SAGA Design Pattern:

- The SAGA pattern is a design pattern used in microservices architecture to manage complex transactions that involve multiple services.
- In a microservices architecture, each service is responsible for its own data and performs a specific business function.
- However, there are some transactions that require coordination between multiple services, which can be challenging to manage.

  Ex:

  order_service : will place order

  payment_service : will process payment

  order_fullfill_service : will confirm/fail order

  - First user will place order and will proceed for payment.
  - If payment is successful, then order should be confirmed else order should be failed.
  - Based on payment event order_fullfill_service will take action.

Note: To complete user request, co-ordination is required between above services

- As these 2 services using 2 diff databases, managing transaction commit and rollback is very challenging.

Note: We need to implement Distributed Transaction to handle this scenario.

- **SAGA** design pattern is used to manage distributed transactions in the application.
- SAGA Transactions means sequence of transactions.
- A saga is a sequence of local transactions, where each local transaction updates the data within a single service, and a coordinator manages the overall flow of the saga.
- The coordinator is responsible for making sure that each step of the saga is executed in the correct order and handling any errors that may occur.
- SAGA pattern is used to simplify distributed transaction management in microservices architecture.
- It allows for greater flexibility and scalability by breaking down the transaction into smaller, manageable steps, and provides a mechanism for handling errors and maintaining data consistency.
- However, it can also add complexity to the system and requires careful coordination between services. Therefore, it should be used judiciously and only for transactions that truly require it.

Types of SAGA Patterns:

1. Choreography-based SAGA pattern (event based)
2. Orchestration-based SAGA pattern (command based / controller)

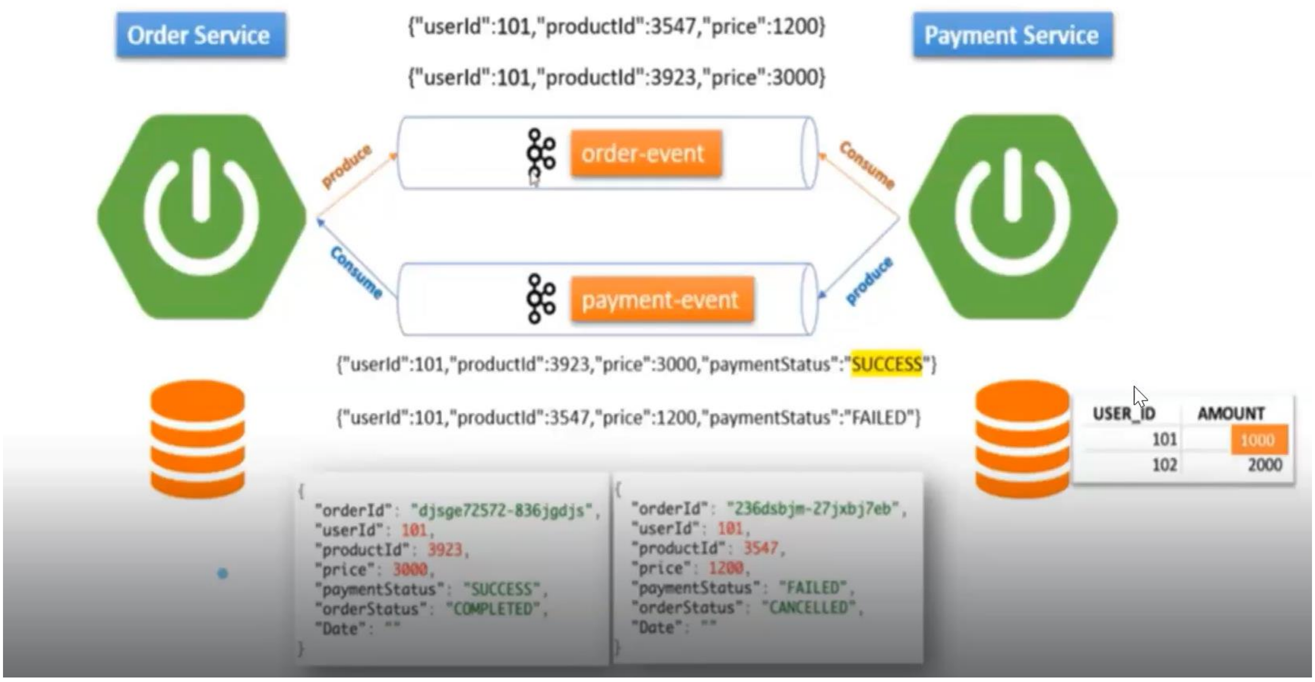
Choreography-based SAGA pattern:

- In a choreography-based SAGA pattern, each service in the transaction communicates with other services directly to coordinate their actions.
- There is no central coordinator or controller to manage the transaction.
- Instead, each service publishes events to notify other services of its actions and subscribes to events to receive notifications from other services.
- This way, each service knows what actions to take based on the events it receives from other services.
- Choreography patterns work based on events (We will use message broker to publish & subscribe to events).
- Choreography-based SAGA patterns are more decentralized, as there is no central coordinator, and services can act independently, making the system more scalable.
- However, it can be more complex to manage and debug, as it can be difficult to determine the order of events and which service is responsible for which action.

Orchestration-based SAGA pattern:

- In an orchestration-based SAGA pattern, a central coordinator or controller service manages the transaction.
- The coordinator sends messages to each service to instruct it on what actions to take and in what order.
- The coordinator is responsible for managing the flow of the transaction and coordinating the execution of each step in the saga.
- Orchestration-based SAGA patterns are more centralized, which makes it easier to manage and coordinate the transaction.
- However, it can be less flexible, as any change in the transaction requires changes to the coordinator, and it can become a bottleneck as the system scales.

Project Setup: Choreography-based SAGA Project



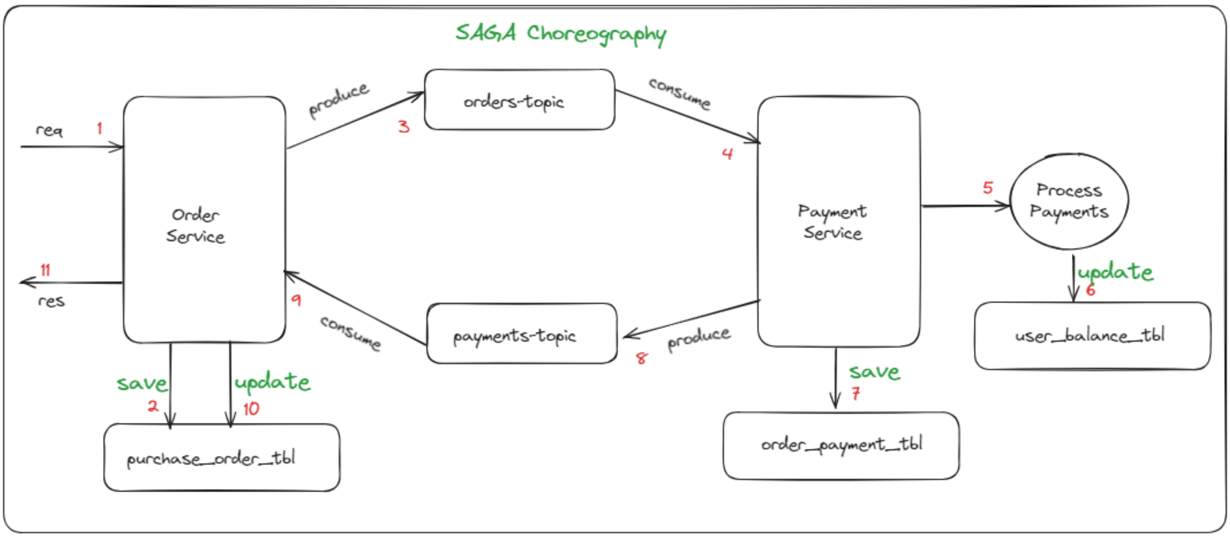
```
{"userId":101,"productId":3547,"price":1200}
{"userId":101,"productId":3923,"price":3000}
```

```
{"userId":101,"productId":3923,"price":3000,"paymentStatus":"SUCCESS"}
{"userId":101,"productId":3547,"price":1200,"paymentStatus":"FAILED"}
```

USER_ID	AMOUNT
101	1000
102	2000

```
{
  "orderId": "djsge72572-836jgdjs",
  "userId": 101,
  "productId": 3923,
  "price": 3000,
  "paymentStatus": "SUCCESS",
  "orderStatus": "COMPLETED",
  "Date": ""
}
```

```
{
  "orderId": "236dsbjm-27jxbj7eb",
  "userId": 101,
  "productId": 3547,
  "price": 1200,
  "paymentStatus": "FAILED",
  "orderStatus": "CANCELLED",
  "Date": ""
}
```



1. Create Maven Multi Module Project SAGA_Choreography_App
    - Create Common-Bindings Module
    - Create Order-Service Module
    - Create Payment-Service Module
2. Add required dependencies in parent pom.xml,

```
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <dependency>
                <groupId>com.mysql</groupId>
                <artifactId>mysql-connector-j</artifactId>
                <scope>runtime</scope>
        </dependency>

        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <scope>test</scope>
        </dependency>

        <dependency>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <optional>true</optional>
        </dependency>

        <dependency>
                <groupId>io.projectreactor</groupId>
                <artifactId>reactor-test</artifactId>
                <scope>test</scope>
        </dependency>
        <dependency>
                <groupId>org.springframework.kafka</groupId>
                <artifactId>spring-kafka-test</artifactId>
                <scope>test</scope>
        </dependency>
</dependencies>
```

3. Common-Bindings Project Development
    a. Create Entity Classes

        - PurchaseOrder.java   (purchase_orders_tbl)

        - PurchaseOrderPayments.java (purchase_orders_payments_tbl)

        - UserBalance.java (user_bal_tbl)

    b. Create Repositories for entity classes.
    c. Create Required Request & Response Binding Classes

        - OrderRequestDto.java

        - OrderResponseDto.java

        - OrderStatusEnum.java

        - PaymentStatusEnum.java

    d. Add Common-Bindings project as a dependency in order-service & payment-service.

4. Order-Service Project Development
    a. Add Below Dependencies,

```
<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-webflux</artifactId>
        </dependency>

        <dependency>
```

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

            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-devtools</artifactId>
                <scope>runtime</scope>
                <optional>true</optional>
            </dependency>


            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <scope>test</scope>
            </dependency>

            <dependency>
                <groupId>io.projectreactor</groupId>
                <artifactId>reactor-test</artifactId>
                <scope>test</scope>
            </dependency>

            <dependency>
                <groupId>nk.honnur</groupId>
                <artifactId>Common-Bindings</artifactId>
                <version>0.0.1-SNAPSHOT</version>
            </dependency>

        </dependencies>
        <dependencyManagement>
            <dependencies>
                <dependency>
                    <groupId>org.springframework.cloud</groupId>
                    <artifactId>spring-cloud-dependencies</artifactId>
                    <version>${spring-cloud.version}</version>
                    <type>pom</type>
                    <scope>import</scope>
                </dependency>

            </dependencies>
        </dependencyManagement>
```

b. KafkaProduceConfig.java

c. KafkaConsumerConfig.java

d. RestController

- createOrder()
- getOrders()

e. Service

- createOrder (save in db & publish msg to order-topic)
- getAllOrdersFromDB
- consumePaymentsTopicMsg (check pay status & confirm/cancel order)

5. Payments-Service Project Development

a. Add below Dependencies,

```
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-webflux</artifactId>
        </dependency>
        <dependency>
                <groupId>org.springframework.kafka</groupId>
                <artifactId>spring-kafka</artifactId>
        </dependency>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-devtools</artifactId>
                <scope>runtime</scope>
                <optional>true</optional>
        </dependency>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <scope>test</scope>
        </dependency>
        <dependency>
                <groupId>io.projectreactor</groupId>
                <artifactId>reactor-test</artifactId>
                <scope>test</scope>
        </dependency>
        <dependency>
                <groupId>nk.honnur</groupId>
                <artifactId>Common-Bindings</artifactId>
                <version>0.0.1-SNAPSHOT</version>
```

```
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

b. KafkaProduceConfig.java
c. KafkaConsumerConfig.java
d. Service

- handleOrderPayment (listen to orders-topic)

- check order amt & user balance

- if user having sufficent bal then deduct bal and update payment completed

- produce payment status to payments-topic

6. Kafka Setup

a.

Step-1: Download Zookeeper from below URL,

URL : http://mirrors.estointernet.in/apache/zookeeper/

Step-2: Download Apache Kafka from below URL,

URL : http://mirrors.estointernet.in/apache/kafka/

Step-3: Set Path to ZOOKEEPER in Environment variables upto bin folder.

Step-4: Copy zookeeper.properties and server.properties files from kafka/config folder to kafka/bin/windows folder.

Step-5 : Start Zookeeper server using below command from folder,

kafka/bin/windows

Command : zookeeper-server-start.bat zookeeper.properties

Step-6: Start Kafka Server using below command from folder, kafka/bin/windows

Command : kafka-server-start.bat server.properties

    b. Open Kafka tool and connect to it (Offset Explorer)

7. Run Order Service (order-topic will be created)
8. Run Payment Service (payments-topic will be created)
9. Send Post req to order service and verify data