(/cs/) (https://www.baeldung.com/cs/)

Saga Pattern in Microservices

Last modified: June 23, 2021

by baeldung (https://www.baeldung.com/cs/author/baeldung)

Programming (https://www.baeldung.com/cs/category/core-concepts/programming)

1. Introduction

From its core principles and true context, **a microservice (/spring-microservices-guide)-based application is a distributed system**. The overall system consists of multiple smaller services, and together these services provide the overall application functionality. Although this architectural style provides numerous benefits, it has several limitations as well. One of the major problems in a microservice architecture is how to handle a transaction that spans across multiple services (/transactions-across-microservices).

In this tutorial, we'll explore the Saga architecture pattern that lets us manage distributed transactions in a microservice architecture.

2. Database per Service Pattern

One of the benefits of microservice architecture is that it lets us choose the technology stack per service. For instance, we can decide to use a relational database for service A and opt for a NoSQL database for service B.

This model lets the services manage domain data independently on a data store that best suites its data types and schema. Further, it also lets the service scale its datastores on-demand and insulates it from the failures of other services.

However, at times a transaction (/transactions-intro) can span across multiple services, and ensuring data consistency across the service database is a challenge. In the next section, let us examine the challenge of distributed transaction management with an example.

3. Distributed Transaction

To demonstrate the use of distributed transactions, we'll take an example of a railway seat reservation system implemented with microservice architecture. There is a microservice to block the seats, one that accepts payment and another one that allocates the booked seat. Each of these microservices performs a local transaction to implement the individual functionalities:

To ensure a successful ticket booking, all three microservices must complete the individual local transactions. If any of the microservice fails to complete its local transaction, all of the completed preceding transactions should roll back to ensure data integrity. This is an example of a distributed transaction as the transaction boundary crosses multiple services and databases.

4. Challenges of Distributed Transaction

In the previous section, we've provided a real-life example of a distributed transaction. Distributed transactions in a microservice architecture pose two key challenges.

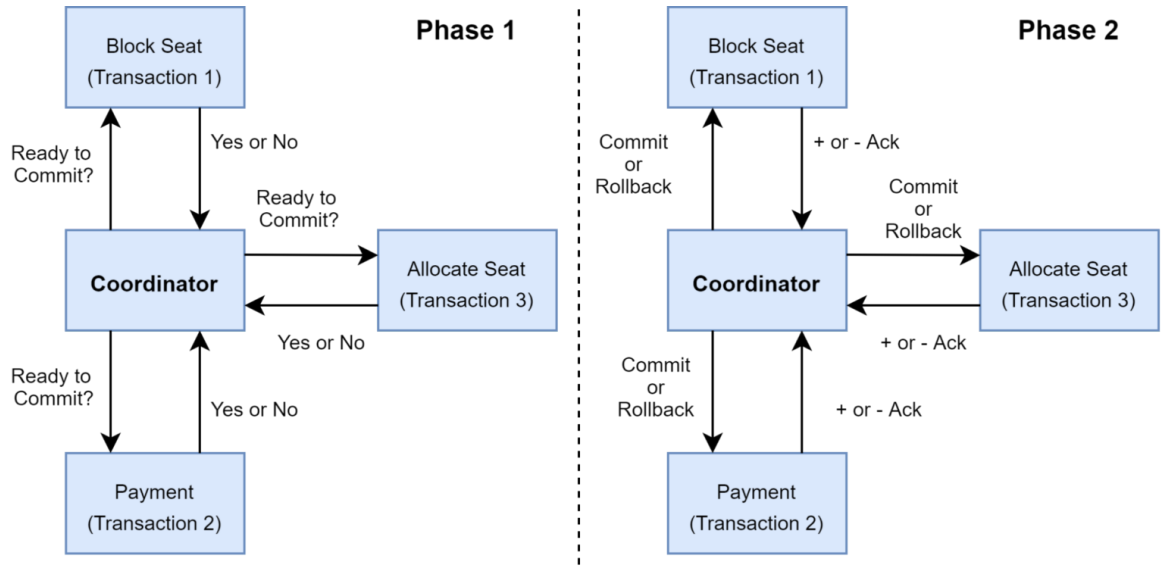
The first one is maintaining ACID. To ensure the correctness of a transaction, it must be atomic, consistent, isolated, and durable (ACID). The atomicity ensures that all or none of the steps of a transaction should complete. The consistency takes data from one valid state to another valid state. Isolation guarantees that concurrent transactions should produce the same result that sequentially transactions would have produced. Lastly, durability means that committed transactions remain committed irrespective of any type of system failure. **In a distributed transaction scenario, as the transaction spans across several services, it always remains a key concern to ensure ACID**.

The second one is managing the transaction isolation level. It specifies the amount of data that is visible in a transaction when the other services access the same data simultaneously. In other words, if one object in one of the microservice is persisted in the database while another request reads the data, should the service return the old or new data?

5. Understanding Two-Phase Commit

The two-phase commit protocol (2PC) **a widely used pattern to implement distributed transactions**. We can use this pattern in a microservice architecture to implement distributed transactions.

In a two-phase commit protocol, there is a coordinator component that is responsible for controlling the transaction and contains the logic to manage the transaction. The other component is the participating nodes (e.g., the microservices) that execute their local transactions:



As the name indicates, the two-phase commit protocol executes a distributed transaction in two phases:

1. **Prepare Phase:** The coordinator asks the participating nodes whether they are ready to commit the transaction. The participants returned with a *yes* or *no*
2. **Commit Phase:** If all the participating node responds affirmatively in phase 1, then the coordinator asks all of them to commit. If at least one node returns negative, the coordinator asks all participants to roll back their local transactions

6. Problems with 2PC

Although 2PC is useful a means to implement a distributed transaction, it has the following shortcomings:

- The **onus of the transaction is on the coordinator node,** and it can become the single point of failure
- All other services need to wait until the slowest service finishes its confirmation. Thus, the overall performance of the transaction is bound by the slowest service
- The two-phase commit protocol is **slow by design due to the chattiness and dependency on the coordinator**. Thus, it can lead to scalability and performance issue in a microservice-based architecture involving multiple services
- Two-phase commit protocol is **not supported in NoSQL databases**. Thus, in a microservice architecture where one or more services use NoSQL databases, a two-phase commit cannot be applied

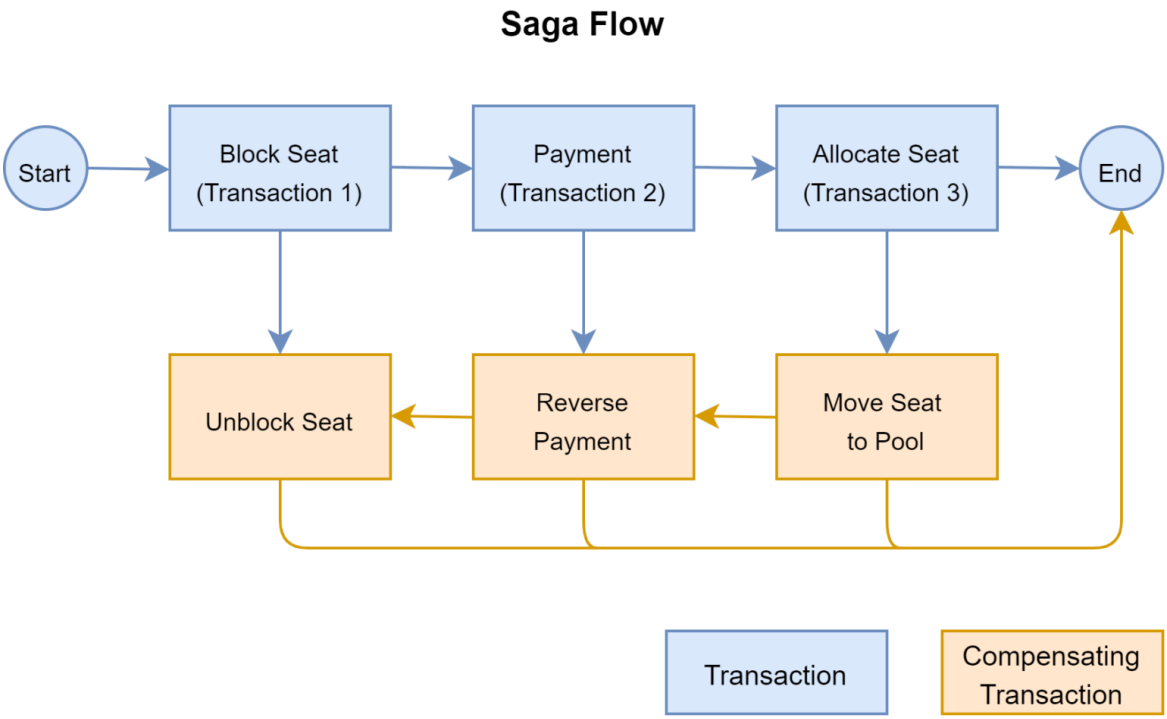
7. Introduction to Saga

7.1. What Is Saga Architecture Pattern?

The Saga architecture pattern **provides transaction management using a sequence of local transactions**. A local transaction is the unit-of-work performed by a saga participant. Every operation that is part of the Saga can be rolled back by a compensating transaction.

Further, the Saga pattern guarantees that either all operations complete successfully or the corresponding compensation transactions are run to undo the work previously completed.

In the Saga pattern, **a compensating transaction must be *idempotent* and *retryable***. These two principles ensure that a transaction can be managed without any manual intervention. The Saga Execution Coordinator (SEC) ensures guarantees these principles:

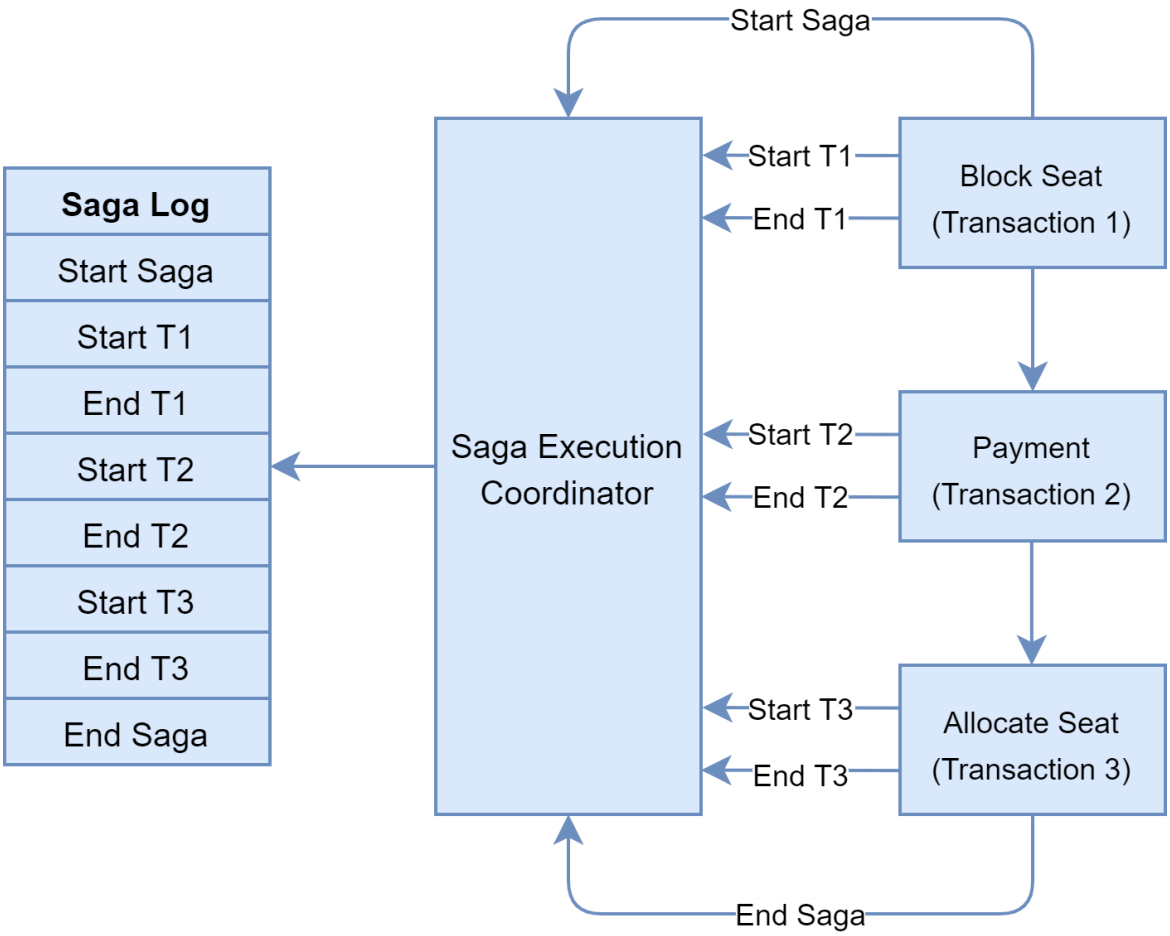


The above diagram shows how to visualize the Saga pattern for the previously discussed railway ticket booking scenario.

7.2. The Saga Execution Coordinator

The **Saga Execution Coordinator is the central component to implement a Saga flow**. It contains a Saga log that captures the sequence of events of a distributed transaction. For any failure, the SEC component inspects the Saga log to identify the components impacted and the sequence in which the compensating transactions should execute.

For any failure in the SEC component, it can read the Saga log once it's coming back up. It can then identify the transactions successfully rolled back, which ones are pending, and can take appropriate actions:

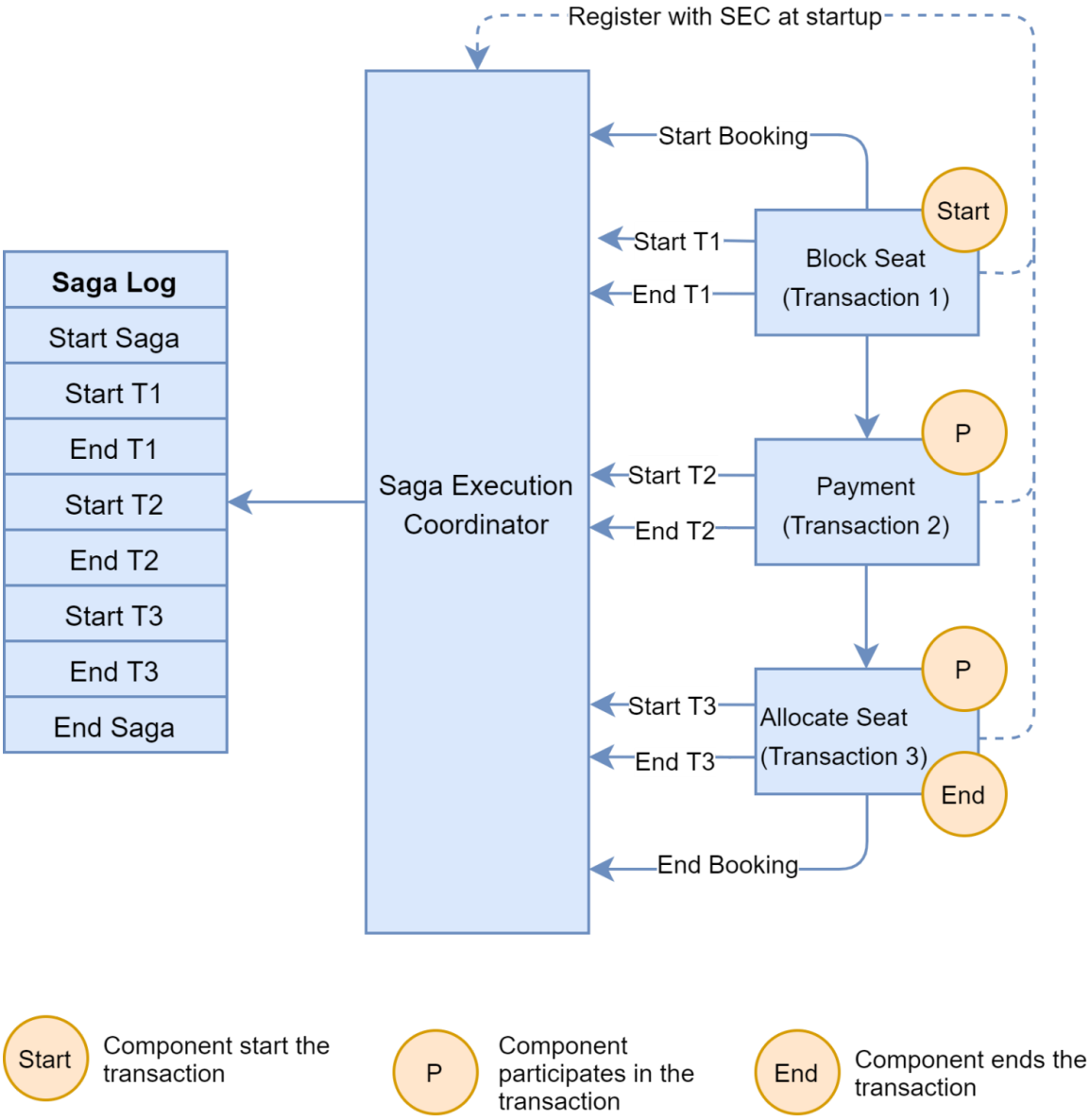


There are two approaches to implement the Saga pattern: choreography and orchestration. Let's discuss them in the next sections.

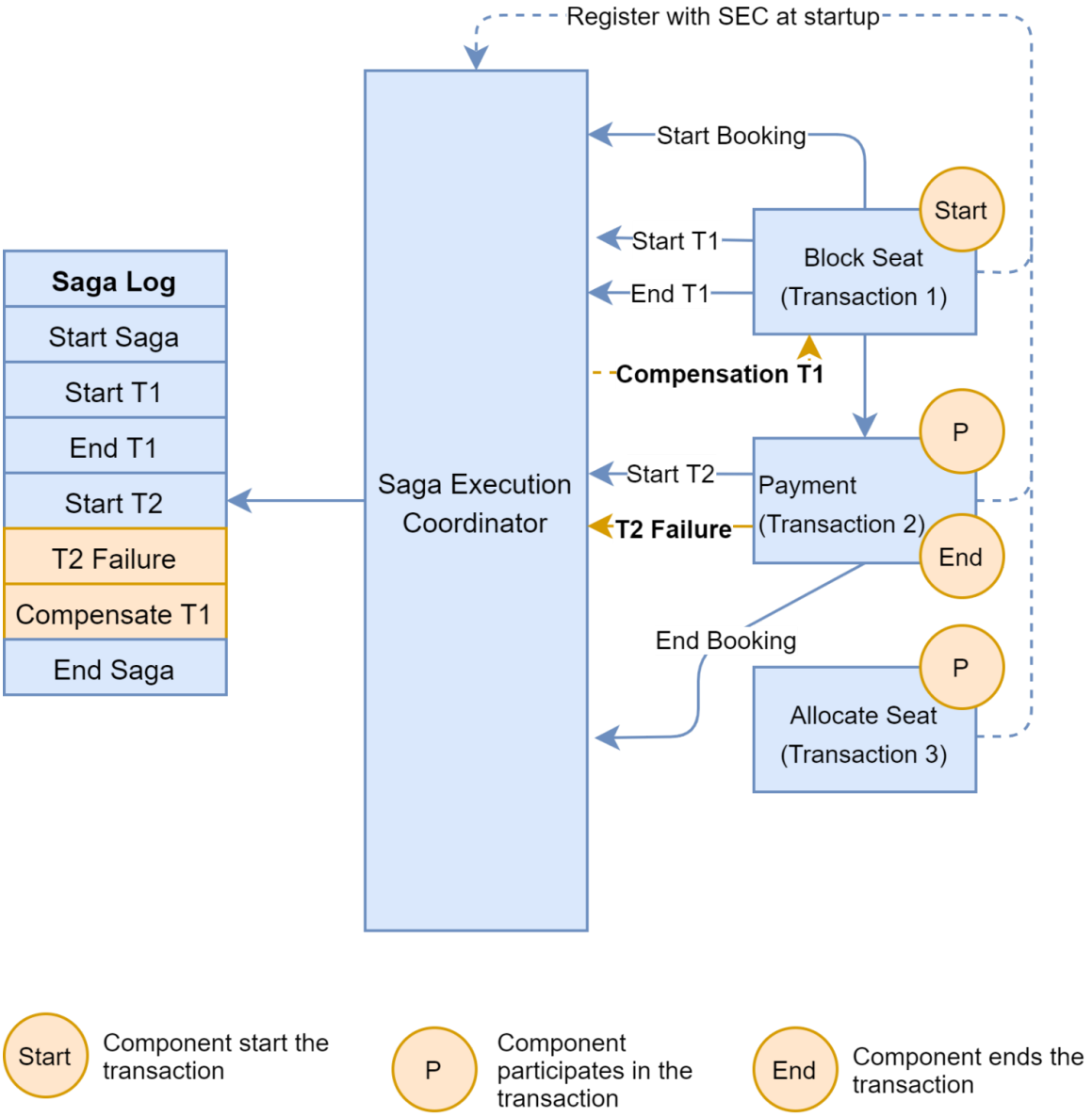
7.3. Implementing Saga Choreography Pattern

In the Saga Choreography pattern, **each microservices that is part of the transaction publishes an event that is processed by the next microservice**. To use this pattern, one needs to make a decision whether the microservice will part of the Saga. Accordingly, the microservice needs to use the appropriate framework to implement Saga. In this pattern, the Saga Execution Coordinator is either embedded within the microservice or cab be a standalone component.

In the Saga, choreography flow is successful if all the microservices complete their local transaction and there is no failure reported by any of the microservice. The following diagram demonstrates the successful Saga flow for the booking application:



In the event of a failure, **the microservice reports the failure to SEC, and it is the responsibility of the SEC to invoke the relevant compensation transactions:**



In this example, the Payment microservice reports a failure, and the SEC invokes the compensating transaction to unblock the seat. If the call to the compensating transaction fails, it is the responsibility of the SEC to retry it until it is successfully completed. Recall that in Saga, a compensating transaction must be *idempotent* and *retryable.*

The Choreography pattern is **suitable for greenfield microservice application development**. Also, this pattern is suitable when there is a fewer participant in the transaction.

Following are a few frameworks available to implement the choreography pattern:

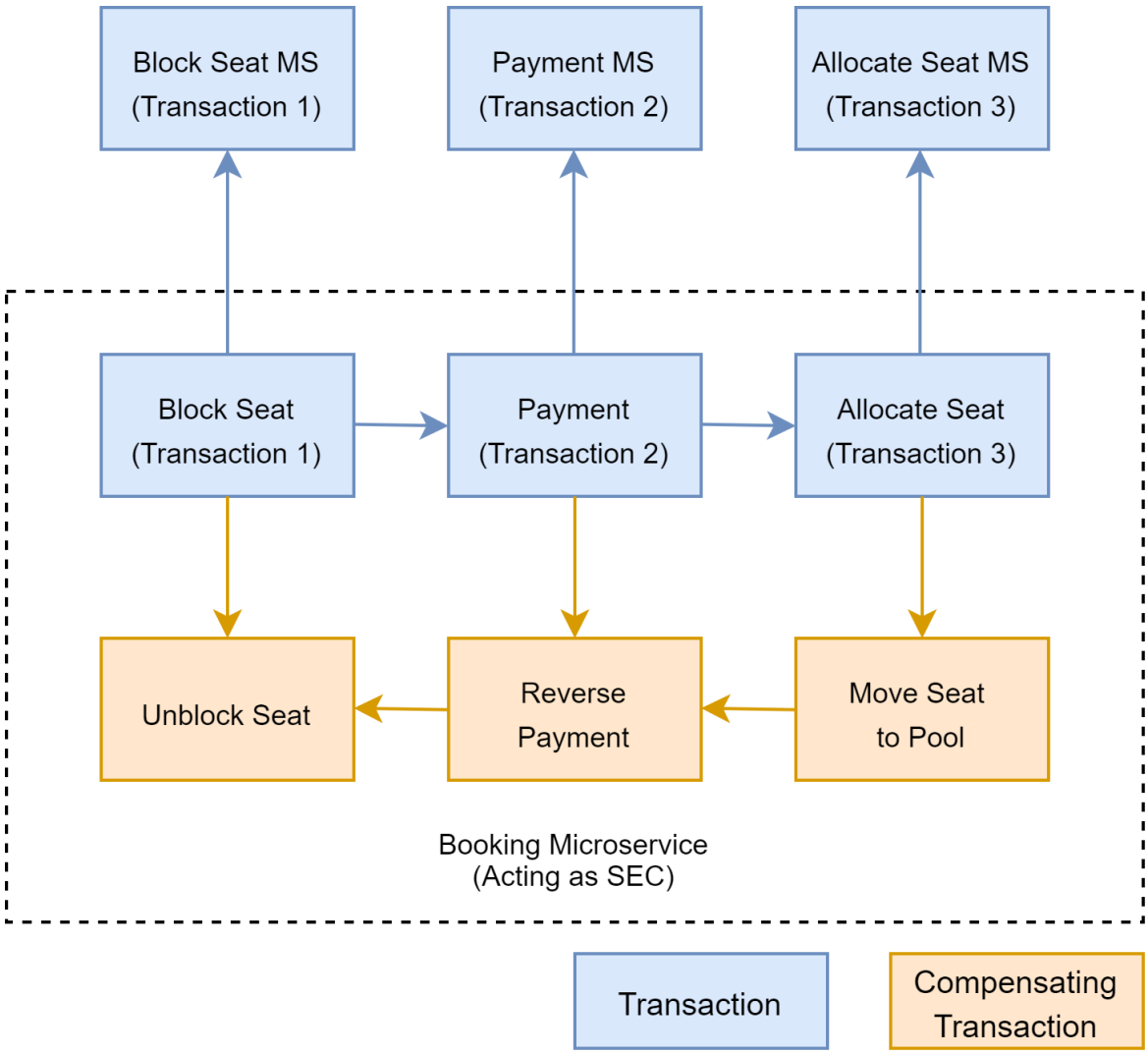
- **Axon Saga (https://docs.axoniq.io/reference-guide/v/3.1/part-ii-domain-logic/sagas)**: A lightweight framework and widely used with Spring Boot based microservices
- **Eclipse MicroProfile LRA (https://github.com/eclipse/microprofile-lra)**: Implementation

of distributed transactions in Saga for HTTP transport based on REST principles

- **Eventuate Tram Saga (https://eventuate.io/docs/manual/eventuate-tram/latest/getting-started-eventuate-tram-sagas.html)**: Saga orchestration framework for Spring Boot and Micronaut based microservices
- **Seata (http://seata.io/en-us/docs/dev/mode/saga-mode.html)**: Open source distributed transaction framework with high-performance and easy-to-use distributed transaction services

7.4. Implementing Saga Orchestration Pattern

In the Orchestration pattern, **a single orchestrator is responsible for managing the overall transaction status**. If any of the microservice encounters a failure, then the orchestrator is responsible for invoking the necessary compensating transactions:



The Saga orchestration pattern is **useful for brownfield microservice application development architecture**. In other words, this pattern is suitable if we already have a set of microservices and would like to implement the Saga pattern in the application. We need to define the appropriate compensating transactions to proceed with this pattern.

Following are a few frameworks available to implement the orchestrator pattern:

- **Camunda (https://camunda.com/)**: This is a Java-based framework that supports Business Process Model and Notation (BPMN) standard for workflow and process automation.
- **Apache Camel (https://camel.apache.org/components/latest/eips/saga-eip.html)**: Provides the implementation for Saga Enterprise Integration Pattern (EIP)

8. Conclusion

In this article, we discussed the Saga architecture pattern to implement distributed transactions in a microservice-based application.

We introduced the challenges of implementing a distributed transactions in a microservice-based application. We then explore the two-phase commit protocol, a popular alternative of Saga, and examined its limitation to implement distributed transactions in microservice-based applications.

Lastly, we discussed the Saga architecture pattern, how it works, and the two main approaches to implement the Saga pattern in microservice-based applications.

Comments are closed on this article!

CATEGORIES