Recent Posts

Spring

WebFlux

Aggregation

Event Carried State Transfer – Microservice Design Patterns

3 Comments / Architectural Design Pattern, Architecture, Articles, Best Practices, Data Stream / Event Stream, Design Pattern, Framework, Kafka, MicroService, Spring, Spring Boot, Spring WebFlux / By vIns / December 6, 2020

Overview:

In this tutorial, I would like to show you one of the Microservice Design Patterns – **Event Carried State Transfer** to achieve the data consistency among the Microservices.

Event Carried State Transfer:

Modern application technology has changed a lot. In the traditional monolithic architecture which consists of all the modules for an application, we have a database which contains all the tables for all the modules. When we move from the monolith application into Microservice Architecture, we also split our big fat DB into multiple data sources. Each and every service manages its own data.

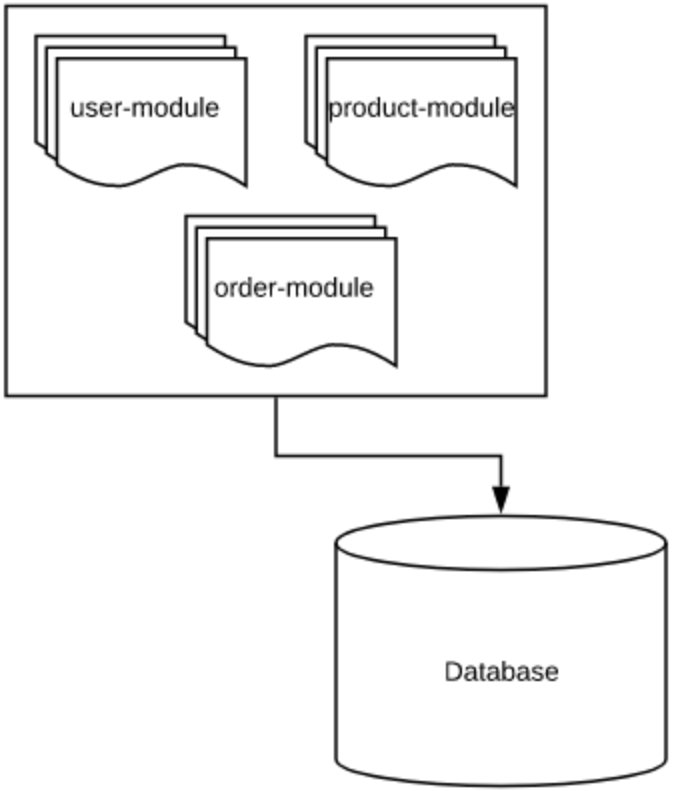
Having different databases and data models bring advantages into our distributed systems architecture. However when we have multiple data sources, obvious challenge would be how to maintain the data consistency among all the

Choreography Saga Pattern With Spring Boot
Spring WebFlux WebSocket gRPC Web Example
Orchestration Saga Pattern With Spring Boot
Selenium WebDriver - How

Microservices when one of the modifies the data. The idea behind **Event Carried State Transfer** pattern is – when a Microservice inserts/modifies/deletes data, it raises an event along with data. So the interested Microservices should consume the event and update their own copy of data accordingly.

Sample Application:

In this example, let's consider a simple application as shown here. A monolith application has modules like user-module, product-module and order-module.



Our DB for the above application has below tables.

```
CREATE TABLE users(
   id serial PRIMARY KEY,
   firstname VARCHAR (50),
   lastname VARCHAR (50),
   email varchar(50)
);

CREATE TABLE product(
   id serial PRIMARY KEY,
   description VARCHAR (500),
   price numeric (10,2) NOT NULL,
   qty_available integer NOT NULL
);

CREATE TABLE purchase_order(
    id serial PRIMARY KEY,
    user_id integer references users (id),
    product_id integer references product (id),
    price numeric (10,2) NOT NULL
);
```

Multiple Test Environments

Selenium WebDriver - Design Patterns in Test Automation - Factory Pattern

Kafka Stream With

When I need to find all the user's orders, I can write a simple join query like this, fetch the details and show it on the UI.

```
select
    u.firstname,
    u.lastname,
    p.description,
    po.price
from
    users u,
    product p,
    purchase_order po
where
    u.id = po.user_id
and p.id = po.product_id
order by u.id;
```

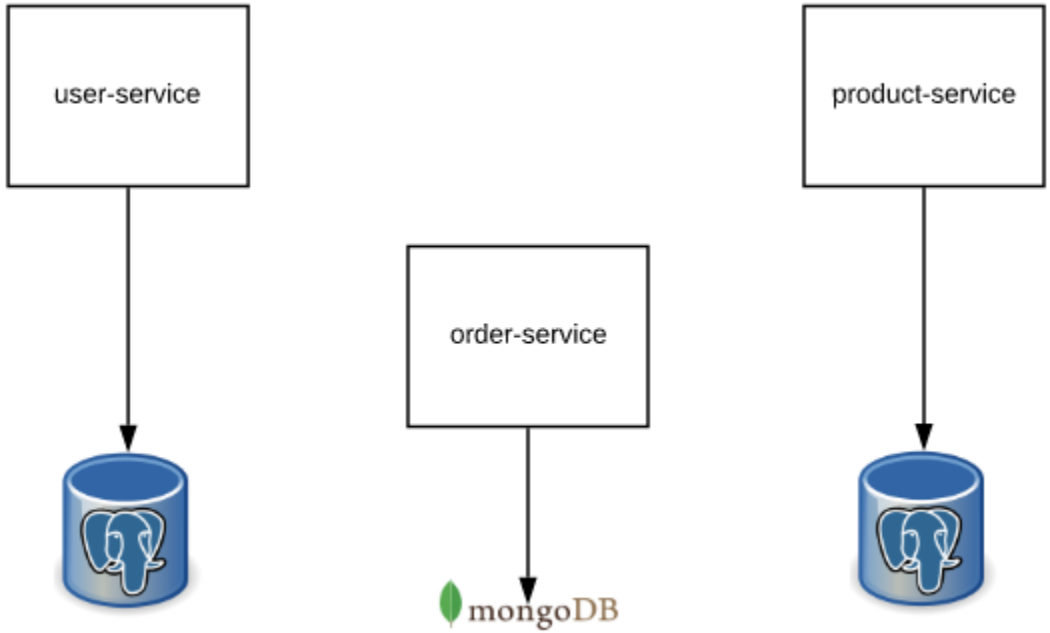
	firstname character varying (50)	lastname character varying (50)	description character varying (500)	price numeric (10,2)
1	vins	guru	ipad	300.00
2	michael	jackson	iphone	650.00
3	michael	jackson	ipad	250.00
4	slim	shady	tv	320.00

Spring Boot


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


JMeter - Distributed Load Testing

That was easy! Now let's assume that we move into Microservice architecture. We have a user-service, product-service and order-service. Each service has it own database.



- user-service
  - Microservice responsible for managing user related application functionalities
  - user-service connects to a PostgreSQL DB which contains users table

using Docker

JMeter - How To Test REST API / MicroServices

JMeter - Property File Reader - A custom config element

Data Output Explain Messages

	id [PK] integer	firstname character varying (50)	lastname character varying (50)	email character varying (50)
1	1	vins	guru	admin@vinsguru.com
2	2	michael	jackson	mj@vinsguru.com
3	3	slim	shady	shady@vinsguru.com

- product-service
  - Microservice responsible for managing product related application functionalities
  - product-service connects to a PostgreSQL DB which contains product table

	id [PK] integer	description character varying (500)	price numeric (10,2)	qty_available integer
1	1	ipad	300.00	10
2	2	iphone	650.00	98
3	3	tv	320.00	560

- order-service
  - Connects to a MongoDB and contains all the user orders along with the product information, price etc.
  - MongoDB contains a collection called purchase_order which has information like this.

Selenium WebDriver - How To Run Automated Tests Inside A Docker Container - Part 1

Categories

```
[
   {
      "userId":1,
      "productId":1,
      "price":300.00
   },
   {
      "userId":2,
      "productId":1,
      "price":250.00
   },
   {
      "userId":2,
      "productId":2,
      "price":650.00
   },
   {
      "userId":3,
      "productId":3,
      "price":320.00
   }
]
```

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)

Now in the above case, when we look for all the user's order, we can not simply write a join query across all the different data sources as we did earlier. We need to first send a request to order-service. Once we get the response from the order-service, based on the **userId** and **productId** it has, We also need to send a request to user-service and product-service to get the user and product details, process the data and show it on the UI. It looks like a lot of work, HTTP calls, network latency to deal with and they are all going to affect performance of the application very badly.

It also creates tight coupling among microservices which is bad! What will happen when the user-service is not available? It will also make the order-service FAIL which we do not want!!!

One possible solution which might sound very bad advice to you is having the user and product information in the **purchase_order** collection itself in the MongoDB as shown here.

```
{
   "user":{
      "id":1,
      "firstname":"vins",
      "lastname":"guru",
      "email: "admin@vinsguru.com"
   },
```

Design Pattern (41)
Architectural Design Pattern (26)
Factory Pattern (1)
Kubernetes Design Pattern (18)
Strategy Pattern (1)
Distributed Load Test (9)
Docker (24)
ElasticSearch (2)

```
    "product":{
       "id":1,
       "description":"ipad"
    },
    "price":300.00
}
```

In this approach, order-service itself has all the information for us to show the data on the UI. It does not depend on other services like user-service, product-service to provide the information we need. It is loosely coupled.

Advantages:

- No more table join.
- Less network calls
- Improved performance
- Loose coupling

Why it might sound very bad is because, data is redundantly stored and what if user changes his name / email? or what if the product description is updated? In the traditional approach, It was not a problem. Now order-service would not have the updated information. It would have stale data if user or product info is updated.

EMail Validation (1)
Framework (104)
Functional Test Automation (83)
Puppeteer (1)
QTP (10)
Selenium (76)
Extend WebDriver (11)
Ocular (2)
Page Object Design (17)

Disadvantages:

- Stale data (user-service updates an user info, order-service will have stale data)
- Redundant data (means additional disk space)

Redundant data/Additional disk space is really not a problem nowadays as data storage is very very cheap! But We can update the user details in the order-service whenever user-details are updated in the user-service. It would be happening asynchronously. Eventual consistency is the trade off for the performance / resilient design we get!

Lets see how we can maintain updated data across all the microservices using Kafka to avoid the above mentioned problem!

Kafka Infrastructure Setup:

We need to have Kafka cluster up and running along with ZooKeeper. Take a look at these articles first If you have not already!

- Kafka – Local Infrastructure Setup Using Docker Compose
- Kafka – Creating Simple Producer & Consumer Applications Using Spring Boot

As part of this article, We are going to update order-service's user details whenever there is an update on user details in the user-service asynchronously. For that we are going to create a topic called **user-service-event** in our Kafka cluster.

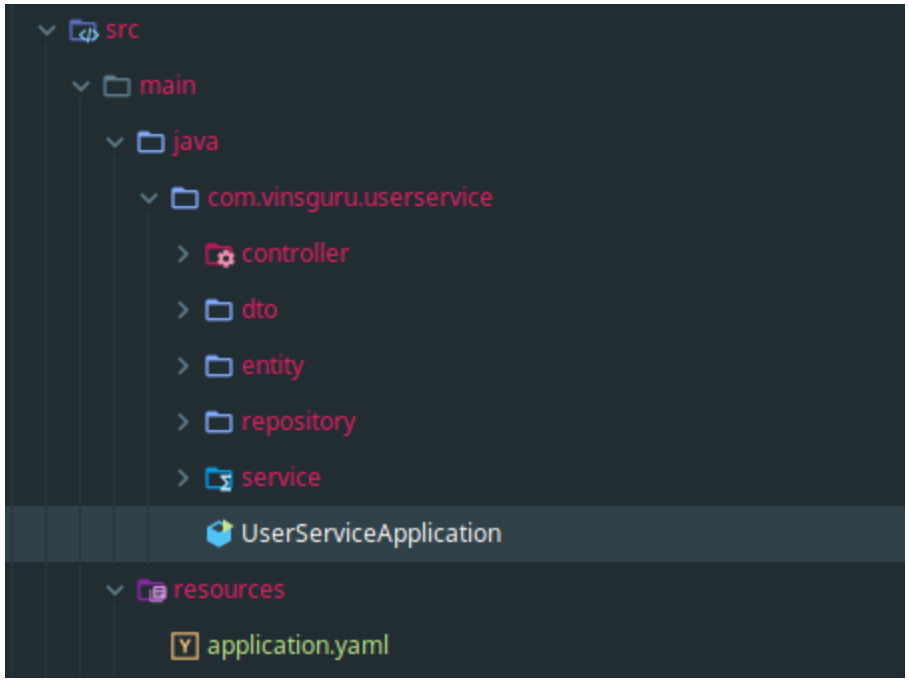
User Service:

- I am creating a simple spring boot application for user-service with below dependencies.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
<dependency>
```

```
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

- This user-service would be the Kafka-producer. It will add the events to **user-service-event** topic whenever user details are updated.
- User-service project structure is as shown below.



- User Entity

```
@Entity
public class Users {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstname;
    private String lastname;
    private String email;

   // getters and setters

}
```

- User Repository

```
@Repository
public interface UsersRepository extends JpaRepository<Users,
}
```

- User DTO

Spring Data (11)
Spring WebFlux (62)
Udemy Courses (5)
Utility (20)
WebSocket (2)

```
public class UserDto {
    private Long id;
    private String firstname;
    private String lastname;
    private String email;

    // getters & setters

}
```

- Service
    - UserServiceImpl is responsible for updating the user details and adding the details into the Kafka topic
    - **Updating the database and publishing the event must occur in a single transaction.**

```
public interface UserService {
    Long createUser(UserDto userDto);
    void updateUser(UserDto userDto);
}

@Service
```

```
public class UserServiceImpl implements UserService {

    private static final ObjectMapper OBJECT_MAPPER = new Obje

    @Autowired
    private UsersRepository usersRepository;

    @Autowired
    private KafkaTemplate<Long, String> kafkaTemplate;

    @Override
    public Long createUser(UserDto userDto) {
        Users user = new Users();
        user.setFirstname(userDto.getFirstname());
        user.setLastname(userDto.getLastname());
        user.setEmail(userDto.getEmail());
        return this.usersRepository.save(user).getId();
    }

    @Override
    @Transactional
    public void updateUser(UserDto userDto) {
        this.usersRepository.findById(userDto.getId())
                .ifPresent(user -> {
```

```
                    user.setFirstname(userDto.getFirstname());
                    user.setLastname(userDto.getLastname());
                    user.setEmail(userDto.getEmail());
                    this.raiseEvent(userDto);
                });
    }

    private void raiseEvent(UserDto dto){
        try{
            String value = OBJECT_MAPPER.writeValueAsString(dt
            this.kafkaTemplate.sendDefault(dto.getId(), value)
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

- User Controller

```
@RestController
@RequestMapping("/user-service")
public class UserController {
```

```
    @Autowired
    private UserService userService;

    @PostMapping("/create")
    public Long createUser(@RequestBody UserDto userDto){
        return this.userService.createUser(userDto);
    }

    @PutMapping("/update")
    public void updateUser(@RequestBody UserDto userDto){
        this.userService.updateUser(userDto);
    }

}
```

- Application.yaml and Kafka configuration

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/userdb
    username: vinsguru
    password: admin
  kafka:
```

```
    bootstrap-servers:
      - localhost:9091
      - localhost:9092
      - localhost:9093
    template:
      default-topic: user-service-event
    producer:
      key-serializer: org.apache.kafka.common.serialization.Lo
      value-serializer: org.apache.kafka.common.serialization.
```

- At this point, we should be able to successfully run the user-service. We should be able to create users / update users. Whenever user info is updated, we raise an event to the Kafka topic. So that interested microservices can subscribe to that.

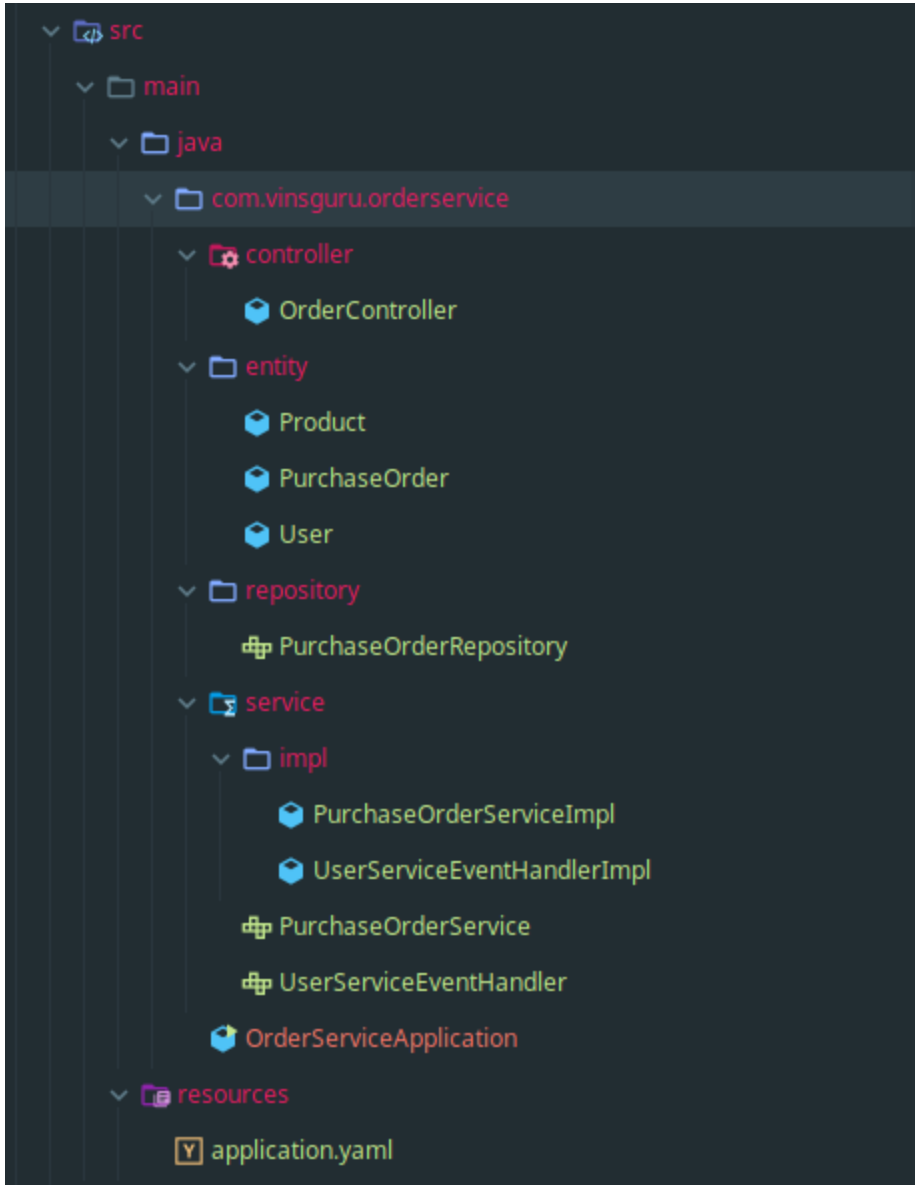
Order-Service

- I am creating another microservice for order-service with below dependencies. MongoDB would be the backend for this service.

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

```
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

- This service also needs Kafka dependency as it would be subscribing to the **user-service-event** topic. This service would be a Kafka consumer.
- Order-service's project structure would be as shown here.



- Purchase Order Entity

```
@Document
public class PurchaseOrder {

    @Id
    private String id;
    private User user;
    private Product product;
    private double price;

    // Getters & Setters

}

public class Product {

    private long id;
    private String description;

    // Getters & Setters

}

public class User {
```

```
        private Long id;
        private String firstname;
        private String lastname;
        private String email;

        // Getters & Setters

}
```

- Purchase Order data access layer

```
@Repository
public interface PurchaseOrderRepository extends MongoReposit

    @Query("{ 'user.id': ?0 }")
    List<PurchaseOrder> findByUserId(long userId);

}
```

- Service
    - This service class simply retrieves all the data from the DB
    - It also make entries when there is a new order

- This service class is not responsible for updating user information

```
public interface PurchaseOrderService {
    List<PurchaseOrder> getPurchaseOrders();
    void createPurchaseOrder(PurchaseOrder purchaseOrder);
}

@Service
public class PurchaseOrderServiceImpl implements PurchaseOrder

    @Autowired
    private PurchaseOrderRepository purchaseOrderRepository;

    @Override
    public List<PurchaseOrder> getPurchaseOrders() {
        return this.purchaseOrderRepository.findAll();
    }

    @Override
    public void createPurchaseOrder(PurchaseOrder purchaseOrde
        this.purchaseOrderRepository.save(purchaseOrder);
    }
```

```
}
```

- User Service Event Handler
    - This service class is responsible for subscribing to a Kafka topic.
    - Whenever user-service raises an event, the message would be consumed here immediately and user details would be updated.

```
public interface UserServiceEventHandler {
    void updateUser(User user);
}

@Service
public class UserServiceEventHandlerImpl implements UserServic

    private static final ObjectMapper OBJECT_MAPPER = new Obje

    @Autowired
    private PurchaseOrderRepository purchaseOrderRepository;

    @KafkaListener(topics = "user-service-event")
    public void consume(String userStr) {
```

```
            try{
                User user = OBJECT_MAPPER.readValue(userStr, User
                this.updateUser(user);
            }catch(Exception e){
                e.printStackTrace();
            }
        }

        @Override
        @Transactional
        public void updateUser(User user) {
            List<PurchaseOrder> userOrders = this.purchaseOrderRep
            userOrders.forEach(p -> p.setUser(user));
            this.purchaseOrderRepository.saveAll(userOrders);
        }
    }
```

- Order Controller

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

```
    @Autowired
    private PurchaseOrderService purchaseOrderService;

    @GetMapping("/all")
    public List<PurchaseOrder> getAllOrders(){
        return this.purchaseOrderService.getPurchaseOrders();
    }

    @PostMapping("/create")
    public void createOrder(@RequestBody PurchaseOrder purchas
        this.purchaseOrderService.createPurchaseOrder(purchase
    }

}
```

- Application.yaml and Kafka consumer configuration

```
spring:
  data:
    mongodb:
      host: localhost
      port: 27017
      database: order-service
```

```
  kafka:
    bootstrap-servers:
      - localhost:9091
      - localhost:9092
      - localhost:9093
    consumer:
      group-id: user-service-group
      auto-offset-reset: earliest
      key-serializer: org.apache.kafka.common.serialization.Lo
      value-serializer: org.apache.kafka.common.serialization.
```

- At this point, Order-service is also up and running fine. It listens to Kafka topic.
- I created a purchase order and called the GET request to get the below response.

```
[
    {
        "id": "5dcfb1056637311008e17f80",
        "user": {
            "id": 1,
            "firstname": "vins",
```

```
            "lastname": "guru",
            "email": "admin@vinsguru.com"
        },
        "product": {
            "id": 1,
            "description": "ipad"
        },
        "price": 300
    }
]
```

- Then I send the below PUT request to my user-service.

```
{
    "id": 1,
    "firstname":"vins",
    "lastname": "gur",
    "email": "admin-updated@vinsguru.com"
}
```

- Now I call the purchase order GET request once again. User detail updates are getting reflected immediately in the order-service.

```
[
    {
        "id": "5dcfb10566373110008e17f80",
        "user": {
            "id": 1,
            "firstname": "vins",
            "lastname": "guru",
            "email": "admin-updated@vinsguru.com"
        },
        "product": {
            "id": 1,
            "description": "ipad"
        },
        "price": 300
    }
]
```


Source Code:

The source is available here.

Summary:

We were able to maintain data consistency across all the microservices using Kafka. This approach avoids many unnecessary network calls among microservices, improves the performance of microservices and make the microservices loosely coupled. For ex: Order-service does not have to be up and running when user details are updated via user-service. User-service would be raising an event. Order-service can subscribe to that whenever it is up and running. So that information is not going to be lost! In the old approach, it makes microservices tightly coupled in such a way that all the dependent microservices have to be up and running together. Otherwise it would make the system unavailable.

« Redis Lua Script With Spring Boot

Scatter Gather Pattern – Microservice Design Patterns »

3 thoughts on “Event Carried State Transfer – Microservice Design Patterns”

alisson pedrina

March 3, 2021 at 8:10 AM

Cool, great post, I hope you keep doing this kind of content, that is very useful.

Reply

Neha

March 15, 2021 at 5:13 PM

Dear Vinoth,

Thanks for such a great tutorials. it looks like this particular code example is not complete. The product-service is completely missing and things are not getting clear to me . Could you please update the required stuff?

Reply

vIns

March 17, 2021 at 4:58 PM

First I explain the high level concept – just theory. For the demo, I had mentioned that – As part of this article, We are going to update order-

service's user details whenever there is an update on user details in the user-service asynchronously. – So it was intentional.

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