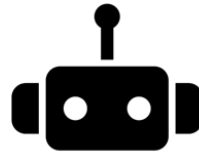




**PUCP**

## SESIÓN DE LABORATORIO 2

### Microservicios Parte 1



*HORARIO 10M1*

Empezaremos a las 7:10 p.m

Gracias!



# ¿Por qué una arquitectura debe utilizar microservicios?

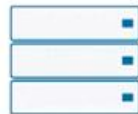


# Arquitectura de Microservicios

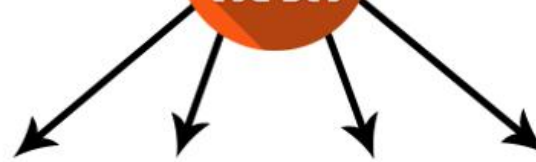
*Monolithic Architecture*



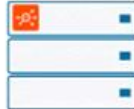
*App Services*



*Microservices Architecture*



*Microservice*



*Microservice*



*Microservice*



*Microservice*

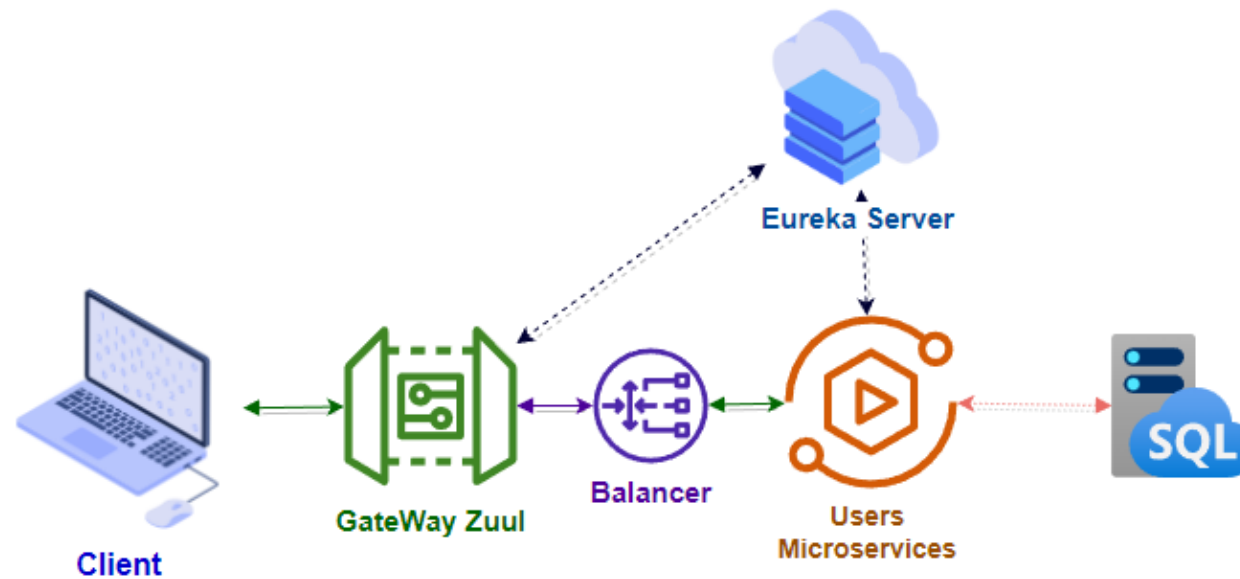


# Ejercicio

## 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.



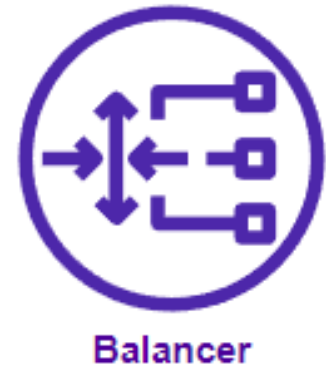
# Gateway Zuul

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

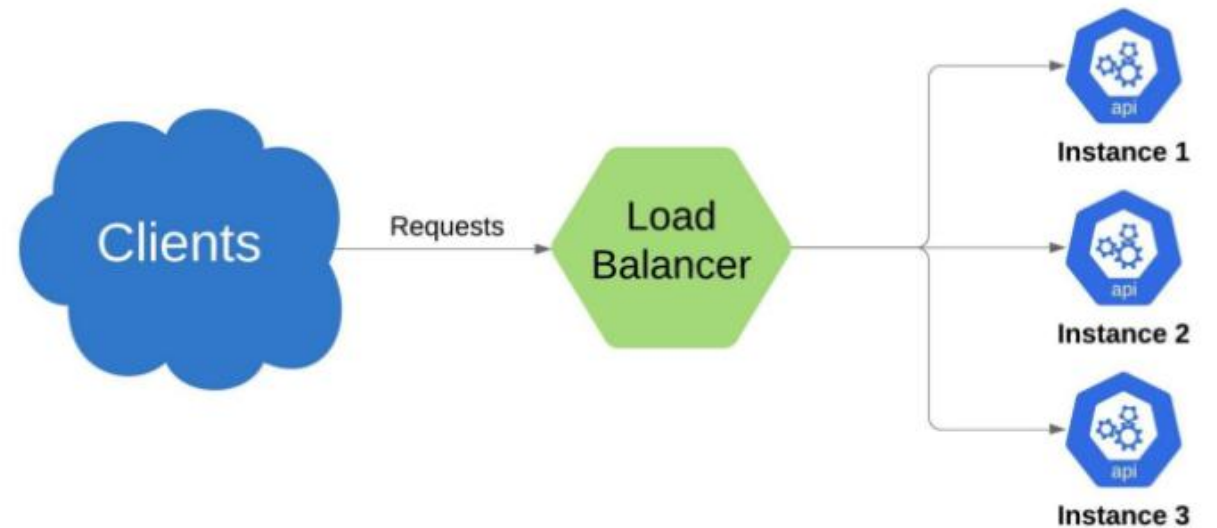
El balanceo de carga es el proceso de distribuir el tráfico de red entrante entre múltiples servidores para optimizar el uso de recursos, garantizar la disponibilidad y mejorar la capacidad de respuesta del sistema. Esta distribución puede realizarse de manera uniforme o basada en reglas específicas,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





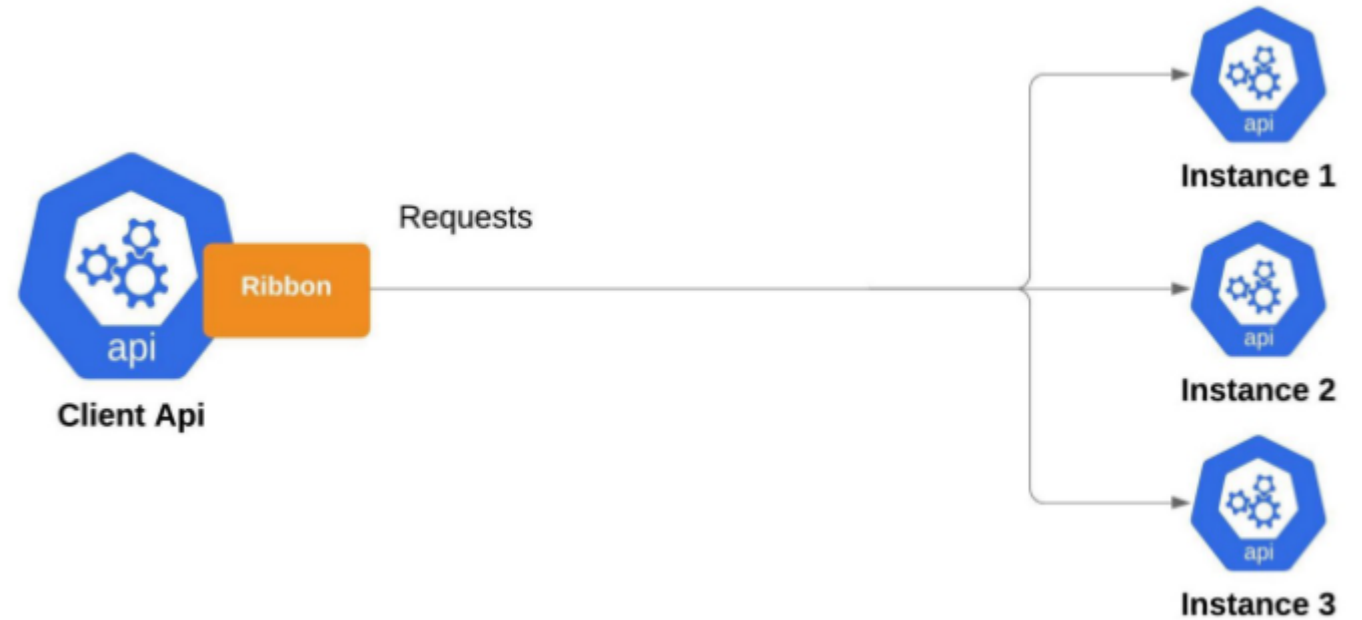
# Server Side Load balancer

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.



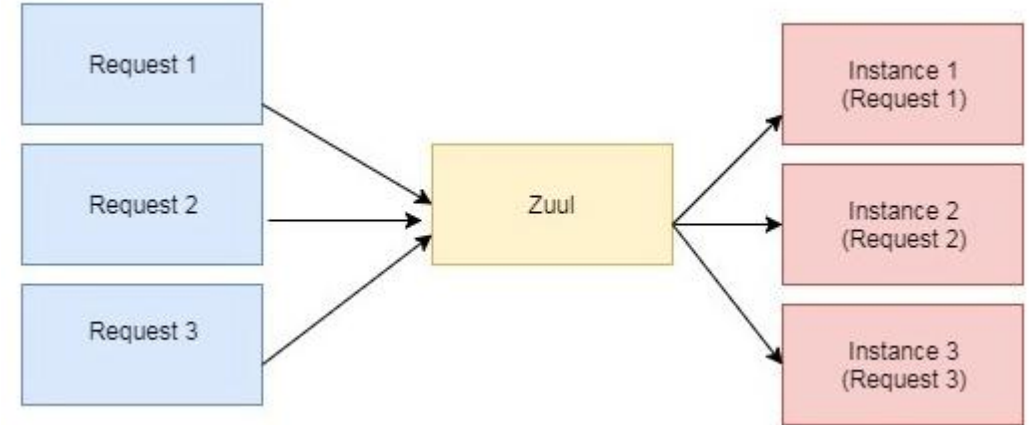
# Zuul Load Balancer

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.



Eureka 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



Eureka Server  
**Spring Cloud**

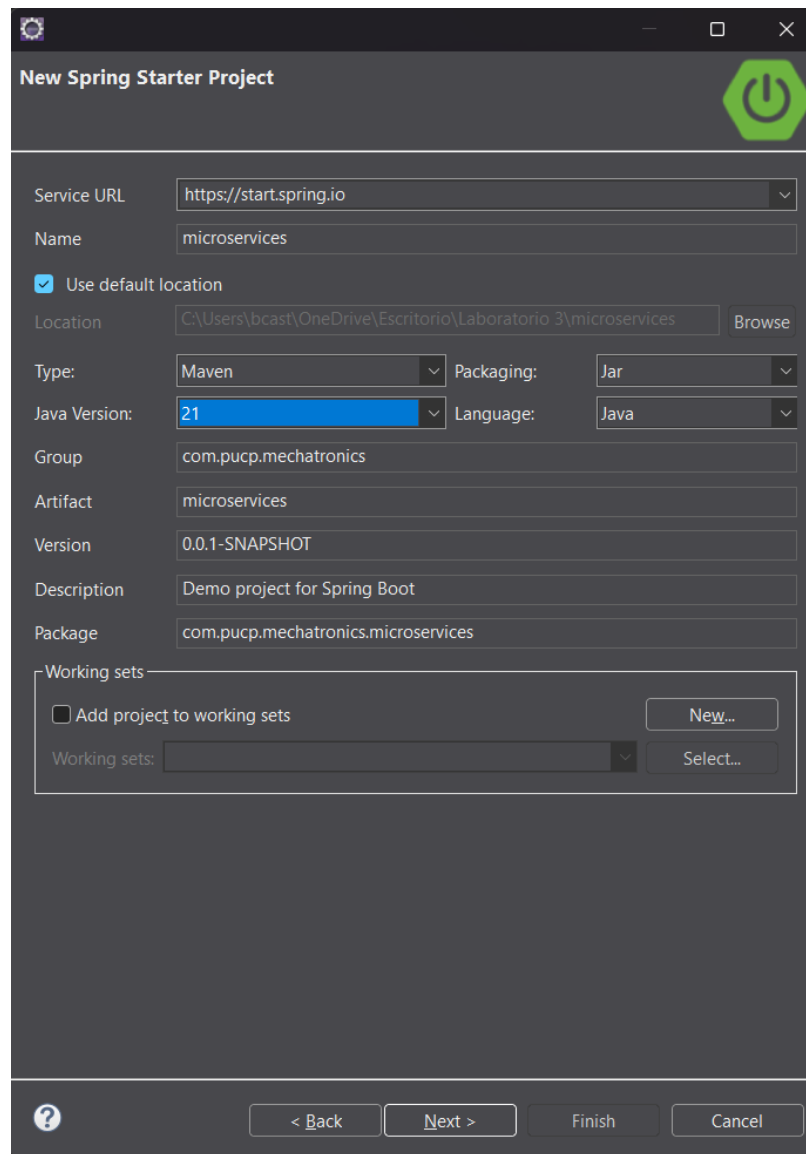
# Consideraciones

Vamos a continuar usando Spring Tools 4 y Java 21



# Desarrollo – Creando el microservicio

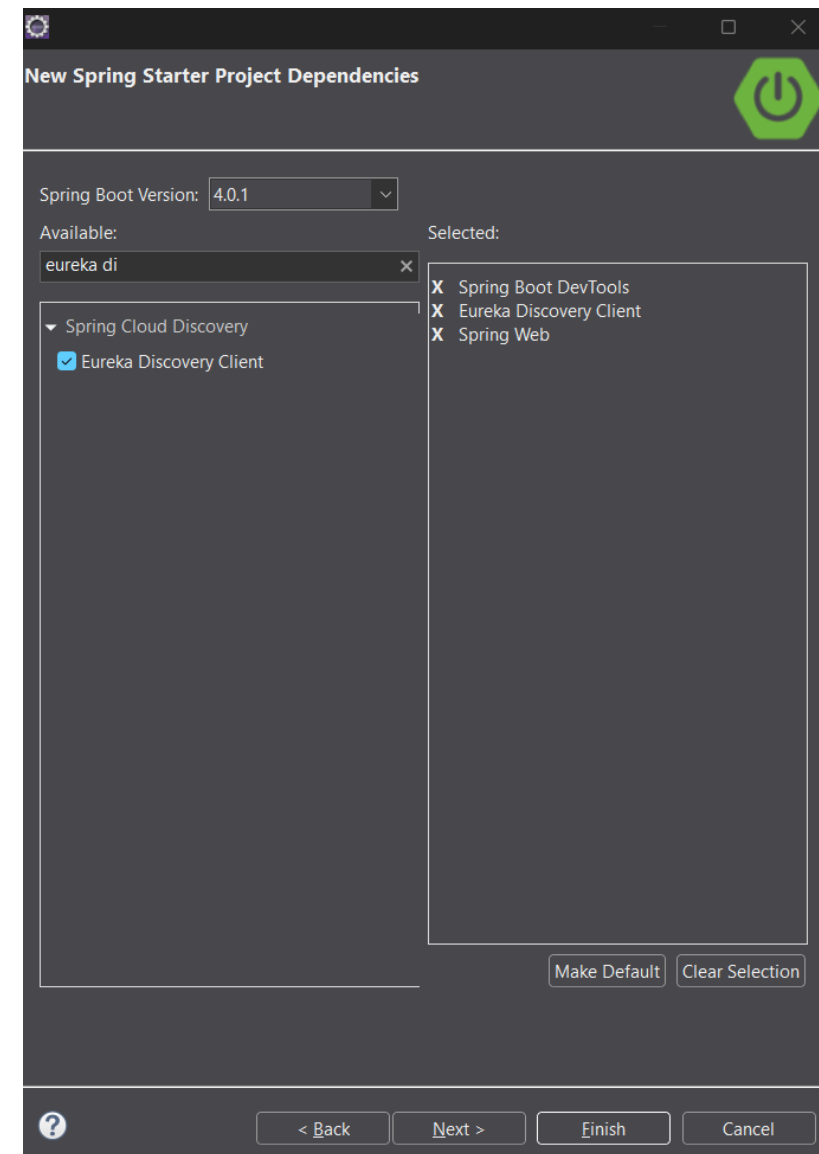
Aquí utilizaremos nuevamente el Spring Starter Project y además para las dependencias utilizaremos Spring Boot DevTools, Spring Web y Eureka Discovery client



The 'New Spring Starter Project' dialog is shown with the following configuration:

- Service URL: `https://start.spring.io`
- Name: `microservices`
- ☒ Use default location
- Location: `C:\Users\bcast\OneDrive\Escritorio\Laboratorio 3\microservices` (with a 'Browse' button)
- Type: `Maven` (dropdown)
- Packaging: `Jar` (dropdown)
- Java Version: `21` (dropdown)
- Language: `Java` (dropdown)
- Group: `com.pucp.mechatronics`
- Artifact: `microservices`
- Version: `0.0.1-SNAPSHOT`
- Description: `Demo project for Spring Boot`
- Package: `com.pucp.mechatronics.microservices`
- Working sets section:
  - ☐ Add project to working sets (with a 'New...' button)
  - Working sets: (dropdown menu) (with a 'Select...' button)

Navigation buttons at the bottom: `< Back`, `Next >`, `Finish`, and `Cancel`.



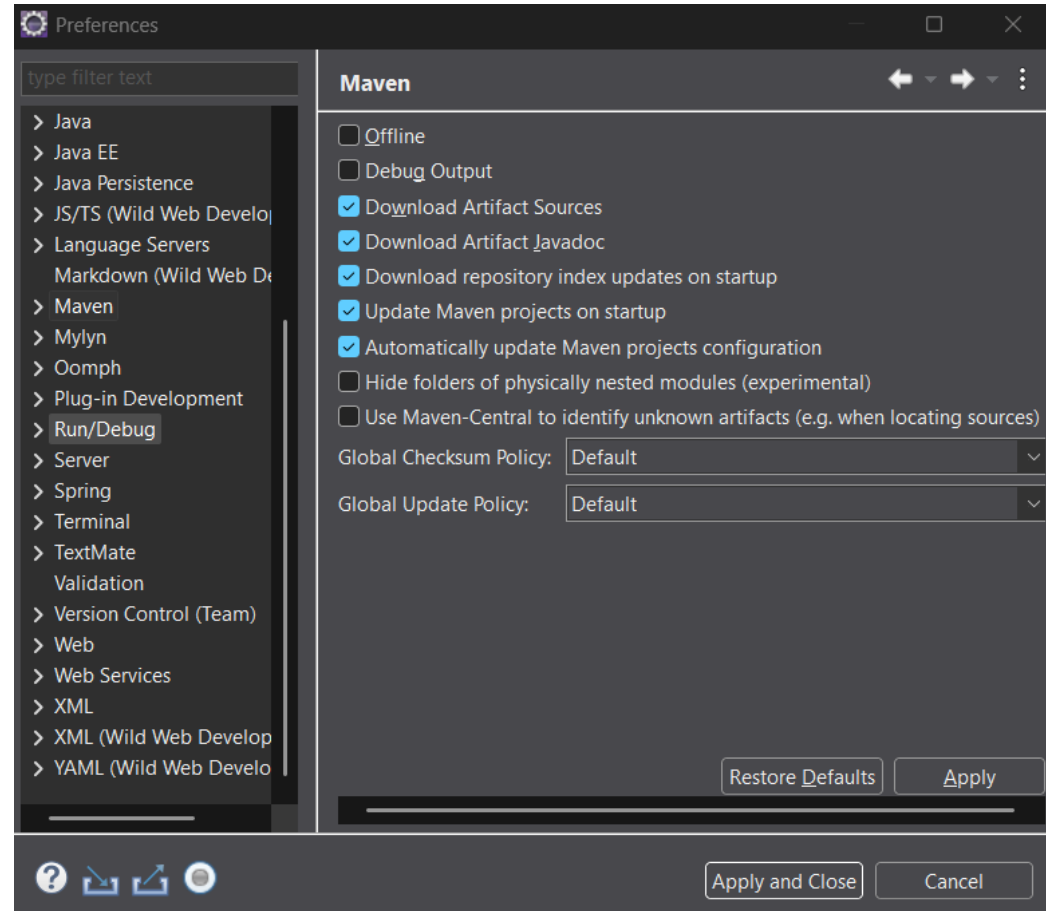
The 'New Spring Starter Project Dependencies' dialog is shown with the following configuration:

- Spring Boot Version: `4.0.1` (dropdown)
- Available: `eureka di` (dropdown)
- Selected: (list of dependencies with checkboxes)
  - ☒ Spring Boot DevTools
  - ☒ Eureka Discovery Client
  - ☒ Spring Web
- Buttons at the bottom right: `Make Default` and `Clear Selection`

Navigation buttons at the bottom: `< Back`, `Next >`, `Finish`, and `Cancel`.

# 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





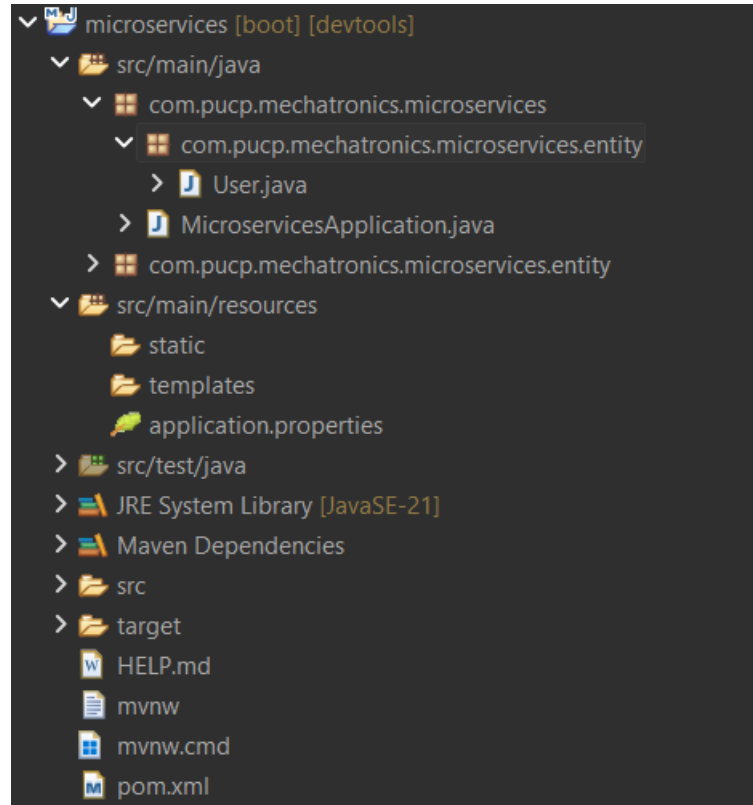
# La Capa entidad

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

Generate Constructor using Fields

Select super constructor to invoke:  
Object()

Select fields to initialize:

<input checked="" type="checkbox"/>	id
<input checked="" type="checkbox"/>	Name
<input checked="" type="checkbox"/>	Lastnames
<input checked="" type="checkbox"/>	email

Select All  
Deselect All  
Up  
Down

Insertion point:  
After 'User()'

Access modifier  
☒ public ☐ protected ☐ package ☐ private

☐ Generate constructor comments  
☐ Omit call to default constructor super()

The format of the constructors may be configured on the [Code Templates](#) preference page.

4 of 4 selected.

Generate Cancel

Generate Constructor using Fields

Select super constructor to invoke:  
Object()

Select fields to initialize:

<input type="checkbox"/>	id
<input checked="" type="checkbox"/>	Name
<input checked="" type="checkbox"/>	Lastnames
<input checked="" type="checkbox"/>	email

Select All  
Deselect All  
Up  
Down

Insertion point:  
After 'User(long, String, String, String)'

Access modifier  
☒ public ☐ protected ☐ package ☐ private

☐ Generate constructor comments  
☐ Omit call to default constructor super()

The format of the constructors may be configured on the [Code Templates](#) preference page.

3 of 4 selected.

Generate Cancel

Generate Getters and Setters

Select getters and setters to create:

>	<input checked="" type="checkbox"/>	email
>	<input checked="" type="checkbox"/>	id
>	<input checked="" type="checkbox"/>	Lastnames
>	<input checked="" type="checkbox"/>	Name

Select All  
Deselect All  
Select Getters  
Select Setters

☐ Allow setters for final fields (remove 'final' modifier from fields if necessary)

Insertion point:  
After 'User(String, String, String)'

Sort by:  
Fields in getter/setter pairs

Access modifier  
☒ public ☐ protected ☐ package ☐ private  
☐ final ☐ synchronized

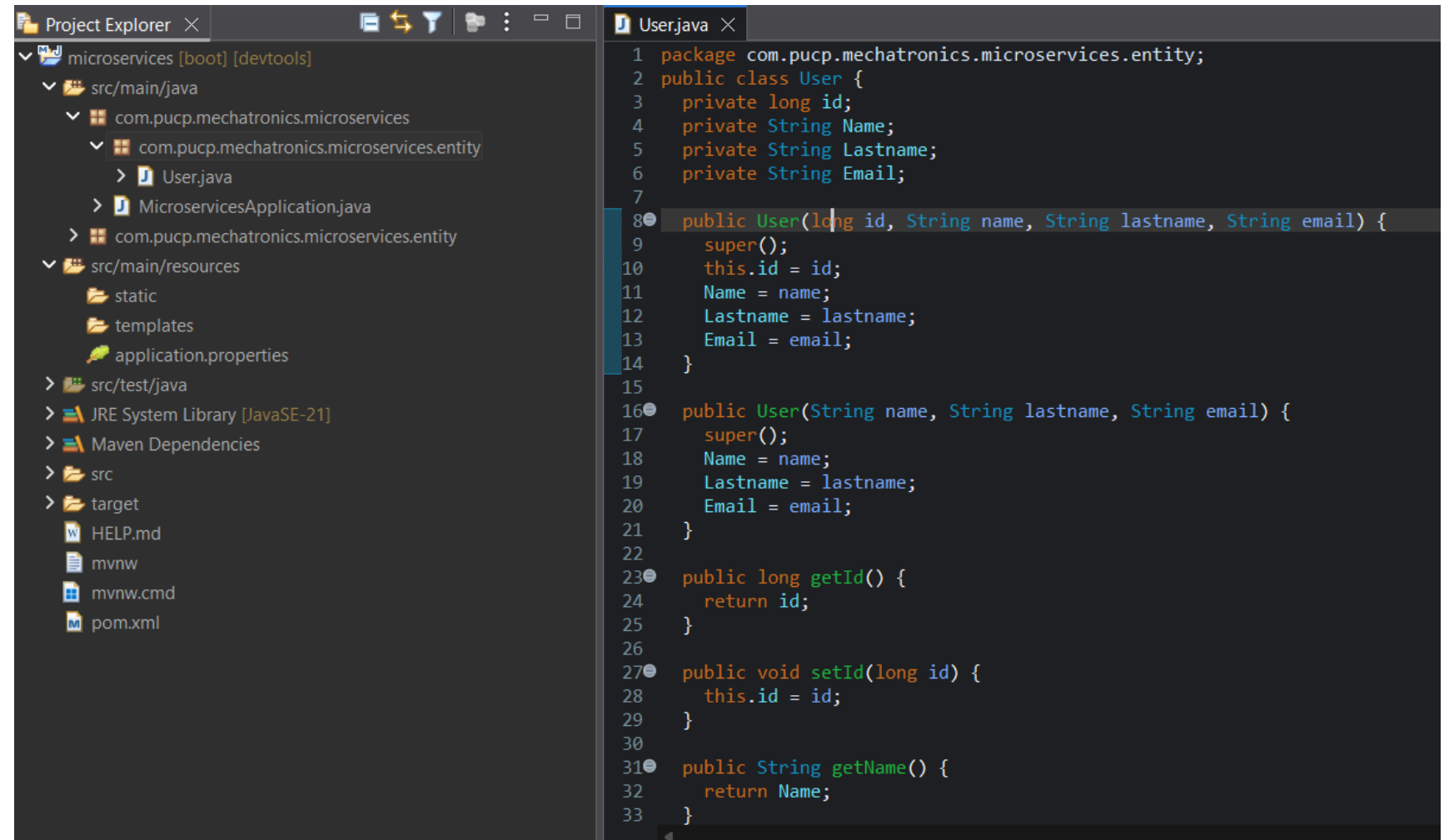
☐ Generate method comments

The format of the getters/setters may be configured on the [Code Templates](#) preference page.

8 of 8 selected.

Generate Cancel

# Clase User



The screenshot displays an IDE interface. On the left, the Project Explorer shows a project structure with the following hierarchy:

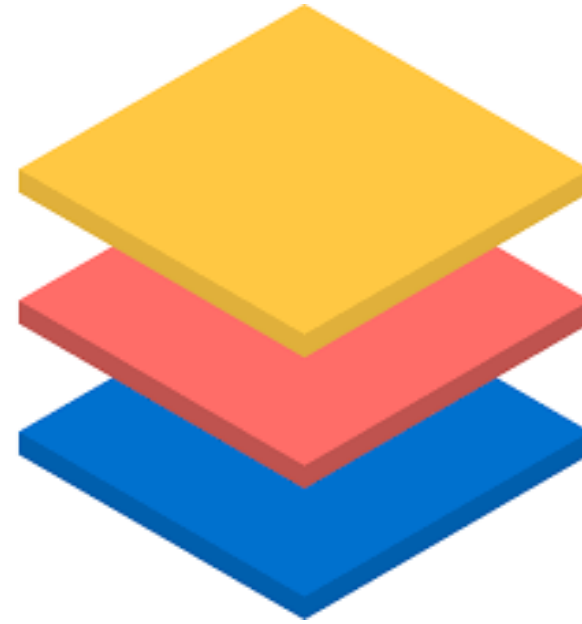
- microservices [boot] [devtools]
  - src/main/java
    - com.pucp.mechatronics.microservices
      - com.pucp.mechatronics.microservices.entity
        - User.java (selected)
        - MicroservicesApplication.java
  - src/main/resources
    - static
    - templates
    - application.properties
  - src/test/java
  - JRE System Library [JavaSE-21]
  - Maven Dependencies
  - src
  - target
  - HELP.md
  - mvnw
  - mvnw.cmd
  - pom.xml

On the right, the source code of User.java is shown:

```
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    }
34 }
```

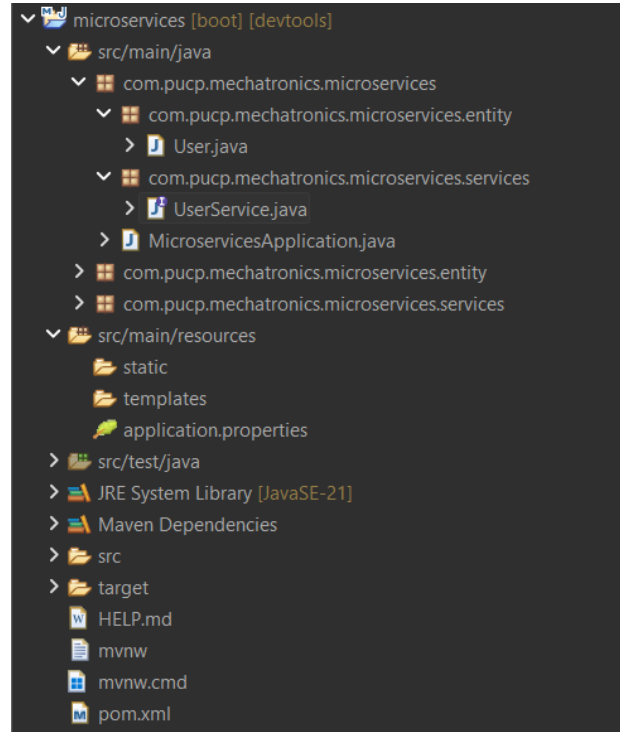
# La Capa de Servicio

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 {

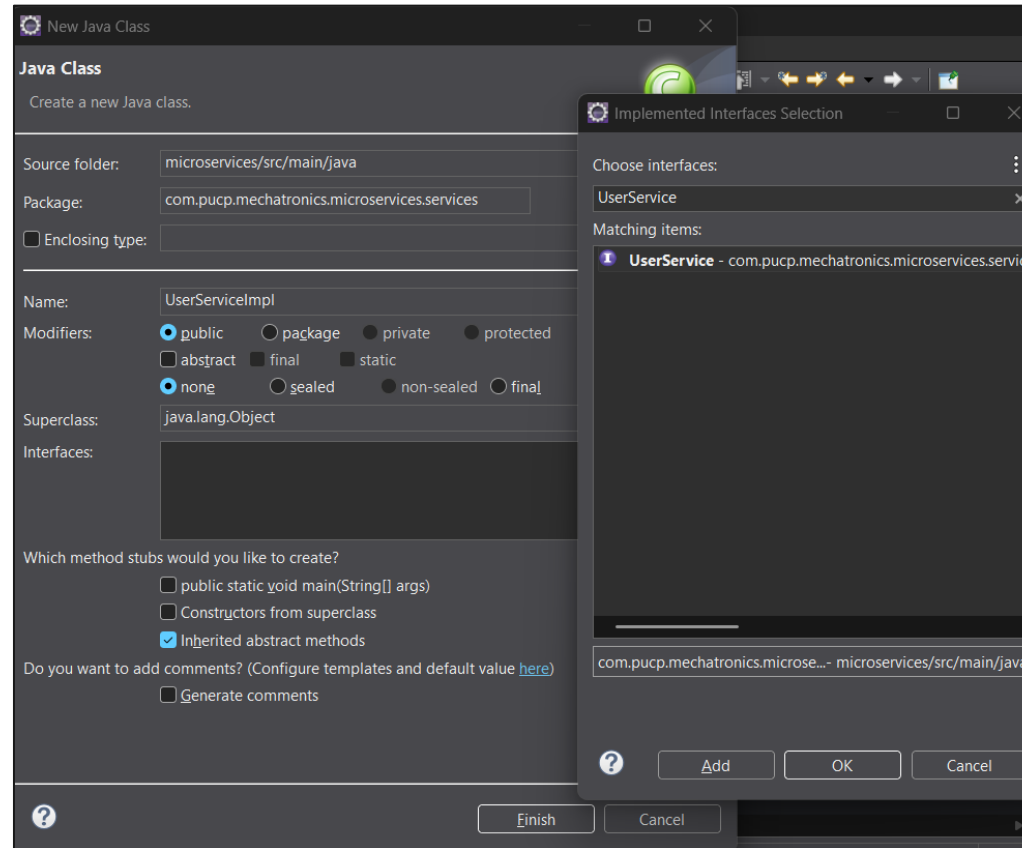
    public Iterable < User > findAll();
    public Optional < User > findById(Long id);
    public User save(User user);
    public void deleteById(Long id);

}
```

# Clase UserServiceImpl (@Service)

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

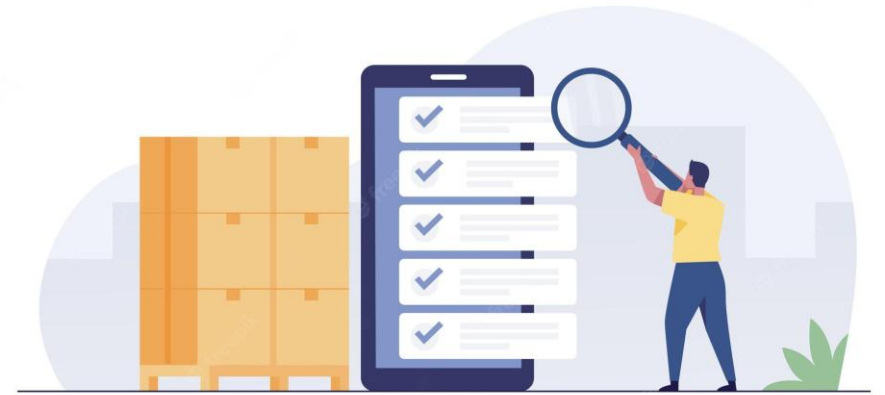
Lo único que tenemos que agregar aquí es la etiqueta @Service



```
2
3 import java.util.Optional;
4
5 import org.springframework.stereotype.Service;
6
7 import com.pucp.mechatronics.microservices.entity.User;
8 @Service
9 public class UserServiceImpl implements UserService {
10
11     @Override
12     public Iterable<User> findAll() {
13         // TODO Auto-generated method stub
14         return null;
15     }
16
17     @Override
18     public Optional<User> findById(Long id) {
19         // TODO Auto-generated method stub
20         return Optional.empty();
21     }
22
23     @Override
24     public User save(User user) {
25         // TODO Auto-generated method stub
26         return null;
27     }
28
29     @Override
30     public void deleteById(Long id) {
31         // TODO Auto-generated method stub
32     }
33 }
```

# La Capa Controlador

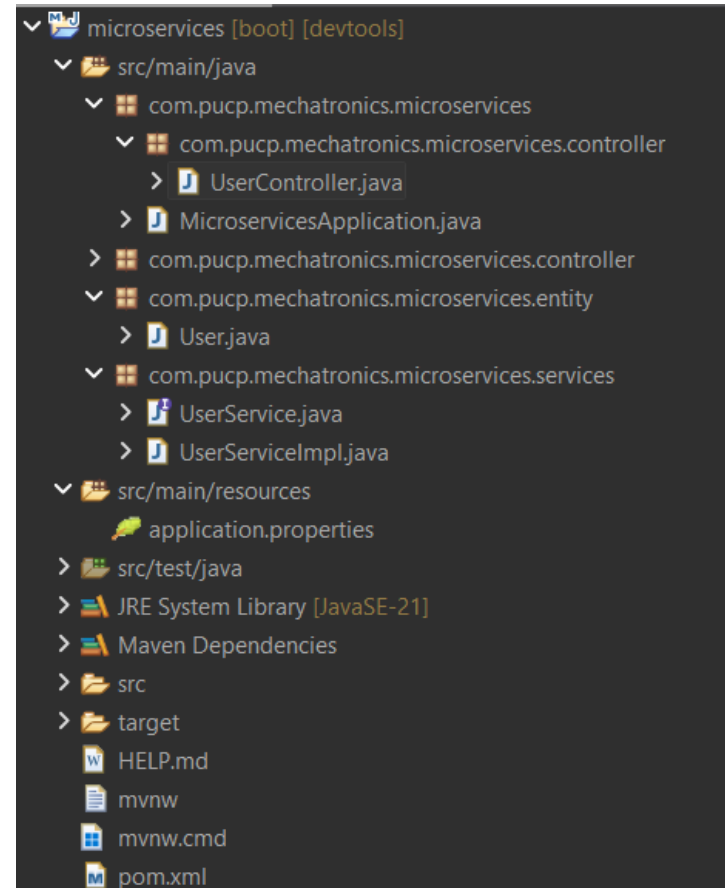
El controlador asigna todas las peticiones a cada proceso y ejecuta las entradas solicitadas. En un proyecto puede haber múltiples controladores para diferentes propósitos, pero todos ellos se refieren al mismo servidor. Además, en el controlador puede haber asignaciones Get, Post y Delete.





# 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);
}
```

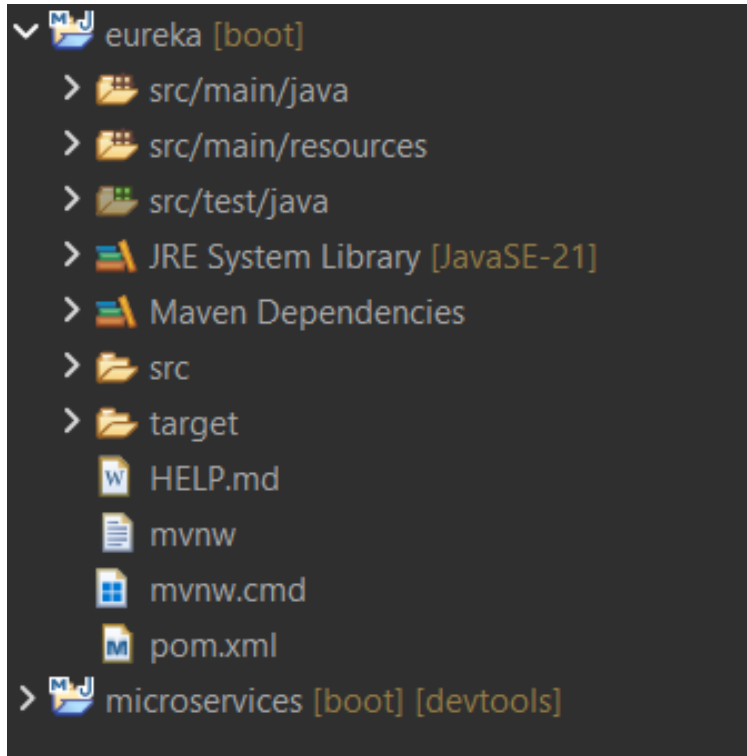
```
@PutMapping
public ResponseEntity<?>
edit(@RequestBody User user,@PathVariable Long id){
    Optional <User> o =
    service.findById(id);
    if (o != null && !o.isPresent() || o
== null) {
        return
```

```
        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



Service URL:

Name:

☒ Use default location

Location:

Type:  Packaging:

Java Version:  Language:

Group:

Artifact:

Version:

Description:

Package:

Working sets

☐ Add project to working sets

Working sets:

Available:

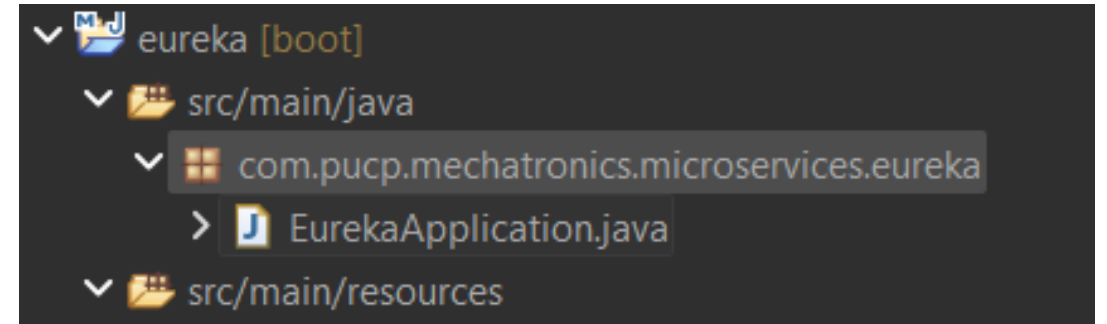
Selected:

Spring Cloud Discovery

☒ Eureka Server

# Habilitando el Servidor

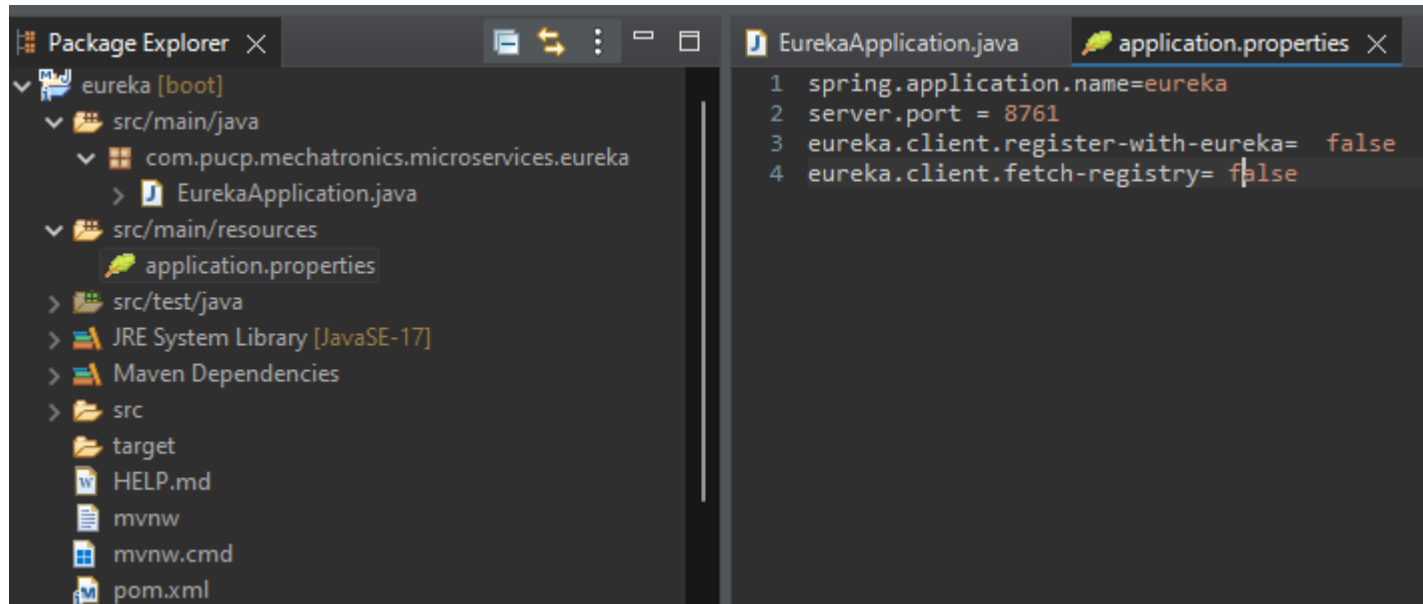
```
EurekaApplication.java ×
1 package com.pucp.mechatronics.microservices.eureka;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
6
7 @EnableEurekaServer
8 @SpringBootApplication
9 public class EurekaApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(EurekaApplication.class, args);
13     }
14
15 }
16
```



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

@EnableEurekaServer
```

# Editando las propiedades de eureka

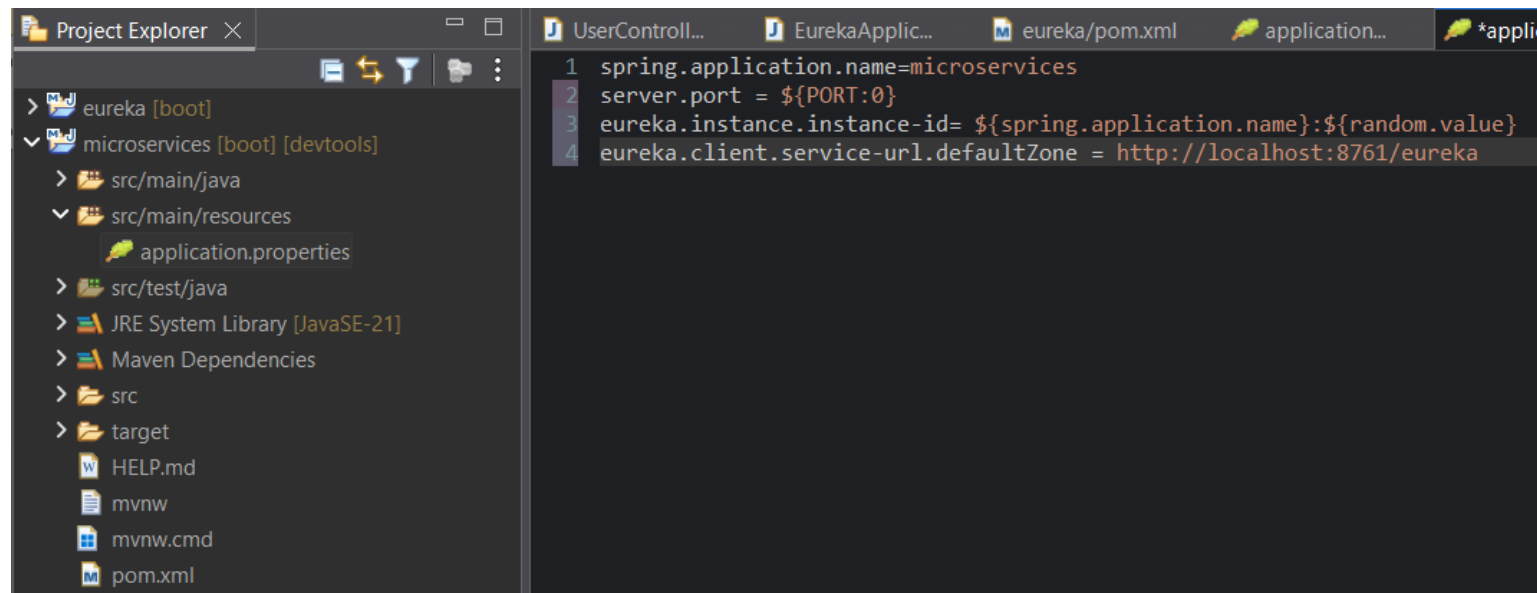


The screenshot shows an IDE interface. On the left, the Package Explorer displays the project structure: 'eureka [boot]' contains 'src/main/java' (with package 'com.pucp.mechatronics.microservices.eureka' and file 'EurekaApplication.java') and 'src/main/resources' (containing 'application.properties'). On the right, the 'application.properties' file is open, showing the following properties:

```
1 spring.application.name=eureka
2 server.port = 8761
3 eureka.client.register-with-eureka= false
4 eureka.client.fetch-registry= false
```

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

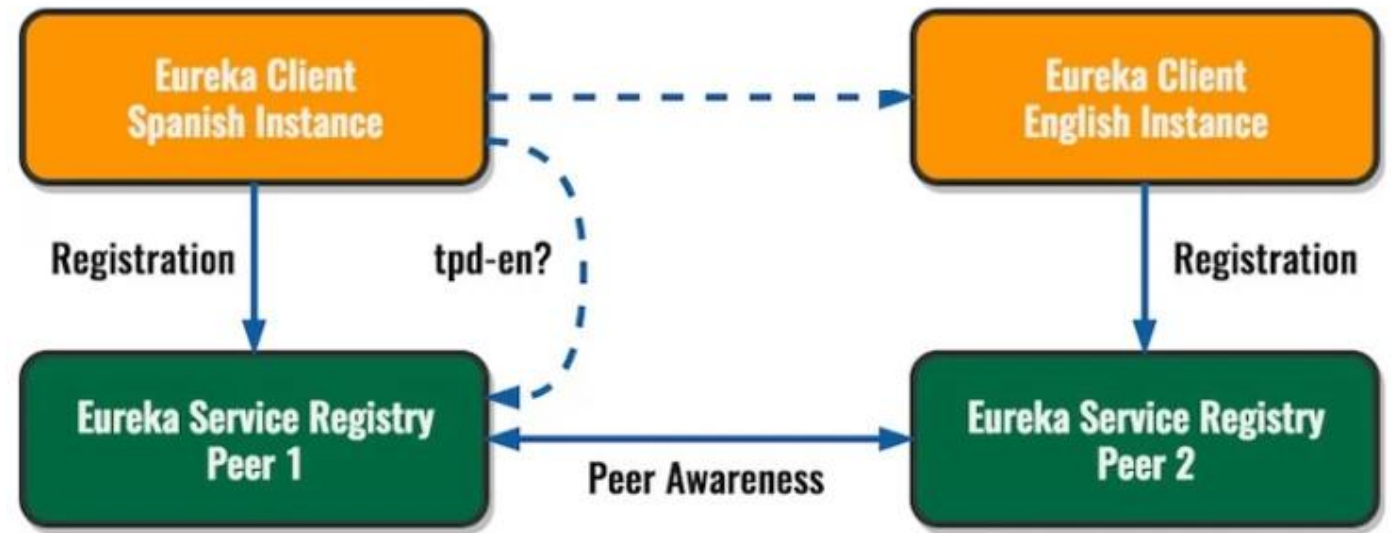
# Conectando el user Service a Eureka en microservices



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.

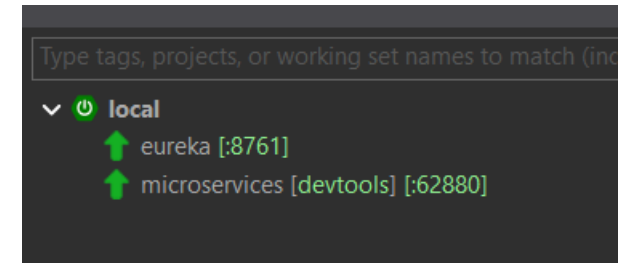
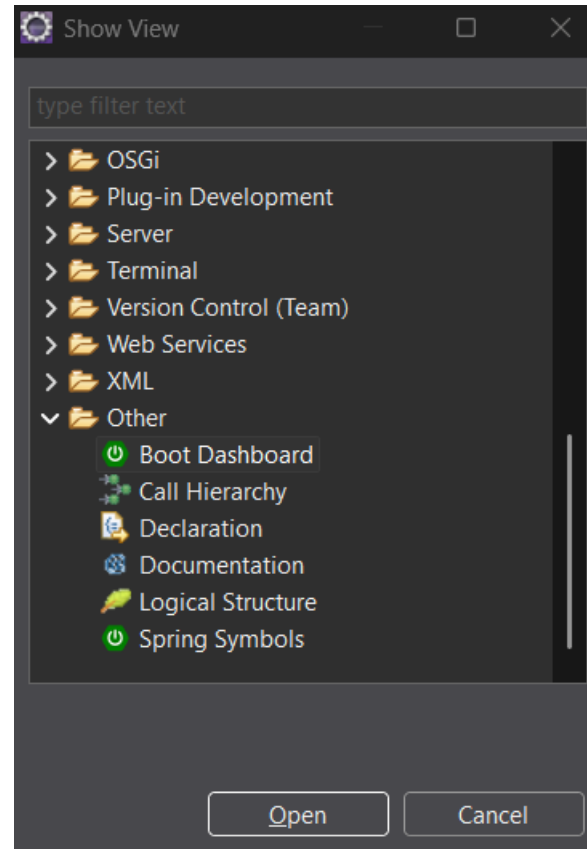





# Probando el Servicio corriendo ambos proyectos

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](http://localhost:8761)

 HOME LAST 1000 SINCE STARTUP

### System Status

Environment	test	Current time	2026-01-20T23:52:27 -0500
Data center	default	Uptime	00:06
		Lease expiration enabled	true
		Renews threshold	3
		Renews (last min)	4


### DS Replicas

[localhost](#)

### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICES	n/a (1)	(1)	UP (1) - <a href="#">microservices:eb4a2d2d229f4bf0a96f5c02a0163c04</a>

# Nuevas Instancias : Nuevo up Servicios

HOME LAST 1000 SINCE STARTUP

### System Status

Environment	test	Current time	2026-01-20T23:54:19 -0500
Data center	default	Uptime	00:08
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	4


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
MICROSERVICES	n/a (2)	(2)	UP (2) - <a href="#">microservices:eb4a2d2d229f4bf0a96f5c02a0163c04</a> , <a href="#">microservices:4c82133c817eebe887da4292d5bcab9</a>

HOME LAST 1000 SINCE STARTUP

### System Status

Environment	test	Current time	2026-01-20T23:55:33 -0500
Data center	default	Uptime	00:10
		Lease expiration enabled	true
		Renews threshold	5
		Renews (last min)	8

### DS Replicas

[localhost](#)

### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICES	n/a (2)	(2)	UP (2) - <a href="#">microservices:eb4a2d2d229f4bf0a96f5c02a0163c04</a> , <a href="#">microservices:4c82133c817eebe887da4292d5bcab9</a>

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".