**WEEK – 5**

**MICROSERVICES WITH API GATEWAY**

**EXERCISE – 1:** Creating Microservices For Account And Loan.

1. **Setting Up Each Microservice.**

**1.1 Generate the Spring Boot Projects.**

* Use Spring Initializr (<https://start.spring.io>):
  + Account Service: Group com.cognizant, Artifact account
  + Loan Service: Group com.cognizant, Artifact loan
  + Add dependencies: Spring Web and Spring Boot DevTools

This sets up independent projects with their own pom.xml and embedded servers.

* 1. **Import & Build.**

cd account

mvn clean package

cd ../loan

mvn clean package

Import into Eclipse or IntelliJ as standalone applications.

**2. Implementing the REST Controllers**

**2.1 Account Controller**

@RestController

@RequestMapping("/accounts")

public class AccountController {

@GetMapping("/{number}")

public Account getAccount(@PathVariable String number) {

return new Account(number, "savings", 234343);

}

}

public record Account(String number, String type, double balance) {}

**2.2 Loan Controller**

@RestController

@RequestMapping("/loans")

public class LoanController {

@GetMapping("/{number}")

public Loan getLoan(@PathVariable String number) {

return new Loan(number, "car", 400000, 3258, 18);

}

}

public record Loan(String number, String type, int loan, int emi, int tenure) {}

These return hard-coded responses without a database—ideal for simple testing or demos.

1. **Running & Testing Endpoints**

**1. Account Service:**

* Run on port **8080** (default).
* Test with: http://localhost:8080/accounts/00987987973432

**2. Loan Service**:

* + Add server.port=8081 to application.properties
  + Run while the account service is active.
  + Test with: http://localhost:8081/loans/H00987987972342

You now have *two independent, running microservices.*

**4. Why This Microservices Setup Matters**

* + **Loose Coupling**: Each service works independently and can be scaled or modified without affecting the other.
  + **Isolation of Responsibility**: Account-related logic is decoupled from loan logic.
* Ideal base for integrating advanced **microservice architecture patterns**:
* **Service Discovery** (e.g., **Eureka**)
* **API Gateway**, **Resilience Patterns** (Circuit Breaker via Resilience4j)

**5. Scaling Up: Integrating with a Real Microservices Ecosystem**

**5.1 Add Service Discovery**

* Create a Eureka Server, annotate with @EnableEurekaServer
* Configure services to register with Eureka.
* Enables dynamic discovery for load balancing and scaling.

**5.2 Use an API Gateway**

* Introduce Spring Cloud Gateway or Zuul for single-entrypoint routing.
* Enforce centralized concerns like auth and request throttling.

**5.3 Add Resilience**

* Include Resilience4j to implement retries, circuit breakers
* Example scenario: loan service querying a rate service.

**5.4 Centralized Config**

* Use Spring Cloud Config to manage configuration centrally for both services

**5.5 Asynchronous Messaging**

* Use Kafka or RabbitMQ for inter-service communication beyond REST
* Enables event-driven flows (e.g., account created → sends message → loan service subscribes)

**5.6 Packaging & Deployment**

* Consider Docker or Kubernetes for containerized deployment.
* Enables microservices orchestration with CI/CD pipelines.

**6. Additional Resources & Examples**

* **Core banking microservices** using Docker + PostgreSQL: includes account, loan, transaction services.
* **Banking app on GitHub**: user, account, transactions, fund transfers + service registry + API gateway.
* **Auth0 tutorial**: demonstrates building secure microservices with Eureka, Gateway, and Resilience4j .
* **Multi-module projects**: share domain models via Maven modules for clean structure .