**Eureka Server and API Gateway and Service Registry:**

**monolithic vs microservice**



monolithic vs microservice

When comparing monolithic and microservice architectures, here's how they stack up:

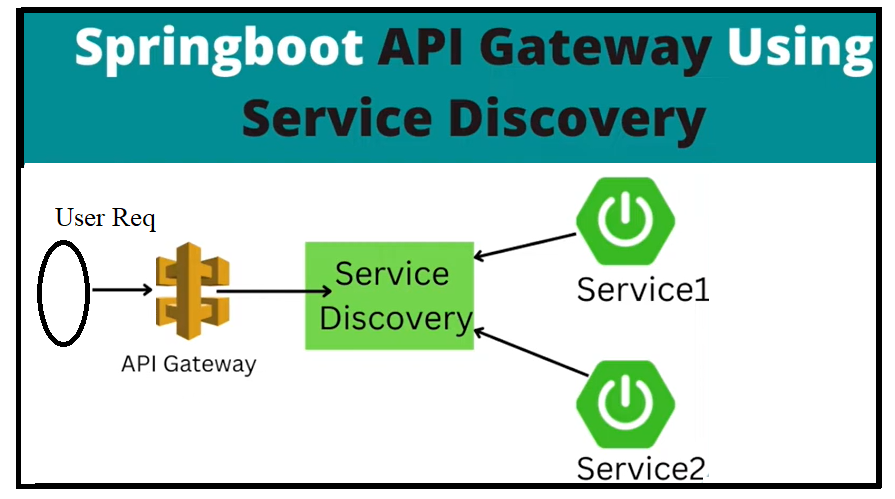
### Monolithic Architecture

1. **Single Codebase**: All components are integrated into one large codebase.
2. **Tightly Coupled**: Components are interdependent.
3. **Easy Deployment**: Single unit deployment makes it simpler to manage initially.
4. **Scalability Limitations**: Harder to scale specific parts.
5. **Simple Testing**: Testing is straightforward as it's one unit.
6. **Slower Development**: Large, complex codebase can slow down development.

### Microservice Architecture

1. **Multiple Services**: Each component is a separate service.
2. **Loosely Coupled**: Services operate independently.
3. **Independent Deployment**: Services can be deployed and updated independently.
4. **Scalability**: Easy to scale individual services.
5. **Complex Testing**: Requires testing of interactions between services.
6. **Faster Development**: Teams can work on different services simultaneously.

**API gateway work flow:**



Here we are going to discussion about spring boot api gateway and service discovery :

So as per above diagram to implements api gatway and service discover we need to build two service and one service discover and one api gateway.

In this diagram first part will execute by Api Gateway any request which is raised by user it will reach to api gateway then it will reach to service discover then it will reach to corresponding specific service

Here in microservice architecture API gate way work like Front controller like dispatch servlet in spring boot

**API Gateway**

An **API Gateway** is a server that acts as an API front-end, routing requests from clients to microservices. It handles various responsibilities such as request routing, composition, and protocol translation.

**Key Features of an API Gateway:**

1. **Request Routing**: Directs incoming client requests to the appropriate service.
2. **Load Balancing**: Distributes client requests across multiple instances of a service.
3. **Security**: Manages authentication, authorization, and rate limiting.
4. **Protocol Translation**: Converts between different protocols (e.g., HTTP to WebSockets).
5. **Response Caching**: Stores responses to improve performance for subsequent requests.
6. **Request Aggregation**: Combines multiple client requests into a single request to a backend service.

**Examples of API Gateways:**

* **Zuul**: A gateway service developed by Netflix.
* **Spring Cloud Gateway**: A gateway solution in the Spring Cloud ecosystem.
* **Kong**: An open-source API gateway.
* **Apigee**: Google's API management platform.

### Service Discovery

**Service Discovery** is the process by which microservices automatically detect and communicate with each other. It helps manage service instances' dynamically changing IP addresses and locations.

#### Key Components of Service Discovery:

1. **Service Registry**: A database where services register themselves with their network locations.
2. **Service Discovery Clients**: Microservices that query the service registry to find other services.
3. **Health Check Mechanisms**: Periodically verify the health of services registered in the service registry.

#### Types of Service Discovery:

1. **Client-Side Discovery**:
   * Each client is responsible for determining the network locations of available service instances.
   * Examples: Netflix Eureka, Consul.
2. **Server-Side Discovery**:
   * Clients make requests to a load balancer, which queries the service registry and routes the request to an appropriate service instance.
   * Examples: AWS Elastic Load Balancer (ELB), Kubernetes.

### Combining API Gateway and Service Discovery

An API Gateway can leverage service discovery to dynamically route requests to the appropriate service instances. For example, Spring Cloud Gateway can integrate with Netflix Eureka for service discovery.

#### Example:

1. **API Gateway**: Spring Cloud Gateway. (Front controller)
2. **Service Discovery**: Netflix Eureka. (load balancing and IP discovery) these two we will do now.

### Spring Cloud Configuration Example:

To achieve this implementation we need two dependencies   
  
<dependencies>

<!-- Spring Cloud Gateway -->

<dependency>

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

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

</dependency>

<!-- Netflix Eureka -->

<dependency>

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

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

</dependency>

</dependencies>

**Step1 : Create Service Discovery**

**Step2: Create min Two services for Api Gateway integrations**

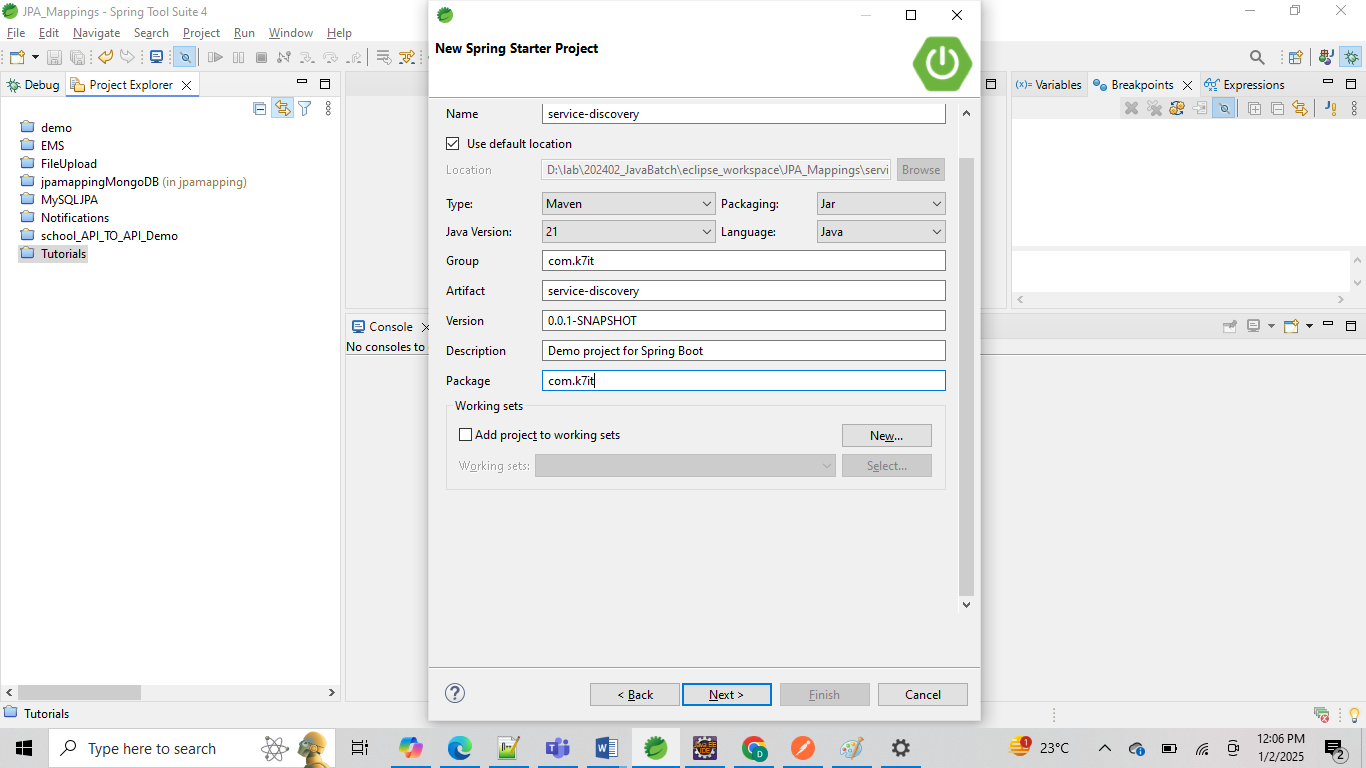
**Step:3 Register Two services with Service Discovery Eureka server**

**Step4: Create Api Gateway**

**Step5: Register all 3 Services i.e Api Gateway and other two services with our Service Discovery**

Step1: Lets start with Service Discovery

Create Spring Boot project with following dependencies



And Add Eureka server dependency only one dependency is enough and

<dependency>

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

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

</dependency>

But some time we may receive this error while starting server but that’s not a issue for our server ,

o.s.v.b.OptionalValidatorFactoryBean : Failed to set up a Bean Validation provider: jakarta.validation.NoProviderFoundException: Unable to create a Configuration, because no Jakarta Bean Validation provider could be found. Add a provider like Hibernate Validator (RI) to your classpath.

in case if you don’t want to see this error , then add below dependency in pom.xml along with Eureka server dependency.

<!-- Hibernate Validator -->

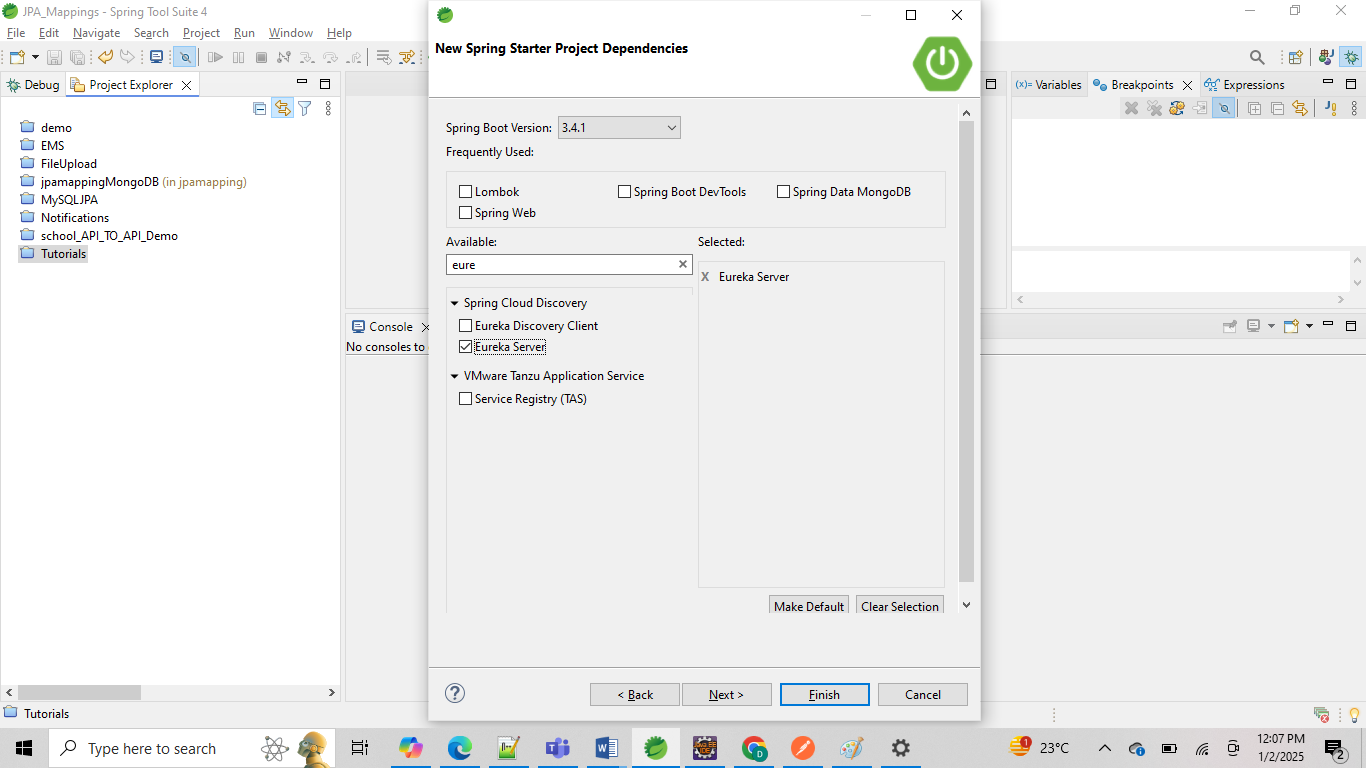
<dependency>

<groupId>org.hibernate.validator</groupId>

<artifactId>hibernate-validator</artifactId>

<version>8.0.2.Final</version>

</dependency>



**Add these properties in Application.Properties file  
=======================================**

spring.application.name=eureka-namming-service

server.port=8761

# Disable Eureka client (since this is a server)

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

# Disable the import check for Spring Cloud Config

spring.cloud.config.import-check.enabled=false

can you explaine more about these properties server.port=8761 # Disable Eureka client (since this is a server) eureka.client.register-with-eureka=false eureka.client.fetch-registry=false # Disable the import check for Spring Cloud Config spring.cloud.config.import-check.enabled=false

Sure! Let me explain these properties in detail.

**1. server.port=8761**

This property sets the port number on which your Spring Boot application will run. In this case, it is set to 8761.

* **Usage**: server.port=8761
  + **Example**: The Eureka Server will be accessible via http://localhost:8761.

**2. eureka.client.register-with-eureka=false**

This property is used to disable the registration of the Eureka server itself as a client. Since the Eureka server acts as a registry, it doesn’t need to register with itself.

* **Usage**: eureka.client.register-with-eureka=false
  + **Purpose**: To prevent the Eureka server from trying to register itself with another Eureka server.

**3. eureka.client.fetch-registry=false**

This property disables the fetching of the registry from another Eureka server. Since this application is a Eureka server, it doesn’t need to fetch registry information from another server.

* **Usage**: eureka.client.fetch-registry=false
  + **Purpose**: To prevent the Eureka server from trying to fetch the registry of services from another Eureka server.

**4. spring.cloud.config.import-check.enabled=false**

This property disables the import check for Spring Cloud Config.

* **Usage**: spring.cloud.config.import-check.enabled=false
  + **Purpose**: When set to false, it allows the application to start even if some optional configurations imported from the Spring Cloud Config server are missing or not accessible. This is particularly useful in development or testing environments where not all configurations may be available.

**Example Scenario:**

Imagine you are setting up a microservices architecture with Eureka for service discovery and Spring Cloud Config for externalized configuration. Your Eureka server will run on port 8761 and will act as a registry where other services register themselves. The Eureka server itself does not need to register or fetch the registry from another server, hence the properties eureka.client.register-with-eureka=false and eureka.client.fetch-registry=false. Additionally, you want to ensure that your application can start even if some optional configurations from the Spring Cloud Config server are not available, so you set spring.cloud.config.import-check.enabled=false.

Then Enable EurekaServer At SpringBootApplication class

**package** com.k7it;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

**public** **class** ServiceDiscoveryApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ServiceDiscoveryApplication.**class**, args);

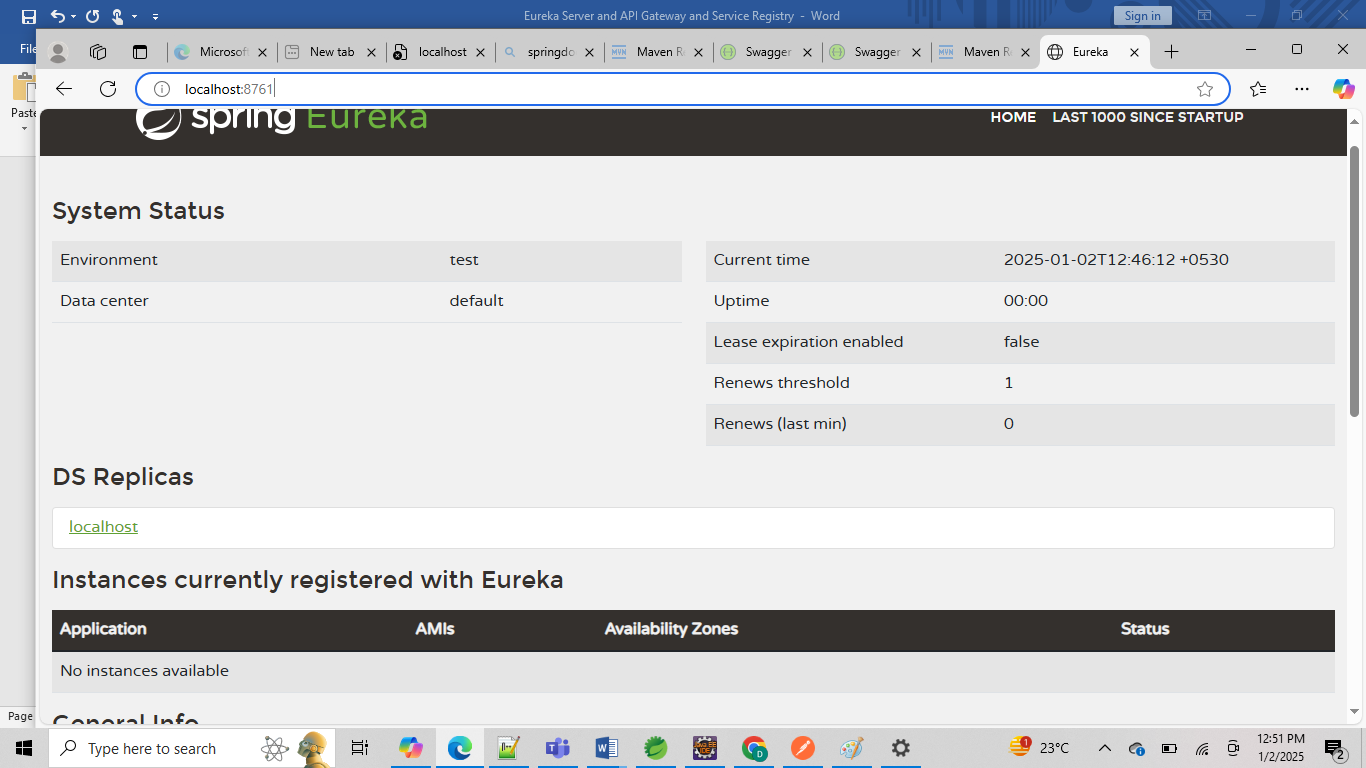
}

}

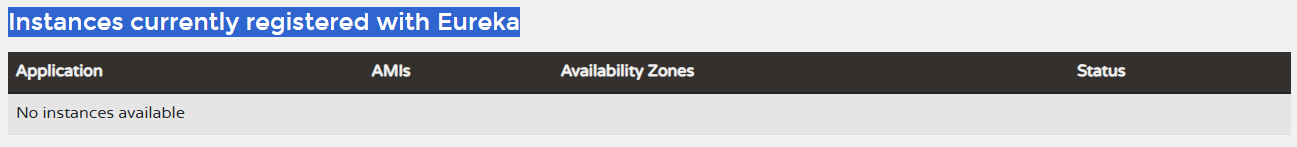
**Then next start our Eureka server service discovery application.**

**Note: We need to start Eureka server application or Service Discovery application should be first before starting any of our business Services or Api Gateway services.**

**Open** [**http://localhost:8761**](http://localhost:8761) **url in browser and see either you can able to see the Eureka server dashboard or not**



**If you can see Server Dash board like above your server is started successfully.**

**Here you can see this section**

**This is for our registry services will comes under this section once we start our api gateway and other services implementation and register with service discovery**

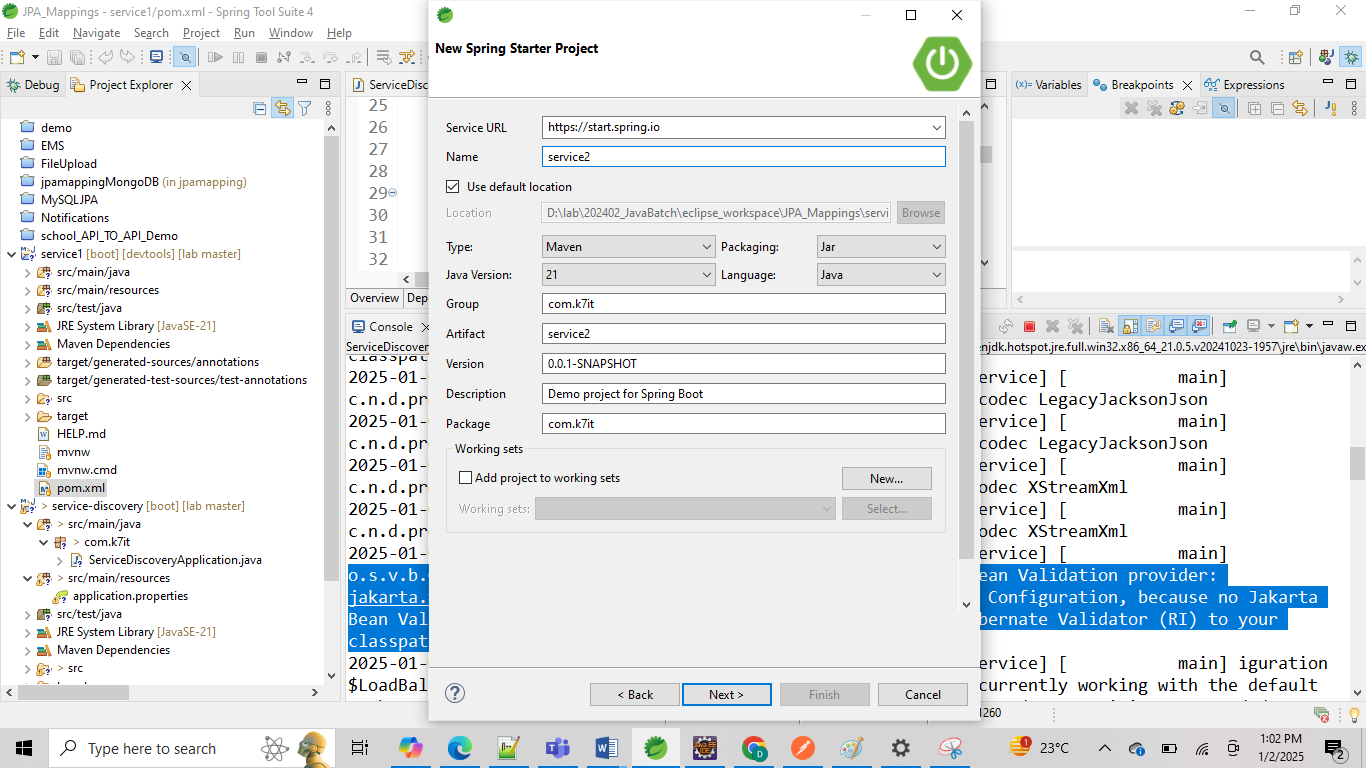
**Step2: Create Services**

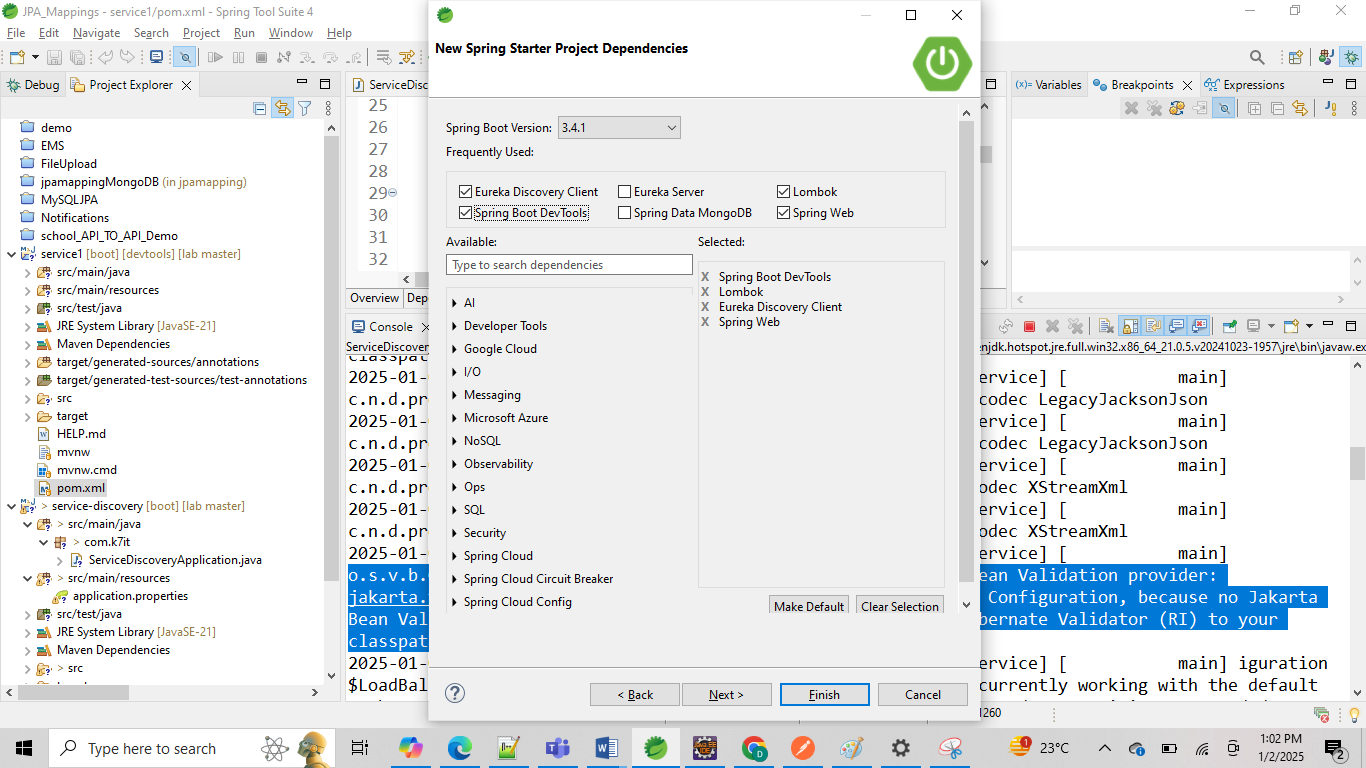
These Services are act like Eureka Clients service which we are going to register with Eureka server

For this we need Eureka client dependency along with our normal Spring boot dependencies.  
<dependency>

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

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





Then Lets create Just One HelloWorld Controller class and give one Get API for say HelloWorld output, but if we want we can add our database operations like CRUD operations for any of the Entity but as of now we are doing simple testing so we are going with HelloWorld.

**package** com.k7it.controller;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/")

**public** **class** HelloWorldController {

@GetMapping(name ="/sayHello", value ="/sayHello")

**public** String sayHello() {

**return** "HelloWorld from from Service1";

}

}

**Then Update Application.properties file: with specific port and service name**

spring.application.name=service1

server.port=9001

# Eureka server URL

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

Third property you can keep in application.properties file or application.yml file separately , better to keep

Eureka server client related info in separate yml file like bellow and remove from application.properties file

**Then add application.xml with Eureka Client registry info:**

Application.yml  
================

eureka:

client:

service-url:

defaultZone: http://localhost:8761/eureka/

Here   
**eureka.client.service-url.defaultZone**: Configures the Eureka server URL.

This application.yml format can be used in your Spring Boot Eureka client application to ensure it registers with the Eureka server at <http://localhost:8761/eureka/>

<http://localhost:8761/eureka/> : its is our eureka server dash board url

so we are registering ourselves with eureka server from our service1

**Then finally Enable our Service as Eureka Client application using   
@EnableDiscoveryClient**  
  
**package** com.k7it;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** Service1Application {

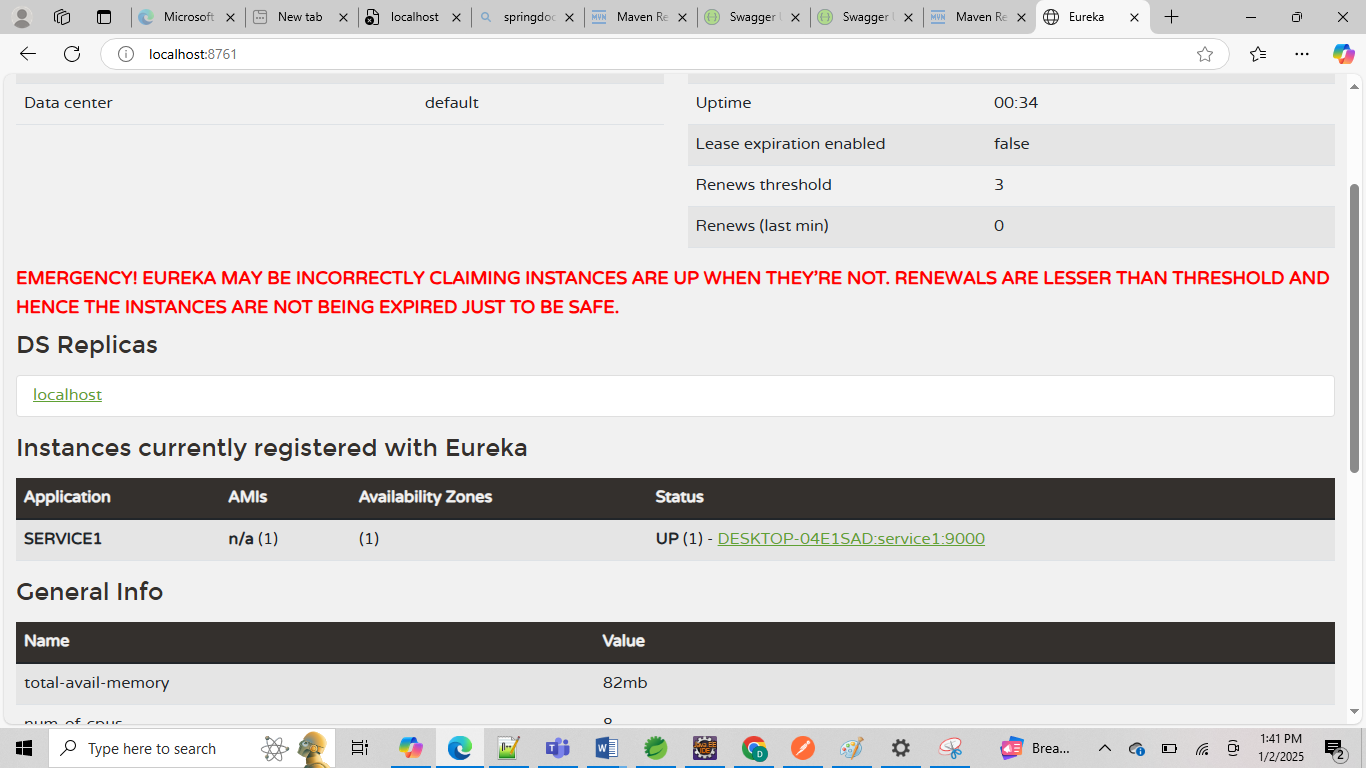
**public** **static** **void** main(String[] args) {

SpringApplication.*run*(Service1Application.**class**, args);

}

}

Then start Service1 and check either this service is registered with our Eureka server or not from Eureka dashboard url

  
Here we can see our service1 is registered with eureka server.

**Service2:**

Then Same Steps do it for Service2 and start service2 with another port number with   
servcer.port= 9001

And register that service with Eureka server dashboard

From .yml file same as service1 and start the application

**Applciation class:**

**package** com.k7it;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** Service2Application {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(Service2Application.**class**, args);

}

}

**Controller class:**

**package** com.k7it.controller;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/")

**public** **class** HelloWorldController {

@GetMapping(name ="/sayHello", value ="/sayHello")

**public** String sayHello() {

**return** "HelloWorld from from Service2";

}

}

**Application.properties:**

spring.application.name=service2

server.port=9002

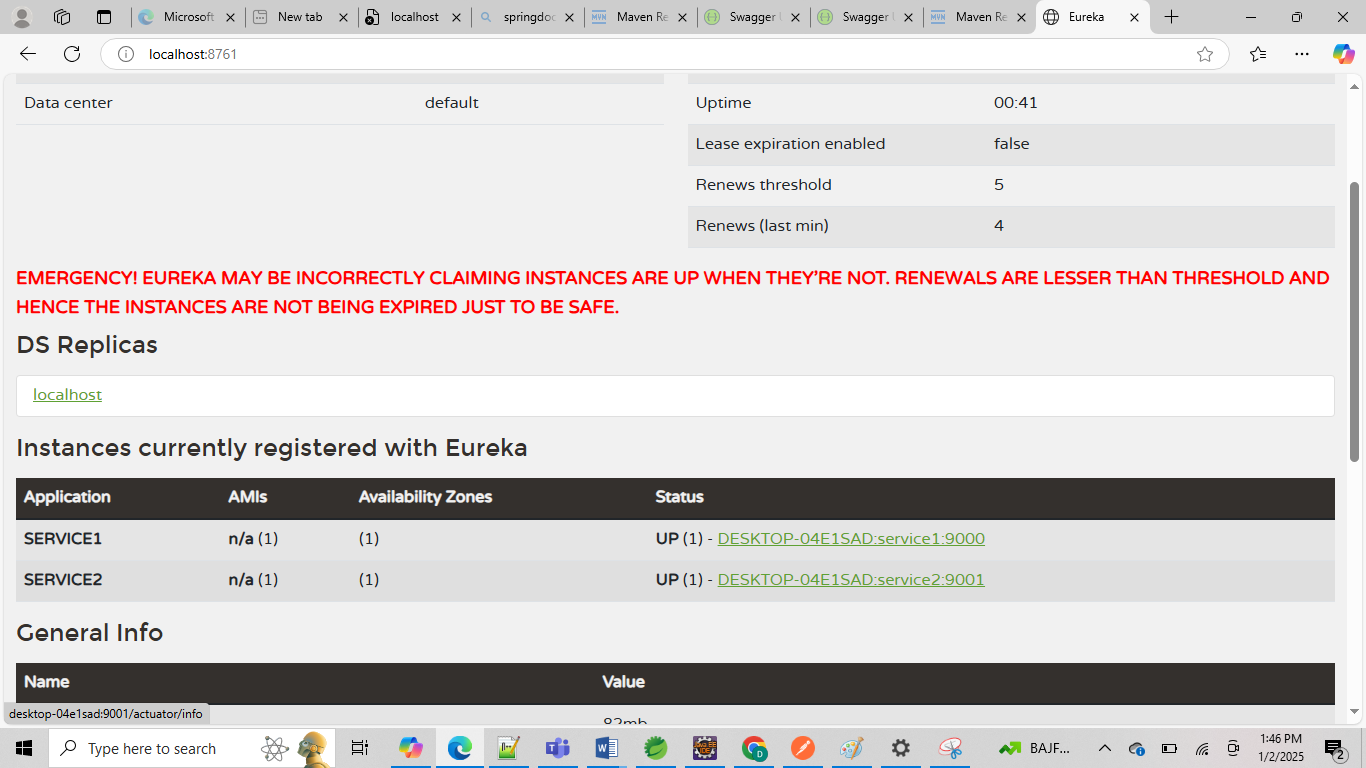
**application.yml:**

eureka:

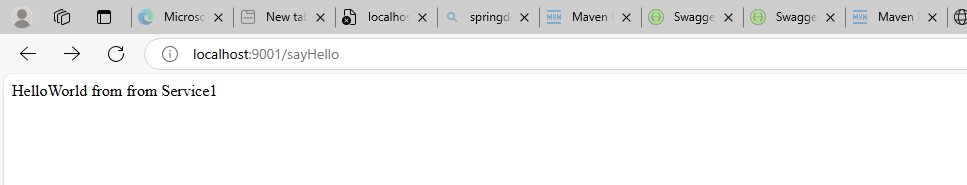
client:

service-url:

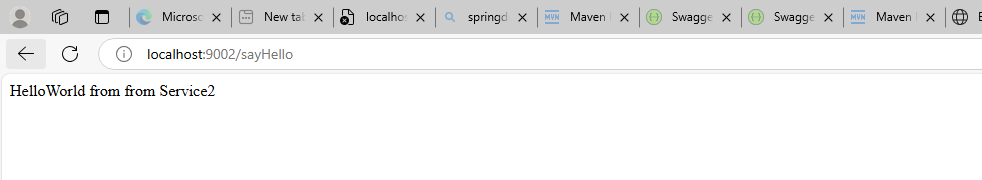
defaultZone: http://localhost:8761/eureka/



Now we can test our services using our service ip and port numbers from browser or post man

[localhost:9001/sayHello](http://localhost:9001/sayHello)   
  


[localhost:9002/sayHello](http://localhost:9001/sayHello)



So here two servers are running in two different port numbers right , but we can access with help of Eureka server and API gateway we can access with single port number for the services i.e service1 and service2.

**Step4: Create Api Gateway**

Lets create new Springboot project for API Gateway with min two dependencies   
1. Eureka Discovery Client   
2. Gateway spring cloud routing

<properties>

<java.version>21</java.version>

<spring-cloud.version>2024.0.0</spring-cloud.version>

</properties>

<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>

<dependencies>

<!--<dependency>

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

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

</dependency>-->

<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-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

Note: if we add api gateway dependency from spring starter it will add spring-cloud-starter-gateway-mvc type, but we need only spring-cloud-starter-gateway

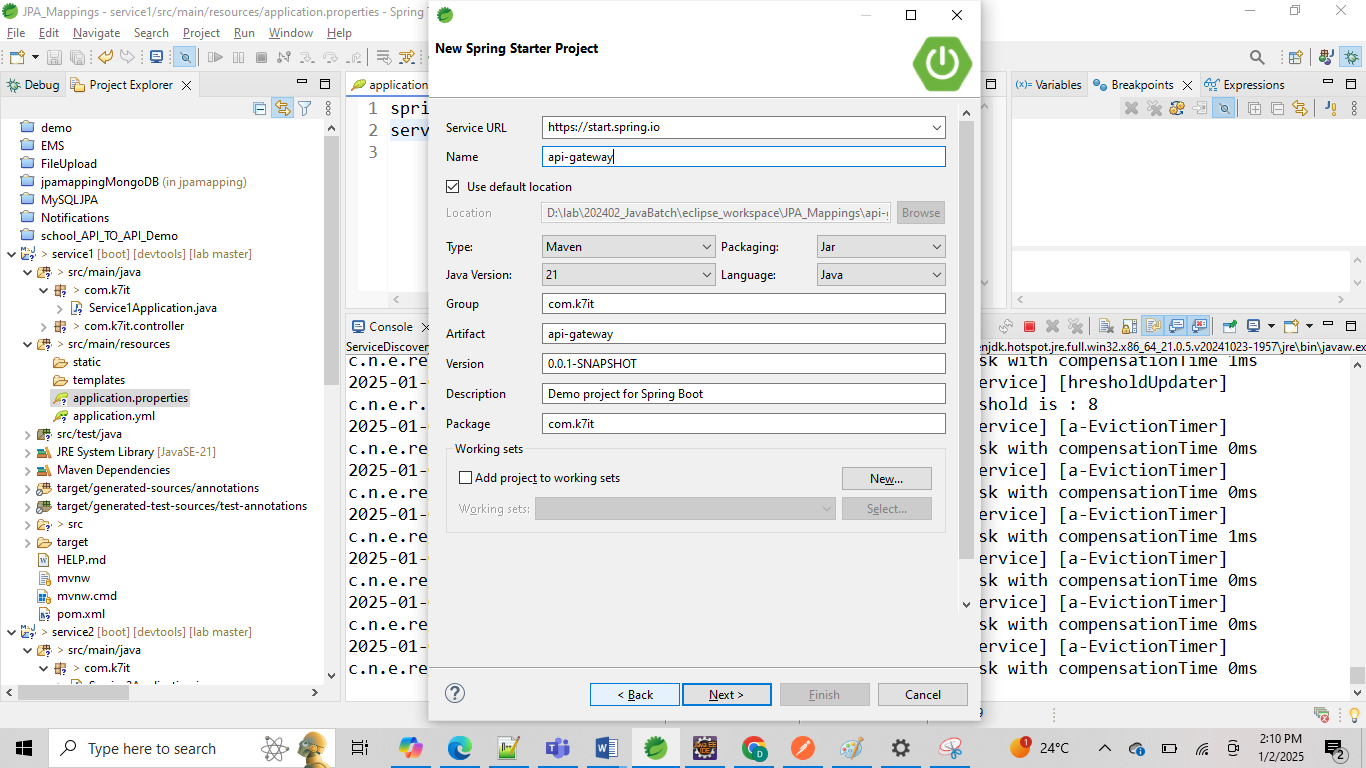
So we need to update pom.xml with this api gateway dependency instead of gateway-mvc

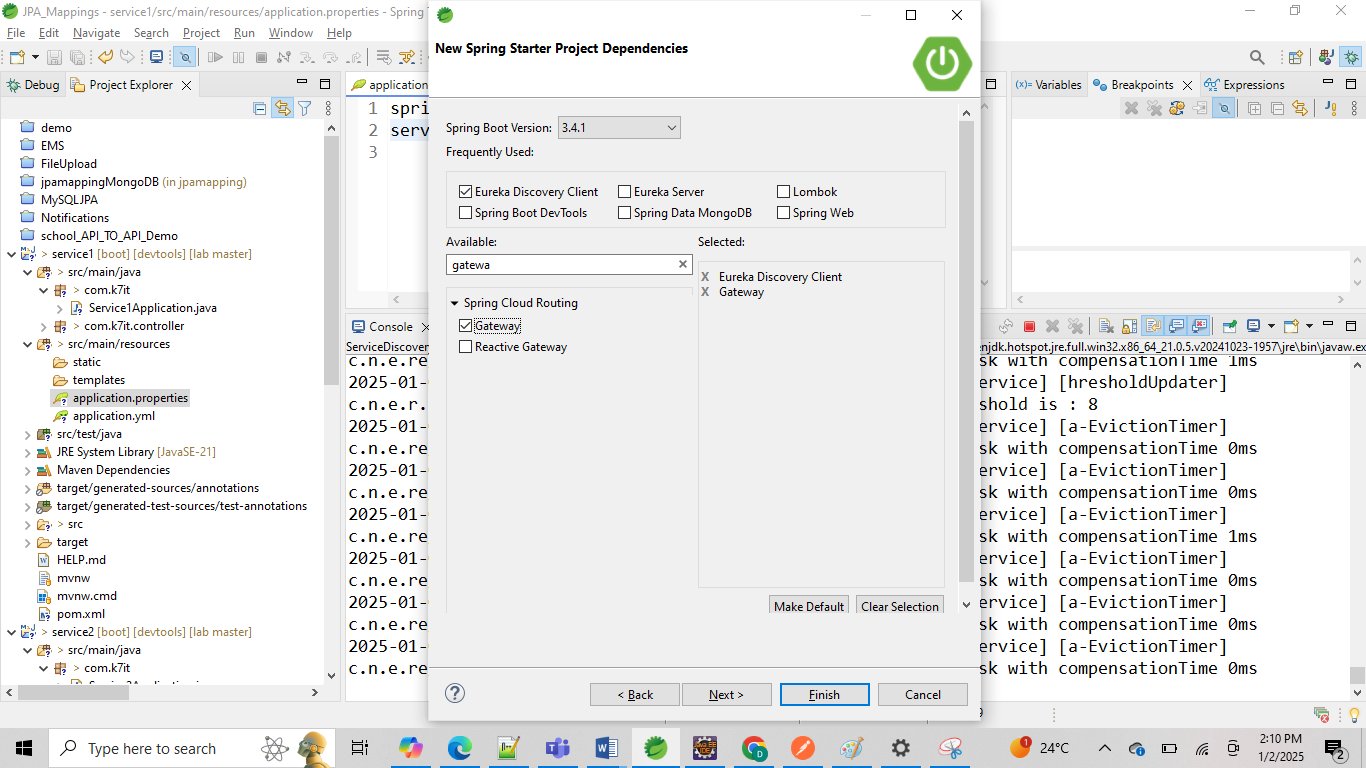
<dependency>

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

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

</dependency>





Next Ennable Eureka Cleint serice from our Api Gateway with Eureka server as same as other services using   
@EnableDiscoveryClient

**package** com.k7it;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** ApiGatewayApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ApiGatewayApplication.**class**, args);

}

}

Then Register Our Api Gateway with Eureka server and Register our services with API Gateway routers to use single port using application.yml file

**Application.properties**  
  
server.port=8080

**Application.yml file:**

spring:

application:

name: api-gateway

cloud:

gateway:

routes:

- id: service1

uri: lb://service1

predicates:

- Path=/service1/\*\*

filters:

- StripPrefix=1

- id: service2

uri: lb://service2

predicates:

- Path=/service2/\*\*

filters:

- StripPrefix=1

eureka:

client:

service-url:

defaultZone: http://localhost:8761/eureka/

server:

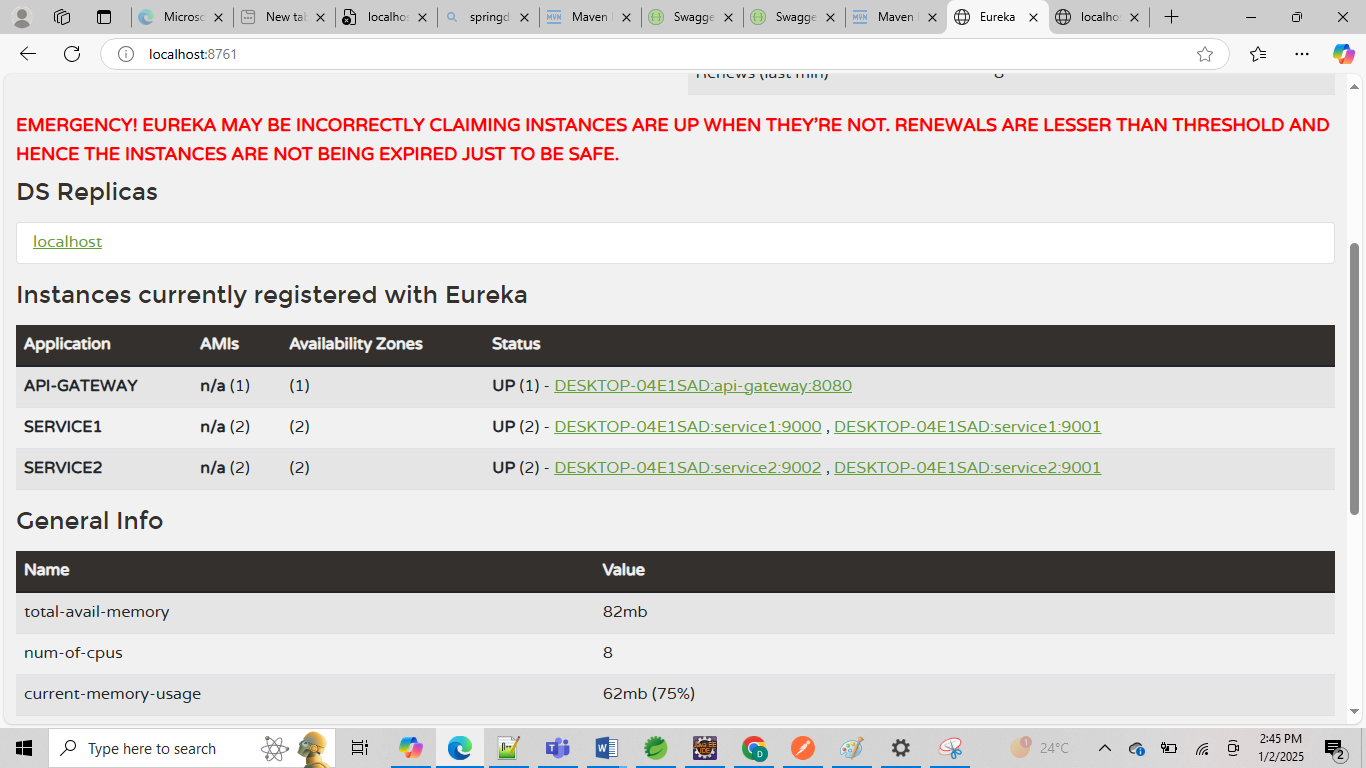
port: 8080

logging:

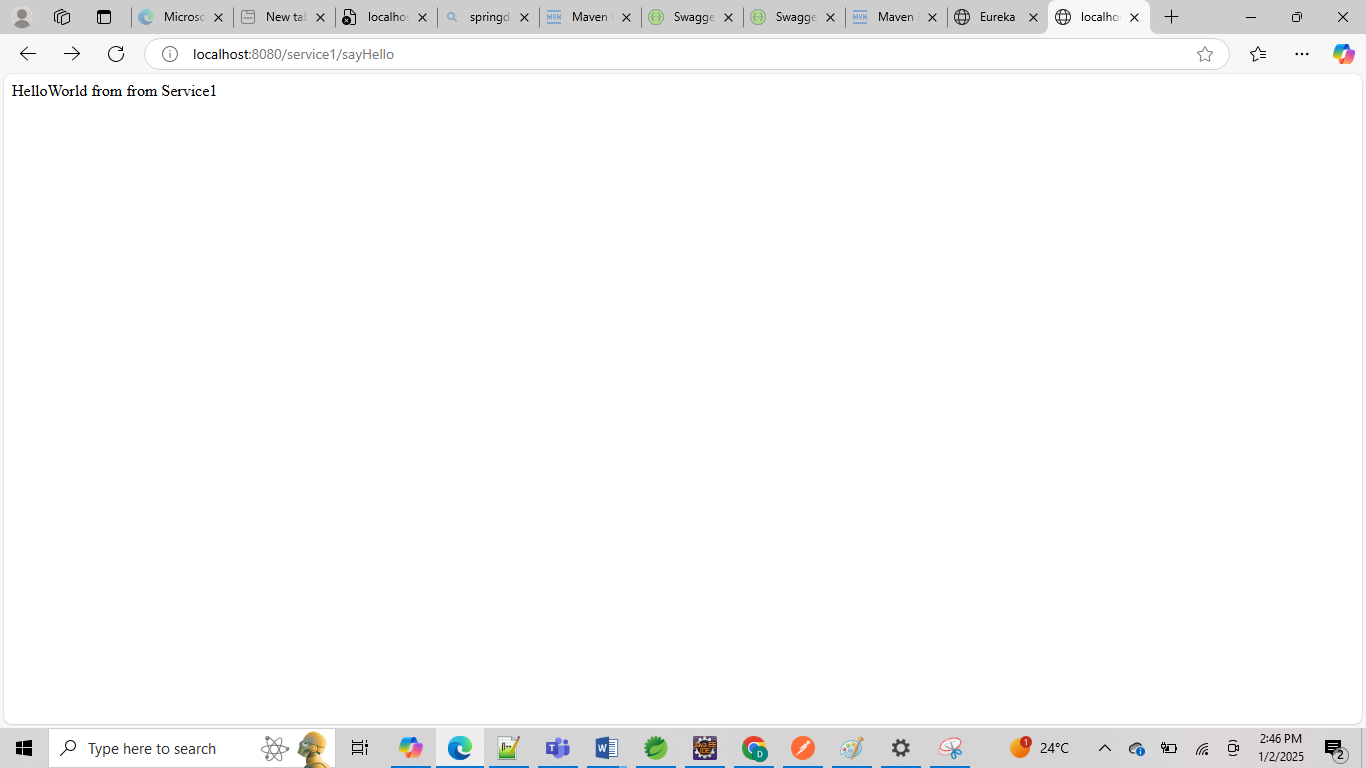
level:

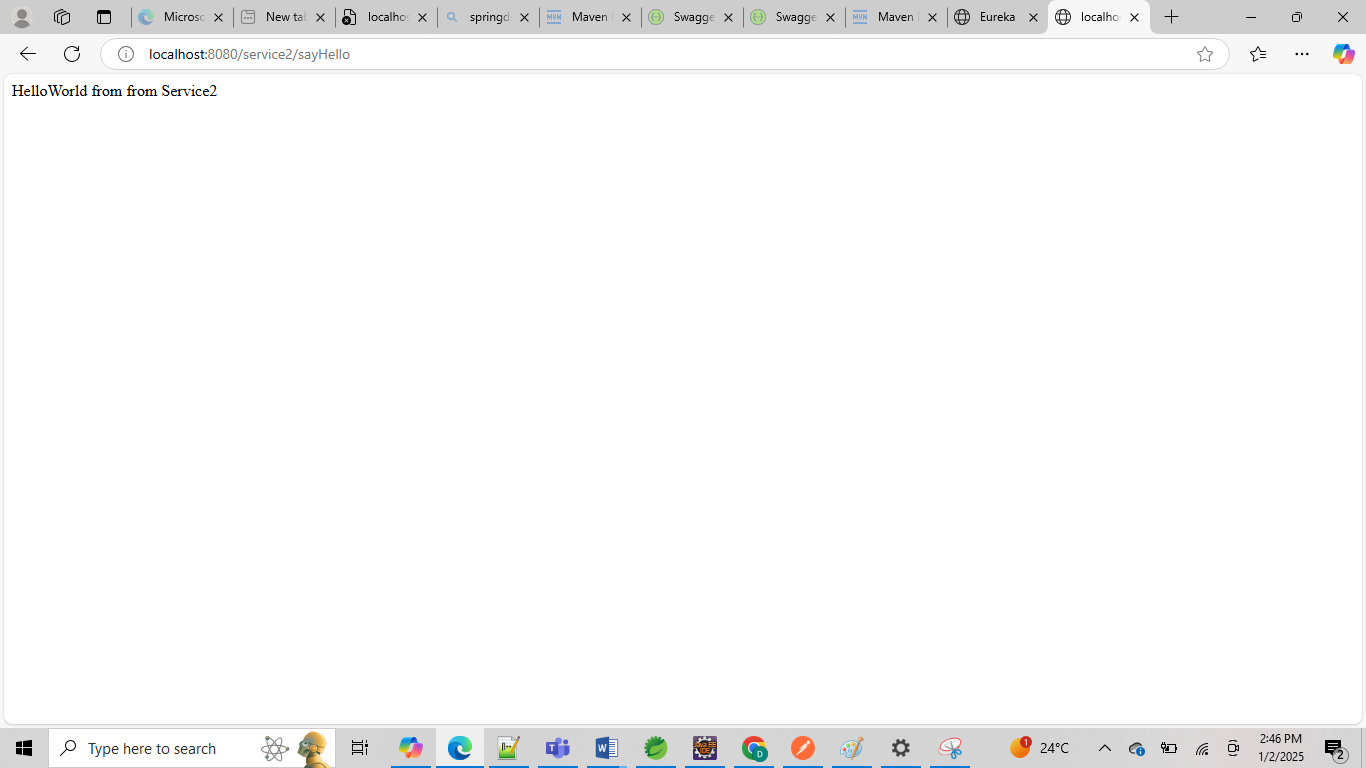
org.springframework.cloud.gateway: DEBUG

Then Run the application and check with this Api Gateway is registered with eureka server or not   
if its registered successfully then our Services also will registered with API Gateway.



Then test our service from ApiGate way port number i.e 8080 instead of 9001 and 9002





Here   
[localhost:8080/service1/sayHello](http://localhost:8080/service2/sayHello)  
[localhost:8080/service2/sayHello](http://localhost:8080/service2/sayHello)

Here service1 and service2 are names which we given at yml file for service registry with router names.