

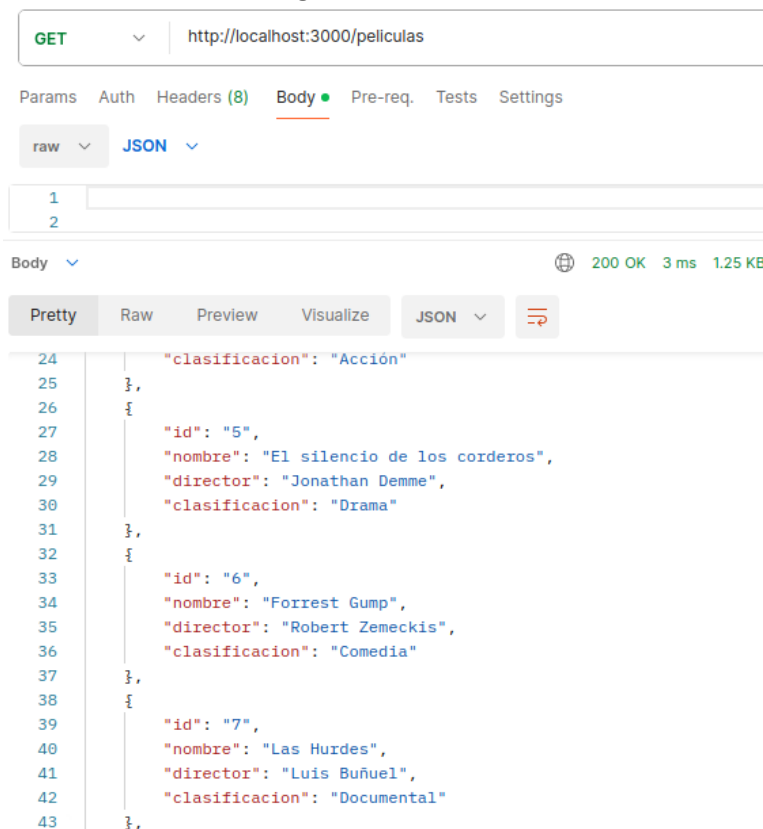
Tarea 1 – UD6

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 ⌵ http://localhost:3000/peliculas

Params Auth Headers (8) **Body** ● Pre-req. Tests Settings

raw ⌵ **JSON** ⌵

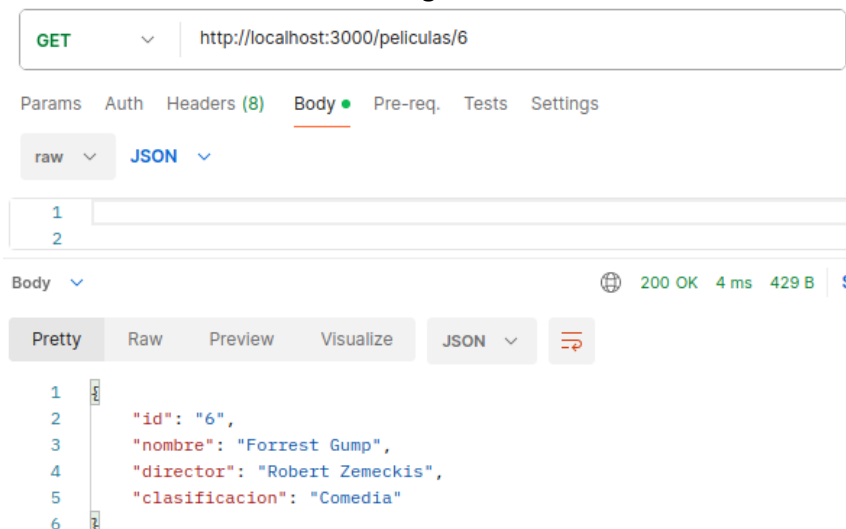
1
2

Body ⌵ 200 OK 3 ms 1.25 KB

Pretty Raw Preview Visualize **JSON** ⌵ 

```
24   "clasificacion": "Acción"
25   },
26   {
27     "id": "5",
28     "nombre": "El silencio de los corderos",
29     "director": "Jonathan Demme",
30     "clasificacion": "Drama"
31   },
32   {
33     "id": "6",
34     "nombre": "Forrest Gump",
35     "director": "Robert Zemeckis",
36     "clasificacion": "Comedia"
37   },
38   {
39     "id": "7",
40     "nombre": "Las Hurdes",
41     "director": "Luis Buñuel",
42     "clasificacion": "Documental"
43   },
44   }
```

GET de una entidad existente: código 200 ok




GET ⌵ http://localhost:3000/peliculas/6

Params Auth Headers (8) **Body** ● Pre-req. Tests Settings

raw ⌵ **JSON** ⌵

1
2

Body ⌵ 200 OK 4 ms 429 B **S**

Pretty Raw Preview Visualize **JSON** ⌵ 

```
1   {
2     "id": "6",
3     "nombre": "Forrest Gump",
4     "director": "Robert Zemeckis",
5     "clasificacion": "Comedia"
6   }
```

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

GET http://localhost:3000/peliculas/10

Params Auth Headers (8) Body Pre-req. Tests Settings

raw JSON

1
2

Body 404 Not Found 6 ms 387 B

Pretty Raw Preview Visualize Text

1 Not Found

POST de una entidad nueva: código 201 created.

POST http://localhost:3000/peliculas

Params Auth Headers (8) Body Pre-req. Tests Settings

raw JSON

```
1 {
2   "id": "9",
3   "nombre": "Criadas y Señoras",
4   "director": "Tate Taylor",
5   "clasificacion": "Drama"
6 }
```

Body 201 Created 9 ms 434 B

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": "9",
3   "nombre": "Criadas y Señoras",
4   "director": "Tate Taylor",
5   "clasificacion": "Drama"
6 }
```

POST de una entidad existente:

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

POST http://localhost:3000/peliculas/5

Params Auth Headers (8) Body Pre-req. Tests Settings

raw JSON

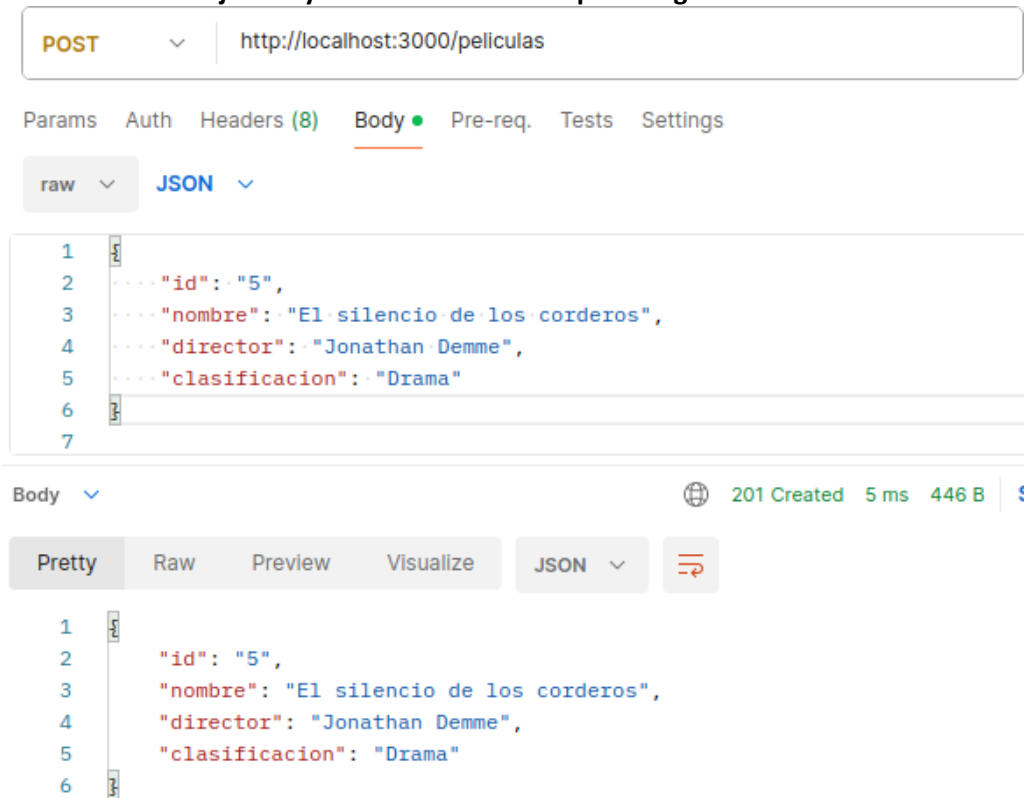
1
2

Body 500 Internal Server Error 3 ms 345 B

Pretty Raw Preview Visualize Text

1 Unexpected end of JSON input

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



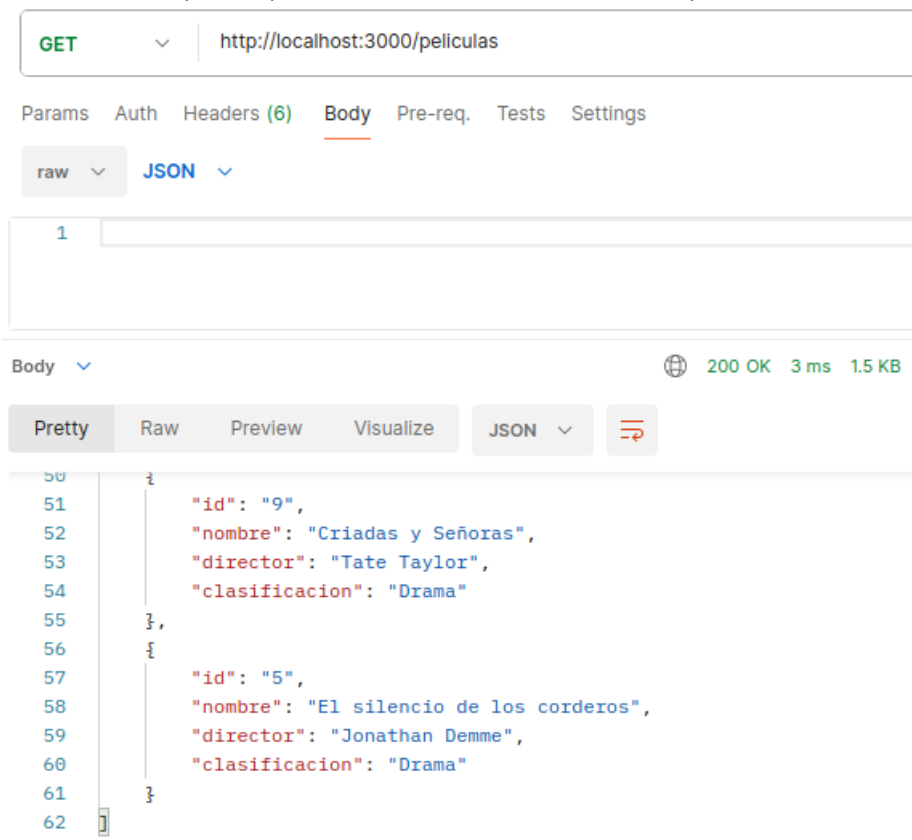
The screenshot shows a REST client interface with a POST request to `http://localhost:3000/peliculas`. The request body is a JSON object:

```
{
  "id": "5",
  "nombre": "El silencio de los corderos",
  "director": "Jonathan Demme",
  "clasificacion": "Drama"
}
```

The response status is **201 Created** with a response time of 5 ms and a body size of 446 B. The response body is shown in the 'Pretty' view:

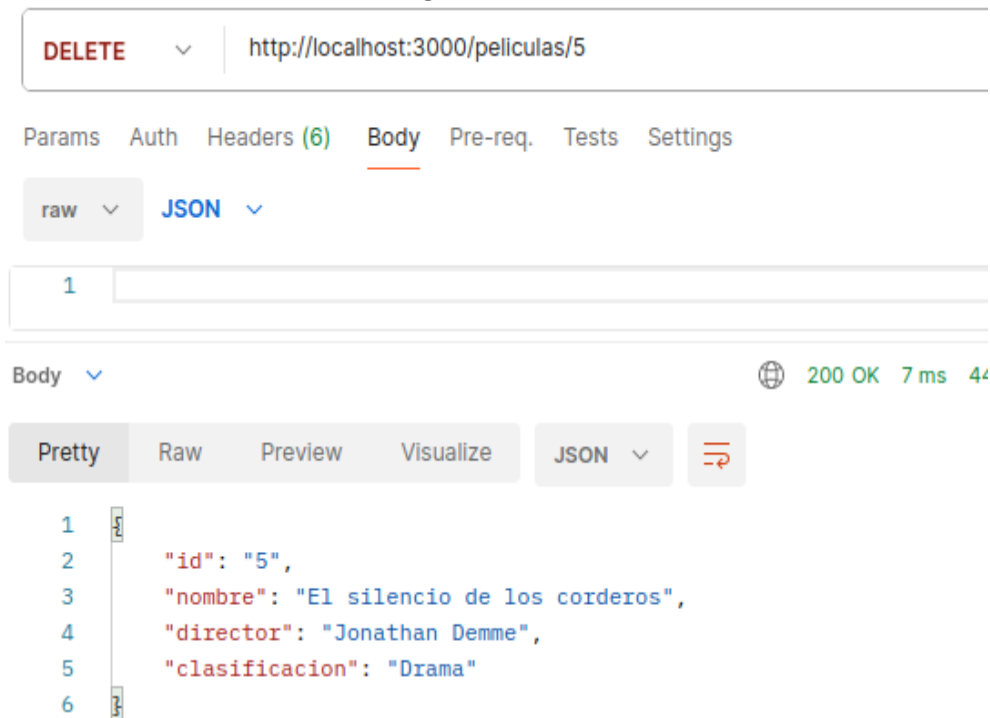
```
{
  "id": "5",
  "nombre": "El silencio de los corderos",
  "director": "Jonathan Demme",
  "clasificacion": "Drama"
}
```

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.



The screenshot shows a REST client interface with a GET request to `http://localhost:3000/peliculas`. The response status is **200 OK** with a response time of 3 ms and a body size of 1.5 KB. The response body is shown in the 'Pretty' view:

```
[
  {
    "id": "9",
    "nombre": "Criadas y Señoras",
    "director": "Tate Taylor",
    "clasificacion": "Drama"
  },
  {
    "id": "5",
    "nombre": "El silencio de los corderos",
    "director": "Jonathan Demme",
    "clasificacion": "Drama"
  }
]
```

DEL de una entidad existente: código 201 ok.

DELETE

Params Auth Headers (6) Body Pre-req. Tests Settings

raw

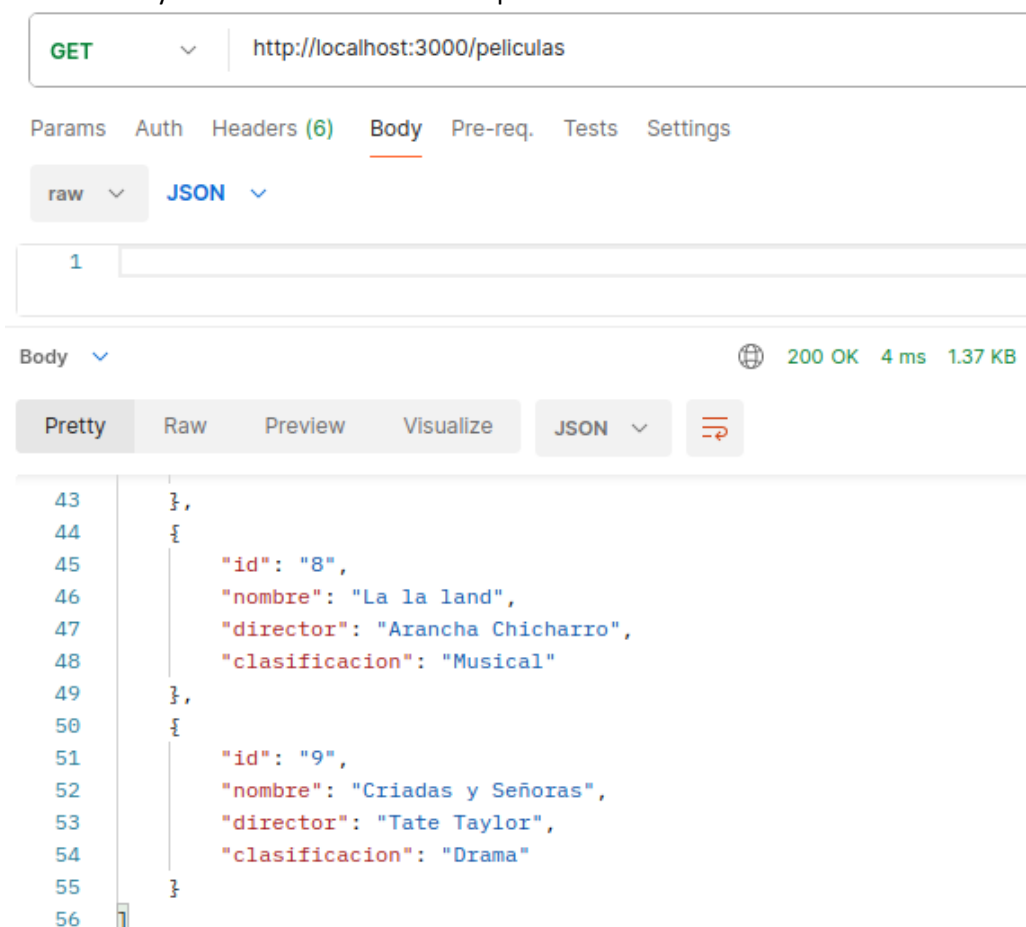
1

Body

Pretty Raw Preview Visualize

```
1 {
2   "id": "5",
3   "nombre": "El silencio de los corderos",
4   "director": "Jonathan Demme",
5   "clasificacion": "Drama"
6 }
```

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



GET

Params Auth Headers (6) Body Pre-req. Tests Settings

raw

1

Body

Pretty Raw Preview Visualize

```
43 },
44 {
45   "id": "8",
46   "nombre": "La la land",
47   "director": "Arancha Chicharro",
48   "clasificacion": "Musical"
49 },
50 {
51   "id": "9",
52   "nombre": "Criadas y Señoras",
53   "director": "Tate Taylor",
54   "clasificacion": "Drama"
55 }
56 ]
```

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

DELETE ▼ http://localhost:3000/peliculas/12

Params Auth Headers (6) Body Pre-req. Tests Settings

raw ▼ JSON ▼

1

Body ▼ 404 Not Found 6 ms 38

Pretty Raw Preview Visualize Text ▼

1 Not Found

PUT de una entidad existente: código 200 ok.

He actualizado el director de la película con id 8.

PUT ▼ http://localhost:3000/peliculas/8

Params Auth Headers (8) Body ● Pre-req. Tests Settings

raw ▼ JSON ▼

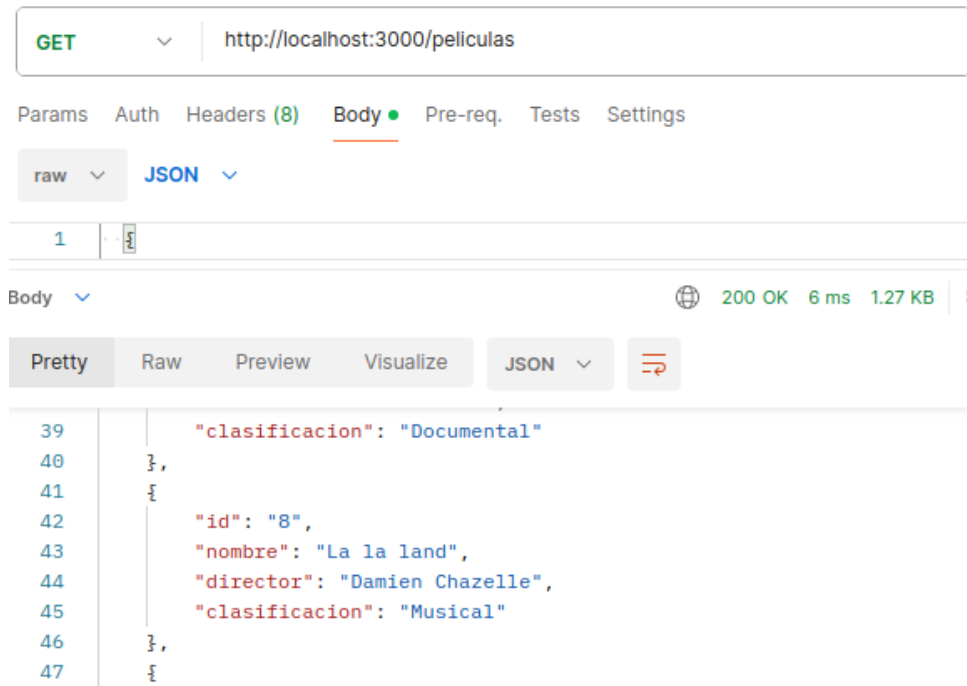
```
1 {
2   "id": "8",
3   "nombre": "La la land",
4   "director": "Damien Chazelle",
5   "clasificacion": "Musical"
6 }
```

Body ▼ 200 OK 4 ms 427 B

Pretty Raw Preview Visualize JSON ▼

```
1 {
2   "id": "8",
3   "nombre": "La la land",
4   "director": "Damien Chazelle",
5   "clasificacion": "Musical"
6 }
```

Lo compruebo en la lista nuevamente:



GET http://localhost:3000/peliculas

Params Auth Headers (8) Body ● Pre-req. Tests Settings

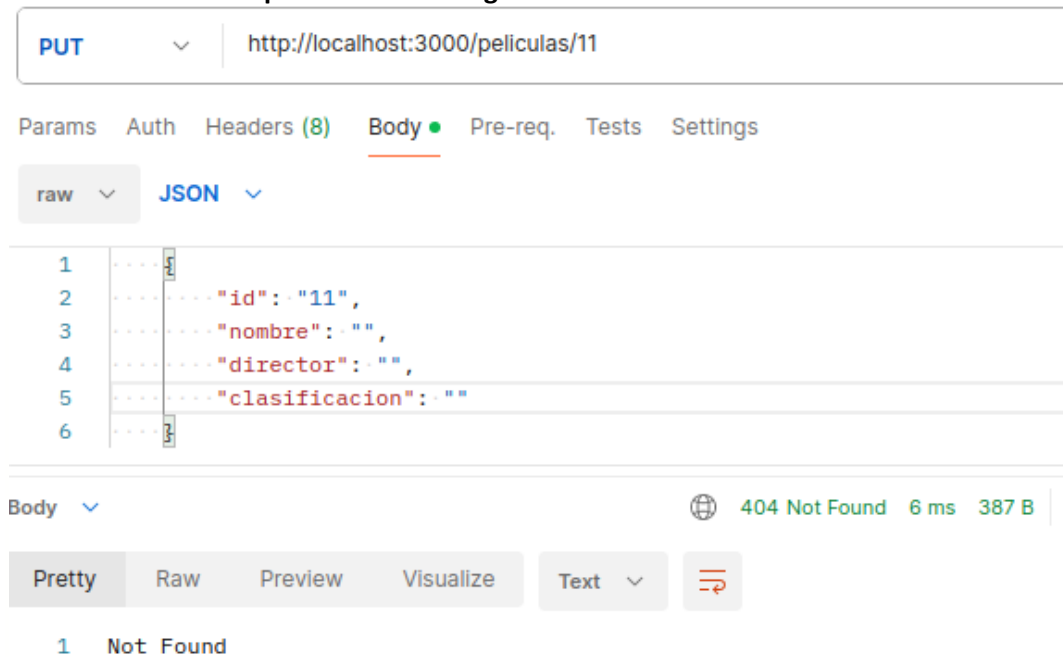
raw JSON

1 {

Body 200 OK 6 ms 1.27 KB

Pretty Raw Preview Visualize JSON

```
39   "clasificacion": "Documental"
40 },
41 {
42   "id": "8",
43   "nombre": "La la land",
44   "director": "Damien Chazelle",
45   "clasificacion": "Musical"
46 },
47 }
```

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

PUT http://localhost:3000/peliculas/11

Params Auth Headers (8) Body ● Pre-req. Tests Settings

raw JSON

```
1 {
2   "id": "11",
3   "nombre": "",
4   "director": "",
5   "clasificacion": ""
6 }
```

Body 404 Not Found 6 ms 387 B

Pretty Raw Preview Visualize Text

1 Not Found

PATCH de una entidad existente: código 200 ok.

Pruebo a modificar el campo clasificación del elemento 9.

REST client interface showing a PATCH request to `http://localhost:3000/peliculas/9`. The request body is a JSON object:

```
{  "id": "9",  "nombre": "Criadas y Señoras",  "director": "Tate Taylor",  "clasificacion": "Comedia"}
```

The response is `200 OK` with a status of `3 ms`.

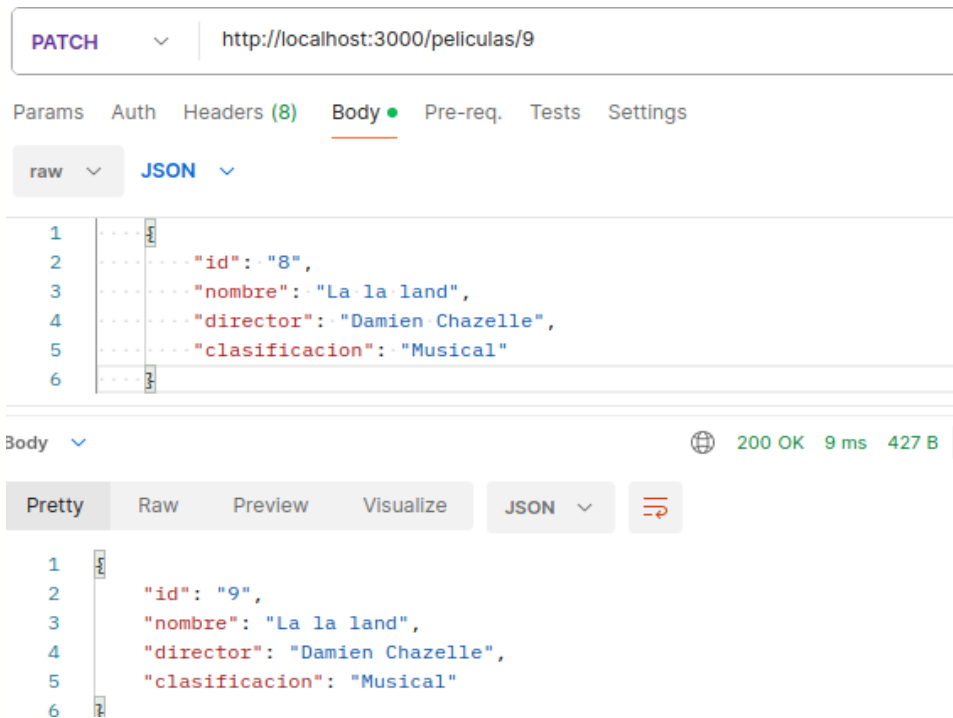
Lo compruebo visualizando la lista:

REST client interface showing a GET request to `http://localhost:3000/peliculas`. The response is a JSON array of two movie objects:

```
[  {    "id": "8",    "nombre": "La la land",    "director": "Damien Chazelle",    "clasificacion": "Musical"  },  {    "id": "9",    "nombre": "Criadas y Señoras",    "director": "Tate Taylor",    "clasificacion": "Comedia"  }]
```

The response is `200 OK` with a status of `3 ms`.

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



PATCH ▼ http://localhost:3000/peliculas/9

Params Auth Headers (8) **Body** ● Pre-req. Tests Settings

raw ▼ **JSON** ▼

```
1  {
2    "id": "8",
3    "nombre": "La la land",
4    "director": "Damien Chazelle",
5    "clasificacion": "Musical"
6  }
```

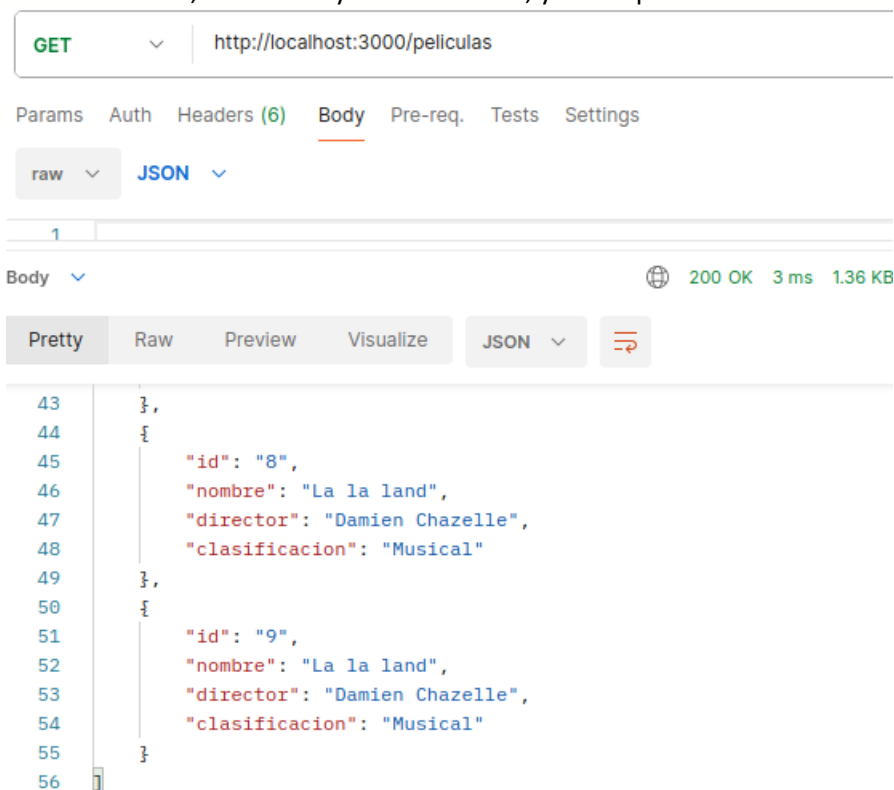
Body ▼ 🌐 200 OK 9 ms 427 B

Pretty Raw Preview Visualize **JSON** ▼ ≡

```
1  {
2    "id": "9",
3    "nombre": "La la land",
4    "director": "Damien Chazelle",
5    "clasificacion": "Musical"
6  }
```

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):



GET ▼ http://localhost:3000/peliculas

Params Auth Headers (6) **Body** Pre-req. Tests Settings

raw ▼ **JSON** ▼

1

Body ▼ 🌐 200 OK 3 ms 1.36 KB

Pretty Raw Preview Visualize **JSON** ▼ ≡

```
43  },
44  {
45    "id": "8",
46    "nombre": "La la land",
47    "director": "Damien Chazelle",
48    "clasificacion": "Musical"
49  },
50  {
51    "id": "9",
52    "nombre": "La la land",
53    "director": "Damien Chazelle",
54    "clasificacion": "Musical"
55  }
56  ]
```

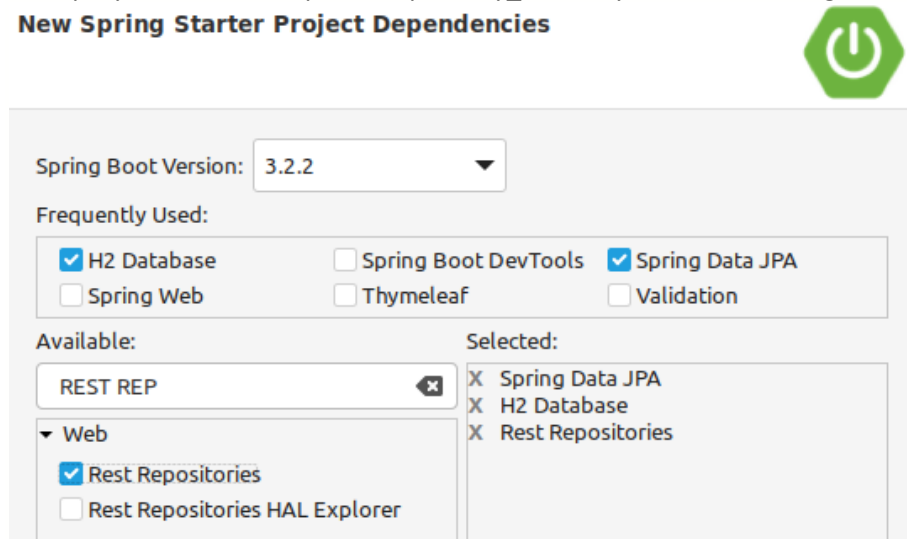

Apartado 2.- API con REST Repository.

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:

New Spring Starter Project Dependencies



Spring Boot Version: 3.2.2

Frequently Used:

- ☒ H2 Database
- ☐ Spring Boot DevTools
- ☒ Spring Data JPA
- ☐ Spring Web
- ☐ Thymeleaf
- ☐ Validation

Available:

- REST REP
- Web
 - ☒ Rest Repositories
 - ☐ Rest Repositories HAL Explorer

Selected:

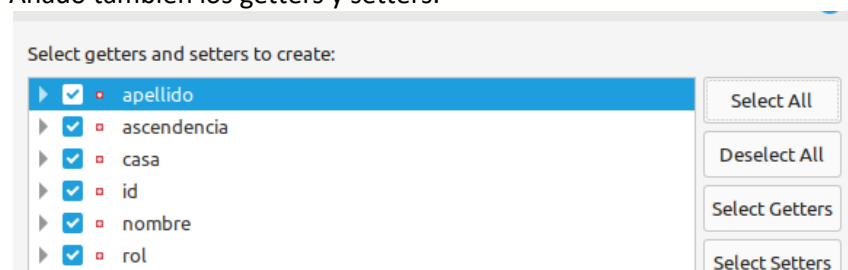
- X Spring Data JPA
- X H2 Database
- X Rest Repositories

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:

```
QuoteController.java ComsumoApiRestApplication.java *Personaje.java
1 package daw.dwes.ud6;
2
3 import jakarta.persistence.Entity;
4 import jakarta.persistence.GeneratedValue;
5 import jakarta.persistence.GenerationType;
6 import jakarta.persistence.Id;
7
8 @Entity
9 public class Personaje {
10
11     @Id
12     @GeneratedValue(strategy = GenerationType.AUTO)
13     private long id;
14
15     private String nombre;
16     private String apellido;
17     private String rol;
18     private String casa;
19     private String ascendencia;
```

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

Añado también los getters y setters.



Select getters and setters to create:

- ☒ apellido
- ☒ ascendencia
- ☒ casa
- ☒ id
- ☒ nombre
- ☒ rol

Select All

Deselect All

Select Getters

Select Setters

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

```

QuoteController.java  ConsumoApiRestApplicatio  *Personaje.java  *PersonajesRepositorio.j
1  package daw.dwes.ud6;
2
3  import java.util.List;
4
5  import org.springframework.data.repository.CrudRepository;
6  import org.springframework.data.repository.PagingAndSortingRepository;
7  import org.springframework.data.repository.query.Param;
8  import org.springframework.data.rest.core.annotation.RepositoryRestResource;
9
10 @RepositoryRestResource(collectionResourceRel = "personajes", path = "personajes")
11 public interface PersonajesRepositorio extends
12     PagingAndSortingRepository<Personaje, Long>,
13     CrudRepository<Personaje, Long>{
14
15     List<Personaje> findByNombreCompleto(@Param("nombre") String nombre);
16
17 }
```

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.**

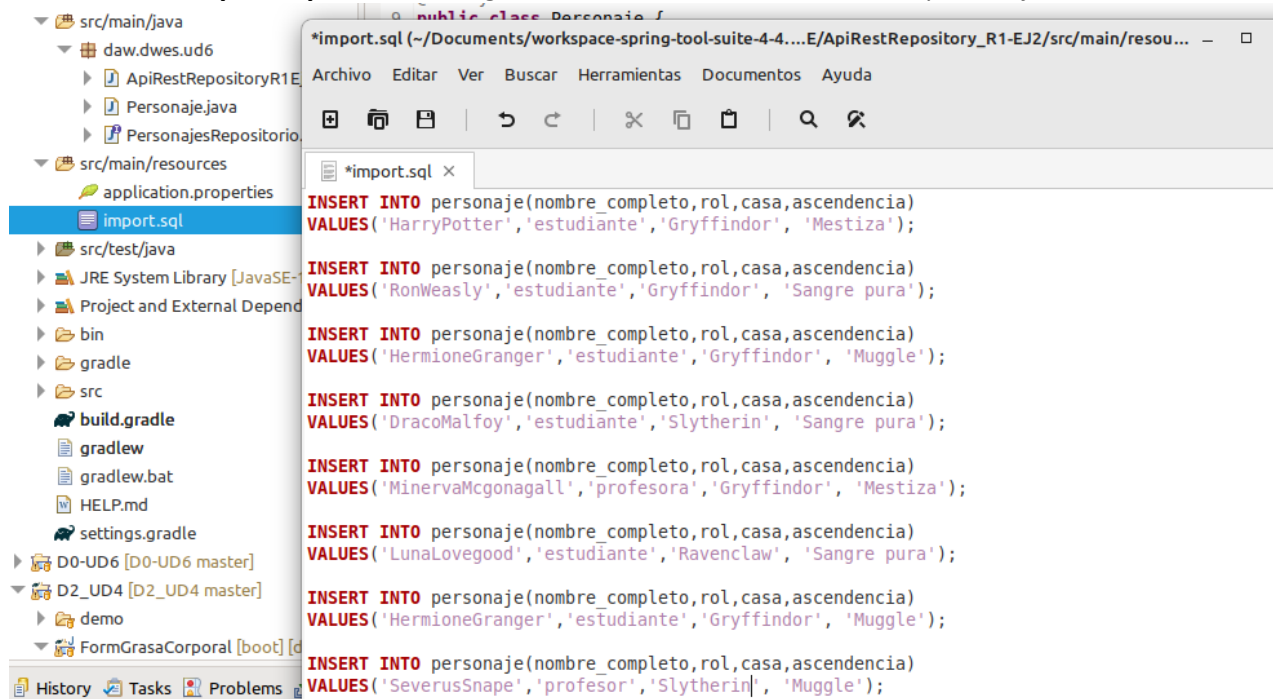
Antes de ello voy a eliminar el **id**, para que lo autogenera la bbdd, para ello en la clase Personaje, en el campo del id, en lugar de **AUTO**, ponemos **IDENTITY**.

```

*Personaje.java x PersonajesRepositorio.java ApiRestRepositoryR1E
6 import jakarta.persistence.Id;
7
8 @Entity
9 public class Personaje {
10
11     @Id
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private long id;
14
15     private String nombreCompleto;

```

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



```

*import.sql x
INSERT INTO personaje(nombre_completo,rol,casa,ascendencia)
VALUES('HarryPotter','estudiante','Gryffindor', 'Mestiza');

INSERT INTO personaje(nombre_completo,rol,casa,ascendencia)
VALUES('RonWeasley','estudiante','Gryffindor', 'Sangre pura');

INSERT INTO personaje(nombre_completo,rol,casa,ascendencia)
VALUES('HermioneGranger','estudiante','Gryffindor', 'Muggle');

INSERT INTO personaje(nombre_completo,rol,casa,ascendencia)
VALUES('DracoMalfoy','estudiante','Slytherin', 'Sangre pura');

INSERT INTO personaje(nombre_completo,rol,casa,ascendencia)
VALUES('MinervaMcgonagall','profesora','Gryffindor', 'Mestiza');

INSERT INTO personaje(nombre_completo,rol,casa,ascendencia)
VALUES('LunaLovegood','estudiante','Ravenclaw', 'Sangre pura');

INSERT INTO personaje(nombre_completo,rol,casa,ascendencia)
VALUES('HermioneGranger','estudiante','Gryffindor', 'Muggle');

INSERT INTO personaje(nombre_completo,rol,casa,ascendencia)
VALUES('SeverusSnape','profesor','Slytherin', 'Muggle');

```

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**:

```

11 @Id
12 @GeneratedValue(strategy = GenerationType.IDENTITY)
13 private long id;
14
15 private String nombre;
16 private String rol;
17 private String casa;
18 private String ascendencia;

```

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

```

10 @RepositoryRestResource(collectionResourceRel = "personajes", path = "personajes")
11 public interface PersonajesRepositorio extends
12     PagingAndSortingRepository<Personaje, Long>,
13     CrudRepository<Personaje, Long>{
14
15     List<Personaje> findByNombre(@Param("nombre") String nombre);
16

```

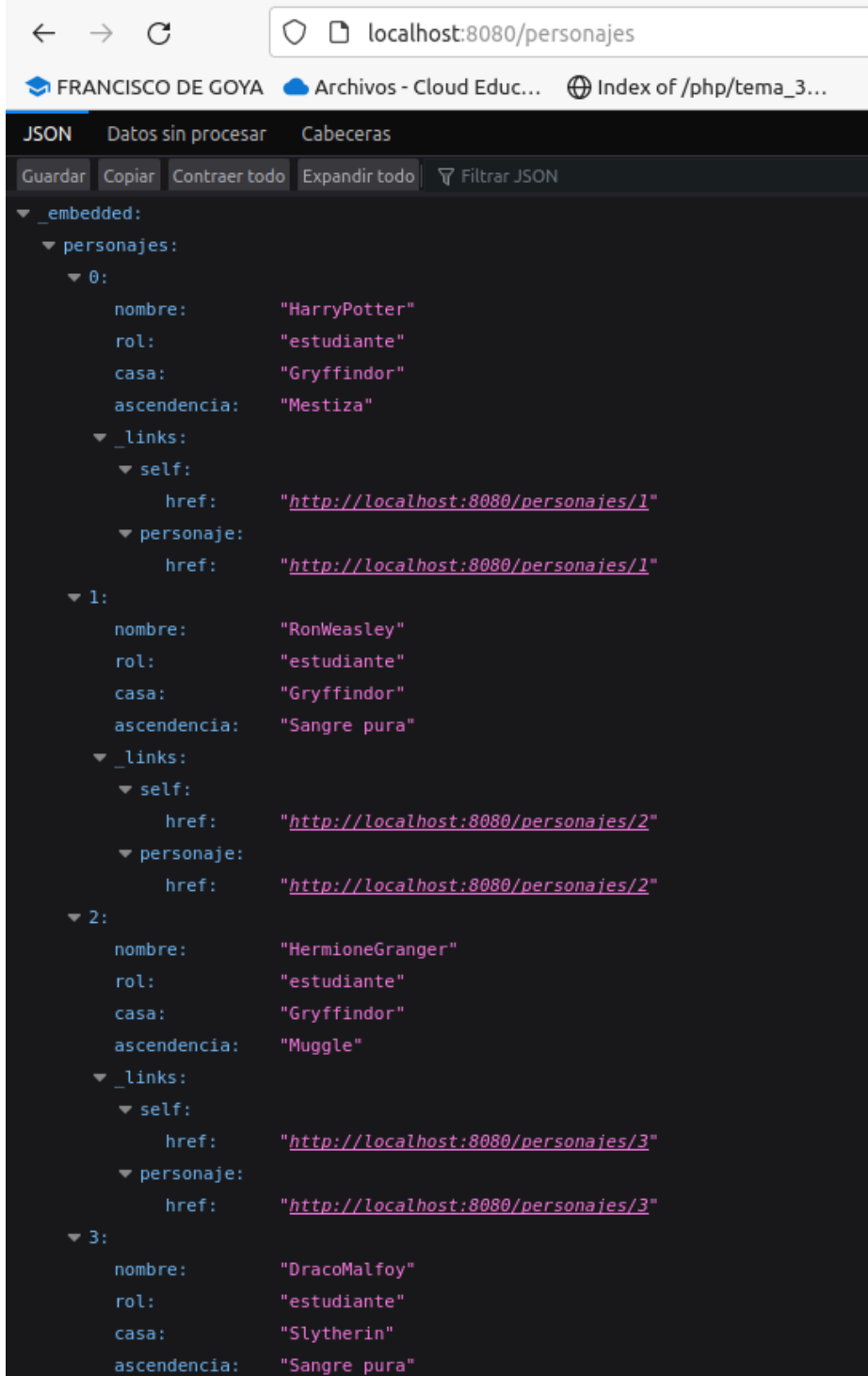
Y ejecutamos nuevamente. Ningún error en consola.

```

2024-01-24T10:07:05.300+01:00 INFO 14256 --- [ restartedMain] o.s.b.a.s.spring.factories.entitymanagerfactory.NoDataEntityManagerFactoryBean : HH0000489: N
2024-01-24T10:07:05.852+01:00 INFO 14256 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized
2024-01-24T10:07:05.882+01:00 WARN 14256 --- [ restartedMain] JpaBaseConfigurationsJpaWebConfiguration : spring.jpa.c
2024-01-24T10:07:06.374+01:00 INFO 14256 --- [ restartedMain] o.s.b.a.o.OptionalLiveReloadServer : LiveReload s
2024-01-24T10:07:06.396+01:00 INFO 14256 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat starte
2024-01-24T10:07:06.404+01:00 INFO 14256 --- [ restartedMain] d.s.u.ApiRestRepositoryYRIE2Application : Started Apif
2024-01-24T10:07:21.819+01:00 INFO 14256 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[/localhost./] : Initializing
2024-01-24T10:07:21.819+01:00 INFO 14256 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing
2024-01-24T10:07:21.821+01:00 INFO 14256 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed 100

```

Y probamos en el navegador:



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

```

Root WebApplicationContext: initialization completed in 716 ms
HikariPool-1 - Starting...
HikariPool-1 - Added connection conn0: url=jdbc:h2:mem:020bc6e1-7c8e-4ab1-a4bd-e6885e43fdbf user=SA
HikariPool-1 - Start completed.
H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:020bc6e1-7c8e-4ab1-a4bd-e6885e43fdbf'
HHH000204: Processing PersistenceUnitInfo [name: default]
HHH000412: Hibernate ORM core version 6.4.1.Final
HHH000026: Second-level cache disabled
No LoadTimeWeaver setup: ignoring JPA class transformer
HHH000489: No JTA platform available (set 'hibernate.transaction.jta.platform' to enable JTA platform integration)

```

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

GET toda la lista: código 200 ok

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/personajes`. The response status is 200 OK, with a response time of 20 ms and a body size of 2.96 KB. The response body is displayed in JSON format, showing an array of two objects representing characters.

```

{
  "_embedded": {
    "personajes": [
      {
        "nombre": "HarryPotter",
        "rol": "estudiante",
        "casa": "Gryffindor",
        "ascendencia": "Mestiza",
        "_links": {
          "self": {
            "href": "http://localhost:8080/personajes/1"
          },
          "personaje": {
            "href": "http://localhost:8080/personajes/1"
          }
        }
      },
      {
        "nombre": "RonWeasley",
        "rol": "estudiante",
        "casa": "Gryffindor",
        "ascendencia": "Sangre pura",
        "_links": {
          "self": {
            "href": "http://localhost:8080/personajes/2"
          },
          "personaje": {
            "href": "http://localhost:8080/personajes/2"
          }
        }
      }
    ]
  }
}

```

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

GET

Params Auth Headers (6) Body Pre-req. Tests Settings

Query Params

Key	Value
-----	-------

Body 200 OK 7 ms 542 B

Pretty Raw Preview Visualize JSON

```
1 {
2   "nombre": "MinervaMcGonagall",
3   "rol": "profesora",
4   "casa": "Gryffindor",
5   "ascendencia": "Mestiza",
6   "_links": {
7     "self": {
8       "href": "http://localhost:8080/personajes/5"
9     },
10    "personaje": {
11      "href": "http://localhost:8080/personajes/5"
12    }
13  }
14 }
```

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

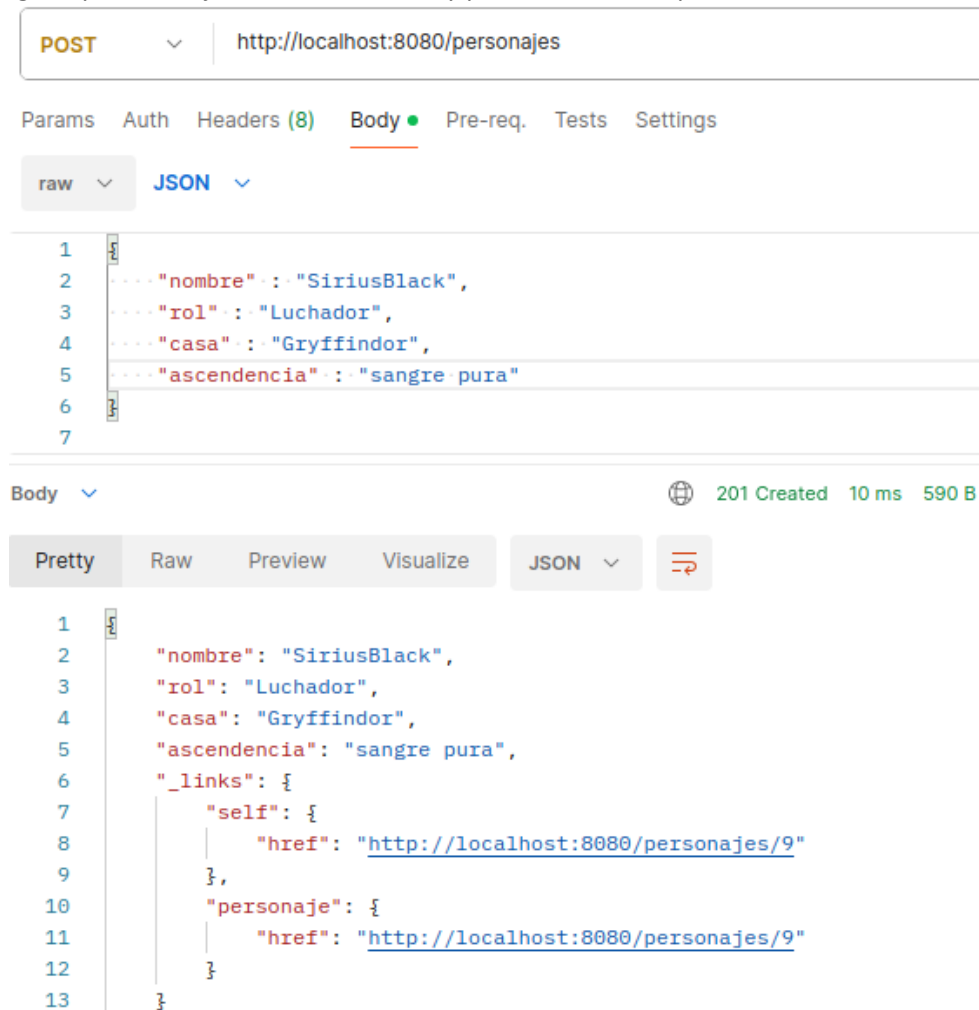
Params Auth Headers (8) Body ☒ Pre-req. Tests Settings

raw JSON

```
1 {
2   "nombre": "SiriusBlack",
3   "rol": "Luchador",
4   "casa": "Gryffindor",
5   "ascendencia": "sangre pura",
6   "_links": {
7     "self": {
8       "href": "http://localhost:8080/personajes/9"
9     },
10    "personaje": {
11      "href": "http://localhost:8080/personajes/9"
12    }
13  }
14 }
```

POST elemento existente: **201 created**.

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



POST ▼ http://localhost:8080/personajes

Params Auth Headers (8) **Body** ● Pre-req. Tests Settings

raw ▼ **JSON** ▼

```

1 {
2   "nombre": "SiriusBlack",
3   "rol": "Luchador",
4   "casa": "Gryffindor",
5   "ascendencia": "sangre pura"
6 }
7

```

Body ▼ 🌐 201 Created 10 ms 590 B

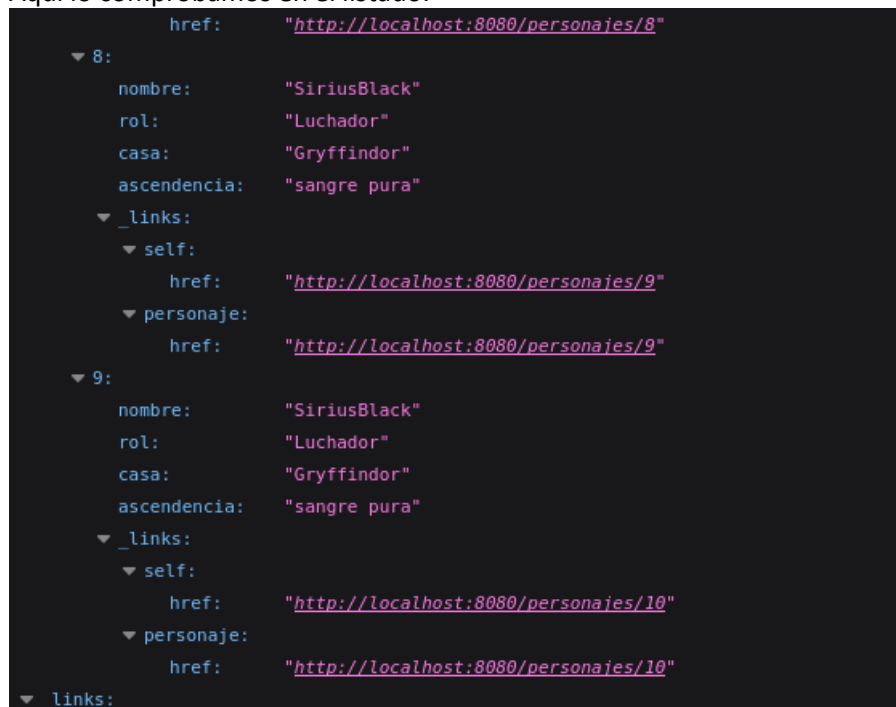
Pretty Raw Preview Visualize **JSON** ▼ ≡

```

1 {
2   "nombre": "SiriusBlack",
3   "rol": "Luchador",
4   "casa": "Gryffindor",
5   "ascendencia": "sangre pura",
6   "_links": {
7     "self": {
8       "href": "http://localhost:8080/personajes/9"
9     },
10    "personaje": {
11      "href": "http://localhost:8080/personajes/9"
12    }
13  }
14 }

```

Aquí lo comprobamos en el listado:



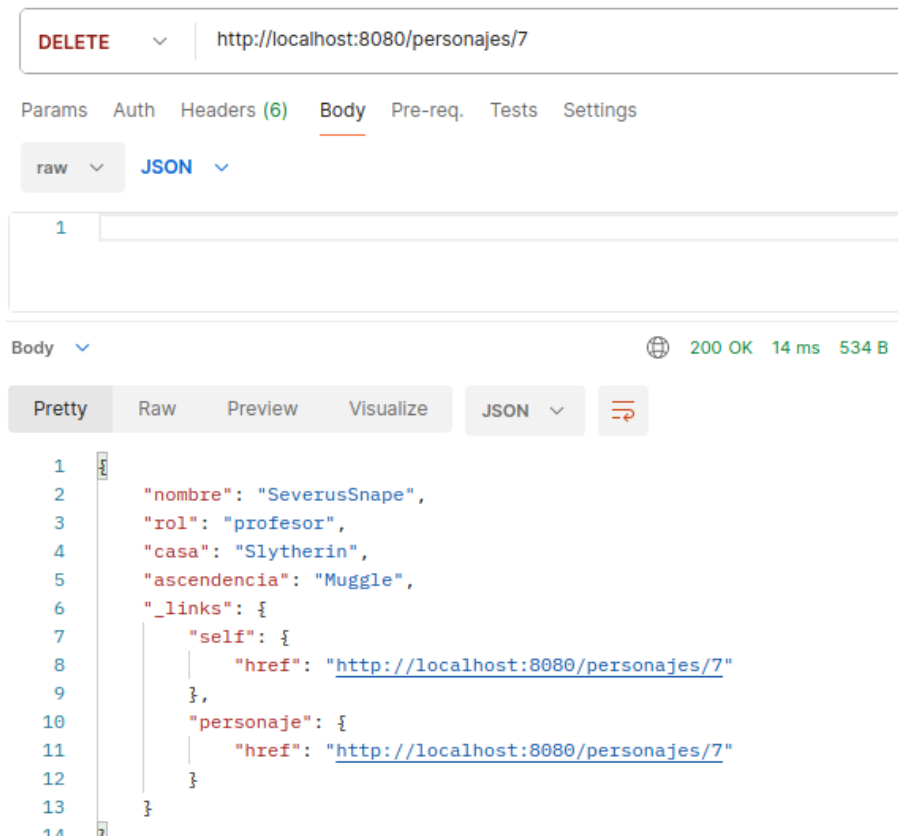
```

  href: "http://localhost:8080/personajes/8"
  ▼ 8:
    nombre: "SiriusBlack"
    rol: "Luchador"
    casa: "Gryffindor"
    ascendencia: "sangre pura"
    ▼ _links:
      ▼ self:
        href: "http://localhost:8080/personajes/9"
      ▼ personaje:
        href: "http://localhost:8080/personajes/9"
  ▼ 9:
    nombre: "SiriusBlack"
    rol: "Luchador"
    casa: "Gryffindor"
    ascendencia: "sangre pura"
    ▼ _links:
      ▼ self:
        href: "http://localhost:8080/personajes/10"
      ▼ personaje:
        href: "http://localhost:8080/personajes/10"
  ▼ _links:

```

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:




DELETE ▼ | http://localhost:8080/personajes/7

Params Auth Headers (6) Body Pre-req. Tests Settings

raw ▼ JSON ▼

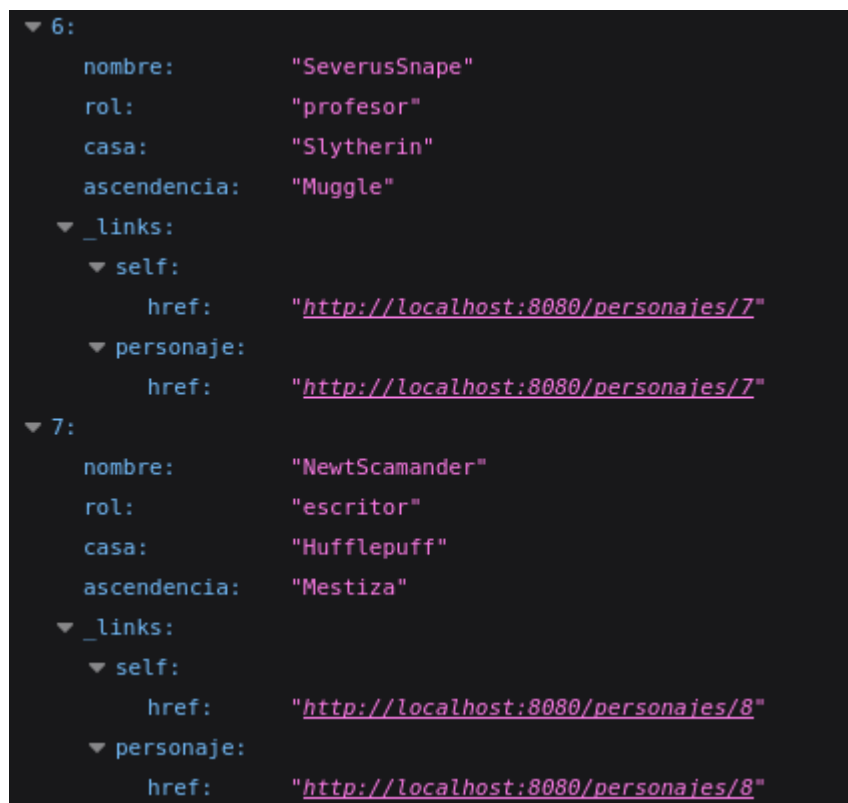
1

Body ▼ 200 OK 14 ms 534 B

Pretty Raw Preview Visualize JSON ▼ 

```

1  {
2    "nombre": "SeverusSnape",
3    "rol": "profesor",
4    "casa": "Slytherin",
5    "ascendencia": "Muggle",
6    "_links": {
7      "self": {
8        "href": "http://localhost:8080/personajes/7"
9      },
10     "personaje": {
11       "href": "http://localhost:8080/personajes/7"
12     }
13   }
14 }
```



6:

```

nombre: "SeverusSnape"
rol: "profesor"
casa: "Slytherin"
ascendencia: "Muggle"
_links:
  self:
    href: "http://localhost:8080/personajes/7"
  personaje:
    href: "http://localhost:8080/personajes/7"
7:
```

7:

```

nombre: "NewtScamander"
rol: "escritor"
casa: "Hufflepuff"
ascendencia: "Mestiza"
_links:
  self:
    href: "http://localhost:8080/personajes/8"
  personaje:
    href: "http://localhost:8080/personajes/8"
```


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

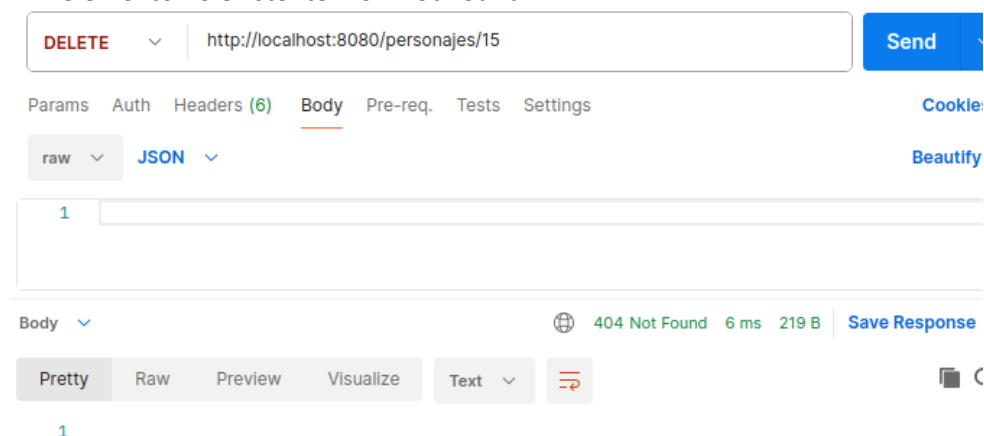
Se ha borrado un elemento, y por lo tanto la posición de los elementos lógicamente cambia, pero su id sigue siendo el mismo, por lo que hay que tener cuidado ahora en buscar los elementos por el nº del id y no por la posición, ya que no es el mismo.

Por ejemplo:

El elemento 7, SiriusBlack, 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:

```
▼ 7:
  nombre:      "SiriusBlack"
  rol:         "Luchador"
  casa:        "Gryffindor"
  ascendencia: "sangre pura"
  _links:
    ▼ self:
      href:     "http://localhost:8080/personajes/9"
    ▼ personaje:
      href:     "http://localhost:8080/personajes/9"
```

PUT ▼ http://localhost:8080/personajes/9 Send ▼

Params Auth Headers (8) **Body** ● Pre-req. Tests Settings Cookies

raw ▼ **JSON** ▼ Beautify

```
1
2
3
4
5
6
```

Body ▼ 200 OK 22 ms 585 B Save Response ▼

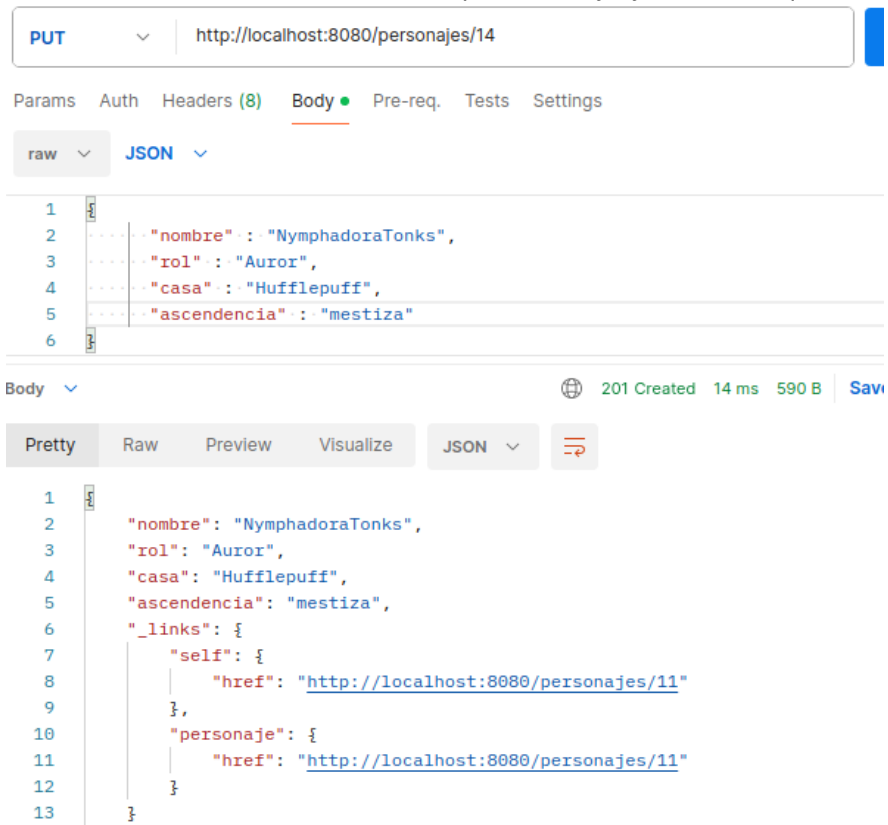
Pretty Raw Preview Visualize **JSON** ▼ 🔍

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

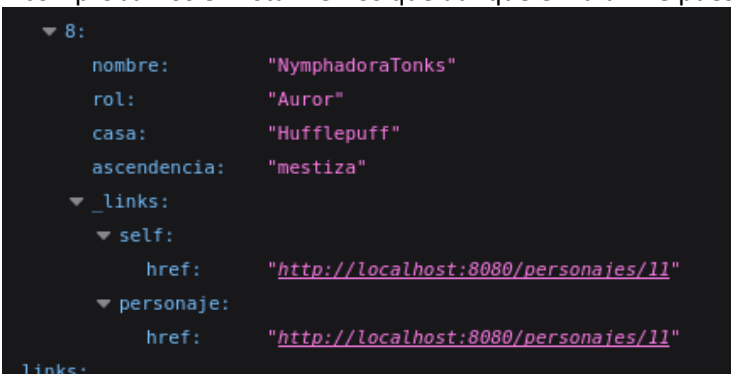
```
▼ 7:
  nombre:      "SiriusBlack"
  rol:         "Prófugo"
  casa:        "Gryffindor"
  ascendencia: "sangre pura"
  _links:
    ▼ self:
      href:     "http://localhost:8080/personajes/9"
    ▼ personaje:
      href:     "http://localhost:8080/personajes/9"
```

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

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

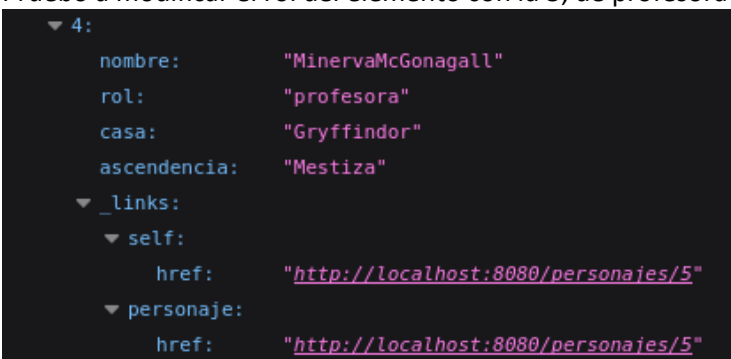


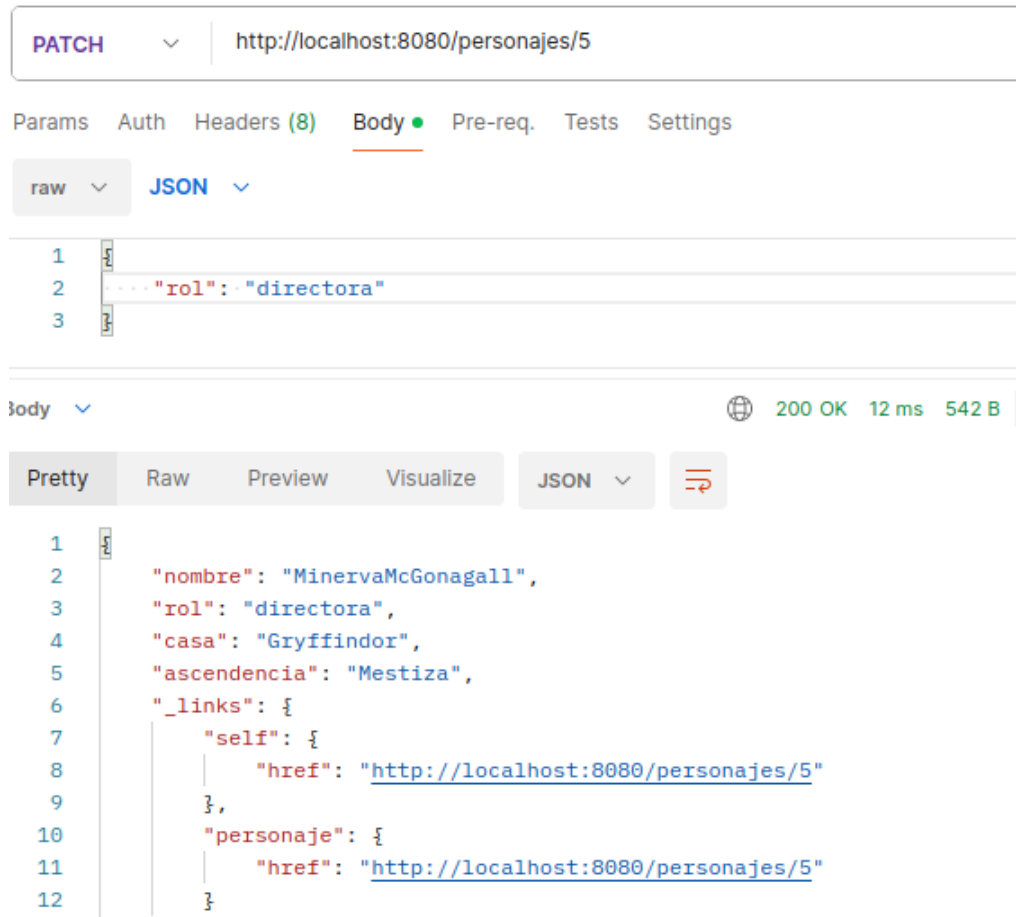
Y comprobamos en lista. Vemos que aunque en la url he puesto el id 14, el json se ha creado con el id 11.



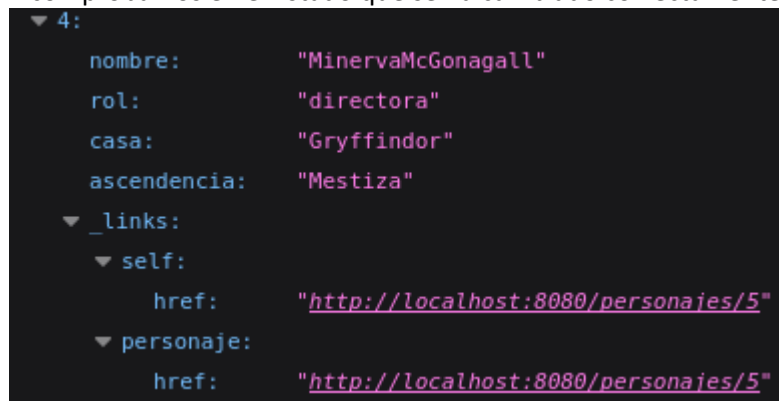
PATCH elemento existente: **código 200 ok.**

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.

REST client interface showing a PATCH request to `http://localhost:8080/personajes/4`. The request body is a JSON object:

```

1 {
2   "nombre": "MinervaMcGonagall",
3   "rol": "jefa de estudios",
4   "casa": "Gryffindor",
5   "ascendencia": "Mestiza",
6   "_links": {
7     "self": {
8       "href": "http://localhost:8080/personajes/4"
9     },
10    "personaje": {
11      "href": "http://localhost:8080/personajes/5"
12    }
13  }

```

The response is a 200 OK status with a JSON body:

```

1 {
2   "nombre": "MinervaMcGonagall",
3   "rol": "jefa de estudios",
4   "casa": "Gryffindor",
5   "ascendencia": "Mestiza",
6   "_links": {
7     "self": {
8       "href": "http://localhost:8080/personajes/4"
9     },
10    "personaje": {
11      "href": "http://localhost:8080/personajes/4"
12    }
13  }

```

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

```

▼ 3:
  nombre:      "MinervaMcGonagall"
  rol:         "jefa de estudios"
  casa:        "Gryffindor"
  ascendencia: "Mestiza"
  ▼ _links:
    ▼ self:
      href:     "http://localhost:8080/personajes/4"
    ▼ personaje:
      href:     "http://localhost:8080/personajes/4"
▼ 4:
  nombre:      "MinervaMcGonagall"
  rol:         "directora"
  casa:        "Gryffindor"
  ascendencia: "Mestiza"
  ▼ _links:
    ▼ self:
      href:     "http://localhost:8080/personajes/5"
    ▼ personaje:
      href:     "http://localhost:8080/personajes/5"

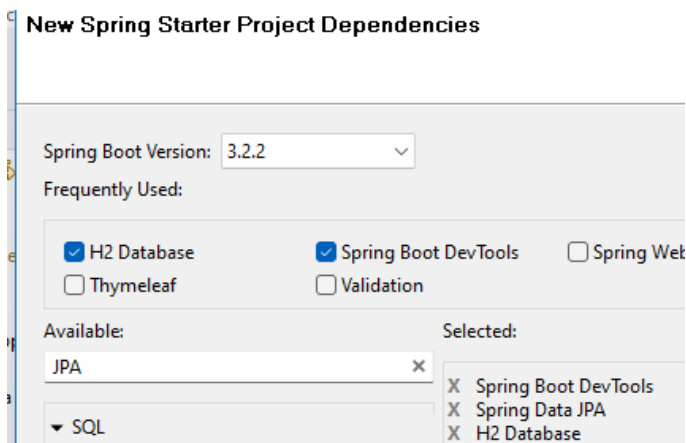
```

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.

```

PersonajeControlador.java  PersonajesRepositorio.java  Personaje.java >
1 package daw.dwes.ud6;
2
3 import jakarta.persistence.Entity;
4
5 @Entity
6 public class Personaje {
7
8     @Id
9     @GeneratedValue(strategy = GenerationType.IDENTITY)
10    private long id;
11
12    private String nombre;
13    private String rol;
14    private String casa;
15    private String ascendencia;
16
17    public long getId() {
18        return id;
19    }
20    public void setId(long id) {
21        this.id = id;
22    }
23    public String getNombre() {

```

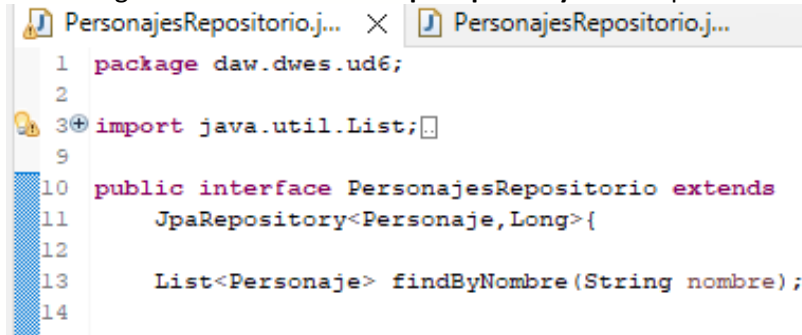
Copiamos la interfaz PersonajeRepositorio pero modificamos lo siguiente:

```

@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:

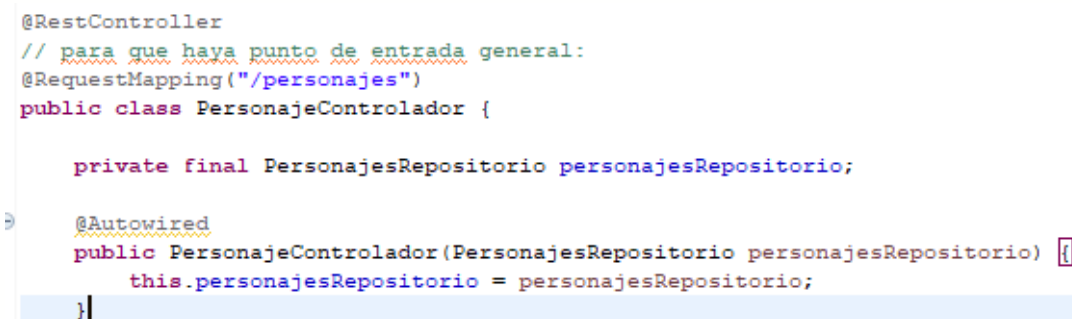


```

1 package daw.dwes.ud6;
2
3 import java.util.List;
4
5
6
7
8
9
10 public interface PersonajesRepository extends
11     JpaRepository<Personaje, Long>{
12
13     List<Personaje> findByNombre(String nombre);
14
15 }

```

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



```

@RestController
// para que haya punto de entrada general:
@RequestMapping("/personajes")
public class PersonajeControlador {

    private final PersonajesRepository personajesRepository;

    @Autowired
    public PersonajeControlador(PersonajesRepository personajesRepository) {
        this.personajesRepository = personajesRepository;
    }
}

```

- **@RestController**: lo aplicamos ya que no podemos usar **RestRepository**.
- **@RequestMapping("/personajes")**: como punto de entrada general en todos los métodos.
- **private final PersonajesRepository personajesRepository**: declaramos esta variable para representar el repositorio, y la vamos a utilizar para acceder a los datos de los personajes.
- **@Autowired**: para indicar a Spring que inyecte una instancia de **PersonajesRepository** en este constructor cuando se cree una instancia de **PersonajeControlador**. Permite que el repositorio esté disponible para usarlo en este controlador.
- **public PersonajeControlador(PersonajesRepository personajesRepository)**: este es el constructor e inicializa la variable **personajesRepository**. Se usa para inyectar el repositorio en el controlador.

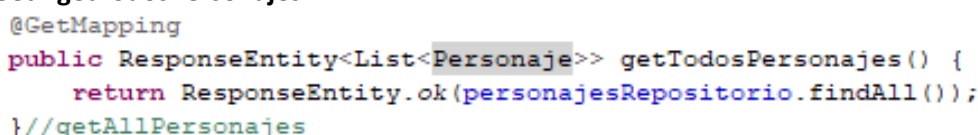
Métodos para las rutas (peticiones y respuestas):

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**:



```

@GetMapping
public ResponseEntity<List<Personaje>> getTodosPersonajes() {
    return ResponseEntity.ok(personajesRepository.findAll());
} //getAllPersonajes

```

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 **personajesRepository**.

Get - GetPersonajeId:

```
@GetMapping("/{id}")
public ResponseEntity<Personaje> getPersonajeById(@PathVariable("id") Long id) {
    return ResponseEntity.ok(personajesRepositorio.findById(id).get());
} //getId
```

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

```
@PostMapping
public ResponseEntity<Personaje> crearPersonaje(@RequestBody Personaje personaje) {
    return ResponseEntity.status(HttpStatus.CREATED)
        .body(personajesRepositorio.save(personaje));
} //postCrearPersonaje
```

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:

```
@PutMapping("/{id}")
public ResponseEntity<Personaje> actualizarPersonaje(
    @PathVariable("id") Long id,
    @RequestBody Personaje actualizadoPersonaje) {
    return ResponseEntity.ok(personajesRepositorio.save(actualizadoPersonaje));
} //putActualizar
```

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:

```
@DeleteMapping("/{id}")
public ResponseEntity<Void> borrarPersonaje(@PathVariable("id") Long id) {
    personajesRepositorio.deleteById(id);
    return ResponseEntity.noContent().build();
} //deleteBorrar
```

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

Borramos el personaje 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:

```
import.sql x
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('HarryPotter', 'estudiante', 'Gryffindor', 'Mestiza');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('RonWeasley', 'estudiante', 'Gryffindor', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('HermioneGranger', 'estudiante', 'Gryffindor', 'Muggle');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('DracoMalfoy', 'estudiante', 'Slytherin', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('MinervaMcGonagall', 'profesora', 'Gryffindor', 'Mestiza');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('LunaLovegood', 'estudiante', 'Ravenclaw', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('SeverusSnape', 'profesor', 'Slytherin', 'Muggle');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('GinnyWeasley', 'estudiante', 'Gryffindor', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('NevilleLongbottom', 'estudiante', 'Gryffindor', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('FredWeasley', 'estudiante', 'Gryffindor', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('GeorgeWeasley', 'estudiante', 'Gryffindor', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('ChoChang', 'estudiante', 'Ravenclaw', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('CedricDiggory', 'estudiante', 'Hufflepuff', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('SiriusBlack', 'prisionero', 'Gryffindor', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('RemusLupin', 'profesor', 'Gryffindor', 'Mestiza');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('BellatrixLestrange', 'mortifaga', 'Slytherin', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('Dobby', 'elfo doméstico', 'N/A', 'Sangre pura liberado');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('KingsleyShacklebolt', 'auror', 'N/A', 'Mestiza');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('NymphadoraTonks', 'auror', 'Hufflepuff', 'Sangre mestiza');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('MadEyeMoody', 'auror', 'N/A', 'Muggle');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('AlbusDumbledore', 'profesor', 'Gryffindor', 'Mestiza');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('SybillTrelawney', 'profesora', 'Ravenclaw', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('FiliusFlitwick', 'profesor', 'Ravenclaw', 'Sangre mestiza');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('LordVoldemort', 'mortifago', 'Slytherin', 'Mestiza');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('LuciusMalfoy', 'mortifago', 'Slytherin', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('NarcissaMalfoy', 'mortifago', 'Slytherin', 'Sangre pura');
INSERT INTO Personaje (nombre, rol, casa, ascendencia) VALUES ('PeterPettigrew', 'mortifago', 'Gryffindor', 'Sangre mestiza');
```

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:

The image displays two side-by-side browser windows showing JSON data from a web application. Both windows are at the URL `localhost:8080/personajes`.

The left window shows the JSON data for 14 characters, indexed from 0 to 13. The data is as follows:

Index	id	nombre	rol	casa	ascendencia
0	1	HarryPotter	estudiante	Gryffindor	Mestiza
1	2	RonWeasley	estudiante	Gryffindor	Sangre pura
2	3	HermioneGranger	estudiante	Gryffindor	Muggle
3	4	DracoMalfoy	estudiante	Slytherin	Sangre pura
4	5	MinervaMcGonagall	profesora	Gryffindor	Mestiza
5	6	LunaLovegood	estudiante	Ravenclaw	Sangre pura
6	7	SeverusSnape	profesor	Slytherin	Muggle
7	8	GinnyWeasley	estudiante	Gryffindor	Sangre pura
8	9	NevilleLongbottom	estudiante	Gryffindor	Sangre pura
9	10	FredWeasley	estudiante	Gryffindor	Sangre pura
10	11	GeorgeWeasley	estudiante	Gryffindor	Sangre pura
11	12	ChoChang	estudiante	Ravenclaw	Sangre pura
12	13	CedricDiggory	estudiante	Hufflepuff	Sangre pura
13	14	SiriusBlack	prisionero	Gryffindor	Sangre pura

The right window shows the same JSON data, but the 10th element (index 10, id 11, GeorgeWeasley) is highlighted, indicating it is the selected character.

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

```

JSON  Raw Data  Headers
Save Copy Collapse All Expand All Filter JSON
id: 10
nombre: "FredWeasley"
rol: "estudiante"
casa: "Gryffindor"
ascendencia: "Sangre pura"

```

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:

```

rol: "profesor"
casa: "Ravenclaw"
ascendencia: "Sangre mestiza"
▼ 23:
id: 24
nombre: "LordVoldemort"
rol: "mortifago"
casa: "Slytherin"
ascendencia: "Mestiza"
▼ 24:
id: 25
nombre: "LuciusMalfoy"
rol: "mortifago"
casa: "Slytherin"
ascendencia: "Sangre pura"
▼ 25:
id: 26
nombre: "NarcissaMalfoy"
rol: "mortifago"
casa: "Slytherin"
ascendencia: "Sangre pura"
▼ 26:
id: 27
nombre: "PeterPettigrew"
rol: "mortifago"
casa: "Gryffindor"
ascendencia: "Sangre mestiza"
▼ 27:
id: 28
nombre: "BillyWeasly"
rol: "Auror"
casa: "Gryffindor"
ascendencia: "Sangre pura"

```

POST ▼ http://localhost:8080/personajes

Params Authorization Headers (8) **Body** Pre-request Script Tests

none form-data x-www-form-urlencoded raw binary **JSON**

```

1
2 "nombre": "BillyWeasly",
3 "rol": "Auror",
4 "casa": "Gryffindor",
5 "ascendencia": "Sangre pura"
6

```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON ▼

```

1
2 "id": 28,
3 "nombre": "BillyWeasly",
4 "rol": "Auror",
5 "casa": "Gryffindor",
6 "ascendencia": "Sangre pura"
7

```

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

```

▼ 0:
id: 1
nombre: "HarryPotter"
rol: "estudiante"
casa: "Gryffindor"
ascendencia: "Mestiza"

```

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:

The screenshot shows a REST client interface. On the left, a list of characters is displayed with their attributes: id, nombre, rol, casa, and ascendencia. The last character in the list has id 28, nombre "HarryPotter", rol "estudiante", casa "Gryffindor", and ascendencia "Mestiza". On the right, the POST request details are shown, including the URL http://localhost:8080/personajes and the JSON body. The response status is 201 Created, and the response body shows the same character data as the last item in the list on the left.

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:

The screenshot shows an IDE dialog box titled "Generate hashCode() and equals()". It prompts the user to "Select the fields to include in the hashCode() and equals() methods:". The fields listed are ascendencia, casa, id, nombre, and rol. The 'nombre' field is checked, indicating it will be included in the hashCode() and equals() methods.

```

@Override
public int hashCode() {
    return Objects.hash(nombre);
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Personaje other = (Personaje) obj;
    return Objects.equals(nombre, other.nombre);
}

```

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:

```

@PostMapping
public ResponseEntity<?> crearPersonaje(@RequestBody Personaje personaje) {
    Personaje existePersonaje = personajesRepositorio
        .findByNombre(personaje.getNombre());
    if (existePersonaje != null)
        return ResponseEntity
            .body("El personaje ya existe");
    return ResponseEntity
        .body(personaje);
} //postCrearPersonaje

```

Type mismatch: cannot convert from List<Personaje> to Personaje

3 quick fixes available:

- 1. Add cast to 'Personaje'
- 2. Change type of 'existePersonaje' to 'List<Personaje>'
- 3. Change return type of 'findByNombre(...)' to 'Personaje'

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 opción 3, voy a ir a la clase PersonajesRepositorio y cambio el método findByNombre de List<Personaje> a Personaje:

```

0 public interface PersonajesRepositorio extends
1     JpaRepository<Personaje, Long>{
2
3     Personaje findByNombre(String nombre);
4
5 }

```

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:

```

/main/resources 64 return ResponseEntity.status(HttpStatus.FOUND);

```

404 Not Found.
See Also:
[HTTP/1.1: Semantics and Content, section 6.5.4](http://1.1: Semantics and Content, section 6.5.4)

- MOVED_TEMPORARILY : HttpStatus - HttpStatus
- MULTI_STATUS : HttpStatus - HttpStatus
- MULTIPLE_CHOICES : HttpStatus - HttpStatus
- NETWORK_AUTHENTICATION_REQUIRED : HttpStatus - HttpStatus
- NO_CONTENT : HttpStatus - HttpStatus
- NON_AUTHORITATIVE_INFORMATION : HttpStatus - HttpStatus
- NOT_ACCEPTABLE : HttpStatus - HttpStatus
- NOT_EXTENDED : HttpStatus - HttpStatus
- NOT_FOUND : HttpStatus - HttpStatus

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 existe `Personaje = 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:

```
@PostMapping
public ResponseEntity<?> crearPersonaje(@RequestBody Personaje personaje) {
    Personaje existePersonaje = personajesRepositorio
        .findByNombre(personaje.getNombre());
    if (existePersonaje != null) {
        return ResponseEntity.status(HttpStatus.METHOD_NOT_ALLOWED)
            .body("El personaje con nombre " + personaje.getNombre() +
                " ya existe.");
    }
    return ResponseEntity.status(HttpStatus.CREATED)
        .body(personajesRepositorio.save(personaje));
} //postCrearPersonaje
```

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

The screenshot shows a REST client interface. On the left, a list of existing characters is displayed:

- 0: id: 1, nombre: "HarryPotter", rol: "estudiante", casa: "Gryffindor", ascendencia: "Mestiza"
- 1: id: 2, nombre: "RonWeasley", rol: "estudiante", casa: "Gryffindor", ascendencia: "Sangre pura"
- 2: id: 3, nombre: "HermioneGranger", rol: "estudiante", casa: "Gryffindor", ascendencia: "Muggle"

The main area shows a POST request to `http://localhost:8080/personajes` with a JSON body:

```
{
  "nombre": "HarryPotter",
  "rol": "estudiante",
  "casa": "Gryffindor",
  "ascendencia": "Mestiza"
}
```

The response is a 405 Method Not Allowed status, with a message: "El personaje con nombre HarryPotter ya existe."

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

The screenshot shows a REST client interface with a list of characters on the left and a POST request details on the right.

Character List (Left):

- rol: "mortífago", casa: "slytherin", ascendencia: "Mestiza"
- 24: id: 25, nombre: "LuciusMalfoy", rol: "mortífago", casa: "slytherin", ascendencia: "Sangre pura"
- 25: id: 26, nombre: "NarcissaMalfoy", rol: "mortífago", casa: "slytherin", ascendencia: "Sangre pura"
- 26: id: 27, nombre: "PeterPettigrew", rol: "mortífago", casa: "Gryffindor", ascendencia: "Sangre mestiza"
- 27: id: 28, nombre: "AranchaChicharro", rol: "estudiante", casa: "Gryffindor", ascendencia: "Muggle"

POST Request Details (Right):

Method: POST, URL: http://localhost:8080/personajes

Body (JSON):

```

1
2   .... "nombre": "AranchaChicharro",
3   .... "rol": "estudiante",
4   .... "casa": "Gryffindor",
5   .... "ascendencia": "Muggle"
6

```

Response (Pretty):

```

1
2   "id": 28,
3   "nombre": "AranchaChicharro",
4   "rol": "estudiante",
5   "casa": "Gryffindor",
6   "ascendencia": "Muggle"
7

```

Status: 201 Created, 16 ms, 268 B

Console: Not connected to a Postman account

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:

The screenshot shows a REST client interface. On the left, a list of characters is displayed with details like id, nombre, rol, casa, and ascendencia. Item 21 is highlighted. On the right, a PUT request is configured to `http://localhost:8080/personajes/21`. The request body is in JSON format, containing the updated data for the character with id 21. The response status is 200 OK.

```

PUT http://localhost:8080/personajes/21

{
  "nombre": "AlbusDumbledore",
  "rol": "director",
  "casa": "Gryffindor",
  "ascendencia": "Sangre pura"
}
  
```

```

19: {
  id: 20,
  nombre: "MadEyeMoody",
  rol: "auror",
  casa: "N/A",
  ascendencia: "Muggle"
}
20: {
  id: 21,
  nombre: "AlbusDumbledore",
  rol: "profesor",
  casa: "Gryffindor",
  ascendencia: "Mestiza"
}
21: {}
22: {}
23: {}
24: {}
25: {}
26: {}
27: {
  id: 28,
  nombre: "AlbusDumbledore",
  rol: "director",
  casa: "Gryffindor"
}
  
```

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:

```

@PutMapping("/{id}")
public ResponseEntity<Personaje> actualizarPersonaje(
    @PathVariable("id") Long id,
    @RequestBody Personaje actualizadoPersonaje) {
    Personaje existePersonaje = personajesRepositorio.findById(id).get();
    if (existePersonaje != null) {
        existePersonaje.setNombre(actualizadoPersonaje.getNombre());
        existePersonaje.setRol(actualizadoPersonaje.getRol());
        existePersonaje.setCasa(actualizadoPersonaje.getCasa());
        existePersonaje.setAscendencia(actualizadoPersonaje.getAscendencia());
        return ResponseEntity.ok(personajesRepositorio.save(existePersonaje));
    } else {
        //si no existe el personaje:
        return ResponseEntity.notFound().build();
    }
}
  
```

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:

The screenshot shows a REST client interface with a list of characters on the left and a PUT request details panel on the right.

Character List (Left):

- 19: id: 20, nombre: "MadEyeMoody", rol: "auror", casa: "N/A", ascendencia: "Muggle"
- 20: id: 21, nombre: "AlbusDumbledore", rol: "director", casa: "Gryffindor", ascendencia: "Sangre pura"
- 21: id: 22, nombre: "SybillTrelawney", rol: "profesora", casa: "Ravenclaw", ascendencia: "Sangre pura"
- 22: id: 23, nombre: "FiliusFlitwick", rol: null, casa: null, ascendencia: null

PUT Request (Right):

- Method: PUT
- URL: http://localhost:8080/personajes/21
- Body (JSON):


```
{
  "nombre": "AlbusDumbledore",
  "rol": "director",
  "casa": "Gryffindor",
  "ascendencia": "Sangre pura"
}
```
- Status: 200 OK, 52 ms, 265 B

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):

The screenshot shows the REST client interface after the second PUT request.

Character List (Left):

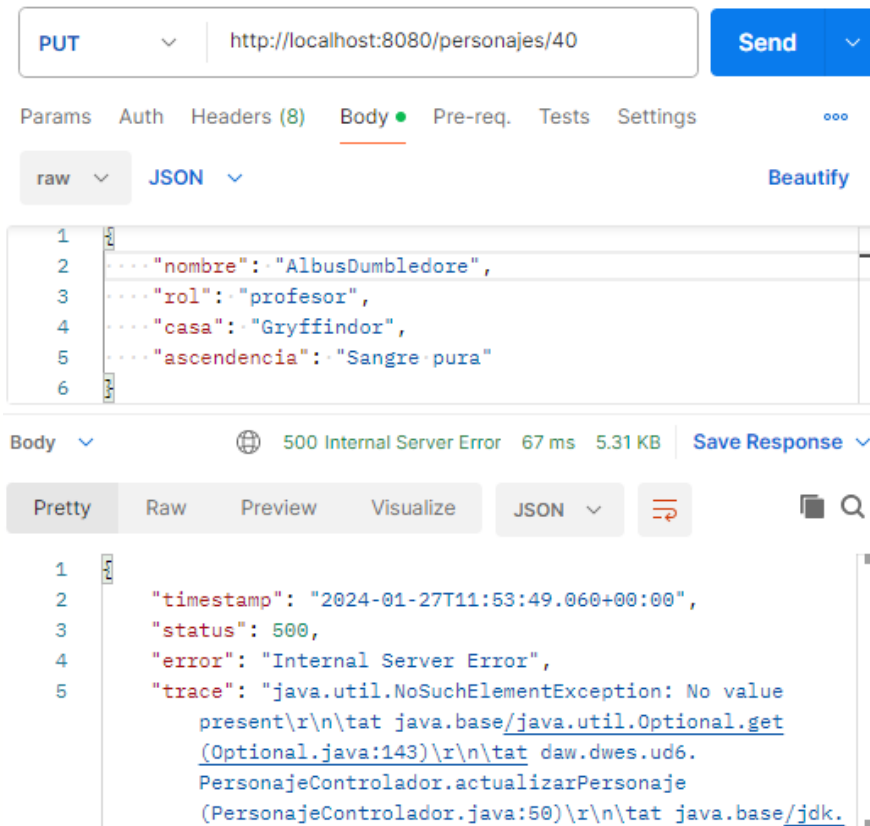
- 18: id: 19, nombre: "NymphadoraTonks", rol: "auror", casa: "Hufflepuff", ascendencia: "Sangre mestiza"
- 19: id: 20, nombre: "MadEyeMoody", rol: "auror", casa: "N/A", ascendencia: "Muggle"
- 20: id: 21, nombre: null, rol: "profesor", casa: null, ascendencia: null
- 21: id: 22, nombre: "SybillTrelawney", rol: "profesora", casa: "Ravenclaw", ascendencia: "Sangre pura"

PUT Request (Right):

- Method: PUT
- URL: http://localhost:8080/personajes/21
- Body (JSON):


```
{
  "rol": "profesor"
}
```
- Status: 200 OK, 175 ms, 235 B

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:

```
@PutMapping("/{id}")
public ResponseEntity<?> actualizarPersonaje(
    @PathVariable("id") Long id,
    @RequestBody Personaje actualizadoPersonaje) {
    Personaje existePersonaje = personajesRepositorio.findById(id).orElse(null);
    if (existePersonaje != null) {
        //Personaje existePersonaje = optionalPersonaje.get();
        existePersonaje.setNombre(actualizadoPersonaje.getNombre());
        existePersonaje.setRol(actualizadoPersonaje.getRol());
        existePersonaje.setCasa(actualizadoPersonaje.getCasa());
        existePersonaje.setAscendencia(actualizadoPersonaje.getAscendencia());
        return ResponseEntity.ok(personajesRepositorio.save(existePersonaje));
    } else {
        //si no existe el personaje:
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body("El personaje con id " + id + " no existe");
    }
}
}
}
```

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

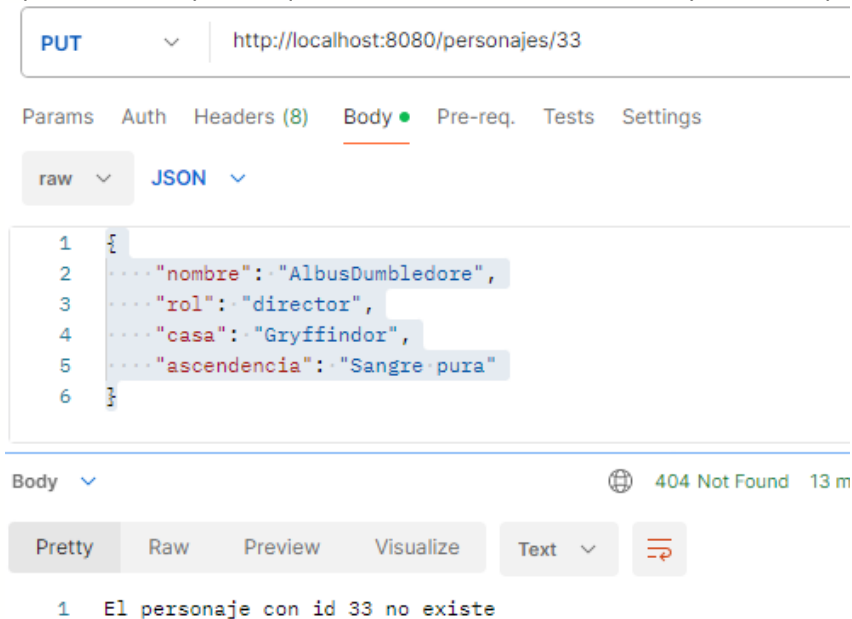
Personaje existe `Personaje = personajesRepositorio.findById(id).orElse(null);`

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.

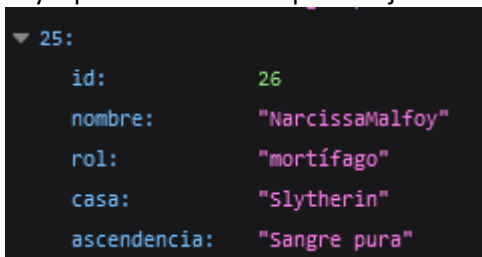
Vamos ahora a hacer una nueva prueba de este PUT con un elemento que no existe.

Y podemos comprobar que sale el estado **404 Not Found**, 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:

The screenshot shows a REST client interface. At the top, a DELETE request is made to `http://localhost:8080/personajes/26`. The response is `204 No Content` with a status of `40 ms` and `112 B`. The response body is empty. On the left, a list of characters is displayed, including Filius Flitwick, Lord Voldemort, Lucius Malfoy, and Peter Pettigrew.

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**.

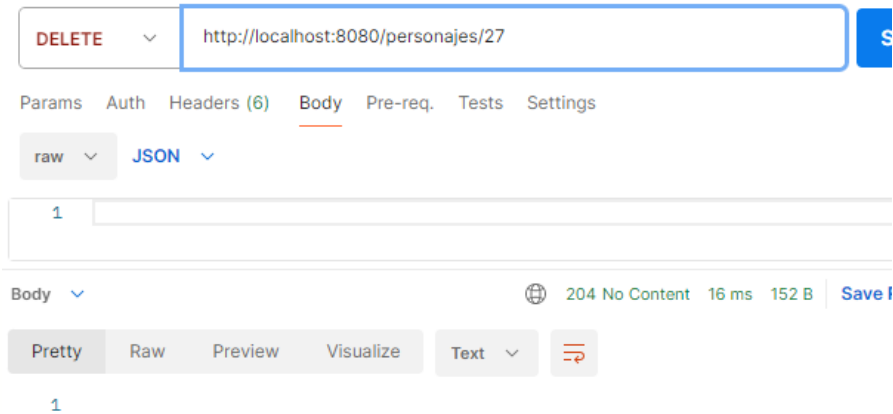
Por lo que vuelvo al código del controlador y añado también una condición de si existe que lo borre y salga este mensaje de **no content**, pero si no existe, que salga error, igual que los métodos Post y Put. Y aprovecho para también añadir mensajes personalizados, tanto cuando existe el elemento como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

```
@DeleteMapping("/{id}")
public ResponseEntity<?> borrarPersonaje(@PathVariable("id") Long id) {
    Personaje existePersonaje = personajesRepositorio.findById(id).orElse(null);
    if (existePersonaje != null) {
        personajesRepositorio.deleteById(id);
        return ResponseEntity.status(HttpStatus.NO_CONTENT)
            .body("El personaje con id " + id + " ha sido borrado.");
    } else {
        //si no existe el personaje:
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body("El personaje con id " + id + " no existe");
    }
}
}
}
```

Pruebo ahora a borrar el elemento con id 27:

The screenshot shows a REST client interface. At the top, a DELETE request is made to `http://localhost:8080/personajes/27`. The response is `204 No Content` with a status of `40 ms` and `112 B`. The response body is empty. On the left, a list of characters is displayed, including Peter Pettigrew, Lord Voldemort, Lucius Malfoy, and Peter Pettigrew.

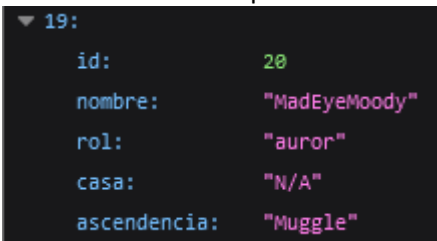
El elemento se borra, pero no sale el mensaje personalizado.



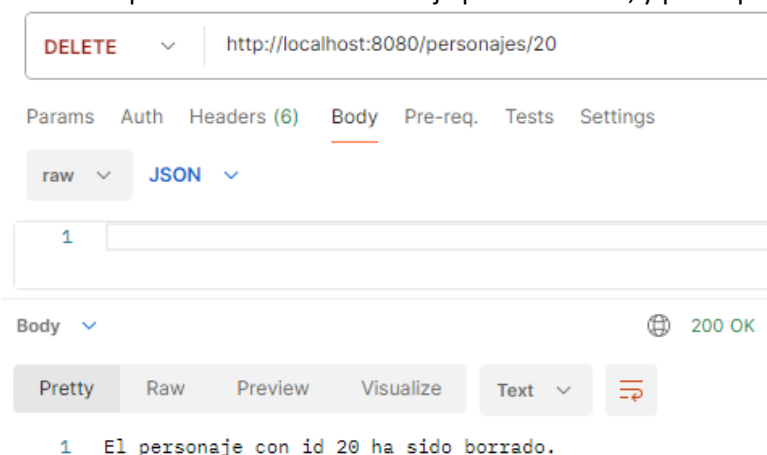
Tras revisar el código, que no veía ningún fallo, me di cuenta que el status que había puesto, el **no Content**, es cuando la acción se ha realizado pero la respuesta no incluye entidad. Esto me di cuenta gracias al enlace de las referencias del tema, indicado anteriormente (<https://restfulapi.net/http-methods/>).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

```
@DeleteMapping("/{id}")
public ResponseEntity<?> borrarPersonaje(@PathVariable("id") Long id) {
    Personaje existePersonaje = personajesRepositorio.findById(id).orElse(null);
    if (existePersonaje != null) {
        personajesRepositorio.deleteById(id);
        return ResponseEntity.status(HttpStatus.OK)
            .body("El personaje con id " + id + " ha sido borrado.");
    } else {
        return ResponseEntity.status(HttpStatus.NO_CONTENT).body(null);
    }
}
```

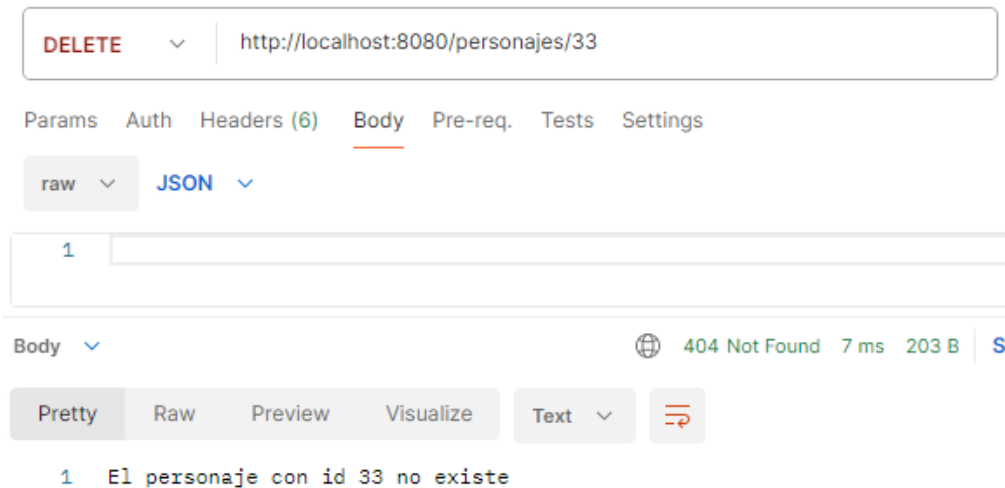
Y vuelvo a hacer una prueba de borrar. Voy a borrar el elemento con id 20:



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:

```
@PatchMapping("/{id}")
public ResponseEntity<?> actualizarAtributoPersonaje(
    @PathVariable("id") Long id,
    @RequestBody Personaje actualizadoPersonaje) {
    Personaje existePersonaje = personajesRepositorio.findById(id).orElse(null);
    if (existePersonaje != null) {
        if (actualizadoPersonaje.getNombre() != null) {
            existePersonaje.setNombre(actualizadoPersonaje.getNombre());
        }
        if (actualizadoPersonaje.getRol() != null) {
            existePersonaje.setRol(actualizadoPersonaje.getRol());
        }
        if (actualizadoPersonaje.getCasa() != null) {
            existePersonaje.setCasa(actualizadoPersonaje.getCasa());
        }
        if (actualizadoPersonaje.getAscendencia() != null) {
            existePersonaje.setAscendencia(actualizadoPersonaje.getAscendencia());
        }
        return ResponseEntity.ok(personajesRepositorio.save(existePersonaje));
    } else {
        // Si no existe el personaje:
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body("El personaje con id " + id + " no existe");
    }
}
//patch
```

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

The screenshot shows a REST client interface with a list of characters on the left and a PATCH request details panel on the right.

Character List (Left):

- id: 4, nombre: "MinervaMcGonagall", rol: "profesora", casa: "Gryffindor", ascendencia: "Mestiza"
- id: 4, nombre: "DracoMalfoy", rol: "estudiante", casa: "Slytherin", ascendencia: "Sangre pura"
- id: 5, nombre: "MinervaMcGonagall", rol: "Jefa de estudios", casa: "Gryffindor", ascendencia: "Sangre pura"
- id: 6, nombre: "LunaLovegood", rol: "estudiante", casa: "Ravenclaw", ascendencia: "Sangre pura"
- id: 7, nombre: "SeverusSnape", rol: "profesor", casa: "Slytherin", ascendencia: "Muggle"

PATCH Request (Right):

- Method: PATCH
- URL: http://localhost:8080/personajes/5
- Body (JSON):


```
{
  "rol": "Jefa de estudios",
  "ascendencia": "Sangre pura"
}
```
- Status: 200 OK, 163 ms, 274 B

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

The screenshot shows a REST client interface with a PATCH request to a non-existent character ID.

PATCH Request (Top):

- Method: PATCH
- URL: http://localhost:8080/personajes/35

Request Body (JSON):

```
{
  "rol": "jefa de estudios",
  "ascendencia": "Sangre pura"
}
```

Response (Bottom):

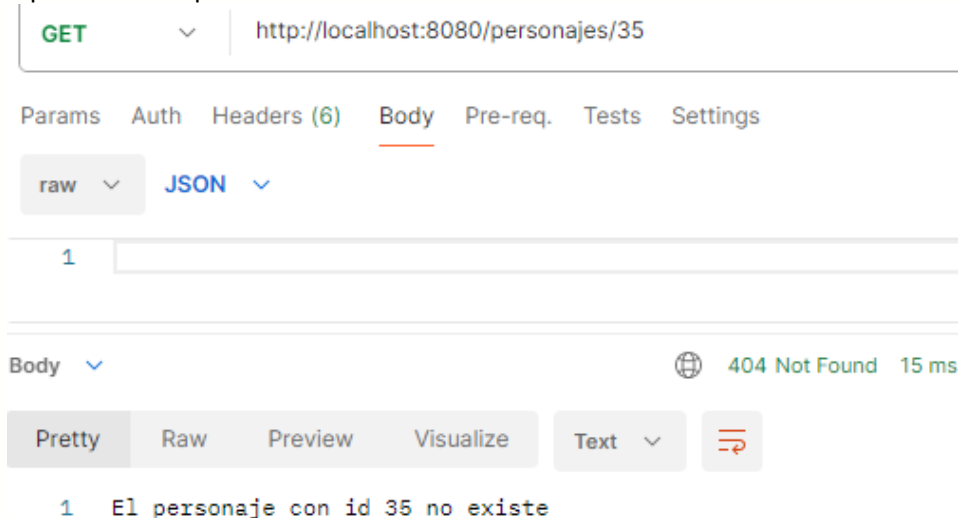
- Status: 404 Not Found, 13 ms, 203 B
- Body (Text):


```
1 El personaje con id 35 no existe
```

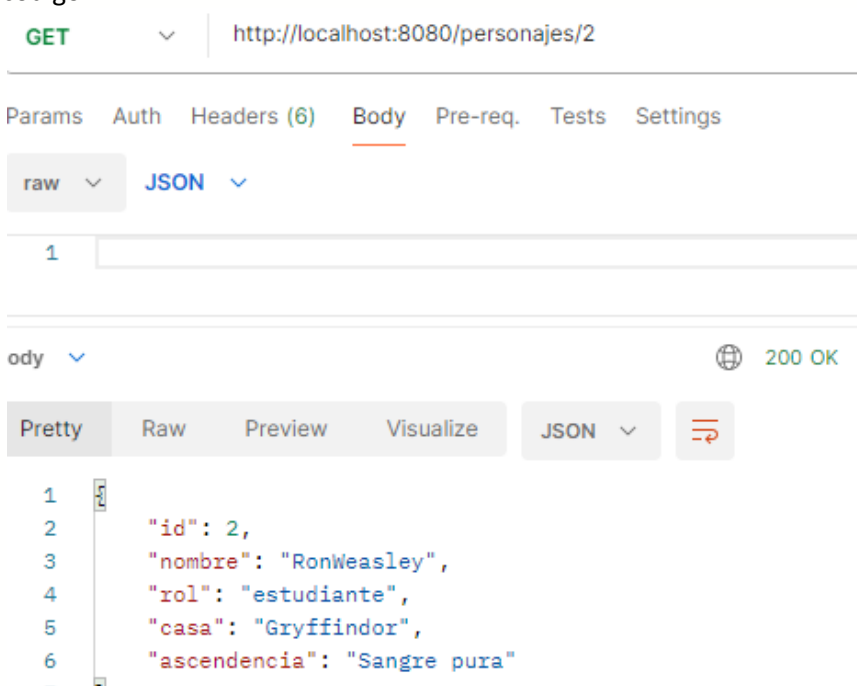
Ahora voy a volver al método **getPersonajeById**, para añadir también un mensaje personalizado en el caso de que no exista el elemento, por lo que siguiendo el ejemplo de los demás métodos, también tengo que crear variable de tipo personaje de si existe. El código quedaría así:

```
@GetMapping("/{id}")
public ResponseEntity<?> getPersonajeById(@PathVariable("id") Long id) {
    Personaje existePersonaje = personajesRepositorio.findById(id).orElse(null);
    if (existePersonaje != null) {
        return ResponseEntity.ok(existePersonaje);
    } else {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body("El personaje con id " + id + " no existe");
    }
} //getId
```

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.

```

15 @ExtendWith(MockitoExtension.class)
16 public class UnitarioTest_PersonajeControlador {
17
18     List<Personaje> personajes;
19
20     @InjectMocks
21     private PersonajeControlador personajeControlador;
22
23     @Mock
24     private PersonajesRepositorio personajesRepositorio;
25
26     @BeforeEach
27     //método que se ejecuta antes de cada prueba:
28     void setUp() {
29         personajes = personajesRepositorio.findAll();
30     }

```

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
void getTodosPersonajes() {
    // Lo que vamos a simular
    when(personajesRepositorio.findAll()).thenReturn(List.copyOf(personajes))

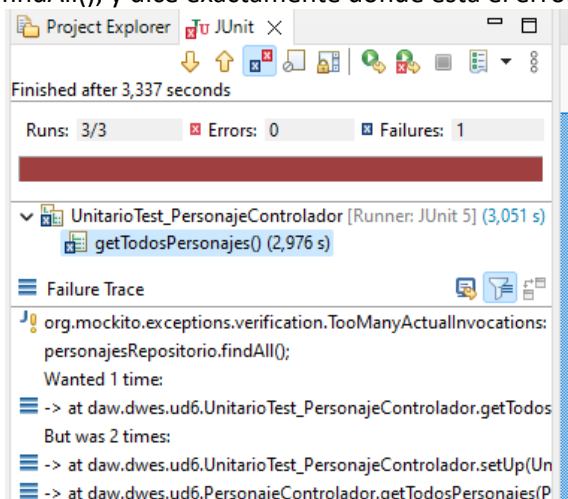
    // Test
    ResponseEntity<List<Personaje>> responseEntity =
        personajeControlador.getTodosPersonajes();

    // Comprobaciones
    assertEquals(
        () -> assertNotNull(responseEntity),
        () -> assertEquals(HttpStatus.OK, responseEntity.getStatusCode())
    );

    // Verificamos que se ha llamado al método
    verify(personajesRepositorio, times(1)).findAll();
} // 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.
- **assertEquals(...)**: 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:

```

46 @BeforeEach
47 //método que se ejecuta antes de cada prueba:
48 void setUp() {
49     personajes = personajesRepositorio.findAll();
50 }
51
52 @Test
53 void getTodosPersonajes() {
54     // Lo que vamos a simular
55     when(personajesRepositorio.findAll()).thenReturn(List.copyOf(personajes));
56
57     // Test
58     ResponseEntity<List<Personaje>> responseEntity =
59         personajeControlador.getTodosPersonajes();
60
61     // Comprobaciones
62     assertAll(
63         () -> assertNotNull(responseEntity),
64         () -> assertEquals(HttpStatus.OK, responseEntity.getStatusCode())
65     );
66
67     // Verificamos que se ha llamado al método
68     //quito esta linea porque ya se está ejecutando en el setup:
69     //verify(personajesRepositorio, times(1)).findAll();
70 }

```

*** 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_Existeld():

```

@Test
void getPersonajeById_Si_Existeld() {
    Long id = 1L;
    Personaje personaje = new Personaje();
    personaje.setId(id);
    when(personajesRepositorio.findById(id)).thenReturn(Optional.of(personaje));

    ResponseEntity<?> responseEntity = personajeControlador.getPersonajeById(id);

    // Assert: comprobaciones
    assertAll(
        () -> assertNotNull(responseEntity),
        () -> assertEquals(HttpStatus.OK, responseEntity.getStatusCode()),
        () -> assertTrue(responseEntity.getBody() instanceof Personaje),
        () -> assertEquals(personaje, responseEntity.getBody())
    );

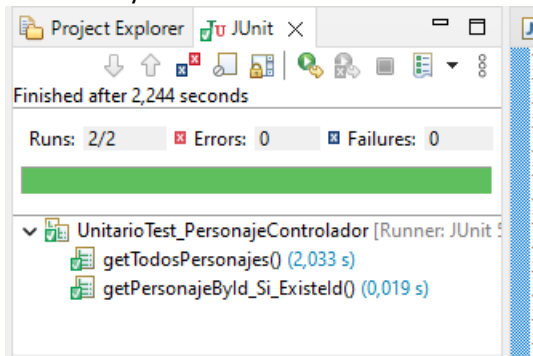
    // Verificamos que se ha llamado al método
    verify(personajesRepositorio, times(1)).findById(id);
}

```

- **@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_Existeld

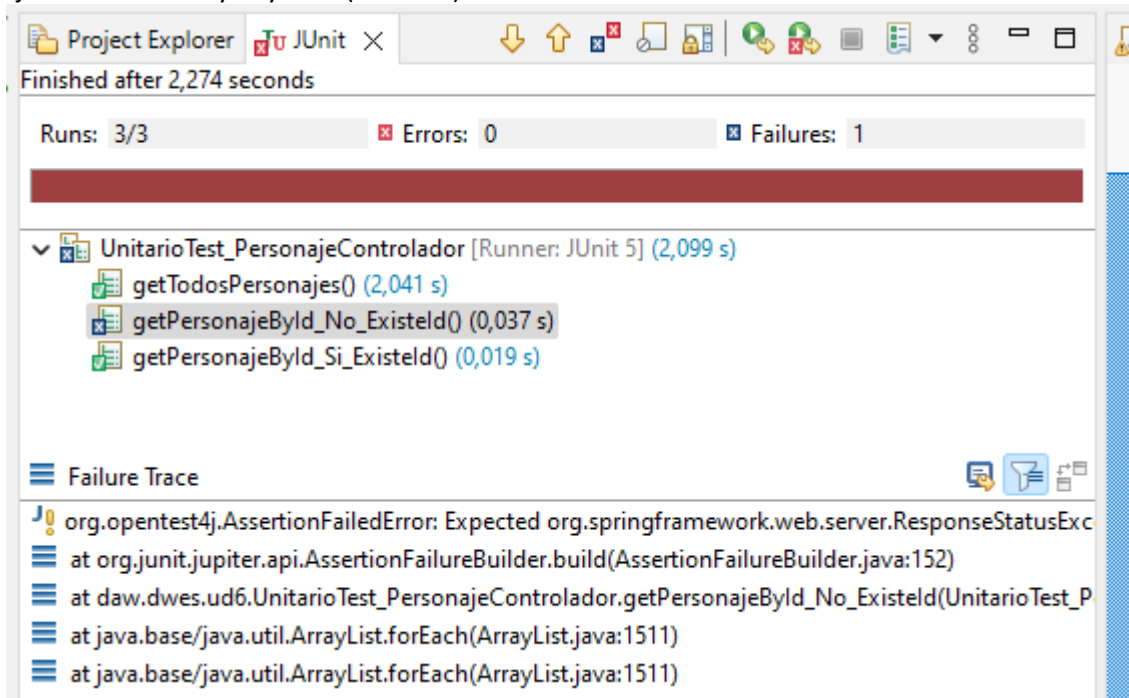
```
@Test
void getPersonajeById_No_Existeld() {
    Long id = -100L;
    when(personajesRepositorio.findById(id)).thenReturn(Optional.empty());

    //salta la excepcion
    //assertThrows para verificar que se lanza una ResponseStatusException
    var res = assertThrows(ResponseStatusException.class, () -> {
        personajeControlador.getPersonajeById(id);
    });

    // Comprobamos que la excepción es la esperada
    assert (res.getMessage().contains("El personaje con id " +
        id + " no existe"));

    // Verificamos que se ha llamado al método
    verify(personajesRepositorio, times(1)).findById(id);
} //getNoId
```

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_Existeld**. 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:

```
else {
    //return ResponseEntity.status(HttpStatus.NOT_FOUND).body("El personaje con
```

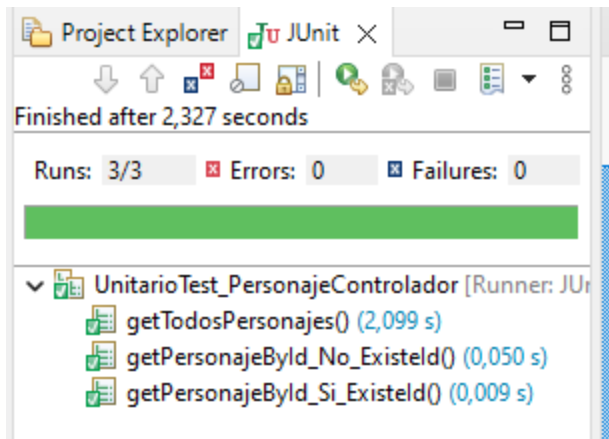
Por esta con throw:

```
throw new ResponseStatusException(HttpStatus.NOT_FOUND,
    "El personaje con id " + id + " no existe");
```

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

```
@GetMapping("/{id}")
public ResponseEntity<?> getPersonajeById(@PathVariable("id") Long id) {
    Personaje existePersonaje = personajesRepositorio.findById(id).orElse(null);
    if (existePersonaje != null) {
        return ResponseEntity.ok(existePersonaje);
    } else {
        //return ResponseEntity.status(HttpStatus.NOT_FOUND).body("El personaje c
        //cambio la línea:
        //nos aseguramos de que lance una ResponseStatusException:
        throw new ResponseStatusException(HttpStatus.NOT_FOUND,
            "El personaje con id " + id + " no existe");
    }
}
} //getId
```

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.

```
@SpringBootTest
public class IntegracionTest_PersonajeControlador {

    @Autowired
    private PersonajeControlador personajeControlador;
```

- **@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:

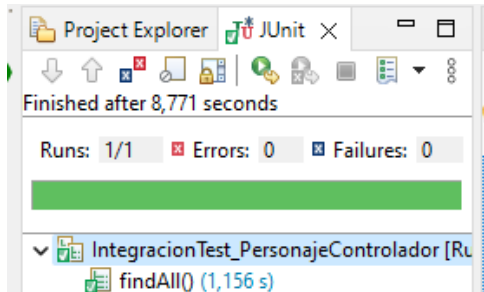
```
@Test
void getTodosPersonajes() {
    //var list = personajesRepositorio.findAll();
    ResponseEntity<List<Personaje>> response =
        personajeControlador.getTodosPersonajes();

    assertAll(
        () -> assertNotNull(response),
        () -> assertEquals(HttpStatus.OK, response.getStatusCode())
    );
} //findAll
```

- **@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_ExistId`:

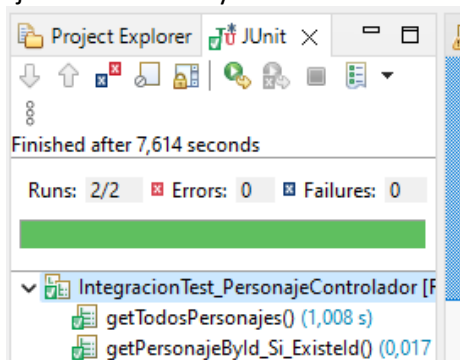
```
@Test
void getPersonajeById_Si_ExistId() {
    Long id = 1L;

    ResponseEntity<?> responseEntity =
        personajeControlador.getPersonajeById(id);

    // Assert: comprobaciones
    assertEquals(
        () -> assertNotNull(responseEntity),
        () -> assertEquals(HttpStatus.OK, responseEntity.getStatusCode()),
        () -> assertTrue(responseEntity.getBody() instanceof Personaje)
    );
} //getSiId
```

- **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_ExistId:

```

@Test
void getPersonajeById_No_ExistId() {
    Long id = -100L;

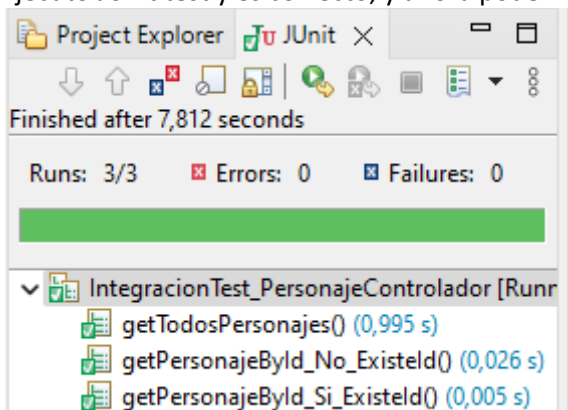
    //assertThrows para verificar que se lanza una ResponseStatusEx
    var res = assertThrows(ResponseStatusException.class, () -> {
        personajeControlador.getPersonajeById(id);
    });

    // Comprobamos que la excepción es la esperada
    assert (res.getMessage().contains("El personaje con id " +
        id + " no existe"));
} //getNoId

```

- **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 saber 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.