SERVICE DISCOVERY WITH EUREKA

SPRING BOOT MICROSERVICES

Rohan Thapa
thaparohan2019@gmail.com

What is Service Discovery?

In a **microservice ecosystem**, there are **multiple services** communicating with each other.

Service discovery allows services to **find each other** dynamically, without the need for **hardcoded URLs**.

Eureka is one of the most popular service discovery solutions. It's part of the **Netflix OSS suite** and integrated with **Spring Cloud** for easy use.

Why Use Eureka?

- **Dynamic Scaling:** Services can come and go (scale up or down), and with Eureka, they can register or deregister themselves automatically.
- **Load Balancing:** Eureka works with **Ribbon** or **Spring Cloud LoadBalancer** to distribute traffic across instances of a service.
- **Resiliency:** Eureka helps in building **resilient systems** by **retrying connections** or **rerouting requests** if a service instance is down.

Key Components of Eureka

- **Eureka Server:** The **registry** where all services register themselves.
- **Eureka Client:** A **microservice** that registers itself with the Eureka Server and can **discover other services** from the registry.
- **Service Discovery:** Eureka clients can fetch the list of available services and make requests dynamically.

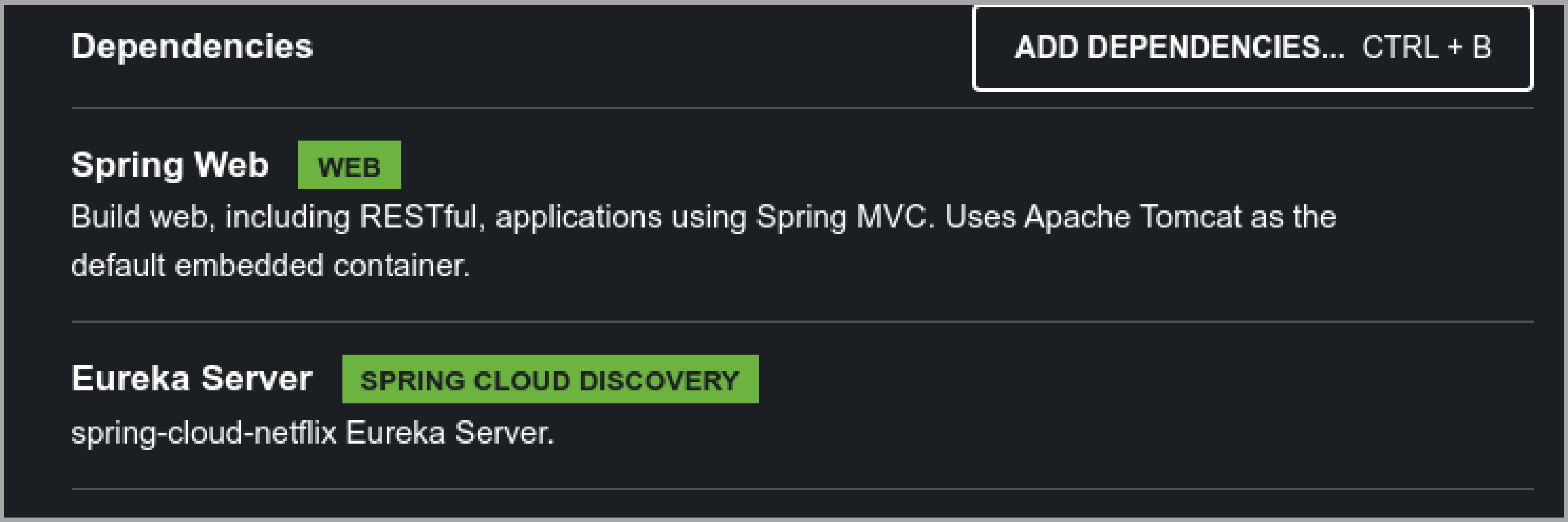
Step-by-Step Example:

We will set up **three components**:

1. **Eureka Server:** Central registry for service discovery.
2. **Hotel Service:** A **microservice** that registers with Eureka.
3. **Room Service:** Another **microservice** that uses Eureka to discover and communicate with the Product Service.

Setting Up the Eureka Server

Create a **New Spring Boot Project** for the Eureka Server.

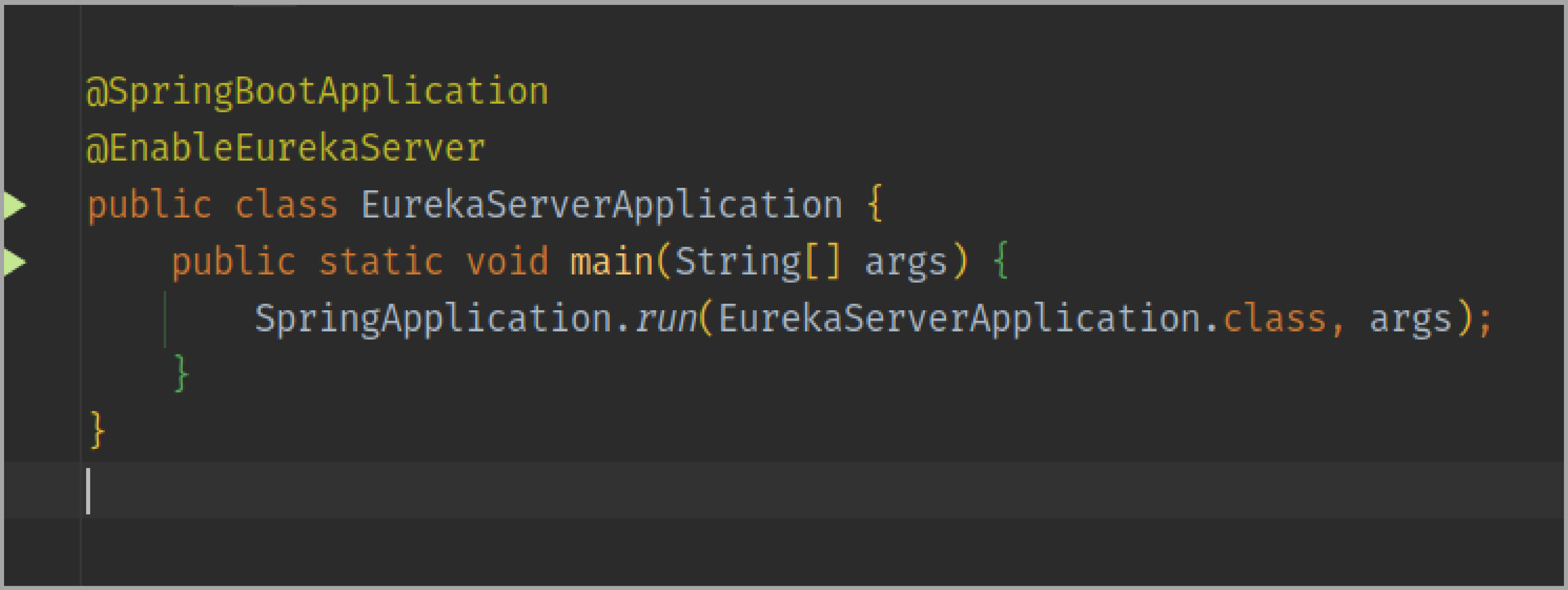


Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Eureka Server SPRING CLOUD DISCOVERY
spring-cloud-netflix Eureka Server.

Enable the Eureka Server

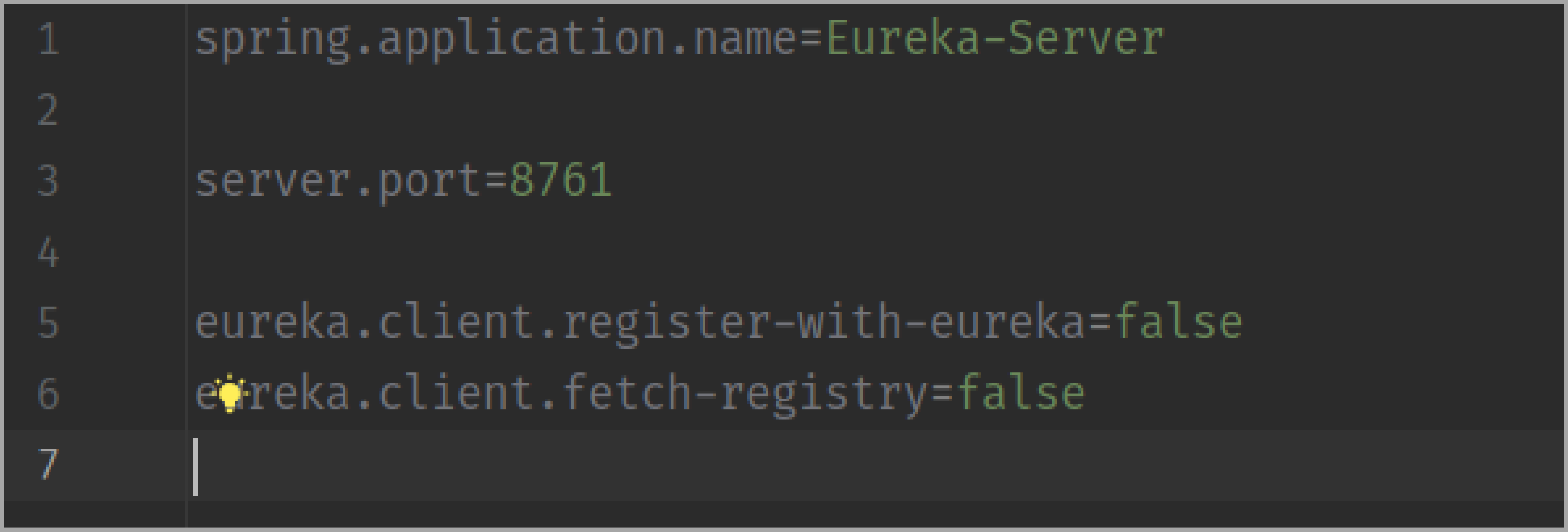
Enable the **Eureka Server** by annotating the main class with **@EnableEurekaServer**:



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

Configure the Eureka Server

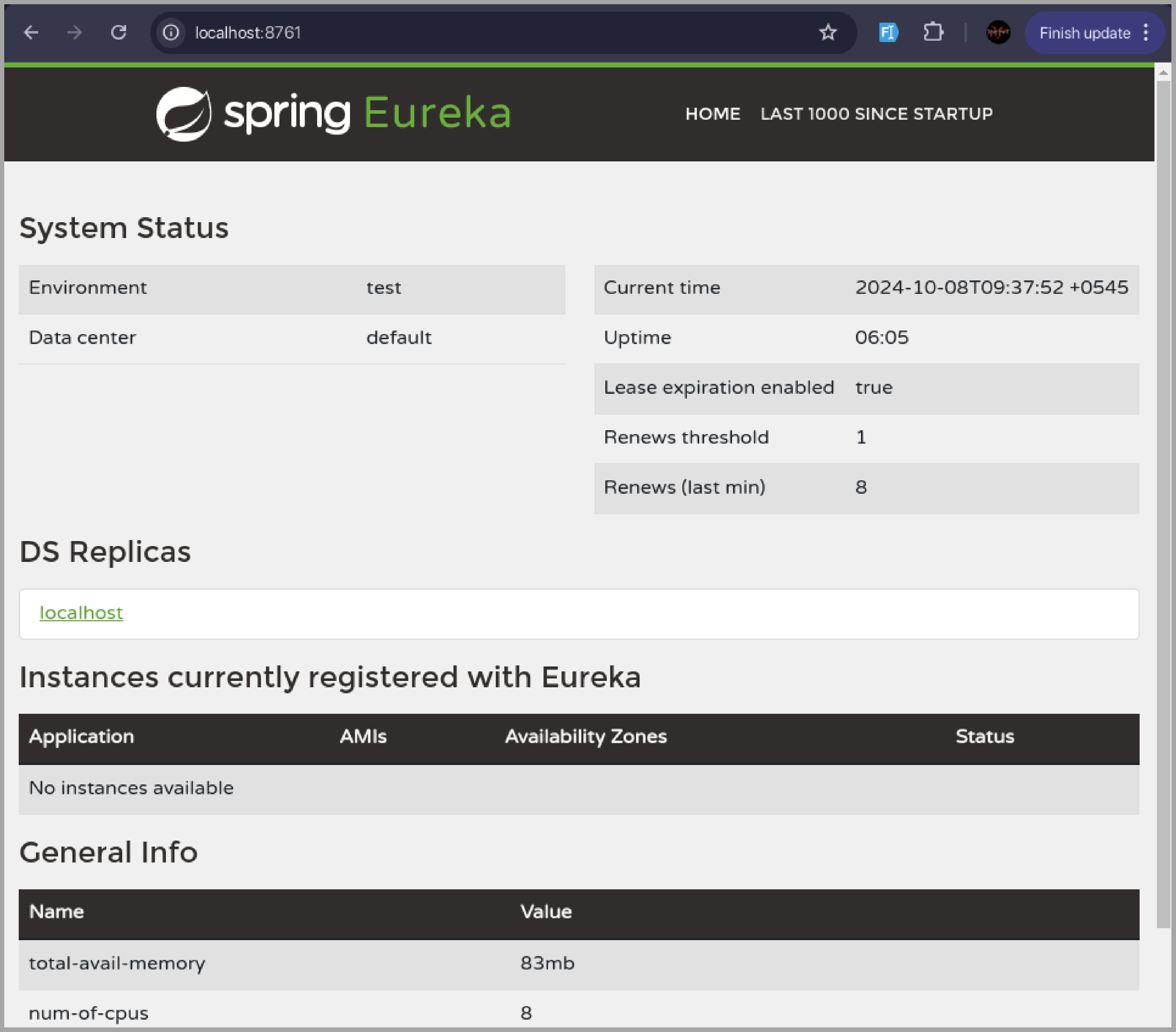
Configure the Eureka Server in application.properties or application.yml to not register as a client.



```
spring.application.name=Eureka-Server

server.port=8761

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false

```

Run the Eureka Server

Run the Eureka Server and go to **http://localhost:8761** to see the **Eureka dashboard**. It should show the status of registered services **(not registered yet)**.



System Status

Environment	test
Data center	default

Current time	2024-10-08T09:37:52 +0545
Uptime	06:05
Lease expiration enabled	true
Renews threshold	1
Renews (last min)	8

DS Replicas

localhost

Instances currently registered with Eureka

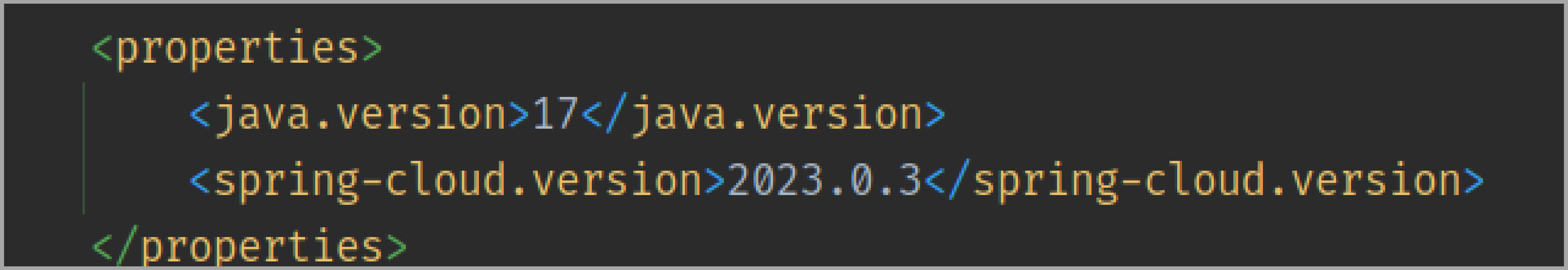
Application	AMIs	Availability Zones	Status
No instances available			

General Info

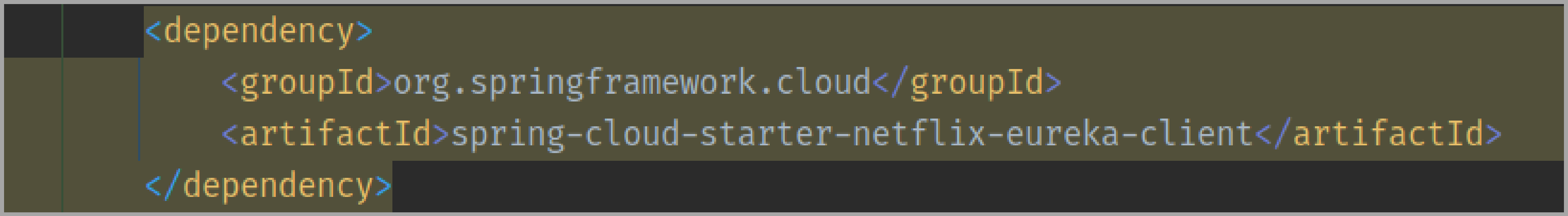
Name	Value
total-avail-memory	83mb
num-of-cpus	8

Setting Up the Hotel Service & Room Service (Eureka Client)

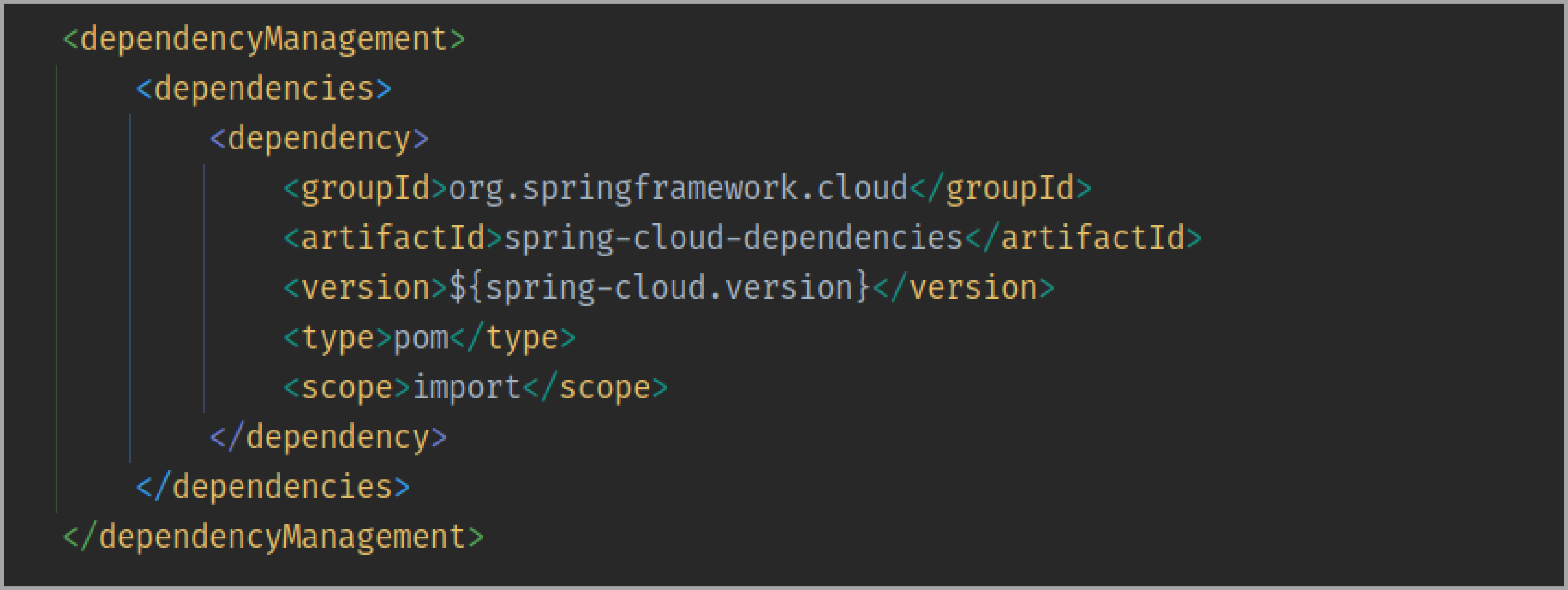
Add the dependencies in your **pom.xml**:



```xml
<properties>
    <java.version>17</java.version>
    <spring-cloud.version>2023.0.3</spring-cloud.version>
</properties>
```



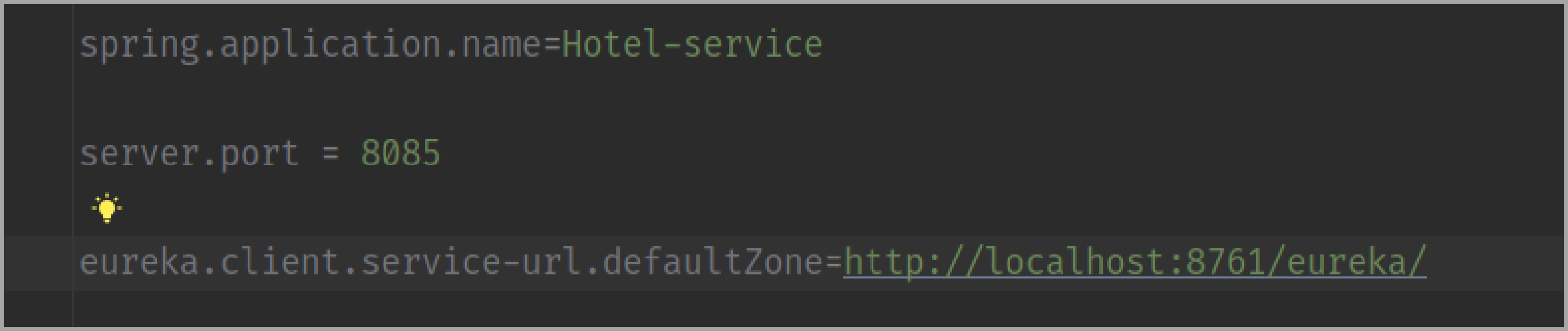
```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```



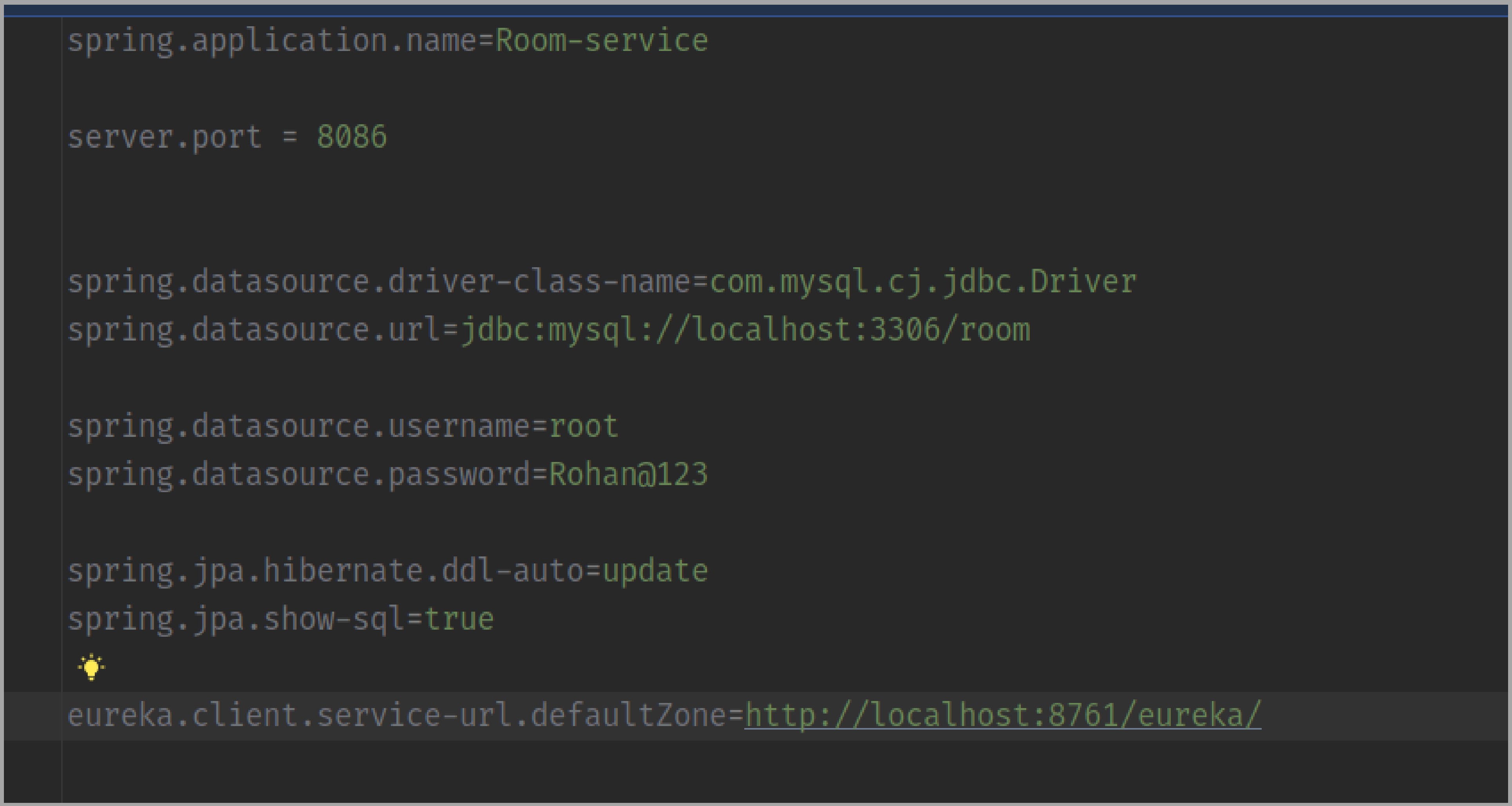
```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Configure Eureka

Configure Eureka in application.yml:
for both services



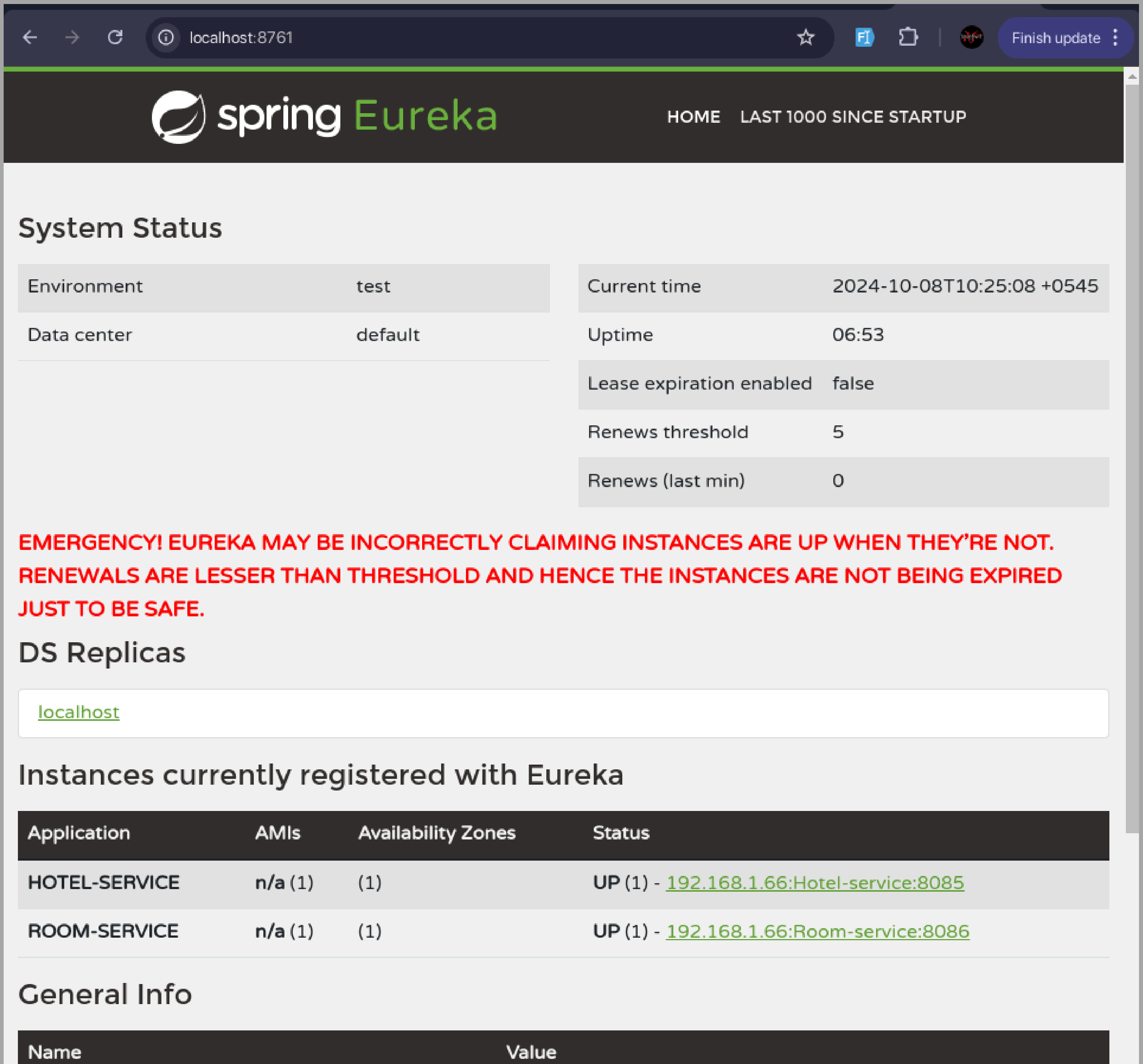
```
spring.application.name=Hotel-service

server.port = 8085

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



```
spring.application.name=Room-service

server.port = 8086


spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/room

spring.datasource.username=root
spring.datasource.password=Rohan@123

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

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

Run the Both Service

Run the Hotel Service and Room Service and it will register itself with the Eureka Server. Check the **Eureka dashboard**, and you should see both service registered.



System Status

Environment	test
Data center	default

Current time	2024-10-08T10:25:08 +0545
Uptime	06:53
Lease expiration enabled	false
Renews threshold	5
Renews (last min)	0

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

localhost

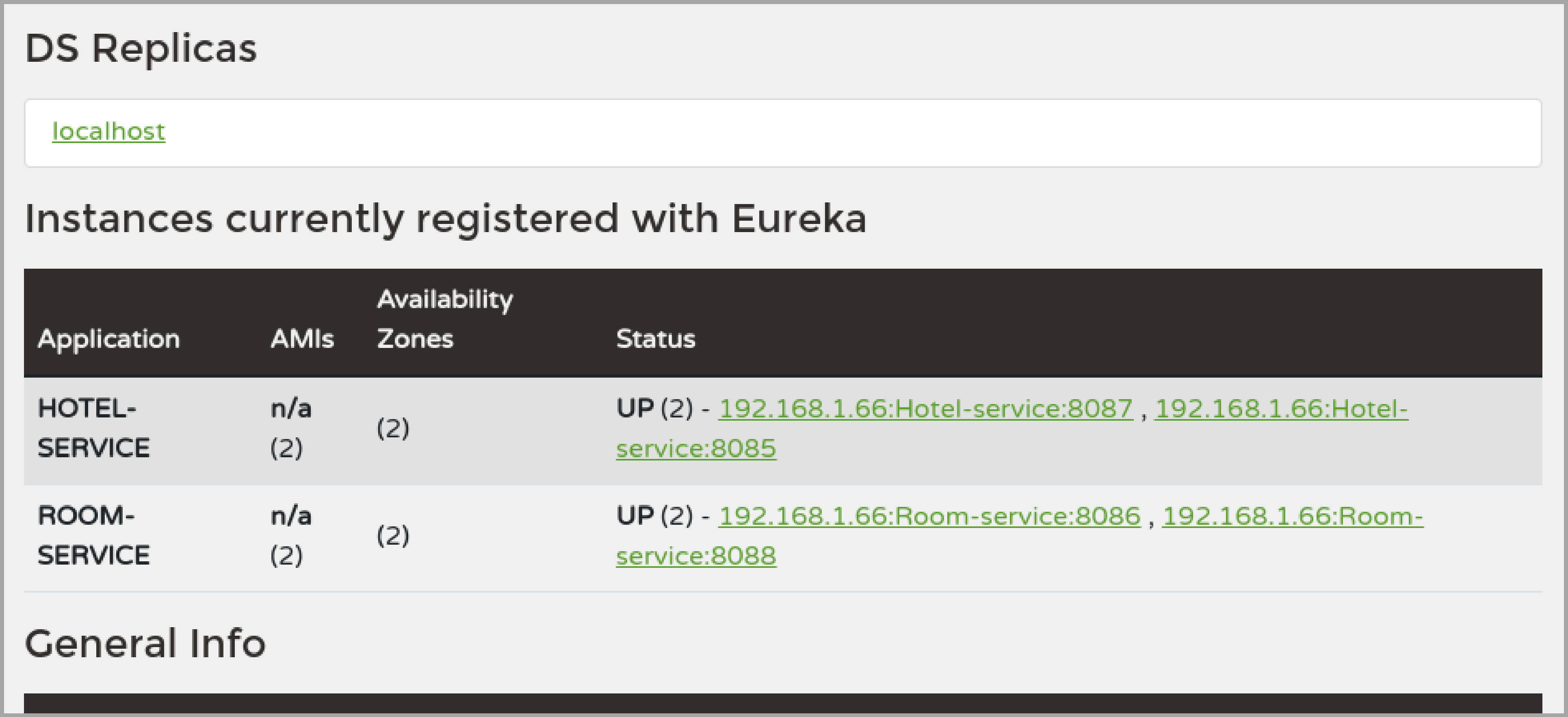
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
HOTEL-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.66:Hotel-service:8085
ROOM-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.66:Room-service:8086

General Info

Name	Value

Running multiple instance



DS Replicas

localhost

Instances currently registered with Eureka

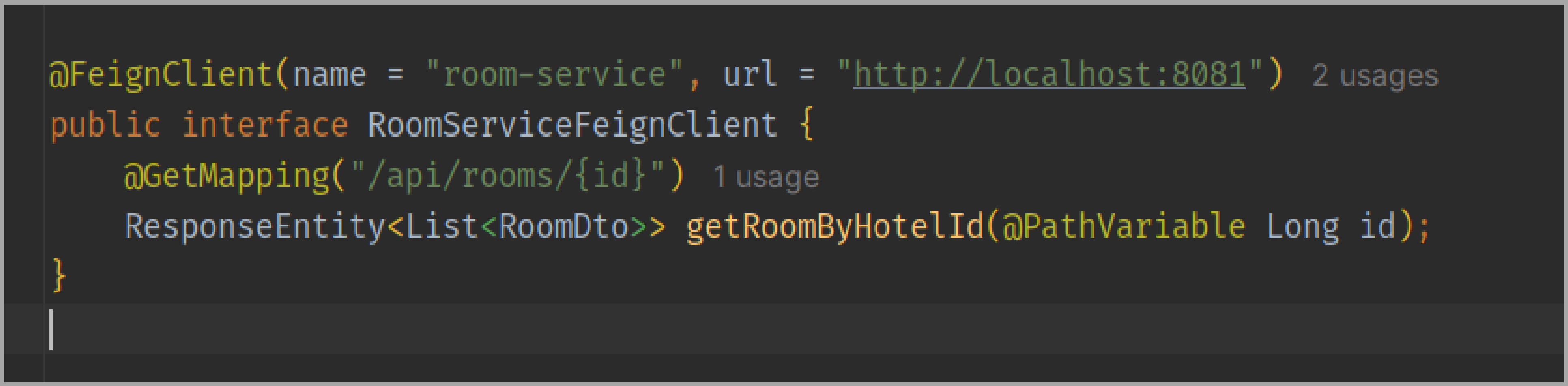
Application	AMIs	Availability Zones	Status
HOTEL-SERVICE	n/a (2)	(2)	UP (2) - 192.168.1.66:Hotel-service:8087 , 192.168.1.66:Hotel-service:8085
ROOM-SERVICE	n/a (2)	(2)	UP (2) - 192.168.1.66:Room-service:8086 , 192.168.1.66:Room-service:8088

General Info

Feign Client

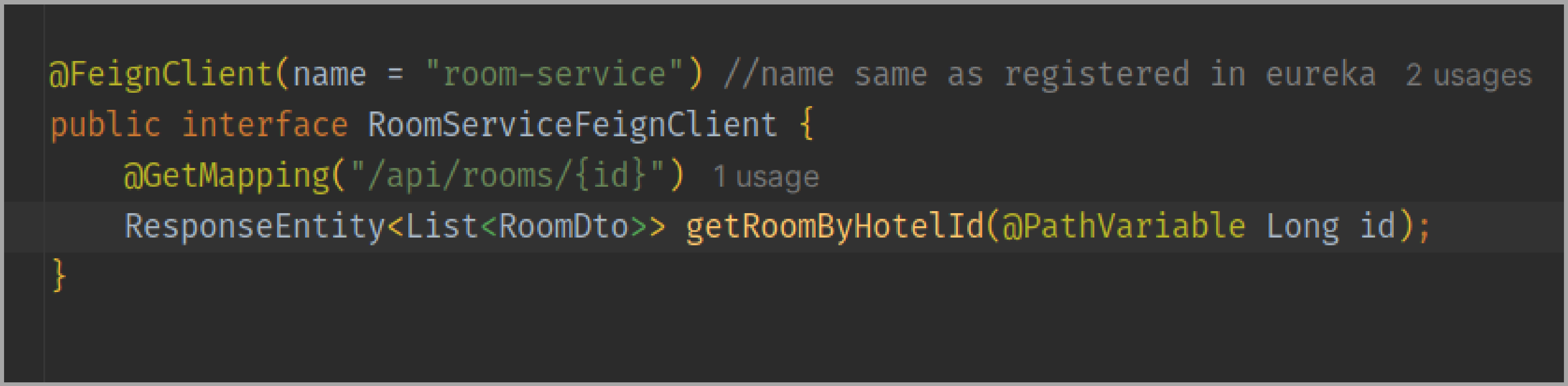
Now feign client service name can be same as registered in Eureka so that no URL should be defined manually.

Before Eureka:



```
@FeignClient(name = "room-service", url = "http://localhost:8081")  2 usages
public interface RoomServiceFeignClient {
    @GetMapping("/api/rooms/{id}")  1 usage
    ResponseEntity<List<RoomDto>> getRoomByHotelId(@PathVariable Long id);
}
```

After Eureka:



```
@FeignClient(name = "room-service") //name same as registered in eureka  2 usages
public interface RoomServiceFeignClient {
    @GetMapping("/api/rooms/{id}")  1 usage
    ResponseEntity<List<RoomDto>> getRoomByHotelId(@PathVariable Long id);
}
```

How Eureka Works

- **Service Registration:** Each service **(Hotel, Room)** registers itself with the **Eureka Server at startup**.
- **Heartbeat Mechanism:** Eureka clients send regular **heartbeats** to the **Eureka server** to indicate they are **still alive**.
- **Service Discovery:** When the Hotel Service needs to call the Room Service, it queries the Eureka registry to find the available instances of the Room Service.
- **Resilience:** If a **service fails** or stops sending heartbeats, Eureka removes it from the registry. This ensures that the Hotel Service won’t try to call a dead instance.

Additional Feature

- **Self-Preservation Mode:** Eureka enters a self-preservation mode when it detects too many service failures (to avoid purging healthy instances).

- **Load Balancing:** Eureka works well with **Ribbon** or **Spring Cloud LoadBalancer** to distribute requests across multiple instances of a service.

- **Clustered Eureka Servers:** For **redundancy**, you can run multiple Eureka servers that sync with each other.

Conclusion

Eureka simplifies **service discovery** in a **microservice architecture**.

With Eureka Server acting as a registry and Eureka Clients dynamically registering themselves, you can scale and manage service communication seamlessly.

Using **Feign Client** on top of **Eureka further** abstracts away HTTP calls, making **inter-service communication declarative** and easy to manage.

Thank You

Rohan Thapa
thaparohan2019@gmail.com