



[Architecture](#) ▾

[Development](#) ▾

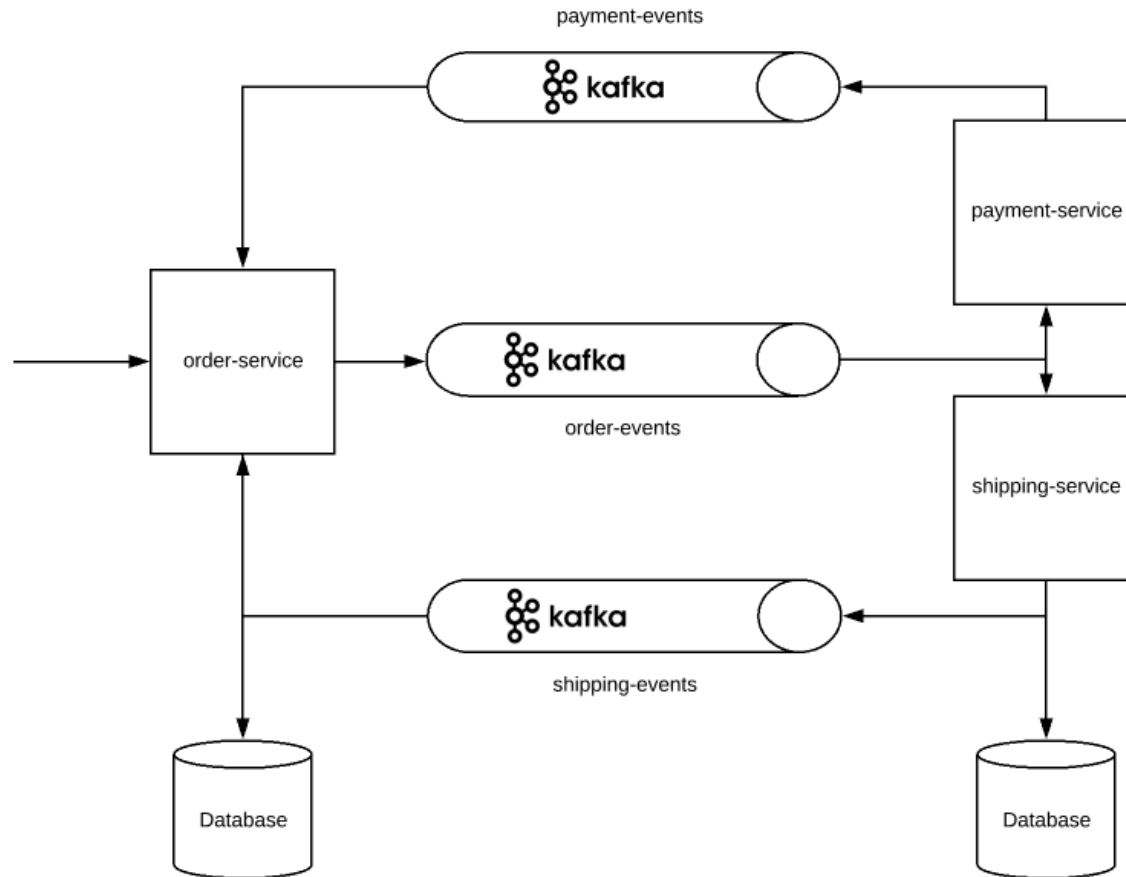
[DevOps](#) ▾

[Test Automation](#) ▾

[Downloads](#)

[About Me](#)

[Topics](#)



Architectural Pattern – Choreography Saga Pattern With Spring Boot + Kafka

5 Comments / Architectural Design Pattern, Architecture, Articles, Data Stream / Event Stream, Design Pattern, Framework, Java, Kafka, Kubernetes Design Pattern, MicroService, Reactive Programming, Reactor, Spring, Spring Boot, Spring WebFlux / By vlns / April 25, 2020



Overview:

Over the years, MicroServices have become very popular. MicroServices are distributed systems. They are smaller, modular, easy to deploy and scale etc. Developing a single microservice application might be interesting! But handling a business transaction which spans across multiple microservices is not fun! MicroService architectures have specific responsibilities. In order to complete an application workflow / a task, multiple MicroServices might have to work together.

Lets see how difficult it could be in dealing with transactions / data consistency in the distributed systems in this article.

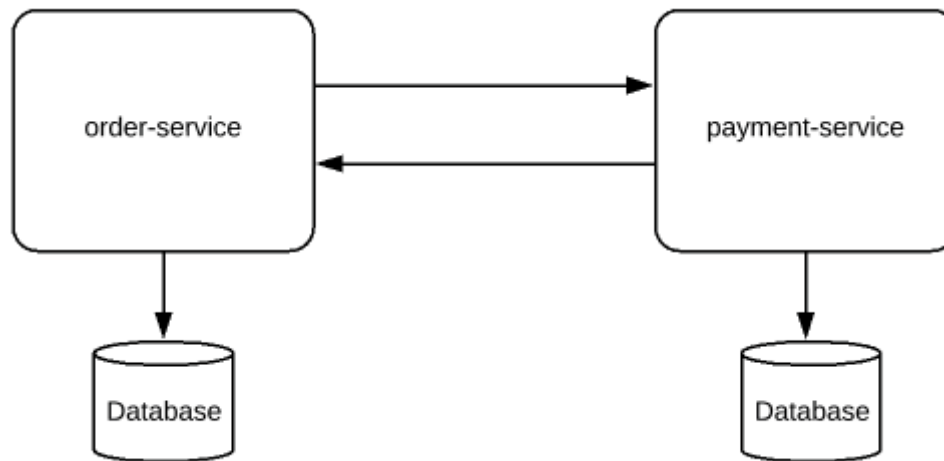
Challenges:

Let's assume that our business rule says, when an user places an order, order will be fulfilled if the product's price is within the user's credit limit/balance. Otherwise it will not be fulfilled. It looks really simple.

This is very easy to implement in a monolith application. The entire workflow can be considered as 1 single transaction. It is easy to commit / rollback when everything is in a single DB. With distributed systems with

multiple databases, It is going to be very complex! Lets look at our architecture first to see how to implement this.

We have an order-service with its own database which is responsible for order management. Similarly we also have payment-service which is responsible for managing payments. So the order-service receives the request and checks with the payment-service if the user has the balance. If the payment-service responds OK, order-service completes the order and payment-service deducts the payment. Otherwise, the order-service cancels the order. For a very simple business requirement, here we have to send multiple requests across the network.



In the traditional system design approach, order-service simply sends a HTTP request to get the information about the user's credit balance. The problem with this approach is order-service assumes that payment-service will be up and running always. Any network issue or performance issue at the payment-service will be propagated to the order-service. It could lead to poor user-experience & we also might lose revenue. Let's see how we could handle transactions in the distributed systems with loose coupling by using a pattern called **Saga** with [Event Sourcing](#) approach.

We had also discussed designing resilient microservices using below patterns. Take a look at them if you are interested.

[01 – Spring Boot – Resilient MicroService Design – Timeout Pattern](#)

[02 – Spring Boot – Resilient MicroService Design – Retry Pattern](#)

[03 – Spring Boot – Resilient MicroService Design – Circuit Breaker Pattern](#)

[04 – Spring Boot – Resilient MicroService Design – Bulkhead Pattern](#)

[05 – Spring Boot – Resilient MicroService Design – Rate Limiter Pattern](#)

Saga Pattern:

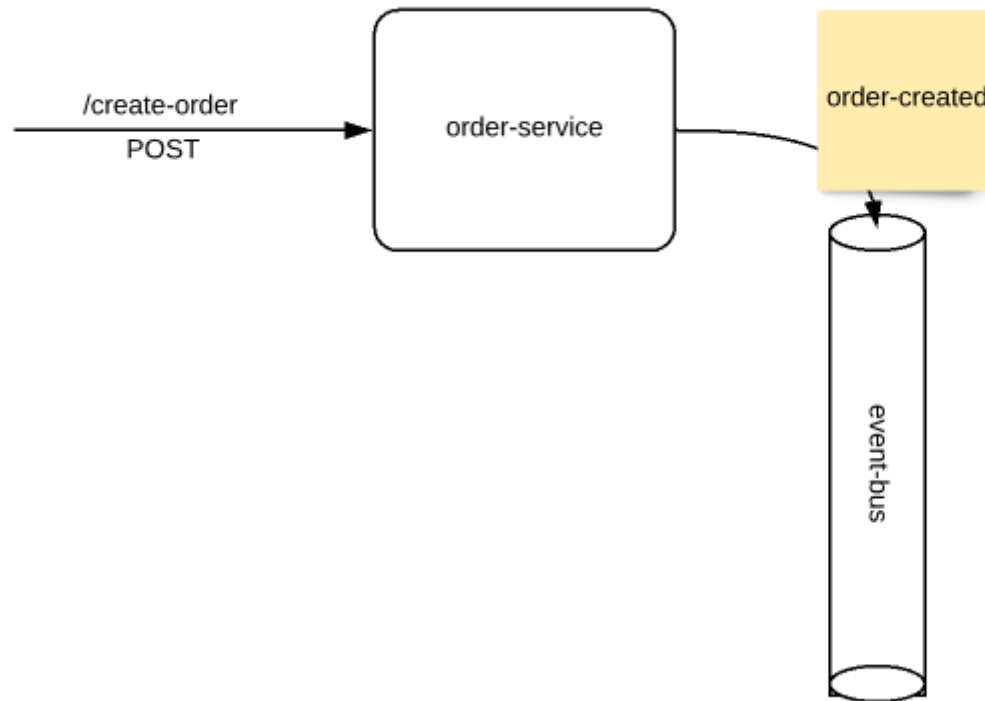
Each business transaction which spans multiple microservices are split into micro-service specific local transactions and they are executed in a sequence to complete the business workflow. It is called Saga. It can be implemented in 2 ways.

- Choreography approach
- Orchestration approach

In this article, we will be discussing the choreography based approach by using **event-sourcing**. For orchestration based Saga, check [here](#).

Event-Sourcing:

In this approach every change to the state of an application is captured as an **event**. This event is stored in the database (for tracking purposes) and is also published in the event-bus for other parties to consume.



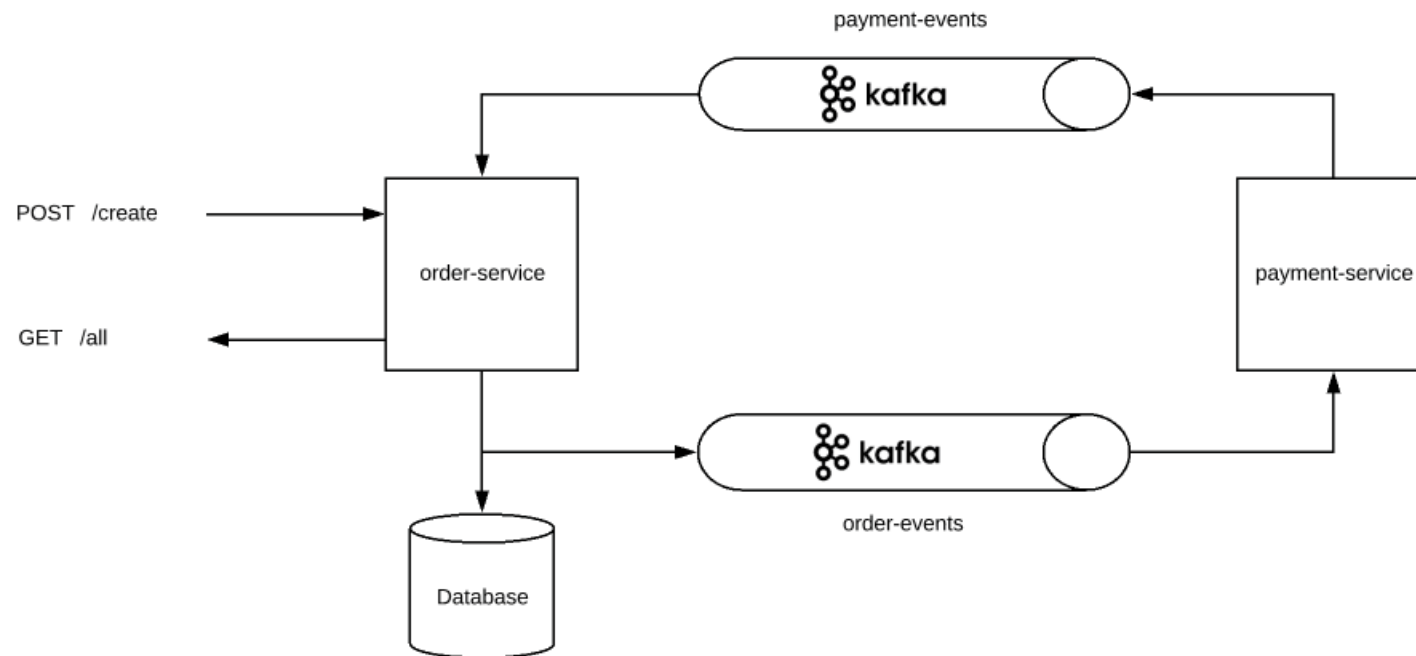
The order-service receives a command to create a new order. This request is processed and raised as an order-created event. Couple of things to note here.

1. Order created event basically informs that a new order request has been received and kept in the PENDING status by order-service. It is not yet fulfilled.
2. The event object will be named in the past tense always as it already happened!

Now the payment-service could be interested in listening to those events and approve/reject payment. Even these could be treated as an event. Payment approved/rejected event. Order-service might listen to these events and fulfill / cancel the order request it had originally received.

This approach has many advantages.

- There is no service dependency. Payment-service does not have to be up and running always.
- Loose coupling
- Horizontal scaling
- Fault tolerant



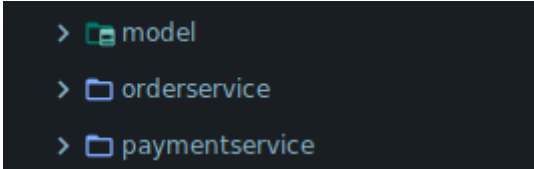
The business workflow is implemented as shown here.

- order-services receives a POST request for a new order
- It places an order request in the DB in the ORDER_CREATED state and raises an event
- payment-service listens to the event, confirms about the credit reservation

- order-service fulfills order or rejects the order based on the credit reservation status.

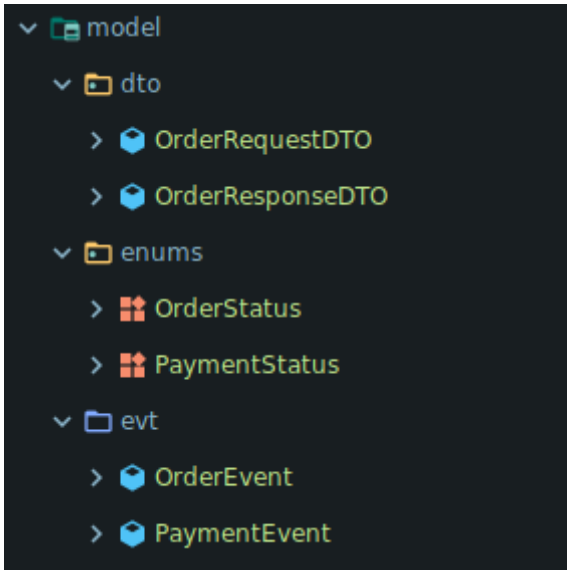
Implementation:

- I created a simple spring-boot project using kafka-cloud-stream. I have basic project structure like this.



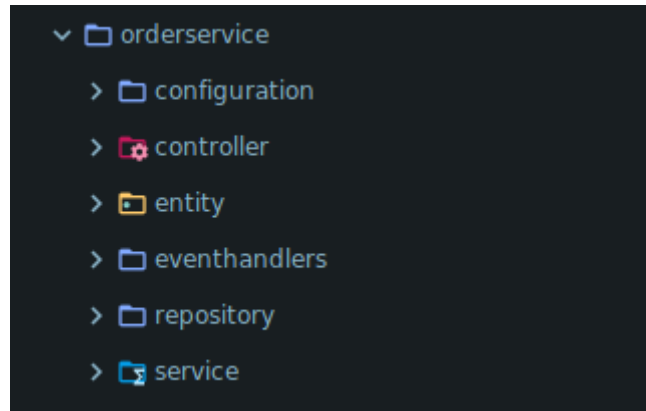
```
> model
> orderservice
> paymentservice
```

- My model package is as shown below.
 - It contains the basic DTOs, Enums and Event objects.

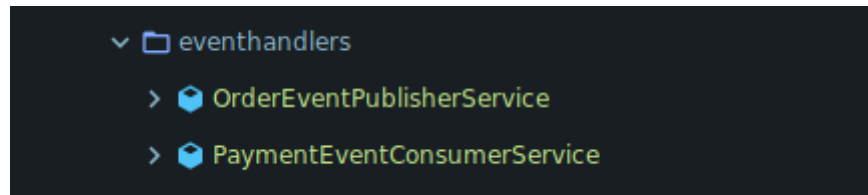


```
▼ model
  ▼ dto
    > OrderRequestDTO
    > OrderResponseDTO
  ▼ enums
    > OrderStatus
    > PaymentStatus
  ▼ evt
    > OrderEvent
    > PaymentEvent
```

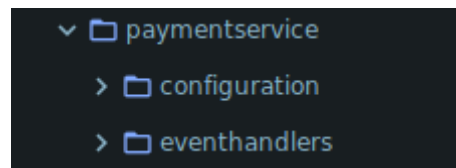
- My order-service project structure is as shown below.



- I have 2 different event handlers here in my order-service
 - one for publishing order related events
 - another one for consuming payment events



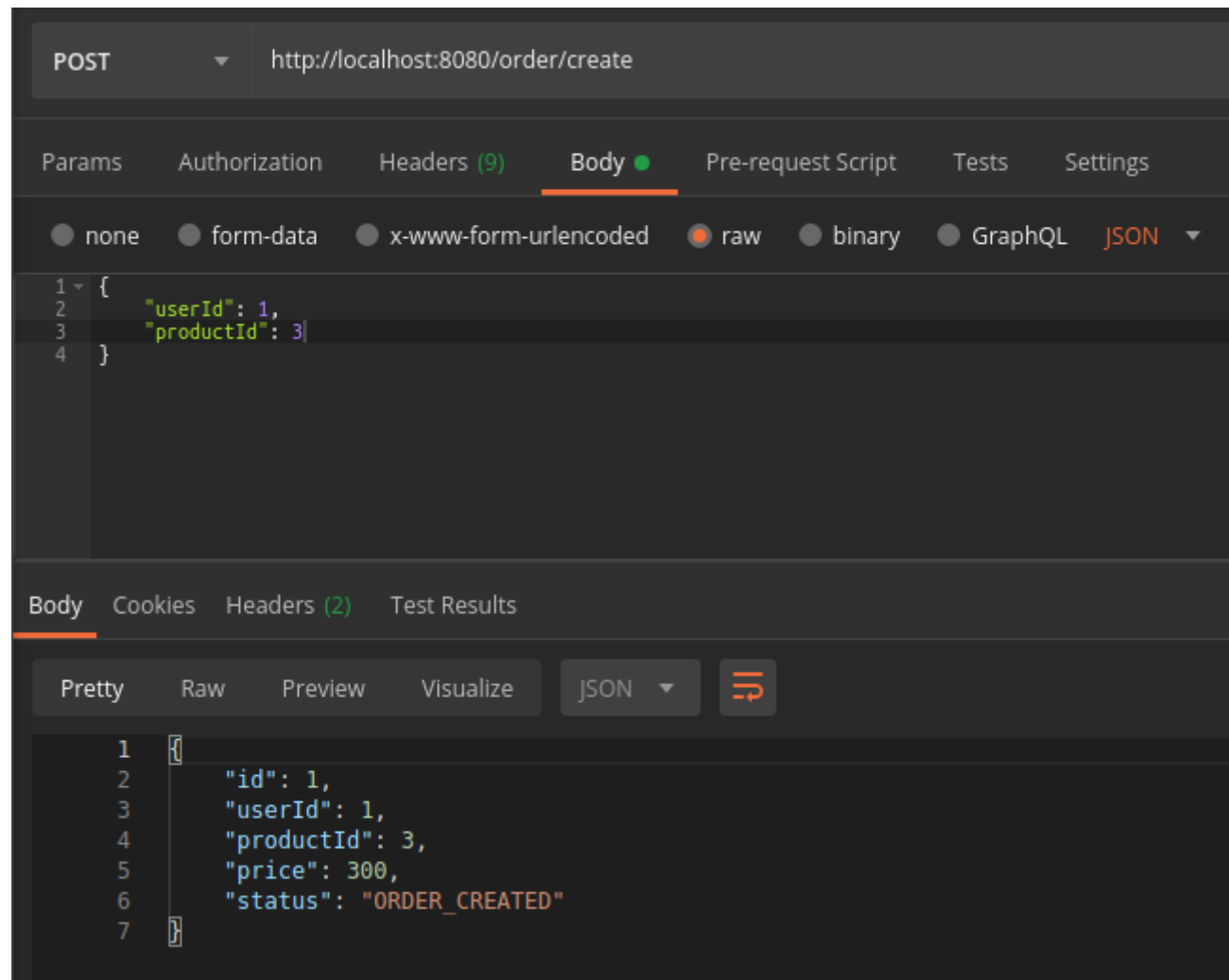
- The payment-service is very simple in my example.



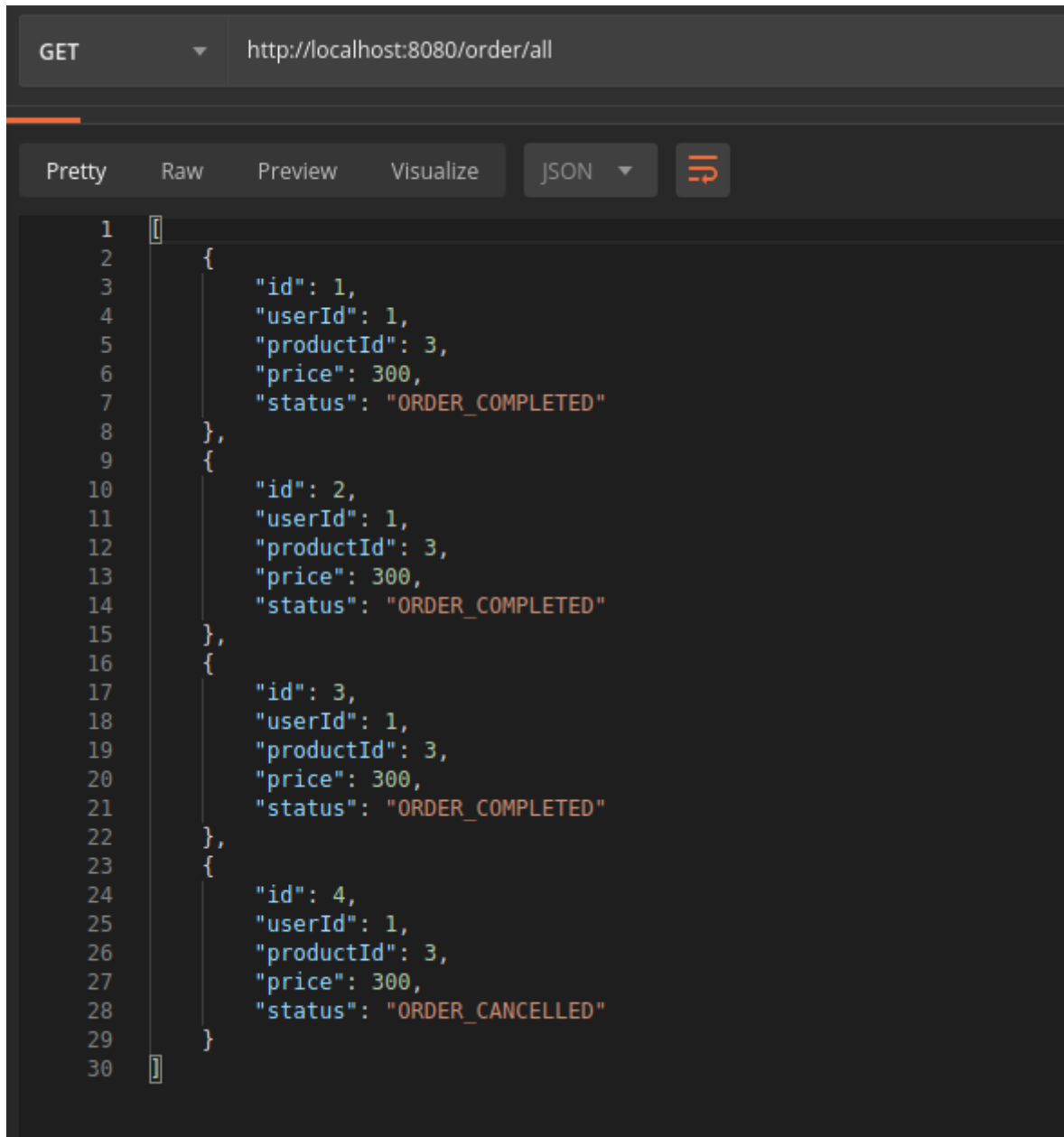
I have a complete working project [here](#). You can clone the code and run to follow the demo here. Ensure that you have a local kafka cluster up and running. You can use this [docker-compose file](#) for running a local kafka cluster.

Demo:

- Once I start my application, I send an order request. The productId=3 costs \$300 and user's credit limit is \$1000.
- As soon as I send a request, I get the immediate response saying the order_created / order_pending.
- I send 4 requests.



- If I send /order/all requests to see all the orders, I see that 3 orders in fulfilled and the 4th order in cancelled status as the user does not have enough balance to fulfill the 4th order.

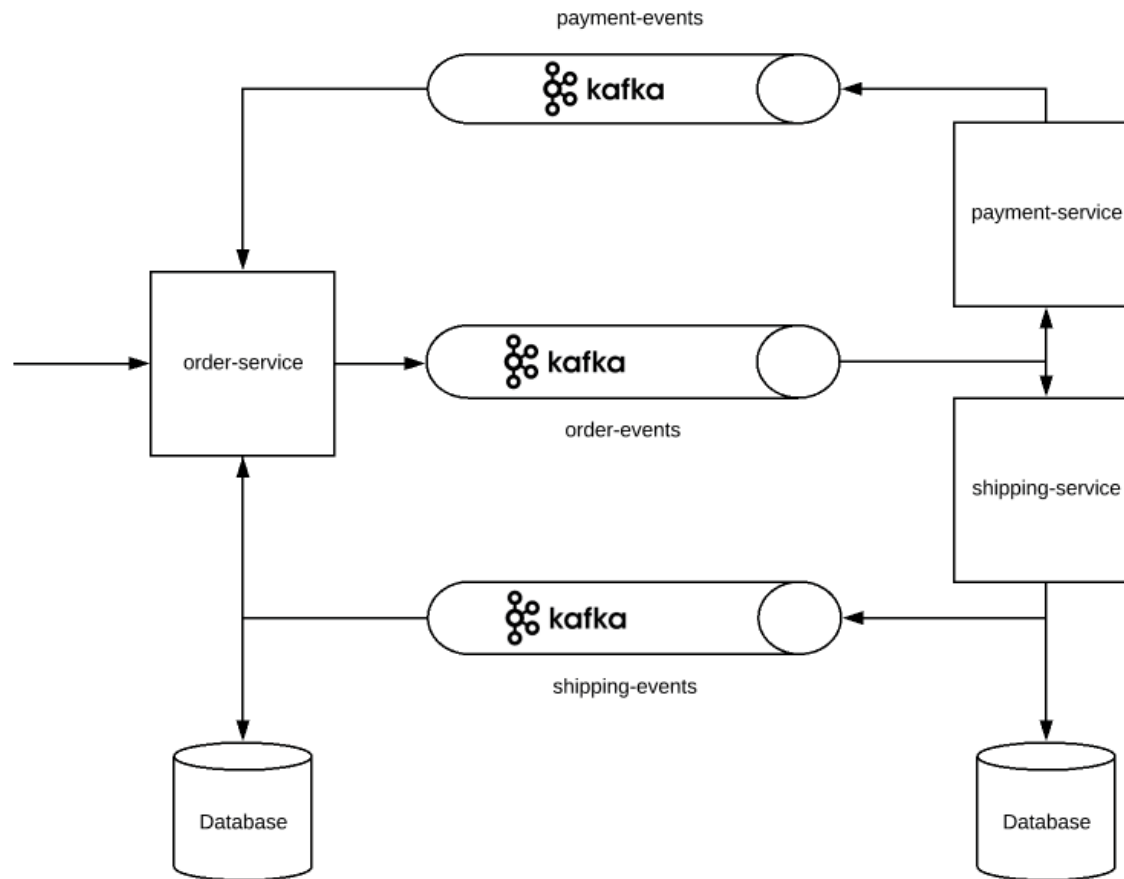


```
GET http://localhost:8080/order/all

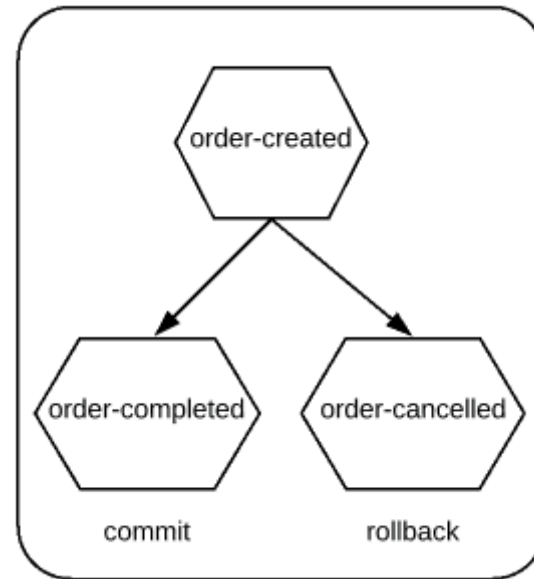
Pretty Raw Preview Visualize JSON ↺

1  [
2    {
3      "id": 1,
4      "userId": 1,
5      "productId": 3,
6      "price": 300,
7      "status": "ORDER_COMPLETED"
8    },
9    {
10     "id": 2,
11     "userId": 1,
12     "productId": 3,
13     "price": 300,
14     "status": "ORDER_COMPLETED"
15   },
16   {
17     "id": 3,
18     "userId": 1,
19     "productId": 3,
20     "price": 300,
21     "status": "ORDER_COMPLETED"
22   },
23   {
24     "id": 4,
25     "userId": 1,
26     "productId": 3,
27     "price": 300,
28     "status": "ORDER_CANCELLED"
29   }
30 ]
```

- We can add additional services in the same way. For ex: once order-service fulfills the order and raises an another event. A shipping service listening to the event and takes care of packing and shipping to the given user address. Order-service might again listen to those events updates its DB to the order_shipped status.



As we had mentioned earlier, committing/rolling back a transaction which spans multiple microservices is very challenging. Each service should have the event-handlers, for committing / rolling back the transaction to maintain the data consistency, for every event it is listening to!



Happy coding 😊

Share This:

5 thoughts on “Architectural Pattern – Choreography Saga Pattern With Spring Boot + Kafka”



srinivas

May 13, 2020 at 8:12 AM

Thanks for this. I have couple of query's on this

- 1) you have created payment and order service in one component, it should be separate components right?
- 2) Let's say, we have order and workflow services, and the workflow id in order, think that workflow created successfully and while creating order it failed. In this case, we have to delete workflow entry, how we can handle this scenario?

Reply



vlns

May 13, 2020 at 1:38 PM

Hi,

1. You are right. In real life, order and payment services should be 2 different microservices. Just for this article, I had placed them in 2 different packages to quickly run and demo. I could have used multi-module maven project.
2. This is a great question. MicroServices communicate among themselves by raising an event. In this case, when the payment failed, order-service cancelled the order. Similarly, in your case, the workflow might have been created successfully. But when the order fails for some reason, it has to raise an event. Workflow service will consume the event and cancel the status / delete etc.

Reply



srinivas

May 13, 2020 at 5:05 PM

Thanks for the quick reply. When comes to delete case:

- 1) Delete event can be raised either if the transaction is not successful or for any run time exception from catch block, right?
- 2) Is there any mechanism instead of raising a delete event, the publisher gets the response back, if failure we can delete?

Reply



Shivam Kumar

July 16, 2020 at 5:30 AM

Hi Vins,

I want to remove kafka with solace it that possible?

Reply



vins

July 16, 2020 at 3:06 PM

We could use any streaming platform like kafka, redis, pulsar..etc. I used Kafka just for this demo.

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

☐ Save my name, email, and website in this browser for the next time I comment.

☐ Notify me of follow-up comments by email.

☐ Notify me of new posts by email.

Post Comment

search..



Recent Posts

NATS – Cloud Native

Messaging Server

Reactor – Parallel Flux

Architectural Pattern –

Orchestration Saga Pattern

With Spring Boot + Kafka

RSocket – Uploading Files

With Reactive Programming

RSocket – Integrating With

Spring Boot



Selenium WebDriver -
How To Test REST
API



Introducing PDFUtil -
Compare two PDF
files textually or
Visually



JMeter - How To Run
Multiple Thread
Groups in Multiple
Test Environments



Selenium WebDriver -
Design Patterns in

Test Automation -
Factory Pattern



Kafka Streams - Real-
Time Data Processing
Using Spring Boot



JMeter - Real Time
Results - InfluxDB &
Grafana - Part 1 - Basic
Setup



JMeter - Distributed
Load Testing using
Docker



JMeter - How To Test
REST API /
MicroServices



JMeter - Property File
Reader - A custom
config element



Selenium WebDriver -

How To Run

Automated Tests

Inside A Docker

Container - Part 1

Categories

Architecture (40)

Arquillian (9)

Articles (170)

AWS / Cloud (17)

AWS (4)

Best Practices (75)

CI / CD / DevOps (51)

Data Stream / Event Stream
(17)

Database (2)

Design Pattern (37)

Architectural Design Pattern
(22)

Factory Pattern (1)

Kubernetes Design Pattern (17)

Strategy Pattern (1)

Distributed Load Test (9)

Docker (23)

ElasticSearch (2)

Email Validation (1)

Framework (100)

Functional Test Automation
(83)

Puppeteer (1)

QTP (10)

Selenium (76)

Extend WebDriver (11)

Ocular (2)

Page Object Design (17)

Report (8)

Selenium Grid (10)

TestNG (7)
gRPC (7)
Java (46)
Guice (2)
Reactor (22)
Jenkins (17)
Kafka (9)
Kubernetes (6)
Maven (7)
messaging (1)
MicroService (52)
Monitoring (13)
FileBeat (1)
Grafana (5)
InfluxDB (7)
Kibana (2)
Multi Factor Authentication (2)
nats (1)
Performance Testing (43)
Extend JMeter (5)
JMeter (43)
Workload Model (2)

Little's Law (1)
Web Scraping (1)
Protocol Buffers (8)
Reactive Programming (26)
Redis (6)
rsocket (3)
Slack (2)
SMS (1)
Spring (46)
Spring Boot (39)
Spring WebFlux (39)
Udemy Courses (4)
Utility (20)