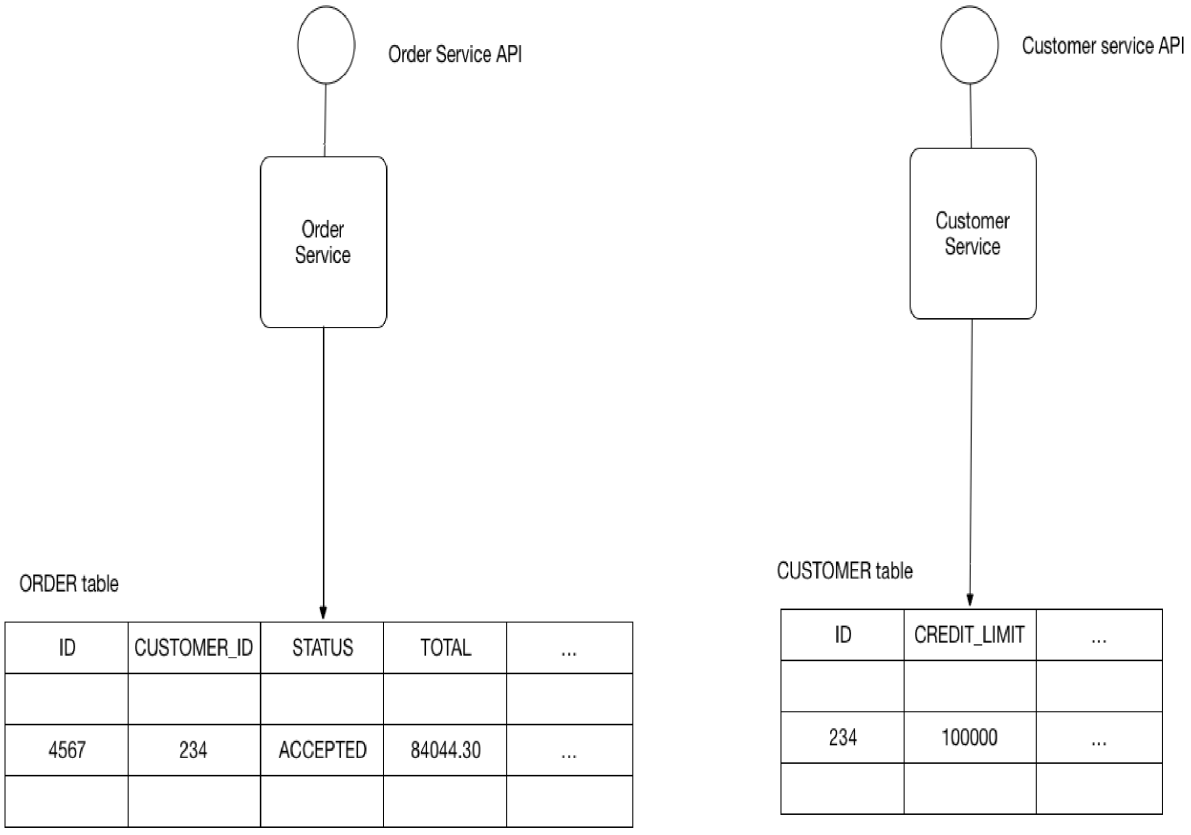
Pattern: Database per service

Context

Let's imagine you are developing an online store application using the Microservice architecture pattern. Most services need to persist data in some kind of database. For example, the Order Service stores information about orders and the Customer Service stores information about customers.



Problem

What's the database architecture in a microservices application?

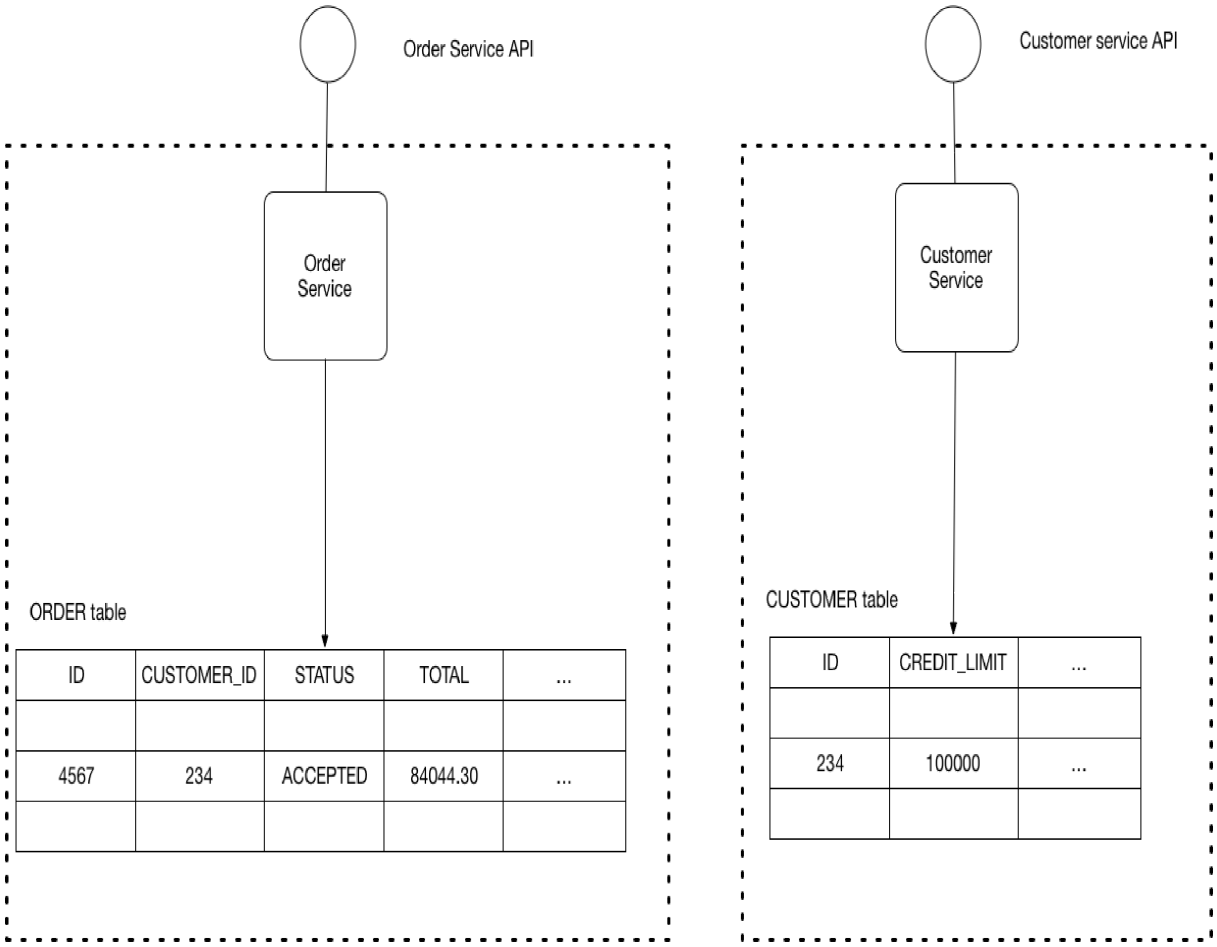
Forces

- 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. See the Scale Cube.
- 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.

The following diagram shows the structure of this pattern.



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.

Resulting context

Using a database per service has the following benefits:

- Helps ensure that the services are loosely coupled. Changes to one service's database does not impact any other services.
- Each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo4j.

Using a database per service has the following drawbacks:

- Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided because of the CAP theorem. Moreover, many modern (NoSQL) databases don't support them. The best solution is to use the Saga pattern. Services publish events when they update data. Other services subscribe to events and update their data in response.
- Implementing queries that join data that is now in multiple databases is challenging.
- Complexity of managing multiple SQL and NoSQL databases

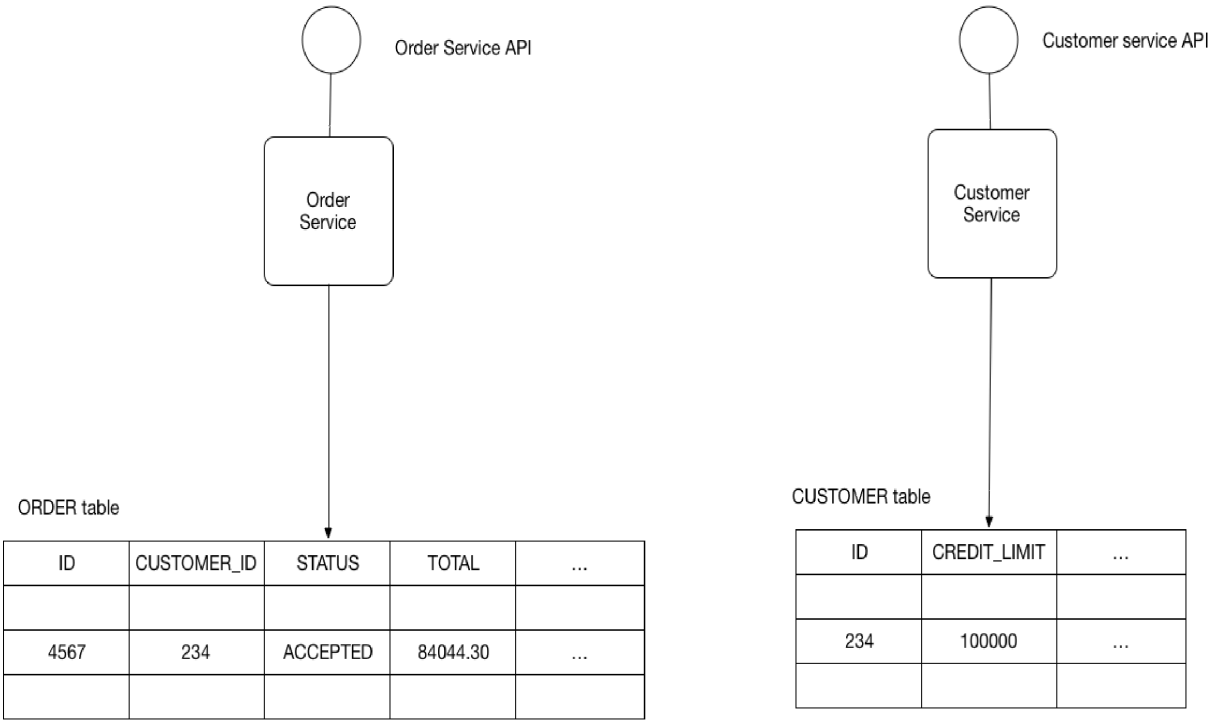
There are various solutions:

- **API Composition -** the application performs the join rather than the database. For example, a service (or the API gateway) could retrieve a customer and their orders by first retrieving the customer from the customer service and then querying the order service to return the customer's most recent orders.
- **Command Query Responsibility Segregation (CQRS)** - maintain one or more materialized views that contain data from multiple services. The views are kept by services that subscribe to events that each services publishes when it updates its data. For example, the online store could implement a query that finds customers in a particular region and their recent orders by maintaining a view that joins customers and orders. The view is updated by a service that subscribes to customer and order events.

Pattern: Shared database

Context

Let's imagine you are developing an online store application using the Microservice architecture pattern. Most services need to persist data in some kind of database. For example, the Order Service stores information about orders and the Customer Service stores information about customers.



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. See the Scale Cube.
- 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
Use a (single) database that is shared by multiple services. Each service freely accesses data owned by other services using local ACID transactions.

Example
The OrderService and CustomerService freely access each other's tables. For example, the OrderService can use the following ACID transaction ensure that a new order will not violate the customer's credit limit.

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

The database will guarantee that the credit limit will not be exceeded even when simultaneous transactions attempt to create orders for the same customer.

Resulting context
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.

Pattern: Saga

Context

You have applied the Database per Service pattern. Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to ensure data consistency across services. For example, lets 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 the application cannot simply use a local ACID transaction.

Problem

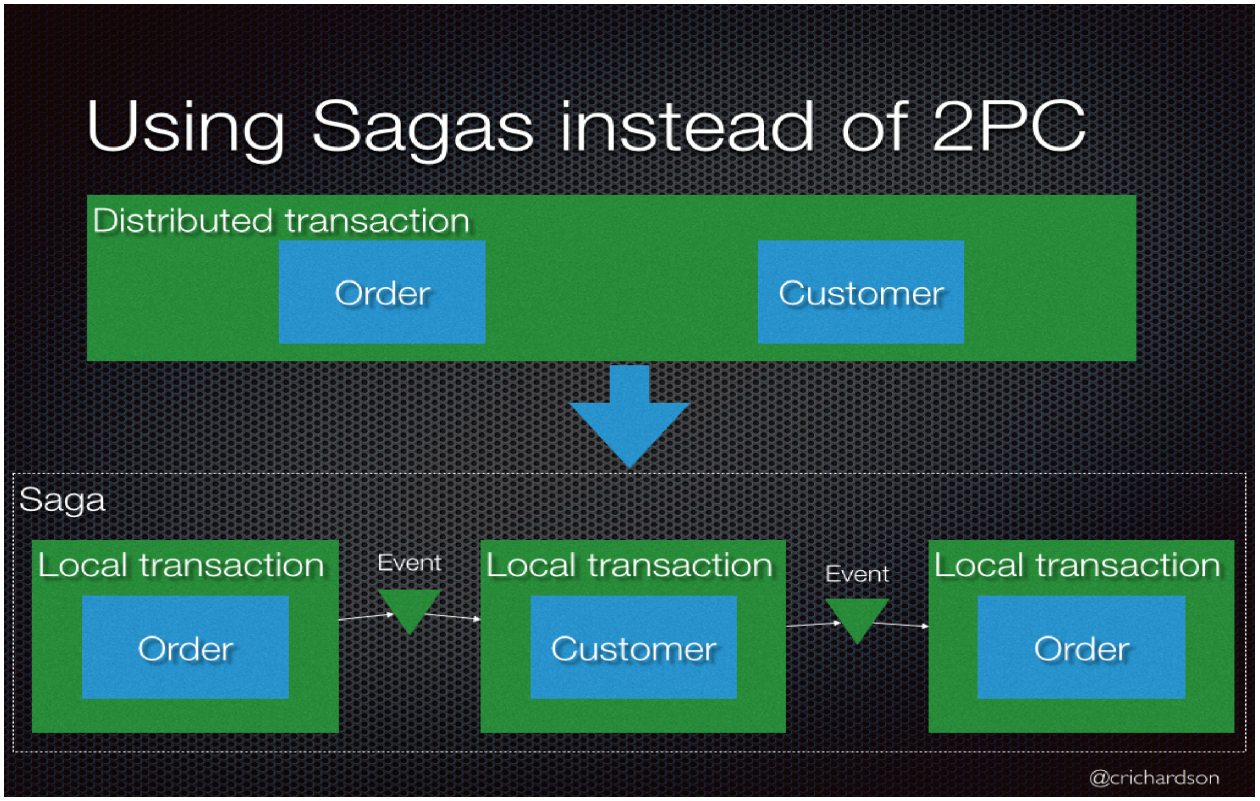
How to maintain data consistency across services?

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.

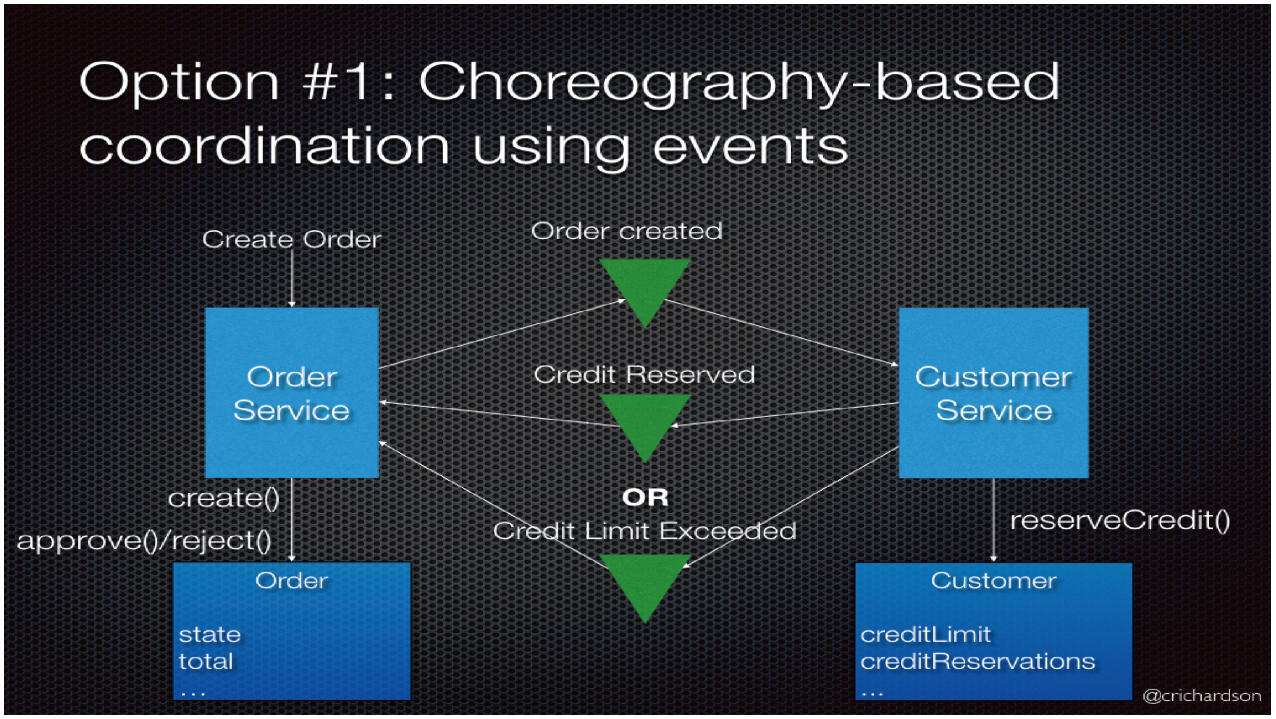


There are two ways of coordination sagas:

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

execute

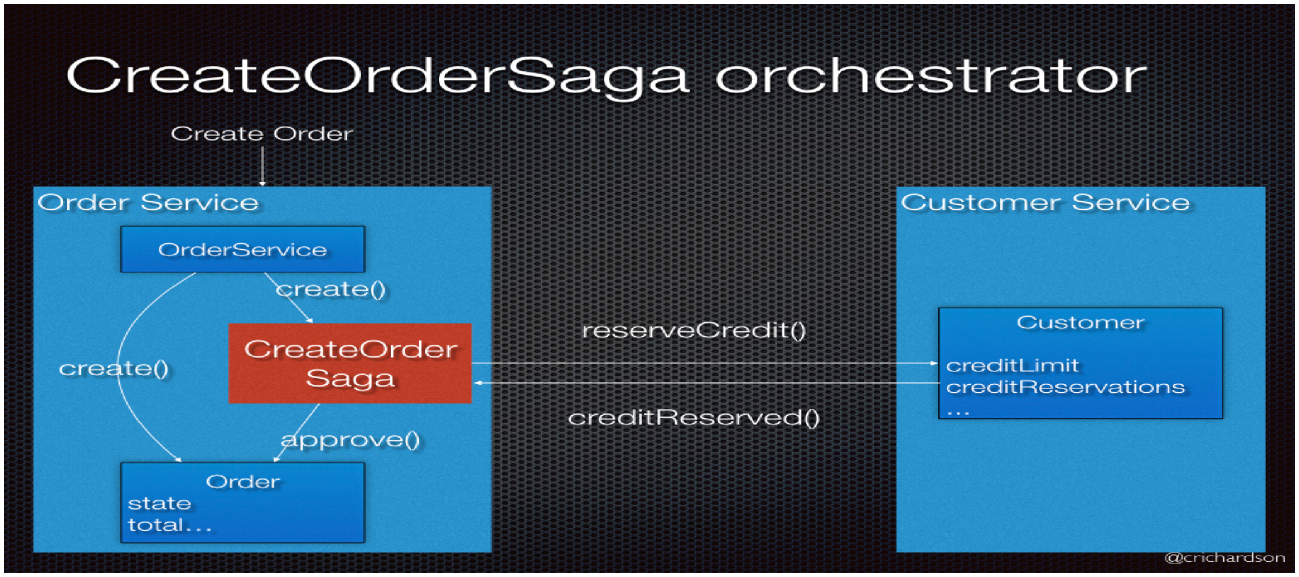
Example: Choreography-based saga



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

- The Order Service creates an Order in a pending state and publishes an OrderCreated event
- The Customer Service receives the event attempts to reserve credit for that Order. It publishes either a Credit Reserved event or a CreditLimitExceeded event.
- The Order Service receives the event and changes the state of the order to either approved or cancelled

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:

- The Order Service creates an Order in a pending state and creates a CreateOrderSaga
- The CreateOrderSaga sends a ReserveCredit command to the Customer Service
- The Customer Service attempts to reserve credit for that Order and sends back a reply
- The CreateOrderSaga receives the reply and sends either an ApproveOrder or RejectOrder command to the Order Service
- The Order Service changes the state of the order to either approved or cancelled

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.

Related patterns

- The Database per Service pattern creates the need for this pattern
- The following patterns are ways to atomically update state and publish messages/events:
    - Event sourcing
    - Application events
- A choreography-based saga can publish events using Aggregates and Domain Events

Pattern: API Composition

Context

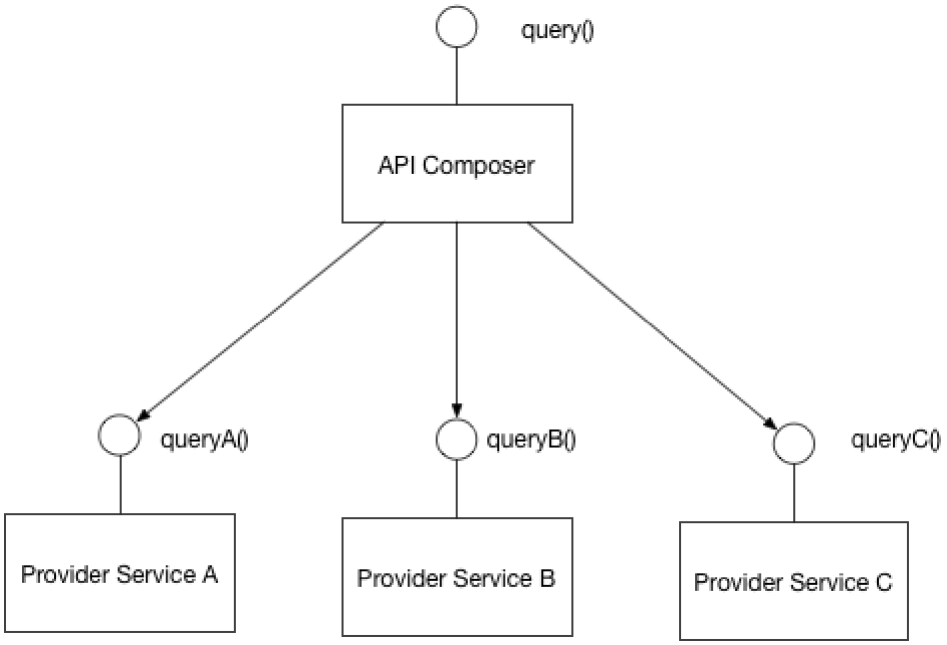
You have applied the Microservices architecture pattern and the Database per service pattern. As a result, it is no longer straightforward to implement queries that join data from multiple services.

Problem

How to implement queries in a microservice architecture?

Solution

Implement a query by defining an API Composer, which invoking the services that own the data and performs an in-memory join of the results.



Example

An API Gateway often does API composition.

Resulting context

This pattern has the following benefits:

- It a simple way to query data in a microservice architecture

This pattern has the following drawbacks:

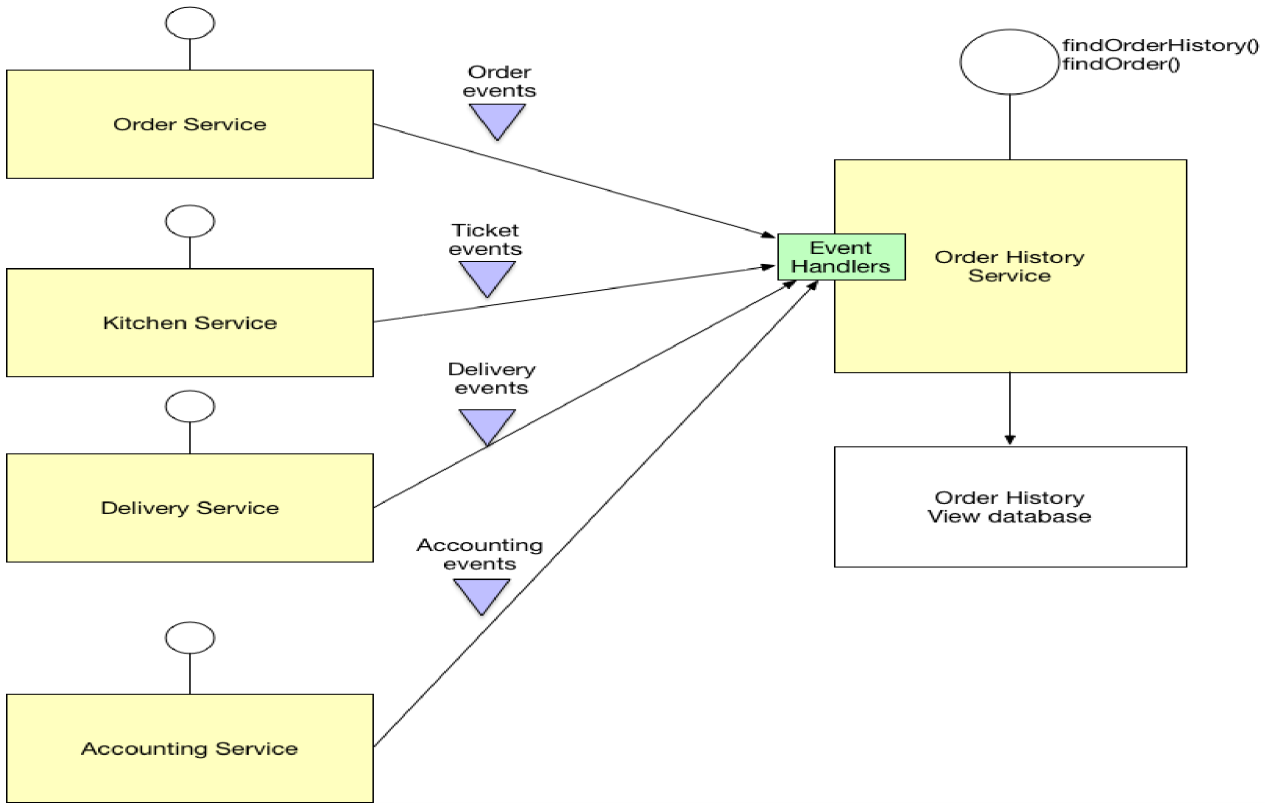
- Some queries would result in inefficient, in-memory joins of large datasets.

Pattern: Command Query Responsibility Segregation (CQRS)

Context
You have applied the Microservices architecture pattern and the Database per service pattern. As a result, it is no longer straightforward to implement queries that join data from multiple services. Also, if you have applied the Event sourcing pattern then the data is no longer easily queried.

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

Pattern: Event sourcing

Context
A service typically need to atomically update the database and publish messages/events. For example, perhaps it uses the Saga pattern. In order to be reliable, each step of a saga must atomically update the database and publish messages/events. Alternatively, it might use the Domain event pattern, perhaps to implement CQRS. In either case, it is not viable to use a distributed transaction that spans the database and the message broker to atomically update the database and publish messages/events.

Problem
How to reliably/atomically update the database and publish messages/events.

Forces
2PC is not an option

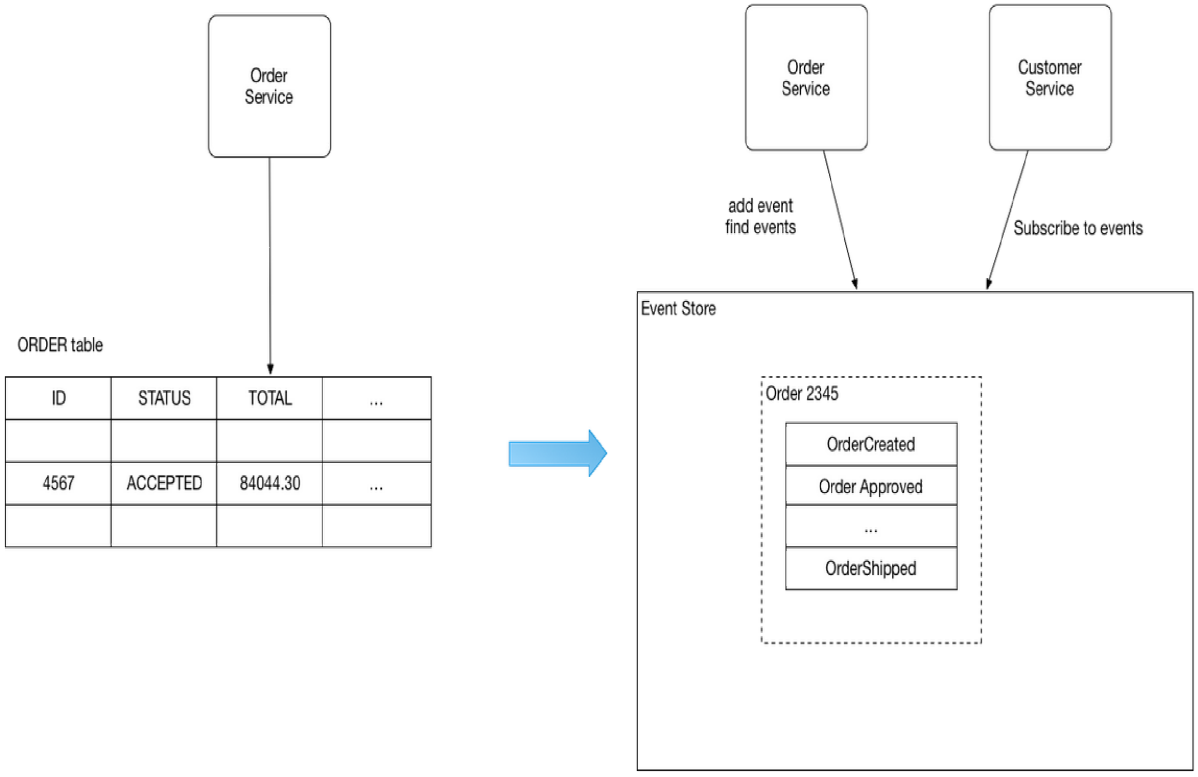
Solution
A good solution to this problem is to use event sourcing. Event sourcing persists the state of a business entity such an Order or a Customer as a sequence of state-changing events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic. The application reconstructs an entity's current state by replaying the events.

Applications persist events in an event store, which is a database of events. The store has an API for adding and retrieving an entity's events. The event store also behaves like a message broker. It provides an API that enables services to subscribe to events. When a service saves an event in the event store, it is delivered to all interested subscribers.

Some entities, such as a Customer, can have a large number of events. In order to optimize loading, an application can periodically save a snapshot of an entity's current state. To reconstruct the current state, the application finds the most recent snapshot and the events that have occurred since that snapshot. As a result, there are fewer events to replay.

Example
Customers and Orders is an example of an application that is built using Event Sourcing and CQRS. The application is written in Java, and uses Spring Boot. It is built using Eventuate, which is an application platform based on event sourcing and CQRS.

The following diagram shows how it persist orders.



Instead of simply storing the current state of each order as a row in an ORDERS table, the application persists each Order as a sequence of events. The CustomerService can subscribe to the order events and update its own state.

Here is the Order aggregate:

```
public class Order extends ReflectiveMutableCommandProcessingAggregate<Order, OrderCommand>
{

 private OrderState state;
 private String customerId;

 public OrderState getState() {
  return state;
 }

 public List<Event> process(CreateOrderCommand cmd) {
  return EventUtil.events(new OrderCreatedEvent(cmd.getCustomerId(), cmd.getOrderTotal()));
 }

 public List<Event> process(ApproveOrderCommand cmd) {
  return EventUtil.events(new OrderApprovedEvent(customerId));
 }

 public List<Event> process(RejectOrderCommand cmd) {
```

```
    return EventUtil.events(new OrderRejectedEvent(customerId));
  }

  public void apply(OrderCreatedEvent event) {
    this.state = OrderState.CREATED;
    this.customerId = event.getCustomerId();
  }

  public void apply(OrderApprovedEvent event) {
    this.state = OrderState.APPROVED;
  }


  public void apply(OrderRejectedEvent event) {
    this.state = OrderState.REJECTED;
  }
```

Here is an example of an event handler in the CustomerService that subscribes to Order events:

```
@EventSubscriber(id = "customerWorkflow")
public class CustomerWorkflow {

  @EventHandlerMethod
  public CompletableFuture<EntityWithIdAndVersion<Customer>> reserveCredit(
        EventHandlerContext<OrderCreatedEvent> ctx) {
    OrderCreatedEvent event = ctx.getEvent();
    Money orderTotal = event.getOrderTotal();
    String customerId = event.getCustomerId();
    String orderId = ctx.getEntityId();

    return ctx.update(Customer.class, customerId, new ReserveCreditCommand(orderTotal, orderId));
  }

}
```

It processes an OrderCreated event by attempting to reserve credit for the orders customer.

There are several example applications that illustrate how to use event sourcing.
Resulting context

Event sourcing has several benefits:

It solves one of the key problems in implementing an event-driven architecture and makes it possible to reliably publish events whenever state changes.

Because it persists events rather than domain objects, it mostly avoids the object-relational impedance mismatch problem.

It provides a 100% reliable audit log of the changes made to a business entity

It makes it possible to implement temporal queries that determine the state of an entity at any point in time.

Event sourcing-based business logic consists of loosely coupled business entities that exchange events. This makes it a lot easier to migrate from a monolithic application to a microservice architecture.

Event sourcing also has several drawbacks:

It is a different and unfamiliar style of programming and so there is a learning curve.

The event store is difficult to query since it requires typical queries to reconstruct the state of the business entities. That is likely to be complex and inefficient. As a result, the application must use Command Query Responsibility Segregation (CQRS) to implement queries. This in turn means that applications must handle eventually consistent data.