Microservices

small, autonomous and harmonic subsystem.

→ It is an architectural pattern that arranges an application as a collection of loosely-coupled, fine-grained services, communicating through lightweight protocol.

→ like you have an massive product, lets say facebook or instagram, that product is broken into subproblems and each problem is then handled by an microservice. And that service is designed in a way to solve that problem in very optimal way.

Why? ① codebase grows over the time.

→ So having to have so-much teams to work on one single code-base

→ It reduce overall developement velocity.

ⅱ Scaling become predictable

→ Only one code base, so if you want to scale something entire monolith is scaled.

→ Each service can scale independly.

ⅲ Autonomous and isolation

→ Own tech stack

→ Contract for communication (HTTP's / TCP). deploy code whenever because of the auto. nature

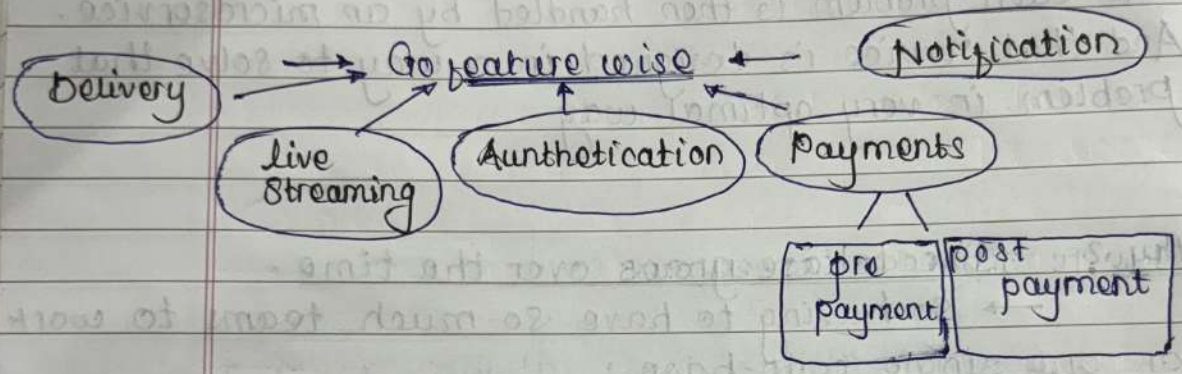
ⅳ Fault tolerance

→ If one service is down, it doesnt effect other

→ partially working product.

(v) Upgrade becomes simpler
→ You can evolve your service ← Tech stack, langauge, DB, Better Infra
→ Can evolve as long as they have same API contract

★ How to force Microservice



■ Concurrency
→ Concurrency refer to ability of system to execute multiple tasks or processes simultaneously
→ It allows different part of system to make progress independtly and in parallel.

→ Concurrency deals with managing multiple task concurrently.
→ parallelism involves executing multiple task simultaneously.

Thread based :- Using thread to acheive concurrency, where multiple thread execute independtly within a single process.

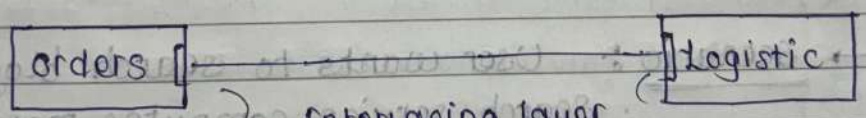
Event driven :- Utilizing events and event loop to handle asynchronous I/O operation and manages concurrent task without blocking.

- **Latency** :-
  → Latency refers to delay between initiation of a task or request and the completion of that task or reciept of response.
  → Time taken by data to travel from source to destination.

2 How to scope Microservices

→ A good micro-service is always designed by keeping the following two-concept in check.

① **Loose coupling** :- Change in one service should not require change in other service.



→ So long as the interfacing layer and API contracts remain the same, change in order service should be transperant to the logistic.

(i) → **How to achieve** A service should know as little as it needs about the other service.

ⓐ public API ✓ ⓑ Rate Limit ✓
ⓒ Authetication ✓ ⓓ communication protocol ✓
ⓕ Database used ✗

* Isolation is one of the key requirement for Microservices.

(ii) High cohesion :- Relative behaviour sits together.

→ Unrelated behaviour sits separetly.

CORE IDEA :- Service should operate independtly.

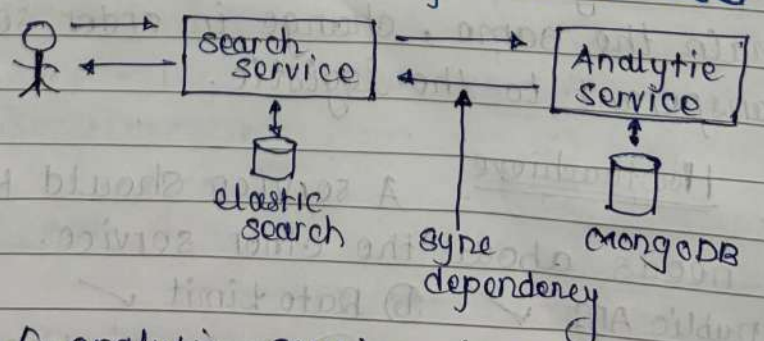
→ whenever you're sharing codebase or database keep in mind that service should be loosely coupled

→ Is communication is robust, transperent enough.

→ Are all the related behaviour / functionality are in one service or not.

3 Handling timeout in Microservice

→ Scenario :- User wants to search blogs. Search service computes most relavant blogs. Then analytic service gets blog views and then return the response to user.



→ If analytic service is not responding

How long we will wait to send the response to user.

what could go wrong ?

(i) Request never reached analytic service ?
(ii) Response never reach search service.
(iii) To long (Analytic service).

→ whenever you make a synchronous call add timeout

(A) Approach 1 :- Ignore.
→ Unpredicted UX
Good practice → catch all the expection
i.e yes timeout happen / DB failed

(B) Approach 2 :- Configure and use defaults.
→ Use default value.

(C) Approach 3 :- Retry
→ Retries are simpler when its an read req.

What if.
→ Request is non-idempotent
→ Request is expensive
→ Other service is overloaded.

Good Practice
Retries with exponential backoff
[1s, 2s, 4s ...]

(D) Approach 4 :- Retry only when needed
→ check operation is successful.
→ Like in some cases you get to know if the req succeful or not.

(E) Approach 5 : Rearchitect
→ Try to make asyn.
→ event driven → Ingest required data.

4] Microservice sharing the database.

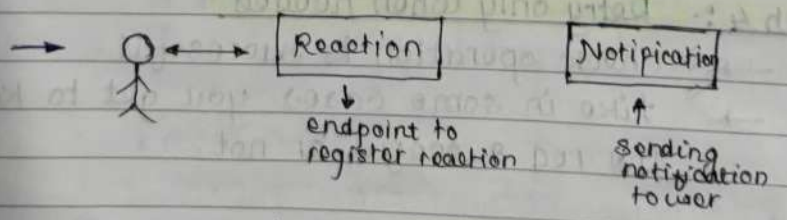
→ Simplest way of integration
→ No latency overhead.
→ quick deployment time.
→ No Middleman.
→ Simpler
→ Better performance.

DisAdvantages.

(i) Challenge 1 :- External parties getting internal access
(ii) Challenge 2 :- Sharing Bussiness logic
(iii) Challenge 3 :- Risk of data correction or deletion.
(iv) Challenge 4 :- Abusing the database.
(heavy computing query will bring load on DB).

① Sharing a DB quick solution, Crunched time.
② when your schema doesnt change often.
③ Read load can be moved to Replica.

5] Syne and Asyne communication between microservice.

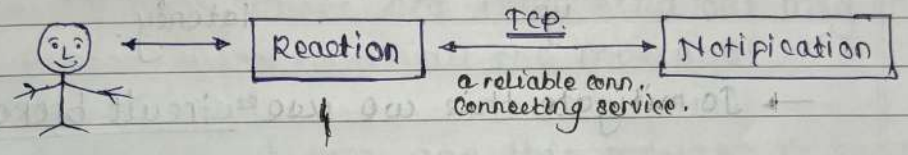


→ Say we are building a social network where we have to notify user A when User B reacts to a post

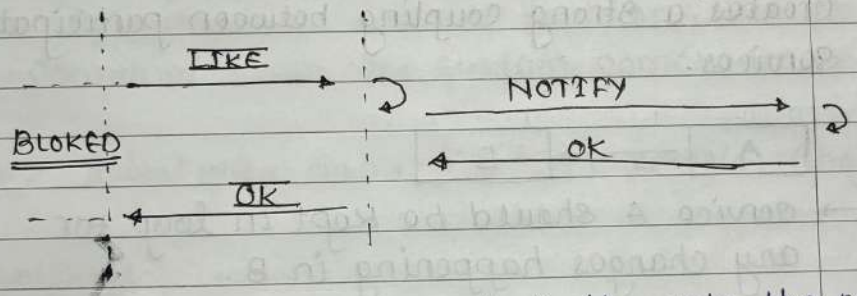
→ How Reaction service talks with Notification service for sending a notification

* In Monolith :- Its simple, its just a function call away.

(A) Synchronous communication



→ send a request and wait for other service to send res.



Synchronous call user definetly gets the notification and also process will be complete. [Blocking till we get res]

Actual Implementation

(i) REST based communication
(ii) GRAPHQL
(iii) GRPC

} HTTP } RAW TCP.

Advantages

(i) communication happen realtime
(ii) simple.

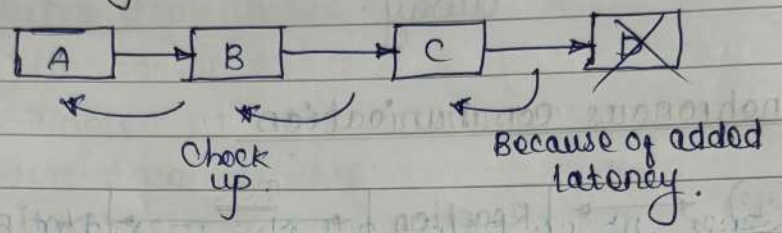
Disadvantages.

(i) caller is blocked untill response is recieved. (timeout)
(ii) server needs to be pro-actively provisioned for peak.

→ In case of synchronous req always ensure enough server else your latency is spikes.

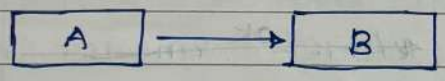
→ always think that you have tested for peak.

→ Cascading failure



→ To mitigate this we use "circuit brekers"

③ Creates a strong coupling between participating services.



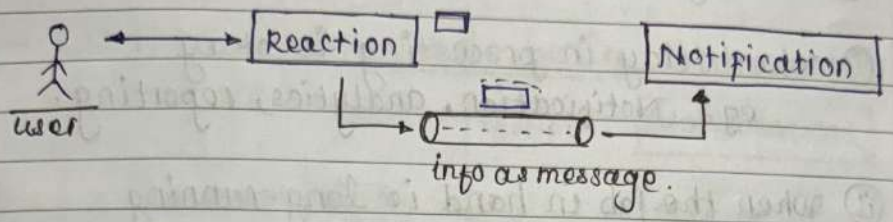
→ service A should be kept in loop for any changes happening in B.

→ say you are changing
  ⓐ versioning
  ⓑ contract (API)
  ⓒ Backward compatibility

⟹ Why / Where.

(i) when you can't move on (you need result before move on). DB query.
(ii) when you want realtime response. [chat checkout]
(iii) when it takes relatively less time to compute or response.

B] Ansynchronous Communication



→ whenever two service talk to each other instead of invoking each others API they send out msg in broker
→ Buffer which holds all the message.

→ Message to broker and this message is consumed by notification service.
→ If Notification service is down, the message is consumed when the system comes online.

Ex :- Rabbit MQ, SQS, KAFKA, Kinesis, Google pubsub.

Advantages.

(i) Service dont need to wait for the response.
(ii) System can handle serge and spike better.
(iii) No need to pro-actively scaled. (Only scale when there is large number of messages). 4+ no of consumer.
(iv) No load balancer required, so no additional network hop
(v) No request drop or data loss.
(vi) Better control over failure.
(vii) Services are truly decoupled.

DisAdvantages

(i) Eventual consistency :- Real time Not eventual consistency
(ii) Broker is backbone :- horizontally scale.
(iii) Harder to track communication

When / Why

(i) when delay in processing is okay
eg:- Notification, analytics, reporting.

(ii) when the job in hand is long-running
eg:- provisioning server, order tracking, DB backups.

(iii) When multiple service need to 'react' to same event

blog → Notify user follower
→ Index search
→ update user analytics.

(iv) when it is OK for you to allow failures and retries
eg:- send notification, retries if fail.

8] Designing workflow in Microservices

Say we are building an e-commerce website, and whenever a user purchase something, we have to

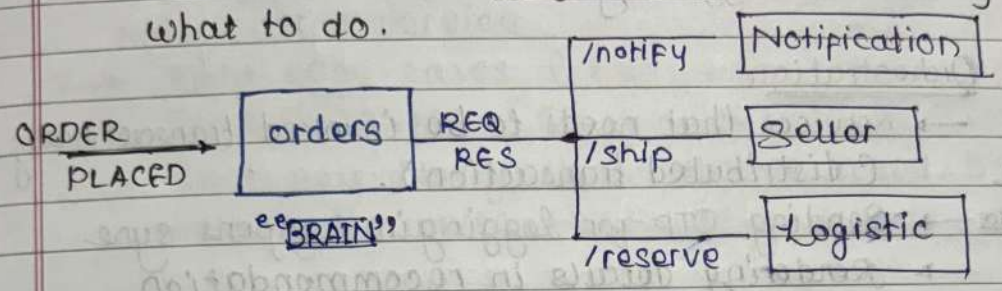
(i) Send an email confirmation to the user.
(ii) Notify the seller to keep the shipment ready.
(iii) Find and assign a logistic delivery partner.

Two architectural pattern.

A] ORCHESTRATION
B] CHOREOGRAPHY

A] Orchestration :- decision logic should be centralized.

→ Let there be an single brain that exactly tells what to do.

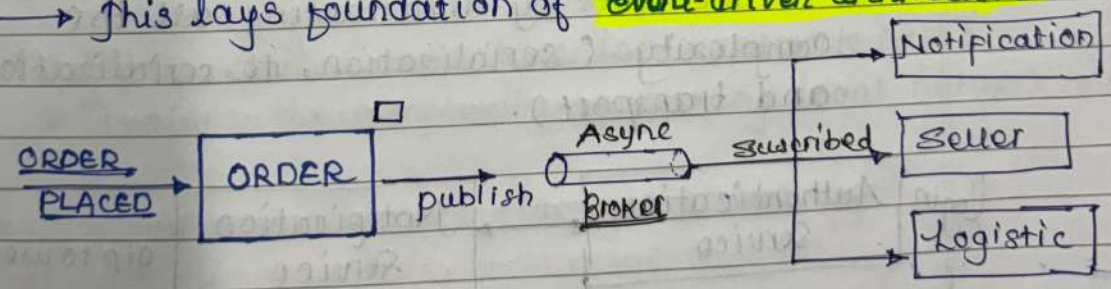


- All the three call need not to be one after the other.

B] Choreography :- decision logic is decentralized.

→ Instead of having a single brain, let each service be independent to think what needs to be done upon getting to know what happened.

→ This lays foundation of Event-driven-architecture



→ WHEN / WHY / WHICH to use ?

- Most modern system uses choreography
- Because of the loose coupling
- Service (each) has its own brain.

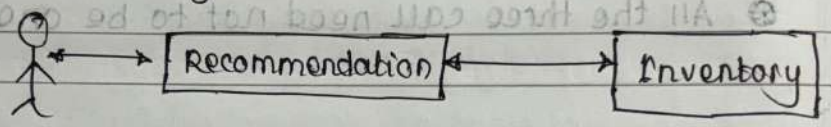
distributed tracing

But in choreography we need to
→ Implement complex observability and track whats happening.

Orchestration
→ Services that needs to be invoked transactionally (distributed transaction).

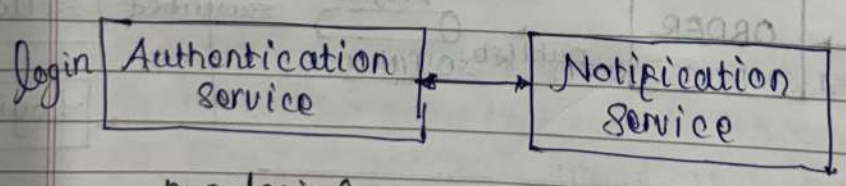
Auth service
↓ sync service
Notificat^n
↓
user

→ Sending OTP for logging in happens sync
→ Rendering details in recommendation requires a sync. communication.



7] Remote Procedure Calls (RPC)

→ Inter-service communication over the network
→ Design to make network call look just like network call.
→ It does this by abrastacting all the complexity (serialization, de-serialization and transport).



pun login (username/email, password):
——
——

notification (email) →
sent a email OTP

How you implement this ?

→ Through REST Api call to api exposed by notification service
→ Right edge-cases (Retry, if server is down).
→ Same thing for (50 services which needs OTP).
→ Lets say Notification service need OTP so we have to write HTTP call.

→ Why do every service need to implement this.

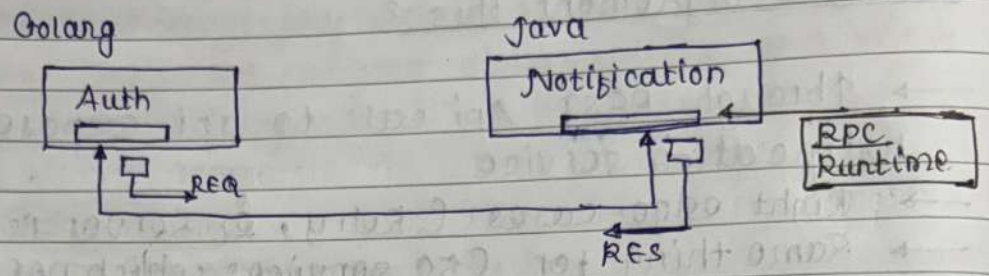
- What if we ABSTRACT out the repetitive and mundane task. like communication protocol, obj creation, failure, retries, compression, streaming. (one soln) and help us the developer to focus on the business logic.

→ So here comes RPC (Remote procedure calls) way to standardize this communication call

→ Network call (looks like local network call)
- It hides the complexity of making local network call

- Happens over network through STUBS.

- Login is on Auth service.
- Notificatio-otp is on notification service.
→ Someone need convert information, basically marsh- al this particular thing into something serialize
→ Someone needs to format this sent to Notificatn service and convert it back to native obj on notificatn service.



→ when in Auth service the notification_otp first thing it interact with STUB

→ Converts into compressed format and sent.

→ STUB on java convert and give it in java subclass.

→ create response (JAVA) and vice-versa.

The stub convert this methods, request type, response type into the format used by the Remote procedure call (RPC).

⇒ <u>Stub interface description langauge</u>

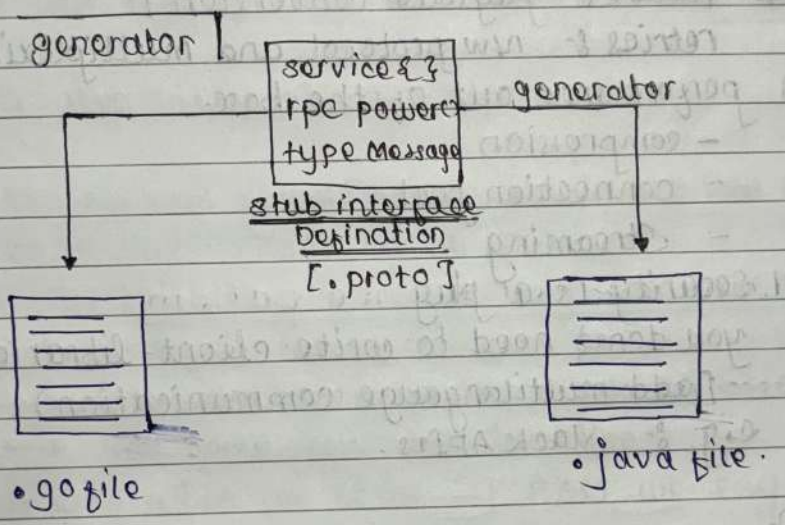
→ Every single RPC implementation will give you a way to define a interface langauge, and it is where you define a type.

→ send Notification req is a type and send notification response is a type. this type needs to be present in Golang struct (Auth service) and also in Java classes (Notification service)

→ Common way to define this, Interface description langauge.

→ GRPC (proto-pup format).
→ Describing in .proto file.

→ Declaring that it is the Interface that both the service would expose and will implement.
write a server-program that implement this interface



→ generator takes .proto file and it will write a necessary peace of component, that is required to it to make a network call, do the retry, everything it would want to (in Java / Golang).

→ you have to write a business logic

→ Communication in Remote procedure call (RPC) ←

You can use whichever protocol you want
→ TCP / UDP
→ GrRPC → HTTP/2

Advantages

(i) Easy to use and strong API contract.
(ii) invoke remote call just like a local call.
(iii) Most RPC (remote procedure call) support modern langauge.
(iv) Most mundane task are obstracted.
→ Failure & payload conversion / retries & n/w protocol and multiplexing
(v) performance out of the box.
- compression
- connection pool
- Streaming
(vi) security is a plug
(vii) you donot need to write client-libraries [add multilangauge communication)
e.g :- Slack APIs.

- **Concerns**

- stub need to be regenerated whenever signature changes (both need to change service).
- testing RPC is non-trivial.
- getting started is little challenging.
- browser support is limited

8] Everything you need to know about REST

→ Representation state transfer (or entities).
→ Every data we have is resource

- Rest just tell how the format of data, how client and server should communicate.
- Client can ask for a particular representation (data).

A] What about this representation

→ Any request comes from the client has a format in which data needs to be send.

REST and HTTP.
→ HTTP methods GET, POST, PUT, DELETE (method)
→ you have your URL (resourse)
that is why when we think of REST we say REST API.

→ URL should be identification of the resourse.

- HTTP and tooling

(i) HTTP client :- CURL, POSTMAN, REQUEST
(ii) web cache :- nginx cache, varnish, ha proxy
(iii) HTTP monitoring tools :- traeing, packet sniffing
(iv) Load balancing :- distributed load uniformly
(v) compression

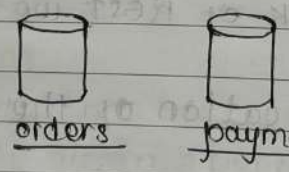
Downside.

(i) consumption is not easy
(ii) Not as simple as STUB in RPC.
(iii) get response in JSON, convert it to native.
(iv) Consumption is repetitive.
(serialization and deserialization logic).
(v) Some server may not support
(vi) HTTP payload are huge.
(vii) You can-not switch protocol easily (TCP → UDP)

- HTTP runs on top of TCP.

8 Database per service pattern

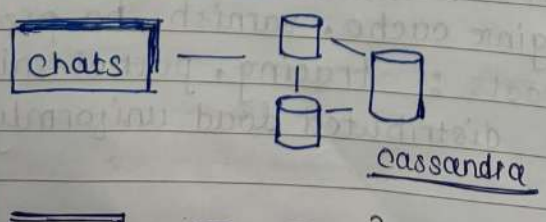
→ All microservice needs a datastore to persist and hold the state of data.



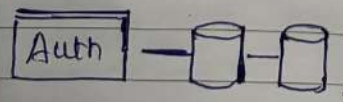
→ loose coupling

→ (i) Independent to build, test, deploy, scale
(ii) autonomous in making there own decision

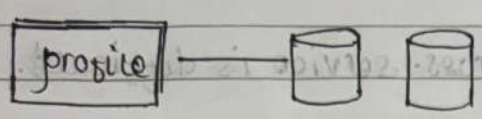
Social Network.



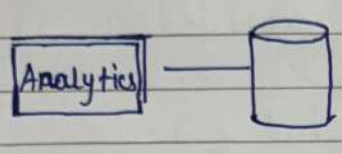
High injection value of data coming in. (High writes).



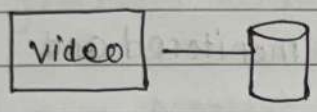
simple db with master and replica setup.



} information is not constant
you may use { MongoDB
Read / Replica setup



} Datawarehouse . { Readshift.



} blob : Amazon S3.

Advantages.

(i) you get loosely-coupled component.
(No direct access).

(ii) you have a very specific DB need for your service.
(graph DB).

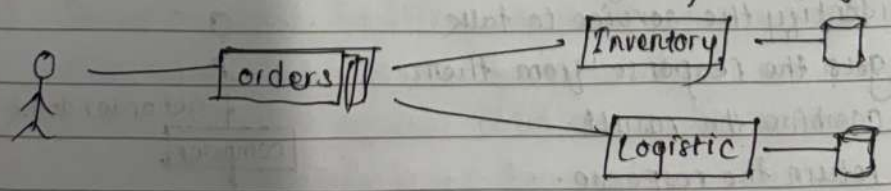
(iii) you want granular control and scaling of your service
eg. horizontal, vertical, partitioned, decentralized.
payment (MYSQL, POSTGRE).

(iv) If db goes down it only affect that service only.

(v) you have separate compliance need for certain type of data

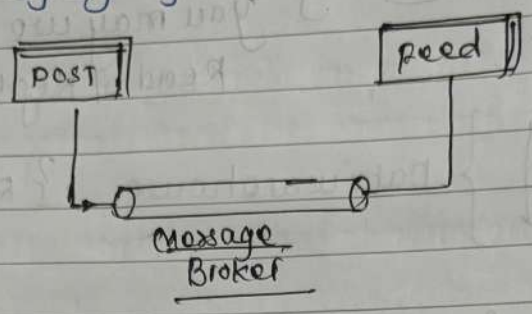
DisAdvantage.

(i) Cross-service transaction are super complex and expensive (Time through put)



→ Distributed transaction.

(ii) Conveying update accross service is difficult.



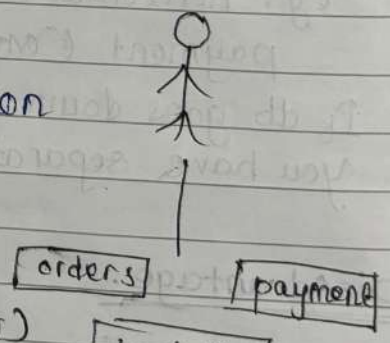
(iii) Multiple infra component to be monitored and managed.

---

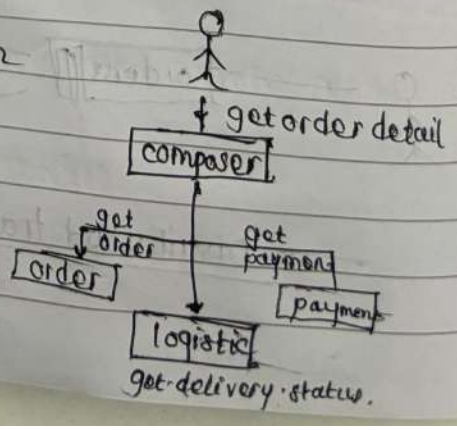
10 API composition pattern in Microservice

→ Say we have developed 7 microservice & its working good, now we want to add a new feature in the product which will be like talking to three service and compile the response.

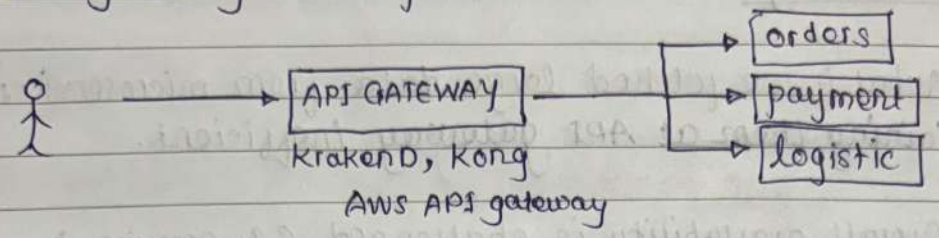
To query microservices, common pattern is used called as API Composition.



(i) put a middle man (composer)
- takes users query
- identify the service to talk
- gets the response from them
- combine the result.
- return the response.



API gateway (composer).



The calls to the individual microservices. can happens.
- sequentially → takes longer time.
- parallel → Require more m/c resources.

- Improving end user experience

→ If we call one by one (without API gateway)
- More roundtrip - (Higher latency).
- Data merge

→ client make one API call to composer and then ⇐

- Branch Composition (Multilevel API composition).

Advantages.

(i) Simple
(ii) user has single point of contact.
(iii) Hides implementation details and complexities.
(iv) Security and limiting can be implemented at composer.
(v) Can cover "bad" design decision.
(vi) Hides legacy system and we can replace it gradually.

DisAdvantages

(i) What if we fetched large data from microservices. joining them at API gateway inefficient.

(ii) Overall availability is challenged (If service is down).

(iii) Having transactional data consistency is very difficult. (Distributed transaction).

(iv) composer need to be managed and maintained.

(v) It may become bottleneck.

11]

Microservice are typically structured around Business needs and are owned by a small team.

→ OPTIMIZE

- Rapid feature delivery
- easy to upgrade the stack

Monolith → Microservice.

(i) club fun and related work and create a microservice

(ii) Start small

(iii) Pick up specific set of module.

(iv) So when you create one add/expose the API to get information.

Characterostics of Microservices

(i) Autonomous :- decision and operation are independent and evolutionary

(ii) Specialized :- focused on solving one problem

(iii) Built for business :- Organized around business need and team.

12 Challenges in adopting and implementing Microservices

(i) Managing Microservices [defining scope]

(ii) Monitoring and logging (blindspot).

(a) Every component / service / server

(b) Debugging a root cause, spanning service

↓

Distributed tracing becomes super critical. (eg. zipkin)

(iii) Service Discovery

(a) central service registry

(b) load balancer based

(c) service mesh.

(iv) Aunthentication and authorization

- a central auth (internal) service issuing JWT tokens

(v) Configuration Management

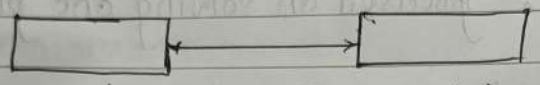
(vi) Theres no going back

(vii) fault tolerance :- More way to fail (because 1000 servers) Outages are inevitable.

- tolerance (loose coupling).

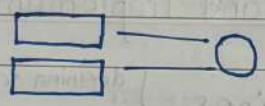
- Async Dependancy

(viii) Internal and External testing.
(ix) Design with failure in mind.
(x) Dependancy Management.
(a) service dependancy.



(sync dependancy may trigger cascading failure).

(b) library / Module dependancy



{ without proper versioning
backward compatible }

(c) Data Dependancy

13 Standardizing Microservice

A good service
- manageable (fire fight any outage)
- observable (knowing what is happening).
- debuggable

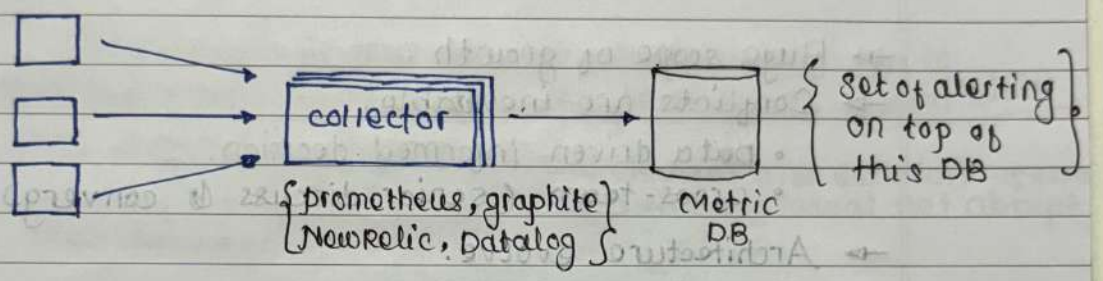
Ensure anyone who add or some functionalities or adopt or build they are on some space.
→ Keeping us coherent & uniform.

① Monitoring :- How services are interacting with each other
→ How the req. flow and res.

Request wise view. → Distributed tracing
(using zipkin, AWS x-ray).

- how every server is doing - CPU, Memory, Data consumption
- how every service is doing - Health check

→ STANDARDIZE



② Interface.

→ How would two service talk to each other. ?
→ How would end user talk to a service ?

- How to define routes ?
- How to name endpoint ?
- How to paginate the documents ?

- Versioning
- Connection timeout (not to small nor large) | Retry stratergy Exponential backoff | payload JSON, XML, TXT

③ Tolerance. :-

Ensure that our Infra is not going down or our service is not going down.

- ration the number of call you get from other service
- limit the number of outgoing calls from a service.
- have ability to cut-off incoming call from a service.
- have ability to cut-off outgoing call to a service.

14 Things to remember while building a Microservice

→ Huge scope of growth.
→ Conflicts are inevitable.
  - Data driven informed decision.
  - cross-team (senior discuss to converg).
→ Architecture evolve.
→ Technical Debt.
  (all inefficient decision).
→ Service template and enforcing standardization
  [creating in github repo] ↓ {provide proper reason}

→ Central template should be collective effort.
→ All engineer should be allowed
→

→ Business >> Engineering

16 Preparing for Integration

Running microservice in isolation does not make a lot of sense. They work together with other services to get things done.

ⓐ Forward and backward compatibility

Make changes to your services, such that the others don't need to change at all } this is compatibility

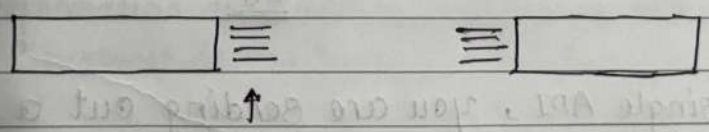
- database migration
- API responses. } changes and deletion of some can be gradual not abrupt
- Message for async communication

Some DO'S AND DONT'S

ⓘ Never change the type of the column / attribute
↳ new column with new type.

ⓘⓘ Never delete column / attribute.
↳ Instead deprecate it

ⓑ Make API interface tech agnostic



↑ an RPC marshalling format that would only work when other service is using JAVA along with a particular lib.

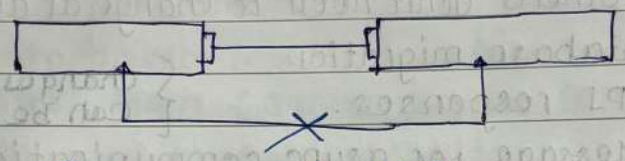
c) Dead simple consumption

→ Keep it simple (As other service are going to integrate).

- Simple API
- Simple data format.
- Common protocol

(consumer facing)

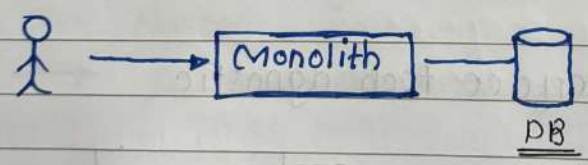
d) Hide Internal implementation details



(Never-ever do this).

- Upgrade become difficult.
- appeat technical debt.

16 BackEnd for FrontEnd



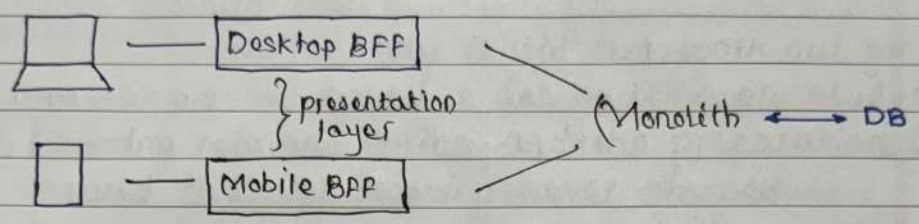
→ In single API, you are sending out a huge response that would help you render complete page.

- Reduce n/w overhead.
- Easy to cache.

- WHEN interface are significantly different.
- Communication format of client is very different.

→ Mobile have very little real estate (screen size).

• BFF (Backend for frontend).

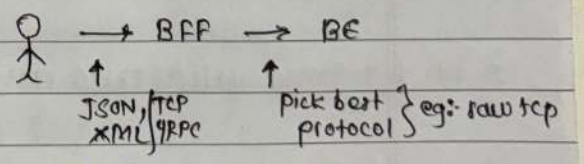


- only presentational layer
- work as API Gateway

Each BFF decides
→ what (data)
→ How (patch)
→ What (Res Data).

Advantages.

- Support isolation interface
- Client specific tweeks
- Hide sensitive information
- Transformation of data (protocol / stack)



- Improved security (checks at BFF layer only)
- Single general purpose backend.
- Act as Request aggregator.

Challenges

(i) fanout (network heavy & stak support).
(ii) code duplication (iii) More moving part.
(iv) Increase latency (slightly).