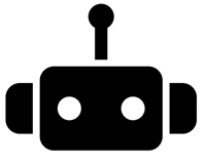

PUCP

SESIÓN DE LABORATORIO 2
Microservicios Parte 1



HORARIO 10M1

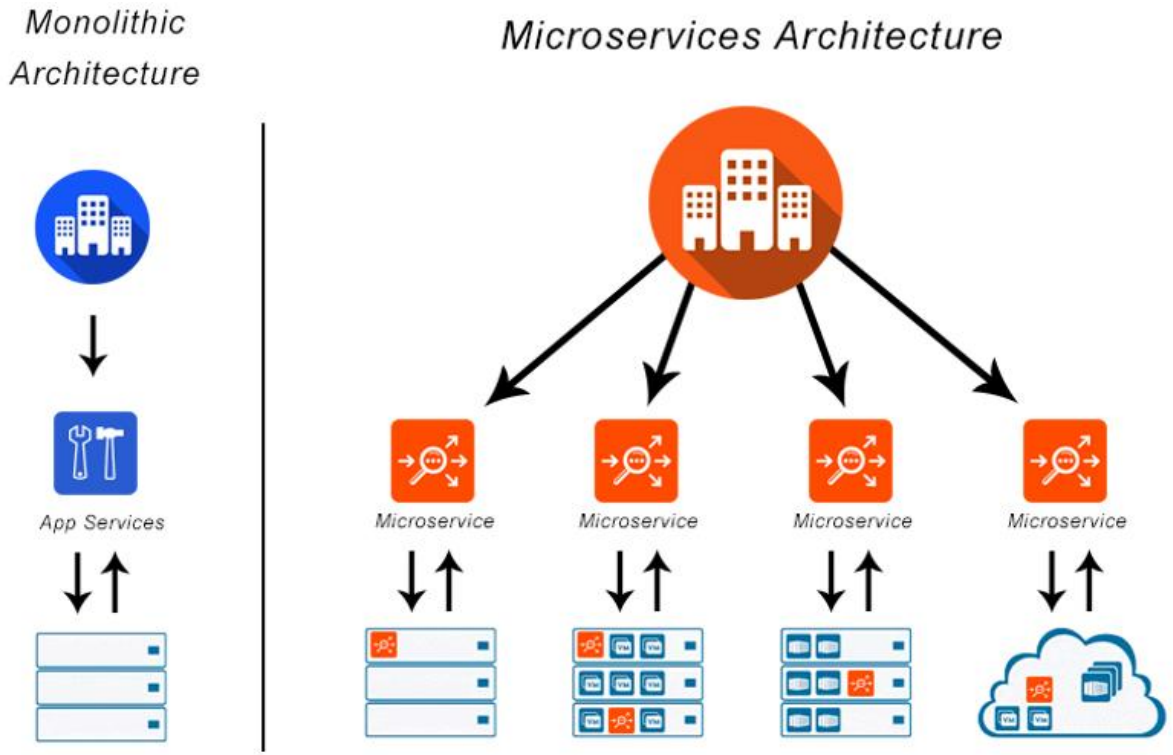
Empezaremos a las 7:10 p.m

Gracias!

PUCP

¿Por qué una arquitectura debe utilizar microservicios?

Arquitectura de Microservicios

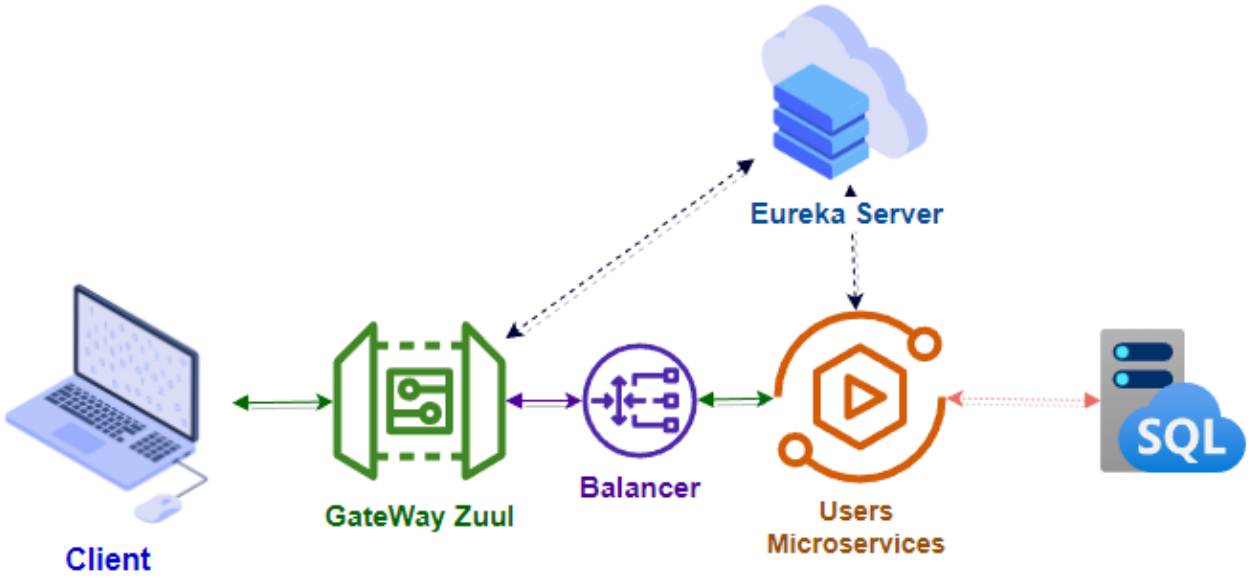


Ejercicio

API Gateway

Spring Cloud

Allows a single and centralized access to our microservices



API Gateway

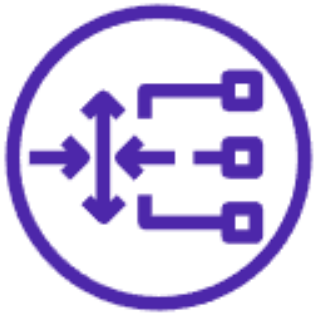
Un **API Gateway** es un componente clave en la arquitectura de microservicios, que actúa como un punto de entrada único para las solicitudes de los clientes hacia el sistema. Su principal función es encapsular la complejidad de la arquitectura interna, proporcionando una interfaz única y simplificada para los consumidores de la API.

Gateway Zuul

Un **API Gateway**, como **Zuul**, es una aplicación que actúa como puerta de entrada única para gestionar todas las solicitudes dirigidas a un sistema de microservicios. Zuul, desarrollado inicialmente por Netflix, no solo maneja las solicitudes entrantes, sino que también realiza el enrutamiento dinámico hacia las aplicaciones de microservicios correspondientes, lo que permite una comunicación eficiente entre los clientes y los servicios internos.

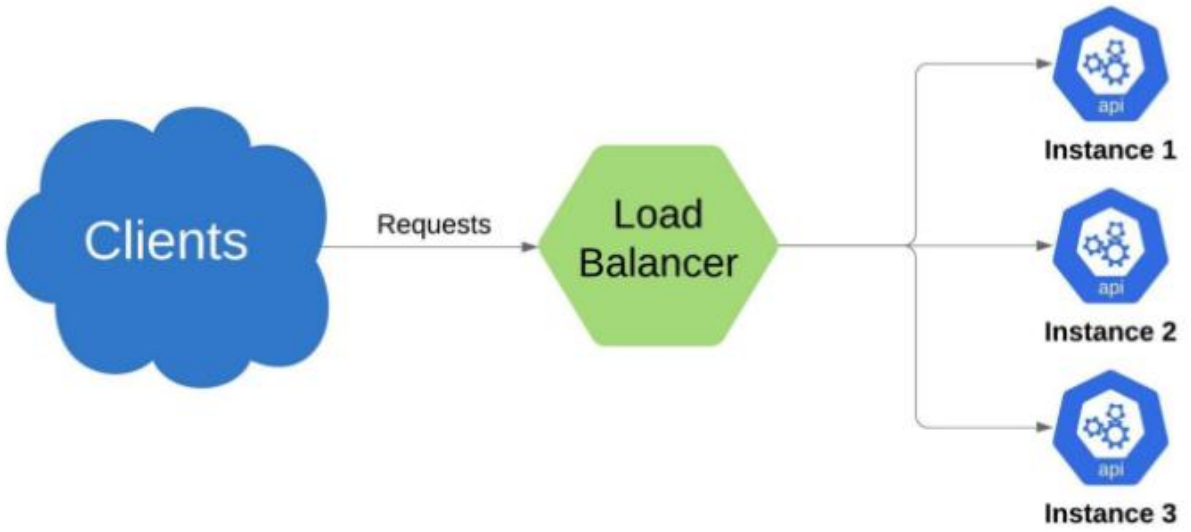
Balancer

El balanceo de carga es el proceso de distribuir el tráfico de red entrante entre múltiples servidores para optimizar el uso de recursos, garantizar la disponibilidad y mejorar la capacidad de respuesta del sistema. Esta distribución puede realizarse de manera uniforme o basada en reglas específicas, como el peso del servidor, la latencia o la carga actual.



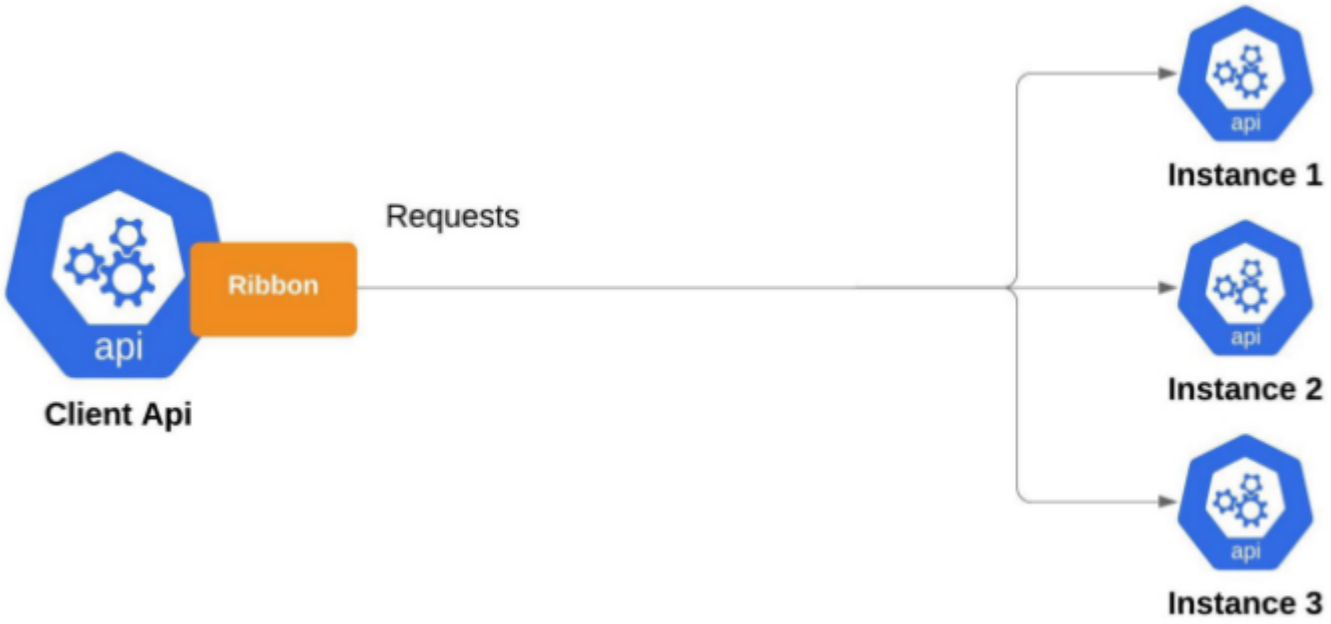
Server Side Load balancer

En este enfoque, un componente central es responsable de distribuir las solicitudes a los servidores disponibles. El cliente solo conoce la dirección del balanceador de carga, y este se encarga de redirigir las solicitudes al servicio más adecuado.



Client Side Load Balancing

En este caso, el cliente realiza el balanceo de carga directamente. Para ello, el cliente mantiene una lista de instancias del servicio y utiliza algoritmos específicos para seleccionar a cuál enviar la solicitud.



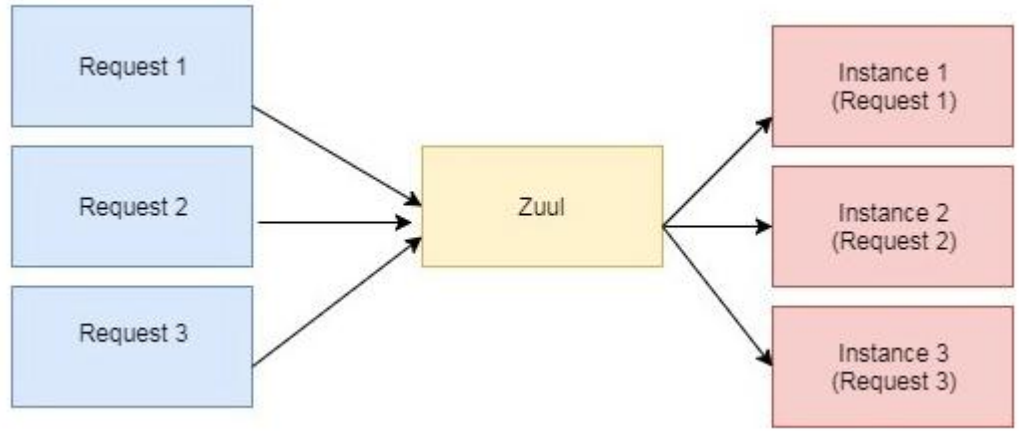
Zuul Load Balancer

Cuando Zuul recibe una solicitud, este realiza los siguientes pasos:

Determinación de la instancia del servicio: Zuul selecciona automáticamente una de las ubicaciones físicas disponibles para el servicio solicitado.

Reenvío de la solicitud: La solicitud se dirige a la instancia de servicio seleccionada, gestionando de forma transparente el proceso de enrutamiento.

Este flujo funciona de manera predeterminada, sin necesidad de configuraciones adicionales, gracias a su integración con las bibliotecas de Netflix.



Eureka Server

Eureka Server es una aplicación que contiene información sobre todas las aplicaciones cliente-servicio. Cada Microservicio se registrará en el servidor Eureka y el servidor Eureka conoce todas las aplicaciones cliente que se ejecutan en cada puerto y dirección IP. Eureka Server viene con el paquete de Spring Cloud. Para ello, tenemos que desarrollar el servidor Eureka y ejecutarlo en el puerto por defecto 8761.

Euerka Server
Spring **Cloud**

Eureka Server Notas importantes

Es un servidor de registros de nombres

Es como un «contenedor» para microservicios

Requiere un identificador para cada servicio

Cada microservicio debe ser un cliente para el autodescubrimiento

Registra los metadatos de los microservicios

Permite el desacoplamiento de IP y Puerto.

Permite la comunicación por nombre e identificador

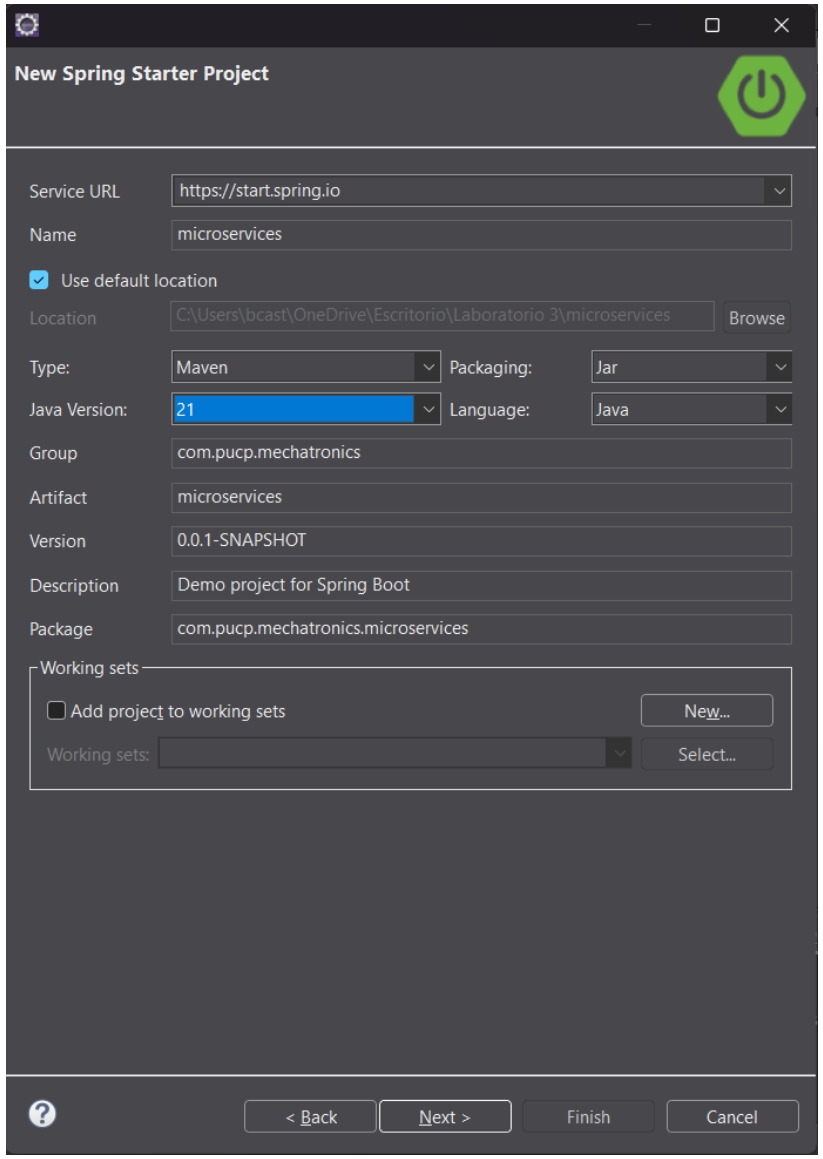
Euerka Server
Spring **Cloud**

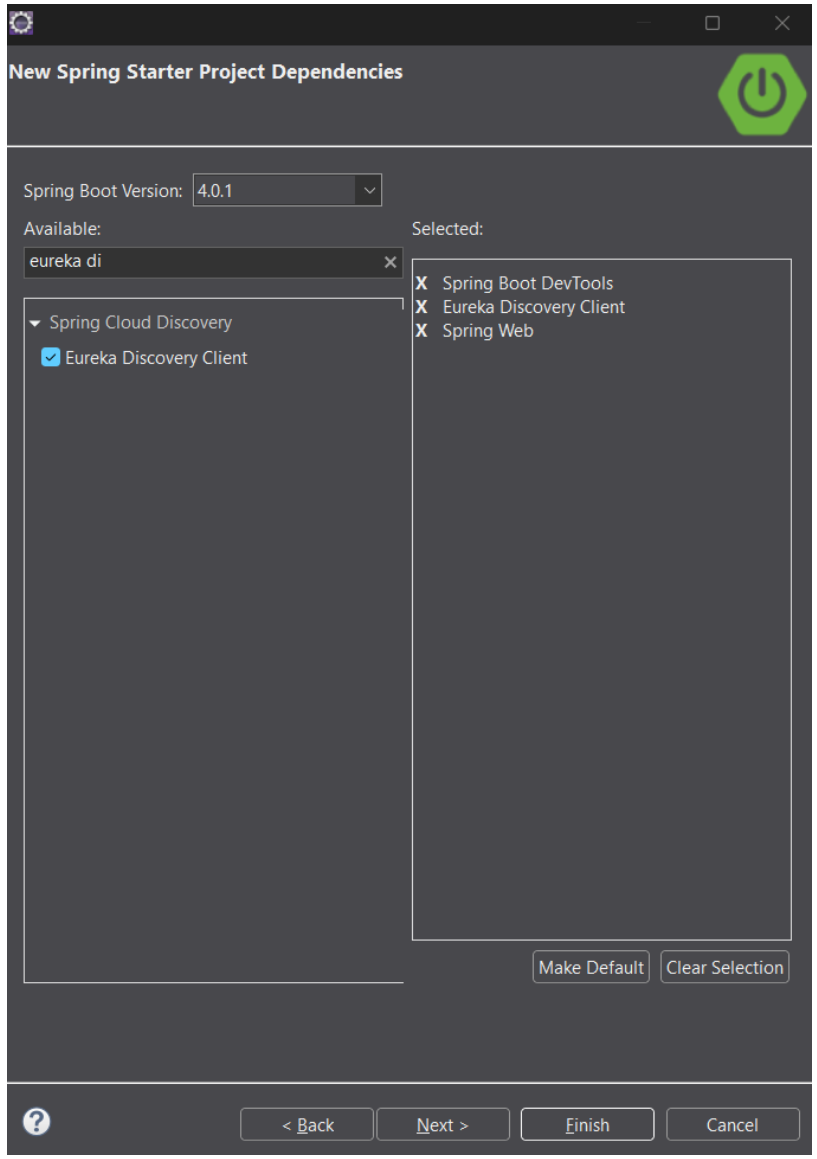
Consideraciones

Vamos a continuar usando Spring Tools 4 y Java 21

Desarrollo – Creando el microservicio

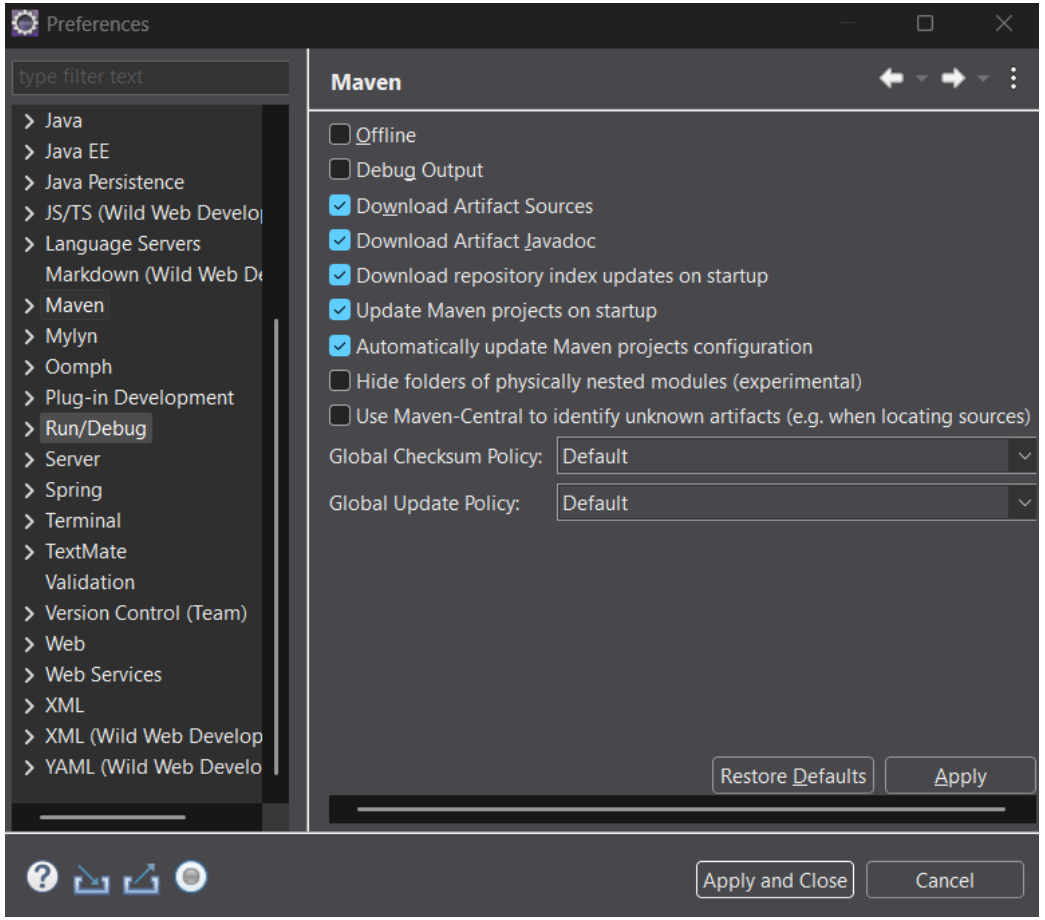
Aquí utilizaremos nuevamente el Spring Starter Project y además para las dependencias utilizaremos Spring Boot DevTools, Spring Web y Eureka Discovery client





Asegurando que todo este activado para las descargas automaticas

Vamos a ir a Window luego a preferences y seleccionamos lo que se muestra a continuación en Maven

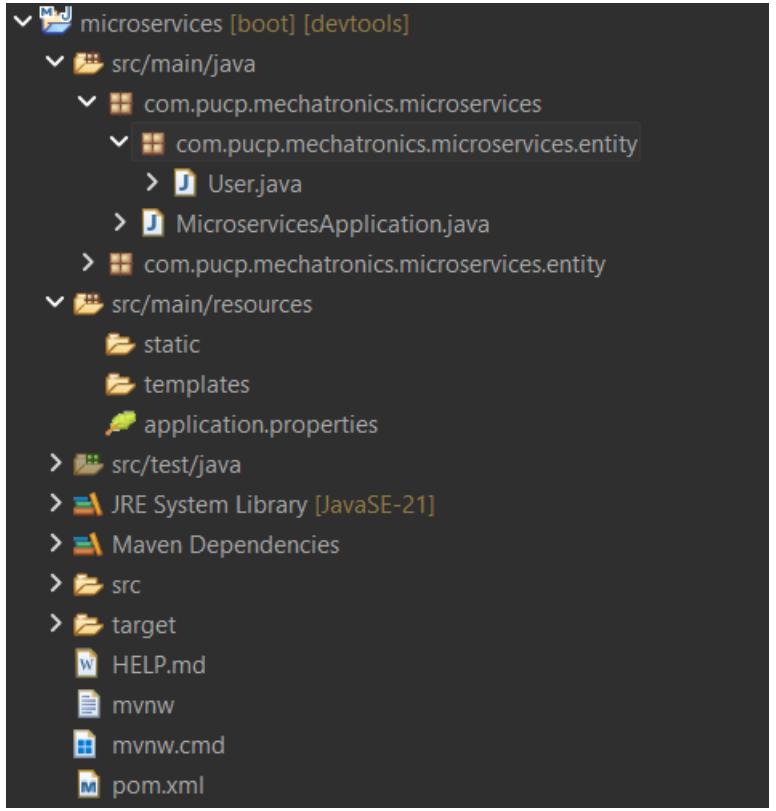


La Capa entidad

También conocida como **capa de modelo** o **modelo de dominio**, se encarga de representar los objetos y datos que reflejan las entidades del negocio. Estas clases suelen estar directamente relacionadas con las tablas de una base de datos en sistemas relacionales, o con documentos en bases de datos NoSQL.

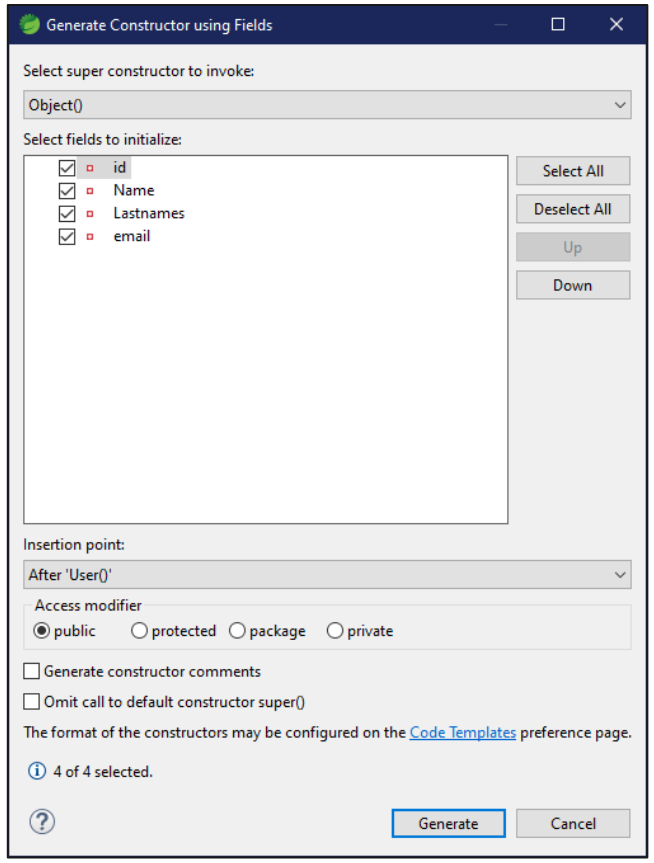
Implementando la capa de entidad

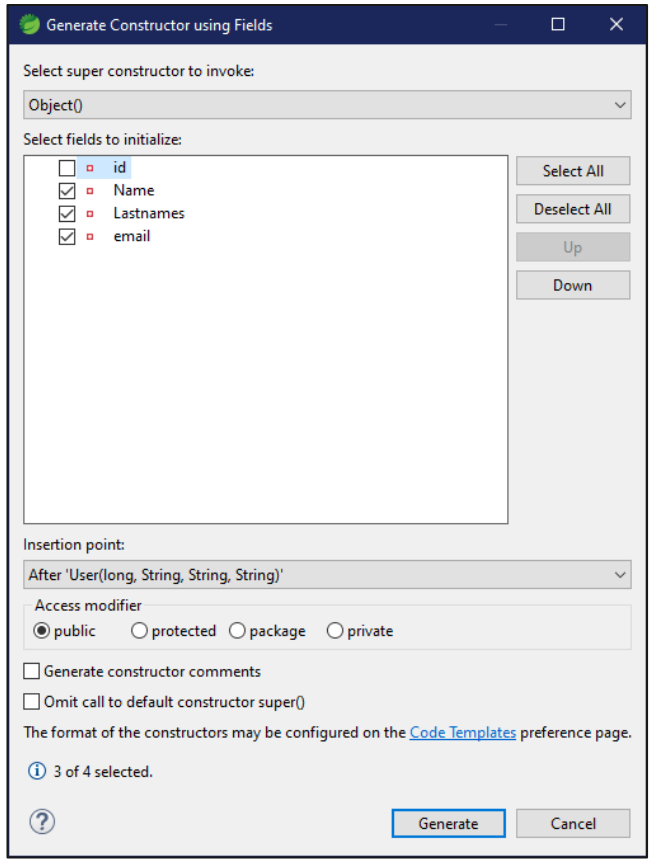
En este caso implementaremos el paquete com.pucp.mechatronics.microservices.entity en conjunto con la clase User

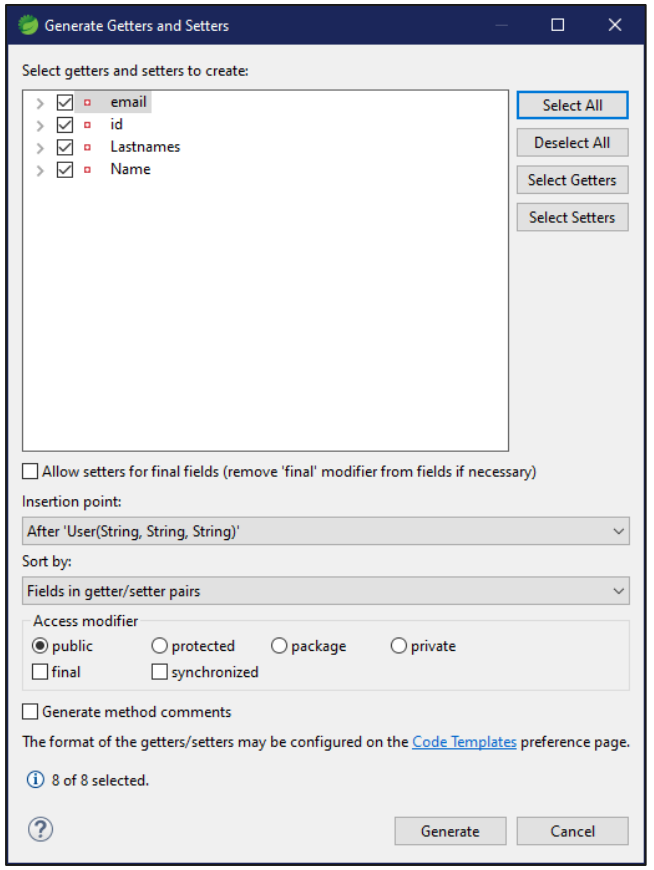



```
package com.pucp.mechatronics.microservices.entity;
public class User {
  private long id;
  private String Name;
  private String Lastname;
  private String Email;
}
```

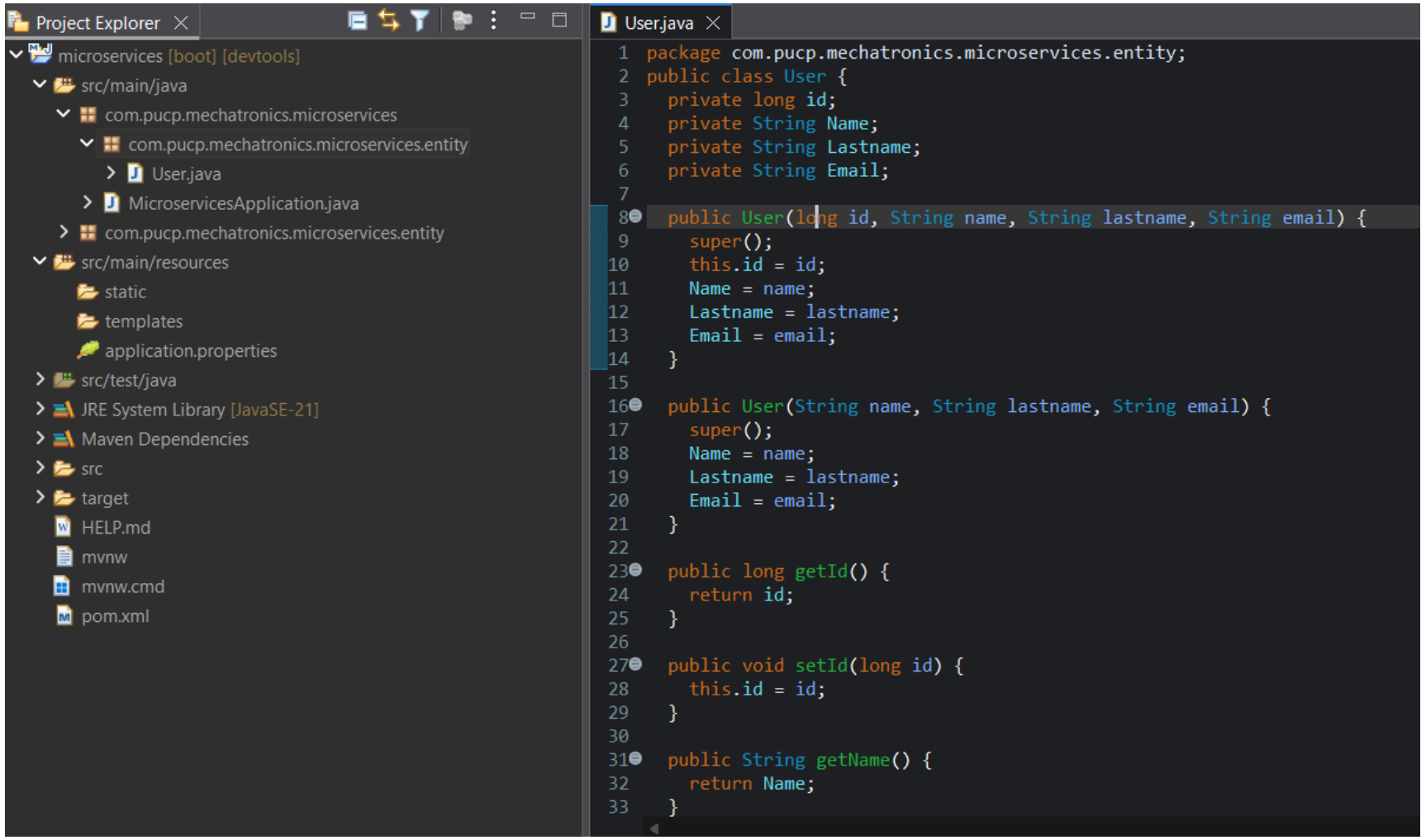
Clase User – Constructos Setters and Getters







Clase User

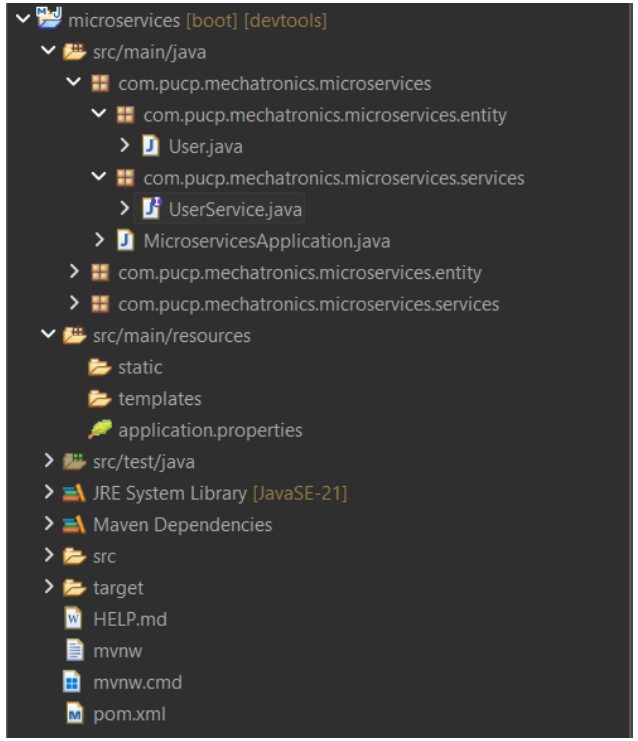

Project Explorer
microservices [boot] [devtools]
src/main/java
com.pucp.mechatronics.microservices
com.pucp.mechatronics.microservices.entity
User.java
MicroservicesApplication.java
com.pucp.mechatronics.microservices.entity
src/main/resources
static
templates
application.properties
src/test/java
JRE System Library [JavaSE-21]
Maven Dependencies
src
target
HELP.md
mvnw
mvnw.cmd
pom.xml
User.java
1 package com.pucp.mechatronics.microservices.entity;
2 public class User {
3 private long id;
4 private String Name;
5 private String Lastname;
6 private String Email;
7
8 public User(long id, String name, String lastname, String email) {
9 super();
10 this.id = id;
11 Name = name;
12 Lastname = lastname;
13 Email = email;
14 }
15
16 public User(String name, String lastname, String email) {
17 super();
18 Name = name;
19 Lastname = lastname;
20 Email = email;
21 }
22
23 public long getId() {
24 return id;
25 }
26
27 public void setId(long id) {
28 this.id = id;
29 }
30
31 public String getName() {
32 return Name;
33 }

La Capa de Servicio

Una capa de servicio es una capa de una aplicación que facilita la comunicación entre el controlador y las demás partes del microservicio. Además, suelen servir de frontera de transacciones y se encargan de autorizarlas.

Implementando la capa de servicio

En este caso implementaremos el paquete com.pucp.mechatronics.microservices.services en conjunto con la interface UserService (New – Interface)

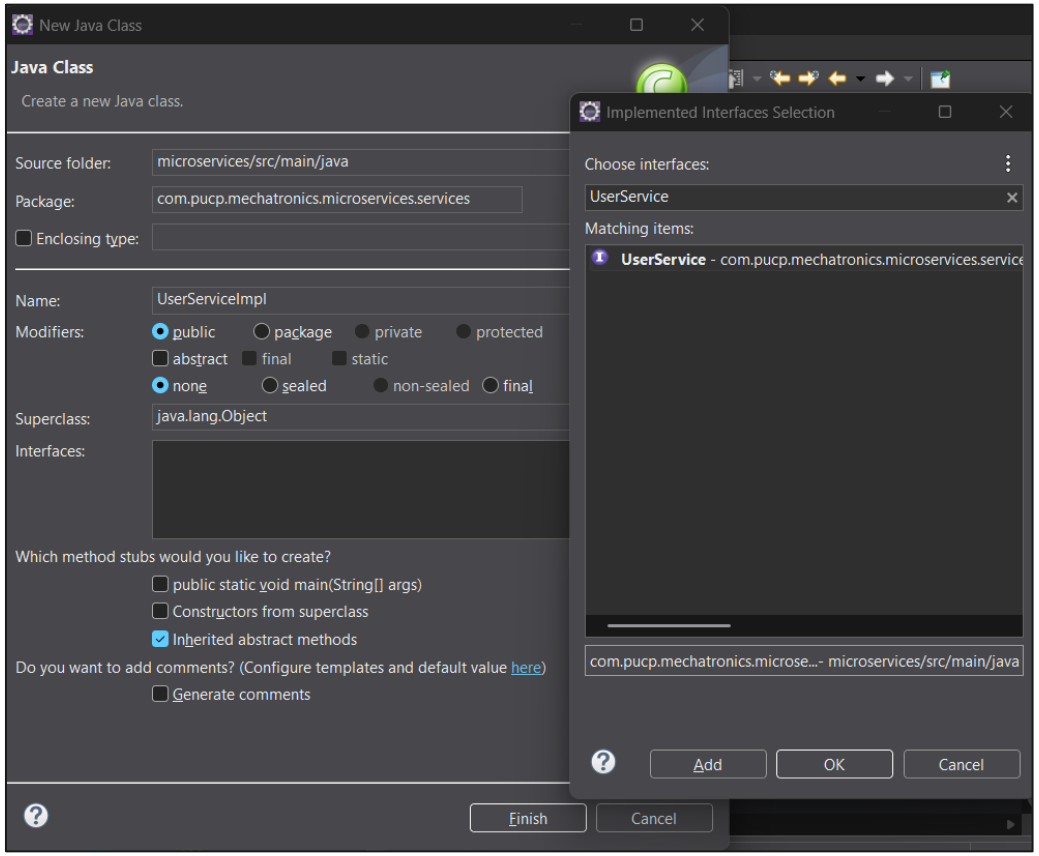



```
package com.pucp.mechatronics.microservices.services;

import java.util.Optional;

import com.pucp.mechatronics.microservices.entity.User;

public interface UserService {

  public Iterable < User > findAll();
  public Optional < User > findById(Long id);
  public User save(User user);
  public void deleteById(Long id);

}
```

Clase UserServiceImpl (@Service)

Adicionalmente creamos la clase UserServiceImpl la cual implementara la interface creada previamente, esto generara todo lo necesario de manera automática.

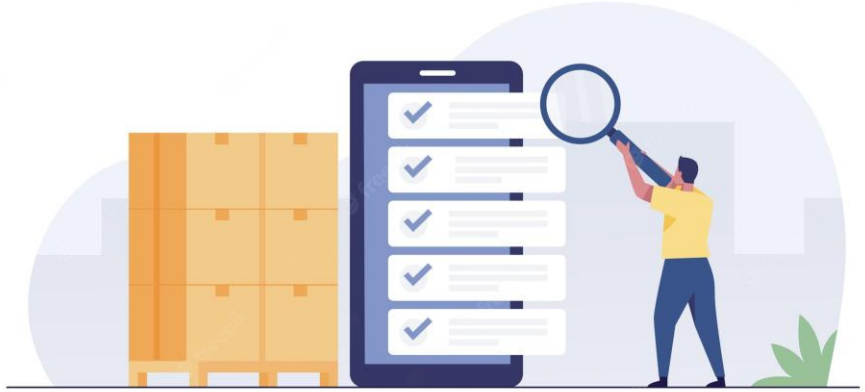
Lo único que tenemos que agregar aquí es la etiqueta @Service




```java
import java.util.Optional;

import org.springframework.stereotype.Service;

import com.pucp.mechatronics.microservices.entity.User;
@Service
public class UserServiceImpl implements UserService {

    @Override
    public Iterable<User> findAll() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public Optional<User> findById(Long id) {
        // TODO Auto-generated method stub
        return Optional.empty();
    }

    @Override
    public User save(User user) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void deleteById(Long id) {
        // TODO Auto-generated method stub

```

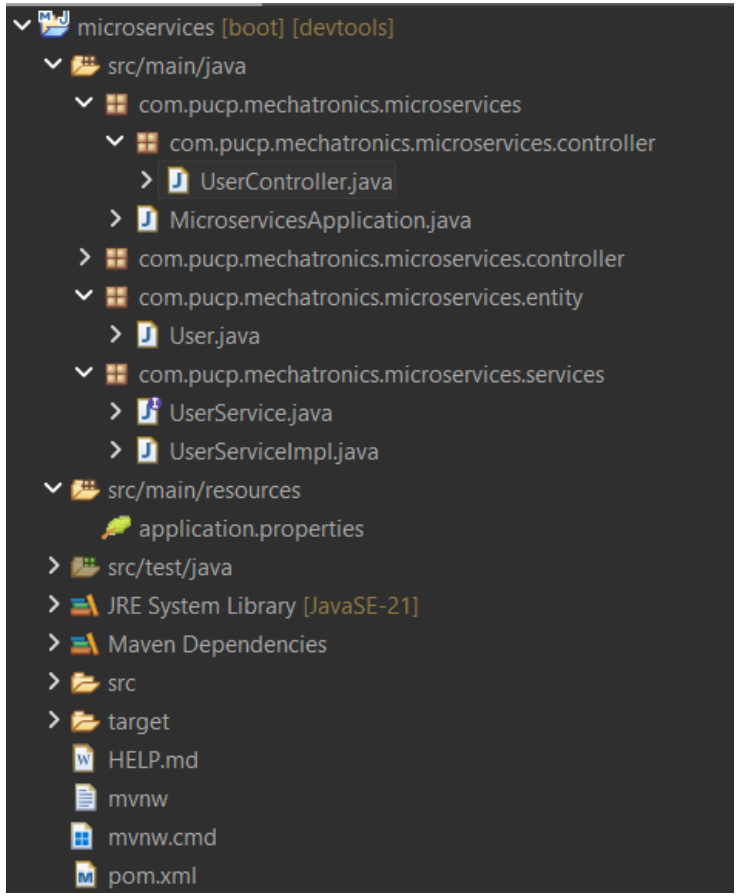
La Capa Controlador

El controlador asigna todas las peticiones a cada proceso y ejecuta las entradas solicitadas. En un proyecto puede haber múltiples controladores para diferentes propósitos, pero todos ellos se refieren al mismo servidor.Además, en el controlador puede haber asignaciones Get, Post y Delete.



Paquete Controller y Clase UserController

Continuando crearemos el paquete com.pucp.mechatronics.microservices.controller y aquí crearemos la clase UserController



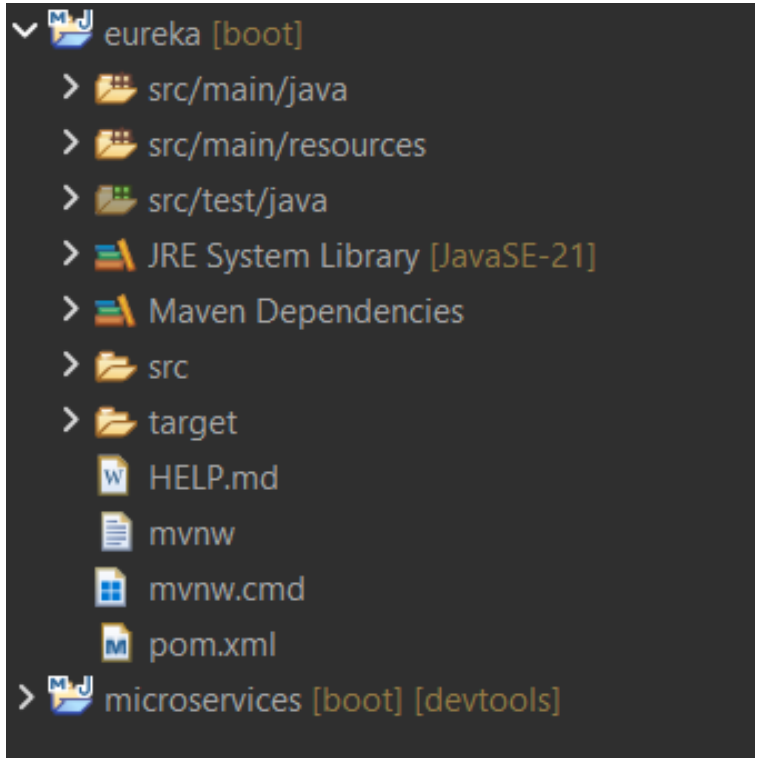
Paquete Controller y Clase UserController

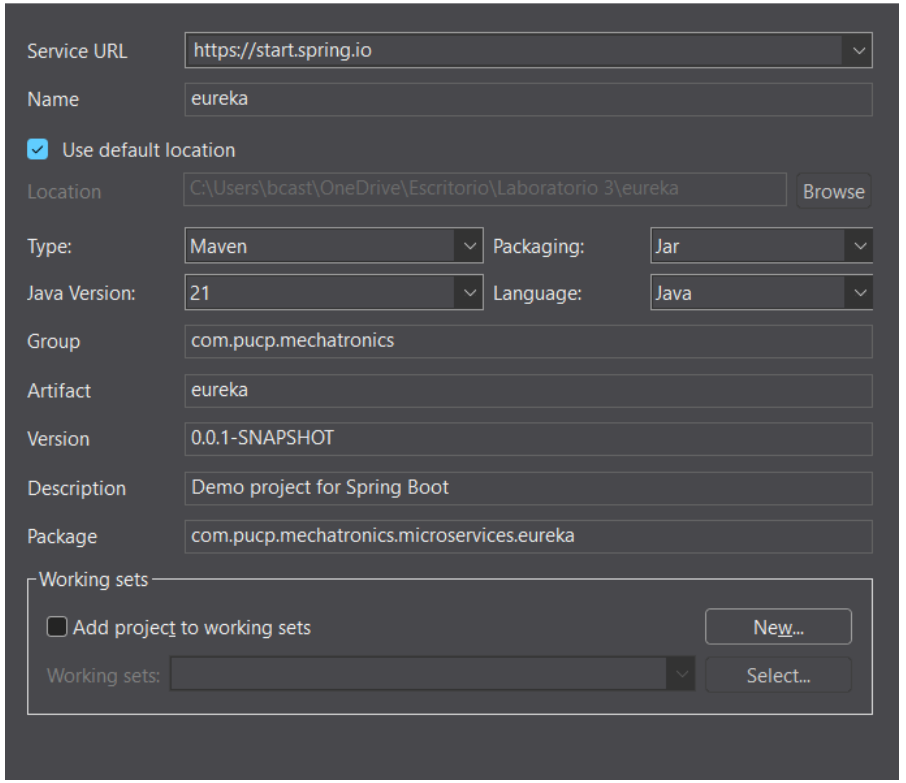
```
package com.pucp.mechatronics.microservices.controller;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.pucp.mechatronics.microservices.entity.User;
import com.pucp.mechatronics.microservices.services.UserService;

@RestController
public class UserController {
        @Autowired UserService service;

        @GetMapping
        public ResponseEntity <?> list(){
                return ResponseEntity.ok().body(service.findAll());
        }

        @GetMapping ("/{id}")
        public ResponseEntity <?> see(@PathVariable Long id){
                Optional <User> o = service.findById(id);
                if (o != null && !o.isPresent() || o == null) {
                        return
ResponseEntity.notFound().build();
                                                                }
                return ResponseEntity.ok().body(o);
        }
```

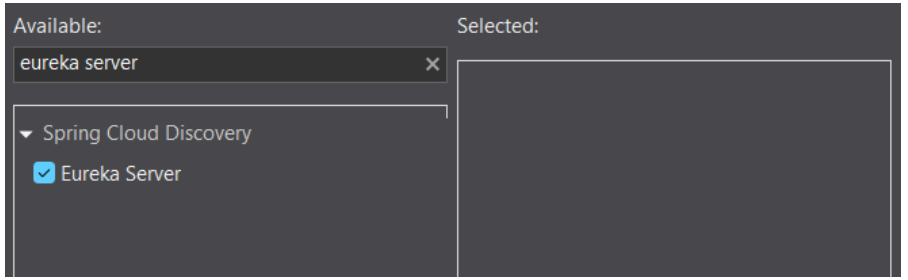
Paquete Controller y Clase UserController

```
        @PostMapping
        public ResponseEntity<?>
create(@RequestBody User user){
                User userDb = service.save(user);
                return
ResponseEntity.status(HttpStatus.CREATED).body(userD
b);
        }

        @PutMapping
        public ResponseEntity<?>
edit(@RequestBody User user,@PathVariable Long id){
                Optional <User> o =
service.findById(id);
                if (o != null && !o.isPresent() || o
== null) {
                        return
ResponseEntity.notFound().build();
                }
                User userDb = o.get();
                userDb.setName(user.getName());
                userDb.setLastname(user.getLastname());
                userDb.setEmail(user.getEmail());

                return
ResponseEntity.status(HttpStatus.CREATED).body(service.save(
userDb));

        }

        @DeleteMapping("/{id}")
        public ResponseEntity<?> delete(@PathVariable
Long id){
                service.deleteById(id);
                return ResponseEntity.noContent().build();
        }
}
```

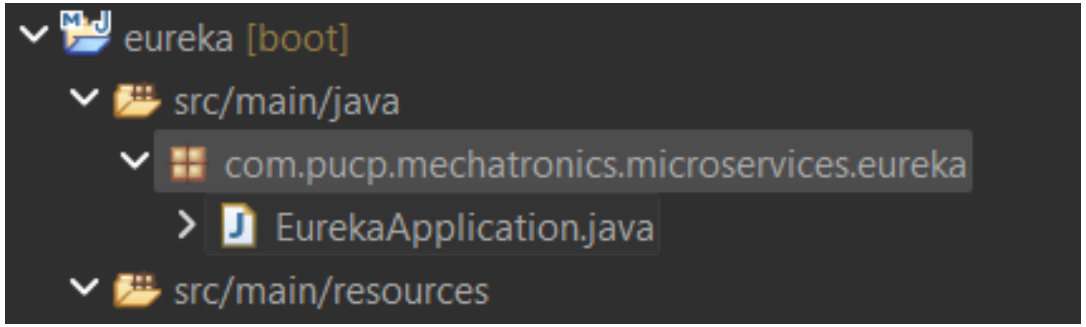
Creando el servidor eureka







Habilitando el Servidor



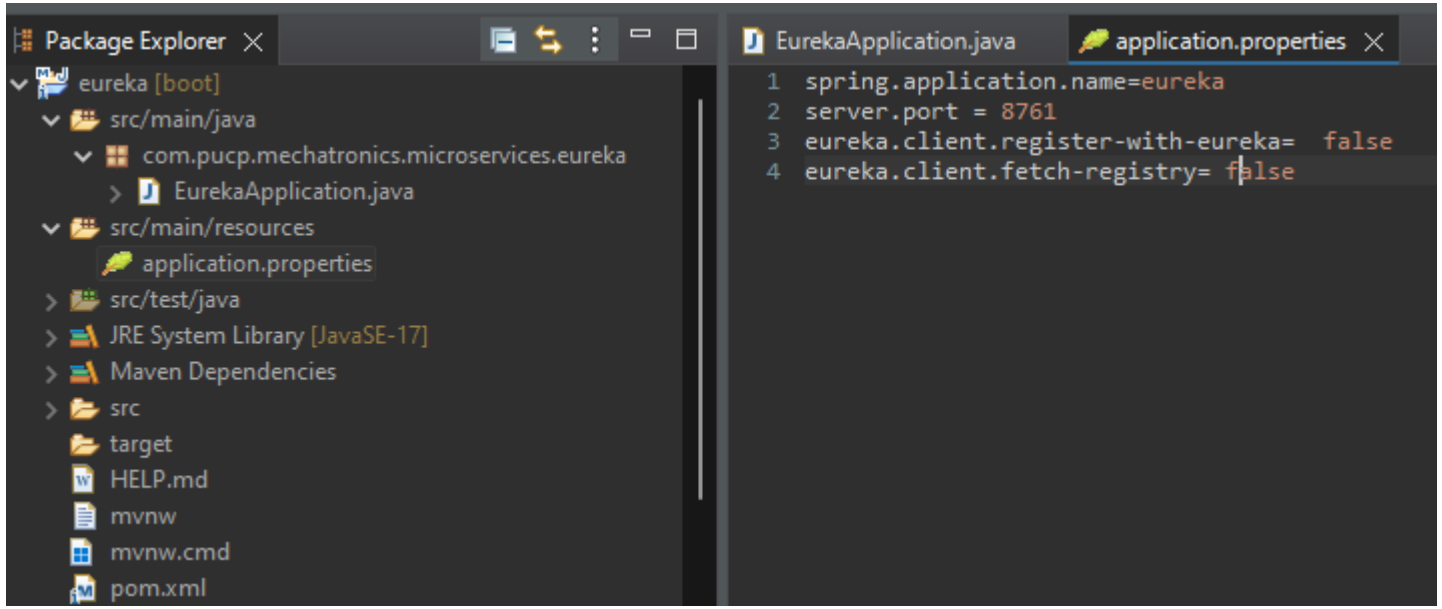
EurekaApplication.java

```java
package com.pucp.mechatronics.microservices.eureka;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer
@SpringBootApplication
public class EurekaApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }

}
```

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

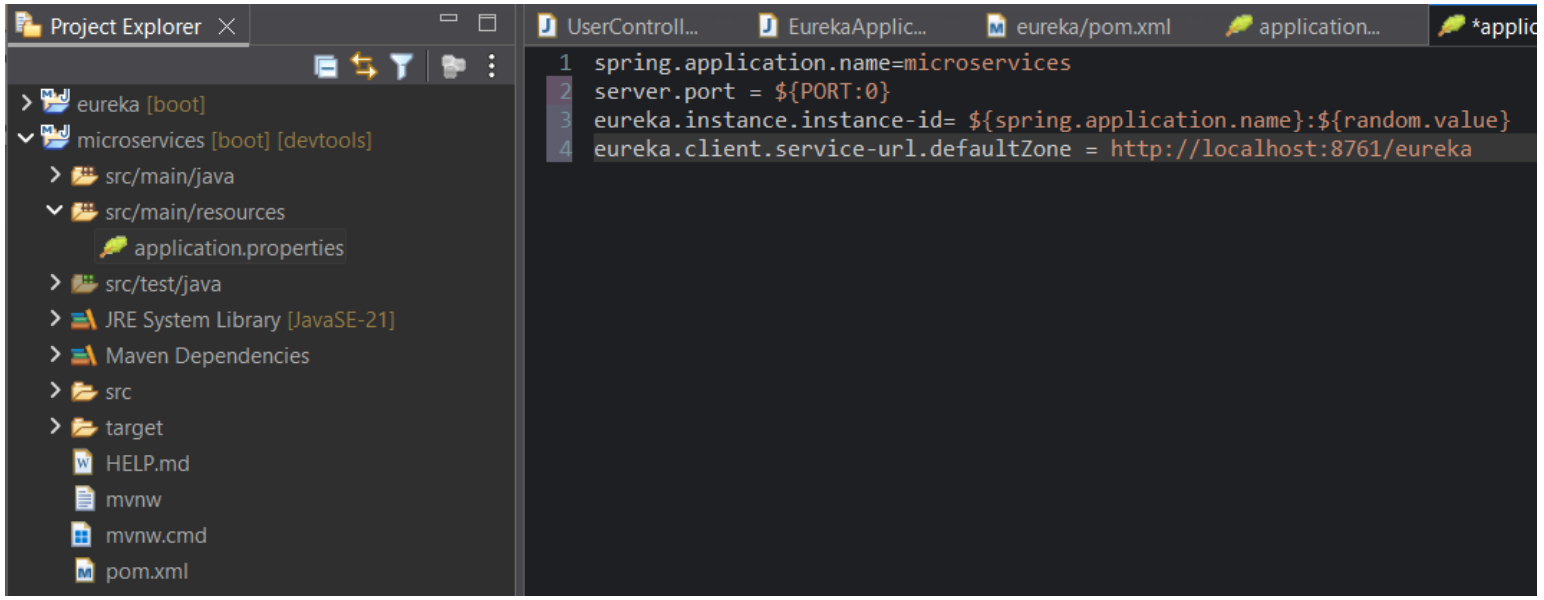
@EnableEurekaServer

Editando las propiedades de eureka



server.port = 8761
eureka.client.register-with-eureka= false
eureka.client.fetch-registry= false

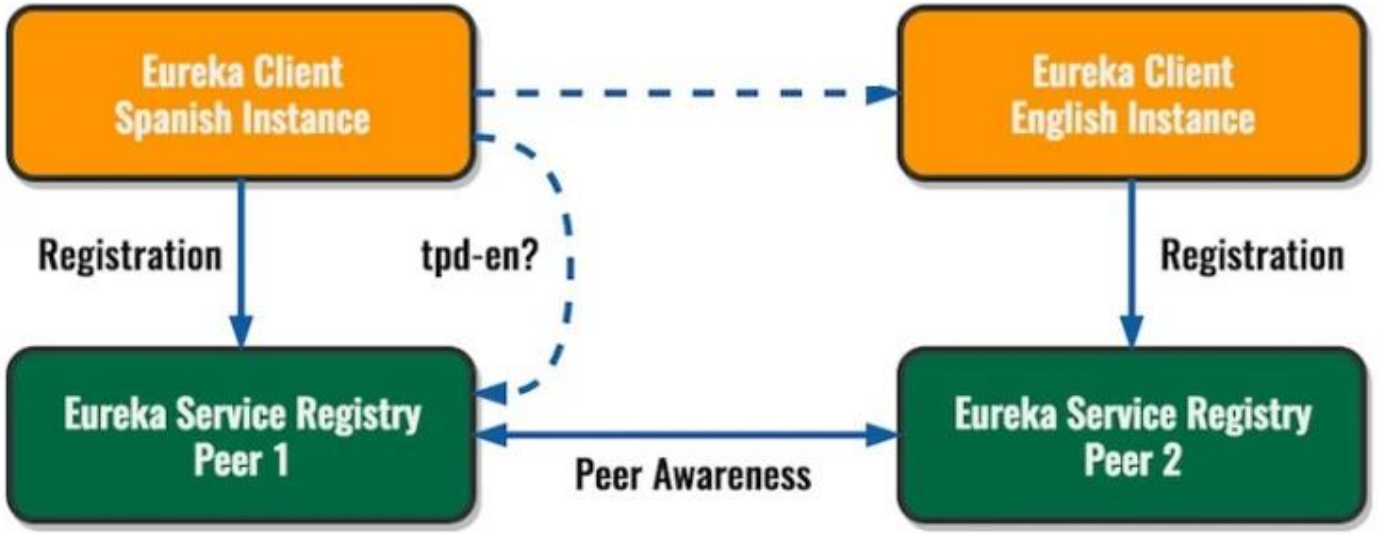
Conectando el user Service a Eureka en microservices



spring.application.name=microservices
server.port = ${PORT:0}
eureka.instance.instance-id=
${spring.application.name}:${random.value}
eureka.client.service-url.defaultZone =
http://localhost:8761/eureka

Eureka Peer Awareness (/Eureka)

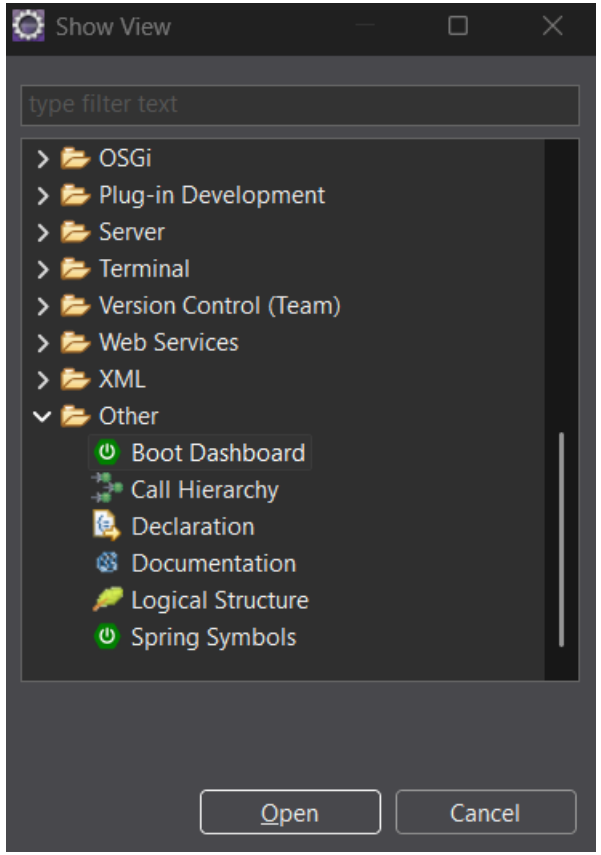
Nos permite tener dos instancias de registro que pueden compartir entre sí el contenido del registro (los clientes registrados) para que puedan implementar la resiliencia y esos clientes puedan encontrarse entre sí poniéndose en contacto con la instancia que conocen.

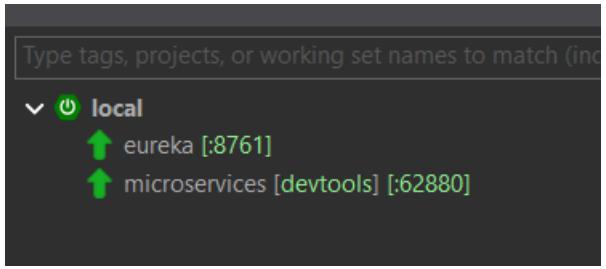


Probando el Servicio corriendo ambos proyectos

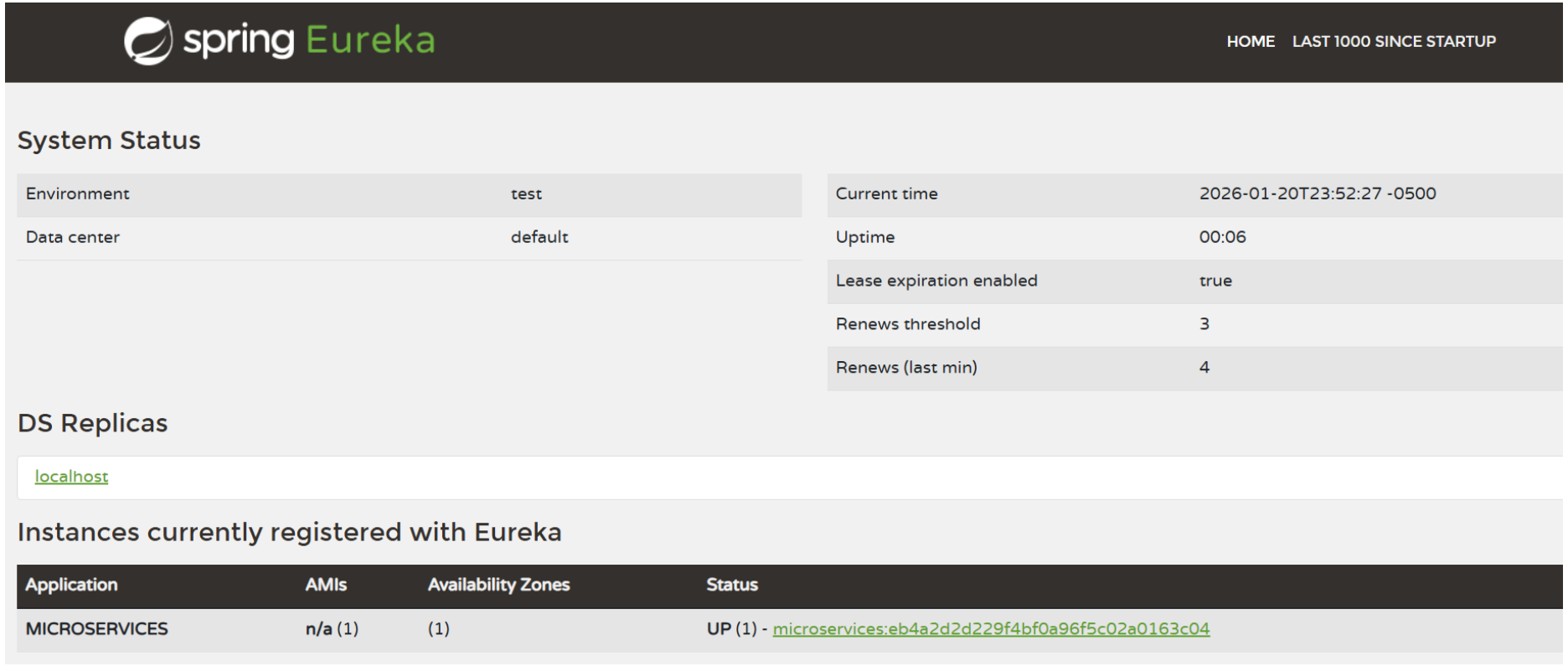
Para esto, iremos a window , show view, other y ahí buscamos Boot Dashboard.

Una vez ya tengamos esto, daremos click derecho a cada proyecto y haremos run as : Spring Boot App

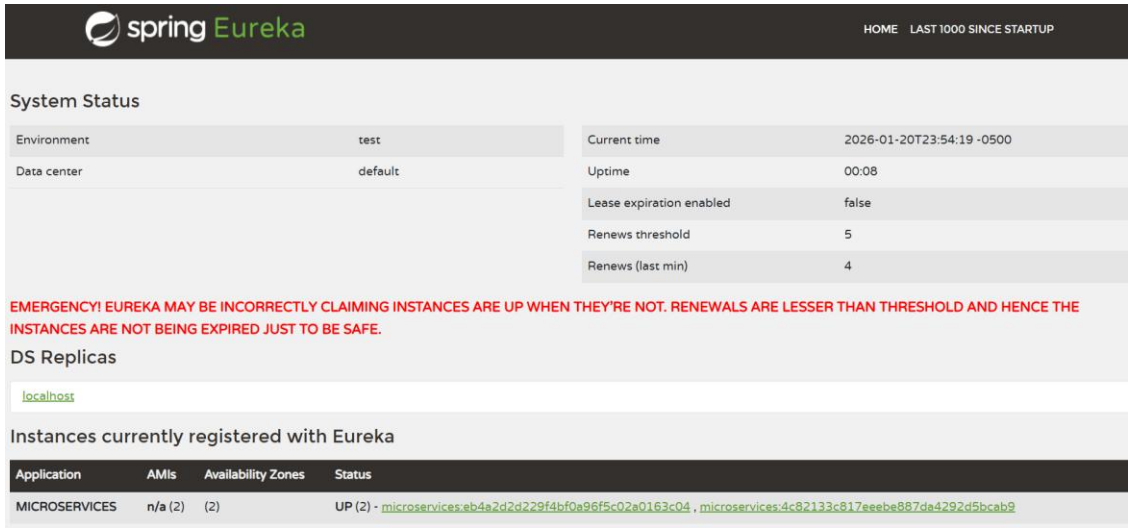


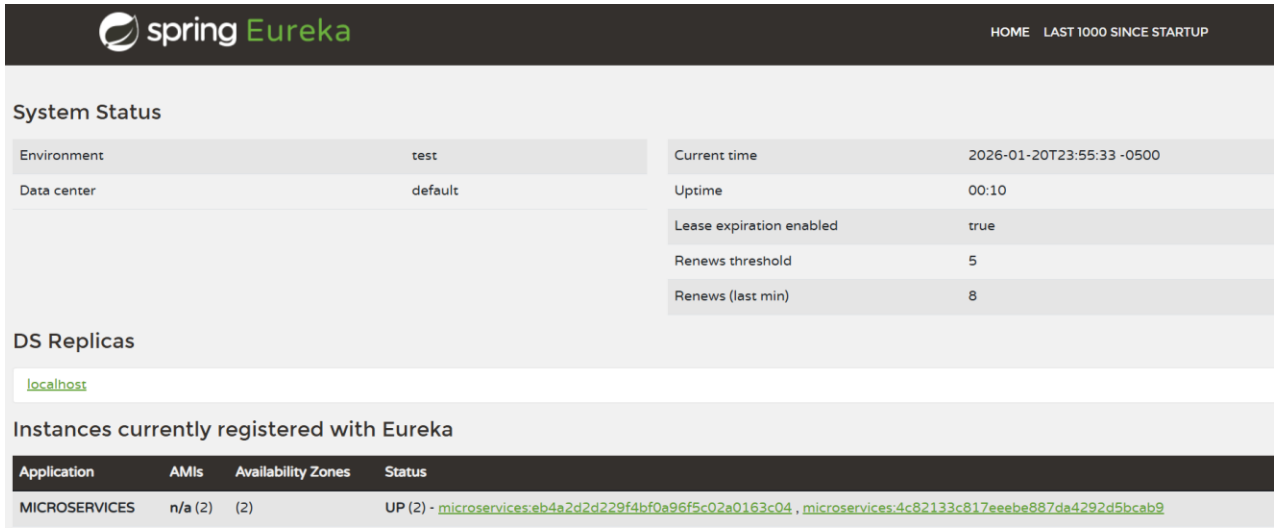


Visualizando los servicios : Http://localhost:8761



Nuevas Instancias : Nuevo up Servicios





Ese mensaje en rojo brillante en el Dashboard de Eureka es la **"Self Preservation Mode"** (Modo de Autopreservación). Es una medida de seguridad de Eureka para evitar que, ante un problema de red, se borren todos los servicios si el servidor deja de recibir sus "heartbeats".