# ­OOP’S Concepts:

OOP is a fundamental concept that revolves around the concept of **Classes** & **Obects.**

1. **Classes & Objects:**

A Class is a blue print or template that defines the behaviour of an object. An object is an instance of classes

Class Car{  
  
int number;

String mode;

// Defult constructor

Public car(){  
}

Public static void main(String[]s ){

Car virtus = new car();

}

}

1. **Encapsulation:**It helps object protect the internal state from external interference and misuse.

We can achieve this by making all the instance variable as private and we will allow the values to be set using getter and setter method so no external classes have the right to access the instance variable for the same.

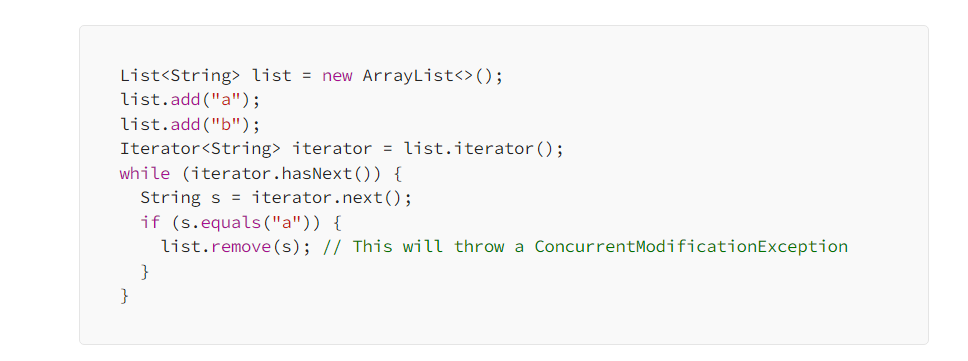
1. Inheritance:  
   It’s a mechanism to allow user to inherit the properties and methods from another class called superclass or parent class.  
     
   The child class inherits all the fields and method from the parent class except Private & Final variables & Methods
2. PLOYMORPHISM:  
   It enables you to write generic code & ability of an object to take on multiple form   
   Method overloading & method overriding are the 2 concept of polymorphism

# Concurrent Modification Exceptions:

It’s a Run time exception that occurs when we try to modify the Collection i.e

1. Vector
2. List
3. Set
4. Map

If we try to modify, update, or delete the value at the time of iteration then we will get the run time exception that is Concurrent Modification Exception and this scenario is called **FAIL FAST.**

****To avoid ConcurrentModificationExceptions, you can:

1. Use thread-safe collections such as Vector or CopyOnWriteArrayList instead of non-thread-safe collections like ArrayList.
2. Use synchronization to ensure that only one thread can modify the collection at a time.
3. Use an iterator that supports the remove() (iterator.remove()) method and call that method instead of modifying the collection directly.

# Vector In Java

Vectors are dsa to store the elements with in it and we use this when we want to preserve the synchronization across the application

Or we want to support the legacy system where we are using the Vector we that support.

As compare to ArrayList<> Vector methods are synchronized in nature which make it thread safe.

# Explain Solid Principles In java:

As we know at some point of time the application will grow we can reduce their complexity and save ourselves a lot of headaches further that can create the use and we can form the reusability components

**Single Responsibility:**

A class should only have one responsibility

Let say if we have a class BOOK then this class contains all the data fields method and logic related to book only

If we have a user class the all the data related to the user must be present there should not be a method for user Authentication & Authorization

For these Two we should have separate classes “UserAuthentication” & “UserAuthorization”.

**Open for extension closed for Modifications:**

We are stopping the class modification here and give us a chance to do a bug free code

Let say we have payment interface and we have an abstract method pay

Different classes we have

1. UPI
2. Credit
3. Debit
4. Cash

All these are the implementing class of the pay interface so we are open for extension if we have to add new payment strategy then we will create one more class like bitcoin payment and implement the pay method accordingly

Main class logic of this {

Payment pay;

setPaymentStrategy(Payment pay){

this.pay = pay;

}

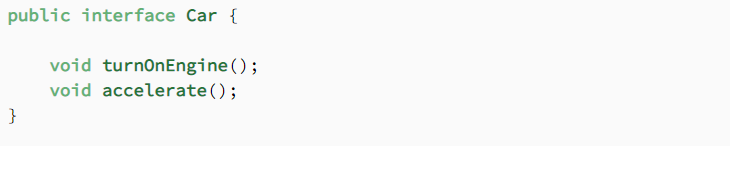
}

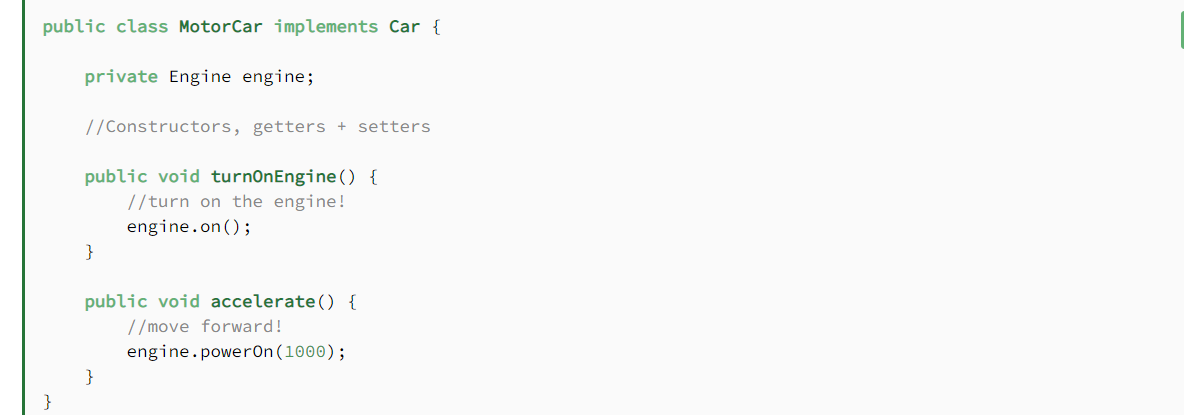
StrategyPattern{

setPaymentStrategy(new paypalPayement(10000));

Liskov Substitution:

If class A is a subtype of Class B we should be able to replace B with A without disrupting the behaviour of our program





**Interface Segregations**:

The I  in SOLID stands for interface segregation, and it simply means that **larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

For this example, we’re going to try our hands as zookeepers. And more specifically, we’ll be working in the bear enclosure.

Let’s start with an interface that outlines our roles as a bear keeper:

**Liskov Substitution principle:**

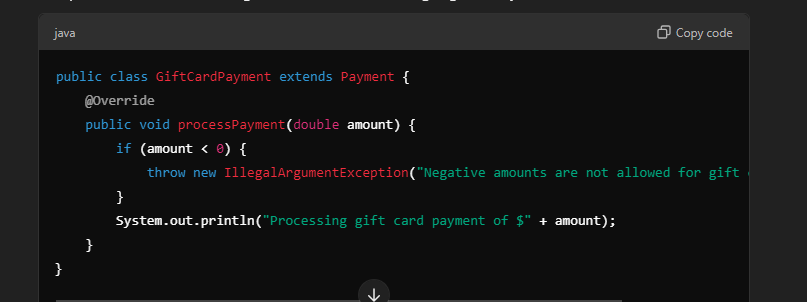
**Real-World Example of LSP: A Payment System**

Consider a scenario where we are building a **payment processing system** for an e-commerce platform. There is a base class Payment and different types of payments, like CreditCardPayment and PayPalPayment.

**Step 1: Base Class**

We start with a base class Payment that has a method processPayment:

Here I have define the payment Abstract class along with the implementation of that payment class with other classes

1. Let Say If One of the child class changes the behaviour of the payment by throwing the exception and it might break the code as we are not explicitly handle the exception
2. 

Payment pay = new GiftCardPayment();

pay.processPayment(-50);

this will break the logic of the code as it will through the exceptions.

**Dependency Injection:**

# Immutable Classes in java:

The classes where we cannot change the value of it

Once the person object is created then the state of that object cannot be changed

If we are going to change the state of it and there is the reference of other classes as well then we need to make sure that it should return the defensive copy of that particular object so that we can’t change the state of our immutable class

Let take an example person:

Class Person{

Private Int id;

Private LocalDateTime ldt;

Private Address address;

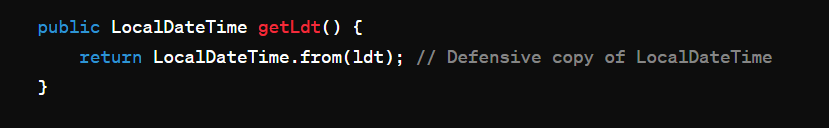
Public person(Int id, LocalDateTime ldt, Address address){

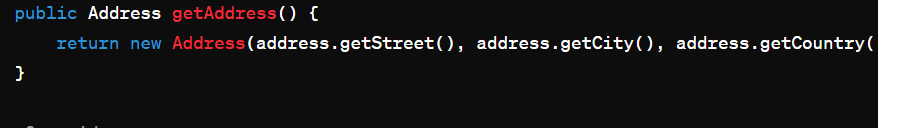
This.id = id;

this.ldt = LocalDateTime.from(ldt) // Defensive copy of LocalDateTImel

this.address = new Address(address.getStreet(), address.getCity());

}





That’s how we will have the defensive copy of other reference classes.

For Collections we will have defensive copies like

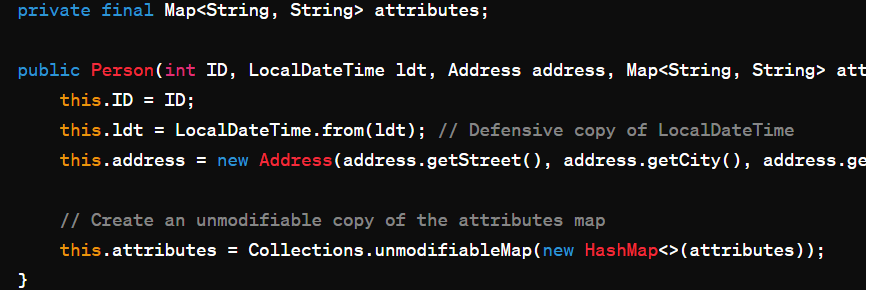
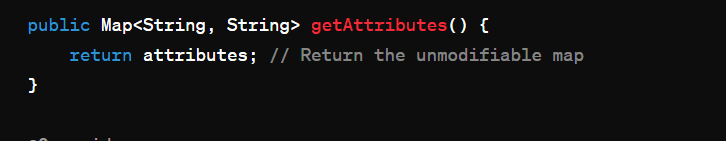
List<String> hobbies;

Constructor(){  
hobbies = Collection.UnmodifiableList(hobbies);

}

Here we will have the defensive copies;

For map :

# Query to find the employees in Each department:

SELECT emp\_department, Count(\*) from employees GROUP BY emp\_department

# Java8 changes in memory:

JVM defines various run time data area which are used to execution of a program. The data areas of thread are created during instantiations and destroy when the thread exists

Note: JVM destroy when JVM exits.

JVM memory divides in the following

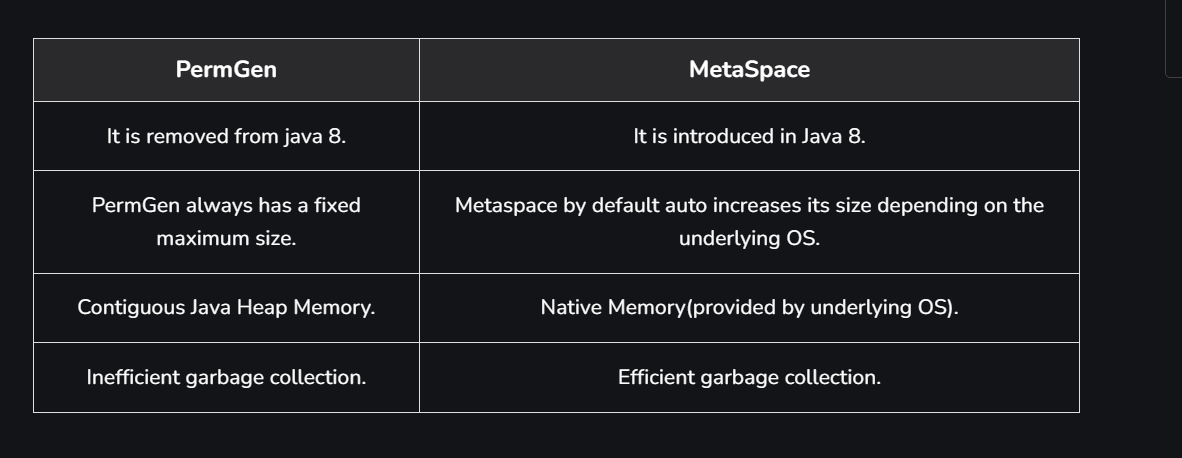
* Heap Area
* Method Area
* JVM Stack
* Native Method stack
* PC Register

All the objects are stored into the heap area The heap are is created when the JVM Starts.

The heap is generally divided into the two parts:

1. Young Generation – All the objects are stored in this generation only When the memory is fill then garbage collection happens this is called Minor Garbage Collections.
2. Old Generation – All the long lived object which has survived many garbage collections is stored in this area

JVM also contains Permeant generation memory apart from heap memory or “PermGen”



# **Write a program to find even numbers from a list of integers and multiply with 2 using stream java 8**

List.stream().filter(x -> x % 2 == 0)

.map(x -> x \* 2).collect(Collectors.toList());

# Why we need hashcode if we have equal what are the contract terms between them.

The hashcode() in java use to generate a unique integer representation for an object based on it’s content. The primary purpose of hashcode() is to support hashing based data structures such as HashMap<>, HashSet and HashTable.

While the **equals()** method is used to compare two objects for equality based on their contents, the **hashCode()** method is used to efficiently distribute objects across buckets in hash-based data structures. Here's why we need both **hashCode()** and **equals()** methods:

Hash-based data structures use the hash code of objects to determine their storage location in the underlying data structure (e.g., buckets in a hash table). When retrieving an object from the data structure, the hash code is used to quickly locate the bucket where the object is stored, thereby improving retrieval performance

To ensure consistency between **hashCode()** and **equals()** methods, the contract between these two methods must be maintained. According to the contract:

If two objects are equal according to the **equals()** method, their hash codes must be equal as well

if two objects have the same hash code, they may or may not be equal according to the **equals()** method. However, it's more efficient for hash-based data structures if objects that are considered equal also have equal hash codes

# What if heap memory is full and we need to optimise the code in java to save the heap memory what can we do?

1. Optimise Data Structure we need to user :
   1. Choose DS that minimize memory usage for the given use case For example use arrays instead of ArrayList If the size is fixed and known in advance as arrays have a smaller memory foot print
2. **Reduce String Usage:**
   1. Strings in Java are immutable and can consume a significant amount of memory, especially when manipulating them frequently. Use StringBuilder or StringBuffer for string manipulation operations that involve concatenation or modification to avoid creating unnecessary string objects.

# How to design the application from scratch from code?

Organized your code base into module,package,Function and domain boundaries For example We could have separate package for Controller,Services,and data domain(Repository), models and utilities

Keep business logic separate from the Data logic and presentation logic

Use the design pattern Such as DAO, MVC, and object creation pattern

* Implement data access logic in repository or DAO classes responsible for interacting with the database. Use ORM (Object-Relational Mapping) frameworks like Hibernate or JPA to abstract away database interactions and simplify data access.

1. We need to use technique such as query optimization , Lazy loading and cache to minimize the latency and response time to the user
2. Implement the filteration and pagination to limit the amount of data to be transfer over the network.

Infrastructure related Design :

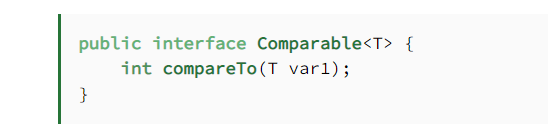
* + - 1. Design scalable solution architecture that can handle growing user loads and data volumes such as use of Load balancer to distribute the load among the instances of that service, Horizontal scaling that is basically increasing the instances of a particular service, use of caching and asynchronous process
* **Define Data Model:**
  + Design a robust data model that represents the entities, relationships, and constraints of the application domain. Use techniques such as entity-relationship modeling and normalization to ensure data integrity and consistency.

# How are you handling the load of 500k concurrent user?

# How to optimise the SQL queries and what are the approaches?

# Comparable VS Comparator

Sometimes when we write a new class we need to compare two objects of that class, It’s especially helpful when we want to use sorted collections There are 2 ways we can do this Comparable interfaces or With the comparator Interfaces.



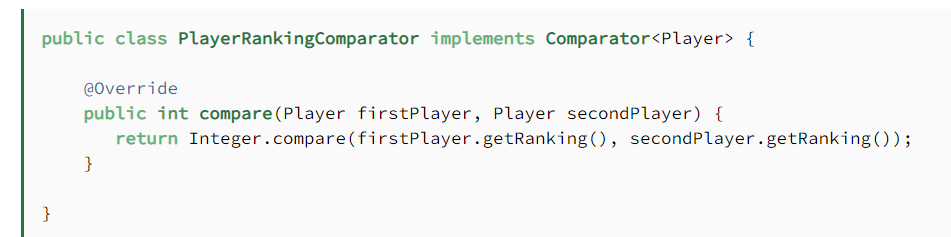
We should implement the interface for the object which we want to be sortedWe should implement the interface for the object which we want to be sorted

It has compareTo method which return int 3 values such as -1,0,1

**Comparable is an interface defining a strategy of comparing an object with other objects of the same type. This is called the class’s “natural ordering**

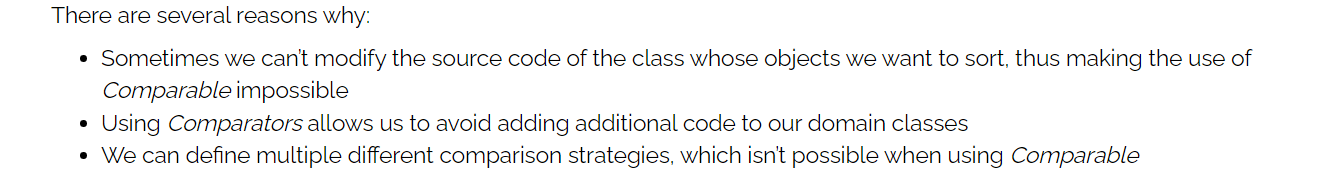


**The Comparator interface defines a compare(arg1, arg2) method** with two arguments that represent compared objects, and works similarly to the Comparable.compareTo() method.





Sort Summary:

The comparable interface is good for natural sorting order or we need the default sorting ordering

# What are the method of Object class what can they do.

* Equals()
* getClass()
* hashcode()
* notify
* notifyAll()
* toString()
* wait()
* Clone()
* Finalize()

# Design Principle Of Micro services

* Modularity :
  + Service should be self contained and have a single well defined Purpose.
* Scalability:
  + Service should be able to scale independently and handle the increase load
* The System Should be Loosely Coupled and decentralized.
* Service should be Available all the time for that we need to define the HA of the System
* Service should able to handle the failure gracefully
* Service should be deployable independently
* Automation in the micro service for scaling and testing

# Design a application where you are getting millions of request how will you design the application:

Use of Load Balancer: A lb distribute the request across the multiple server to ensure that no single server get over load

**Choose the right architecture:**  Microservice, serverless, or monolithic architecture can all work depending on the specific use case. Microservices allow for more flexibility and can handle scale better but are more complex to manage. Serverless architecture can automatically scale to handle the load but may have limitations on customization.

**Use a distributed cache**: A distributed cache can help reduce the load on the database by storing frequently accessed data in memory. This can speed up the application’s response time and reduce the number of database queries.

**Optimize database performance**: Databases are often a bottleneck in highly scalable applications. To optimize database performance, use techniques like indexing, caching and partitioning.

**Implement asynchronous processing**: By using asynchronous processing, the application can handle multiple requests at the same time, improving performance and scalability.

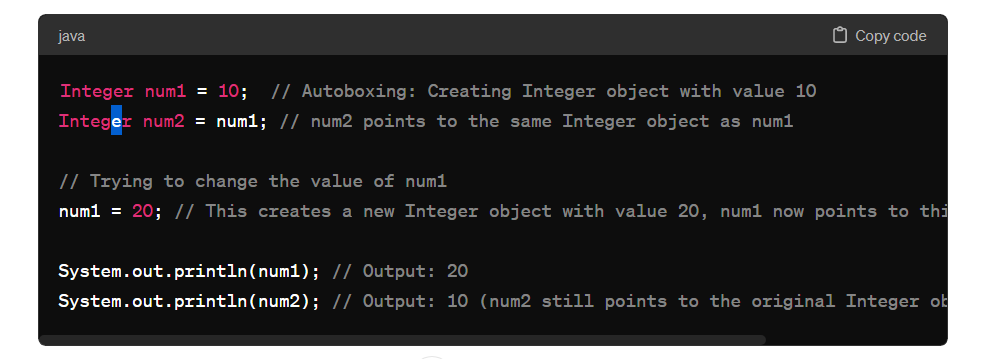
**Use Content Delivery Network (CDN)**: CDN is a network of servers that can distribute content globally, delivering it from the closest server to the user, thus reducing the latency and improving the application’s performance.

**Monitor and optimize**: Monitor the application’s performance and usage patterns and optimize the infrastructure accordingly. Implementing logging, monitoring, and alerting can help detect issues and optimize performance in real time.

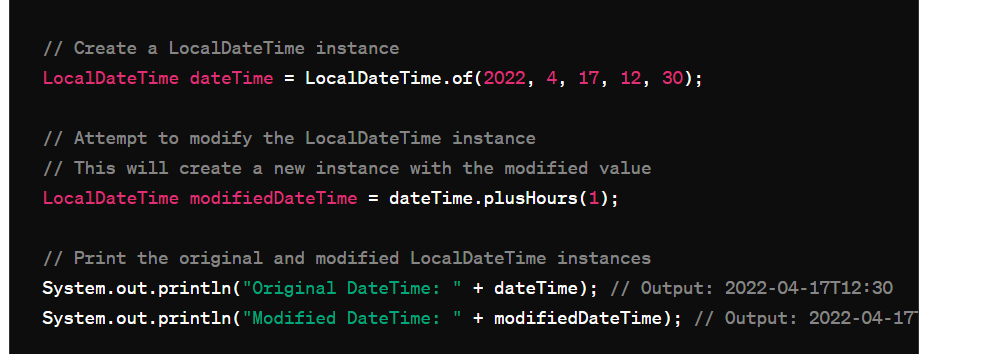
# Inbuilt Immutable Classes in Java:

String: The class represent the sequence of character and is one of the most commonly used classes in java Once a string object is created it’s content cannot be modified.

Integer,Long,Double,Float, Short,Byte, & Character These classes represent primitive data types and are also immutable. Once an Object of any of these classes is created its value cannot be changed;

Example: 

LocalDate, LocalTime, LocalDateTime, and Instant: These classes represent dates and times and are also immutable. They are part of the java.time package introduced in Java 8.



They don’t have the setter method and if we are refering the object with another instance and try to change it then it will create the new instances for the same.

# SQL Query to print the value if it exist otherwise Print the null and create the dynamic query for the same:

DELIMITER //

CREATE PROCEDURE GetWorkersWithSalary(IN salary\_value INT)

BEGIN

SELECT

\*,

CASE

WHEN salary = salary\_value THEN salary\_value

ELSE NULL

END AS Null\_Salary

FROM

worker;

END //

DELIMITER ;

# @Table Annotation In java:

* 1. Specify the name of the table that is mapped to the db with the help of ORM frame work

@Entity

@Table(name = “employees”)

Public class Employee{

}

* 1. Defining the unique constraint to one or more columns of the database table

@Entity

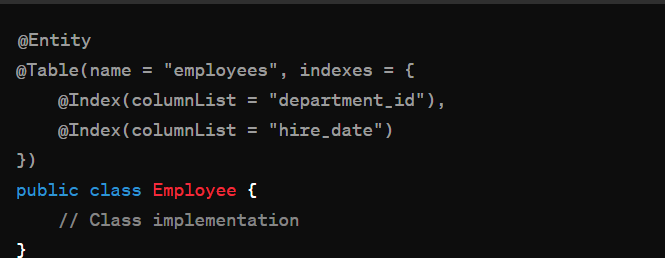
@Table(name = “employees”,

uniqueConstraints = {

@UniqueConstraint(ColumnNames = {“email”,”Phone”})

})

* 1. We can set the index using this annotation



# Clone Object Information and clone able:

In java it is used to define the shallow copy of the object and clone() provide a way to create a new object with same state by default it performs the shallow copy

Colne able interface act as marker interface and we have a clone() in Object class which we override to define logic

* + - 1. Proto type design pattern cloning is used
      2. Immutable object we will use cloning

# Difference btw Mono lithic & Micro services architecture:

Mono take all software components into one single project and microservice divide the application into separate self contained services.

1. Mono lithic app are not separable housing all functionality in a single codebase whereas micro services are modular each responsible for a specific set of task.
2. Deployment of micro app are independent to each other and
3. Mono communicate with in person call which increase the dependency in the classes whereas micro call other services with HTTP/REST protocol or message brokers.

When system is complex and we need to ensure the millions of user a scalable project then we will use micro services.

# Asynchronous & Event Based Communication in micro services:

We have 2 communication One is

1. Synchronous : REST, FEIGN,GRPC,
2. Asynchronous

Asynch Communication:

We can communicate with the help of queue We can send the message to the message broker and we will not wait for the response as soon as service receive the message from broker it will send the response.

If a services is down then in synch communication we will get the error but in async it’s more fault tolerance it will wait for the services to be up as soon as it’s up it will send the message to the services

If MB is down or restart then we will loose our data

Asynch removes the tight coupling from the services communication as we have to wait for the response

1. To make the fault tolerance **Make Replicas of MB**

## Types of Async:

1. Point to point
2. Pub sub

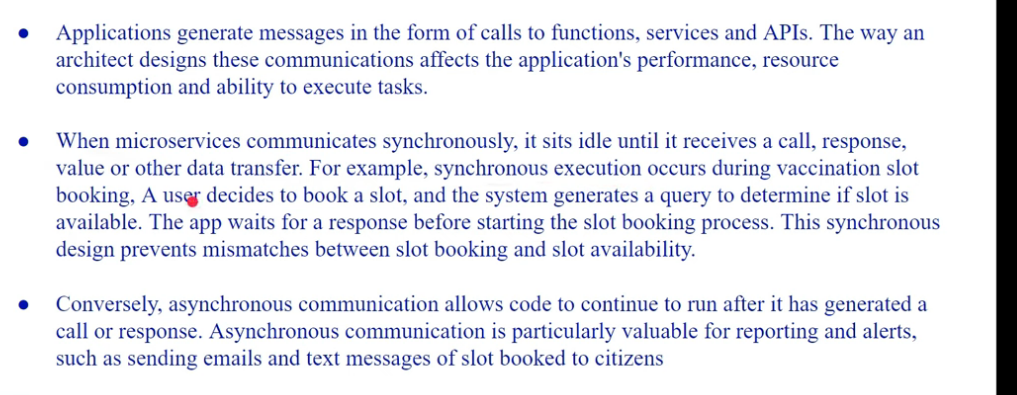
## Point to point:

* A queue will be used for this communication
* We have only 1 producer and 1 consumer that are interacting with each other with the help of queue
* One message send by producer will be consumed by one Consumer & message will be deleted after consumption
* If receiver service is down then message will be in the queue till the time receiver is up
* Popular choice is Rabbit MQ & Active MQ

## Pub – Sub model:

* In this model topic is used for the communication the broker will be used to store the message sent by the publisher and then subscribers that subscribe to that topic will consume that message.
* Unlike p to p the message will be ready to consume for all subscribers and the topic can have one or more subscriber. The message remains persistent in a topic until we delete it.
* Example are Kafka & AWS SNS etc.

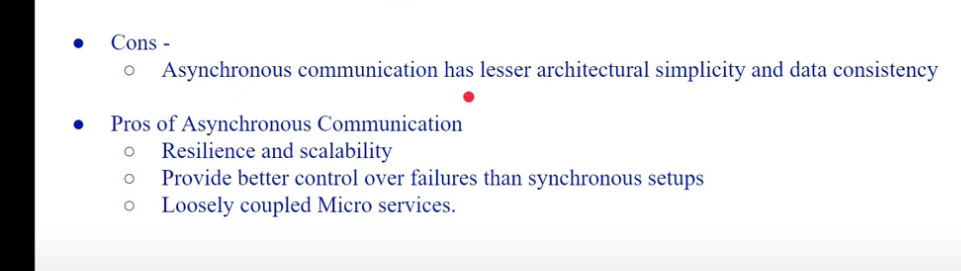
# How does Synch & ASync Communication Works?



Performance impact in sync communication if failure in any of the service

Note: to save yourself from failure we can have the load balancer and more instances of the services

ASYNCHRONOUS:



# When to use Which Micro service Communication:

* When we start from scratch we need to go with synch communication to optimise for speed of evolution
* If your service starts to become complex and architecture grows the start adding the asynchronous communication between the services
* Figure it out if response is needed at the time of call only or we can have the response later then we can have asynchronous communication

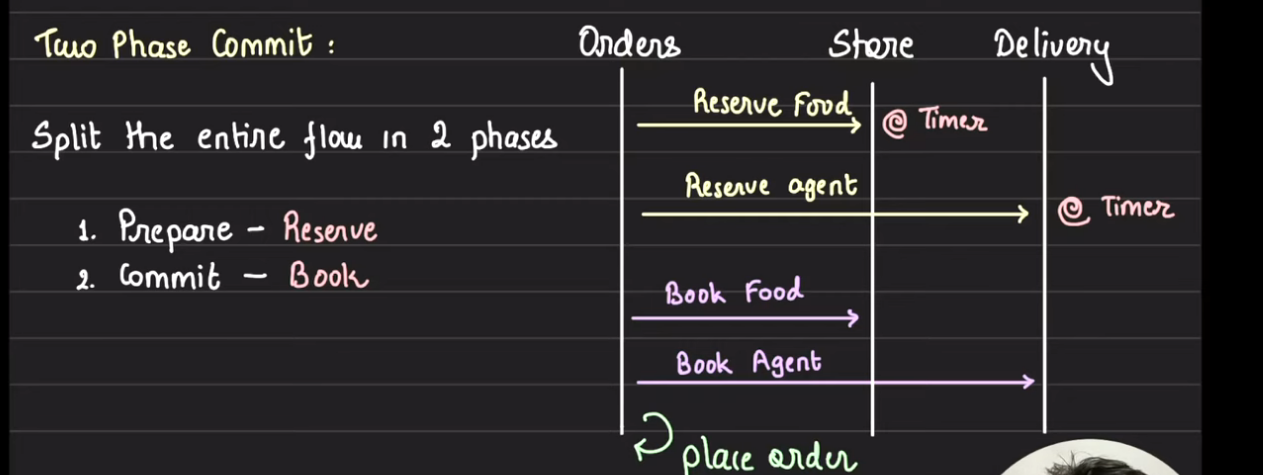
# Two – Phase Commit protocol:

When we have an order and that order is step of transaction need to be done in distributed manner if one of the service is fails then there is a chance of inconsistency and bad user experience.

To overcome this problem We have distributed transaction

## Two phase commit:

1. Prepare Phase
2. Commit phase



So here on coordinator will be there where it will give the service 2 option In prepare phase

Ready to commit yes or no

In commit phase coordinator will send all the service a msg to commit if any of them give –ve ack the coordinator tell each service to roll back.

Advantages:

* Atomic transaction are guaranteed
* Isolation each thread is working independently

DisAdvantages:

* Slow (Because everything is in the sequential)
* Prone to deadlock.(Chance of having a cyclic dependency).

# How to handle fault tolerance in Micro services & what are the ways:

Fault tolerance is a key concept in building resilient microservices. It ensures that a system can continue to operate even when some components fail. In microservices, this is crucial because these systems often rely on multiple interconnected services. If one service goes down, it shouldn't bring the entire application to a halt.

To achieve fault tolerance, several strategies and patterns are commonly used:

* [**Circuit Breaker**](https://www.sayonetech.com/blog/circuit-breaker-in-microservices/)**:** This pattern helps prevent a failure in one service from cascading to others. It monitors for failures and, when a threshold is reached, stops requests to the failing service, allowing it time to recover.
* **Retries:** When a request fails, the system can automatically retry it after a short delay. This is useful for transient failures, such as temporary network issues. However, it's important to manage retries carefully to avoid overwhelming services with repeated requests.
* **Timeouts:** Setting time limits on requests ensures that the system doesn't hang indefinitely waiting for a response. If a service doesn't respond within the specified time, the request is aborted, and a fallback strategy can be employed.
* **Fallbacks:** When a service fails, a fallback method can provide an alternative response. This might involve returning cached data or a default message, ensuring the user still receives a response.
* **Rate Limiters:** These control the number of requests a service can handle within a certain timeframe, protecting it from being overwhelmed by too many requests at once.

Implementing these patterns helps maintain the availability and reliability of microservices, even in the face of failures. By planning for failures and incorporating fault tolerance, developers can build systems that are capable of handling unexpected issues.

**Retry Pattern**

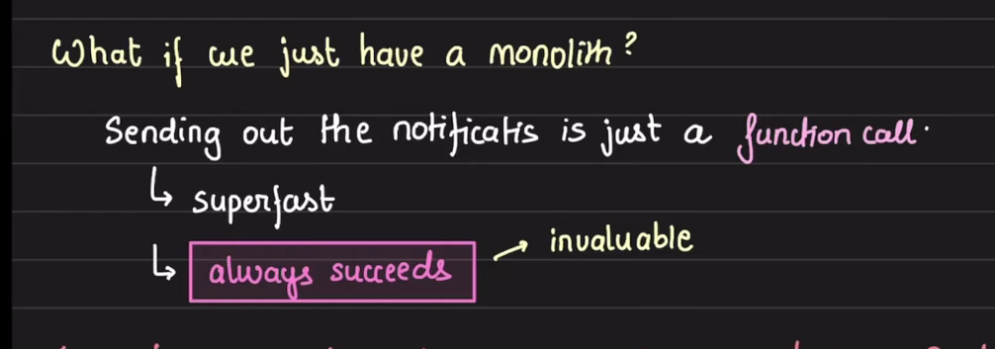
When a service encounters a transient failure, such as a network error or temporary unavailability of a dependent service, the Retry pattern allows the service to attempt the operation again.

The pattern typically involves detecting a failure, waiting for a specified duration before retrying, and retrying a configured number of times or until a timeout is reached.

Implementing the Retry pattern include setting sensible retry limits, using exponential backoff and jitter to avoid retry storms, and ensuring idempotency to prevent unintended side effects.

# Synchronous & Asynchronous Communication:

**Note: We are building a Notification server To notify user B when user A likes/comment on post.**

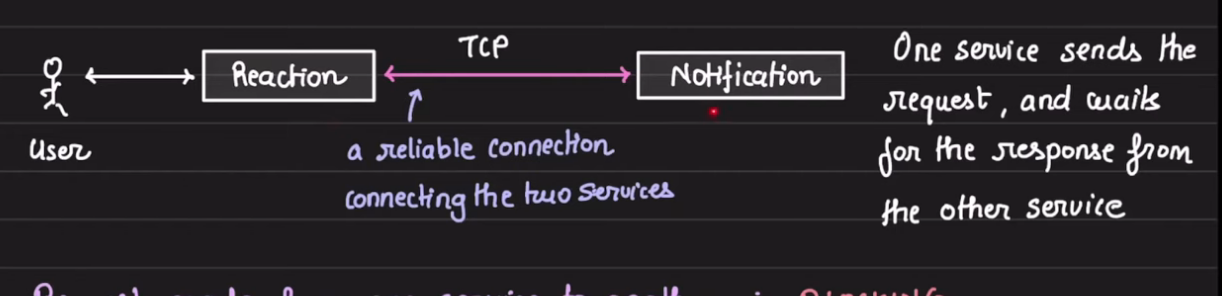
****

Things become extra challenges when we have distributed system

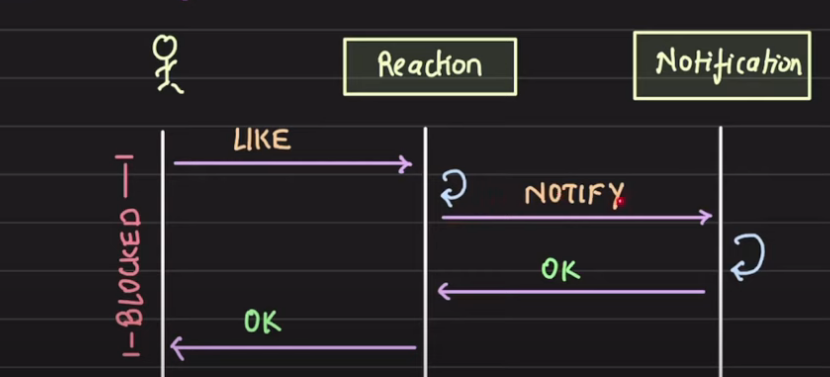
* Network Failure
* Target Service is down
* Target service is overwhelmed
* Target service is not reachable.

**How we make this service talk each other:**

## Synchronous Communication:



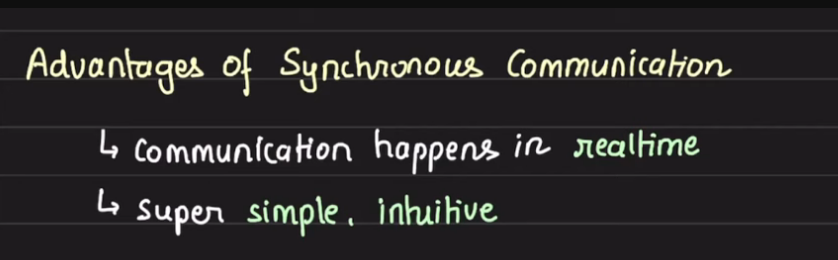
Request from one service to another service is **blocking**



We will have to wait for the notification response to complete the notified to the user the end user will get the response so it’s waiting us for the same.

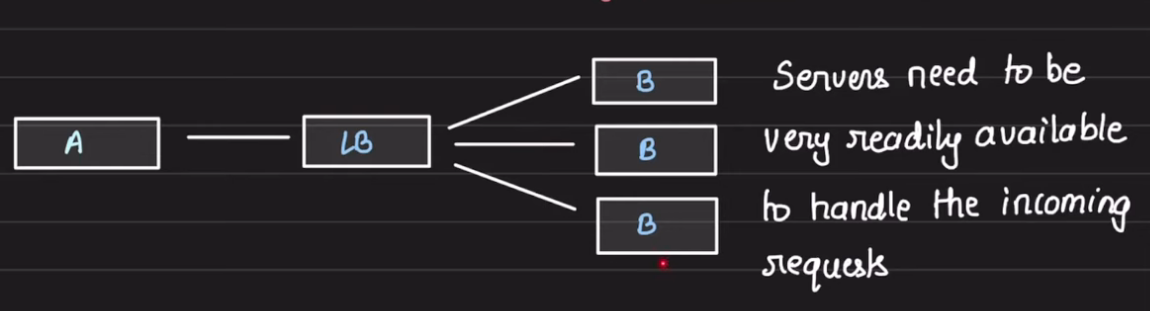
Communication Protocol:

1. Rest
2. GraphQl
3. Grpc



**Disadvantages Of Sync Communications:**

1. Caller is blocked until the response is received(ms,micro second,seconds of response time).
2. We will face the time out error if there is a chain of sync communication

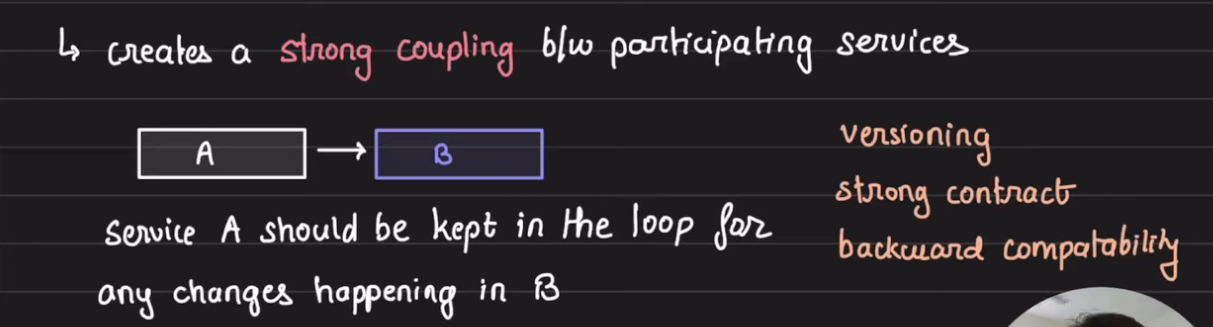


1. Risk of cascading failure :   
   when we have chain of services and each of them communicate then one service is down and so on all the services are going down.



Note: Solution is Circuit Breaker pattern

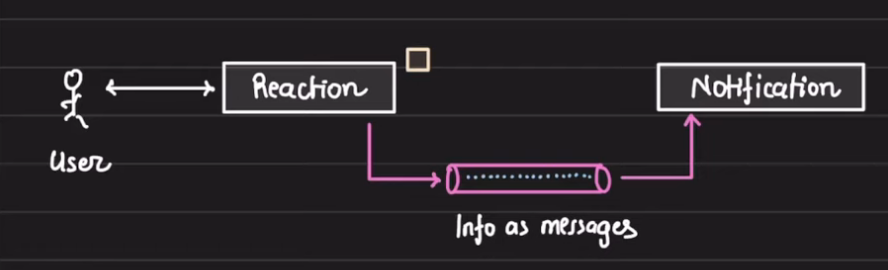
1. We don’t have loose coupling in sync communication which is the hectic thing



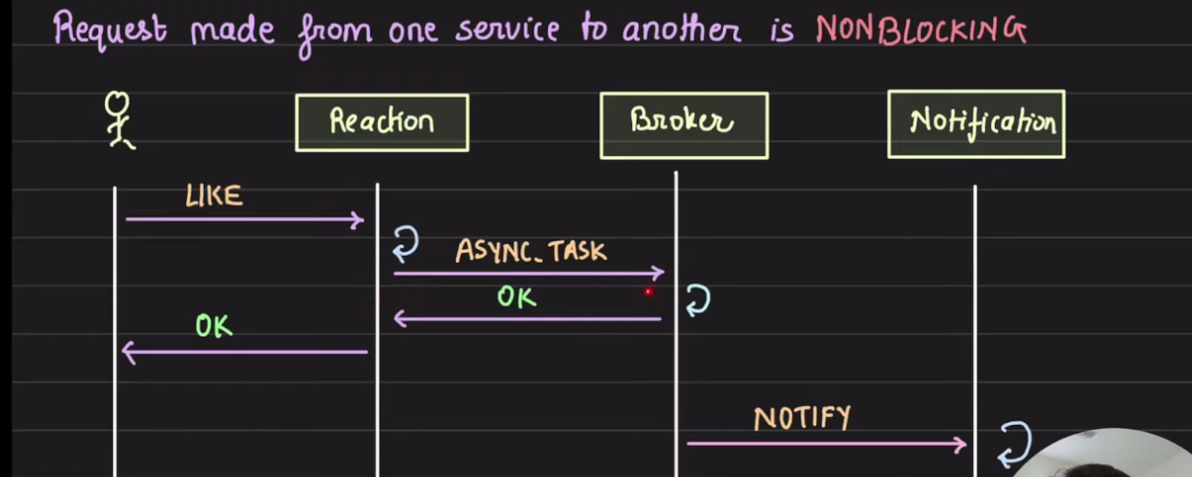
Good services don’t have strong coupling.



## Asynchronous Communication:



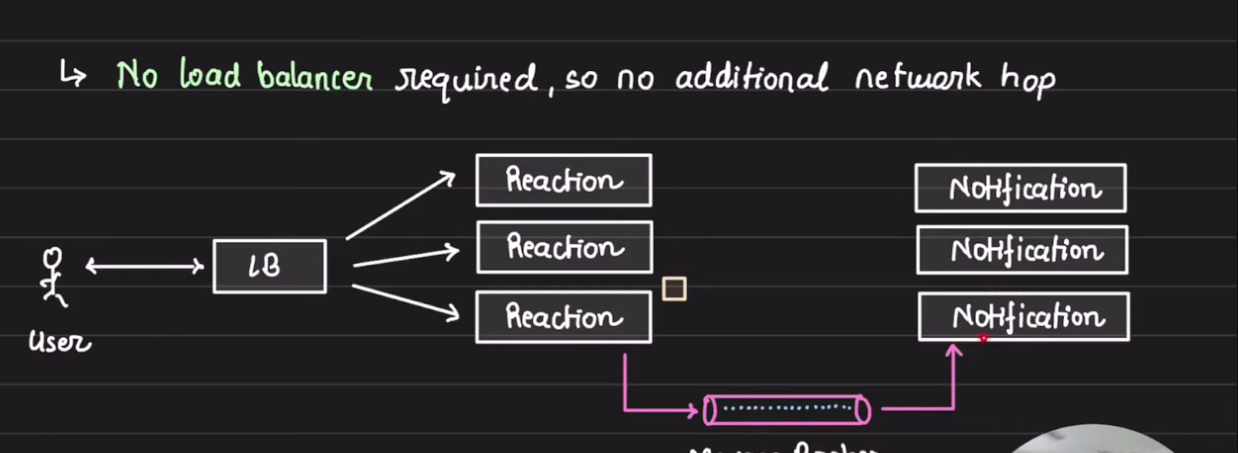
* In this communication we are going to generate the message into the broker that will be consumed by the notification service.
* Interaction between services is an exchange of text message via message broker.
* The message are buffered in the broker and the consuming services will consume them
* If the consumer service is down then the message will be in the queue when the consumer service is up then it will work.
* No risk of cascading failure we will get the response and eventually once the service is up then we will get the notification.



As a broker we can use:

1. Kafka
2. AWS SQS
3. Google Pub sub

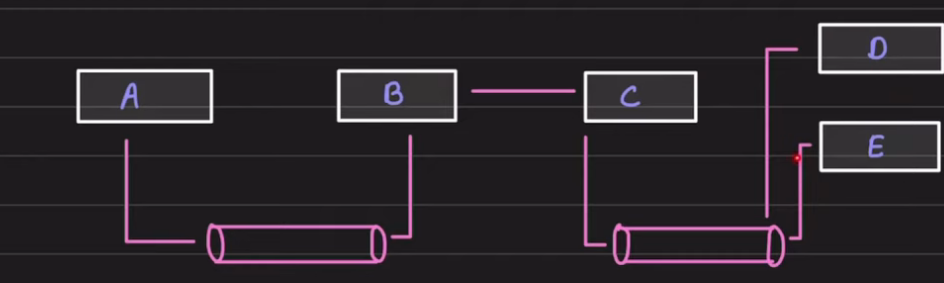
If notification services is overvlemhed there is no issue in that we will put the message into the broker and listener will do the jobs



**Disadvantage of Asynchronous:**

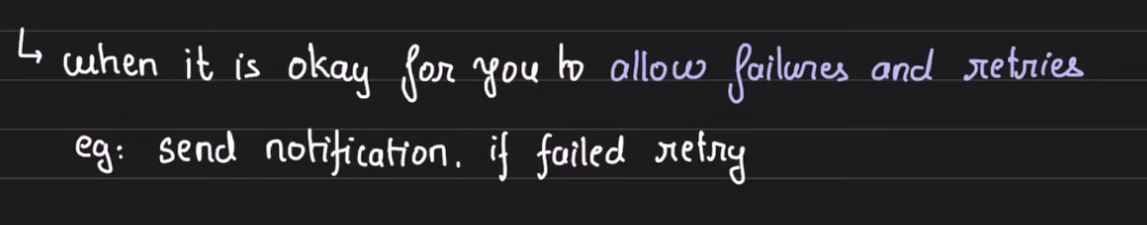
1. **Eventual Consistency:**You can not have a strongly consistent system with broker and hence you have to be okay message to be eventual consistence.
2. **Single Point of failure:**

The broker is backbone if it’s down we need to be cautious for the same

1. **Tracking:  
   **

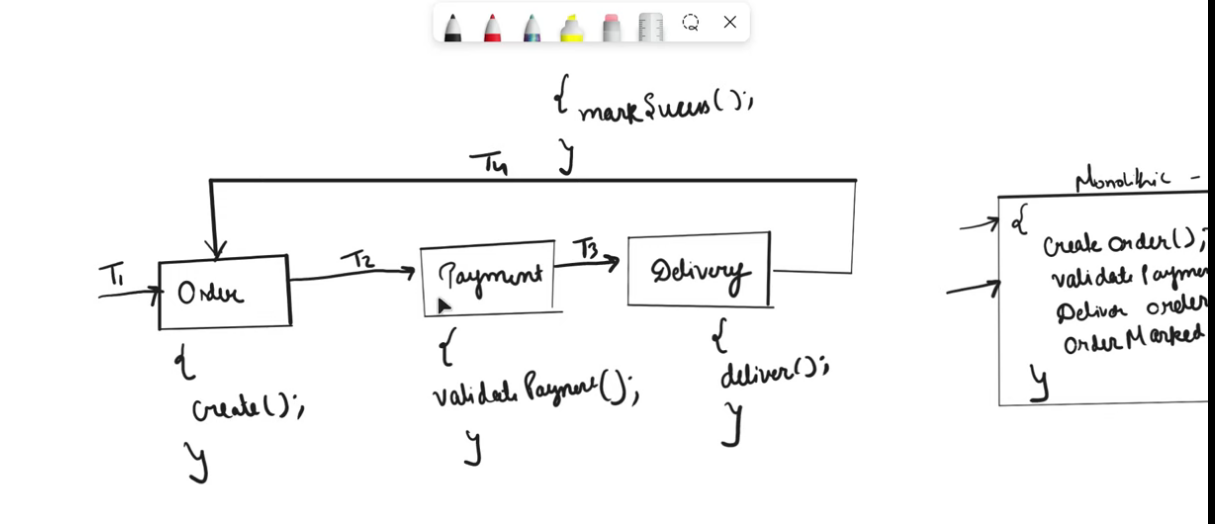
Tracing is hard we don’t know where our messages are.

**When should be use Asynchronous:**

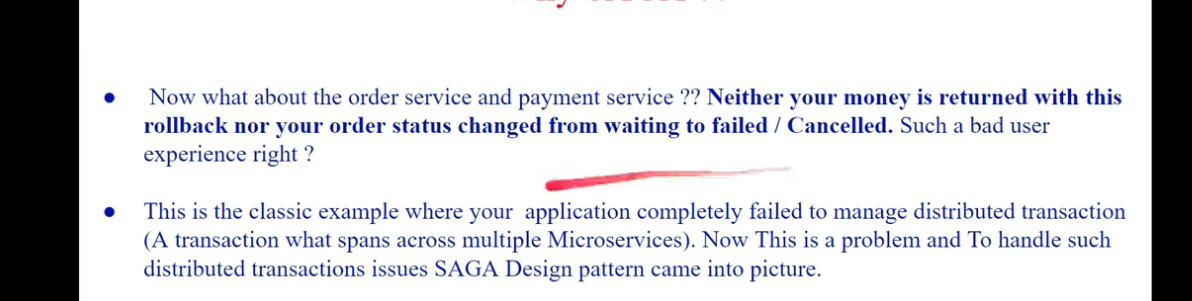
1. When delay in processing in the communication   
   eg. Notification,analytics,reporting
2. When the job at hand is long-running
3. When we have monitoring system and we need to generate the reports.
4. 

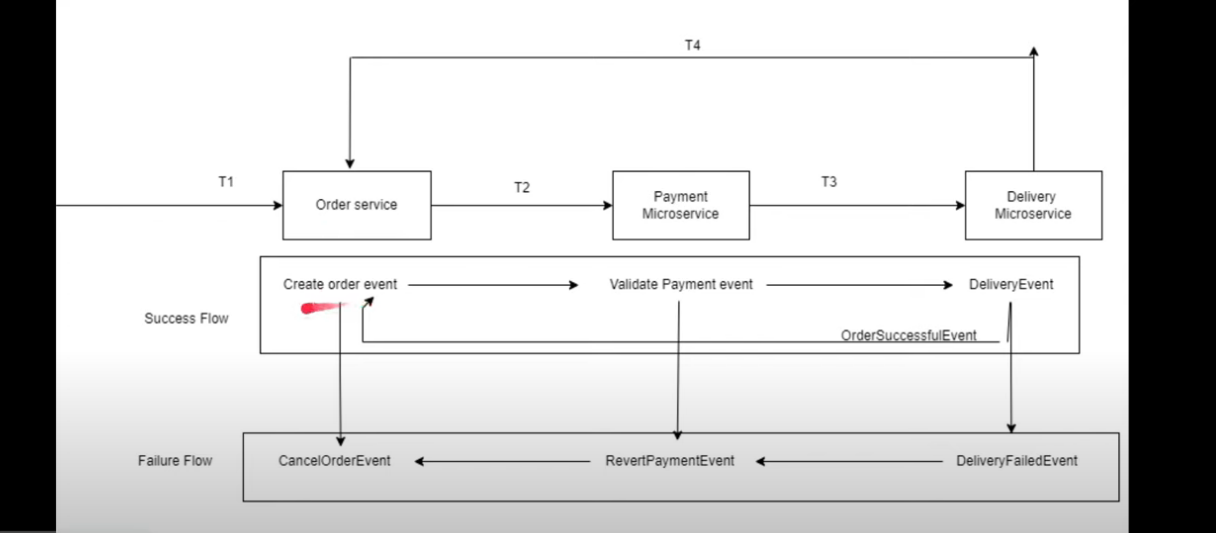
# SAGA Design Pattern:

* When we have a distributed services across the network and each service running on different IP address when
* we have a monolithic Service system then everything is atomic in nature all the tables we have in a single program the if we want to order something using zomato then whole process wrap it up using same application



At some point of time T3 is not executed as partner is not assigned for delivery we can marked as fail but local txn is roll backed in the services of Delivery service but not for Order & Payment txn



How SAGA Change the flow of your execution and handle the events:  


Here we are managing the event between the micro service to have the flow info and have the failure info as well

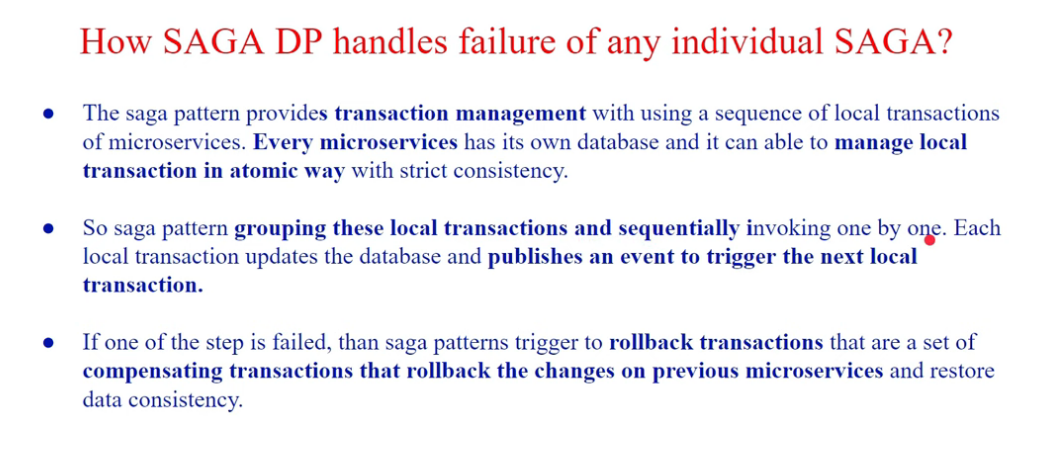
T1: Our execution start likes Order Service called have the local txn perform and generate the order event

T2: Payment Validation happen and generate the payment event

T3: Find delivery partner Not able to find it time out occurs from the delivery service

Now we have the reverse failure event flow where end micro services are the event generator and first one are the consumers of those event

Deliver fails we will generate event of failure into the another queue and let every other service to consume this message and let them roll back.



# Two ways to implement SAGA:

1. Choreography
2. Orchestration

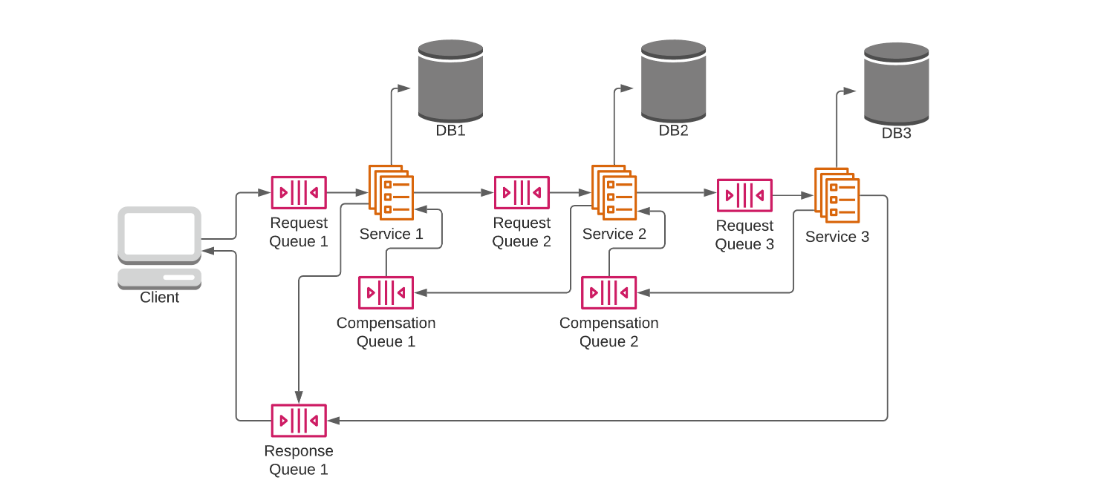
## Choreography Session:

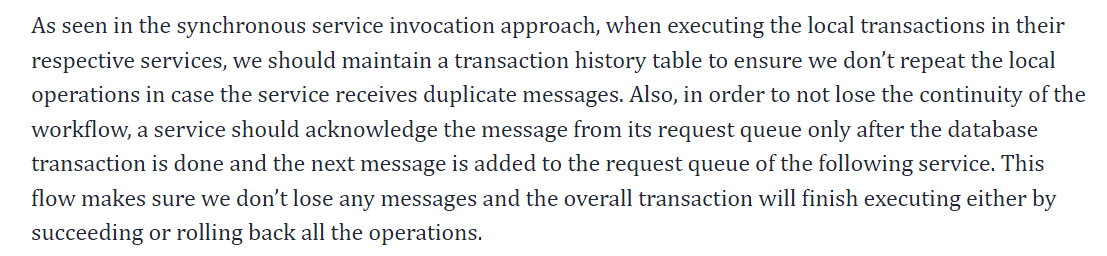
In this approach each of the service is communicate with each other with the help of queue we are maintaining the service operation here

After the successful transaction it will generate the message in the queue for another service 2 and same goes on

If any point of time any service got failed then we will generate the message into the compensation queue to roll back the transaction and it will further give the notice to the other services for giving compensation to the user if the payment is deducted from his/her side

Below is the diagram of the choreography :



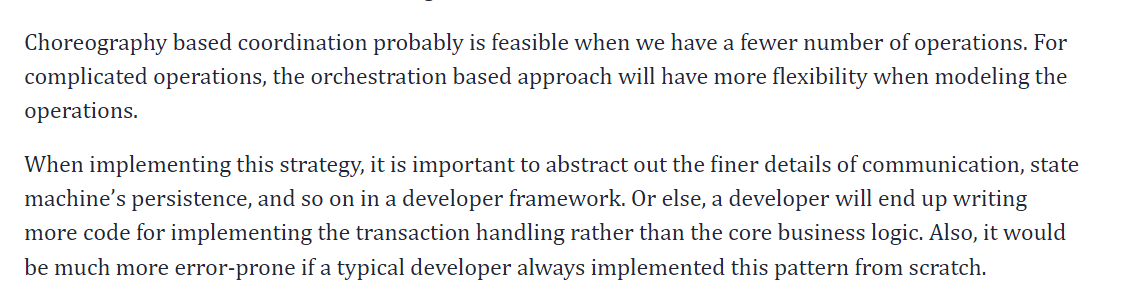


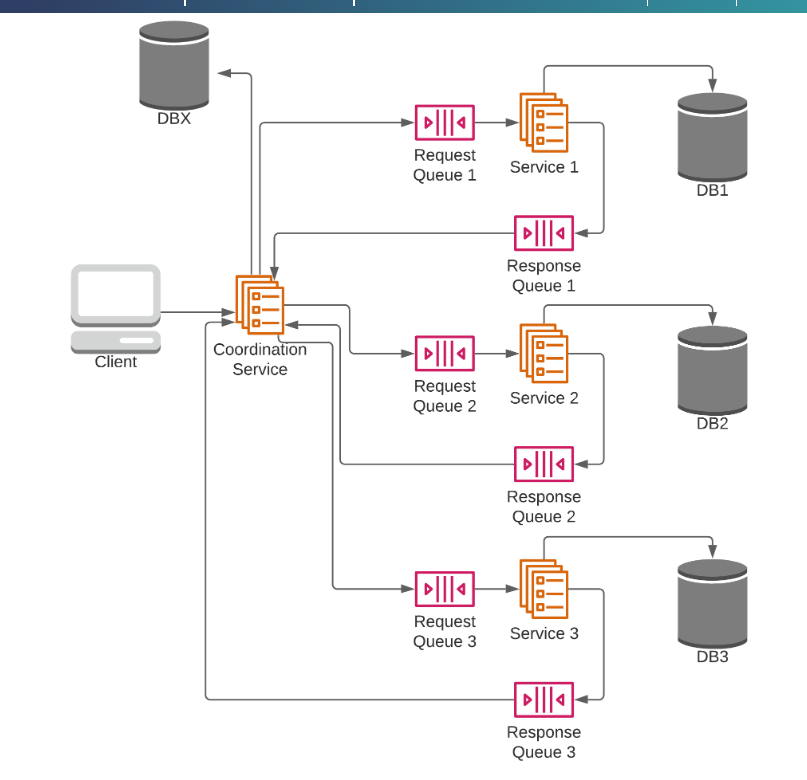
## Orchestration:

It’s a Service where all other services are communicating for further txn of a particular work flow

Advantages:

* Good for complex workflow there can be many participant with the orchestrator and there are new participants as well that can join the same
* Doesn’t introduce the cyclic dependencies
* There is no confusion between the communication of the microservice which one is listen to which command the command give by orchestrator is the only thing that is going to process





# If we have 2 micro services and We have to maintain the Transaction between the services if transaction failed in between how we will able to handle it

**one of the best ways to solve the problem of distributed transactions is to avoid them completely.**

TO ensure that we are having the communication between the micro services and performing the transaction then there are some approaches that we follow to solve this problem:

1. Implement the circuit breaker to check the response time from the called service if the response time convert it into the time out then circuit becomes open and we can conclude and call the “**Fall-back method”**  to give the default and final response to the user

# Use case of Abstract Class over Interfaces:

1. Partial implementation :   
   You have some method that can be implemented by the class and other method needs to implemented by the subclasses
2. Flexiblity in design:

Abstract class give more flexible design as compare to Interfaces They allow for a mix of abstract and concrete method in java

1. **Shared State or Fields**: You need to maintain state or fields that are shared among all subclasses. Abstract classes can have instance variables, constructors, and other features that interfaces cannot contain. This is useful when you have common data or behavior that all subclasses need access to.

# How to maintain the user session in the microservices:

* Centralized Session Management:  
  Implement the services that is responsible for authentication and authorization and validating the user session or the token of that user
* Token Based Authentication:  
  when user logs in to the service then he can generate one token and other services verify the signature of that token and fetch the details from it
* API Gateway:  
  The api gateway can handle the authentication request and validating the user session along with the authorization as well and give us the routing to appropriate service.
* Distributed Caching:  
  We can have the distributed cache for all the microservices where we are maintain the session id with the expiry time for the particular user and we can request the cache for the user session or the state in which state it is and which microservice user wants to access it.

# Relational Model Constraints:

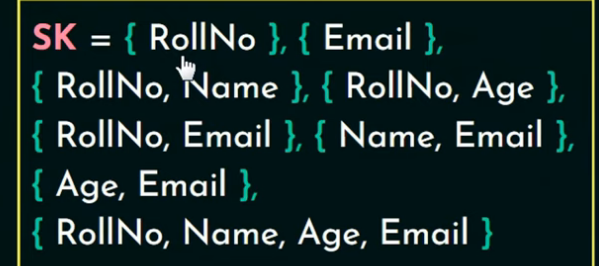
Constraint in database:

1. Schema Based Constraint:

* Must be an atomic Value
* Perform data type check in the give table the age column have the data type integer but it’s storing the value of character which violated the domain constraint.



1. Key Constraints: An attribute that can be uniquely identified each tuple in a relation is called key

* Primary key has to be unique in nature
* Every relationship has at least one SUPERKEY that is a combination of all attributes. 
* A key satisfied 2 constraint two tuple can not have identical value & It is a minimal superkey
* A set of attributes that uniquely identify the tuples in a realtion is called candidate KEY

1. CONSTRAINT ON NULL

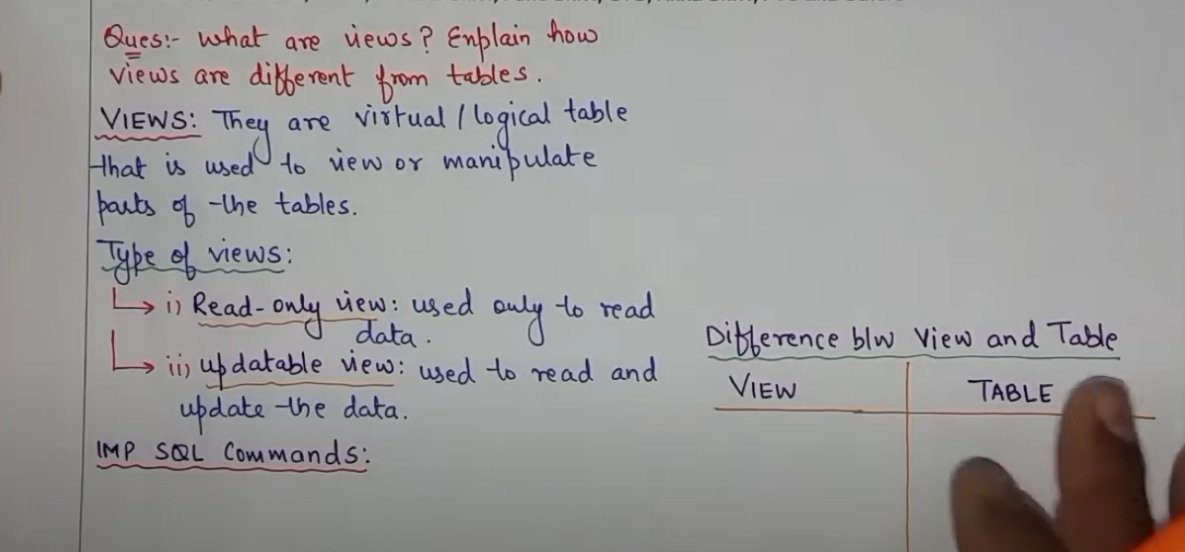
* Specified wether we need null or not if not the define attribute with NOT NULL
* NO Primary key can not have null values

1. REFRENCIAL INTEGRITY Constraint:

This constraint is on 2 tables foreign key concept

* FK must be of same domain in both the table
* In one table FK must be PK

# What Are Views & How it’s different from table:



1. View is a virtual entity no data stores in the physical memory as compare to table

Let say we have a table of Employee( “name”,”Department”,”salary”);

Here I don’t want to give access of this data to all the user so I’ll create users to access this table I’ll grant the access on the table not on the records

USERS: uHr,uAdmin,uFin

**Query: this query will create the view where it will run for department hr only**

**CREATE VIEW vHr as**

**SELECT dept,ename,sal WHERE department = ‘HR’;**

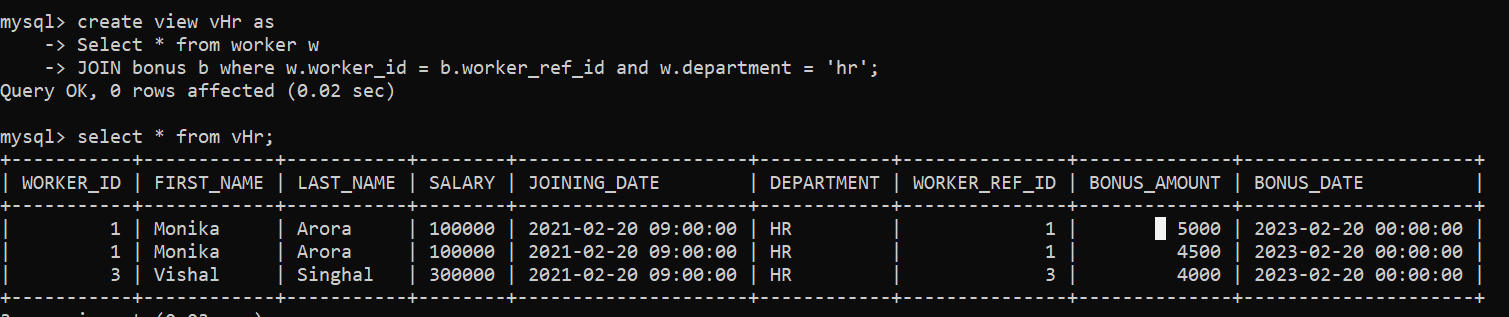
**It provides a consistent way to access the data**

* **At some point of time we can restrict the user to access some sensitive data using view.**
* **Used to extract data from the tables**

**Type of views:**

1. **Read Only View: it’s allowed to read the data no manipulation in the table**
2. **Updatable view : Used to view read and update the data**

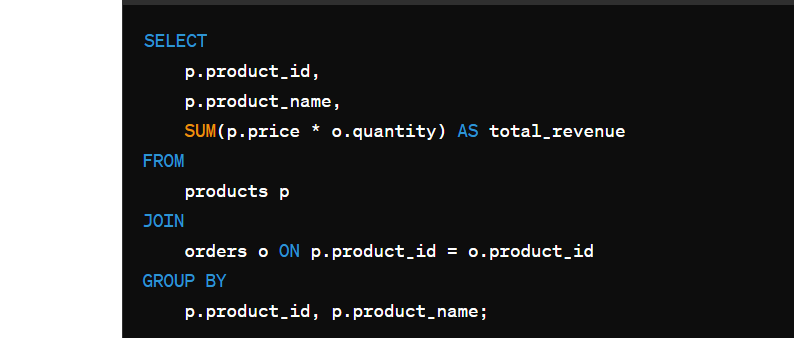
**View on JOIN:**

****

# SQL Queries:

𝗝𝗢𝗜𝗡 𝗢𝗽𝗲𝗿𝗮𝘁𝗶𝗼𝗻𝘀:

→ Given a table products (columns: product\_id, product\_name, price) and a table orders (columns: order\_id, product\_id, quantity), write a SQL query to find the total revenue generated by each product.

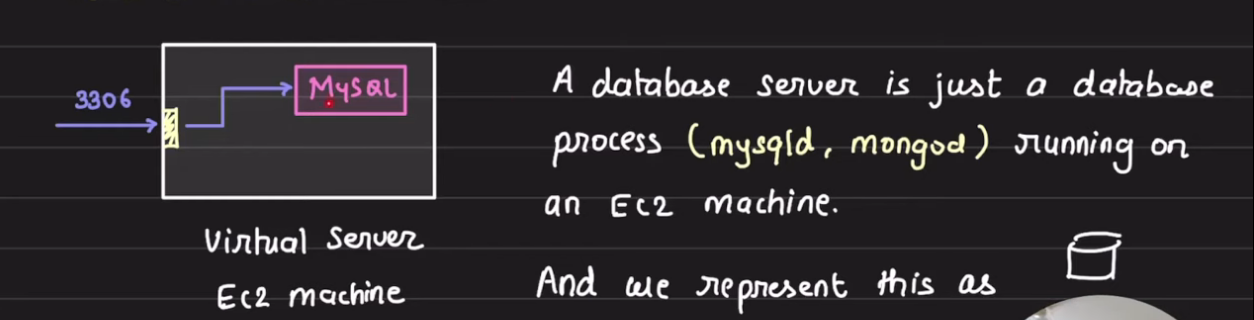
* 

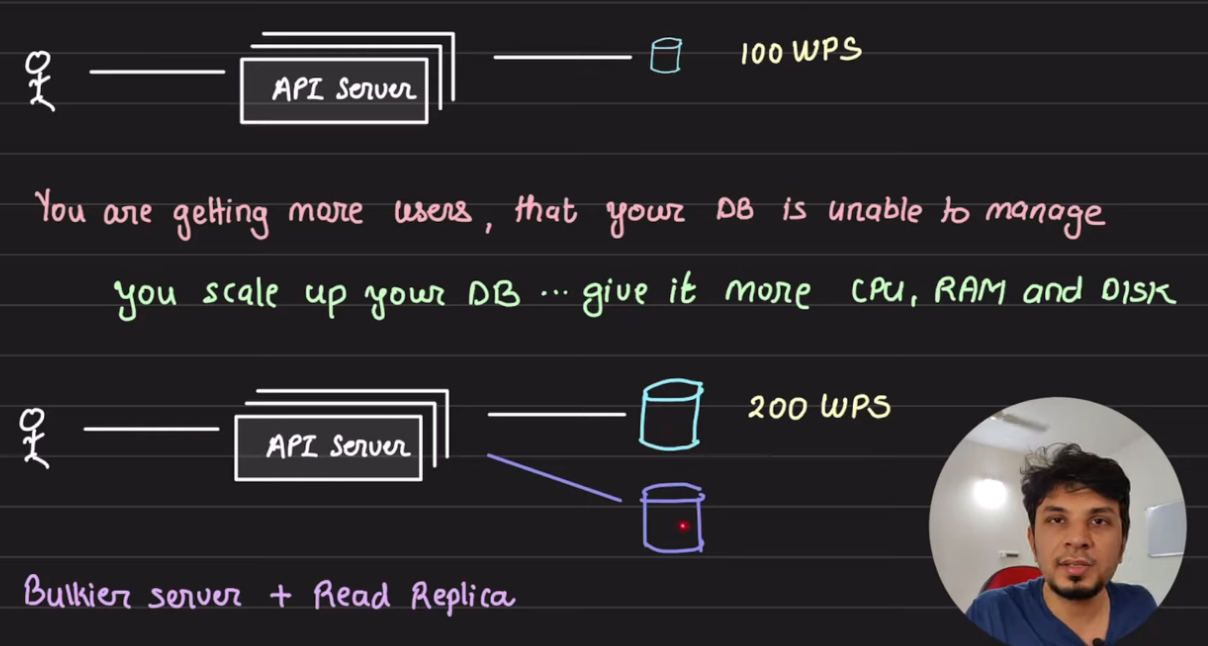
# Data base partitioning:

Sharding: Method of Spliting the data across the machines

Partitioning: splitting a subset of data within the same instances.

## How db scaled?





This above is classic master – slave architecture

Here we have Vertical scale the system for 200WPS (Writes per second)

And added the read replica so that the read request goes to the replica server

Now the load increase and we can’t grow the Vertical scaling as hardware limitation reach at 1000wps

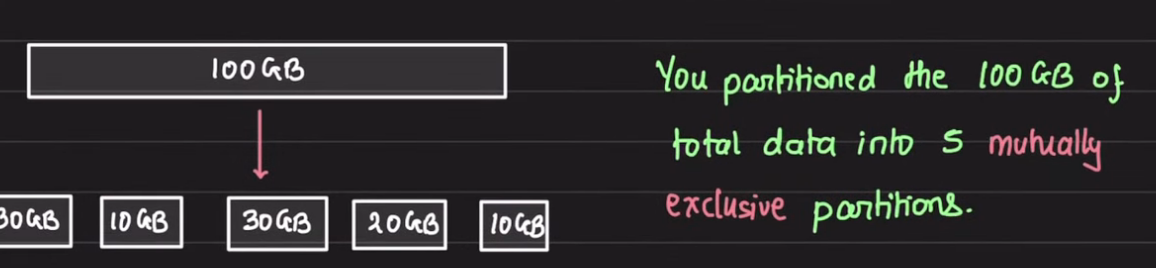
Now the solution is “**Horizontal Scaling”**



By adding one node database server we reduced the load to 750WPS

On each node and thus higher throughput

So this node is known as “SHARD” while data is partitioned

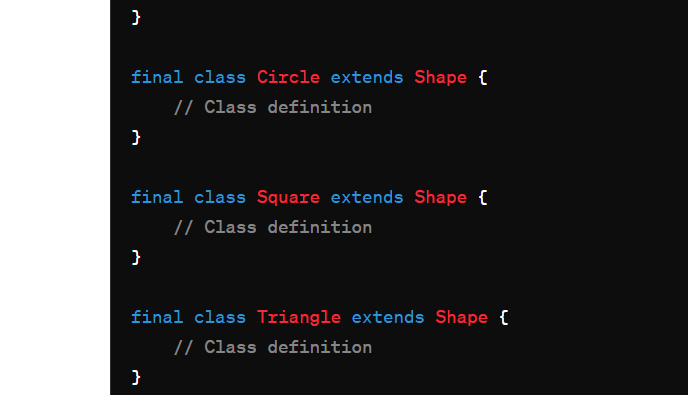


# JAVA 17 latest feature to be introduced:

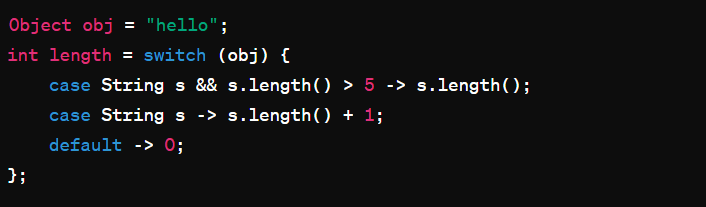
1. Sealed Classes:

Public sealed class Shape permits Circle,Square,Triangle{

}



1. Switch Pattern matching:  
   Pattern matching for switch statements simplifies code by allowing you to combine declaration and conditional extraction in a single step. This feature enhances readability and reduces boilerplate code when working with complex conditional logic



# AWS Interview Question:

AWS people have their own data centre



Accordingly we can increase and decrease the server

Services:

* S3
* EC2
* Billing and cost management
* Support
* RDS(relational Database service)
* Cloud formations

**S3** is storage bucket which help us to store the huge amount of data into the virtual storage room

**EC2** it’s an instance where our deployed service will run

**Amazon VPC**: virtual private cloud where I need to create one private cloud and aws allows that where we can deploy our services that is not accessible to you publically

Enable us to define subnet & routing

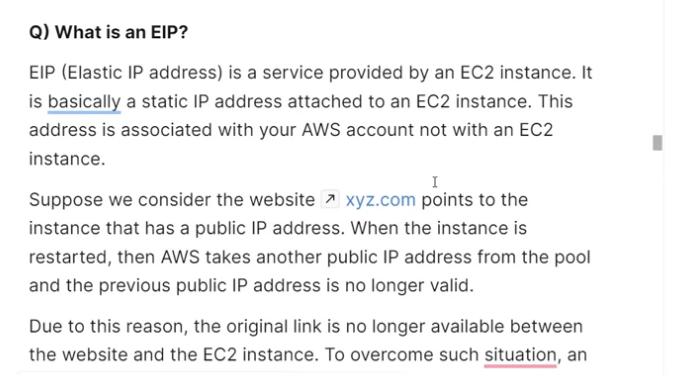
**Amazon IAM**: Identity access management with the help of this we can manage the users to access the aws account

**AWS Lambda**: Lambda is serverless computing service that enables you to run your code without provisioning or managing servers.

Lambda automatically handles the infrastructure and scales it based on incoming requests.

**AWS SNS:** Simple notification service that allows you to send the message to subscribed endpoint or clients. It supports various delivery protocols including email, sms, push notification, & HTTP/HTTPS end points.

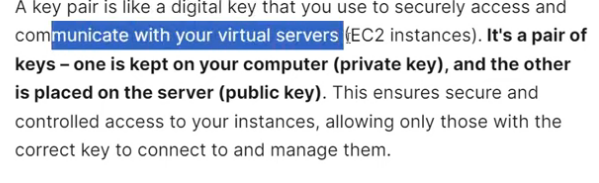
**AWS Cloud Watch:** Cloud watch is a monitoring tool to our application or services which track and monitor the log if the exception comes into our application and give us the notification to the dev team or product team.



## What are KEY – PAIRS ?

An amazon EC2 instance use public key cryptography which Is used to encrypt and decrypt the login information.

In public key that is used to encrypt the information while at the server side private key is used to decrypt the data.



# How to design Microservices:

1. Start with a monolithic services
2. Split the services in micro services
3. How services are going to communicate Synchronous or asynchronous communication
4. We need to focus on fault tolerance if service is down, we can give the response for the same
5. To test the same end to end Integration testing
6. CI/CD for deployment

Scenario where we should go Micro Or MONO:

1. Ecommerce Site we need that site to be scalable as more user are coming to the website and we need to handle the load on the different service and we have tha flexibility of different services.

**How do we make sure to handle the increasing load of users:**

1. Load balancing
2. Horizontal scaling : Add more instances of the services
3. Auto scaling: Provide by the Cloud providers
4. Caching: to store the frequent access data instead of quering every time
5. Database scaling:
6. Asynchronous processing:

# Comparable & Comparator:

* The most fundamental difference between the comparable & comparator in java is the number of sorting sequence they support

1. Comparable: Supports single sorting sequence logic on an object   
   natural ordering (Often define by a single field).
2. Comparator: supports multiple sorting sequence order based on your custom logic that you will define

In comparable we have CompareTo() & Comparator we have compare().

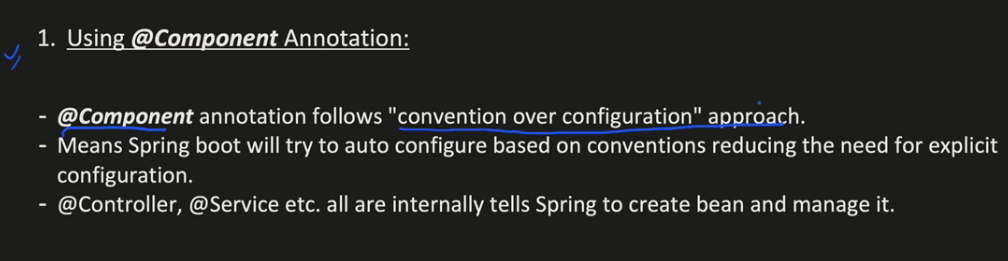
* Uses the compare(Object o1, Object o2) method to compare two objects and return a negative integer if the first is less than the second, zero if they are equal, and a positive integer if the first is greater.
* Implemented by the class itself using the compareTo(Object o) method.

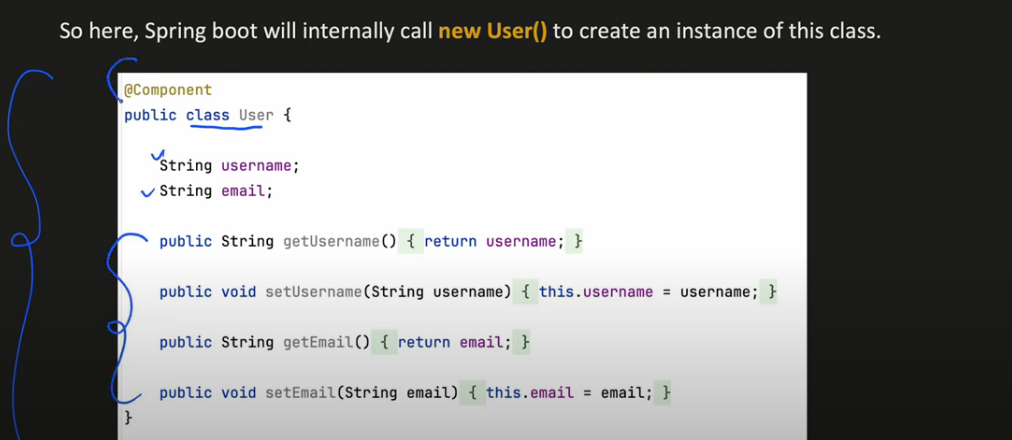
# What is bean & everything about bean

IOC container contains all the bean which get created and managed them.

2 ways to create beans:

* @Component
* @Bean





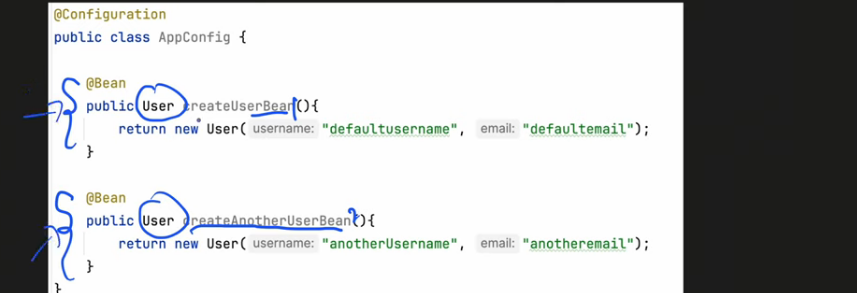
**If we make the argument constructor and we don’t have the default constructor so either we get the error there is no default constructor**

**Or we can have @Bean annotation let see the given example below:**

****

For this we can have our own custom bean configuration for that particular user class

## What if we had 2 beans of same classes?



2 object will be created for the same and managed by IOC only

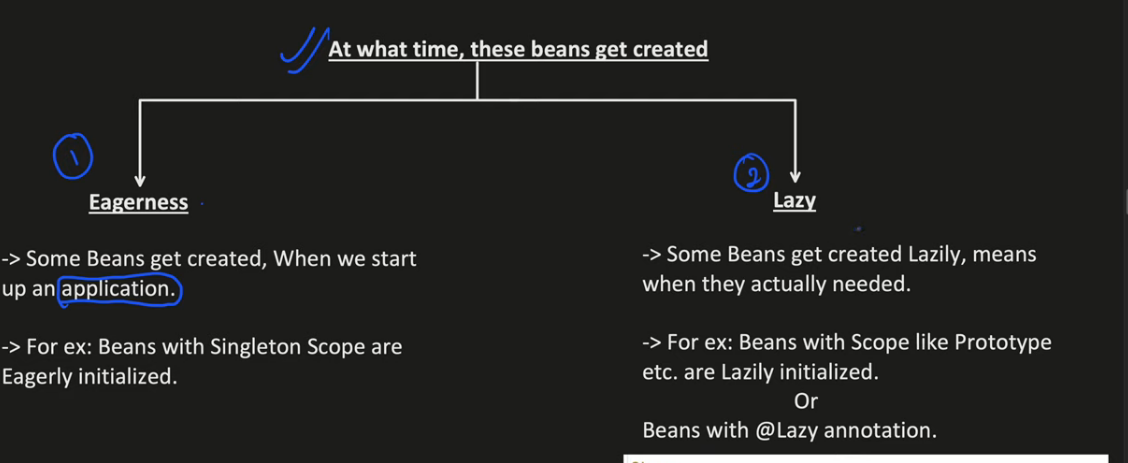
We can have the @Qualifier annotation to use which bean we have to use.

## How spring boot find these beans?

@ComponentScan(basepackages=””)

This annotation will scan all the packages & Sub package and look into the annotation of the package and start creating the bean for the classes.

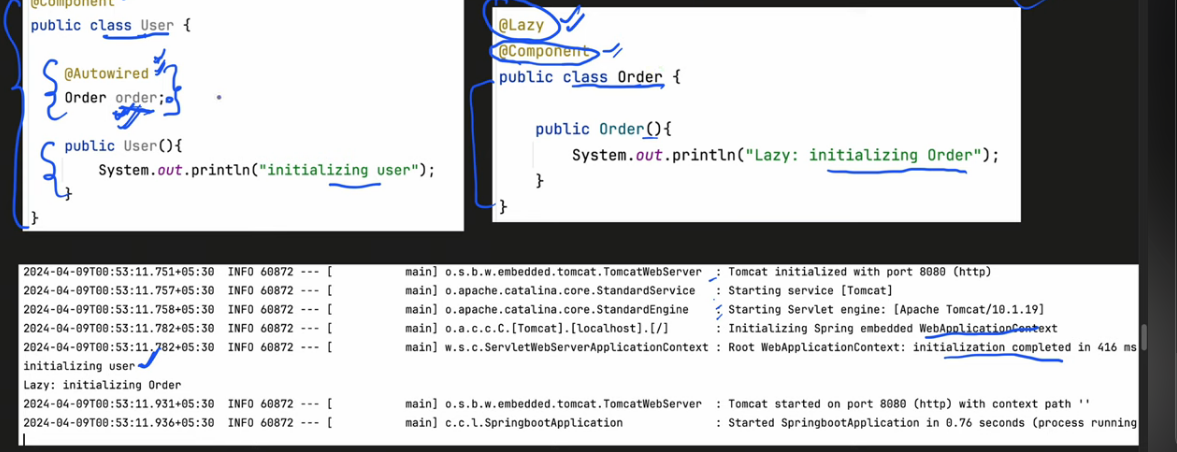
@Configuration also help us to find the beans from the classe.



## Life cycle of a bean:

Application Start 🡪 IOC container 🡪 Construct Bean 🡪 Inject dependenies int the constructef bean

* @Postconstruct 🡪 Use the bean 🡪 @PreDestroy 🡺 Bean Destroyed.



First IOC inject the dependency of the class and then it will inject the other dependent class

# Dependency Injections

# Native Query in Spring data Jpa & SQL Queries

**@Query(“Select e From Employee e”)**

**Public List<Employee> findAllEmployee();**

Example 2 : JPQL Queries with parameter  
**@Query(“Select e From employee e WHERE e.department = :department”)**

**List<Employee> FindByDepartment(@param(“department”) String department)**

## Write A query to fetch the 1:M mapping details from Emp ----- Project

Select \* from Employee e

JOIN emp\_project ep On e.emp\_id = ep.emp\_id

JOIN Project p ON p.proj\_Id = ep.proj\_id

Where e.Department = ‘IT’;

# Spring Boot Dynamically Initialized Values

Unsatisfied dependency will be solved by @qualifier annotations

Class User{  
  
@Autowired

@Qualifier(“onlineOrder”)  
private Order order;

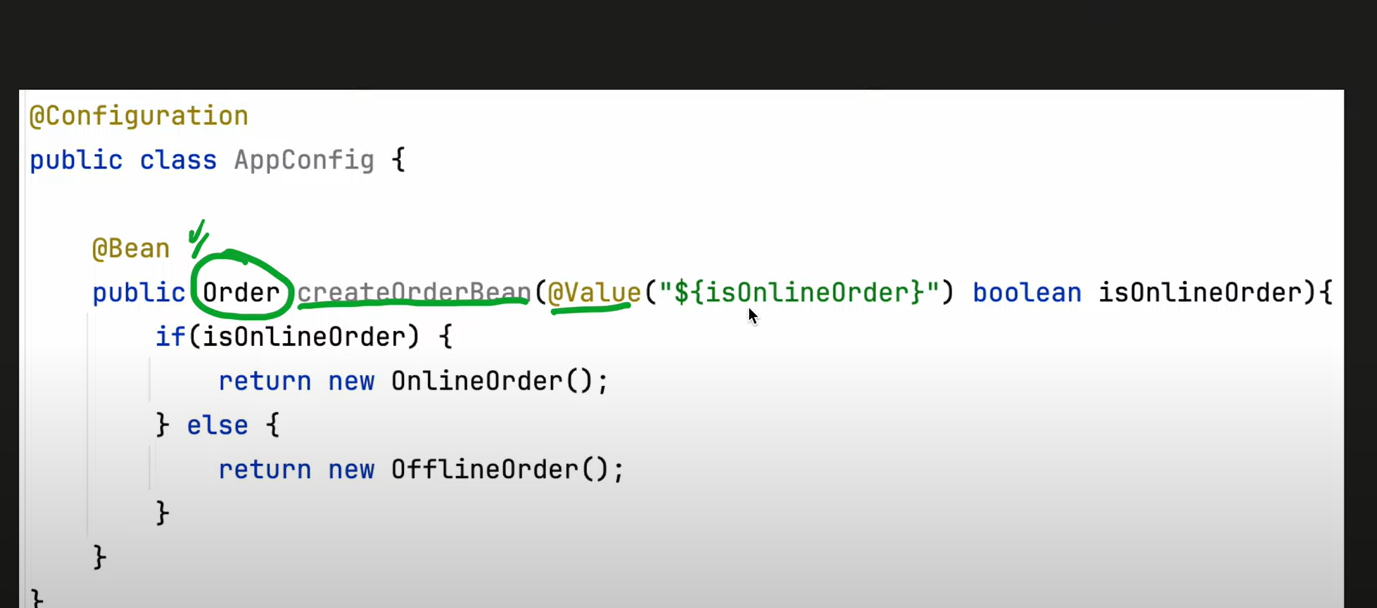
}

In the above code we made the dependency inversion fail as we always get the Online order impl what if we want offlineImpl in Order Service..

Order Interface has 2 implementation one is OfflineOrder and OnlineOrder

**Solution** we will create 2 object of Order with different Qualifier name.

One more solution is this to use @Value Annotation and in the configuration we will take the bean of order according to our requirement if isOnlineOrder is false then we will get the offlineOrder.



From the properties @Value(“${isOnlineOrder}”) it will read it and give us the bean

The flow of the program is like first it will create the User class bean then check the dependency ->   
after that it will check for that dependency, we have configuration yes, we have then it will read the value and create the bean according to the properties.