7. Spring Boot Microservices Architecture 🚀

In this section, you will learn:
✅ **What are Microservices?**
✅ **Why Use Microservices?**
✅ **Building Microservices in Spring Boot**
✅ **Service Discovery with Eureka**
✅ **API Gateway with Spring Cloud Gateway**
✅ **Inter-Service Communication (Feign & RestTemplate)**
✅ **Centralized Configuration with Spring Cloud Config**
✅ **Resilience & Fault Tolerance with Resilience4j**

7.1 What are Microservices? 🤔

Microservices is an architectural style where applications are **broken down into smaller, independent services** that:
✅ Are loosely coupled
✅ Can be developed & deployed independently
✅ Communicate via APIs (HTTP, gRPC, Messaging, etc.)

🔹 **Example of a Microservices-based E-commerce System:**

- 🛒 **Order Service** (Handles orders)
- 📦 **Inventory Service** (Manages stock)
- 💳 **Payment Service** (Processes payments)
- 👤 **User Service** (Manages user authentication)

Each service **runs independently** but communicates using APIs.

7.2 Why Use Microservices? 🤔

🔹 **Advantages:**
✅ **Scalability** – Scale individual services as needed.
✅ **Flexibility** – Each service can be written in different technologies.
✅ **Faster Deployment** – Services can be deployed independently.
✅ **Fault Isolation** – If one service fails, others continue running.

🔹 **Challenges:**
❌ **Complexity** – Requires good management.
❌ **Inter-Service Communication** – Services must interact efficiently.
❌ **Data Consistency** – Handling transactions across services is challenging.

7.3 Creating Microservices in Spring Boot

We will create two microservices:
1️⃣ **Order Service** (Handles orders)
2️⃣ **Inventory Service** (Checks stock)

7.3.1 Order Service (`order-service`)

Step 1: Add Dependencies (`pom.xml`)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

✅ **Spring Web**, **JPA**, and **OpenFeign** (for API calls).

Step 2: Create Order Entity (`Order.java`)

```
import jakarta.persistence.*;
import lombok.*;

@Entity
@Table(name = "orders")
@Getter @Setter
@AllArgsConstructor @NoArgsConstructor
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String productCode;
    private int quantity;
}
```

Step 3: Create Order Repository (`OrderRepository.java`)

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface OrderRepository extends JpaRepository<Order, Long> {
}
```

Step 4: Create Inventory Client (Feign) to Call Inventory Service

```
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@FeignClient(name = "inventory-service")
public interface InventoryClient {
    @GetMapping("/inventory/check")
    boolean checkStock(@RequestParam String productCode, @RequestParam int
quantity);
}
```

✅ Calls **Inventory Service** to check stock availability.

Step 5: Create Order Service (`OrderService.java`)

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class OrderService {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private InventoryClient inventoryClient;

    public String placeOrder(String productCode, int quantity) {
        boolean isAvailable = inventoryClient.checkStock(productCode,
quantity);
        if (isAvailable) {
            Order order = new Order();
            order.setProductCode(productCode);
            order.setQuantity(quantity);
            orderRepository.save(order);
            return "Order placed successfully!";
        } else {
            return "Out of stock!";
        }
```

```
    }
}
```

✅ **Checks inventory before placing an order.**

Step 6: Create Order Controller (`OrderController.java`)

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/orders")
public class OrderController {

    @Autowired
    private OrderService orderService;

    @PostMapping("/place")
    public String placeOrder(@RequestParam String productCode,
@RequestParam int quantity) {
        return orderService.placeOrder(productCode, quantity);
    }
}
```

✅ **API to place an order.**

7.4 Creating Inventory Service (`inventory-service`)

Step 1: Create Inventory Controller (`InventoryController.java`)

```
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/inventory")
public class InventoryController {

    @GetMapping("/check")
    public boolean checkStock(@RequestParam String productCode,
@RequestParam int quantity) {
        return quantity < 10; // Assume stock is available if quantity < 10
    }
}
```

✅ **Checks if stock is available.**

7.5 Service Discovery with Eureka

Eureka helps in **service discovery** so microservices can find each other dynamically.

Step 1: Add Eureka Dependencies (`pom.xml`)

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Step 2: Create Eureka Server (`eureka-server`)

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

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

✅ **All microservices will register with Eureka**.

7.6 API Gateway with Spring Cloud Gateway

The API Gateway acts as a single entry point.

Step 1: Add Gateway Dependencies (`pom.xml`)

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Step 2: Configure Gateway Routes (`application.yml`)

```
spring:
  cloud:
    gateway:
      routes:
        - id: order-service
          uri: lb://order-service
          predicates:
            - Path=/orders/**
        - id: inventory-service
          uri: lb://inventory-service
          predicates:
            - Path=/inventory/**
```

✅ **Routes requests to correct services.**

7.7 Handling Fault Tolerance with Resilience4j

To prevent failures, we use **Resilience4j Circuit Breaker**.

Step 1: Add Dependencies (`pom.xml`)

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

Step 2: Use Circuit Breaker in Order Service

```
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;

@Service
public class OrderService {

    @CircuitBreaker(name = "inventoryService", fallbackMethod =
"fallbackMethod")
    public String placeOrder(String productCode, int quantity) {
        return inventoryClient.checkStock(productCode, quantity) ? "Order
placed!" : "Out of stock!";
    }

    public String fallbackMethod(String productCode, int quantity,
Throwable ex) {
        return "Service unavailable. Try again later!";
    }
}
```

✅ **Handles failures gracefully**.