1. **What is the need for microservices architecture.**

Ans:

**Advantages of microservices:**

* **Accelerate scalability**: Due to independent operation of each service within the microservice architecture. They can choose best language, technology for each of the services without compatibility issue.
* **Types of Scalabilities:**
* **Horizontal Scaling: -** Adding more instances of a service or server to distribute the load. If one instance can handle 100 requests per second, adding another instance would increase the capacity to 200 requests per second. A load balancer is often used to distribute traffic among these instances.

* **Vertical Scaling: -** This is called resource level Scaling which Increasing the resources (CPU, RAM, Storage) of an existing instance to handle more load. If a single instance could handle 100 requests per second, after increasing its CPU and RAM, it might handle 200 requests per second, but it is still a single instance.

Load Balancer is not in picture in this case.

* **Improve fault isolation (Fault Tolerance):** If any of the server is encounter fault or failure. other services will not get affected.

* **Enhance team productivity:** Team can focus on particular service development, deployment, and maintenance without being burdened by the complexities of the entire system.
* **Quicker deployment time:** it gives faster release because service is independent and no risk (no impact on other service).

* **Increase Cost-efficiency:** Team is dedicated on small, well-defined service. Teams focus on specific functionality, ensuring resources are used efficiently without redundancy or excess capacity.

**Disadvantages of microservices:**

* **Increased Complexity:** managing service communication can be challenging in distributed system. Developer may have to write extra code to smooth communication between modules.

* **Deployment and versioning challenges:** Coordinating deployments and managing version control across multiple services can be complex

* **Testing complexity:** conducting integration testing across various services can increase complexity.

* **Debugging difficulties:** A single business process can run across multiple machines simultaneously. Each has their own set of logs.

* **Data management challenges**: Difficulty in Data consistency and transaction in distributed architecture.

2. **Monolithic Architecture**

Ans:

**Advantages of using a Monolithic Architecture: -**

* **Simplicity:** Easy to understand since all the code of application at one place.
* **Deployment Speed:** Deployment speed is fast because all the modules are tightly connected to each other and there is single jar or war must be deployed.
* **Deployment:** Reduce the risk of deployment error and easy to rollback since it has only one Artifact.
* **Debugging:** Debugging is easy because code is in one place
* **Latency:** It has low Latency because of less Network call.

**Disadvantages of using a Monolithic Architecture: -**

* **Complexity:** Application grows and will harder to manage in long lime.
* **Scalability:** Monolithic Application can be challenging to scale. especially when certain component must be scale but due to monolithic architecture scale must be done for entire application.
* **Technology Stack:** entire application is in same tech stack.
* **Deployment:** Deployment is time consuming. Any small changes will force to deploy entire application.
* **Fault Tolerance:** Application with Monolithic architecture is not fault Tolerant. if single component fails it can bring down the entire system.

3. **When Microservices Should Not Be Implemented**

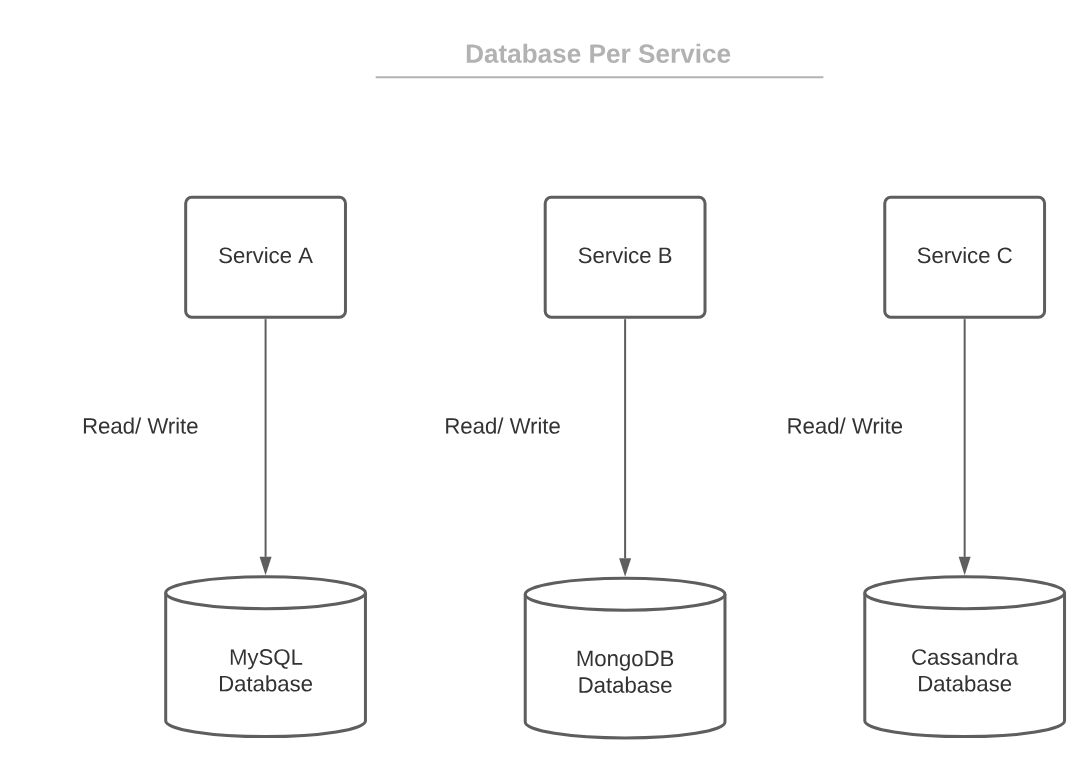
Ans:

* Small or Simple Applications
* Low Traffic or Load
* Lack of Expertise and Resources
* Tight Coupling and Strong Interdependencies

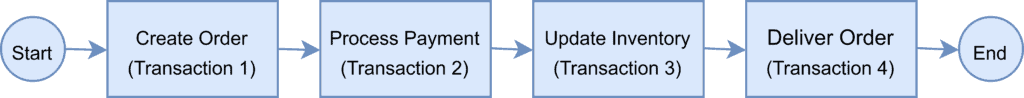
4. **Microservice Design Patterns (Deep understanding, Use case, Implementation by keeping distributed system in knowledge| Multiple instances | Load Balancing | Request Flow with distributed system ):**

Ans:

* **Database per service pattern:** The data owned by one service may be largely relational, while a second service might be better served by a NoSQL solution and a third service may require a vector database. In this scenario, using dedicated services for each database could help you manage them more easily.

****

* **Saga design pattern:** The saga design pattern is a style of software architecture that's intended to manage distributed transactions. It helps maintain data consistency and integrity across multiple services without using distributed transactions, which are complex and hard to manage. The Saga pattern breaks a transaction into a series of smaller, isolated steps that are executed one by one. If one step fails, the pattern triggers compensating transactions to undo the work done by the preceding steps.



This is an example of a distributed transaction as the transaction boundary crosses multiple services and databases.

To ensure a successful order processing service, all four microservices must complete the individual local transactions. If any of the microservices fail to complete its local transaction, all the completed preceding transactions should roll back to ensure data integrity.

**When to use a saga design pattern:** The saga pattern is well-suited for business workflows that must reset in their entirety if there is a failure in any one of the workflow's steps. Order processing workflows and supply chain management systems are typical use cases.

There are two approaches to implement a saga: orchestration and choreography.

**orchestration**: In an orchestrated saga, a central controller manages the steps and compensations within a sequence.

**choreography:** In the choreographed approach to a saga design pattern, each step acts independently with no central controller. One step knows its subsequent step in the process, so when it completes it calls the next step. If a call fails, the called step executes its compensation and then throws an error.

**Use Case:** Order Fulfilment Process

* **Circuit breaker design pattern:** Circuit Breaker is a design pattern which help to cascade failure in case of other microservice is failed to send response.

- Circuit Breakers in Microservices serve as a protective shield, ensuring that the system remains operational, even when some parts face issues.

- This pattern is usually applied between services that are communicating synchronously.

- A developer might decide to utilize the circuit breaker when a service is exhibiting high latency or is completely unresponsive.

- Therefore, calls won't be piling up and using the system resources, which could cause significant delays within the app or even a string of service failures.

- Implementing this pattern as a function in a circuit breaker design requires an object to be called to monitor failure conditions. When a failure condition is detected, the circuit breaker will trip. Once this has been tripped, all calls to the circuit breaker will result in an error and be directed to a different service. Alternatively, calls can result in a default error message being retrieved.

**There are three states of the circuit breaker pattern: -**

**Open:** A circuit breaker pattern is open when the number of failures has exceeded the threshold. When in this state, the microservice gives errors for the calls without executing the desired function.

**Closed:** When a circuit breaker is closed, it's in the default state and all calls are responded to normally. This is the ideal state developers want a circuit breaker microservice to remain in — in a perfect world, of course.

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

* **API Gateway Pattern:** An API gateway is a type of middleware that acts as a central entry point for requests between clients and microservices in an application. The API Gateway is responsible for request routing, composition, protocol translation, and more. It acts as a reverse proxy, handling all the client requests and forwarding them to the appropriate microservice. It can help address challenges that arise from microservices architecture, such as direct client-to-microservice communication and coupling.

**Benefits of API Gateway Pattern: -**

**Centralized Routing:** Provides a single endpoint for clients, reducing the complexity of managing multiple microservice endpoints.

**Security:** The API gateway hides the services and the underlying infrastructure from being exposed publicly, which makes it difficult for attackers to bring down the application. It can also apply Centralizes authentication, authorization, rate limiting, and other security concerns against attacks, and other security features such as request validation, auth, circuit breaking, and policy enforcement.

**Routing:** The API gateway handles client requests and routes to the Eureka server for actual microservices where request will come back with microservices details and hit appropriate microservices.

**Implementation Flow**: Set Up Project using Spring Cloud Gateway --> Creating the API Gateway Application with @EnableEurekaClient --> application properties changes for add other microservices to it --> Create other 2 services --> Order services, customer services --> Create Eureka --> Register all services with Eureka server --> while calling use API Gateway URL.

Ex: <http://localhost:8080/orders/123>

<http://localhost:8080/customers/456>

* **Service Discovery Pattern**: Service Discovery is a design pattern used to dynamically find the network locations of services in a microservices architecture. The primary goal of service discovery is to decouple service consumers from service providers

**Service Discovery Mechanism**: The mechanism by which the service consumers discover available instances of a service. There are two types of service discovery mechanisms:

**Client-Side Discovery**: The service consumer is responsible for querying the Service Registry to obtain the network location of a service instance and then makes a direct request to the service. Examples: Ribbon with Eureka, Netflix Feign.

**Server-Side Discovery**: The service consumer makes a request to a router (e.g., API Gateway), which is responsible for querying the Service Registry and routing the request to an available service instance. Examples: Zuul, Spring Cloud Gateway with Eureka.

* Load Balancing (Working of load balancing in service discovery):
* Event Sourcing:
* CQRS (Command Query Responsibility Segregation):
* Sidecar Pattern:

4. **Advantages in Spring Boot Over Spring**

Ans: 1. You can use it to create standalone applications. we are using main method to start the server.

2. No need to deploy WAR file into the tomcat location.

3. It Does not require XML configuration (Spring Boot is fully Annotation Based)

4. Offer production ready feature (Spring Boot Actuator). - take a view of UI

5. Inbuild Serves Support

5.1. Tomcat (Default) 1st priority

5.2. Undertow 2nd priority

5.3. Jetty 3rd priority

Note: Spring boot provides 3 built-in servers in it.

1. If required to exclude any server (Tomcat, Undertow, Jetty). That needs to be excluded in pom.xml from spring-boot-starter-web.

Ex:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

2. if required to include any third-party server (jboss) then exclude all 3 server (Tomcat, Undertow, Jetty) and add dependency of jboss.

Ex:

<dependency>

<groupId>org.wildfly.bom</groupId>

<artifactId>wildfly-javaee8</artifactId>

<version>23.0.2.Final</version>

<type>pom</type>

<scope>provided</scope>

</dependency>

6. Automate Version management (not required to provide the version in pom.xml it will auto fetch according to the parent version)

7. Inbuilt Database support (h2 Database)

8. Auto Configuration (It will scan the classpath, so whatever beans present in the form of jar it will create the object for those beans and make ready inside the container)

5. **How to restrict hibernate in Springboot applications**

Ans:

1. Exclude Hibernate Dependency by Using Pom.xml

Ex:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

<exclusions>

<exclusion>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

</exclusion>

</exclusions>

</dependency>

2. Exclude Auto-Configuration from main method by using @SpringBootApplication.

Ex:

@SpringBootApplication(exclude = HibernateJpaAutoConfiguration.class)

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

3. Remove Hibernate-Specific Properties by Using property file

6. **What is the difference between Hibernet and JPA Data.**

Ans: 1. JPA is a specification and does not provide actual implementations.

2. Hibernate is an implementation of JPA and provides concrete functionality.

4. What is API Gateway in microservices

Ans: An API gateway is a type of middleware that acts as a central entry point for requests between clients and microservices in an application.

It can help address challenges that arise from microservices architecture, such as direct client-to-microservice communication and coupling.

Security:

The API gateway hides the services and the underlying infrastructure from being exposed publicly, which makes it difficult for attackers to bring down the application.

It can also apply rate limiting against attacks, and other security features such as request validation, auth, circuit breaking, and policy enforcement.

Routing:

The API gateway handles client requests and routes to the Eureka server for actual microservices where request will come back with microservices details and hit appropriate microservices.

5. **What is Eureka Server in microservices.**

Ans:

4.1. Eureka Server is a service registry for microservices that helps manage service registration and discovery.

4.2. It was originally developed by Netflix and is now open source and part of the Spring Cloud umbrella.

4.3. Eureka Server's main purpose is to enable efficient service discovery by maintaining a registry of all active service instances, providing a real-time directory of microservices within a system.

8**. What is Docker and how does it work in Microservices application.**

Ans: Docker is perfect for deploying microservices architecture, which builds a single application by breaking it into a collection of independent, loosely coupled services.

9. **Blue Green Deployment**

Ans: Blue-green deployment is a release management strategy for software deployment that aims to reduce downtime and risk by running two identical production environments.

10. Configurations Management in spring boot application

Ans:

1. Configuring Spring Boot with properties and profiles. (4)

we can create other profiles by creating properties like below

application\_prod.properties -----------> For Prod profile

application\_qa.properties -----------> For QA profile

1. Applying external Configuration with spring Boot.

Using Command line - highest precedence (1)

Limitation – it is complex and time taking deployment processand due to this complexity can make error and if you have multiple instances for one micro service and if you have n number of microservices

Using JVM Properties - (2)

Environment variable - (3)

Limitation - Admin can see the environment variables . so it is not safe

1. Implementing Configuration server with Spring boot cloud config server.

Udemy Questions:

1. What is microservices architecture and how it is different from monolithic and SOA architectures

2. How to build production ready microservices using Java, Spring, SpringBoot and Spring Cloud

3. How to document microservices using Open API Specification (API spec in yml file ) and Swagger(API Documentation tool)

4. How to right size microservices and identify service boundaries

5. Role of Docker in microservices and how to build docker images, containers

6. Role of Docker compose and how to use it to run all the microservices inside a application

7. What are cloud native apps & 15 factor methodology behind them

8. Configuration management in microservices using Spring Cloud Config Server

9. Service Discovery and Registration pattern inside microservices and how to implement using Spring Eureka server

10. Handling Cross cutting concerns and routing inside microservices using Spring Cloud Gateway

11. Building resilient microservices using RESILIENCE4J framework

12. Implementing observability and monitoring using Prometheus, Loki, Promtail, Tempo and Grafana

13. Securing microservices using OAuth2, OpenID connect and Spring Security

14. How to build event driven microservices using RabbitMQ, Kafka, Spring Cloud Functions and Spring Cloud Stream

15. Role of Kubernetes in microservices as a container orchestration framework.

16. How to setup a Kubernetes cluster inside GCP using Google Kubernetes Engine and deploy microservices inside it

17. What is Helm & it's role in microservices world

18. Most commonly used Docker, Kubernetes and Helm commands