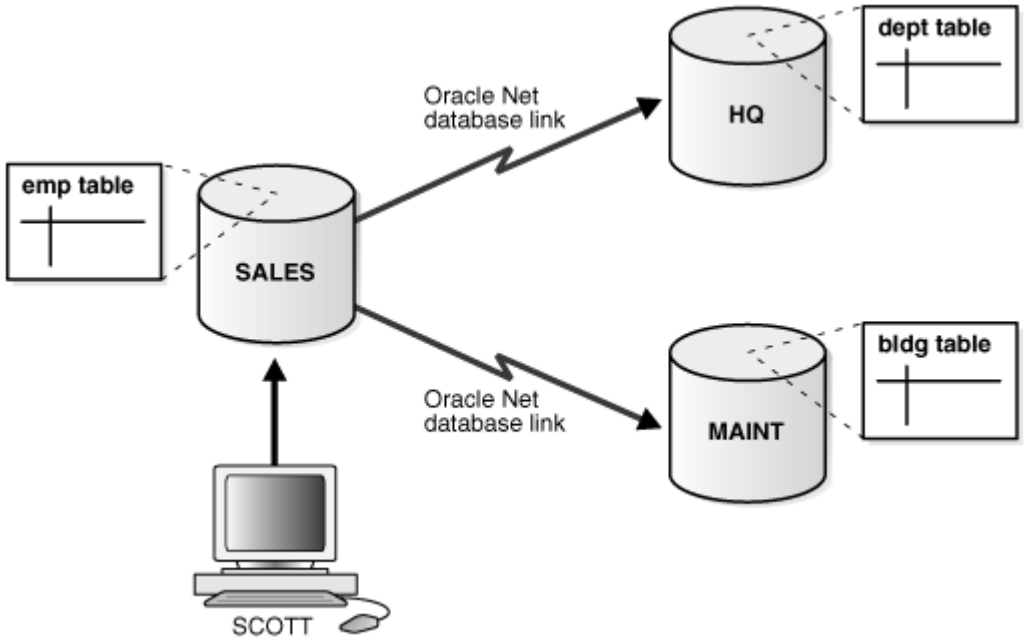
Transactions in distributed systems



What are Transactions?

A **transaction** symbolizes a unit of work performed within a database management system (or similar system) against a database and treated in a coherent and reliable way independent of other transactions. A transaction generally represents any change in a database. Transactions in a database environment have two main purposes:

1. To provide reliable units of work that allow correct recovery from failures and keep a database consistent even in cases of system failure. For example, when execution prematurely and unexpectedly stops (completely or partially), many operations upon a database remain uncompleted, with unclear status.

2. To provide isolation between programs accessing a database concurrently. If this isolation is not provided, the programs' outcomes are possibly erroneous.

When breaking apart our databases, we've already touched on some of the problems that can result. Maintaining referential integrity becomes problematic, latency can increase, and we can make activities like reporting more complex. We've looked at various coping patterns for some of these challenges, but one big one remains: what about transactions? Making changes to our database in a transaction can make our systems much easier to reason about, and therefore easier to develop and maintain. We rely on our

database to ensure the safety and consistency of our data, leaving us to worry about other things. But when we split data across databases, we lose the benefit of using a database transaction to apply changes in state in an atomic fashion. Before we explore how to tackle this issue, let's look briefly at what a normal database transaction gives us.

ACID Transactions

Typically, when we talk about database transactions, we are talking about ACID transactions. ACID is an acronym outlining the key properties of database transactions that lead to a system we can rely on to ensure the durability and consistency of our data storage. and here is what these properties give us:

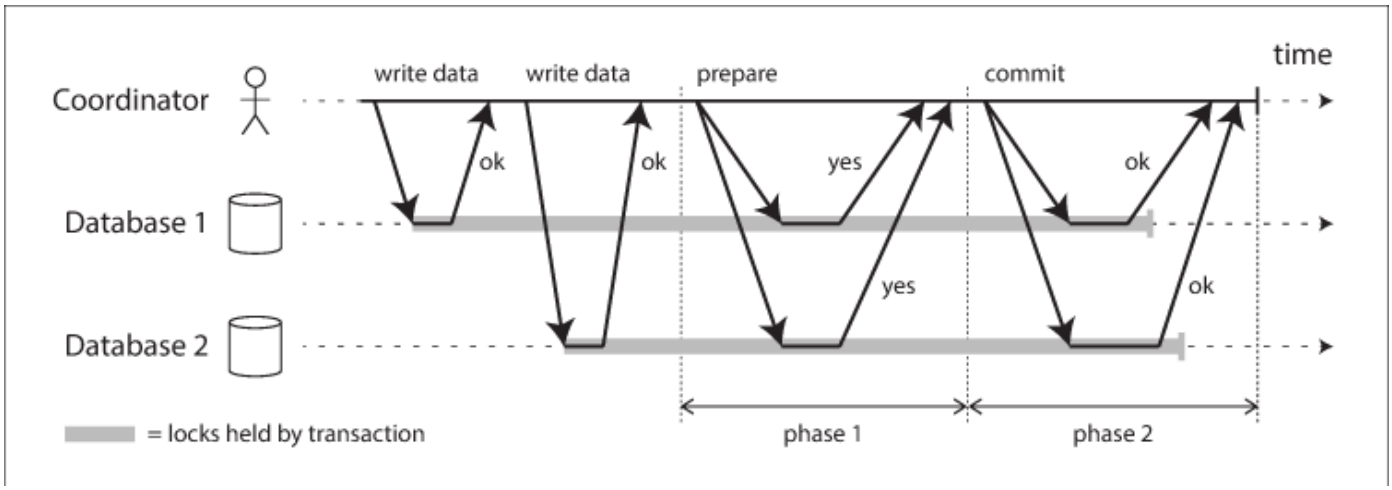
- **Atomicity**: Ensures that all operations completed within the transaction either all complete or all fail. If any of the changes we're trying to make fail for some reason, then the whole operation is aborted, and it's as though no changes were ever made.
- **Consistency**: When changes are made to our database, we ensure it is left in a valid, consistent state.
- **Isolation**: Allows multiple transactions to operate at the same time without interfering.
  This is achieved by ensuring that any interim state changes made during one transaction are invisible to other transactions.
- **Durability:** Makes sure that once a transaction has been completed, we are confident the data won't get lost in the event of some system failure.

I want to be clear that we can still use ACID-style transactions when we split databases apart, but the scope of these transactions is reduced, as is their usefulness. With a single database, this is done in the scope of a single ACID database transaction either the new rows are written, or neither is written, We could exactly do the same thing in a distributed database, but now each
change is made in a different database. This means there are two transactions to consider, each of which could work or fail independently of the other

This lack of atomicity can start to cause significant problems, especially if we are migrating systems that previously relied on this property. It's at this point that people start to look for other solutions to give them some ability to reason about changes being made to multiple

services at once. Normally, the first option that people start considering is distributed transactions. Let's look at one of the most common algorithms for implementing distributed transactions, the two-phase commit, as a way of exploring the challenges associated with distributed transactions as a whole.

Two-Phase Commits



The two-phase commit algorithm (sometimes shortened to 2PC) is frequently used to attempt to give us the ability to make transactional changes in a distributed system, where multiple separate processes may need to be updated as part of the overall operation. I want to let you know upfront that 2PCs have limitations, which we'll cover, but they're worth knowing about. Distributed transactions, and two-phased commits more specifically, are frequently raised by teams moving to microservice architectures
as a way of solving challenges they face. But as we'll see, they may not solve your problems and may bring even more confusion to your system.
The algorithm is broken into two phases (hence the name two-phase commit): a voting phase and a commit phase. During the voting phase, a central coordinator contacts all the workers who are going to be part of the transaction, and asks for confirmation as to whether or not some state change can be made.

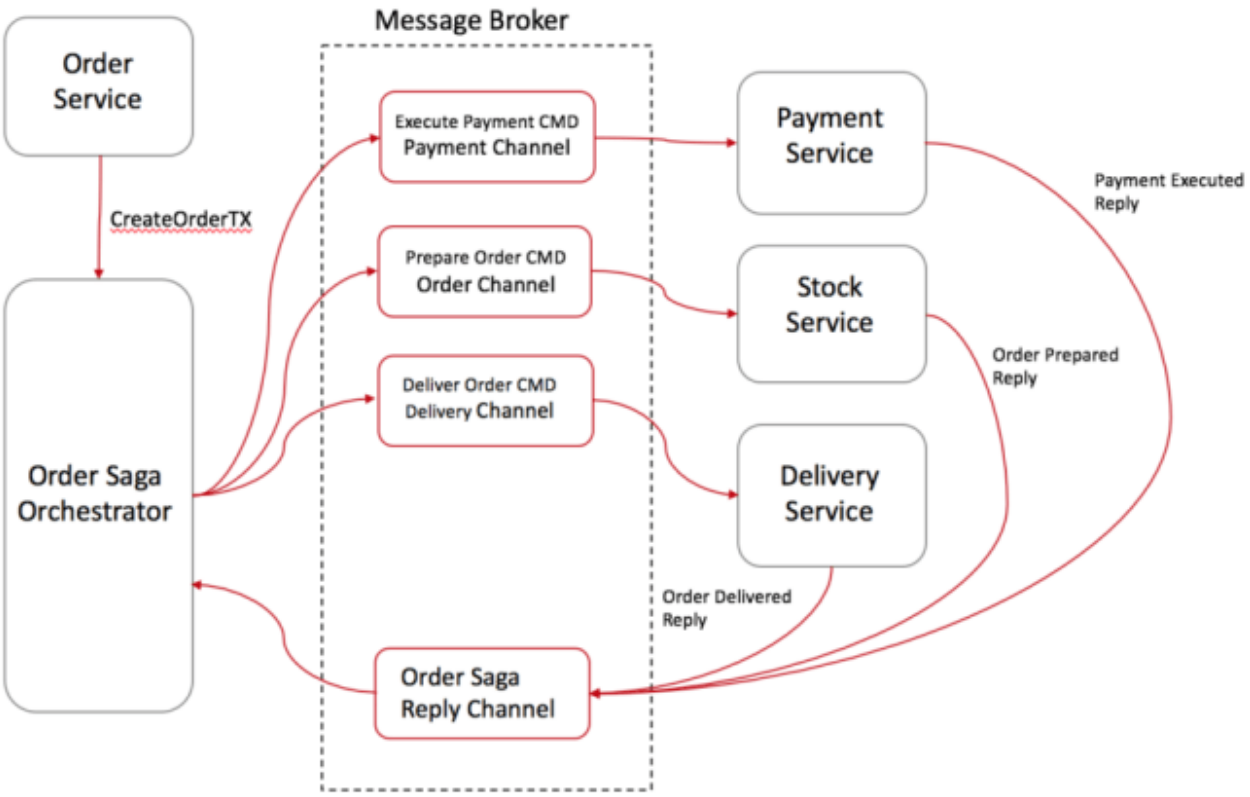
When two-phase commits work, at their heart they are very often just coordinating distributed locks. The workers need to lock local resources to ensure that the commit can take place during the second phase. Managing locks, and avoiding deadlocks in a single-process system, isn't fun. Now imagine the challenges of coordinating locks among multiple participants. It's not pretty. There are a host of failure modes associated with two-phase

commits that we don't have time to explore. Consider the problem of a worker voting to proceed with the transaction, but then not responding when asked to commit. What should we do then? Some of these failure modes can be handled automatically, but some can leave the system in such a state that things need to be manually unpicked. The more participants you have, and the more latency you have in the system, the more issues a two-phase commit will have. They can be a quick way to inject huge amounts of latency into your system, especially if the scope of locking is large, or the duration of the transaction is large. It's for this reason two-phase commits are typically used only for very short-lived operations. The longer the operation takes, the
longer you've got resources locked for!

Why using Distributed Transactions is not a good decision?

For all these reasons outlined so far, I strongly suggest you avoid the use of distributed transactions like the two-phase commit to coordinate changes in the state across your Distributed system. So what else can you do?
Well, the first option could be to just not split the data apart in the first place. If you have pieces of state that you want to manage in a truly atomic and consistent way, and you cannot work out how to sensibly get these characteristics without an ACID-style transaction, then leave that state in a single database, and leave the functionality that manages that state in a single service (or in your monolith). If you're in the process of working out where to split your monolith and working out what decompositions might be easy (or hard), then you could well decide that splitting apart data that is currently managed in a transaction is just too hard to handle right now. Work on some other area of the system, and come back to this later. But what happens if you really do need to break this data apart, but you don't want all the pain of managing distributed transactions? How can we carry out operations in multiple services but avoid locking? What if the operation is going to take minutes, days, or perhaps even months? In cases like this, we can consider an alternative approach: sagas.

Sagas



Unlike a two-phase commit, a saga is by design an algorithm that can coordinate multiple changes in state, but avoids the need for locking resources for long periods of time. We do this by modeling the steps involved as discrete activities that can be executed independently. It comes with the added benefit of forcing us to explicitly model our business processes, which can have significant benefits. The core idea, first outlined by Hector Garcia-Molina and Kenneth Salem, 10 reflected on the challenges of how best to handle operations of what they referred to as long-lived transactions (LLT). These transactions might take a long time (minutes, hours, or perhaps even days), and as part of that process require changes to be made to a database.

If you directly mapped an LLT to a normal database transaction, a single database transaction would span its entire life cycle of it. This could result in multiple rows or even full tables being locked for long periods of time while the LLT is taking place, causing significant issues if other processes are trying to read or modify these locked resources. Instead, the authors of the paper suggest we should break down these LLTs into a sequence of transactions, each of which can be handled independently. The idea is that the duration of each of these “sub” transactions will be shorter-lived, and will modify only part of the data affected by the entire LLT. As a result, there will be far less contention in the underlying database as the scope and duration of locks is

greatly reduced. While sagas were originally envisaged as a mechanism to help with LLTs acting against a single database, the model works just as well for coordinating change across multiple services. We can break a single business process into a set of calls that will be made to collaborating services as part of a single saga.

Saga Failure Modes

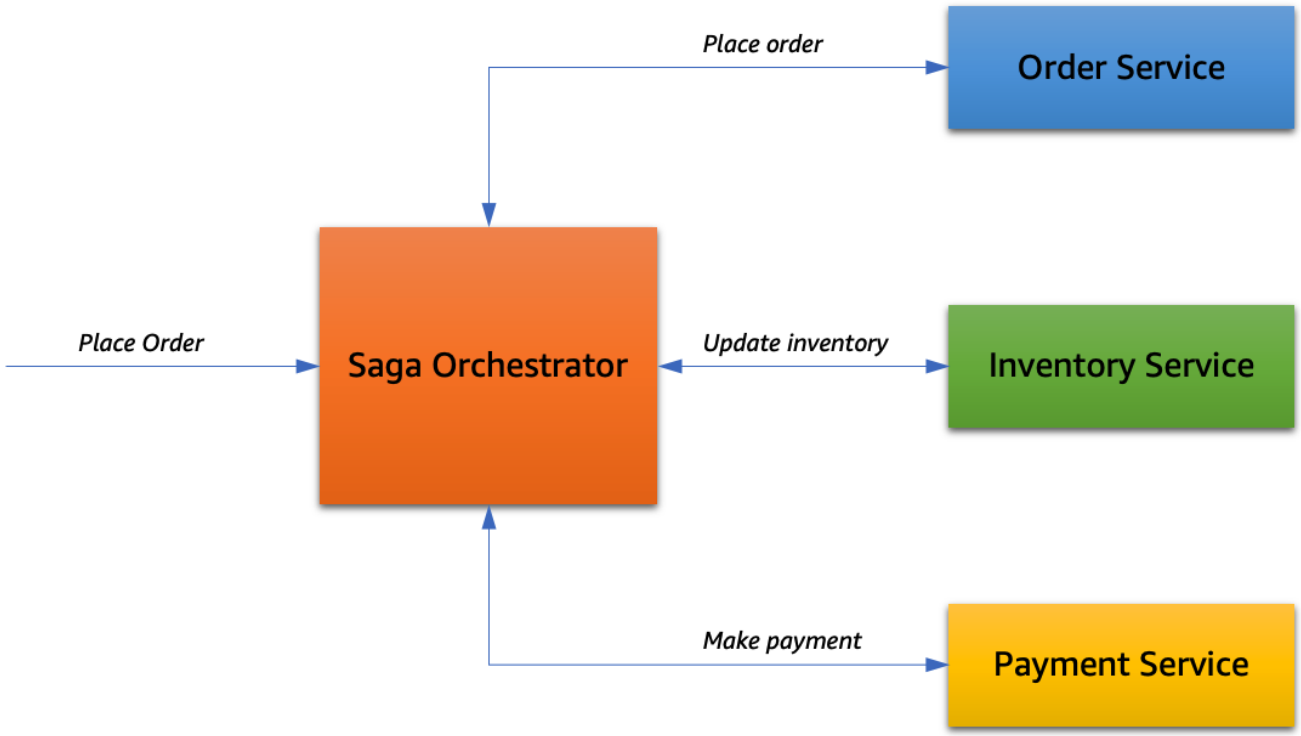
With a saga being broken into individual transactions, we need to consider how to handle failure, more specifically, how to recover when a failure happens. The original saga paper describes two types of recovery: backward recovery and forward recovery. Backward recovery involves reverting the failure and cleaning up afterward a rollback. For this to work, we need to define compensating actions that allow us to undo previously committed transactions. Forward recovery allows us to pick up from the point where the failure occurred, and keep processing. For that to work, we need to be able to retry transactions, which in turn implies that our system is persisting enough information to allow this retry to take place. Depending on the nature of the business process being modeled, you may consider that any failure mode triggers a backward recovery, a forward recovery, or perhaps a mix of the two.

Implementing Sagas

So far, we've looked at the logical model for how sagas work, but we need to go a bit deeper to examine ways of implementing the saga itself. We can look at two styles of saga implementation. Orchestrated sagas more closely follow the original solution space and rely primarily on centralized coordination and tracking. These can be compared to choreographed sagas, which avoid the need for centralized coordination in favor of a more loosely coupled model, but which can make tracking the progress of a saga more complicated.

Orchestrated sagas

Orchestration is a way to centralize the workflow of logic for a business process. It coordinates the workflow by sending commands to the appropriate service and consuming the resulting events. In contrast to choreography, Orchestration tells other services what action/command to perform rather than those services being reactive to other events in the system.



Taking the order fulfillment process shown in the image blow , let's see how this central coordination process would work as a set of collaborating services, Here, our central Order Processor, playing the role of the orchestrator, coordinates our fulfillment process. It knows what services are needed to carry out the operation, and it decides when to make calls to those services. If the calls fail, it can decide what to do as a result. These orchestrated processors tend to make heavy use of request/ response calls between services: the Order Processor sends a request to services (such as a Payment Gateway), and expects a response letting it know if the request was successful and providing the results of the request. Having our business process explicitly modeled inside the Order Processor is extremely beneficial. It allows us to look at one place in our system and understand how this process is supposed to work. That can make the onboarding of new people easier, and help impart a better understanding of the core parts of the system.

Benefits & drawbacks of Orchestration

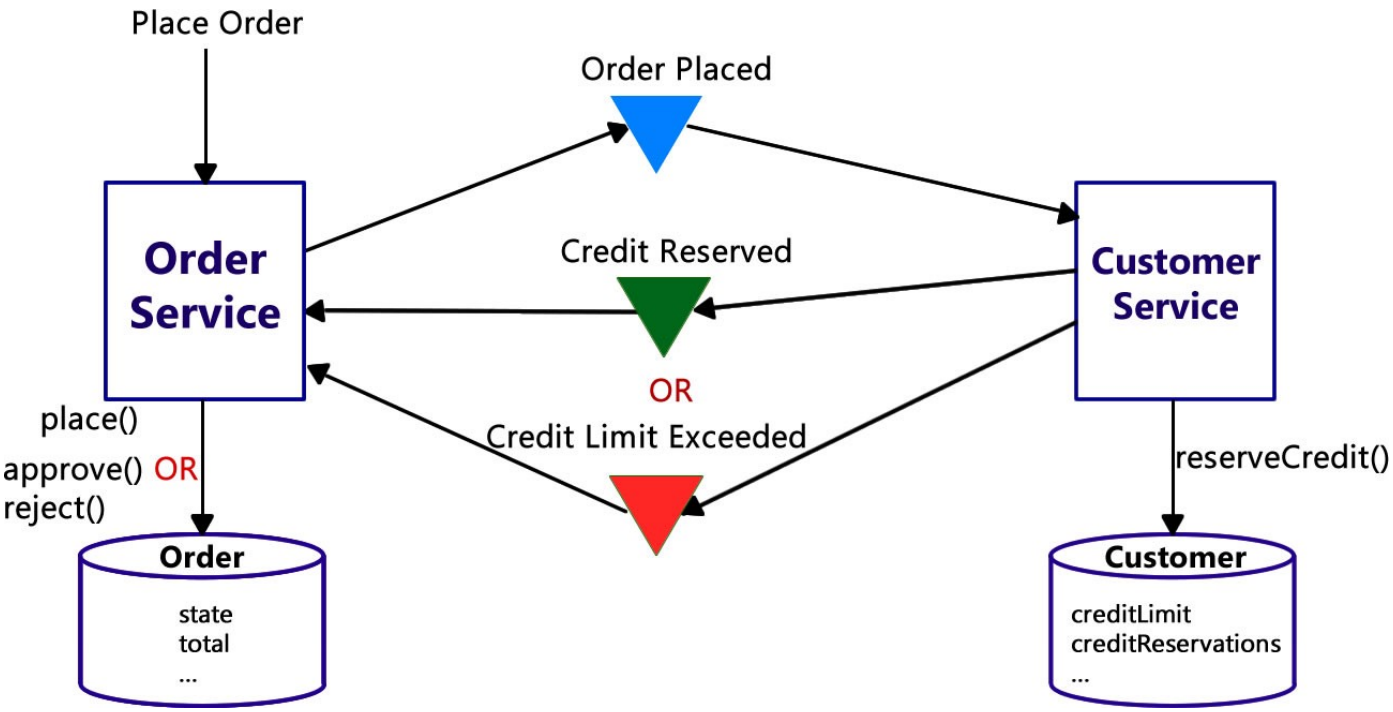
- Centralized logic: this can be good and bad
- Easier to understand the workflow since its defined in a central location
- Full control over the workflow steps via commands
- Point of failure

- Easier to debug and test

Choreographed sagas

Choreographed sagas aim to distribute responsibility for the operation of the saga among multiple collaborating services. If orchestration is command-and-control, choreographed sagas represent a trust-but-verify architecture. As we'll see in our example in the image, choreographed sagas will often make heavy use of events for collaboration between services.

There's quite a bit going on here, so it's worth exploring in more detail. First, these services are reacting to events being received. Conceptually, events are broadcast in the system, and interested parties are able to receive them. You don't send events to a service; you just fire them out, and the services that are interested in these events are able to receive them and act accordingly. In our example, when the Warehouse service receives that first Order Placed event, it knows its job to reserve the appropriate stock and fire an event once that is done. If the stock couldn't be received, the Warehouse would need to raise an appropriate event (an Insufficient Stock event perhaps), which might lead to the order being aborted



Benefits & drawbacks of Choreography

- No centralized logic: this can be good and bad
- Useful for small/simple workflows

- Difficult to conceptualize if a lot of services are involved.
- Hader to debug & test if a lot of services are involved

Choreography or Orchestration?

Implementing choreographed sagas can bring with it ideas that may be unfamiliar to you and your team. They typically assume heavy use of event-driven collaboration, which isn't widely understood. However, in my experience, the extra complexity associated with tracking the progress of a saga is almost always outweighed by the benefits associated with having a more loosely coupled architecture. Stepping aside from my own personal tastes, though, the general advice I give regarding orchestration versus choreography is that I am very relaxed in the use of orchestrated sagas when one team owns the implementation of the entire saga. In such a situation, the more inherently coupled architecture is much easier to manage within the team boundary. If you have multiple teams involved, I greatly prefer the more
decomposed choreographed saga as it is easier to distribute responsibility for implementing the saga to the teams, with the more loosely coupled architecture allowing these teams to work more in isolation.

Sagas Versus Distributed Transactions

As I hope I have broken down by now, distributed transactions come with some significant challenges, and outside of some very specific situations are something I tend to avoid. Pat Helland, a pioneer in distributed systems, distills the fundamental challenges of implementing distributed transactions for the kinds of applications we build today. explicitly modeling business processes as a saga avoids many of the challenges of distributed transactions, while at the same time has the added benefit of making what might otherwise be implicitly modeled processes much more explicit and obvious to your developers. Making the core business processes of your system a first-class concept will have a host of benefits. A fuller discussion of implementing orchestration and choreography, along with the various implementation details.

Conclusion

manage transactions throw distributed systems is a big topic I will put some references to help you improve yourself on this topic.

- Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith chapter 4
- The Saga Pattern in a Reactive Microservices Environment
- Distributed Transactions and Distributed Concurrency Control