Microservices , Agile & Jira created by mayuresh ratnaparkhi

==========

======Microservices Architecture======

==========

Microservice :- Every single service is a different component with its dedicated database and there is an API Gateway to guard the client apps and Route them. microservices are Loosely coupled.

Microservices are an architectural style that develops a single application as a set of small services. Each service runs in its own process. The services communicate with clients, and often each other, using lightweight protocols, often over messaging or HTTP.

The fundamental of microservices are below key concepts which define how microservices are built and modeled.

1. **High Coheshion & Low Coupling** :--

   Coheshion :-- means (every functionality ,features, business logic related to respecitive service) should be close to that respective service or inside the service.

   Example :-- Suppose there is an Order service, so every functionality ,features, business logic related to Order service should be close to that respective service or inside the service. That means service is cohesion. and that microservice must be high Coheshion.

   coupling :-- in the microservices architecture services do talk to each other. when they do talk to each other. they should not be coupled with each other.

   coupling means if the services are too dependent on each other or they know too much about each other or they share a common data source or a database that would mean that the services are coupled with each other.

2. **Independent Deployablity :--** independent deployability that means , that you should be able to deploy your services independently.

3. **Independent Scaling :--** with independent deployability we also get a feature with microservices which is independent scaling. **So scale up and scale down as per request traffic.**

4. **Business Domain Modelling :--** micro Services architecture the services are divided as per the business domain or business features. **each service will deal only related it's business needs.**

Example :-- suppose we have Order service, so Order service will only deal with anything related to order like placing order or sending order to payment service. or mainitaing the stae of order like (order is accepted, rejected or delivered)

5. **Observability :--** Since you have different services, so it will be very easy and helpful to observe each microservice and can trace or find out that the respective microservice is working properly not.

6. **Resilience :--** in case of microservices if one of the service is degraded or one of the databases is down the other service is not affected. for that you need to implement fault tolerance in your service. so that the failures do not Cascade or travel across the service.

with this you can actually just focus on fixing the broken part of your services and the rest of the services will continue working without any issue.

7. **Reusability :--** in micro Services architecture, whatever component we creates that can be re-usable in other micro Services.

==========

======SPRING BOOT MICRO SERVICES======

==========

Microservices Architecture: - In a microservices architecture, **the application is divided into small, independently deployable services.** Each service focuses on a specific business capability and communicates with other services through APIs.

Microservice has almost **6 major components**, which are required for building a scalable microservices environment

1. Service Discovery & Service Registry
2. Load Balancer
3. API Gateway
4. Circuit Distributed Tracing Breaker
5. Service Monitoring Distributed Tracing Breaker (Sleuth)
6. Configuration Management Config Server

Service Discovery & Service Registry (Eureka): - Spring Cloud Eureka is a service registry and discovery server. It allows microservices to register themselves and discover other services in the ecosystem. This enables dynamic scaling and routing.

Load Balancer :- Netflix's Ribbon is a load-balancing library that provides client-side load balancing for RESTful applications. Using Ribbon with Spring Boot microservices can improve service availability, scalability, and fault tolerance.

Note :- Resilience4j is a lightweight fault tolerance library that provides a variety of fault tolerance and stability patterns to a web application.

API Gateway :- Spring Cloud Zuul acts as an API gateway that manages the routing, filtering, and load balancing of requests to different microservices. It helps simplify the client-side communication with multiple services. It acts as gatekeeper to the outside world, not allowing any unauthorized external requests.

Circuit Breaker (Hystrix):- Spring Cloud Hystrix helps prevent system failures due to service outages. It provides a circuit breaker pattern to handle and manage failures gracefully.

Service Monitoring Circuit Distributed Tracing Breaker (Sleuth):- Spring Cloud Sleuth provides distributed tracing to track requests across various microservices. It helps diagnose and troubleshoot performance issues.

Configuration Management (Config Server):- Spring Cloud Config Server manages externalized configuration for microservices. It allows you to centralize configuration properties for different environments and services.

==========

======The monolith design pattern======

==========

Monolith :-- A monolith, in the context of software architecture, refers to a single, unified application where all the components and functionalities are tightly integrated and operate as a single unit.

Key Characteristics of a Monolith

Single Codebase: -- The entire application is developed and maintained in a single codebase. All components, such as the user interface, business logic, and data access layers, are part of the same codebase.

Unified Deployment: -- The application is built, tested, and deployed as a single unit. Any change, even a minor one, requires redeploying the entire application.

Tightly Coupled Components: -- Components within a monolith are often tightly coupled and dependent on each other, making it challenging to modify or replace individual parts without affecting the whole system.

Shared Database: -- A monolithic application typically uses a single, centralized database for storing data.

==========

======The Strangler design pattern======

==========

The Strangler design pattern is a technique used to incrementally migrate a monolith system to a new system by gradually replacing parts of the old system with new functionality. This pattern allows for a smooth transition without the need for a complete system overhaul at once, reducing risk and ensuring continuous operation.

Key Concepts of the Strangler Design Pattern

Incremental Migration: -- Instead of rewriting the entire legacy system in one go, parts of the system are gradually replaced with new code.

Parallel Operation: -- Both the legacy system and the new system run in parallel during the transition. This allows for testing and validation of the new components without disrupting existing functionality.

==========

======The SOLID design principles======

==========

The SOLID design principles are a set of five guidelines that help developers create more maintainable, understandable, and flexible software systems.

1. **Single Responsibility Principle (SRP) :--** "a class should have only one reason to change" which means every class should have a single responsibility or single job.
Any classes you create will have only one responsibility. And that responsibility must be followed by that particular class.

Example :-- suppose there is BankService class this BankService class will performs only Bank Service related functionality like (Withdraw, Deposit, Print Pass Book, Get Loan Info,Send OTP).

Purpose: This principle encourages a modular design where each class focuses on a single functionality, making the system easier to understand, maintain, and test.

2. **Open/Closed Principle (OCP)** :-- This principle states that software components should be easily extendable for new features but without modifying their existing code. **Any Class or Any Interface you create should not be open for modification, it should be close for modification. Whatever extra functionality that you need to add by extending the class or interface.**

Example :--
```
public class SpeedCalculator {
   public BigDecimal calculateSpeed(Vehicle vehicle) {
      if (vehicle instanceof Car) {
         // Calculate Speed here…
      } else if (vehicle instanceof Airplane) {
        //Calculate Speed here…
      } else if (vehicle instanceof Bicycle) {
       //Calculate Speed here…
      }
   } }
```

There is a problem with the above code, what if there is a new requirement to calculate the speed of trucks and other vehicles, you have to add lots of if conditions in the calculateSpeed method. There may be a chance that you will introduce some new bugs while adding the changes. This type of code change leads to poor code readability and is very hard to maintain.

We can rewrite the above code to ensure code extensibility and reusability.

```
public interface Vehicle{
   public BigDecimal calculateSpeed();
}
public class Car implements Vehicle {
   public BigDecimal calculateSpeed() {
     //calculate speed to car
   }
}
public class Airplane implements Vehicle {
   public BigDecimal calculateSpeed() {
      //calculate speed to car
   }
}
```

Purpose: This principle helps in building systems that can be easily extended with new features without changing existing code, thus reducing the risk of introducing bugs.

3. **Liskov Substitution Principle (LSP)** :-- In object-oriented programming, we always try and use inheritance, even if it's not really the right thing to do just for the sake of reusing the code, but in LSP **subtypes should be replaceable by their base types** , means you should able to swap / substitute the child of parent object without breaking code.

Note :- You should always be careful when you use inheritance, you should only inherit when your superclass is replaceable by subclasses in all instances.

Liskov principal says that whenever you create parent child relationship that is the parenet class and the sub type class. So both the class should be interchangable and should be substitutable, that means whenever , we pass the object of child or whenever we pass the object of parent , both code should work.

Guidelines

- Every Child class should be completely substitutable of there base class.
- No new exceptions can be thrown by the subtype
- Clients should not know which specific subtype they are calling
- New derived classes just extend without replacing functionality of Old Classes.

4. **Interface Segregation Principle (ISP)** :-- The principle states that the larger interfaces split into smaller ones. Because the implementation classes use only the methods that are required. We should not force the client to use the methods that they do not want to use.

Purpose: This principle encourages creating smaller, more specific interfaces so that classes only need to implement methods that are relevant to them, reducing unnecessary dependencies.

5. **Dependency Inversion Principle (DIP) :**-- The principle states that we must use abstraction **(abstract classes and interfaces)** instead of concrete implementations.

We should create the interface in such way that those interfaces should have a client required methods only and that needs to be implemented. don't create interface which have un necessary and un-required methods.

Purpose: High-level modules should not depend on the low-level module but both should depend on the abstraction. Because the abstraction does not depend on detail but the detail depends on abstraction.

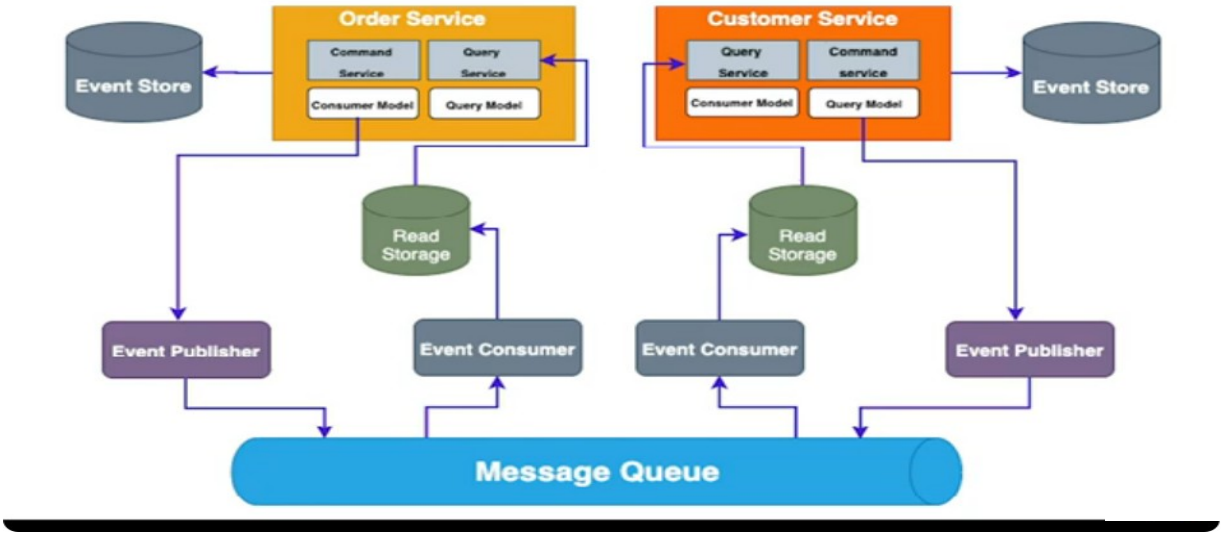
==========

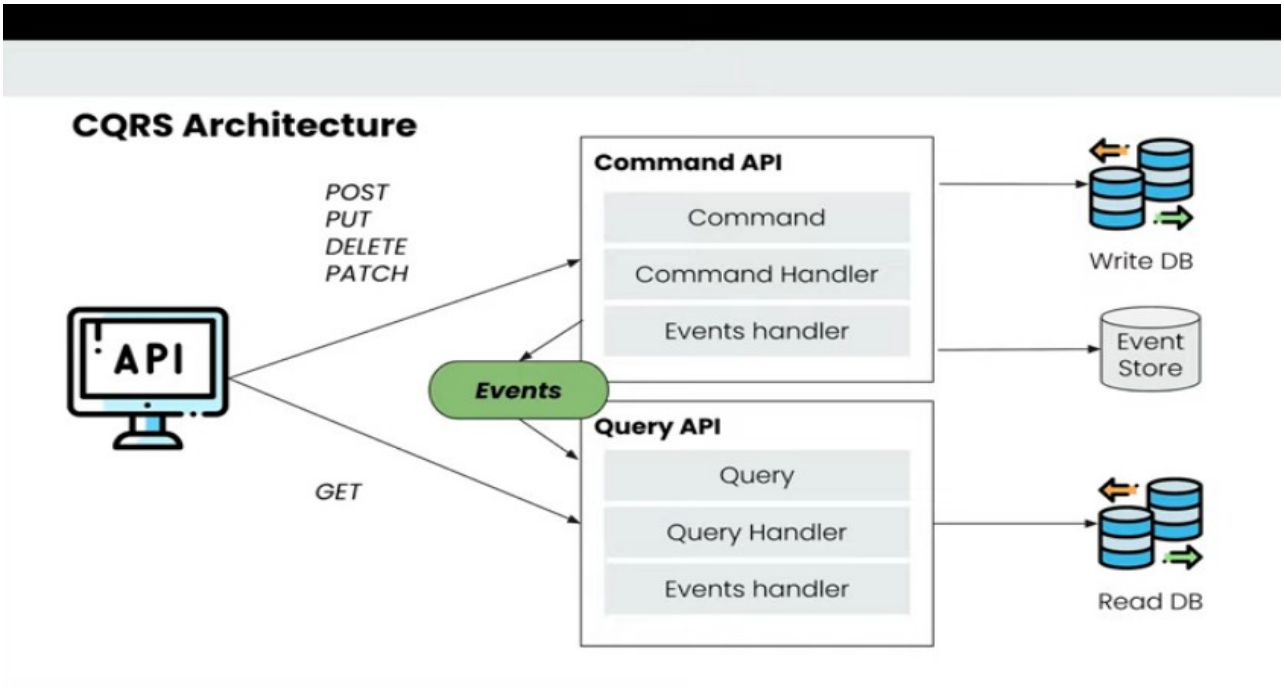
======**The CQRS design pattern**======

==========

CQRS **(Command Query Responsibility Segregation)** is a design pattern used in software architecture that separates the operations that read data (queries) from the operations that update data (commands). This pattern is especially useful in complex systems where read and write operations have different requirements for performance, scalability, and security.

Note : - We implement CQRS using **Event Driven Microservices and use Event sourcing to interact each other.**





Key Concepts:

Command and Query Separation:

Command: An operation that changes the state of the system. Commands are typically write operations such as create, update, or delete.

Query: An operation that retrieves data without modifying it. Queries are read-only operations that do not affect the system's state.

Advantages of CQRS:

Improved Performance: By optimizing reads and writes separately, you can achieve better performance.

Scalability: Easier to scale parts of the system independently.

Maintainability: Separation of concerns leads to cleaner, more maintainable code.

Flexibility: Allows for different models and storage mechanisms for reading and writing data.

==========

======The Saga Choreography design pattern======

==========

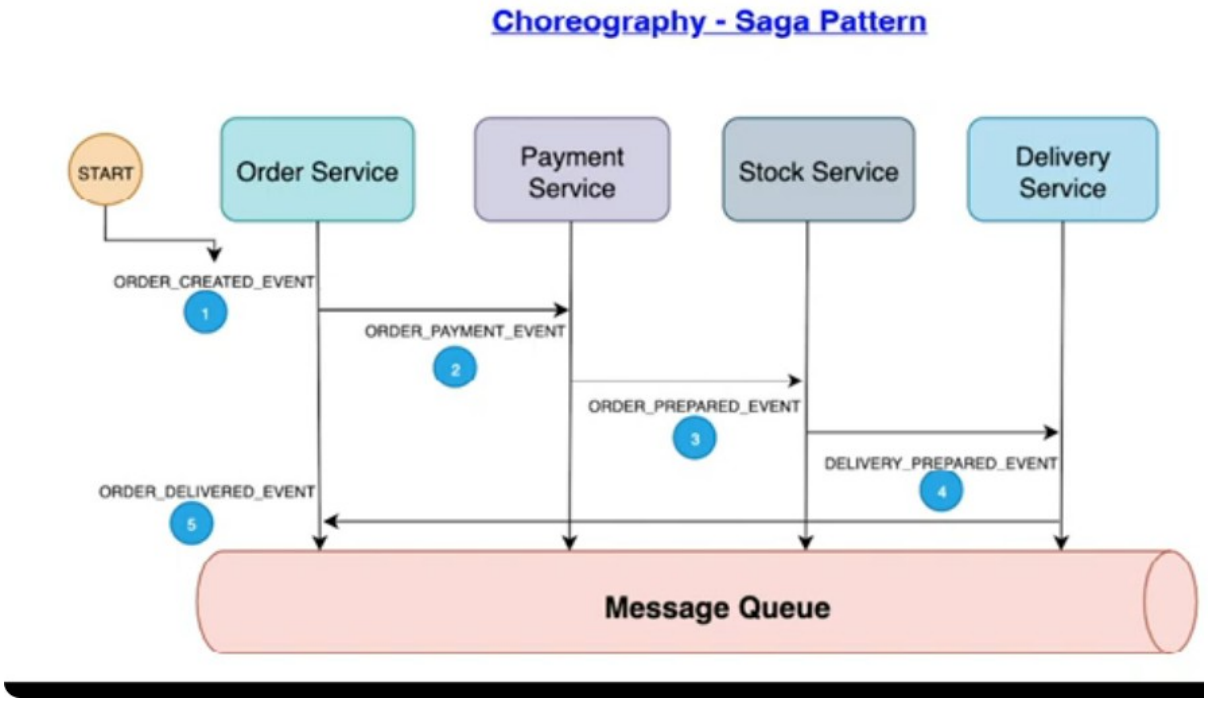
Saga pattern is a sequence of local transactions across micorservices, where each transaction updates data and publish the message or event to next local transaction.

Usages :-- Saga is used for Long Live **distributed transactions**

The Saga Choreography design pattern is a distributed transaction management pattern used in microservices architecture to ensure data consistency across multiple services. It is especially useful when traditional ACID transactions are not feasible due to the distributed nature of microservices. In the context of the Saga pattern, choreography refers to a decentralized approach where each service involved in the saga is responsible for

Choreography means the tasks execute independently. Once one task is completed, it invokes the next tasks in the sequence. In case if the next task fails then it invokes compensating task for the previous tasks.
coordinating its own actions and reactions to events.

Choreography doen't have central co-ordination, each service produces and listen to other services event and decides action should be taken or not.
Each local transactions publish the domain event that trigger local transaction in other services.



Key Concepts

Saga: A saga is a sequence of transactions that updates multiple services. Each transaction in a saga is called a step, and each step has a corresponding compensating transaction to undo its effect in case of a failure.

Choreography: In the choreography-based saga, there is no central coordinator. Instead, services communicate through events to inform each other about the completion of transactions and the need to perform compensating actions if necessary.

How Choreography-Based Saga Works

Event-Driven Communication: Each service publishes events when it performs an operation. Other services subscribe to these events and react accordingly.

Local Transactions: Each service performs a local transaction and publishes an event if the transaction is successful.

Compensating Transactions: If a service fails, it publishes an event indicating the failure. Other services that have already completed their transactions in response to the initial event will listen to this failure event and perform compensating transactions to revert the previous actions.

==========
======The Saga Orchestration design pattern======
==========

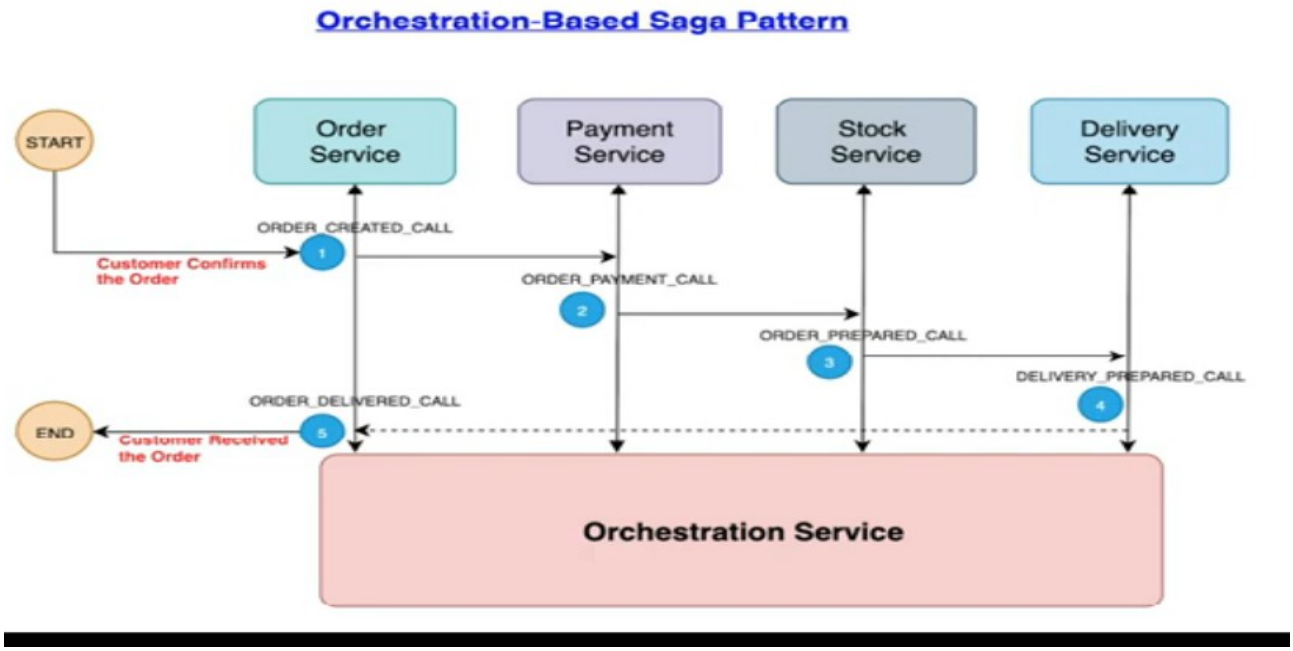
Saga pattern is a sequence of local transactions across micorservices, where each transaction updates data and publish the message or event to next local transaction.

Usages :-- Saga is used for Long Live distributed transactions

Saga Orchestration is a design pattern used to manage distributed transactions in a microservices architecture. Saga Orchestration involves a central orchestrator that coordinates the entire transaction process across multiple services.

Orchestration means the tasks are invoked by another parent task. It plays the role of an orchestrator.
It calls each tasks in sequence and based on their response decides whether to call the next task or the compensating tasks.

In Orchestration base saga co-ordinator service is responsible for centralizing the saga's decision making and sequencing business logic. this co-ordinator service also called Orchestrator service that tells the participant what local transactions to execute.



Key Concepts

Saga: A saga is a sequence of operations that update multiple services. Each operation is a step in the saga, and each step has a corresponding compensating operation to undo its effect in case of a failure.

Orchestrator: The orchestrator is a central controller that manages the saga. It invokes the necessary operations in each service, listens for their completion, and triggers the next steps or compensations as needed.

How Orchestration-Based Saga Works

Centralized Control: The orchestrator starts the saga by invoking the first service operation. It waits for the result and, based on the outcome, decides the next step.

Step-by-Step Execution: The orchestrator sequentially triggers each step of the saga. If any step fails, the orchestrator triggers the compensating operations for the previous steps to maintain consistency.

Compensating Transactions: If a step fails, the orchestrator initiates compensating transactions to undo the effects of the previous successful steps.

==========
======Object-Oriented Analysis and Design (OOAD)======
==========

Object-Oriented Analysis and Design (OOAD) is a software engineering approach that uses object-oriented programming principles to design systems. It involves analyzing and designing a system by modeling it using objects, which are instances of classes, encapsulating both data and behavior. **OOAD helps in building complex software systems that are modular, maintainable, and scalable.**

Key Concepts

Objects: Object is a instance of class and object has **state and behaviour.**
state :- state means properties (class variables)
behaviour :- behaviour means (class functions/methods)

Classes: Class is a blueprint representation of an object, not the object itself. Class is collection of State & Behaviour.

Encapsulation:

- The whole idea behind encapsulation is to hide the implementation details from users.
- If a data member is private it means it can only be accessed within the same class.
- No outside class can access private data member (variable) of other class.
- Using public getter and setter methods we update and reads the private data fields
- The outside class can access those private data fields via public methods.
- The private fields and their implementation hidden for outside classes.
- That’s why encapsulation is known as data hiding.

Inheritance: Inheritance is the **"is a releationship"** where A subclass inherits all the members **(fields, methods, and nested classes)** from its superclass. **extends is keyword which is used for inheritace.**
IS-A relationship based on Inheritance, which can be of two types (Class Inheritance or Interface Inheritance).
Has-a relationship is composition relationship which is productive way of code reuse.

Polymorphism: Polymorphism means to process objects differently based on their data type.
one method can be defined multiple times with different data types.

Polymorphism could be static and dynamic both.

Overloading is static polymorphism :- Overloading in simple words means two methods having same method name but takes different input parameters. This called static because,which method to be invoked will be decided at the time of compilation

Overriding is dynamic polymorphism :- Overriding means a derived class is implementing a method of its super class.

Types of polymorphism :- There are two types of polymorphism in java

Runtime polymorhism(Dynamic polymorphism) :-- Method overriding is a perfect example of runtime polymorphism.
in method overriding both the classes(base class and child class) have same method, compile doesn’t figure out which method to call at compile-time.
In this case JVM(java virtual machine) decides which method to call at runtime that’s why it is known as runtime or dynamic polymorphism.

Compile time polymorphism (static polymorphism) :-- Method overloading is a perfect example of runtime polymorphism. class can have more than one methods with same name but with different number of arguments or different types of arguments or both. compiler is able to figure out the method call at compile-time so it known as compile time polymorphism.

Abstraction: In Java Abstraction is achieved using Abstract classes, and Interfaces.
Abstraction is a process of hiding the implementation details and showing only functionality to the user.

OOAD Process

Object-Oriented Analysis (OOA): Focuses on understanding the problem domain by identifying the objects and their interactions within the system. Involves gathering requirements, identifying use cases, and creating conceptual models.

Object-Oriented Design (OOD): Focuses on designing the solution domain by defining how the system will be implemented using object-oriented principles. Involves creating detailed class diagrams, defining object interactions, and refining the conceptual model into a logical design.

Benefits of OOAD

Modularity: OOAD promotes modularity by encapsulating data and behavior within objects.
Reusability: Inheritance and polymorphism enable code reuse and reduce redundancy.
Maintainability: Encapsulation and modular design make the system easier to maintain and update.
Scalability: Object-oriented design allows for scalable and flexible systems that can evolve with changing requirements.

OOAD is a systematic approach that leverages object-oriented principles to create robust and maintainable software systems. By focusing on objects and their interactions, OOAD facilitates the development of software that closely aligns with real-world scenarios.

==========

======Microservices Design Patterns======

Designing microservices effectively requires understanding and applying various design patterns and principles that ensure the system is scalable, maintainable, and resilient. Here are some key microservices design patterns and principles:

Microservices Design Patterns

Decomposition Patterns:

Domain-Driven Design (DDD): Break down the system into bounded contexts, each representing a specific business domain. Each bounded context corresponds to a microservice.

Domain means The core business logic and rules. The domain layer is where the business objects and their interactions reside. Domain-Driven Design is used for designing and building software systems that is particularly well-suited for microservices architecture.

Key Concepts of DDD

a) Domain: The core business logic and rules. The domain layer is where the business objects and their interactions reside.

b) Entity: An object that has a distinct identity that runs through time and different states.

c) Value Object: An immutable object that is defined only by its attributes.

d) Aggregate: A cluster of related entities and value objects that are treated as a single unit.

e) Repository: A mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects.

f) Service: Contains the business logic that doesn't naturally fit within an entity or value object.

g) Bounded Context: Defines the boundary within which a particular model is defined and applicable.

Subdomain Decomposition: Identify and decompose subdomains within the main domain, creating microservices for each subdomain.

Integration Patterns:

API Gateway: A single entry point for all client requests, which then routes the requests to the appropriate microservices.

Aggregator: A microservice that invokes multiple microservices to achieve the desired functionality and returns the aggregated result to the client.

Proxy: Similar to an API Gateway, but also handles tasks such as load balancing, caching, and security.

Communication Patterns:

Synchronous Communication: Uses HTTP/REST or gRPC for direct service-to-service communication.

Asynchronous Communication: Uses messaging systems like RabbitMQ, Kafka, or AWS SQS for decoupled and resilient communication.

Database Patterns:

Database per Service: Each microservice manages its own database to ensure loose coupling and autonomy.

Shared Database: Multiple microservices share a single database, suitable for scenarios where strong consistency is required.

CQRS (Command Query Responsibility Segregation): Separate the read and write operations into different models to optimize performance and scalability.

Data Management Patterns:

Event Sourcing: Persist the state of a service as a sequence of events rather than storing the current state directly.

Saga Pattern: Manage distributed transactions by coordinating multiple microservices using either choreography or orchestration.

Resilience Patterns:

Circuit Breaker: Prevent cascading failures by stopping calls to a service that is failing or not responding.

Retry: Automatically retry failed operations to handle transient faults.

Bulkhead: Isolate critical resources to prevent failure in one part of the system from affecting others.

Observability Patterns:

Log Aggregation: Collect and aggregate logs from all services for centralized analysis and monitoring.

Distributed Tracing: Trace requests as they travel through different services to diagnose latency issues and track failures.

Metrics: Collect and monitor metrics like response times, error rates, and throughput to ensure system health.

Design Principles

Single Responsibility Principle (SRP): Each microservice should have a single responsibility, focusing on a specific business capability.

Interface Segregation Principle (ISP): Design services with specific, fine-grained interfaces rather than a single general-purpose interface.

Loose Coupling: Ensure that microservices are loosely coupled and communicate via well-defined interfaces, reducing dependencies and improving resilience.

High Cohesion: Group related functionality within a single microservice to enhance maintainability and reduce the complexity of interactions.

Design for Failure: Assume that failures will happen and design microservices to handle failures gracefully. Implement patterns like circuit breakers, retries, and timeouts.

Decentralized Data Management: Avoid centralized databases to prevent bottlenecks and enable microservices to manage their own data independently.

Automation: Automate the deployment, scaling, and monitoring of microservices to ensure consistent and reliable operations.

Security: Implement security best practices such as authentication, authorization, and encryption to protect microservices and data.

==========

======Spring Cloud Gateway======

==========

Spring Cloud Gateway is API gateway which is implemented by Spring Cloud team.

It provides the simple and effective way of routing the API request to downstream microservices based on the different parameters and configuration.

API Gateway is one of the design pattern in microservices architecture.

Important parts of Gateway

1. **Route :-** Route is the block which contains the URL to which incoming request to be forwarded.

2. **Predicate :-** Predicate is the block which contains the condition criteria which should match to forward the incoming request to route URL. There are different types of Predicates.

a) Path Route	e) Bewtween Route	j) Query Route
b) Header Route	g) Cookie Route	k) RemoteAddr Route
c) After Route	h) Host Route	l) Weight Route
d) Between Route	i) Method Route	

3. **Filter :-** Filter the block where we can modify the incoming request before sending to route URL.

==========

======Resilience4j is a fault tolerance mechanism======

==========

In a microservice architecture, it’s common for a service to call another service. And there is always the possibility that the other service being called is unavailable or unable to respond, In this scenario Resilience4j is used to protect the service resources by throwing an exception depending on the fault tolerance pattern.

Resilience4j is use to fault tolerance library designed for Java applications. It provides various features like Circuit Breaker, Retry, Rate Limiting which help to build resilient and fault-tolerant systems.

Resilience4j have different modules

1. Circuit Breaker :-- It acts like a switch that opens when a any microservice unavailable or unable to respond and when a failure threshold is met.

A Circuit Breaker has three primary states: **(Closed , Open Half-Open)**

Closed is the initial state of circuit breaker. When microservices run and interact smoothly, circuit breaker is Closed.

Example :-- Suppose one Microservice is trying to consume the another microsevice but if that microsevice is unavailable or unable to respond, so these requests are failing, in this case , we can define the threshold to identify how many requests are failing , suppose , if you define the sliding window of 10 requests and y**ou defined the threshold as 50 % , so if 50 or more than 50% requests are failing in that case state will change from close to open.**

And when circuit is open state , then no request will be allowed, here you can define the wait duration for your circuit breaker to be in the open state.

Example :-- so if you define the 5 second wait duration , so for this 5 second this circuit breaker will be in open state. and after that it will change to Half-open , Half-Open is nothing but half of the **circuit is open and half of the circuit is closed.** so few of the request will be pass to this circuit breaker , just to check that the service which are down is up or not.

so once it comes in to Half-Open state , again you can define threshold, How many request should be allowed within the Half-open state.

Example :-- suppose you define that 3 request are allowed in Half-Open state and your thresold is 50 % again if 50 or more than 50% requests are failing
then again it will go back to in open state. and if it found there is no failing , then it will change to close state.

2. Retry :-- Retry is simple thing , where you define the how many retries you should do , when there is a failure in your service.

Example :-- suppose , if you define 3 retries when there is failure, as well as time duration.
so it will try 3 times and after 3 retries still request is failing, then it will call callback method and will return response.

3. Rate Limiter:-- Rate Limiter is nothing but to identify or check your system, that how many request are allowed within time duration.

Example :-- suppose , your application can handle only thousand request per second, then you can define the rate limiter for this particular URL
which will only hit thousand request per second.

4. Time Limiter :-- Time limiter ensures that request should be completed within decided time frame, if Request is not completed decided time frame, then it will call callback method and will return response..

Example :-- suppose, there is lot of delay while getting response from given request , in that case Time limiter will call callback method and will return the default response.

5. BulkHead :-- The name Bulkhead comes from the technique building the boats , where the boat is divided into multiple sections, such that if the single
section is flooded, then it does not flood the entire boat.

Note :- In Bulk Head pattern, The application isolated into pools , so if one service fails so the other service will still continue to function.

BulkHead pattern is used to limiting number of concurrent execution.

There are two types in Bulk Head pattern

1. **semaphore :-** in this approach, we limit the concurrent requests to the service,It will reject the incoming requests once the limit is hit.

   for this, we defines the (maxWaitDuration, maxConcurrentCalls) attributes in application.properties.

2. **Fixed Thread Pool BulkHead :--** In thin approach, We isolate a set of thread pool from system resources, using only that thead pool for the service. if pool are full then request will get rejected with BulkHeadFull Exception.

   for this, we defines the (maxThreadPoolSize, coreThreadPoolSize and QueueCapacity) attributes in application.properties.

6. Cache :- Cache module provides a mechanism for caching the results of method calls to improve performance and reduce the load on services. By caching results, you can avoid repeated calls to slow or expensive operations, improving the overall efficiency and responsiveness of your application.

==========

Microservices , Agile & Jira created by mayuresh ratnaparkhi

======AGILE and JIRA SHORT INFORMATION======

==========

How Agile Works.

DSU :-- what we did yesterday and what is my today's plan

Agile sprint Planning :-- We take decision, in which we discuss, which ticket / story has to be completed in this sprint and also assign story points/ estimations for the decided ticket. as well as we decides the priority of the tasks.

Refirment Call :-- This calls schedule before 2,3 days the sprint ends , In this we decide what will be the tickets for the future sprint.

Demo call :-- We gives the demo of completed tasks.

Retrospective Call :-- we analys the overall sprint and rate ourself depending on the task , which we have done in the current sprint , and also
we analys what went wrong in this sprint and how it can be improve in the future sprint.

What is story point :-- Story points are units of measure for expressing an estimate of the overall effort required to fully implement a product backlog item

sprint Cycle :-- 3 weeks

what are the primary factors which you have to consider being a developer

1. Requirement Phase is most important, where we need to understand the requirement.
2. base on the Requirement understanding, then We do the documentation and verifies this from client.
3. also we clears all doubt and finally we gives the estimates.
4. after that we decides , which design pattern can be use for coding and coding standards.
5. We create tdd (techinical document design) and Application Version documents.

How do you estimate the story point?

Answer :- Every user story have units and that units called story points. story point are unit, not days or hours and base on below factors we decides the story point

1. Amount of Work
2. Complexity
3. Risk/Dependencies

Story points creates base on fibonacci Series
Example 0,1,1,2,3,5,8,13,21,34

What is your team velocity and capacity planning?

velocity :-- In every sprint , how many story points are completed that is called velocity.
we basically calculate the average of previous sprint and that average called velocity.

How do you track your team work/progress ?
Answer :-- using Jira

JIRA :-- in backlogs , we create the stories, and we add these stories in sprint.

PROCESS :-- We will create the **EPIC**, then will create the **backlogs** and **in backlogs, we will create stories.**
then we will select and add those stories in current sprint. after that we maintain progress of task with below status

1. todo
2. inprogresss
3. completed/done
4. testing

==========