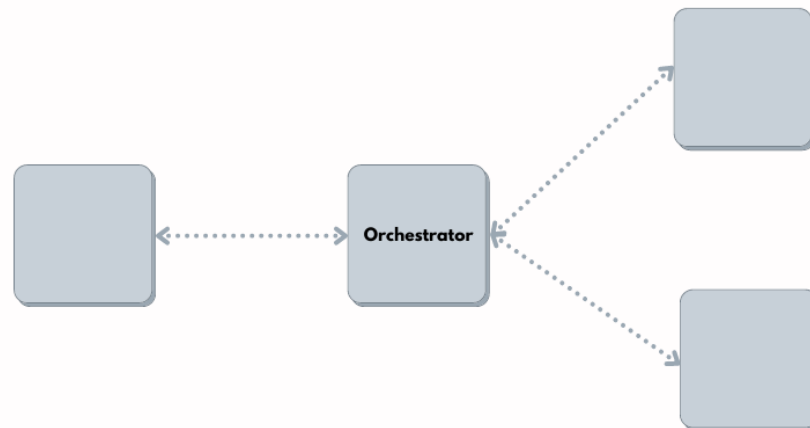


[Udemy Courses](#)[Architecture](#) ▾[Development](#) ▾[DevOps](#) ▾[Test Automation](#) ▾[Downloads](#)[About Me](#)[Topics](#)

Saga Pattern



Recent Posts

[Spring](#)[WebFlux](#)[Aggregation](#) ^

Orchestration Saga Pattern With Spring Boot

27 Comments / Architectural Design Pattern, Architecture, Articles, Data Stream / Event Stream, Design Pattern, Java, Kafka, Kubernetes Design Pattern, MicroService, Reactive Programming, Reactor, Spring, Spring Boot, Spring WebFlux / By vlns / January 30, 2022

Overview:


In this tutorial, I would like to show you a simple implementation of **Orchestration Saga Pattern with Spring Boot**.

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! In order to complete an application workflow / a task, multiple Microservices might have to work together.

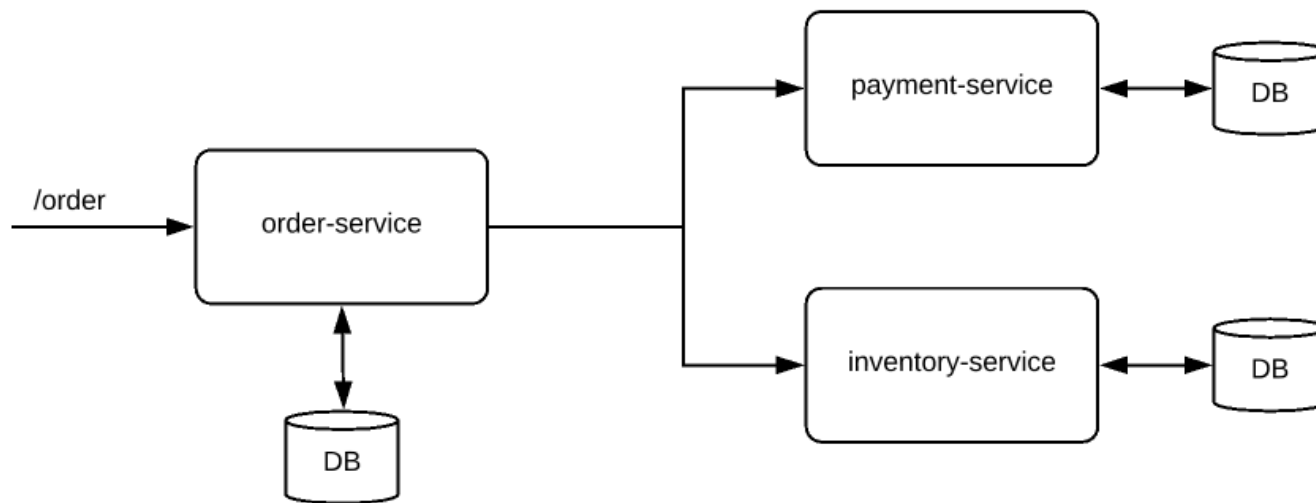
Let's see how difficult it could be in dealing with transactions / data consistency in the distributed systems in this article & how **Orchestration Saga Pattern** could help us.

A Simple Transaction:

Choreograph
hy Saga
Pattern With
Spring Boot
Spring
WebFlux
WebSocket
gRPC Web
Example
Orchestratio
n Saga
Pattern With
Spring Boot

 Selenium
WebDriv
er - How

Let's assume that our business rule says, when a user places an order, order will be fulfilled if the product's price is within the user's credit limit/balance & the inventory is available for the product. 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! Let's look at our architecture first to see how to implement this.



We have below Microservices with its own DB.

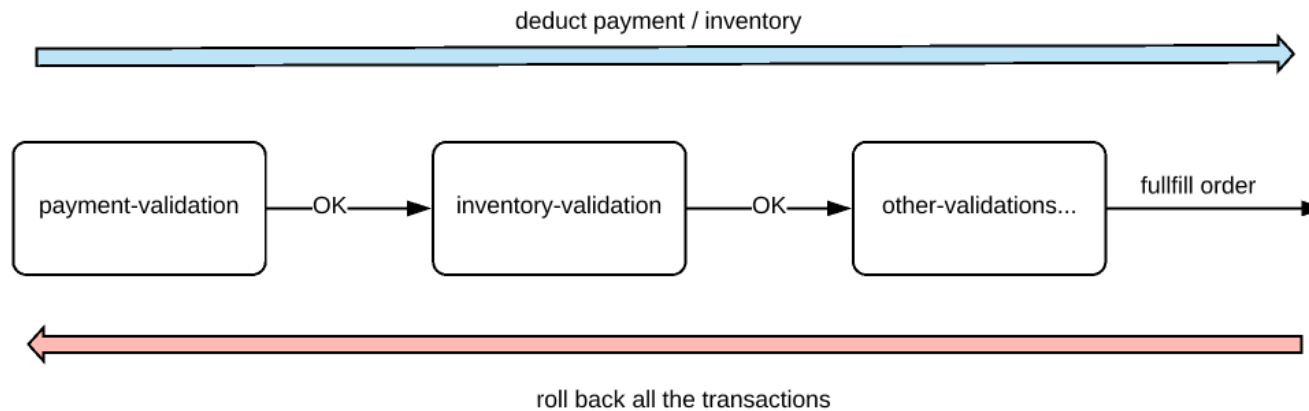
- **order-service**
- **payment-service**
- **inventory-service**

To Test
REST API

Introduci
ng
PDFUtil -
Compare
two PDF
files
textually
or
Visually

JMeter -
How To
Run
Multiple
Thread
Groups ii ^

When the order-service receives the request for the new order, It has to check with the payment-service & inventory-service. We deduct the payment, inventory and fulfill the order finally! What will happen if we deducted payment but if inventory is not available? How to roll back? It is difficult when multiple databases are involved.



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

Multiple
Test
Environm
ents



Selenium

WebDriv

er -

Design

Patterns

in Test

Automati

on -

Factory

Pattern



Kafka

Stream

With



In this article, we will be discussing the Orchestration based saga. For more information on Choreography based saga, check [here](#).

Orchestration Saga Pattern:

In this pattern, we will have an orchestrator, a separate service, which will be coordinating all the transactions among all the Microservices. If things are fine, it makes the order-request as complete, otherwise marks that as cancelled.

Let's see how we could implement this. Our sample architecture will be more or less like this.!

- In this demo, communication between orchestrator and other services would be a simple HTTP in a non-blocking asynchronous way to make this stateless.
- We can also use Kafka topics for this communication. For that we have to use [scatter/gather pattern](#) which is more of a stateful style.

Spring
Boot



JMeter -

Real

Time

Results -

InfluxDB

&

Grafana -

Part 1 -

Basic

Setup



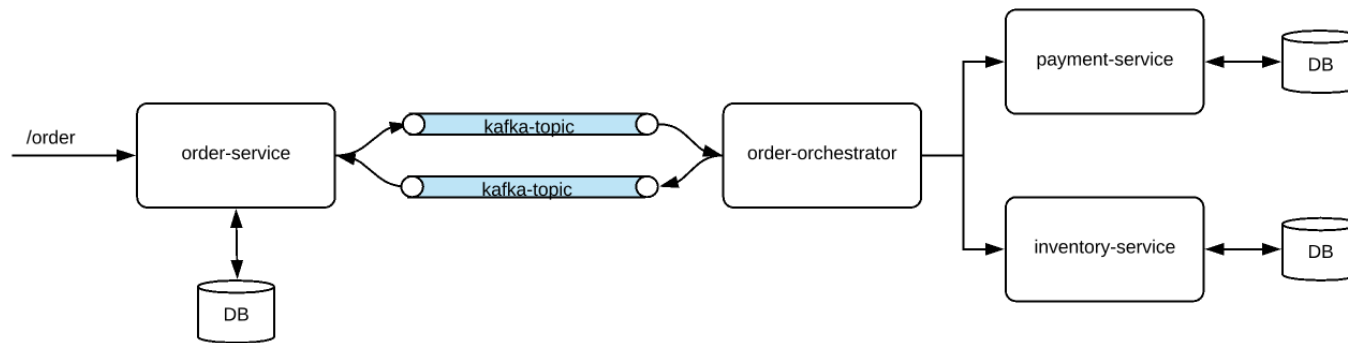
JMeter -

Distribut

ed Load

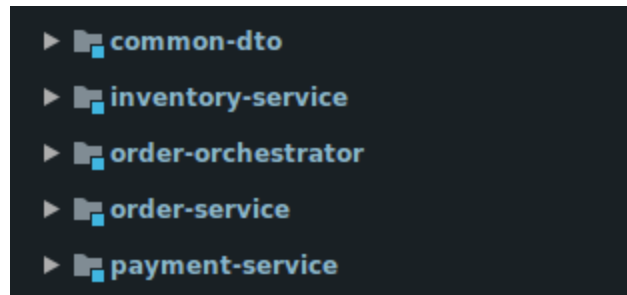
Testing





Common DTOs:

- First I create a Spring boot multi module maven project as shown below.



- I create common DTOs/models which will be used across all the microservices. (I would suggest you to follow [this approach](#) for DTOs)

using

Docker

JMeter -

How To

Test

REST API

/

MicroSer

vices

JMeter -

Property

File

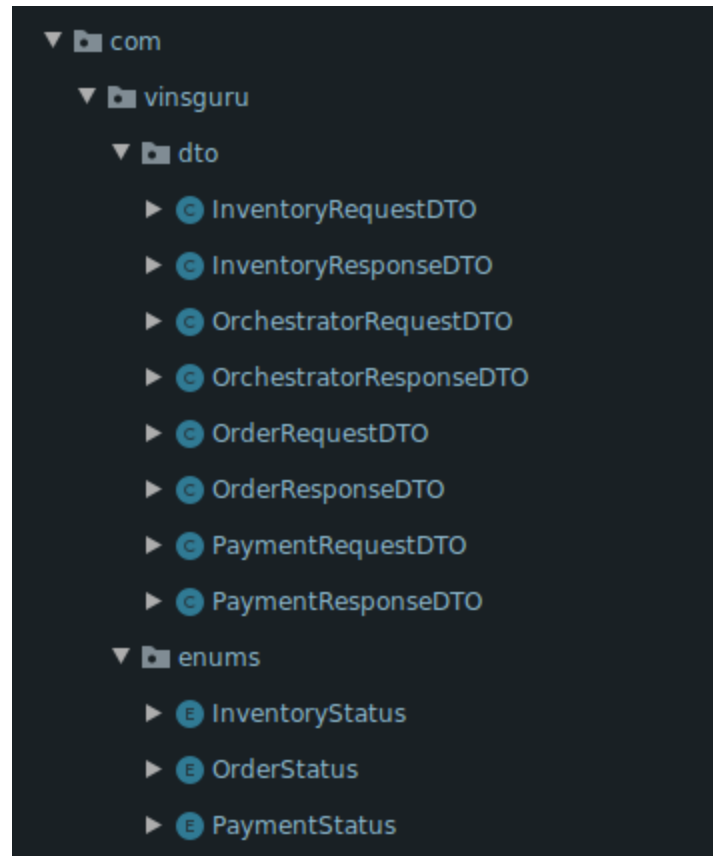
Reader -

A custom

config

element





Selenium
WebDriv
er - How
To Run
Automat
ed Tests
Inside A
Docker
Containe
r - Part 1

Inventory Service:

Each microservice which will be coordinated by orchestrator is expected to have at least 2 endpoints for each entity! One is deducting and other one is for resetting the transaction. For example. if we deduct inventory first and then later when we come to know that insufficient balance from payment system, we need to add the inventory back.

Note: I used a map as a DB to hold some inventory for few product IDs.

Categori
es

```
@Service
public class InventoryService {

    private Map<Integer, Integer> productInventoryMap;

    @PostConstruct
    private void init(){
        this.productInventoryMap = new HashMap<>();
        this.productInventoryMap.put(1, 5);
        this.productInventoryMap.put(2, 5);
        this.productInventoryMap.put(3, 5);
    }

    public InventoryResponseDTO deductInventory(final Integer productId,
        int quantity = this.productInventoryMap.getOrDefault(productId, 0);
        InventoryResponseDTO responseDTO = new InventoryResponseDTO();
        responseDTO.setOrderId(requestDTO.getOrderId());
        responseDTO.setUserId(requestDTO.getUserId());
        responseDTO.setProductId(requestDTO.getProductId());
        responseDTO.setStatus(InventoryStatus.UNAVAILABLE);
        if(quantity > 0){
            responseDTO.setStatus(InventoryStatus.AVAILABLE);
            this.productInventoryMap.put(productId, productInventoryMap.get(productId) - quantity);
        }
    }
}
```

Architecture

(62)

Arquillian (9)

Articles

(204)

AWS / Cloud

(17)

AWS (4)

Best

Practices

(75)

CI / CD /

DevOps (51)

Data Stream

/ Event

Stream (27)

Database (9)

^


```

        return responseDTO;
    }

    public void addInventory(final InventoryRequestDTO request) {
        this.productInventoryMap
            .computeIfPresent(requestDTO.getProductId(), (key, value) -> {
                // ...
            });
    }
}

```

- controller

```

@RestController
@RequestMapping("inventory")
public class InventoryController {

    @Autowired
    private InventoryService service;

    @PostMapping("/deduct")
    public InventoryResponseDTO deduct(@RequestBody final InventoryRequestDTO request) {
        return this.service.deductInventory(requestDTO);
    }
}

```

Design

Pattern (41)

Architectural

Design

Pattern (26)

Factory

Pattern (1)

Kubernetes

Design

Pattern (18)

Strategy

Pattern (1)

Distributed

Load Test

(9)

Docker (24)

ElasticSearch

h (2)



```
@PostMapping("/add")
public void add(@RequestBody final InventoryRequestDTO req
    this.service.addInventory(requestDTO);
}

}
```

Payment Service:

It also exposes 2 endpoints like inventory-service. I am showing only the important classes. For more details please check the github link at the end of this article for the complete project source code.

```
@Service
public class PaymentService {

    private Map<Integer, Double> userBalanceMap;

    @PostConstruct
    private void init(){
        this.userBalanceMap = new HashMap<>();
        this.userBalanceMap.put(1, 1000d);
        this.userBalanceMap.put(2, 1000d);
    }
}
```

Email

Validation (1)

Framework

(104)

Functional

Test

Automation

(83)

Puppeteer

(1)

QTP (10)

Selenium

(76)

Extend

WebDriver

(11)

Ocular (2)



```

        this.userBalanceMap.put(3, 1000d);
    }

    public PaymentResponseDTO debit(final PaymentRequestDTO requestDTO) {
        double balance = this.userBalanceMap.getOrDefault(requestDTO.getUserId(), 0.0d);
        PaymentResponseDTO responseDTO = new PaymentResponseDTO();
        responseDTO.setAmount(requestDTO.getAmount());
        responseDTO.setUserId(requestDTO.getUserId());
        responseDTO.setOrderId(requestDTO.getOrderId());
        responseDTO.setStatus(PaymentStatus.PAYMENT_REJECTED);
        if(balance >= requestDTO.getAmount()){
            responseDTO.setStatus(PaymentStatus.PAYMENT_APPROVED);
            this.userBalanceMap.put(requestDTO.getUserId(), balance - requestDTO.getAmount());
        }
        return responseDTO;
    }

    public void credit(final PaymentRequestDTO requestDTO){
        this.userBalanceMap.computeIfPresent(requestDTO.getUserId(), (key, value) -> value + requestDTO.getAmount());
    }
}

```

Page Object

Design (17)

Report (8)

Selenium

Grid (10)

TestNG (7)

gRPC (15)

Java (81)

Guice (2)

Reactor (41)

Jenkins (17)

Kafka (9)

Kubernetes

(8)

Linkerd (2)

Maven (7)

messaging

(11)



- controller

```

@RestController
@RequestMapping("payment")
public class PaymentController {

    @Autowired
    private PaymentService service;

    @PostMapping("/debit")
    public PaymentResponseDTO debit(@RequestBody PaymentRequest request) {
        return this.service.debit(request);
    }

    @PostMapping("/credit")
    public void credit(@RequestBody PaymentRequestDTO request) {
        this.service.credit(request);
    }
}

```



Order Service:

MicroService

(76)

Mongo (4)

Monitoring

(13)

FileBeat (1)

Grafana (5)

InfluxDB (7)

Kibana (2)

Multi Factor

Authenticati

on (2)

nats (4)

Performance

Testing (44)

Extend

JMeter (5)

JMeter (43) ^

Our order service receives the create order command and raises an **order-created** event using spring boot kafka binder. It also listens to **order-updated** channel/kafka topic and updates order status.

- controller

```
@RestController
@RequestMapping("order")
public class OrderController {

    @Autowired
    private OrderService service;

    @PostMapping("/create")
    public PurchaseOrder createOrder(@RequestBody OrderRequest
        requestDTO.setOrderId(UUID.randomUUID());
        return this.service.createOrder(requestDTO);
    }

    @GetMapping("/all")
    public List<OrderResponseDTO> getOrders(){
        return this.service.getAll();
    }
}
```

Workload
Model (2)
Little's Law
(1)
Web
Scraping (1)
Protocol
Buffers (15)
r2dbc (4)
Reactive
Programmin
g (40)
Redis (8)
rsocket (7)
Slack (3)
SMS (1)
Spring (73)

^

```
}
```

- service

```
@Service
public class OrderService {

    // product price map
    private static final Map<Integer, Double> PRODUCT_PRICE =
        1, 100d,
        2, 200d,
        3, 300d
    );

    @Autowired
    private PurchaseOrderRepository purchaseOrderRepository;

    @Autowired
    private FluxSink<OrchestratorRequestDTO> sink;

    public PurchaseOrder createOrder(OrderRequestDTO orderReq
        PurchaseOrder purchaseOrder = this.purchaseOrderRepos:
```

Spring Boot

(62)

Spring Data

(11)

Spring

WebFlux (62)

Udemy

Courses (5)

Utility (20)

WebSocket

(2)

^

```
        this.sink.next(this.getOrchestratorRequestDTO(orderReq  
        return purchaseOrder;  
    }  
}
```

```
public List<OrderResponseDTO> getAll() {  
    return this.purchaseOrderRepository.findAll()  
        .stream()  
        .map(this::entityToDto)  
        .collect(Collectors.toList());  
}
```

```
private PurchaseOrder dtoToEntity(final OrderRequestDTO dt  
    PurchaseOrder purchaseOrder = new PurchaseOrder();  
    purchaseOrder.setId(dto.getOrderId());  
    purchaseOrder.setProductId(dto.getProductId());  
    purchaseOrder.setUserId(dto.getUserId());  
    purchaseOrder.setStatus(OrderStatus.ORDER_CREATED);  
    purchaseOrder.setPrice(PRODUCT_PRICE.get(purchaseOrder  
    return purchaseOrder;  
}
```

```
private OrderResponseDTO entityToDto(final PurchaseOrder p  
    OrderResponseDTO dto = new OrderResponseDTO();  
    dto.setOrderId(purchaseOrder.getId());
```

^

```
        dto.setProductId(purchaseOrder.getProductId());
        dto.setUserId(purchaseOrder.getUserId());
        dto.setStatus(purchaseOrder.getStatus());
        dto.setAmount(purchaseOrder.getPrice());
        return dto;
    }

    public OrchestratorRequestDTO getOrchestratorRequestDTO(Or
    OrchestratorRequestDTO requestDTO = new OrchestratorRe
    requestDTO.setUserId(orderRequestDTO.getUserId());
    requestDTO.setAmount(PRODUCT_PRICE.get(orderRequestDT
    requestDTO.setOrderId(orderRequestDTO.getOrderId());
    requestDTO.setProductId(orderRequestDTO.getProductId(
    return requestDTO;
}

}
```

Order Orchestrator:

This is a microservice which is responsible for coordinating all the transactions. It listens to order-created topic. As and when a new order is created, It immediately builds separate request to each service like payment-service/inventory-service etc

^

and validates the responses. If they are OK, fulfills the order. If one of them is not, cancels the order. It also tries to reset any of local transactions which happened in any of the microservices.

We consider all the local transactions as 1 single workflow. A workflow will contain multiple workflow steps.

- Workflow step

```
public interface WorkflowStep {  
  
    WorkflowStepStatus getStatus();  
    Mono<Boolean> process();  
    Mono<Boolean> revert();  
  
}
```

- Workflow

```
public interface Workflow {  
  
    List<WorkflowStep> getSteps();  
  
}
```

^

```
}
```

- In our case, for the Order workflow, we have 2 steps. Each implementation should know how to do local transaction and how to reset.
- Inventory step

```
public class InventoryStep implements WorkflowStep {  
  
    private final WebClient webClient;  
    private final InventoryRequestDTO requestDTO;  
    private WorkflowStepStatus stepStatus = WorkflowStepStatus:  
  
    public InventoryStep(WebClient webClient, InventoryRequest  
        this.webClient = webClient;  
        this.requestDTO = requestDTO;  
    }  
  
    @Override  
    public WorkflowStepStatus getStatus() {  
        return this.stepStatus;  
    }  
}
```

^

@Override

```
public Mono<Boolean> process() {  
    return this.webClient  
        .post()  
        .uri("/inventory/deduct")  
        .body(BodyInserters.fromValue(this.requestDTO))  
        .retrieve()  
        .bodyToMono(InventoryResponseDTO.class)  
        .map(r -> r.getStatus().equals(InventoryStatus.  
        .doOnNext(b -> this.stepStatus = b ? Workflow.  
    }  
}
```

@Override

```
public Mono<Boolean> revert() {  
    return this.webClient  
        .post()  
        .uri("/inventory/add")  
        .body(BodyInserters.fromValue(this.request  
        .retrieve()  
        .bodyToMono(Void.class)  
        .map(r -> true)  
        .onErrorReturn(false);  
}
```

^

```
}  
}
```

- Payment step

```
public class PaymentStep implements WorkflowStep {  
  
    private final WebClient webClient;  
    private final PaymentRequestDTO requestDTO;  
    private WorkflowStepStatus stepStatus = WorkflowStepStatus:  
  
    public PaymentStep(WebClient webClient, PaymentRequestDTO  
        this.webClient = webClient;  
        this.requestDTO = requestDTO;  
    }  
  
    @Override  
    public WorkflowStepStatus getStatus() {  
        return this.stepStatus;  
    }  
  
    @Override  
    public Mono<Boolean> process() {
```

^

```
        return this.webClient
            .post()
            .uri("/payment/debit")
            .body(BodyInserters.fromValue(this.requestDTO))
            .retrieve()
            .bodyToMono(PaymentResponseDTO.class)
            .map(r -> r.getStatus().equals(PaymentStatus.COMPLETED))
            .doOnNext(b -> this.stepStatus = b ? WorkStatus.COMPLETED : WorkStatus.FAILED)
    }
}
```

```
@Override
public Mono<Boolean> revert() {
    return this.webClient
        .post()
        .uri("/payment/credit")
        .body(BodyInserters.fromValue(this.requestDTO))
        .retrieve()
        .bodyToMono(Void.class)
        .map(r -> true)
        .onErrorReturn(false);
}
}
```

- service / coordinator

```
@Service
public class OrchestratorService {

    @Autowired
    @Qualifier("payment")
    private WebClient paymentClient;

    @Autowired
    @Qualifier("inventory")
    private WebClient inventoryClient;

    public Mono<OrchestratorResponseDTO> orderProduct(final Order
        Workflow orderWorkflow = this.getOrderWorkflow(request);
        return Flux.fromStream(() -> orderWorkflow.getSteps())
            .flatMap(WorkflowStep::process)
            .handle(((aBoolean, synchronousSink) -> {
                if(aBoolean)
                    synchronousSink.next(true);
                else
                    synchronousSink.error(new WorkflowException());
            })))
}
```

```
.then(Mono.fromCallable(() -> getResponseDTO(i
.onErrorResume(ex -> this.revertOrder(orderWo

}

private Mono<OrchestratorResponseDTO> revertOrder(final Wo
return Flux.fromStream(() -> workflow.getSteps().stre
    .filter(wf -> wf.getStatus().equals(WorkflowS
    .flatMap(WorkflowStep::revert)
    .retry(3)
    .then(Mono.just(this.getResponseDTO(requestDT

}

private Workflow getOrderWorkflow(OrchestratorRequestDTO i
    WorkflowStep paymentStep = new PaymentStep(this.payme
    WorkflowStep inventoryStep = new InventoryStep(this.i
    return new OrderWorkflow(List.of(paymentStep, invento

}

private OrchestratorResponseDTO getResponseDTO(Orchestrato
    OrchestratorResponseDTO responseDTO = new Orchestrator
    responseDTO.setOrderId(requestDTO.getOrderId());
    responseDTO.setAmount(requestDTO.getAmount());
    responseDTO.setProductId(requestDTO.getProductId());
```

^

```
responseDTO.setUserId(requestDTO.getUserId());  
responseDTO.setStatus(status);  
return responseDTO;  
}
```

```
private PaymentRequestDTO getPaymentRequestDTO(Orchestrator  
    PaymentRequestDTO paymentRequestDTO = new PaymentRequestDTO();  
    paymentRequestDTO.setUserId(requestDTO.getUserId());  
    paymentRequestDTO.setAmount(requestDTO.getAmount());  
    paymentRequestDTO.setOrderId(requestDTO.getOrderId());  
    return paymentRequestDTO;  
}
```

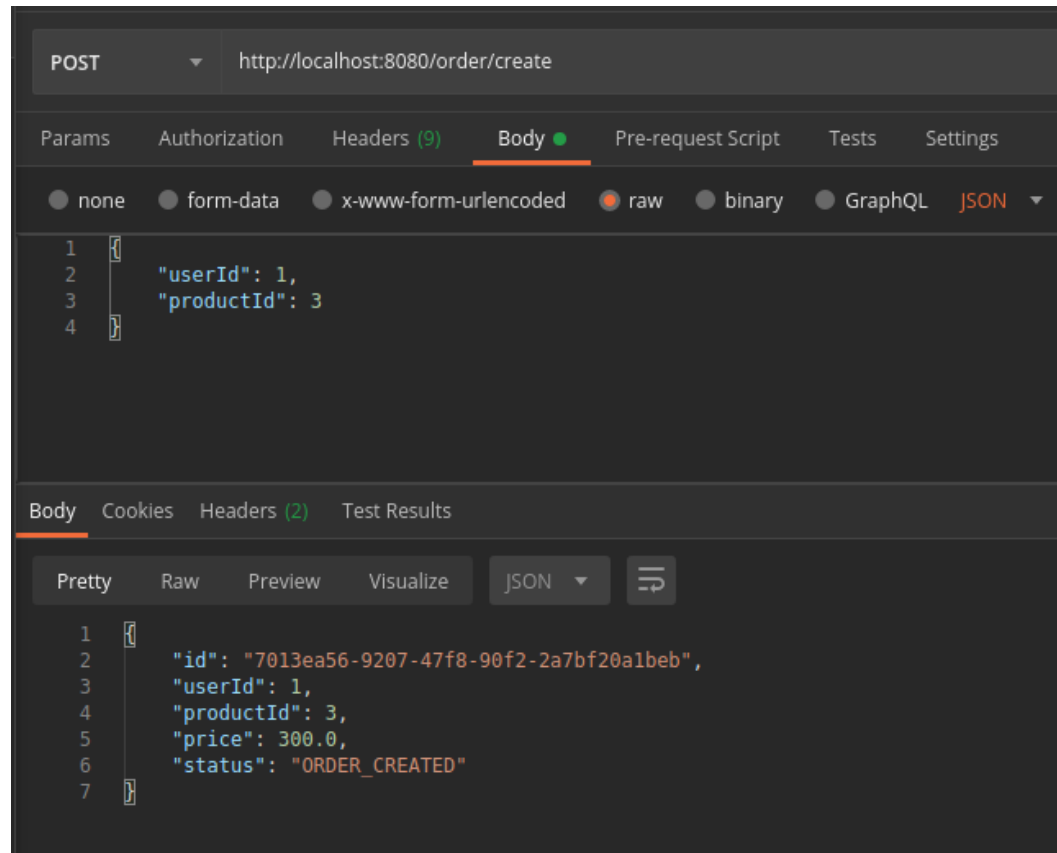
```
private InventoryRequestDTO getInventoryRequestDTO(Orchestrator  
    InventoryRequestDTO inventoryRequestDTO = new InventoryRequestDTO();  
    inventoryRequestDTO.setUserId(requestDTO.getUserId());  
    inventoryRequestDTO.setProductId(requestDTO.getProductId());  
    inventoryRequestDTO.setOrderId(requestDTO.getOrderId());  
    return inventoryRequestDTO;  
}
```

```
}
```

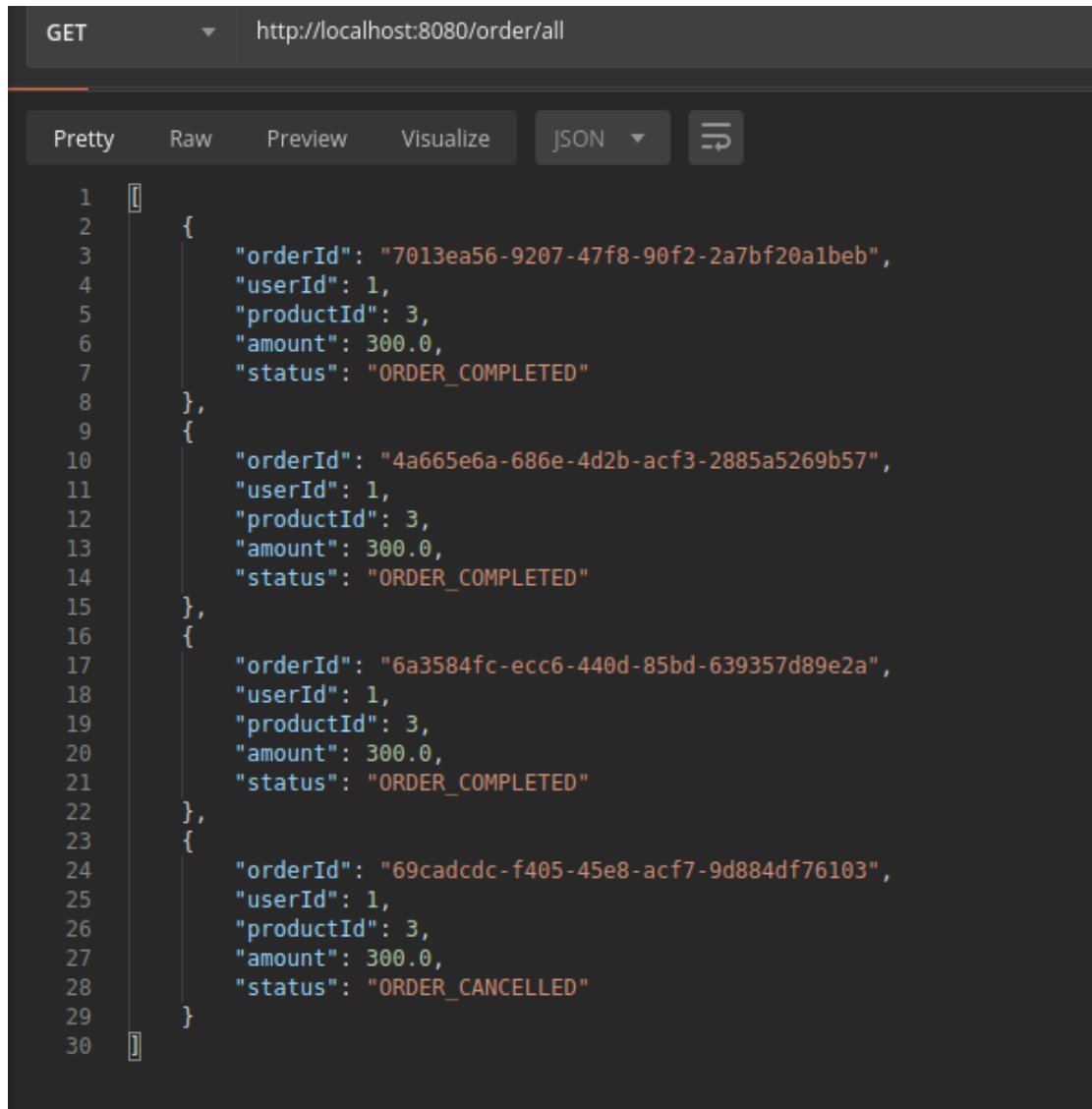

I have provided only high level details here. For the complete source, check [here](#).

Orchestration Saga Pattern – Demo:

- Once all the services are up and running, I send a POST request to create order. I get the order created status.
 - Do note that user 1 tries to order product id 3 which costs \$300
 - The user's credit limit is \$1000



- I sent 4 requests. So 3 requests were fulfilled. Not the 4th one as the user would have only \$100 left and we can not fulfill the 4th order. So the payment service would have declined.

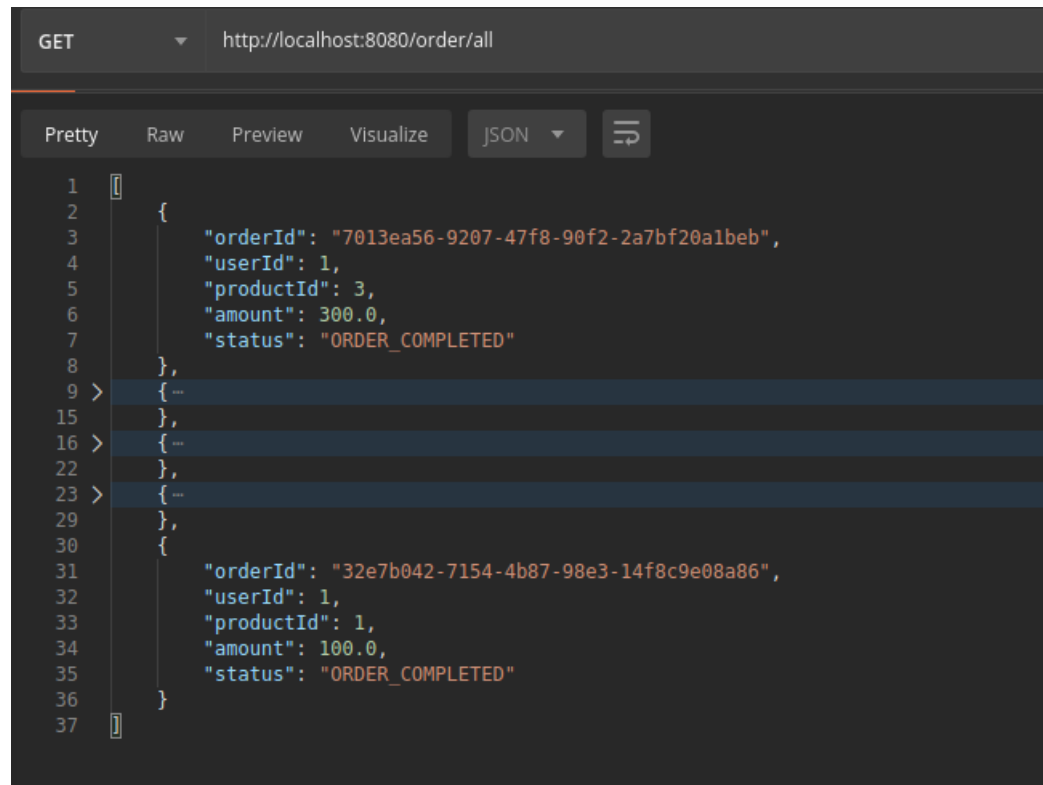


```
GET http://localhost:8080/order/all

Pretty Raw Preview Visualize JSON

1 [
2   {
3     "orderId": "7013ea56-9207-47f8-90f2-2a7bf20a1beb",
4     "userId": 1,
5     "productId": 3,
6     "amount": 300.0,
7     "status": "ORDER_COMPLETED"
8   },
9   {
10    "orderId": "4a665e6a-686e-4d2b-acf3-2885a5269b57",
11    "userId": 1,
12    "productId": 3,
13    "amount": 300.0,
14    "status": "ORDER_COMPLETED"
15  },
16  {
17    "orderId": "6a3584fc-ecc6-440d-85bd-639357d89e2a",
18    "userId": 1,
19    "productId": 3,
20    "amount": 300.0,
21    "status": "ORDER_COMPLETED"
22  },
23  {
24    "orderId": "69cadcdc-f405-45e8-acf7-9d884df76103",
25    "userId": 1,
26    "productId": 3,
27    "amount": 300.0,
28    "status": "ORDER_CANCELLED"
29  }
30 ]
```

- The user 1 with this available balance \$100, he can buy product id 1 as it costs only \$100.



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

Pretty Raw Preview Visualize JSON ↺

1  [
2    {
3      "orderId": "7013ea56-9207-47f8-90f2-2a7bf20a1beb",
4      "userId": 1,
5      "productId": 3,
6      "amount": 300.0,
7      "status": "ORDER_COMPLETED"
8    },
9    { ...
15  },
16  { ...
22  },
23  { ...
29  },
30  {
31    "orderId": "32e7b042-7154-4b87-98e3-14f8c9e08a86",
32    "userId": 1,
33    "productId": 1,
34    "amount": 100.0,
35    "status": "ORDER_COMPLETED"
36  }
37 ]
```

Summary:

We were able to successfully demonstrate the **Orchestration Saga Pattern with Spring Boot**. Handling transactions and maintaining data consistency among all the microservices are difficult in general. When multiple services are involved like payment, inventory, fraud check, shipping check...etc it would be very difficult to manage such a complex workflow with multiple steps without a coordinator. By

introducing a separate service for orchestration, order-service is freed up from these responsibilities.

Check the project source code [here](#).

Learn more about Microservices Patterns.

- [Bulkhead Pattern – Microservice Design Patterns](#)
- [Circuit Breaker Pattern – Microservice Design Patterns](#)
- [CQRS Pattern – Microservice Design Patterns](#)

Happy coding 😊

Share This:

« [Flux Expand vs ExpandDeep with Examples](#)

[gRPC Web Example](#) »

27 thoughts on “Orchestration Saga Pattern With Spring Boot”



**Sambhav Dave**

August 12, 2020 at 4:38 PM

Wonderful sir. I am your youtube and udemy subscriber. Please keep on teaching students like me. I am working as test automation engineer at Tcs Nagpur. I love your way of teaching, and especially bringing new topics out of the traditional scenarios that we face. #BigFan

Reply

**vlns**

August 14, 2020 at 1:59 PM

Thanks Sambhav.

Reply

**Sankar**

August 26, 2020 at 8:53 PM

great article on spring boot and kafka. Thanks Vinoth.

great article on spring boot and kafka. Thanks Vinod

Reply



Milton Chapilliquen

September 1, 2020 at 10:26 AM

Hello dear,

I have a couple of questions about your excellent post:

I have my client API, which I have created 3 spring boot projects (independent): customer (GET /customers/{id-customer}, POST, PUT, etc), addresses (GET /{id-customer}/addresses, POST, PUT , etc) and contacts (GET /{id-customer}/contacts, POST, PUT, etc). Could you use Stream, Flux and kafka so that when I request GET /customers/{id} it returns the complete 'customers' information, including the 'address' and 'contacts' service information in parallel? It means that the request must wait for the data of each service to be obtained and return all the information of a customer. Do you have any suggestions or examples?

About the display of orders, payments and products, is there a way that the response from the execution of the POST /order/create returns the ORDER_CANCELLED status? I mean does the POST request "wait" for the process to finish completely and the updated status to be sent when an error occurs?

Currently the POST /order/create returns only ORDER_CREATED, and the status

changes it in the background

changes it in the background.

Reply



vlns

September 1, 2020 at 2:07 PM

Hi,

You have multiple questions here.

Most of the applications are more read heavy than write. So when you need customer info, you could avoid calling multiple services internally to join all the info. Because when you need to query 3 services, when one service is down, the entire application becomes kind of down! not accessible. There is a separate pattern for this. Event carried state transfer – Check this **one**.

If you check the screenshots, order-service always keeps the order in pending status first as soon as it gets a request. The orchestraor service is the one which coordinates all the activities and decides it can be fulfilled or cancelled. So, order-service does not wait. It is asynchronous.

Thanks.



Reply



Hien HOANG

October 26, 2020 at 2:46 AM

Hi Vins,

I followed your blog for really long time, and I saw that you had several topic related to Kafka. I am just curious about how do you perform the testing for streaming application using Kafka. May you create a post about each testing level for Kafka?

Many thanks in advanced.

Reply



Leclerc HOUNMENOU

November 23, 2020 at 5:04 AM

Just say thanks you!!

Very satisfied!

You are a master!

Never give up!!

Don't



vlns

November 23, 2020 at 3:13 PM

spam?

Reply



jagadeesh

December 11, 2020 at 10:03 AM

Hi ..good one..plz provide ur udemy details and you tube..are u providing any tutorial for microservices?..

Reply



Rajugaru

January 2, 2021 at 4:02 AM

after reading a dozen stuff over the internet and tried and got tired. This is one of the most satisfying among all other. and thank you very much for all the



explanation. Loved it. Looking forward to read some more interesting stuff.

Reply



Akash

January 19, 2021 at 6:44 PM

Awesome tutorial. I struggled a lot to find distributed transaction examples and tutorials. This article is my destination!

Reply



Martin

January 23, 2021 at 10:28 AM

I wonder, what will happen when during the reverting process – let's say after revert payment but before revert inventory (or vice versa) – the coordinator failed and go down. With this implementation we will leave our system in inconsistent state.

Reply



vlns

January 24, 2021 at 4:40 PM

Yes, It could happen. For that we could use Kafka topics for the



communication between the orchestrator and other services.

Or this orchestrator will be maintaining its own DB for the state. That is – It has to get the confirmation for the reverting process. Otherwise it will retry.

Reply



K

March 5, 2021 at 2:33 AM

Is this orchestration approach recommended if we have one service calling 7-8 micro services?

Also, we could call the inventory and payment service using Test template. Is a message broker or Kafka mandatory?

Could you please let me know? Thanks

Reply



vlns

March 5, 2021 at 3:35 AM

Each approach has its own advantage!

RestTemplate is simple and easy. Using message brokers helps with async processing and calling service does not have to worry about service unavailability. I would not say it is recommended. It just a demo wherever this pattern will be useful.

Reply



Raj

February 21, 2022 at 8:17 AM

Hello vlns

Nice article.

I am just wondering where in the code you invoke the orchestration service. If you could direct me to that code snippet?

Thanks

Raj

Reply



vlns

February 26, 2022 at 3:24 PM



Check here for the complete source code.

Reply



Camilo Cortes

March 28, 2022 at 12:23 AM

Hi vlns

excellent article.

I have a question.

I Can implement the project dont using Kafka? only calling the order-orchestration service with a HTTP request?

Reply



vlns

June 26, 2022 at 8:22 PM

Yes. Please check this – <https://www.vinsguru.com/spring-webflux-aggregation/>

Reply





Neha

June 16, 2022 at 2:53 AM

Hi,

I see your current code don't return correct "status": "ORDER_CANCELLED" on 4th attempts. Could you please double check?

Reply



vlns

June 26, 2022 at 7:32 PM

Thanks a lot. Looks like it broke with recent version of Spring. I have updated the code as few things got deprecated as well. It seems to work now.

Reply



Neha

July 11, 2022 at 4:13 PM

Hey, I do see code still not working with the latest code, I double

check many times. something weird happening. would you want to double check again ?

Reply



Neha

July 11, 2022 at 5:53 PM

I think there is confusion with the Status – it always return ORDER_CREATED irrespective of success or failure. Its happening because you've hardcoded in OrderService.java dtoToEntity method need some correction. Please do the needful and inform back.

Reply



vlns

July 11, 2022 at 8:01 PM

Yes, it is ORDER CREATED. Not Completed or Failed.
The status will get updated eventually!!





sai

June 20, 2022 at 10:31 AM

same me too, Wondering why the flow is not going to order -orchestrator processor.

Reply



vlns

June 26, 2022 at 7:33 PM

It works fine now as I had to update few things for latest version of Spring.

Reply



sai

June 20, 2022 at 10:51 AM

I just Found, Using rest end point we can see cancelled order as shown in the below

images

images.

<http://localhost:8080/order/all>

Thanks For the article @Vinsguru

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

