

23-5-2021

APIUNI

API REST con Spring y MySQL



Escuela Técnica Superior de Ingeniería Informática

Grado en ingeniería informática

Ingeniería del Software

Alejandro Blanco Pérez

Álvaro Gómez Pérez

API REST APIUNI

Contenido

Introducción.....	2
Tecnologías	2
Spring	3
Base de Datos.....	6
Herramientas	7
Github	7
Discord	8
Eclipse	8
MySql WorkBench.....	8
Clever Cloud	9
Pasos a seguir para lanzar el Proyecto.....	10
Modelo de Entidades	10
Diagrama	10
Entidades.....	11
Serializaciones.....	13
Relaciones	14
Métodos API.....	17
Introducción.....	17
Llamadas más interesantes.....	17
Respuestas de la API	19
Scrapping.....	19
Conclusiones	20
Bibliografía	21

Introducción

En este documento se va a exponer los distintos componentes que conforman el proyecto desarrollado por el equipo de trabajo. En concreto, este proyecto se basa en el desarrollo de una API REST basada en un modelo de clases relacionadas con el ámbito de la Universidad. Antes de nada, hay que recalcar que tanto el modelo de datos como la aplicación en sí misma es una visión reducida de las muchas entidades que pueden llegar a intervenir en el ámbito universitario, ya que debido al tiempo que el equipo disponía no se ha podido ampliar más el alcance del proyecto.

Para la realización de este proyecto se ha utilizado (principalmente) las tecnologías de Spring para la lógica de la aplicación y MySQL como encargado de gestionar la base de datos, dichas tecnologías y las correspondientes herramientas usadas para coordinar el trabajo del equipo y dar soporte, se explicarán más adelante.

La aplicación ofrecerá a la persona que haga uso de ella una serie de llamadas mediante métodos GET y POST en las cuales se podrá interactuar con la base de datos del sistema con funciones como obtener información de los registros de cualquier tabla disponible de la base de datos o modificar los propios registros, ya sea eliminándolos, modificándolos o incluso creando relaciones entre uno o varios objetos. Esto al igual que las tecnologías se detallará más adelante.

Cómo puntos “extras” a comentar en este documento vamos a encontrar una pequeña guía para lanzar el proyecto y poder probar todas las funciones que dispone la aplicación, además de una serie de problemas y sus respectivas soluciones que nos ha ido surgiendo durante el desarrollo del proyecto y que creemos que es interesante comentar para tener una visión más cercana a la experiencia que hemos tenido.

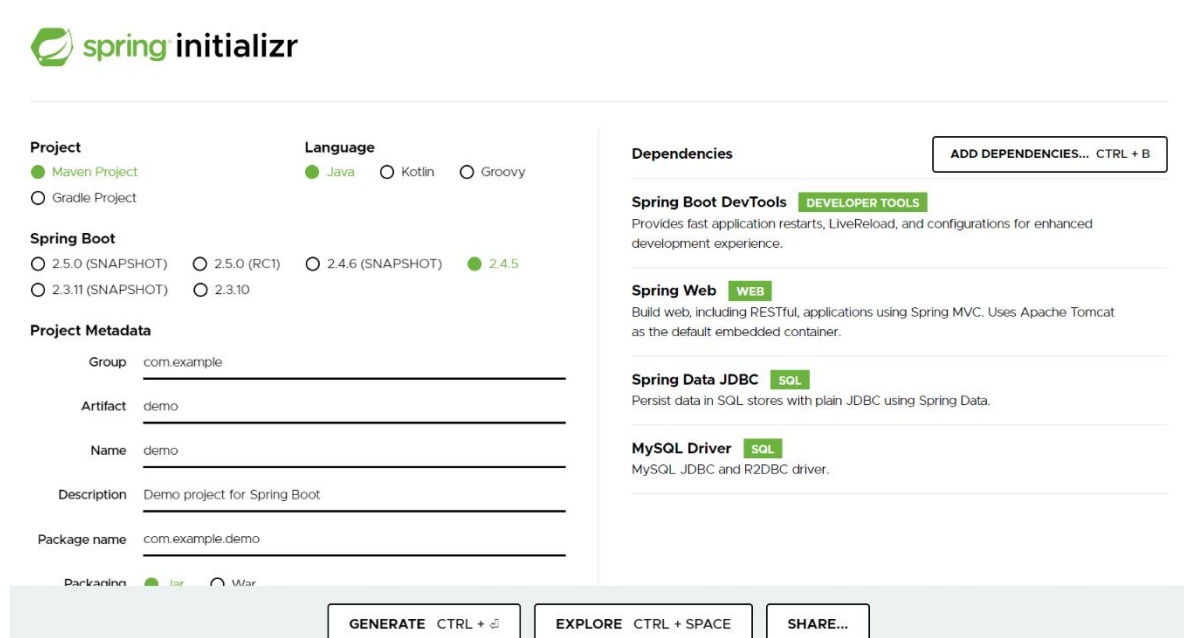
Tecnologías

En esta sección hablaremos sobre la arquitectura de nuestra aplicación web y de las tecnologías que hemos usado para desarrollarla. Hablaremos de Spring, que actuará como backend, de la base de datos, que será MySQL y de las herramientas utilizadas para desarrollar la aplicación web.

Spring

El framework escogido para la aplicación es Spring, un framework de java que nos permite por medio de inyección de dependencias hacer aplicaciones web de una forma muy simple, además debido a la experiencia previa de otros proyectos por parte de los dos miembros del equipo su inicialización y puesta a punto nos ha resultado más cómoda.

Para crear el proyecto Spring, hemos usado una herramienta online muy interesante que ofrece el propio framework, llamada **Spring Initializr**. Esta herramienta permite crear el esqueleto del proyecto de forma gráfica y sencilla, ya que solo le tenemos que indicar parámetros como la versión, lenguaje y nombre del proyecto entre otras. Además, también nos permite añadir las dependencias de una lista separada en categorías, por lo que no tendremos que buscarlas de manera externa para añadirlas al pom.xml



The screenshot shows the Spring Initializr web application interface. It features a logo at the top left and a main content area divided into several sections:

- Project:** Includes radio buttons for "Maven Project" (selected) and "Gradle Project".
- Language:** Includes radio buttons for "Java" (selected), "Kotlin", and "Groovy".
- Spring Boot:** Includes radio buttons for versions: "2.5.0 (SNAPSHOT)", "2.5.0 (RC1)", "2.4.6 (SNAPSHOT)", "2.4.5" (selected), "2.3.11 (SNAPSHOT)", and "2.3.10".
- Project Metadata:** Includes input fields for "Group" (com.example), "Artifact" (demo), "Name" (demo), "Description" (Demo project for Spring Boot), and "Package name" (com.example.demo).
- Dependencies:** Includes a button "ADD DEPENDENCIES... CTRL + B" and a list of dependencies with category tags: "Spring Boot DevTools" (DEVELOPER TOOLS), "Spring Web" (WEB), "Spring Data JDBC" (SQL), and "MySQL Driver" (SQL).

At the bottom, there are three buttons: "GENERATE CTRL + G", "EXPLORE CTRL + SPACE", and "SHARE...".

FIGURA 1. Spring Initializr.

Nuestra aplicación usará el patrón MVC (Modelo-vista-controlador). Para conectar nuestra base de datos con Spring, necesitaremos inyectar algunas dependencias indispensables, la forma de hacerlo es mediante el archivo pom.xml.

```

<!-- Conector/libreria de MYSQL para java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>

```

FIGURA 1. Pom.xml con inyección del driver mysql connector

Además, tenemos que añadir a las propiedades del proyecto la cadena de conexión a la base datos junto al usuario y la contraseña, para ello, lo indicamos en el archivo de application.properties.

```

1  spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
2 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
3
4 #spring.datasource.url=jdbc:mysql://127.0.0.1:3306/apiuni?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC
5 #spring.datasource.username=root
6 #spring.datasource.password=Strong-P@$$w0rd!
7 spring.datasource.url=jdbc:mysql://bncipwhkpb248p7flayx-mysql.services.clever-cloud.com:3306/bncipwhkpb248p7flayx?useUnicode=true&useJDBCCompliantTimezoneShift=tru
8 spring.datasource.username=ulx1sh1hzd4sgfyv
9 spring.datasource.password=NpZdC3bNPhxgi6scTTs
10
11 spring.jpa.hibernate.ddl-auto=create-drop
12 spring.jpa.show-sql=true
13 spring.jpa.properties.hibernate.format_sql=true
14 server.port=8090
15 spring.datasource.initialization-mode=always
16
17 #desactivar generate-ddl para que no aparezca el sql en la consola
18 spring.jpa.generate-ddl=true
19 server.error.include-stacktrace=never

```

FIGURA 2. Application.properties con datos necesarios para conectar a la base de datos.

Entre las propiedades más interesantes mostradas en la captura anterior encontramos:

- **Spring.datasource.url:** La cual indica la url a la que se tiene que conectar la aplicación para acceder a la base de datos.
- **Spring.datasource.username y spring.datasource.password:** Son las credenciales que la aplicación debe usar para acceder correctamente a la base de datos de la url indicada.
- **Spring.jpa.hibernate.ddl-auto:** Indica que cada vez que la aplicación se reinicie se deben eliminar las tablas completamente y volver a crear junto con sus registros correspondientes. En nuestro caso decidimos de seleccionar esta opción debido a que cómo las entidades relacionadas en la aplicación no son demasiadas, no nos influía el hecho de esperar que se crearan las tablas por cada reinicio de la aplicación a cambio de realizar un desarrollo más limpio.

Una vez añadido este código, la aplicación esta lista para usar la base de datos.

Lo primero que hacemos con Spring es crear las tablas de nuestra base de datos, para ello creamos entidades. Las entidades son clase de Java, que contienen anotaciones necesarias

para la conversión a tablas de la base de datos. Para indicar el nombre de la tabla, añadimos la anotación `@Entity` encima del nombre de la clase. Si no especificamos nada, la tabla tendrá por defecto el nombre de la clase. Para crear las columnas de la tabla, creamos las propiedades del objeto. Si no indicamos con la anotación `@Column` el nombre de la columna, por defecto se creará con el nombre que tenga la propiedad del modelo.

A parte de generar las tablas, Spring nos permite también hacer consultas a la base de datos, para ello crearemos repositorios. Los repositorios son Interfaces de java con la anotación `@Repository`, en nuestro caso extenderemos a `CrudRepository`, que tiene los métodos más comunes de consultas a la base de datos, como por ejemplo `findAll()`, que obtiene todos los registros de la tabla y los convierte a objetos java. De forma que podremos tratar los datos mediante métodos. `CrudRepository` nos ofrece todos los métodos necesarios para el CRUD (Create, Read, Update, Delete).

En el siguiente ejemplo podemos observar el método para obtener un alumno de la base de datos filtrando por un número de teléfono dado. El formato que sigue el repositorio de Spring para hacer una consulta de lectura como esta es `findBy` seguido de la propiedad del modelo a través de la cual queremos filtrar, introduciéndole el dato como parámetro del método.

```
1 package com.apiuni.apiuni.repositorio;
2
3 import java.util.List;
4
5
6
7
8
9
10
11 @Repository
12 public interface AlumnoRepository extends CrudRepository<Alumno, Long>{
13
14     public abstract List<Alumno> findByTelefono(Integer telefono);
15
16
17 }
18
```

FIGURA 3. Repositorio para Alumno

También generaremos los servicios, donde desarrollaremos los métodos no triviales que no hagamos mediante consultas simples en el repositorio. En el servicio se añadirá la anotación `@Service` encima del nombre de la clase. Para poder hacer las consultas a la base de datos, añadiremos el repositorio del objeto al que queremos hacérselas con la anotación `@Autowired`, que inyectara el repositorio para poder usar los métodos anteriormente descritos. A continuación, vemos un ejemplo del servicio de Alumno con algunos métodos:

```

1 package com.apiuni.apiuni.servicio;
2
3 import java.util.List;
4
5
6
7
8
9
10 @Service
11 public class AlumnoService {
12
13     @Autowired
14     private AlumnoRepository alumnoRepository;
15
16     public List<Alumno> findAllAlumno() {
17
18         return (List<Alumno>) this.alumnoRepository.findAll();
19     }
20
21     public Alumno findAlumnoById(long id) {
22         return this.alumnoRepository.findById(id).orElse(null);
23     }
24
25     public void save(Alumno a) {
26         alumnoRepository.save(a);
27     }
28
29     public long saveId(Alumno d) {
30         return this.alumnoRepository.save(d).getId();
31     }
32 }

```

FIGURA 4. Servicio para Alumno

Por último, veremos los controladores, que son clases java anotadas con `@Controller`. En el controlador obtendremos las url de la aplicación web para devolver los objetos de nuestra API. Para ello, tendremos que añadir las anotaciones `@RestController` (Para indicar que el controlador trabaja con una API REST) y `@RequestMapping` (Para indicar la ruta raíz del controlador). En el controlador se inyectarán los servicios de nuevo con la anotación `@Autowired`.

Dependiendo de la ruta que obtenga el controlador, accederá a uno de los métodos anotados con `@GetMapping`, `@PostMapping`, `@PutMapping` o `@DeleteMapping` para devolver la respuesta en la vista. De este tipo de clases veremos ejemplos más concretos cuando pasemos a exponer las distintas llamadas que dispone nuestra Api Rest.

Base de Datos

La base de datos utilizada para recoger la información de la web es MySQL, que es un sistema de gestión de bases de datos relacionales muy famoso para el desarrollo de aplicaciones web, esto se debe a que trabaja con un sistema centralizado de gestión de datos, por lo que nos permite realizar cambios en un solo archivo sin tener que tocar el resto para ver los cambios efectuados, algo que resulta muy útil a los desarrolladores ya que les agiliza el trabajo.

La información que guardamos es sobre datos universitarios, en concreto, nos hemos basado en la Universidad de Sevilla. Las tablas que almacena la base de datos son alumnos, profesores, departamentos, asignaturas y titulaciones, siendo todos los datos introducidos datos reales de la propia web de la universidad de Sevilla mediante un proceso de scrapping, de forma que se han introducido en el archivo “data.sql” de nuestro proyecto, el cual Spring utiliza para inyectar mediante queries a la base de datos de la aplicación. Esta información es consultada por nuestro backend, que se trata de Spring Boot, ya que este framework simplifica mucho la creación de consultas, haciendo las más básicas sin necesidad de crear la query manualmente, como son las de consultar, guardar y borrar objetos. Las tablas de la base de datos también se generan automáticamente tras crear las entidades, que no son más que clases java que contienen los atributos de cada entidad, estos atributos se traducen a columnas de cada tabla.

En la siguiente imagen podemos ver la inserción de los datos de forma manual en la base de datos a través del archivo data.sql.

```
1
2
3 INSERT INTO `departamento` VALUES (2076,'admon-sec@dte.us.es,director@dte.us.es',
4 'TECNOLOGÍA ELECTRÓNICA','ESCUELA POLITÉCNICA SUPERIOR','95.455.27.64','http://www.dte.us.es/'),(2074,'preventivaysaludpublica@us.es'
5
6
7 INSERT INTO `titulaciones` VALUES (2668,'Máster Universitario en Ingeniería Biomédica y Salud Digital'),
8 (2686,'Máster Universitario en Ingeniería Informática'),(2710,'Máster Universitario en Lógica, Computación e Inteligencia Artificial'
9
10
11 INSERT INTO `asignatura` VALUES (2718,'1','Optativa','6','Cuatrimestral (Segundo Cuatrimestre)'
12 ,'Procesamiento del Lenguaje Natural',2055,2710),(2714,'1','Optativa','6','Cuatrimestral (Primer Cuatrimestre)','Computación Bioinspi
13
14
15 INSERT INTO `profesor` VALUES (2416,'soledad@us.es','SOLEDAD FERNANDEZ GARCIA','954559841',2057)
16 ,(2411,'smartin@us.es','SERGIO MARTIN GUILLEN','954557095',2076),(2412,'delapaz@us.es','SERGIO MONTSERRAT DE LA PAZ','El número de te
17
18
19 INSERT INTO `asignatura_profesores` VALUES (2568,2274),(2568,2237),(2568,2154),(2568,2084),(2464,2374),(2464,2306),(2464,2276),
20 (2464,2264),(2463,2338),(2463,2245),(2579,2141),(2463,2227),(2579,2133),(2522,2121),(2521,2408),(2521,2255),(2520,2359),(2632,2242),(
21
```

FIGURA 5. Archivo data.sql con datos reales introducidos para rellenar la base de datos.

Herramientas

Entre las herramientas que hemos usado en relación con el proyecto destacan GitHub, Discord, Eclipse IDE, MySQL WorkBench y Clever Cloud.

GitHub

La aplicación para la gestión del código fuente que hemos elegido para trabajar conjuntamente es GitHub, ya que hemos hecho durante la carrera muchos proyectos con esta herramienta, por lo que ya nos resulta muy familiar y cómoda de usar, además de que es gratuita y tiene interfaz web y de escritorio. El repositorio en el que se encuentra nuestro proyecto es: <https://github.com/alvgomper1/APIUNI/tree/master/apiuni>

Se trata de un repositorio público, en el que hemos creado 2 ramas a partir de máster, una de develop, que es donde probaremos el código que se vaya subiendo y otra llamada Blanco, que es para Alejandro Blanco y se ha creado por comodidad a la hora de subir el código.

Discord

Esta herramienta se ha usado para la comunicación entre los integrantes del proyecto, ya que es una aplicación que ambos utilizamos frecuentemente y estamos acostumbrados a usar para otras tareas. La hemos elegido por la facilidad para compartir contenido, hablar por voz, videollamada o incluso retransmitir en directo la pantalla para trabajar telemáticamente entre ambos.

Eclipse

El entorno de desarrollo elegido para desarrollar el código fuente de la aplicación es eclipse, ya que, al igual que GitHub, es una aplicación que llevamos usando desde el principio de la carrera, por lo que no tenemos que aprender a usar otro diferente y nos agiliza el trabajo. Además, nos ofrece la posibilidad de integrar git para poder gestionar el proyecto de una forma más rápida.

MySQL WorkBench

Esta herramienta nos sirve para poder visualizar la base de datos de nuestro proyecto, es una de las más usadas por los desarrolladores, ya que es gratuita y tiene una gran variedad de funcionalidades, aunque nosotros solo la usaremos para poder ver los datos de las tablas, editarlos y borrarlos, ya que el diseño de la base de datos se hace mediante Spring. Otro motivo por el que hemos usado MySQL Workbench es por la facilidad a la hora de instalar en nuestras máquinas, ambas Windows 10.

Esta herramienta nos permite visualizar la base de datos tanto en local como en la nube, simplemente añadiendo los datos de acceso que son host, puerto, usuario y contraseña como podemos ver en la siguiente imagen.

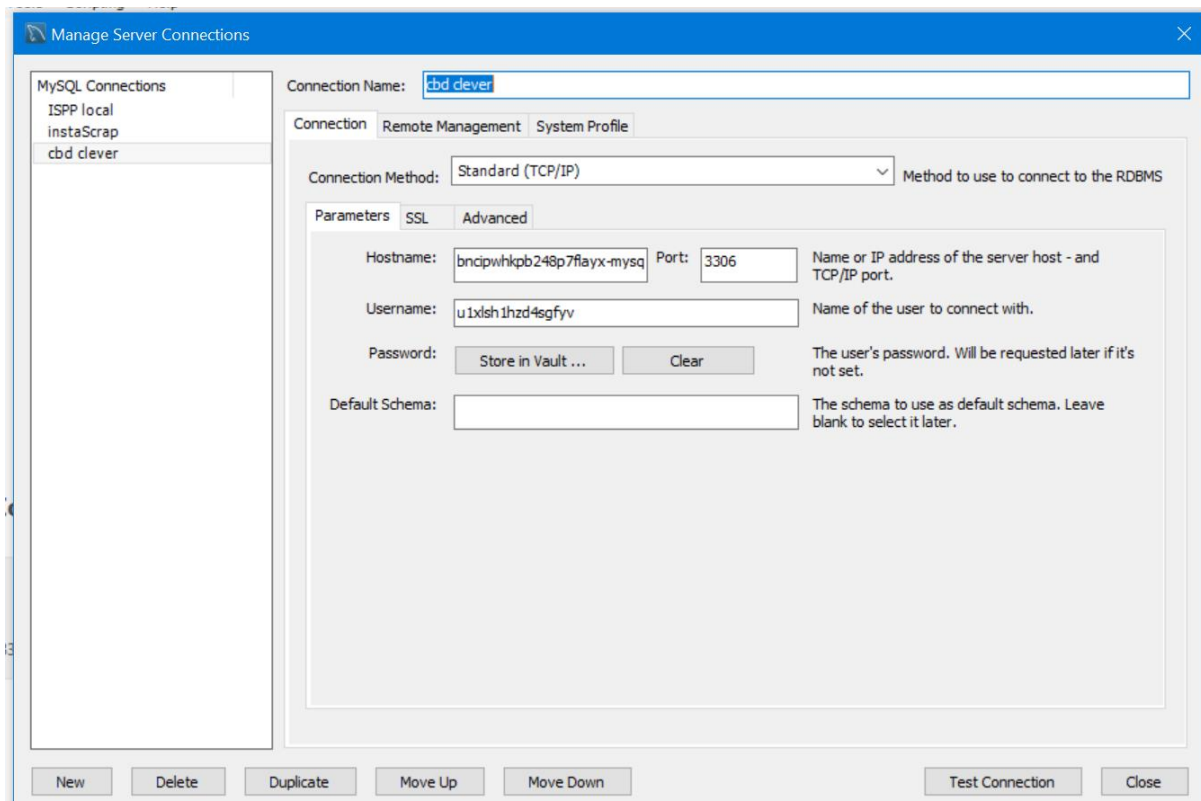


FIGURA 6. Crear conexión a la base de datos desde MySQL Workbench con datos de acceso.

Clever Cloud

Para desplegar nuestra base de datos en un servidor hemos usado Clever Cloud, ya que es muy fácil crear nuestra base de datos y es gratis. Los pasos para crear la base de datos son registrarse con un correo electrónico y validarlo, después en el panel de control crear un add-on, seleccionar MySQL y ponerle un nombre a la base de datos para poder acceder a ella más fácilmente en caso de que tengas más. Tras esto, seleccionaremos el plan gratuito que nos da una base de datos de 10 MB y 5 conexiones simultaneas, que, para nuestro caso, nos sirve ya que no necesitaremos mucha más memoria ni conexiones a la vez. En caso de que necesitáramos una base de datos con más recursos, habría que buscar un plan de pago.

Una vez se ha creado la base de datos, obtenemos los datos de acceso que son host, puerto, usuario y contraseña, que viene en el panel de administración de la base de datos como podemos ver en la siguiente imagen.

MySQL by Clever Cloud

AdminPHPMyAdminDocumentation

TYPE	PLAN	CLUSTER	VERSION	REGION	STATUS	CREATION DATE	ID
MySQL	Dev	par-mysql-c6	8.0	par	ACTIVE	2021-05-21	mysql_fabaa7d2-be04-4605-b849-8bcf43572096

Database Credentials

Get credentials for manual connections to this database.

Export Environment Variables

Host

bncipwhkpb248p7flayx-mysql.services.clever-cloud.com

Database Name

bncipwhkpb248p7flayx

User

ulxlshlhzd4sgfyv

Password

.....

Port

3306

Connection URI

.....

FIGURA 7. Panel de administración de base de datos en Clever Cloud con datos de conexión.

Pasos a seguir para lanzar el Proyecto

En este apartado vamos a presentar una pequeña guía de cómo poner en marcha la aplicación desarrollada para su posterior uso y pruebas. Debido al tiempo que hemos podido dedicar al proyecto, no hemos podido desplegar la aplicación en un servidor de pruebas como puede ser Heroku, que era nuestra intención inicial, así que su uso deberá ser en local.

Para Conseguir que nuestra aplicación se lance, se debe importar el proyecto en cualquier IDE que se disponga, con sólo dos condiciones que el IDE en el que se importa el proyecto tenga la disponibilidad de utilizar Java para la ejecución de los proyectos y que el proyecto sea importado como Maven Project. Cumpliendo con estas dos simples condiciones el proyecto deberá funcionar correctamente en el ámbito local de cualquier máquina. Además, cómo se ha desplegado la base de datos en un servidor de Clever Cloud, no hará falta ninguna configuración o instalación extra para iniciar la aplicación.

Modelo de Entidades

Diagrama

En nuestro diagrama UML podemos observar las entidades que crearemos para representar las tablas, sus atributos que serán las columnas de dichas tablas y las relaciones entre ellas,

que se crearán en Spring con anotaciones como `@ManyToMany` para relaciones N-N, `@ManyToOne`, para relaciones N-1 y `@OneToMany`, para relaciones 1-N.

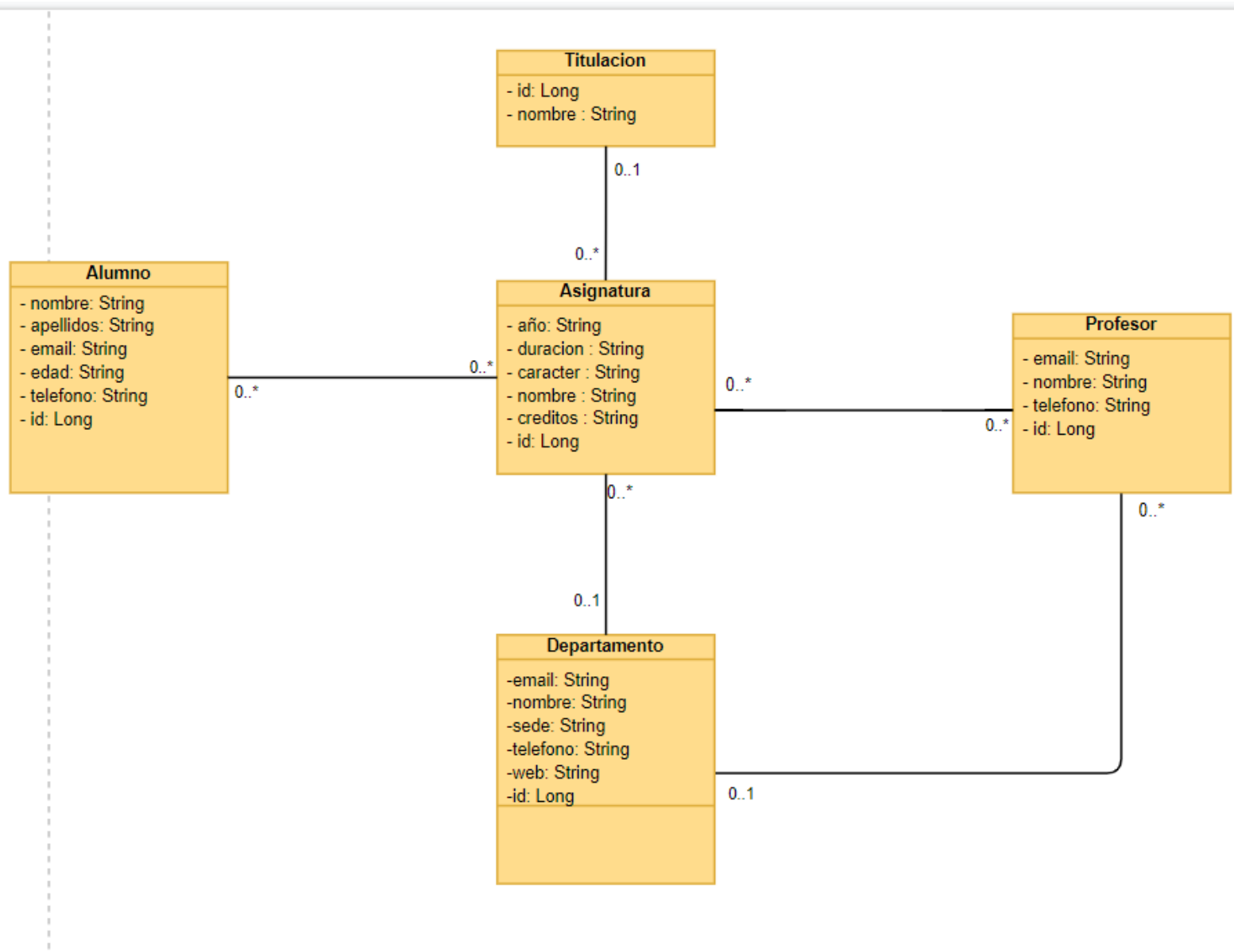


FIGURA 8. Diagrama UML.

Entidades

Las entidades corresponden a las clases obtenidas previamente del diagrama UML, que son Alumno, Profesor, Departamento, Titulación y Asignatura, cada una de ellas con sus atributos.

La entidad se anotará como `@Entity` para crear la tabla como hemos visto anteriormente, en caso de que queramos ponerle un nombre personalizado a una tabla usaremos la anotación `@Table`, pasándole como parámetro *name* el nombre que queremos que muestre en nuestra

base de datos. Todas las entidades tendrán un id que identificará cada objeto, de manera que no habrá 2 objetos de una misma entidad que compartan id, esto se indica con la anotación @id y con el parámetro unique = true dentro de la anotación @Column.

Para acceder a los atributos del objeto, generamos getter y setters para todas las propiedades.

En las entidades también se crean las relaciones entre objetos, que serán las que se describen en el diagrama UML y se explicarán en el siguiente apartado más detalladamente.

Para nuestra API REST, decidimos que algunos campos de las entidades no se mostrarán en el formato Json cuando hiciéramos una petición en la aplicación web, esta decisión fue tomada debido a la gran cantidad de datos que devolvían los objetos a causa de las relaciones que hay entre ellos, ya que al ser sobre datos universitarios, todo está estrechamente relacionado como podemos observar en el diagrama, de manera que si queremos crear una asignatura por ejemplo, habría que indicar el departamento que tiene, y al ser Departamento un objeto, el Json mostraría de nuevo el objeto Departamento dentro, que a su vez tiene más relaciones con objetos de tipo Profesor y Asignatura, por lo que en lugar de mostrar tantos objetos anidados, solo mostraremos atributos de tipo String a la hora de crear objetos mediante la API, no obstante, también crearemos métodos para que nuestra aplicación permita añadir estas relaciones entre objetos de manera más cómoda y sencilla.

Para poder ocultar los atributos que no nos interesan a la hora de crear objetos usamos la anotación @JsonIgnore encima de la propiedad que queremos ocultar.

```

@Entity
public class Alumno {

    @Id
    @Column(unique = true, nullable = false)

    private Long id;

    private String nombre;
    private String apellidos;
    private Integer edad;
    private String email;

    private String telefono;

    @JsonIgnore()
    @ManyToMany(mappedBy = "alumnos" )
    private List<Asignatura> asignaturas;

    @PreRemove

```

FIGURA 9. Ejemplo de la entidad Alumno con anotación @JsonIgnore()

Serializaciones

Para evitar el problema de los objetos anidados expuesto en el punto anterior, a la hora de mostrar resultados también necesitábamos modificar el Json, de forma que optamos por crear una clase extra por cada entidad, de manera que se dé un formato a los objetos para poder mostrar los objetos solo por su nombre, y no mostrar el resto de los atributos, por ejemplo, las asignaturas del alumno, que solo mostraríamos el nombre y no el objeto completo.

Para ello, extendemos la clase JsonSerializer y creamos el método serialize con un objeto JsonGenerator para poder darle el formato que queramos al Json que mostraremos en nuestra aplicación, de manera que, con una llamada a este método en el controlador, obtengamos la serialización del objeto que queramos.

Un ejemplo sería el de alumno en la siguiente imagen.

```

11 import java.io.IOException;
12
13 @JsonComponent
14 public class AlumnoJsonSerializer extends JsonSerializer<Alumno> {
15
16     @Override
17     public void serialize(Alumno alumno, JsonGenerator jsonGenerator, SerializerProvider serializers)
18         throws IOException {
19
20         jsonGenerator.writeStartObject();
21
22         jsonGenerator.writeStringField("alumnoId", alumno.getId().toString());
23         jsonGenerator.writeStringField("email", alumno.getEmail().toString());
24         jsonGenerator.writeStringField("apellidos", alumno.getApellidos().toString());
25         jsonGenerator.writeStringField("nombre", alumno.getNombre().toString());
26         jsonGenerator.writeStringField("telefono", alumno.getTelefono().toString());
27         jsonGenerator.writeStringField("edad", alumno.getEdad().toString());
28         jsonGenerator.writeStringField("asignaturas",
29             alumno.getAsignaturas().stream().map(x -> x.getNombre()).collect(Collectors.toList()).toString());
30
31         jsonGenerator.writeEndObject();
32     }
33 }

```

FIGURA 10. Ejemplo de la clase AlumnoJsonSerializer con su formato.

Relaciones

Una de las partes más importantes y a la vez más complejas que forman nuestra aplicación son las relaciones existentes entre las distintas entidades que disponemos, a continuación, se va a detallar cada una de ellas mostrando cómo se han implementado gracias a las distintas anotaciones @ManyToMany, @OneToOne, @OneToMany y @ManyToOne:

- Titulación <--> Asignatura

```

31 @JsonIgnore
32 @OneToMany(cascade = CascadeType.ALL, mappedBy = "titulacion")
33 private List<Asignatura> asignaturas;
34

```

FIGURA 11. Implementación en la entidad Titulación de su relación con Asignatura .

```

29 @JsonIgnore
30 @ManyToOne
31 @JoinColumn(name = "titulacion_id")
32 private Titulacion titulacion;
33

```

FIGURA 12. Implementación en la entidad Asignatura de su relación con Titulación .

Esta relación es una relación bidireccional, ya que cada Titulación registrada en la base de datos contiene de 0 a “Muchas” asignaturas, por lo tanto, hemos usado la anotación @OneToMany, cómo se puede ver en la anterior captura. En el caso de la entidad Asignatura ocurre lo mismo, pero a la inversa, por lo tanto, hemos usado la anotación @ManyToOne. En el caso del @OneToMany usado en la entidad Titulación, podemos observar que contiene dos propiedades las cuales se verán de ahora en adelante en las siguientes relaciones mostradas:

- Cascade : Indica cómo se deben actualizar los datos de la entidad al otro lado de la relación, en nuestro caso, hemos utilizado el CascadeType.ALL para que se propaguen todo tipo de acciones de una entidad a otra.
- mappedBy: Es una propiedad que indica que la entidad en la que aparece es la entidad “padre” de la relación existente entre las dos entidades.

Por último, también se ha mostrado en la segunda captura, la notación @JoinColumn, la cual indica el nombre de la columna que se va a crear en la entidad “hija” para apuntar a la entidad “padre” que se marca en esta relación, esto es opcional y no afecta al funcionamiento de la aplicación.

A continuación, vamos a mostrar las relaciones en las que interviene la entidad Departamento, la cual se relaciona con Asignatura y Profesor:

```

39     @OneToMany(cascade = CascadeType.ALL, mappedBy = "departamento")
40     private List<Asignatura> asignaturas;
41
42
43     @OneToMany(cascade = CascadeType.ALL, mappedBy = "departamento")
44     private List<Profesor> profesores;
45

```

FIGURA 13. Implementación en la entidad Departamento de sus relaciones .

Cómo podemos ver en la captura anterior, en ambas relaciones utilizamos la notación @OneToMany debido a que cada departamento posee una serie de Asignaturas y una serie de profesores que pertenecen a un departamento en concreto. En cuanto a las propiedades que contienen las notaciones, podemos comprobar que se tratan de las anteriormente explicadas.

```

40     @ManyToOne
41     @JoinColumn(name = "departamento_id")
42     private Departamento departamento;
43

```

FIGURA 14. Implementación en la entidad Asignatura de su relación con Departamento.

```

30     @ManyToOne
31     private Departamento departamento;
32

```

FIGURA 15. Implementación en la entidad profesor de su relación con Departamento.

En las entidades con las que se relaciona Departamento utilizamos el @ManyToOne para indicar que tanto cada profesor como cada asignatura pertenece a un departamento concreto. En estas dos capturas se puede observar que una no tiene el @JoinColumn con la propiedad name, esto simplemente pondrá en las tablas de Profesor una columna con el nombre que Spring tome como defecto para apuntar a la entidad Departamento.

Por último, vamos a ver las relaciones que aún no se han visto en las que intervenga la entidad Asignatura, las cuales son :

- Asignatura - Profesor
- Asignatura- Alumno

Empezamos por la relación de Asignatura con Profesor, la cual es una relación ManyToMany, ya que Cualquier Profesor tiene la posibilidad de tener más de una asignatura y cualquier asignatura la posibilidad de que sea impartida por más de un profesor (casi siempre se da ambos casos). En el caso de la relación de Asignatura con Alumno pasa exactamente lo mismo, por lo tanto, su implementación será la misma en ambas relaciones.

```

35  @JsonIgnore
36  @ManyToMany()
37  private List<Profesor> profesores;
38
39  @JsonIgnore
40  @ManyToMany()
41  @JoinTable(name = "asignatura_alumnos", joinColumns = @JoinColumn(name = "FK_ASIGNATURA", nullable = false),
42  inverseJoinColumns = @JoinColumn(name = "FK_ALUMNOS", nullable = false))
43  private Set<Alumno> alumnos;

```

FIGURA 16. Implementación en la entidad Asignatura de su relación con Profesor y Alumno.

Cómo se puede observar en ambos casos las notaciones usadas son las @ManyToMany, las cuales indican la relación explicada anteriormente y crean una tabla intermedia con las id's de cada uno de los objetos relacionados entre sí. En el caso de la propiedad alumno, se muestra que además usamos la notación @JoinTable, la cual de una forma similar a la notación @JoinColumn, indica los nombres de cada una de las columnas de la tabla intermedia creada por el @ManyToMany.

```

32  @JsonIgnore()
33  @ManyToMany(mappedBy = "alumnos" )
34  private List<Asignatura> asignaturas;
35
36  @PreRemove
37  private void removeAlumnosFromAsignatura() {
38      for (Asignatura u : asignaturas) {
39          u.getAlumnos().remove(this);
40      }
41  }

```

FIGURA 17. Implementación en la entidad Alumno de su relación con Asignatura.

```

29  @JsonIgnore
30  @ManyToMany(mappedBy = "profesores" )
31  private List<Asignatura> asignaturas;
32
33  @PreRemove
34  private void removeProfesoresFromAsignatura() {
35      for (Asignatura u : asignaturas) {
36          u.getProfesores().remove(this);
37      }
38  }

```

FIGURA 18. Implementación en la entidad Profesor de su relación con Asignatura.

Al igual que en los casos anteriores en la notación @ManyToMany se ha añadido mappedBy para indicar que ambas entidades son las “padres” de la relación indicadas. Además, estas relaciones tienen un método con la notación @PreRemove, esta notación indica que el método que contiene realiza su ejecución antes de que se realice una operación de eliminación de la entidad en la que se encuentra. Por lo tanto, cada vez que se elimine un Profesor o un alumno, se van a eliminar las relaciones existentes entre las entidades y la

entidad Asignatura para que en su posterior eliminación los objetos de la tabla Asignatura no se eliminen por propagación.

Métodos API

Introducción

En la Api Rest implementada en la aplicación encontramos distintas llamadas de cada una de las operaciones básicas CRUD de cualquier servicio de Api Rest. Además, hemos implementado variantes de estas llamadas básicas que pueden dar bastante funcionalidad y utilidad a la aplicación desarrollada. Estas variantes las vamos a exponer más detalladamente a continuación junto a su correspondiente implementación. Dichas funcionalidades que se muestran de las siguientes implementaciones o cualquier otra, está documentada en la propia vista de Swagger implementada en la aplicación como lugar plataforma donde poder realizar pruebas en cada una de las llamadas de la aplicación.

En las distintas implementaciones que se van a mostrar vamos a encontrar distintas notaciones que nos han ayudado a realizar la documentación de la Api Rest en Swagger:

- **@Operation**: Indica los datos de la operación que va a realizar el método que engloba, las propiedades que usamos dentro de esta anotación son:
 - **Summary**: indica el nombre de la operación
 - **Description**: Indica una breve descripción de lo que el método o la llamada puede realizar.
 - **Tags**: Indica a que grupo pertenece la operación que se realiza en la llamada, en nuestro caso hemos asignado el nombre de la entidad a la que pertenezca la operación.
- **@ApiResponse**: Indica las distintas respuestas que se puede obtener ejecutando la llamada mediante la url indicada de cada uno de los métodos que se indican con el **@GetMapping** o **@PostMapping** en función del tipo de llamada que se realice.

Llamadas más interesantes

Para los ejemplos de las implementaciones de las siguientes llamadas, hemos decidido usar los métodos de la entidad Asignatura, ya que es la que más relaciones tiene y por lo tanto la que es más compleja hablando en relación con la base de datos.

- **POST (crear)**

Esta llamada implementada en la aplicación crea un objeto en base de datos con las propiedades que se pasan a través del body de la Request.

```
74 @Operation(summary = "Crear Asignatura", description = "Esta operación crea una nueva Asignatura y lo inserta en la base de datos", tags = "Asignatura")
75 @ApiResponses(value = {
76     @ApiResponse(responseCode = "200", description = "Se ha ejecutado la consulta correctamente", content = {
77         @Content(array = @ArraySchema(schema = @Schema(implementation = Asignatura.class))) },
78     @ApiResponse(responseCode = "409", description = "No se puede crear con ese id porque ya existe en la base de datos", content = {
79         @Content(array = @ArraySchema(schema = @Schema(implementation = ErrorObject409.class)), mediaType = "application/json") },
80     @ApiResponse(responseCode = "500",
81         description = "No se ha podido crear la Asignatura porque se añadieron atributos que no están creados en la base de datos", content = @Content),
82     @ApiResponse(responseCode = "404", description = "No se ha encontrado la asignatura con ese id", content = {
83         @Content(array = @ArraySchema(schema = @Schema(implementation = ErrorObject404.class)), mediaType = "application/json") },
84
85     @ApiResponse(responseCode = "400", description = "Solicitud errónea",
86         content = @Content(array = @ArraySchema(schema = @Schema(implementation = ErrorObject400.class)), mediaType = "application/json")) })
87 @PostMapping(path = "/añadir", consumes = "application/json")
88 public ResponseEntity<Asignatura> guardarAsignatura(@RequestBody AsignaturaRequest a) {
89
```

FIGURA 19. Implementación de la llamada Post de la entidad Asignatura.

En este método el parámetro, el cual está con la notación `@RequestBody` es la entidad `AsignaturaRequest`, la cual se ha creado para que mediante el body de la request se solicite sólo los parámetros y propiedades de la propia entidad `Asignatura`, para con ello, evitar que se cree relaciones a la hora de crear una entidad.

El método devuelve un `ResponseEntity`, el cual devuelve el estado de la respuesta que devuelve el servidor de la aplicación provocado por las acciones con la base de datos, además, en algunas respuestas que se verán a continuación, se devuelve el objeto creado o relacionado con la llamada implicada.

- DELETE (eliminar)

Esta llamada implementada en la aplicación elimina el objeto que tenga la id pasada como variable en la url de la llamada.

```
123 @Operation(summary = "Borrar Asignatura",
124           description = "Esta operación permite eliminar una asignatura introduciendo como parámetro su identificador (id)", tags = "Asignatura")
125 @ApiResponse(value = {
126     @ApiResponse(responseCode = "200", description = "Se ha borrado la asignatura de la base de datos correctamente", content = {
127         @Content(array = @ArraySchema(schema = @Schema(implementation = Asignatura.class))) }),
128     @ApiResponse(responseCode = "404", description = "No se ha encontrado la asignatura con ese id", content = {
129         @Content(array = @ArraySchema(schema = @Schema(implementation = ErrorObject404.class)), mediaType = "application/json") }),
130     @ApiResponse(responseCode = "400", description = "Solicitud errónea",
131         content = @Content(array = @ArraySchema(schema = @Schema(implementation = ErrorObject400.class)), mediaType = "application/json")) })
132
133 @DeleteMapping(path = "/eliminar/{id}")
134 public ResponseEntity<Asignatura> eliminarPorId(@PathVariable("id") Long id) {
135
```

FIGURA 20. Implementación de la llamada Delete de la entidad Asignatura.

En este método el parámetro que se recibe la id de la asignatura que se quiere eliminar, el cual comprueba si existe la base de datos y si existe, lo elimina de la base de datos del sistema.

- PUT (actualizar)

En esta llamada se realiza una actualización del objeto que se indique en la llamada a la Api. En el caso que vamos a mostrar a continuación, esta actualización se va a usar como método para añadir y crear relaciones entre las entidades `Asignaturas` y `alumnos`.

```
151 @Operation(summary = "Añadir alumno nuevo a la asignatura",
152           description = "Esta operación crea un nuevo alumno y lo añade a la asignatura", tags = "Asignatura")
153 @ApiResponse(value = {
154     @ApiResponse(responseCode = "200", description = "Se ha ejecutado la consulta correctamente", content = {
155         @Content(array = @ArraySchema(schema = @Schema(implementation = Asignatura.class))) }),
156     @ApiResponse(responseCode = "500", description = "No se ha podido añadir el alumno", content = @Content),
157     @ApiResponse(responseCode = "404", description = "No se ha encontrado la asignatura con ese id", content = {
158         @Content(array = @ArraySchema(schema = @Schema(implementation = ErrorObject404.class)), mediaType = "application/json") }),
159     @ApiResponse(responseCode = "409", description = "No se puede crear con ese id porque ya existe en la base de datos", content = {
160         @Content(array = @ArraySchema(schema = @Schema(implementation = ErrorObject409.class)), mediaType = "application/json") }),
161     @ApiResponse(responseCode = "400",
162         description = "Solicitud errónea", content = @Content(array = @ArraySchema(schema = @Schema(implementation = ErrorObject400.class)),
163         mediaType = "application/json")) })
164 @PutMapping("/{id_asignatura}/añadirAlumno")
165 public ResponseEntity<Asignatura> anadeAlumnoAsignatura(@PathVariable("id_asignatura") final long id,
166 @RequestBody final AlumnoRequest a) {
167
```

FIGURA 21. Implementación de la llamada PUT de la entidad Asignatura.

En este método se puede ver que los parámetros que reciben son la id de la asignatura que se pretende actualizar, la cual se recibe mediante variable en la url de la llamada. Luego, también se recibe mediante el cuerpo de la request los parámetros necesarios para crear un

alumno, los cuales poseen las mismas validaciones que el propio método de creación del alumno.

La respuesta de la aplicación es similar a las respuestas mostradas en las llamadas anteriores.

GET (obtener) Este método obtiene datos y registros pertenecientes de la base de datos de la entidad que se le solicite, en este caso vamos a mostrar un ejemplo de un Get un poco más avanzado que hay implementado en la aplicación para obtener todos los profesores que tienen como nombre el parámetro introducido en la llamada mediante una variable en la url.

```
--
69 @Operation(summary = "Obtener profesores por nombre",
70 description = "Esta operacion devuelve todos los profesores de la pagina web que tengan el nombre introducido como parametro", tags = "Profesor")
71 @ApiResponse(value = {
72     @ApiResponse(responseCode = "200", description = "Se han obtenido todos los resultados de profesores de la base de datos correctamente", content = {
73         @Content(array = @ArraySchema(schema = @Schema(implementation = Profesor.class)),
74         mediaType = "application/json" )}),
75     @ApiResponse(responseCode = "404", description = "No se ha encontrado el profesor con ese id", content = {
76         @Content(array = @ArraySchema(schema = @Schema(implementation = ErrorObject404.class)), mediaType = "application/json" )}),
77     @ApiResponse(responseCode = "400", description = "Solicitud erronea",
78     content = @Content(array = @ArraySchema(schema = @Schema(implementation = ErrorObject400.class)), mediaType = "application/json" )})
79 @GetMapping("/{nombre}")
80 public String obtenerProfesorPorNombre(@PathVariable("nombre") final String nombre) throws JsonProcessingException {
81     ...
--
```

FIGURA 22. Implementación de la llamada GET de la entidad Profesor.

Respuestas de la API

A continuación, vamos a exponer las distintas posibles respuestas que se pueden obtener por parte de la aplicación, la mayoría son respuestas que se dan en cualquier servidor que podamos usar en la web:

- **Código 200:** En el caso de que se reciba este código, obtendremos una respuesta correcta por parte del servidor.
- **Código 400:** En el caso de que se reciba este código, quiere decir que la request enviada por parte del cliente tiene un formato erróneo.
- **Código 404:** En el caso de que se reciba este código, quiere decir que uno de los parámetros no se ha encontrado en la base de datos, en nuestro caso suele ser la id que se pasa mediante url.
- **Código 409:** En el caso de que se reciba este código, quiere decir que hay un conflicto entre varios registros de la base de datos y por lo tanto no se puede realizar la consulta correctamente.
- **Código 500:** En el caso de que se reciba este código, quiere decir que hay un problema interno en el servidor de la aplicación en la propia base de datos.

Scrapping

Cómo se ha comentado anteriormente en este documento, se ha realizado una inyección inicial de datos reales del ámbito universitario, más concretamente los datos inyectados son los datos relacionados con la Escuela Técnica Superior de Ingeniería Informática junto a todas las titulaciones que se imparten en ella. Estos datos son extraídos mediante una técnica llamada Scrapping la cual mediante el código HTML de la página se extraen los datos deseados.

El código que se ha usado para la extracción de estos datos no pertenece a este proyecto, sino a un proyecto en desarrollo llamado DlgaApp de uno de los integrantes del grupo, el cual se puede revisar en el siguiente repositorio de GitHub (<https://github.com/A-Blanco/DlgaApp>). Los datos son extraídos de las distintas secciones y repositorio de datos que ofrece públicamente la página principal de la universidad de Sevilla (<https://www.us.es/>).

Al no estar relacionado directamente el apartado de la obtención de los datos iniciales en este proyecto, el equipo ha decidido no añadir su implementación en la aplicación, por lo tanto, gracias a MySQL WorkBench hemos exportado los datos de la base de datos del proyecto DlgaApp mediante una función que ofrece dicho programa.

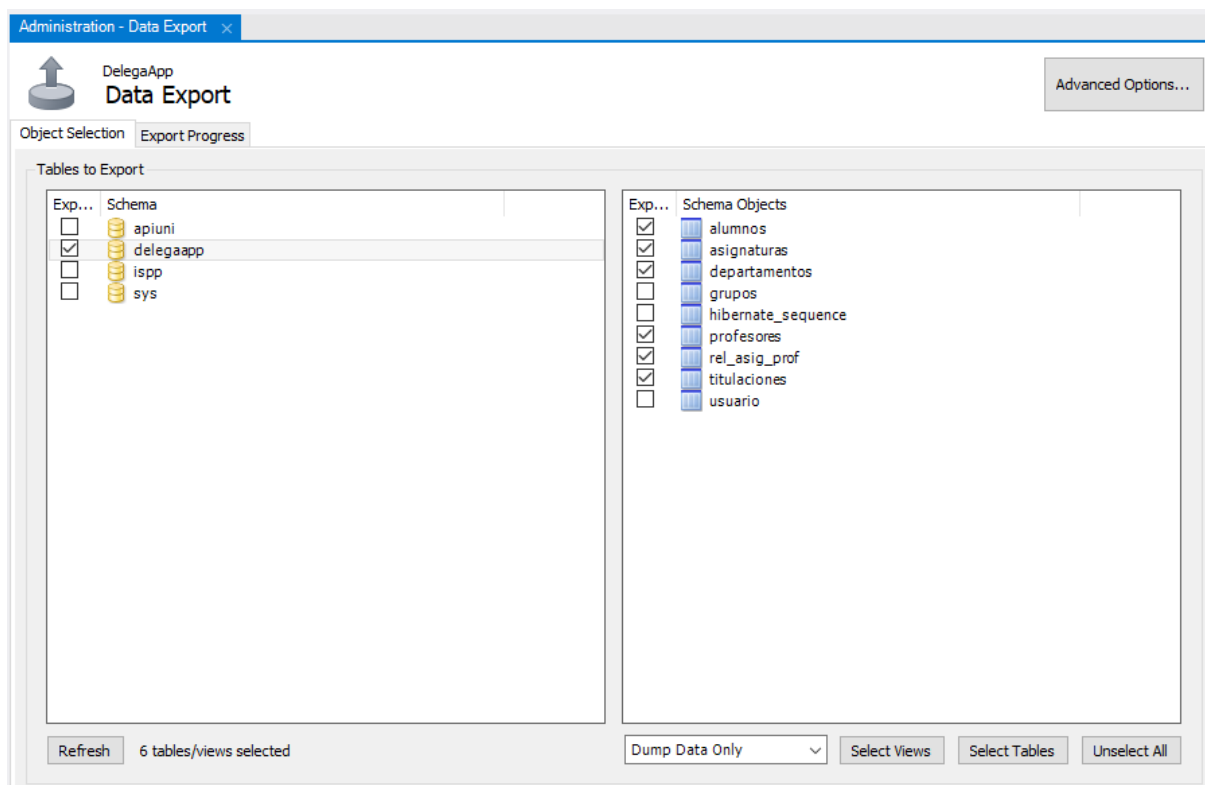


FIGURA 23. Opción Data Export MySQLWorkBench.

Estos datos han sido introducidos mediante consultas INSERT indicadas en el archivo data.sql anteriormente comentado. Previamente se han modificado algunas incompatibilidades existentes entre las bases de datos de los distintos proyectos.

Conclusiones

Como conclusión después de realizar este proyecto, las sensaciones son positivas, ya que hemos aprendido a implementar nuestra propia API REST con la base de datos. Lo aprendido en este proyecto seguro que nos ayuda de cara al Trabajo de fin de grado e incluso cuando empezemos a trabajar, ya que es algo nuevo que hemos aprendido.

También ha reforzado nuestros conocimientos sobre Spring, ya que teníamos algunos previos, pero esto nos ha reforzado algo más, sobre todo con la parte de integrar una base de datos al proyecto.

En cuanto a la base de datos del proyecto, nos ha ayudado a conocer un poco más profundo cómo funciona mejor las relaciones entre las distintas entidades ya que, aunque previamente teníamos un conocimiento general del funcionamiento de Spring respecto a las entidades y su funcionamiento, no teníamos un conocimiento tan profundo como el que hemos necesitado a la hora de relacionar algunas entidades del proyecto como puede ser la entidad Asignatura, la cual tiene varias relaciones a un mismo “nivel” con distintas entidades o tablas de la base de datos. Por lo tanto, nos ha ayudado a conocer algunos funcionamientos que desconocíamos de otros proyectos.

Bibliografía

1. (Documentación sobre @PreRemove) <https://www.programcreek.com/java-api-examples/?api=javax.persistence.PreRemove>
2. (Enlace repositorio de proyecto de apoyo DlegaApp) <https://github.com/A-Blanco/DlgaApp>
3. (Documentación sobre la función de Data Export) <https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html>
4. (Documentación del uso ResponseEntity) <https://www.baeldung.com/spring-response-entity>
5. https://www.youtube.com/watch?v=ghn9p6d_Yc&list=WL&index=7
6. <https://www.youtube.com/watch?v=cIN5JOUakVs&list=WL&index=9&t=140sw.baeldung.com/spring-response-entity>
7. <https://www.youtube.com/watch?v=mViFmjcdOoA&list=WL&index=11>