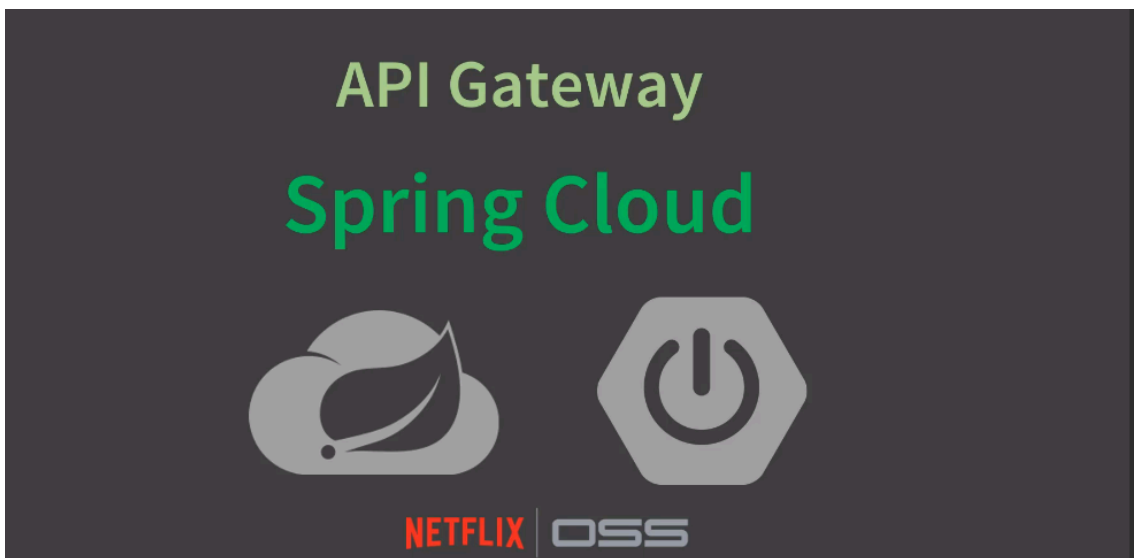


**Samuel Santiago Falla Alfaro - 90180**

## CREACIÓN DE PUERTA DE ENLACE GATEWAY

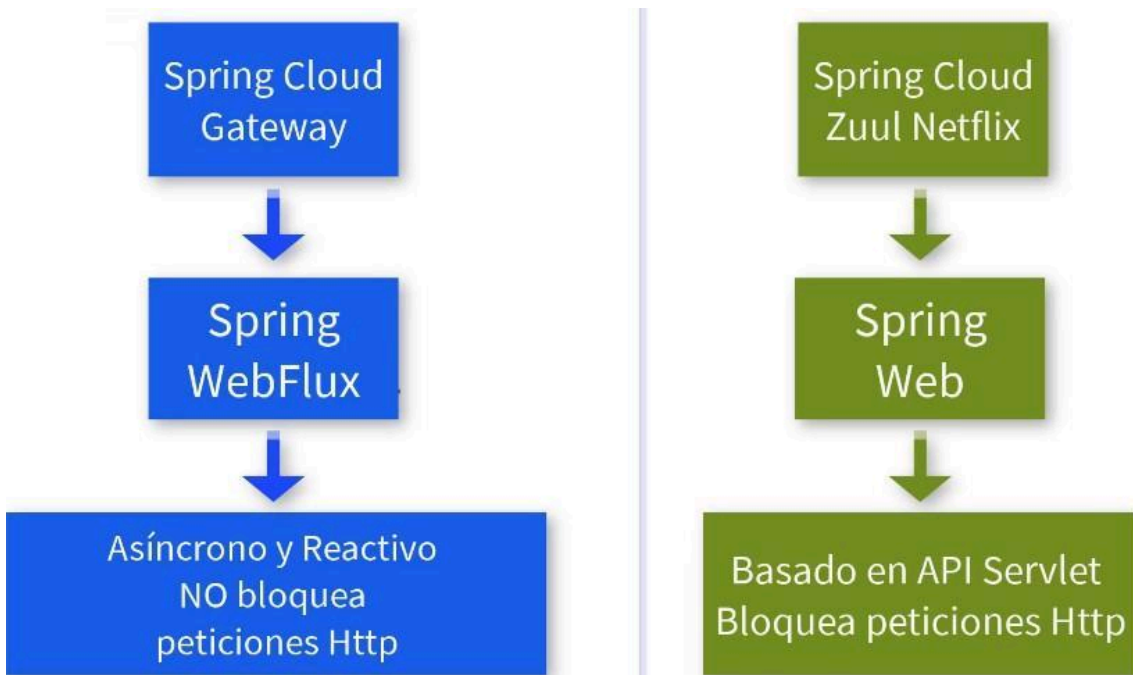
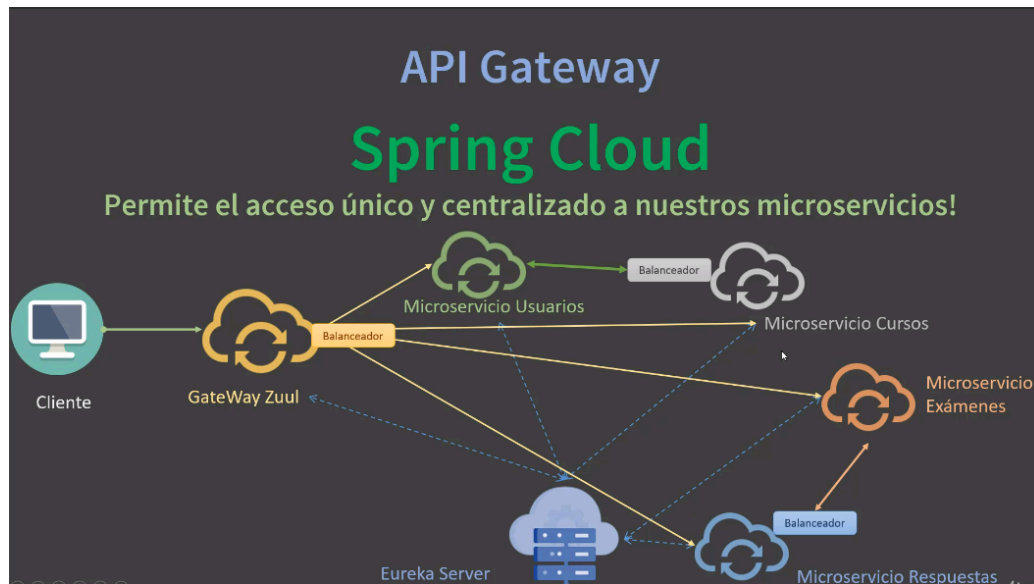


### 1.1 Buscar y poner en esta parte el concepto de Spring Cloud Gateway

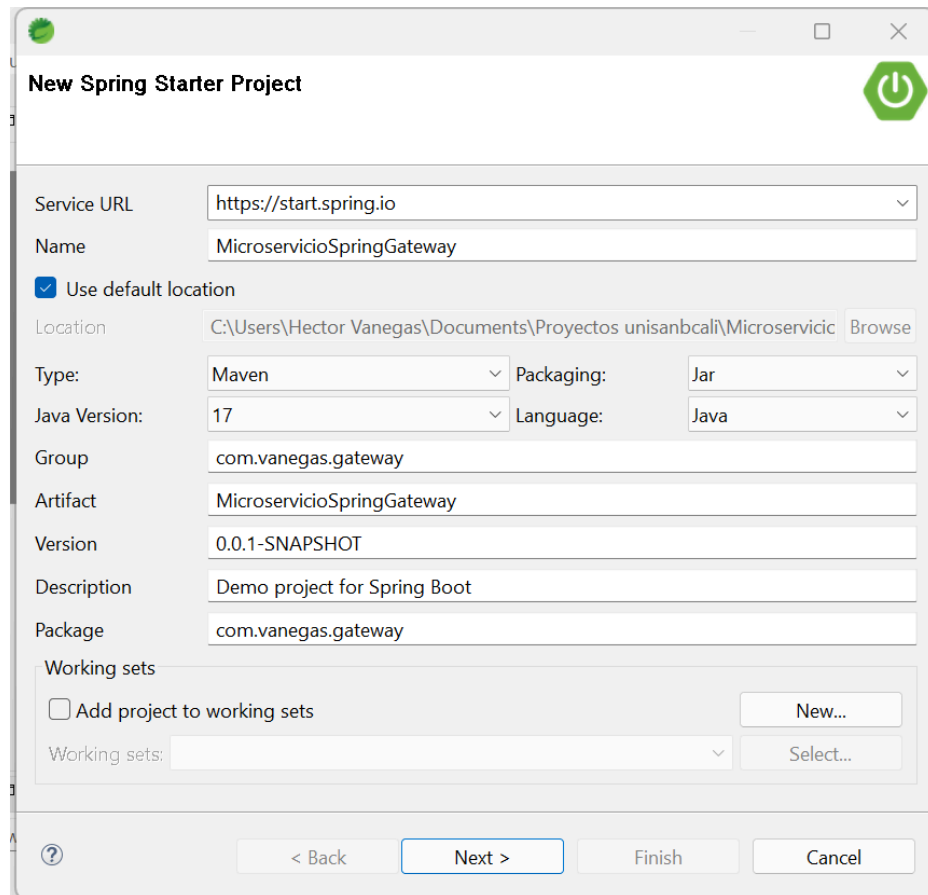
API Gateway

Spring Cloud

- Zuul Netflix y Spring Cloud Gateway
- Puerta de enlace, acceso centralizado
- Enrutamiento dinámico de los microservicios
- Balanceo de carga
- Maneja filtros propios
- Permite extender funcionalidades



1.2 Se debe crear un nuevo servicio que servirá como apigateway



**New Spring Starter Project**

Service URL:

Name:

☒ Use default location

Location:

Type:  Packaging:

Java Version:  Language:

Group:

Artifact:

Version:

Description:

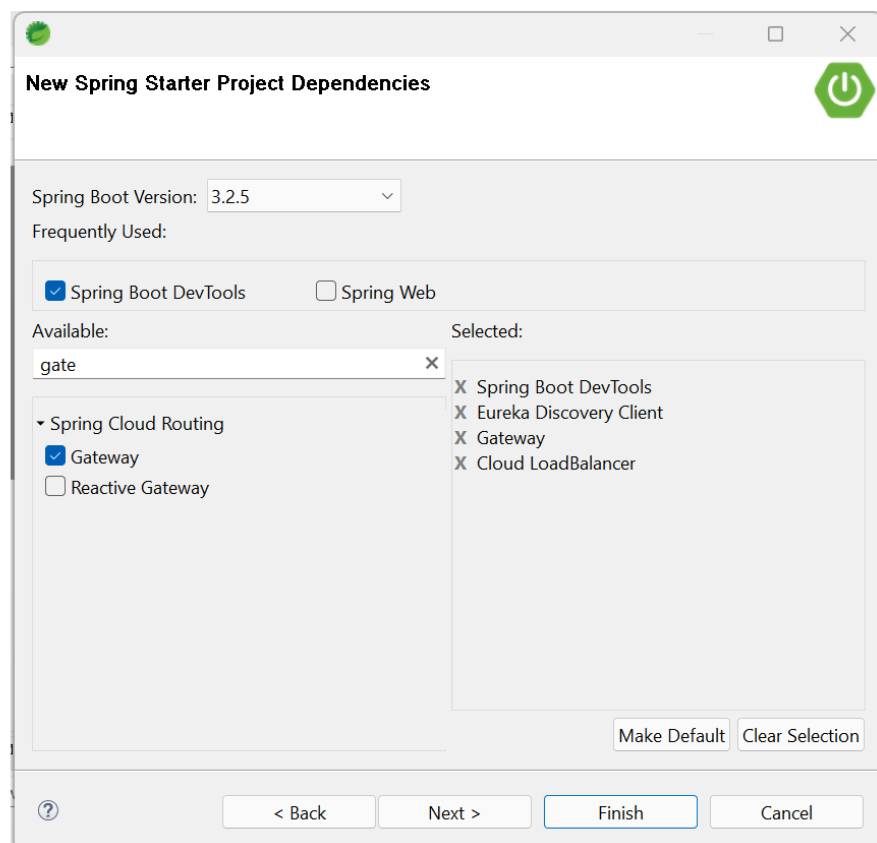
Package:

Working sets

☐ Add project to working sets

Working sets:

1.3 Las dependencias a usar son las siguientes:



**New Spring Starter Project Dependencies**

Spring Boot Version:

Frequently Used:

☒ Spring Boot DevTools ☐ Spring Web

Available:

☒ Spring Cloud Routing

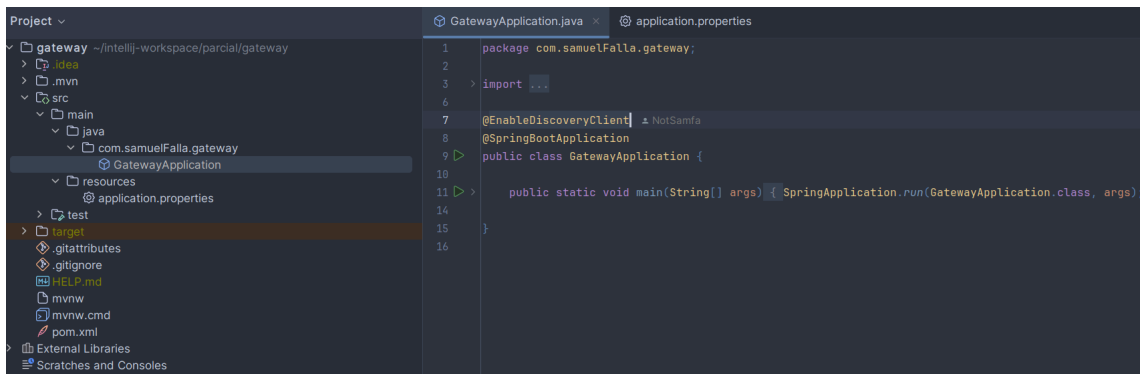
☒ Gateway ☐ Reactive Gateway

Selected:

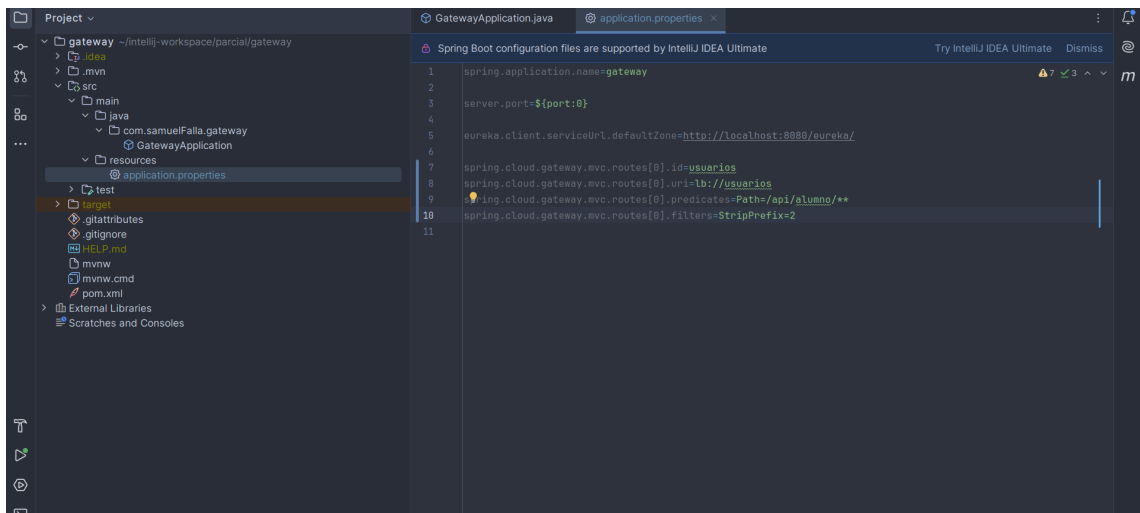
- X Spring Boot DevTools
- X Eureka Discovery Client
- X Gateway
- X Cloud LoadBalancer

1.4 Luego de haber creado el proyecto en la clase principal colocamos a nivel de la clase la anotación

@EnableDiscoveryClient



1.5 En el archivo de propiedades colocamos las siguientes propiedades relacionadas con Gateway



1.6 con la orientación del profesor colocar en este parte del documento el significado de cada una de las propiedades del anterior archivo

**spring.cloud.gateway.routes[0].id:** Define un identificador único para la ruta del gateway.

**spring.cloud.gateway.routes[0].uri:** Indica al gateway a donde debe acceder el lb (balanceador de cargas), es decir, buscar el servicio con nombre “usuarios” en el respectivo registro Eureka.

**spring.cloud.gateway.routes[0].predicates:** Indica la dirección a la que se puede acceder al servicio, solo con esta ruta “/api/alumno/”. Lo demás indicado en “\*\*” es para las rutas mapeadas en el servicio correspondiente.

**spring.cloud.gateway.routes[0].filters:** La solicitud en realidad quitará 2 de las secciones delimitadas por “/”, es decir las 2 partes que trae /api/alumno. De

esta manera, la solicitud sería, por ejemplo, /api/alumno/balaceador-test -> /balaceador-test.

**Referencia:** <https://cloud.spring.io/spring-cloud-gateway/reference/html/>

1.7. Para permitir la comunicación y exposición del servicio usuarios a través del balanceador de carga, colocamos en el archivo de propiedades de ese servicio la siguiente anotación, lo cual corresponde una variable de entorno:

config.balaceador.test=\${BALANCEADOR\_TEST: string por defecto}

Esta variable nos permitirá que instancia esta siendo invocada por el balanceador

1.8 Luego inyectamos esa variable en el controlador

```
23 @Autowired
24 AlumnoService service;
25
26 @Value("${config.balaceador.test}")
27 private String balaceadorTest;
28
29 @GetMapping
30 public ResponseEntity<?> listarAlumno() {
31
32     return ResponseEntity.ok().body(service.findAll());
33 }
34
35 @GetMapping("/")
36 public ResponseEntity<?> listarAlumnoConBalaceadorTest() {
```

1.9 Ahora creamos un método de prueba dentro del controlador que nos retorne el valor de esa variable de entorno junto a la lista de alumnos dentro de un HasMap , de la siguiente forma

```

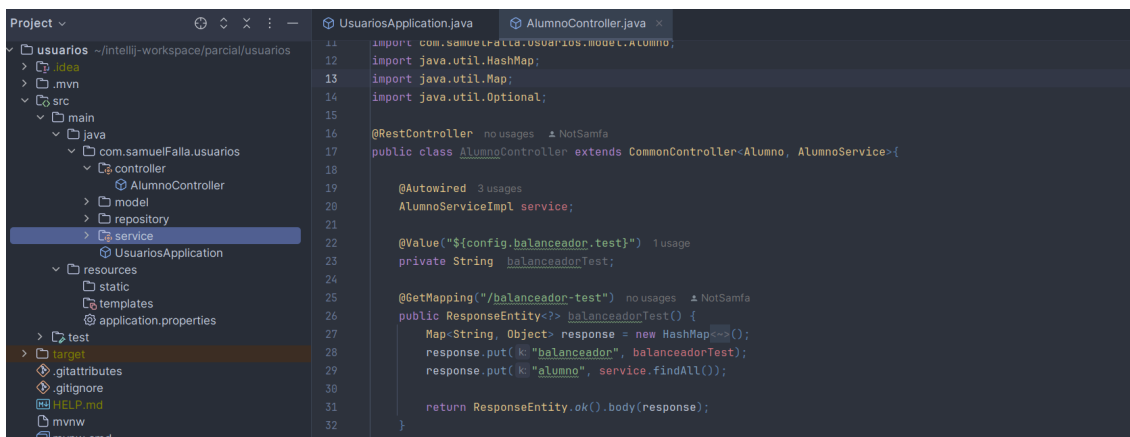
28 @Value("${config.balaceador.test}")
29 private String balaceadorTest;
30
31 @GetMapping("/balanceador-test")
32 public ResponseEntity<?> balanceadorTest() {
33     Map<String, Object> response = new HashMap<String, Object>();
34     response.put("balanceador", balaceadorTest);
35     response.put("alumno", service.findAll());
36
37     return ResponseEntity.ok().body(response);
38 }
39
40 @GetMapping

```

El anterior método nos permite validar el funcionamiento del balanceador de carga.

2.0 para probar los servicio levantamos primero a eureka server, luego el microservicio usuarios y por ultimo el Gateway.

Podemos ver como están corriendo los 3 servicios



Y vamos a eureka para ver que realmente estén registrados allí

The screenshot shows the Spring Eureka web interface running on localhost:8080. The interface has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into several sections:

- System Status:** A table showing environment details.
 

Environment	test	Current time	2024-11-10T12:39:54 -0500
Data center	default	Uptime	00:01
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	0
- DS Replicas:** A section for distributed storage replicas.
- Instances currently registered with Eureka:** A table listing registered applications.
 

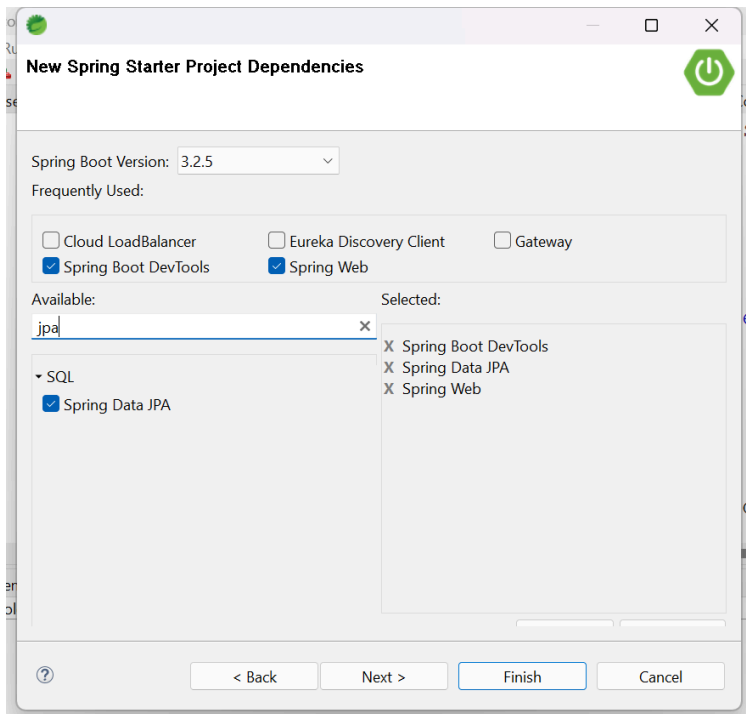
Application	AMIs	Availability Zones	Status
GATEWAY	n/a (1)	(1)	UP (1) - <a href="#">fedora:gateway:0</a>
USUARIOS	n/a (1)	(1)	UP (1) - <a href="#">fedora:usuarios:0</a>
- General Info:** A section for general information.

3. Vamos a crear un componente genérico que nos va a permitir generar reuso de código.

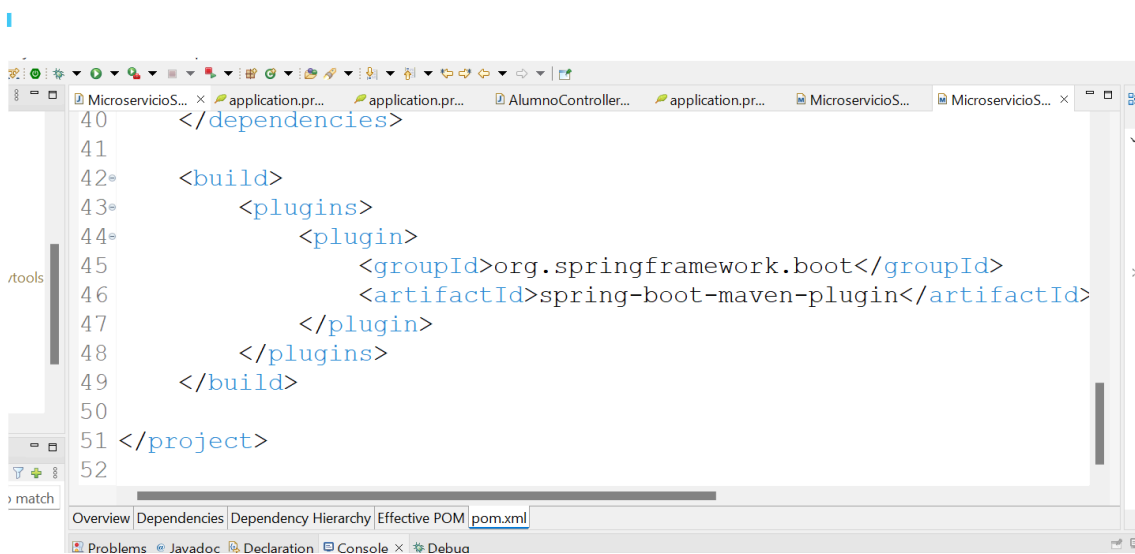
Antes de continuar investiga y ponlo en esta parte de la actividad que son los genéricos en java , sus ventajas y usos.

**Genéricos en java:** Permiten trabajar con diferentes tipos de datos, sin perder la seguridad al momento de ejecución. Gracias a esto se pueden crear clases o métodos flexibles y, por lo tanto, reusables, que puedan gestionar lo que sea que se instancie.

3.1 Creamos un nuevo proyecto con las siguientes dependencias

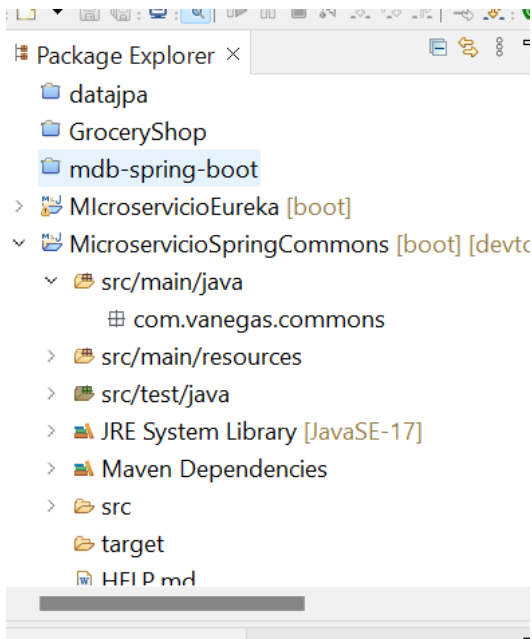


3.2 Luego de creado el proyecto nos dirigimos al pom.xml y eliminamos el plugin de Maven como se ve a continuación:

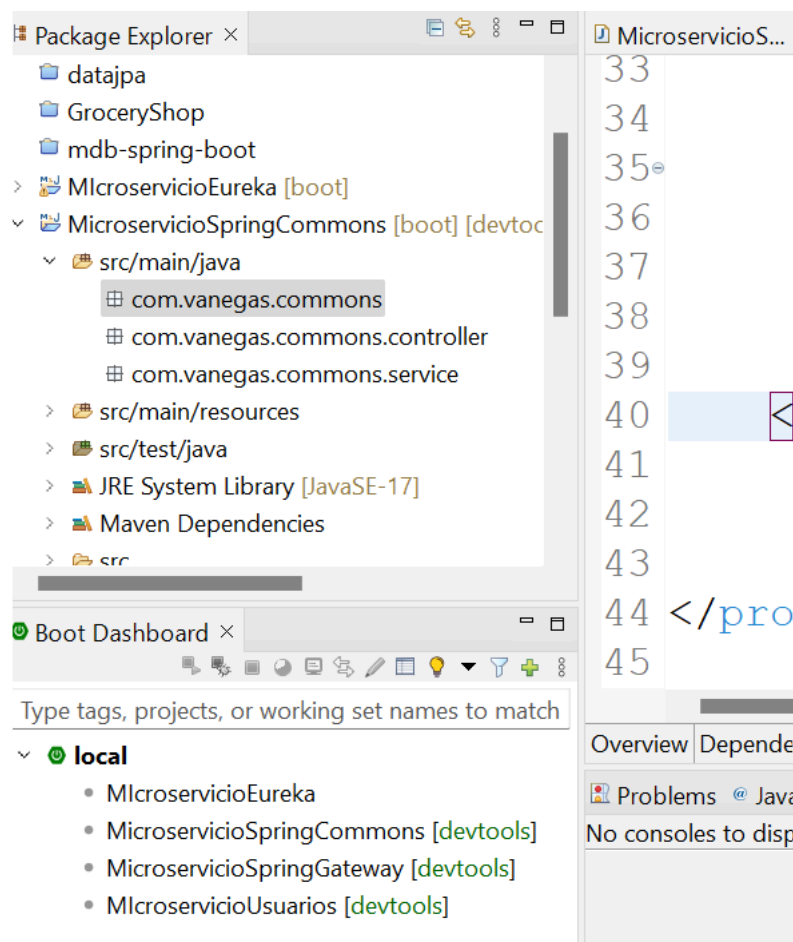


3.3 Luego eliminamos la clase principal ya que lo que se quiere en construir una librería

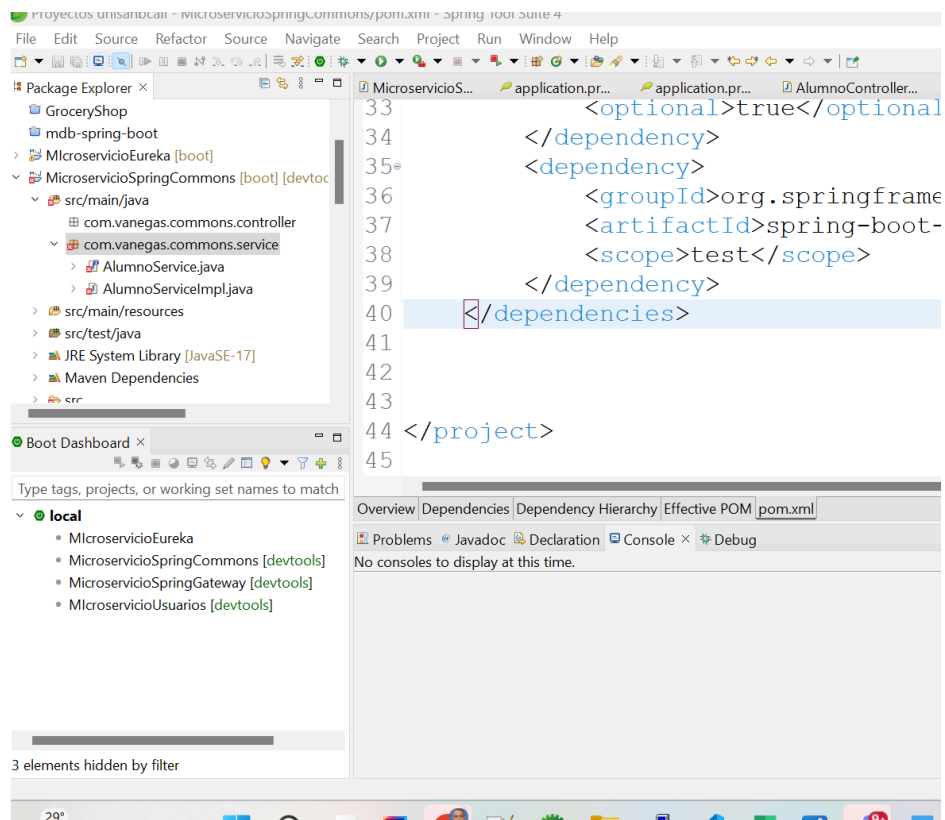




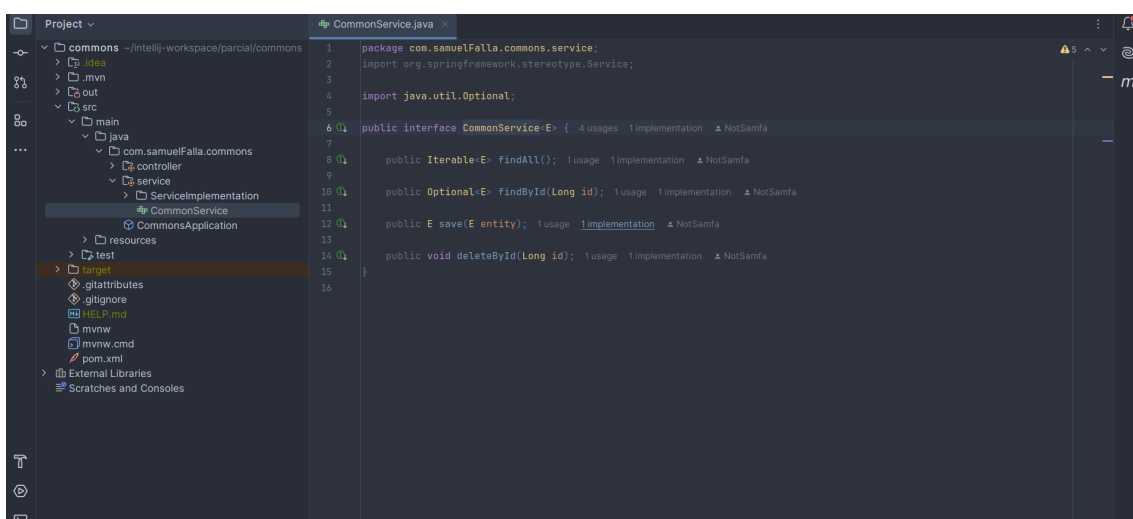
3.4 Luego creamos dos paquetes uno con el path de service y el otro de controller



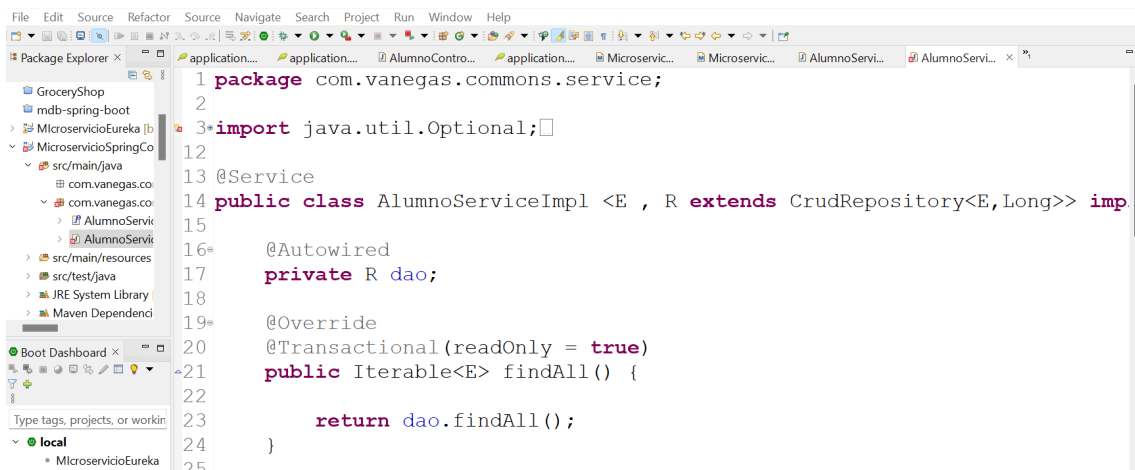
### 3.5 Dentro del paquete service copiamos las clases AlumnoService y AlumnoServiceImpl que tenemos en el proyecto MicroservicioUsuarios



### 3.6 Vamos a hacer algunos cambios en la interface, quietamos lo que tenga la entidad Alumno y trabajaremos con el api generic de java y quedaría así:



### 3.7 ahora modificamos la implementación del service de la siguiente manera



3.8 Cambiamos la inyección del componente repository para que sea genérico , allí se puede ver que estamos usando dos genéricos la E que corresponde a la entidad de la bd y la R que corresponde a la capa de acceso a datos.

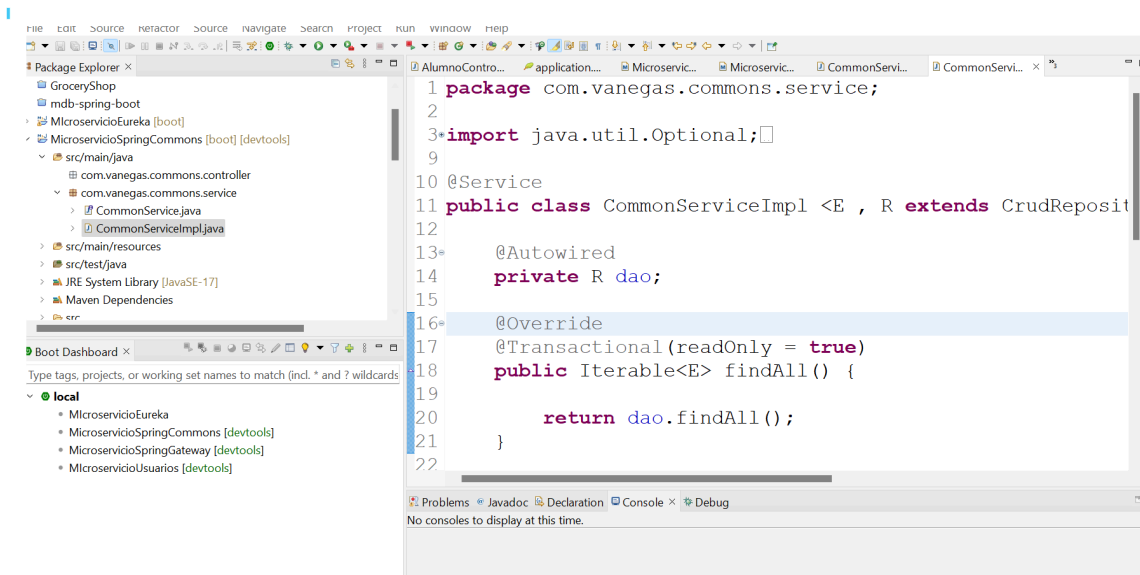
Y en todas las partes de esta clase donde se encuentre la clase Alumno lo cambiamos por E y la palabra alumno por entity

```

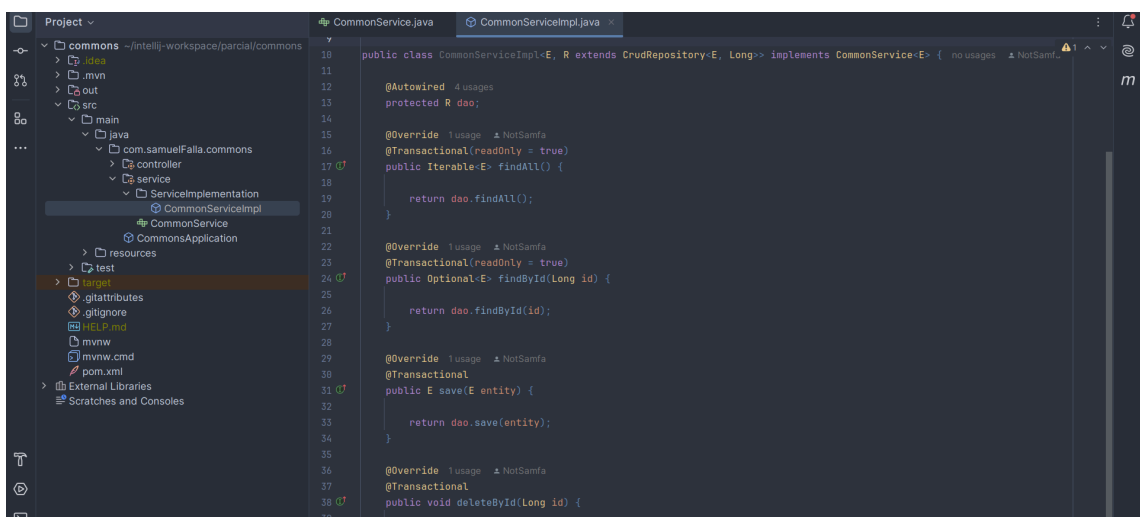
1 @Override
2 @Transactional
3 public E save(E entity) {
4
5     return dao.save(entity);
6 }

```

3.9 Y luego por buenas practicas renombramos las clases que hemos modificado con CommonService y CommonServiceImpl



Y como no es un componente que vamos a inyectar en la clase CommonServiceImpl quitamos el decorador @Service



3.10 ahora se tiene que inyectar este servicio en el microservicio en el de usuarios para ello copiamos el

`groupId` , `artifactId` y `version`

Estas dependencias del servicio de Common , deben ser llevadas al servicio de Usuarios como una dependencia y de momento dejamos allí