Session - 1

Date
Page

(1) Saga Design Pattern

(a) Why SAGA?

→ The developers faced issue when moved from Monolithic application to Micro Service Architecture.

→ These problems are solved using SAGA pattern

→ Let's take an example of Swiggy, zomato food delivering applications

- a) Choose our meal.
- b) Add them to Cart and checkout
- c) Make Payment
- d) Order gets delivered
- e) Our order is marked as completed after delivery is successful.

↳ In Monolithic it's not a problem as we have 1 database, multiple tables like orders, Payments, Delivery Etc.

↳ Now in 1 single atomic transaction we can do all these steps and if payment fails, everything gets rolled back.

Monolithic

In Single transaction →

```
{
    Create order();
    Validate Payment();
    Deliver order();
    Order marked Successful;
}
```

① U have an Order service it accepts your order and it creates and updates order at order table. (when updation done at the table). (SAGA 1)

② Next events (T_2) gets triggered i.e. Payment services validates the payment done. (SAGA 2)

③ Next events (T_3) gets triggered at delivery services where after successful transaction delivery is scheduled by the delivery partner to our doorsteps. (SAGA-3)

Eventually delivery partners makes / confirms order as successfully delivered.

Now moving to microservice Architecture

(a) Now when we moved to microservices architecture and segregate (divide) the whole Zomato and Swiggy food delivering applications to multiple different microservices like :-

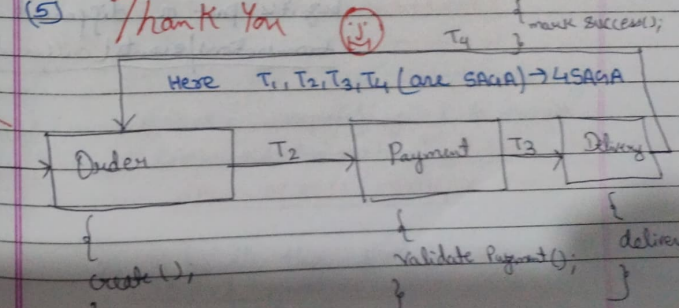
a) Order Service
b) Payment Service
c) Delivery Service

(b) The Happy Case will be as

Steps :-

(1) Your order service accepts the order.
(2) Payment Service validates the payment done
(3) & Delivery service is responsible for delivery of your order at your doorsteps.
(4) When delivered successfully the orders is marked completed in the application.

(5) Thank You :)



(c) If delivery fails / delivery partner not assigned to us:-

Worst Case:

(1) Payment done.
(2) money got deducted and now NO food.
Expectation (3) At least we need to get the money back and order must be marked as cancelled.

Now the problems arisses here in microservices is

→ ~~The~~ If tx

(a) If delivery is failed then the transaction can only rollback to delivery point (deliver()) ~~delivery service~~

and marks delivery as failed but payment can't be reversed bcz it is part of different transactions / events / different applications

Date
Page

Bookish | Madam Points

For this to happen we need a Transaction rollback.
→ Transaction did get rolled back, but only the scope of transaction was in delivery service.
→ The boundary for this transaction ended in Delivery service.

Now what about the order service and payment service?

a) Neither your money is refund/returned with this rollback.

b) Nor your order status changed from waiting to failed/cancelled.

c) Bad User Experience.

Why SAGA introduced?

NOTE:-

WVI
Why SAGA Introduced Interview

1) This is the classic example where your application completely failed to manage distributed transaction [A transaction that spans across multiple microservices].

2) To handle this problem & to handle such distributed transactions issues SAGA Design Pattern came into picture.

What is SAGA ?

a) A SAGA is a sequence of local transactions. ~~Where each transaction updates the data within a service and publishes an event.~~

b) Each Saga has 2 jobs to Do

↳ Updates the current Microservice and make required changes.

↳ Publish events to trigger the next transaction for the next microservices.

What is SAGA ?

a) A SAGA is a sequence of local transactions. ~~Where each transaction updates the data within a service and publishes an event.~~

b) Each Saga has 2 jobs to Do

↳ Updates the current Microservice and make required changes.

↳ Publish events to trigger the next transaction for the next microservices.

How SAGA DP handles failure of any individual SAGA?

In SAGA pattern, when a part (or individual step) of the transaction fails, it handles the failure by compensating actions to undo the work of the previous successful steps:-

Let's understand how it works?

(1) Steps in the SAGA

a) Each service (or microservice) peforms it's local transaction and if successful, triggers the next step.

(b) Example → Place Order → Reserve Inventory → Process Payment.

(2) Failure at a Step

(a) If one step fails (eg Payment service fails to process payment), the SAGA stops further processing.

③ Compensating Actions

(a) The SAGA runs undo actions for the already completed steps to roll back the system to it's previous state.

Example :-

- Cancel Inventory Reservation
  To release the reserved items
- Mark Order as Canceled in the order service.

④ Event - Driven Rollback

The compensating actions are usually triggered by events / transactions. either automatically (or) by a central Orchestrator.

Simple Example:-

Suppose we are booking a trip:-

Steps:-

(1) Book a flight (success)
(2) Reserve a hotel (success)
(3) Book a car rental (fails)

If Step 3, fails, the SAGA will

(a) Cancel the hotel reservation.
(b) Cancel the flight booking

This way, the system ensures that no incomplete (or) inconsistent states are left behind.

Ways to Implement SAGA?

There are 2 types of Saga implementation ways

(a) Choreography-based SAGA

(b) Orchestration-based SAGA

(a) Choreography-based SAGA

- Each microservice listens for events and reacts accordingly.
- There is no central controller, services coordinates by observing and responding to events.

(b) Orchestration-based SAGA

- A central orchestrator (or coordinator) controls the workflow.
- The orchestrator sends commands to services to execute specific tasks in the transaction.

Task of Orchestrator

Disadvantages (choreography)

(a) There's a risk of cyclic dependency between Saga participants because they have to consume each other's commands.

(b) Integration testing is difficult because all services must be running to simulate a transaction

Advantages of Choreography Saga Pattern

(a) Good for simple workflows that require few participants and don't need a coordination logic.

(b) Doesn't require additional service implementation and maintenance.

Disadvantages (Orchestration)

(a) Good for complex workflows involving many participants (or) new participants added over time.

Disadvantages (orchestration)

(a) Additional design complexity requires an implementation of a coordination logic.