**Saga Pattern**

The Saga pattern will only be efficient in micro service with distributed transactions. In microservice, to complete a process we will perform multiple steps/transaction in multiple services, these transactions in each service are represented as a Saga as per Saga pattern (multiple transactions can be present in each service, so all the transaction in single service is considered to be one Saga).

The beauty of Saga pattern is Sagas can either be executed or compensated (i.e., undone) in case of a failure. For example, if there is a failure occurs in one service, ***the Saga can initiate compensating transactions in other services to undo any partial changes.***

Eg: Below is the distributed transaction flow of a food ordering app

A diagram of a service and restaurant

Description automatically generated

*When a person orders, transaction for order will be completed, then he moves to payment page, then transaction for payment will be started and so on. How we can say it as distributed transaction?*

Even though the user experience might be divided into steps, multiple services are working together behind the scenes to complete the order. These services (Order Service, Payment Service, Restaurant Service) each manage their own data and need to be coordinated to ensure overall consistency.

So, In above example if one Service fail to complete the transaction, all other transactions done by previous service should be rolled back. This is the place where we need to use Saga pattern

**Achieving Saga pattern:**

There are two ways to achieve Saga pattern using

* Orchestration pattern
* Choreography pattern

**Orchestration pattern**

The Orchestration pattern is a design coordinate workflows across multiple services. It involves a central coordinator(Conductor) component that manages the overall flow of execution and ensures all participating

A diagram of a company

Description automatically generated

In above ER diagram, Conductor is the service or queue which manages all the workflow. When a person orders, Conductor will be the single point of service, conductor send request to each service to complete the transaction. Whenever something fails conductor will send some more request to respective service to rollback/compensate the transaction which is made preciously.

*Biggest disadvantage: Single point of failure.*

How to implement this in Spring boot:

* Create a conductor rest service along with all other necessary service required (Order, Restaurant, Payment, Delivery, Notification)
* Whenever a user tries to order food, request first need to go to Conductor service
* From conductor service, rest call to other services should be initiated one by one.
* Whenever something fails conductor need to request to respective service to rollback/compensate the transaction which is made preciously.
* Make sure conductor service more reliable and available

**Choreography** **pattern:**

The Choreography approach is a design pattern for coordinating workflows across multiple services. It differs from the Orchestration pattern by taking a decentralized approach, where services communicate directly with each other rather than relying on a central coordinator(conductor). It can achieved by Event bus.

**A diagram of a restaurant service

Description automatically generated**

In above ER diagram, We are not using a service for single point of contact. Communication between the service is independent. This is achieved by using an event bus like Kafka, JMS etc..

An Event will published by order service(Producer) then the consumer service consumes the event and preforms the transaction and publish an event to another topic present in another queue, then this will be consumed by respective consumer and performs transaction. In case of any failure, an event will be published to a Failure topic then all the consumer consumes it and performs rollback/compensate transaction.

How to implement this in Spring boot:

* Create necessary Services, brokers and topics for the service/as needed (Order, Restaurant, Payment, Delivery, Notification)
* Whenever a user tries to order food, Order service needs to publish event on a topic which will be consumed by Payment service for completing payment transaction. Then payment service will to publish event on a topic which will be consumed by restaurant service and the same process goes on.
* Whenever something fails, an event need to be published on a topic created for failure which will be consumed by all other services to rollback /compensate the transactions if already made.
* Now the services communicate directly with each other without coordinator.