


DZone (/) > Microservices Zone (/microservices-news-tutorials-tools) > Distributed Transactions and Saga Patterns

Distributed Transactions and Saga Patterns



(/users/3367019/ayushprashar.html) by

Ayush Prashar (/users/3367019/ayushprashar.html)  MVB ·

Sep. 04, 18 · Microservices Zone (/microservices-news-tutorials-tools) · Tutorial

 Like (12)  Comment (1)  Save  Tweet

In a Knolx session organized by Knoldus, we discussed the idea of following Saga Patterns. For that to be more accessible, I'd like to share the session.

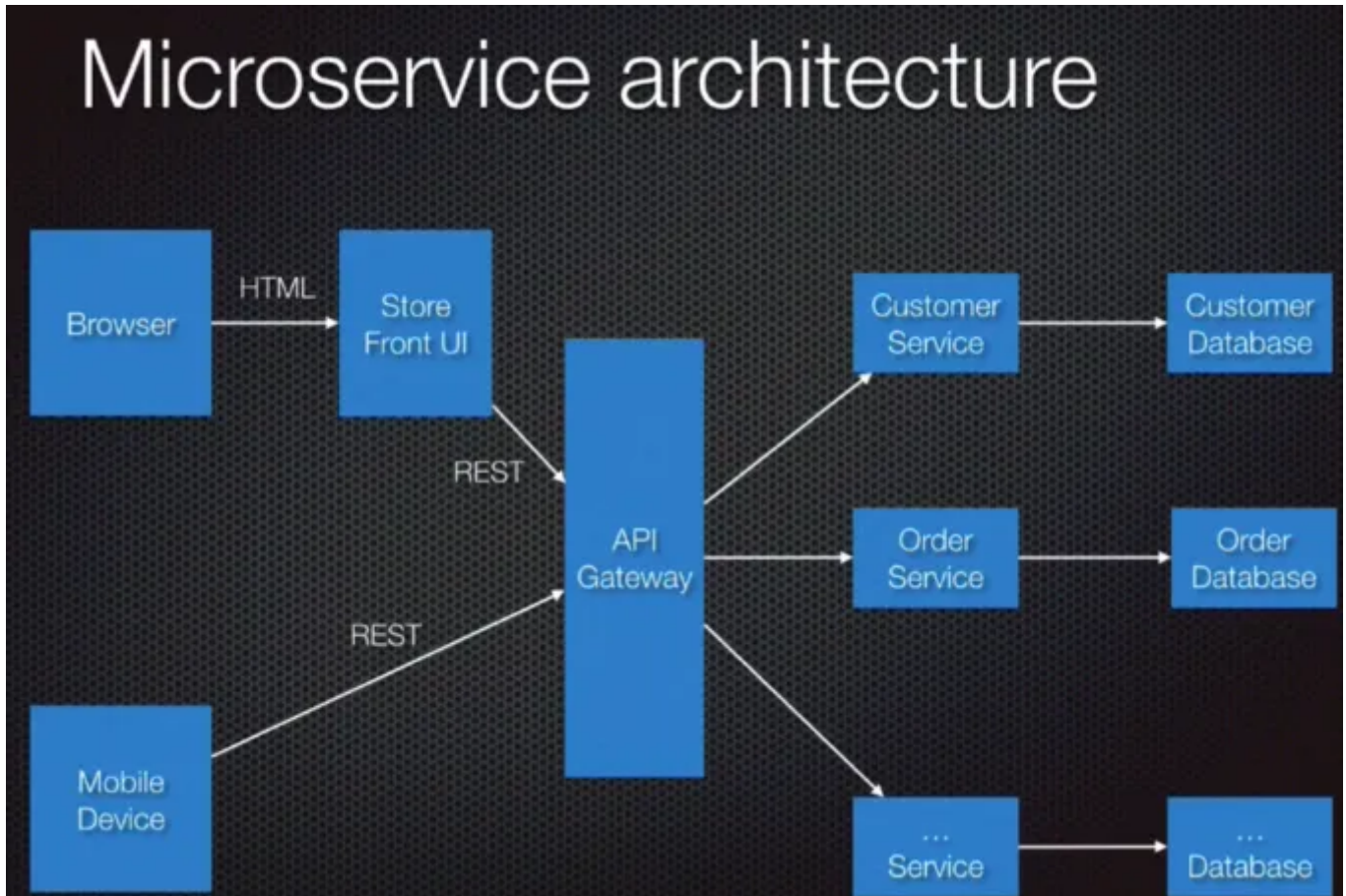
Service-oriented architecture has given us enough advantages to be a predominant architecture in our Industry, but it can't be all sunshine and rainbows. There are use cases where monoliths are not only better but technically the only practically possible option. One such case is the feasibility of **transactions**.

A transaction is a logically atomic unit of work which may span multiple database queries. To say the least about it, they ensure locks over resources (Eg: tables it accesses) they acquire in order to maintain the consistency of the database. All this is done with the help of the famous properties. ACID here stands for:

- **Atomicity:** It means that either a transaction happens in full or doesn't happen at all. At any point, if the transaction feels it can't process, it'll rollback.
- **Consistency:** It means that the state of the database remains consistent before a transaction begins and after the transaction ends.
- **Isolation:** It means that multiple transactions can run in parallel without disrupting the consistency of the database.
- **Durability:** It means that any changes made in the database actually persist.

Apart from the ACID properties, the ability to make transactions serializable increases the throughput.

But all this is guaranteed in a single database. What happens when we start using microservices where each service houses a separate database? Do we have any means of performing anything similar to a **distributed transaction**? Let's find out.



Distributed Transactions

A distributed transaction would do what a transaction would do but on multiple databases. Think about an e-commerce website. In a monolith design, it would comprise of tables such as **CUSTOMER_DETAILS**, **INVENTORY**, **PRODUCT_DETAILS**, **ORDER_DETAILS**, **PAYMENT_DETAILS**, and so on. Thus in order to buy a product, we would select a customer from **CUSTOMER_DETAILS**, a product from **PRODUCT_DETAILS**, check it's status in the **INVENTORY** and make changes in the table if feasible. After which we will add the order in **ORDER_DETAILS** and record payments in **PAYMENT_DETAILS**.

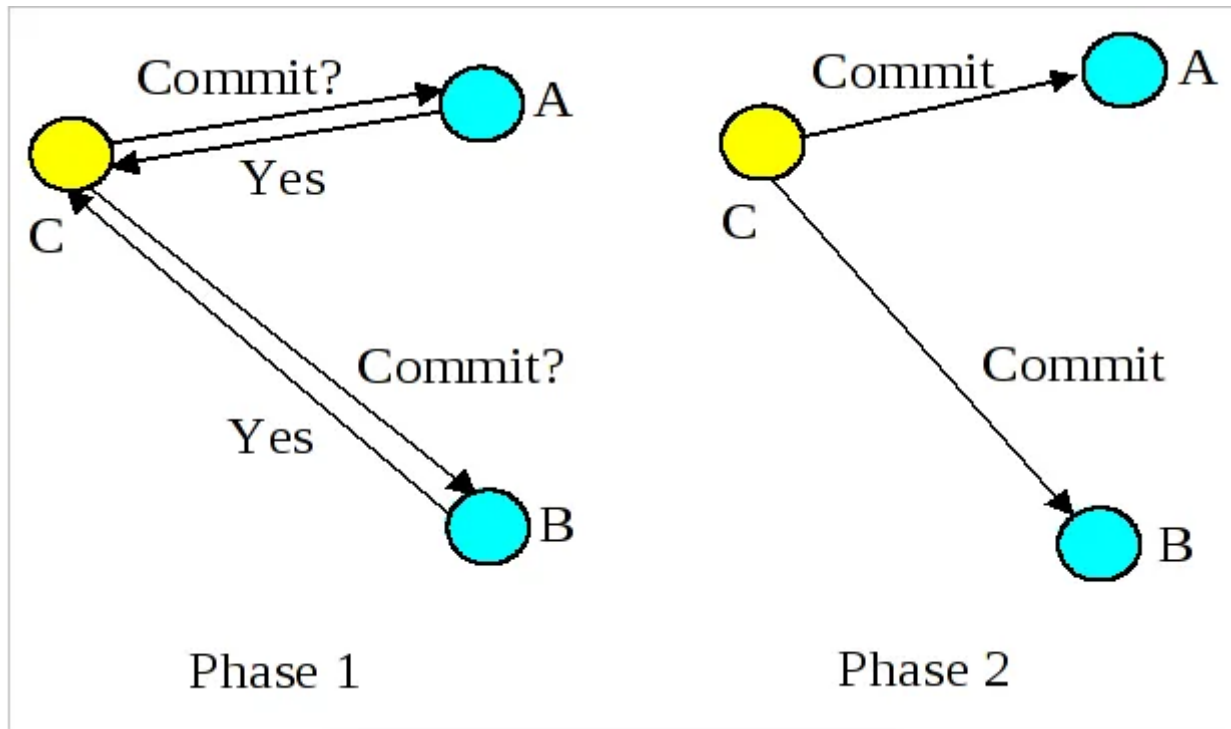
In a distributed scenario the architecture would be split into services like user handling service, payment gateway service etc who would house a respective database, and all the aforementioned actions would be performed in the respective databases.

One way of doing it is the **Two-Phase Commit**. In a two-phase commit, we have a

controlling node which houses most of the logic, and we have a few participating nodes on which the actions would be performed. It works in two phases. (/search)

REFCARDZ (/refcardz) RESEARCH (/research) WEBINARS (/webinars) ZONES (/zones)

1. *Prepare Phase*: In this phase, the controlling node would ask all the participating nodes if they are ready to commit. The participating nodes would then respond in yes or no.
2. *Commit Phase*: Then, if *all* the nodes had replied in affirmative, the controlling node would ask them to commit, else even if one node replied in negative, it'll ask them to roll back.



This solves our problem to a decent extent but as you can see, it brings in some faults as well:

1. Whole logic gets concentrated in a single node, and in case that node goes down, the whole system fails.
2. The whole system is bound by the slowest resource since any ready node will have to wait for confirmation from a slower node which is yet to confirm its status.

Thus, even though we can kind of implement it, it's not really a feasible option. It's clear from the two-phase commit that atomicity can hamper our performance. Therefore, we make use of a design pattern which trades this atomicity with resource availability. Something like this was published in a paper titled SAGAS from 1987. Which brings us to the **Saga pattern**.

Saga pattern is one of the ways by which we ensure data consistency in a distributed architecture but it doesn't really qualify as a standard transaction. A major reason for



DZone

that being absence of atomicity. So in order to implement this pattern, we perform the constituent operations one after another. For example, in the e-commerce website, we talked about, we first do the transaction pertaining to the selection of customer. Once that is complete, we start selecting a product and so on. So all these constituent transactions together will be known as a Saga.

RESEARCH (/research/)

RESEARCH (/research/)

WEBINARS (/webinars/)

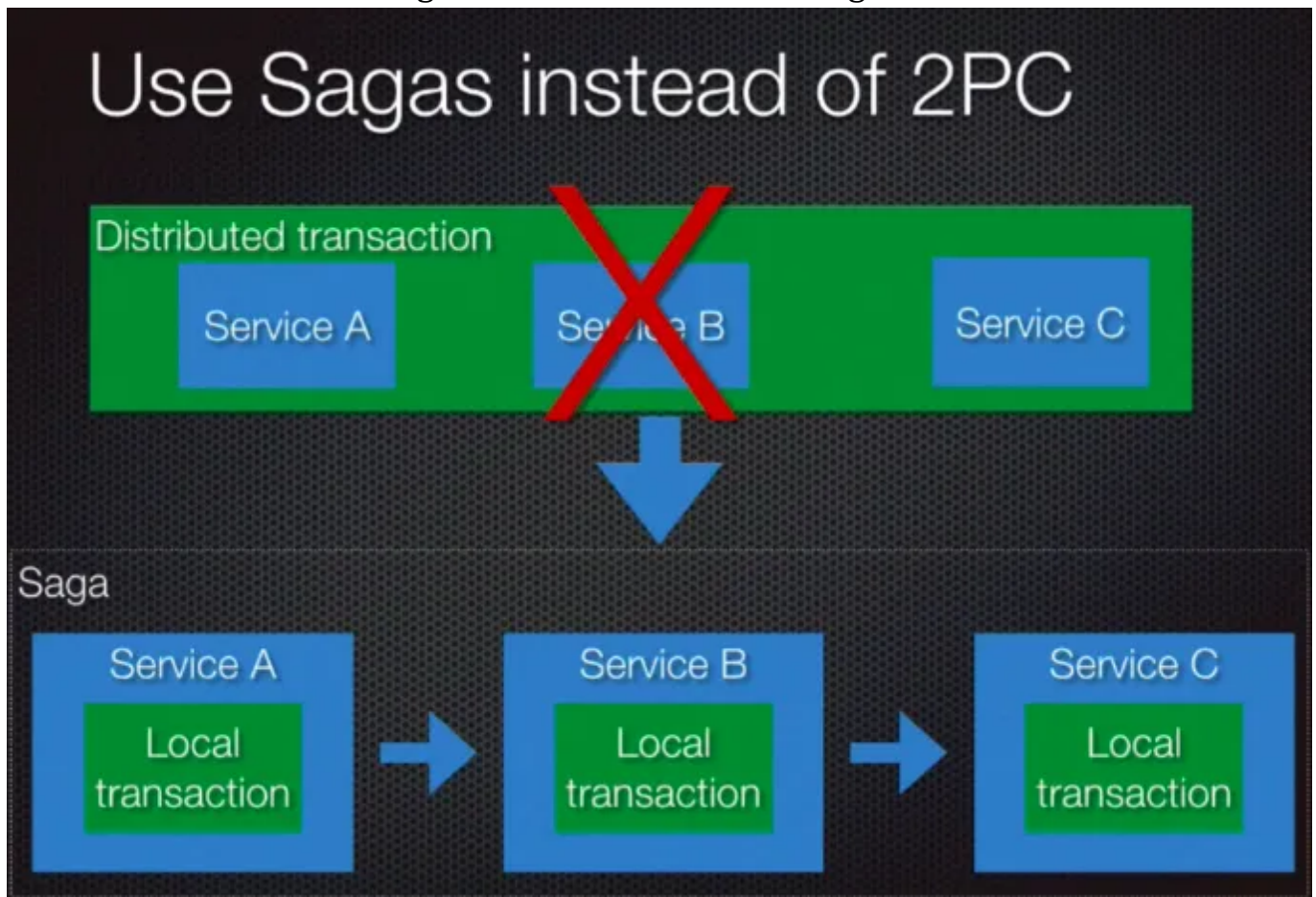
ZONES



(/users/login.html)



(/search/)



A successful saga looks something like this :

1. Start Saga
2. Start T1
3. End T1
4. Start T2
5. End T2
6. Start T3
7. End T3
8. End Saga

But things rarely go as straight. Sometimes, we might not be in a position to perform a transaction in the middle of the saga. At that point, the previously successful transactions would've already committed. So apart from not continuing with the saga, we also need to undo whatever changes we may have already committed. For

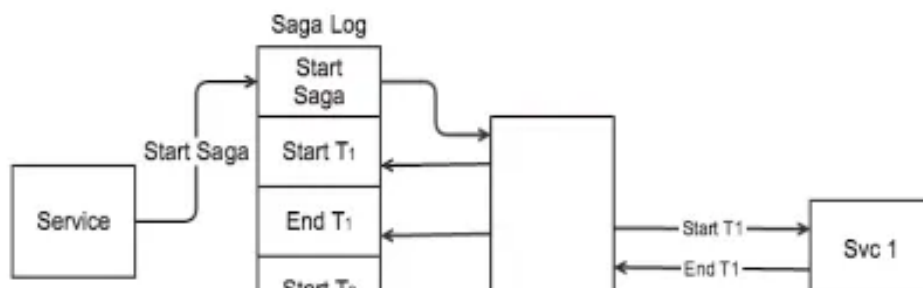
saga, we also need to undo whatever changes we may have already committed. For this, we apply **compensatory transactions**. Thus, for each transaction T_i , we implement a compensatory transaction C_i , which tries to semantically nullify T_i . It's not always possible to get back to the exact same state. For example, if T_i involves sending out an email, we can't really undo that. So we send a corrective email which semantically undoes T_i . So a failed saga looks something like this:

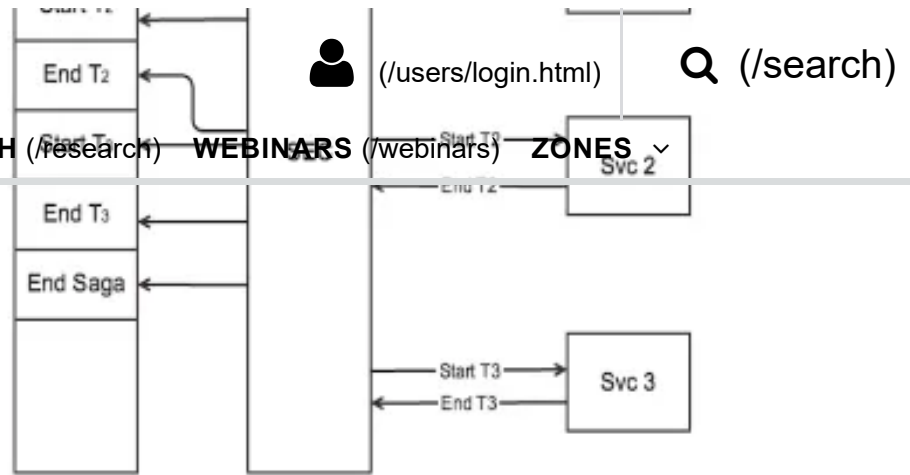
1. Begin Saga
2. Start T_1
3. End T_1
4. Start T_2
5. Abort Saga
6. Start C_2
7. End C_2
8. Start C_1
9. End C_1
10. End Saga

There can be various ways for implementation of saga pattern, but to actually implement it in a scalable manner, we can introduce a central process aka **Saga Execution Coordinator or SEC**. SEC is merely a process that handles the flow of execution of transactions or compensatory transactions. It helps us centrally locate the logic of execution. However, unlike the initiating node in a two-phase commit, we can prevent it from being a single point of failure. This will be explained after we actually understand the full structure and flow.

Another important constituent in order to implement our form of saga pattern in the **Saga Log**. Just like a database log, it's a basic, durable but distributed source of information. Every event that SEC executes is documented in a saga log. A good example of that could be a Kafka topic.

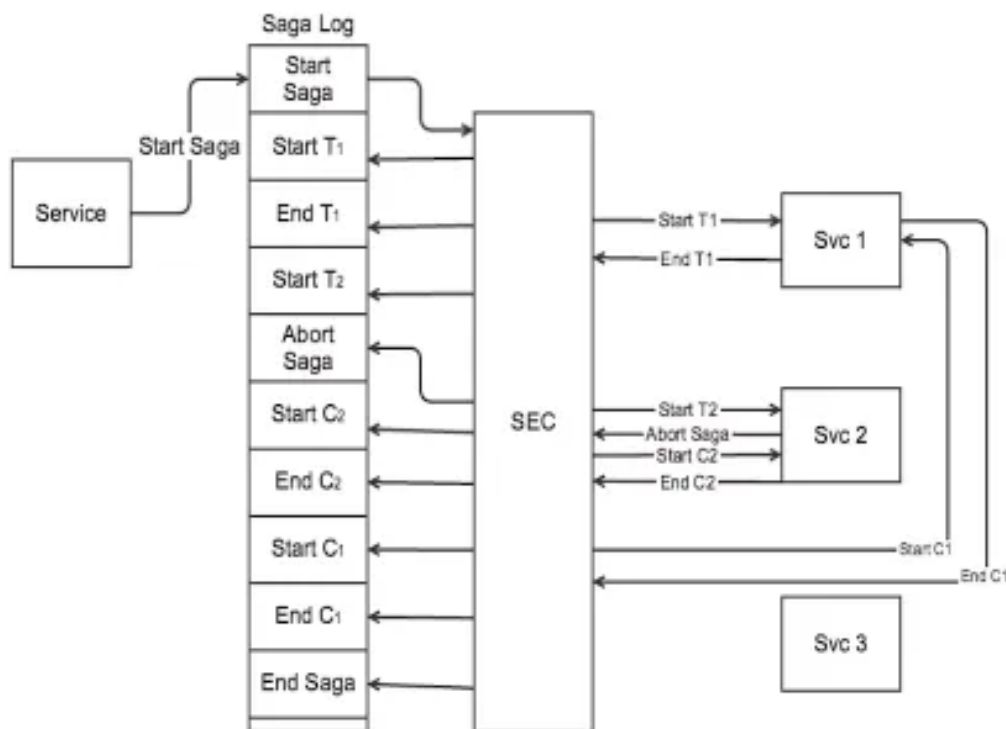
So a successful saga could be represented something like this.





Here, when we get a hit on the service, it marks the start of the saga on the topic where the SEC is listening. Then SEC logs the start of T1 and hits the service 1 for its transaction. The service responds to the SEC who then logs it as the end of T1. After this, SEC similarly logs transaction T2 and moves in an identical manner. When the last transaction has returned success, the SEC logs it and then logs the end of the Saga.

Now let's discuss the failure case as well.



Everything works in the similar fashion as above till the start of T2, where we encounter some issues and service indicates a failure. The SEC after receiving this message logs to abort the saga. It then undoes the changes made by T2 (if at all) by issuing compensatory transaction C2, which is first logged and then hit on service 2. The response marks the end of C2 which is logged and then the SEC tries to undo T1

REGARDZ (/regardz) **RESEARCH** (/research) **WEBINARS** (/webinars) **ZONES** (/zones)
One question here could be "What if the compensatory transaction fails?"


Well, simply put, we can't afford that. We need to undo any changes made by the canceled transaction in order to maintain data consistency. So we try again repeated till the time it succeeds. We make them idempotent.

Now coming back to the SEC. The SEC is not all that special because we CAN actually afford to lose it. There can be two cases in consideration

1. Safe state: Safe state exists when the SEC failed at the point of time where we were already under abort or when we had logged the end of the last transaction and not started a new one. In abort, since compensatory transactions are idempotent, we are already guaranteed a consistent state so we needn't worry about it. And in the other case, well we can quickly spin up another SEC who could carry on from the point of failure of the previous one.
2. An unsafe state would be when SEC fails at a point of time when we have started the transaction and not yet received any confirmation. It could have persisted with the changes, not even started the execution of the transaction, or even failed the transaction. We could never be certain. Which is why we would spin up another SEC and ask it to abort the current saga.

This probably gives us a fair insight into distributed transactions and saga patterns. Feel free to reach out to us in case of any queries. We appreciate multiple POVs.

Topics: MICROSERVICES, DISTRIBUTED TRANSACTIONS, SOFTWARE ARCHITECTURE, PATTERNS, DISTRIBUTED APPLICATIONS

Published at DZone with permission of Ayush Prashar, DZone MVB. [See the original article here.](#)  (<https://blog.knoldus.com/distributed-transactions-and-saga-patterns/>)

Opinions expressed by DZone contributors are their own.

Popular on DZone

- **Introducing JSON Tables** (/articles/introducing-json-tables?fromrel=true)
- **Generative Adversarial Networks (GANs) Business Use Cases** (/articles/generative-adversarial-networks-business-use-cases?fromrel=true)