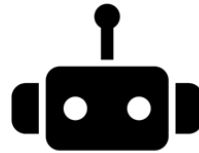




**PUCP**

## SESIÓN DE LABORATORIO 2

### Microservicios Parte 1



*HORARIO 10M1*

Empezaremos a las 7:10 p.m

Gracias!



# ¿Por qué una arquitectura debe utilizar microservicios?

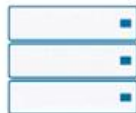


# Arquitectura de Microservicios

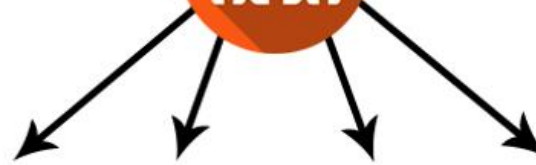
*Monolithic Architecture*



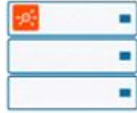
*App Services*



*Microservices Architecture*



*Microservice*



*Microservice*



*Microservice*



*Microservice*

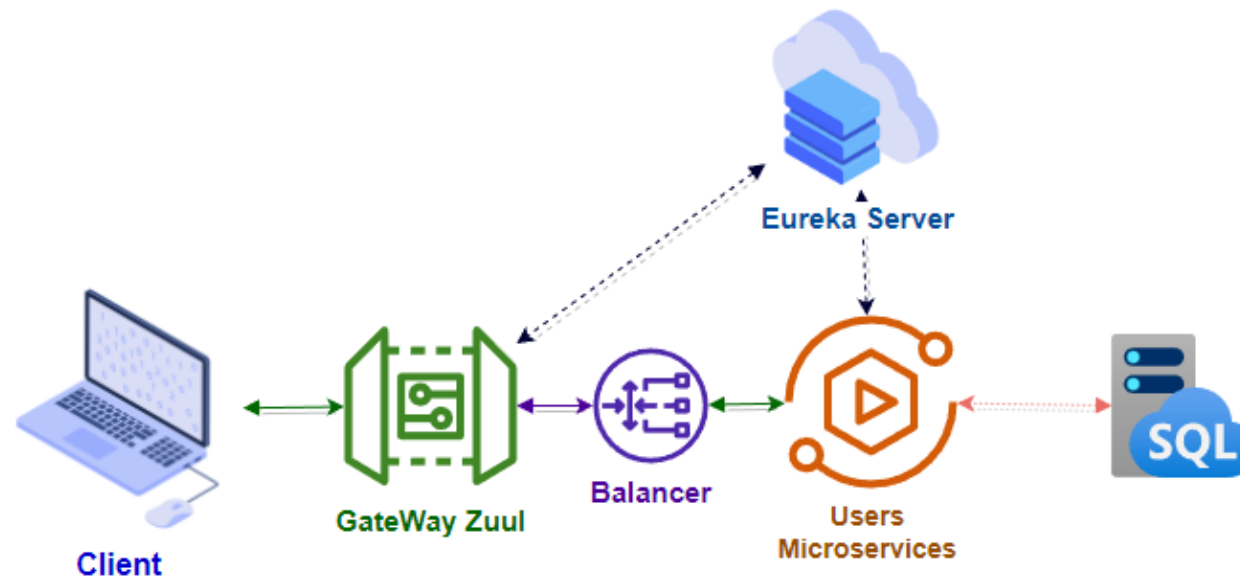


# Ejercicio

## 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

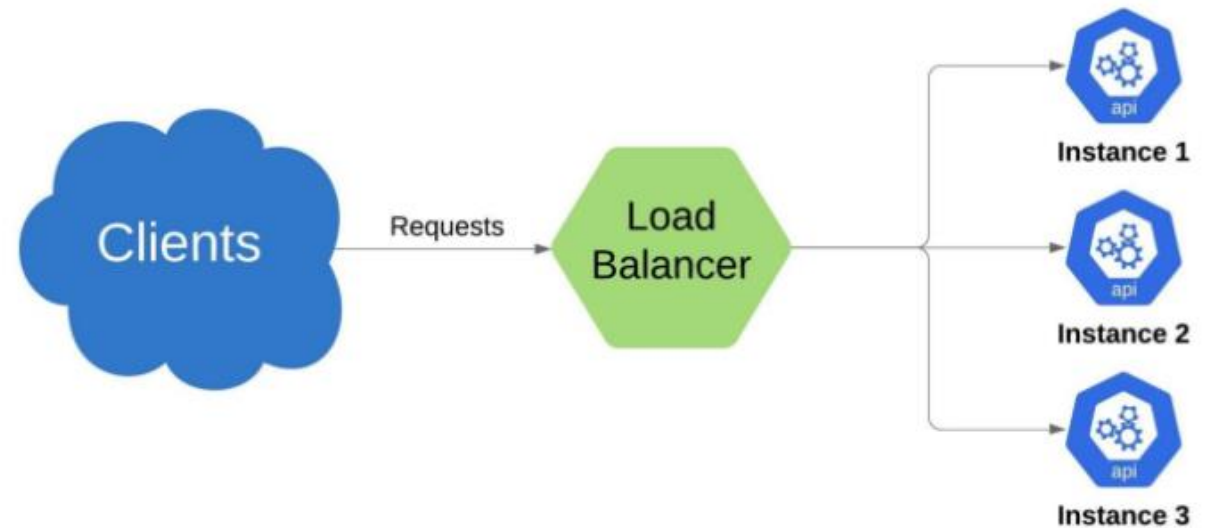
El balanceo de carga es el proceso de distribuir el tráfico de red entrante entre múltiples servidores para optimizar el uso de recursos, garantizar la disponibilidad y mejorar la capacidad de respuesta del sistema. Esta distribución puede realizarse de manera uniforme o basada en reglas específicas,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





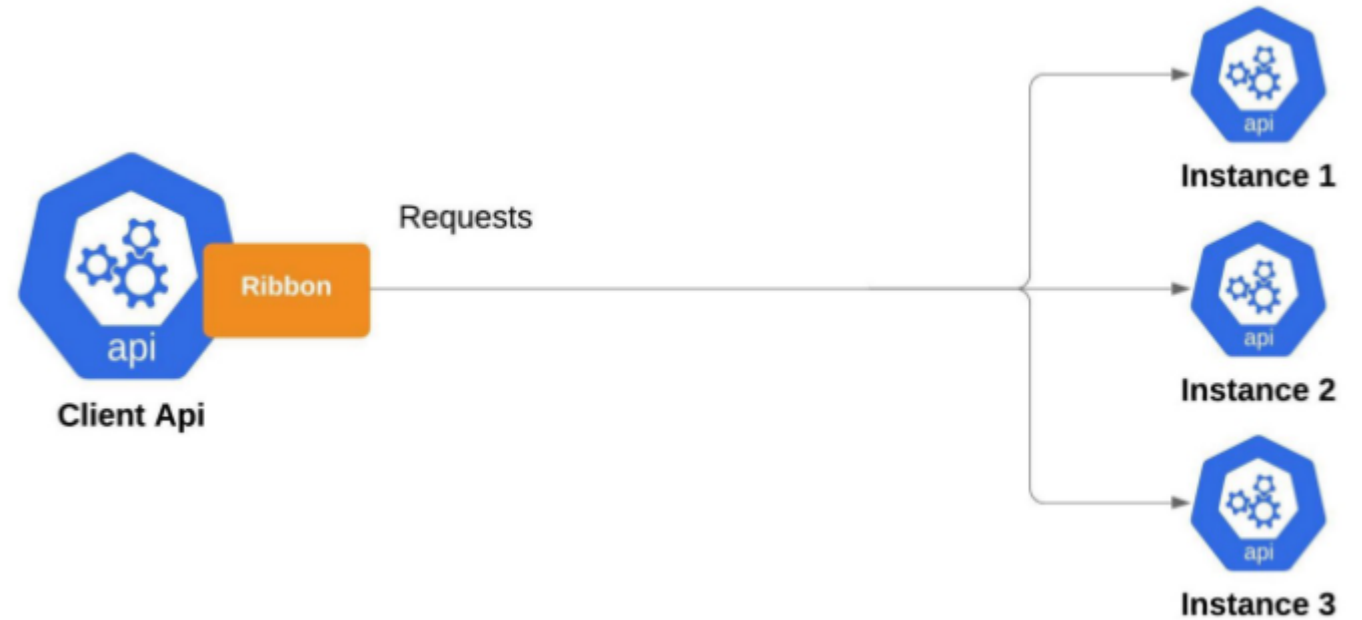
# Server Side Load balancer

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.



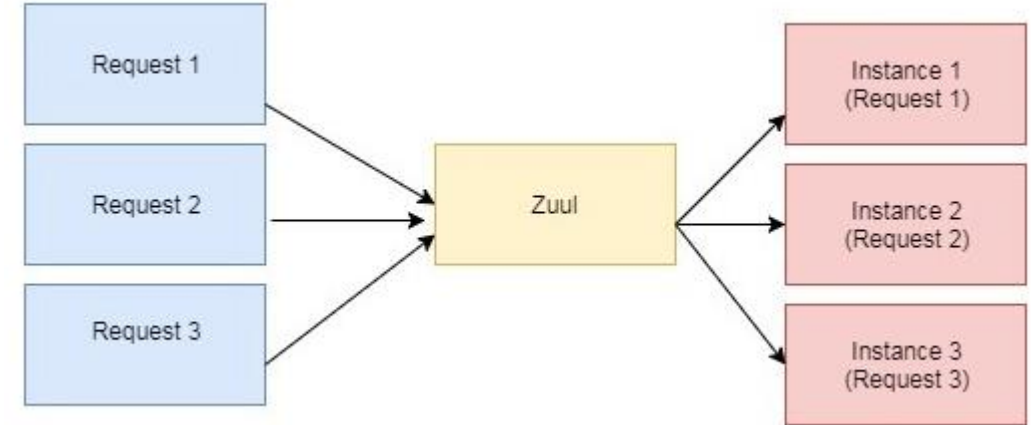
# Zuul Load Balancer

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



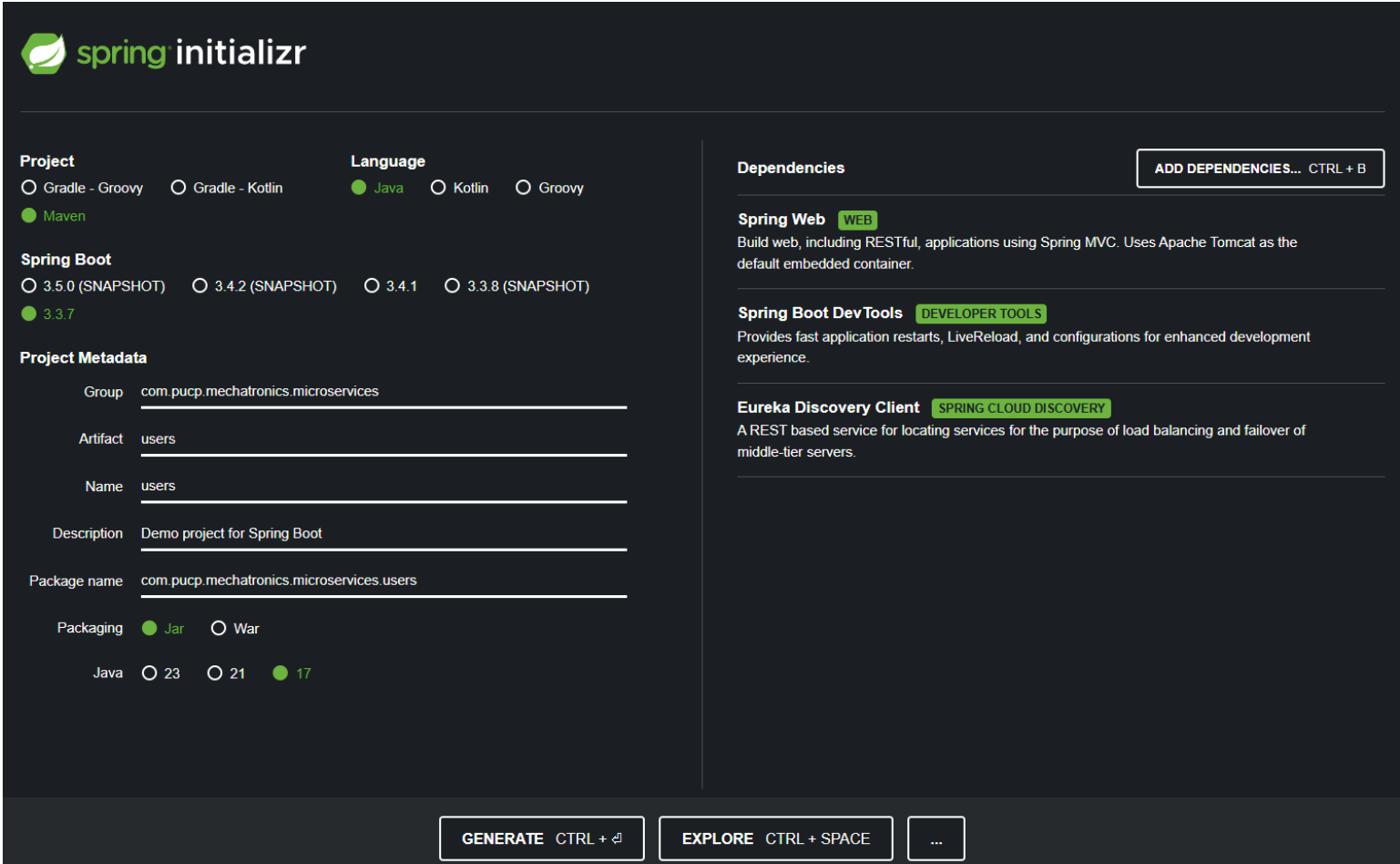
Eureka Server  
**Spring Cloud**

# Consideraciones

Vamos a continuar usando Spring Tools 4 y Java 8



# Desarrollo – Creando el microservicio

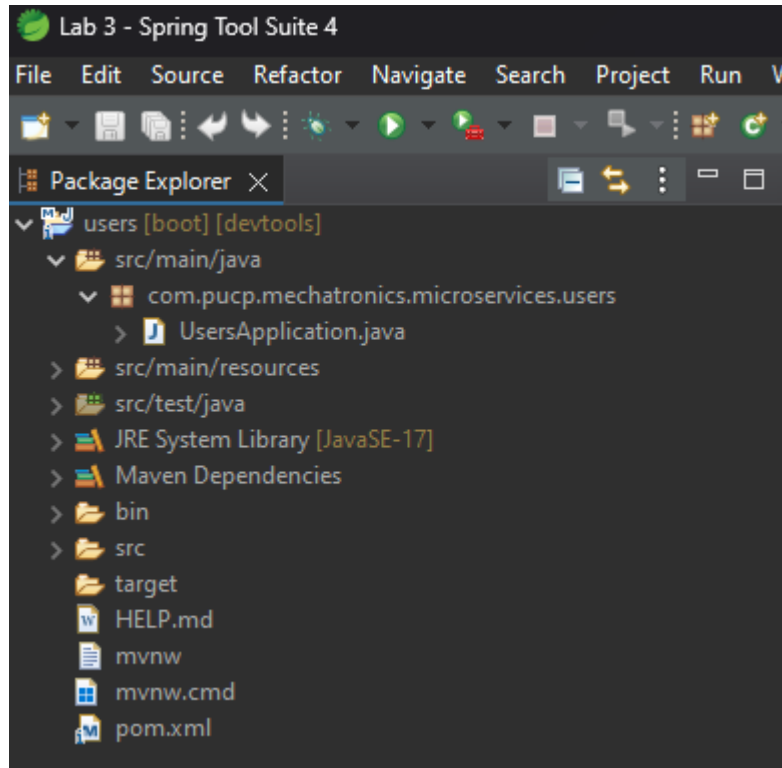


The image shows the Spring Initializr web interface, a tool for generating Spring project boilerplate. The interface is dark-themed and organized into several sections:

- Project:** Includes radio buttons for build tools (Gradle - Groovy, Gradle - Kotlin, Maven) and languages (Java, Kotlin, Groovy). Maven and Java are selected.
- Spring Boot:** Includes radio buttons for versions (3.5.0, 3.4.2, 3.4.1, 3.3.8, 3.3.7). Version 3.3.7 is selected.
- Project Metadata:** A form with fields for Group (com.pucp.mechatronics.microservices), Artifact (users), Name (users), Description (Demo project for Spring Boot), and Package name (com.pucp.mechatronics.microservices.users).
- Dependencies:** A list of dependencies with checkboxes. Selected dependencies include Spring Web, Spring Boot DevTools, and Eureka Discovery Client. A button "ADD DEPENDENCIES... CTRL + B" is present.
- Packaging:** Includes radio buttons for Jar and War. Jar is selected.
- Java:** Includes radio buttons for versions (23, 21, 17). Version 17 is selected.

At the bottom, there are three buttons: "GENERATE CTRL + G", "EXPLORE CTRL + SPACE", and a button with three dots.

# Proyecto Creado y modificando el Pom a Java 8



```
21     <developer/>
22   </developers>
23   <scm>
24     <connection/>
25     <developerConnection/>
26     <tag/>
27     <url/>
28   </scm>
29   <properties>
30     <java.version>1.8</java.version>
31     <spring-cloud.version>2023.0.5</spring-cloud.version>
32   </properties>
33   <dependencies> Add Spring Boot Starters...
34     <dependency>
35       <groupId>org.springframework.boot</groupId>
36       <artifactId>spring-boot-starter-web</artifactId>
37     </dependency>
38     <dependency>
39       <groupId>org.springframework.cloud</groupId>
40       <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
41     </dependency>
42
43     <dependency>
44       <groupId>org.springframework.boot</groupId>
45       <artifactId>spring-boot-devtools</artifactId>
46       <scope>runtime</scope>
47       <optional>true</optional>
48     </dependency>
```

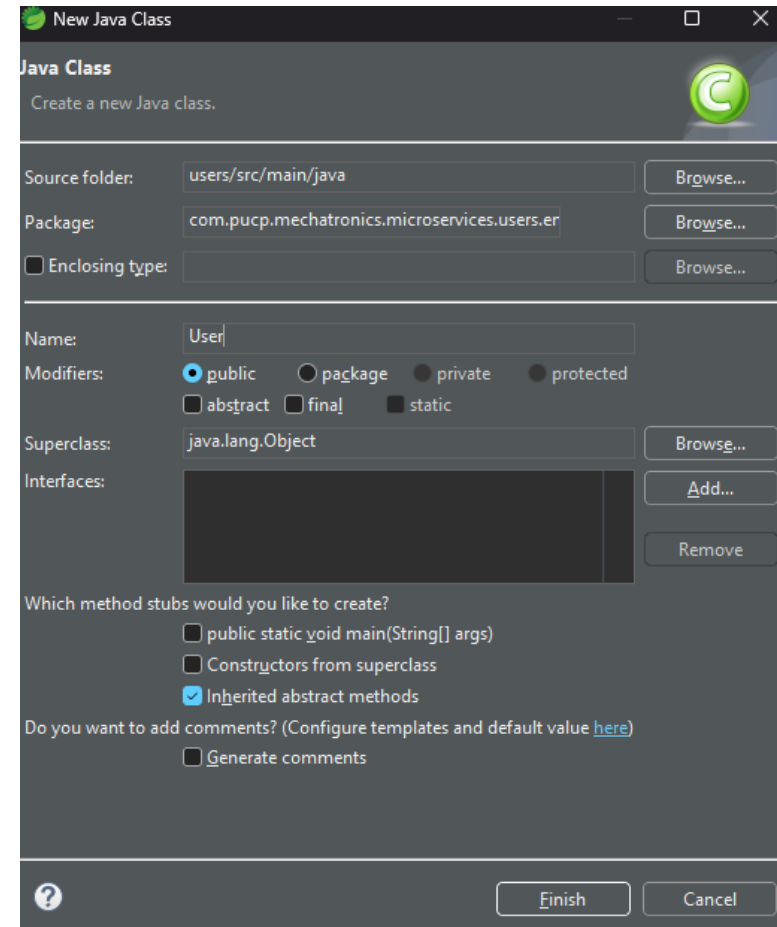
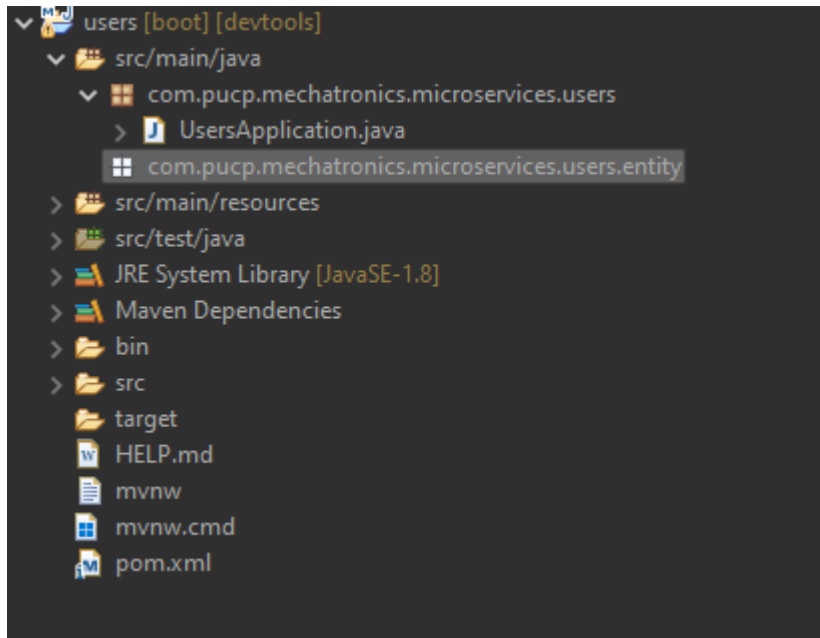


# La Capa entidad

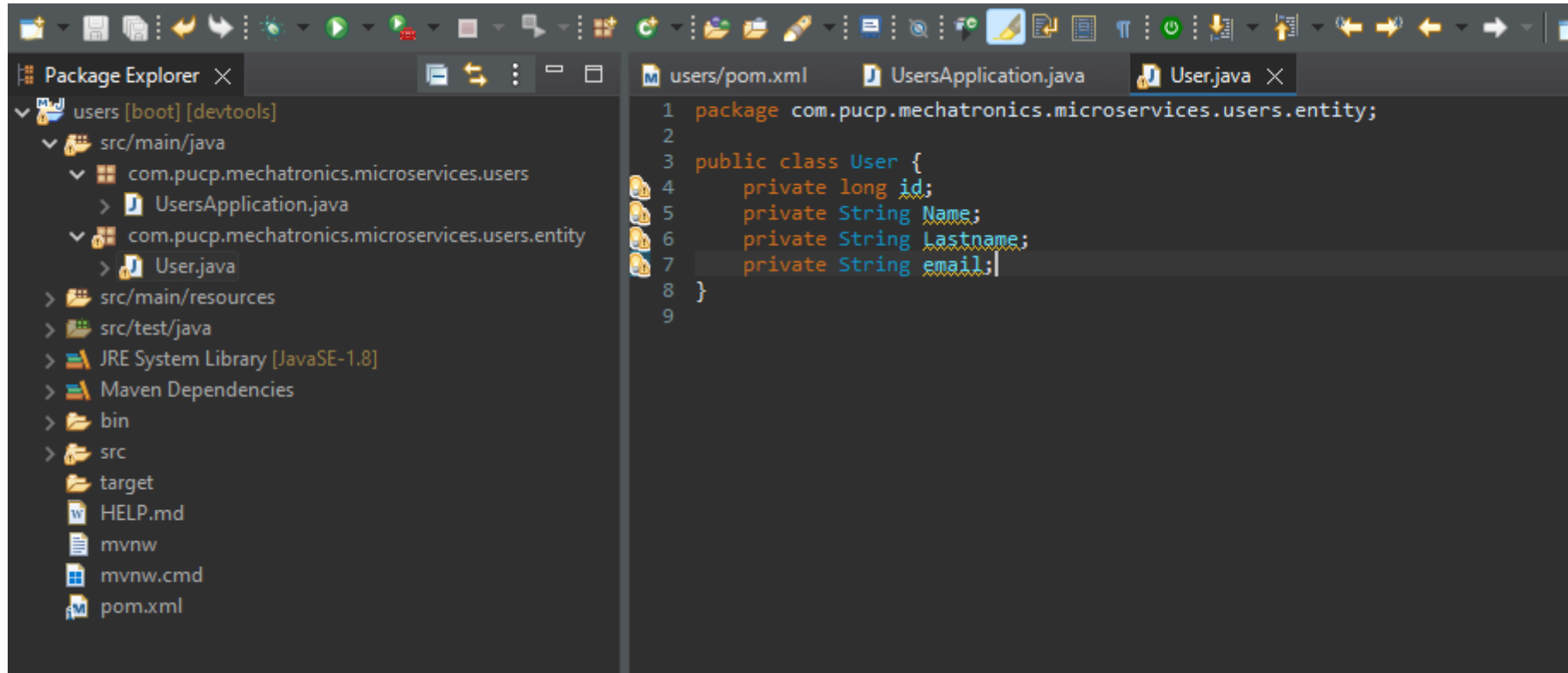
También conocida como **capa de modelo** o **modelo de dominio**, se encarga de representar los objetos y datos que reflejan las entidades del negocio. Estas clases suelen estar directamente relacionadas con las tablas de una base de datos en sistemas relacionales, o con documentos en bases de datos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



# Implementando la capa de entidad



# Clase User



The screenshot shows an IDE interface with a dark theme. On the left, the 'Package Explorer' displays a project structure for 'users [boot] [devtools]'. The 'src/main/java' directory is expanded, showing the package 'com.pucp.mechatronics.microservices.users'. Inside this package, there is a sub-package 'com.pucp.mechatronics.microservices.users.entity' which contains the 'User.java' file. The main editor on the right shows the code for 'User.java'. The code defines a package, imports a class, and declares a public class 'User' with three private attributes: 'id' (long), 'Name' (String), and 'email' (String).

```
1 package com.pucp.mechatronics.microservices.users.entity;
2
3 public class User {
4     private long id;
5     private String Name;
6     private String Lastname;
7     private String email;
8 }
9
```

# Clase User

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

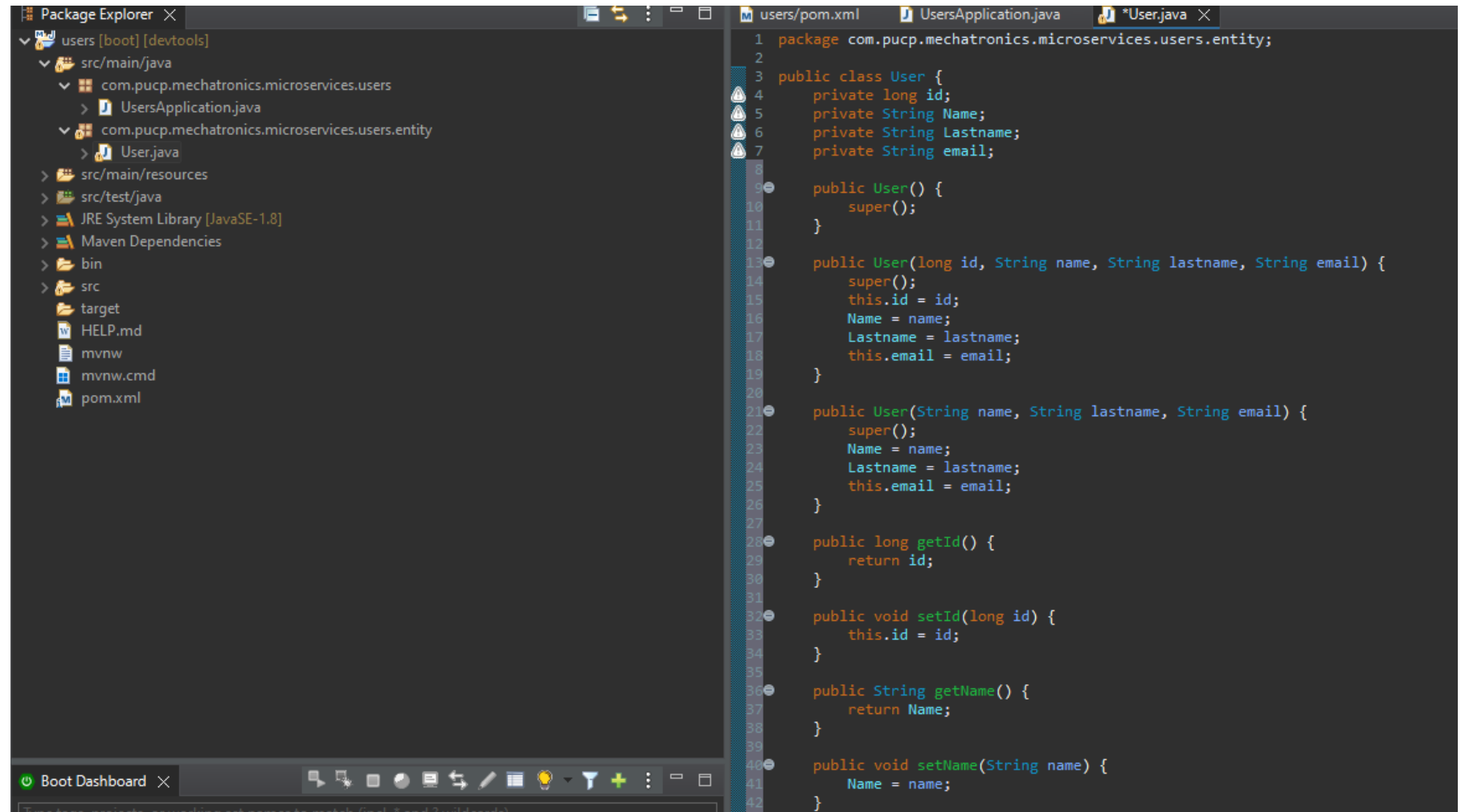
☐ Generate method comments

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 shows an IDE with the Package Explorer on the left and the code editor on the right. The Package Explorer displays the project structure for 'users [boot] [devtools]', including packages like 'com.pucp.mechatronics.microservices.users' and 'com.pucp.mechatronics.microservices.users.entity', and files like 'User.java'. The code editor shows the 'User.java' file with the following code:

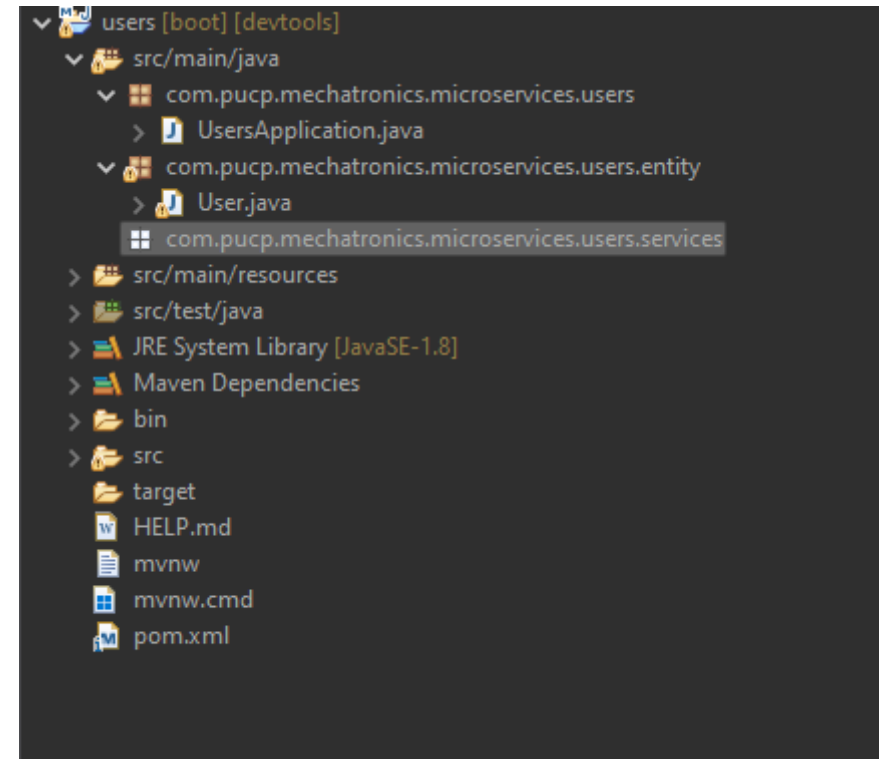
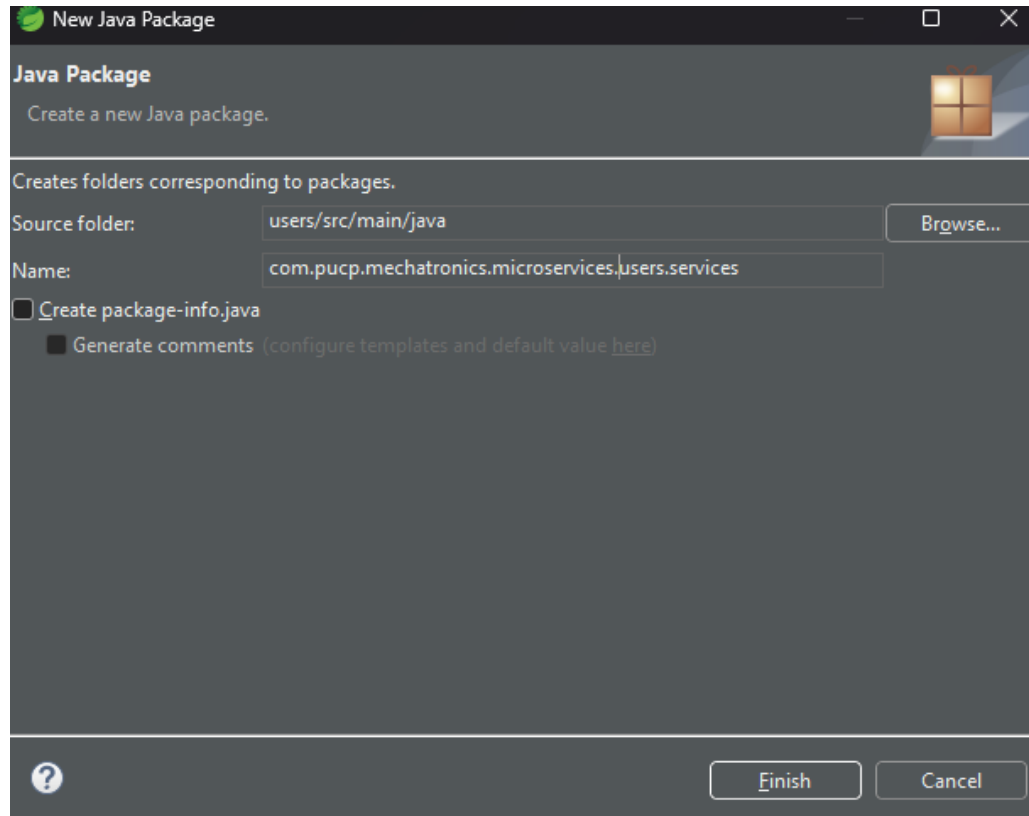
```
1 package com.pucp.mechatronics.microservices.users.entity;
2
3 public class User {
4     private long id;
5     private String Name;
6     private String Lastname;
7     private String email;
8
9     public User() {
10         super();
11     }
12
13     public User(long id, String name, String lastname, String email) {
14         super();
15         this.id = id;
16         Name = name;
17         Lastname = lastname;
18         this.email = email;
19     }
20
21     public User(String name, String lastname, String email) {
22         super();
23         Name = name;
24         Lastname = lastname;
25         this.email = email;
26     }
27
28     public long getId() {
29         return id;
30     }
31
32     public void setId(long id) {
33         this.id = id;
34     }
35
36     public String getName() {
37         return Name;
38     }
39
40     public void setName(String name) {
41         Name = name;
42     }
43 }
```

# La Capa de Servicio

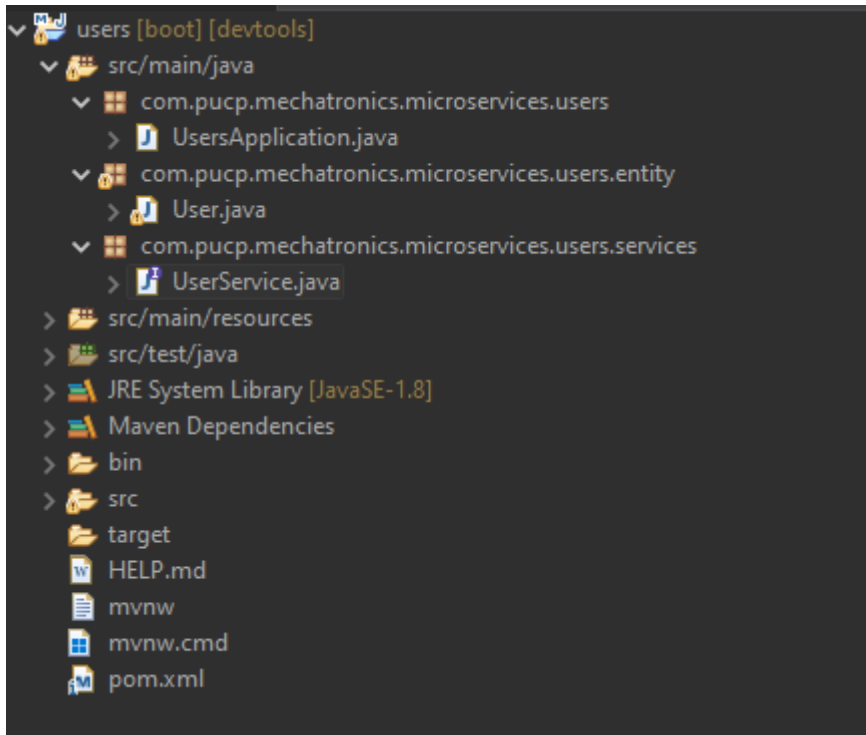
Una capa de servicio es una capa de una aplicación que facilita la comunicación entre el controlador y las demás partes del microservicio. Además, suelen servir de frontera de transacciones y se encargan de autorizarlas.



# Implementando la capa de servicio



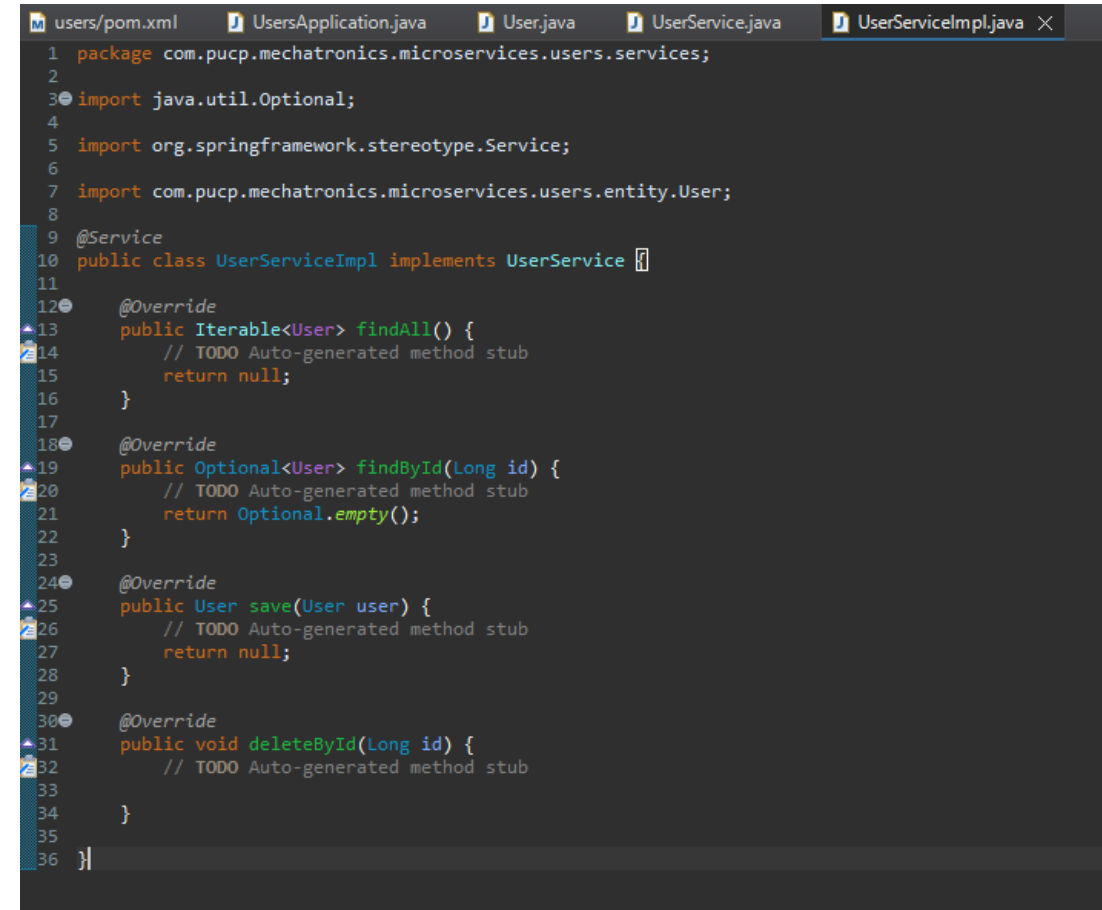
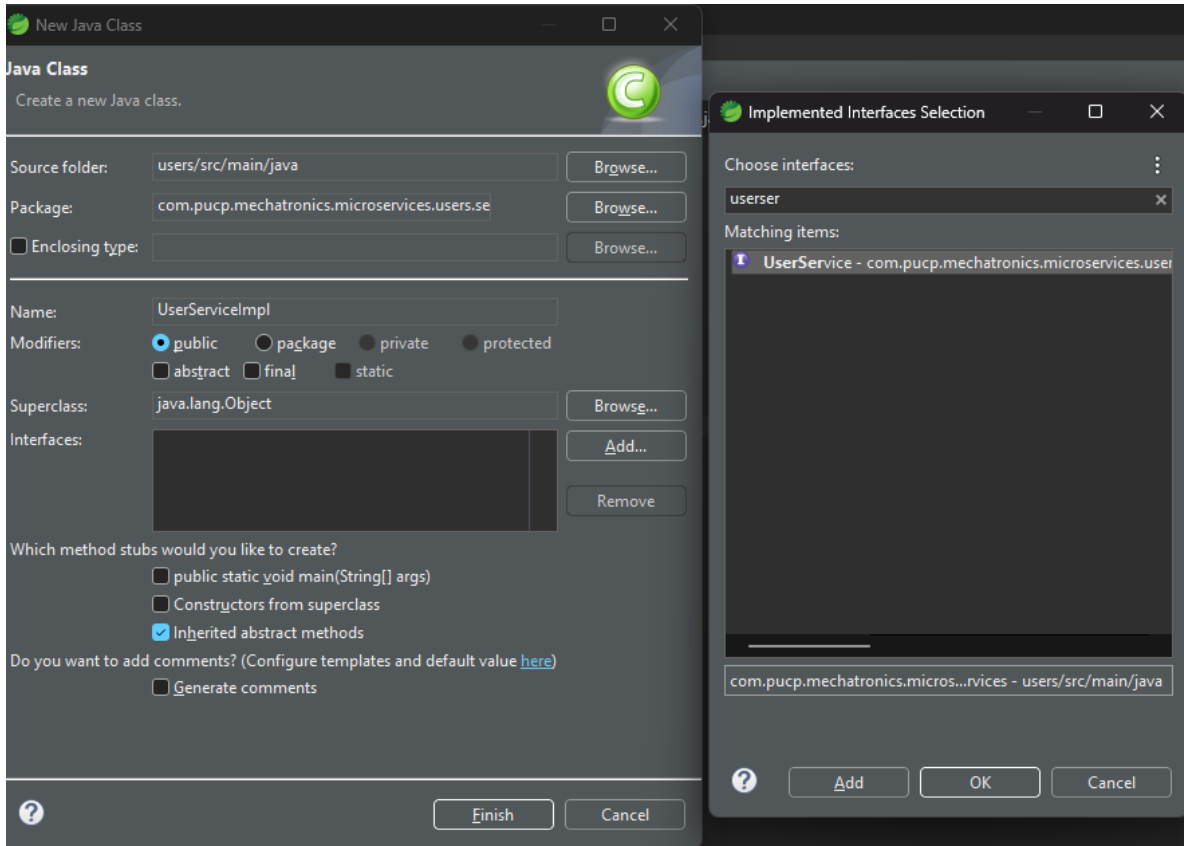
# Interfaz UserService



```
users/pom.xml  UsersApplication.java  *User.java  *UserService.java X
1 package com.pucp.mechatronics.microservices.users.services;
2
3 import java.util.Optional;
4
5 import com.pucp.mechatronics.microservices.users.entity.User;
6
7 public interface UserService {
8
9     public Iterable<User> findAll();
10    public Optional<User> findById(Long id);
11    public User save (User user);
12    public void deleteById (Long id);
13
14 }
15
```

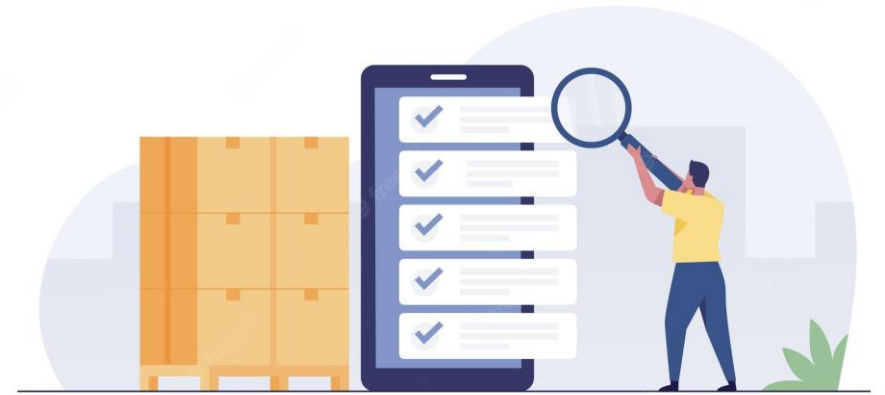


# Clase UserServiceImpl (@Service)

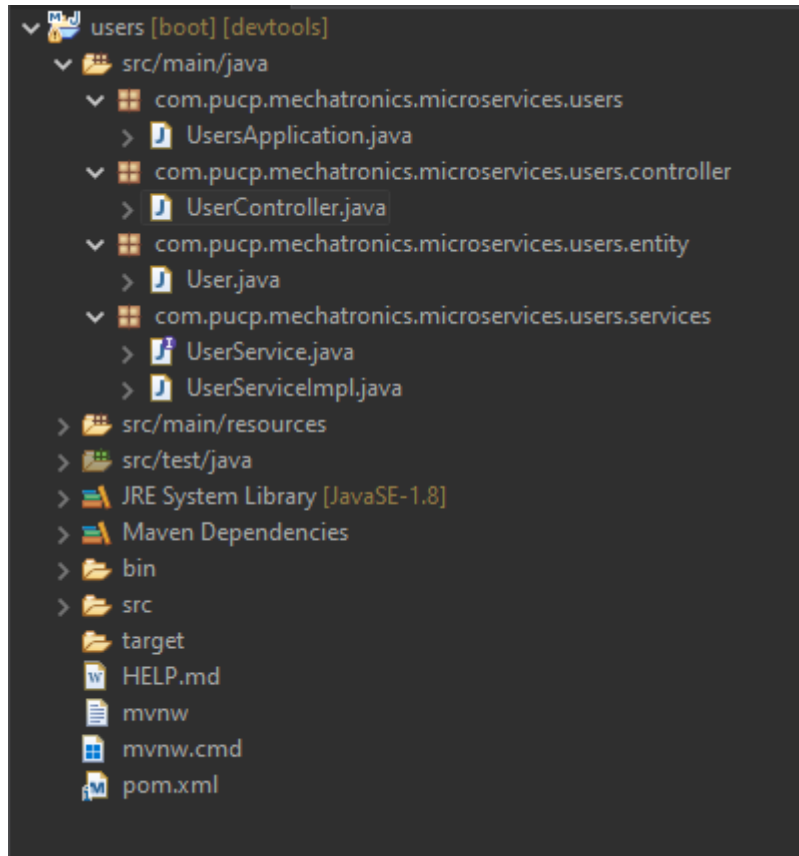


# La Capa Controlador

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.

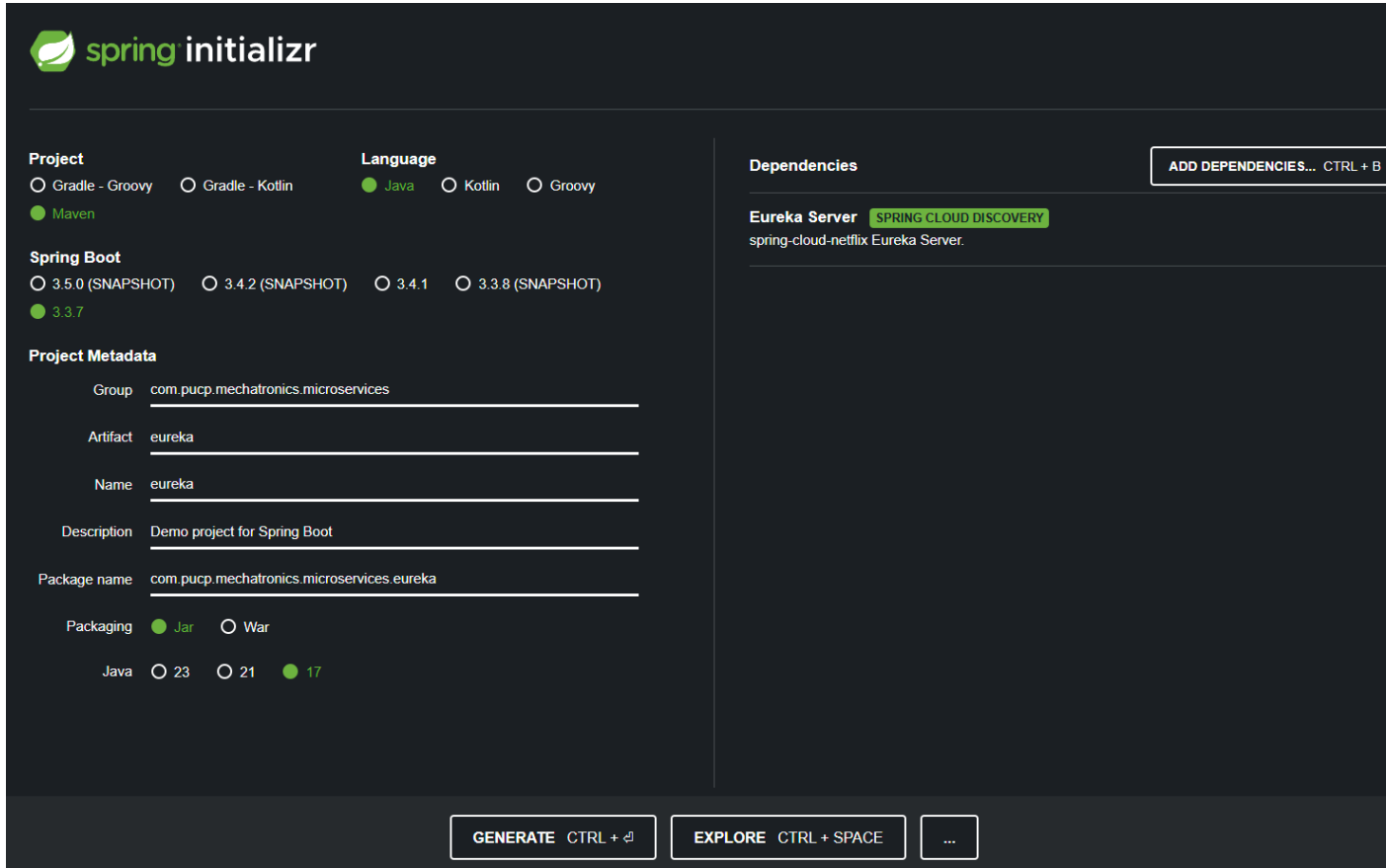


# Paquete Controller y Clase UserController



```
1 package com.pucp.mechatronics.microservices.users.controller;
2
3 import java.util.Optional;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.http.HttpStatus;
7 import org.springframework.http.ResponseEntity;
8 import org.springframework.web.bind.annotation.DeleteMapping;
9 import org.springframework.web.bind.annotation.GetMapping;
10 import org.springframework.web.bind.annotation.PathVariable;
11 import org.springframework.web.bind.annotation.PostMapping;
12 import org.springframework.web.bind.annotation.PutMapping;
13 import org.springframework.web.bind.annotation.RequestBody;
14 import org.springframework.web.bind.annotation.RestController;
15
16 import com.pucp.mechatronics.microservices.users.entity.User;
17 import com.pucp.mechatronics.microservices.users.services.UserService;
18
19 @RestController
20
21 public class UserController {
22     @Autowired UserService service;
23
24     @GetMapping
25     public ResponseEntity <?> list(){
26         return ResponseEntity.ok().body(service.findAll());
27     }
28
29     @GetMapping("/{id}")
30     public ResponseEntity <?> see(@PathVariable Long id){
31         Optional <User> o = service.findById(id);
32         if (o != null && !o.isPresent() || o == null) {
33             return ResponseEntity.notFound().build();
34         }
35         return ResponseEntity.ok().body(o);
36     }
37
38     @PostMapping
39     public ResponseEntity<?> create(@RequestBody User user){
40         User userDb = service.save(user);
41         return ResponseEntity.status(HttpStatus.CREATED).body(userDb);
42     }
43
44     @PutMapping
```

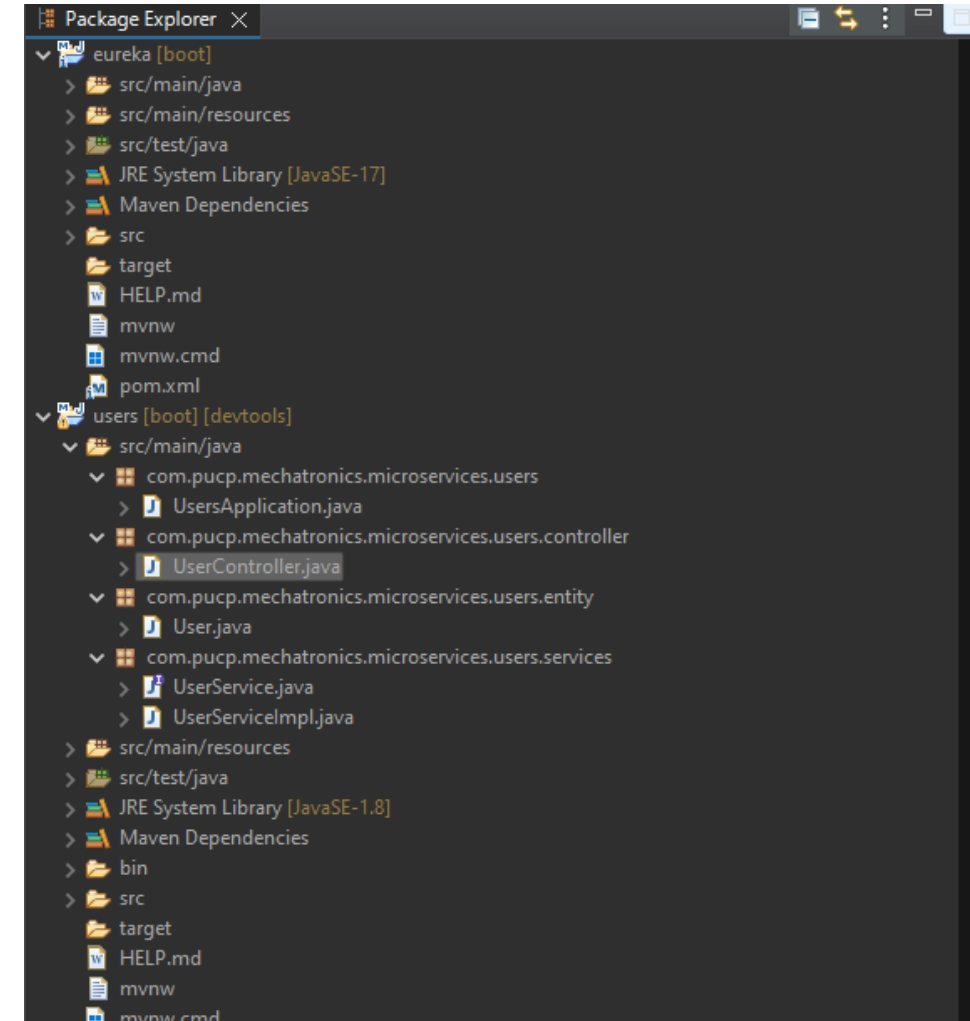
# Creando el servidor eureka



The Spring Initializr interface shows the configuration for a new project. The 'Project' section has 'Maven' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '3.3.7' selected. The 'Project Metadata' section shows the following details:

- Group: com.pucp.mechatronics.microservices
- Artifact: eureka
- Name: eureka
- Description: Demo project for Spring Boot
- Package name: com.pucp.mechatronics.microservices.eureka
- Packaging: Jar
- Java: 17

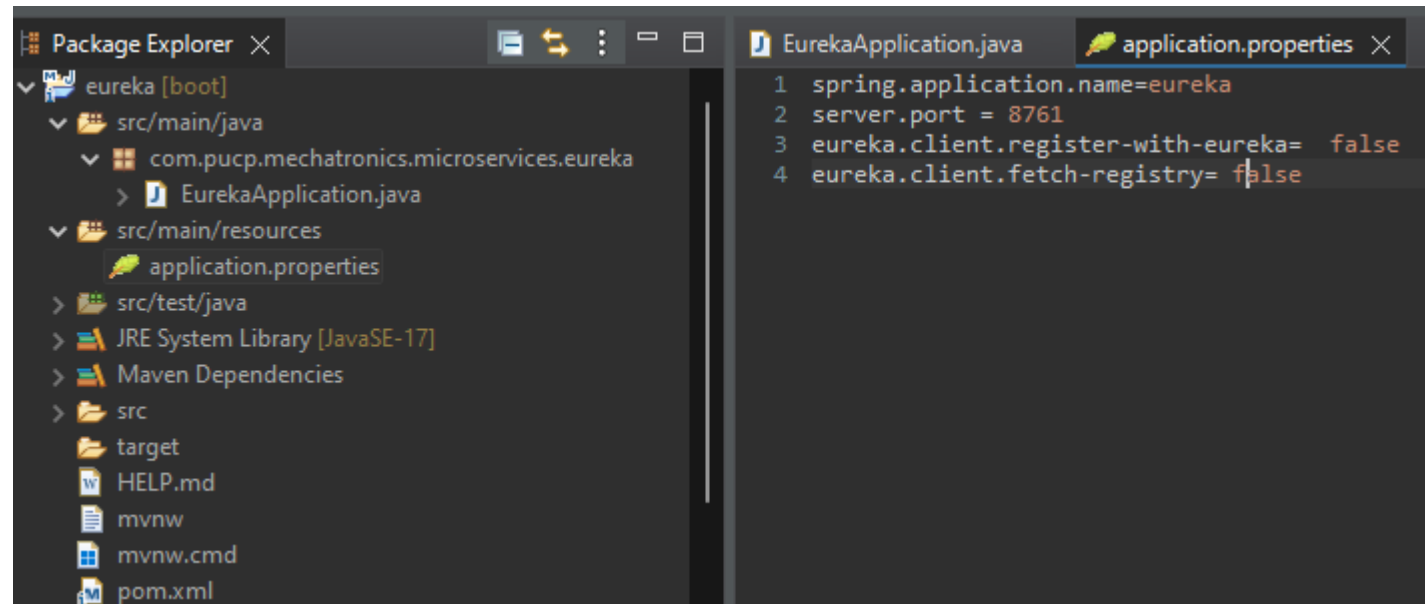
The 'Dependencies' section shows 'Eureka Server' with the dependency 'spring-cloud-netflix Eureka Server'. The 'ADD DEPENDENCIES... CTRL + B' button is visible. At the bottom, there are buttons for 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and a menu icon.



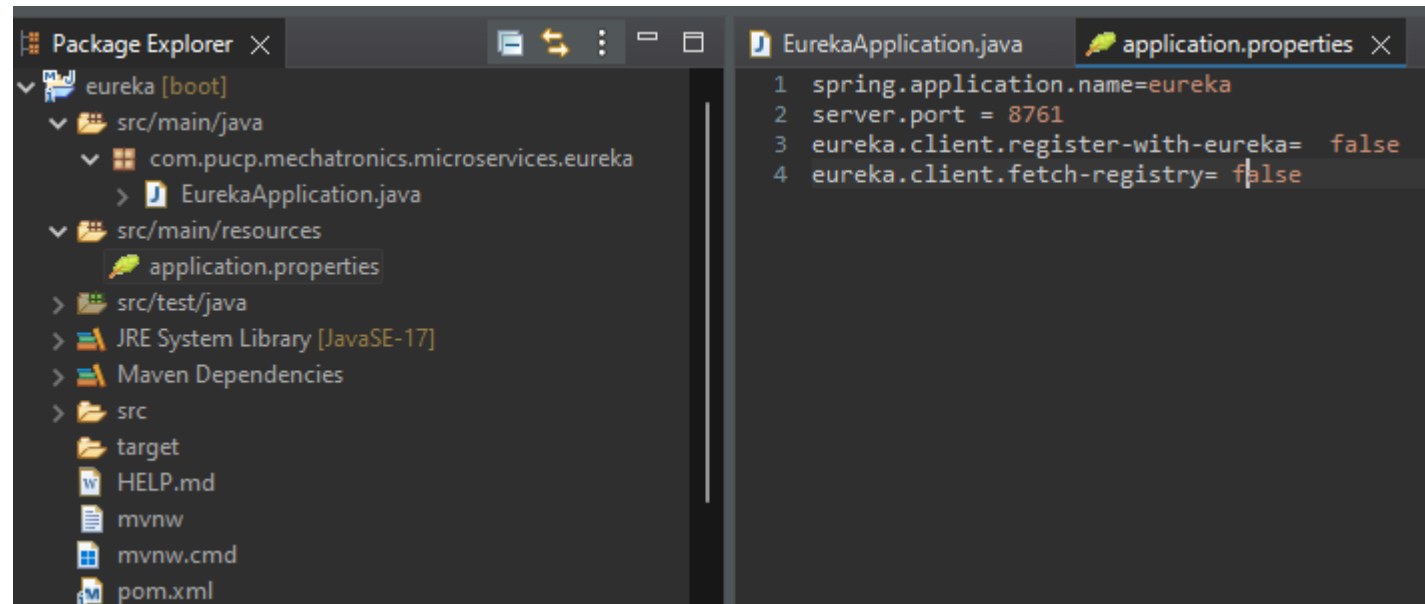
# Habilitando el Servidor

```
EurekaApplication.java X
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
```

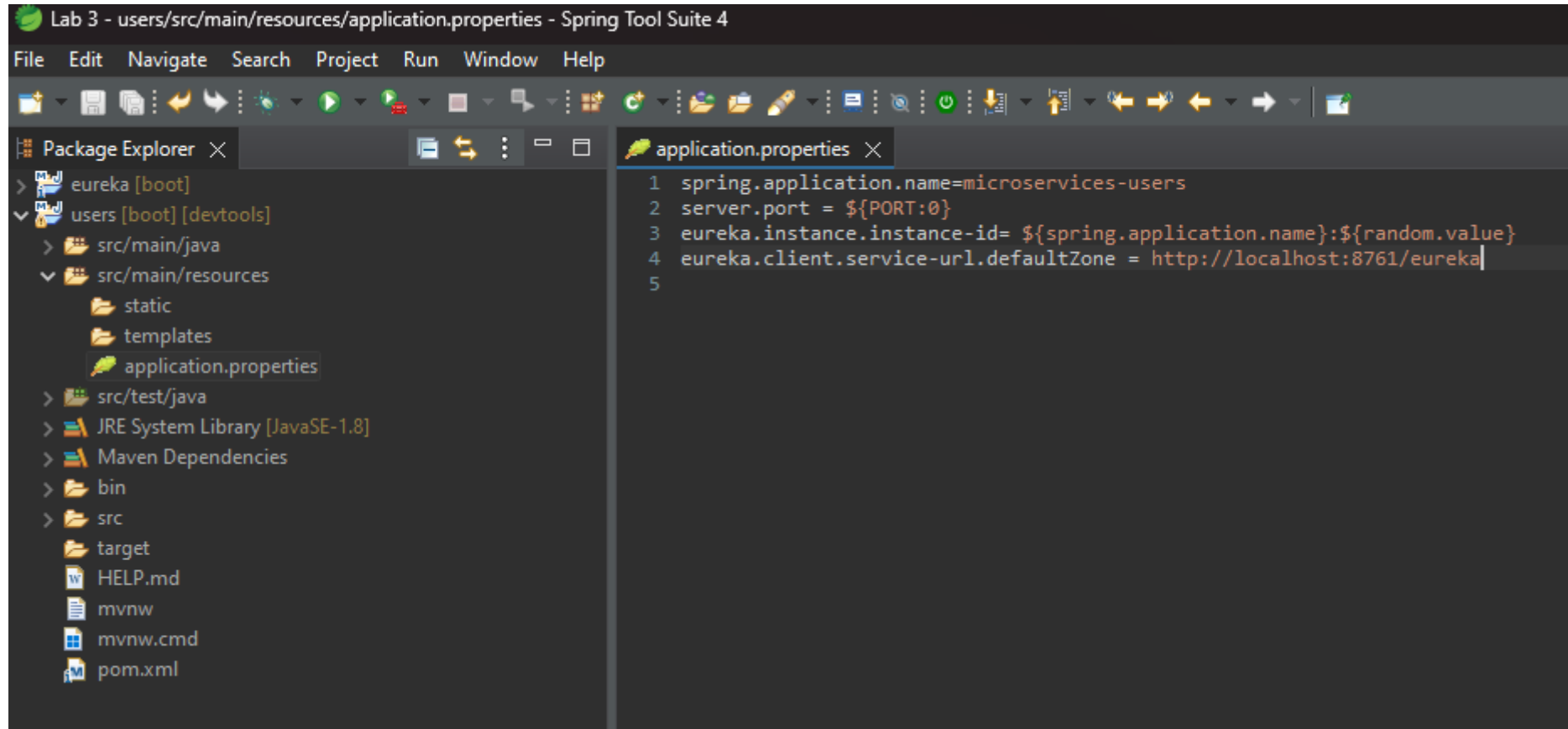
# Editando las propiedades



# Editando las propiedades



# Conectando el user Service a Eureka



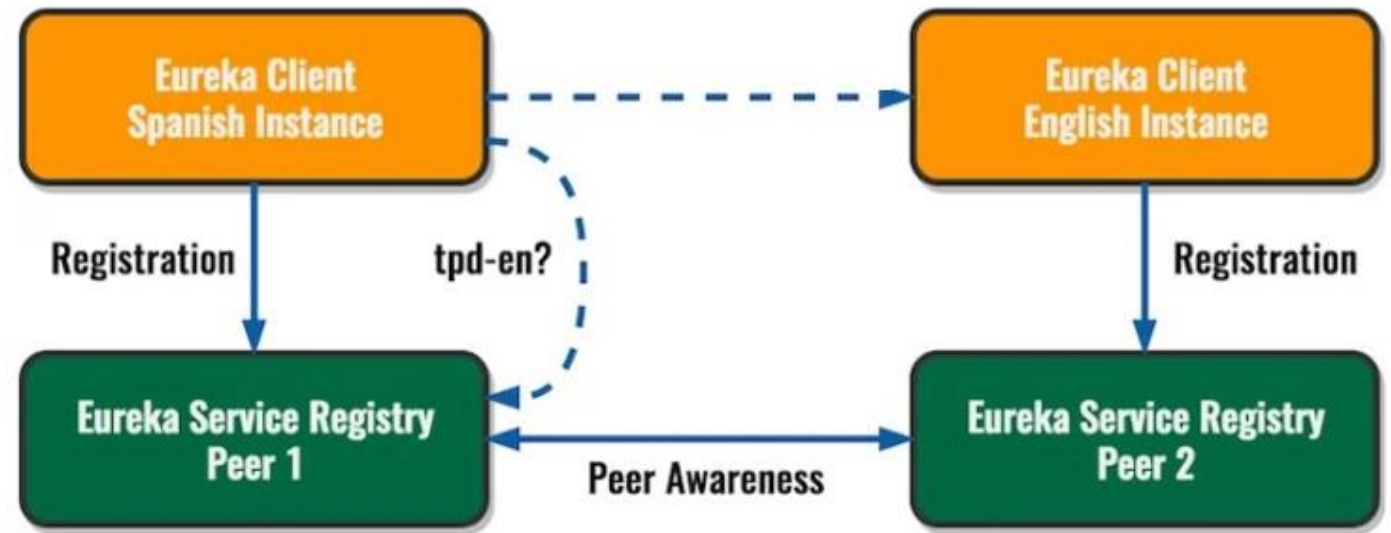
The screenshot shows the Spring Tool Suite 4 IDE. The Package Explorer on the left displays the project structure: eureka [boot], users [boot] [devtools], src/main/java, src/main/resources (containing static, templates, and application.properties), src/test/java, JRE System Library [JavaSE-1.8], Maven Dependencies, bin, src, target, HELP.md, mvnw, mvnw.cmd, and pom.xml. The application.properties file is open in the editor, showing the following configuration:

```
1 spring.application.name=microservices-users
2 server.port = ${PORT:0}
3 eureka.instance.instance-id= ${spring.application.name}:${random.value}
4 eureka.client.service-url.defaultZone = http://localhost:8761/eureka
5
```



# Eureka Peer Awareness (/Eureka)

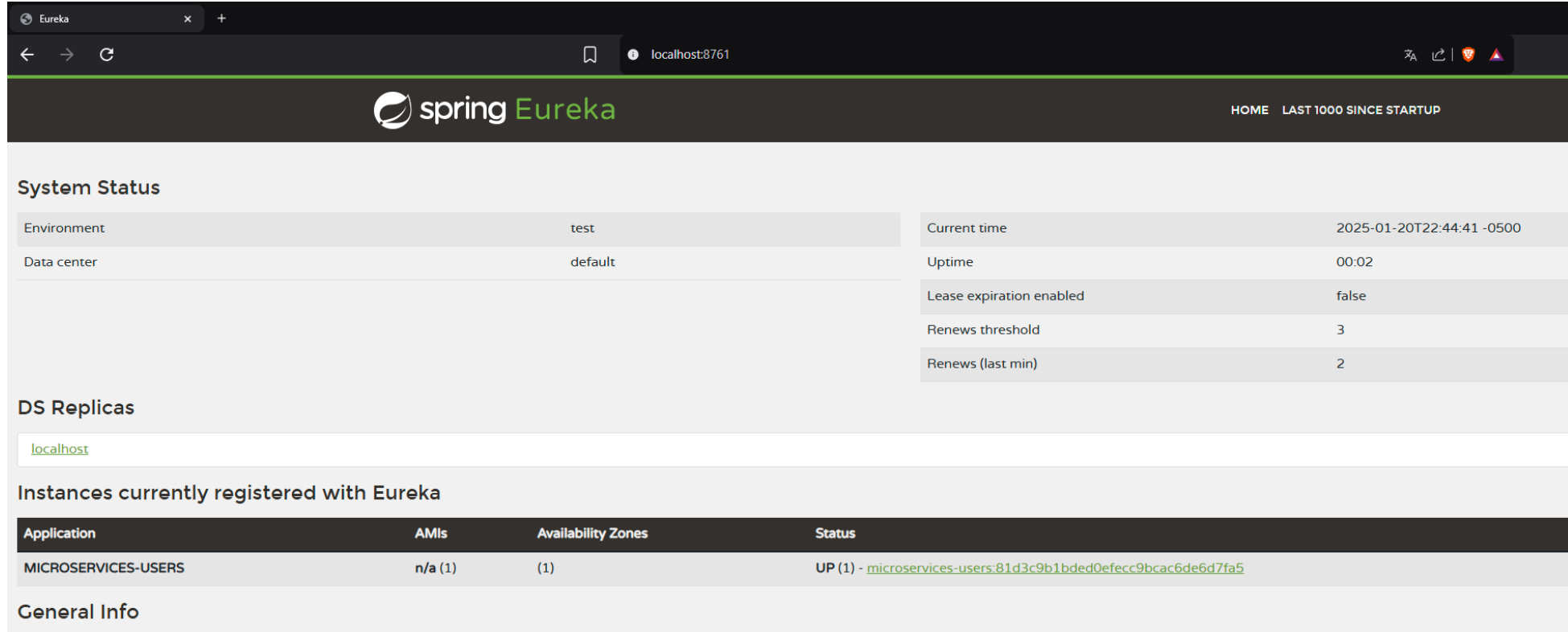
Nos permite tener dos instancias de registro que pueden compartir entre sí el contenido del registro (los clientes registrados) para que puedan implementar la resiliencia y esos clientes puedan encontrarse entre sí poniéndose en contacto con la instancia que conocen.



# Probando el Servicio corriendo ambos proyectos

The screenshot displays the Eclipse IDE interface. On the left, the Package Explorer shows two projects: 'eureka [boot]' and 'users [boot] [devtools]'. Below it, the Boot Dashboard shows the status of the running applications: 'eureka [8761]' and 'users [devtools] [51388]'. The main console window shows the startup logs for the 'microservices-users' application (v3.3.7). The logs include a series of 'restartedMain' messages and 'INFO 38600' status updates, indicating the application is running successfully.

# Visualizando los servicios : Http://localhost:8761



The screenshot shows the Spring Eureka web interface in a browser window. The address bar shows 'localhost:8761'. The page title is 'spring Eureka'. The navigation bar includes 'HOME' and 'LAST 1000 SINCE STARTUP'.

### System Status

Environment	test	Current time	2025-01-20T22:44:41 -0500
Data center	default	Uptime	00:02
		Lease expiration enabled	false
		Renews threshold	3
		Renews (last min)	2

### DS Replicas


[localhost](#)

### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICES-USERS	n/a (1)	(1)	UP (1) - <a href="#">microservices-users.81d3c9b1bde0efecc9bcac6de6d7fa5</a>

### General Info

# Nuevas Instancias : Nuevo up Servicios

HOME LAST 1000 SINCE S

### System Status

Environment	test	Current time	2025-01
Data center	default	Uptime	00:03
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	4

### DS Replicas

[localhost](#)

### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICROSERVICES-USERS	n/a (2)	(2)	UP (2) - <a href="#">microservices-users:81d3c9b1bded0efecc9bcac6de6d7fa5</a> , <a href="#">microservices-users:451d6fcf33b7c426d948615d9a6ef46f</a>

### General Info