Micro Services design pattern

1. Gateway Pattern / API Gateway Pattern/ 2) API Composition Pattern

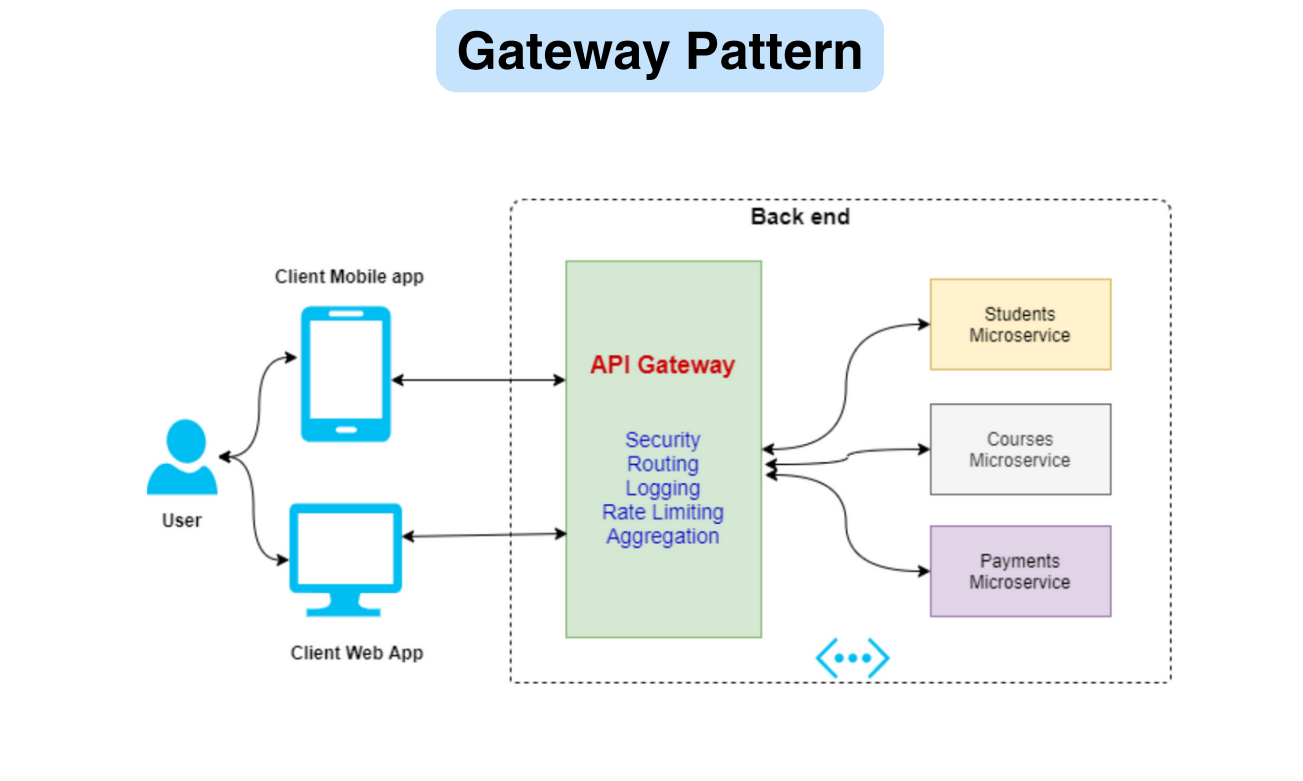
Imagine an Amazon site (product service run 8080, Rating 9090, Order 7070). To place an order, you would need to check rating, product info and then place order

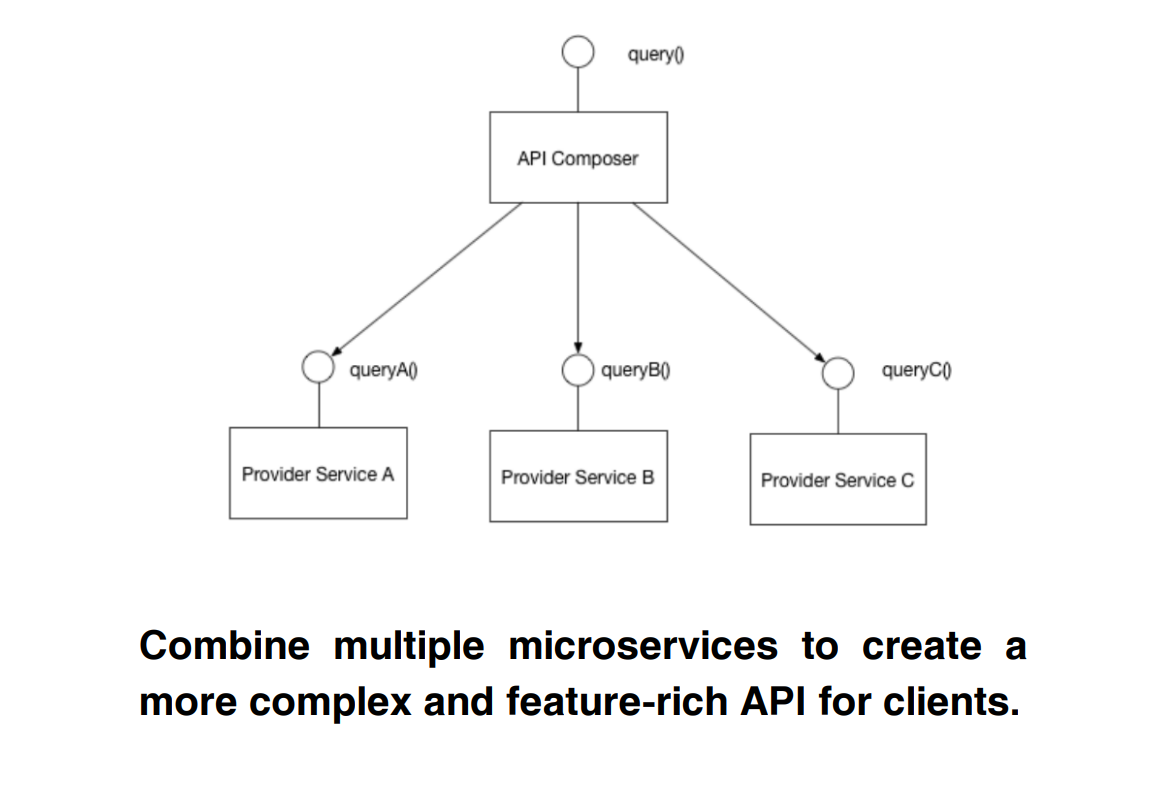
So ideally client or user need to call each service independently

This is bad user experience

To solve this, we have an API Gateway Pattern

* It **reduces # of direct call** that can happen between user and each service. you just make a call from client 🡪API GATEWAY🡪(internally will communicate with services) 🡪API GATEWAY will give you response
* API gateway is a common entry point and this point will internally communicate with all services that may be in the same network (Saving time)
* Without this API gateway, to get any information from any services you always need to authenticate
* So, **API gateway has the responsibility of Authentication too (first call would be HTTPS then all other call with diff services will be HTTP or RPC or whatever)**
* **API gateway also has the information of how many instances of each service are running so it can automatically do the load balancing if needed in case of huge traffic**
* Insulates the clients from how the application is partitioned into microservice. Insulates the clients from the problem of determining the locations of service instances
* Provides the optimal API for each client
* **You can have API gateway one for web, desktop and client so that whoever request the info according how detailed info need to be served can be decided. (With pagination for cell, without for desktop) 🡪 called BACK IN FOR FRONT END GATEWAY**
* **Implement: download gateway dependencies and if all of your microservices project running on Eureka server you can enable to search so that depending upon the request you can be redirected to that particular microservices**
* **You can aggregate the response from each service and combine them and return as a response to user. So, no need of aggregation complexity on Front end side**

****



1. Service Registry Pattern/ Service Discovery

We have a Load Balancer and let say we have 2 microservices Service A and service B

Now assume Service A want to communicate with Service B

How will they communicate? They should know server name and Port address

Hence to keep track of all the information of each services address port number, we have a Service Discovery.

As soon as any instance is newly created or destroyed (in case of autoscaling or not) instance will register itself with Service Discovery. The service discovery could be a fixed IP address so that instance can register/deregister them with it

All the info about IP address port for each service and instance would be present inside a Linked list or HashMap. this is called Service Registry

But if you see then service discovery becomes really important component and failure of it can stop communication between microservices

So how does it happen?

Micro services A wants to communicate with Microservices B

A-> connect to Load balancer (here info about service registry is cached)

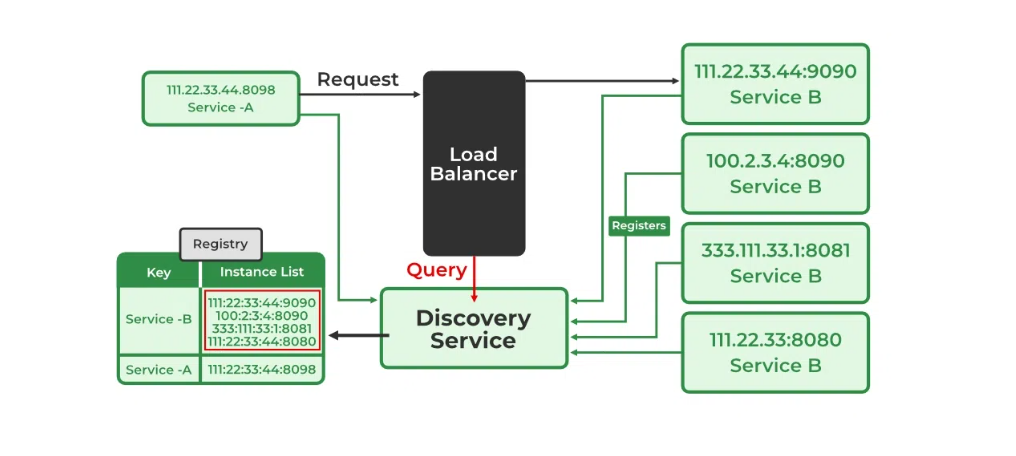
At first If B info is not present, LB 🡪 Service discovery -> Get info and cache it -> Now if LB says n instance of B present -> then LB will pass the request to instance with low load

So responsibility of Load Balancer and Service Discovery is different

Implementation wise : you have annotation of @@EnableEurekaServer

Which will allow you to register on Service discovery

It can be done from Client side and Server Side as well



**Types of Service Discovery**

There are two types of Service Discovery

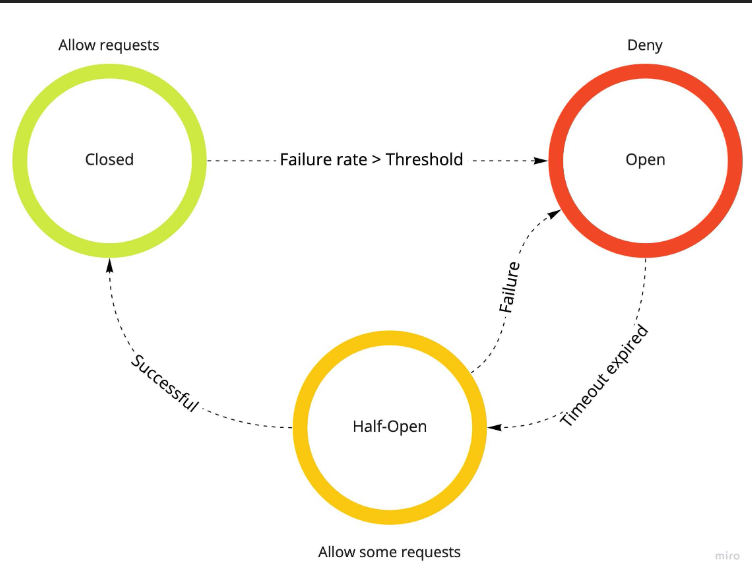
* Client-Side Service Discovery
* Server-Side Service Discovery

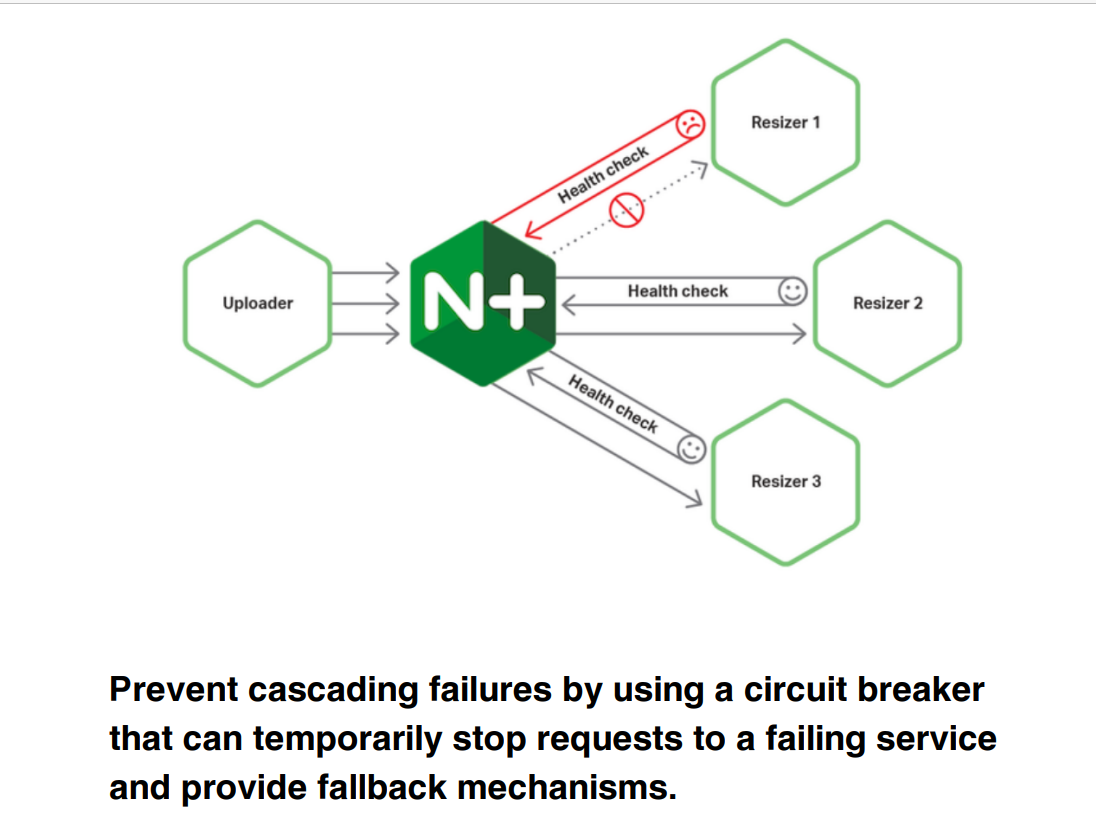
**Point to Remember:**

* **Client-Side Service Discovery Example**: Netflix Eureka, Zookeeper, Consul
* **Server-Side Service Discovery Example**: NGNIX, AWS ELB

1. Circuit Breaker Pattern

* This pattern is use when microservice A is synchronously waiting response from microservices B
* Assume for any reason (network or Database down) B get down, then A is waiting causes multiple resources to starve (thread memory and connection to stay alive or block for long time)
* In order to avoid it we have a circuit break pattern. @EnableCircuitBreaker, it has 2 parameters name 1) name of it 2) The fallback method
* The fallback method should return same type on which Circuit breaker method is called. It can return dummy response
* There are 3 states involved in Circuit Breaker pattern 1) CLOSED 2) OPEN 3) HALF CLOSE
* CLOSED is when for a request you are getting response
* Transition happens from CLOSED to OPEN happen when failure rate > Threshold
* Open to half open happen when we wait to send a response in some time
* If even after time expired, we are unable to serve we go to open state else it will go to the Closed state
* The below diagram transition is useful to understand the behaviour of it





# SAGA Pattern /How to do Distributed Transactions the RIGHT way? OR(6) EVEN DRIVEN ARCHITECTURE

We know that in a microservices, each services have its own DB. Let say Order and Transaction are 2 microservices. When an order is placed, we have to make sure that we need to have an entry in order database and payment records is persisted

So, both the database should be updated then in order to have system check and heath intact.

One can argue to achieve distributed transaction by using 2 Phase Commit (2PC). In a 2Phase commit we ensure that each database commit happen then only we say TRUE COMMIT else it would be a rollback. In the below diagram step1, 2 -> VOTE PHASE step 3 and 4 -> Commit phase

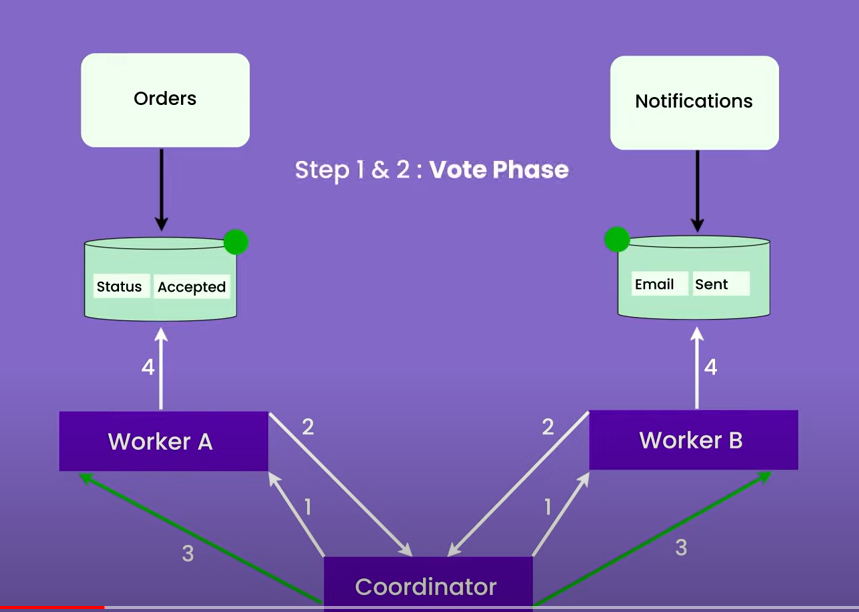
In the above diagram coordinator ask worker thread of each DB, are they ready to change the state of their respectively database. If yes. the coordinator asks worker thread to lock tables

This is called Voting phase

Once the tables are lock, the thread applies the changes and commit the changes in the database rows. This is called Commit phase

But locking a table or rows when we have multiple billion of transaction updating at same time may lead to inconsistency, high latency and finally bad user experience

So ideally, we shouldn’t go for 2 Phase commit in such a large system

 and step 3, 4 -> COMMIT PHASE

So here what we can do is we follow Forward Strategy or Backward Strategy or Combination of Both

What is Forward Strategy

Order placed 🡪 Payment -> Tracking -> Delivery Notification

Here if we move in a forward back (commit one by one ). any failure in any steps will be undoing it and send Failed delivery Notification

What is Backward Strategy

Here if we move in a forward back (commit one by one). any failure in any steps will be like undoing all the steps from failed state side to start phase and send Failed delivery Notification

To implement a SAGA pattern, we can follow 2 strategies

* 1. Orchestration Strategy
  2. Choreography Strategy

In Orchestration Strategy: here we have controller who will give the request to each module

And depending upon the response of each module, the subsequent request will go to next module

SO here we have some controller who will place order if success then payment is asked then success then tracking info is updated and then if success then delivery notification

In the below diagram User is orchestrator

User 🡪 Course (success) 🡪 Payment (success) -> Notification (success)

Now next is Choreography strategy:

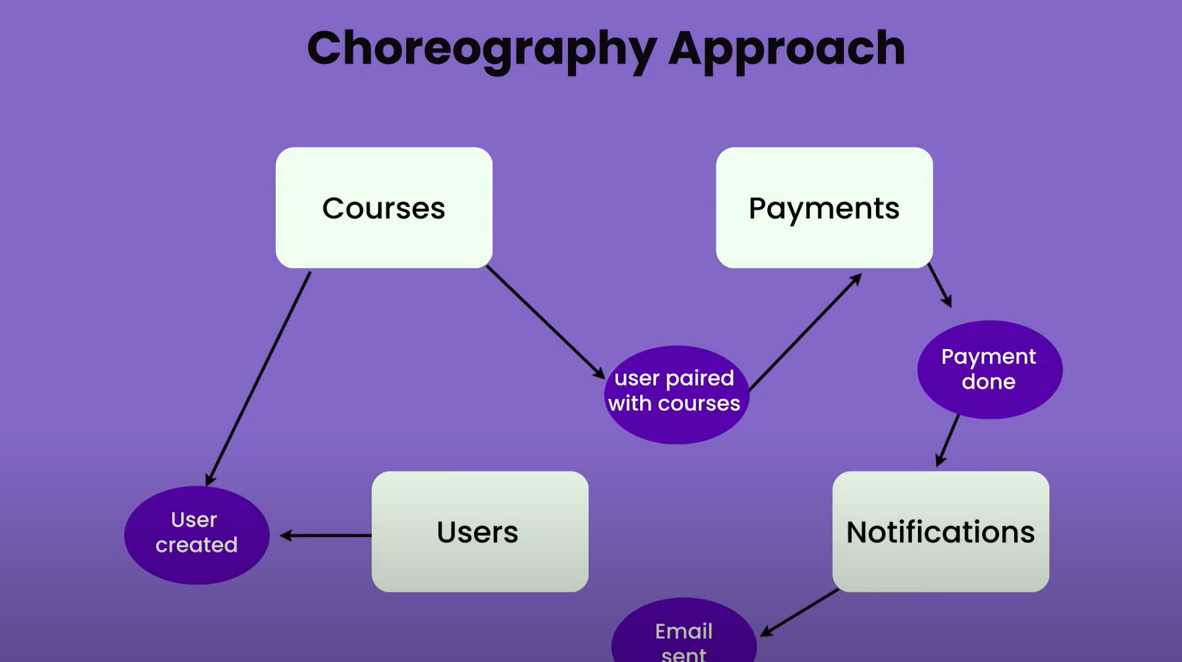
It’s a complete event based (Asynchronous)

Here its like if user place order, order services will publish a message “ORDER\_CREATED”

Payment services will listen to each message and ask for payment from user and then published “PAYEMENT\_DONE” once successfully receive it

Similarly tracking and notification will listen to diff message topic and take action based on it

Here no one is controller everyone take action depending upon the message they get whole operation E2E done



* 1. CQRS
* It stands for Command Query Responsibility Segregation

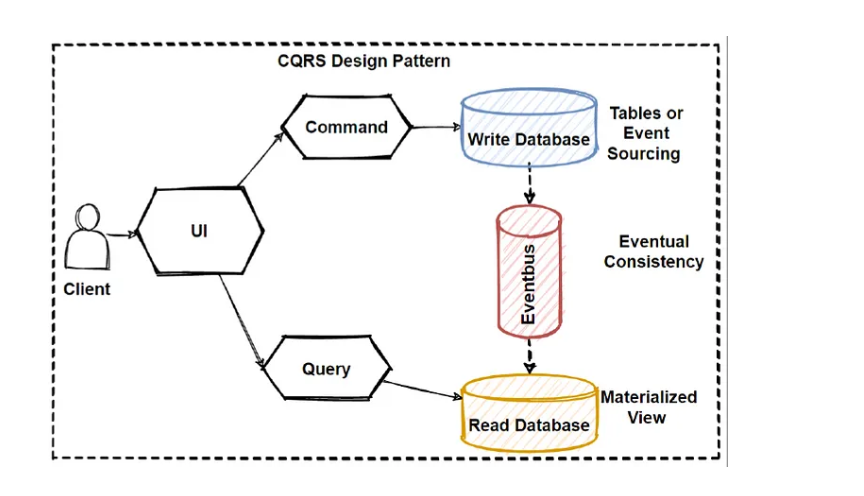
Consider a scenario where a request gets a response which join 10 tables 🡪 all the tables are lock. Meanwhile any write operation on these tables won’t be allowed or other way around. This will make the system unavailable, HIGH latency and bad user experience

Hence a line or separation of concerns should happen in case of Command (any update using put. post. delete, patch) and query (get) command

In case of commands, you can opt to use RDBMS but in case of Query you can use no SQL such as Cassandra

So here we will maintain 2 databases write operation can happen on RDMS and query on no SQL. Both the database can achieve consistency using Kafka (Eventual consistency)

Instagram use the above pattern



* 1. BULKHEAD PATTERN

Bulk ahead pattern is similar to Circuit Break pattern

It got it name form Shipping business where each compartment isolated from the other. So that if anyone of them get flooded other don’t get flooded too

Similar when it comes to Microservices, we don’t want failure or slowness of one component may affect others too

Let say we have a thread pool of 900. And we have three services service A, B and C

For some reason service A is slow. so, all the threads which has request for A may get blocked while thread for B and C will return with faster response,

If the slowness still continues at some point, it will happen that majority of threads get blocked with A leading to slowness in B and C

Hence in order to avoid that we have Bulk ahead pattern

Here we divide the thread pool among 3 services. Let say 900 threads, we can divided for A (300) B (300) C (300) . max per allowed services is 300. If 301request came for any service it has to wait for some time (which is defined in the configuration) if that time passes it will give failure response

Implementation wise it similar like circuit breaker

We have @BulkAheadOperation with name and fall-back method

For each service we will define

1. Max Concurrent Threads/Request allowed
2. Time : in millseconds : Any extra request will wait for this time . If it get unblocked its good else it will return with failure response which is defined in fallback method
3. Even you can implement via Conditional Sempahore
   1. SIDE CAR Pattern

Consider a scenario where we have a system which used to handle all upcoming HTTP traffic

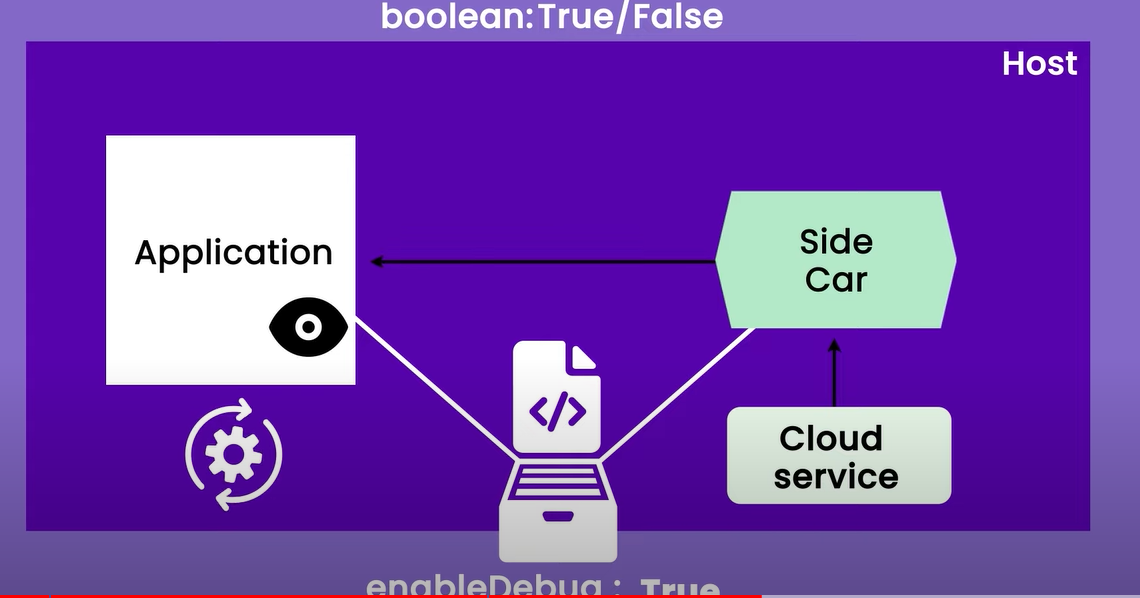
With growing demands and need, it has been decided to move from HHTP to HTTPS traffic

There is multiple option

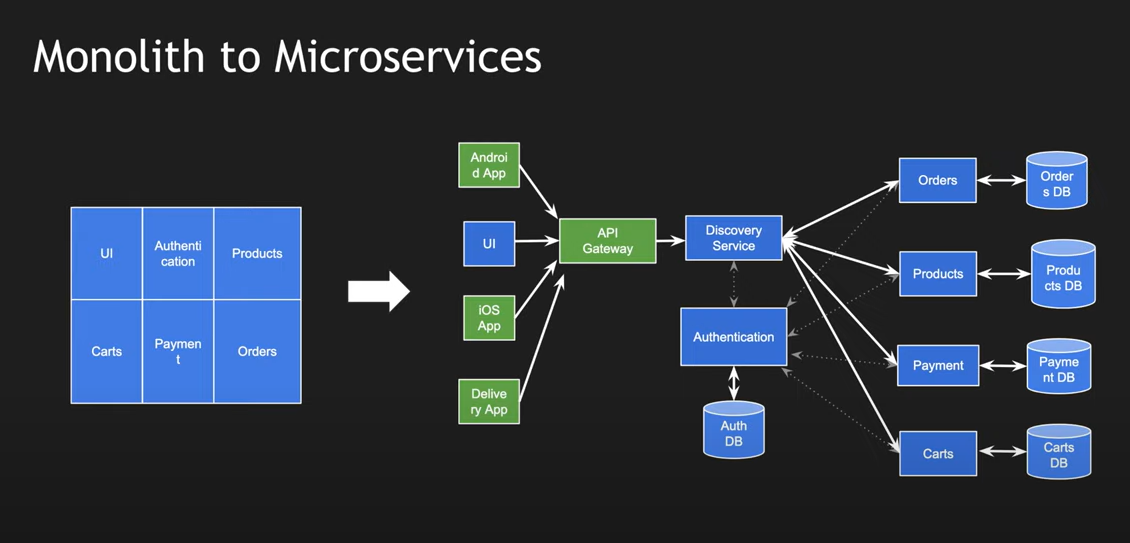
* One is to rewrite the application once again but its time consuming
* Second is to have side component which will take HTTTPS request validate it and internally will use legacy system to handle it. so, side component has the responsibility of taking HTTPS traffic and convert it into HTTP and forward it to legacy one
* **But point to be noted that both components (legacy and side component) can be language independent but communication among then must be lang independent. Both need to be deployed on same machine sharing same memory and network**

Another example is CCM (Application read values from CCM). if we need to change any configuration one shouldn’t not change in code because it will require whole application deployment. Rather change in configuration (static file) should happen via CCM (side car)

Once value is changed side car may send a sync mgs’ to application which will get restart on its own to read updated value from the configuration file



* 1. Database Per Service Pattern 🡪 self-explanatory
  2. Retry Pattern ->self-explanatory @Retry annotation will be used. Configuration are max-attempts and Wait for Duration in milliseconds
  3. Configuration Externalization Pattern -> (self-explanatory) Here the config need to be kept externally at third party product, so that same code can run in DEV , QA , perf and Prod 🡪 CCM
  4. Strangler Flag Pattern
* This pattern is used when we are migrating from Monolith to Microservices



* 1. Leader Election Pattern
* This pattern is used when system want to recover from failure. It as an automated way of System Recovery

We have 3 diff algo for the same

1. LCR algorithm (n^2)

* Here 3 assumptions are
* 1) Every node will have unique UUID
* 2) UUID are comparable
* 3) Every node knows its right-hand side node (imagine a circular ring with multiple nodes)
* Imagine a scenario that suppose leader goes down and whoever detect it would send a **“synchronous” message that election will happen**
* Each node will vote by sending its own UUID to its RHS node
* Now each node on receiving mgs will check if own UUID > Received UUI it will discard received one and forward it own
* In case id its own UUID < Received UUID it will discard its own msg and forward the received vote
* There will be a single node who will received mgs’ where its own UUID= received UUID=> this node is a new a Leader
* Now this new leader need to send HALT msg to its RHS node and so on to inform that new leader is elected

1. HS Algorithm (nlogn)

* It is similar as above here node knows both of its immediate LHS and RHS node
* The leader is selected in phase wise
* In each phase I , we send message to 2^I nodes
* A node sends its UUID to LHS and RHS

Here if node UUID is less than received its get eliminated but would still replay message to immediate nodes

Here if node UUI is greater than received one, it won’t send ACK back. Hence the source node would get eliminated

And at time if node received its own votes 🡪 it seems node is a leader

So in every phase you are moving from Local Leader to global Leader

1. Time Slice Leader Election Algorithm

Let say we have nodes 3, 4, 5 6, 7, 8, 9,

In phase 1 node with 1 is send, but since we don’t have node 1 so phase expired

In phase 2 node with 2 is send, but since we don’t have node 2 so phase expired

In phase 3 node with 3 is send, but since we have node 3 so phase stops and least node 3 is elected as leader

Now node 3 will send mgs’ to every other node that I am the leader

Its very slow and highly impractical algo used