Servicio Web Rest

29/11/2021

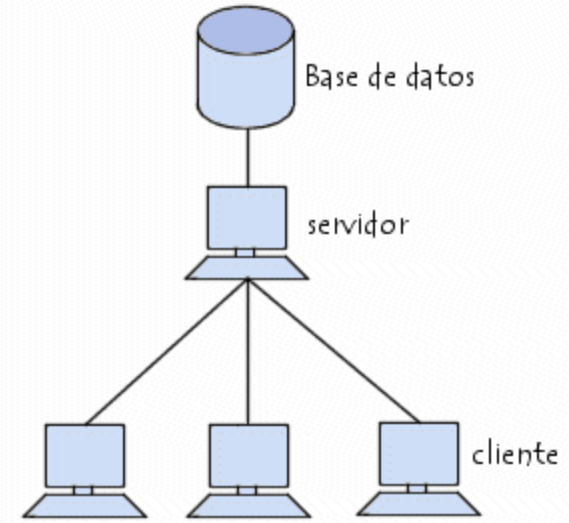
Miguel Ángel Parejo Morillas
IES Julio Verne
Programación de Servicios y Procesos

Servicio Web Rest

En esta práctica, he desarrollado una Api Rest, con el framework Spring Boot. Ésta Api consta de 3 partes interrelacionadas, las cuales definen la estructura de éste documento. Iré explicando de manera detallada la estructura de cada una de estas partes y explicando el código que las compone:

- Base de datos: La base de datos que se pide en el enunciado de la práctica es MySQL. Para conectarse a ésta, he usado el cliente "MySQL Workbench", conectándose a mi base de datos mediante "root", "root".
- Aplicación Java: Mi aplicación en java está desarrollada en el IDE "Spring tool Suite 4", el cual es una versión mejorada de Eclipse, adaptada a éste framework.
- Cliente: El cliente usado para pedir recursos a la API es "PostMan". Los datos se muestran en JSON.

Como he dicho anteriormente, éstos 3 componentes son los que conforman ésta API REST(más bien serían la BD y el programa, el cliente únicamente accede al servicio, pero es necesario para probar la aplicación).



Base de datos

Una vez entrado en materia, el primer paso que debemos hacer para la creación de nuestra API, es el diseño de nuestra base de datos, donde se recogerán y almacenarán los objetos que se muestran en el cliente por el controlador. Mi base de datos se llama "ServicioWebRest", contiene 3 tablas(Alumno, profesor y grupo).

Relaciones

Alumno - Grupo

Un Alumno pertenece a un Grupo.

En un Grupo pueden estar matriculados varios Alumnos.

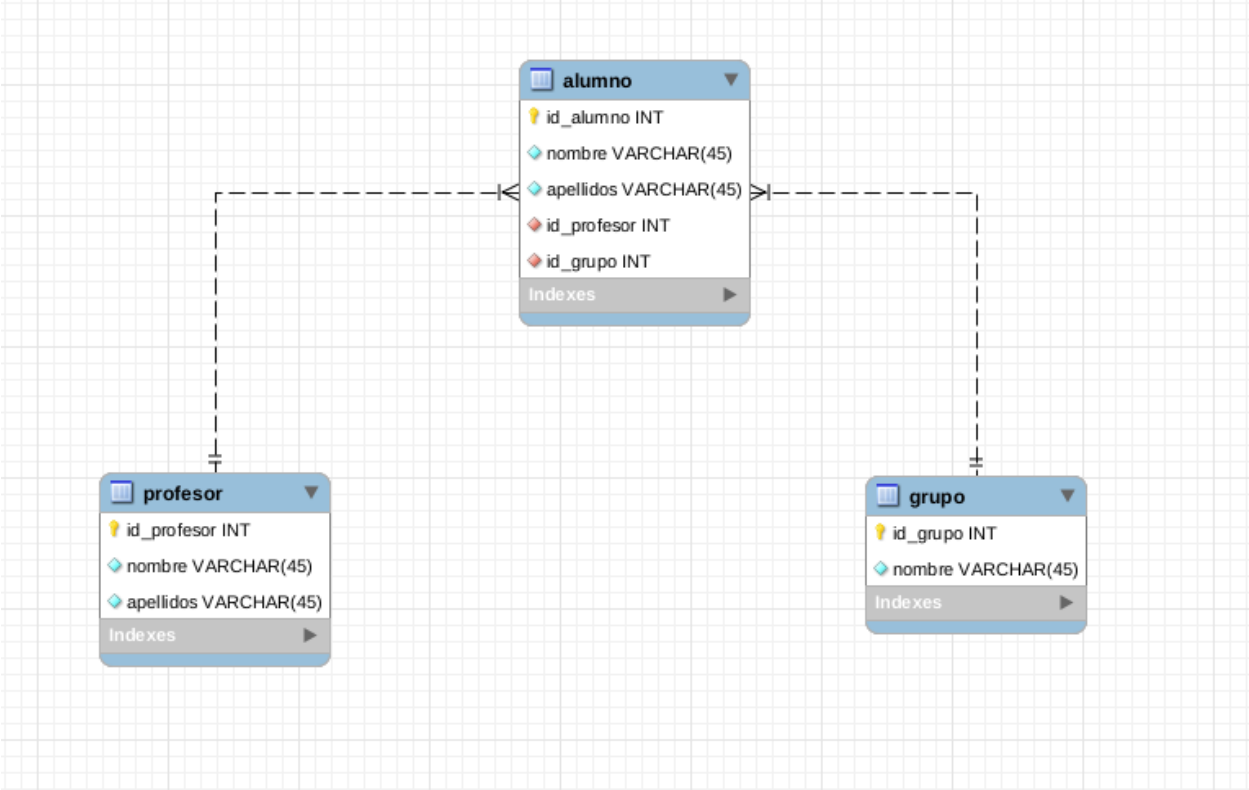
Alumno - Profesor

A un Alumno le imparte clase un profesor.

Un Profesor imparte clase a varios Alumnos.

Modelo E/R

El modelo Entidad/Relación es el siguiente:



Como se puede observar, las dos relaciones son 1:N. También cabe destacar que Workbench, usa como nombre por defecto para las Foreign Keys el nombre de la tabla padre + nombre del campo de la clave primaria de la clase padre. Spring boot es muy delicado con el tema de las nomenclaturas de los campos de la BD, por lo cual es conveniente que se le cambie el nombre a las claves foráneas. También es importante que ninguna de las tablas comiencen llamándose por mayúscula, ya que puede dar problemas.

También he incluido el archivo de creación de la base de datos (archivo que acaba en .mwb).

Aplicacion Spring boot con Java

Una vez que tengo la BD diseñada correctamente, queda crear la aplicación con spring boot para poder gestionar los datos. Para ello, he descargado el Entorno de desarrollo "Spring tools Suite 4" de la página oficial.

Para explicar esta parte, no voy a ir poco a poco, ya que spring tiene mil y una anotaciones y configuraciones que dependiendo de cada caso se usarán unas u otras. Por lo que, voy a mostrar directamente mi jerarquía de proyecto, he iré explicando de menor a mayor nivel cada una de las clases que lo componen, para que, al finalizar la explicación, pasemos a la parte del cliente y veamos el funcionamiento del programa que he detallado. Sin más preámbulo, voy a mostrar en primer lugar la configuración básica de la aplicación.

POM

En el archivo POM, he incluido las siguientes dependencias:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.modelmapper</groupId>
        <artifactId>modelmapper</artifactId>
        <version>2.3.5</version>
    </dependency>
</dependencies>
```

He necesitado tanto las dependencias necesarias para que el framework de Spring Boot funcione, como las dependencias para conectar la base de datos MySQL, model Mapper, para poder tratar los objetos con las clases DTO y convertir las entidades de la base de datos en objetos de la aplicación y también para poder dar una vista personalizada a éstos objetos en el cliente en formato JSON y por último, lombok. En ésta última dependencia, no bastó únicamente con agregar esas líneas de código, también tuve que descargar el instalador en ".jar" e instalarlo en proyecto.

Application Properties

En éste archivo, es donde indico la configuración general necesaria para que la BD se conecte correctamente a la base de datos que tengo creada en el ordenador(localhost), tiene el siguiente código:

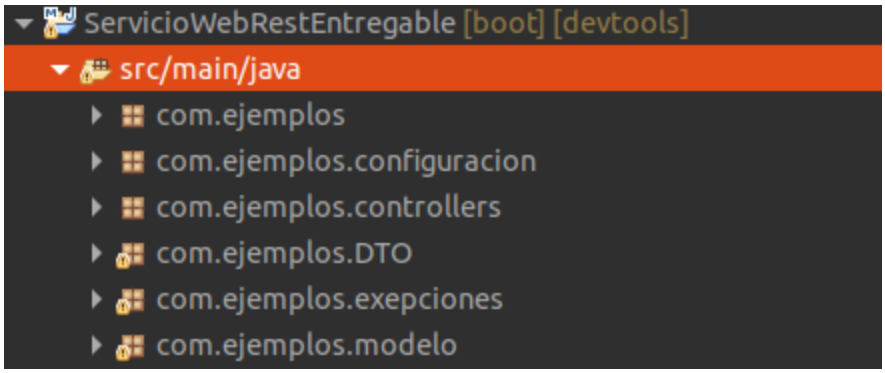
```
1  server.port=8099
2  #Data source
3  #Indica el driver/lib para conectar java a mysql
4  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
5
6  #Url donde esta el servicio de tu mysql y el nombre de la base de datos
7  spring.datasource.url=jdbc:mysql://localhost:3306/ServicioWebRest
8
9  #Usuario y contrasena para tu base de datos descrita en la linea anterior
10 spring.datasource.username=root
11 spring.datasource.password=root
```

He indicado que el puerto de la API Rest será el 8099, como se indica en el enunciado de la entrega. También es necesario indicar el driver de mysql, la URL de la base de datos, la cual debe corresponderse con el nombre de la base de datos que hemos elegido a la hora de crearla en WorkBench. Por último, las credenciales de conexión a dicha BD.

Nota: En la imagen aparecen como credenciales root, root. Sin embargo, el proyecto que se ha entregado tiene configurado ya el usuario, usuario. Como indica la práctica.

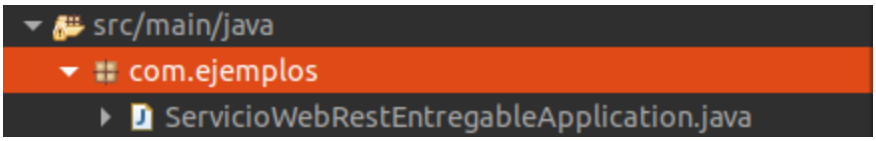
Estructura del proyecto

Una vez detallada y explicada la configuración general, voy a pasar con las clases que componen mi API Rest. Los paquetes están ordenados de la siguiente manera:

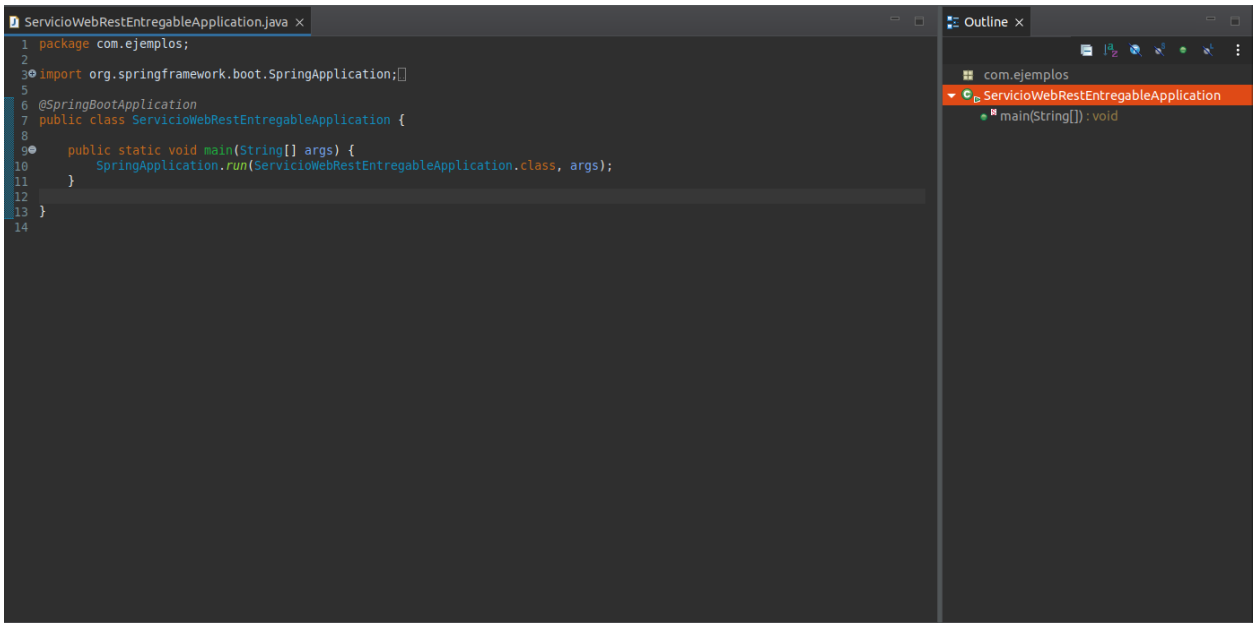


Paquete "com.ejemplos"

En éste paquete se encuentra la siguiente clase:



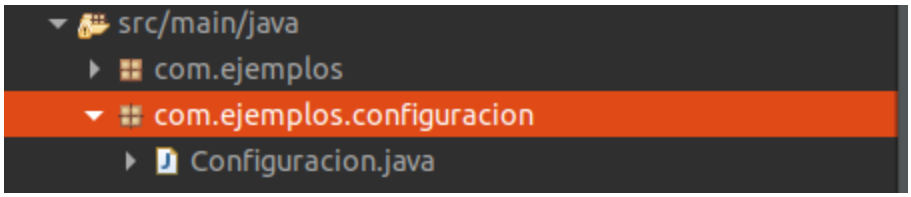
Tiene el siguiente código:



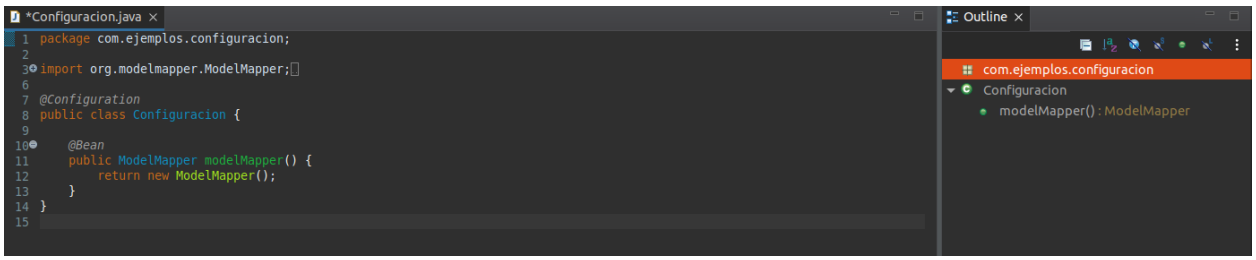
Éste código es el que inicia la API de manera automática, se genera solo cuando añadimos las dependencias de Spring Boot y no necesitamos tocar nada.

Paquete “com.ejemplos.configuración”

En éste paquete se encuentra la siguiente clase:



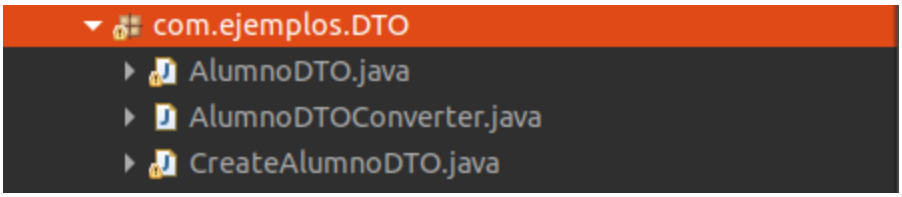
Tiene el siguiente código:



Ésta clase únicamente sirve como configuración de la dependencia “ModelMapper”. Si no incluimos ésto, no podremos hacer uso de manera automática de los DTO y tendremos que instanciarlos manualmente.

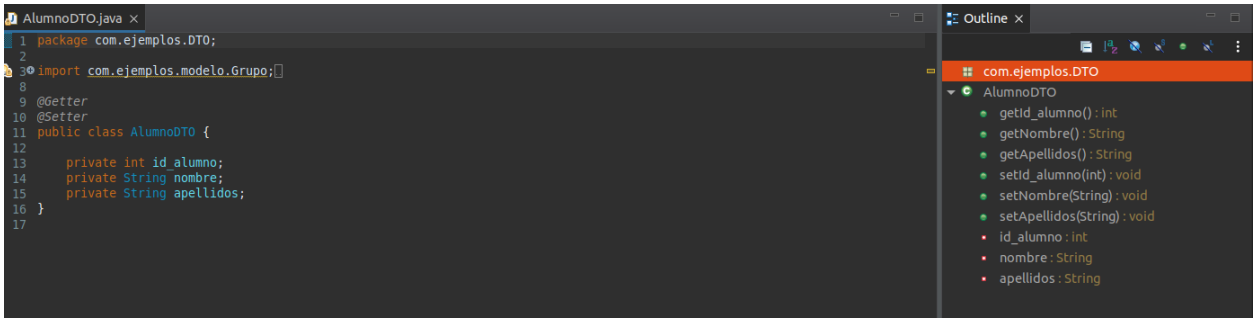
Paquete “com.ejemplos.DTO”

En éste paquete se incluyen las siguientes 3 clases:



Aquí se pueden diferenciar dos tipos de clases. El primer tipo son las dos clases DTO(AlumnoDTO.java y CreateAlumnoDTO.java). Ambas clases sirven para poder tratar los objetos con un formato determinado. De esta manera, podremos tratar en la aplicación java los objetos de una forma en concreto(usando AlumnoDTO.java) y recogerlos del cliente con otro formato distinto(usando CreateAlumnoDTO.java). Para poder realizar estas conversiones, se ha creado entre el controller y éstas clases, un converter. Veamos el código de cada una de las clases para ver el formato de cada una de las vistas.

AlumnoDTO



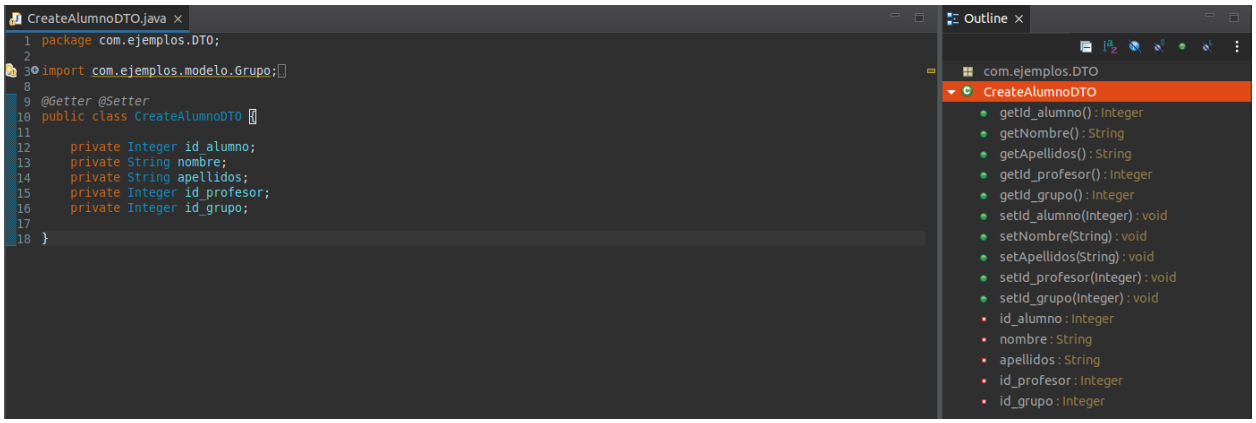
Como se puede observar en la imagen, se han añadido dos anotaciones de la dependencia "Lombok", "@Getter" y "@Setter". Estas dos anotaciones sirven para que lombok cree y gestione de manera automática los getters y setters que se usan para acceder a los datos que se recogen o muestran en el cliente. Ésto se puede confirmar mirando a la derecha, el apartado "OutLine". Ésta pestaña muestra los métodos y atributos de la clase. Como cabe esperar después de incluir las anotaciones, aparte de los 3 atributos, se contemplan getters y setters de todos ellos, sin necesidad de escribirlos.

Éste DTO se usará para mostrar los datos de un Alumno con el formato:

```
{
   "id_alumno":Integer,
   "nombre": String,
   "apellidos": String
}
```

El objeto ALumno, como hemos visto en la creación de la BD y como veremos más adelante en el modelo, tiene más atributos, sin embargo, si no lo incluimos en ésta clase, no se mostrarán, aunque si se pasan al cliente.

CreateAlumnoDTO



Éste DTO también incluye las dos anotaciones de lombok para los métodos getters y setters. Además, tiene un modelo distinto. Esto se debe a que esta clase es usada para recoger un objeto de tipo Alumno, el cual está siendo enviado desde el cliente, ya sea para editar un alumno con un determinado ID o para crear uno nuevo. Los datos que deberá introducir el usuario tendrán que tener el siguiente formato:

```
{

   "id_alumno":Integer,

   "nombre": String,

   "apellidos": String,

   "id_profesor":{

        "id_profesor": Integer

   },

   "id_grupo":{

        "id_grupo":Integer

   }

}
```

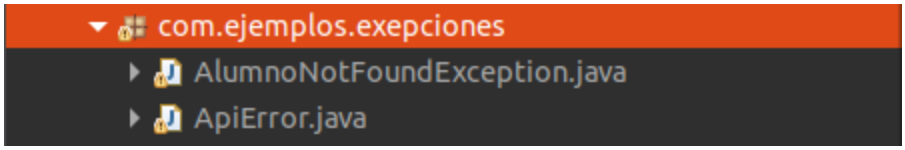
AlumnoDTOConverter

En ésta clase se han añadido dos anotaciones de dos dependencias distintas. La primera anotación, "@Component", es propia de Spring. Y la segunda, es de Lombok. "@RequiredArgsConstructor" es una anotación que sirve para que Lombok gestione de manera automática, como aparece en OutLine, el método constructor.

Después tenemos dos métodos, uno para convertir a partir de un objeto de tipo Alumno, el cual hace referencia al formato de la BD, a un objeto de tipo AlumnoDTO. Ésto servirá para que en caso de que el cliente haga una petición "Get" a la API, ésta transforme el objeto de tipo Alumno a uno de tipo DTO y se lo envíe a la vista. Así podremos mostrar el alumno con el formato definido en la clase "AlumnoDTO". El otro método es justamente para lo contrario, es decir, cuando desde el cliente, el usuario introduzca con el formato definido en la clase "CreateAlumnoDTO" los datos del alumno, estos serán tratados y convertidos al objeto de tipo Alumno, el cual se corresponde al formato de la base de datos. Ésto se usará para que en caso de que se realice una petición "Post" o "Put"(crear o editar, respectivamente), se pueda hacer dicho tratamiento a la información.

Paquete "com.ejemplos.exepciones"

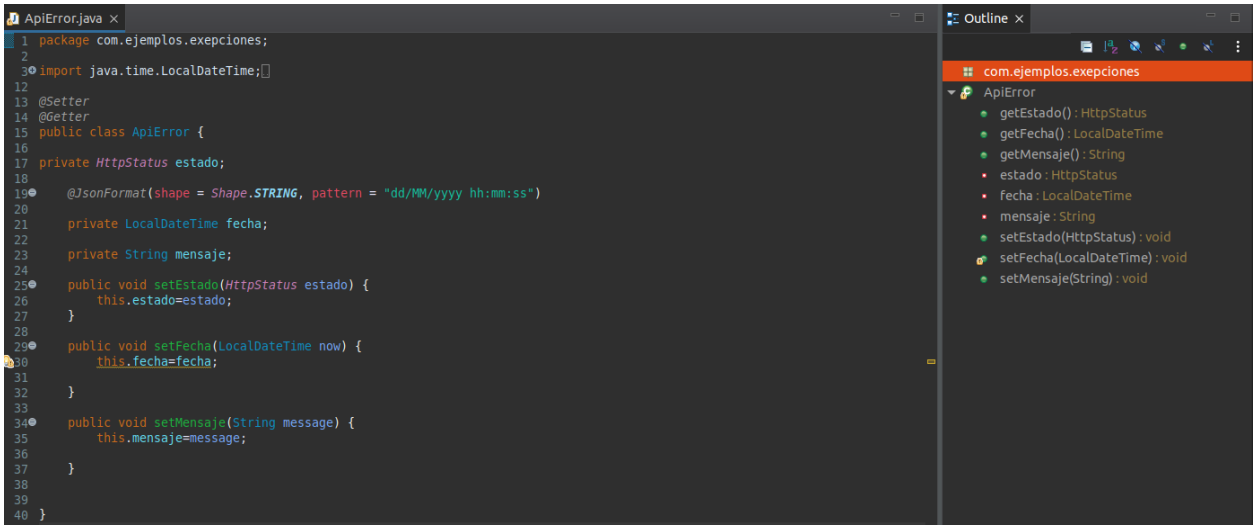
En éste paquete se encuentran las siguientes 2 clases:



Desde éste paquete trataré las excepciones que crea conveniente. De ésta forma, podré determinar con qué errores se muestran qué mensaje, además de poder incluir metadatos en la información del mensaje del error. Para entenderlo mejor, mostraré el código de ambas clases.

También podré modificar que dependiendo la excepción que lance el error, la API envíe el código de error que crea más conveniente.

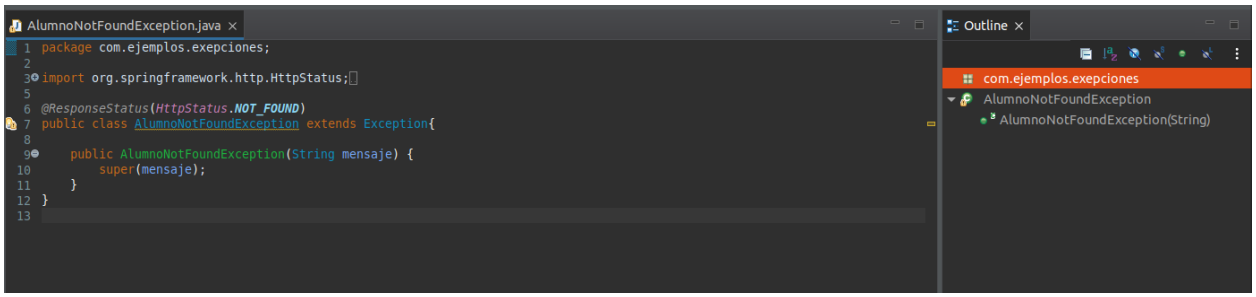
ApiError



Como se puede apreciar en la imagen, ésta clase también incluye los getters y setters de Lombok.

Ésta clase sirve para poder gestionar el estado de error http que envía la API al cliente en caso de que se dé una excepción que yo controle mediante código. Es decir, cuando en el controlador se lance una determinada excepción que haya controlado, se creará un objeto de éste tipo, y se enviará al cliente como cuerpo o body de la respuesta, con el estado de error que yo haya controlado.

AlumnoNotFoundException



En ésta clase simplemente defino la excepción personalizada para crearla y lanzarla cuando yo crea conveniente en el controlador.

Paquete “com.ejemplos.modelo”

En el paquete modelo, incluye tanto las clases que definen las entidades de mi BD en el programa, como los repositorios que incluyen los métodos de JPA, para poder realizar consultas y operaciones en la BD. En primer lugar, voy a mostrar el código de los 3 repositorios(alumnoRepository, ProfesorRepository y GrupoRepository), ya que comparten la misma sintaxis y no hay que especificar ningun metodo ni atributo, simplemente añadir las anotaciones necesarias para que la clase incluya la dependencia de JPA, la cual es gestionada por Spring. El código de las clases es el siguiente:

*AlumnoRepositorio.java

```java
package com.ejemplos.modelo;

import org.springframework.data.jpa.repository.JpaRepository;

public interface AlumnoRepositorio extends JpaRepository<Alumno, Integer>{

}

```

*GrupoRepositorio.java

```java
package com.ejemplos.modelo;

import org.springframework.data.jpa.repository.JpaRepository;

public interface GrupoRepositorio extends JpaRepository<Grupo, Integer>{

}

```

*ProfesorRepositorio.java

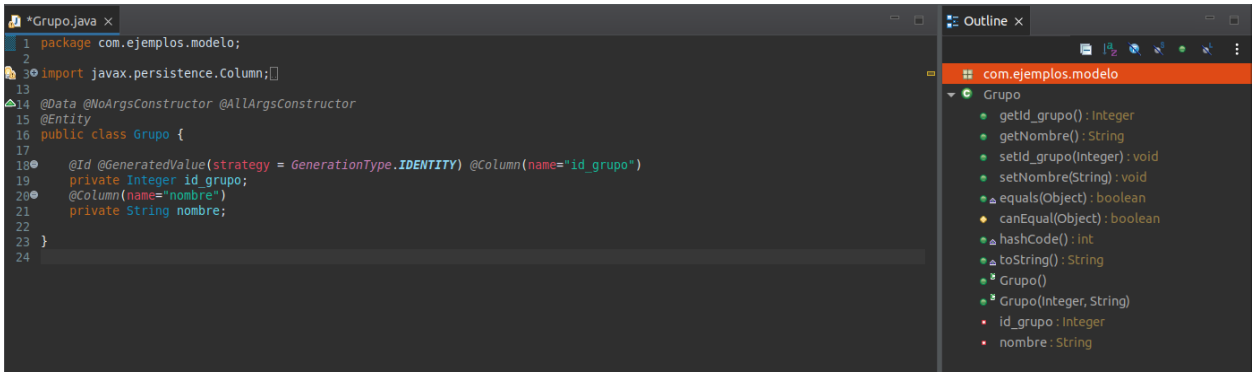
```java
package com.ejemplos.modelo;

import org.springframework.data.jpa.repository.JpaRepository;

public interface ProfesorRepositorio extends JpaRepository<Profesor, Integer>{

}

```

Como se puede observar en las imágenes, éstas clases únicamente extienden de JpaRepository. En ésta extensión, se le indica tanto el tipo de objeto(definido en el modelo con las clases Alumno, Grupo y Profesor respectivamente) y el tipo de dato que va a componer la clave primaria de dicha clase.

Una vez definido esto, ya disponemos de los métodos propios de JPA, sin necesidad de implementar nada.

Ahora mostraré las clases Profesor y Grupo, las cuales no tienen claves foráneas, y por lo tanto, no contienen relaciones en sí mismas, con lo cual son las dos clases más simples.

Clases Profesor y Grupo



En ambas clases, se incluyen las mismas anotaciones. Las cuales realizan la siguiente función:

- @DATA → Anotación de lombok, la cual incluye los métodos que se pueden ver en el OutLine, es decir, getters y setter. Con esta anotación también estamos haciendo uso de @RequiredArgsConstructor y otras más.
- @NoArgsConstructor → Anotación de lombok, la cual incluye un constructor de la clase vacío.
- @AllArgsConstructor → Anotación de lombok, la cual incluye un constructor de la clase que pide por parámetros todos los atributos de dicha clase.
- @Entity → Anotación de la librería "Persistance", la cual indica que esa clase será la que definirá una tabla del modelo de mi BD.

Cuando nos adentramos en el código de la clase, podemos observar que tenemos dos tipos de atributos, la clave primaria de la tabla y los atributos que hacen referencia a campos simples de la tabla.

La clave primaria tiene los siguiente atributos:

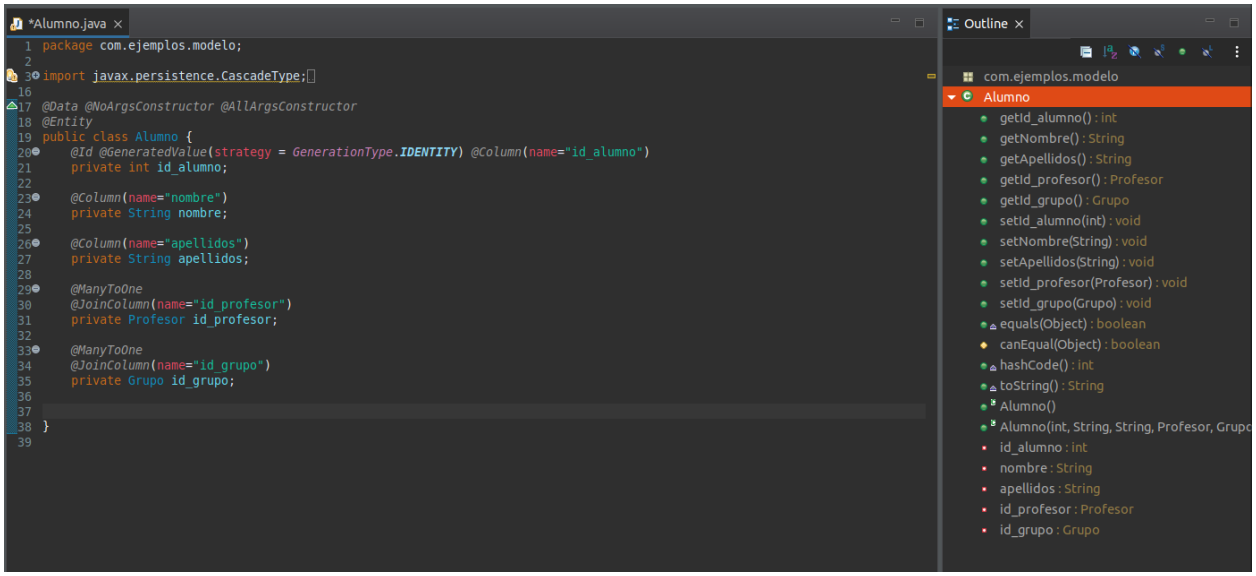
- @ID → Para indicar que ese atributo es el campo que almacenará la clave primaria de la entidad. Es importante que éste atributo sea del mismo tipo y que se llame igual que en la base de datos para evitar incompatibilidades.
- @GeneratedValue(strategy = GenerationType.IDENTITY) → Para indicarle que la clave primaria de la tabla se va a crear de manera automática.
- @Column(name="id_profesor") || @Column(name="id_grupo") → Para indicarle el nombre de la columna de la tabla de la BD que está representando este atributo. Repito que es muy importante que coincidan.

Los campos simples tienen el siguiente atributo:

- @Column(name="") → Para indicar el nombre de la columna de la tabla de la BD que está representando este atributo.

Es importante mencionar que éstas clases, tengan relaciones o no, deben tener tantos atributos como campos tenga la tabla que representan de la BD, coincidiendo tanto el nombre del atributo con el del campo, como el tipo de dato que almacena.

Clase Alumno



A nivel de clase, incluye las anotaciones ya mencionadas anteriormente de lombok y persistance.

La clave primaria y campos simples también son iguales que las otras dos clases del modelo.

La parte importante de ésta clase son las claves foráneas. Estas son del tipo de objeto el cual hace referencia dicha clave. Es decir, en el caso de la clave foránea de id_profesor, el tipo de objeto de dicho atributo será de tipo Profesor, ya que ese campo en la base de

datos, hace referencia al código de un profesor. Para optimizar el proceso, Spring directamente obtiene el objeto de tipo profesor mediante el repositorio de JPA. Igual en el caso de la clave foránea de grupo.

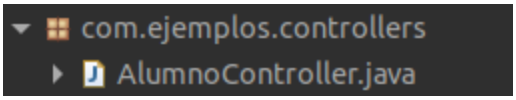
Cabe a destacar las anotaciones que incluyen ambos campos:

- @ManyToOne → Indica la cardinalidad de la relación(1:N en este caso).
- @JoinColum → Indica el nombre de la columna en la BD.

Tanto el nombre de la columna a la que hace referencia, como el atributo, como el campo de la BD, deben llamarse igual. De nuevo, esto se debe a posibles incompatibilidades.

Paquete "com.ejemplos.controllers"

En éste paquete incluiré los controladores que gestionan los recursos diseñados en los demás paquetes, dependiendo del tipo de petición (Get, Post, Put y Delete) y URL que use el cliente. Como en el enunciado de la práctica se detalla que únicamente se requiere el CRUD de la entidad con más relaciones, solo he hecho Controller de la entidad Alumno, por ello, el paquete incluye únicamente una clase:



La clase AlumnoController es la que responderá a las peticiones del cliente y realizará la acción correspondiente. También gestionará la excepción contemplada en el paquete "com.ejemplos.exceptions". Tiene el siguiente código:

*AlumnoController.java

```
1  package com.ejemplos.controllers;
2
3  import java.time.LocalDateTime;
26
27 @RestController
28 @RequiredArgsConstructor
29 public class AlumnoController {
30
31     private final AlumnoRepositorio alumnoRepositorio;
32     private final AlumnoDTOConverter alumnoDTOConverter;
33
34     @GetMapping("/alumno")
35     public ResponseEntity<?> obtenerAlumnos() {
36         List<Alumno> result = alumnoRepositorio.findAll();
37         if(result.isEmpty()) {
38             return ResponseEntity.notFound().build();
39         }else {
40             List<AlumnoDTO> dtoList = result.stream().map(alumnoDTOConverter::convertirADto).collect(Collectors.toList());
41
42             return ResponseEntity.ok(dtoList);
43         }
44     }
45
46     @GetMapping("/alumno/{id}")
47     public ResponseEntity<?> obtenerAlumnoID(@PathVariable int id) throws AlumnoNotFoundException{
48         Alumno result = alumnoRepositorio.findById(id).orElse(null);
49         if(result==null) {
50             throw new AlumnoNotFoundException("El alumno con ID: " + id + " no existe.");
51         }else {
52             return ResponseEntity.ok(alumnoDTOConverter.convertirADto(result));
53         }
54     }
55
```

La clase usa la anotación @RestController, para indicarle a Spring, que esta clase va a ser un controlador y @RequiredArgsConstructor, para crear con lombok un constructor de la clase con todos los atributos requeridos.

Es necesario definir como constante tanto el repositorio JPA del alumno y el DTOConverter, para poder gestionar las vistas, tanto a la hora de enviar como recibir datos del cliente.

Método obtenerAlumnos()

Responde a la URL "/alumno" y devuelve un ResponseEntity<?>, es decir, una entidad de Spring, definida en el modelo, en este caso será una lista de alumnos. En caso de que no haya alumnos en la tabla devolverá un error 404 Not found. El tipo de petición que debe de hacer el cliente para poder acceder a éste método será "Get". A la hora de enviar la lista de alumnos, se hace una conversión usando el DTOConverter, para pasar del objeto de tipo Alumno, al objeto de tipo AlumnoDTO. De ésta manera, los datos de los alumnos enviados en la lista se mostrarán con el formato definido en la clase DTO.

Método obtenerAlumnoID(id Alumno)

Responde a la URL "/alumno/id", es decir, tendremos que especificar en la ruta, el ID del alumno que queremos recibir. En caso de que el ID no coincida con ninguna clave primaria registrada en la BD, devolverá un error 404 Not found. En éste caso, ya que debemos introducir un dato por parámetros mediante la URL, debemos indicarlo con la anotación @PathVariable. El tipo de petición que debe de hacer el cliente para poder acceder a éste método será "Get". A la hora de enviar el Alumno obtenido, se hace una conversión haciendo uso de la clase DTOConverter, para pasar del tipo Alumno al tipo AlumnoDTO. De esta manera, los datos del alumno enviado se mostrarán en el formato definido en la clase DTO. En caso de introducir de manera errónea el id del alumno, nos saltará la excepción AlumnoNotFound, con el código de error 404 Not Found y un mensaje personalizado.

```java
@PostMapping("/alumno")
public ResponseEntity<?> nuevoAlumno(@RequestBody Alumno nuevo){
    Alumno saved = alumnoRepositorio.save(nuevo);
    return ResponseEntity.status(HttpStatus.CREATED).body(saved);
}

@PutMapping("/alumno/{id}")
public ResponseEntity<?> editarAlumno(@RequestBody Alumno editar, @PathVariable int id) throws AlumnoNotFoundException{
    if(alumnoRepositorio.existsById(id)) {
        editar.setId_alumno(id);
        return ResponseEntity.ok(alumnoRepositorio.save(editar));
    }else {
        throw new AlumnoNotFoundException("El alumno con ID: " + id + " no existe.");
    }
}
```

Método nuevoAlumno(Alumno alumno)

Como se indica en la anotación @PostMapping, la petición que haga el cliente deberá ser de tipo “Post”, en la URL “/alumno”. Como parámetro, en este caso, en vez de pedir el dato por URL, se pedirá como cuerpo de la petición, usando la anotación @RequestBody. Éste método creará una nueva entrada en la tabla alumno, con el objeto pasado por el cliente. El alumno deberá tener los campos definidos en la clase createAlumnoDTO.

Método editarAlumno(Alumno alumno, int idAlumno)

Como se indica en la anotación @PutMapping, la petición que haga el cliente deberá ser de tipo “Put” y en la URL “/alumno/id”. En el caso de éste método, además de pasarle por URL el id del alumno que deseamos editar, tendremos que pasar en el cuerpo de la petición el objeto alumno para introducir los campos nuevos a dicho alumno. En caso de que se especifique un id incorrecto, se lanzará la excepción AlumnoNotFound, mostrando el código de error 404 Not found, además de un mensaje personalizado.

```
    @DeleteMapping("/alumno/{id}")
    public ResponseEntity<?> borrarAlumno(@PathVariable int id){
        if(alumnoRepositorio.existsById(id)) {
            alumnoRepositorio.deleteById(id);
            return ResponseEntity.noContent().build();
        }else {
            return ResponseEntity.notFound().build();
        }
    }

    @ExceptionHandler(AlumnoNotFoundException.class)
    public ResponseEntity<ApiError> handleAlumnoNoEncontrado(AlumnoNotFoundException ex){
        ApiError apiError = new ApiError();
        apiError.setEstado(HttpStatus.NOT_FOUND);
        apiError.setFecha(LocalDateTime.now());
        apiError.setMensaje(ex.getMessage());
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(apiError);
    }
}
```

Método borrarAlumno(int idAlumno)

En este caso, como se indica con la anotación @DeleteMapping, el cliente deberá hacer una petición de tipo “Delete”, en la URL “/alumno/id”. Deberá introducir el número de la clave primaria o ID del alumno que desea eliminar.

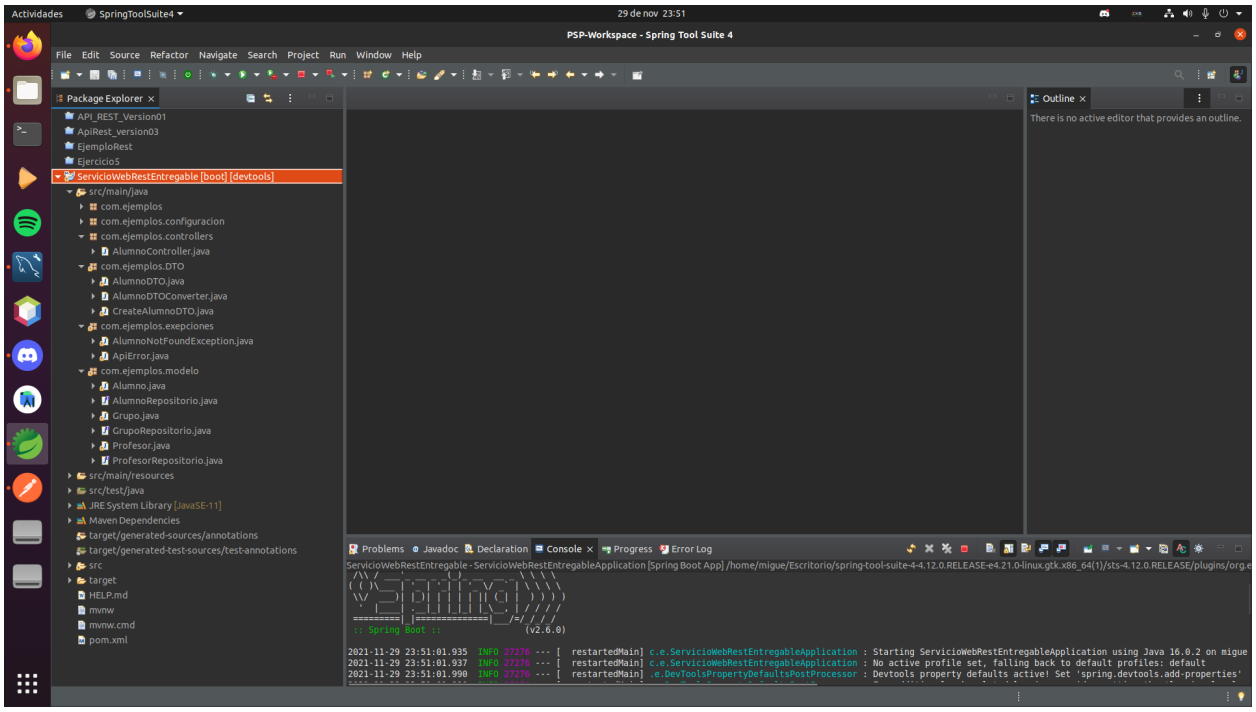
Método handleAlumnoNoEncontrado(AlumnoNotFoundException ex)

Éste método sirve para lanzar la excepción y tratar el código de error que va a incluir en el cuerpo de la misma.

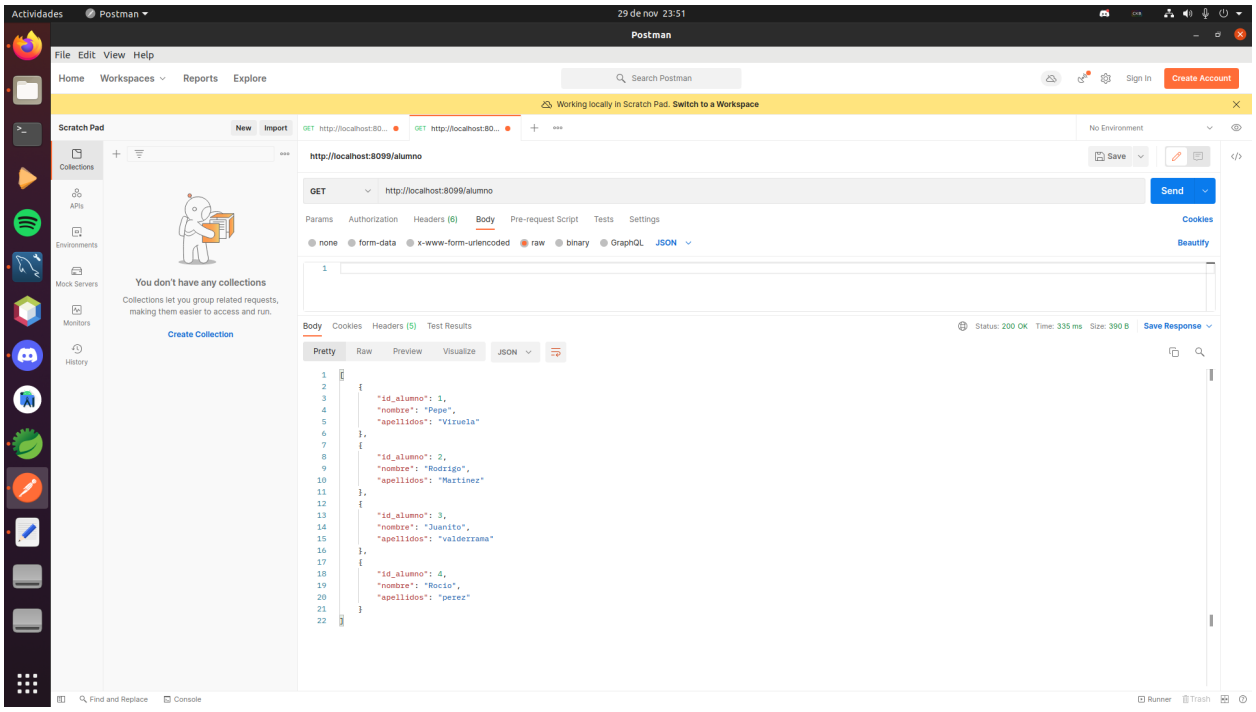
Prueba de la API con PostMan

Una vez explicadas tanto la parte de la BD como la de la aplicación, pasaré a probar con PostMan, cada uno de los métodos que hay en el controler.

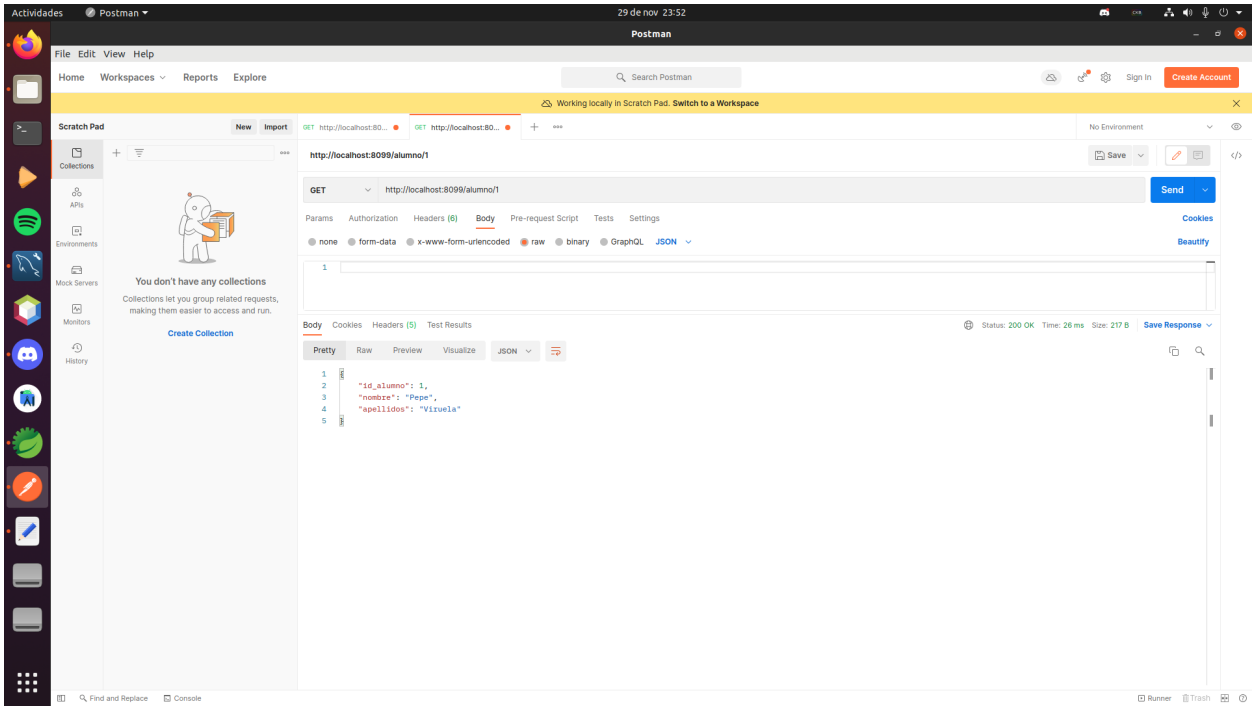
Para ello, lanzaré la aplicación con spring:



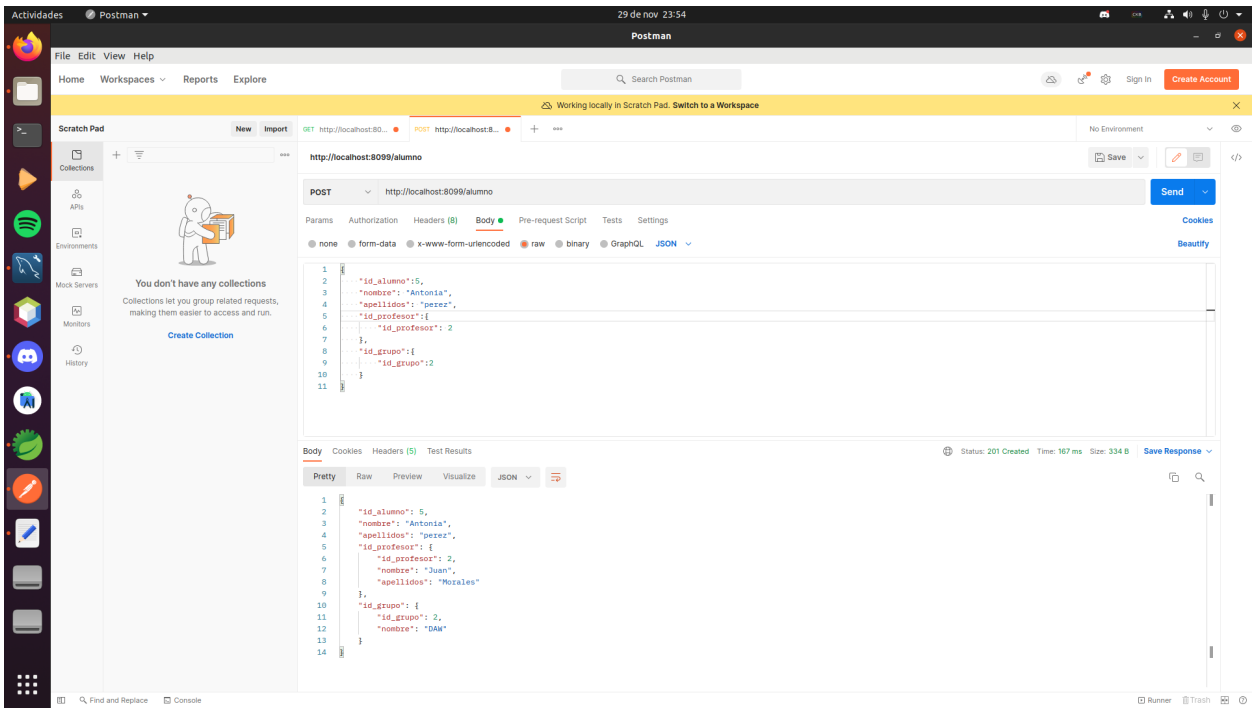
Método obtenerAlumnos()



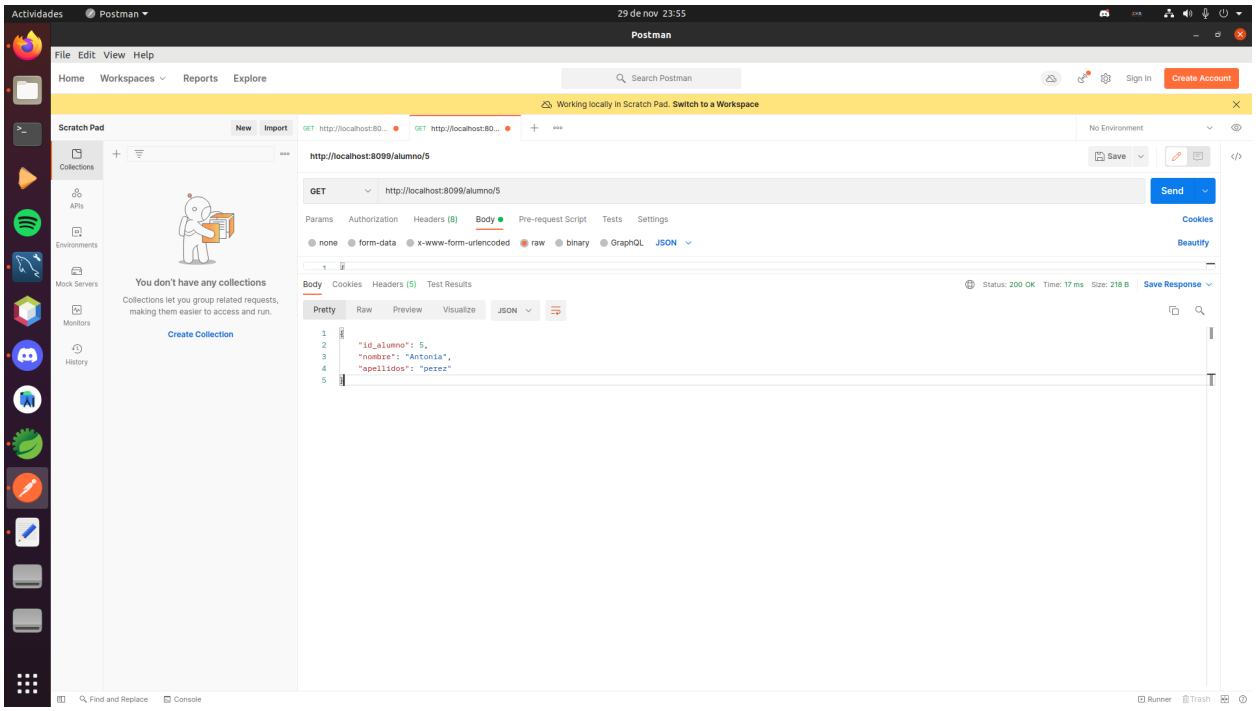
Método obtenerAlumnoID(id Alumno)



Método nuevoAlumno(Alumno alumno)

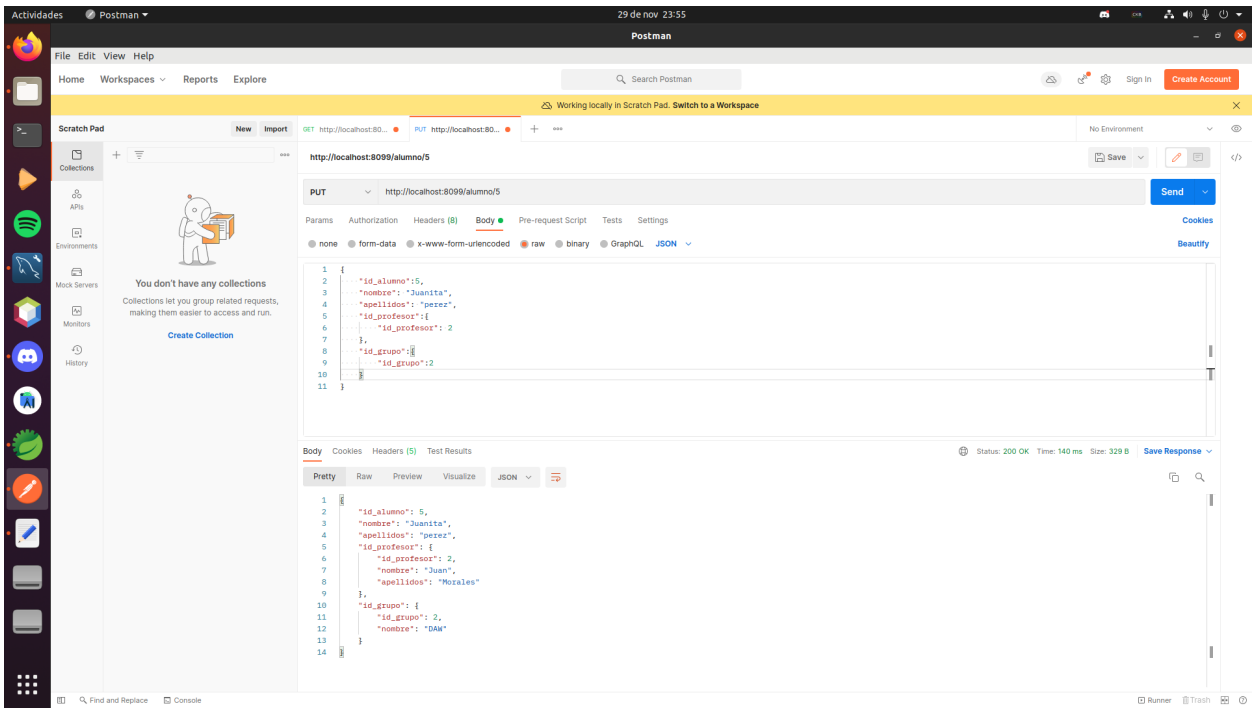


Si ahora obtenemos el alumno 5:



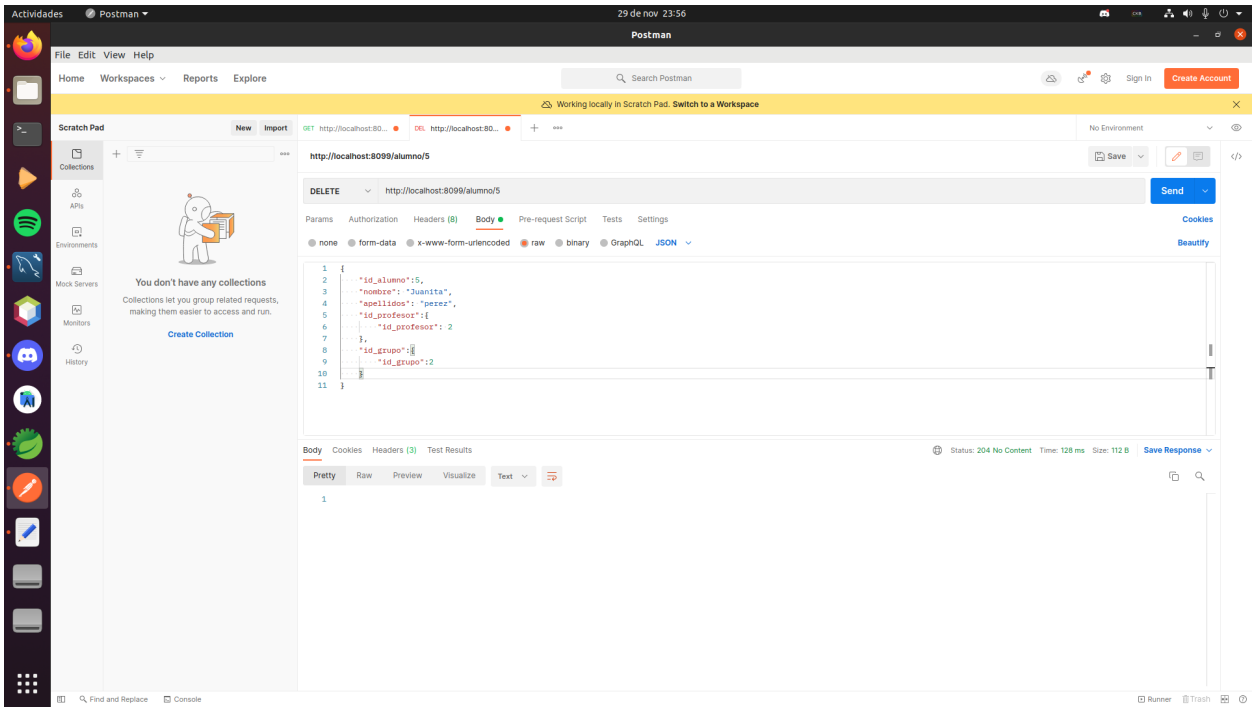
Método editarAlumno(Alumno alumno, int idAlumno)

Edito el nombre del alumno 5:

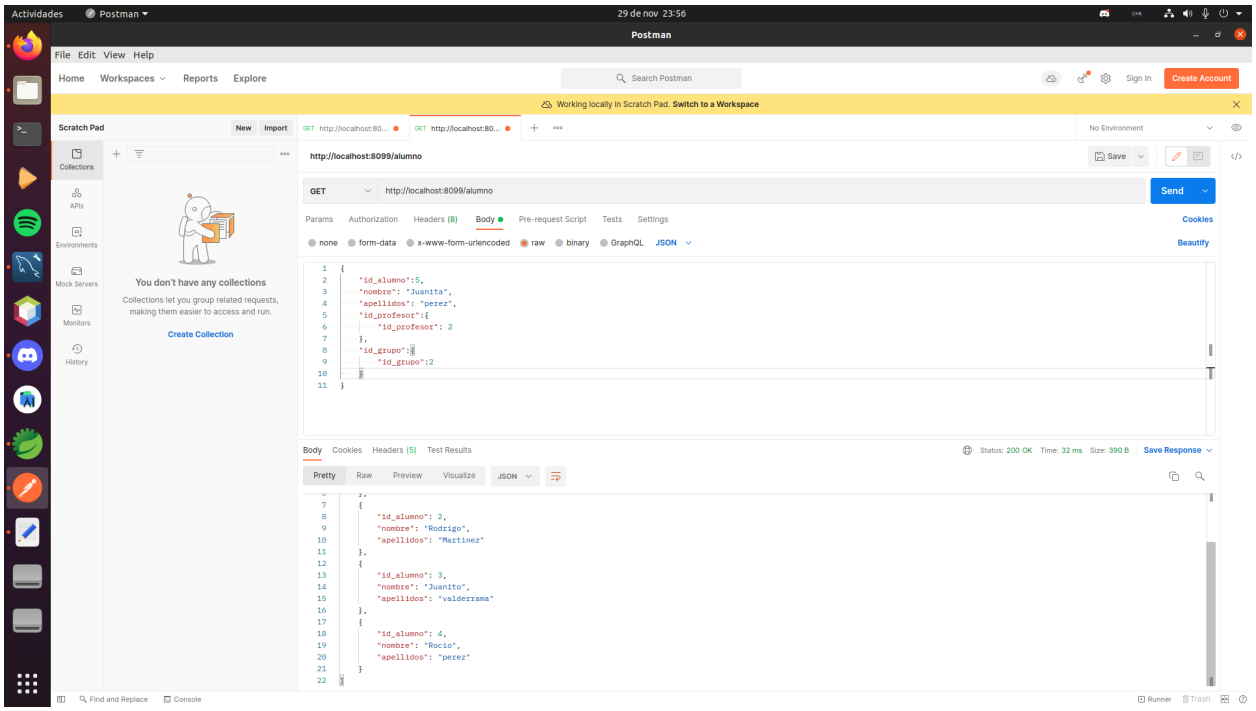


Método borrarAlumno(int idAlumno)

Borrar el alumno 5:



Si ahora obtengo la lista de alumnos, solo hay 4:



Método handleAlumnoNoEncontrado(AlumnoNotFoundException ex)

Intento acceder al alumno 10: