Codez Up
Code the Way Up

Saga Pattern: Manage Distributed Transactions | Microservice

By Vikas | January 13, 2021 0 Comment

In this article, we will be learning about the saga pattern which is an alternative to a two-phase commit to managing distributed transactions.

Saga Pattern

A saga can be considered a sequence of local transactions in different microservices.

Each local transaction updates the database of that microservice and then publishes a message or event to trigger the next local transaction in the saga.

If one local transaction fails, the saga pattern executes a series of compensating transactions that rollback of local transactions forming part of the distributed transaction that has already been executed.

Types of Saga Patterns

There are two main different types of Saga pattern implementations.

1. Choreography-Based Sagas

The first is **choreography-based sagas**, where each local transaction publishes domain events that will trigger local transactions in other services until the saga is completed.

So in this example here where the payment service obtains payment from a customer, then the order is created and then the products are reserved for the customer, the order service first sends a payment request event to the message broker to start the saga.

The payment service, which has subscribed to the payment requests event receives this message and attempts to obtain payment from the customer.

It then sends a message indicating if the payment has been received successfully which will be picked up by the order service.

If the payment was not received successfully the saga ends and no further actions are needed.

However, if the payment was received successfully, the order service attempts to create the order and sends an event indicating if this was successful which is picked up by both the product service and the payment service.

If the order creation was not successful then the payment service must execute the compensating transactions to rollback the actions taken to obtain payment from the customer which would most likely involve a refund and sending an email to the customer or notifying them on the screen.

If the order was created successfully, the product service which subscribes to this event will attempt to reserve the products requested for the order.

It will then send an event indicating whether the products were reserved successfully.

If so then the saga ends as no further actions are required.

However, if the products were not reserved successfully both the order service and the payment service will roll back the actions they have performed in this saga through compensating transactions.

As you can see, the way distributed transactions are handled is different from two-phase commit where transactions would not be committed to the database before these have been approved by all participating microservices.

2. Orchestrator-Based Sagas

The alternative way of implementing sagas is by using an orchestrator.

In **orchestrator-based sagas**, an orchestrator which is usually spawned specifically for each saga instructs the microservices involved in the saga what local transactions to execute, so basically it coordinates the whole saga.

To use the same example that was taken for the choreography- based saga and implement it using an orchestrator.

The ordered service creates an order saga object which doesn't need to be a separate service it can just be an object residing inside the order service.

The order creation saga pattern will instruct the payment service to request payment and will wait for a message indicating if payment was received successfully.

If not then the order creation saga terminates immediately as no more actions are required.

However, if payment was received successfully, the creation saga sends a message to the order service to create an order.

The ordered service will then reply to the order creation saga on whether the order was created successfully.

If the order wasn't created successfully the order creation saga instructs the payment service to roll back its previous actions taken to request payment and then this saga terminates.

On the other hand, if the order was created successfully the order creation saga sends a request to the product service to reserve product the product service with a new like with a message indicating if the products were reserved successfully.

If so then the saga can terminate as no further actions are required in the saga.

However if the product service didn't manage to reserve the products, then the order creation saga will notify both the order service and payment service to roll back the previous actions they have taken in the saga.

The overall advantages of the saga pattern are that it enables an application to maintain data consistency across multiple services.

However, as you can see the development process becomes more complex and also takes longer as compensating actions need to be developed for each transaction to be able to roll back.

Also, ideally, the microservice should publish the event and update the database atomically to prevent race conditions and potential data inconsistencies.

This implies that certain data related patterns should be used that support this.

One of the most popular is event sourcing which we discussed in earlier articles but it's also possible with other less common patterns such as using a database trigger to write events to a table that is pulled by a process that publishes these events.

I hope you guys like the article, feel free to drop any comments in the comment section down below.

Sharing is Caring :

Category: Microservice Tags: Microservice