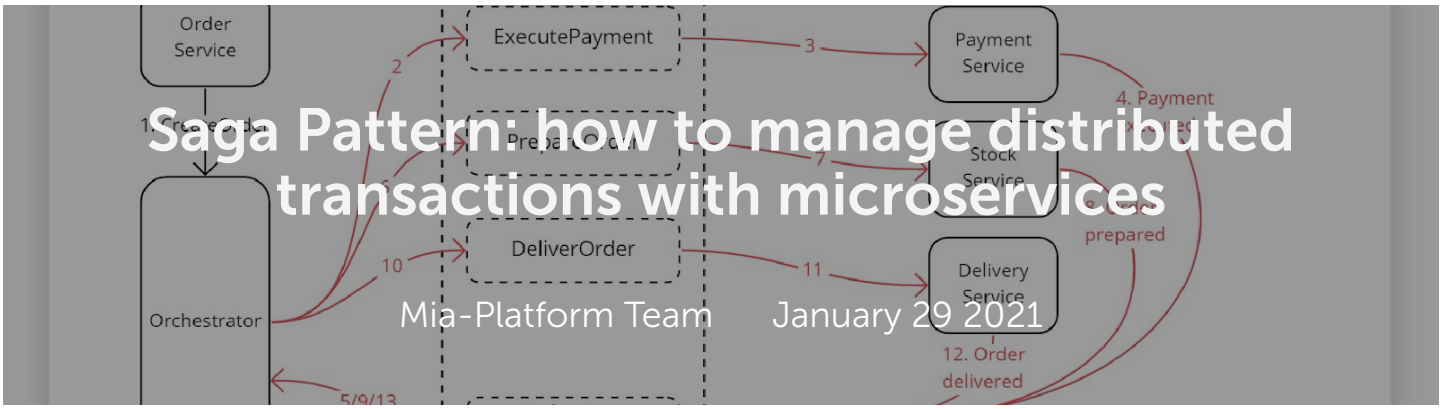
mia Platform



Saga Pattern: how to manage distributed transactions with microservices

Mia-Platform Team January 29 2021

In the context of microservices, one of the problems that we often encounter is **the need to guarantee data consistency, updated in parallel by different actors.**

In this case, the Saga architectural pattern is becoming popular: let's find out how it works and which problems it solves.

Data Consistency: what is it and which problems does it imply?

Guaranteeing data consistency means ensuring that data is significantly and efficiently usable in the company's applications.

An example of data consistency comes from an e-commerce website: after the payment of an order, it is necessary to update both the payments tables and the order tables, to ensure that the order is paid and that it can be shipped. These updates potentially come from different sources and must preserve data coherence.

Securing data consistency is a sensitive subject.

- ensuring data consistency in a distributed operation among different systems;
- guaranteeing *error safety*.

In order **to secure the consistency of data** is fundamental to ensure that the execution flow of the different operations on diverse microservices respect a certain order. This implies **guaranteeing *error safety***, that is the presence of a remediation mechanism which restores the right situation in case of a mistake.

To give a concrete example, let's analyze **a modern food service delivery**: the operations that "carry on" an order are multiple and often involve different actors, each of which executes one or more operations, as the use case below.

- The user clicks on "Order" button;
- The order is created;
- The payment service authorizes the user to pay the order;
- The user pays the order;
- The payment service updates the order;
- The service of the order notifies the restaurant;
- The restaurant confirms and enters a time for the withdrawal;
- The delivery service notifies the rider;
- The rider takes and delivers the order;
- The order is closed.

It is not always easy, with all the steps and actors listed above, to guarantee data consistency in all the points and to avoid mistakes along this chain.

This is where **a data management pattern,** that is becoming more and more popular as a preferred response in this kind of situation, comes to help: the *Saga Pattern*.

What is the Saga Pattern?

The idea behind the Saga Pattern, as described in the previous paragraph, is the management of distributed transactions, from start to finish, each of which is, indeed, a saga.

that are part of one saga, or that must be performed in a predetermined order and whose effect is not compromised by any intermediate transactions belonging to different sagas.

Therefore, this pattern helps **to manage the consistency of data in the execution of distributed transactions among various microservices**. It involves different actors (services) that go to act on the same entity through individual transactions aimed at updating a common data.

The goal is twofold: to maintain the identity of data and carry out compensation actions to restore it in the event of an error.

Comparison of different approaches to Saga Pattern

There are mainly **two approaches** to manage the pattern under consideration:

- **events/choreography** → no coordinator, the services carry on the saga by working together, without having someone to control them;
- **commands/orchestration** → centralized control, managed by an orchestrator.

Both of the above approaches have pros and cons, and clearly, there is not a universal rule that establishes whether one approach should be used rather than the other: it always depends on the project to be implemented.

Events/choreography approach

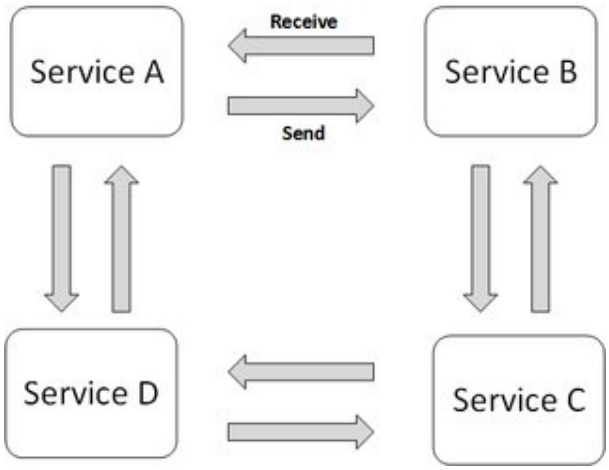
The events/choreography approach establishes that all microservices involved work triggered by specific events, like proper choreography.

There isn't a coordinator that manages everything, **but every service knows what to do and when to do it**. As a matter of fact, the union of the work of each service enables to start, progress and finish the saga.

In this way, we have a decentralized logic, distributed among the various services that deal with the saga.

Taking up the previous example of the food delivery order system, the choreography of events could be as follows:

- **order-service**: opens the order → sends *"open order" event*
- **payment- service**: receives *"open order"* event → processes payment → sends *"order paid"* event
- **delivery-service**: receives *"paid order"* event → sends the order to the user's address → sends *"order sent"* event
- .....



As you can see in the use case above, each service knows exactly its environment, what to do and when to do it. The saga progresses naturally despite the fact that it involves different microservices, which one does a different thing: they do not need to communicate directly, despite carrying out a sequence of activities indirectly linked to each other.

This approach has **several advantages**:

- there is no strong dependence on a central component → **there is no single point of failure**;
- it is ideal for transactions involving a few actors, as it is simple and quick to develop;
- *Agile prone*: each team can work on their own service regardless of the other teams / microservices (once the format and nomenclature of messages/events have been agreed).

However, it also has some **disadvantages**:

- it is not suitable for managing complex sagas:
  - it is a system that, with the addition of new services, tends to quickly become more complex;

manages a specific event and, consequently, testing the system becomes more complex;

- when many actors are involved and enrich a common database, it may happen that you do not always have full control over the data if you do not use an actor responsible for ensuring its persistence;

- without an "overview" of the saga, **the risk of cycles among dependencies increases**.

Commands/Orchestration Approach

This approach, unlike the previous one, involves the use of **a central service that manages and controls the entire flow of the saga**. The service is, as the name of the approach suggests, the orchestrator.

The orchestrator has the role of controller and all the actors involved in the transaction interface with it.

According to this approach, the orchestrator directs the flow, sending **commands** to each service in the form of messages through the *Message Broker*. Upon receiving this command, the service implements its own logic to carry on the saga and responds by sending an **event** via the *Message Broker*, which will be managed by the orchestrator.

In this way, the orchestrator will be able to carry on the saga by directing its "orchestra" of services.

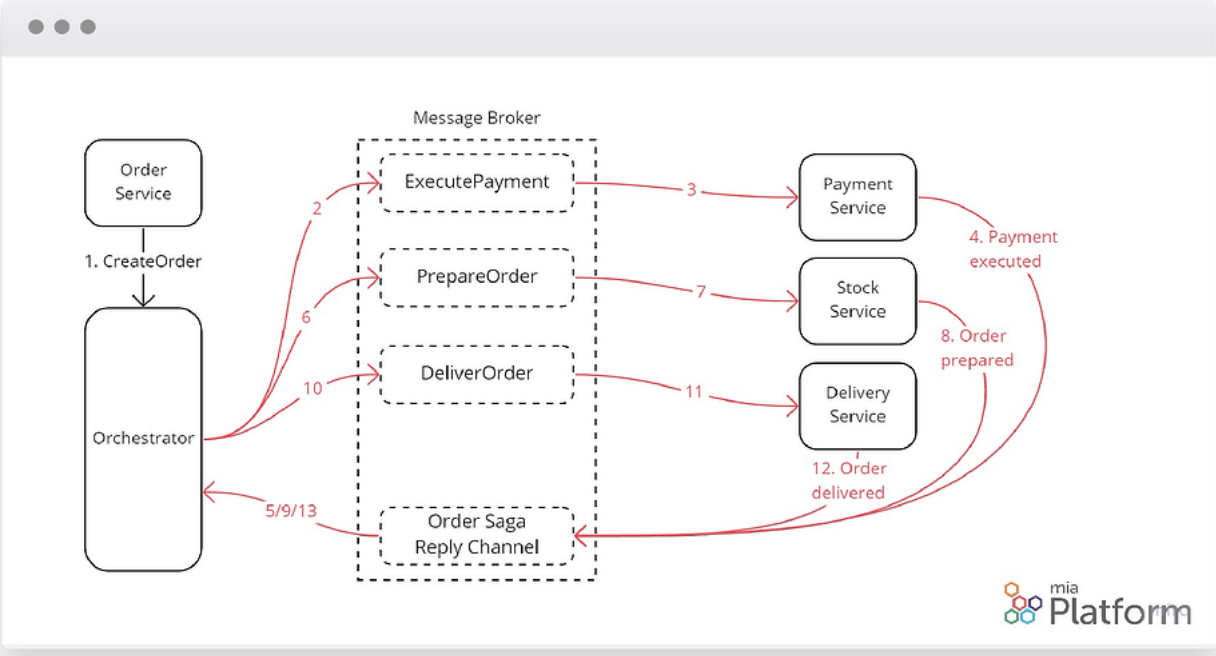
The previous example of the order on a food delivery system could be, this time, the following:

- **order-service**: opens the order → sends *"open order"* event

- **orchestrator**: receives *"open order"* event → sends *"payment attempt"* command

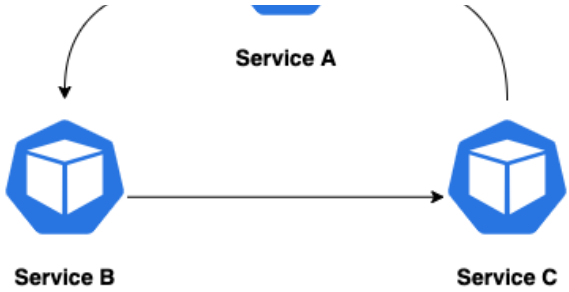
- **orchestrator**: receives *"paid order"* event → sends *"order shipping"* command
- **delivery-service**: receives *"order dispatch"* command → ships order to the user's address → sends *"order shipped"* event
- ....

Also in this case, the services know their environment only and implement that small part of the logic linked to the saga. However, this time they have a sort of reference point, as can be seen in the following diagram.



The advantages of this approach:

- having a central point that provides an overview helps **to avoid cycles among services** or the possibility that N services are called cyclically,



- **the business logic is concentrated in a single point**, consequently:
  - it can be easily managed and modified;
  - the addition of a new step involves only implementing the functionality independently, and then modifying the orchestrator to update the saga flow, which is a transparent operation for the other services;

- *Agile prone*: as with the choreography approach, also in this case the services are separated from each other and **the teams can work independently** (once the format and nomenclature of messages/events have been agreed upon);

- Easier **rollback management**.

The disadvantages are the following:

- centralized logic can also represent a risk since there is ***a single point of failure***. In this way, if the orchestrator doesn't work, all the flow is blocked;

- it is necessary to implement ***an additional service***, besides to the services that manage the saga: the orchestrator;

- **the number of interactions is exactly duplicated**: each message (event) of the choreography approach corresponds to two messages (command + event) in the orchestrator approach.

Which approach would you choose?

In the previous chapter, we described the pros and cons of both the *events/choreography* and *commands/orchestration* approaches. Which one to

a saga that involves two microservices and a very simple flow: the added value obtained from the introduction of an orchestrator would be canceled out by a considerable increase in complexity, especially at the beginning, linked to the *commands/orchestrator approach*.

Which approach we chose

At Mia-Platform we had to face this problem and, consequently, we had to choose which approach best suited our case.
Analyzing the needs, the various steps of the saga flow and the designeted actors, we opted for the **Commands / Orchestration approach.** As we didn't find on the web a ready-to-use implementation of an orchestrator, we decided to develop a new platform component: the Flow Manager.

Read this article to learn more about Flow Manager, the saga orchestrator of Mia-Platform.

The article was written by Francesco Francomano, Senior Full Stack Developer, and Giuseppe Manzi, Full Stack Specialist.

Leave your comment

First Name*