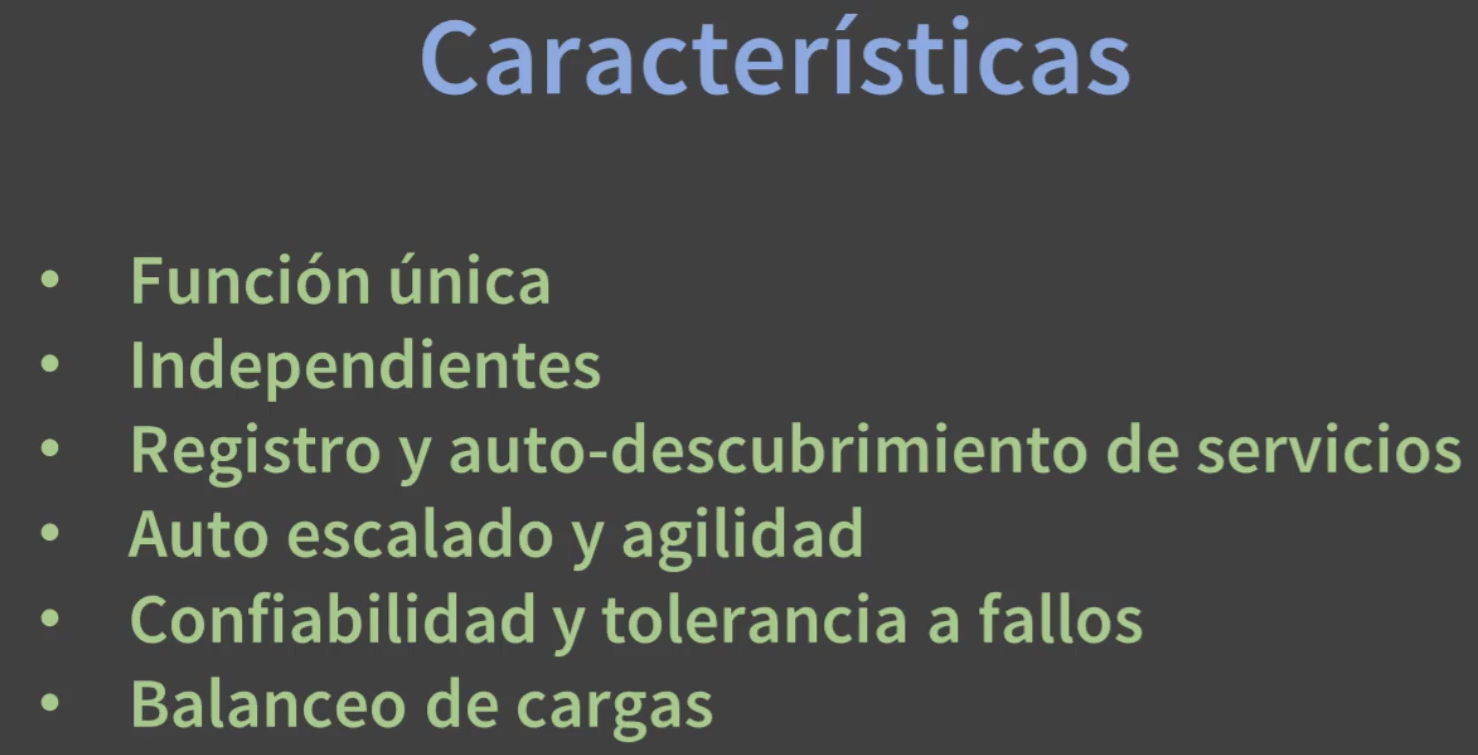
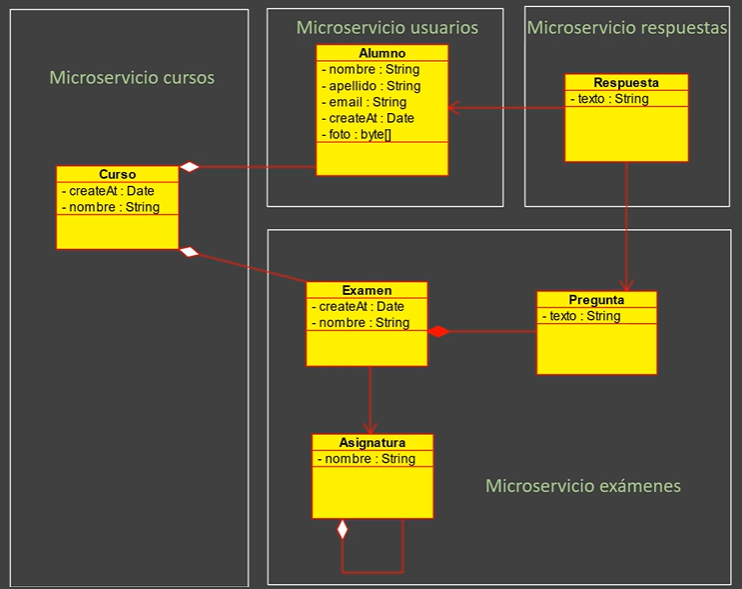


Servidor de nombre Eureka

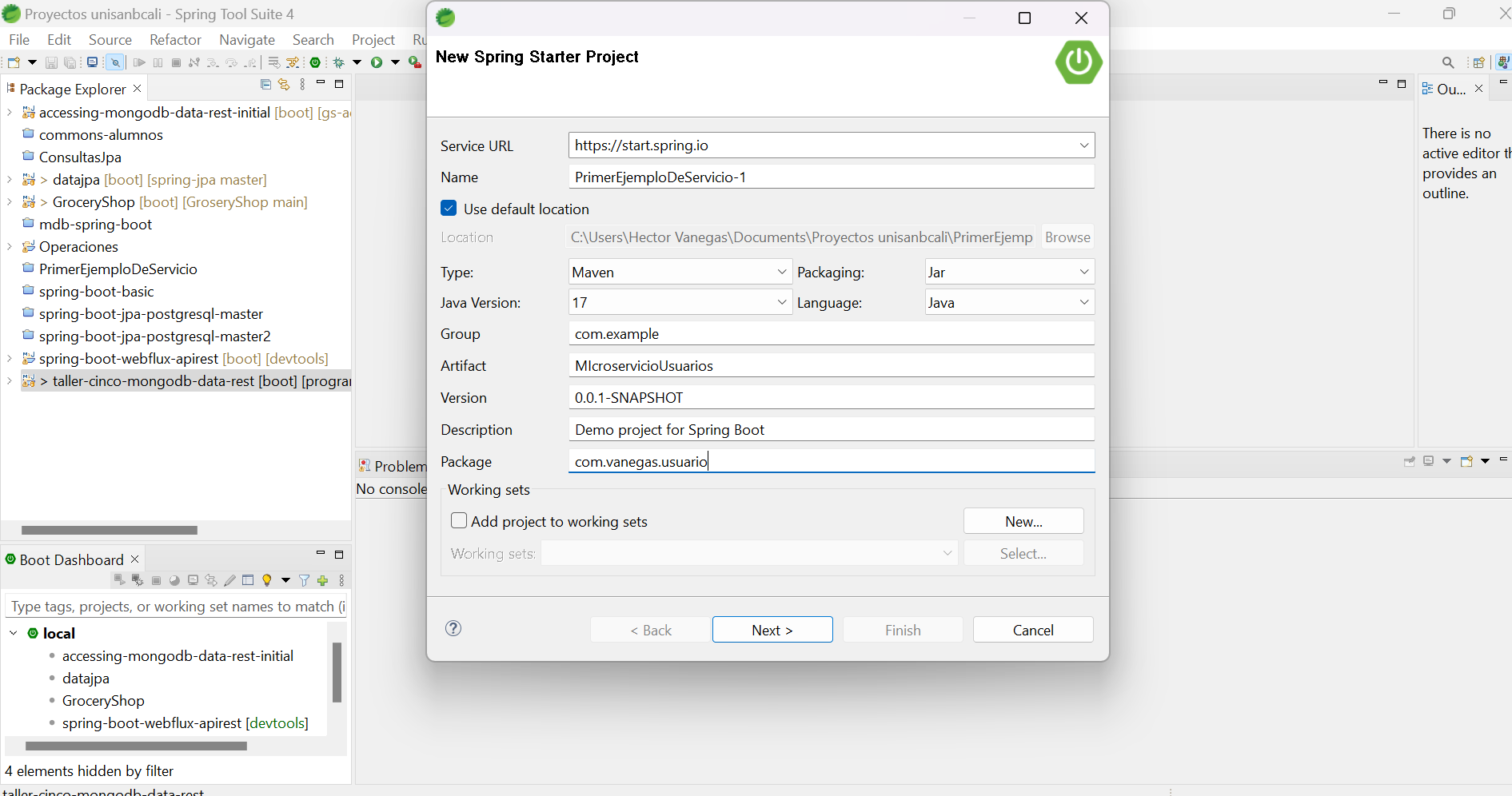


Llegaremos a construir este diagrama



Actividades a desarrollar:

1. **CREACIÓN DE SERVICIO USUARIOS**
2. Creamos el primer microservicio con el nombre de Usuarios y lo generamos a través de spring.io con las siguientes dependencias:



Spring web

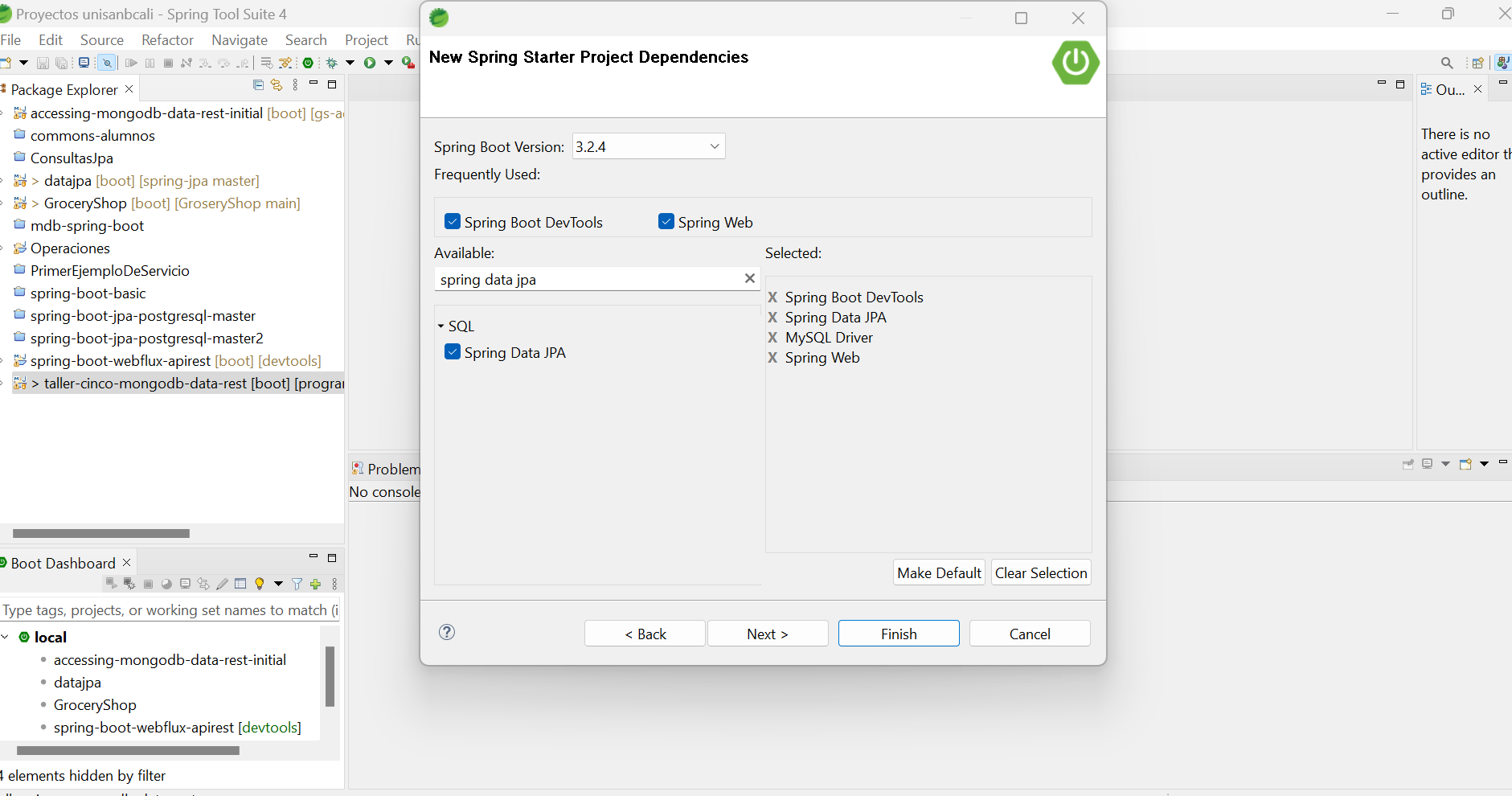
Spring Dev Tools

Spring Data Jpa

MySql Driver

Y que debe tener como paquete principal el siguiente nombre:

com.co.(su apellido).usuarios



1. Dentro de este mismo documento deberá investigar y colocar el significado de cada una de las dependencias de spring descritas anteriormente

**1. Spring Web**

**Funcionalidad:** Spring Web forma parte del proyecto Spring MVC (Model-View-Controller) y permite a los desarrolladores crear aplicaciones web y servicios RESTful en Java.

**Bondades:**

* Simplifica la creación de APIs RESTful: Con anotaciones como @RestController y @RequestMapping, permite crear endpoints que manejan peticiones HTTP (GET, POST, PUT, DELETE) de forma sencilla.
* Controladores MVC: Facilita la construcción de controladores para manejar peticiones y respuestas, ideal para aplicaciones web tradicionales.
* Configuración sencilla: Proporciona opciones de configuración y manejo de excepciones centralizado, reduciendo el código repetitivo y haciendo que el desarrollo sea más eficiente.
* Soporte para JSON y XML: Permite enviar y recibir datos en formato JSON y XML de manera nativa, lo cual es esencial para aplicaciones que consumen y exponen APIs.

**2. Spring DevTools**

**Funcionalidad:** Spring DevTools es una dependencia que acelera el ciclo de desarrollo y pruebas de aplicaciones Spring.

**Bondades:**

* Recarga automática: Permite que los cambios en el código (como actualizar controladores, servicios, etc.) se reflejen de inmediato sin tener que reiniciar el servidor manualmente, lo cual ahorra mucho tiempo durante el desarrollo.
* Reinicio de la aplicación: Ofrece una funcionalidad de "reinicio rápido" que reinicia la aplicación de manera más rápida comparado con un reinicio completo, lo cual es útil cuando se realizan cambios en el código base.
* Desactivación de la caché: Al desactivar la caché de plantillas HTML y CSS, permite ver inmediatamente los cambios en el front-end sin necesidad de limpiar el caché, facilitando la depuración.
* Configuraciones específicas para desarrollo: Activa configuraciones útiles solo en el entorno de desarrollo, evitando que estén activas en producción.

**3. Spring Data JPA**

**Funcionalidad:** Spring Data JPA simplifica el acceso a datos al integrar Spring con JPA (Java Persistence API), que es una especificación para el mapeo objeto-relacional (ORM). Esto permite a los desarrolladores trabajar con bases de datos sin necesidad de escribir mucho código SQL.

**Bondades:**

* Reducción de código repetitivo: Spring Data JPA crea automáticamente las consultas más comunes en los repositorios, como findById, findAll, save, delete, etc., lo que evita tener que escribir consultas SQL desde cero.
* Consultas derivadas: Permite crear métodos de consulta personalizados simplemente mediante el nombre del método, por ejemplo, findByNombre o findByEdadGreaterThan. Spring interpreta estos nombres y genera las consultas automáticamente.
* Consultas con JPQL y SQL nativo: Aun cuando genera consultas automáticamente, también permite definir consultas personalizadas usando JPQL (Java Persistence Query Language) o SQL nativo cuando se requiere mayor control.
* Paginación y ordenamiento: Facilita la implementación de paginación y ordenamiento en las consultas, lo cual es útil cuando se trabajan con grandes volúmenes de datos.

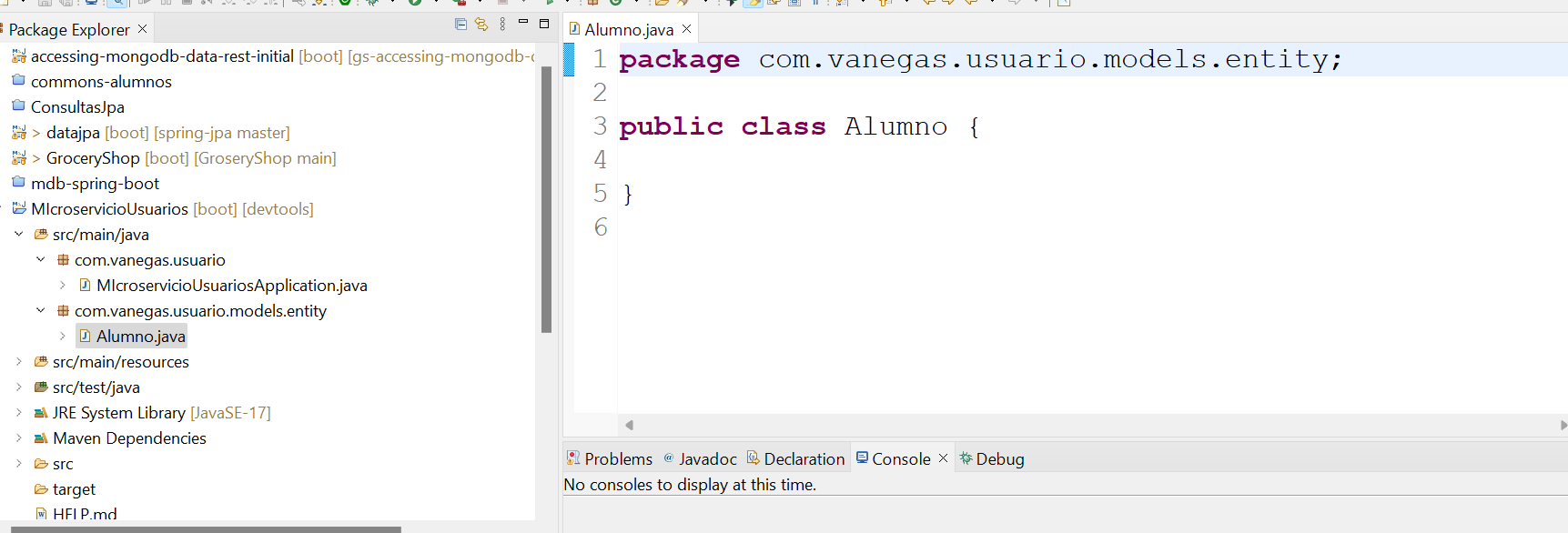
**4. MySQL Driver**

**Funcionalidad:** Este controlador JDBC (Java Database Connectivity) permite a las aplicaciones Java conectarse y comunicarse con una base de datos MySQL.

**Bondades:**

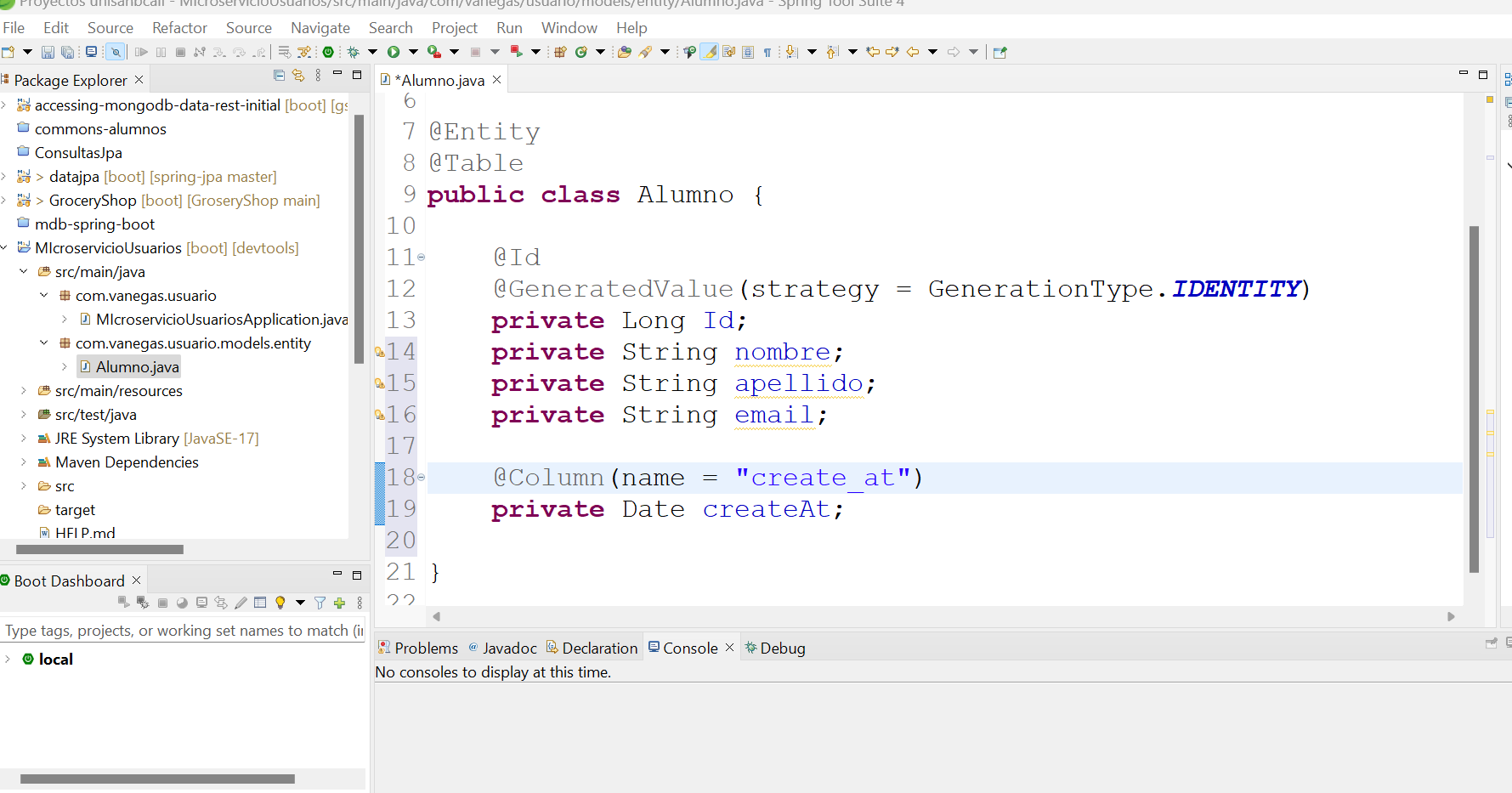
* Interfaz para bases de datos MySQL: Es esencial para cualquier aplicación Java que necesite conectarse a una base de datos MySQL, ya que actúa como el puente entre la aplicación y la base de datos.
* Compatibilidad con Spring Data JPA: Al combinarse con Spring Data JPA, permite a los desarrolladores interactuar con la base de datos de manera sencilla y con el soporte de ORM, facilitando la persistencia de datos sin tener que manejar conexiones y consultas manualmente.
* Manejo de transacciones: Gracias a las configuraciones de transacciones de Spring, se pueden gestionar conexiones de forma eficiente, garantizando que las operaciones de inserción, actualización y eliminación se realicen de manera segura y en bloque (para mantener la consistencia de los datos).

1. En el proyecto anterior crearemos el crud de la clase Alumno, para ello se creará primero un paquete a partir del paquete principal con el nombre de com.co.(su apellido).usuarios.entity.
2. Dentro del paquete anterior se debe construir una clase concreta con el nombre de Alumno.

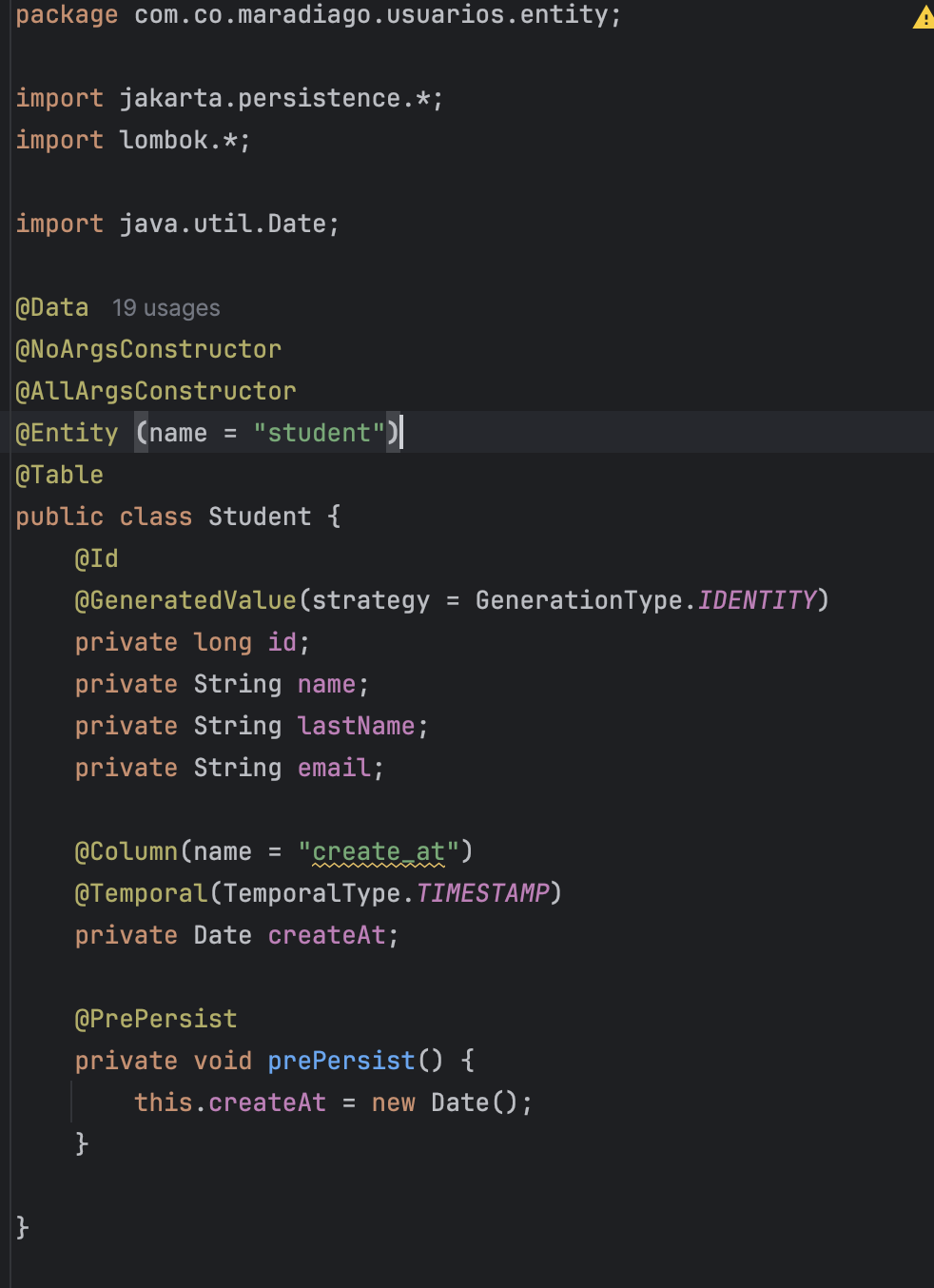


1. Colocaremos a nivel de clase los decoradores @Entity y @Table importándolo de la siguiente librería: jakarta.persistence
2. Agregamos los siguientes atributos de clase:

Id, nombre apellidos, email, fecha de creación tal como se ve en la siguiente imagen:



1. Debe crear un método que cada vez que guarde un nuevo alumno se almacene la fecha de creación de ese alumno de la siguiente forma:



1. Debe investigar y colocar en este documento el significado de cada uno de los decoradores que se encuentra en esta clase.

**Decoradores de JPA para Clases de Entidad y Tablas**

@Entity

Descripción: Marca la clase como una entidad JPA, lo que significa que será mapeada a una tabla en la base de datos.

Uso: Requerido para que JPA reconozca una clase como entidad. Generalmente, se usa en combinación con @Table y otros decoradores de JPA.

@Table

Descripción: Especifica detalles sobre la tabla en la base de datos a la que está mapeada la entidad, como el nombre de la tabla y otras propiedades.

Uso: Se usa junto con @Entity para personalizar el nombre de la tabla, aplicar restricciones de índices, y definir esquemas.

**Decoradores de JPA**

Estos son usados para mapear clases de Java a tablas en una base de datos relacional.

@Id

Descripción: Marca un campo como la clave primaria de la entidad.

Uso: Se usa en una clase de entidad (es decir, una clase que representa una tabla en la base de datos) para identificar qué campo será la clave primaria.

@GeneratedValue

Descripción: Indica que el valor del campo será generado automáticamente por la base de datos o por JPA.

Uso: Suele usarse junto con @Id para que la clave primaria se genere automáticamente, ya sea secuencialmente, aleatoriamente o con alguna estrategia definida.

Estrategias:

GenerationType.AUTO: Deja que el JPA decida la estrategia de generación basada en la base de datos.

GenerationType.IDENTITY: Usa una columna de auto-incremento en la base de datos.

GenerationType.SEQUENCE: Usa una secuencia en la base de datos.

GenerationType.TABLE: Usa una tabla específica para manejar la generación de claves.

@Column

Descripción: Especifica el nombre y las propiedades de la columna en la base de datos a la que se mapeará el campo de la entidad.

Uso: Se usa para personalizar los detalles de una columna, como el nombre, el tamaño, si es única, si es nullable, etc.

@Temporal

Descripción: Indica que el campo está asociado a un tipo de dato Date o Calendar y especifica si representa solo una fecha (DATE), solo una hora (TIME), o ambas (TIMESTAMP).

Uso: Se usa para controlar cómo se almacena un campo de tipo fecha o tiempo en la base de datos.

@PrePersist

Descripción: Marca un método que se ejecutará antes de que la entidad sea persistida en la base de datos.

Uso: Útil para inicializar o modificar valores antes de guardar el objeto en la base de datos, como establecer una fecha de creación.

1. Agregar los métodos getter and setter, además del constructor con todos los campos y el constructor por defecto vacío y sobreescribir el método ToString (este paso se omitió porque se utilizaron los decoradores con Lombok)

**Decoradores de Lombok**

Estos decoradores son proporcionados por la librería Lombok y ayudan a reducir la cantidad de código repetitivo en clases de Java, especialmente en clases de modelo.

@Data

Descripción: Genera automáticamente los métodos getters, setters, equals, hashCode, y toString para todos los campos de la clase.

Uso: Es útil en clases de modelo o entidades donde necesitas estos métodos sin escribirlos manualmente.

@NoArgsConstructor

Descripción: Genera un constructor sin argumentos para la clase.

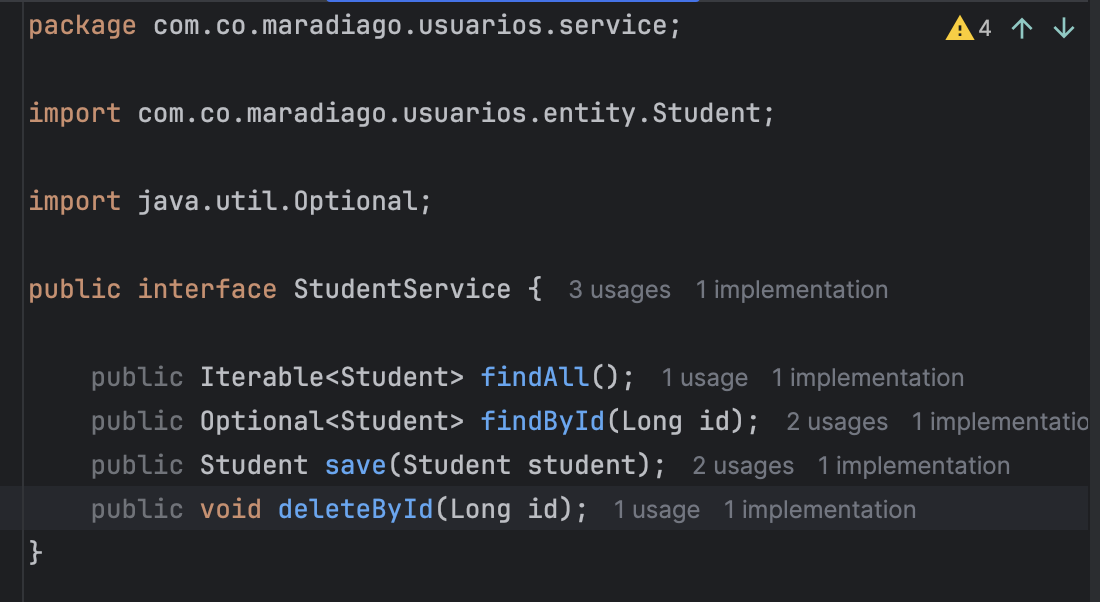
Uso: Es útil cuando necesitas un constructor vacío, especialmente en el contexto de JPA, ya que se requiere un constructor sin argumentos para las entidades.

@AllArgsConstructor

Descripción: Genera un constructor que incluye todos los campos de la clase como parámetros.

Uso: Es útil para crear fácilmente instancias de una clase con todos los valores inicializados.

1. Luego creamos un nuevo paquete con el nombre com.co.(su apellido).usuarios.repository dentro de éste creamos una interfaz con el nombre AlumnoRepository que extienda de CrudRepository
2. Luego creamos otro paquete con el nombre com.co.(su apellido).usuarios.service y dentro de este una interfaz con el nombre AlumnoService y dentro de ella los siguientes métodos abstractos:



1. Luego creamos dentro de ese mismo paquete una clase concreta que implemente la interfaz y por ende que implemente los métodos concretos de esa interfaz, esa clase debe estar con el decorador service, además a nivel de cada método colocaremos el decorador Transactional indicando si es no de lectura o escritura tal como se ve en la siguiente imagen:



1. Investigar y colocar dentro del este documento el concepto del decorador Transaction.

**@Service**

**Descripción:** Es una anotación de Spring que marca una clase como un "Service" (servicio) en la capa de servicio de la aplicación.

Uso: Se utiliza principalmente en la capa de negocio, donde se colocan las clases que contienen la lógica de negocio o procesamiento de datos. Aunque es similar a @Component (también registra la clase como un bean en el contenedor de Spring), @Service se usa por convención para mejorar la claridad del código, indicando que la clase está en la capa de servicio.

**Beneficios:**

* Hace que la clase sea reconocida como un bean de Spring, permitiendo la inyección de dependencias y otras funcionalidades de Spring.
* Facilita la separación de capas (controladores, servicios, repositorios) al mejorar la legibilidad y mantener una estructura clara.

**@Autowired**

Descripción: Es una anotación de inyección de dependencias en Spring. Permite que Spring inyecte automáticamente una instancia de un bean en otro bean.

**Uso:** Generalmente, se usa para inyectar dependencias en una clase (ya sea en atributos, constructores o métodos). Spring buscará automáticamente el bean adecuado y lo inyectará. Si hay solo un candidato para inyección, lo usará directamente; si hay más de uno, se puede combinar con @Qualifier para especificar cuál se debe inyectar.

**Beneficios:**

* Simplifica la inyección de dependencias, permitiendo que el contenedor de Spring maneje la creación y la administración de objetos.
* Permite la inyección de dependencias sin necesidad de configuración XML, lo cual mejora la modularidad y el mantenimiento del código.

**Inyección en el constructor o en el setter**: Además de inyectar en campos, también puedes usar @Autowired en el constructor o en métodos setter, lo cual es una práctica recomendada porque facilita la prueba y el diseño orientado a interfaces.

**@Transactional**

**Descripción:** Marca un método o una clase como transaccional, lo que significa que las operaciones de base de datos realizadas dentro de ese contexto se ejecutarán en una única transacción.

**Uso:** Generalmente, se usa en métodos de servicio que ejecutan operaciones de base de datos y deben garantizar que se completen en su totalidad o se reviertan en caso de error. Si el método marcado con @Transactional falla, cualquier cambio que haya realizado en la base de datos será revertido (rollback), asegurando la consistencia de los datos.

**Beneficios:**

* Asegura la atomicidad de las operaciones: Al utilizar transacciones, todas las operaciones de base de datos en un método @Transactional se consideran una única unidad de trabajo. Si una operación falla, se deshacen todas las demás.
* Aislamiento: Puedes configurar el nivel de aislamiento de la transacción para controlar la visibilidad de los datos entre transacciones concurrentes, evitando problemas como lecturas sucias y repeticiones inconsistentes.
* Manejo automático de errores: Al combinarse con excepciones, permite que el código se recupere automáticamente de ciertos fallos mediante el rollback.

**Nivel de transacción**: @Transactional puede configurarse con varios atributos, como propagation (controla cómo se propaga la transacción) y isolation (controla el nivel de aislamiento).

1. Inyectamos en esta clase la interface AlumnoRepository e implementamos cada uno de los métodos como se muestra en la siguiente imagen:



1. Creamos ahora el paquete controller y dentro de él una clase concreta con el nombre AlumnoController y decorada a nivel de la clase con la etiqueta @RestController.
2. Dentro del controlador inyectamos la interface AlumnoService y allí implementamos los métodos que se encuentran en la siguiente imagen



1. Definir que significa los siguientes conceptos en Spring Boot :
2. ResponseEntity<?>
3. Optional
4. @PathVariable

**a. ResponseEntity<?>**

Descripción: ResponseEntity es una clase de Spring Framework que representa toda la respuesta HTTP, incluidas las cabeceras (headers), el cuerpo y el código de estado (status code). El <?> es un comodín que indica que puede contener cualquier tipo de dato.

Uso: Se utiliza en controladores de Spring para devolver respuestas personalizadas a los clientes de una API REST. Con ResponseEntity, puedes especificar tanto el cuerpo de la respuesta como el estado HTTP, lo cual permite un mayor control sobre cómo se responde a las solicitudes.

Beneficios:

Permite configurar el estado HTTP de la respuesta (por ejemplo, 200 OK, 404 Not Found, 500 Internal Server Error, etc.).

Facilita la adición de cabeceras HTTP personalizadas.

Proporciona un mecanismo para retornar diferentes tipos de datos en el cuerpo de la respuesta, gracias al uso de tipos genéricos (<?>).

Uso del tipo genérico (<?>): <?> significa que puedes devolver cualquier tipo de objeto, lo cual permite que ResponseEntity se adapte a diferentes tipos de datos según el contexto.

**b. Optional**

Descripción: Optional es una clase de Java 8 que representa un contenedor que puede o no contener un valor no nulo. Ayuda a evitar errores NullPointerException y promueve el manejo seguro de valores que podrían ser null.

Uso: Optional se usa principalmente para envolver un objeto que podría ser null. En el contexto de Spring y JPA, Optional se utiliza a menudo en métodos de repositorio (por ejemplo, findById) para indicar que el objeto que se devuelve podría estar ausente.

Beneficios:

Mejora la legibilidad del código al hacer explícito el hecho de que un valor puede estar ausente.

Facilita el manejo de valores null mediante métodos como isPresent(), ifPresent(), y orElse(), reduciendo la probabilidad de excepciones NullPointerException.

Permite realizar operaciones encadenadas para transformar o manejar el valor solo si está presente.

Manejo seguro de null: Optional ayuda a lidiar con valores potencialmente nulos sin causar errores, promoviendo un estilo de programación más seguro y limpio.

**c. @PathVariable**

Descripción: @PathVariable es una anotación de Spring MVC que se usa para extraer valores de la URI (ruta) de una solicitud HTTP y asignarlos a parámetros en un método de controlador.

Uso: Se utiliza en controladores REST para capturar variables de la URL. Por ejemplo, si tienes una URL como /persona/{id}, el valor {id} se puede obtener usando @PathVariable y pasar como argumento al método controlador.

Beneficios:

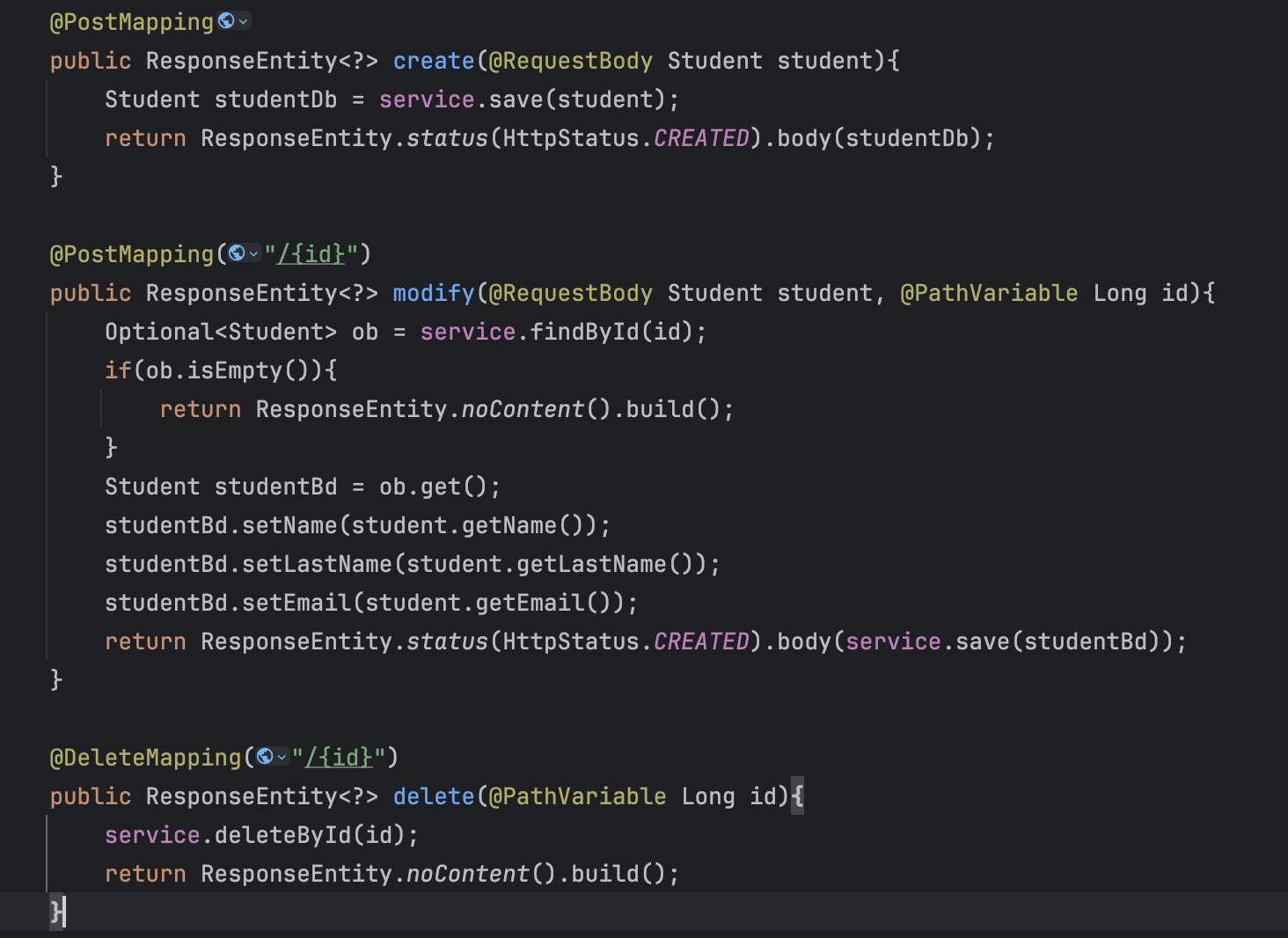
Permite hacer que las rutas sean más dinámicas y significativas, ya que puedes capturar partes de la ruta como parámetros.

Facilita el desarrollo de APIs RESTful que cumplan con convenciones de URLs REST.

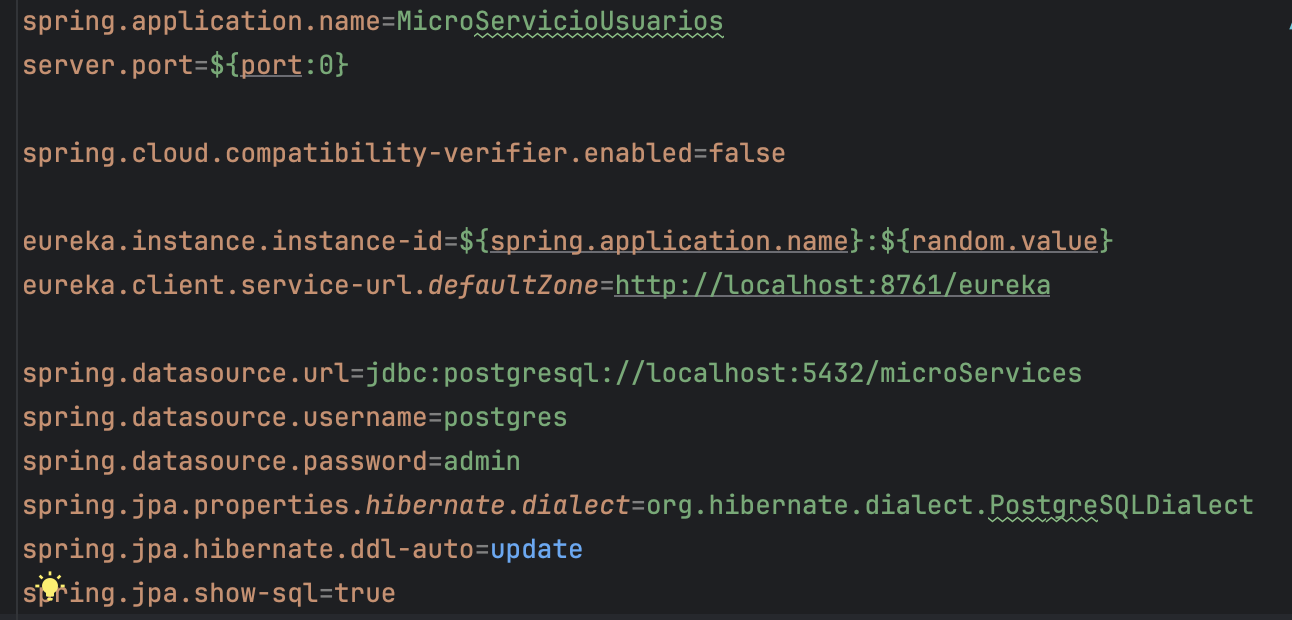
Ofrece flexibilidad para mapear directamente valores de la URL a variables en el controlador, simplificando el manejo de rutas dinámicas.

Parámetros de URL: @PathVariable es ideal para capturar valores directamente de la URL y es común en APIs RESTful, ya que permite capturar IDs o nombres de recursos directamente en las rutas.

Y los otros métodos son :



1. Crear el string de conexión a la bd de su preferencia, en mi caso es postgres y el string de conexión es el siguiente:



**CREACIÓN SERVIDOR EUREKA**

1. Investigar y colocar en este parte del documento lo que es el servidor EUREKA como parte de Spring Cloud

Eureka Server es un servicio de descubrimiento que forma parte de Spring Cloud Netflix, una suite de herramientas de Netflix OSS para aplicaciones distribuidas en el ecosistema de Spring Cloud. Es ampliamente utilizado en arquitecturas de microservicios para permitir que las aplicaciones encuentren y se comuniquen entre sí sin necesidad de una configuración manual de las ubicaciones de cada servicio. A continuación, te explico sus componentes y funcionalidades principales:

**1. Funcionalidad de Eureka Server**

Servicio de descubrimiento: Eureka actúa como un "registro de servicios" donde todos los microservicios se registran con su dirección y puerto. Otros servicios pueden consultar el registro para descubrir la ubicación de los servicios que necesitan.

Registro de servicios: Cuando un microservicio se inicia, se registra en Eureka Server, proporcionando su dirección IP, puerto y nombre de servicio. Esto permite que otros servicios lo descubran.

Balanceo de carga: Eureka, en combinación con otros componentes de Spring Cloud como Ribbon, puede distribuir las solicitudes de manera equitativa entre las instancias de un servicio, proporcionando balanceo de carga en las comunicaciones internas de la aplicación.

Alta disponibilidad y tolerancia a fallos: Eureka puede desplegarse en un clúster, y los servicios realizan "self-healing" (autorrecuperación), dado que cada instancia de servicio envía un "ping" o "heartbeat" a Eureka para indicar que está activa. Si un servicio no envía este "ping" en un tiempo determinado, Eureka lo elimina del registro, para evitar que otros servicios intenten conectarse a una instancia inactiva.

**2. Cómo funciona Eureka Server**

Registro de servicios (Service Registration): Cuando un microservicio se inicia, se registra en el Eureka Server. Este registro incluye la ubicación del servicio (IP y puerto) y un identificador único (por ejemplo, el nombre del servicio).

Descubrimiento de servicios (Service Discovery): Los microservicios cliente pueden consultar a Eureka para obtener la ubicación de otros servicios. Esto se hace mediante el EurekaClient, que permite que las aplicaciones encuentren automáticamente los servicios que necesitan consumir, usando el nombre de servicio como clave de búsqueda.

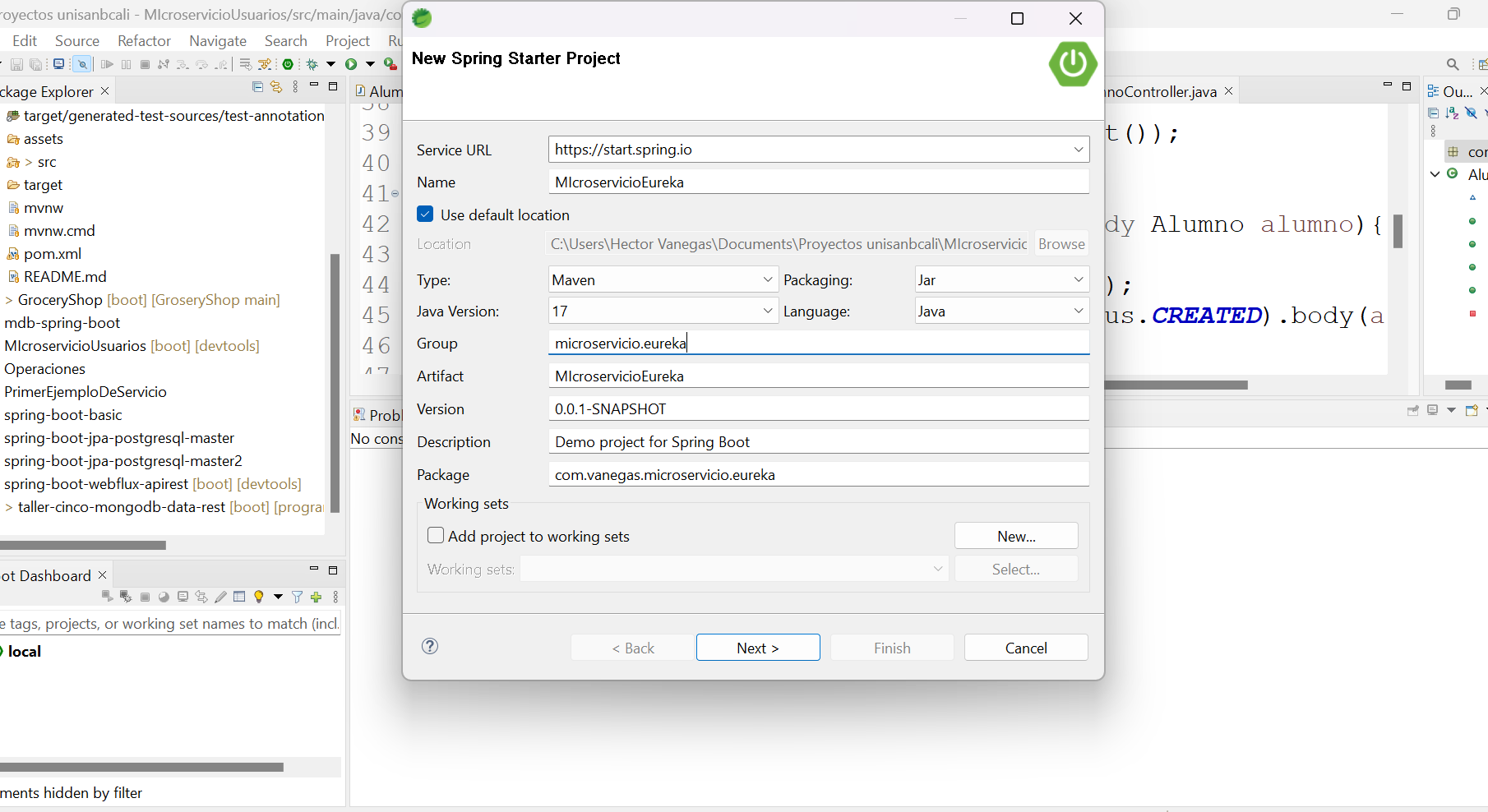
Actualización de registro (Heartbeats): Las instancias de servicio enviarán periódicamente un "heartbeat" al servidor Eureka para indicarle que están activas. Si el servidor no recibe este "heartbeat" dentro de un tiempo específico, marcará la instancia como inactiva y eventualmente la eliminará del registro, para evitar intentos de conexión a servicios que ya no están activos.

**3. Componentes de Eureka**

Eureka Server: Es el componente central que mantiene el registro de servicios y gestiona el descubrimiento. Lo configuras en tu aplicación Spring Boot mediante la dependencia spring-cloud-starter-netflix-eureka-server. Solo necesitas anotarlo con @EnableEurekaServer en la clase principal para activar el servicio.

Eureka Client: Los microservicios que necesitan descubrir otros servicios se configuran como "clientes de Eureka" y utilizan el @EnableEurekaClient para habilitar su registro en el servidor. Los clientes también se encargan de consultar a Eureka Server para descubrir otros servicios.

1. Se debe crea un nuevo proyecto con Spring.io tal y como hicimos el servicio anterior



Con las siguientes dependencias:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

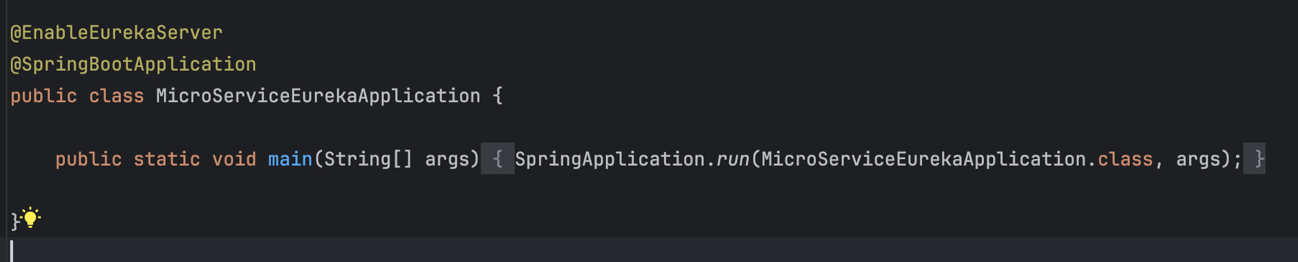
<dependency>

<groupId>org.glassfish.jaxb </groupId>

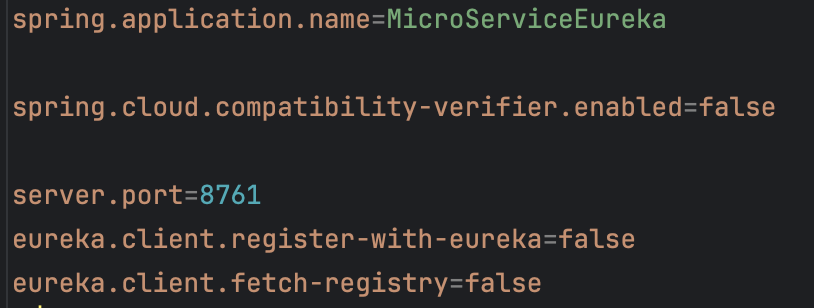
<artifactId>jaxb-runtime</artifactId>

</dependency>

1. Una vez creado el proyecto coloco el decorador de Eureka a nivel de la clase principal del proyecto como se muestra a continuación:



1. En el archivo de propiedades colocamos la siguiente configuración:



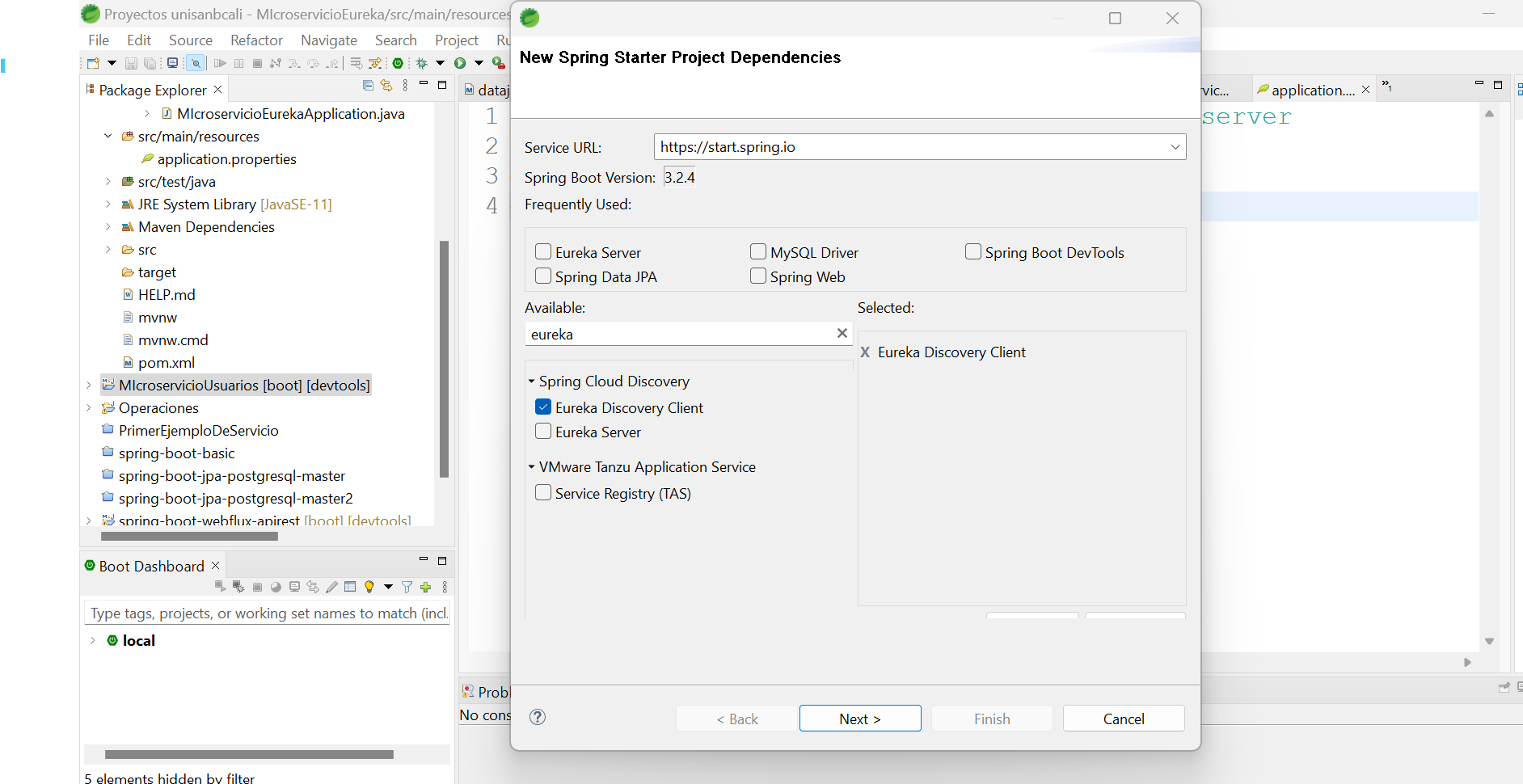
Y eso es todo de momento con el servicio de Eureka

1. Volvemos al servicio de Usuarios para agregar la dependencia de eureka cliente, se la siguiente forma

Interfaz de usuario gráfica, Aplicación, Word

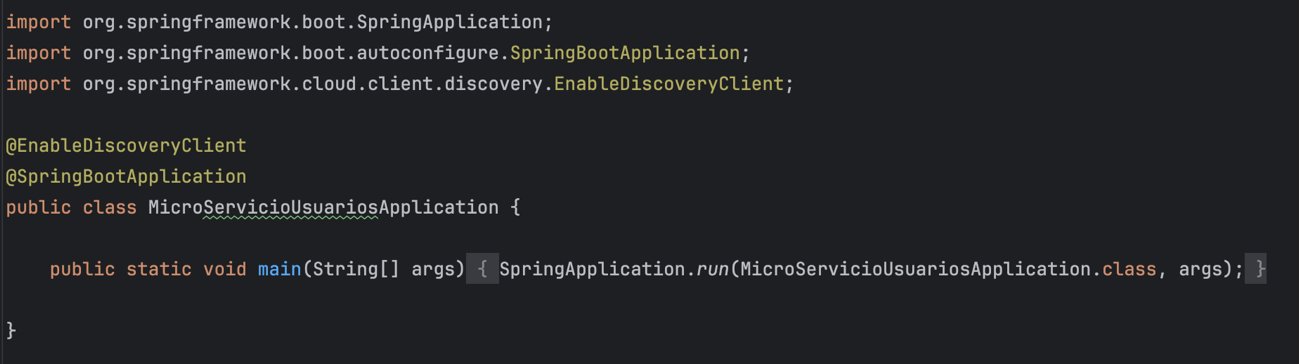
Descripción generada automáticamente

En la imagen anterior queremos editar las dependencias y agregamos la dependencia de EurekaClient

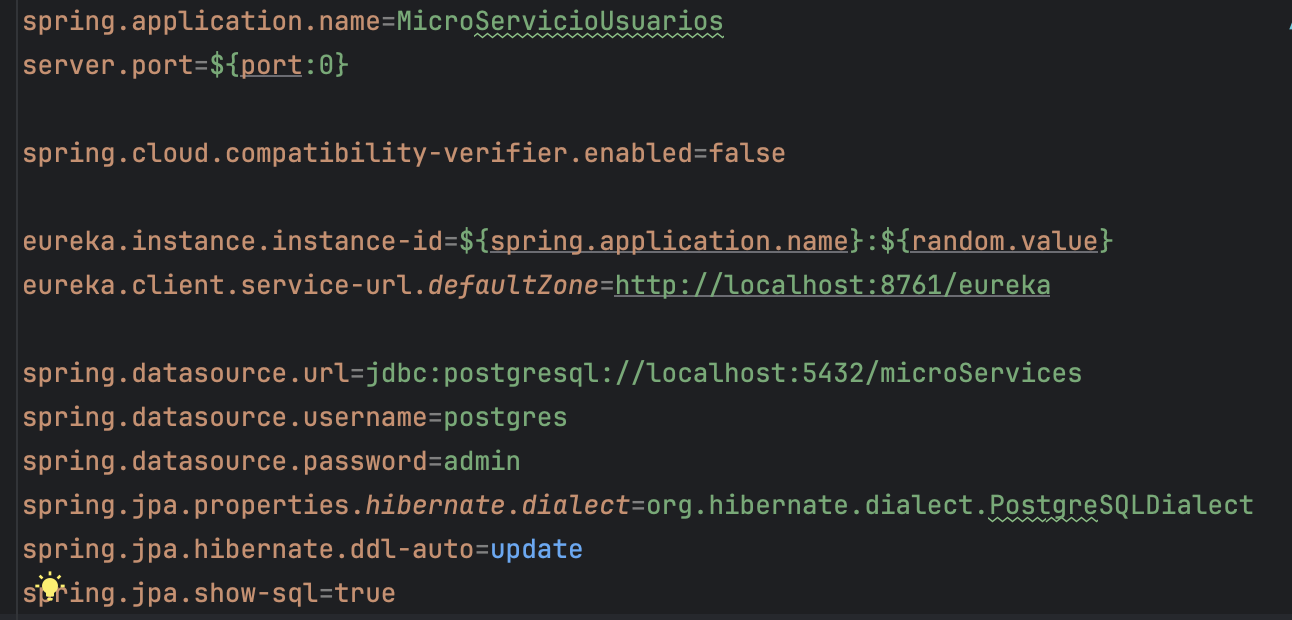


Luego se actualiza el pom de ese proyecto.

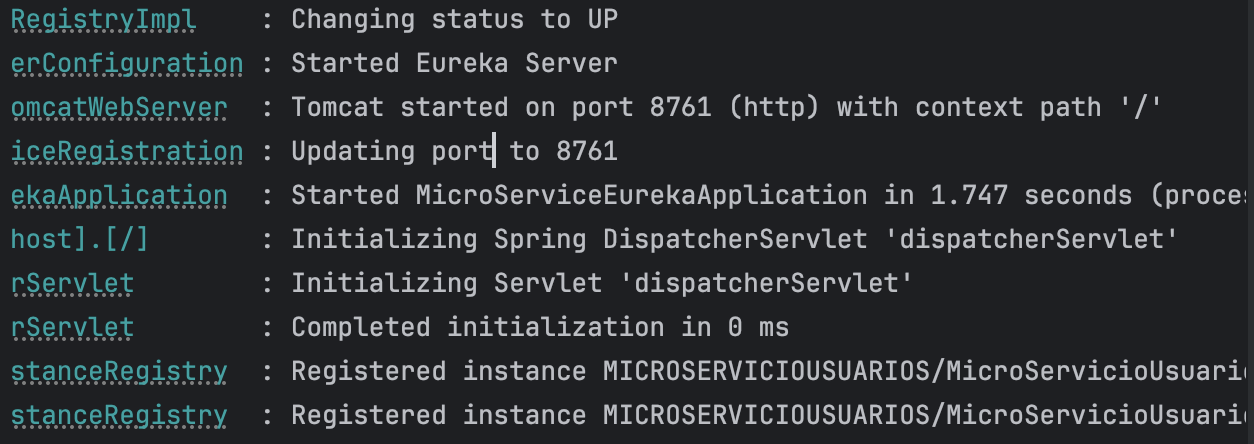
1. Se agrega el decorador de eureka cliente en la clase principal como se ve en la siguiente imagen



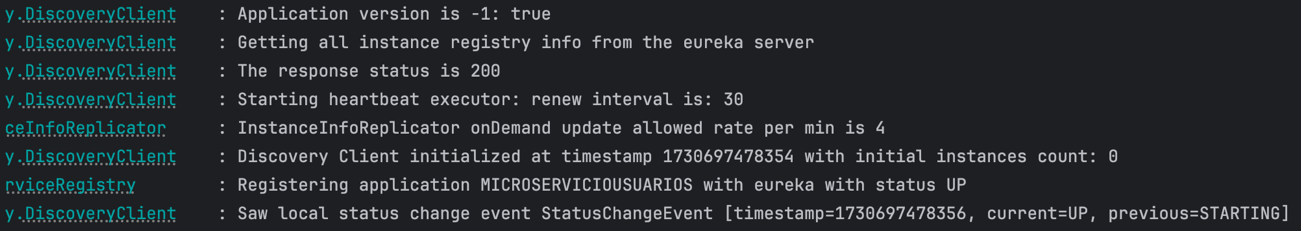
1. Le damos un nombre a este servicio a través del archivo de propiedades y le colocamos un puerto RAMDOM para escalar, como se ve a continuación:



1. Se debe levantar primero el servicio de Eureka y luego el servicio de Usuarios
2. Probar que todo esta bien, primero debe salir el log en el ms de eureka el siguiente log:



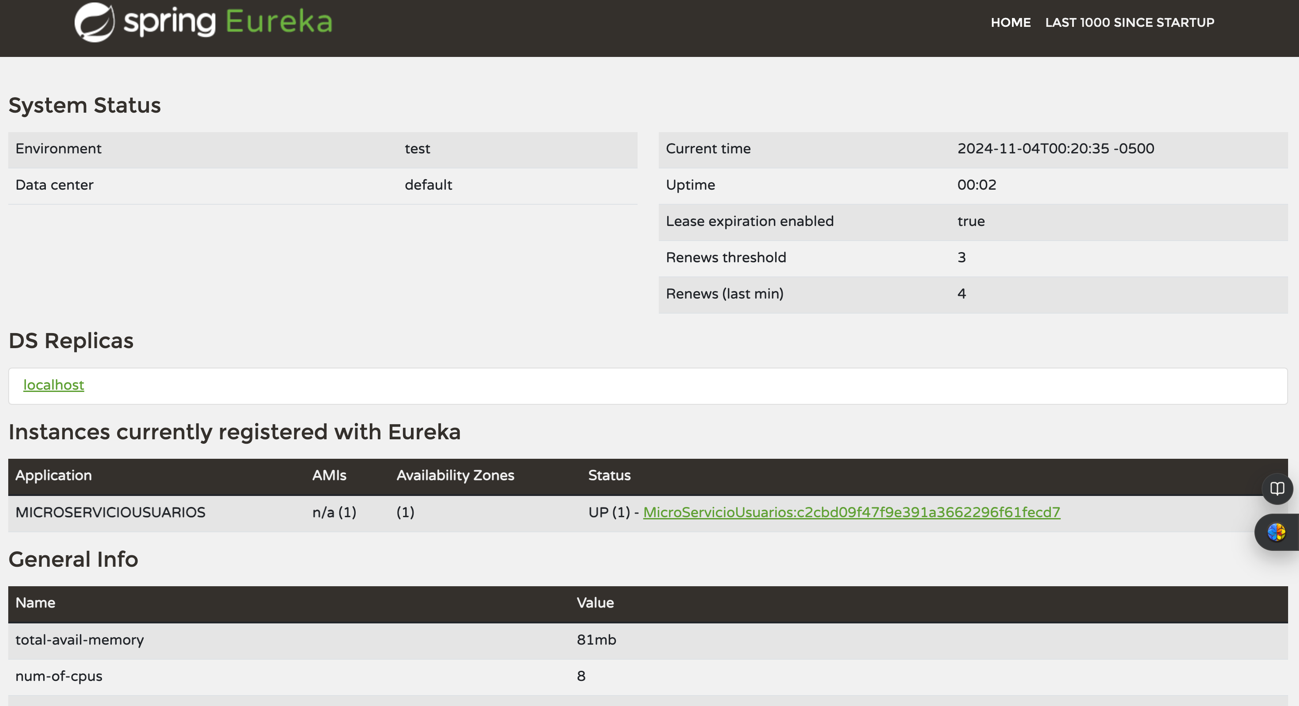
Luego abrimos el log del servicio de Usuarios y debe ver que esta enviadno señal al servidor de nombres cada 30 segundos como se ve en la imagen



Y por ultimo vamos al navegador y colocamos la siguiente url

<http://localhost:8761>

y deberá mostrar lo siguiente:



Si has llegado hasta aquí, acabas de completar el primer reto de esta actividad , donde has creado un servicio y lo has registrado en un servidor de nombre de Spring Cloud