INTRODUCCIÓN AL DESARROLLO DE APIs EN SPRING BOOT

1. ¿Qué es una API?

Una **API** (Application Programming Interface en inglés, o Interfaz de programación de aplicaciones en español) es un conjunto de definiciones y reglas bien definidas que permiten la comunicación de diferentes programas entre sí.

Las APIs permiten que sus productos y servicios se comuniquen con otros, sin necesidad de saber cómo están implementados. Esto simplifica el desarrollo de las aplicaciones y permite ahorra tiempo y dinero al aportar flexibilidad.

• API REST

Hay un tipo de API que se llama **API REST**, en la cual me voy a centrar, que es una interfaz de comunicación entre sistemas de información que usar el protocolo HTTP para obtener datos o ejecutar operaciones sobre esos datos en formatos distintos como pueden ser XML o JSON.

REST viene de **RE**presentational **S**tate **T**ransfer, es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP. REST se compone de una lista de reglas que se deben cumplir en el diseño de la arquitectura de una API.

API REST se basa en el modelo cliente-servidor, donde el cliente es el que solicita obtener los recursos o realizar alguna operación sobre esos datos, mientras que el servidor es el que se encarga de entregar o procesar dichos datos de la solicitud del cliente.

- ### Criterios de una API REST

Puesto que no todas las API son REST, hay diversos criterios para identificar si una API es de tipo REST, como los siguientes:

- Debe usar una arquitectura cliente-servidor.
- Las ejecuciones de la API no deben considerar el estado del cliente, el estado de peticiones anteriores o algún indiciador almacenado que haga variar su comportamiento. Es decir, la comunicación debe ser sin estado (stateless).
- Ha de estar orientada a recursos, usando las operaciones estándar de los verbos HTTP, que se explicarán más adelante.
- Debe hacer uso de la URL como identificador único de los recursos.
- Debe ser hipermedia: cuando se consulte un recurso, este debe contener links o hipervínculos de acciones o recursos que lo complementen.

2. ¿Qué y cuáles son los verbos HTTP?

Los verbos HTTP son aquellos verbos propios del protocolo HTTP que fueron tomados para definir operaciones muy puntuales y específicas sobre los recursos de la API. Los más utilizados son:

1. **GET**: Sirve para el listado de recursos.
2. **POST**: Creación de un recurso.
3. **PUT**: Modificación total de un recurso.
4. **DELETE**: Eliminación de un recurso. Muchas veces es un ***soft delete***, es decir, no se elimina definitivamente el recurso, sino que únicamente es marcado como eliminado o desactivado.
5. **PATCH**: Modificación parcial de un recurso.

- ### Diferencias entre RESTful y RESTless

Muchas veces se escuchan o se leen estos términos en el momento del diseño, construcción o interacción con una API, por lo que se va a explicar la diferencia.

Es sencilla la diferencia, se llaman **RESTful** a todas las APIs que cumplen completamente los crietrios REST antes explicados, mientras que se llaman **RESTless** a aquellas APIs que no cumplen del todo con los criterios REST.

Por ejemplo, si una API utiliza el verbo HTTP POST para todas sus operaciones no es una API RESTful, sino una API RESTless.

- ### HTTP Status

El protocolo HTTP tiene estatus de respuesta propios que fueron tomados para informar sobre el resultado de la operación solicitada. Los más comunes en API Rest son los siguientes:

1. **200**: OK
2. **201**: Creado
3. **204**: Contenido vacío
4. **400**: Bad request
5. **401**: Desautorizado
6. **403**: Prohibido
7. **404**: No encontrado / Not found
8. **500**: Error del servidor interno / Internal Server Error

- ## API Spectification

La especificación de una API es aquella documentación donde se describe el comportamiento de una API, a esto también se le conoce como el contrato de la API.

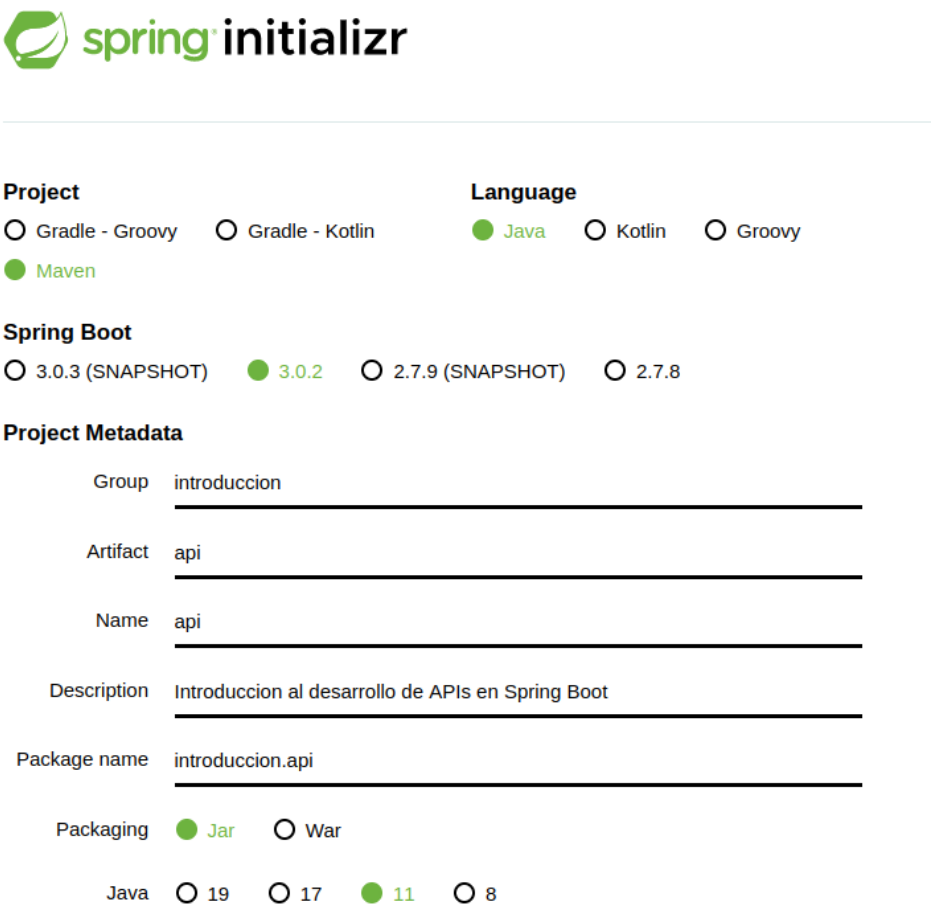
La finalidad de dicha documentación es guiar al desarrollador que va a integrar el uso de la API en u sistema. Es tanta la importancia que tiene esto que actualmente existen diversas herramientas y estándares creados específicamente para describir una API REST, algunas de estas herramientas son RAML, el estándar OpenAPI, o Swagger, de esta última se hablará más adelante. En el contrato de la API se especifican los verbos HTTP y los HTTP status.

3. Código con ejemplo

A continuación, desarrollaré una API REST con Spring Boot, una tecnologgía de Spring. Cabe mencionar que Spring framework es la tecnología más utilizada para el lenguaje de programación Java.

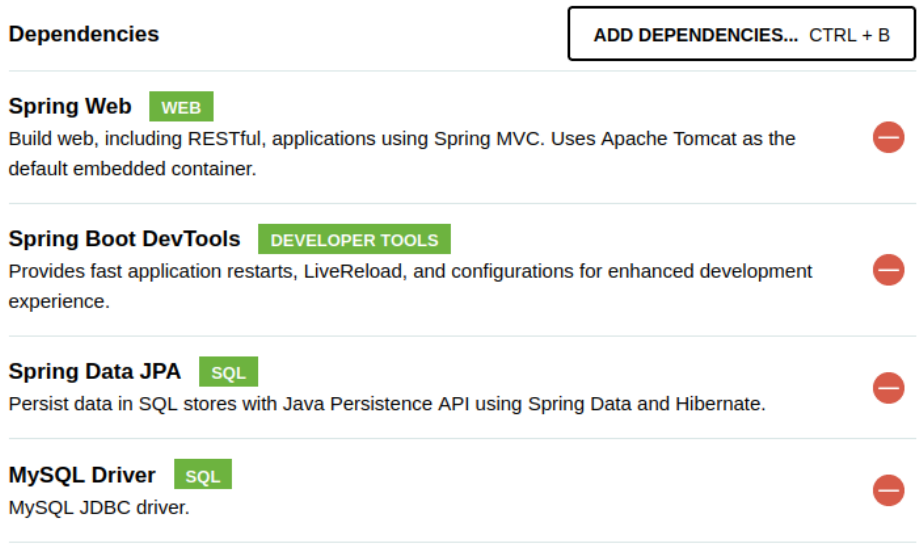
Gracias a Spring Boot nos facilitaremos las configuraciones como las dependencias de conexión con la base de datos, que en este caso será MySQL, despliegue del servidor, etc. Para el gestor de aplicación utilizaré Maven.

Para crear el proyecto con Spring Boot se realizará gracias a la herramienta gratuita de internet Spring Initializr, cuya página web es https://start.spring.io/

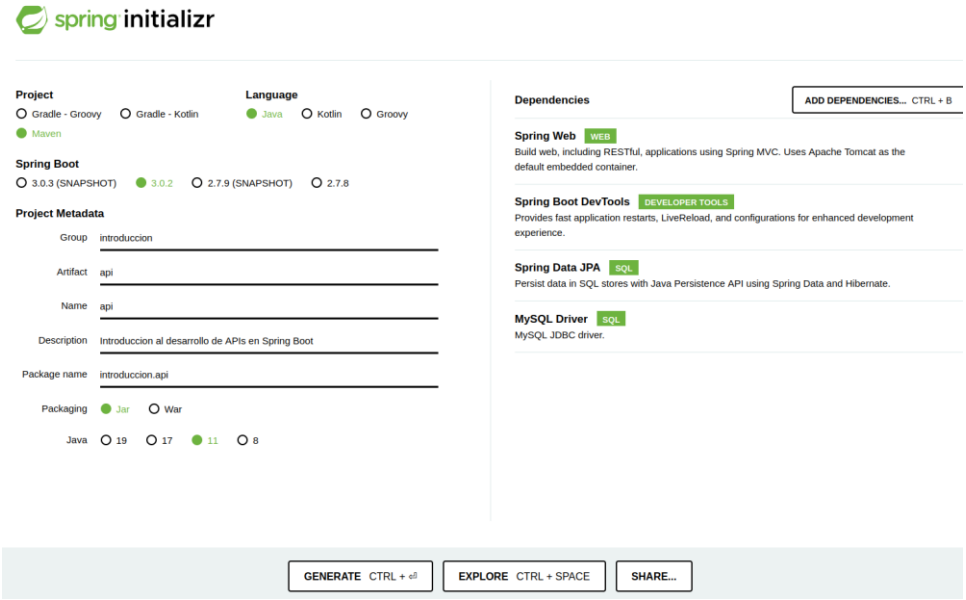


Como se observa en la imagen, el tipo de proyecto será Maven como antes mencionado, el lenguaje Java, la versión de Spring Boot será la útilma actual que no sea SNAPSHOT, y el

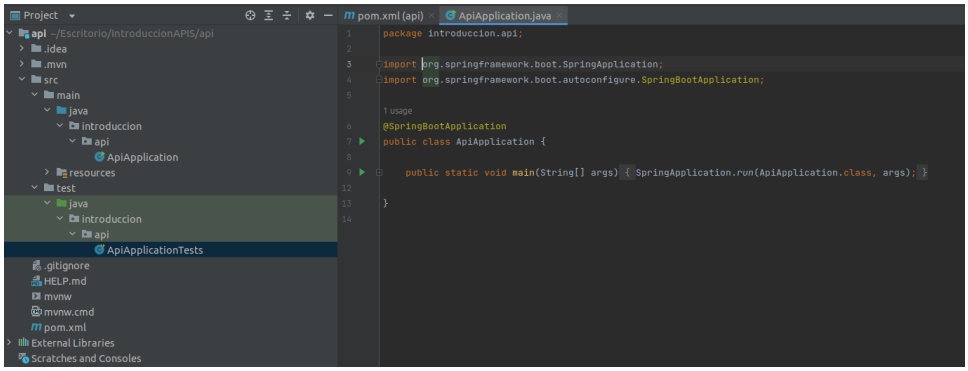
metadata del proyecto lo que se observa en la imagen, con el packaging jar y version de Java 11.



A continuación, se muestran las dependencias que se utilizarán, es recomendable añadirlas ahora y no luego, así cuando se cree el proyecto viene todo integrado ya.

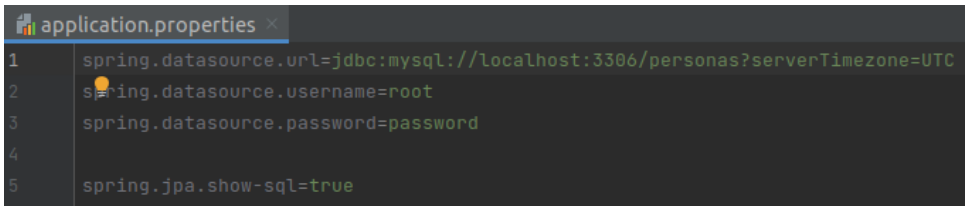


Damos clic en Generate para descargar el proyecto, y una vez descargado se descomprime.
Para el desarrollo de la aplicación yo utilizaré IntelliJ pero se puede utilizar la herramienta con la que cada uno esté más a gusto. Una vez importado se espera para que se resuelvan las dependencias y se descargue bien todo y a su vez se coja el proyecto como un maven proyect.



Como se observa en la imagen ya se ha creado todo bien y están las dependencias en el pom.xml.

En el archivo **application.properties** que se encuentra en la ruta de acceso src/main/resources/ se agrega el string de conexión para ingresar a la base de datos que crearemos:

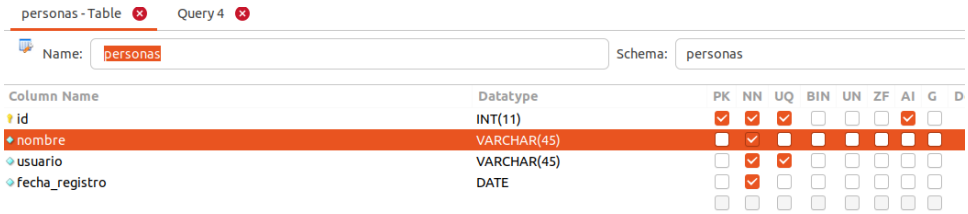


En el caso de password se pone la de la base de datos, yo pongo password debido a que es asi como lo tengo definido en MySQL.

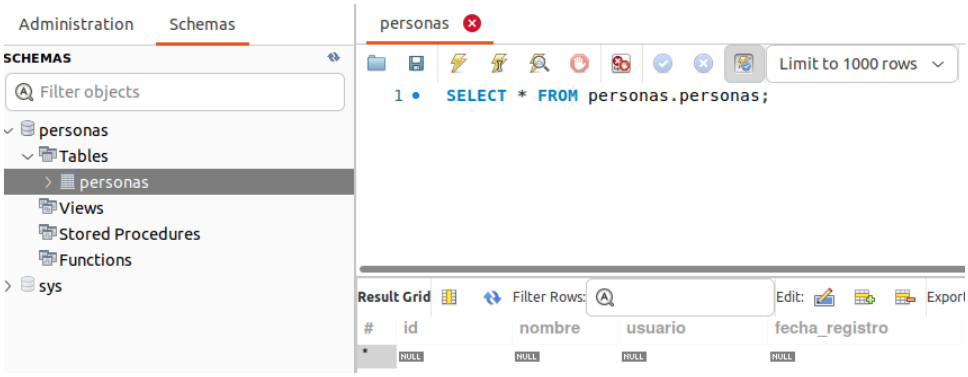
Una vez hecho esto vamos al MySQL y ejecutamos el comando

CREATE DATABASE personas;

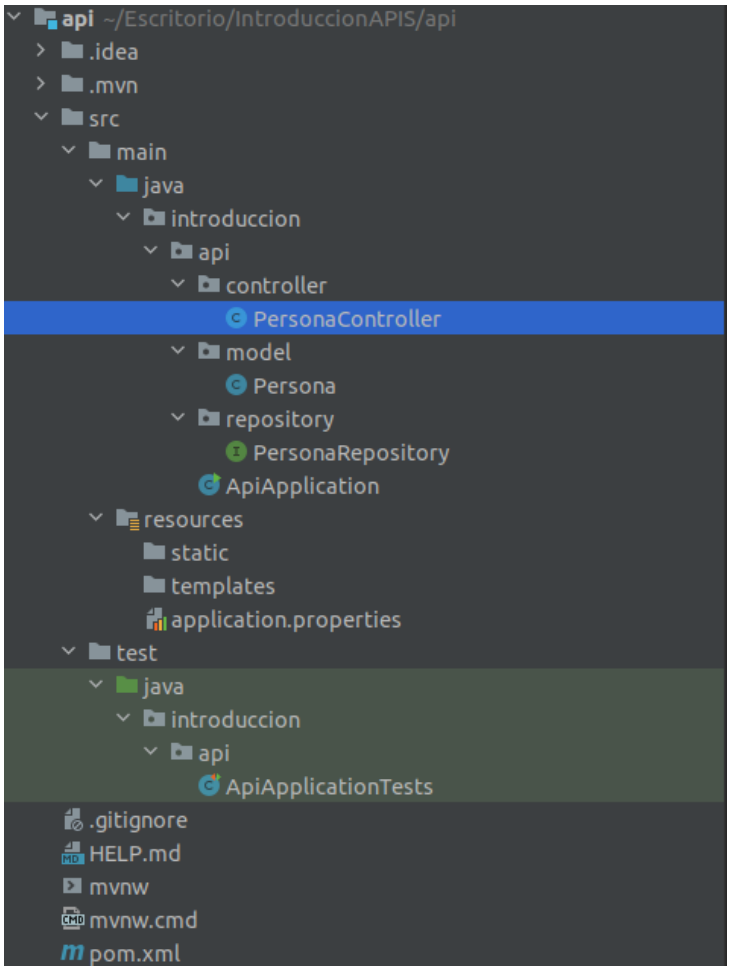
Tras esto creamos una tabla nueva que tendrá los siguientes campos:



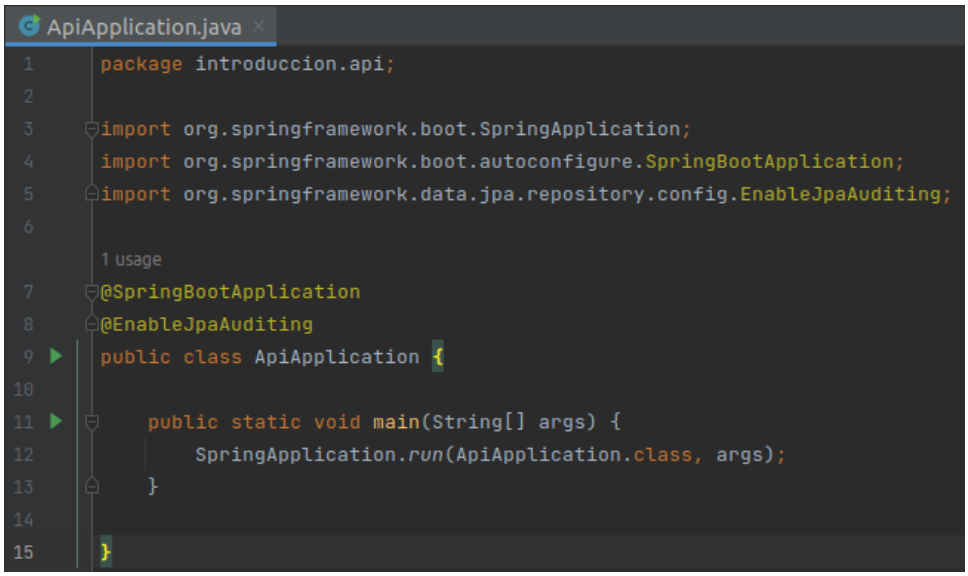
Una vez dado a aplicar el resultado tiene que ser el siguiente:



Ahora procederemos al código del IntelliJ, pero antes tenemos que definir la estructura, como es un ejemplo será sencilla y solo contendrá repositorio, modelo y controlador además de la aplicación para que funcione con Spring Boot.



- **ApiApplication**



Contiene lo necesario para conectar a la BBDD y que vaya bien con Spring Boot. Es la clase que arranca cuando iniciamos la aplicación.

- **Controller**

La forma de realizar las operaciones será bien a partir de una id o el nombre de usuario, para elegir el método que se desee. Normalmente es recomendable la ID porque eso nunca cambiará ya que el nombre de usuario puede cambiarse.

PersonaController.java

```java
package introduccion.api.controller;

import introduccion.api.model.Persona;
import introduccion.api.repository.PersonaRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class PersonaController {

    10 usages
    @Autowired
    private PersonaRepository personaRepository;

    @GetMapping("/personas")
    public List<Persona> getPersonas() {
        return personaRepository.findAll();
    }

    @GetMapping("/personas/nombre")
    public List<Persona> getPersonasByNombre(@PathVariable("nombre") String nombre) {
        return personaRepository.findByNombre(nombre);
    }

    @GetMapping("/personas/usuario")
    public Persona getPersonaByUsuario(@PathVariable("usuario") String usuario) {
        return personaRepository.findByUsuario(usuario);
    }

    @GetMapping("/personas/usuario/existe")
    public boolean existeUsuario(@PathVariable("usuario") String usuario) {
        return personaRepository.existsByUsuario(usuario);
    }
```

PersonaController.java

```java
        return personaRepository.findByNombre(nombre);
    }

    @GetMapping("/personas/usuario")
    public Persona getPersonaByUsuario(@PathVariable("usuario") String usuario) {
        return personaRepository.findByUsuario(usuario);
    }

    @GetMapping("/personas/usuario/existe")
    public boolean existeUsuario(@PathVariable("usuario") String usuario) {
        return personaRepository.existsByUsuario(usuario);
    }

    @PostMapping("/persona")
    public Persona crearPersona(@RequestBody Persona persona) {
        return personaRepository.save(persona);
    }

    @PutMapping("/persona/{id}")
    public Persona actualizarPersona(@PathVariable int id, @RequestBody Persona persona) {
        return personaRepository.save(persona);
    }

    @PutMapping("/persona/usuario/{usuario}")
    public Persona actualizarPersonaByUsuario(@PathVariable String usuario, @RequestBody Persona persona) {
        Persona personaActual = personaRepository.findByUsuario(usuario);
        personaActual.setNombre(persona.getNombre());
        personaActual.setUsuario(persona.getUsuario());
        return personaRepository.save(personaActual);
    }

    @DeleteMapping("/persona/{id}")
    public void eliminarPersona(@PathVariable("id") Long id) { personaRepository.deleteById(id); }

    @DeleteMapping("/persona/usuario/{usuario}")
    public void eliminarPersonaByUsuario(@PathVariable String usuario) { personaRepository.deleteByUsuario(usuario); }
}
```

Esta clase le indica a Spring que la clase va a ser un controlador de una API REST, nos proveerá de todos los métodos para un CRUD de la clase Persona.

- **Model**

Persona.java

```java
package introduccion.api.model;

import javax.persistence.*;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import java.util.Date;

15 usages
@Entity
@Table(name = "personas")
@EntityListeners(AuditingEntityListener.class)
public class Persona {
    4 usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    4 usages
    @Column(name = "nombre")
    private String nombre;

    4 usages
    @Column(name = "usuario")
    private String usuario;

    4 usages
    @Column(name = "fecha_registro")
    @CreatedDate
    private Date fechaRegistro;

    public Persona() {
    }
```

Persona.java

```java
    public Persona() {
    }

    public Persona(Long id, String nombre, String usuario, Date fechaRegistro) {
        this.id = id;
        this.nombre = nombre;
        this.usuario = usuario;
        this.fechaRegistro = fechaRegistro;
    }
```

Y a su vez los setters y getters además del toString

```java
@Override
public String toString() {
    return "Persona{" + "id=" + id + ", nombre='" + nombre + '\'' + ", usuario='" + usuario + '\'' +
            ", fechaRegistro=" + fechaRegistro + '}';
}
```

Es la entidad de nuestro objeto Persona, clase que va a mapear los atributos como campos de la base de datos de MySQL.

- **Repository**

PersonaRepository.java

```java
package introduccion.api.repository;

import introduccion.api.model.Persona;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface PersonaRepository extends JpaRepository<Persona, Long> {

    List<Persona> findByNombre(@Param("nombre") String nombre);
    Persona findByUsuario(@Param("usuario") String usuario);
    boolean existsByUsuario(@Param("usuario") String usuario);
    void deleteByUsuario(@Param("usuario") String usuario);
}
```

Esta clase nos ayuda a realizar operaciones en nuestra base de datos sin necesidad de escribir tantas líneas de código. Spring Data lo realiza por nosotros, operaciones como delete, save, findAll se realizan automáticamente.

- **pom.xml**

Al darme un error a mí en los archivos de springboot 3.0.2 he decidido cambiar la versión a la 2.7.0 y he vuelto a dar a
- Download Sources and Documentation, Genereate Sources and Update Folders y a Reload project, tras esto se da run al ApiApplication.

pom.xml (api)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>introduccion</groupId>
    <artifactId>api</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>api</name>
    <description>Introduccion al desarrollo de APIs en Spring Boot</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</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>
```

pom.xml (api)

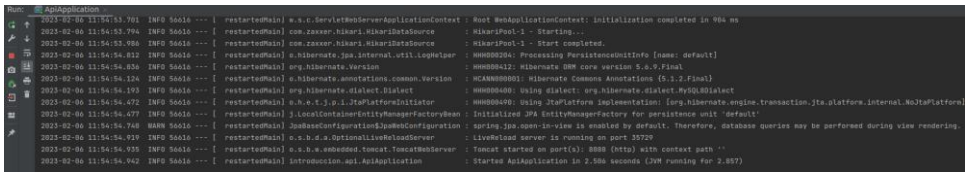
```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </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.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

</project>
```

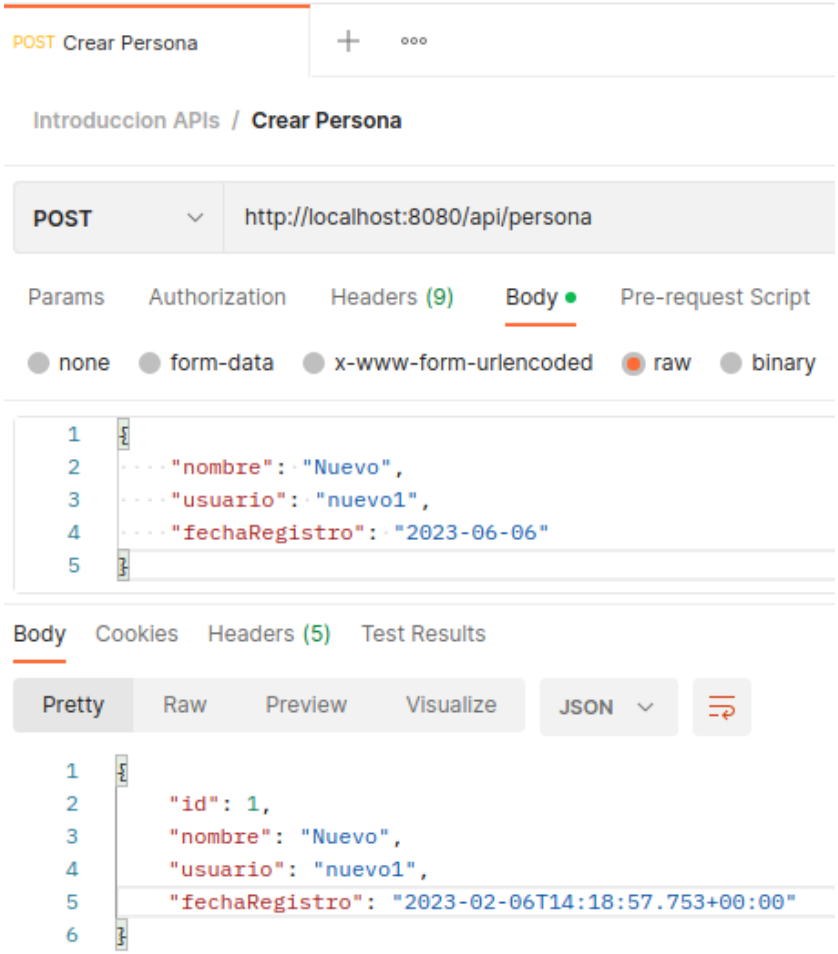
A continuación, se le da al run de ApiApplication y tiene que aparecer lo siguiente:



Pruebas con Postman

Para realizar las pruebas correspondientes utilizaré Postman, una aplicación diseñada para probar APIs.

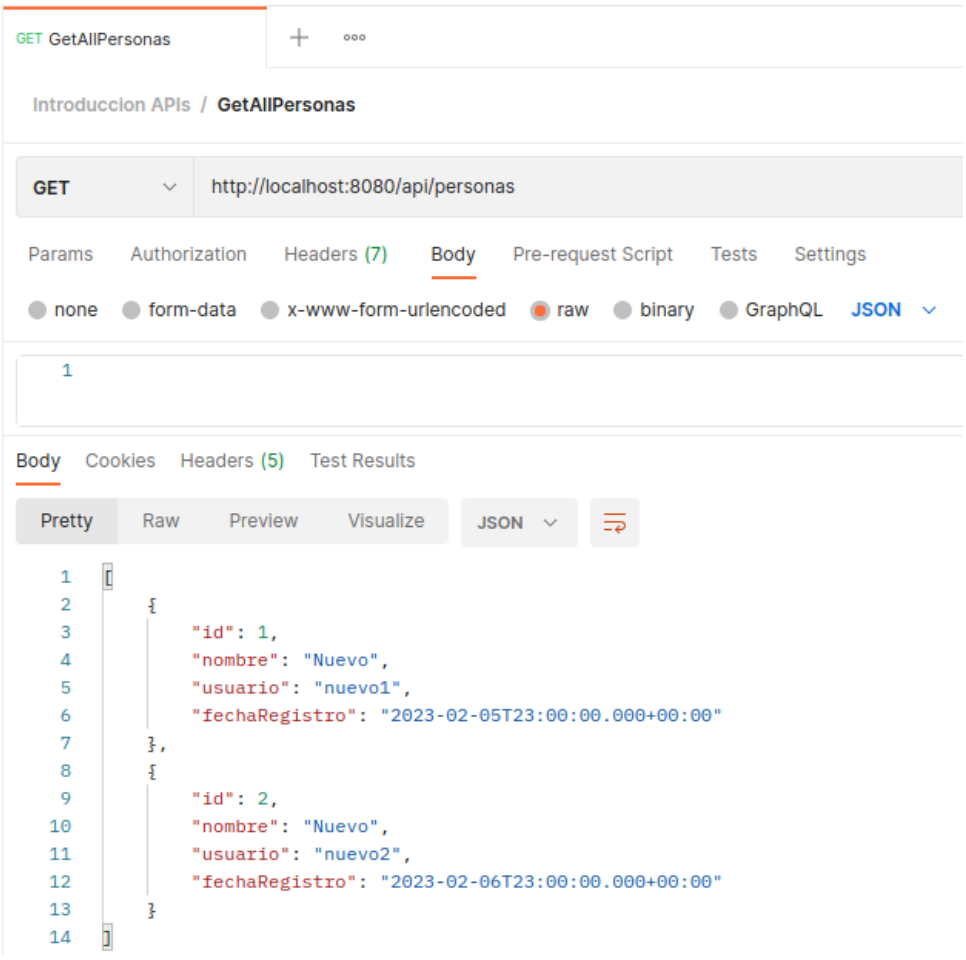
- Post



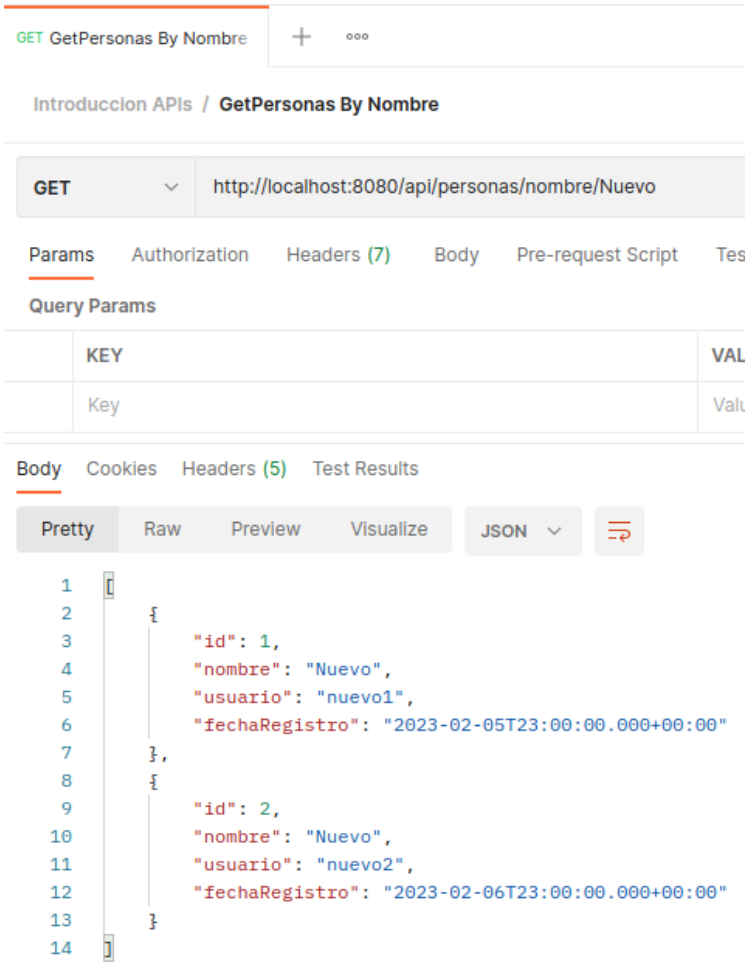
Como se observa se utiliza el verbo HTTP POST y se crea una persona pasandole un JSON con el nombre,usuario y una fechaRegistro que deseemos que luego esa se pone como la actual.

- Get

Para estas pruebas voy a crear un nuevo usuario, llamado Nombre y de usuario nuevo2.

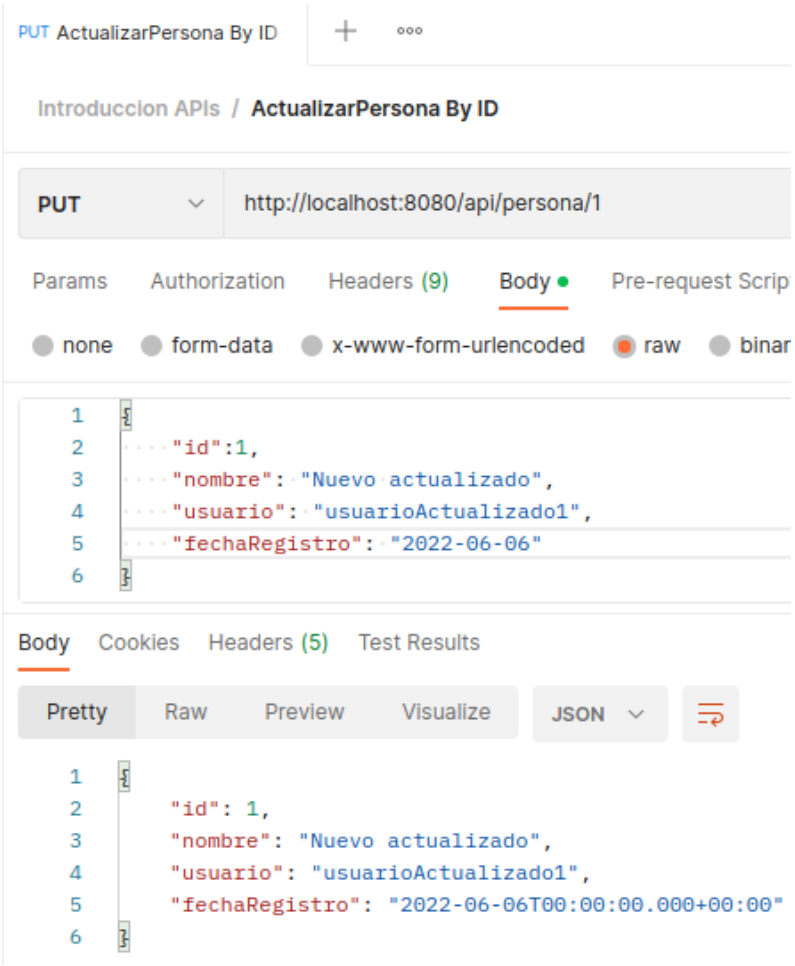


Se observa que el get de todas las personas salen los 2 usuarios.



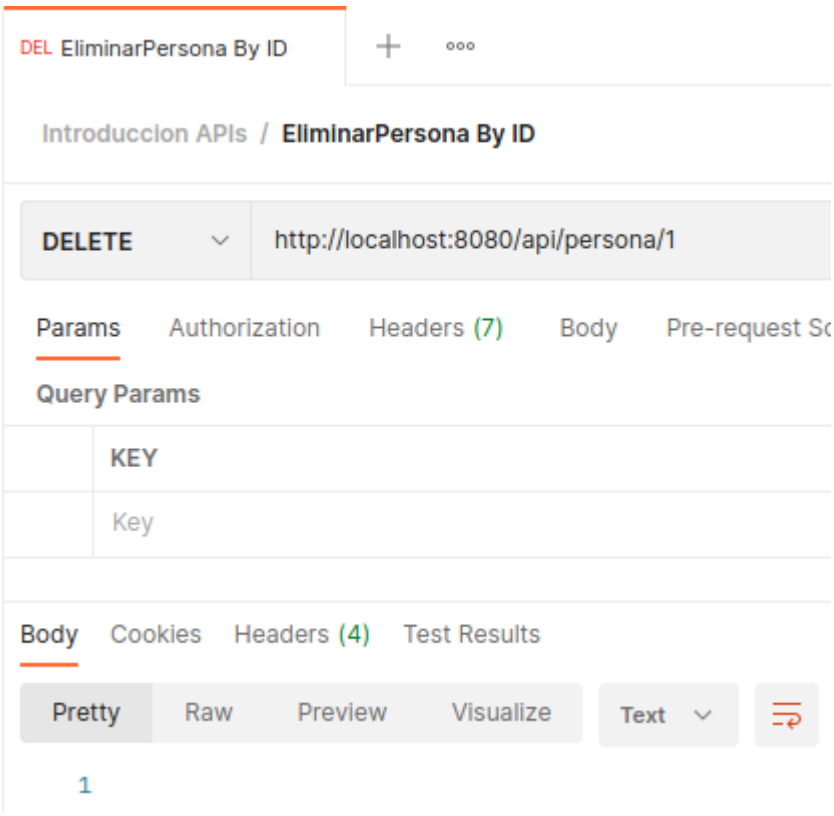
Ahora con un parámetro en la URL, se observa que se muestran todos los usuarios que se llamen de esa manera.

- Put

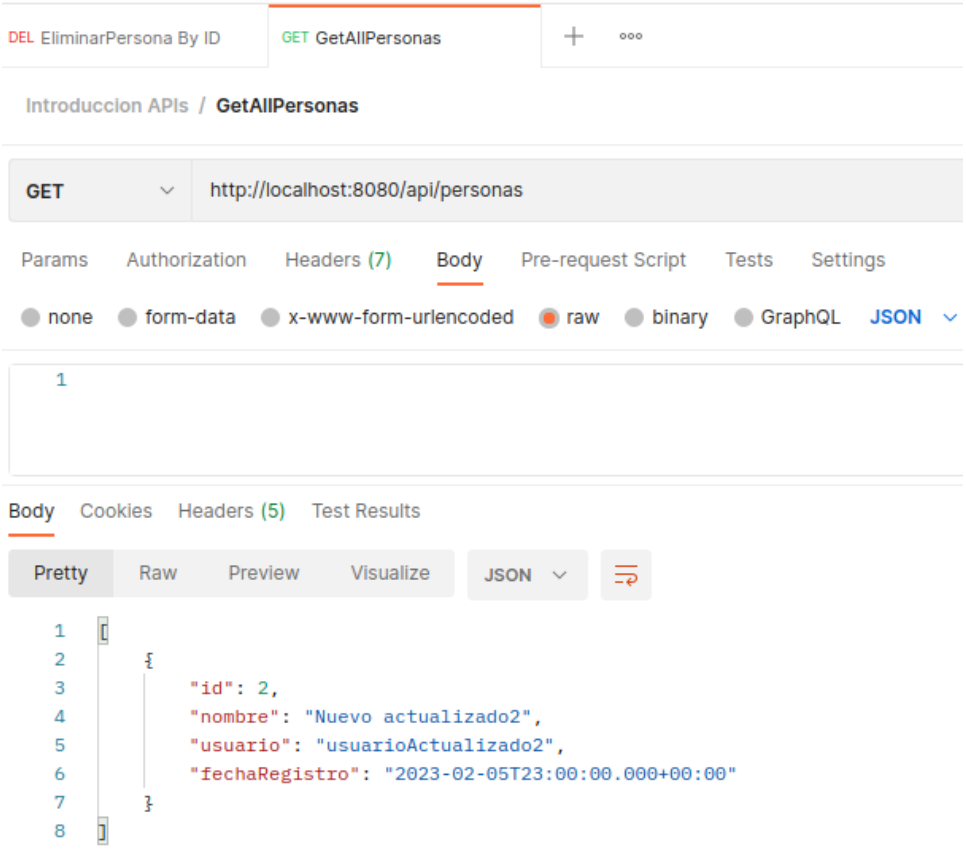


Se actualiza el usuario que deseemos con los nuevos valores que queremos.

- Delete

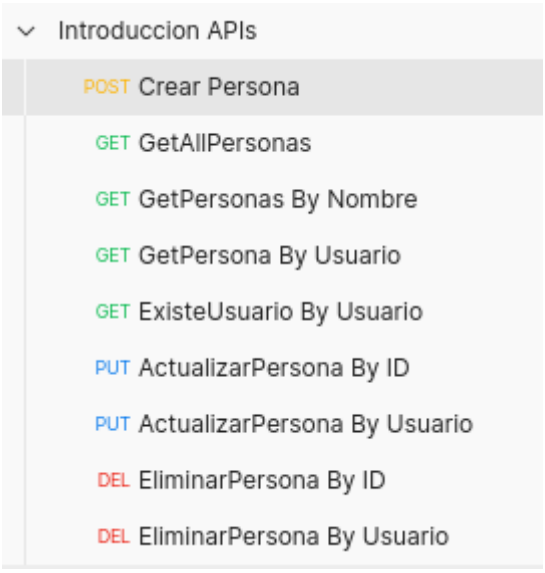


Se observa que no sale ningún mensaje, se puede ccambiar en el código o bien ver el get de todas las personas o el ID:



Como se observa funciona todo bien y la API REST está activa y operativa para hacer operaciones con los usuarios. No se han mostrado fotos de la BBDD, pero para que esto funcione es necesario que en la BBDD esté todo correcto y como se observa lo está.

Resumen de todas las acciones vistas desde Postman:



4. Swagger de la API

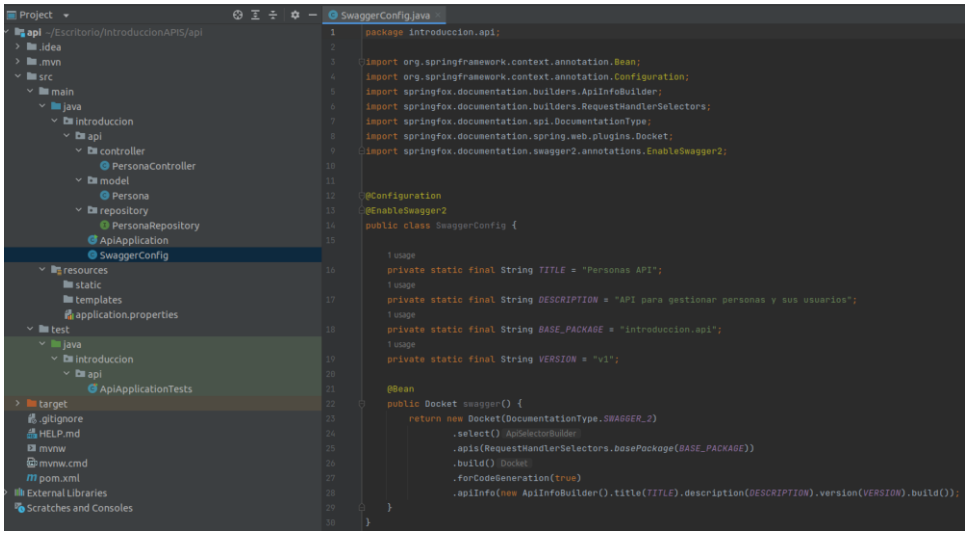
Como se ha dicho antes es necesario saber el contrato de una API fácilmente por lo que se utilizan diferentes herramientas.

Una de estas herramientas es Swagger, que con simplemente una configuración en el código y unas dependencias se sabe desde una página web propia de Swagger cuales son los verbos HTTP, la descripción de cada verbo y las respuestas que estos ofrecen.

Los cambios que hay que hacer son los siguientes:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

Primero en el pom.xml se añaden estas 2 dependencias. Tras esto es necesario añadir una nueva clase que llamaremos SwaggerConfig.java



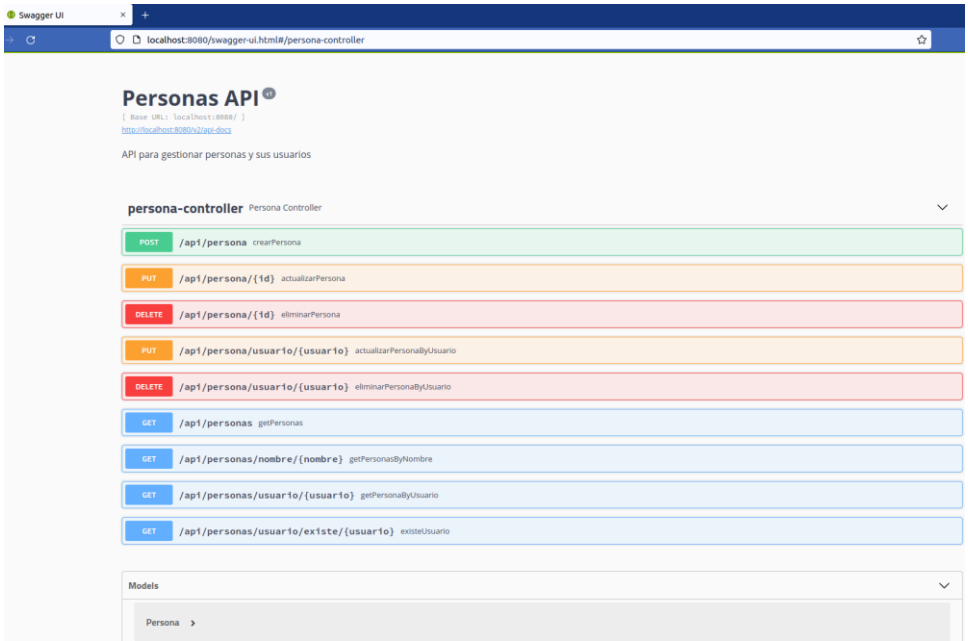
Esta clase tendrá este código y estará en la carpeta de api.

Por último, es necesario modificar el application.properties y añadir una línea de código:

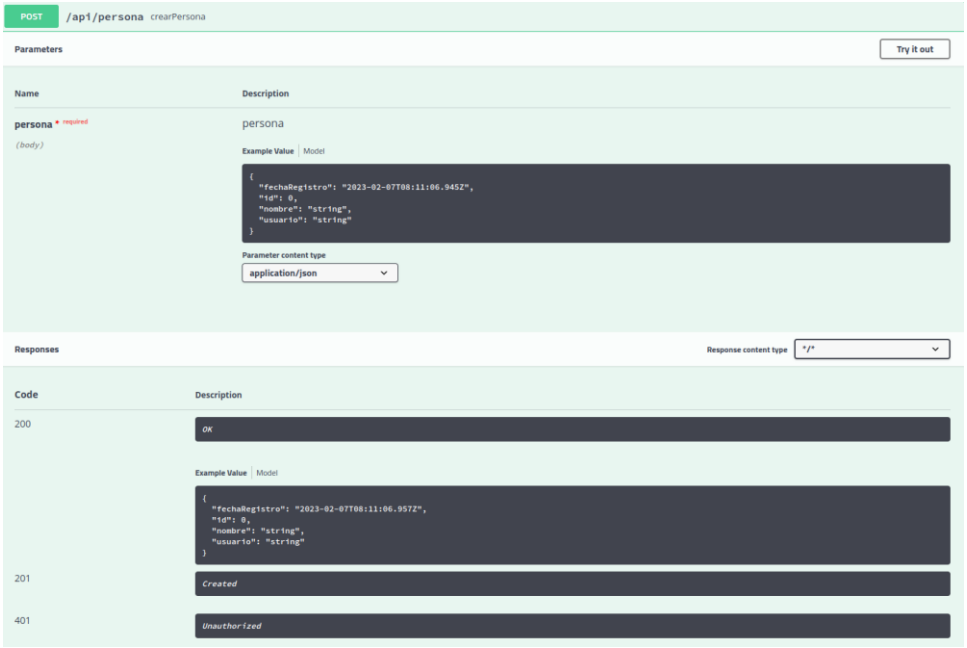
spring.mvc.pathmatch.matching-strategy = ANT_PATH_MATCHER

Se añade al final para que todo vaya correcto.

Ahora ya se puede abrir la página de swagger, una vez esté corriendo nuestra aplicación vamos a la URL: http://localhost:8080/swagger-ui.html#/



Como se observa salen todos los verbos del controlador y como se le llama a cada uno, a su vez también está el modelo de Persona antes creado.



Aquí se puede ver cómo sería una llamada a la API, con un ejemplo de cómo llamarlo y las respuestas.

Se observa que esto es muy útil, porque siendo este un proyecto super pequeño ya se puede ver mucho, en un proyecto grande será super importante.