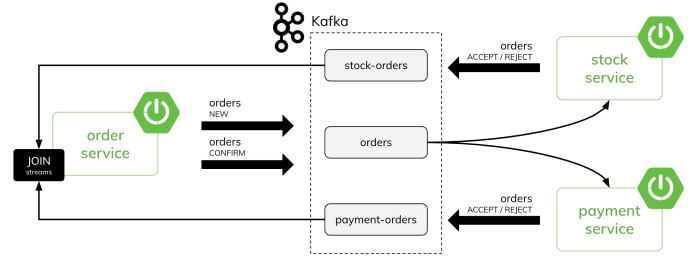
**Distributed Transactions in Microservices with Kafka Streams and Spring Boot**

In this article, you will learn how to use Kafka Streams with Spring Boot. We will rely on the Spring Kafka project. In order to explain well how it works, we are going to implement a saga pattern. The saga pattern is a way to manage distributed transactions across microservices. The key phase of that process is to publish an event that triggers local transactions. Microservices exchanges such events through a message broker. It turns out that Kafka Streams may help us here. Let’s see how!

**Architecture**

We will create a simple system that consists of three microservices. The order-service sends orders to the Kafka topic called orders. Both other microservices stock-service and payment-service listen for the incoming events. After receiving them they verify if it is possible to execute the order. For example, if there are no sufficient funds on the customer account the order is rejected. Otherwise, the payment-service accepts the order and sends a response to the payment-orders topic. The same with stock-service except that it verifies a number of products in stock and sends a response to the stock-orders topic.

Then, the order-service joins two streams from the stock-orders and payment-orders topics by the order’s id. If both orders were accepted it confirms a distributed transaction. On the other hand, if one order has been accepted and the second rejected it performs rollback. In that case, it just generates o new order event and sends it to the orders topic. We may treat the orders topic as a stream of the order’s status changes or just like a table with the last status. Here’s the picture that visualizes our scenario.



**Kafka Streams with Spring Boot**

Let’s begin our implementation from the order-service. Surprisingly there is no Spring Boot starter for Kafka (unless we use Spring Cloud Stream). Therefore we need to include the spring-kafka dependency. In order to process streams we also need to include the kafka-streams module directly. Since the order-service exposes some REST endpoints it is required to add the Spring Boot Web starter.

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

</dependency>

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-streams</artifactId>

</dependency>

The order-service is the most important microservice in our scenario. It acts as an order gateway and a saga pattern orchestrator. It requires all the three topics used in our architecture. In order to automatically create topics on application startup we need to define the following beans:

@Bean

public NewTopic orders() {

return TopicBuilder.name("orders")

.partitions(3)

.compact()

.build();

}

@Bean

public NewTopic paymentTopic() {

return TopicBuilder.name("payment-orders")

.partitions(3)

.compact()

.build();

}

@Bean

public NewTopic stockTopic() {

return TopicBuilder.name("stock-orders")

.partitions(3)

.compact()

.build();

}

Then, let’s define our first Kafka stream. To do that we need to use the StreamsBuilder bean. The order-service receives events from the payment-service (in the payment-events topic) and from the stock-service (in the stock-events topic). Every single event contains the id previously set by the order-service. If we join both these streams into a single stream by order’s id we will be able to determine the status of our transaction. The algorithm is pretty simple. If both payment-service and stock-service accepted the order the final status of transaction is CONFIRMED. If both services rejected the order the final status is REJECTED. The last option is ROLLBACK – when one service accepted the order, and one service rejected it. Here’s the described method inside the OrderManageService bean.

@Service

public class OrderManageService {

public Order confirm(Order orderPayment, Order orderStock) {

Order o = new Order(orderPayment.getId(),

orderPayment.getCustomerId(),

orderPayment.getProductId(),

orderPayment.getProductCount(),

orderPayment.getPrice());

if (orderPayment.getStatus().equals("ACCEPT") &&

orderStock.getStatus().equals("ACCEPT")) {

o.setStatus("CONFIRMED");

} else if (orderPayment.getStatus().equals("REJECT") &&

orderStock.getStatus().equals("REJECT")) {

o.setStatus("REJECTED");

} else if (orderPayment.getStatus().equals("REJECT") ||

orderStock.getStatus().equals("REJECT")) {

String source = orderPayment.getStatus().equals("REJECT")

? "PAYMENT" : "STOCK";

o.setStatus("ROLLBACK");

o.setSource(source);

}

return o;

}

}

Finally, the implementation of our stream. We need to define the KStream bean. We are joining two streams using the join method of KStream. The joining window is 10 seconds. As the result, we are setting the status of the order and sending a new order to the orders topic. We use the same topic as for sending new orders.

@Autowired

OrderManageService orderManageService;

@Bean

public KStream<Long, Order> stream(StreamsBuilder builder) {

JsonSerde<Order> orderSerde = new JsonSerde<>(Order.class);

KStream<Long, Order> stream = builder

.stream("payment-orders", Consumed.with(Serdes.Long(), orderSerde));

stream.join(

builder.stream("stock-orders"),

orderManageService::confirm,

JoinWindows.of(Duration.ofSeconds(10)),

StreamJoined.with(Serdes.Long(), orderSerde, orderSerde))

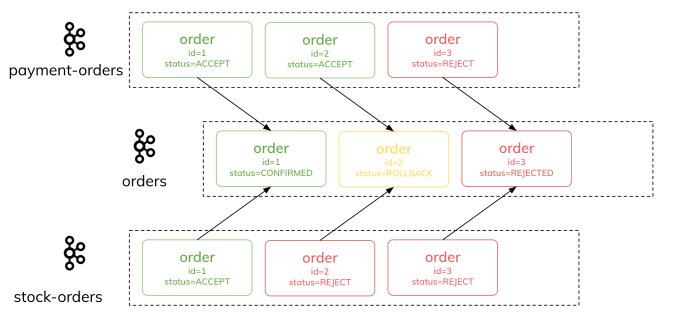
.peek((k, o) -> LOG.info("Output: {}", o))

.to("orders");

return stream;

}

Let’s see it also in the picture.



**Configuration for Spring Boot**

In Spring Boot the name of the application is by default the name of the consumer group for Kafka Streams. Therefore, we should set in application.yml. Of course, we also need to set the address of the Kafka bootstrap server. Finally, we have to configure the default key and value for events serialization. It applies to both standard and streams processing.

spring.application.name: orders

spring.kafka:

bootstrap-servers: 127.0.0.1:56820

producer:

key-serializer: org.apache.kafka.common.serialization.LongSerializer

value-serializer: org.springframework.kafka.support.serializer.JsonSerializer

streams:

properties:

default.key.serde: org.apache.kafka.common.serialization.Serdes$LongSerde

default.value.serde: org.springframework.kafka.support.serializer.JsonSerde

spring.json.trusted.packages: "\*"

**Sending and receiving events from Kafka topics**

In the previous section, we discussed how to create a new Kafka stream as a result of joining two other streams. Now, let’s see how to process incoming messages. We can consider it on the example of the payment-service. It listens for the incoming orders. If it receives a new order it performs reservation on the customer’s account and sends a response with a reservation status to the payment-orders topic. If it receives confirmation of the transaction from the order-service, it commits the transaction or rollbacks it. In order to enable Kafka listener, we should annotate the main class with @EnableKafka. Additionally, the listening method must be annotated with the @KafkaListener. The following method listens for events on the orders topic and runs in the payment consumer group.

@SpringBootApplication

@EnableKafka

public class PaymentApp {

private static final Logger LOG = LoggerFactory.getLogger(PaymentApp.class);

public static void main(String[] args) {

SpringApplication.run(PaymentApp.class, args);

}

@Autowired

OrderManageService orderManageService;

@KafkaListener(id = "orders", topics = "orders", groupId = "payment")

public void onEvent(Order o) {

LOG.info("Received: {}" , o);

if (o.getStatus().equals("NEW"))

orderManageService.reserve(o);

else

orderManageService.confirm(o);

}

}

Here’s the implementation of the OrderManageService used in the previous code snippet. If it receives the order in the NEW status it performs reservation. During the reservation, it subtracts the order’s price from the amountAvailable field and adds the same value to the amountReserved field. Then it sets the status of the order and sends a response to the payment-orders topic using KafkaTemplate. During the confirmation phase, it doesn’t send any response event. It can perform a rollback, which means – subtracting the order’s price from the amountReserved field and adding it to the amountAvailable field. Otherwise it just “commits” the transaction by subtracting the price from the amountReserved field.

@Service

public class OrderManageService {

private static final String SOURCE = "payment";

private static final Logger LOG = LoggerFactory.getLogger(OrderManageService.class);

private CustomerRepository repository;

private KafkaTemplate<Long, Order> template;

public OrderManageService(CustomerRepository repository, KafkaTemplate<Long, Order> template) {

this.repository = repository;

this.template = template;

}

public void reserve(Order order) {

Customer customer = repository.findById(order.getCustomerId()).orElseThrow();

LOG.info("Found: {}", customer);

if (order.getPrice() < customer.getAmountAvailable()) {

order.setStatus("ACCEPT");

customer.setAmountReserved(customer.getAmountReserved() + order.getPrice());

customer.setAmountAvailable(customer.getAmountAvailable() - order.getPrice());

} else {

order.setStatus("REJECT");

}

order.setSource(SOURCE);

repository.save(customer);

template.send("payment-orders", order.getId(), order);

LOG.info("Sent: {}", order);

}

public void confirm(Order order) {

Customer customer = repository.findById(order.getCustomerId()).orElseThrow();

LOG.info("Found: {}", customer);

if (order.getStatus().equals("CONFIRMED")) {

customer.setAmountReserved(customer.getAmountReserved() - order.getPrice());

repository.save(customer);

} else if (order.getStatus().equals("ROLLBACK") && !order.getSource().equals(SOURCE)) {

customer.setAmountReserved(customer.getAmountReserved() - order.getPrice());

customer.setAmountAvailable(customer.getAmountAvailable() + order.getPrice());

repository.save(customer);

}

}

}

A similar logic is implemented on the stock-service side. However, instead of the order’s price, it uses the field productCount and performs reservation for the desired number of ordered products. Here’s the implementation of the OrderManageService class in the stock-service.

@Service

public class OrderManageService {

private static final String SOURCE = "stock";

private static final Logger LOG = LoggerFactory.getLogger(OrderManageService.class);

private ProductRepository repository;

private KafkaTemplate<Long, Order> template;

public OrderManageService(ProductRepository repository, KafkaTemplate<Long, Order> template) {

this.repository = repository;

this.template = template;

}

public void reserve(Order order) {

Product product = repository.findById(order.getProductId()).orElseThrow();

LOG.info("Found: {}", product);

if (order.getStatus().equals("NEW")) {

if (order.getProductCount() < product.getAvailableItems()) {

product.setReservedItems(product.getReservedItems() + order.getProductCount());

product.setAvailableItems(product.getAvailableItems() - order.getProductCount());

order.setStatus("ACCEPT");

repository.save(product);

} else {

order.setStatus("REJECT");

}

template.send("stock-orders", order.getId(), order);

LOG.info("Sent: {}", order);

}

}

public void confirm(Order order) {

Product product = repository.findById(order.getProductId()).orElseThrow();

LOG.info("Found: {}", product);

if (order.getStatus().equals("CONFIRMED")) {

product.setReservedItems(product.getReservedItems() - order.getProductCount());

repository.save(product);

} else if (order.getStatus().equals("ROLLBACK") && !order.getSource().equals(SOURCE)) {

product.setReservedItems(product.getReservedItems() - order.getProductCount());

product.setAvailableItems(product.getAvailableItems() + order.getProductCount());

repository.save(product);

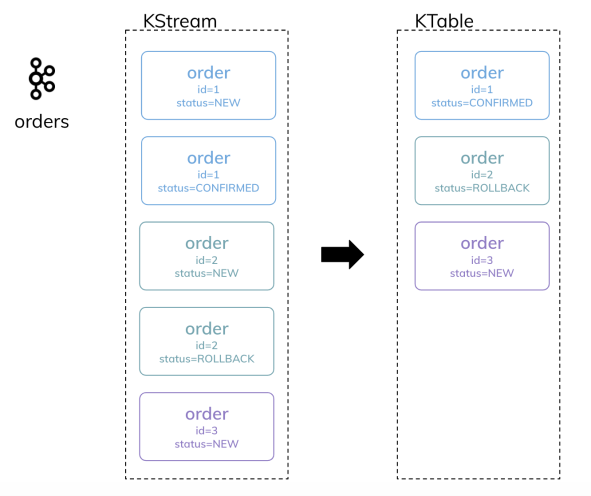
}

}

}

**Query Kafka Stream with Spring Boot**

Now, let’s consider the following scenario. Firstly, the order-service receives a new order (via REST API) and sends it to the Kafka topic. This order is then received by both other microservices. Once they send back a positive response (or negative) the order-service process them as streams and change the status of the order. The order with a new status is emitted to the same topic as before. So, where we are storing the data with the current status of an order? In Kafka topic. Once again we will use Kafka Streams in our Spring Boot application. But this time, we take advantage of KTable. Let’s at the visualization of our scenario.



Ok, so let’s define another Kafka Streams bean in the order-service. We are getting the same orders topic as a stream. We will convert it to the Kafka table and materialize it in a persistent store. Thanks to that we will be able to easily query the store from our REST controller.

@Bean

public KTable<Long, Order> table(StreamsBuilder builder) {

KeyValueBytesStoreSupplier store =

Stores.persistentKeyValueStore("orders");

JsonSerde<Order> orderSerde = new JsonSerde<>(Order.class);

KStream<Long, Order> stream = builder

.stream("orders", Consumed.with(Serdes.Long(), orderSerde));

return stream.toTable(Materialized.<Long, Order>as(store)

.withKeySerde(Serdes.Long())

.withValueSerde(orderSerde));

}

If we run more than one instance of the order-service on the same machine it is also important to override the default location of the state store. To do that we should define the following property unique per every instance:

spring.kafka.streams.state-dir: /tmp/kafka-streams/1

Unfortunately, there is no built-in support in Spring Boot for interactive queries of Kafka Streams. However, we can use auto-configured StreamsBuilderFactoryBean to inject KafkaStreams instance into the controller. Then we can query the state store under the “materialized” name. That’s of course very trivial sample. We are just getting all orders from KTable.

@GetMapping

public List<Order> all() {

List<Order> orders = new ArrayList<>();

ReadOnlyKeyValueStore<Long, Order> store = kafkaStreamsFactory

.getKafkaStreams()

.store(StoreQueryParameters.fromNameAndType(

"orders",

QueryableStoreTypes.keyValueStore()));

KeyValueIterator<Long, Order> it = store.all();

it.forEachRemaining(kv -> orders.add(kv.value));

return orders;

}

In the same OrderController there is also a method for sending a new order to the Kafka topic.

@PostMapping

public Order create(@RequestBody Order order) {

order.setId(id.incrementAndGet());

template.send("orders", order.getId(), order);

LOG.info("Sent: {}", order);

return order;

}

**Testing Scenario**

Before we run our sample microservices, we need to start the local instance of Kafka. Usually, I’m using Redpanda for that. It is a Kafka API compatible streaming platform. In comparison to Kafka, it is relatively easy to run it locally. All you need to do is to install the rpk CLI locally (here is the [**instruction**](https://vectorized.io/docs/quick-start-macos) for macOS). After that, you can create a single-node instance using the following command:

$ rpk container start

After running, it will print the address of your node. For me, it is 127.0.0.1:56820. You should put that address as a value of the spring.kafka.bootstrap-servers property. You can also display a list of created topics using the following command:

$ rpk topic list --brokers 127.0.0.1:56820

Then, let’s run our microservices. Begin from the order-service since it is creating all the required topics and building the Kafka Streams instance. You can send a single order using the REST endpoint:

$ curl -X 'POST' \

'http://localhost:8080/orders' \

-H 'Content-Type: application/json' \

-d '{

"customerId": 10,

"productId": 10,

"productCount": 5,

"price": 100,

"status": "NEW"

}'

There is some test data inserted while payment-service and stock-service start. So, you can set the value of customerId or productId between 1 and 100 and it will work for you. However, you can use as well a method for generating a random stream of data. The following bean is responsible for generating 10000 random orders:

@Service

public class OrderGeneratorService {

private static Random RAND = new Random();

private AtomicLong id = new AtomicLong();

private Executor executor;

private KafkaTemplate<Long, Order> template;

public OrderGeneratorService(Executor executor, KafkaTemplate<Long, Order> template) {

this.executor = executor;

this.template = template;

}

@Async

public void generate() {

for (int i = 0; i < 10000; i++) {

int x = RAND.nextInt(5) + 1;

Order o = new Order(id.incrementAndGet(), RAND.nextLong(100) + 1, RAND.nextLong(100) + 1, "NEW");

o.setPrice(100 \* x);

o.setProductCount(x);

template.send("orders", o.getId(), o);

}

}

}

You can start that process by calling the endpoint POST /orders/generate.

@PostMapping("/generate")

public boolean create() {

orderGeneratorService.generate();

return true;

}

No matter if decide to send single order or generate multiple random orders you can easily query the status of orders using the following endpoint:

$ curl http://localhost:8080/orders

Here’s a structure of topics generated by the application and by Kafka Streams to perform join operation and save the orders KTable as a state store.

