1. Build a User and Order Management System

Problem: Create two microservices:

* User Service to manage users.
* Order Service to manage orders placed by users.

Requirements:

* Use REST APIs.
* Communicate between services using WebClient (Spring WebFlux) or

OpenFeign.

* Store data in MySQL or PostgreSQL.

**Solution:**

**1. Project Structure**

This system will include two independent Spring Boot applications:

* **User Service**
* **Order Service**

Each service will have its own database and expose RESTful APIs. Communication between services will be handled using OpenFeign or WebClient.

**2. User Service**

**Dependencies (pom.xml):**

<dependencies>

<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>mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

</dependency>

</dependencies>

**User Entity:**

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

}

**User Repository:**

public interface UserRepository extends JpaRepository<User, Long> {

}

**User Controller:**

@RestController

@RequestMapping("/users")

public class UserController {

@Autowired

private UserRepository userRepository;

@PostMapping

public User createUser(@RequestBody User user) {

return userRepository.save(user);

}

@GetMapping("/{id}")

public User getUser(@PathVariable Long id) {

return userRepository.findById(id).orElse(null);

}

}

**application.properties:**

spring.datasource.url=jdbc:mysql://localhost:3306/userdb

spring.datasource.username=root

spring.datasource.password=your\_password

spring.jpa.hibernate.ddl-auto=update

**3. Order Service**

**Dependencies (pom.xml):**

<dependencies>

<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>mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

</dependency>

</dependencies>

**Order Entity:**

@Entity

public class Order {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private Long userId;

private String product;

private Double amount;

}

**Order Repository:**

public interface OrderRepository extends JpaRepository<Order, Long> {

List<Order> findByUserId(Long userId);

}

**Order Controller:**

@RestController

@RequestMapping("/orders")

public class OrderController {

@Autowired

private OrderRepository orderRepository;

@PostMapping

public Order createOrder(@RequestBody Order order) {

return orderRepository.save(order);

}

@GetMapping("/user/{userId}")

public List<Order> getOrdersByUser(@PathVariable Long userId) {

return orderRepository.findByUserId(userId);

}

}

**Feign Client (Optional for calling User Service):**

@FeignClient(name = "user-service", url = "http://localhost:8081")

public interface UserClient {

@GetMapping("/users/{id}")

User getUserById(@PathVariable Long id);

}

**Main Application Class:**

@EnableFeignClients

@SpringBootApplication

public class OrderServiceApplication {

public static void main(String[] args) {

SpringApplication.run(OrderServiceApplication.class, args);

}

}

**application.properties:**

spring.datasource.url=jdbc:mysql://localhost:3306/orderdb

spring.datasource.username=root

spring.datasource.password=your\_password

spring.jpa.hibernate.ddl-auto=update

**4. Communication Between Services**

To communicate between the services, OpenFeign can be used in the Order Service to fetch user details from the User Service.

Alternatively, WebClient from Spring WebFlux can be used if a reactive approach is preferred.

**5. Database Configuration**

Both services should have their own schema in MySQL or PostgreSQL. Configuration is done through the respective application.properties files.

**2.Inventory Management System with Service Discovery**

Problem: Create:

* Product Service: Manage products and stock.
* Inventory Service: Track stock levels for each product.

Requirements:

* Use Spring Cloud Netflix Eureka for service discovery.
* Implement centralized configuration using Spring Cloud Config Server.

### Solution:

#### 1. Setup: Spring Cloud Netflix Eureka for Service Discovery

##### 1.1 Eureka Server

**Dependencies (pom.xml):**

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

</dependencies>

**Main Class:**

@EnableEurekaServer

@SpringBootApplication

public class EurekaServerApplication {

public static void main(String[] args) {

SpringApplication.run(EurekaServerApplication.class, args);

}

}

**application.properties:**

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

Start the Eureka server at port 8761.

#### 2. Product Service

**Dependencies (pom.xml):**

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

</dependency>

</dependencies>

**Main Class:**

@EnableEurekaClient

@SpringBootApplication

public class ProductServiceApplication {

public static void main(String[] args) {

SpringApplication.run(ProductServiceApplication.class, args);

}

}

**Product Entity:**

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private int availableStock;

}

**Product Repository:**

public interface ProductRepository extends JpaRepository<Product, Long> {

}

**Product Controller:**

@RestController

@RequestMapping("/products")

public class ProductController {

@Autowired

private ProductRepository productRepository;

@PostMapping

public Product addProduct(@RequestBody Product product) {

return productRepository.save(product);

}

@GetMapping("/{id}")

public Product getProduct(@PathVariable Long id) {

return productRepository.findById(id).orElse(null);

}

}

**application.properties:**

spring.application.name=product-service

server.port=8081

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

spring.datasource.url=jdbc:mysql://localhost:3306/productdb

spring.datasource.username=root

spring.datasource.password=your\_password

spring.jpa.hibernate.ddl-auto=update

#### 3. Inventory Service

**Dependencies and Configuration** are similar to Product Service.

**Inventory Entity:**

@Entity

public class Inventory {

@Id

private Long productId;

private int stockLevel;

}

**Inventory Repository:**

public interface InventoryRepository extends JpaRepository<Inventory, Long> {

}

**Inventory Controller:**

@RestController

@RequestMapping("/inventory")

public class InventoryController {

@Autowired

private InventoryRepository inventoryRepository;

@PostMapping

public Inventory updateStock(@RequestBody Inventory inventory) {

return inventoryRepository.save(inventory);

}

@GetMapping("/{productId}")

public Inventory getStock(@PathVariable Long productId) {

return inventoryRepository.findById(productId).orElse(null);

}

}

**application.properties:**

spring.application.name=inventory-service

server.port=8082

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

spring.datasource.url=jdbc:mysql://localhost:3306/inventorydb

spring.datasource.username=root

spring.datasource.password=your\_password

spring.jpa.hibernate.ddl-auto=update

#### 4. Spring Cloud Config Server (Centralized Configuration)

**Config Server Setup**

**Dependencies:**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

**Main Class:**

@EnableConfigServer

@SpringBootApplication

public class ConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigServerApplication.class, args);

}

}

**application.properties (for config server):**

server.port=8888

spring.cloud.config.server.git.uri=https://github.com/your-repo/config-repo

spring.cloud.config.server.git.clone-on-start=true

Each client (Product and Inventory services) will include:

spring.config.import=optional:configserver:http://localhost:8888

spring.application.name=product-service

**3.Implement an API Gateway**

Problem: Create an API Gateway to route requests to:

* Customer Service
* Billing Service

Requirements:

* Use Spring Cloud Gateway.
* Implement rate limiting, caching, and path rewriting.

### Exercise 3: Implement an API Gateway

#### Problem Statement:

Create an API Gateway to route requests to:

* Customer Service
* Billing Service

#### Requirements:

* Use Spring Cloud Gateway
* Implement rate limiting, caching, and path rewriting

### Solution:

#### 1. API Gateway Setup Using Spring Cloud Gateway

**Dependencies (pom.xml):**

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot3</artifactId>

</dependency>

</dependencies>

**Main Application Class:**

@EnableEurekaClient

@SpringBootApplication

public class ApiGatewayApplication {

public static void main(String[] args) {

SpringApplication.run(ApiGatewayApplication.class, args);

}

}

#### 2. application.yml Configuration

spring:

application:

name: api-gateway

cloud:

gateway:

discovery:

locator:

enabled: true

routes:

- id: customer-service

uri: lb://CUSTOMER-SERVICE

predicates:

- Path=/customers/\*\*

filters:

- RewritePath=/customers/(?<segment>.\*), /${segment}

- name: RequestRateLimiter

args:

redis-rate-limiter.replenishRate: 5

redis-rate-limiter.burstCapacity: 10

- id: billing-service

uri: lb://BILLING-SERVICE

predicates:

- Path=/billing/\*\*

filters:

- RewritePath=/billing/(?<segment>.\*), /${segment}

- name: RequestRateLimiter

args:

redis-rate-limiter.replenishRate: 3

redis-rate-limiter.burstCapacity: 6

eureka:

client:

service-url:

defaultZone: http://localhost:8761/eureka

Note: Redis dependency and configuration are required for rate limiting.

#### 3. Implementing Rate Limiting

To enable rate limiting:

* Add Redis as a dependency (spring-boot-starter-data-redis)
* Configure Redis host in application.yml

spring:

redis:

host: localhost

port: 6379

#### 4. Path Rewriting

Using the RewritePath filter, incoming paths like /customers/getAll will be internally routed as /getAll to the customer service.

#### 5. Optional: Caching Layer

For simple in-memory or distributed caching, Spring Cloud Gateway does not include native caching. However, caching can be implemented:

* At service level using @Cacheable with Redis or Caffeine
* Or by adding a gateway filter to cache responses (via third-party filters)

**4. Resilient Microservices with Circuit Breaker**

Problem: A Payment Service calls a slow third-party API.

Requirements:

* Implement Circuit Breaker and fallback logic using Resilience4j.
* Log and monitor fallback events.

### Solution:

#### 1. Setup Resilience4j in Payment Service

**Dependencies (pom.xml):**

<dependencies>

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot3</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

#### 2. External API Call Using RestTemplate or WebClient

**Service with Circuit Breaker (Example using RestTemplate):**

@Service

public class PaymentService {

@Autowired

private RestTemplate restTemplate;

@CircuitBreaker(name = "paymentCB", fallbackMethod = "fallbackForPayment")

public String callPaymentProvider() {

return restTemplate.getForObject("http://thirdparty.com/api/payment", String.class);

}

public String fallbackForPayment(Throwable t) {

// Log the error and return a fallback response

return "Payment service is currently unavailable. Please try again later.";

}

}

**RestTemplate Configuration:**

@Configuration

public class AppConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

#### 3. application.yml Configuration

resilience4j:

circuitbreaker:

instances:

paymentCB:

registerHealthIndicator: true

slidingWindowSize: 10

minimumNumberOfCalls: 5

failureRateThreshold: 50

waitDurationInOpenState: 10s

permittedNumberOfCallsInHalfOpenState: 3

automaticTransitionFromOpenToHalfOpenEnabled: true

eventConsumerBufferSize: 10

#### 4. Logging Fallback Events

You can configure logging in application.properties or logback-spring.xml:

logging.level.io.github.resilience4j.circuitbreaker=INFO

Or log manually in the fallback method:

public String fallbackForPayment(Throwable t) {

Logger logger = LoggerFactory.getLogger(PaymentService.class);

logger.error("Fallback executed due to: {}", t.getMessage());

return "Fallback: payment service unavailable.";

}

#### 5. Monitoring

Resilience4j integrates with Micrometer for metrics. You can visualize circuit breaker states using tools like Prometheus and Grafana, or expose actuator endpoints.

Add actuator dependency:

<dependency>

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

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

</dependency>

Enable actuator endpoint:

management:

endpoints:

web:

exposure:

include: resilience4j.circuitbreakers