MODULE 10

Java – Micro services with Spring Boot, Spring Cloud

1. Microservices with Spring Boot and Spring Cloud

- ### What are Microservices?

Definition and characteristics of Microservices architecture.

Microservices Architecture is an architectural style that structures an application as a collection of loosely coupled, independently deployable services. Each service corresponds to a specific business function and can operate autonomously, allowing them to be developed, deployed, and maintained independently.

In a microservices architecture, each service is small, self-contained, and performs a specific function, and they typically communicate over lightweight protocols like HTTP or messaging queues.

Characteristics:

- Decoupled Services: Independent services that can be developed, deployed, and scaled separately.
- Scalability: Services can be scaled independently based on demand.
- Independent Deployment: Each service can be deployed without affecting others.

Key principles: Decoupled services, scalability, independent deployment.

1. Decoupling: Services are independent and do not depend on each other's internal implementation.
2. Scalability: Services can scale independently, optimizing resource use.
3. Independent Deployment: Each service can be updated or deployed without impacting the entire system.

- ### Advantages of Microservices Over Monolithic Architecture:

- **Scalability**:
    - Microservices allow independent scaling of services based on demand. In contrast, monolithic systems require scaling the entire application, which can be inefficient.

- **Fault Isolation**:
    - In microservices, failures are isolated to specific services, ensuring that issues in one service don't affect the entire system. Monolithic systems have higher risk of cascading failures across the entire application.
- **Flexibility**:
    - Microservices allow using different technologies, frameworks, and databases for different services, offering greater flexibility. Monolithic architectures typically rely on a single technology stack.
- **Faster Deployment**:
    - With microservices, each service can be updated and deployed independently, leading to faster release cycles. Monolithic systems often require redeploying the entire application for any change, slowing down deployment.

- ## Components of Microservices Architecture:

1. **API Gateway**:

- The API Gateway acts as the entry point for client requests. It routes and load-balances incoming requests to the appropriate microservices, handling cross-cutting concerns like authentication, rate-limiting, and logging.

2. **Service Registry (e.g., Eureka)**:

- A Service Registry keeps track of all the available microservices and their locations (IP address, port, etc.). It allows services to register themselves and discover other services dynamically.

3. **Circuit Breaker**:

- The Circuit Breaker pattern helps manage failures by detecting service faults and preventing further requests to a failing service. It allows the system to fail gracefully and prevents cascading failures.

4. **Load Balancer**:

- A Load Balancer distributes incoming requests across multiple instances of a service to ensure efficient resource usage and prevent any single service instance from becoming overwhelmed.

2. Introduction to Microservice Architecture

- **Microservice vs. Monolithic Architecture:**

  Monolithic Architecture: A single, large application where all functionalities are interconnected and run as one unit.

Microservices: An application is divided into smaller, self-contained services that are independently developed, deployed, and scaled.

- **Key Characteristics:**
  Decentralization: Each microservice has its own database.
  Inter-Service Communication: Services communicate using lightweight protocols like HTTP or messaging systems like RabbitMQ

3.Developing and Deploying a Microservice Application Locally

Theory:

Building a microservice-based application involves creating multiple, independent services that can function autonomously. Below are the steps involved in developing and deploying a microservice application locally.

Steps to Build a Microservice:

1. **Develop Each Service Independently**:
   - Break the application into smaller, independent services, each with its own functionality.
2. **Use Spring Boot for Development**:
   - Set up Spring Boot projects for each service.
   - Develop RESTful APIs and business logic.
3. **Package and Deploy**:
   - **Docker**: Create a Dockerfile for each service, build Docker images, and run containers.
   - **Localhost**: Run services directly on different local ports (e.g., localhost:8081, localhost:8082).
4. **Service Communication**:
   - Use REST APIs or message brokers (e.g., Kafka) for communication between services.
5. **Testing Locally**:
   - Test each service and its interactions using tools like Postman or cURL.

4. Introduction to Service Discovery: Eureka Server

- **Service Discovery**

Service Discovery: In a microservices architecture, the individual services can start, stop, or scale dynamically. As a result, each service might not always have a fixed address or endpoint. To solve this problem, **Service Discovery** becomes essential. It allows services

to register themselves and discover other services even if their locations (IP addresses or ports) change dynamically. A **Service Registry** is a centralized directory where these services are registered and where other services can query to discover their locations.

- **What is Eureka?**

  Eureka is a Service Registry tool created by **Netflix** for managing and facilitating service discovery in microservices architectures. Eureka allows microservices to register themselves with the Eureka Server and discover other services easily. It acts as a registry where services can communicate with each other by querying the registry, even if the services' instances are dynamically changing.Eureka Server and Eureka Client:

Eureka Server and Eureka Client:

1. **Eureka Server:**

   The Eureka Server acts as a **Service Registry** where all registered services are stored.

   The server maintains a list of all available service instances, including their metadata (such as IP address and port).

   The Eureka Server is responsible for handling requests from client services who want to register or discover other services.

2. **Eureka Client:**

   The Eureka Client is a microservice that registers itself with the Eureka Server upon startup.

   The client also queries the Eureka Server to discover other available services that it might need to communicate with.

   The Eureka Client is typically configured with the Eureka Server's URL so it knows where to send requests for registration or discovery.

5. Client-Side and Server-Side Discovery Patterns

- **Client-Side Discovery:**

  Definition: The client is responsible for discovering services by interacting with the **Eureka Server** and selecting an instance.
  How it Works:
  Client queries Eureka for available services.
  Selects a service instance and sends the request.

Advantages:
No single point of failure.
More control for clients (e.g., load balancing).
Disadvantages:
Increased complexity for clients (service discovery, load balancing).
Tight coupling with service registry.

- **Server-Side Discovery:**

  Definition: The client makes a request to an **API Gateway** or **Load Balancer**, which handles service discovery and forwards the request to an appropriate service.
  How it Works:
  Client sends requests to API Gateway/Load Balancer.
  Server handles discovery, selects a service, and forwards the request.
  Advantages:
  Simplifies client logic.
  Centralized routing and load balancing.
  Disadvantages:
  Single point of failure (API Gateway or Load Balancer).
  Potential bottleneck.

6. Load Balancing Configuration

- **What is Load Balancing?**
  Load balancing is the process of distributing incoming network or application traffic across multiple instances of a service. The goal is to ensure better performance, availability, and fault tolerance by preventing any single service instance from becoming overwhelmed with too many requests.

  Types of Load Balancers:

- **Client-Side Load Balancer:**
  Managed at the client-side (e.g., Ribbon).
  The client is responsible for deciding which service instance to send the request to.
  The client queries the service registry (like Eureka) for available service instances and uses a load balancing algorithm (e.g., round-robin) to choose an instance.
  Advantages:
  No dependency on a centralized load balancer.
  Clients have full control over load balancing decisions.
  Disadvantages:
  Increases complexity on the client side.

- **Server-Side Load Balancer:**
  Managed centrally (e.g., API Gateway, Nginx).

The client sends a request to a load balancer (like API Gateway or Nginx), which then distributes the requests to the available service instances.
Advantages:
Simplifies the client-side logic.
Centralized management of load balancing and routing.
Disadvantages:
Creates a single point of failure (the load balancer).