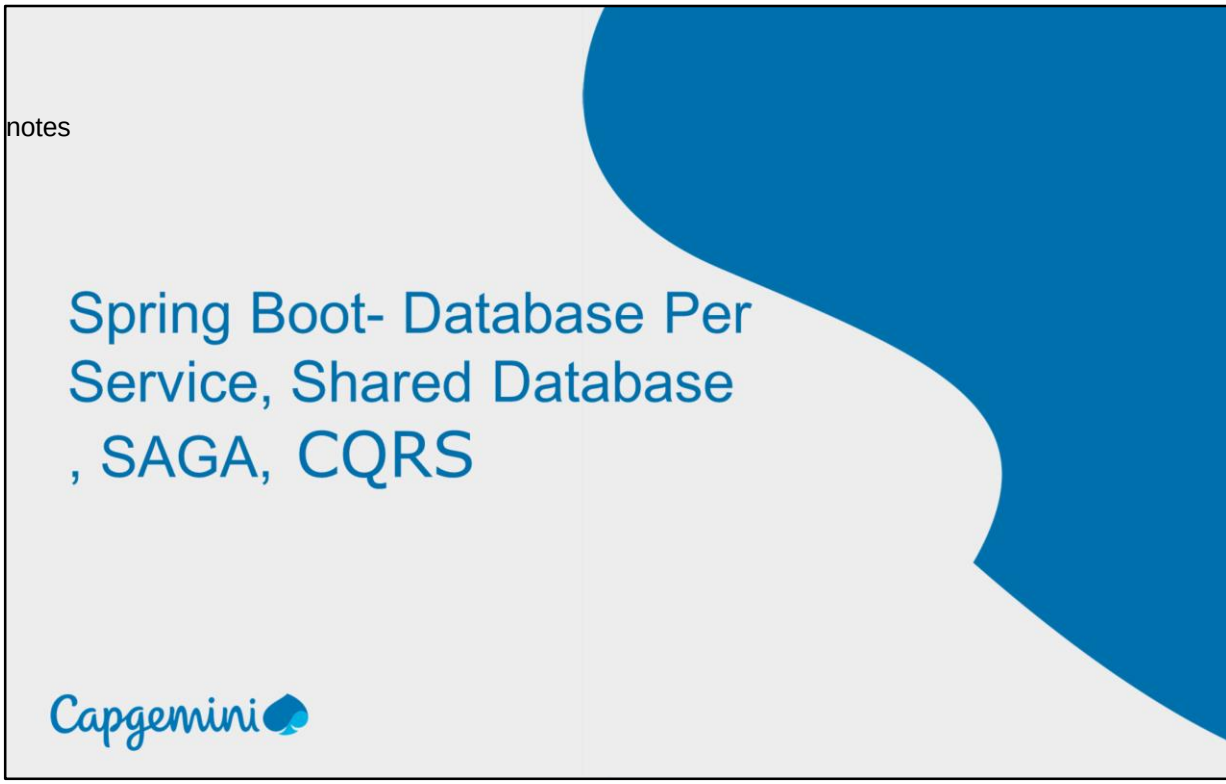
Add instructor notes here.



Objective

- Database Per Service
- Shared Database
- SAGA
- CQRS

Database Per Service

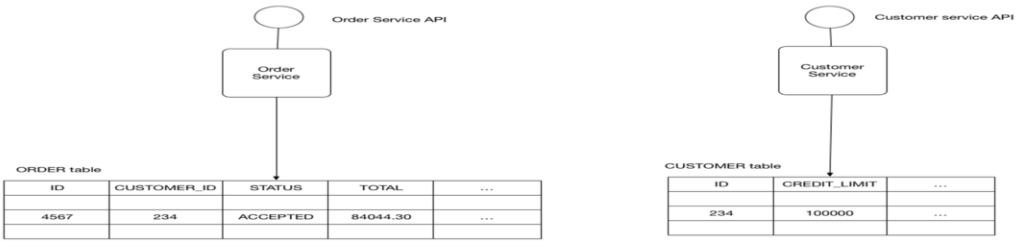
- Each microservice manages its own data. What this implies is that no other microservice can access that data directly. Communication or exchange of data can only happen using a set of well-defined APIs.
- Applications usually are not so well demarcated. Usually, microservices need data from each other for implementing their logic. This leads to spaghetti-like interactions between various services in your application.
- The success of this pattern hinges on effectively defining the bounded contexts in your application. For a new application or system, it is easier to do so. But for large and existing monolithic systems, it is troublesome.
- Other challenges include implementing business transactions that span several microservices. Another challenge could be implementing queries that want to expose data from two or three different bounded contexts.
- However, if done properly, the major advantages of this pattern are loose coupling between microservices. We can save your application from impact-analysis hell.
- Also, we could scale up microservices individually. It can provide freedom to the developers to choose a specific database solution for a given microservice.

Problem
What's the database architecture in a microservices application?
Forces Services must be loosely coupled so that they can be developed, deployed and scaled independently
Some business transactions must enforce invariants that span multiple services. For example, the Place Order use case must verify that a new Order will not exceed the customer's credit limit. Other business transactions, must update data owned by multiple services.
Some business transactions need to query data that is owned by multiple services. For example, the View Available Credit use must query the Customer to find the creditLimit and Orders to calculate the total amount of the open orders.
Some queries must join data that is owned by multiple services. For example, finding customers in a particular region and their recent orders requires a join between customers and orders.
Databases must sometimes be replicated and sharded in order to scale.
Different services have different data storage requirements. For some services, a relational database is the best choice. Other services might need a NoSQL database such as MongoDB, which is good at storing complex, unstructured data, or Neo4J, which is designed to efficiently store and query graph data.

Solution

Keep each microservice’s persistent data private to that service and accessible only via its API. A service’s transactions only involve its database.

Database Per Service



The service's database is effectively part of the implementation of that service. It cannot be accessed directly by other services.

There are a few different ways to keep a service's persistent data private. You do not need to provision a database server for each service. For example, if you are using a relational database then the options are:

Private-tables-per-service – each service owns a set of tables that must only be accessed by that service

Schema-per-service – each service has a database schema that's private to that service

Database-server-per-service – each service has it's own database server.

Private-tables-per-service and schema-per-service have the lowest overhead. Using a schema per service is appealing since it makes ownership clearer. Some high throughput services might need their own database server.

It is a good idea to create barriers that enforce this modularity. You could, for example, assign a different database user id to each service and use a database access control mechanism such as grants. Without some kind of barrier to enforce encapsulation, developers will always be tempted to bypass a service's API and access it's data directly.

Shared Database

- A shared database could be a viable option if the challenges surrounding Database Per Service become too tough to handle for our team.
- This approach tries to solve the same problems. But it does so by adopting a much more lenient approach by using a shared database accessed by multiple microservices.
- Mostly, this is a safer pattern for developers as they are able to work in existing ways. Familiar ACID transactions are used to enforce consistency.
- This approach takes away most of the benefits of microservices. Developers across teams need to coordinate for schema changes to tables. There could also be run-time conflicts when multiple services are trying to access the same database resources.

Problem

What's the database architecture in a microservices application?

Forces

Services must be loosely coupled so that they can be developed, deployed and scaled independently

Some business transactions must enforce invariants that span multiple services. For example, the Place Order use case must verify that a new Order will not exceed the customer's credit limit. Other business transactions, must update data owned by multiple services.

Some business transactions need to query data that is owned by multiple services. For example, the View Available Credit use must query the Customer to find the creditLimit and Orders to calculate the total amount of the open orders.

Some queries must join data that is owned by multiple services. For example, finding customers in a particular region and their recent orders requires a join between customers and orders.

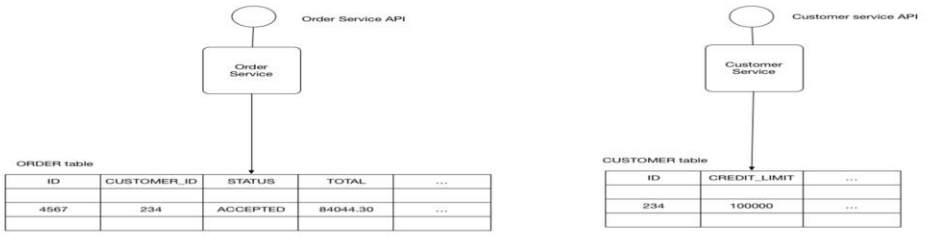
Databases must sometimes be replicated and sharded in order to scale.

Different services have different data storage requirements. For some services, a relational database is the best choice. Other services might need a NoSQL database such as MongoDB, which is good at storing complex, unstructured data, or Neo4J,

which is designed to efficiently store and query graph data.

Shared database

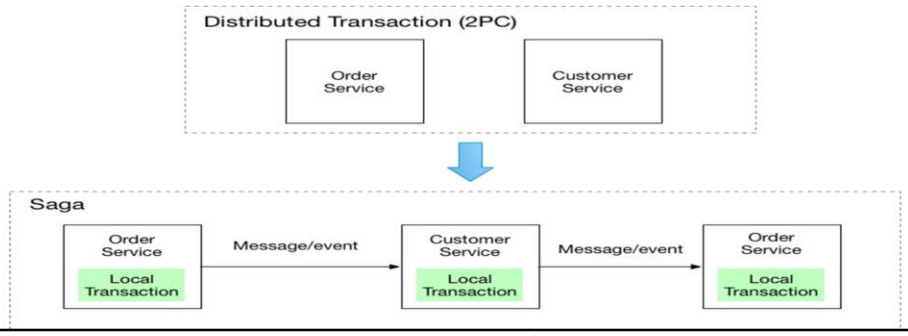
```
BEGIN TRANSACTION
SELECT ORDER_TOTAL
 FROM ORDERS WHERE CUSTOMER_ID = ?
SELECT CREDIT_LIMIT
FROM CUSTOMERS WHERE CUSTOMER_ID = ?
INSERT INTO ORDERS …
COMMIT TRANSACTION
```



The benefits of this pattern are:
A developer uses familiar and straightforward ACID transactions to enforce data consistency
A single database is simpler to operate
The drawbacks of this pattern are:
Development time coupling - a developer working on, for example, the OrderService will need to coordinate schema changes with the developers of other services that access the same tables. This coupling and additional coordination will slow down development.
Runtime coupling - because all services access the same database they can potentially interfere with one another. For example, if long running CustomerService transaction holds a lock on the ORDER table then the OrderService will be blocked.
Single database might not satisfy the data storage and access requirements of all services.

Saga Pattern

- A Saga is basically a sequence of local transactions. For every transaction performed within a Saga, the service performing the transaction publishes an event.
- The subsequent transaction is triggered based on the output of the previous transaction. And if one of the transactions in this chain fails, the Saga executes a series of compensating transactions to undo the impact of all the previous transactions.



Problem
How to maintain data consistency across 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

Saga Pattern

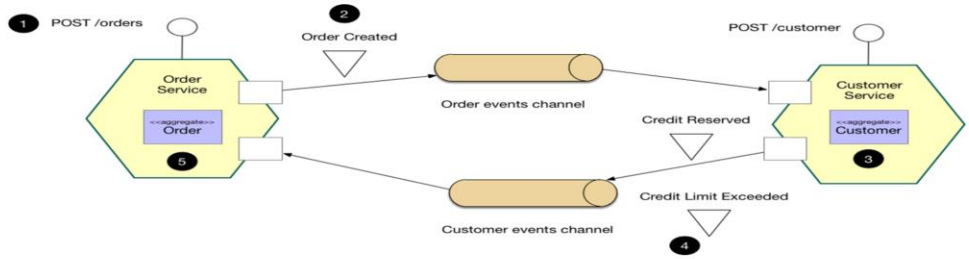
Example –Food Delivery Application

- Food Order service creates an order. At this point, the order is in a PENDING state. A Saga manages the chain of events.
- The Saga contacts the restaurant via the Restaurant service.
- The Restaurant service attempts to place the order with the chosen restaurant. After getting confirmation, it sends back a reply.
- The Saga receives the reply. And, depending on the reply, it can Approve the order or Reject the order.
- The Food Order service then changes the state of the order. If the order was Approved, it would inform the customer with the next details. If Rejected, it will also inform the customer with an apology message.
- We can see, this is drastically different from the usual point-to-point call approach. This approach adds complexity.
- Sagas are a very powerful tool to solve some tricky challenges. But they should be used sparingly.

SAGA -Example

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

- The Order Service receives the POST /orders request and creates an Order in a PENDING state
- It then emits an Order Created event
- The Customer Service's event handler attempts to reserve credit
- It then emits an event indicating the outcome
- The OrderService's event handler either approves or rejects the Order



The Saga pattern is the solution to implementing business transactions spanning multiple microservices.

A **Saga** is basically a sequence of local transactions. For every transaction performed within a Saga, the service performing the transaction publishes an event. The subsequent transaction is triggered based on the output of the previous transaction. And if one of the transactions in this chain fails, the Saga executes a series of compensating transactions to undo the impact of all the previous transactions.

To understand this better, let's take a simple example. Assume that there is a food-delivery app. When a customer tries to order food, the below steps can occur:

Food Order service creates an *order.* At this point, the order is in a PENDING state. A Saga manages the chain of events.

The Saga contacts the restaurant via the Restaurant service.

The Restaurant service attempts to place the order with the chosen restaurant. After getting confirmation, it sends back a reply.

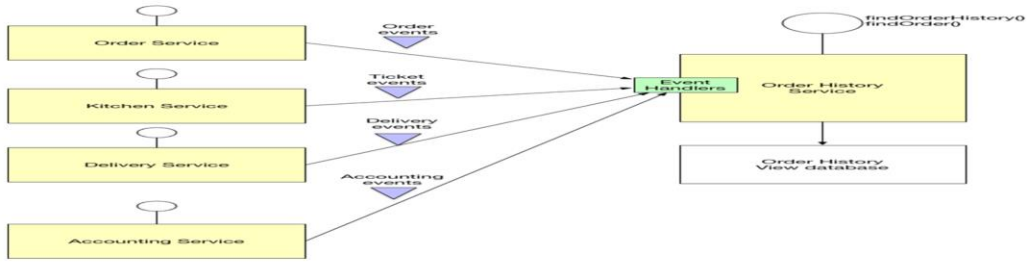
The Saga receives the reply. And, depending on the reply, it can Approve the order or Reject the order.

The Food Order service then changes the state of the order. If the order was Approved, it would inform the customer with the next details. If Rejected, it will also inform the customer with an apology message.

As you can see, this is drastically different from the usual point-to-point call approach. This approach adds complexity. However, in my view, Sagas are a very powerful tool to solve some tricky challenges. But they should be used sparingly.

CQRS

- CQRS, or Command Query Responsibility Segregation, is an attempt to get around the issues with API Composition pattern.
- An application listens to domain events from other microservices and updates the view or query database. We can serve complex aggregation queries from this database. We could optimize the performance and scale up the query microservices accordingly.
- The downside to this is an increase in complexity. All of a sudden, our microservice should be handling events. This can cause latency issues where the view database is eventually consistent rather than always consistent. It can also increase code duplication.



Problem

How to implement a query that retrieves data from multiple services in a microservice architecture?

Solution

Define a view database, which is a read-only replica that is designed to support that query. The application keeps the replica up to data by subscribing to Domain events published by the service that own the data.

Examples

My book's FTGO example application has the Order History Service, which implements this pattern.

There are several Eventuate-based example applications that illustrate how to use this pattern.

Resulting context

This pattern has the following benefits:

Supports multiple denormalized views that are scalable and performant

Improved separation of concerns = simpler command and query models

Necessary in an event sourced architecture

This pattern has the following drawbacks:

Increased complexity

Potential code duplication

Replication lag/eventually consistent views

Demo

demoCQRS