

Article

Solving distributed transaction management problem in microservices architecture using Saga

[Site feedback](#)

Advice on how Saga fits in the microservices world and its different subpatterns

 Save  Like

By Nampreet Pal Singh, Amit Deshpande

Published September 17, 2020

One of the problems to solve in a microservices architecture is how to handle a transaction across multiple services. Saga is an architectural pattern that provides an elegant approach to implement a transaction that spans multiple services and is asynchronous and reactive in nature. Each service in a Saga performs its own local transaction and publishes an event. The successive services listen to that event and perform the next local transaction. If one transaction fails for some reason, the Saga also executes compensating transactions to undo the impact of the preceding transactions.

Estimated time

It should take you less than 10 minutes to read this article.

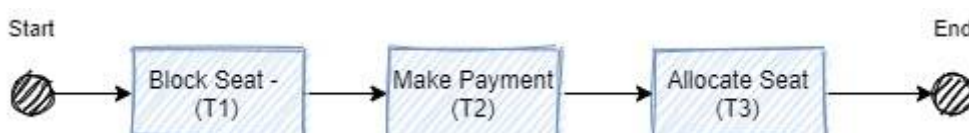
Complexity in distributed transaction management in microservices architecture

A microservice, from its core principles and in its true context, is a distributed system. A transaction is distributed to multiple services that are called sequentially or parallelly to complete the entire transaction. With a microservices architecture, the most common pattern is database per microservice, so transactions also need to span across different databases.

Site feedback

[Figure 1](#) shows an example of a simple airline flight booking scenario implemented using a microservices architecture. There would be one microservice to block a seat, another to accept payments, and finally, another microservice to allocate the blocked seat, each implementing a local transaction to realize their functionalities. To successfully complete the flight booking process for a traveler, all three steps must be completed. If any of the steps fails, all of the completed preceding steps must roll back. Since the overall transaction boundary crosses multiple services and databases, it is considered to be a distributed transaction.

Figure 1. Airline flight booking scenario



With the advent of microservices architecture, there are two key problems with respect to distributed transaction management:

- **How to maintain a transaction's automaticity.** Automaticity implies that all of the steps in the transaction must be successful or if a step fails, then all of the previously completed steps should be rolled back. However, in a microservices architecture, a transaction can consist of multiple local transactions handled by different microservices. Therefore, if one of the local transactions fails, how can you roll back the successful transactions that previously completed?
- **How to manage the transaction isolation level for concurrent requests.** The transaction isolation level specifies the amount of data that is visible to a statement in

a transaction, specifically when the same data source is accessed by multiple service calls simultaneously. If an object from any one of the microservices is persisted to the database while another request reads the same object at the same time, should the service return the old data or new?

It is crucial to address these two problems while designing microservices-based applications. Below are the two approaches to address these problems:

1. Two-phase commit (2PC)
2. Saga

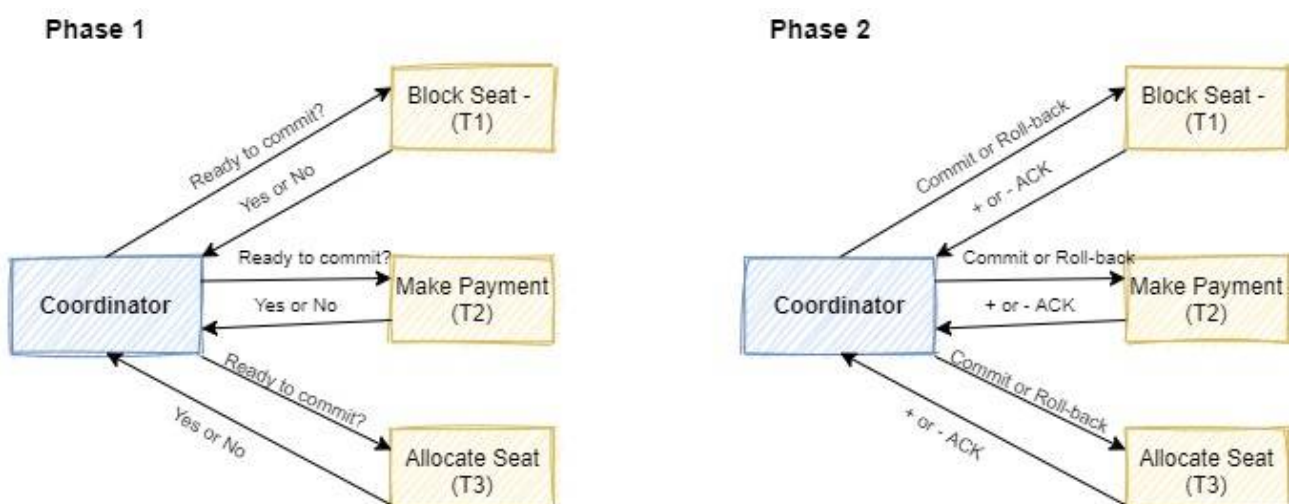
[Site feedback](#)

Two-phase commit (2PC)

Two-phase commit is a well known pattern in database systems. This pattern can also be used for microservices to implement distributed transactions. In a two-phase commit, there is a controlling node that houses most of the logic and participating nodes (microservices) on which the actions are performed. It works in two phases:

- **Prepare phase (Phase 1):** The controlling node asks all of the participating nodes if they are ready to commit. The participating nodes respond with yes or no.
- **Commit phase (Phase 2):** If all of the nodes replied in the affirmative, then the controlling node asks them to commit. Even if one node replies in the negative, the controlling node asks them to roll back.

Figure 2. Two-phase commit




Even though 2PC can help provide transaction management in a distributed system, it also becomes the single point of failure as the onus of a transaction falls onto the coordinator. With the number of phases, the overall performance is also impacted. Because of the chattiness of the coordinator, the whole system is bound by the slowest resources since any ready node has to wait for confirmation from a slower node. Also, typical implementations of such a coordinator are synchronous in nature, which can lead to a reduced throughput in the future. 2PC still has the following shortcomings:

- If one microservice becomes unavailable in the commit phase, there is not a mechanism to roll back the other transaction.
- Other services must wait until the slowest service finishes its confirmation. The resources used by the services are locked until the whole transaction is complete.
- Two-phase commits are slow by design due to their dependence on the transaction coordinator. This can cause scalability issues, particularly in a microservices-based application and in a roll-back scenario involving many services.

[Site feedback](#)

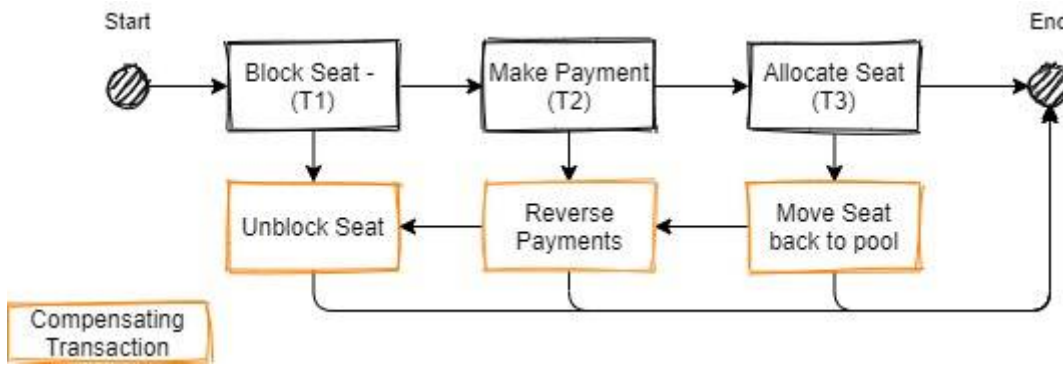
Saga

A Saga, as described by Hector Garcia-Molina and Kenneth Salem their 1987 Association for Computing Machinery [article](#) , is a sequence of operations performing a specific unit of work and are generally interleaved with each other. Every operation that is part of the Saga can be rolled back by a compensating action. The Saga guarantees that either all operations complete successfully or the corresponding compensation actions are run for all executed operations to roll back any work previously done.

A compensating action must be *idempotent* and must have the capability to be *retried* until it is executed successfully, essentially making it an action that just cannot fail and no manual intervention is required to solve its failure. The Saga Execution Coordinator (SEC) provides that guarantee and capability to the overall flow, making it a transaction that is either successful or aborted successfully with necessary rollbacks.

The following diagram shows how a Saga can be visualized for the aforementioned flight booking scenario.

Figure 3. Saga flow



How the Saga pattern helps in a distributed transaction scenario

Microservices introduced another set of problems for managing transactions, as each of the domain-driven services is deployed individually and running in isolation. With a microservices architecture, a single business process brings multiple microservices together to provide an overall solution. It is very difficult to implement ACID (Atomicity, Consistency, Isolation, Durability) transactions using a microservices architecture and it's impossible in some cases. For example, in the aforementioned flight booking example, a microservice with the block seat functionality can't acquire a lock on the payment database, since it is an external service in most cases. But some form of transaction management is still required, so these transactions are referred to as *BASE* transactions: Basic Availability, Soft state, Eventual consistency. Compensating actions must be taken to revert anything that occurred as part of the transaction.

This is where the Saga pattern fits perfectly, as it helps to:

- Maintain data consistency across multiple microservices without tight coupling.
- Perform better compared to 2PC.
- Offer no single point of failure.
- Keep the overall state of the transaction eventually consistent.

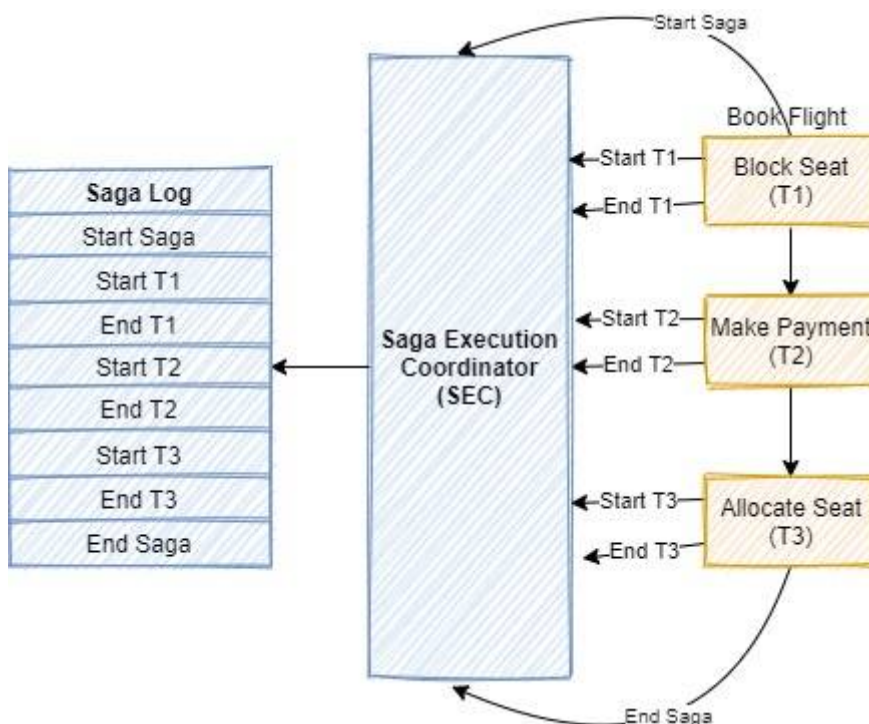
Saga Execution Coordinator

The Saga Execution Coordinator (SEC) is the core component for implementing a successful Saga flow. It maintains a Saga log that contains the sequence of events of a particular flow. If a failure occurs within any of the components, the SEC queries the logs

and helps identify which components are impacted and in which sequence the compensating transactions must be executed. Essentially, the SEC helps maintain an eventually consistent state of the overall process.

If the SEC component itself fails, it can read the SEC logs when coming back up to identify which of the components are successfully rolled back, identify which ones were pending, and start calling them in reverse chronological order.

Figure 4. Saga Execution Coordinator



Different ways to implement the Saga pattern

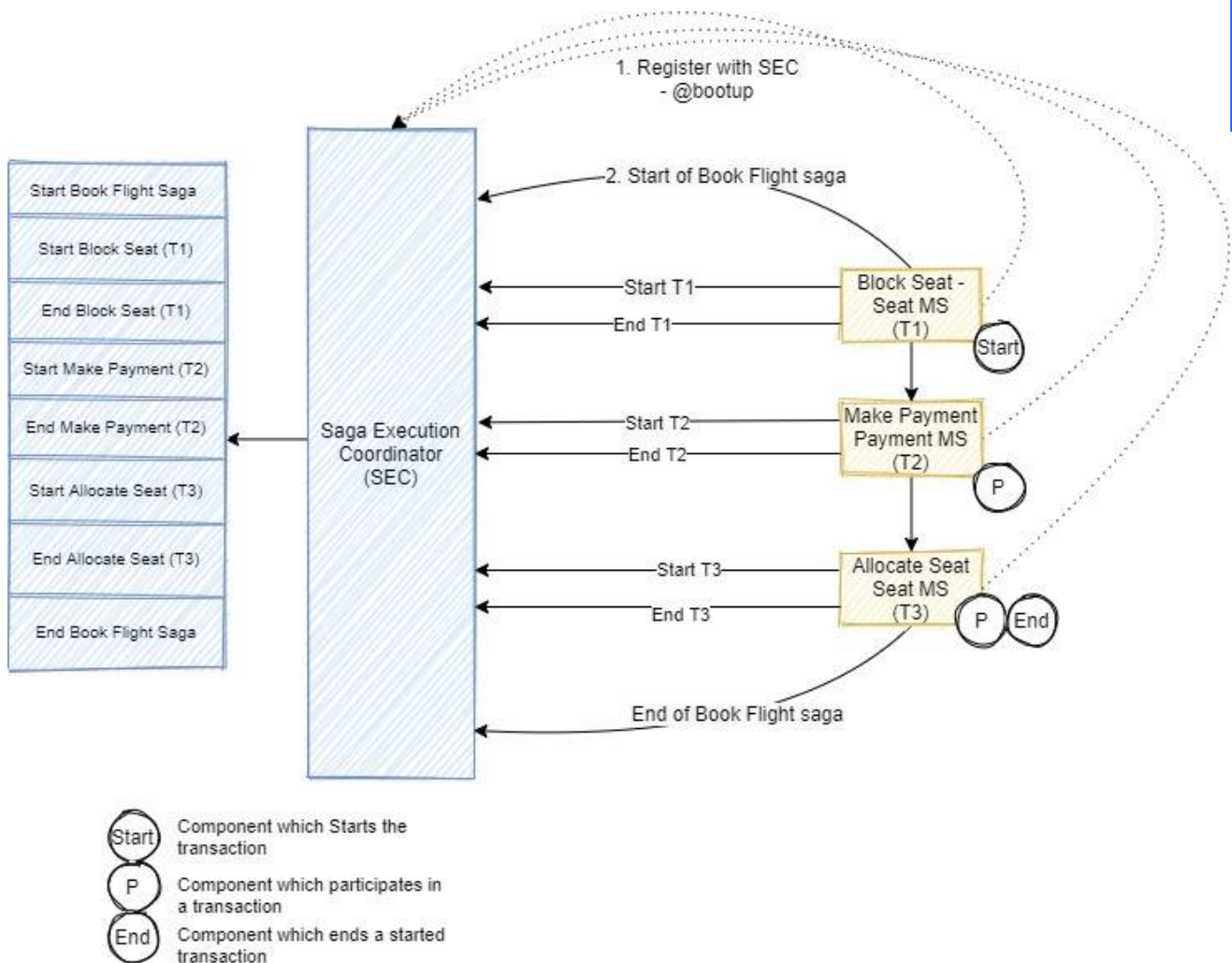
There are two logical ways to implement the Saga pattern: choreography and orchestration.

Choreography

In the Saga choreography pattern, each individual microservice that is part of a process publishes an event that is picked up by the successive microservice. You must make

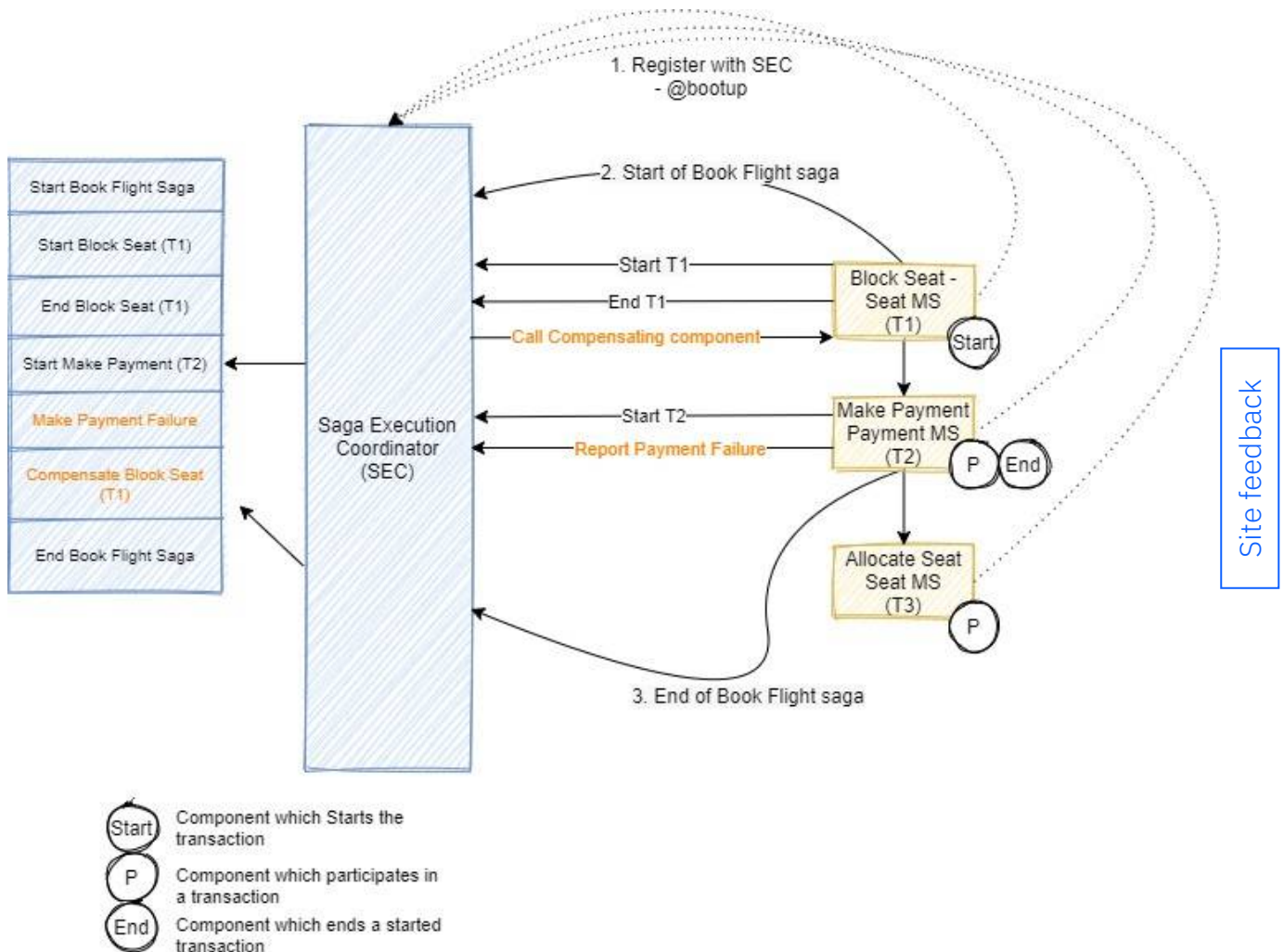
decisions early in the microservice development lifecycle to understand if it will be part of a Saga pattern or not, since you must choose an appropriate framework that will help implement this pattern. To adopt a particular framework code, the microservice must be decorated with annotations, class initializations, or other configuration changes. In the Saga choreography pattern, the SEC can be embedded within the microservice or in most of the scenarios is a standalone component.

Figure 5. Saga choreography pattern success scenario



Whenever a service comes up, it registers itself with the SEC which makes it available to be part of a transaction that may span various microservices. The SEC maintains the sequence of events in its log, which helps it make a decision about the compensating services to call in case of failure and the sequence. The following diagram shows how the logs are maintained in a failure scenario.

Figure 6. Saga choreography pattern failure scenario




The participating microservice informs the SEC of the failure and then it is the responsibility of the SEC to roll back all of the other transactions that were completed before it. The diagram shows that the payment microservice fails, which triggers the SEC to roll back the block seat service. If the call to the block seat service fails, it is the responsibility of the SEC to retry it until it succeeds.

The Saga choreography pattern is ideal when you start your microservices journey (essentially a greenfield development) and understand that it is necessary to introduce process microservices in due course. The following frameworks are available to implement the Saga choreography pattern:

- [Eclipse MicroProfile LRA](#) [↗](#) (Long Running Actions) is an implementation of distributed transactions in Saga for HTTP transport based on REST principles. A release candidate is currently in progress.
- [Eventuate Tram Saga](#) [↗](#) is a Saga orchestration framework for Spring Boot and Micronaut microservices, and is based on the Eventuate Tram framework.
- [Axon Saga](#) [↗](#) is a lightweight framework that helps you build scalable and extensible applications by addressing these concerns directly in the architecture. An Axon Saga framework implementation can be integrated with the majority of Java frameworks to

implement a Saga flow. It's a popular choice with microservices based on the Spring Boot framework.

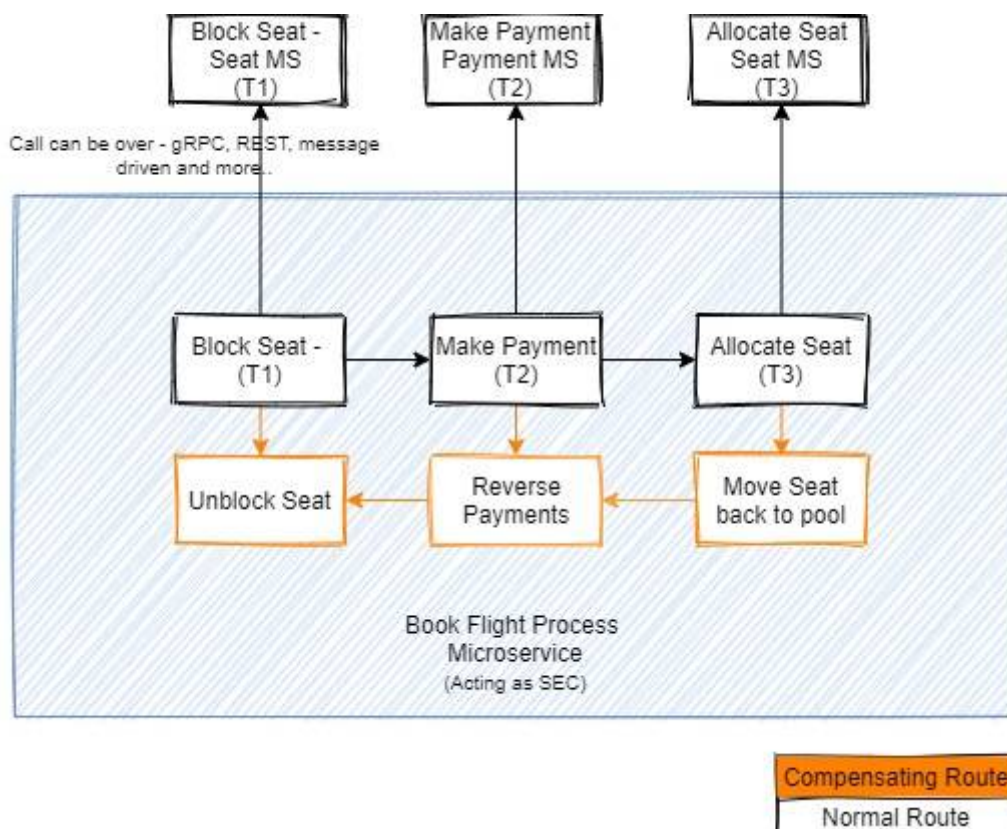
- [Seata](#)  is an open source, distributed transaction solution dedicated to providing high performance and easy-to-use distributed transaction services. Seata also integrates well with the majority of Java frameworks and is popularized by Alibaba for solving their distributed transaction problems.

Orchestration

As the name of the Saga orchestration pattern suggests, there is a single orchestrator component that is responsible for managing the overall process flow. If the process encounters an error while calling any individual microservice, then it is responsible for calling the compensating service too. The orchestrator helps model the Saga flow, but also relies on the underlying framework to call the services in a sequence and make compensating calls if any of the services fail.



In the following diagram, the book flight process microservice encapsulates calls to various microservices in a particular sequence. If any one of those calls fail, then rollback happens by calling the compensating action of those microservices.

Figure 7. Saga orchestration pattern



The Saga orchestration pattern is ideal for scenarios where you already built your microservices and now want to create a process flow using these microservices. There must be a compensating service available and adding another piece of code is not necessary. This is unlike the Saga choreography pattern, where you must align the code with a framework that you chose to implement the Saga.

The following frameworks are available to implement the Saga orchestration pattern:

- [Camunda](#)  is a Java-based framework that supports the Business Process Model and Notation (BPMN) standard for workflow and process automation.
- [Apache Camel](#)  provides implementation for the Saga EIP (Enterprise Integration Pattern), a way to define a series of related actions in a Camel route that should be either completed successfully (all of them) or not-executed or compensated.
- [IBM App Connect](#) allows you to draw out a flow using various built-in adapters and configure its properties appropriately to create a Saga flow.

[Site feedback](#)

Summary

In a microservices world where each microservice has its own data store, it is complex and difficult to keep the consistency of data. Saga is one of the best ways to implement BASE transactions and ensure the eventual consistency of the data in a distributed architecture. Optimal implementation of Saga requires a mindset change for both business and development teams.



Legend 

Categories



Databases

Microservices

Table of Contents

