PAKHI SHARMA

Exercises on Microservices with Spring Boot 3.0

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 **MySǪL** or **PostgreSǪL**.

**SOLUTION**

**User Service**

@Entity

public class User {

@Id @GeneratedValue

private Long id;

private String name;

private String email;

}

public interface UserRepository extends JpaRepository<User, Long> {}

@RestController

@RequestMapping("/users")

@RequiredArgsConstructor

public class UserController {

private final UserRepository repo;

@PostMapping

public User saveUser(@RequestBody User user) {

return repo.save(user);

}

@GetMapping("/{id}")

public User getUser(@PathVariable Long id) {

return repo.findById(id).orElseThrow();

}

}

**Order Service**

@FeignClient(name = "user-service", url = "http://localhost:8080")

public interface UserClient {

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

User getUser(@PathVariable Long id);

}

@Entity

public class Order {

@Id @GeneratedValue

private Long id;

private Long userId;

private String product;

}

@RestController

@RequestMapping("/orders")

@RequiredArgsConstructor

public class OrderController {

private final OrderRepository repo;

private final UserClient userClient;

@PostMapping

public Order createOrder(@RequestBody Order order) {

userClient.getUser(order.getUserId());

return repo.save(order);

}

}

1. **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**

**Eureka Server Application**

@SpringBootApplication

@EnableEurekaServer

public class EurekaServerApplication {

public static void main(String[] args) {

SpringApplication.run(EurekaServerApplication.class, args);

}

}

application.yml (Eureka Server)

server:

port: 8761

eureka:

client:

register-with-eureka: false

fetch-registry: false

Product Service

@EnableDiscoveryClient

@SpringBootApplication

public class ProductServiceApp {

public static void main(String[] args) {

SpringApplication.run(ProductServiceApp.class, args);

}

}

**Inventory Service**

* Uses Feign/WebClient to call Product Service for product details.
* Tracks stock in DB.

1. **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**.

**SOLUTION**

**Spring Cloud Gateway Application**

@SpringBootApplication

@EnableDiscoveryClient

public class ApiGatewayApplication {

public static void main(String[] args) {

SpringApplication.run(ApiGatewayApplication.class, args);

}

}

**application.yml**

spring:

cloud:

gateway:

routes:

- id: customer\_service

uri: lb://CUSTOMER-SERVICE

predicates:

- Path=/customers/\*\*

filters:

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

- id: billing\_service

uri: lb://BILLING-SERVICE

predicates:

- Path=/billing/\*\*

default-filters:

- name: RequestRateLimiter

args:

redis-rate-limiter.replenishRate: 5

redis-rate-limiter.burstCapacity: 10

1. **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**

**Payment Service** (Resilience4j)

@RestController

@RequestMapping("/payments")

@RequiredArgsConstructor

public class PaymentController {

private final PaymentService paymentService;

@GetMapping("/{id}")

public String pay(@PathVariable Long id) {

return paymentService.processPayment(id);

}

}

@Service

public class PaymentService {

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

public String processPayment(Long id) {

try { Thread.sleep(5000); } catch (InterruptedException e) {}

return "Payment successful for ID: " + id;

}

public String paymentFallback(Long id, Throwable t) {

return "Payment service is busy. Please try later.";

}

}

**application.yml**

resilience4j:

circuitbreaker:

instances:

paymentCB:

register-health-indicator: true

sliding-window-size: 5

failure-rate-threshold: 50

wait-duration-in-open-state: 5s