**WEEK 5-MANDATORY HANDS ON EXERCISES**

**Module 8 - Microservices with Spring Boot 3 and Spring Cloud**

**Creating Microservices for Account and Loan — My Hands-on Journey**

In this practical session, I created two independent Spring Boot microservices for a bank — one to handle Accounts, and the other for Loans. Each service is a separate Maven-based Spring RESTful web application with its own pom.xml. Instead of combining everything into one monolith (old-school style), the system is split into two clean, focused services.

Both services are basic — no backend/database connectivity, just dummy data for now. Think of it as the skeleton before we flesh it out.

**Account Microservice**

**Setup:**

1. Created a folder with my employee ID on the D: drive.

2. Inside it, created a microservices folder to store all future sample microservices.

3. Jumped over to [https://start.spring.io](https://start.spring.io/):

o **Group:** com.cognizant

o **Artifact:** account

o Selected:

§ Spring Boot DevTools

§ Spring Web

4. Generated and downloaded the zip, then extracted it to the microservices folder.

**Development:**

5. Navigated into the account folder via command prompt and ran:

mvn clean package

6. Imported the project into Eclipse.

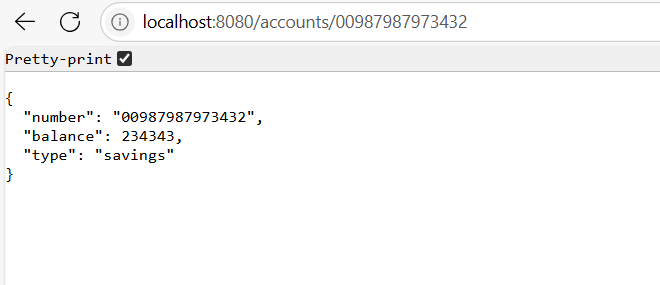
7. Created a controller with a GET endpoint:

/accounts/{number}

8. Returned dummy data (no DB):

9. Ran the application and tested on port 8080.

**OUTPUT**:



**Loan Microservice**

**Setup:**

1. Repeated the same steps from the Account service.

2. This time, the Artifact was loan.

**Development:**

3. Implemented a controller with this GET endpoint:

/loans/{number}

4. Returned dummy loan details

5. Modified application.properties of the Loan microservice:

server.port=8081

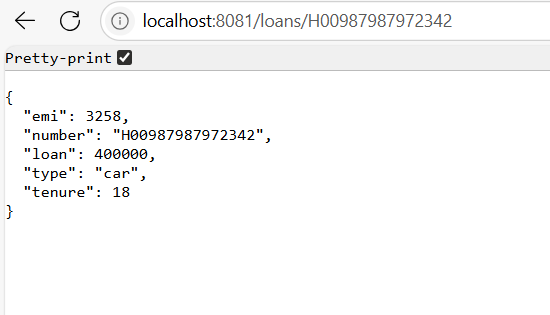
6. Relaunched — now running smoothly on port 8081.

**Outcome**

Both microservices are now:

* Independently runnable
* Exposing simple REST endpoints
* Running on separate ports (8080 for Account, 8081 for Loan)

**Output:**



**WEEK 5-ADDITIONAL HANDS ON EXERCISES**

**Create Eureka Discovery Server and register microservices**

**Setting up Eureka Discovery, Registering Microservices, and Routing via Spring Cloud API Gateway**

Microservices are like that group project where each member finally does their part independently — and Eureka is keeping track of who's present. Here's how I went from zero to a functional discovery ecosystem with API routing and request logging.

**Step 1: Creating Eureka Discovery Server**

Eureka Discovery Server is where all our microservices announce, “Hey, I’m online!” Consumers can then discover and call these services without hardcoding URLs.

**Setup:**

1. Went to [start.spring.io](https://start.spring.io/) and configured:

o **Group:** com.cognizant

o **Artifact:** eureka-discovery-server

o **Dependencies:** Spring Cloud Discovery > Eureka Server

2. Downloaded the zip, extracted, and built it:

mvn clean package

3. Imported into **Eclipse**, and in the main class, added:

@EnableEurekaServer

4. Then I opened application.properties and dropped in:

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

logging.level.com.netflix.eureka=OFF

logging.level.com.netflix.discovery=OFF

5. Ran the app and visited [http://localhost:8761](http://localhost:8761/) — Eureka dashboard showed up! (Empty for now, but not for long.)

**Step 2: Registering Microservices (Account & Loan)**

Next up: get the **Account** and **Loan** microservices to introduce themselves to Eureka.

**For Each Microservice:**

1. Headed to [start.spring.io](https://start.spring.io/) and reconfigured:

o **Group:** com.cognizant

o **Artifact:** account or loan

o **Dependencies:**

§ Spring Web

§ Spring Boot DevTools

§ Eureka Discovery Client

2. Used the “Explore” button to grab the generated pom.xml and pasted it into the existing account/loan project to update dependencies.

3. Rebuilt with:

mvn clean install

4. In AccountApplication.java / LoanApplication.java, added:

@EnableDiscoveryClient

5. Updated application.properties:

spring.application.name=account-service # or loan-service

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

6. Killed all running services in Eclipse (Eureka, Account, Loan) — used the Console View > Monitor icon > Terminate.

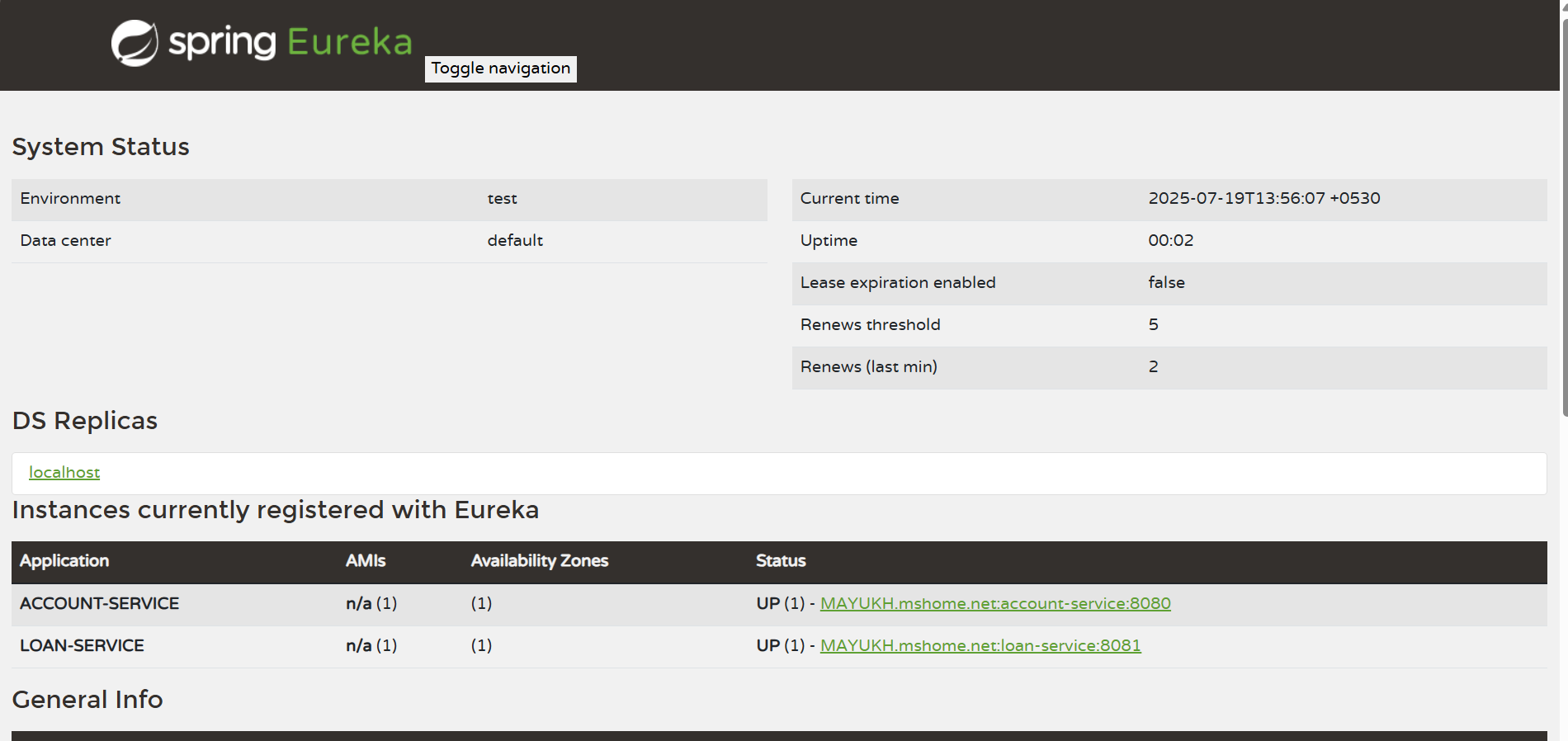
7. Restarted in this order:

o Eureka Server (localhost:8761)

o Account Service

o Loan Service

8. Boom — both services appeared in the Eureka dashboard under **Instances currently registered with Eureka**



**Step 3: Creating an API Gateway & Logging Requests**

Now to make things fancy — a gateway to route all requests + log them globally.

**Greet Service**

1. Created a new microservice greet-service via Spring Initializer:

o **Dependencies:**

§ Spring Web

§ Eureka Discovery Client

*2. Configured application.properties:*

spring.application.name=greet-service

server.port=8082

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

3. Made a simple controller:

@RestController

public class GreetController {

@GetMapping("/greet")

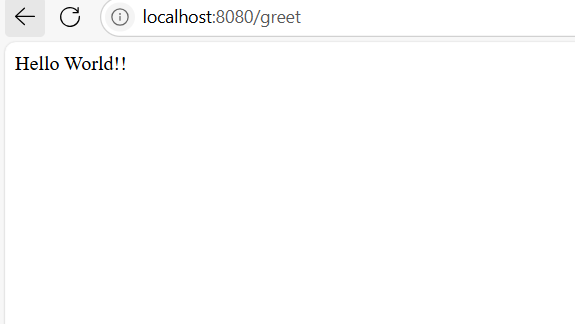
public String greet() {

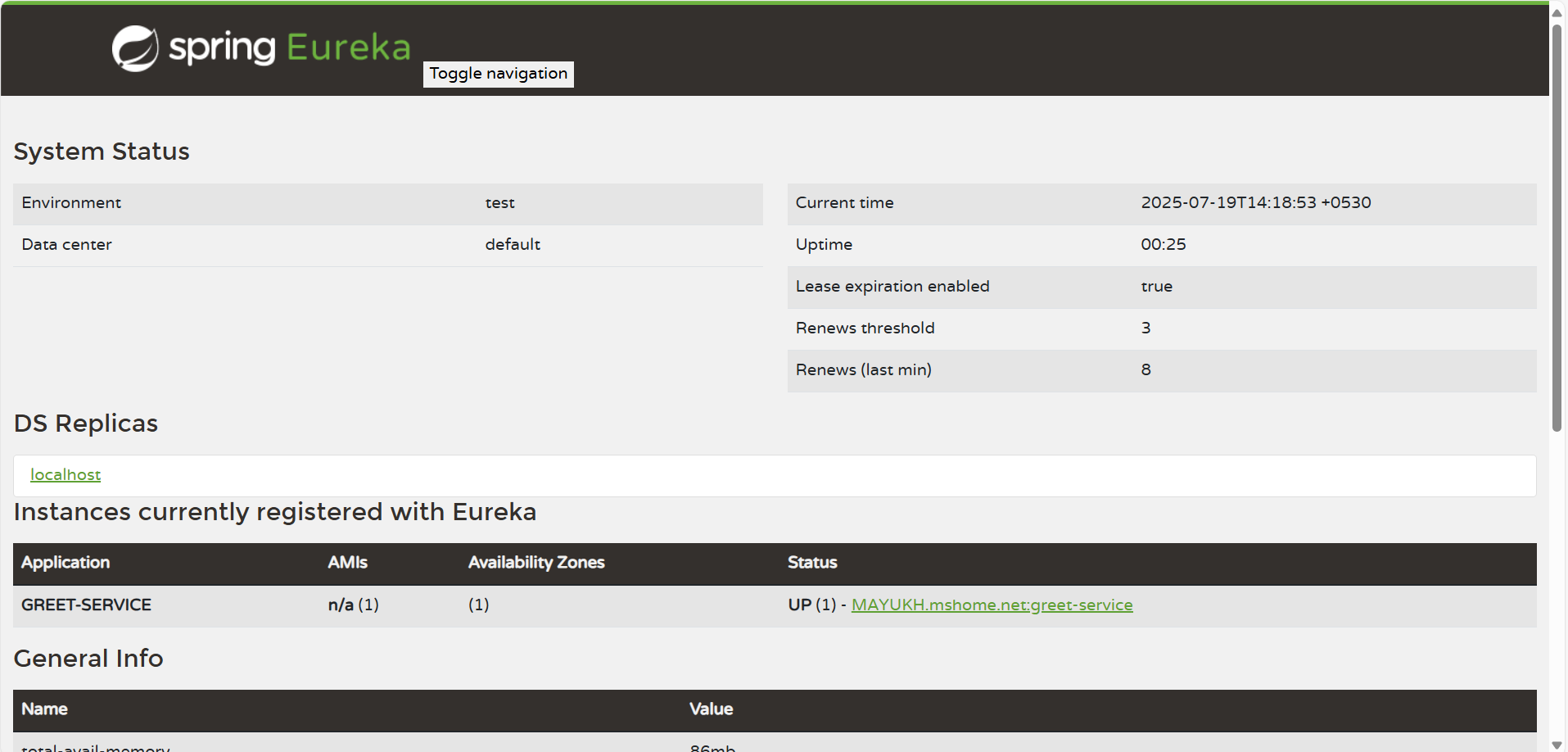
return "Hello World";

}

}

4. Ran it, confirmed it registered with Eureka, and /greet worked on port 8080.





**API Gateway**

1. Created another service: api-gateway with:

o Spring Cloud Gateway

o Eureka Discovery Client

2. Configured application.properties:

spring.application.name=api-gateway

server.port=9090

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

spring.cloud.gateway.discovery.locator.enabled=true

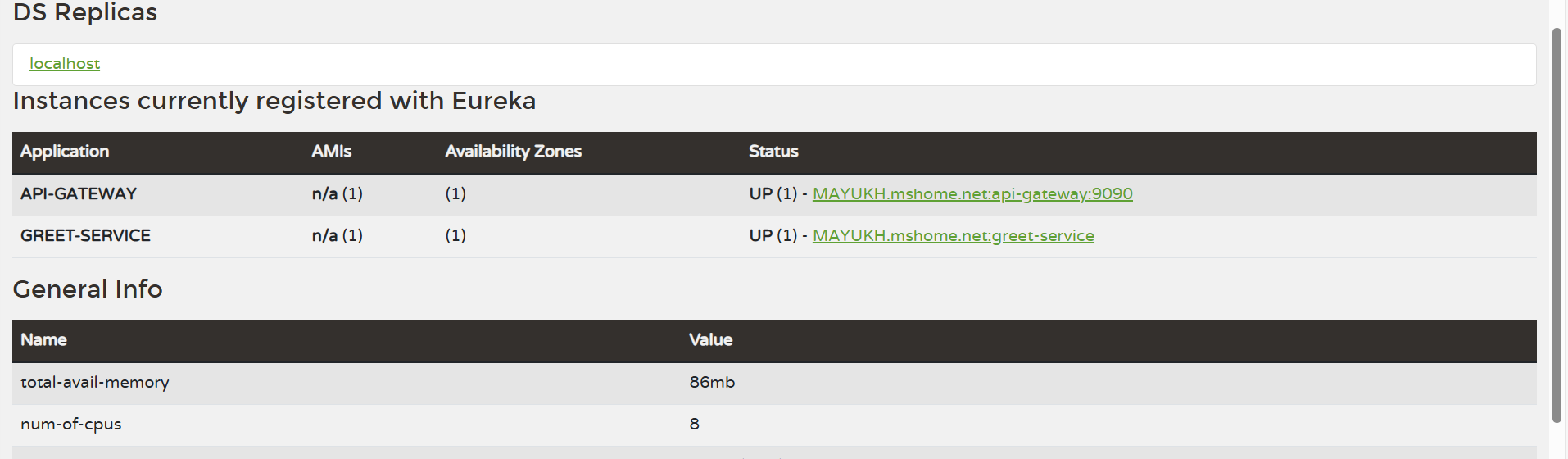
spring.cloud.gateway.discovery.locator.lower-case-service-id=true

3. Confirmed api-gateway showed up in Eureka.

4. Accessed the greet service via:

http://localhost:9090/greet-service/greet

It routed correctly



**Logging All Incoming Requests — Global Filter**

To track every hit to the API Gateway (because devs *love* logs):

1. Created LogFilter.java in api-gateway:

@Component

public class LogFilter implements GlobalFilter, Ordered {

@Override

public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

System.out.println("Incoming Request: " + exchange.getRequest().getURI());

return chain.filter(exchange);

}

@Override

public int getOrder() {

return -1;

}

}

2. Accessed:

http://localhost:9090/greet-service/greet

3. Checked the Gateway console — got:

Incoming Request: /greet-service/greet

And that’s how I made my gateway paranoid enough to log every incoming guest

**Final Result**

| **Component** | **Port** | **Registered with Eureka?** | **Role** |
| --- | --- | --- | --- |
| Eureka Server | 8761 | No | Service registry |
| Account Microservice | 8080 | Yes | Provides account info |
| Loan Microservice | 8081 | Yes | Provides loan info |
| Greet Microservice | 8080 | Yes | Returns "Hello World" |
| API Gateway | 9090 | Yes | Routes & logs requests |