Microservices with Spring Boot

1. Introduction to Microservices

- **Microservices Architecture** is an approach to building enterprise applications as a **collection of small, independent services** that communicate with each other using lightweight protocols (mostly REST or messaging).

- Each microservice is:

    - **Loosely coupled** → Can be developed, deployed, and scaled independently.

    - **Focused on a single business capability** (e.g., Payment Service, Order Service).

    - **Technology agnostic** → Different services may use Java, Python, Node.js, etc.

- **Spring Boot + Spring Cloud** is the most popular combination for building production-ready microservices because it simplifies configuration, service discovery, communication, and resilience.

---

2. Key Characteristics of Microservices

1. **Independence**: Services can be built and deployed without affecting others.

2. **Scalability**: Each service can scale based on demand (e.g., scaling only the payment service on Black Friday).

3. **Resilience**: Failure in one service should not crash the entire application.

4. **Decentralized Governance**: Each team manages its own service lifecycle.

5. **Polyglot Development**: Use of multiple technologies per service if required.

---

3. Implementing Microservices with Spring Boot and Spring Cloud

3.1 Service Discovery with Eureka

- In a distributed system, microservices run on different servers/ports, making it hard to manage them.
- **Eureka Server (Service Registry)** acts as a directory where services register themselves and discover others.

Steps:

1. Create **Eureka Server**:

```
@EnableEurekaServer
@SpringBootApplication
public class DiscoveryServerApp {
    public static void main(String[] args) {
        SpringApplication.run(DiscoveryServerApp.class, args);
    }
}
```

2. Register a microservice with Eureka:

```
@EnableEurekaClient
@SpringBootApplication
public class ProductServiceApp {
    public static void main(String[] args) {
        SpringApplication.run(ProductServiceApp.class, args);
    }
}
```

- Now, **Order Service** can discover and call **Product Service** using the Eureka registry.

3.2 API Gateway with Spring Cloud Gateway

- API Gateway is a **single entry point** for all client requests.
- Responsibilities:
  - Routing requests to correct microservice.
  - Load balancing.
  - Security (authentication/authorization).
  - Rate limiting and monitoring.

Example Configuration (application.yml):

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: product-service
          uri: lb://PRODUCT-SERVICE
          predicates:
            - Path=/products/**
        - id: order-service
          uri: lb://ORDER-SERVICE
          predicates:
            - Path=/orders/**
```

- Requests to `/products/**` are routed to **Product Service**.
- Requests to `/orders/**` go to **Order Service**.

3.3 Inter-Service Communication

- Microservices often need to talk to each other. Two approaches:

(a) Feign Client (Declarative REST Client)

- Simplifies communication by defining Java interfaces with annotations.

```
@FeignClient(name = "PRODUCT-SERVICE")
public interface ProductClient {
    @GetMapping("/products/{id}")
    Product getProduct(@PathVariable("id") Long id);
}
```

(b) RestTemplate / WebClient (Imperative/Reactive)

```
@Autowired
private RestTemplate restTemplate;

Product product =
restTemplate.getForObject("http://PRODUCT-SERVICE/products/1",
Product.class);
```

3.4 Resilience with Circuit Breakers (Hystrix/Resilience4j)

- Microservices may fail due to downtime or network issues.
- Circuit breakers prevent cascading failures by:
  - Monitoring calls.
  - Providing fallbacks when a service is down.

Example (Resilience4j with Feign):

```
@FeignClient(name = "PRODUCT-SERVICE", fallback =
ProductFallback.class)
public interface ProductClient {
    @GetMapping("/products/{id}")
    Product getProduct(@PathVariable("id") Long id);
}
```

```java
@Component
public class ProductFallback implements ProductClient {
    @Override
    public Product getProduct(Long id) {
        return new Product(id, "Default Product", 0.0);
    }
}
```

3.5 Centralized Configuration with Spring Cloud Config

- Instead of hardcoding configs in every service, we store them in a **central Git repo**.
- **Config Server** serves these configs to microservices at runtime.

Example:

```yaml
spring:
  application:
    name: product-service
  cloud:
    config:
      uri: http://localhost:8888
```

3.6 Deployment Patterns for Microservices

1. **Single Service per Container (Docker/Kubernetes).**
   - Each microservice runs in a container.
   - Kubernetes handles orchestration, scaling, and auto-healing.
2. **CI/CD Pipelines:**
   - Automated testing, build, and deployment.
   - Jenkins, GitHub Actions, or GitLab CI commonly used.

3. **Service Mesh (Istio/Linkerd):**
    - Advanced traffic routing, security, and monitoring for microservices.

---

4. Real-Time Example (E-commerce Application)

- **Product Service** → Manages product catalog.
- **Order Service** → Manages customer orders.
- **Payment Service** → Handles transactions.
- **Eureka** → Service discovery between all services.
- **Spring Cloud Gateway** → Single entry for customers.
- **Resilience4j** → Fallback for payment failures.
- **Config Server** → Central configuration for all services.
- **Docker/Kubernetes** → Deployment and scaling.

---

5. Advantages of Microservices with Spring Boot

- Independent deployment and scaling.
- Faster development with smaller, focused teams.
- Resilient and fault-tolerant architecture.
- Technology flexibility (polyglot systems).
- Easy integration with cloud-native environments.

---

6. Challenges of Microservices

- Increased complexity in service communication.
- Requires monitoring, logging, and distributed tracing (ELK Stack, Zipkin).
- More DevOps effort (CI/CD, container orchestration).
- Data consistency across services (eventual consistency vs transactions).

---

7. Conclusion

Microservices architecture, powered by **Spring Boot** and **Spring Cloud**, is the foundation for **scalable, resilient, and cloud-ready applications**. With **Eureka** for service discovery, **Spring Cloud Gateway** as API gateway, **Feign clients** for communication, **Resilience4j** for fault tolerance, and **centralized configuration**, Spring simplifies the otherwise complex microservice ecosystem.