Saga Pattern is a direct result of Database-per-service pattern. In Database-per-service pattern, each service has its own database. In other words, each service is responsible only for its own data.

This leads to a tricky situation.

Some business transactions require data from multiple services. Such transactions may also need to update or process data across services. Therefore, a mechanism to handle data consistency across multiple services is required.

This situation or use-case forms the basis of the Saga Pattern.

In this series of posts, we will implement Saga Pattern using Axon Framework and Spring Boot.

Axon Framework is a microservices framework that makes it easy to build distributed systems. It provides great support for Spring Boot and we will be using the same to build a sample application.

Defining the Saga Pattern

Saga Pattern proposes implementing distributed transactions in the form of Sagas.

A Saga is nothing but a sequence of local transactions. These local transactions are occurring at the service level. Whenever a local transaction occurs, it publishes a message or an event. Such an event is responsible for triggering the next transaction in the Saga.

But then, you might ask what happens when a single transaction in the Saga sequence fails?

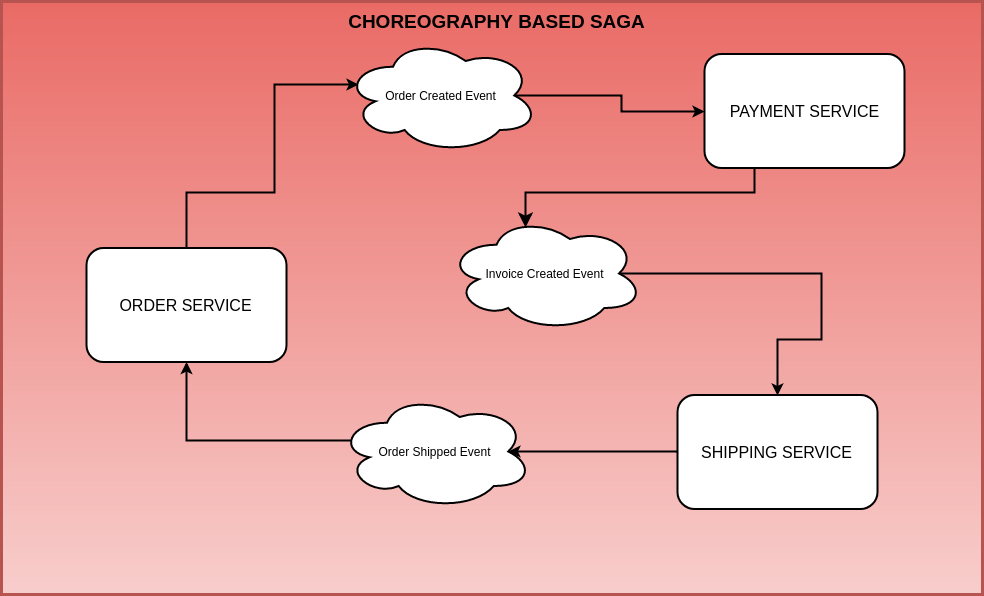
In that case, the Saga executes a series of compensating transactions. These transactions basically undo the changes made by the preceding transactions.

Types of Saga

There are basically two types of Saga. In other words, there are two ways in which you can implement the Saga Pattern.

Choreography-Based Saga

In this type of Saga Implementation, each service publishes one or more domain events. These domain events trigger local transactions in other microservices.



Let’s understand what is happening in the above example.

Order Service is responsible for creating an Order. It also publishes an event for the same.

The Payment Service listens to that event and creates an Invoice.

When the Invoice is created, the Shipping Service creates the shipment.

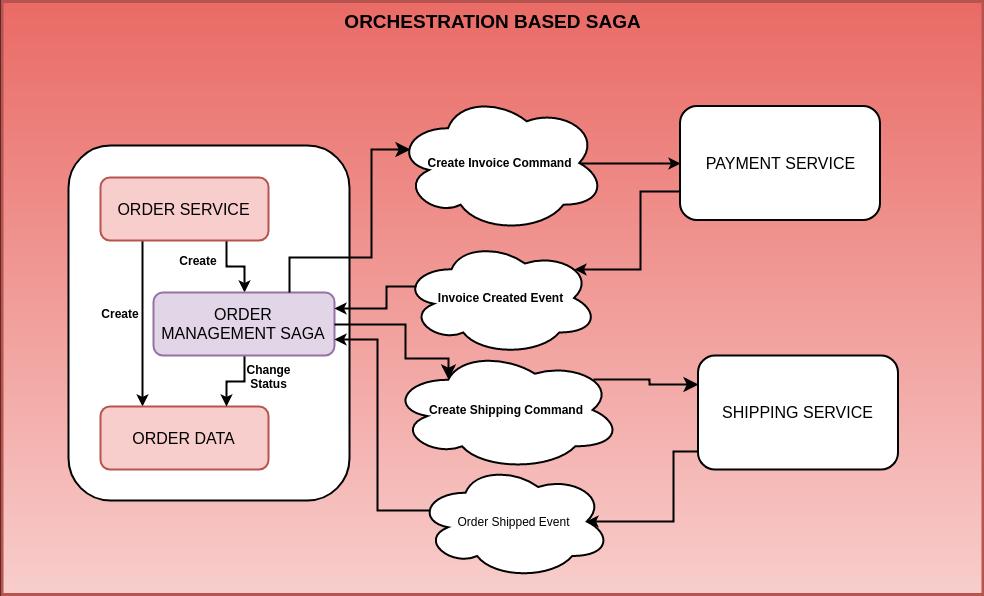
When the Order is shipped, the Order Service updates the status of the Order.

Now, this is a very simple example. In real life situations, there would be many business rules and steps involved to carry out such a distributed transaction. However, this example is only meant to demonstrate how Saga Pattern works.

The important thing to note here is that each service plays its part in the choreography. Each service is basically dependent on the events coming out from other services.

Orchestration-Based Saga

In Orchestration-Based Saga, there is an orchestrator. An orchestrator can also be thought of as a manager that directs the participant services to execute local transactions.



Below is how it can look like for the previous example.

orchestration based saga pattern implementation

Let’s understand what’s happening in this approach:

Order Service creates an Order. Then, it also creates the Order Management Saga.

The Order Management Saga sends a Create Invoice Command to the Payment Service.

The Payment Service creates the Invoice and responds back to the Order Management Saga. Note that these responses can be totally asynchronous and message-driven as well.

In the next step, the Order Management Saga issues the Create Shipping Command to the Shipping Service.

The Shipping Service does the needful and creates the Shipping. It also replies back to the Order Management Saga.

The Order Management Saga changes the status of the Order and ends the Saga’s life-cycle.

Saga Pattern – Benefits and Drawbacks

The biggest benefit of Saga Pattern is that it allows an application to maintain data consistency across multiple services. This is very important from a Microservices Architecture point-of-view because it allows us to write our individual services without tight coupling. In case we run into a case for distributed transactions, we can utilize the Saga Pattern.

However, there are also drawbacks in this approach. The main point going against Saga Pattern is the fact that this pattern makes the programming model more complex. A simple example is developing compensating transactions in case anything goes wrong. Also, developers are usually not completely comfortable with this programming model. At least, not yet.

Other challenges include the topic of atomically updating the Aggregate state as well as publishing domain events.

Usually, Saga Pattern is applied in conjunction with some other patterns such as Event Sourcing and CQRS. These patterns specifically try to handle the scenarios around atomically updating the Aggregate’s state and publish events.

Implementing Saga Pattern

Now that we have understood what is Saga Pattern, we will look at implementing it.

We will be look at Orchestration-Based Implementation. And we will be using the same case as the Order Service described above.

Some of the tools and frameworks we would be using are as follows:

Axon Framework – This is a Java-based Microservices Framework that helps build applications using Domain Driven Design techniques. Building Scalable Microservices using Axon Framework describes the framework in more detail.

Axon Server – From version 4 onward, Axon has becomes more of a platform. The Axon Platform comprises of the core Axon Framework and also the Axon Server. The Axon Server facilitates communication between microservices.

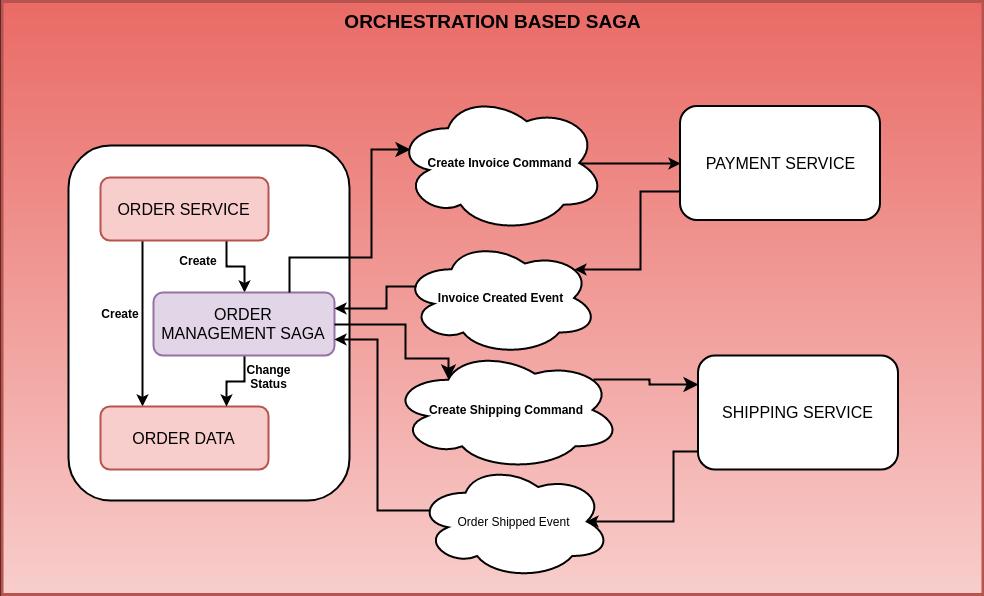
Spring Boot – Spring Boot is probably the most popular Java-based framework for building enterprise application. It’s ease of use and wide integration with other frameworks makes it an ideal choice for building microservices. Building Fastest Production-Ready Microservices using Spring Boot talks in detail about Spring Boot Microservices.

The Problem Statement for Saga Pattern Implementation

As we discussed earlier as well, we will be implementing Saga Pattern using the Orchestration-based Saga approach. As a refresher, Orchestration-based Saga uses an orchestrator to manage the Saga.

In other words, an orchestrator is simply a manager that directs the participating services to execute one or more local transactions.

A typical example of this approach is shown in the below illustration.



orchestration based saga pattern implementation

Here, we are looking at Orchestration-Based Saga for solving the problem of Order Management. This Order Management problem can apply to any e-commerce store, food delivery apps, or any other similar use-case.

Of course, for demo purposes, we have made this problem much simpler than a real production level complexity.

High-Level Components

There are 5 major parts of this application with regards to our Saga Pattern Implementation. The parts are as follows:

Order Service – This service exposes APIs that help creating an Order in the system. Also, the service manages the Order Aggregate. Order Aggregate is nothing but an entity that maintains the Order related information. However, the Order Service also acts as the home for the actual Order Management Saga implementation.

Payment Service – The Payment Service acts upon the Create Invoice Command issued by the Order Management Saga. Once it finishes its job, it publishes an event. This event pushes the Saga forward onto the next step.

Shipping Service – This service takes care of creating a shipment in the system corresponding to the Order. It acts upon a command issued by the Saga Manager. Once it does it’s job, it also publishes an event that pushes the Saga forward.

Core-APIs – This is not a service as such. However, Core-APIs acts as the integration-glue between various services that form a part of the Saga. In our case, the Core-APIs will consist of the various commands and event definitions required for our Saga implementation to function

Axon Server – Axon Server is part of the Axon Platform. We will be using Axon Framework to manage our Aggregates such as Order, Payment, Shipping. Also, we will be using Axon Server to handle the communication between the three services. You can check out my post on in-depth view of Axon Server if you are looking for more details about it.

Below is the overall structure of our application.

.

├── core-apis

├── order-service

├── payment-service

├── pom.xml

├── saga-axon-server-spring-boot.iml

└── shipping-service

As you can see, it is a multi-maven module structure. Each service is a maven module and is part of the overall project.

The main POM.xml file glues all of them together.

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 <http://maven.apache.org/xsd/maven-4.0.0.xsd>">

<modelVersion>4.0.0</modelVersion>

<groupId>com.progressivecoder.saga-pattern</groupId>

<artifactId>saga-axon-server-spring-boot</artifactId>

<packaging>pom</packaging>

<version>1.0-SNAPSHOT</version>

<modules>

<module>order-service</module>

<module>payment-service</module>

<module>core-apis</module>

<module>shipping-service</module>

</modules>

</project>

Order Service Implementation

To implement Order Service (and all of the other services as well), we will create typical Spring Boot applications. If you are not aware of Spring Boot or want a refresher, refer to my post about Spring Boot Microservices.

Below are the dependencies for our application:

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- Axon -->

<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>

<!-- Swagger -->

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.9.2</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.9.2</version>

</dependency>

<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>1</version>

</dependency>

<dependency>

<groupId>com.progressivecoder.saga-pattern</groupId>

<artifactId>core-apis</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

Some of the core things to consider in this are as follows:

We are using Axon Spring Boot Starter (version 4.0.3). This brings in support for Axon Framework plus Axon Server.

To make testing our application easy, we have also included Swagger.

We are using Spring Boot Starter Data JPA and H2 in-memory database for our persistence layer.

Also, we have another module called core-apis. We will come back to it in some time.

The Order Aggregate

Order Aggregate is one of the most important pieces of our Saga Pattern implementation. It forms the base around which the Order Management Saga will work. Let’s see how it looks like:

@Aggregate

public class OrderAggregate {

@AggregateIdentifier

private String orderId;

private ItemType itemType;

private BigDecimal price;

private String currency;

private OrderStatus orderStatus;

public OrderAggregate() {

}

@CommandHandler

public OrderAggregate(CreateOrderCommand createOrderCommand){

AggregateLifecycle.apply(new OrderCreatedEvent(createOrderCommand.orderId, createOrderCommand.itemType,

createOrderCommand.price, createOrderCommand.currency, createOrderCommand.orderStatus));

}

@EventSourcingHandler

protected void on(OrderCreatedEvent orderCreatedEvent){

this.orderId = orderCreatedEvent.orderId;

this.itemType = ItemType.valueOf(orderCreatedEvent.itemType);

this.price = orderCreatedEvent.price;

this.currency = orderCreatedEvent.currency;

this.orderStatus = OrderStatus.valueOf(orderCreatedEvent.orderStatus);

}

@CommandHandler

protected void on(UpdateOrderStatusCommand updateOrderStatusCommand){

AggregateLifecycle.apply(new OrderUpdatedEvent(updateOrderStatusCommand.orderId, updateOrderStatusCommand.orderStatus));

}

@EventSourcingHandler

protected void on(OrderUpdatedEvent orderUpdatedEvent){

this.orderId = orderId;

this.orderStatus = OrderStatus.valueOf(orderUpdatedEvent.orderStatus);

}

}

As you can see, this is a typical entity class. The major things to note here are the Axon specific annotations @Aggregate and @AggregateIdentifier. These annotations allow Axon Framework to manage the Order Aggregate instances.

Also, we are using Event Sourcing to store the events occurring on this aggregate. Event Sourcing is another Microservices Architecture pattern that deals with storing the aggregate information in the form of domain events. I have a detailed post on Event Sourcing implementation if you want to read more about it.

The Order Service

To facilitate creation of Orders, we have also defined an Order Service interface and its corresponding implementation.

public interface OrderCommandService {

public CompletableFuture<String> createOrder(OrderCreateDTO orderCreateDTO);

}

@Service

public class OrderCommandServiceImpl implements OrderCommandService {

private final CommandGateway commandGateway;

public OrderCommandServiceImpl(CommandGateway commandGateway) {

this.commandGateway = commandGateway;

}

@Override

public CompletableFuture<String> createOrder(OrderCreateDTO orderCreateDTO) {

return commandGateway.send(new CreateOrderCommand(UUID.randomUUID().toString(), orderCreateDTO.getItemType(),

orderCreateDTO.getPrice(), orderCreateDTO.getCurrency(), String.valueOf(OrderStatus.CREATED)));

}

}

This service implementation uses Axon Framework’s Command Gateway to issue a command to the Aggregate. The command is handled in the Aggregate class we declared earlier.

The Order Controller

The Order Controller class is the place where we create our API end-points. At this point, for the purposes of our demo, we have only one end-point.

@RestController

@RequestMapping(value = "/api/orders")

@Api(value = "Order Commands", description = "Order Commands Related Endpoints", tags = "Order Commands")

public class OrderCommandController {

private OrderCommandService orderCommandService;

public OrderCommandController(OrderCommandService orderCommandService) {

this.orderCommandService = orderCommandService;

}

@PostMapping

public CompletableFuture<String> createOrder(@RequestBody OrderCreateDTO orderCreateDTO){

return orderCommandService.createOrder(orderCreateDTO);

}

}

To help Swagger discover these end-points, we also configure Swagger using Docket Bean setup.

@Configuration

@EnableSwagger2

public class SwaggerConfig {

@Bean

public Docket apiDocket(){

return new Docket(DocumentationType.SWAGGER\_2)

.select()

.apis(RequestHandlerSelectors.basePackage("com.progressivecoder.ordermanagement"))

.paths(PathSelectors.any())

.build()

.apiInfo(getApiInfo());

}

private ApiInfo getApiInfo(){

return new ApiInfo(

"Saga Pattern Implementation using Axon and Spring Boot",

"App to demonstrate Saga Pattern using Axon and Spring Boot",

"1.0.0",

"Terms of Service",

new Contact("Saurabh Dashora", "progressivecoder.com", "coder.progressive@gmail.com"),

"",

"",

Collections.emptyList());

}

}

The Order Management Saga

The heart of the Saga Pattern implementation is the Order Management Saga. In a nutshell, this is also a typical Java class that describes the various handlers for the individual Saga steps.

The individual Saga steps can be managed in a declarative manner. In other words, this makes it extremely easy for a developer to understand the flow of the Saga at a single-glance.

@Saga

public class OrderManagementSaga {

@Inject

private transient CommandGateway commandGateway;

@StartSaga

@SagaEventHandler(associationProperty = "orderId")

public void handle(OrderCreatedEvent orderCreatedEvent){

String paymentId = UUID.randomUUID().toString();

System.out.println("Saga invoked");

//associate Saga

SagaLifecycle.associateWith("paymentId", paymentId);

System.out.println("order id" + orderCreatedEvent.orderId);

//send the commands

commandGateway.send(new CreateInvoiceCommand(paymentId, orderCreatedEvent.orderId));

}

@SagaEventHandler(associationProperty = "paymentId")

public void handle(InvoiceCreatedEvent invoiceCreatedEvent){

String shippingId = UUID.randomUUID().toString();

System.out.println("Saga continued");

//associate Saga with shipping

SagaLifecycle.associateWith("shipping", shippingId);

//send the create shipping command

commandGateway.send(new CreateShippingCommand(shippingId, invoiceCreatedEvent.orderId, invoiceCreatedEvent.paymentId));

}

@SagaEventHandler(associationProperty = "orderId")

public void handle(OrderShippedEvent orderShippedEvent){

commandGateway.send(new UpdateOrderStatusCommand(orderShippedEvent.orderId, String.valueOf(OrderStatus.SHIPPED)));

}

@SagaEventHandler(associationProperty = "orderId")

public void handle(OrderUpdatedEvent orderUpdatedEvent){

SagaLifecycle.end();

}

}

Let’s understand what is going on here.

The @StartSaga annotation signifies the start of the Saga. It tells Axon to create a new Saga. The Saga is also associated with a particular instance of the Aggregate. This is done using the associationProperty specified with the @SagaEventHandler. In this case, we are associating the Saga with an instance of the Order Aggregate using the property orderId. We have also specified the event for which the method should be called. In our case, it is the Order Created Event.

The other methods annotated with @SagaEventHandler signify the other transactions that are part of the Saga.

We also associate the Saga with other concepts such as Payment and Shipping using SagaLifecycle.associateWith() method. By allowing the clients to generate the identifier, we don’t have to follow a request-response model. This allows us to easily associate an identifier with the Saga.

The Saga life-cycle is finished once we call the SagaLifecycle.end() method.

Core-APIs

Now would be a good time to look at the Core-APIs module.

The Core-APIs are nothing but a bunch of classes that define the Commands and Events that are going to be a part of our Saga Pattern Implementation.

Commands

The Create Order Command is triggered when a new Order is created in our application. This command is handled by the Order Aggregate.

public class CreateOrderCommand {

@TargetAggregateIdentifier

public final String orderId;

public final String itemType;

public final BigDecimal price;

public final String currency;

public final String orderStatus;

public CreateOrderCommand(String orderId, String itemType, BigDecimal price, String currency, String orderStatus) {

this.orderId = orderId;

this.itemType = itemType;

this.price = price;

this.currency = currency;

this.orderStatus = orderStatus;

}

}

Next, the Create Invoice Command is triggered by the Order Management Saga when the Order is created.

public class CreateInvoiceCommand{

@TargetAggregateIdentifier

public final String paymentId;

public final String orderId;

public CreateInvoiceCommand(String paymentId, String orderId) {

this.paymentId = paymentId;

this.orderId = orderId;

}

}

The Create Shipping Command is also triggered by the Order Management Saga when the invoice creation and payment processing is done.

public class CreateShippingCommand {

@TargetAggregateIdentifier

public final String shippingId;

public final String orderId;

public final String paymentId;

public CreateShippingCommand(String shippingId, String orderId, String paymentId) {

this.shippingId = shippingId;

this.orderId = orderId;

this.paymentId = paymentId;

}

}

Lastly, we have the Update Order Status Command. When the shipping is done, this command is triggered.

public class UpdateOrderStatusCommand {

@TargetAggregateIdentifier

public final String orderId;

public final String orderStatus;

public UpdateOrderStatusCommand(String orderId, String orderStatus) {

this.orderId = orderId;

this.orderStatus = orderStatus;

}

}

Events

The first event in the overall process is the Order Created Event. This event is also responsible for starting up the Saga as we saw earlier.

public class OrderCreatedEvent {

public final String orderId;

public final String itemType;

public final BigDecimal price;

public final String currency;

public final String orderStatus;

public OrderCreatedEvent(String orderId, String itemType, BigDecimal price, String currency, String orderStatus) {

this.orderId = orderId;

this.itemType = itemType;

this.price = price;

this.currency = currency;

this.orderStatus = orderStatus;

}

}

Next event is the Invoice Created Event. The Invoice Service publishes this event. We will see the implementation for the same in the next post.

public class InvoiceCreatedEvent {

public final String paymentId;

public final String orderId;

public InvoiceCreatedEvent(String paymentId, String orderId) {

this.paymentId = paymentId;

this.orderId = orderId;

}

}

After that, we have the Order Shipped Event. This event is published by the Shipping Service after it has done the needful.

public class OrderShippedEvent {

public final String shippingId;

public final String orderId;

public final String paymentId;

public OrderShippedEvent(String shippingId, String orderId, String paymentId) {

this.shippingId = shippingId;

this.orderId = orderId;

this.paymentId = paymentId;

}

}

Lastly, we have the Order Updated Event. This event is published by the Order Aggregate after it has updated the status of the Order.

public class OrderUpdatedEvent {

public final String orderId;

public final String orderStatus;

public OrderUpdatedEvent(String orderId, String orderStatus) {

this.orderId = orderId;

this.orderStatus = orderStatus;

}

}

Conclusion

At this point, we have managed to come pretty far in implementing two major pieces of our application – the Order Service and the Core-APIs.

The Order Service houses our main Saga Pattern Implementation code. On the other hand, the Core-APIs are the backbone of our Order Management Saga.

Payment Service Implementation

Payment Service is another typical Spring Boot application. In the context of our application, this service takes care of creating an invoice after the Order is created

The Payment Aggregate

The Payment Aggregate is the core for this service. It stores the invoice related information and the relationship with the Order Aggregate.

@Aggregate

public class InvoiceAggregate {

@AggregateIdentifier

private String paymentId;

private String orderId;

private InvoiceStatus invoiceStatus;

public InvoiceAggregate() {

}

@CommandHandler

public InvoiceAggregate(CreateInvoiceCommand createInvoiceCommand){

AggregateLifecycle.apply(new InvoiceCreatedEvent(createInvoiceCommand.paymentId, createInvoiceCommand.orderId));

}

@EventSourcingHandler

protected void on(InvoiceCreatedEvent invoiceCreatedEvent){

this.paymentId = invoiceCreatedEvent.paymentId;

this.orderId = invoiceCreatedEvent.orderId;

this.invoiceStatus = InvoiceStatus.PAID;

}

}

As you can see it’s a pretty simple entity class. While in a real production case, this will have lots of other details. However, for the purposes of our sample app, I have kept it intentionally simple.

Similar to the Order Service, we are using Event Sourcing to store the Aggregate information. If you want to know more about Event Sourcing, I have a detailed post on Event Sourcing using Axon and Spring Boot.

To elaborate further, the Payment Aggregate has a handler for Create Invoice Command. When this command is received, a new instance of the payment is created and a Invoice Created Event is published.

Application Properties File

We will run the Payment Service on a different port. For doing so, we will set the server.port property in the application.properties file.

Also, note that we are providing an application.name property. This is a good practice as this will help us easily identify our application when it connects to the Axon Server.

spring.application.name=payment-service

server.port=8081

hipping Service Implementation

The responsibility of the Shipping Service is to create a shipment. This service waits for the appropriate command from the Order Management Saga. On receiving the command, it creates a shipment.

The Shipping Aggregate

The Shipping Aggregate is the core of the Shipping Service. This Aggregate stores the relationship between the shipment and the order.

Below is the how the Aggregate class definition looks like:

@Aggregate

public class ShippingAggregate {

@AggregateIdentifier

private String shippingId;

private String orderId;

private String paymentId;

public ShippingAggregate() {

}

@CommandHandler

public ShippingAggregate(CreateShippingCommand createShippingCommand){

AggregateLifecycle.apply(new OrderShippedEvent(createShippingCommand.shippingId, createShippingCommand.orderId, createShippingCommand.paymentId));

}

@EventSourcingHandler

protected void on(OrderShippedEvent orderShippedEvent){

this.shippingId = orderShippedEvent.shippingId;

this.orderId = orderShippedEvent.orderId;

}

}

As can be seen, the Aggregate has a handler for the Create Shipping Command. This command is issued by the Order Management Saga.

Once the command is received, the Aggregate publishes the Order Shipped Event. It also sets the values on the Aggregate instance.

Application Properties File

We will update the application.properties file to set the server.port property. Also, we will set the application.name to shipping-service.

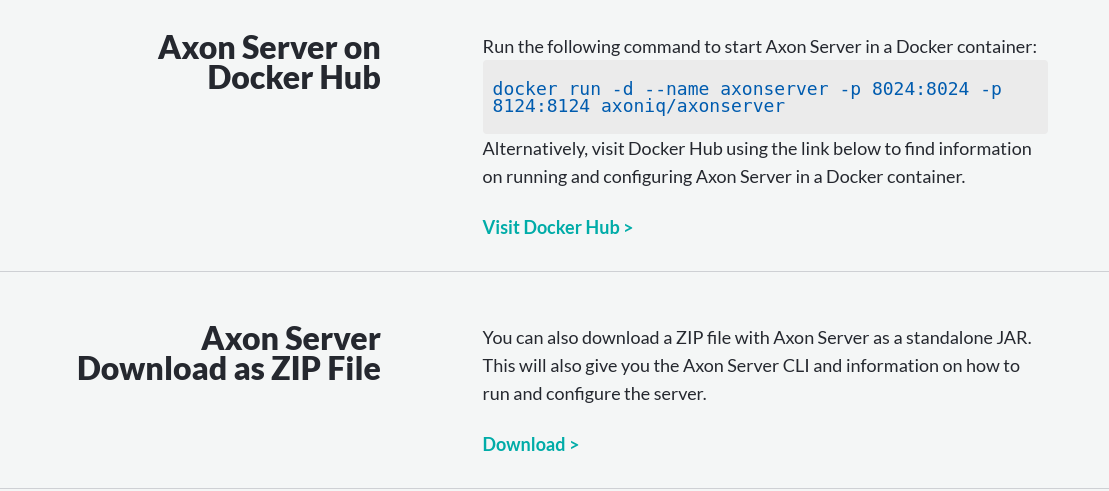
spring.application.name=shipping-service

server.port=8082

Setting up Axon Server

Next step for us is to start up the Axon Server. On a high-level, Axon Server is primarily used to facilitate communication between the various services in our Saga Pattern Implementation. If you want to know more, I have a detailed post on Axon Server.

To start up Axon Server, we can download the Axon Server JAR file in zip format from AxonIQ site.



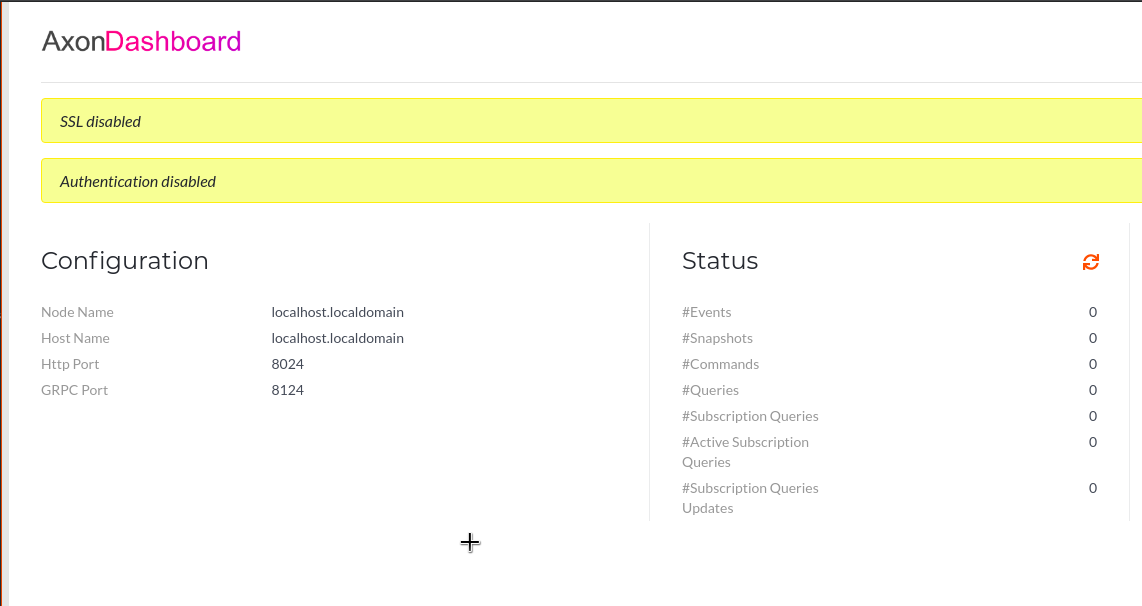
As you can see there are several options available. We can run Axon Server as a Docker Image or download the entire platform. However, we will use the Axon Server as ZIP file approach.

Download the ZIP file and extract it somewhere on your system. Then, you can simply get into the directory and run the below command in your command prompt or terminal to run Axon Server.

java -jar axonserver.jar

The Axon Server should start up on port 8024

Next, you can visit <http://localhost:8024>. If everything has worked fine, you should be presented with the below screen.



Starting up the other services

Now that Axon Server is up and running, we need to also start the other services that are part of our Saga Pattern Implementation. These are the Order-Service, the Payment-Service and the Shipment-Service.

To start the three services, we can use the below commands at the root directory of our multi-maven module project structure.

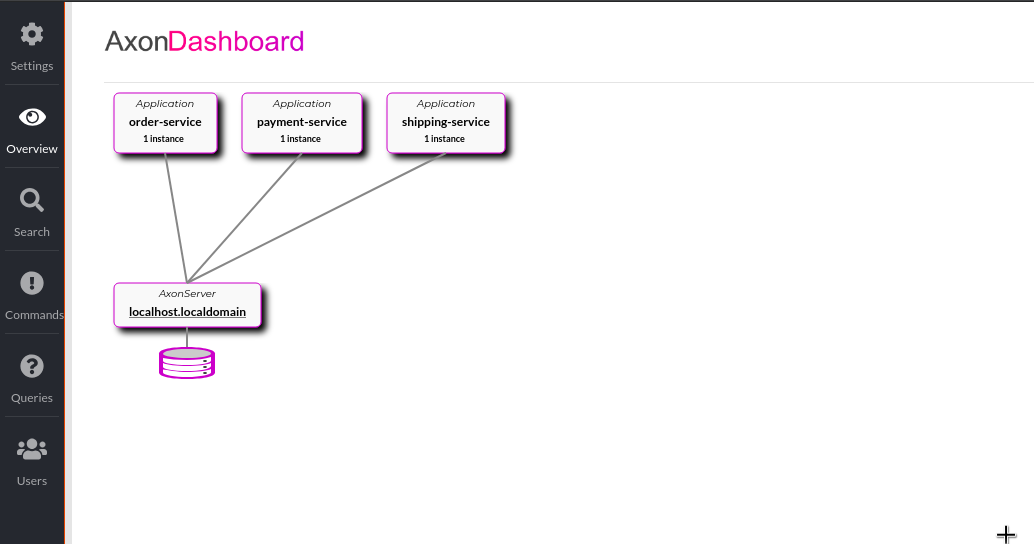
clean package spring-boot:run -pl order-service --also-make

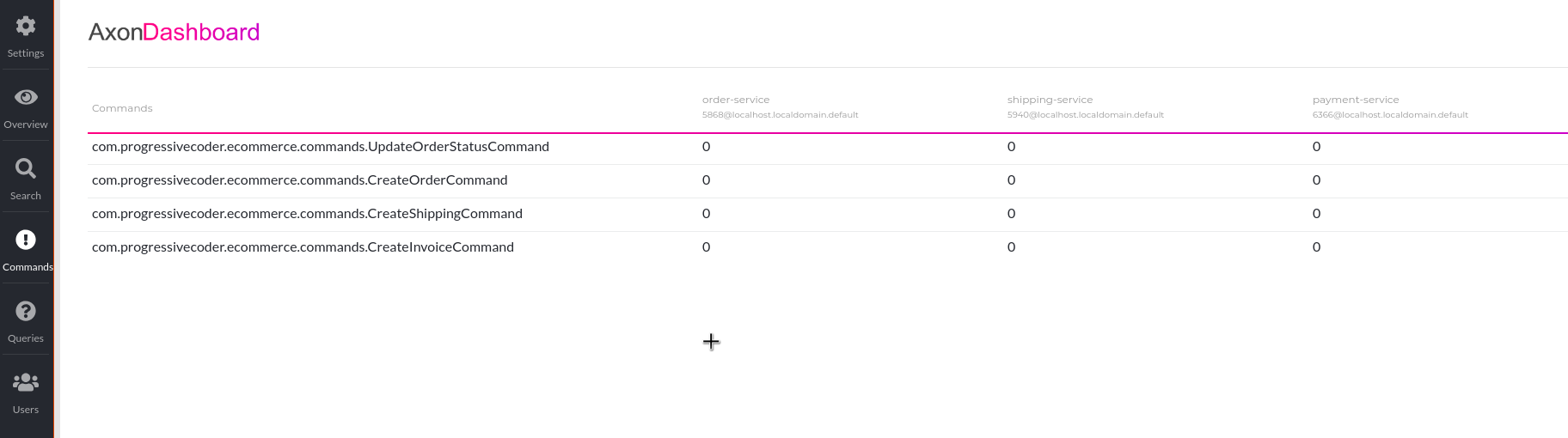
clean package spring-boot:run -pl payment-service --also-make

clean package spring-boot:run -pl shipping-service --also-make

Basically, here we are just starting up the individual services. The –also-make parameter ensures that the dependencies for the project are also bundled up in the JAR file that gets created.

Once the three applications have started up successfully, we can see them in the Overview page of the Axon Server Dashboard.





Initiating the Saga

/ap1/orders

{

"currency": "USD",

"itemType":"LAPTOP",

"price":1000

}

This will also initiate the Order Management Saga.