**EJERCICIO 1**:

**Propuesta**:

**1. – Descarga un proyecto utilizando Spring Initizalizr añadiendo los siguientes módulos:**

* Spring Web
* Spring Data JPA
* H2 Database
* Spring Boot Actuator
* Spring Boot DevTools

**2. – Arranca la aplicación y comprueba los endpoints del actuator.**

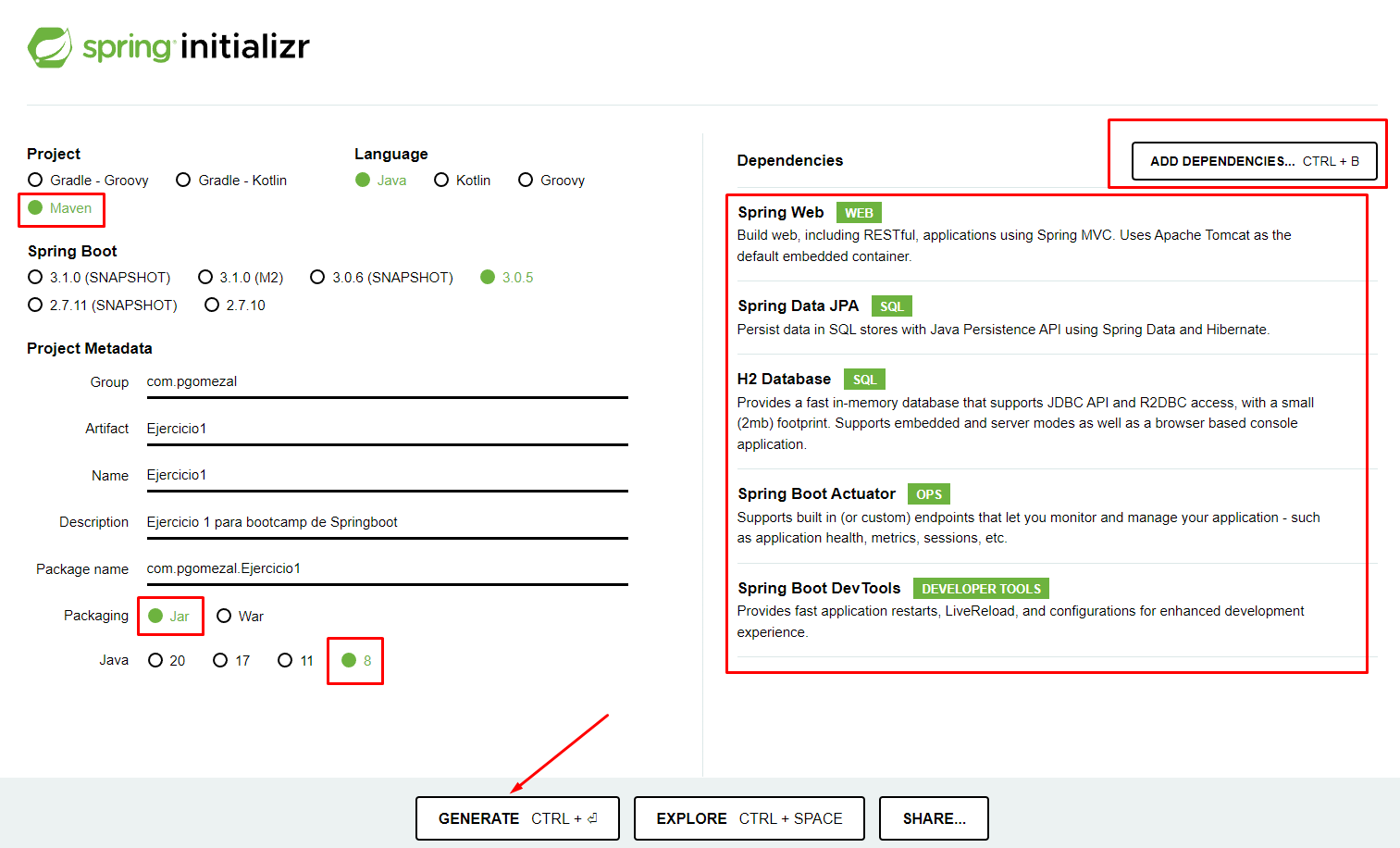
**3. – Crea un contenedor llamado MessageRestController con la etiqueta @RestController que devuelva el mensaje “Hello World!” al realizar una petición GET a la url:**

* <http://localhost:8080/api/v1/message>

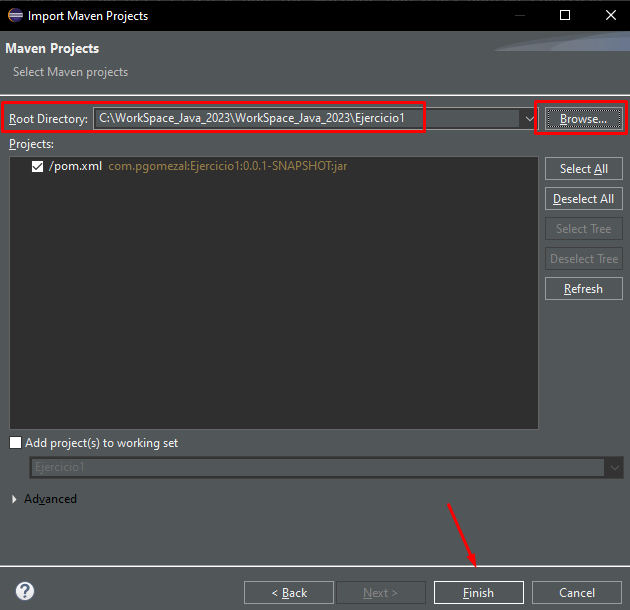
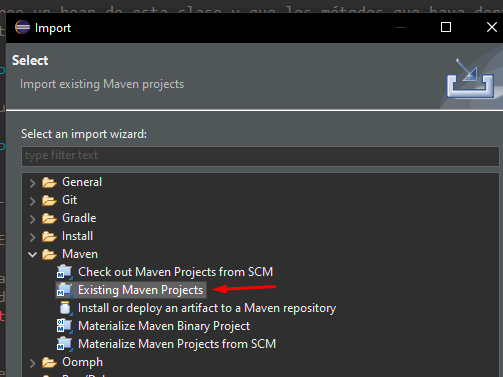
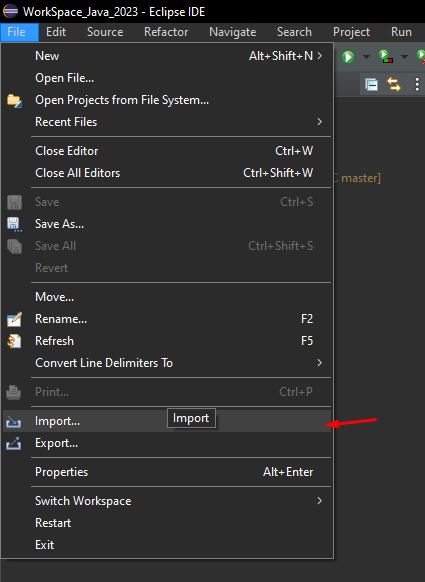
**Solución:**

**Paso 1**:

Accedemos a la url <https://start.spring.io/> , lo configuramos, añadimos los módulos indicados y le damos a “GENERATE”:



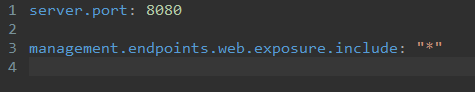
Nos descargará un zip con el proyecto. Lo descomprimimos en nuestro workspace y lo importamos este proyecto en Eclipse como un Maven Project:



**Paso 2**

Antes de nada, vamos a cambiar el application.properties por application.yml. Simplemente lo refactorizamos y listo.

Una vez hecho eso, lo editamos y añadimos lo siguiente:

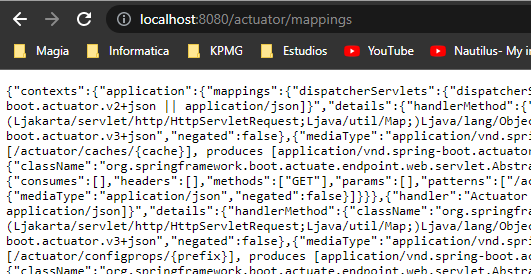
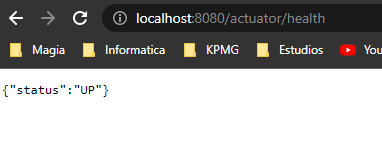


Con la primera linea estamos cambiando el puerto por defecto (en realidad el de por defecto es el 8080 pero podríamos cambiarlo asi si lo deseamos).

Con la segunda línea, le estamos diciendo que nos incluya todos los endpoints del modulo actuator.

Ahora arrancamos la aplicación y nos vamos al navegador web y escribimos la url correspondiente para ver los endpoint del actuator.

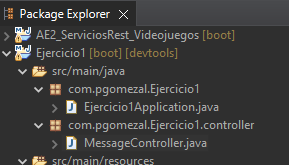
[http://localhost:8080/actuator/{nombre\_endpoint}](http://localhost:8080/actuator/%7bnombre_endpoint%7d):



Podríamos visualizarlo mejor en Postman, haciendo una petición GET.

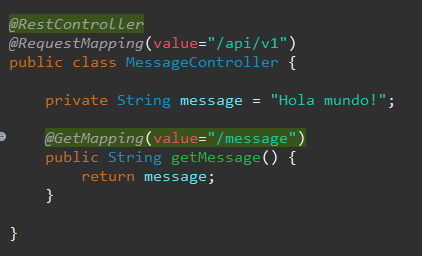
**Paso 3**

Creamos un paquete controlador y creamos la clase **MessageController**.



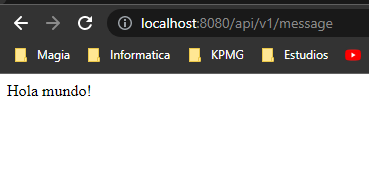
Anotaremos esta clase como **@RestController**. Recordemos que donde esté esta notación, es donde se encontrarán configurados nuestros endpoints.

Dejaremos la clase de esta manera:



Con **@RequestMapping** estamos indicándole en el **value** cuál es la raíz de nuestros endpoints de esta clase.

Ahora tan solo tenemos que crear un método que devuelva el mensaje especificado en el ejercicio y anotar dicha clase con **@GetMapping** para que esa salida la devuelva como String a cualquier petición GET que se haga a la ruta especificada en el value. De esta forma, si escribimos en un navegador <http://localhost:8080/api/v1/message> nos devolverá dicho mensaje:



**Ejercicio 2**

**Propuesta**:

1. Cambia el servidor embebido de la aplicación generada a undertow

2. Definir el mensaje de salida del endpoint creado anteriormente en el application.yml

3. Cambiar el puerto del servidor al 8090

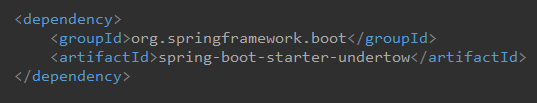
4. Comprobar que la URL del endpoint ahora es la siguiente:

<http://localhost:8090/api/v1/message>

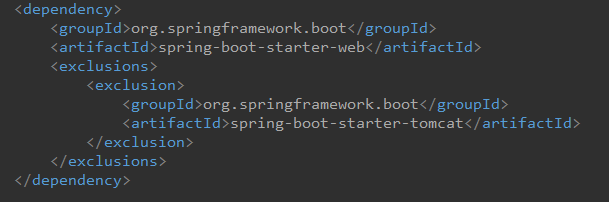
**Solución**:

**Paso 1**

Para cambiar el servidor embebido por undertow, añadimos al pom.xml la siguiente dependencia:

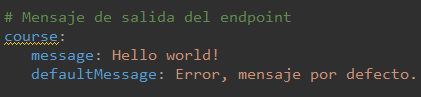


Por supuesto, debemos desactivar el que viene por defecto, el **tomcat**, modificando la dependencia de Spring Web añadiendo una exclusión.



**Paso 2**

Para definir el mensaje de salida del endpoint creado anteriormente en el application.yml tan sólo tenemos que añadir en dicho archivo lo siguiente:



Y ahora, en nuestra clase controladora, añadimos la siguiente anotación antes del String que almacena el mensaje para que coja esos valores del yml (y obviamente no lo inicializamos):



**Paso 3:**

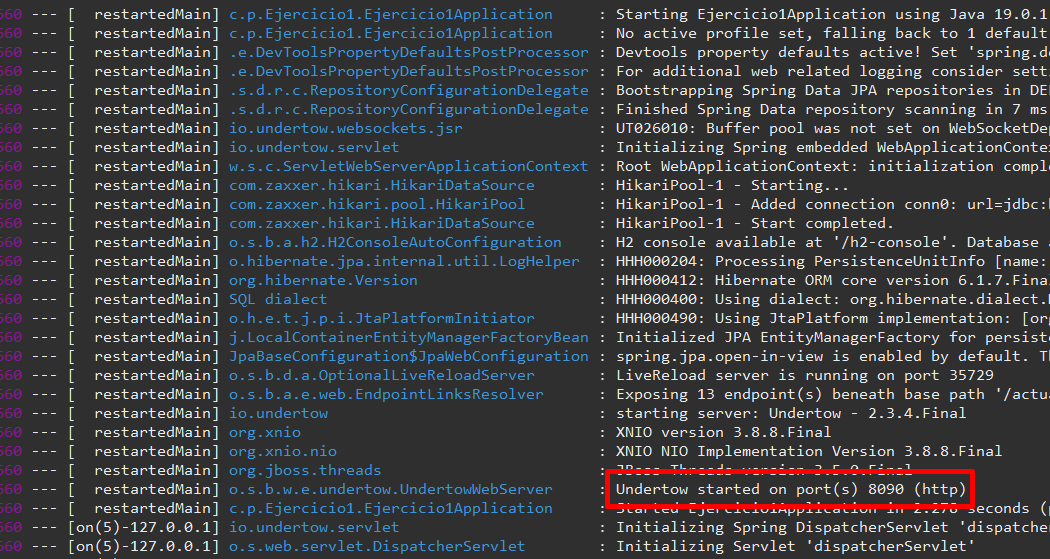
Para cambiar el puerto al 8090 es muy sencillo y lo vimos en el primer ejercicio:

Tan solo tenemos que añadir/cambiar este valor en el application.yml:



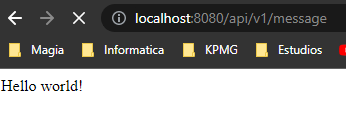
**Paso 4**:

Si arrancamos el servidor veremos que los cambios aplicados anteriormente se han realizado correctamente:



Como vemos, se ha iniciado el server undertow en el puerto 8090

Y si hacemos GET a <http://localhost:8090/api/v1/message> veremos que lo hace correctamente.



**Ejercicio 3**

**Propuesta**:

1. Crear ficheros de configuración adicionales para los profiles “dev” y “prod”.

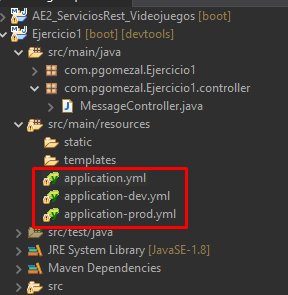
2. Configurar diferentes mensajes para cada uno de los perfiles.

3. Probar la aplicación con y sin profiles.

**Solución**:

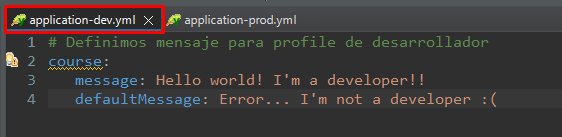
**Paso 1**

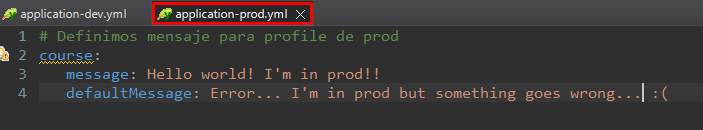
Tan solo creamos en la carpeta /src/main/resources dos archivos: application-dev.yml y application-prod.yml:



**Paso 2**

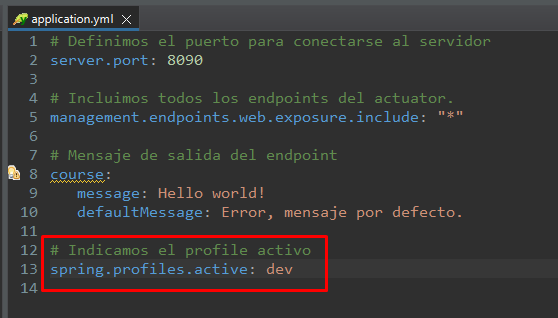
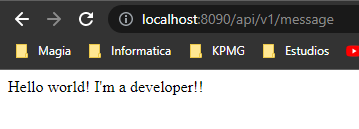
Al igual que lo teníamos en el application.yml configuramos mensajes para cada uno de los profiles:

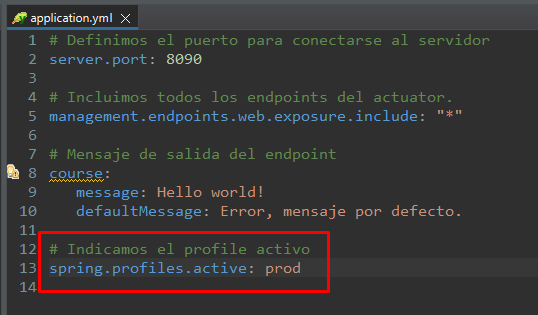
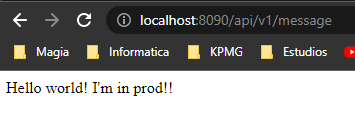


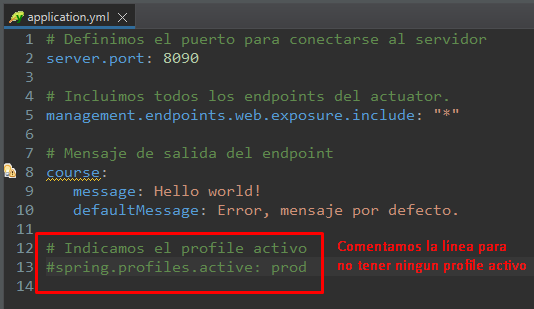
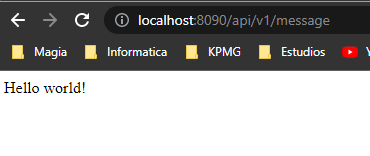


**Paso 3**

Ahora indicamos el perfil activo en el application.yml

**Ejercicio 4**

**Propuesta**:

1. Creación de una interfaz MessageService que tenga el método String getMessage().

2. Crear la implementación del MessageService que recoja el mensaje de los ficheros YAML.

3. Inyectar la dependencia en el controlador creado anteriormente

**Solución**:

Algo importante de los microservicios es que están divididos en 3 capas:

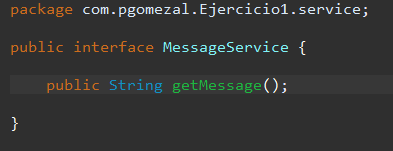
Capa de request/controlador: Trata todos los endpoints

Capa de servicio: Lleva toda la lógica

Capa de acceso de la base de datos/repository.

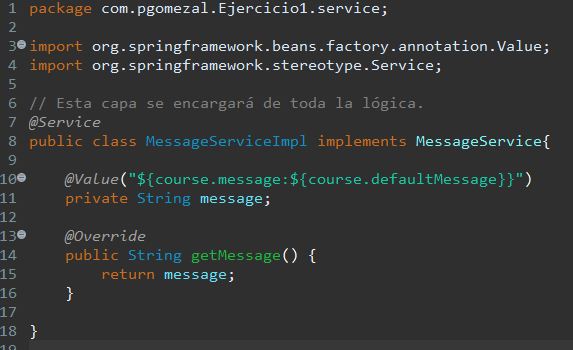
**Paso 1**

Creamos el paquete de servicio y en él creamos la interfaz.

**Paso 2**

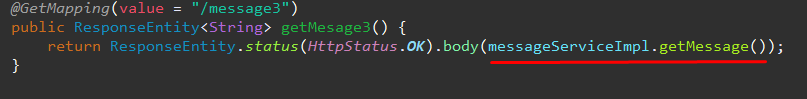
Creamos la clase que implemente la anterior interfaz y sobrescribimos el método creando la funcionalidad que antes la hacía la capa de controlador. Además, la anotamos con **@Service**:



**Paso 3**

Inyectamos con **@Autowired** el servicio **MessageServiceImpl** en el controlador **MessageController** para, posteriormente, poder utilizar su método en el endpoint:





**Ejercicio 5**

**Propuesta**:

1. Creación de una entidad User con los campos Id (numérico), name y birthdate (LocalDate).

2. Crear el repositorio de la entidad.

3. Crear un servicio que contenga las operaciones básicas CRUD y accedan al servicio siguiendo los estándares de definición de URLs para servicios REST.

Para configurar la base de datos H2:

<https://www.baeldung.com/spring-boot-h2-database>

**Solución**:

La configuración de la BBDD la pongo al final, después del paso 3

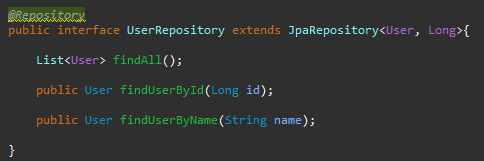
**Paso 1:**

Creamos la entidad User. Yo la he añadido a un paquete llamado entities:



**Paso 2**

Creo el repositorio de la entidad:



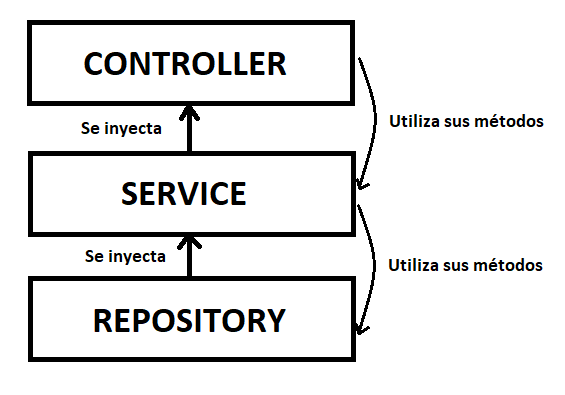
Recordemos que tiene que ser una interfaz que extienda de JpaRepository, y se le tiene que pasar el objeto y el tipo de dato de su Id, en este caso, Long.

Además, hemos añadido algunos de los métodos que necesitamos.

He tenido que añadir los datos a mano.

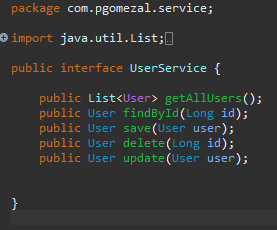
**Paso 3**

Antes de nada, recordar este esquema:



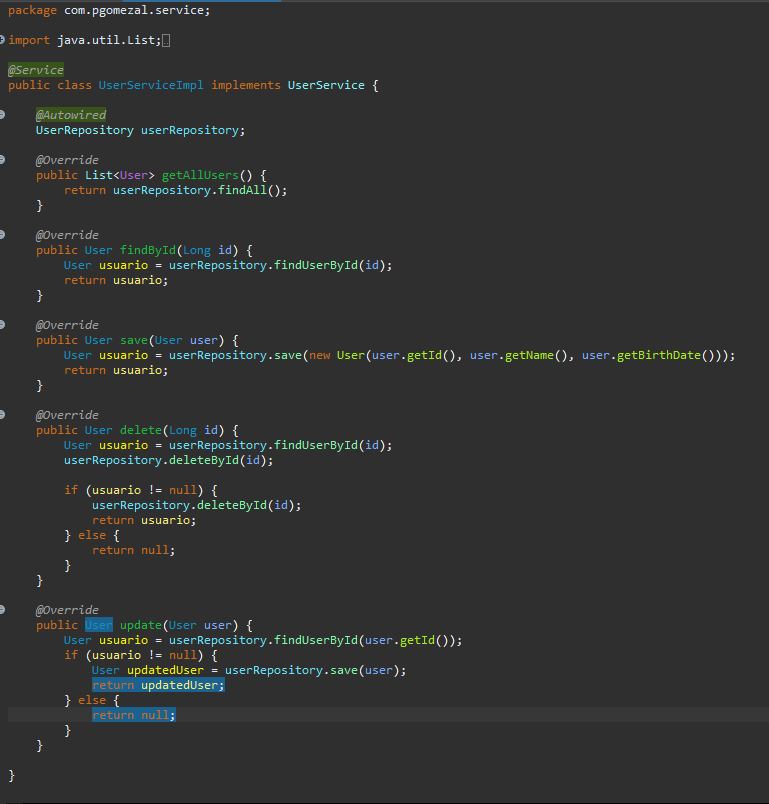
La clase controlador utilizará los métodos del Service mediante inyección con @Autowired, y esta hará lo mismo con el repositorio de la entidad User.

Vamos allá. En primer lugar creo la capa de servicio añadiendo primero una interfaz UserService que contenga todos los métodos necesarios para el CRUD:

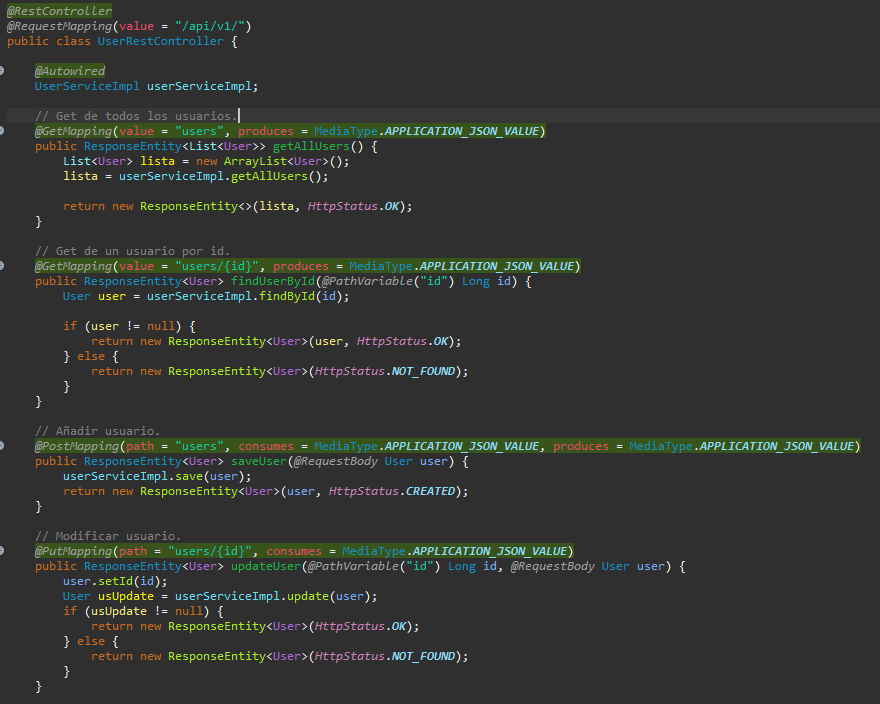


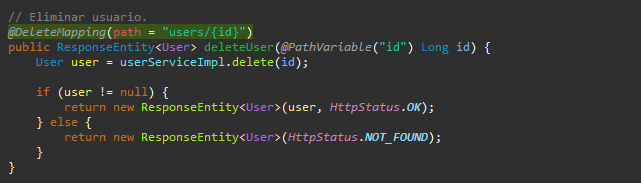
Posteriormente, creo una clase que implemente esta inferfaz llamada UserServiceImpl e implementamos sus métodos.

Como he indicado anteriormente, inyectará el UserRepository mediante la anotación @Autowired y posteriormente utilizará sus métodos dentro de cada uno de los métodos implementados de la interfaz.



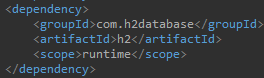
Por último, creamos una clase UserRestController y crearemos los endpoints para conectarse a la BBDD y realizar las peticiones deseadas. Para ello, inyectaremos el UserServiceImpl y utilizaremos los métodos creados anteriormente:



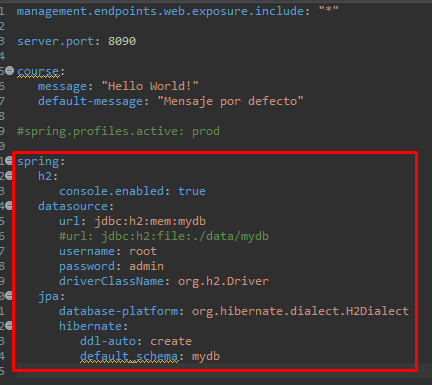


**Paso 4 – Configuración de la BBDD**

Nos aseguramos de tener la siguiente dependencia en el pom.xml



Añadimos las siguientes líneas al application.yml:



Con esto, levantará una BBDD en memoria (lo que significa que cuando se pare la aplicación perderemos los datos persistidos).

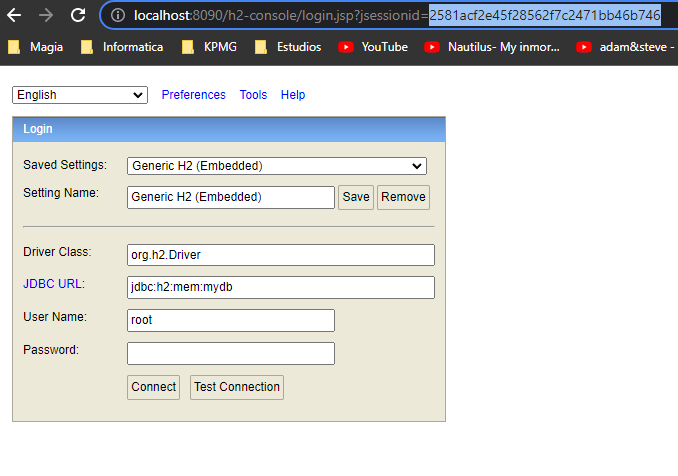
Cuando arranquemos la aplicación, levantará la BBDD y creará las tablas según las clases que tengamos anotadas con @Entity.

Posteriormente, para acceder a la consola, nos vamos al navegador y escribimos la url:

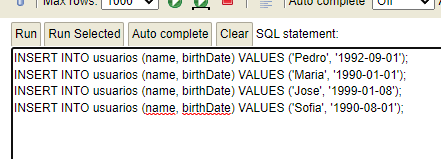
[http://localhost:{puerto\_configurado\_en\_el\_yaml}/h2-console/](http://localhost:%7bpuerto_configurado_en_el_yaml%7d/h2-console/)

En nuestro caso: <http://localhost:8090/h2-console/>

Nos saldrá una consola como esta donde tendremos que introducir los datos del application.yml

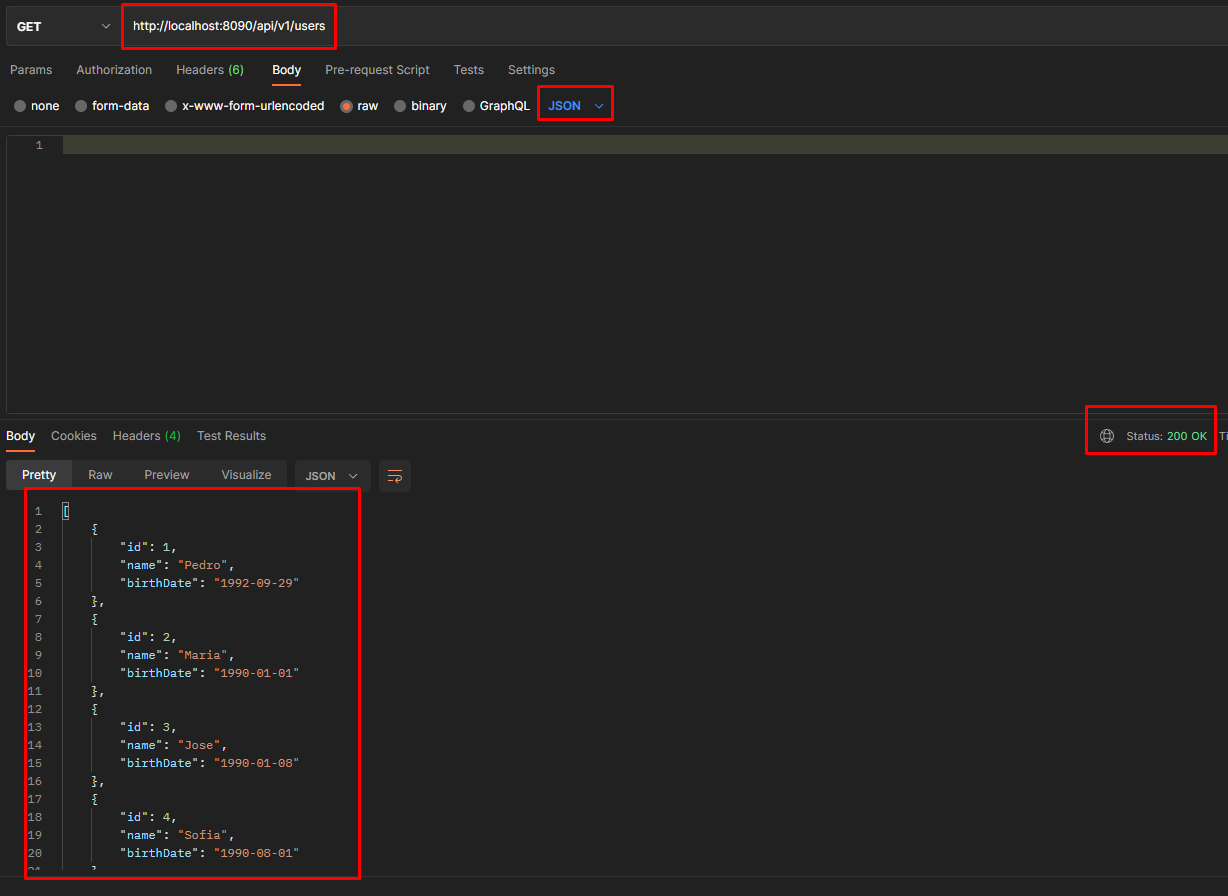


Una vez nos acceda, introducimos los datos deseados para hacer las pruebas.

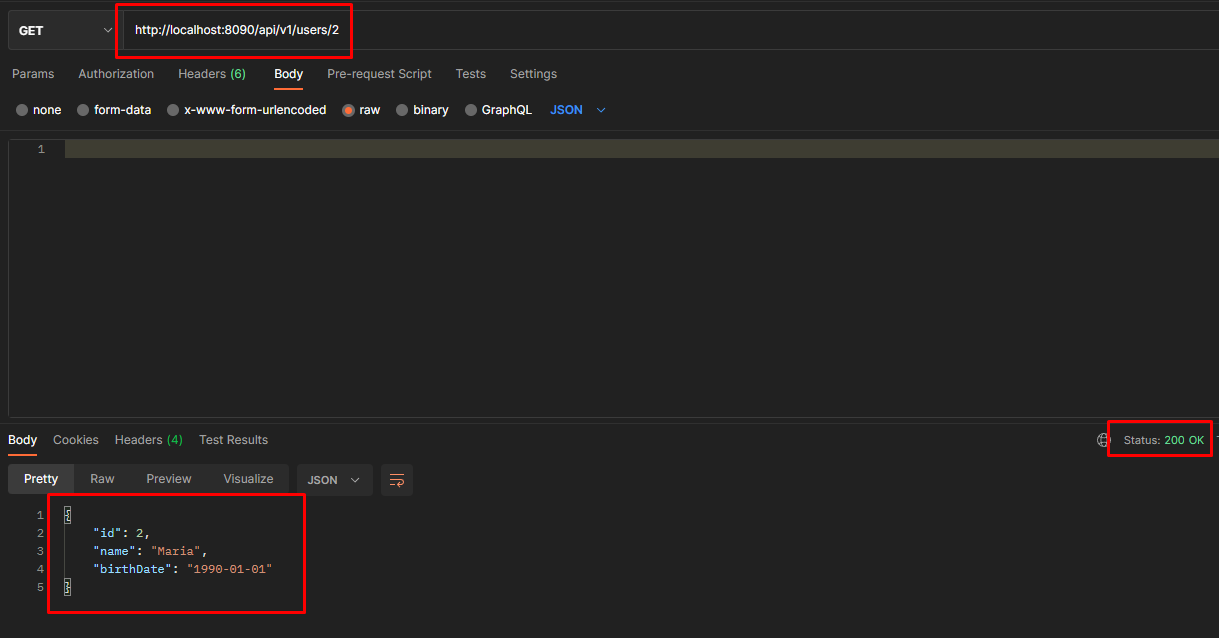


Después, con una aplicación como POSTMAN, realizaremos las diferentes pruebas para comprobar que todo funciona correctamente.

**GET DE TODOS LOS USUARIOS:**

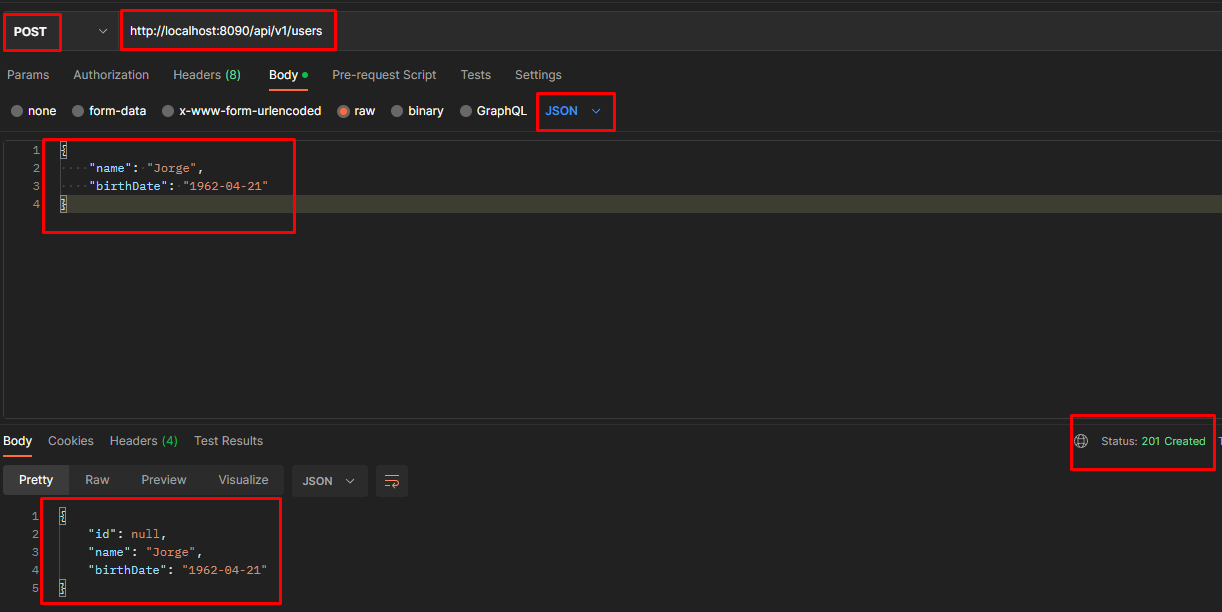


**GET DE USUARIO POR ID**

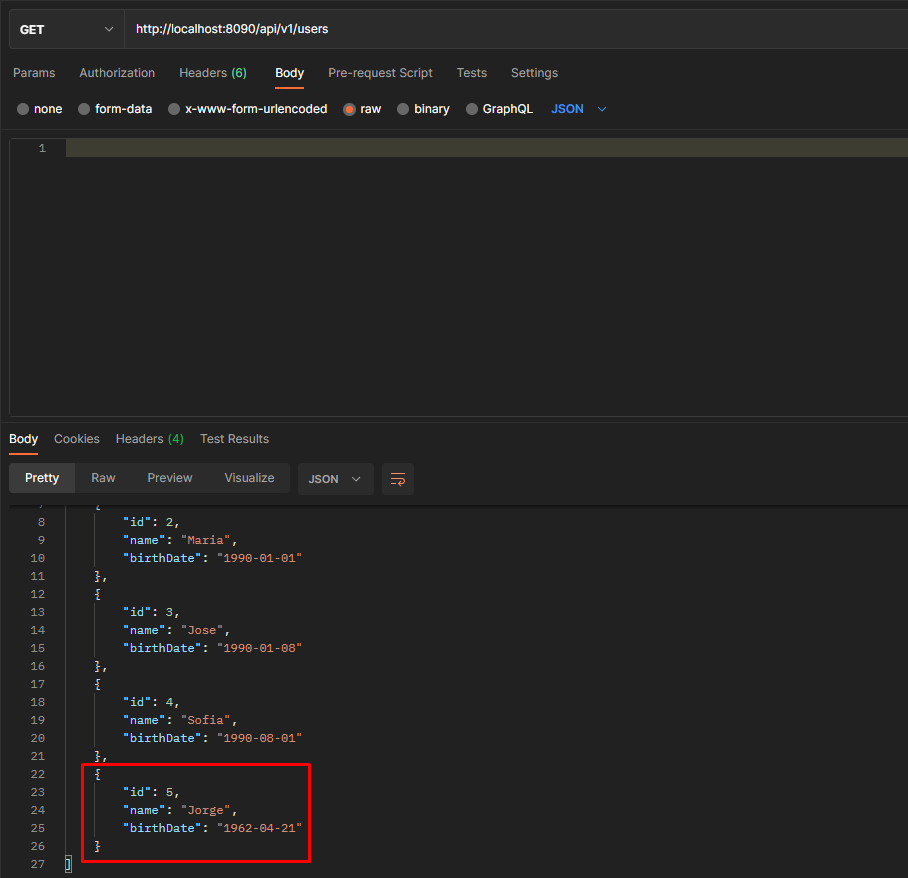


**ALTA DE USUARIO**

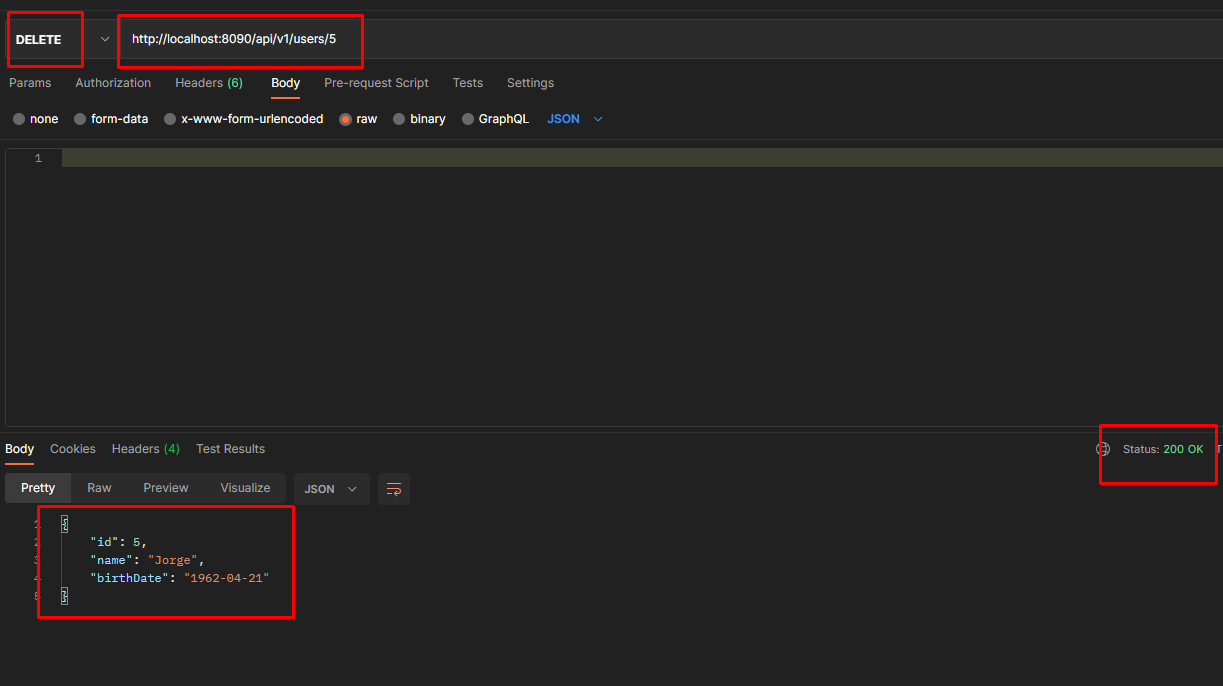
Recordemos que tal como lo tenemos programado, tenemos que pasarle un JSON (**SIN EL ID**) con los datos del usuario.



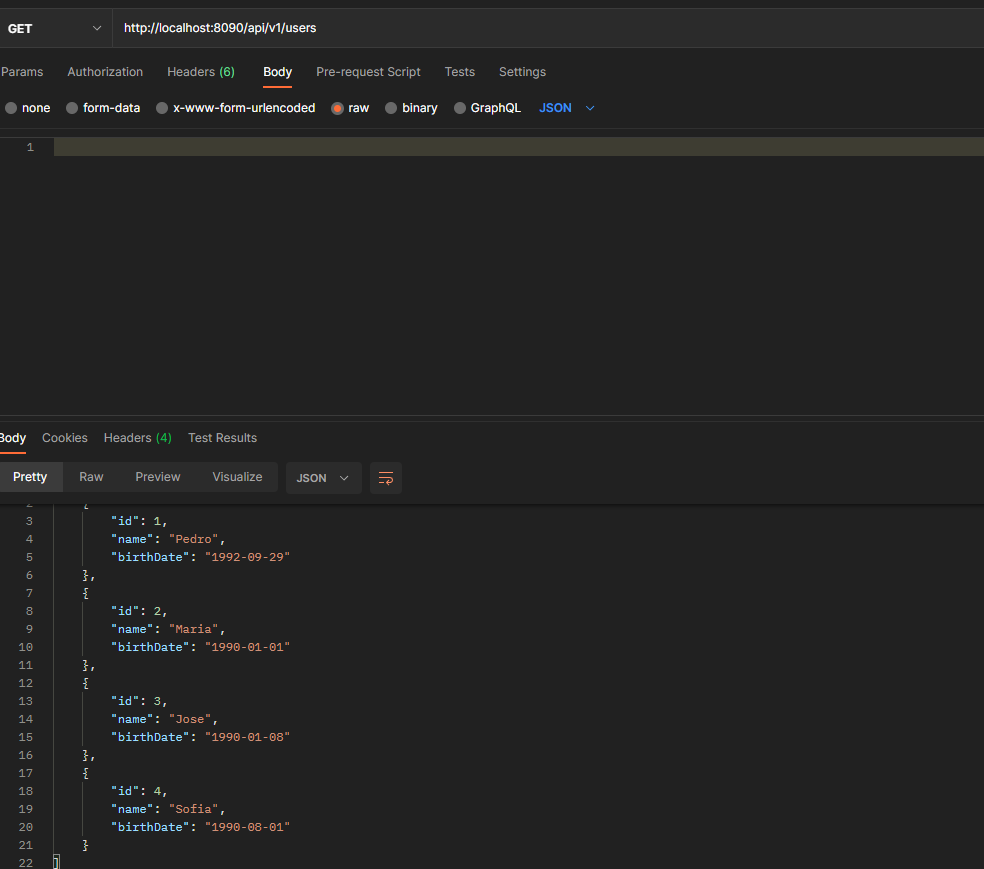
Si ahora hacemos un get de los usuarios veremos que está creado correctamente con el ID autoincrementado:



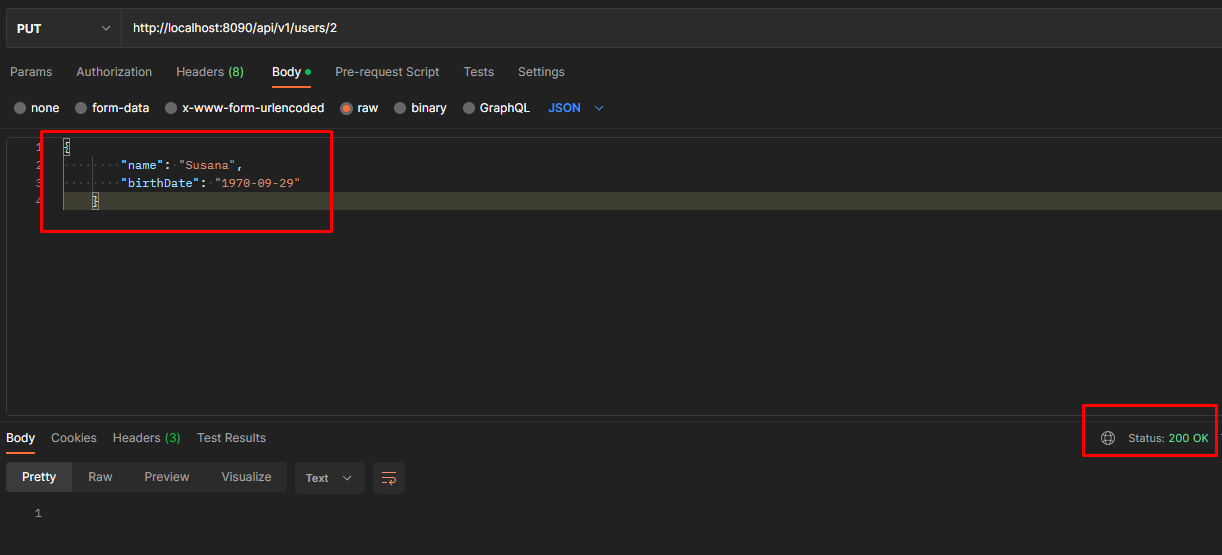
BAJA DE USUARIO

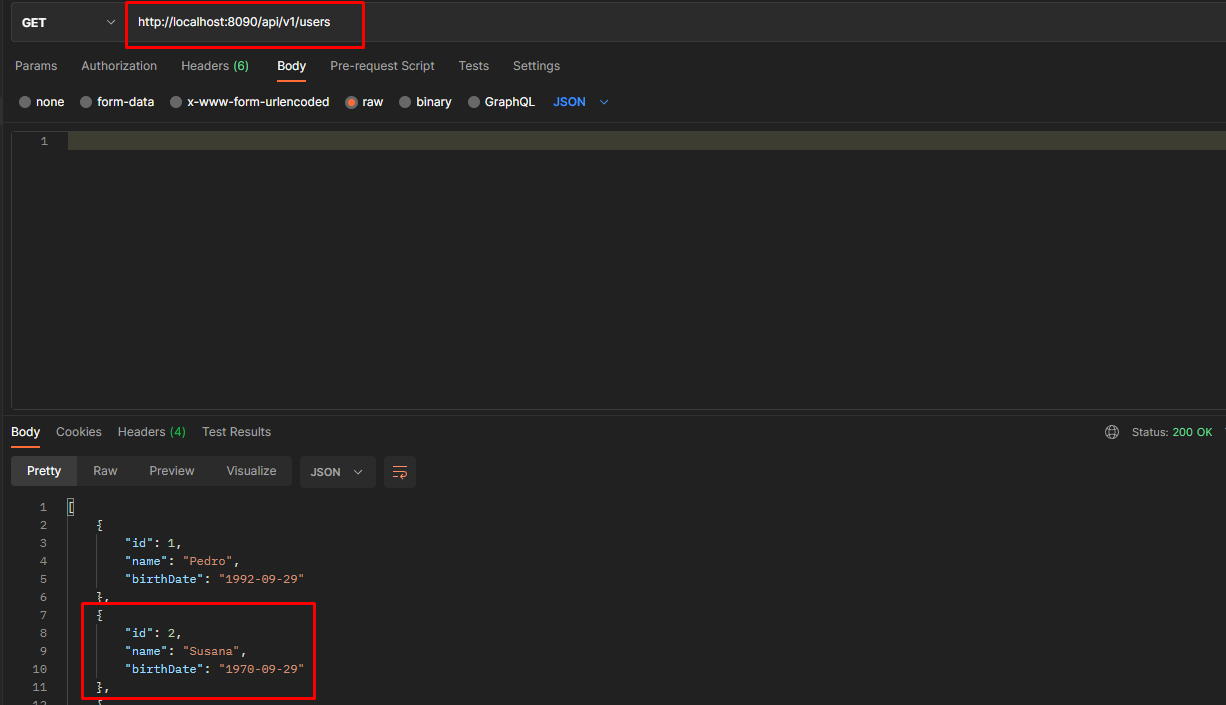


Como vemos, se ha eliminado correctamente:



**ACTUALIZACIÓN DE USUARIO:**

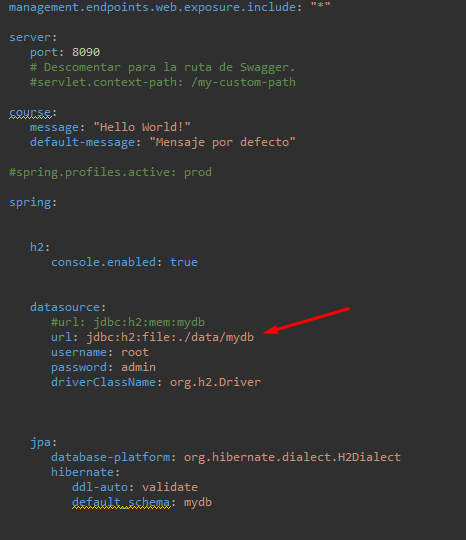




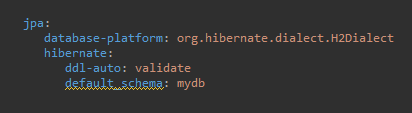
Como vemos todo funciona correctamente.

**PASOS ADICIONALES**

Para hacer que funcione la base de datos de forma permanente, tenemos que cambiar el archivo application.yml y modificar la url por un archivo local (.db). Si el archivo no existe, se creará en la ruta especificada.

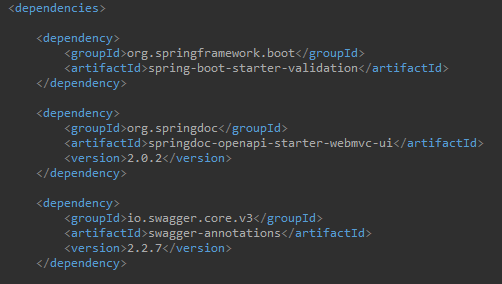


Ojo! Asegurarse también de que ddl-auto esté en “validate”:

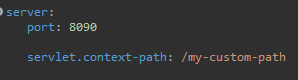


**Ejercicio extra – Configurar swagger**

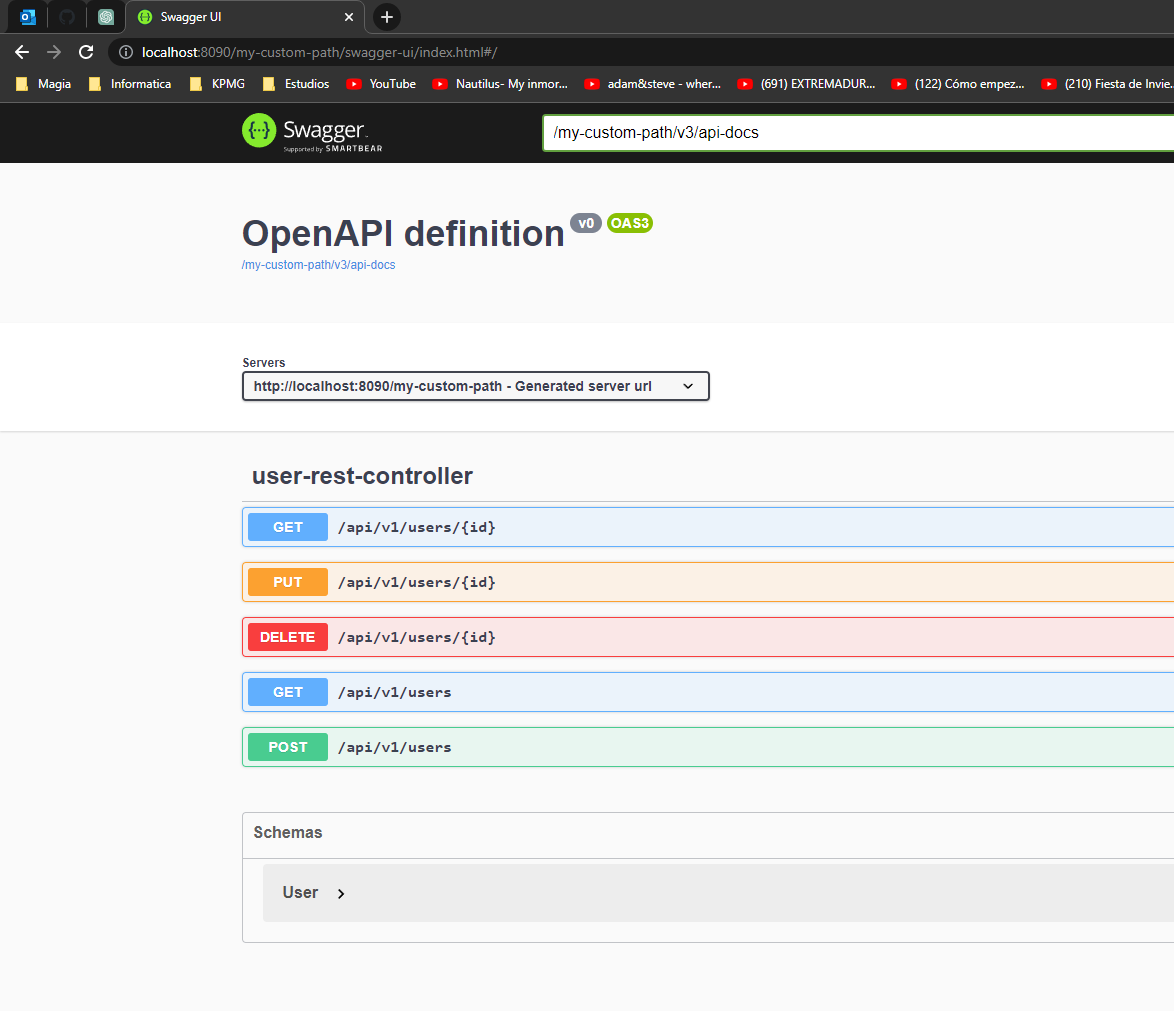
Para configurar swagger, lo único que tenemos que hacer es añadir las siguientes dependencias a nuestro pom.xml



Posteriormente, en el application.yml añadimos lo siguiente:



Con esto ya podríamos acceder al swagger en la ruta indicada en esa linea.



**OJO**: Me he dado cuenta de que si intentamos acceder a la consola de h2 con esa linea incorporada al application.yml no consigue acceder. Si queremos volver a acceder a la consola, comentamos esa linea (solución temporal hasta que averigüe si hay otra forma).

**Ejercicio 7**

**Propuesta**:

1. Añade una validación en el **@Service** que impida persistir User si no tiene informados la fecha de nacimiento o el nombre.

2. Crea un **ExceptionHandler** que controle el error de validación y devuelva un código de error acorde al problema.

3. Crea una respuesta por defecto para el resto de excepciones que devuelva el siguiente Json:

{

“type”: “{Exception class}”,

“title”: “Unexpected error”,

“status”: 400,

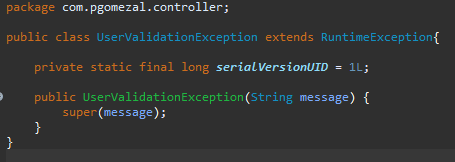
“detail: “An unexpected error occurred”

}

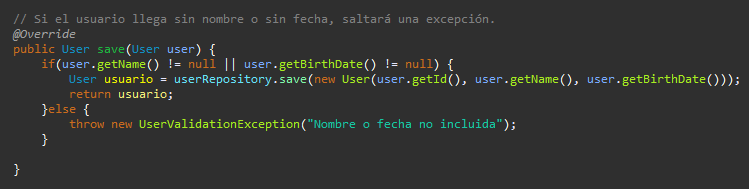
**Solución**:

**Paso 1**:

En primer lugar, creamos una Excepción personalizada a la que llamaremos **UserValidationException**. Se le pasará un mensaje por el constructor.

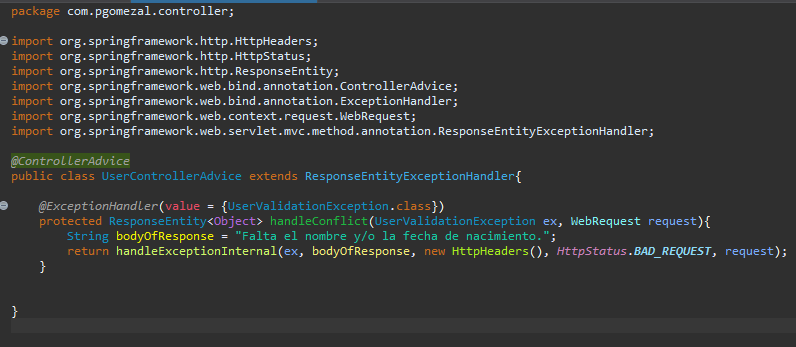


A continuación, en el método “save” del UserServiceImpl indicamos que si se le pasa un null por parámetro, lanzará la excepción anteriormente creada.



**Paso 2**:

Creamos una clase UserControllerAdvice que extienda de ResponseEntityExceptionHandler para manejar las excepciones y añadimos un método con la anotación @ExceptionHandler para controlar las excepciones del tipo que hemos creado, y que devuelva el tipo de error 400 (BAD\_REQUEST) y el mensaje solicitado.



No he sido capaz de realizar el paso 3 aún.