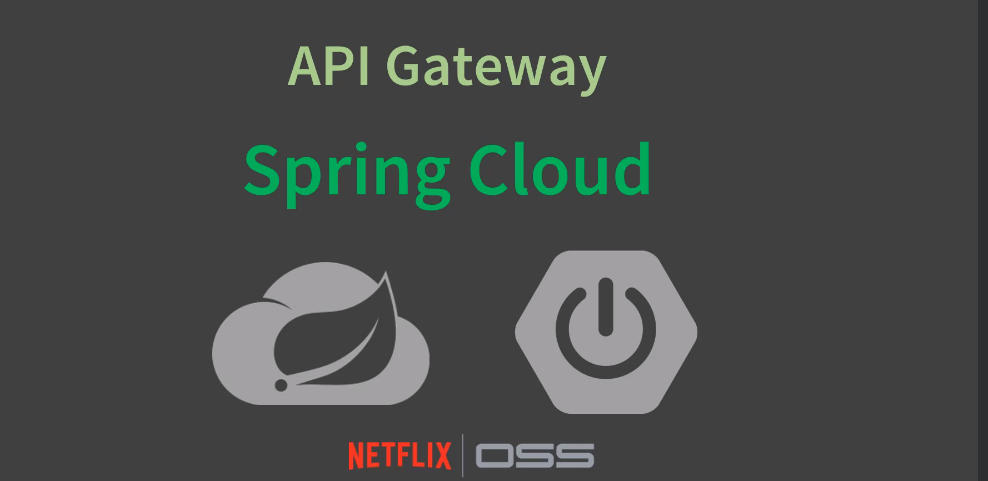
Actividades a desarrollar:

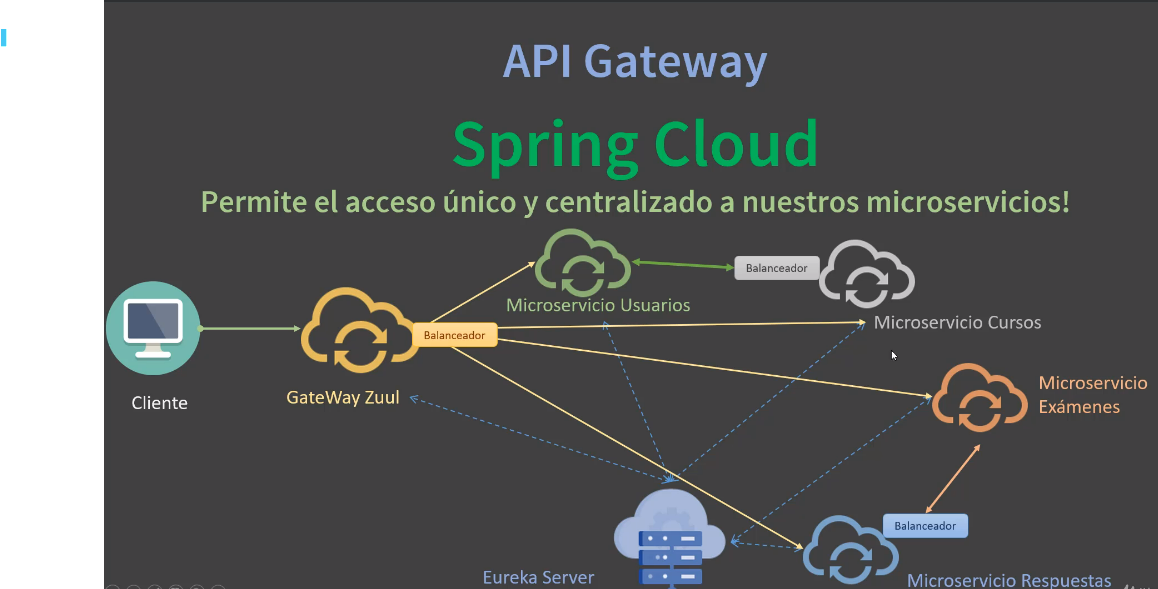
CREACIÓN DE PUERTA DE ENLACE GATEWAY

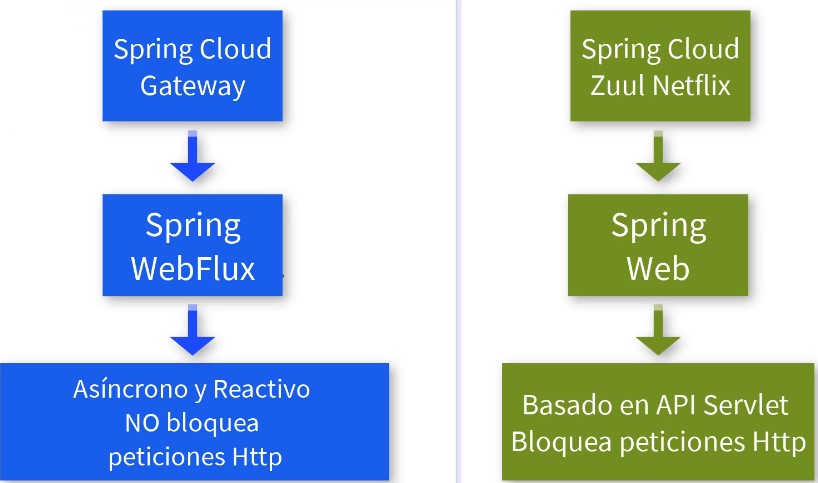


1.1 Buscar y poner en esta parte el concepto de Spring Cloud Gateway

Texto

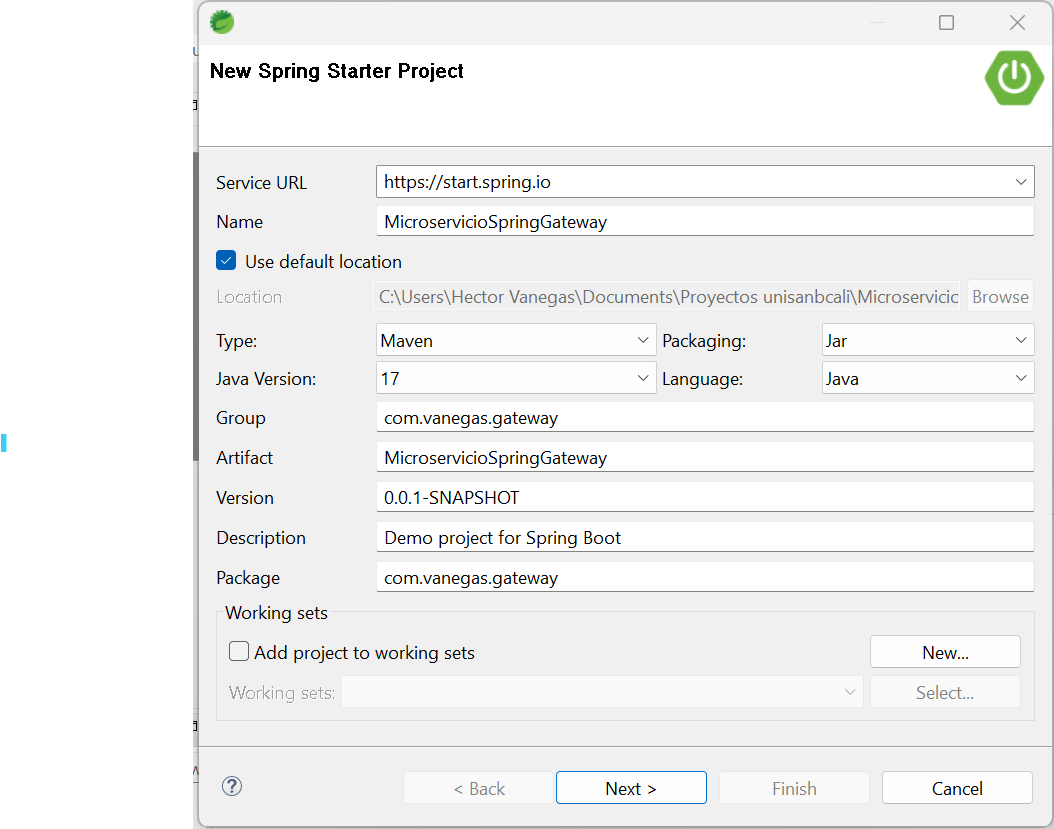
Descripción generada automáticamente



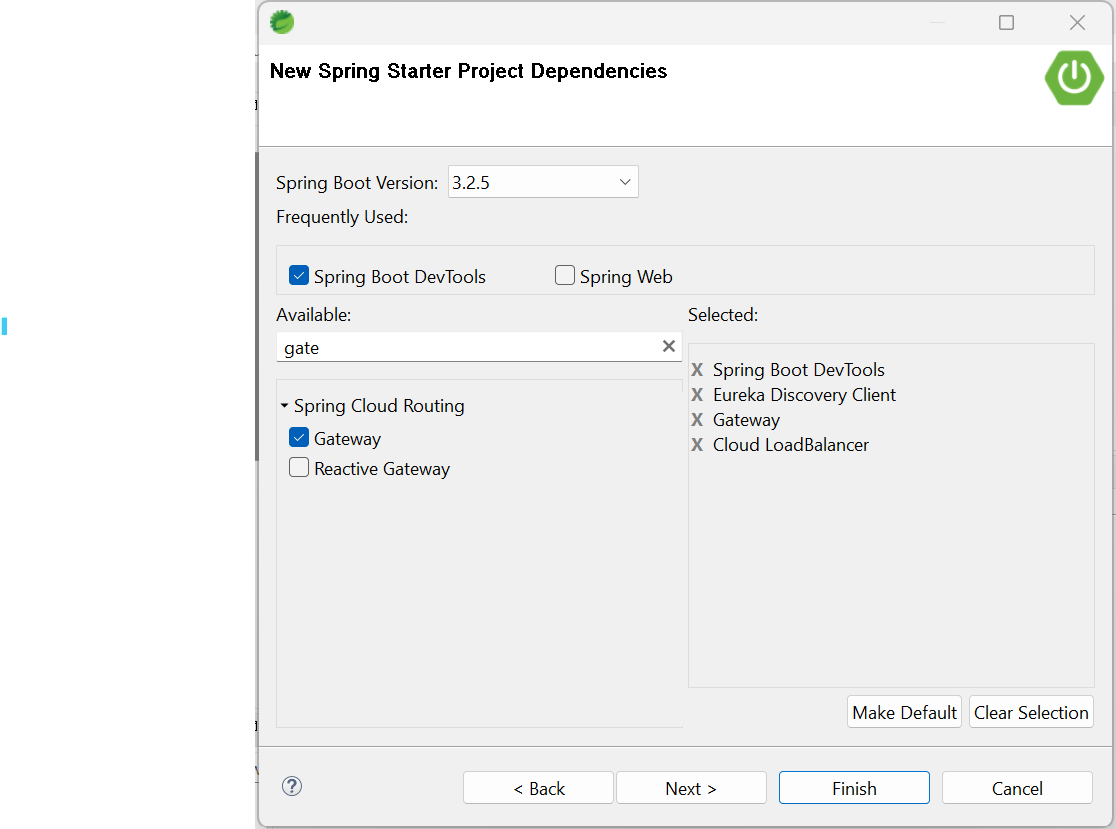


1.2 Se debe crear un nuevo servicio que servirá como apigateway

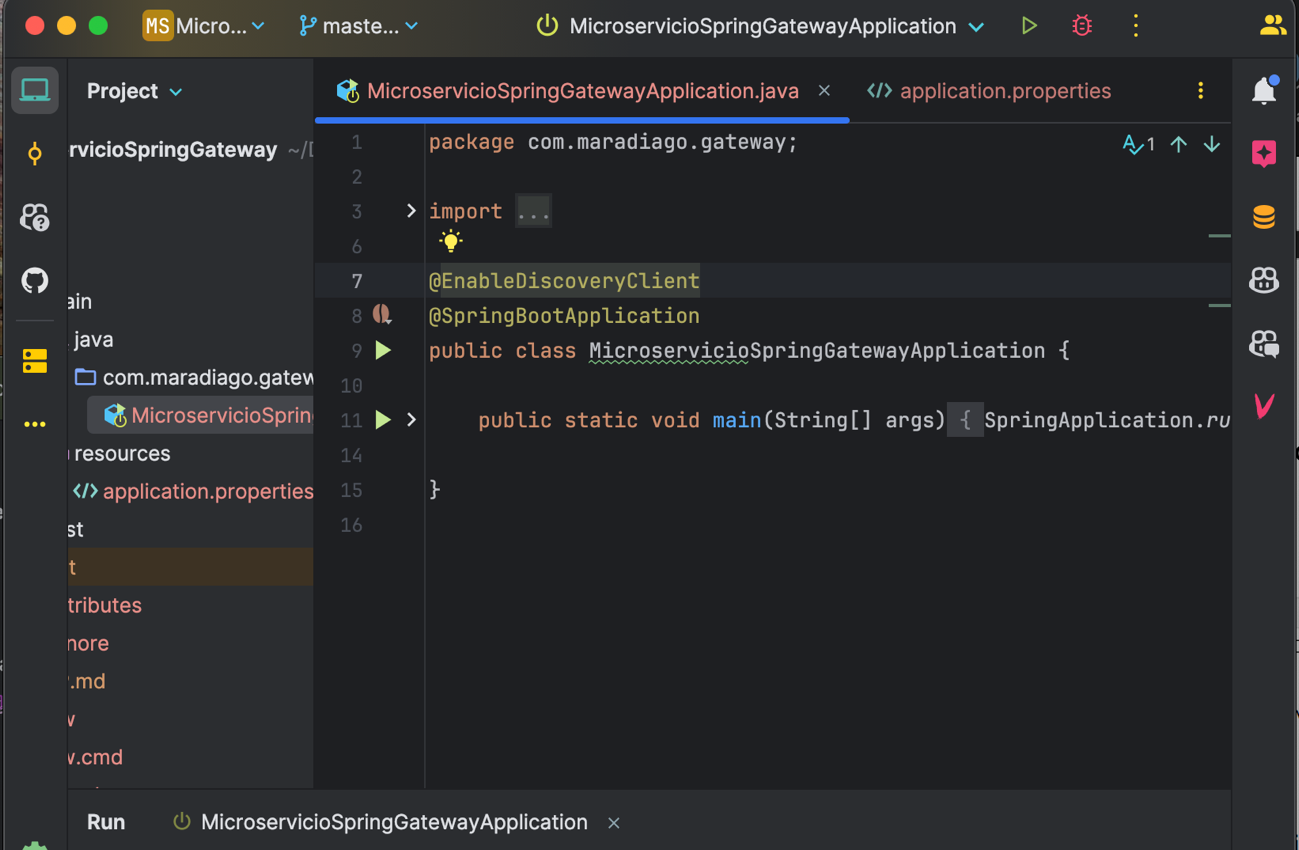
1. **CREACIÓN DE APIGATEWAY**

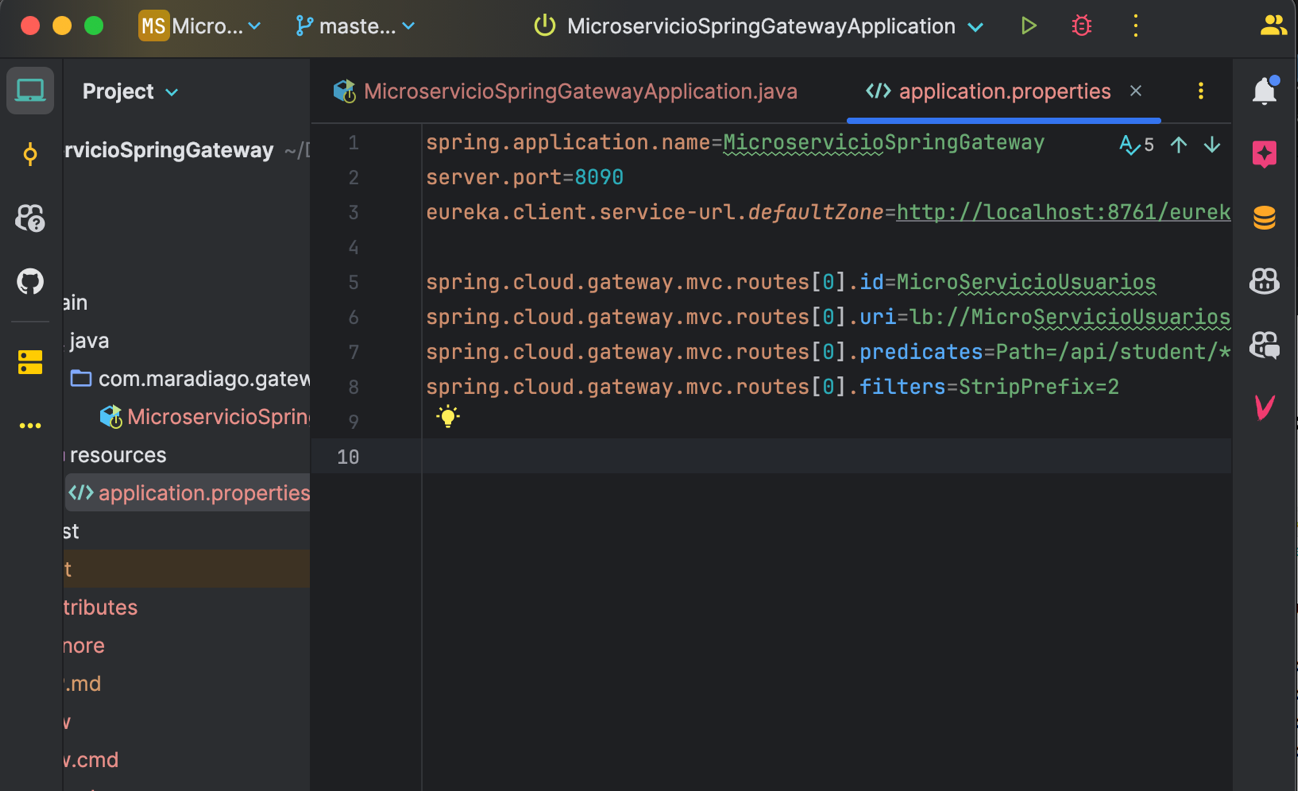


1.3 Las dependencias a usar son las siguientes:



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

**server.port=8090**

Esta propiedad especifica el puerto en el que se ejecutará el servidor de tu aplicación Spring Boot. En este caso, el servidor estará disponible en localhost:8090.

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

Esta propiedad configura la URL del servidor Eureka al cual el cliente se conectará para registrarse y descubrir otros servicios. localhost:8761/eureka indica que el servidor Eureka se está ejecutando localmente en el puerto 8761 y en la ruta /eureka.

**spring.cloud.gateway.mvc.routes[0].id=MicroServicioUsuarios**

Esta propiedad define un identificador único para la ruta del gateway. En este caso, MicroServicioUsuarios es el nombre que se le asigna a la ruta. Este identificador es útil para distinguir esta ruta de otras en la configuración del gateway.

**spring.cloud.gateway.mvc.routes[0].uri=lb://MicroServicioUsuarios**

Esta propiedad especifica la URI de destino para las solicitudes que pasen por esta ruta del gateway. El prefijo lb:// indica que el gateway debe utilizar un cliente de carga balanceada (load balancer) para redirigir la solicitud a uno de los servicios registrados en Eureka con el nombre

MicroServicioUsuarios.

**spring.cloud.gateway.mvc.routes[0].predicates=Path=/api/student/\*\***

Aquí se define una condición (predicate) para esta ruta. En este caso, la condición es que las solicitudes con una ruta que coincide con /api/student/\*\* pasarán por esta ruta del gateway. El /\*\* indica que cualquier subruta después de /api/student/ también será incluida en esta coincidencia.

**spring.cloud.gateway.mvc.routes[0].filters=StripPrefix=2**

Esta propiedad aplica un filtro StripPrefix a la ruta. StripPrefix=2 indica que se deben eliminar los primeros dos segmentos del camino (path) de la solicitud antes de redirigirla al servicio. Por ejemplo, si la solicitud original es /api/student/user/details, el gateway eliminará los dos primeros segmentos (/api/student), y el servicio recibirá la solicitud como /user/details.

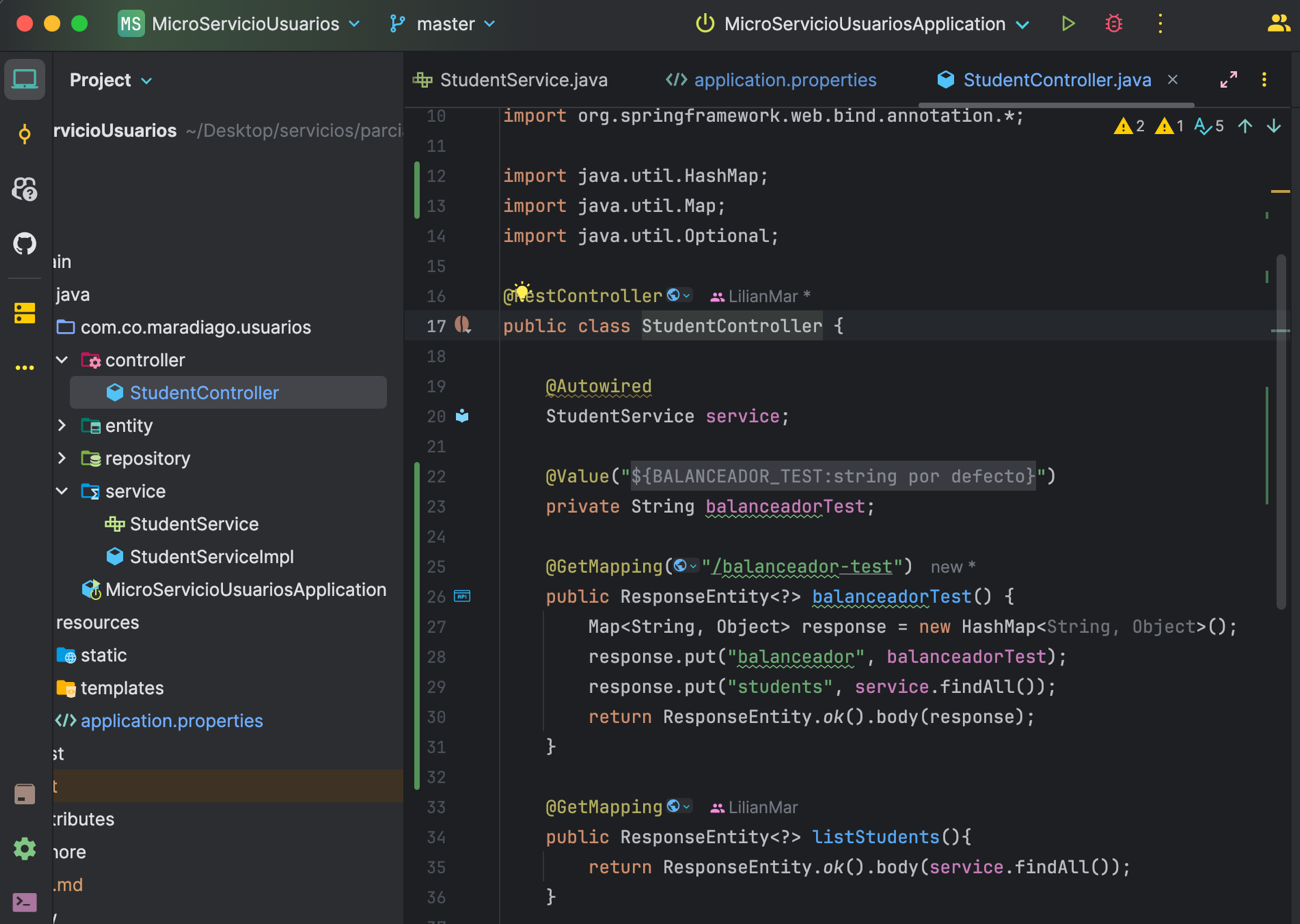
Esta configuración establece un gateway en el que las solicitudes que coinciden con /api/student/\*\* se dirigen al microservicio MicroServicioUsuarios, con los dos primeros segmentos de la ruta eliminados para simplificar la ruta al llegar al microservicio.

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:

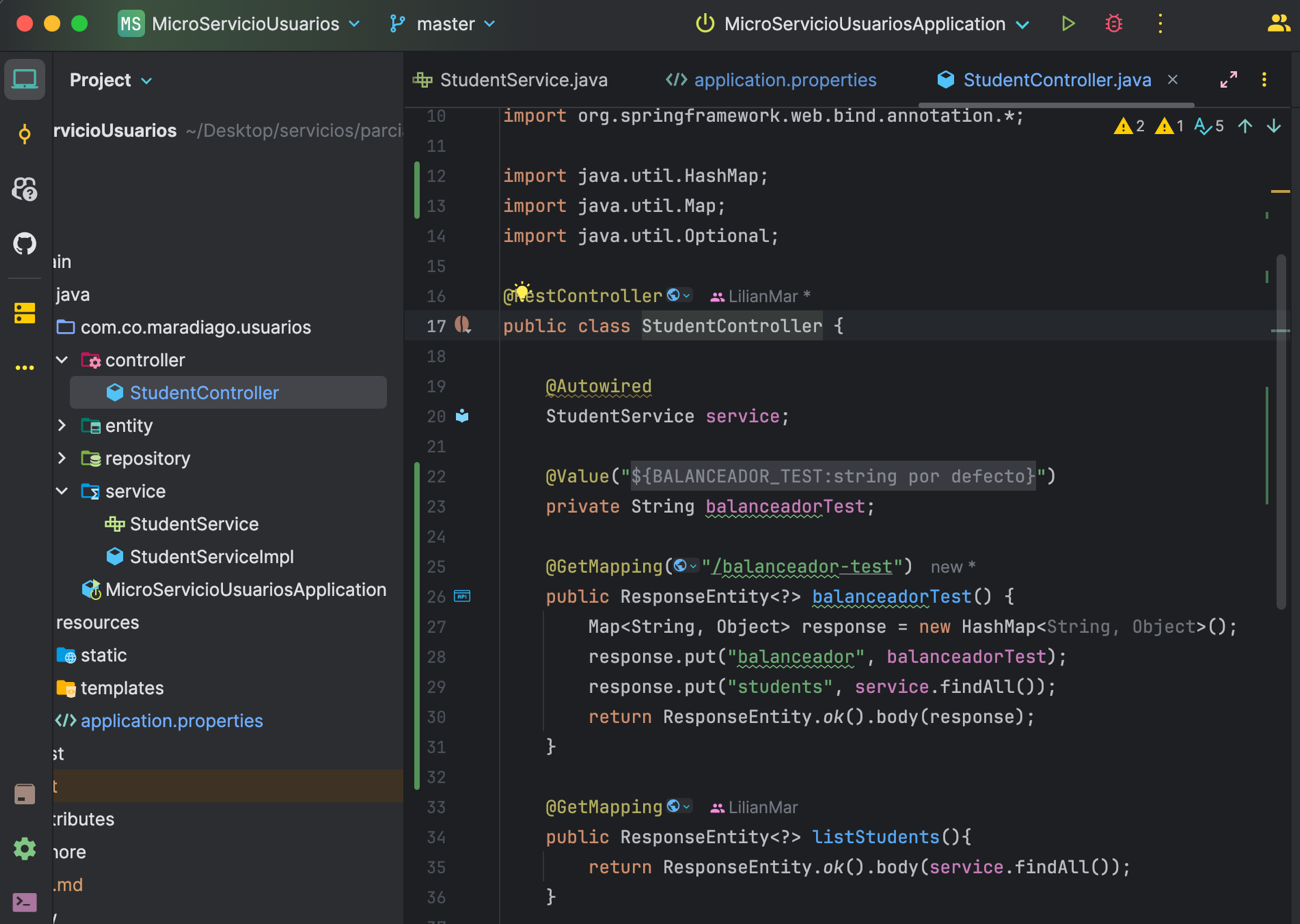
config.balanaceador.test=${BALANCEADOR\_TEST: string por defecto}

Esta variable nos permitirá validar que instancia está siendo invocada por el balanceador

1.8 Luego inyectamos esa variable en el controlador



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

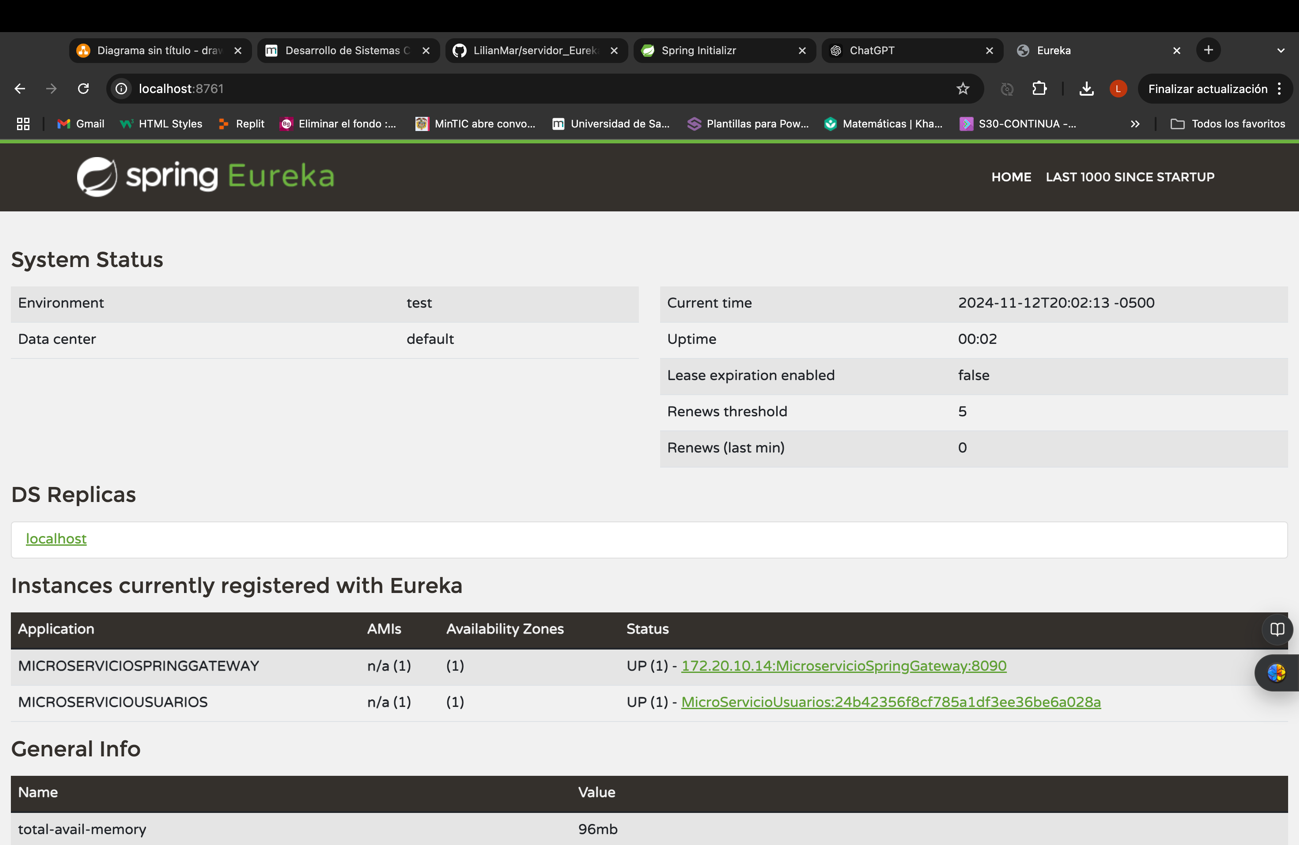


El anterior método nos permite validar el funcionamiento del balanceador de carga.

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

Podemos ver cómo están corriendo los 3 servicios

Y vamos a eureka para ver que realmente estén registrados allí



**3. CREACIÓN DE LIBRERÍA**

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

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

En Java, los genéricos son una característica que permite que las clases, interfaces y métodos operen sobre tipos de datos especificados en el momento de su uso. Esto permite definir clases y métodos que pueden trabajar con cualquier tipo de dato, proporcionando una mayor flexibilidad sin perder seguridad en el tipo de datos.

¿Cómo funcionan los genéricos?

Cuando se usa un genérico, se especifica un parámetro de tipo entre los símbolos <>, permitiendo que el tipo de dato se determine en el momento de la instanciación. Por ejemplo:

List<String> nombres = new ArrayList<>();

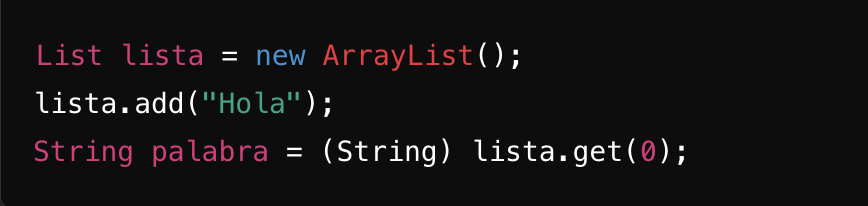
En este caso, List<String> especifica que la lista contendrá solo elementos de tipo String, y si intentas agregar un tipo diferente, se generará un error de compilación.

Ventajas de los Genéricos

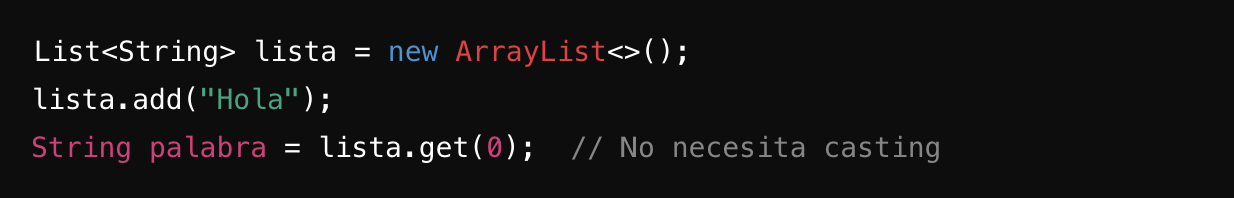
Seguridad en el tipo: Los genéricos ayudan a evitar errores de tipo en tiempo de ejecución al asegurarse en tiempo de compilación de que los tipos de datos son consistentes. Por ejemplo, si declaras List<Integer>, no puedes accidentalmente agregar un String.

Eliminación de Casts (conversiones explícitas): Sin genéricos, se requiere hacer conversiones explícitas al recuperar elementos de colecciones genéricas. Con genéricos, el tipo está asegurado y se elimina la necesidad de conversiones.

Sin genéricos:



Con genéricos



Reutilización de código: Puedes crear clases y métodos que funcionen con cualquier tipo de datos, lo que mejora la reutilización del código. Por ejemplo, List<T> puede contener cualquier tipo T, en lugar de crear una lista específica para cada tipo de dato.

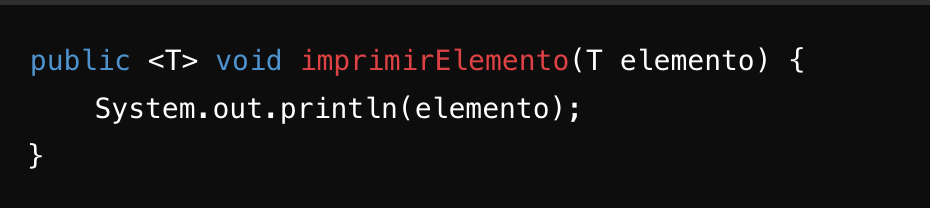
Mejor legibilidad y mantenimiento del código: Los genéricos hacen el código más claro y fácil de leer al definir explícitamente el tipo de datos que se espera, lo cual facilita el mantenimiento.

Uso de los Genéricos en Java

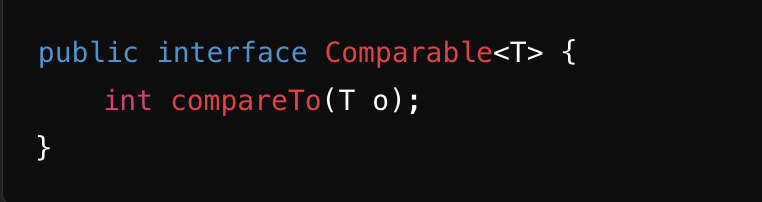
Clases Genéricas: Puedes crear clases genéricas que operen en múltiples tipos. Por ejemplo:

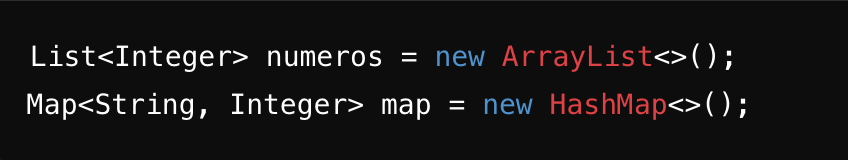


Métodos Genéricos: Permiten definir métodos que utilizan parámetros de tipo, sin hacer genérica a toda la clase. Ejemplo:



Interfaz Genérica: Al igual que las clases, las interfaces también pueden ser genéricas.



Genéricos en Colecciones: Las clases de la biblioteca de colecciones de Java, como List, Map, y Set, están basadas en genéricos y ayudan a manejar elementos del mismo tipo.

Wildcards en Genéricos (?)

Los wildcards o comodines son útiles cuando no se conoce el tipo exacto, o si necesitas ser más flexible:

? representa cualquier tipo desconocido.

? extends T restringe el tipo a T o subclases de T (limitado por la parte superior).

? super T restringe el tipo a T o superclases de T (limitado por la parte inferior).



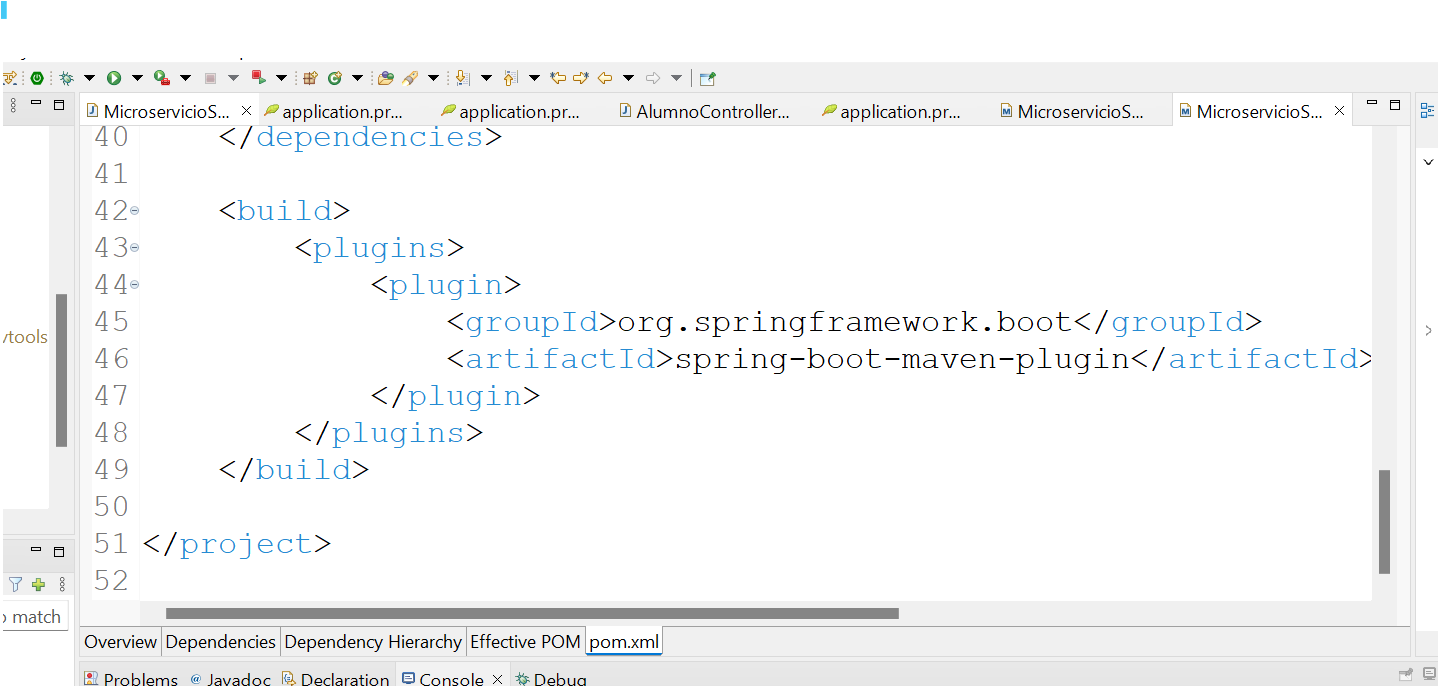
En resumen, los genéricos en Java proporcionan **seguridad de tipo, eliminan la necesidad de casting**, y promueven la **reutilización del código y el mantenimiento**. Son fundamentales al trabajar con estructuras de datos y clases que necesitan ser flexibles para manejar diferentes tipos de datos.

3.1 Creamos un nuevo proyecto con las siguientes dependencias

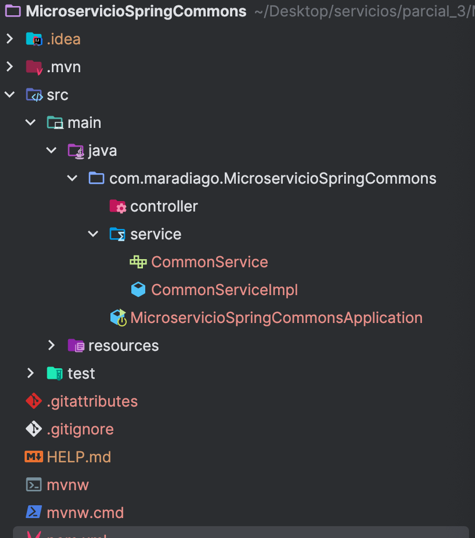
Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

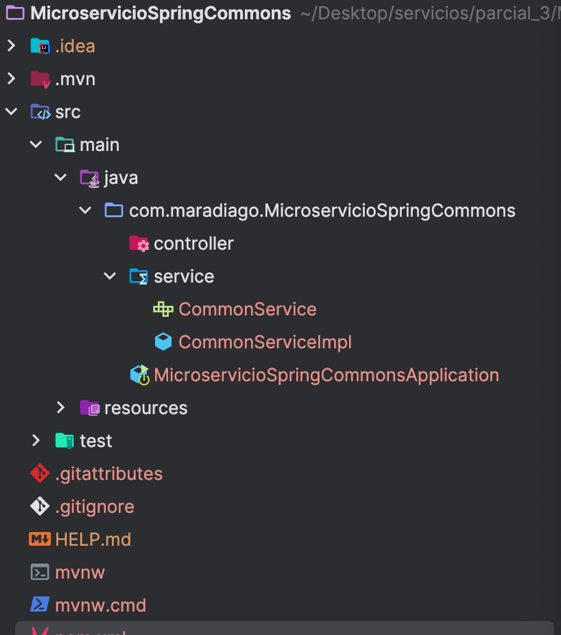
3.2 Luego de creado el proyecto nos dirigimos al pom.xml y eliminamos el plugin de Maven como se ve a continuación:



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

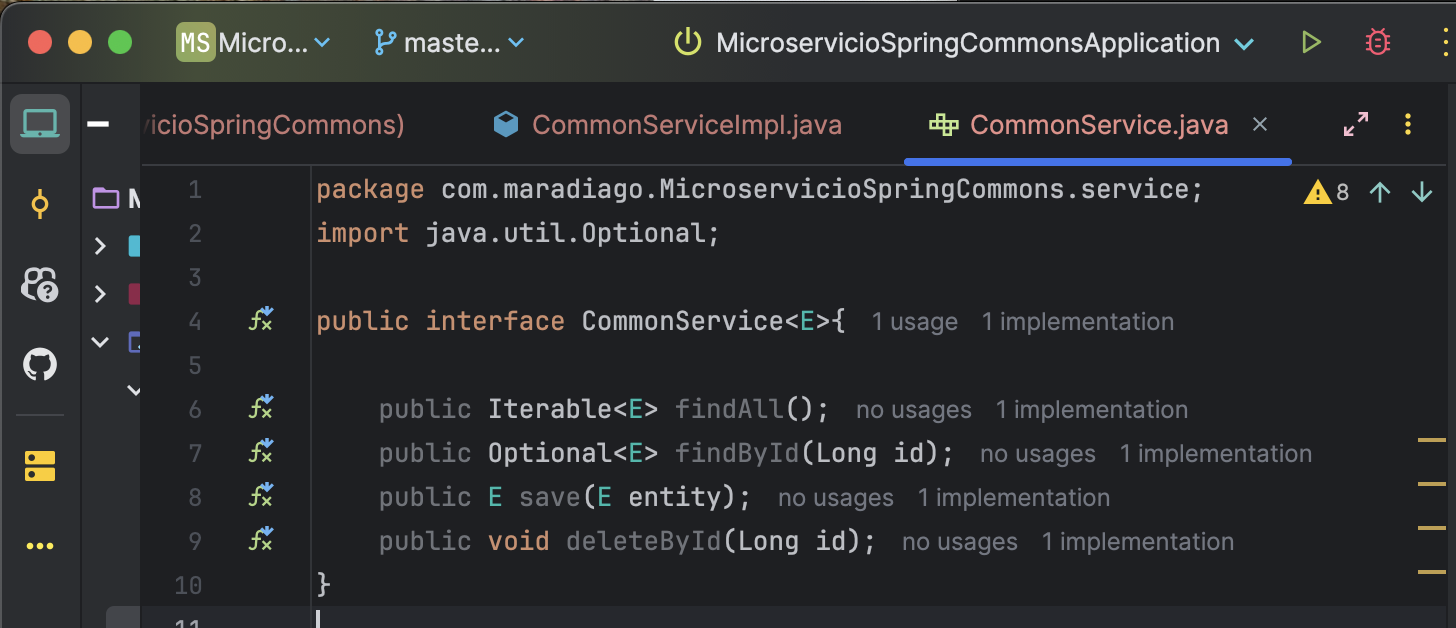


3.4 Luego creamos dos paquetes uno con el path de service y el otro de controller

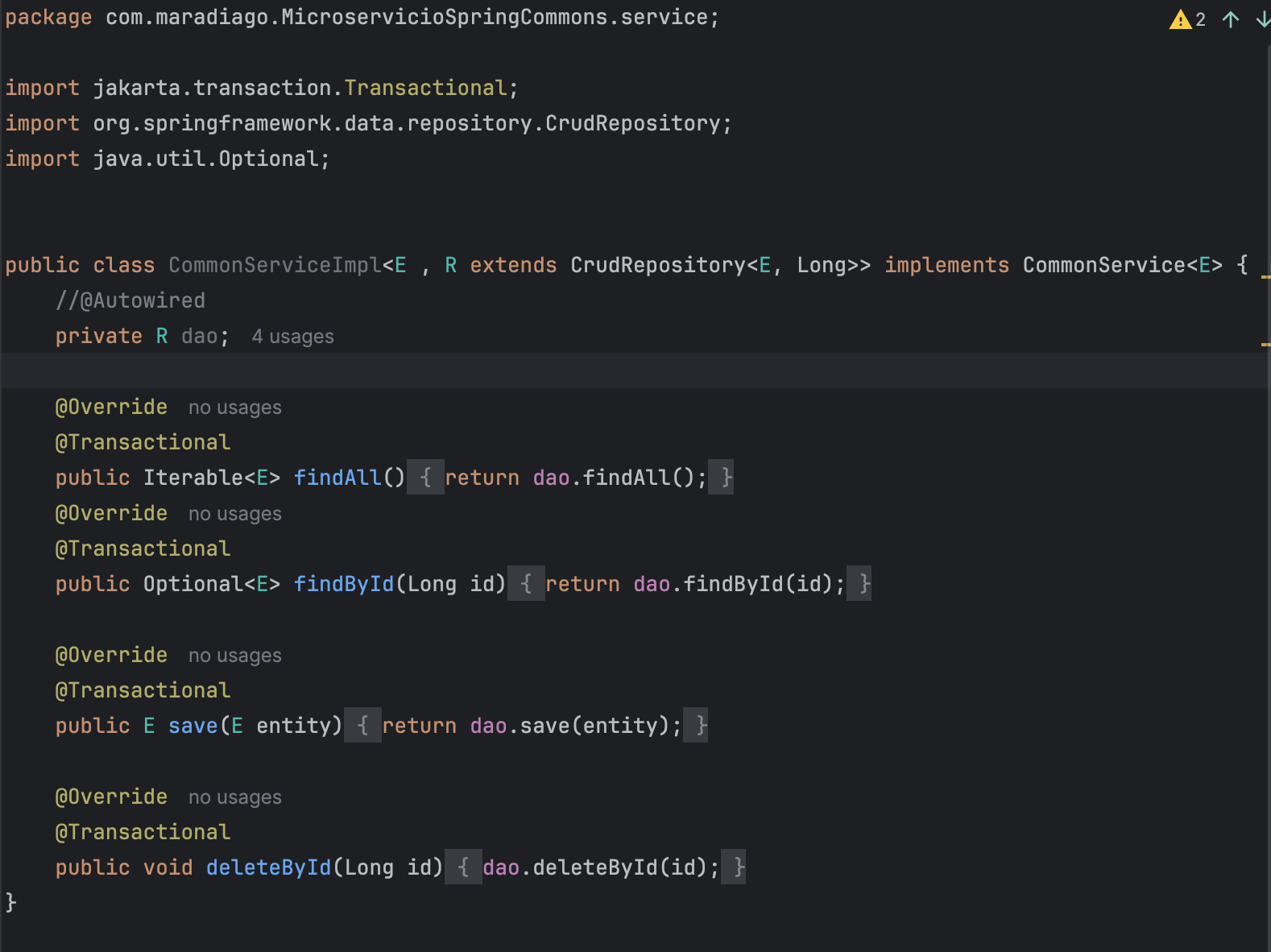


3.5 Dentro del paquete service copiamos las clases AlumnoService y AlumnoServiceImpl que tenemos en el proyecto MicroservicioUsuarios

3.6 Vamos a hacer algunos cambios en la interface, quitamos 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



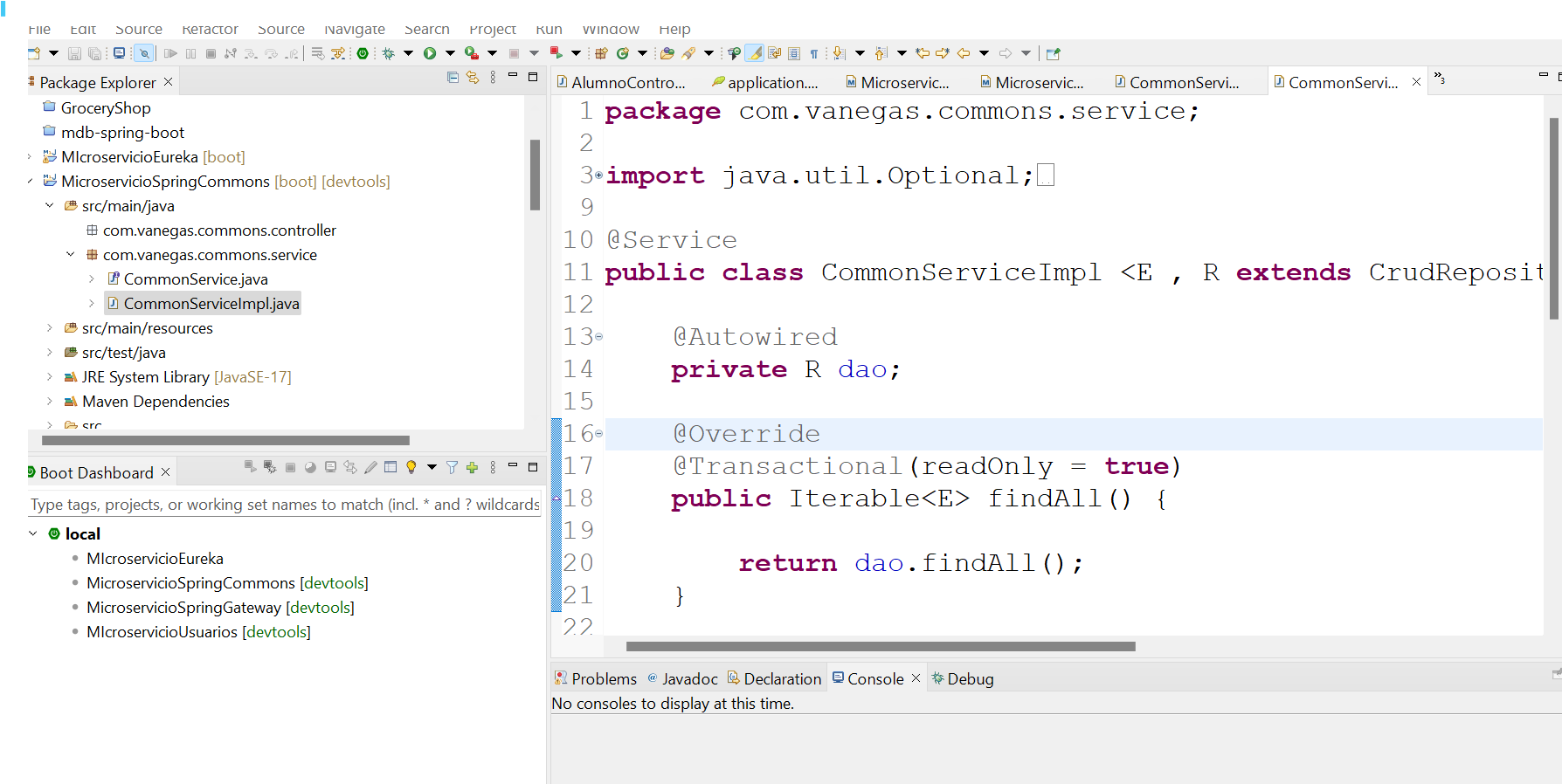
3.8 Cambiamos la inyección del componente repository para que sea genérico, allí se puede ver que estamos usando dos genéricos la E que corresponde a la entidad de la bd y la R que corresponde a la capa de acceso a datos.

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

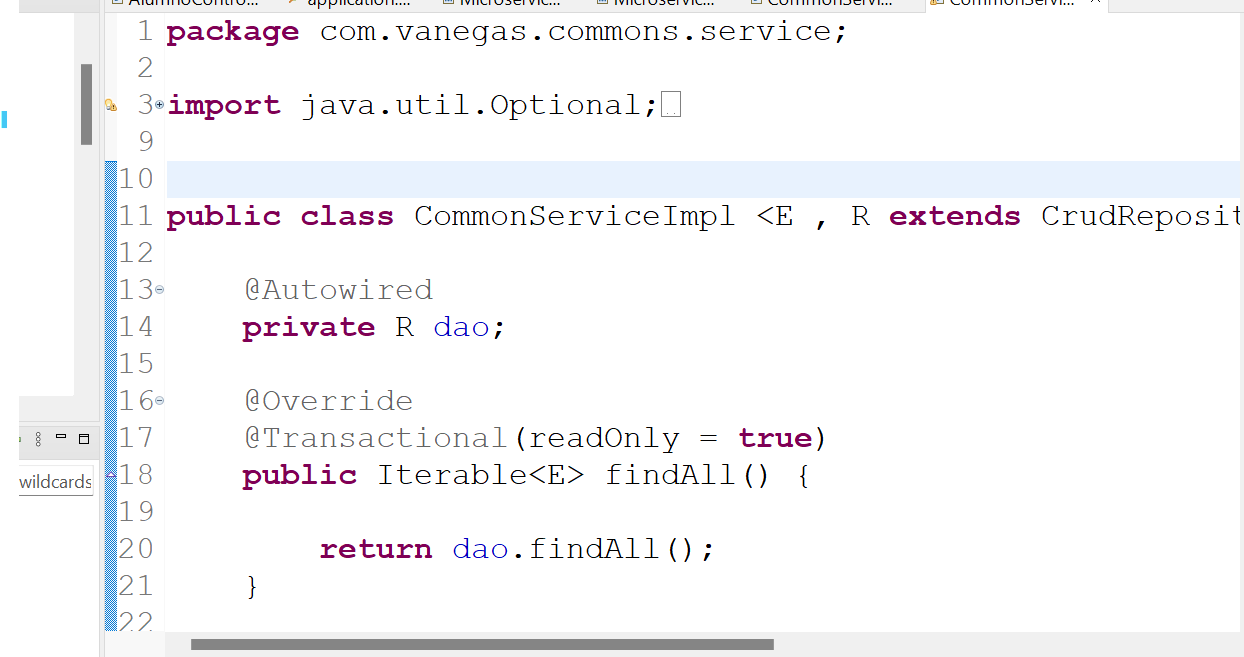
Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

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