

# INTRODUCCIÓN AL DESARROLLO DE APIs EN SPRING BOOT

INTRODUCCIÓN AL DESARROLLO DE APIs EN SPRING BOOT .....	1
1. ¿Qué es una API? .....	1
• API REST .....	1
• Criterios de una API REST .....	2
2. ¿Qué y cuáles son los verbos HTTP? .....	2
• Diferencias entre RESTful y RESTless .....	2
• HTTP Status .....	2
• API Spectification .....	3
3. Código con ejemplo .....	3
Pruebas con Postman .....	11
4. Swagger de la API.....	17

## 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 ahorrar 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 usan 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 indicador 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 criterios 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 Specification**

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 un 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 tecnología 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/>



---

**Project**

☐ Gradle - Groovy   ☐ Gradle - Kotlin   ☒ Java   ☐ Kotlin   ☐ Groovy

☒ Maven

**Spring Boot**

☐ 3.0.3 (SNAPSHOT)   ☒ 3.0.2   ☐ 2.7.9 (SNAPSHOT)   ☐ 2.7.8

**Project Metadata**

Group

introduccion

Artifact

api

Name

api

Description

Introduccion al desarrollo de APIs en Spring Boot

Package name

introduccion.api

Packaging

☒ Jar   ☐ War

Java

☐ 19   ☐ 17   ☒ 11   ☐ 8

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 última 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.

**Dependencies**

ADD DEPENDENCIES... CTRL + B

**Spring Web** WEB  
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

**Spring Boot DevTools** DEVELOPER TOOLS  
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

**Spring Data JPA** SQL  
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

**MySQL Driver** SQL  
MySQL JDBC driver.

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.



**Project**  
☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ Maven

**Language**  
☒ Java ☐ Kotlin ☐ Groovy

**Spring Boot**  
☐ 3.0.3 (SNAPSHOT) ☒ 3.0.2 ☐ 2.7.9 (SNAPSHOT) ☐ 2.7.8

**Project Metadata**  
Group:   
Artifact:   
Name:   
Description:   
Package name:   
Packaging: ☒ Jar ☐ War  
Java: ☐ 19 ☐ 17 ☒ 11 ☐ 8

**Dependencies**

ADD DEPENDENCIES... CTRL + B

**Spring Web** WEB  
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

**Spring Boot DevTools** DEVELOPER TOOLS  
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

**Spring Data JPA** SQL  
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

**MySQL Driver** SQL  
MySQL JDBC driver.

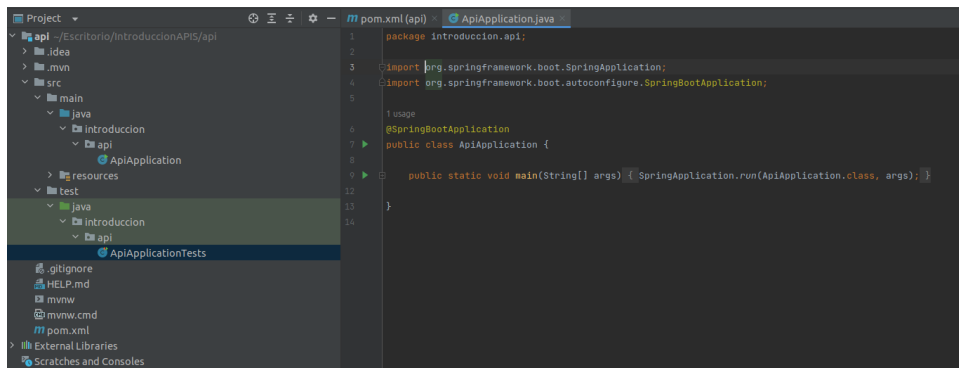
GENERATE CTRL + G

EXPLORE CTRL + SPACE

SHARE...

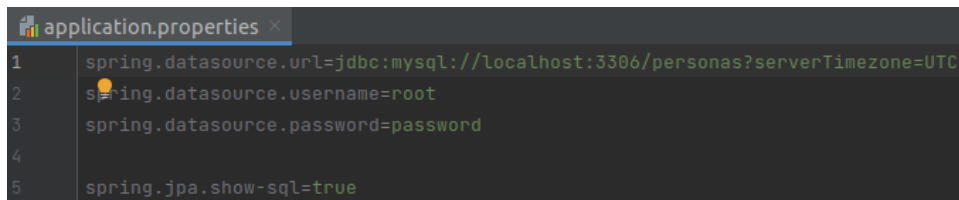
Damos clic en Generate para descargar el proyecto, y una vez descargado se descomprime.

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 resuelven las dependencias y se descargue bien todo y a su vez se coja el proyecto como un maven project.



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 así como lo tengo definido en MySQL.

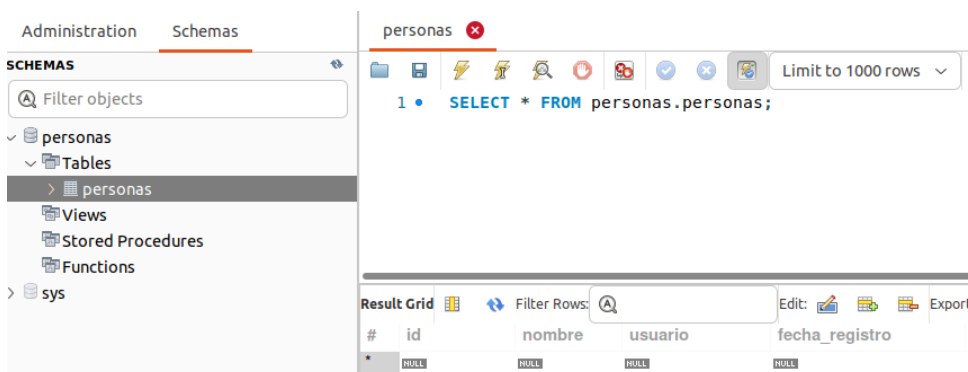
Una vez hecho esto vamos al MySQL y ejecutamos el comando

CREATE DATABASE personas;

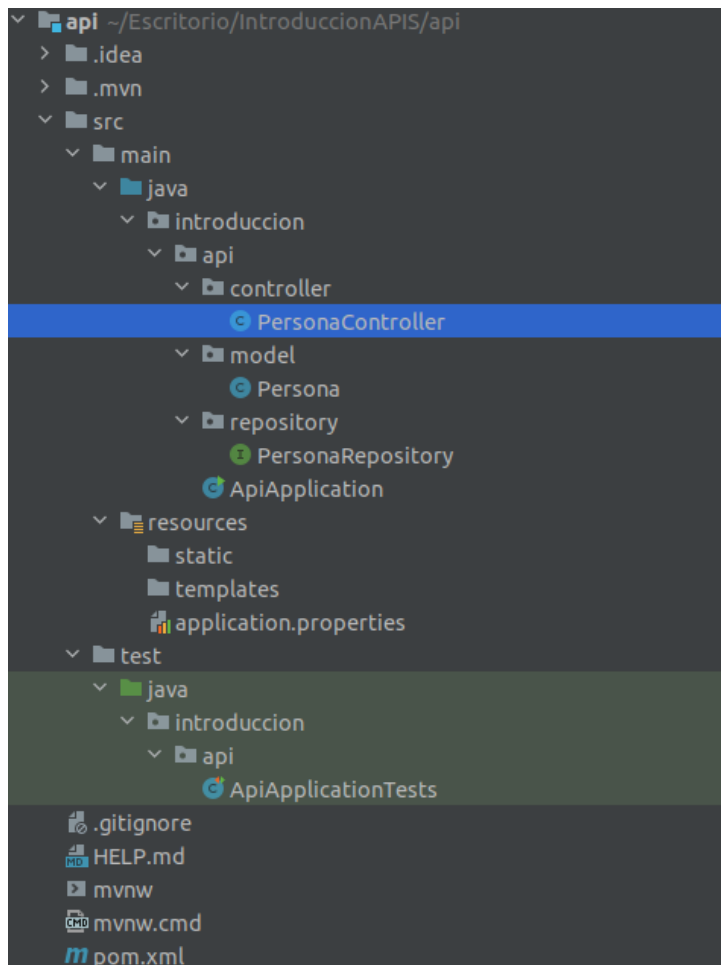
Tras esto creamos una tabla nueva que tendrá los siguientes campos:

personas - Table		Query 4									
Name:	personas	Schema: personas									
Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	G	D	
id	INT(11)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
nombre	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
usuario	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
fecha_registro	DATE	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

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**

The screenshot shows the code for the `ApiApplication.java` file. The code is as follows:

```
1 package introduccion.api;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.data.jpa.repository.config.EnableJpaAuditing;
6
7 @SpringBootApplication
8 @EnableJpaAuditing
9 public class ApiApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(ApiApplication.class, args);
13     }
14
15 }
```

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
1 package introduccion.api.controller;
2
3 import introduccion.api.model.Persona;
4 import introduccion.api.repository.PersonaRepository;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.web.bind.annotation.*;
7
8 import java.util.List;
9
10 @RestController
11 @RequestMapping("/api")
12 public class PersonaController {
13
14     10 usages
15     @Autowired
16     private PersonaRepository personaRepository;
17
18     @GetMapping("/personas")
19     public List<Persona> getPersonas() {
20         return personaRepository.findAll();
21     }
22
23     @GetMapping("/personas/nombre")
24     public List<Persona> getPersonasByNombre(@PathVariable("nombre") String nombre) {
25         return personaRepository.findByNombre(nombre);
26     }
27
28     @GetMapping("/personas/usuario")
29     public Persona getPersonaByUsuario(@PathVariable("usuario") String usuario) {
30         return personaRepository.findByUsuario(usuario);
31     }
32
33     @GetMapping("/personas/usuario/existe")
34     public boolean existeUsuario(@PathVariable("usuario") String usuario) {
35         return personaRepository.existsByUsuario(usuario);
36     }
37 }
```

```

24         return personaRepository.findByNombre(nombre);
25     }
26
27     @GetMapping("/personas/usuario")
28     public Persona getPersonaByUsuario(@PathVariable("usuario") String usuario) {
29         return personaRepository.findByUsuario(usuario);
30     }
31
32     @GetMapping("/personas/usuario/existe")
33     public boolean existeUsuario(@PathVariable("usuario") String usuario) {
34         return personaRepository.existsByUsuario(usuario);
35     }
36
37     @PostMapping("/persona")
38     public Persona crearPersona(@RequestBody Persona persona) {
39         return personaRepository.save(persona);
40     }
41
42     @PutMapping("/persona/{id}")
43     public Persona actualizarPersona(@PathVariable int id, @RequestBody Persona persona) {
44         return personaRepository.save(persona);
45     }
46
47     @PutMapping("/persona/usuario/{usuario}")
48     public Persona actualizarPersonaByUsuario(@PathVariable String usuario, @RequestBody Persona persona) {
49         Persona personaActual = personaRepository.findByUsuario(usuario);
50         personaActual.setNombre(persona.getNombre());
51         personaActual.setUsuario(persona.getUsuario());
52         return personaRepository.save(personaActual);
53     }
54
55     @DeleteMapping("/persona/{id}")
56     public void eliminarPersona(@PathVariable("id") Long id) { personaRepository.deleteById(id); }
57
58     @DeleteMapping("/persona/usuario/{usuario}")
59     public void eliminarPersonaByUsuario(@PathVariable String usuario) { personaRepository.deleteByUsuario(usuario); }
60
61     }
62

```

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 x
1 package introduccion.api.model;
2
3 import javax.persistence.*;
4 import org.springframework.data.annotation.CreatedDate;
5 import org.springframework.data.jpa.domain.support.AuditingEntityListener;
6
7 import java.util.Date;
8
9 @Entity
10 @Table(name = "personas")
11 @EntityListeners(AuditingEntityListener.class)
12 public class Persona {
13     @Id
14     @GeneratedValue(strategy = GenerationType.IDENTITY)
15     private Long id;
16
17     @Column(name = "nombre")
18     private String nombre;
19
20     @Column(name = "usuario")
21     private String usuario;
22
23     @Column(name = "fecha_registro")
24     @CreatedDate
25     private Date fechaRegistro;
26
27     public Persona() {
28     }
```

```
Persona.java x
24
25     public Persona() {
26     }
27
28     public Persona(Long id, String nombre, String usuario, Date fechaRegistro) {
29         this.id = id;
30         this.nombre = nombre;
31         this.usuario = usuario;
32         this.fechaRegistro = fechaRegistro;
33     }
```

Y a su vez los setters y getters además del toString

```
@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**

```

1 package introduccion.api.repository;
2
3 import introduccion.api.model.Persona;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.data.repository.query.Param;
6 import org.springframework.stereotype.Repository;
7
8 import java.util.List;
9
10 @Repository
11 public interface PersonaRepository extends JpaRepository<Persona, Long> {
12
13     List<Persona> findByNombre(@Param("nombre") String nombre);
14     Persona findByUsuario(@Param("usuario") String usuario);
15     boolean existsByUsuario(@Param("usuario") String usuario);
16     void deleteByUsuario(@Param("usuario") String usuario);
17 }

```

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, Generate Sources and Update Folders y a Reload project, tras esto se da run al ApiApplication.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4     <modelVersion>4.0.0</modelVersion>
5     <parent>
6         <groupId>org.springframework.boot</groupId>
7         <artifactId>spring-boot-starter-parent</artifactId>
8         <version>2.7.0</version>
9         <relativePath><!-- lookup parent from repository -->
10    </parent>
11    <groupId>introduccion</groupId>
12    <artifactId>api</artifactId>
13    <version>0.0.1-SNAPSHOT</version>
14    <name>api</name>
15    <description>Introduccion al desarrollo de APIs en Spring Boot</description>
16    <properties>
17        <java.version>11</java.version>
18    </properties>
19    <dependencies>
20        <dependency>
21            <groupId>org.springframework.boot</groupId>
22            <artifactId>spring-boot-starter-data-jpa</artifactId>
23        </dependency>
24        <dependency>
25            <groupId>org.springframework.boot</groupId>
26            <artifactId>spring-boot-starter-web</artifactId>
27        </dependency>
28
29        <dependency>
30            <groupId>org.springframework.boot</groupId>
31            <artifactId>spring-boot-devtools</artifactId>
32            <scope>runtime</scope>
33            <optional>true</optional>
34        </dependency>
35        <dependency>
36            <groupId>mysql</groupId>
37            <artifactId>mysql-connector-java</artifactId>
38            <scope>runtime</scope>
39        </dependency>

```

```

19 <dependencies>
20   <dependency>
21     <groupId>org.springframework.boot</groupId>
22     <artifactId>spring-boot-starter-data-jpa</artifactId>
23   </dependency>
24   <dependency>
25     <groupId>org.springframework.boot</groupId>
26     <artifactId>spring-boot-starter-web</artifactId>
27   </dependency>
28
29   <dependency>
30     <groupId>org.springframework.boot</groupId>
31     <artifactId>spring-boot-devtools</artifactId>
32     <scope>runtime</scope>
33     <optional>true</optional>
34   </dependency>
35   <dependency>
36     <groupId>mysql</groupId>
37     <artifactId>mysql-connector-java</artifactId>
38     <scope>runtime</scope>
39   </dependency>
40   <dependency>
41     <groupId>org.springframework.boot</groupId>
42     <artifactId>spring-boot-starter-test</artifactId>
43     <scope>test</scope>
44   </dependency>
45 </dependencies>
46
47 <build>
48   <plugins>
49     <plugin>
50       <groupId>org.springframework.boot</groupId>
51       <artifactId>spring-boot-maven-plugin</artifactId>
52     </plugin>
53   </plugins>
54 </build>
55
56 </project>

```

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

```

Run:  ApiApplication
2023-02-06 11:53:53.791 INFO 5616 --- [restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 966 ms
2023-02-06 11:54:53.794 INFO 5616 --- [restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2023-02-06 11:54:53.986 INFO 5616 --- [restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2023-02-06 11:54:54.012 INFO 5616 --- [restartedMain] org.hibernate.jpa.internal.util.LogHelper : HHH000120: Processing PersistenceUnitInfo [name: default]
2023-02-06 11:54:54.836 INFO 5616 --- [restartedMain] org.hibernate.Version : HHH000412: Hibernate ORM core version 5.6.9.Final
2023-02-06 11:54:54.124 INFO 5616 --- [restartedMain] org.hibernate.annotations.common.Version : HCA000000: Hibernate Commons Annotations (5.1.2.Final)
2023-02-06 11:54:54.193 INFO 5616 --- [restartedMain] org.hibernate.dialect.Dialect : HHH000400: using dialect: org.hibernate.dialect.MySQL8Dialect
2023-02-06 11:54:54.472 INFO 5616 --- [restartedMain] org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform : HHH000046: using JtaPlatform implementation: org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform
2023-02-06 11:54:54.477 INFO 5616 --- [restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2023-02-06 11:54:54.748 WARN 5616 --- [restartedMain] JpaWebConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering.
2023-02-06 11:54:54.839 INFO 5616 --- [restartedMain] o.s.b.a.o.OptionalLiveReloadServer : LiveReload server is running on port 35729
2023-02-06 11:54:54.935 INFO 5616 --- [restartedMain] o.s.s.s.AnnotationMethodIntrospector : Tomcat started on port(s): 8080 (http) with context path ''
2023-02-06 11:54:54.942 INFO 5616 --- [restartedMain] Introduccion.Api.ApiApplication : Started ApiApplication in 2.586 seconds (JVM running for 2.897)

```

## Pruebas con Postman

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

- Post

POST Crear Persona

+

...

Introduccion APIs / Crear Persona

POST

▼

http://localhost:8080/api/persona

Params

Authorization

Headers (9)

Body ●

Pre-request Script

☐ none

☐ form-data

☐ x-www-form-urlencoded

☒ raw

☐ binary

1

}

2

... "nombre": "Nuevo",

3

... "usuario": "nuevo1",

4

... "fechaRegistro": "2023-06-06"

5

}

Body

Cookies

Headers (5)

Test Results

Pretty

Raw

Preview

Visualize

JSON ▼

≡

1

}

2

"id": 1,

3

"nombre": "Nuevo",

4

"usuario": "nuevo1",

5

"fechaRegistro": "2023-02-06T14:18:57.753+00:00"

6

}

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.

GET GetAllPersonas

+

...

Introduccion APIs / GetAllPersonas

GET

▼

http://localhost:8080/api/personas

ParamsAuthorizationHeaders (7)BodyPre-request ScriptTestsSettings

● none

● form-data

● x-www-form-urlencoded

● raw

● binary

● GraphQL

JSON ▼

1

BodyCookiesHeaders (5)Test Results

Pretty

Raw

Preview

Visualize

JSON ▼

≡

🔍

1

[

2

{

3

"id": 1,

4

"nombre": "Nuevo",

5

"usuario": "nuevo1",

6

"fechaRegistro": "2023-02-05T23:00:00.000+00:00"

7

},

8

{

9

"id": 2,

10

"nombre": "Nuevo",

11

"usuario": "nuevo2",

12

"fechaRegistro": "2023-02-06T23:00:00.000+00:00"

13

}

14

]

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

GET GetPersonas By Nombre + ...

Introduccion APIs / GetPersonas By Nombre

GET http://localhost:8080/api/personas/nombre/Nuevo

Params Authorization Headers (7) Body Pre-request Script Test Results

Query Params

KEY	VAL
Key	Val

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "id": 1,
4     "nombre": "Nuevo",
5     "usuario": "nuevo1",
6     "fechaRegistro": "2023-02-05T23:00:00.000+00:00"
7   },
8   {
9     "id": 2,
10    "nombre": "Nuevo",
11    "usuario": "nuevo2",
12    "fechaRegistro": "2023-02-06T23:00:00.000+00:00"
13  }
14 ]
```

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

– Put

PUT ActualizarPersona By ID + ...

Introduccion APIs / ActualizarPersona By ID

PUT http://localhost:8080/api/persona/1

Params Authorization Headers (9) **Body** Pre-request Scrip

● none ● form-data ● x-www-form-urlencoded ● **raw** ● binar

```
1 {
2   "id": 1,
3   "nombre": "Nuevo actualizado",
4   "usuario": "usuarioActualizado1",
5   "fechaRegistro": "2022-06-06"
6 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "nombre": "Nuevo actualizado",
4   "usuario": "usuarioActualizado1",
5   "fechaRegistro": "2022-06-06T00:00:00.000+00:00"
6 }
```

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

- Delete

DEL EliminarPersona By ID

+ ...

Introduccion APIs / EliminarPersona By ID

DELETE

▼

http://localhost:8080/api/persona/1

ParamsAuthorizationHeaders (7)BodyPre-request Sc

Query Params

	KEY
	Key

BodyCookiesHeaders (4)Test Results

PrettyRawPreviewVisualizeText ▼

1

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:

DEL EliminarPersona By ID

GET GetAllPersonas

+ ...

Introduccion APIs / GetAllPersonas

GET

▼

http://localhost:8080/api/personas

ParamsAuthorizationHeaders (7)BodyPre-request ScriptTestsSettings

● none

● form-data

● x-www-form-urlencoded

● raw

● binary

● GraphQL

JSON ▼

1

BodyCookiesHeaders (5)Test Results

PrettyRawPreviewVisualizeJSON ▼

```
1  [
2    {
3      "id": 2,
4      "nombre": "Nuevo actualizado2",
5      "usuario": "usuarioActualizado2",
6      "fechaRegistro": "2023-02-05T23:00:00.000+00:00"
7    }
8  ]
```



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:

▼ Introduccion APIs
POST Crear Persona
GET GetAllPersonas
GET GetPersonas By Nombre
GET GetPersona By Usuario
GET ExisteUsuario By Usuario
PUT ActualizarPersona By ID
PUT ActualizarPersona By Usuario
DEL EliminarPersona By ID
DEL EliminarPersona By Usuario

## 4. Swagger de la API

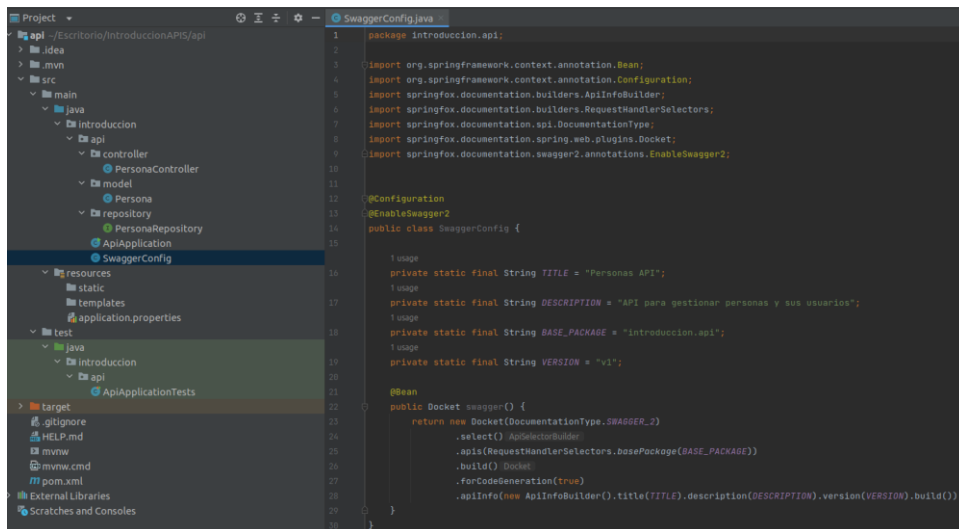
Como se ha dicho antes es necesario saber el contrato de una API fácilmente por lo que se utilizan diferentes herramientas.

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



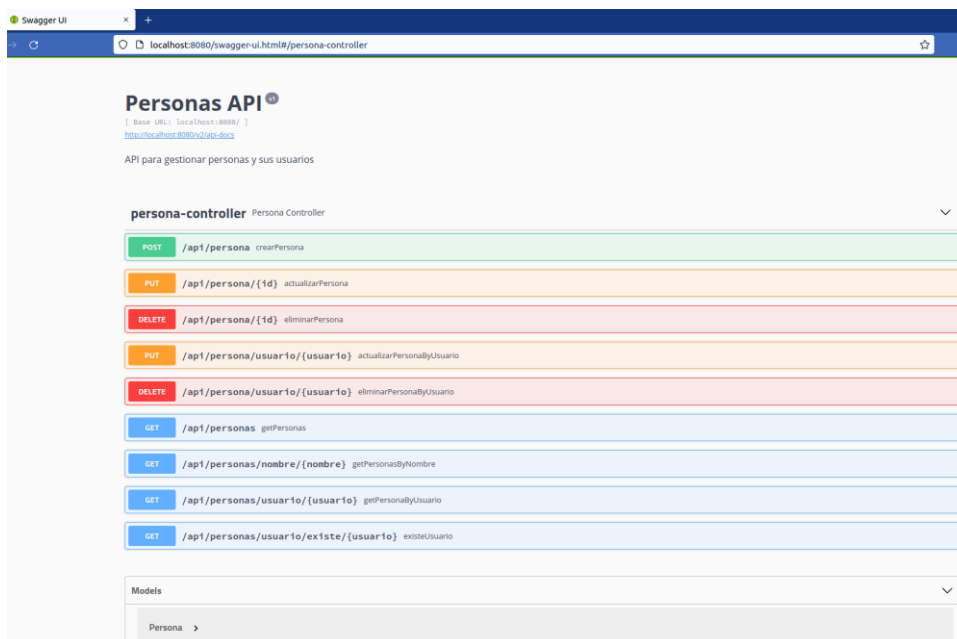
Esta clase tendrá este código y estará en la carpeta de api.

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/#/](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.

POST

/api/persona crearPersona

Try it out

Parameters

Name	Description
persona <span>required</span> (body)	persona <div>Example Value   Model</div> <pre>{  "fechaRegistro": "2023-02-07T08:11:06.945Z",  "id": 0,  "nombre": "string",  "usuario": "string"}  </pre> <div>Parameter content type application/json</div>

Responses

Response content type "/"

Code	Description
200	<div>OK</div> <div>Example Value   Model</div> <pre>{  "fechaRegistro": "2023-02-07T08:11:06.957Z",  "id": 0,  "nombre": "string",  "usuario": "string"}  </pre>
201	<div>Created</div>
401	<div>Unauthorized</div>

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.