Saga Pattern in Microservices

FEATURED VIDEOS

Last updated: June 15, 2023

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

Reviewed by: Michal Aibin
(https://www.baeldung.com/cs/editor/michal-author)

**Software Architecture
(https://www.baeldung.com/cs/category/software-architecture)**

Design Patterns (https://www.baeldung.com/cs/tag/design-patterns)

Microservices (https://www.baeldung.com/cs/tag/microservices)

1. Overview

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 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.

Further reading:

Microservices and Cross-Cutting Concerns (/cs/?post_type=post&p=53628)

Learn about common cross-cutting concerns in a microservices architecture.

Read more (/cs/?post_type=post&p=53628) →

Service Discovery in Microservices (/cs/?post_type=post&p=55784)

An introduction to Service Discovery in the context of microservices.

Read more (/cs/?post_type=post&p=55784) →

Explaining the Context Design Pattern (/cs/?post_type=post&p=64389)

Explore several patterns that use the idea of context.

Read more (/cs/?post_type=post&p=64389) →

2. Database per Service Pattern

One of the benefits of microservice architecture is that we can choose the technology stack per service.

For instance, we can decide to use a relational database for service A and a NoSQL database for service B.

This model lets the service manage domain data independently on a data store that best suites its data types and schema. Further, it also lets the service scale its data stores 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, we'll look closer at 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 an e-commerce application that processes online orders and is implemented with microservice architecture.
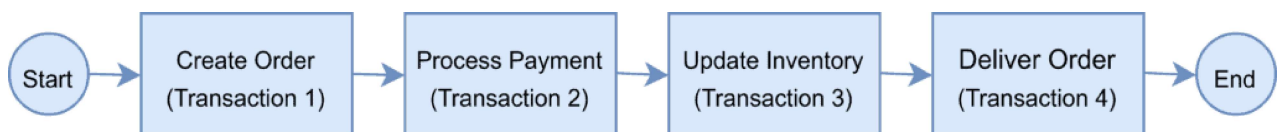
There is a microservice to create the orders, one that processes the payment, another that updates the inventory and the last one that delivers the order.

Each of these microservices performs a local transaction to implement the individual functionalities:

This is an example of a distributed transaction as the transaction boundary crosses multiple services and databases.

To ensure a successful order processing service, all four microservices must complete the individual local transactions. If any of the microservices fail to complete its local transaction, all the completed preceding transactions should roll back to ensure data integrity.

4. Challenges of Distributed Transaction

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

The first challenge 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. *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 several services, ensuring ACID always remains key.**

The second challenge 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 microservices 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) is **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 run their local transactions:
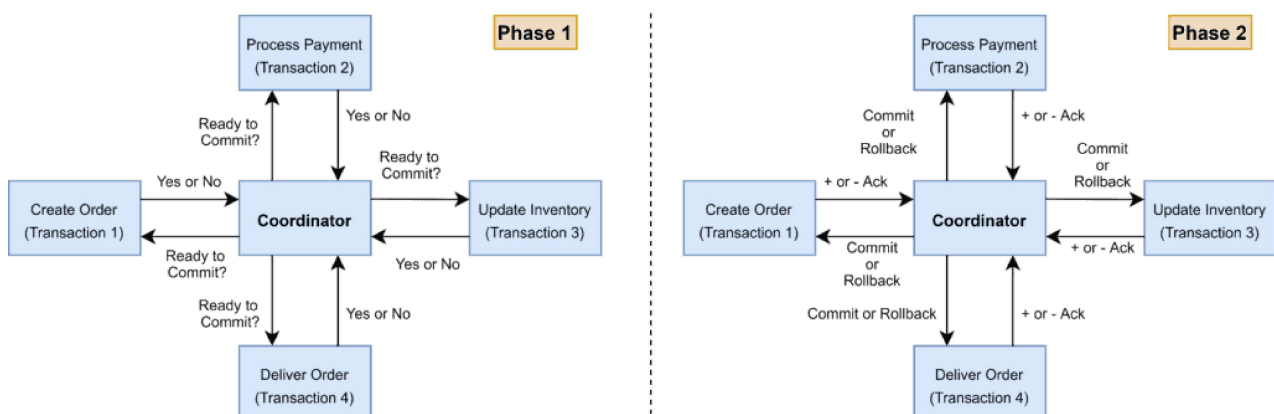
As the name indicates, the two-phase commit protocol runs a distributed transaction in two phases:

1. **Prepare Phase** (/tb4) coordinat(https://www.participating nodes/)
   whether they are ready to commit the transaction. The participants
   returned with a *yes* or *no*.

   (/bael-search)

2. **Commit Phase** – If all the participating nodes respond affirmatively in
   phase 1, 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 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. So, 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.** So, it can lead to
  scalability and performance issues in a microservice-based
  architecture involving multiple services.
- Two-phase commit protocol is **not supported in NoSQL databases.**
  Therefore, in a microservice architecture where one or more services
  use NoSQL databases, we can't apply a two-phase commit.

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 we can manage transactions without any manual intervention.

The Saga Execution Coordinator (SEC) guarantees these principles:
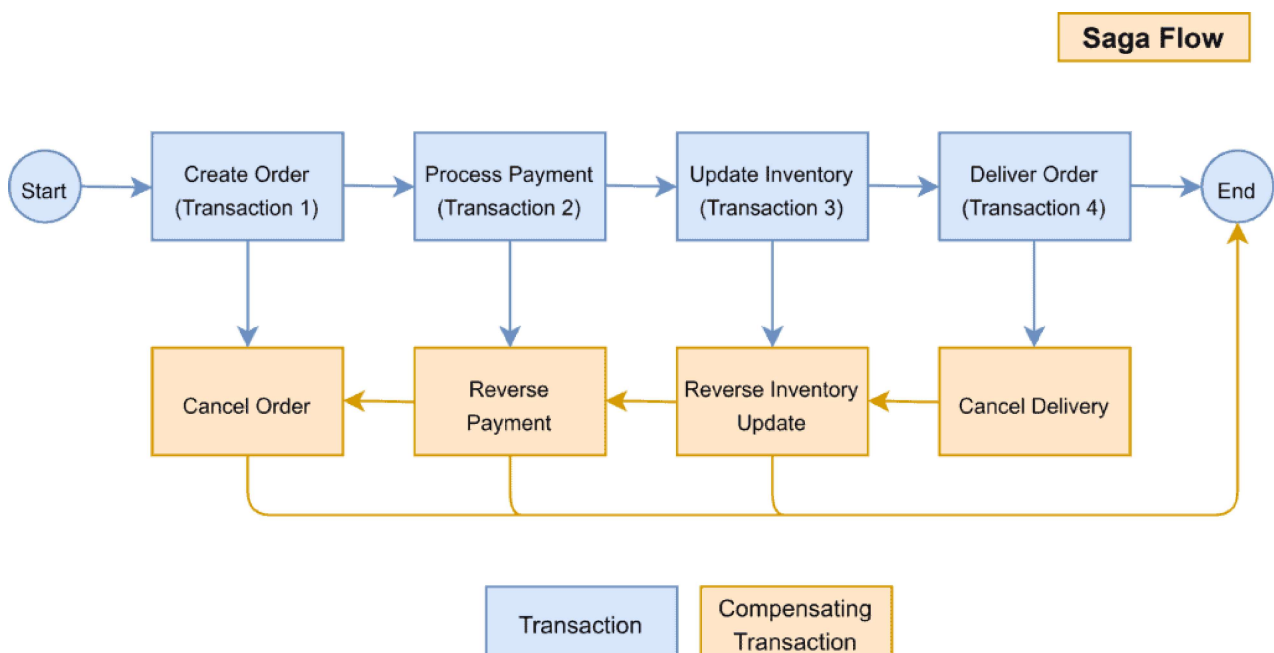
The above diagram shows how to visualize the Saga pattern for the online order processing scenario we discussed earlier.

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. (/bael-search)

For any failure, the SEC component inspects the Saga log to identify the impacted components and the sequence in which the compensating transactions should run.

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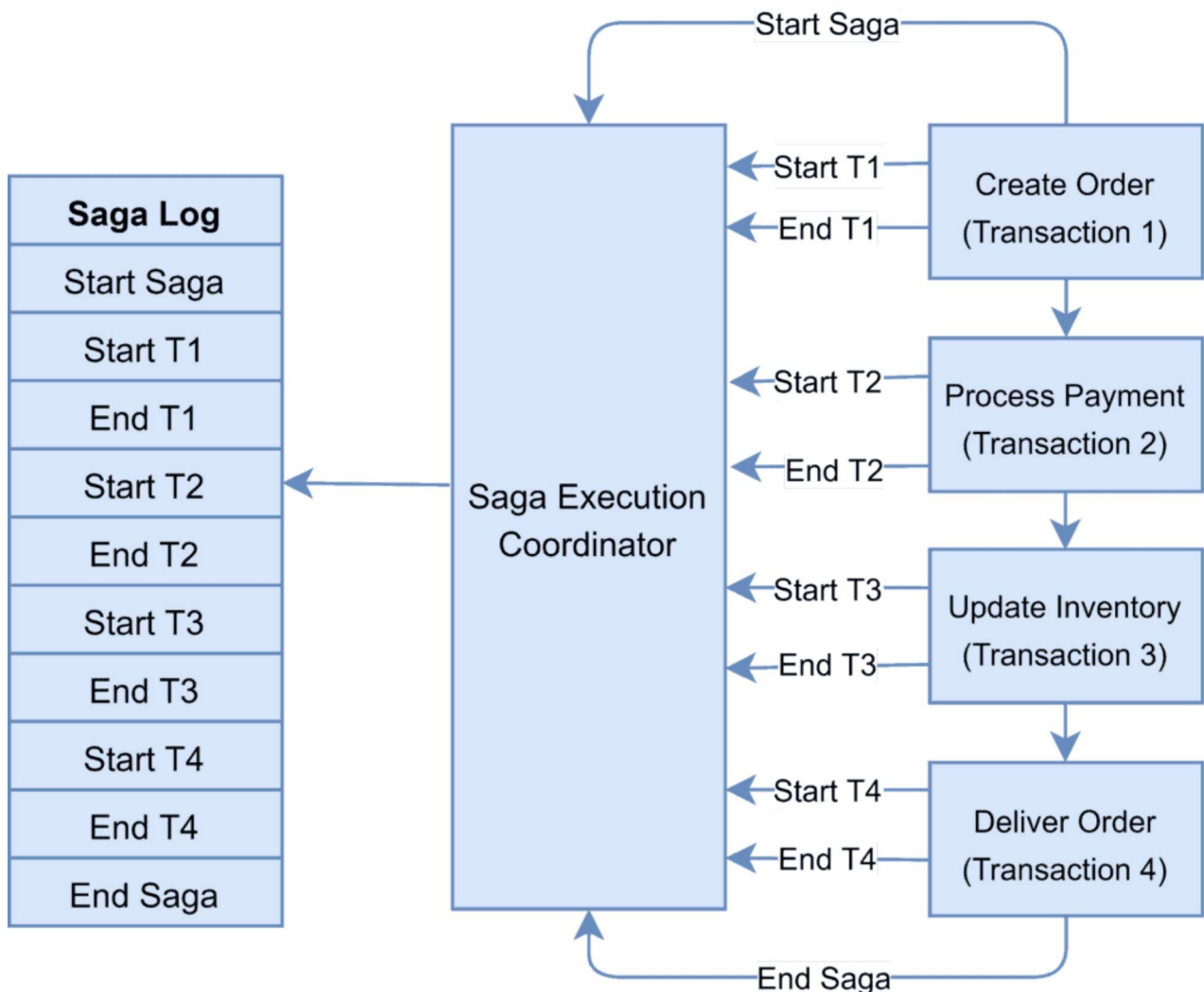
There are two approaches to implement the Saga pattern: choreography and orchestration. Let's discuss them in the next sections.

7.2. Implementing Saga Choreography Pattern

In the Saga Choreography Pattern, **each microservice that is part of the transaction publishes an event that is processed by the next microservice.**
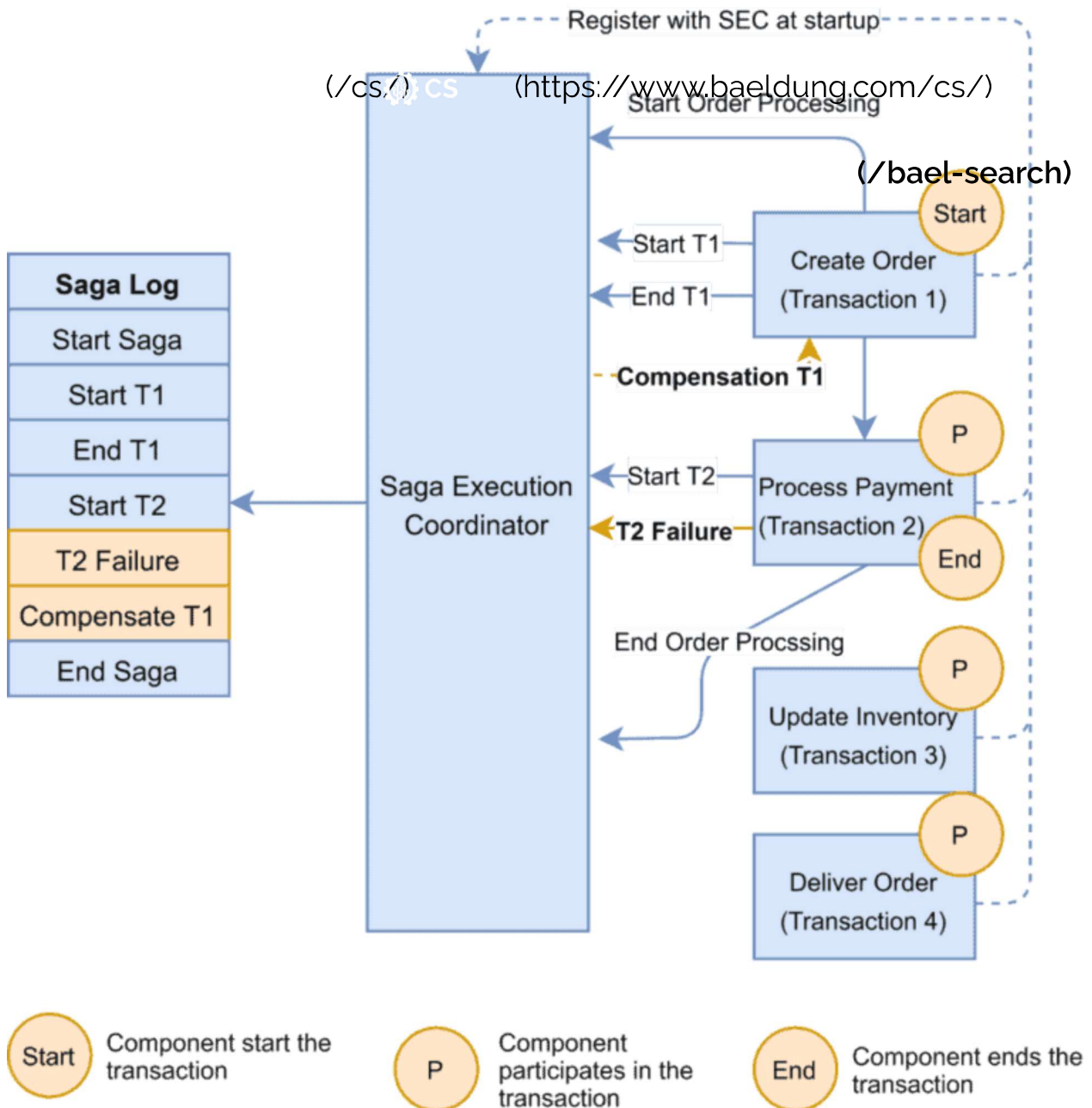
To use this pattern, we need to decide if the microservice will be part of the Saga. Accordingly, the microservice needs to use the appropriate framework to implement Saga. In this pattern, the Saga Execution Coordinator is either embedded within the microservice or can be a standalone component.

In the Saga, choreography flow is successful if all the microservices complete their local transaction, and none of the microservices reported any failure.

The following diagram demonstrates the successful Saga flow for the online order processing application:
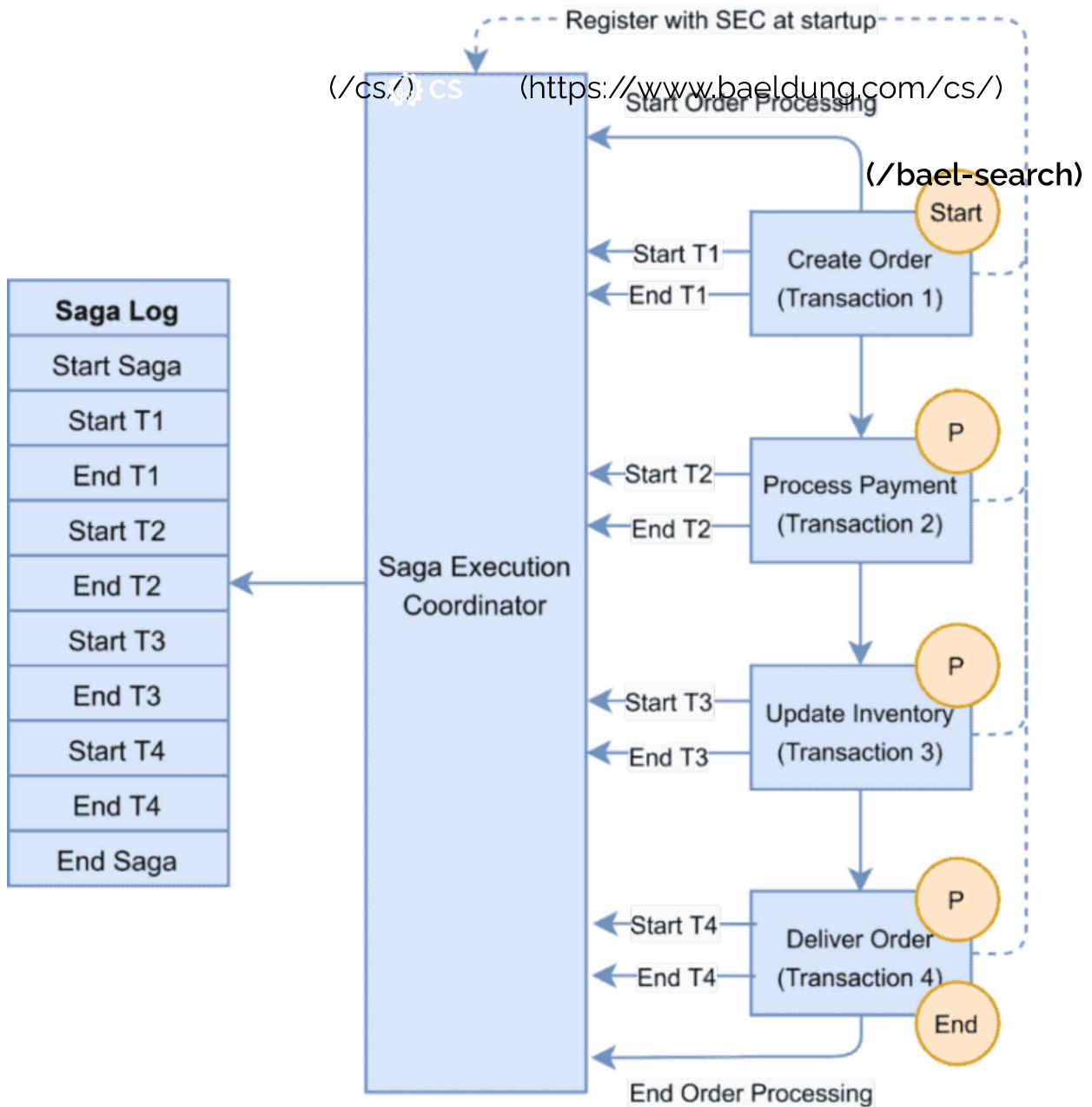
In the event of a failure, **the microservice reports the failure to SEC, and it is the SEC's responsibility to invoke the relevant compensation transactions**:

Saga Log

Start Saga
Start T1
End T1
Start T2
T2 Failure
Compensate T1
End Saga

Saga Execution Coordinator

Start T1 — Create Order (Transaction 1)
End T1

Compensation T1

Start T2 — Process Payment (Transaction 2)
T2 Failure

End Order Procssing

Update Inventory (Transaction 3)

Deliver Order (Transaction 4)

Start	Component start the transaction	
P	Component participates in the transaction	
End	Component ends the transaction	

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 SEC's responsibility to retry it until it is successfully completed. Recall that in Saga, a compensating transaction must be *idempotent* and *retryable*.

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

Here 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 (https://www.seata.io/docs/dev/mode/saga-mode/)** – 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 microservices encounter a failure, the orchestrator is responsible for invoking the necessary compensating transactions:

Order Processing Microservice
(Acting as SEC)

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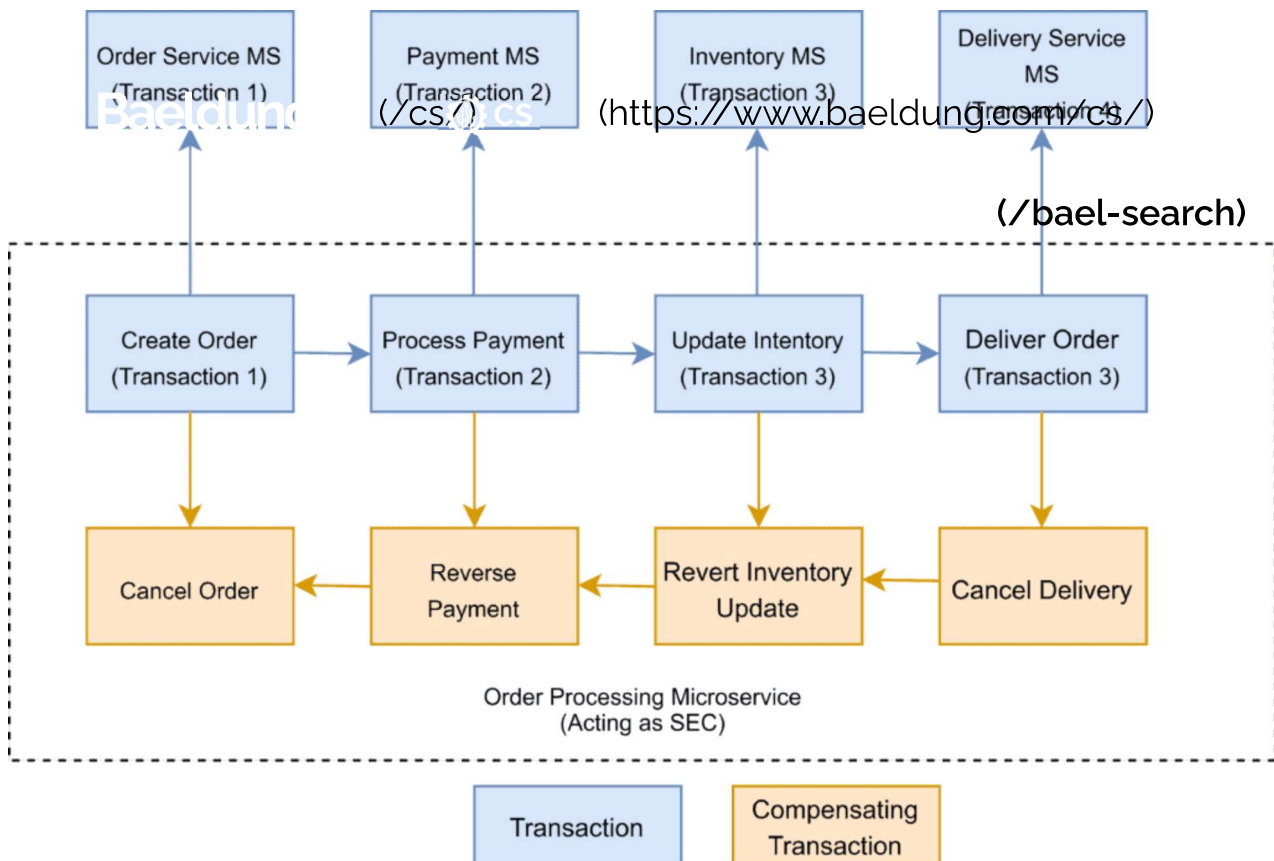
The Saga orchestration pattern is **useful for brownfield microservice application development architecture.** In other words, this pattern works when we already have a set of microservices and would like to implement the Saga pattern in the application. We need to define the appropriate compensating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

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

- **Camunda (https://camunda.com/)** 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 first introduced the challenges of these implementations.

We then explored 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 implementing the Saga pattern in microservice-based applications.

Comments are open for 30 days after publishing a post. For any issues past this date, use the Contact form on the site.

CATEGORIES

SERIES

GRAPHS (HTTPS://WWW.BAELDUNG.COM/CS/GRAPHS-SERIES)

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

ABOUT

(/bael-search)