MICROSERVICES & ITS PATTERNS

On a high level, there are two architecture styles available:

1) Monolithic
2) Microservices

⇒ Monolithic (also kla legacy)

In this, we put all the functionality in a single application. For eg: for an e-commerce applic^{n}, all functionalities like cart, inventory, booking, payment etc. will be put in a single application.

Disadvantage:

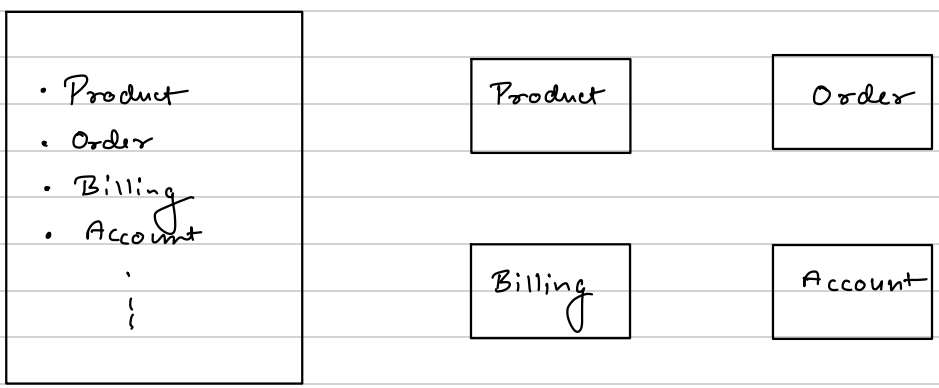
1) Overload IDE: Since everything is present in a single codebase, the size of it becomes huge & development using IDE is very tough.

2) Scaling Is Hard: Scaling is very tough in a monolith application as everything is tightly coupled & testing + debugging takes a lot of time. Deployment & regression is again tough. Also scaling a specific part (let's say ordering in e-commerce app) is not possible so we'd

have to scale the whole application which again is costly.

⇒ Microservices :

In this, the whole application is divided into multiple smaller services & they communicate with each other to achieve the functionalities.



* Advantages

1) All the disadvantages of monoliths are fixed in micro-services i.e easy to scale, managable code base, easy to scale specific parts.

* Disadvantages :

1) If microservices are not loosely coupled, then communic^n blw services will become difficult & will increase latency. So decompos^n of monolith to microservices should be done carefully.

2) Deployments should be done in a planned way & monitoring is tough as change in one microservice can break others. So debugging also becomes tough as it'll include multiple services.

3) Transaction Management : Since each microservice has its own dB, so managing transaction is difficult. In case of any failure, transactions need to be rolled back for multiple services.

⇒ How should we decompose a monolith into microservices or creating new ones - These are a few phases of microservices.

1) Decomposition
   - Decompose by business capability.
   - Decompose by subdomain etc.

2) Database
   - Individual database per service
   - Shared database b/w services etc.

3) Communication
   - API based communication
   - Event based communication etc.

4) Integration
   - API Gateway etc.

5) Observability
   - Monitoring etc.

So these are a few patterns & we've to choose one for each phase while designing microservices.

Let's now discuss these patterns:

1) Decomposition Patterns

There are primarily three types of decompos" patterns:

a) Decomposition by business capability:

In this we divide the microservices by the business functionalities.

For. eg:- In E-commerce application we can divide like this:

- Order Service : Handle orders
- Product Service : Handle product related functions
- Login Service : Handle logins
- Billing Service : Handle billing

⋮

and so on.

For this, having prior knowledge of business functionalities is a must.

b) Decomposition by subdomain:

In this kla Domain Driven Design (DDD). we divide the services on the basis of domain. A domain can have multiple microservices. For eg: Order Management - It can have order, product, inventory etc. Payment Domain - It can have payments, billing services etc.

c) Decomposition by transaction:

This is an appropriate pattern for many transactional operations across multiple components or services. We can choose this option when there are strict consistency requirements. For eg: Consider cases when an insurance claim is submitted The claim request might interact with both a Customers app" & Claims microservices at the same time.

2) Strangler Pattern

This is used whenever we're refactoring a monolith to microservices. We use this design pattern to incrementally transform a monolith application to microservices. This is accomplished by replacing old functionality with a new service - and, consequently - this is how the pattern receives its name. Once the new service is ready to be executed, the old service is "strangled" so the new one can take over.

To accomplish this successful transfer, a facade interface is used by developers that allows them to expose individual services and functions. The targeted functions are broken free from the monolith so they can be "strangled" and replaced.

3) Database Per Service Pattern

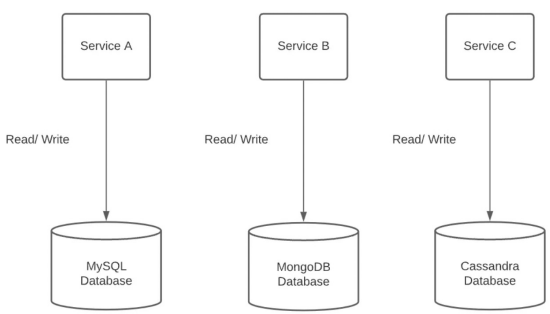
Database organization will affect the efficiency and complexity of the app". The most common options that we can use when determining the organizational architecture of an app" are:

a) Dedicated Database for each service :

In this each service has its own database. It makes much easier to scale & understand from a whole end-to-end business aspect. It is also helpful in cases where the databases have different needs or access requirements.

It also reduces coupling as one service can't tie itself to the tables of other. Services are forced to communicate via published interfaces.

It's downside is that dedicated databases require a failure protection mechanism for event where communication fails.



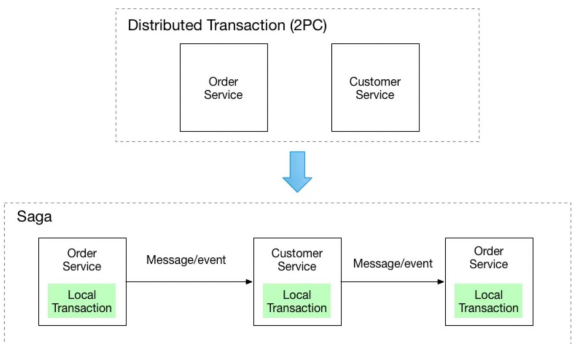
b) Single Database Shared by all services :

It isn't the standard for microservices architecture but may be appropriate in some situations. In those cases as well, it's very important to enforce logical boundaries within the database.

Its disadvantages is that we lose the key benefits like scalability, robustness and independence.

4) SAGA Pattern :

A saga is a series of local transactions. Saga pattern helps in maintaining data consistency during distributed transactions.



So, in this pattern, events are sent from one service to another effectively combining various local transaction. If one of the local transaction fails, compensatory events can be sent to rollback the changes of other local transactions.

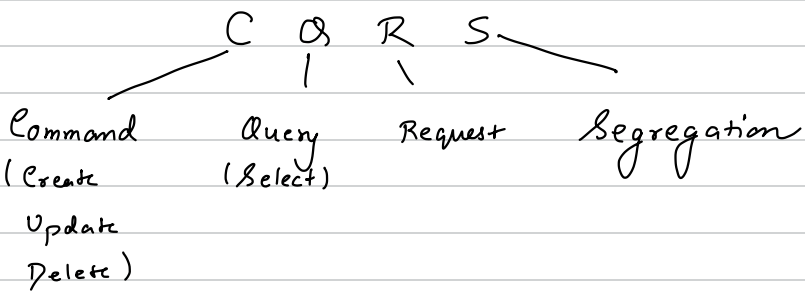
There are primarily two approaches to implement the Saga pattern:

a) Choreography : In this approach, a service will perform a transaction and then publish an event. In some instances, other services will respond to those published events & perform tasks according to presets. Basically in this, one service performs some task, will publish an event, then other services listens to it & do their work.

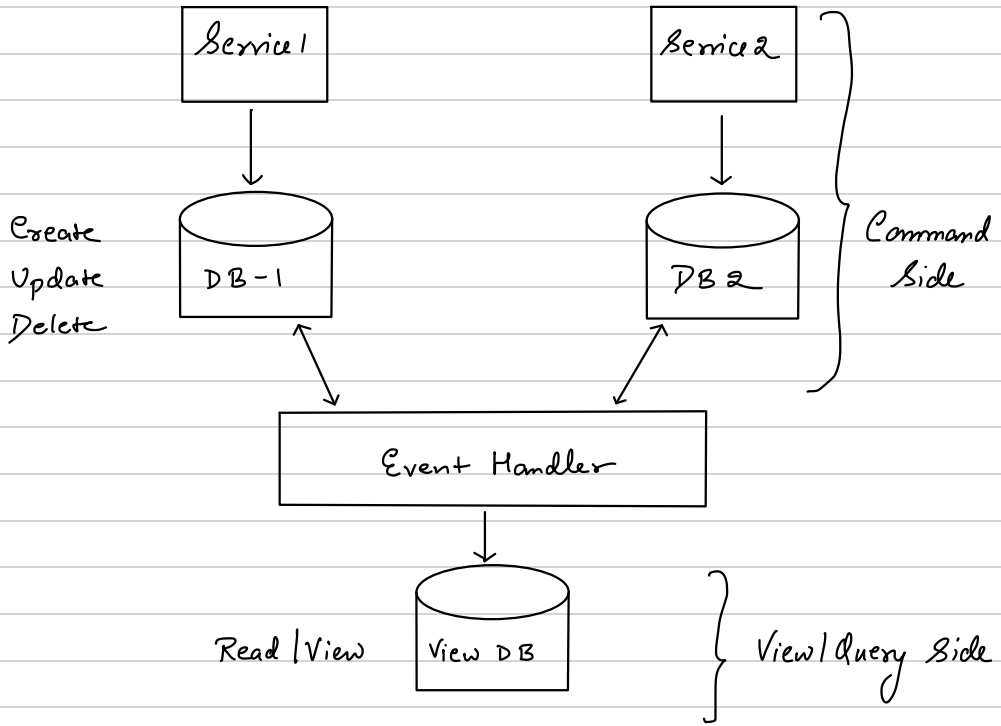
In this, a demerit is that a cycle may form b/w 2 services i.e first perform the task, publish an event, second listen to that event & perform the task & publish the event which then first listens and so on.

b) Orchestration: An orchestration approach will perform transactions and publish events using an object to orchestrate the events, triggering other services to respond by completing their tasks. The orchestrator tells the participants what local transactions to execute. So the responsibility is of the orchestrator to handle & control the tasks.

5) CQRS: CQRS stands for



In CQRS pattern as the name suggests we split/segregate the application in two parts – the command side & the query side. The command side handles the Create, Update & Delete requests. The query side handles the query part by using the materialized views. To connect both we can use DB Triggers/ procedures/events but the event sourcing pattern is generally used to create events for any data change. The query side are kept updated by subscribing to the stream of events.



6) Circuit Breaker Pattern: This pattern is applied usually blw services that are communicating synchronously. A service generally calls other services to retrieve data, and there is a chance that the downstream service may be down. So, the request will keep going to the down service, exhausting network resources & slowing performance.

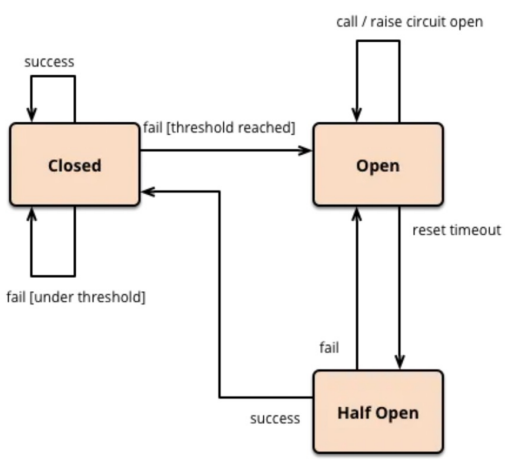
Circuit Breaker Pattern solves this issue. In this, an object is used to monitor failure conditions. When a failure condition is detected (let's say x no. of downstream calls failed), the circuit breaker will trip. Once this has been tripped, all calls to circuit breaker will result in error and be directed to a different service.

So, we can say there are 3 states of Circuit Breaker :

a) Closed : It's the default state & all calls are responded to normally. This is the ideal state we want a circuit breaker microservice to remain in.

b) Open : A circuit breaker pattern is open when the no. of failures has exceeded the threshold. When in this state, the microservice gives errors for the calls without executing the desired function.

c) Half Open : When a circuit breaker is checking for underlying problems, it remains in a half-open state. Some calls may be responded to normally, but some may not be. It depends on why the circuit breaker switched to this state initially.



7) API Gateway Pattern : It is a good option for large applications with multiple clients. It's biggest benefit is that it insulates the client from needing to know how the services are partitioned.

It basically grants a single entry point for a group of microservices by working as a reverse proxy b/w client apps and the services.

The client also doesn't need to know how to find or communicate with a multitude of ever changing services.

We can also create gateways for specific types of clients (eg. backends for frontends).

It also can take care of crucial tasks like authentic", SSL Termination and caching.