	**Made by:** Valerio Bondì
	23th November 2022
Object	**State**
Ipitq OrderManager challenge documentation	Final Draft

Contents

Introduction

The **OrderManager** project consists in a simple Spring Boot application which creates and ships **orders** made by EverythingEverywhere and ACME. Creation and shipping of the orders are managed by **Kafka Streams**, in particular:

- A **Kafka Producer** produces the order;
- A **Kafka Stream** consume, process it and produces an **event** into a different topic.

The project techstack is made by:

- Java 18;
- Spring Boot 2.7.5
- Docker 4.14

Setup

For the setup of the project, it's recommended to use IntelliJ Idea + Docker to run the project inside IntelliJ. Otherwise Java JDK 18 e Docker are mandatory to run the project from the terminal. More details can be found on the *README.md* file.

Structure

The application:

- Uses **Kafka Streams** to consume messages from a Kafka topic named **orders** and map into another topic named **events.** The objects will not be modified because there are no instructions about it;
- Uses **Docker** to run the Kafka image inside a container. After installing Docker, running the **docker-compose.yml** file is enough to the startup of the containers. There is also a **script** folder where is defined a little script to create orders and events topics;
- Uses **Avro Models** to serialize the Kafka messages in an efficient way;
- Uses **Mapstruct** to map the input data into models with just an interface and few lines of codes.

The application follows this package structure:

- **Configuration:** contains the Kafka Configuration's classes;
- **Controller:** here are defined all the REST APIs for the creation and shipping of the orders;
- **Exception:** contains the classes about the exception management;
- **Mapper:** contains a mapper that maps the APIs input to an Avro model.
- **Model:** all the model classes, DTO included;
- **Producer:** the Kafka Producer that produce the creation or a shipping of the order;
- **Service:** contains the standard business logic;
- **Stream:** the Kafka Stream implementation and the Processor class that processes the data from the first topic and creates the event for the second one.

There are also other packages as:

- **Resources:** contains the resources of the application:
  - *application.yml:* contains all the configuration as Kafka Producer and Stream properties, server port etc.;
  - *commands and imports:* contain the Avro Models.
- **Test:** contains the Unit Tests.

Configuration

The configuration when you have to deal with several technologies can be confusing. Here there are just 4 files that configures all the Kafka elements, so streams and producer:

- **Application.yml:** there are defined the properties of kafka as the bootstrap servers and also the streams and producer properties, and the topic names;
- **KafkaConfiguration.java:** the base Kafka Configuration, with the annotations to enable Kafka and the beans that define the production and consumption of the stream;
- **KafkaProducerConfiguration.java:** the custom configuration of the producer, that defines 2 beans: one that reads the configurations from application.yml and merges them with the default properties, and one that simply creates an instance of the KafkaTemplate;
- **KafkaTopics:** it simply gets the 2 topics values from application.yml.

With this configuration, the business logic classes can totally abstract the configuration part and focus just on the logic.

Avro Models

The Avro Models are defined under *resouces/commands* and *resouces/imports*. The avro models are similar to the Model objects, but mapping into Avro models are useful to have a common serialize/deserialize object (SpecificRecord) and avoid some problems (for example consume a custom object with the JsonSerde from the stream with some headers produce some exceptions, avoided using the SpecificRecord and AvroSerde)

Business Logic

The business logic follows this flow:

OrderController: it simply exposes 4 api:

- *createEEOrder:* API to create an EverythingEverywhere Order;
- *createAcmeOrder:* API to create an ACME Order;
- *shipEEOrder:* API to ship an EverythingEverywhere Order;
- *shipAcmeOrder:* API to ship an ACME Order;

The input is **validated** (with annotations inside the model classes) to avoid null values.

The controller has also an interface **OrderApi** where is defined all the APIs docs and schemas.

OrderService: the main business logic class that simply produces the messages to the orders topic. It has 4 methods, one for every API defined in the controller. For every method:

- It creates the avro payload with the mapstruct **OrderMapper;**

- It builds the **kafkaKey** made by:
    - The order ID;
    - The company name (EE or ACME);
    - The event name (CREATED or SHIPPED).
- It sends the message though the Kafka Template.

OrderKafkaProducer: the implementation of the Kafka Producer.

- With the *sendMessage(KafkaKey kafkaKey, SpecificRecord payload, String topic)* method, it send a message made by the kafkaKey and payload provided to the specified topic.
- It provides also *successHandler()* and *failureHandler()* methods, called if the sending is successful or not.

OrderEventStream: the implementation of the Kafka Stream.

It listens for new messages from the order topic. So when the producer successfully send a message to the topic, the OrderEventStream will consume the message, send it to the OrderMessageProcessor to map the payload into an Event and produce it to the event topic.

OrderMessageProcessor: the OrderMessageProcessor simply gets the message from the stream and map the SpecificRecord into an Event object, a map that contains the original data from the payload.

Exception managing

There are no custom exceptions in the project, but there is a **GlobalExceptionHandler** that maps the validation exceptions into a more readable and explaining message. The message provides:

- The error detail;
- The error code;
- The timestamp.

Tests

The project provides **Unit Tests** made by Junit5 and using **Mockito** to mock classes and beans. The coverage is about 84% (not considering the model classes because there are lines generated by Lombok as getters, setters, constructors etc).

There are no automated integration tests, but integration test can be made using the **Postman Collection** provided and test all the APIs. With the **AKHQ** console, you can check the kafka topics and the message generated.

Docs and Credits

All the documentations can be found at:

- README.md file;
- Docs.pdf file;
- OpenAPI Swagger link: Swagger UI

All the project is made by Valerio Bondì.