**Saga Choreography Pattern with Kafka Spring Boot – 2022**

In case of Choreography pattern, each microservice publishes an event to a Kafka Topic and also each microservice creates a listener to get the information from the other microservices to take appropriate action. Basically each microservice interacts with other microservices through Kafka Topics.

As an example, we consider this.

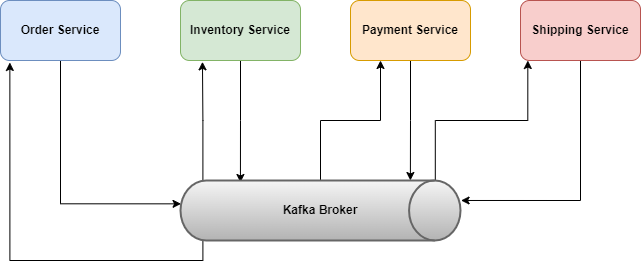
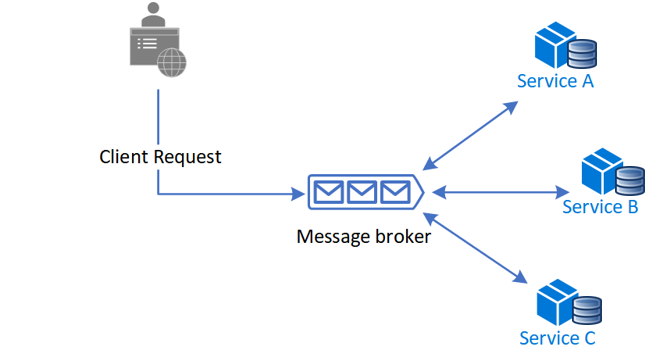


Diagram from Microsoft



We have the following microservices.

**Order Service**, **Inventory Service**, **Payment Service** and **Shipping Service**

We consider the following use cases.

1. An order is created by order microservice.
2. Order is received by Inventory service to check whether item is available/fulfilled or item is out of stock.
   1. If item is available – Success Scenario
   2. If item is out of stock – Failure scenario – Initiates a compensatory action to rollback the operation
3. Inventory service passes the event to Payment Service for payment initiation through Kafka Topic.
   1. If payment is successful – Success Scenario.
   2. If payment is unsuccessful - Failure scenario – Initiates a compensatory action to rollback the operation
4. Payment service passes the event to Shipping Service to ship through Kafka Topic.
   1. If the address is valid – Success Scenario
   2. If the address is invalid - Failure scenario – Initiates a compensatory action to rollback the operation

The json message structure is given below for various scenarios.

**All Good - Successful**

{

"orderId": 123,

"orderName": "Vivo Mobile",

"orderPrice": 13000,

"shipToAddress": "Bengaluru"

}

**Failure Scenario 1 - Vivo Mobile - Inventory Service To Fail**

{

"orderId": 123,

"orderName": "Vivo Mobile",

"orderPrice": 13000,

"shipToAddress": "Bengaluru"

}

**Failure Scenario 2 - Insufficient Fund- Payment Service To Fail**

{

"orderId": 123,

"orderName": "Samsung Mobile",

"orderPrice": 63000,

"shipToAddress": "Bengaluru"

}

**Failure Scenario 2 - Invalid Address- Shipping Service To Fail**

{

"orderId": 123,

"orderName": "Samsung Mobile",

"orderPrice": 13000,

"shipToAddress": "Invalid Bengaluru"

}

**Kafka Topics to be created**

kafka-topics.bat --create --topic **saga-choreo-order-topic** --bootstrap-server localhost:9092

kafka-topics.bat --create --topic **saga-inventory-out-topic** --bootstrap-server localhost:9092

kafka-topics.bat --create --topic **saga-choreo-shipping-topic** --bootstrap-server localhost:9092

kafka-topics.bat --create --topic **saga-choreo-order-complete-topic** --bootstrap-server localhost:9092

kafka-topics.bat --create --topic **saga-choreo-order-cancel-topic** --bootstrap-server localhost:9092

kafka-topics.bat --create --topic **saga-choreo-payment-cancel-topic** --bootstrap-server localhost:9092

kafka-topics.bat --create --topic **saga-cancel-ship-topic** --bootstrap-server localhost:9092

**saga-order-service**

**Controller**

@RestController

**public** **class** OrderController {

@Autowired

**private** OrderServiceImpl orderService;

@PostMapping(path = "/order")

**public** ResponseEntity<String> placeOrder(@RequestBody OrderEntity order) {

orderService.placeOrder(order);

String msg = "Order successfully placed ...";

**return** **new** ResponseEntity<String>(msg, HttpStatus.***CREATED***);

}

}

@Data

**public** **class** OrderEntity { 🡸 **Entity**

**private** **long** orderId;

**private** String orderName;

**private** **int** orderPrice;

**private** String shipToAddress;

}

**Create Order Event**

@Data

**public** **class** OrderEvent {

**private** **long** orderId;

**private** String actionName;

**private** String orderName;

**private** **int** orderPrice;

**private** String shipToAddress;

}

**Publish the Event**

@Service

**public** **class** OrderServiceImpl {

@Autowired

**private** KafkaTemplate<String, OrderEvent> kafkaTemplate;

@Value("${kafka.order.topic.name}")

**private** String orderTopicName;

**public** **void** placeOrder(OrderEntity order) {

OrderEvent orderEvent = **new** OrderEvent();

orderEvent.setOrderId(order.getOrderId());

orderEvent.setOrderName(order.getOrderName());

orderEvent.setActionName("ORDER\_PLACED");

orderEvent.setOrderPrice(order.getOrderPrice());

orderEvent.setShipToAddress(order.getShipToAddress());

// Publish

kafkaTemplate.send(orderTopicName, orderEvent);

System.***out***.println("Order sent to topic ...");

}

}

There are two Listeners, if the order is successfully completed and if the order is cancelled.

@Component

**public** **class** OrderListener { 🡸 Successful order

@KafkaListener(topics = "${kafka.order.complete.topic.name}")

**public** **void** listenOrder(OrderEvent orderEvent) {

**if** (orderEvent.getActionName().equalsIgnoreCase("ORDER\_COMPLETE")) {

System.***out***.println("Your order has been successfully processed, you will receive in 2 days");

}

}

}

Create a cancel Event and listen to it

@Data

**public** **class** CancelEvent {

**private** **long** orderId;

**private** String actionName;

**private** String orderName;

**private** String reason;

}

@Component

**public** **class** CancelOrderListener { 🡸 **For cancellation of order**

@KafkaListener(topics = "${kafka.order.cancel.topic.name}")

**public** **void** listenOrder(CancelEvent cancelEvent) {

**if**(cancelEvent.getActionName().equalsIgnoreCase("CANCEL\_ORDER")) {

System.***out***.println("Your order has been cancelled. Reason: "+cancelEvent.getReason());

}

}

}

**Configuration – application.properties**

server.port=8081

spring.application.name=saga-order-service

spring.profiles.active=dev

# Kafka Consumer

spring.kafka.consumer.bootstrap-servers=localhost:9092

spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer

spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer

spring.kafka.consumer.group-id=saga-order-grp-id

spring.kafka.consumer.auto-offset-reset=latest

# The below line is important for Json Deserialization

spring.kafka.consumer.properties.spring.json.trusted.packages=\*

#Kafka Producer

spring.kafka.producer.bootstrap-servers=localhost:9092

spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer

spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer

# Open API Swagger documentation

springdoc.swagger-ui.path=/index.html

springdoc.swagger-ui.disable-swagger-default-url=true

#Application Specific

kafka.order.topic.name=saga-choreo-order-topic

# Listen from this Topic

kafka.order.complete.topic.name=saga-choreo-order-complete-topic

kafka.order.cancel.topic.name=saga-choreo-order-cancel-topic

**Inventory Service**

Inventory service has a listener who listens whether order has come to inventory service.

@Component

**public** **class** PlacedOrderListener {

@Autowired

**private** InventoryServiceImpl inventoryService;

@KafkaListener(topics = "${kafka.order.topic.name}", groupId = "saga-order-grp-id")

**public** **void** listenOrderPlaced(OrderEvent orderEvent) {

System.***out***.println("Placed Order has come to inventory ...");

System.***out***.println("Order Details:" + orderEvent);

**if**(orderEvent.getActionName().equalsIgnoreCase("ORDER\_PLACED")) {

// Make next call to initiate the payment

**boolean** itemAvlable = isItemAvailable(orderEvent.getOrderName());

System.***out***.println("orderEvent.getShipToAddress()--->"+orderEvent.getShipToAddress());

**if**(itemAvlable) {

// Initiate for Payment

ItemEvent itemEvent = **new** ItemEvent();

itemEvent.setOrderId(orderEvent.getOrderId());

itemEvent.setItemName(orderEvent.getOrderName());

itemEvent.setPrice(orderEvent.getOrderPrice());

itemEvent.setActionName("RECEIVE\_PAYMENT");

itemEvent.setShipToAddress(orderEvent.getShipToAddress());

inventoryService.sendForPaymentInitiation(itemEvent);

} **else** {

CancelEvent cancelEvent = **new** CancelEvent();

cancelEvent.setActionName("CANCEL\_ORDER");

cancelEvent.setReason("Outof Stock");

inventoryService.compensate(cancelEvent);

}

}

}

**private** **boolean** isItemAvailable(String name) {

**if**(name.startsWith("Vivo")) **return** **false**;

**return** **true**;

}

}

Inventory service either publishes for payment or initiates a compensatory operation for failure.

@Service

**public** **class** InventoryServiceImpl { 🡸 Service Implementation class

@Autowired

**private** KafkaTemplate<String, ItemEvent> kafkaTemplate;

@Autowired

**private** KafkaTemplate<String, CancelEvent> cancelKafkaTemplate;

@Value("${kafka.inventory.out.topic.name}")

**private** String topicName;

@Value("${kafka.order.cancel.topic.name}")

**private** String cancelTopicName;

**public** **void** sendForPaymentInitiation(ItemEvent itemEvent) {

System.***out***.println("Initiating payment for the item");

kafkaTemplate.send(topicName, itemEvent);

System.***out***.println("Sent for payment initiation ...");

}

**public** **void** compensate(CancelEvent cancelEvent) {

System.***out***.println("This item is currenlty out of stock, hence order is cancelled...");

cancelKafkaTemplate.send(cancelTopicName, cancelEvent);

}

**public** **void** cancelOrder(CancelEvent cancelEvent) {

System.***out***.println("Order has been cancelled and item is put back in the inventory");

cancelKafkaTemplate.send(cancelTopicName, cancelEvent);

}

}

Inventory service has also a listener for failures.

@Component

**public** **class** FailedOrderListener {

@Autowired

**private** InventoryServiceImpl inventoryService;

@KafkaListener(topics = "${kafka.payment.cancel.topic.name}")

**public** **void** cancelOrder(CancelEvent cancelEvent) {

System.***out***.println("Failed Order Listener: "+cancelEvent);

cancelEvent.setActionName("CANCEL\_ORDER");

inventoryService.cancelOrder(cancelEvent);

}

}

**Configuration – application.properties**

server.port=8082

spring.profiles.active=dev

# Kafka Consumer

spring.kafka.consumer.bootstrap-servers=localhost:9092

spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer

spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer

spring.kafka.consumer.group-id=saga-order-grp-id

spring.kafka.consumer.auto-offset-reset=latest

# The below line is important for Json Deserialization

spring.kafka.consumer.properties.spring.json.trusted.packages=\*

# Kafka Producer

spring.kafka.producer.bootstrap-servers=localhost:9092

spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer

spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer

# Open API Swagger Documentation

springdoc.swagger-ui.path=/index.html

springdoc.swagger-ui.disable-swagger-default-url=true

#Application Specific

kafka.order.topic.name=saga-choreo-order-topic

kafka.inventory.out.topic.name=saga-inventory-out-topic

# To cancel the order

kafka.order.cancel.topic.name=saga-choreo-order-cancel-topic

kafka.payment.cancel.topic.name=saga-choreo-payment-cancel-topic

**Payment Service**

Payment service has also listener whether payment to be initiated or order to be cancelled. In case of payment failure, it initiates a compensatory operation.

@Component

**public** **class** PaymentListener {

@Autowired

**private** PaymentServiceImpl paymentService;

@KafkaListener(topics = "${kafka.inventory.out.topic.name}")

**public** **void** receivePayment(ItemEvent itemEvent) {

**if** (itemEvent.getActionName().equals("RECEIVE\_PAYMENT")) {

**boolean** isReceived = isPaymentReceived(itemEvent.getPrice());

System.***out***.println("itemEvent.getShipToAddress()---->" + itemEvent.getShipToAddress());

**if** (isReceived) {

System.***out***.println("Your payment has been successfull ...");

// Confirm for shipping

ShippingEvent shipEvent = **new** ShippingEvent();

shipEvent.setActionName("SHIP\_ITEM\_ADDRESS");

shipEvent.setItemName(itemEvent.getItemName());

shipEvent.setOrderId(itemEvent.getOrderId());

shipEvent.setShippingAdress(itemEvent.getShipToAddress());

paymentService.shipToAddress(shipEvent);

System.***out***.println("Your package is ready to be shipped...");

} **else** {

// Insufficient balance

CancelEvent cancelEvent = **new** CancelEvent();

cancelEvent.setOrderId(itemEvent.getOrderId());

cancelEvent.setActionName("CANCEL\_ORDER");

cancelEvent.setReason("Insufficient fund...");

cancelEvent.setOrderName(itemEvent.getItemName());

paymentService.cancelOrder(cancelEvent);

System.***out***.println("Order has been cancelled due to insufficient fund...");

}

}

}

**public** **boolean** isPaymentReceived(**int** amount) {

**if** (amount > 50000)

**return** **false**;

**else**

**return** **true**;

}

}

Service Implementation Class

@Service

**public** **class** PaymentServiceImpl {

@Autowired

**private** KafkaTemplate<String, ShippingEvent> kafkaTemplate;

@Value("${kafka.shipping.out.topic.name}")

**private** String topicName;

@Autowired

**private** KafkaTemplate<String, CancelEvent> cancelKafkaTemplate;

@Value("${kafka.payment.cancel.topic.name}")

**private** String cancelTopicName;

**public** **void** shipToAddress(ShippingEvent shipEvent) {

kafkaTemplate.send(topicName, shipEvent);

}

**public** **void** cancelOrder(CancelEvent cancelEvent) {

cancelKafkaTemplate.send(cancelTopicName, cancelEvent);

}

**public** **void** reverseTransaction(CancelEvent cancelEvent) {

System.***out***.println("Your transaction has been reversed, you will get back your amount in 48 hours...");

cancelOrder(cancelEvent);

}

}

**Listener whether Shipping has been cancelled**

@Component

**public** **class** CancelShipListener {

@Autowired

**private** PaymentServiceImpl paymentService;

@KafkaListener(topics = "${kafka.cancel.ship.topic.name}")

**public** **void** listen(CancelEvent cancelEvent) {

**if**(cancelEvent.getActionName().equals("CANCEL\_PAYMENT")) {

paymentService.reverseTransaction(cancelEvent);

}

}

}

**Configuration – application.properties**

server.port=8083

spring.profiles.active=dev

# Kafka Consumer

spring.kafka.consumer.bootstrap-servers=localhost:9092

spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer

spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer

spring.kafka.consumer.group-id=saga-order-grp-id

spring.kafka.consumer.auto-offset-reset=latest

# The below line is important for Json Deserialization

spring.kafka.consumer.properties.spring.json.trusted.packages=\*

# Kafka Producer

spring.kafka.producer.bootstrap-servers=localhost:9092

spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer

spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer

# Open API Swagger Documentation

springdoc.swagger-ui.path=/index.html

springdoc.swagger-ui.disable-swagger-default-url=true

#Application Specific Configuration

kafka.shipping.out.topic.name=saga-choreo-shipping-topic

# Listen from this topic

kafka.inventory.out.topic.name=saga-inventory-out-topic

kafka.payment.cancel.topic.name=saga-choreo-payment-cancel-topic

kafka.cancel.ship.topic.name=saga-cancel-ship-topic

**Shipping Service**

Shipping service has a listener whether to ship the product to cancel the product so that whole operation can be rolled back.

**Listener in Shipping Service**

@Component

**public** **class** ShippingListener {

@Autowired

**private** ShippingServiceImpl shipService;

@KafkaListener(topics = "${kafka.shipping.out.topic.name}")

**public** **void** receiveItemForShipping(ShippingEvent shipEvent) {

System.***out***.println("shipEvent.getShippingAdress()--->" + shipEvent.getShippingAdress());

**boolean** invalidShipAdres = shipEvent.getShippingAdress().startsWith("Invalid");

System.***out***.println("invalidShipAdres : " + invalidShipAdres);

**boolean** validAction = shipEvent.getActionName().equalsIgnoreCase("SHIP\_ITEM\_ADDRESS");

System.***out***.println("validAction : " + validAction);

**if** (validAction && !invalidShipAdres) {

System.***out***.println("Your item has been received by the courier company...");

OrderEvent orderEvent = **new** OrderEvent();

orderEvent.setOrderId(shipEvent.getOrderId());

orderEvent.setActionName("ORDER\_COMPLETE");

shipService.completeOrder(orderEvent);

System.***out***.println("Package has been successfully shipped to your address...");

} **else** {

System.***out***.println("Your order will be cancelled...");

// Cancel shipping

CancelEvent cancelEvent = **new** CancelEvent();

cancelEvent.setActionName("CANCEL\_PAYMENT");

cancelEvent.setOrderId(shipEvent.getOrderId());

cancelEvent.setOrderName(shipEvent.getItemName());

cancelEvent.setReason("This item does not ship to address: " + shipEvent.getShippingAdress());

shipService.cancelOrder(cancelEvent);

}

}

}

**Service Implementation class**

@Service

**public** **class** ShippingServiceImpl {

@Autowired

**private** KafkaTemplate<String, OrderEvent> kafkaTemplate;

@Value("${kafka.order.complete.topic.name}")

**private** String topicName;

@Autowired

**private** KafkaTemplate<String, CancelEvent> cancelKafkaTemplate;

@Value("${kafka.cancel.ship.topic.name}")

**private** String cancelTopicName;

**public** **void** completeOrder(OrderEvent orderEvent) {

kafkaTemplate.send(topicName, orderEvent);

System.***out***.println("Your order processing has been complete");

}

**public** **void** cancelOrder(CancelEvent cancelEvent) {

cancelKafkaTemplate.send(cancelTopicName, cancelEvent);

System.***out***.println("Your order processing has been cancelled due to invalid address");

}

}

**Configuration – application.properties**

server.port=8084

spring.profiles.active=dev

# Kafka Consumer

spring.kafka.consumer.bootstrap-servers=localhost:9092

spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer

spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer

spring.kafka.consumer.group-id=saga-order-grp-id

spring.kafka.consumer.auto-offset-reset=latest

# The below line is important for Json Deserialization

spring.kafka.consumer.properties.spring.json.trusted.packages=\*

# Kafka Producer

spring.kafka.producer.bootstrap-servers=localhost:9092

spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer

spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer

# Kafka Swagger Documentation

springdoc.swagger-ui.path=/index.html

springdoc.swagger-ui.disable-swagger-default-url=true

#Application Specific Configuration

kafka.shipping.out.topic.name=saga-choreo-shipping-topic

kafka.order.complete.topic.name=saga-choreo-order-complete-topic

kafka.cancel.ship.topic.name=saga-cancel-ship-topic

**Advantages of Choreography Pattern**

* Good for simple workflows that require few participants and don't need a coordination logic.
* Doesn't require additional service implementation and maintenance.
* Doesn't introduce a single point of failure, since the responsibilities are distributed across the saga participants.

**Disadvantages of Choreography Pattern**

* Workflow can become confusing when adding new steps, as it's difficult to track which saga participants listen to which commands.
* There's a risk of cyclic dependency between saga participants because they have to consume each other's commands.
* Integration testing is difficult because all services must be running to simulate a transaction.