

Microservice Architecture (/index.html)

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

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

Pattern: Saga

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

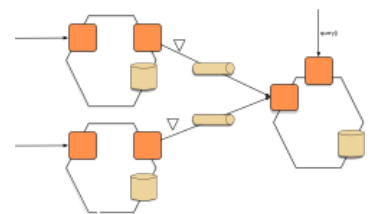
Solution

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.

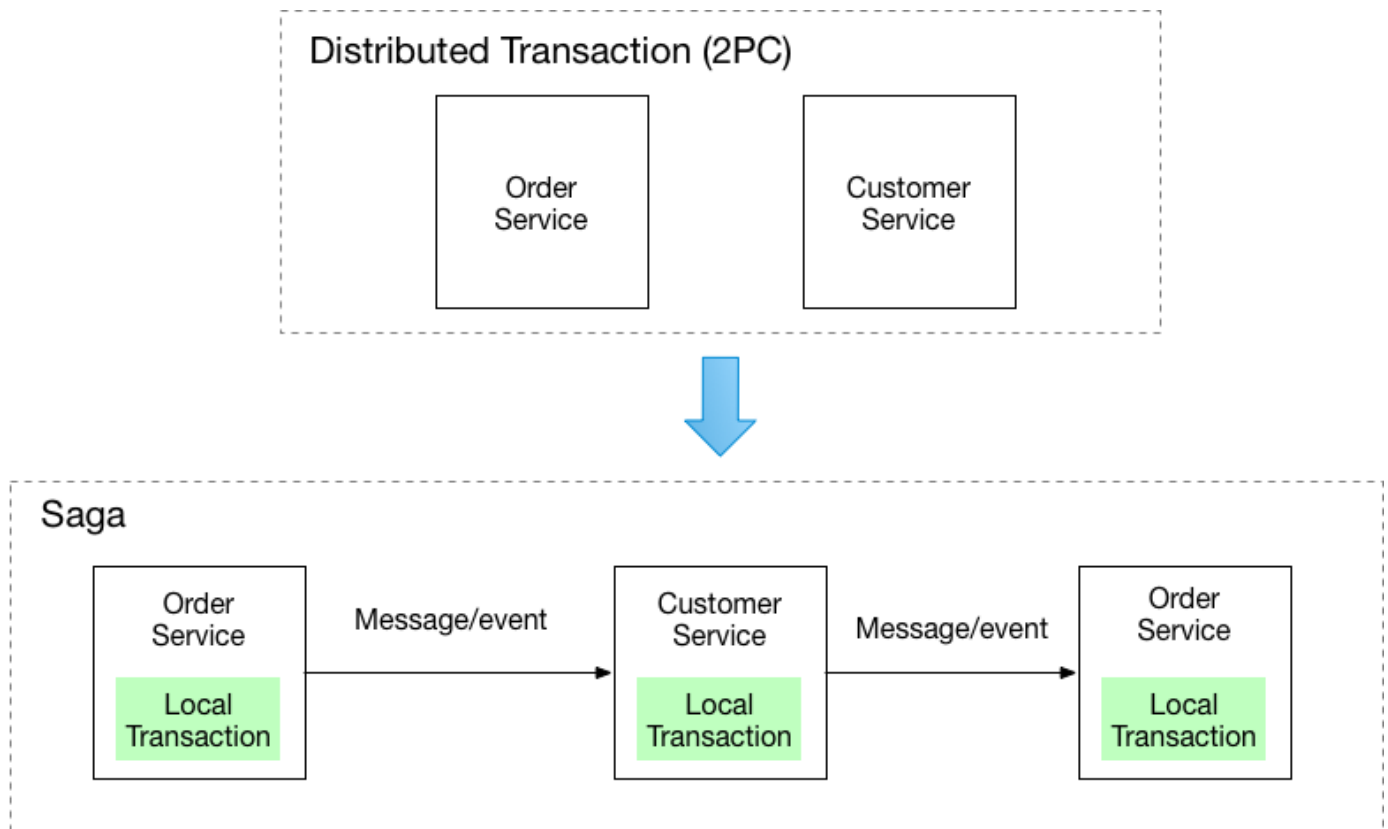
Want to learn more about this pattern?

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.

The regular price is \$395/person but use coupon JUNVCEJE to sign up for \$195 (valid until February 1st, 2023)



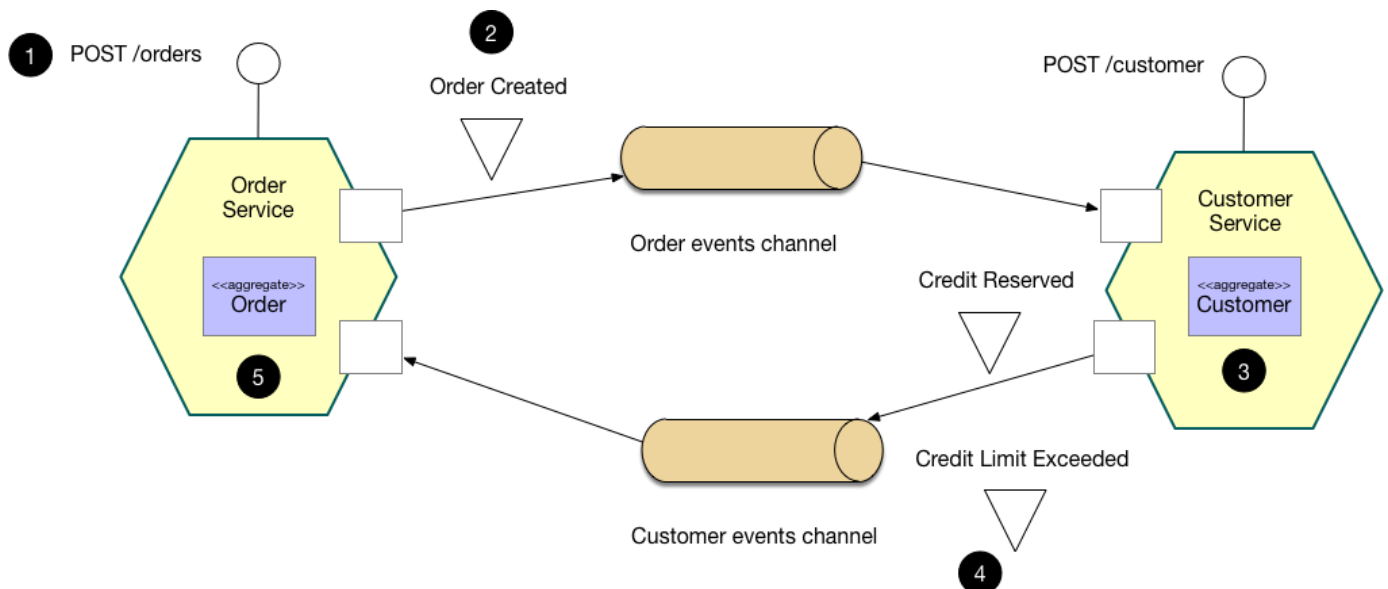
(<https://chrisrichardson.net/virtual-bootcamp-distributed-data-management.html>)



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

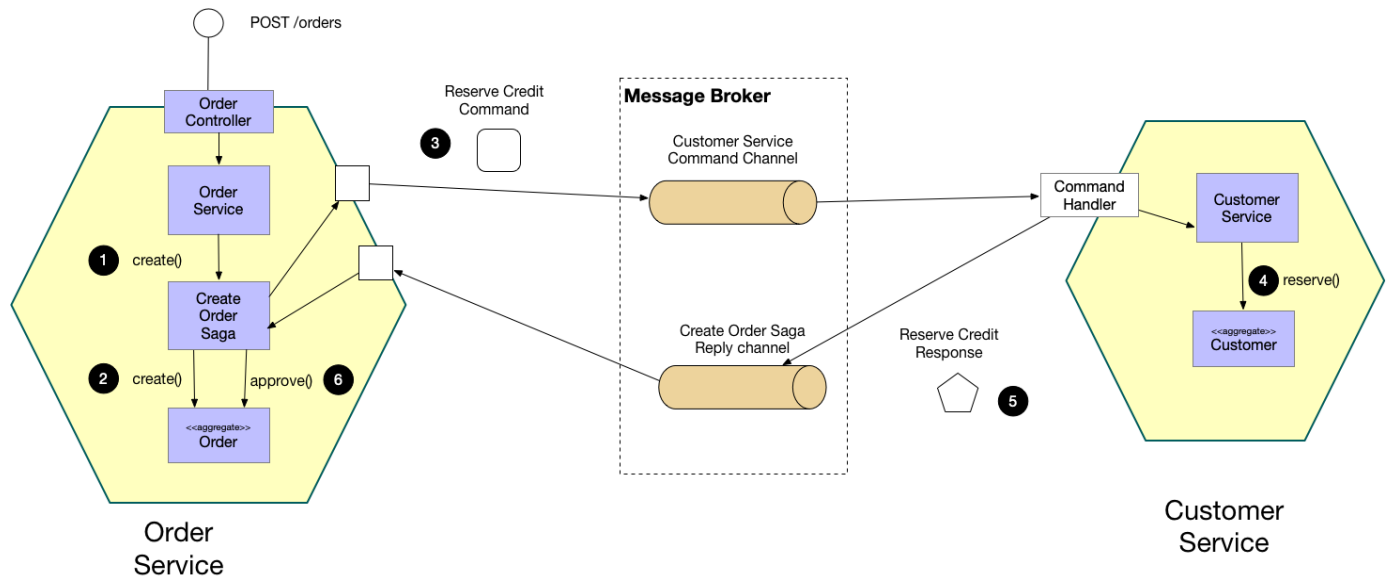
Example: Choreography-based saga



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`

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

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:

- The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.

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 is 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](#))






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 \(/tags/pattern\)](/tags/pattern)
 [transaction management \(/tags/transaction](/tags/transaction)
[management\)](#)
 [sagas \(/tags/sagas\)](/tags/sagas)
 [service collaboration \(/tags/service](/tags/service)
[collaboration\)](#)
 [implementing commands \(/tags/implementing commands\)](/tags/implementing)

Tweet

Follow [@MicroSvcArch](#)

Copyright © 2023 Chris Richardson • All rights reserved • Supported by Kong (<https://konghq.com/>).

ALSO ON MICROSERVICES

5 years ago · 7 comments

API composition

6 years ago · 1 comment

Exception tracking

3

\$

101 Comments

1

Login ▾

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

?

Name

10

Share

Best Newest Oldest

rahul —

🕒 3 years ago edited

Hey Chris, Would the choreography 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