Microservices Notes

Problem:

* Execute synchronized update for bank account. How to validate synchronized DB ?

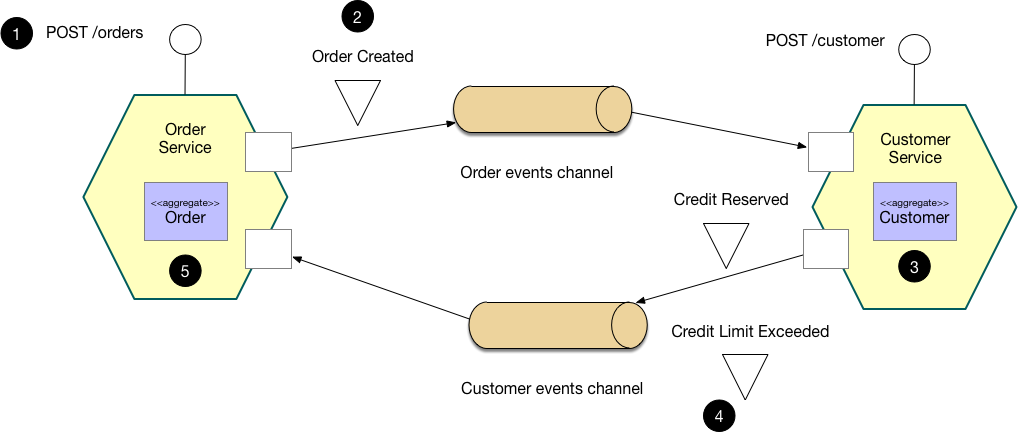
Solution:

1. DB guarantee that by applying 2 phases commit and 2 phases lock protocol by locking updating DB record until executing 1 update then refresh data before executing second update.
2. For distributed system we can apply Event sourcing or Saga design pattern to communicate between microservices.

Problem:

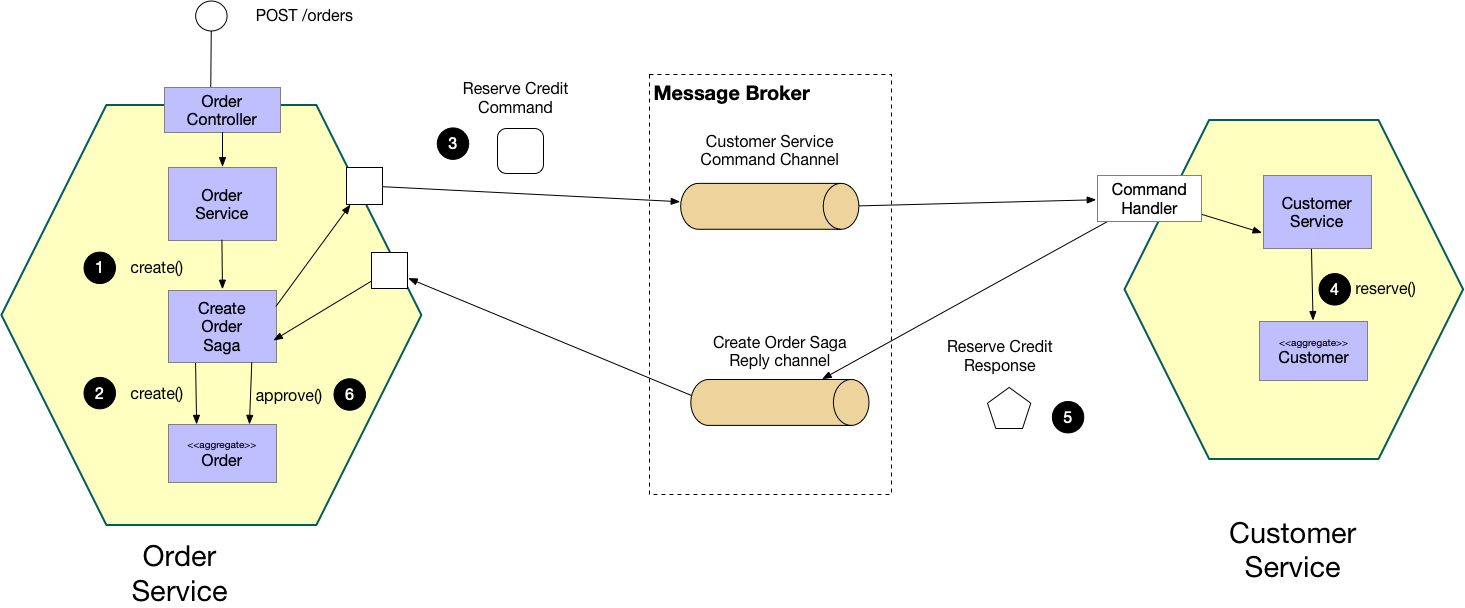
* What is Saga Design Pattern and what is the benefits of applying it ?
* have a credit limit. ensure a new order will not exceed the customer’s credit limit. Orders and Customers are in different databases owned by different services the application cannot simply use a local ACID transaction.

Solution:

* It enables an application to maintain data consistency across multiple services without using distributed transactions
* **Cryptograph based Saga** [no orchestrator in order service]
* 

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

* **Orchestration- based Saga** [orchestrator exist to organize order(send and receive response then decide action)]



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.

Benefits:

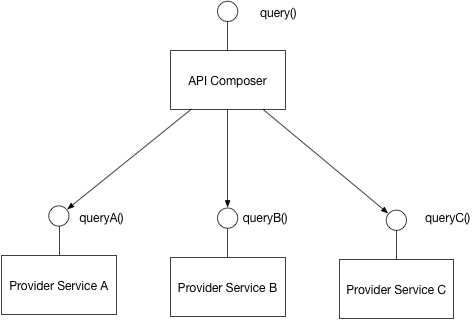
* simple way to query data in a microservice architecture

Drawback:

* Some queries would result in inefficient, in-memory joins of large datasets
* Solution may be using CQRS for segregate reading data from writing.

Note:

* API Gateway apply API Composer Design Pattern [Single Entry Point to access services]

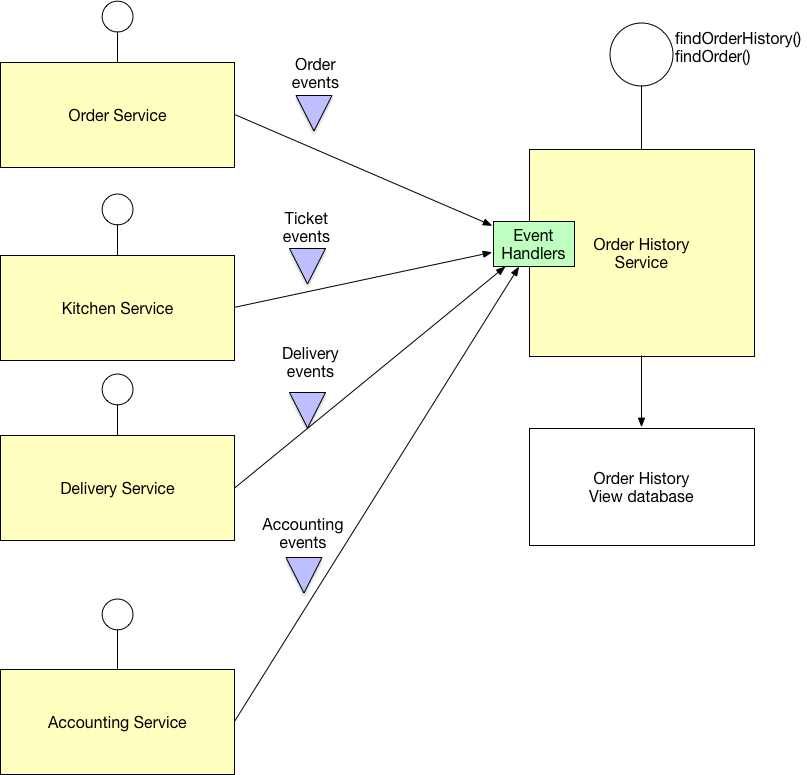


Problem:

* What is CQRS ?
* How to implement a query that retrieves data from multiple services in a microservice architecture?

Solution:

* Command Query Responsibility Segregation
* Define read-only Database replica to support queries only
* Keep replica data up to date using DDD aggregation [Domain event] subscription, [publish events when they created or updated].



**Data integrity is the overall accuracy, completeness, and consistency of data**

Problem:

* What is two-phase commit (2PC) ?

Solution:

* standardized protocol that ensures atomicity, consistency, isolation and durability (ACID) of a transaction
* For distributed System [multiple Databases], It is an atomic commitment protocol
* The database must store a copy of the transaction’s updates before a failure occurs. 2PC ensures that each system accessed by a transaction durably stores its portion of the transaction’s updates before the transaction commits anywhere.

Problem:

* How does two-phase commit work?

Solution:

* special object, known as a coordinator, is required -> arranging activities and synchronizations between distributed servers.
* consists of two phases

Phase 1:

* every resource manager stores their updates in log storage (Every server that is required to commit writes its data records in a log)
* Initiating node requests for all participating nodes
* Waiting response (prepared, Read-only, Abort)
* Modifying data itself -> lock is then placed on the modified tables to prevent them from being read.
* Prepared nodes then wait for either a commit or rollback response from the global coordinator [in-doubt status]

Phase 2:

* tells all resource managers to commit or rollback all if any failure that all commits is stored in logs before committing.

Problem:

* Compare between 2PC vs Saga ?

Solution:

* same goal: to coordinate resources while overlaying operations form a coherent unit of work. But they are using different approaches.
* 2PC: allows programmer to execute transaction as one unit in distributed system by committing or rollback all. It is already used in Most of DB engines in local DBs and may be in distributed DBs as well. No need to program it.
* Saga: allow programmer to execute transaction as one unit in distributed System but allowing execute it in period of time [It is not required immediate exaction]

Problem:

What is Domain-Driven Design (DDD) ?

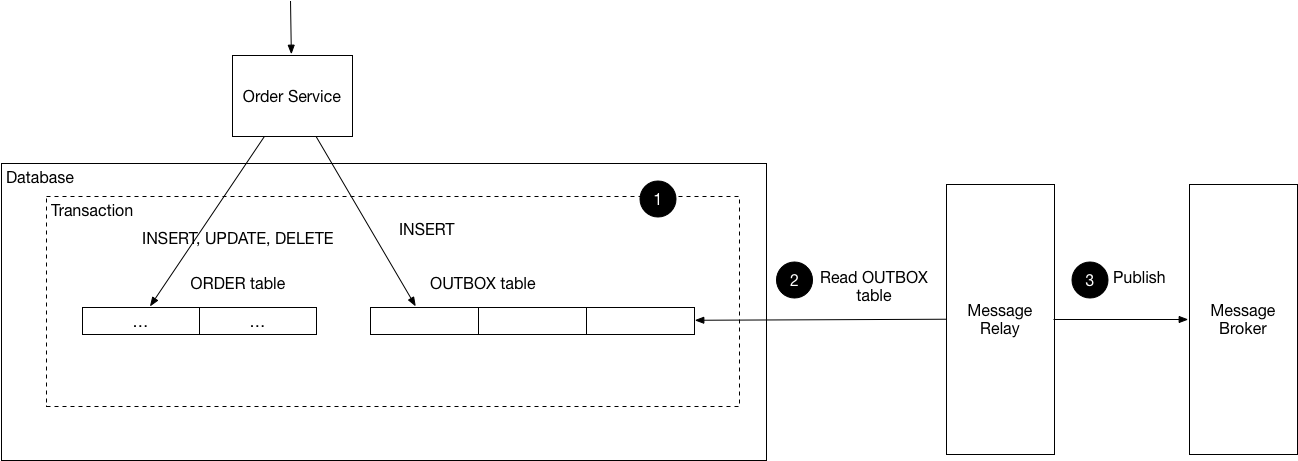
Solution:

* Service publish events when it updates its data
* Organize the business logic of a service as a collection of DDD [aggregates](https://microservices.io/patterns/data/aggregate.html) that emit domain events when they created or updated
* Service publishes these domain events so that they can be consumed by other services

Problem:

* What is Transactional outbox ?
* How to reliably/atomically update the database and send messages/events?
* 2PC is not an option. Message broker may not support 2PC. If does, It is not good practicing to couple between message and transaction. Transaction may be executed and response message craches.

Solution:



* service that uses a **relational** database inserts messages/events into an **outbox table, NoSQL** DB we can use **attribute** in record [Document or Item] being updated.
* Message Relay Process can publish the events inserted in DB to message broker.
* It is useful with **Saga** and **Domain Event** design patterns.
* **Event-Sourcing** design pattern is alternative solution for same target.

Drawback:

* Developer may forget publishing the message after updating DB.
* Message Relay may send the message more than one time. It might happen, after sending the message, system may be crashed before recording message is published.

[This point is not actually issue because it is resolved by default ☺]

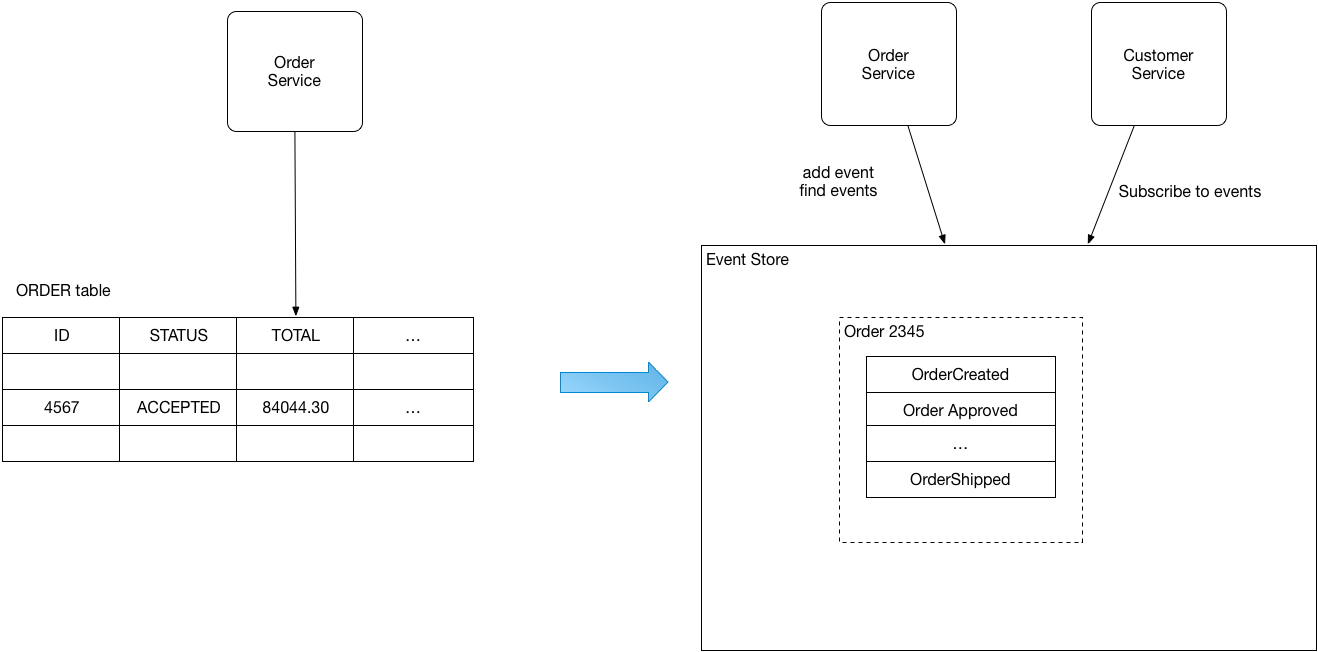
message consumer must be idempotent, perhaps by tracking the IDs of the messages.

Actually any Message Consumers need to be idempotent(handling multiple calls for same message) because any most of message broker support deliver messages more than once.

Problem:

* What is Event Sourcing Design Pattern?

Solution:



* 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 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
* Customer, can have a large number of events, so an application can periodically save a snapshot of an entity’s current state

Advantages:

* It resolves the problem of reliable publish events whenever changing the event’s state.
* It saves events instead of object so no object-relational mismatch problem.
* 100% reliable audit log of business entity changes.
* Possible to make queries to determine the state of business entities at any point of time.
* loosely coupled business entities that exchange events

Drawbacks:

* unfamiliar style of programming and so there is a learning curve
* event store is difficult to query so may use CQRS for queries ☺

Simplify Design Needs ☺

|  |  |  |
| --- | --- | --- |
| Maintain data consistency | Implement queries | Communicate using transactional messaging |
| **Saga** | **CQRS** | **Transactional Outbox**  or  **Event Sourcing** |