Microservice Architecture (/index.html)

Supported by Kong (https://konghq.com/)

Pattern: Saga

🏷 pattern (/tags/pattern) 🏷 transaction management (/tags/transaction management) 🏷 sagas (/tags/sagas) 🏷 service collaboration (/tags/service collaboration) 🏷 implementing commands (/tags/implementing commands)

Context

You have applied the Database per Service (database-per-service.html) pattern. Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to implement transactions that span services. For example, let's imagine that you are building an e-commerce store where customers have a credit limit. The application must ensure that a new order will not exceed the customer's credit limit. Since Orders and Customers are in different databases owned by different services the application cannot simply use a local ACID transaction.

Problem

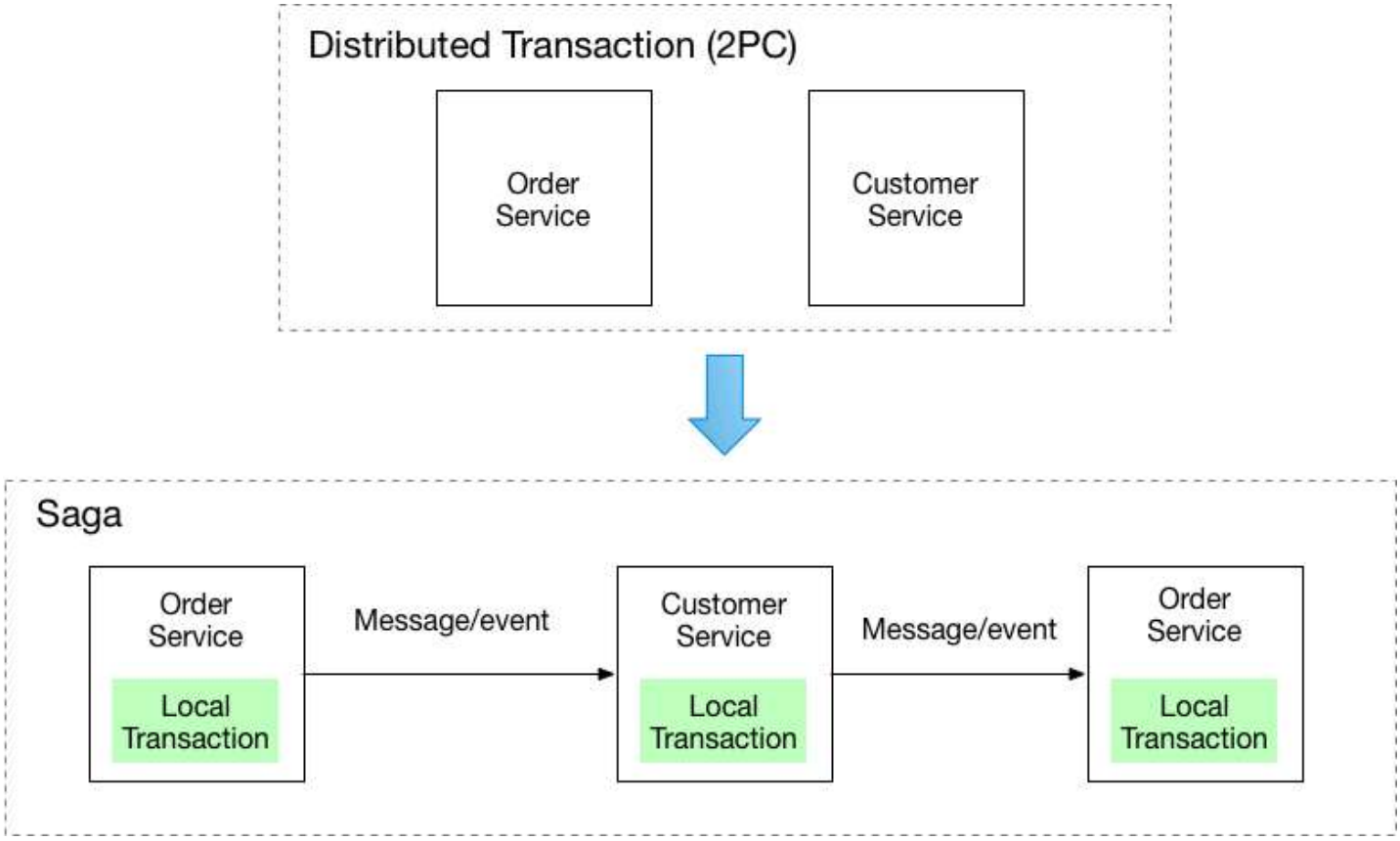
How to implement transactions that span services?

Forces

- 2PC is not an option

Solution

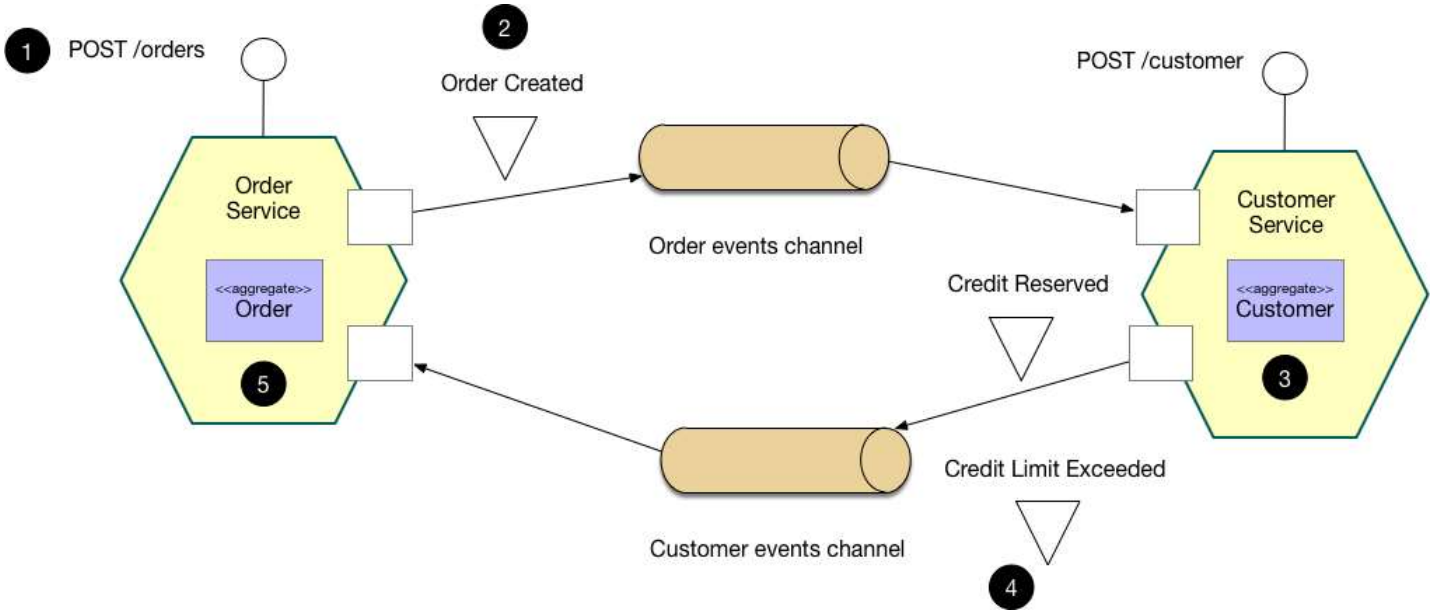
Implement each business transaction that spans multiple services as a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.



There are two ways of coordination sagas:

- Choreography - each local transaction publishes domain events that trigger local transactions in other services
- Orchestration - an orchestrator (object) tells the participants what local transactions to execute

Example: Choreography-based saga



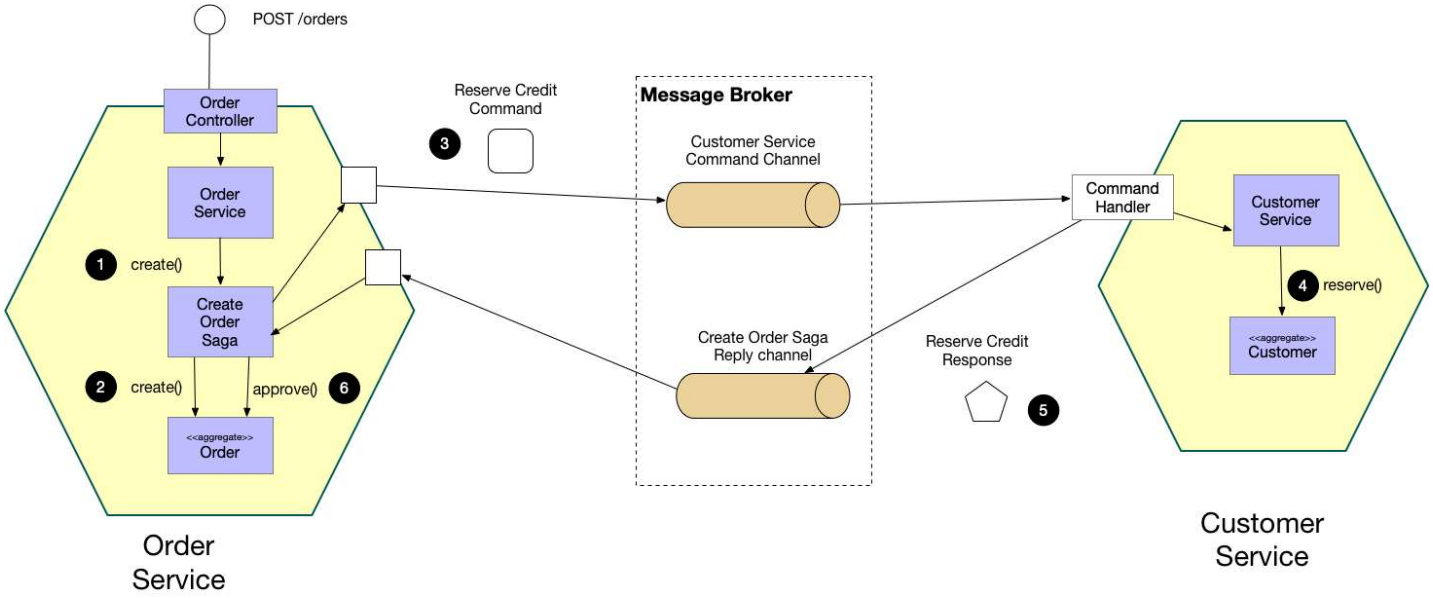
An e-commerce application that uses this approach would create an order using a choreography-based saga that consists of the following steps:

1. The `Order Service` receives the `POST /orders` request and creates an `Order` in a `PENDING` state

2. It then emits an `Order Created` event
3. The `Customer Service` 's event handler attempts to reserve credit
4. It then emits an event indicating the outcome
5. The `OrderService` 's event handler either approves or rejects the `Order`

Take a tour of an example saga (/post/architecture/2024/03/20/tour-of-two-sagas.html)

Example: Orchestration-based saga



An e-commerce application that uses this approach would create an order using an orchestration-based saga that consists of the following steps:

1. The `Order Service` receives the `POST /orders` request and creates the `Create Order` saga orchestrator
2. The saga orchestrator creates an `Order` in the `PENDING` state
3. It then sends a `Reserve Credit` command to the `Customer Service`
4. The `Customer Service` attempts to reserve credit
5. It then sends back a reply message indicating the outcome
6. The saga orchestrator either approves or rejects the `Order`

Take a tour of an example saga (/post/architecture/2024/03/20/tour-of-two-sagas.html)

Resulting context

This pattern has the following benefits:

- It enables an application to maintain data consistency across multiple services without using distributed transactions

This solution has the following drawbacks:

- Lack of automatic rollback - a developer must design compensating transactions that explicitly undo changes made earlier in a saga rather than relying on the automatic rollback feature of ACID transactions
- Lack of isolation (the “I” in ACID) - the lack of isolation means that there’s risk that the concurrent execution of multiple sagas and transactions can use data anomalies. consequently, a saga developer must typical use countermeasures, which are design techniques that implement isolation. Moreover, careful analysis is

needed to select and correctly implement the countermeasures. See Chapter 4/section 4.3 of my book Microservices Patterns (https://livebook.manning.com/book/microservices-patterns/chapter-4/143) for more information.

There are also the following issues to address:

- In order to be reliable, a service must atomically update its database *and* publish a message/event. It cannot use the traditional mechanism of a distributed transaction that spans the database and the message broker. Instead, it must use one of the patterns listed below.
- A client that initiates the saga, which an asynchronous flow, using a synchronous request (e.g. HTTP `POST /orders`) needs to be able to determine its outcome. There are several options, each with different trade-offs:
    - The service sends back a response once the saga completes, e.g. once it receives an `OrderApproved` or `OrderRejected` event.
    - The service sends back a response (e.g. containing the `orderID`) after initiating the saga and the client periodically polls (e.g. `GET /orders/{orderID}`) to determine the outcome
    - The service sends back a response (e.g. containing the `orderID`) after initiating the saga, and then sends an event (e.g. websocket, web hook, etc) to the client once the saga completes.

Related patterns

- The Database per Service pattern (database-per-service.html) creates the need for this pattern
- The following patterns are ways to *atomically* update state *and* publish messages/events:
    - Event sourcing (event-sourcing.html)
    - Transactional Outbox (transactional-outbox.html)
- A choreography-based saga can publish events using Aggregates (aggregate.html) and Domain Events (domain-event.html)
- The Command-side replica (command-side-replica.html) is an alternative pattern, which can replace saga step that query data

Learn more

- My book Microservices patterns (/book) describes this pattern in a lot more detail. The book's example application (https://github.com/microservice-patterns/ftgo-application) implements orchestration-based sagas using the Eventuate Tram Sagas framework (https://github.com/eventuate-tram/eventuate-tram-sagas)
- Take a look at my self-paced, online bootcamp (https://chrisrichardson.net/virtual-bootcamp-distributed-data-management.html) that teaches you how to use the Saga, API Composition, and CQRS patterns to design operations that span multiple services.
- Read these articles (/tags/sagas.html) about the Saga pattern
- My presentations (/presentations) on sagas and asynchronous microservices.

Example code

The following examples implement the customers and orders example in different ways:

- Choreography-based saga (https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders) where the services publish domain events using the Eventuate Tram framework (https://github.com/eventuate-tram/eventuate-tram-core)

- Orchestration-based saga (https://github.com/eventuate-tram/eventuate-tram-sagas-examples-customers-and-orders) where the `Order Service` uses a saga orchestrator implemented using the Eventuate Tram Sagas framework (https://github.com/eventuate-tram/eventuate-tram-sagas)
- Choreography and event sourcing-based saga (https://github.com/eventuate-examples/eventuate-examples-java-customers-and-orders) where the services publish domain events using the Eventuate event sourcing framework (http://eventuate.io/)

🏷 pattern (/tags/pattern) 🏷 transaction management (/tags/transaction management) 🏷 sagas (/tags/sagas) 🏷 service collaboration (/tags/service collaboration) 🏷 implementing commands (/tags/implementing commands)

L **Laurence Grant**

5 years ago

So Chris, this scares the "blank" out of me. It feels a lot like the old days of Informix databases performing a dirty-read. While a multi-table transaction was ongoing, a new query would read the uncommitted data, assuming that 99% of the time the data would be committed. Seems reasonable to some but much of the community hated it in favor of Oracle's approach to ALWAYS show the committed data, meaning it would read for the logs if a transaction was open to get the previous view of the data, because until the commit happens, the data in the table is temporarily in flux.

If I get your pattern, then each service is committing its data independently, and would later reverse the update/insert if the coordinator (or event) indicates something failed downstream, and the transaction must be undone. Meanwhile, that data was readable for a period of time, and perhaps even changeable by other service calls. That would potentially lead to data corruption.

Is it expected that the service is also keeping track of all transactions, either in memory or probably another table, to ensure two updates don't step on each other, or that a read doesn't return data before it's absolutely committed across all coordinated services?

All of this can/would greatly complicate the code, and is basically doing what a db was designed to do. Now I feel as a micro-services developer I'm basically coding the logic of a database transnational system. That's a very complicated system to try and reproduce and I sure don't want a team of application business developers all trying to recreate their own database transaction management system. It seems a bit absurd and a breakdown in the whole idea of micro-service making things easier?

I get why we want to decouple certain workflows, but there must be a point of no return, and it feels like we're encroaching on it. We might allow faster code deployments and elevation, but if it leads to data corruption, then we should never do that, or if it requires all my developers to be designing their own pseudo database to handle it, that doesn't make sense either.

There must be a better way!

12 0 Reply

Chris Richardson Mod → Laurence Grant

5 years ago

Sagas have the *potential* to be very complicated.
But often are not.
And, if you find that your sagas are excessively complicated, then that's time to reconsider how you have divided your application into services.

6 0 Reply

Edgewise → Chris Richardson

4 years ago edited

Honestly, I think this point needs to be highlighted.

I'm just learning about microservice concepts, having a lot of experience developing monolithic services, so take this with a grain of salt. But it sounds to me like you want to rely on sagas almost as a last resort, especially when you can decouple the data so much that precision is not essential.

For this reason, I think the common example of an order service and a customer service isn't the best. Depending on how you approach decomposition, I can see these resources being part of the same service, which would eliminate the need for sagas.

Ideally, you want to decompose your services in such a way as to minimize the number of transactions that you'll need to distribute across them. I think sagas help when you run into the inevitable cross-cutting concerns and it's more painful to redesign your entire architecture around them.

2 0 Reply

Guy Pardon → Laurence Grant

4 years ago

We tried to develop a better way after we got complaints about dirty reads and compensation logic overhead. Check this blog post if you're interested.

Cheers

0 0 Reply

L

Laurence Grant → Guy Pardon

4 years ago

Seems like everyone assumes the data isn't changing between your commit in MS 1 and a commit in MS 2. Consider you update MS1 then MS2 fails so you revert the change in MS1, but someone already changed the data you're trying to put back, so you can't revert it. Boom, data corruption. Without some form of lock across all micro services to ensure consistency until the last service commits its data, you risk data corruption

commits its data, you risk data corruption.

I get 99% of the time the conflict never occurs and everything seems great. But I care about the time that's going to cause corruption.

If I'm a social media site then I probably don't care, and can except some small discrepancy, or a dropped post on someones feed. However, if I'm a financial, banking, or medical system, there's zero tolerance for data corruption.

The complexity of SQL databases didn't just happen by chance. They evolved through necessity, and this new world of micro services feels like it was developed by a bunch of coders, who didn't understand all the advanced features under the hood of a SQL db and what ACID transactions really mean.

I really want to find a way to make it work, but feels like I have to invent my own transaction manager in code, along with a distributed lock manager to allow me consistent locking across micro services, so I'm able to safely roll back. It seems silly that I'm reinventing a DB basically, because I'm too hip to do it in a larger service. Maybe I don't need a monolith, but I need something to coordinate the subtitles of a complex transaction across separate of data.

damned if I do damned if I don't

8 0 Reply

Chris Richardson Mod → Laurence Grant

4 years ago

Yes. A compensating transaction risks overwriting updates made by a different saga/transaction.
There are various options, a.k.a. countermeasures.
Here are a couple:

1. Reorder the saga to eliminate the compensating transaction
2. Exploit the commutative nature of operations: undo a debit() (ie. subtract) with a credit() (i.e. add)
3. Use semantic locking, which is a form of application locking. e.g. the saga that revises an Order changes first changes its state ACCEPTED->REVISION_PENDING and then once its completed changes its state REVISION_PENDING->ACCEPTED. A cancel() transaction would see that the state is REVISION_PENDING and throw an exception (or in principle block).

I discuss these in my book, in my workshops and mention briefly here: https://www.slideshare.net/... (https://microservices.io/mi...

2 0 Reply

K

Kirill → Chris Richardson

4 years ago

Chris,

Do you think a distributed lock service can completely solve the isolation problem? For example, if we lock current Account Id in this example until rollback is completed, other transactions will not be able to change current Account data. I mean, the example application assumes (by business logic) that a particular account can be changed by only one transaction at a time, which means we will not sacrifice maximum availability, even using locks. What are the disadvantages of this approach?

0 0 Reply

Guy Pardon → Laurence Grant

4 years ago

Laurence,

Did you check the blog post I mentioned? We also support XA for microservices, so there is nobody changing the data you want to put back. It's a real rollback.

Cheers

0 0 Reply

Chris Richardson Mod → Guy Pardon

4 years ago

Presumably, you are propagating transaction IDs over REST.

Obviously, that works but the trade-off is reduced availability.

* Chains of REST calls N services long => availability^N
* Transaction can't commit until every participant is available to commit

Right?

3 0 Reply

Guy Pardon → Chris Richardson

4 years ago edited

Hi Chris,

Propagating transactions over REST is possible, we do that if desired. At first we didn't but customers kept on asking for this - so we gave in :-) Now it works.

Transactions can commit in the background due to self-healing recovery, but you are right: in synchronous systems you need more availability than in asynchronous ones.

That's why we also support propagation of transactions over messaging if desired

messaging if desired.

To be clear: it's not for every use case - just like Sagas aren't for everyone.

We did this because our Saga variant TCC got the same remarks every time: too expensive to do the compensation logic and people just don't want to bother yet.

In the future, Sagas or TCC will show useful for interactions across organisations, but not for in-house microservices. At least that is my intuition.

Cheers,
Guy

More info re TCC is here: https://www.atomikos.com/Bl... - in case you're interested. It's very popular in China but not yet in the West.

1 0 Reply

rahul

5 years ago edited

Hey Chris, Would the chorography based saga not create race condition? Lets say two orders came from customer to order service, one order in order service sends event for checking with customer service and marks the order as Pending, meanwhile 2nd order does the same, gets the response, and carries on ... while 1st order (still in pending state), it gets the response later(even when the credit limit would have exceeded due to 1st order eating it), how is this race condition solved? This would result in transaction even when customer did not have credit limit.

9 0 Reply

Chris Richardson Mod → rahul

5 years ago

The Customer Service would handle concurrent reserveCredit() requests using the regularly database concurrency mechanisms: locks or optimistic locking

2 0 Reply

Ilias Mertzanidis → Chris Richardson

4 years ago

Hi Chris,

if I understand the concept of Optimistic Locking, like for example using a annotated variable with @Version in Spring Boot, it is about preventing inconsistency when multiple processes want to update or delete a data record.

In the case of **@rahul** I thing is something totally different. He means I guess, what about if the 1st order gets the answer after 30 minutes let's say. The @Version field is already going to be incremented and the saga class would

not know if this response should be processed as normal. In other words the 1st order would not know that the response of the 2nd order came before the response of the 1st order.

What about this situation? What would you suggest ? I thing your solution is having a central orchestrator that controlls the hall workflow of the system and in this case would know or check the state of the order.

Please let me know if I understand something wrong.

0 0 Reply

ritesh thakur → Ilias Mertzanidis

4 years ago

The idea is, checking of available credit and actually deducting the credit has to be atomic for which optimistic locking on customer credit will be used.This way once the 2nd order has already exhausted the credit the first order will get an out of credit error and thus his 1st order wont succeeds.Hope this answers your question

3 0 Reply

parag3gwl → Ilias Mertzanidis

3 years ago

There are few assumptions -

1. If business requirement is - "We need not to allow more orders if due to previous pending orders credit is not available"

One of the solution could be - Let's say we have 2 tables in customer db. "customer_credit", "reserved_credit". For any new order customer service needs to reduce credit from customer_credit and park order amount in reserved_credit. For any new order customer service will always see available credit in customer_credit. based on available amount either it is going to be succeed or failed. Once any order is confirmed we can remove entry from reserved_credit and if order fails we can reverse(add) order amount back to customer_credit. **@Ilias Mertzanidis** I believe it will solve issue of which order gets fulfilled first.

2. If business Requirement is - "If order is in pending state but still we need to allow customer to place more order till actual credit is reduced"

One of the solution could be - **@ritesh thakur**'s solution will work.

1 0 Reply

Li Fu

5 years ago edited

so what's the difference between SAGA and event-driven architecture? especially choreography

based SAGA? the 3 steps are the same.

6 0 Reply

Chris Richardson Mod → Li Fu

5 years ago

An event-driven architecture is a generic concept.
Choreography-based sagas use events for the specific goal of maintaining data consistency.

4 1 Reply

redbush

5 years ago

I'm confused on how a Choreography based SAGA would work when you horizontally scale the services. What if there are five Customer services consuming the same OrderCreated event and publishing five of the same CreditReserved events? In the microservices book it talks about using a correlationId. Does this mean that the OrderService would need to discard duplicate correlationIds or just handle the duplicates as if the service has never seen the message.

5 0 Reply

Chris Richardson Mod → redbush

5 years ago

You need a mechanism like Kafka consumer groups. This is discussed in more detail in Chapter 3 of my book.

0 0 Reply

L

Lars Johansson

6 years ago

If I implement my microservice as an old-school J(2)EE application, what's preventing me from using a distributed transaction that spans the database and the message broker?

5 0 Reply

Marcus Leite → Lars Johansson

6 years ago

Nothing is preventing you, but you should avoid distributed transaction.

1) Modern technologies won't support it (RabbitMQ, Kafka, etc.);
2) This is a form of using Inter-Process Communication in a synchronized way and this reduces availability;
3) All participants of the distributed transaction need to be avaiable for a distributed commit, again: reduces availability.

Hope it helps.

44 0 Reply

Chris Richardson Mod → Marcus Leite
6 years ago

+1

3 0 Reply

Guy Pardon → Marcus Leite
4 years ago

Hi,

As to 1): yes modern technologies don't support it but maybe they will / should.

As to 2): not necessarily. Modern solutions exist that solve this.

As to 3): if you want to read a message from a broker and update your database (as in sagas?) then you would need availability too. So this argument does not make any sense either.

Hope this helps.

1 2 Reply

B

Bastian Kohnnet → Marcus Leite
6 years ago

Could https://debezium.io/ a solution for distributed transactions? They use Kafka and KafkaConnect.

0 0 Reply

Sachin → Lars Johansson
6 years ago

Do you have multiple IPC calls of microservices with CRUD transaction needed ?

0 0 Reply

L

Lars Johansson → Sachin
6 years ago edited

I was thinking hypothetically, as JEE is the stack we are currently on. If each microservice is implemented as a transactional session bean (deployed in its own EAR, possibly in its own app server) and both the JDBC datasource and the JMS queue support distributed transactions there's no reason the EJB transaction couldn't hold this together?

0 0 Reply

C

Chuck Zheng
6 years ago

what if execution of compensating transaction series fail? who is responsible to

detect/retry/recover?

In a portfolio of hundreds of microservices with dozens of key biz transactions each with 3-5 participating services, how to make sure sagas are implemented properly across all service teams? has anyone done that?

4 0 Reply

Chris Richardson Mod → Chuck Zheng

6 years ago

With choreography-based sagas, a saga participant will publish an event indicating failure (e.g. creditLimitExceeded), which will trigger other participants to execute compensating transactions.

With orchestration-based sagas, a saga participant will reply with a failure message, which tells the saga orchestrator to start executing compensating transaction.

You need to design this one saga at a time.

5 4 Reply

A

Ashwani Solanki → Chuck Zheng

6 years ago

Lets say a microservice's service does all the transactions then an eventbus(guava's eventbus) in service's container can hold all the relevant information for a transaction. Then on any services exception make an eventbus rollback action from catch block. Override the eventbus's rollback method to do condescending transactions for earlier recorded transactions on event bus. I did it in a springrest micro-service container once and works perfectly fine.

1 0 Reply

C

Chuck Zheng → Ashwani Solanki

6 years ago

Thx for the info. If you have a string of services in different deployment containers to propagate the change: S1 -> S2 -> S3 -> S4 -> ... Let's say you have S1, S2, S3 commit well and propagate events well. Then S4 hit a snag. Is there existing support to automate notifying S3 then S2 then S1 to rollback?

- what if one of them fail to rollback? e.g. S2 become (temporarily) unavailable?
- while this lengthy chain is propagating, what if the data committed by S1 is already consumed by other systems? (I guess this is a ISOLATION problem which SAGA does not have support)

6 0 Reply

Sean Xiong → Chuck Zheng

6 years ago

this is a trick scenario, and even if there is a service to be able to roll

back all those 3 changes successfully, its still possible to break the system when there is a credit limit checking operation processed just before the roll backs completed. the credit limit checking result will be processed base on incorrect credit value stored in DB.

however there is a solution to guarantee the correctness of the credit limit value stored in DB if we can guarantee all the write operations, include roll backs, be processed in the order of created time. one approach is put all write operations to a fifo queue.

lets use your use scenario as a example

At T0, the credit value for customer A is 100, and then there are 4 operations to change it.

S4: +400
S3: +300
S2: +200

see more

6 3 Reply

Muhammad Omar Butt → Sean Xiong

5 years ago

using the solution proposed by **@Sean Xiong** the microservice will no longer be horizontally scale-able as no concurrent executions will be possible and it will become bottleneck for performance.

2 0 Reply

L

lmao → Muhammad Omar Butt

5 years ago

We can shard the queue into multiple queues keyed by some identifier in each write request. It is possible that one queue may fail and all those write requests fail to get received by the queue's listener, but as long as each enqueue operation is synchronous and returns an indicator of whether or not the enqueue failed, we can take appropriate action based on this indicator (e.g. send to another working queue).

0 0 Reply

Chris Richardson Mod → Sean Xiong

6 years ago

Take a look at this presentation/video: https://www.slideshare.net/...

Summary: Sagas lack isolation - they are ACD rather than ACID. You have to use countermeasures to enforce isolation at the application

level.

1 2 Reply

C **Chuck Zheng** → Sean Xiong

5 years ago

Granted Distributed Tx is hard and costly. But there is an new design how this can be done relatively efficiently with microservices with distributed DB - pls check out this IEEE paper "GRIT: Consistent Distributed Transactions Across Polyglot Microservices with Multiple Databases". The lead designer of this approach is Guogen Zhang

0 0 Reply

I **Ivan**

5 years ago

There is a time gap between saving order in the db and emitting an event. What if the process crashes right after committing db transaction and event will never be produced? Does it mean that the created order will stuck in pending state? Thank you.

3 0 Reply

Chris Richardson Mod → Ivan

5 years ago

See the related patterns section, which references Event Sourcing and Transactional Outbox patterns.

1 0 Reply

D **Deepak r kiran**

6 years ago

How about creating a Finite state Machine (out of the application) and individual microservice updates their status of transaction. Using zookeeper could also be helpful. Any thoughts ?

3 0 Reply

L **lmao** → Deepak r kiran

5 years ago

I see the choreographer pattern as an implementation of an FSM. Each channel represents a state change, and each microservice is a service node that receives responses for each their requests from a reply channel. Within each node is an FSM that is contained only within the scope of the microservice. Starting from the initial response, the microservice can inspect the state of their request and route it to an appropriate handler within itself, which represents a state change.

0 0 Reply

水 **水球潘** → Deepak r kiran

5 years ago edited

This is exactly what I'm also thinking, finite state machine is well applied in the game aspect, wondering if the concept also well fits in microservices, cuz now it's event-driven, just like a set of game objects who know their own lifecycles.

Every event handler should behave like a finite state machine, and we can visualize it in a finite state machine diagram to make every service maintainable.

0 0 Reply

O

oskarcarlstedt

5 years ago edited

I believe we have to state it clearly; there is no way to guarantee a distributed transaction in computer science. It is logically impossible. However, we can do as much as possible to prevent it from causing inconsistent data. However, I do like the pattern showed here, nice and clearly explained. Good job on you description!!! As already commented there are mechanisms in e.g. Spring Framework managing such best practices automatically. BUT, there are no guarantees a distributed transaction exits in full success without data inconsistency. Here's why:

When two or more components shall cooperate in a distributed transaction they will have to implement local atomic transactions. The first one may succeed, so also the second one but the third one may fail and then call for a rollback. In such case rollback is called on atomic transaction 1 and atomic transaction 2, which are already committed as they are atomic - meaning each rollback must be implemented as its own atomic transaction, which may fail itself/themselves and then causing the original distributed transaction to result in an inconsistent dataset.

2 0 Reply

A

Atul Agrawal

5 years ago

don't you think orchestration-based sagas has single point of failure?

1 0 Reply

Chris Richardson Mod → Atul Agrawal

3 years ago

No. The orchestration logic can be highly available.

0 0 Reply

Golden King → Atul Agrawal

3 years ago

Yes it does. There are some flaws in the example. The orchestrator should not live in the order service and it should not hold any state in-memory. Here's a better illustration of what a saga with a central orchestrator should look like:
https://www.youtube.com/wat...

0 0 Reply

Chris Richardson Mod → Golden King

3 years ago

There are tradeoffs around the location of the orchestrator so blanket statements 'should not' are not true.

Moreover, the state is in the DB (in the Eventuate framework) and not in memory.

0 0 Reply

Hrishikesh

6 years ago

Hi,
Nice and informative article, Please publish kindle edition of your book on Amazon

1 0 Reply

Manuel Silva

6 years ago

(y)

1 0 Reply

Phuc hoang trong