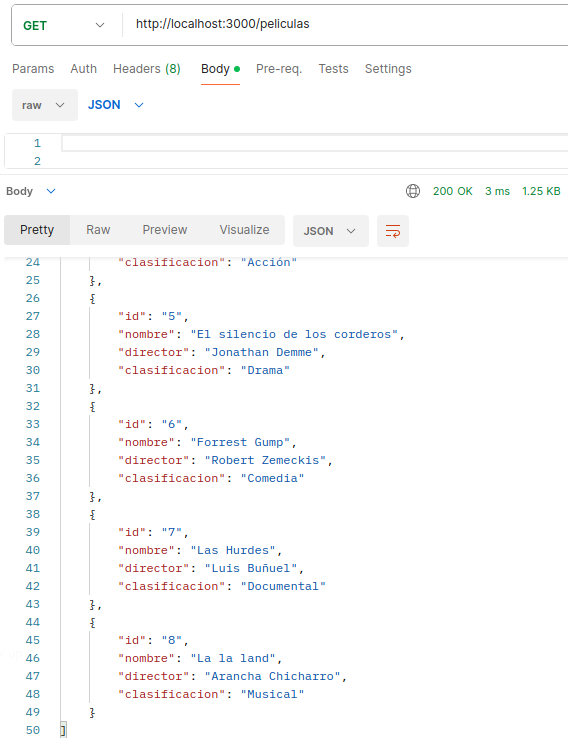
**Tarea 1 – UD6**

**PRIMERA API REST**

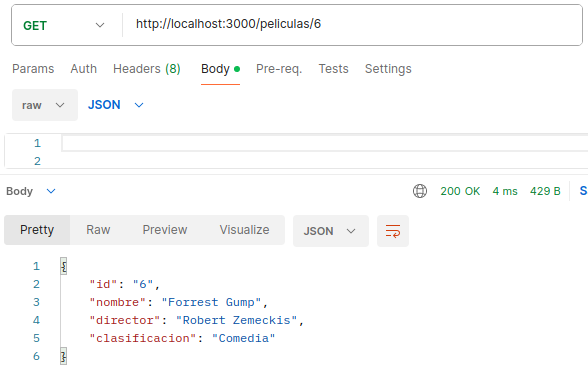
**Apartado 1.- API REST con Json-server:**

Describir el código de retorno y los datos devueltos para cada una de las siguientes peticiones:

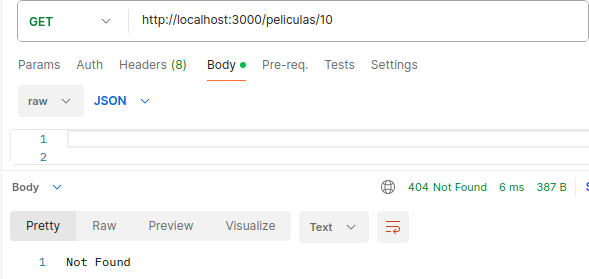
**GET lista entidades: código retorno 200 ok**

****

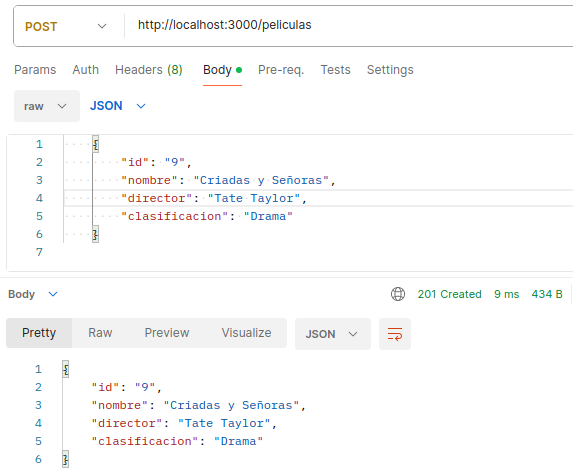
**GET de una entidad existente: código 200 ok**

****

**GET de una entidad que no exista (id inválido): código 404 not found.**

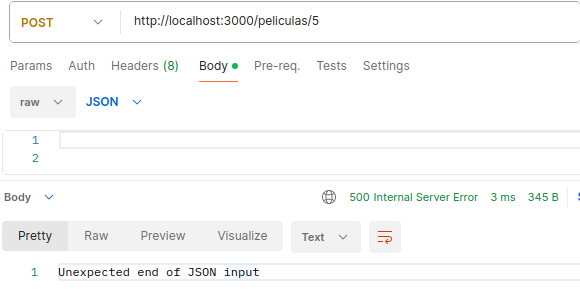
****

**POST de una entidad nueva: código 201 created.**

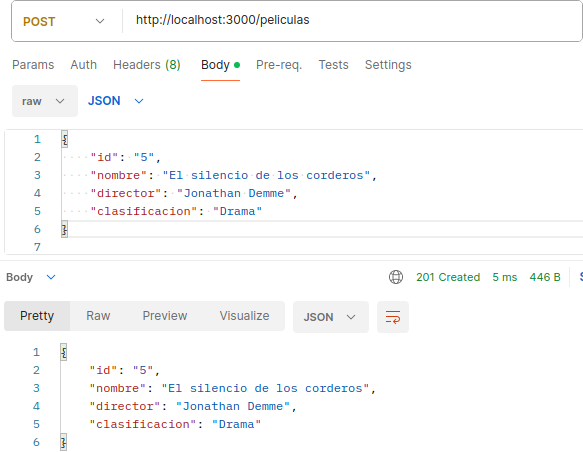
****

**POST de una entidad existente:**

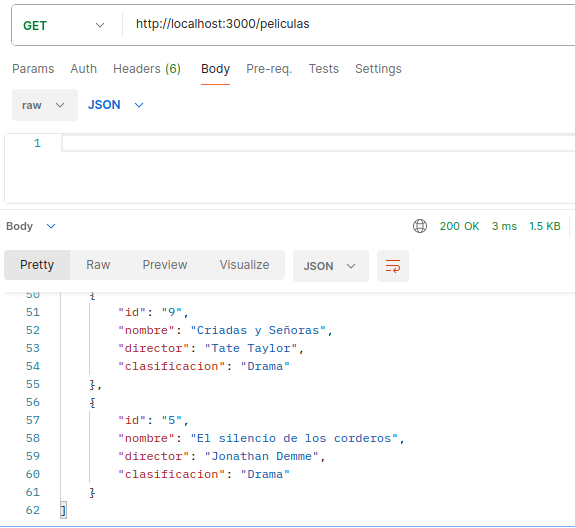
Si no enviamos datos al cuerpo: **código 50 Internal Server Error.**

****

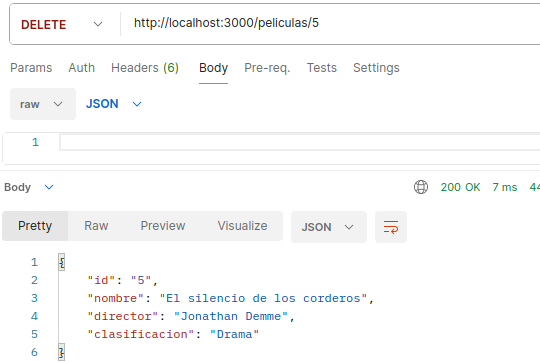
**Si enviamos un obj JSON ya existente en el cuerpo: código 201 created.**



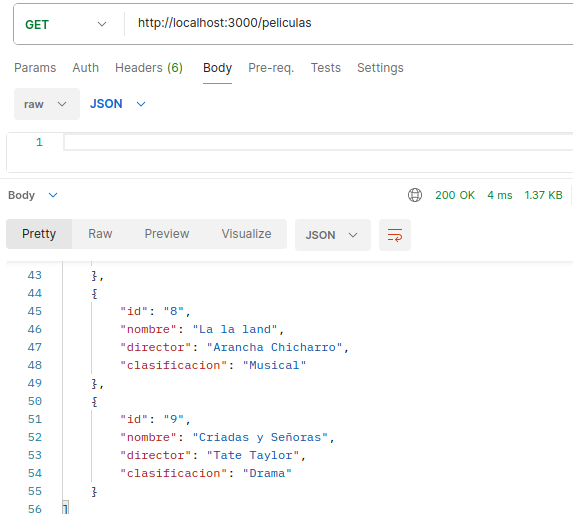
Según la lógica debería salir 200 ok pero sale creado. Veamos la lista completa a ver qué ha pasado. Vemos este elemento al final de la lista, por lo que POST lo ha creado como nuevo elemento y lo sitúa al final de la lista, por lo que ahora tenemos un elemento duplicado.



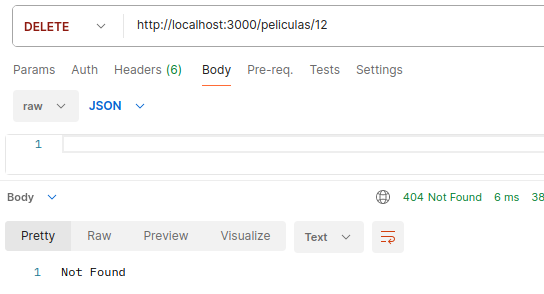
**DEL de una entidad existente: código 201 ok.**

****

Visualizo la lista nuevamente, y vemos que se ha eliminado este elemento 5 que teníamos repetido. De esta manera ya no tenemos elementos repetidos:

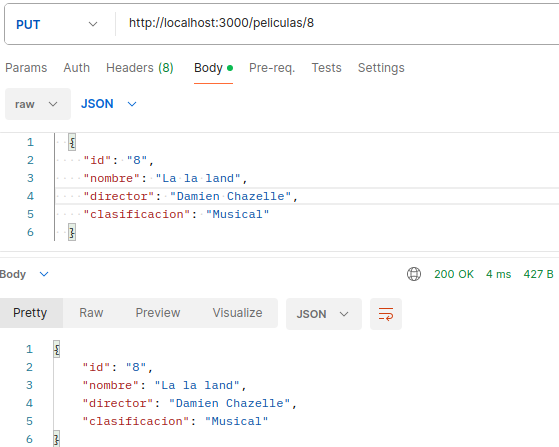


**DEL de una entidad que no existe: código 404 Not Found**

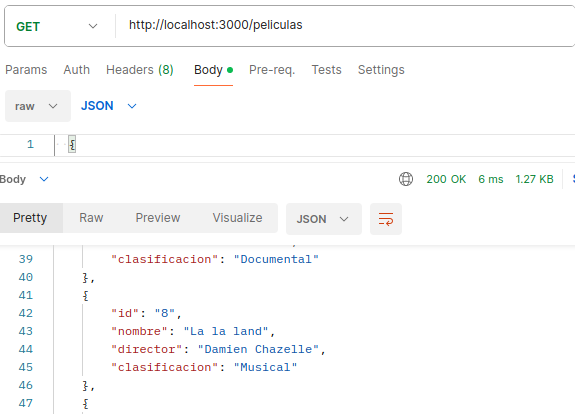
****

**PUT de una entidad existente: código 200 ok.**

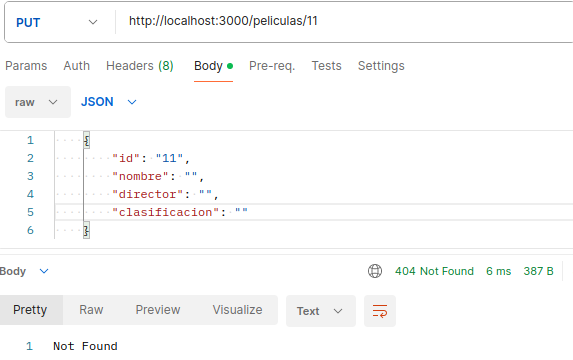
He actualizado el director de la película con id 8.



Lo compruebo en la lista nuevamente:

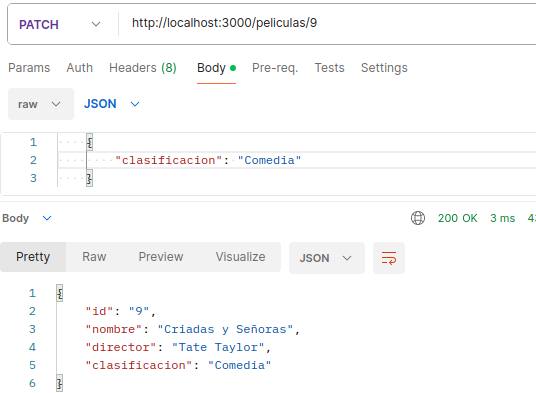


**PUT de una entidad que no exista: código 404 Not Found-**

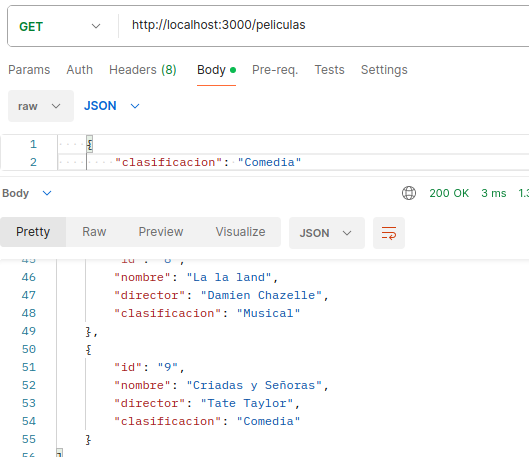


**PATCH de una entidad existente: código 200 ok.**

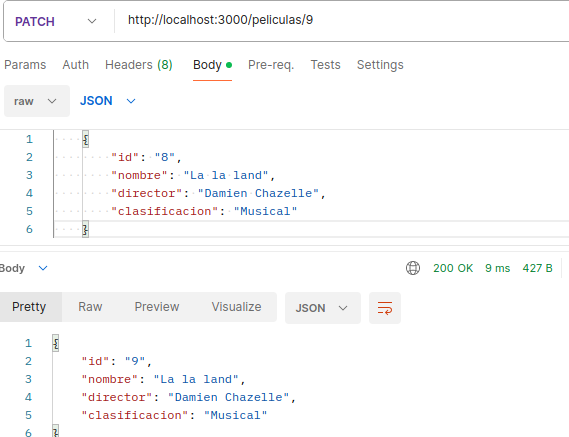
Pruebo a modificar el campo clasificación del elemento 9.



Lo compruebo visualizando la lista:

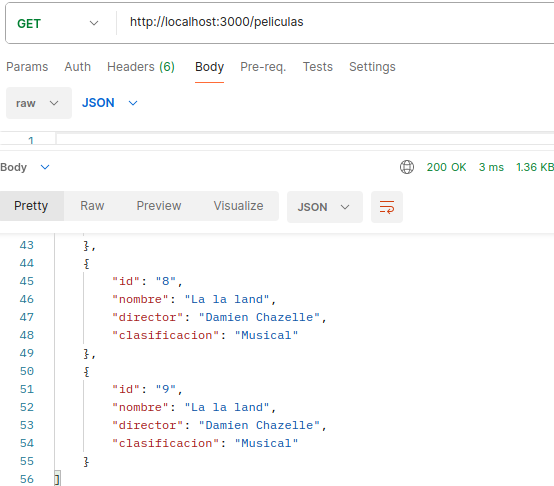


**PATCH de una entidad existente provocando incoherencia entre el id del path y el pasado en json: código 200 ok.**

****

Como ha salido OK, vemos qué ha pasado visualizando la lista.

Ha modificado el elemento con id 9, (el del path) lo ha actualizado con los elementos diferentes al id, es decir el nombre, el director y la clasificación, y los ha puesto con los del elemento 8 (los del body JSON):

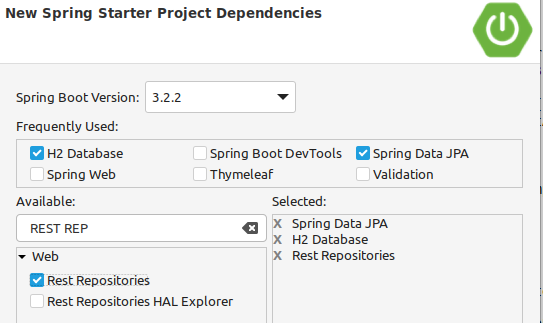


**Apartado 2.- API con REST Repository.**

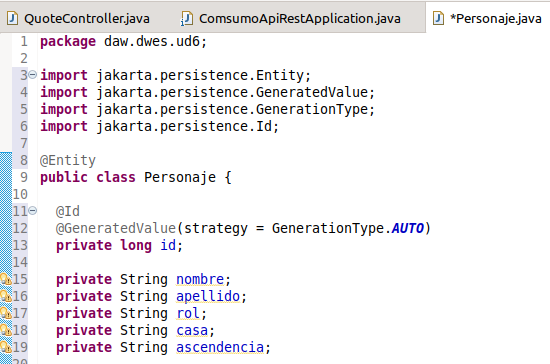
**Implementar el mismo API del apartado anterior pero esta vez con SpringBoot.**

Tutorial: <https://spring.io/guides/gs/accessing-data-rest/>

Creo proyecto llamado ApiRestRepository\_R1-EJ2, y seleccionó las siguientes dependencias:

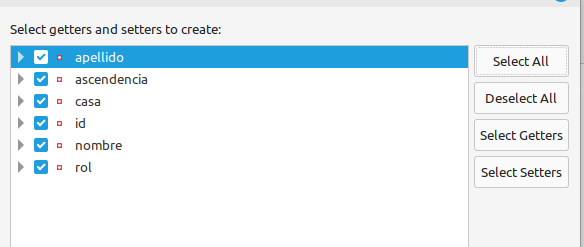


**Creo un nuevo objeto de dominio** para presentar a un personaje (voy a elegir personajes de Harry Potter). Y creo variables de los atributos que quiero que tenga mi objeto:

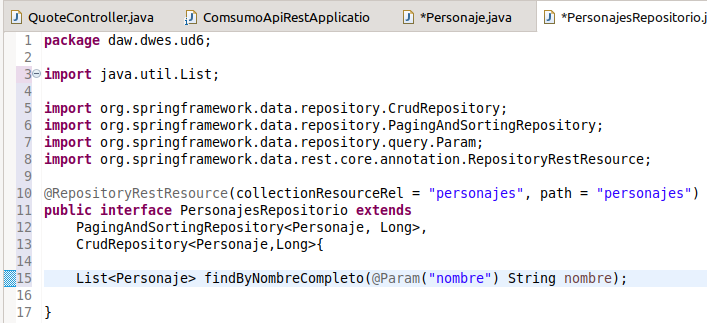


También hay un objeto ID que se configura para ser generado automáticamente gracias a sus anotaciones **GeneratedValue**.

Añado también los getters y setters.



**Creo un repositorio de personajes**. Para ello creo una interfaz:



Este repositorio es una interfaz que permite realizar varias operaciones que implican objetos Personajes.

Spring Data REST implementa automáticamente esta interfaz. Luego utiliza la anotación de @**RepositoryRestResource** para dirigir Spring MVC a crear terminales RESTful en /personajes.

Aquí también se ha definido una consulta personalizada para recuperar una lista de personajes basados en el **nombreCompleto** (he cambiado nombre por **nombreCompleto** y he eliminado **apellido**).

Si nos vamos a la clase application, vemos la anotación @**SpringBootApplication** que añade todo lo siguiente:

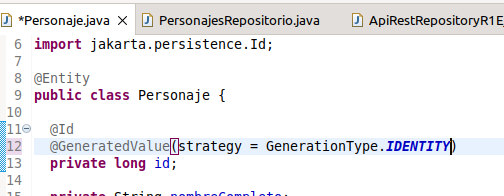
* @**Configuration**: Etiqueta la clase como una fuente de definiciones de bean para el contexto de aplicación.
* @**EnableAutoConfiguration**: Le dice a Spring Boot que comience a añadir beans basados en ajustes de classpath, otros beans y varios ajustes de propiedad.
* @**ComponentScan**: Le dice a Spring que busque otros componentes, configuraciones y servicios en el paquete daw/dwes, dejándole encontrar a los controladores.

Spring Boot activa automáticamente Spring Data JPA (gracias a la dependencia) para crear una implementación de **PersonajeRepository** y configurarlo para comunicarse con una base de datos back-end en memoria mediante JPA.

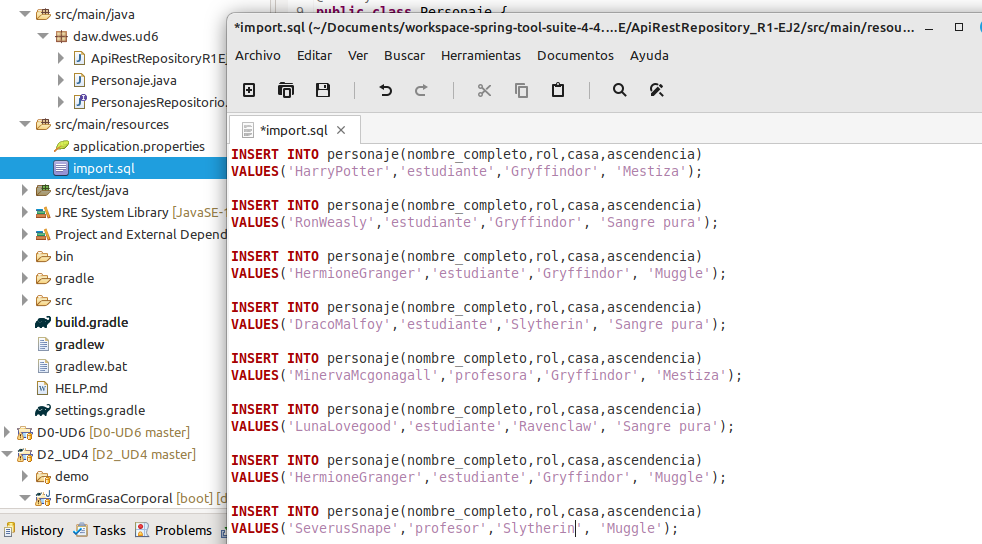
Spring Data REST se basa en Spring MVC. Crea una colección de controladores Spring MVC, convertidores JSON y otros beans para proporcionar una interfaz RESTful. Estos componentes se vinculan al backend de Spring Data JPA. Cuando usas Spring Boot, todo esto se configura automáticamente.

Podemos cargar la base de datos inicialmente. Para ello creamos un archivo import.sql en la carpeta resources, **importante, este archivo tiene que tener este nombre obligatoriamente.**

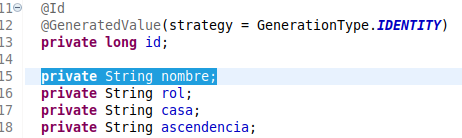
Antes de ello voy a eliminar el **id**, para que lo autogenere la bbdd, para ello en la clase Personaje, en el campo del id, en lugar de **AUTO**, ponemos **IDENTITY**.



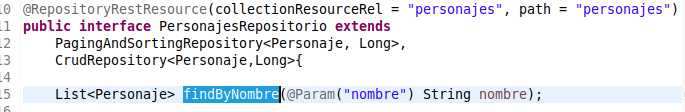
**Creo la bbdd** **import.sql** añadiendo algunos inserts con los datos de varios personajes:



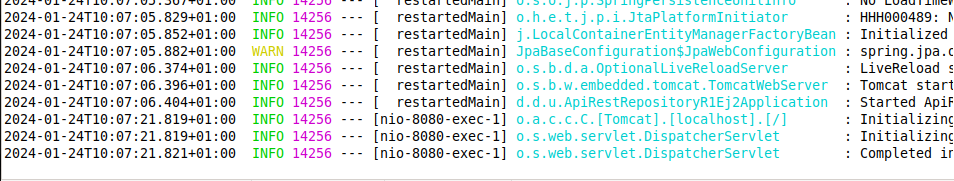
Al ejecutar la app, tenía errores de sintaxis en los insert del sql. Primero estaba poniendo el nombre de la tabla como personaje, pero mi clase es Personaje (la primera en mayúscula), pero seguí saliendo error, y tras corregir varias veces y seguir saliendo errores, decidí cambiar el atributo, y por lo tanto también columna, **nombreCompleto** a solo **nombre**:



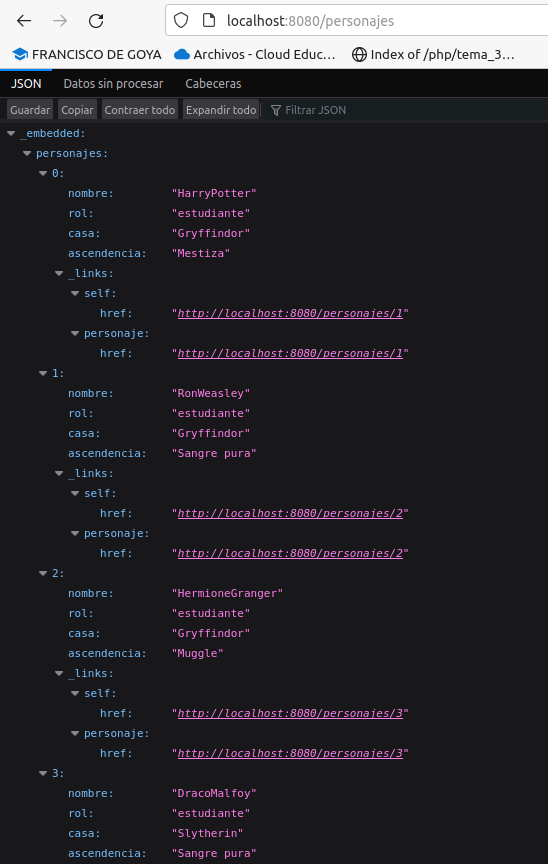
Importante también cambiar el método del nombre de la Lista en el repositorio a **findByNombre**:



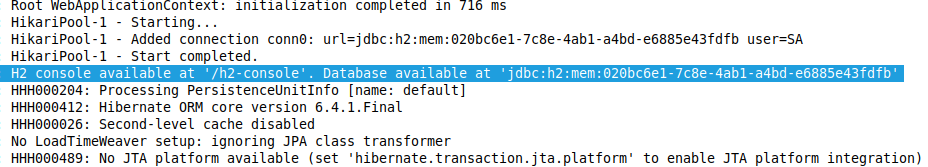
Y ejecutamos nuevamente. Ningún error en consola.



Y probamos en el navegador:

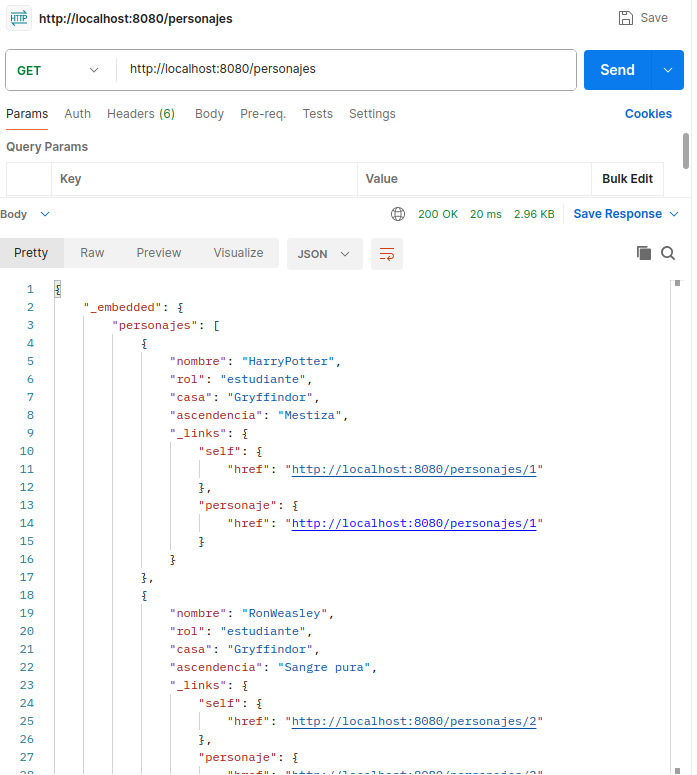


Tal y como indica el enunciado, vemos en los **logs** de la consola de STS la bbdd en memoria:

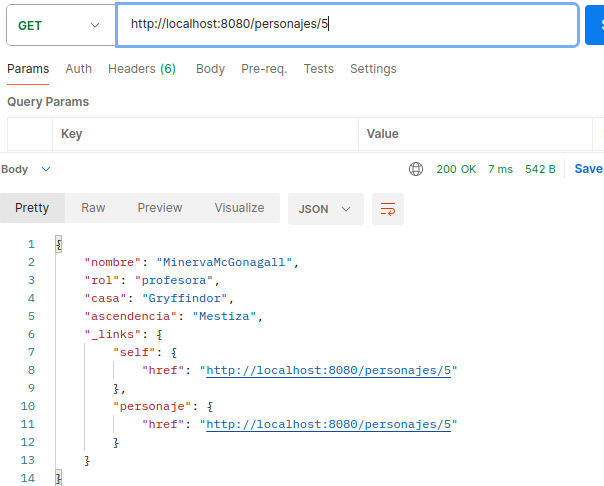


**Ahora vamos a hacer pruebas con Postman, como el ej 1.**

GET toda la lista: código **200 ok**

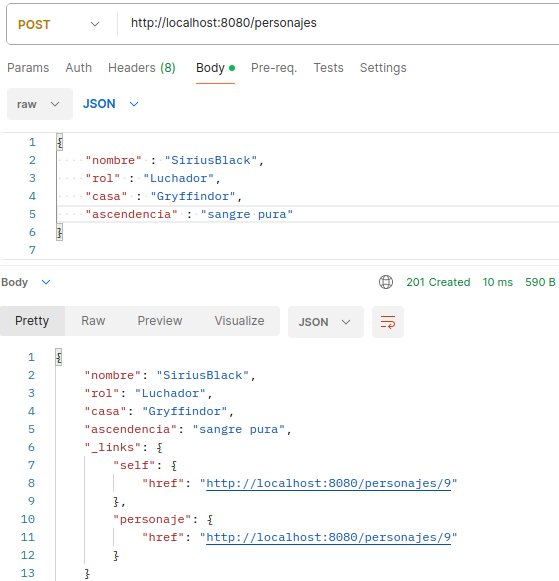


GET de un elemento: código **200 ok.**



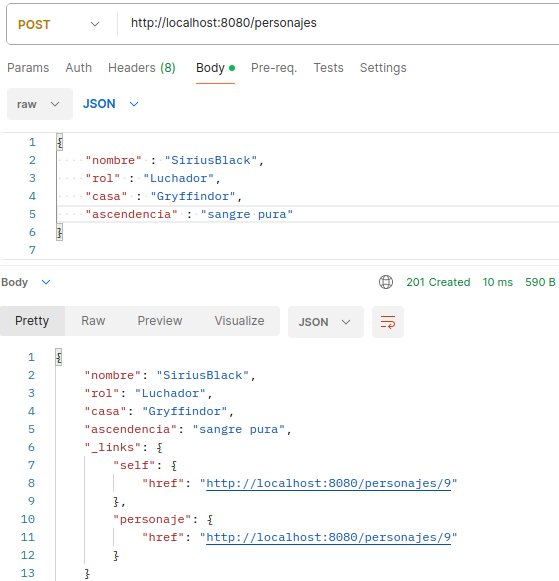
POST elemento nuevo: código **201 created.**

**Aquí vemos una diferencia con el ej1, aquí la respuesta es que también se crean los elementos links, tal cual aparecen los json en el navegador.**

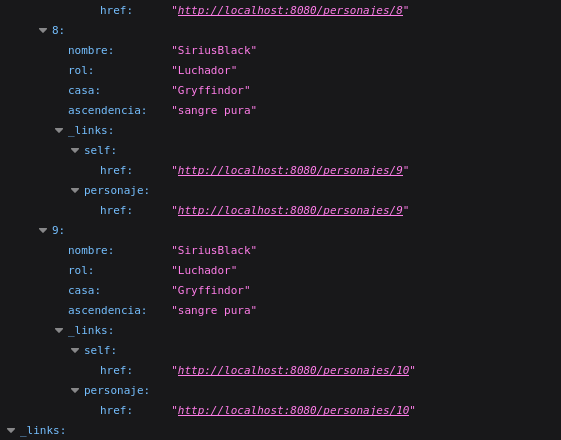


POST elemento existente: **201 created.**

Igual que en el ej1, se vuelve a crear y por lo tanto se duplica, con diferente id.

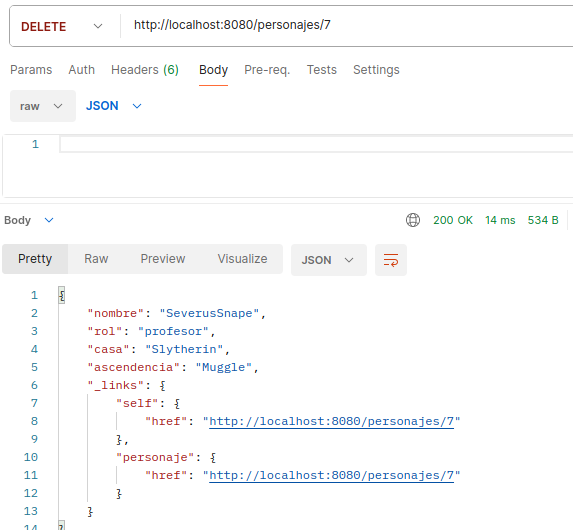


Aquí lo comprobamos en el listado:



DEL elemento existente: **200 ok.**

OJO! Aquí hay una diferencia. Como hemos quitado el id para ponerlo de forma automática en la bbdd, el primer elemento es el 0 y no el 1, por lo tanto al querer borrar el elemento con id 7, he borrado el 6, ya que el elemento con id 6 es el elemento 7:



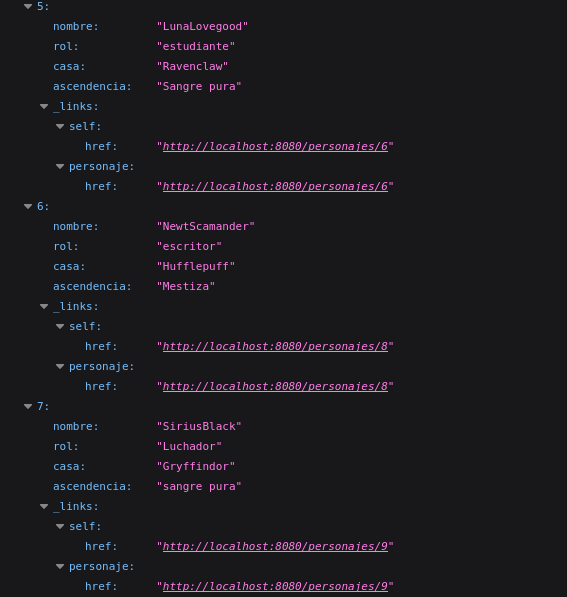


Con esto me doy cuenta de algo importante para las futuras consultas.

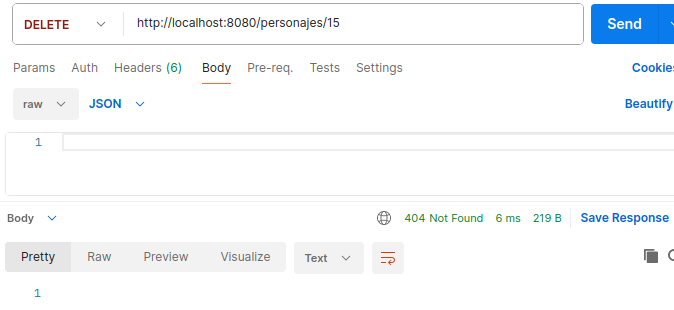
Se ha borrado un elemento, y por lo tanto la posición de los elementos lógicamente cambia, pero su id sigue siendo el mismo, por lo que hay que tener cuidado ahora en buscar los elementos por el nº del id y no por la posición, ya que no es el mismo.

Por ejemplo:

El elemento 7, SirusBlack, tiene id 9, por lo que para acceder a él tenemos que usar el 9.

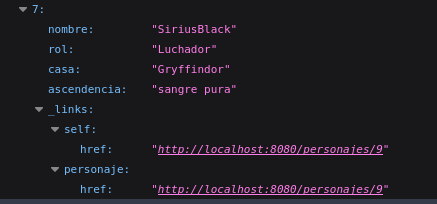


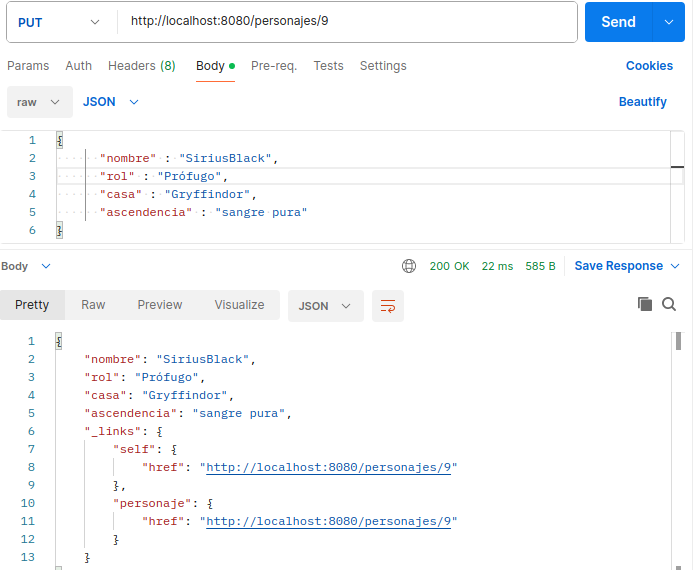
DEL elemento no existente: **404 Not Found.**



PUT elemento existente: **código 200 ok.**

Voy a cambiar el rol del elemento 7, id 9, de luchador a prófugo:

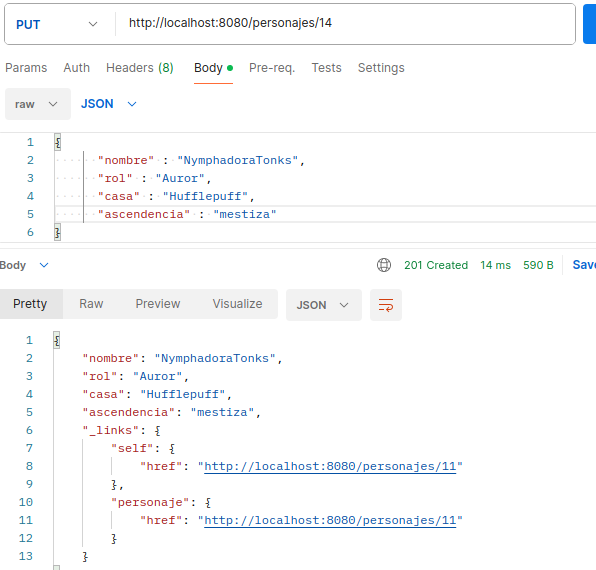






PUT elemento no existente: **código 201 created.**

Crea el elemento entero, a diferencia que con el ej1, json-server, que era **Not Found:**



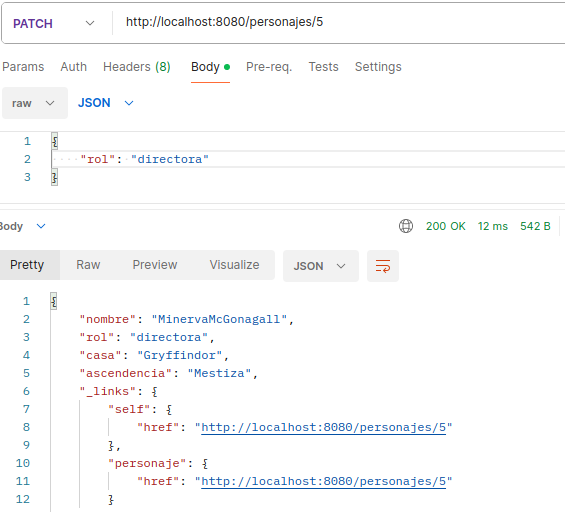
Y comprobamos en lista. Vemos que aunque en la url he puesto el id 14, el json se ha creado con el id 11.



PATCH elemento existente: **código 200 ok.**

Pruebo a modificar el rol del elemento con id 5, de profesora a directora:



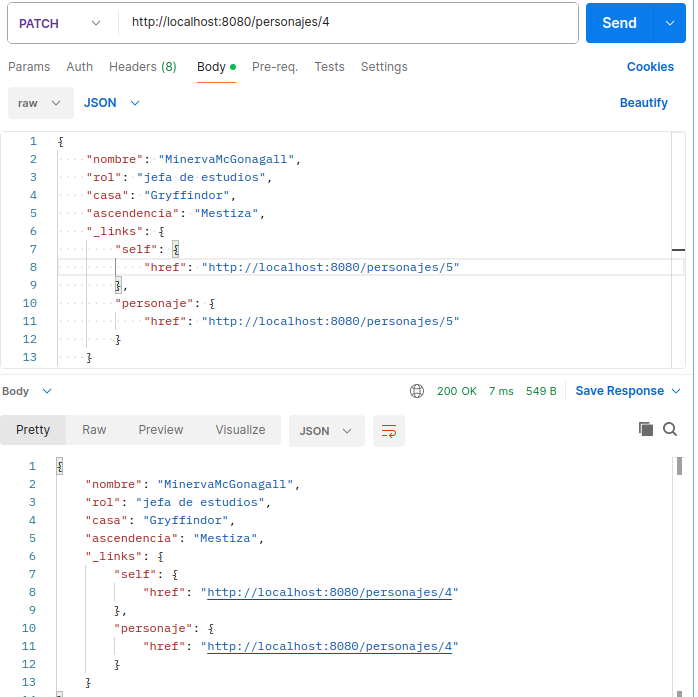


Y comprobamos en el listado que se ha cambiado correctamente:



PATCH de una entidad existente provocando incoherencia entre el id del path y el pasado en json: código **200 ok.**

Pongo en la url el id 4 (draco malfoy), pero en el body pongo el json del id 5 (minerva mcgonagal) cambiando el rol de directora a jefa de estudios.



La salida de consola es el elemento minerva mcgonagall y el id en los links es el 4.

Veamos el listado:

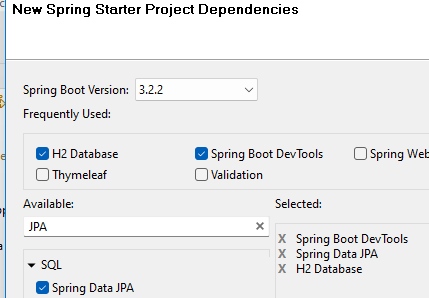


Lo que ha pasado es que ha modificado el elemento del id 4, el puesto en la url, con los datos del elemento con id 5, el del body. Ahora tengo dos elementos duplicados de minerva mcgonagall, con diferente rol y diferente id, y no hay elemento DracoMalfoy. Esto es igual que con json-server.

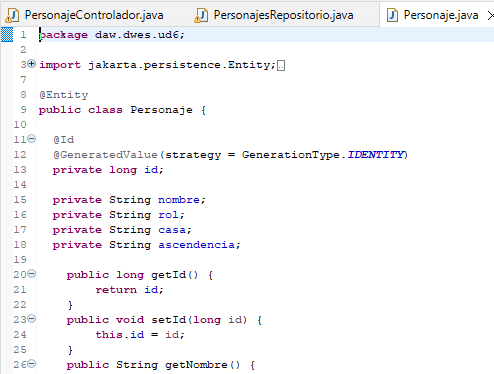
**Apartado 3. (extra)**

**API REST con @RestController (web) pero sin RestRepository.**

Voy a seguir el consejo del profesor, y en vez de usar una colección usaré la dependencia **H2 + JPA** y declarar la interfaz del repositorio. Por esta razón, al crear el proyecto de Spring, importo las dependencias **H2 Database y JPA.**



**Copio la clase Personaje.java y la dejo igual**.



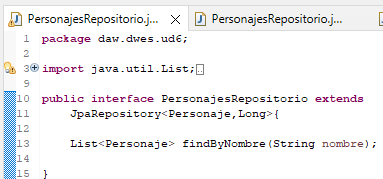
**Copiamos la interfaz PersonajeRepositorio** pero modificamos lo siguiente:

@RepositoryRestResource(collectionResourceRel = "personajes", path = "personajes")

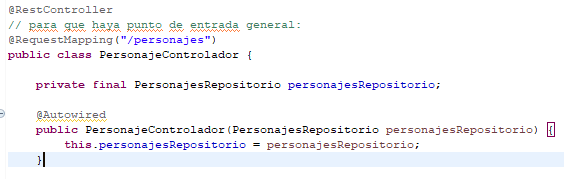
**extends** PagingAndSortingRepository<Personaje, Long>,CrudRepository<Personaje,Long>

@Param("nombre")

Y en su lugar la clase extiende de **JpaRepository**. Por lo que la clase quedaría así:



**Ahora creo la clase controlador llamada PersonajeControlador.**

****

- **@RestController:** lo aplicamos ya que no podemos usar RestRepository.

**- @RequestMapping(“/personajes”):** como punto de entrada general en todos los métodos.

- **private final PersonajesRepositorio personajesRepositorio:** declaramos esta variable para representar el respositorio, y la vamos a utilizar para acceder a los datos de los personajes.

**- @Autowired:** para indicar a Spring que inyecte una instancia de PersonajesRepositorio en este constructor cuando se cree una instancia de PersonajeControlador. Permite que el repositorio esté disponible para usarlo en este controlador.

**- public PersonajeControlador(PersonajesRepositorio personajesRepositorio):** este es el constructor e inicializa la variable personajesRepositorio. Se usa para inyectar el repositorio en el controlador.

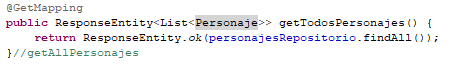
**Métodos para las rutas (peticiones y respuestas):**

Tienen en común que son de tipo **ResponseEntity.** Es una clase que representa la respuesta HTTP, tanto el código de estado, como los encabezados y cuerpo. Y al usarla, nos permite controlar cada aspecto de la respuesta HTTP.

En estos casos, la usamos para que el método devuelva el tipo de respuesta que queremos.

Explico cada método:

**Get - getTodosPersonajes:**



Este método nos devuelve todo el listado de personajes en la bbdd, por lo que tiene que ser de tipo List<Personaje>.

- **return**: ponemos el código de estado **ok,** y con la función **.findAll(),** conseguimos todos los elementos de personajesRepositorio.

**Get - GetPersonajeId:**

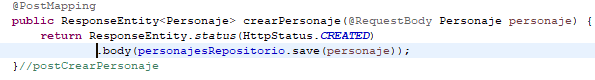


Nos devuelve solo un elemento de tipo Personaje, por lo que ya no es List.

Recibe por parámetro el id, que captamos con la anotación @PathVariable.

- **return**: código de estado **ok,** y con la función **.findById(id)** encuentra el elemento que queremos y con la función **.get(),** accedemos a él.

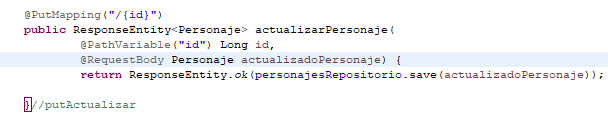
**Post - crearPersonaje**



Creamos un personaje, y como tenemos que indicar todos los atributos, recibe un elemento personaje, que captamos con la anotación **@RequestBody**.

- **return**: usamos la función **.status** que hace que podamos manejar el estado de la petición y con **HttpStatus** indicamos el código que queremos que nos devuelva, **CREATED** en este caso. Y con la función **.body** indicamos qué queremos mostrar en el body de la respuesta, en este caso es el personaje que hemos creado y por lo tanto guardado, con la función **.save.**

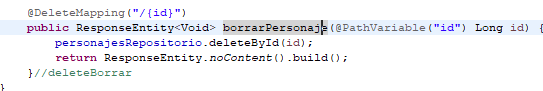
**Put – actualizarPersonaje:**



Actualizamos un personaje accediendo a él mediante un id que captamos con **@PathVariable**, y también recibe en el cuerpo un objeto personaje entero que captamos con **@RequestBody.**

- **return:** indicamos el código de estado **ok,** y mostramos el personaje que hemos actualizado/guardado con la función **.save.**

**Delete - borrarPersonaje:**



Borra un personaje recibiendo el **id**  que captamos con **@PathVariable.**

Borramos el perosnaje con la función **.deleteById(id).**

**- return:** indicamos el código de estado **noContent.** Y como este código no tiene contenido de respuesta, para finalizar la construcción de respuesta se usa la función **.build().**

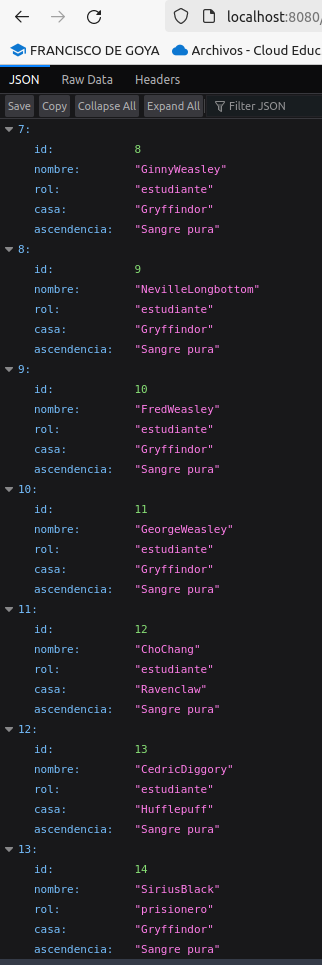
**Copio el import.sql** del ej anterior pero he añadido más personajes, un total de 26 para así hacer pruebas más interesantes:



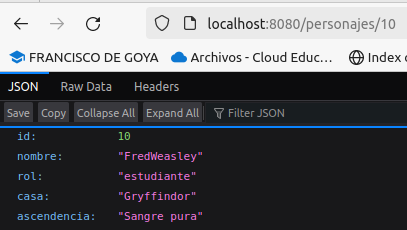
**PRUEBAS EN EL NAVEGADOR:**

En el navegador haremos las consultas de tipo Get.

**- Get getTodosPersonajes**: nos muestra toda la lista de nuestros objetos Personajes en formato Json:

**- Get getPersonajeById**, accedemos al personaje con id 10:



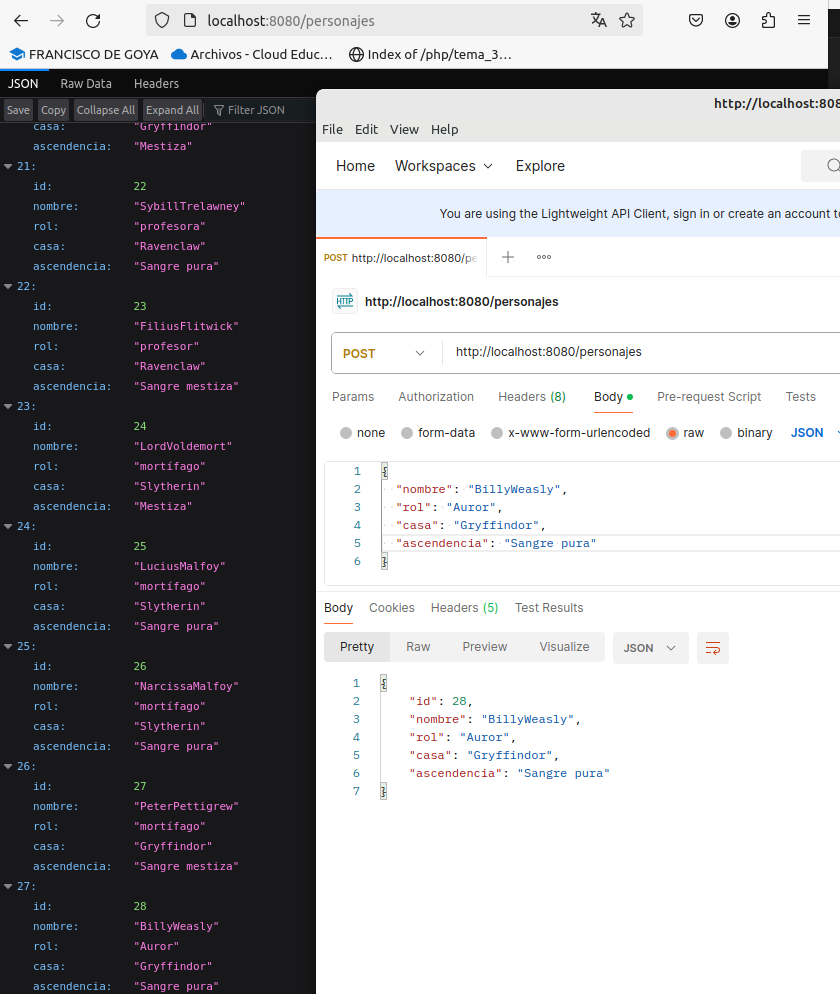
**PRUEBAS POSTMAN:**

En postman hacemos el resto de consultas (POST, PUT, DELETE).

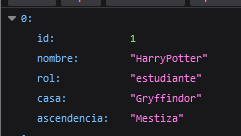
**Post crearPersonaje:**

Creo personaje que no existe, BillyWeasly. No pongo id porque se crea automáticamente.

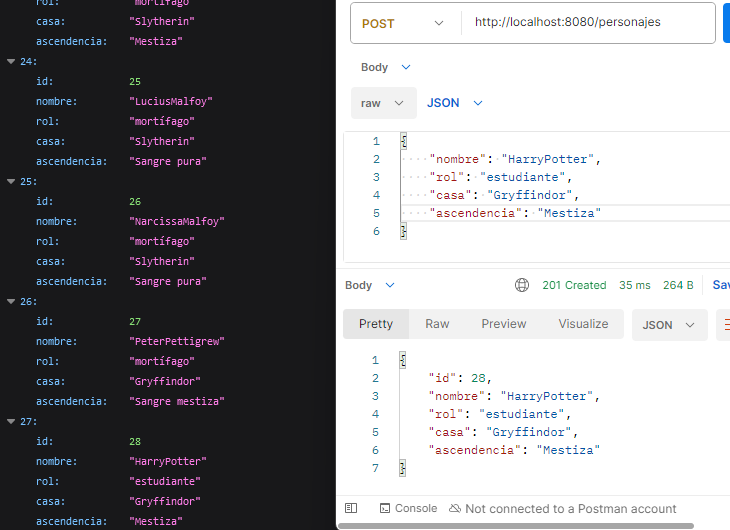
Código respuesta **201 created**, y compruebo en la lista que se ha incluido (ver en la izquierda), se ha creado con id 28 y posición 27:



Ahora creo elemento que ya existe, HarryPotter .Vemos que existe en la lista:



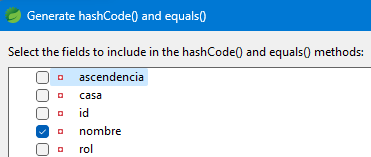
Y probamos. El código de estado es **201 created,** vemos que la salida es el elemento con los mismos datos pero con id 28, por lo que el elemento se ha creado y ahora tenemos dos elementos duplicados, aunque con diferente id:

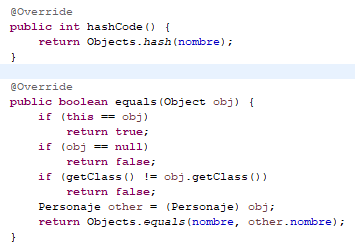


Para evitar esto, siguiendo con la segunda parte del enunciado, voy a modificar el código para que no se permita insertar elementos que ya existen. Además voy a incluir un código de estado y mensaje personalizados.

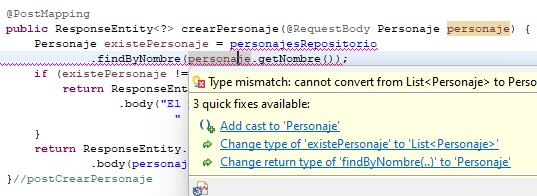
Para ello voy a crear **equals y hashCode** eligiendo el campo **nombre**  para que marque la identidad, pues en el caso de mi aplicación, mis personajes solo tienen el Nombre como atributo único (además del id que se crea automáticamente), pues los demás atributos (rol, casa y ascendencia) son elementos comunes con otros personajes.

Vamos a la clase Personaje, y creo equals y hashCode:

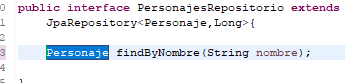




Y ahora lo implemento en el método **POST crearPersonaje** del controlador.

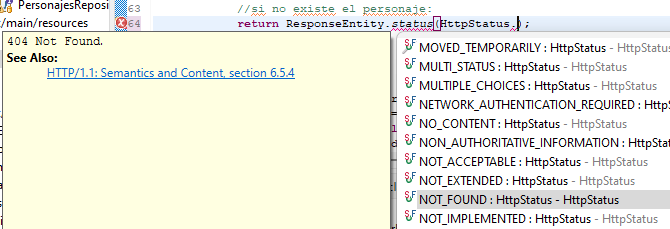
Mi primera modificación es esta, pero sale error y el código propone 3 correcciones:  


Estas tres opciones de corrección es porque con esa línea, estábamos devolviendo una lista, es decir, a la variable Personaje se le está asignando el valor de toda la lista de personajes. Por lo que voy a elegir la opción3, voy a ir a la clase PersonajesRepositorio y cambio el método findByNombre de List<Personaje> a Personaje:



Y para crear el código y mensaje personalizado, ResponseEntity tiene una función de status que permite indicar el código con el que quieres que responda la petición (el que hemos usado en el método post). En vez de indicar el código, voy a indicar el nombre del código (OK, NotFound, Create, etc), y estos mensajes/códigos los saco de este link de las referencias del tema, donde indica qué códigos de estado pueden salir con cada método: <https://restfulapi.net/http-methods/>

Aunque también puedes ver qué status hay, desde sts ya que el código lo reconoce:



En este caso voy a indicar el código de estado **METHOD\_NOT\_ALLOWED** si el personaje ya existe y **CREATED**  si no existe y se crea.

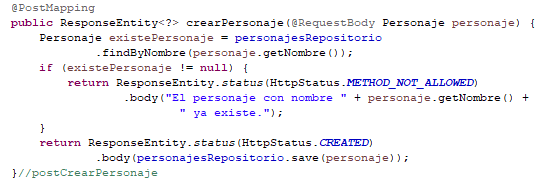
En este último caso, el código detecta error para cambiar el método a que devuelva un string en lugar de un personaje, pero como si existe el personaje lo tiene que devolver, entonces, tras consultar en ChatGpt propone devolver un <?>.

Otro de los cambios que he tenido que hacer es la línea:

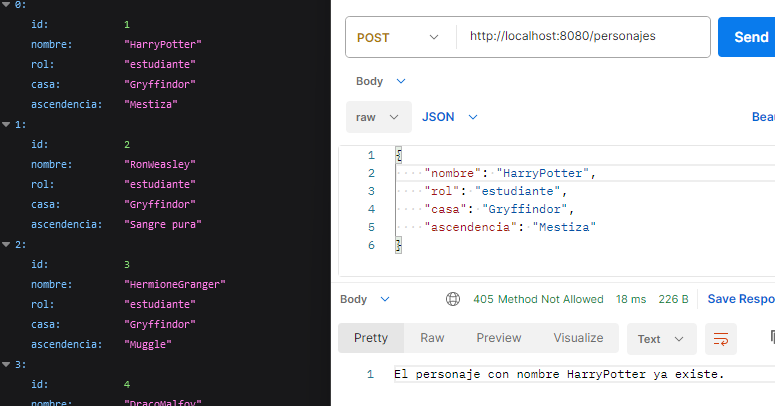
**Personaje existePersonaje = personajesRepositorio.findByNombre(personaje.getNombre());**

Esta línea busca un personaje de la lista con el **nombre** del personaje del parámetro del que cogemos el nombre con **.getNombre** y almacena el resultado en **existePersonaje**. Y mediante el equals y hashcode la aplicación hace la comparación automáticamente. Por lo tanto, si hay un personaje en la lista con ese nombre, si existePersonaje no es null podemos avanzar en el código.

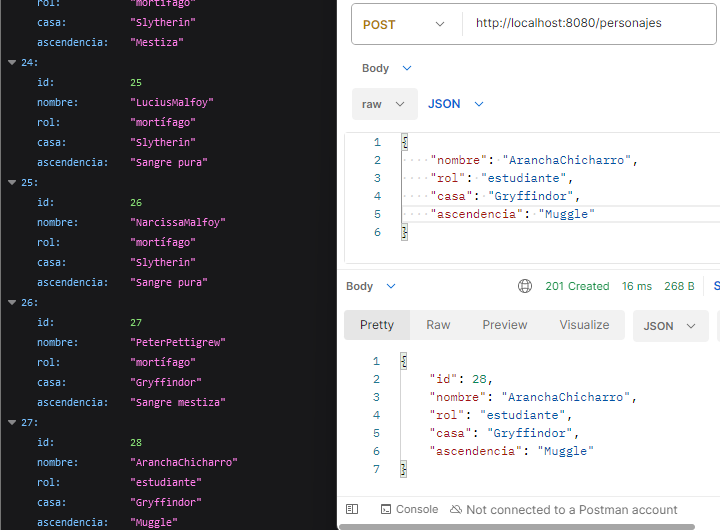
Por lo que el código de post **crearPersonaje,** quedaría así:



Y probamos a añadir el personaje con nombre HarryPotter, que existe, y la respuesta es la que esperábamos: código **404 MetthodNotAllowed.**



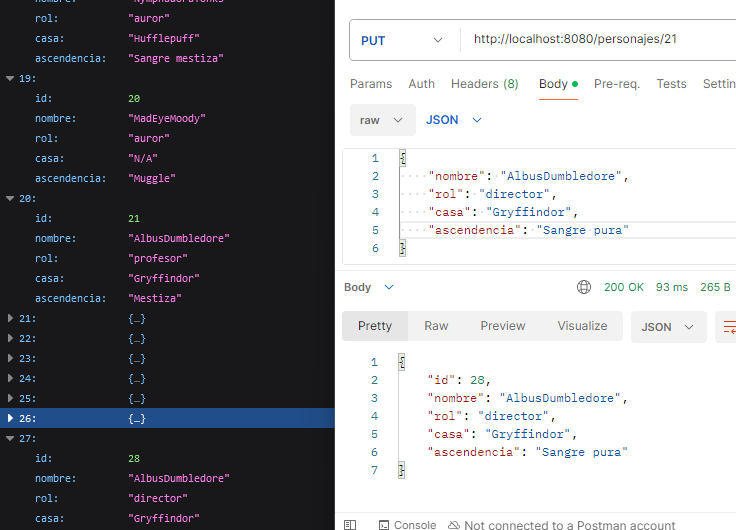
Probamos ahora con un personaje nuevo, para verificar que al modificar el código, sigue funcionando con las dos posibilidades:



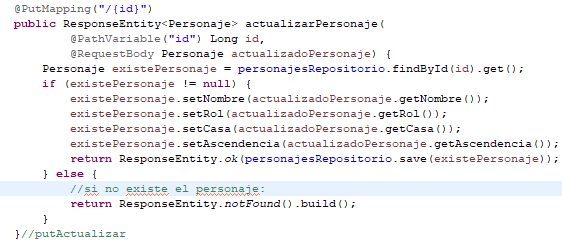
**Put actualizarPersonaje:**

Accedo al personaje con id 21, AlbusDumbledore y cambio su rol de **profesor** a **director**. Y su ascendencia de **mestiza** a **sangre pura.**

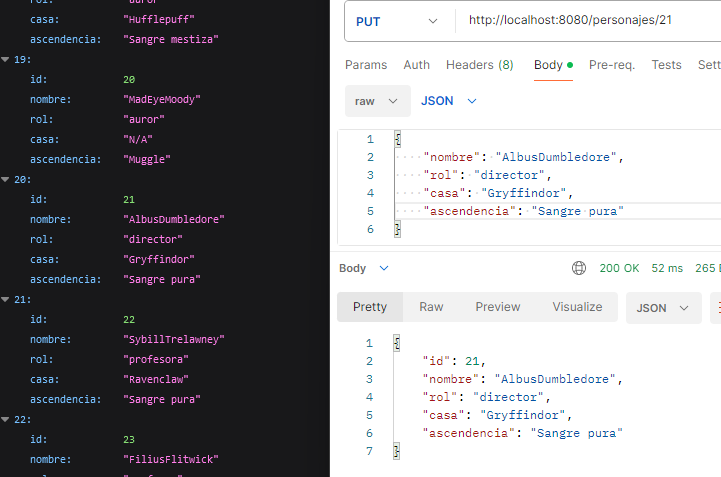
El código es **200 ok**, pero la salida es un elemento con id 28, y vemos en el listado (ver izq.) que se ha creado un elemento nuevo y que el objeto con id 21 no se ha modificado:



Vuelvo al código del controlador, y reviso. Creo condición de “si existe el elemento”, para que se modifique el elemento en cuestión. Para ello, creo variable de tipo personaje, y con los métodos set y get correspondientes por cada atributo, actualizo los valores:

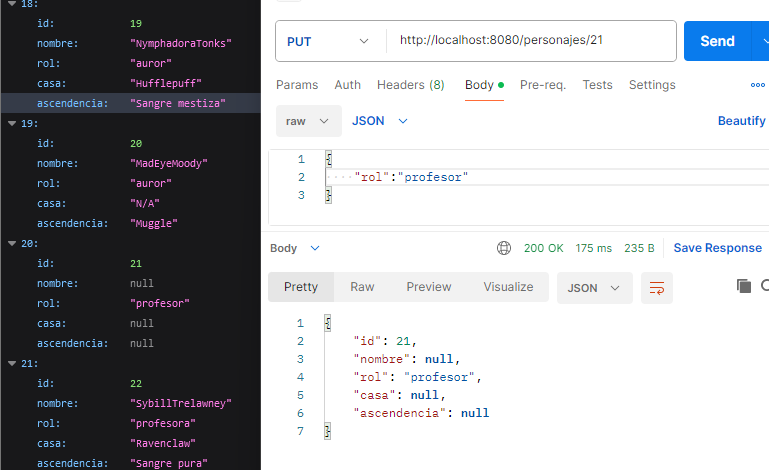


Ahora vuelvo a hacer la misma prueba y es correcto. La salida es el elemento con id 21 y los nuevos valores actualizados, y vemos en la lista que efectivamente el objeto con id 21 se ha actualizado con los nuevos valores:

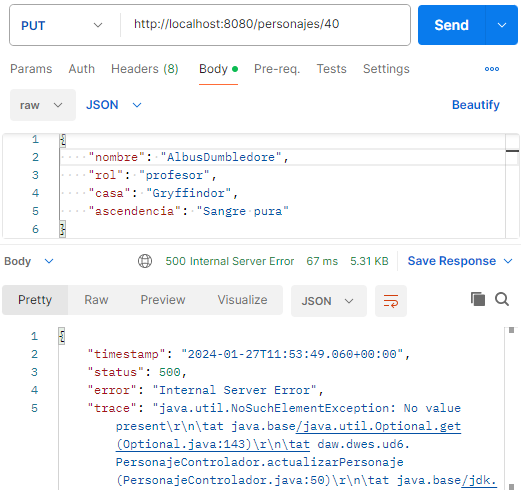


Ahora voy a hacer el PUT poniendo solo el rol, accedo de nuevo al personaje con id 21 y vuelvo a cambiar el rol de director a profesor.

Y vemos que se ha actualizado todo el objeto, pues efectivamente lo que hace PUT es actualizar todo el objeto, por lo que el resto de elementos los ha dejado como null (así lo hemos puesto en el método put, con cada set/get, al no introducir un valor, lo pone como null):



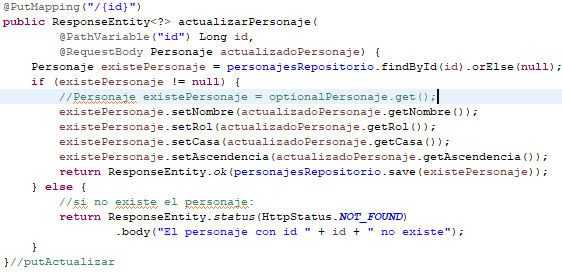
Ahora hago una consulta put de un personaje con id que no existe, por ejemplo el id 40, y sale error 500 con lo que corresponde al servidor, lo normal es que salga algún error 400, del cliente, como not found.



En este caso de PUT lo que quiero es que si se intenta actualizar un personaje con id que no existe, que ponga el código de **not found,** por lo voy a personalizar el código de estado al igual que en el método Post. Además voy a poner un mensaje de error personalizado, algo que también permite el ResponseEntity con su función **body**.

Y como en el caso del método Post tenemos que poner el tipo de método a <?>.

Tras modificar código y dar errores, y por lo tanto, tras hacer varias modificaciones, esté método **actualizarPersonaje** finalmente quedaría así:



Otro de los cambios que he tenido que hacer es la línea:

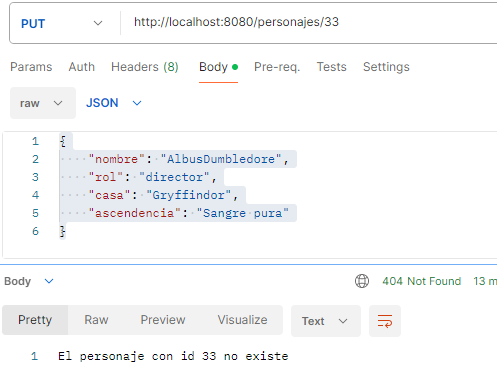
**Personaje existePersonaje = personajesRepositorio.findById(id).orElse(null);**

Esta línea busca un personaje con el **id** especificado con el **.findById(id)** y almacena el resultado en **existePersonaje**. Si el personaje no se encuentra, **existePersonaje** será **null** (gracias al **.orElse(null)**).

Y de esta manera podemos manejar el objeto tanto si tiene valor para que devuelva el personaje y como si no tiene valor, si es null, para que devuelva nuestro mensaje personalizado.

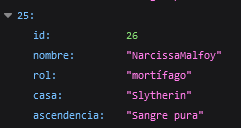
Vamos ahora a hacer una nueva prueba de este PUT con un elemento que no existe.

Y podemos comprobar que sale el estado **404 NotFound**, y en el cuerpo, nuestro mensaje personalizado:

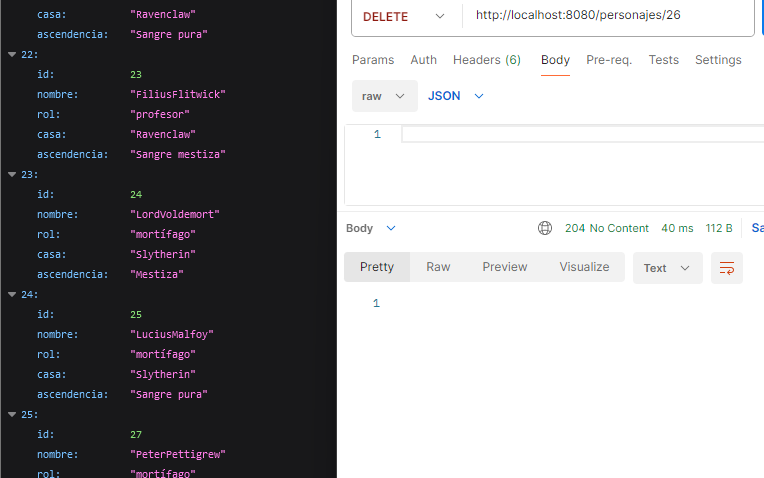


**Delete borrarPersonaje** :

Voy a probar a borrar el personaje con id 26 (NarcissaMalfoy):

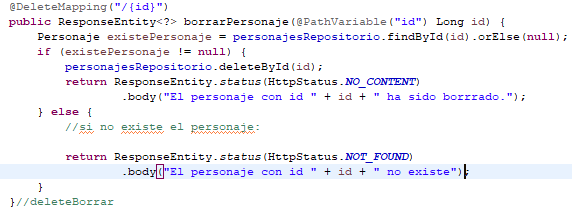


El código es **204 no content** y vemos en la lista (ver izq.) que efectivamente se ha borrado el elemento con id 26:

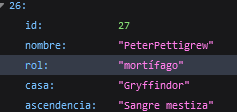


Pruebo a borrar un elemento que no existe, busco el elemento con id 45, y la salida es la misma: **no content** cuando es más “lógico” que salga un **notFound**.

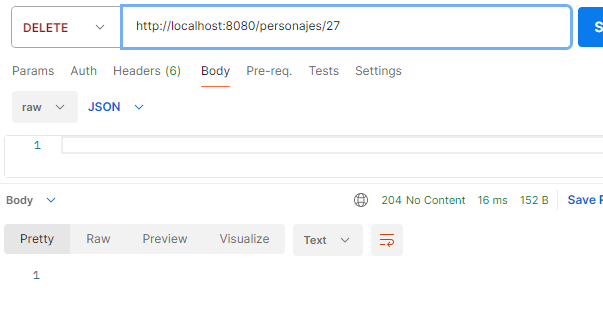
Por lo que vuelvo al código del controlador y añado también una condición de si existe que lo borre y salga este mensaje de **no content**, pero si no existe, que salga error, igual que los métodos Post y Put.  
Y aprovecho para también añadir mensajes personalizados, tanto cuando existe el elemento como si no, fijándome en el método PUT anterior:



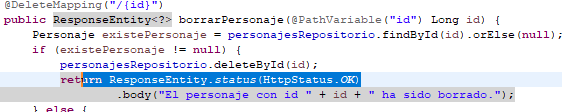
Pruebo ahora a borrar el elemento con id 27:



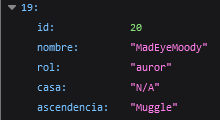
El elemento se borra, pero no sale el mensaje personalizado.



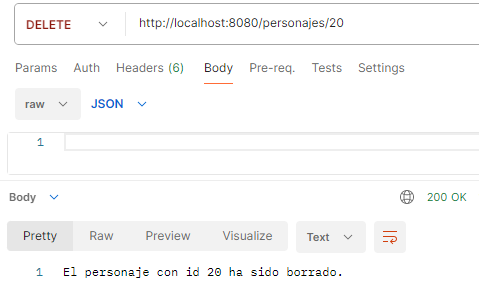
Tras revisar el código, que no veía ningún fallo, me di cuenta que el status que había puesto, el **no Content**, es cuando la acción se ha realizado pero la respuesta no incluye entidad. Esto me di cuenta gracias al enlace de las referencias del tema, indicado anteriormente (<https://restfulapi.net/http-methods/>). Modifico el código poniendo el status **OK**, pues es otras de las respuestas de **DELETE**, y este si permite devolver contenido:



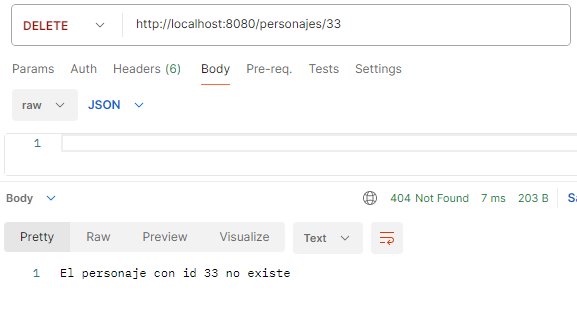
Y vuelvo a hacer una prueba de borrar. Voy a borrar el elemento con id 20:



Y ahora sí que nos muestra el mensaje personalizado, y por supuesto el nuevo código, el **200 ok**:



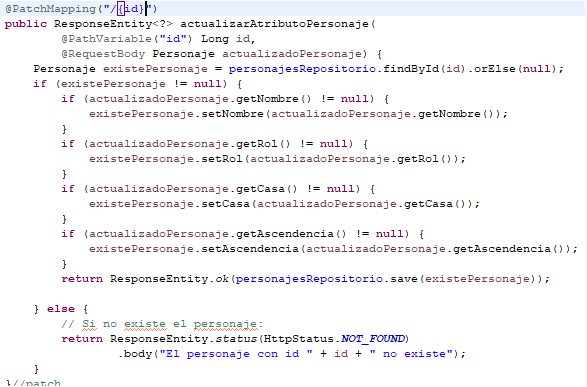
Y ahora pruebo a borrar el elemento 33, que no existe. Y sale en el body nuestro mensaje personalizado:



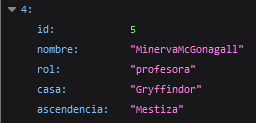
**Patch actualizarAtributoPersonaje:**

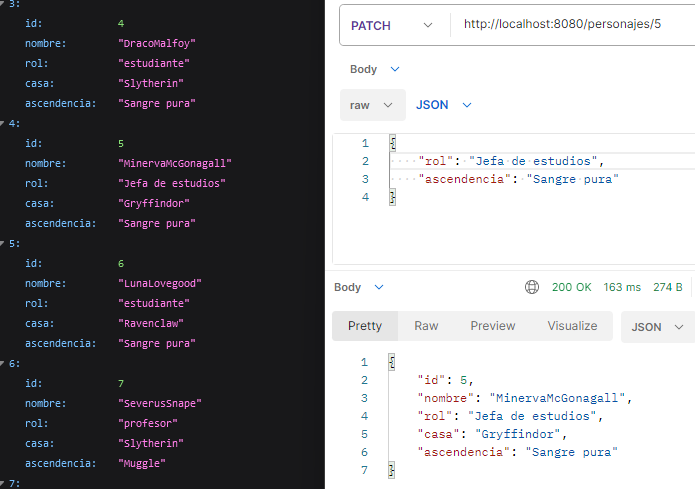
Voy a crear un método patch para así también poder actualizar solo algunos atributos de nuestro objeto personaje. Me voy a basar en el método put, añadiendo también código y mensaje personalizado.

Aquí no podemos poner todos los set/get de cada atributo porque si no se actualizarían todos los atributos igual que en el put, por lo que tendremos que poner condicionales if por cada uno de ellos, siendo la lógica: si el cuerpo recibe nombre, se actualiza nombre, si recibe casa, se actualiza casa, etc. Y por lo tanto si no recibe dicho atributo, no se actualiza y se quedaría igual.  
El código sería este:

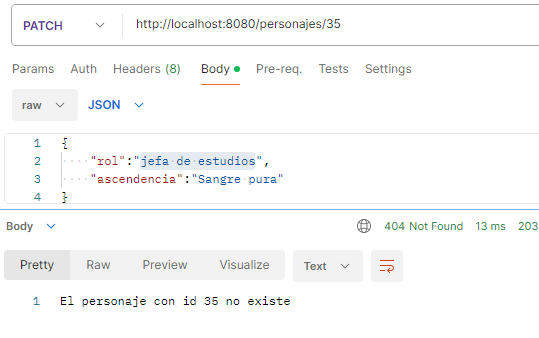


Pruebo a actualizar el personaje con id 5, cambiando su rol de profesora a jefa de estudios y su ascendencia de Mestiza a Sangre pura, por lo que en el cuerpo solo pongo estos datos. Su código es **200 ok**, y vemos en la lista de la izquierda que se ha actualizado correctamente

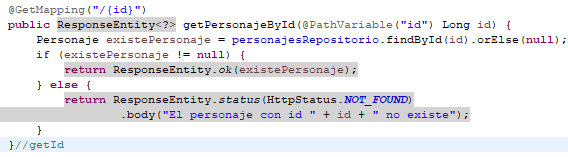




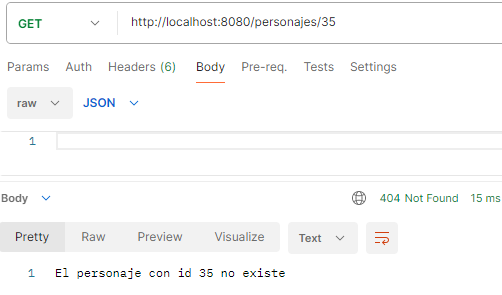
Y ahora pruebo con un elemento que no existe, y sale nuestro mensaje personalizado:



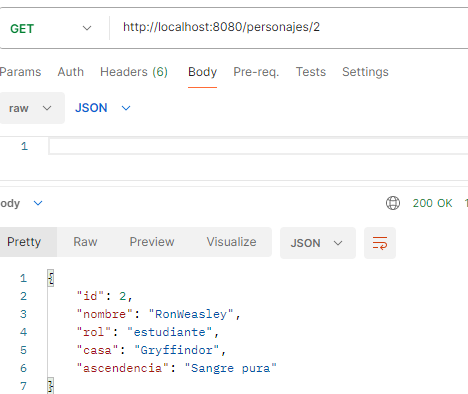
Ahora voy a volver al método **getPersonajeById**, para añadir también un mensaje personalizado en el caso de que no exista el elemento, por lo que siguiendo el ejemplo de los demás métodos, también tengo que crear variable de tipo personaje de si existe. El código quedaría así:



Y probamos en postman:



Probamos también a hacer get de un elemento que existe para comprobar que no hemos estropeado el código:



Y en relación con la otra parte del enunciado:

* **Que no dejamos insertar elementos que ya existan (elegid los campos que marquen la identidad, convendría crear equals y hashCode de nuestra entidad).**

Realizado:

Cómo mi método para insertar elementos es el **POST crearPersonaje,** he aplicado ahí esta funcionalidad.

* **Que no dejamos hacer PUT o PATCH sobre elementos que no existan.**

Realizado:

Al crear la variable Personaje existePersonaje y usar el condicional de si existePersonaje no es null.

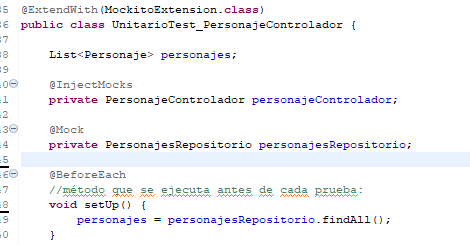
**Apartado 5.- (extra) Automatización de las pruebas.**

Voy a adaptar el ejemplo del tutorial a mi API anterior, personajes Harry Potter. **Por lo que voy a hacer los test en el mismo proyecto.**

**TEST UNITARIOS**

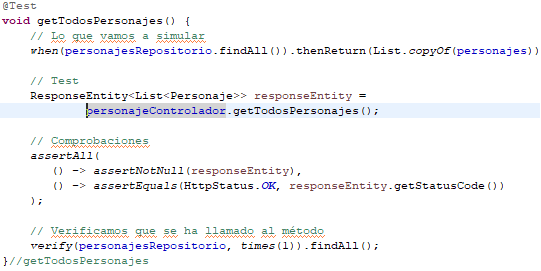
Para probar una unidad de código de forma aislada. Y se suelen utilizar mocks para aislar la unidad de código que estamos probando de las dependencias que tiene.

Siguiendo el segundo ejemplo de este apartado, voy a aplicar el test a mi controlador.



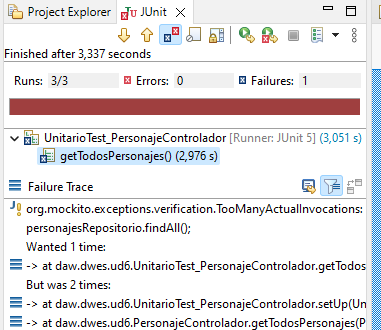
1. **@ExtendWith(MockitoExtension.class)**: anotación que indica que se debe utilizar la extensión **MockitoExtension** para ejecutar los tests. **MockitoExtension** proporciona soporte para la creación y manejo de objetos simulados (mocks) utilizando Mockito.
2. **@InjectMocks**: anotación de Mockito que se utiliza para inyectar automáticamente mocks. En este caso, se está inyectando un mock de **PersonajeControlador** en el campo **personajeControlador**.
   * **private PersonajeControlador personajeControlador;**: necesario crearlo para probar los métodos de la clase **PersonajeControlador**.
3. **@Mock**: anotación de Mockito que se utiliza para crear un mock de la clase **PersonajesRepositorio**.
   * **private PersonajesRepositorio personajesRepositorio;**: se utilizará para simular el repositorio de personajes.
4. **@BeforeEach**: anotación que indica que el método **setUp()** se ejecutará antes de cada método de prueba.
   * **Void setUp():** En este método, se inicializa la lista de personajes obteniendo todos los personajes del repositorio.

**Test del método getTodosPersonajes:**

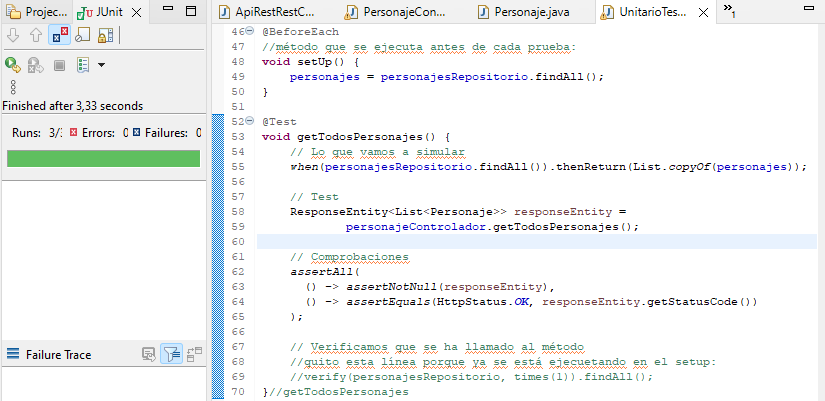


* **@Test**: indica que el método **getTodosPersonajes()** es un método de prueba.
* **when(personajesRepositorio.findAll()).thenReturn(List.copyOf(personajes));**: Aquí se configura el comportamiento del mock **personajesRepositorio**. Se especifica que cuando se llame al método **findAll()** en el mock, se devolverá una copia de la lista de personajes definida en el **setUp()**.
* **ResponseEntity<List<Personaje>> responseEntity = personajeControlador.getTodosPersonajes();**: Se llama al método **getTodosPersonajes()** en el controlador de personajes para obtener una respuesta HTTP que contiene una lista de personajes.
* **assertAll(...)**: Este método de **Assertions** se utiliza para realizar múltiples aserciones al mismo tiempo. Aquí se verifica que la respuesta no sea nula y que el código de estado de la respuesta sea **HttpStatus.OK**.
* **verify(personajesRepositorio, times(1)).findAll();**: Se verifica que el método **findAll()** en el mock **personajesRepositorio** se haya llamado exactamente una vez durante la ejecución del método **getTodosPersonajes()**.

Ejecuto JUnit y no hay error, pero hay fallo, en la descripción dice que hay muchas actuaciones de findAll(), y dice exactamente dónde está el error:



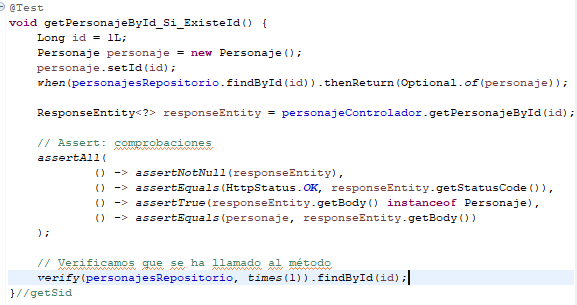
Por lo que reviso el código y es cierto que la línea **personajesRepositorio.findAll()** la estoy aplicando, además de en el propio método, en el void setUp, que con la anotación BeforeEach ordenamos que se ejecute antes de cada prueba, por lo que elimino esta línea del método dejándolo así, y ejecuto test, esta vez sale perfecto:

  
*\*\* Este método ha costado mucho adaptarlo a mi api, muchos fallos incluso al poner algún import (con los assert e incluso con un asList). Pues la api en la que se basa el tutorial es muy diferente a la mía.*

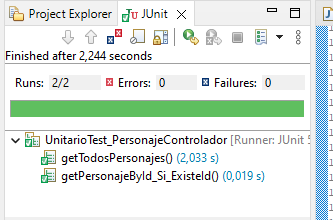
*No he copiado las capturas de todos los fallos y de las demás versiones de código porque he ido haciendo muchos cambios sobre la marcha y no quería perder el hilo.*

La prueba del método getPersonajeById lo he separado en dos pruebas, una en el caso de que el id exista, y otra cuando el id no exista.

**Test del método getPersonajeById\_Si\_ExisteId():**

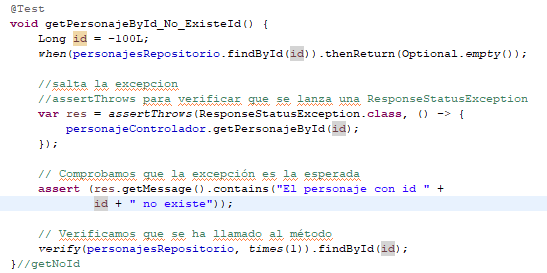
****

* **@Test**: Esta anotación indica que es un método de prueba.
* **Long id = 1L;**: Se define un ID para el personaje que se utilizará en la prueba.
* **Personaje personaje = new Personaje();**: Se crea un nuevo objeto **Personaje** que se utilizará como resultado de la búsqueda en el repositorio.
* **personaje.setId(id);**: Se establece el ID del personaje creado anteriormente.
* **when(personajesRepositorio.findById(id)).thenReturn(Optional.of(personaje));**: Aquí se configura el comportamiento del mock **personajesRepositorio**. Se especifica que cuando se llame al método **findById()** con el ID definido, se devolverá un **Optional** que contiene el personaje creado.
* **ResponseEntity<?> responseEntity = personajeControlador.getPersonajeById(id);**: Se llama al método **getPersonajeById()** en el controlador de personajes para obtener la respuesta HTTP que contiene el personaje.
* **assertAll(...)**: Se realizan múltiples aserciones al mismo tiempo para verificar que la respuesta no sea nula, que el código de estado de la respuesta sea **HttpStatus.OK**, que el cuerpo de la respuesta sea una instancia de **Personaje** y que el personaje devuelto sea igual al personaje creado anteriormente.
* **verify(personajesRepositorio, times(1)).findById(id);**: Se verifica que el método **findById()** en el mock **personajesRepositorio** se haya llamado exactamente una vez durante la ejecución del método **getPersonajeById()**.

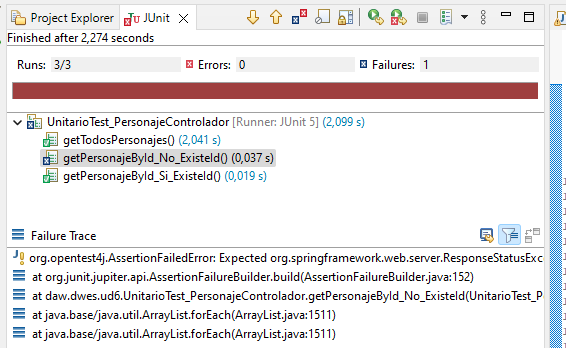
Probamos y sale correcto:  


*\*\* Esta prueba ha sido más sencilla de adaptar, mezclando el ejemplo del tutorial con el código creado del método test anterior.*

**Test método getPersonajeById\_No\_ExisteId**



Ejecuto JUnit test y hay fallo (no error) en este método.



El error que sale es que indica que no se lanzó la excepción esperada en el método **getPersonajeById\_No\_ExisteId**. Al utilizar **assertThrows** verificamos que se lanza una **ResponseStatusException**, pero parece que no se está lanzando ninguna excepción.

Para solucionar este problema, hay que asegurarse de que el método **getPersonajeById** del controlador lance una **ResponseStatusException** cuando no se encuentre el personaje con el ID indicado.

Por lo que voy al código del controlador y voy a configurar el método **getPersonajeById** para lanzar la excepción cuando sea necesario.

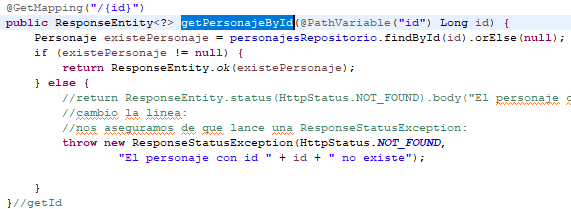
Cambio la línea del return:



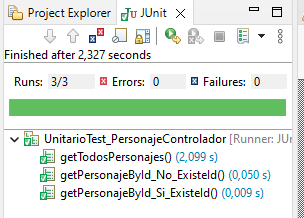
Por esta con throw:



Asi que el método **getPersonajeById** del controlador quedaría así:



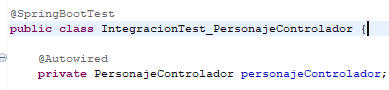
Ahora repito JUnit test y sale correcto, ya podemos ver que los tres test aplicados funcionan correctamente:



**TEST DE INTEGRACIÓN**

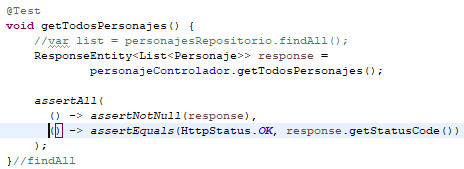
Estos tests se encargan de probar que las distintas unidades de código funcionan correctamente cuando se integran entre ellas. Para ello se suelen utilizar bases de datos en memoria para simular el acceso a datos.

Como vamos a hacer si queremos hacer los test de integración con Spring Boot, debemos usar la anotación @SpringBootTest en la clase de test. De esta manera Spring Boot se encargará de inicializar el contexto de la aplicación y de inyectar las dependencias que necesitemos.



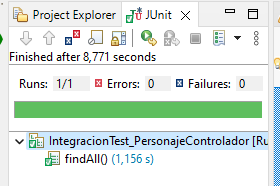
* **@Autowired**: Esta anotación hace que Spring se encargue de crear una instancia del **PersonajeControlador** y asignarla a esta variable automáticamente antes de ejecutar las pruebas.
* **private PersonajeControlador personajeControlador;**: Declaración de la variable **personajeControlador**, que contendrá la instancia del controlador que se inyectará.

**Test del método getTodosPersonajes:**

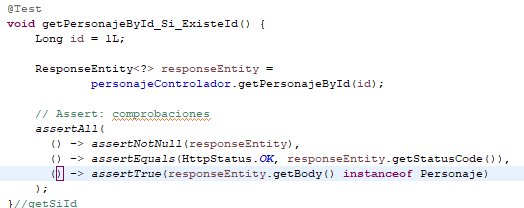
****

* **@Test**: marca el método como una prueba.
* **ResponseEntity<List<Personaje>> response = personajeControlador.getTodosPersonajes();**: llama al método **getTodosPersonajes()** del controlador. El resultado se almacena en una variable **response**, que es de tipo **ResponseEntity<List<Personaje>>**.
* **assertAll(...)**:aserción que se asegura de que las afirmaciones dentro de ella sean verdaderas.
* **assertNotNull(response)**: comprueba si la variable **response** no es nula.
* **assertEquals(HttpStatus.OK, response.getStatusCode())**:verifica si el estado de la respuesta (**response.getStatusCode()**) es igual a **HttpStatus.OK**.

Ejecuto JUnit test y sale correcto:

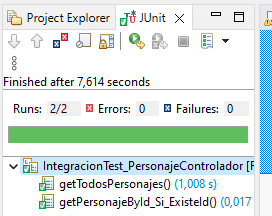


**Test del método getPersonajeById\_Si\_ExisteId:**

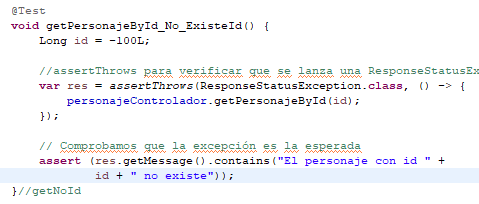
****

* **Long id = 1L;**: Se define un ID para un personaje existente.
* **ResponseEntity<?> responseEntity = personajeControlador.getPersonajeById(id);**: Se llama al método **getPersonajeById** del controlador para obtener un personaje por su ID. El resultado de se almacena en una variable **responseEntity**, que es de tipo **ResponseEntity<?>**.
* **assertNotNull(responseEntity)**: verifica si la variable **responseEntity** no es nula o si se ha recibido una respuesta del controlador.
* **assertEquals(HttpStatus.OK, responseEntity.getStatusCode())**: asegura que la respuesta del controlador sea exitosa (código de estado HTTP 200).
* **assertTrue(responseEntity.getBody() instanceof Personaje)**: verifica si el cuerpo de la respuesta es una instancia de la clase **Personaje**. Esto garantiza que el controlador haya devuelto un objeto de tipo **Personaje**.

Ejecuto JUnit test y es correcto:

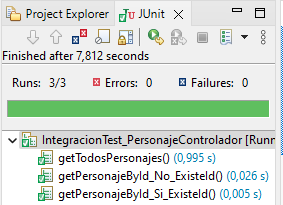


**Test método getPersonajeById\_No\_ExisteId:**



* **assertThrows(ResponseStatusException.class, () -> { personajeControlador.getPersonajeById(id); });**: Esta línea verifica que se lance una excepción de tipo **ResponseStatusException** cuando se intenta obtener un personaje con un ID que no existe.
* **var res = ...**: La excepción que se espera que se lance se captura en la variable **res**.
* **assert (res.getMessage().contains("El personaje con id " + id + " no existe"));**: Esta línea verifica si el mensaje de la excepción contiene el texto "El personaje con id [ID] no existe". Esto asegura que la excepción lanzada corresponda a la situación esperada de que el personaje no exista.

Ejecuto JUnit test y es correcto, y ahora podemos observar los test de los tres métodos aplicados:



**CONCLUSIONES**

Sobre los fallos y correcciones lo he ido comentando a lo largo de todo el documento, ya que lo he ido realizando simultáneamente con las prácticas.

Sobre lo aprendido, he aprendido muchísimo con estas prácticas, primero a usar y sabér qué es JSon Server y compararlo con Spring.

A manejas mis propias solicitudes HTTP par mi propia api según mis reglas y condiciones, e ir comprobando las respuestas en postman.

Me ha gustado mucho poder personalizar el código de estado con ResponseEntity.status y sobre todo el mensaje de respuesta con la función .body.

Reconozco que ha sido una práctica dura en el sentido de todo el tiempo dedicado tanto en clase como en casa, pero que me ha permitido aprender y practicar mucho y con la que acabo muy satisfecha.