

D1-UD6 Nuestra primera API REST completa

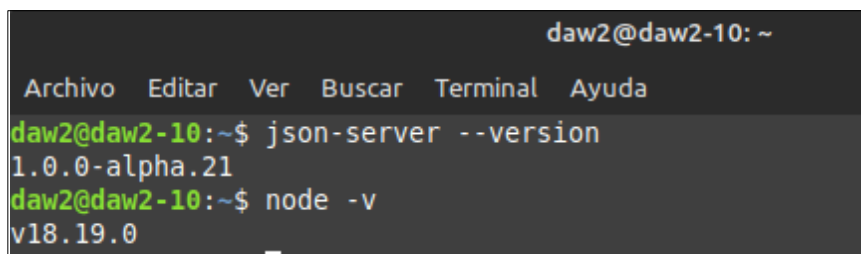
D1-UD6 Nuestra primera API REST completa

Te propongo seguir aprendiendo sobre las API RESTful creando y probando tu propia API con los tres métodos que describo a continuación. Antes de nada, elije una temática de tu interés. Para empezar, puedes hacerlo con un solo tipo de entidad y pocos atributos.

Apartado 1.- API REST con Json-server:

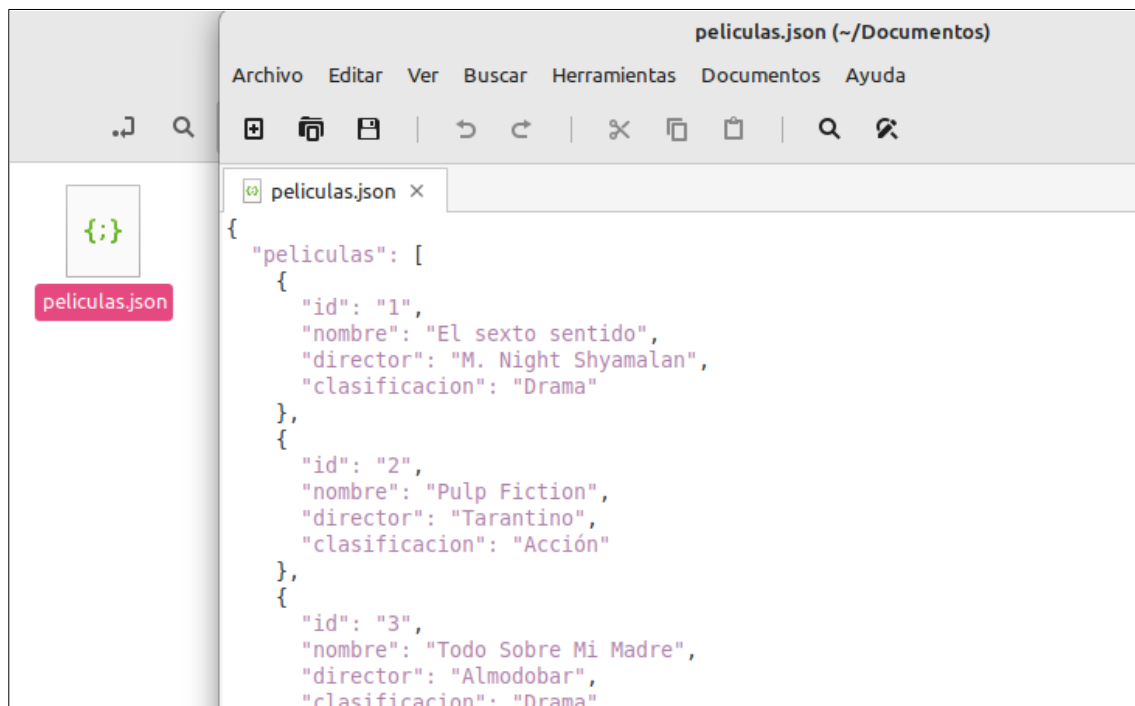
Basándote en este tutorial <https://desarrolloweb.com/articulos/crear-api-rest-json-server.html> (si te diera problema de sintaxis por versión de Node.JS, recomiendo la opción 3 de <https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-22-04>) prueba la API REST generada bien sea con Postman, POST, curl o cualquiera de las alternativas de las referencias.

- Una vez instalado node.js y el servidor json:



```
daw2@daw2-10: ~  
Archivo  Editar  Ver  Buscar  Terminal  Ayuda  
daw2@daw2-10:~$ json-server --version  
1.0.0-alpha.21  
daw2@daw2-10:~$ node -v  
v18.19.0
```

- Creamos el archivo. json que llamaremos peliculas.json con el contenido sugerido:



```
peliculas.json (~/Documentos)  
Archivo  Editar  Ver  Buscar  Herramientas  Documentos  Ayuda  
peliculas.json x  
{  
  "peliculas": [  
    {  
      "id": "1",  
      "nombre": "El sexto sentido",  
      "director": "M. Night Shyamalan",  
      "clasificacion": "Drama"  
    },  
    {  
      "id": "2",  
      "nombre": "Pulp Fiction",  
      "director": "Tarantino",  
      "clasificacion": "Acción"  
    },  
    {  
      "id": "3",  
      "nombre": "Todo Sobre Mi Madre",  
      "director": "Almodobar",  
      "clasificacion": "Drama"  
    }  
  ]  
}
```

- Arrancamos el servidor del API REST Fake:

json-server --watch peliculas.json

```
daw2@daw2-10: ~/Documentos
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
daw2@daw2-10:~$ cd Documentos/
daw2@daw2-10:~/Documentos$ json-server --watch peliculas.json
--watch/-w can be omitted, JSON Server 1+ watches for file changes by default
JSON Server started on PORT :3000
Press CTRL-C to stop
Watching peliculas.json...

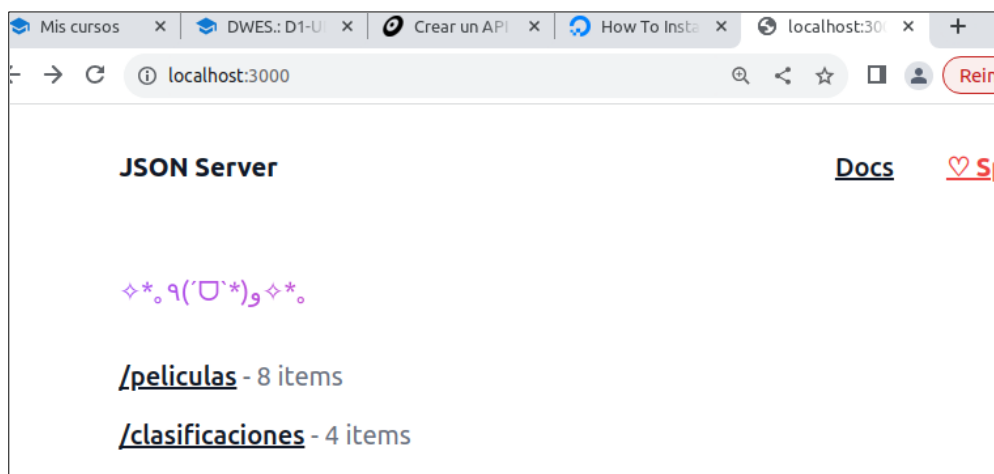
Index:
http://localhost:3000/

Static files:
Serving ./public directory if it exists

Endpoints:
http://localhost:3000/peliculas
http://localhost:3000/clasificaciones
```

- Accedemos al API:

http://localhost:3000



El objetivo es describir el código de retorno y los datos devueltos para cada una de las siguientes peticiones:

- GET de la lista de entidades.

GET de la lista de películas: <http://localhost:3000/peliculas>

The screenshot shows a web client interface with the following details:

- Method:** GET
- URL:** http://localhost:3000/peliculas
- Headers:** 6 hidden
- Body:** Pretty view showing a JSON array of 3 movie objects.

```
1 {
2   {
3     "id": "1",
4     "nombre": "El sexto sentido",
5     "director": "M. Night Shyamalan",
6     "clasificacion": "Drama"
7   },
8   {
9     "id": "2",
10    "nombre": "Pulp Fiction",
11    "director": "Tarantino",
12    "clasificacion": "Acción"
13  },
14  {
15    "id": "3",
16    "nombre": "Todo Sobre Mi Madre",
17    "director": "Almodobar",
18    "clasificacion": "Drama"
19  }
20 }
```
- Status:** 200 OK, 6 ms, 1.25 KB

- GET de una entidad existente: <http://localhost:3000/peliculas/2>

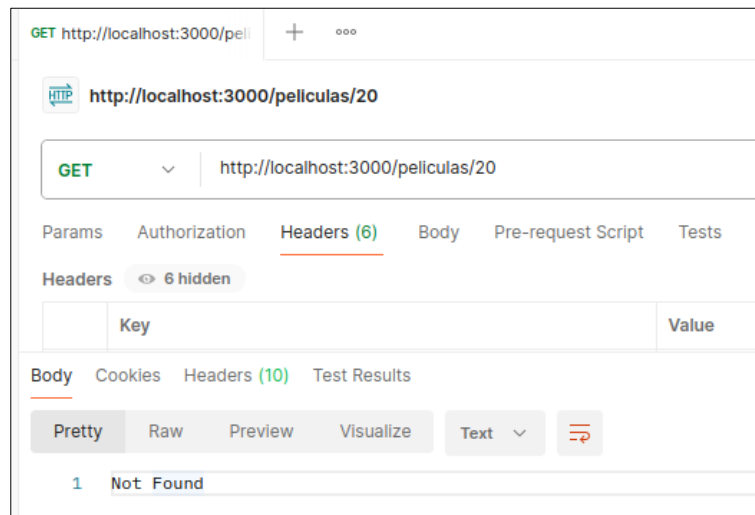
The screenshot shows a web client interface with the following details:

- Method:** GET
- URL:** http://localhost:3000/peliculas/2
- Headers:** 6 hidden
- Body:** Pretty view showing a JSON object for the movie 'Pulp Fiction'.

```
1 {
2   "id": "2",
3   "nombre": "Pulp Fiction",
4   "director": "Tarantino",
5   "clasificacion": "Acción"
6 }
```
- Status:** 200 OK

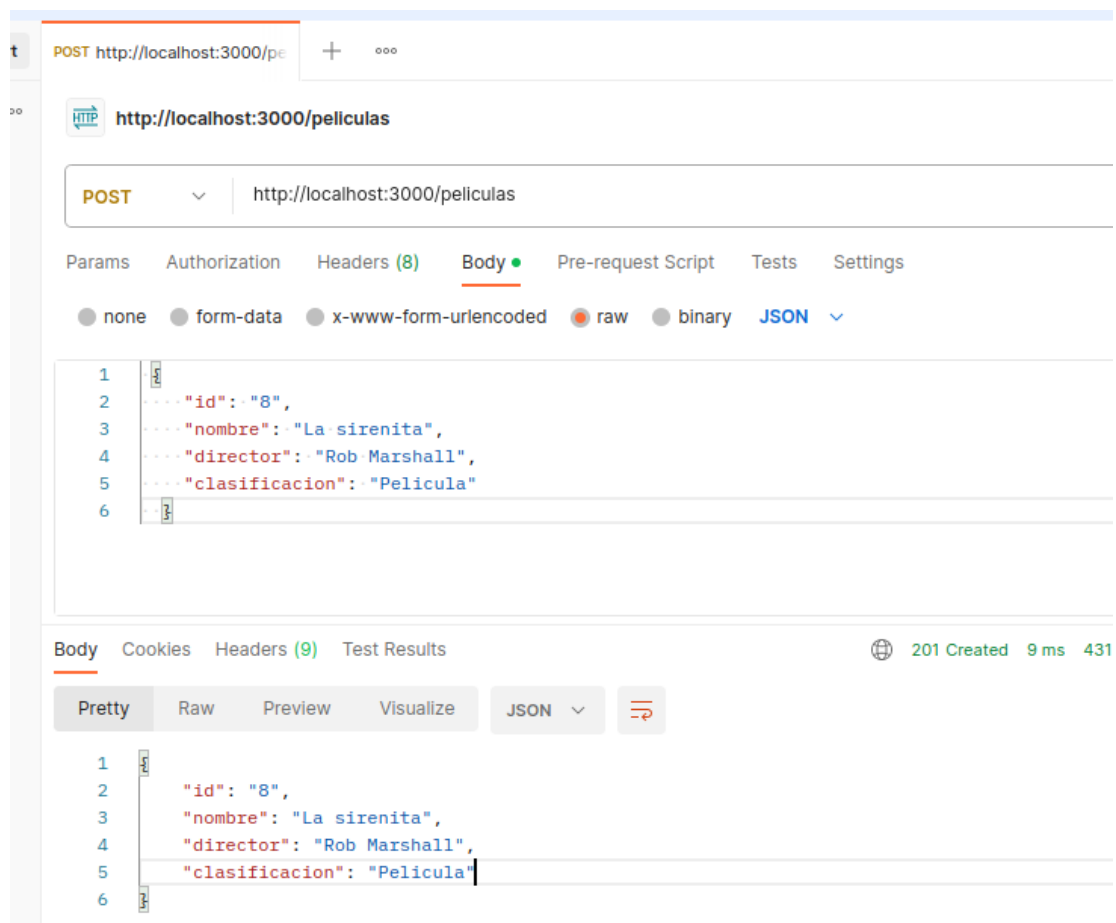
- GET de una entidad que no exista (id inválido, por ejemplo)

http://localhost:3000/peliculas/20



- POST de una entidad nueva.

En esta ocasión he hecho un POST agregando una nueva pelicula "La sirenita"

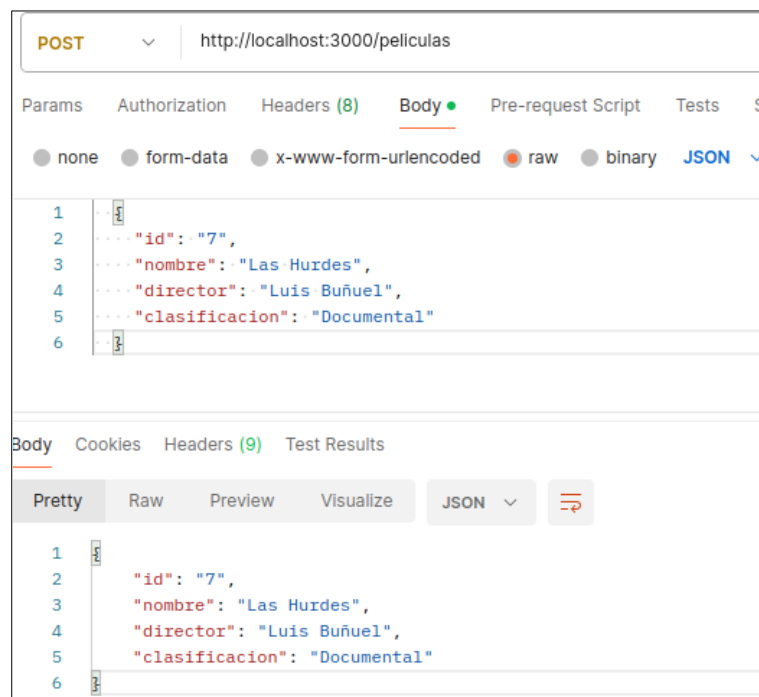


Se agrego correctamente:

```
{
  "id": "6",
  "nombre": "Forrest Gump",
  "director": "Robert Zemeckis",
  "clasificacion": "Comedia"
},
{
  "id": "7",
  "nombre": "Las Hurdes",
  "director": "Luis Buñuel",
  "clasificacion": "Documental"
},
{
  "id": "8",
  "nombre": "La sirenita",
  "director": "Rob Marshall",
  "clasificacion": "Película"
}
}
```

- POST de una entidad ya existente.

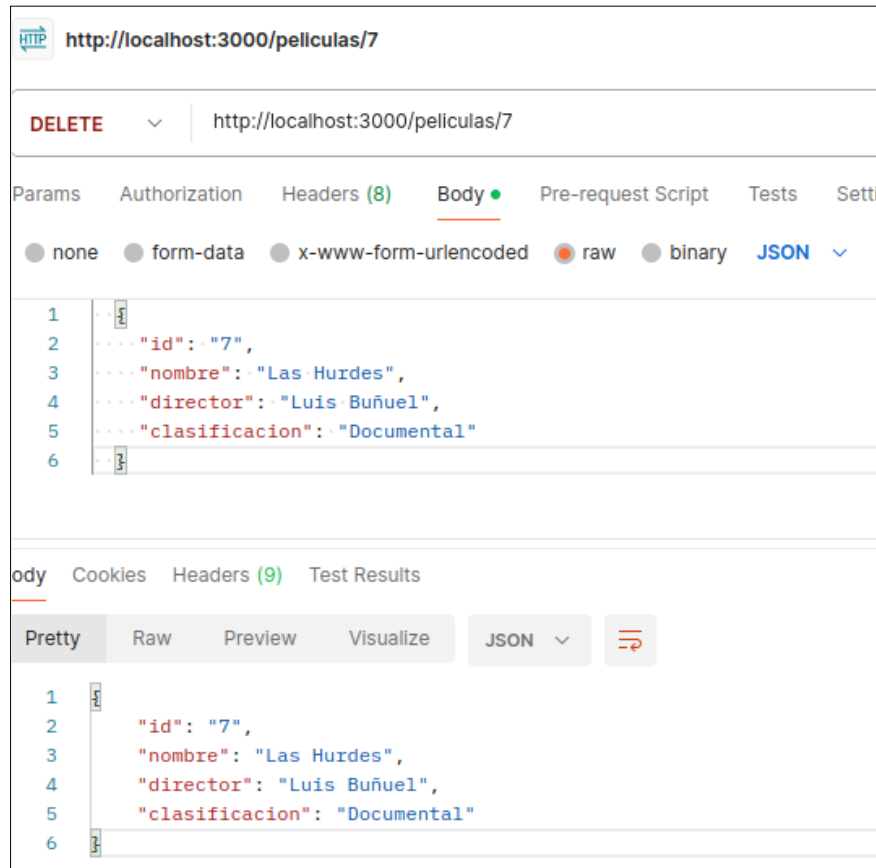
Si hago un POST de una película existente en este caso de la número 7, se duplica:



```
{
  "id": "7",
  "nombre": "Las Hurdes",
  "director": "Luis Buñuel",
  "clasificacion": "Documental"
},
{
  "id": "8",
  "nombre": "La sirenita",
  "director": "Rob Marshall",
  "clasificacion": "Película"
},
{
  "id": "7",
  "nombre": "Las Hurdes",
  "director": "Luis Buñuel",
  "clasificacion": "Documental"
}
}
```

- DEL de una entidad existente.

voy a borrar esa entidad haber que sucede:

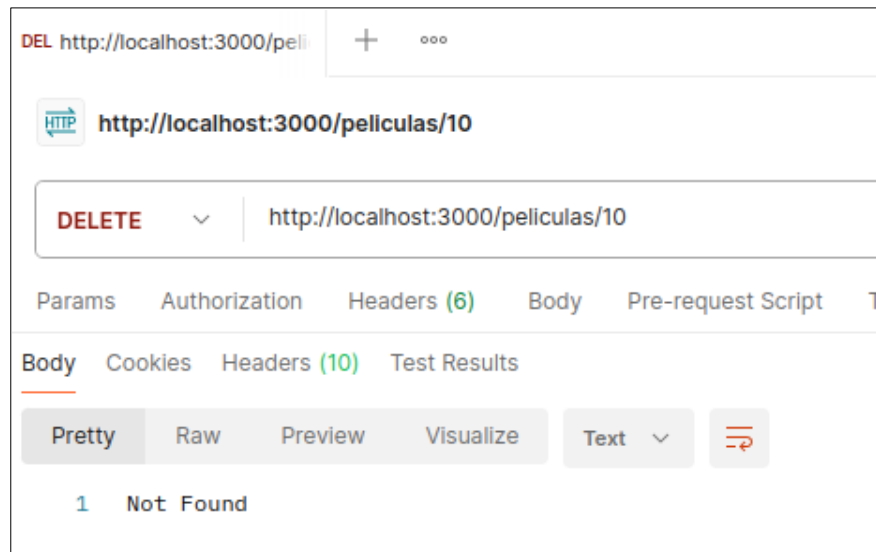


Lo que hace es borrar una a una las películas que tengan id :7.

```
{
  "id": "6",
  "nombre": "Forrest Gump",
  "director": "Robert Zemeckis",
  "clasificacion": "Comedia"
},
{
  "id": "8",
  "nombre": "La sirenita",
  "director": "Rob Marshall",
  "clasificacion": "Película"
}
```

- DEL de una entidad que no exista.

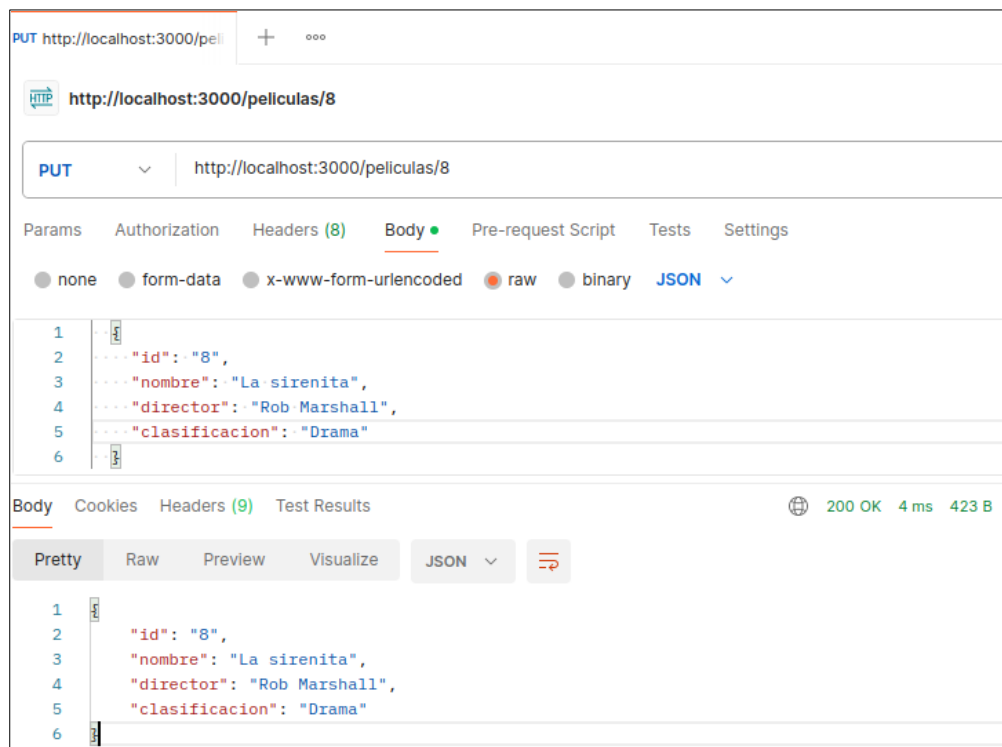
Me dice que no lo encuentra:



- PUT de una entidad existente.

En este caso actualizare la clasificacion de la sirenita ya que no es un valor correcto: lo pondre de "pelicula" a "Drama"

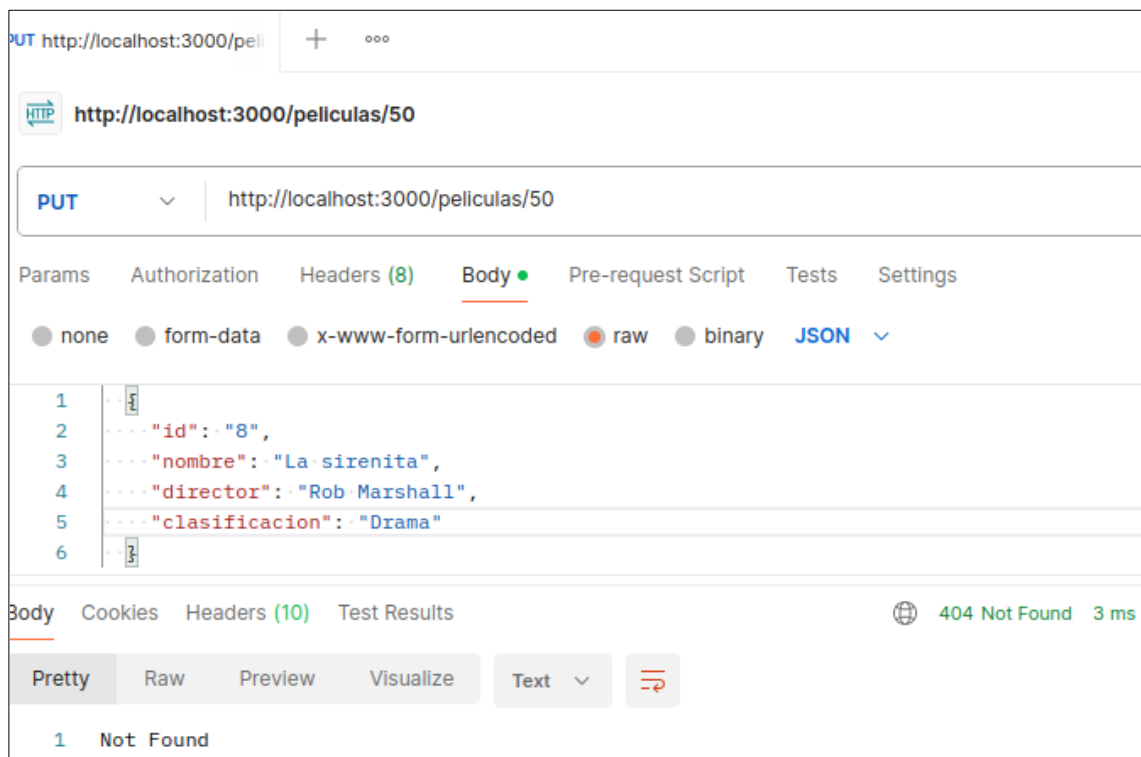
```
{
  "id": "8",
  "nombre": "La sirenita",
  "director": "Rob Marshall",
  "clasificacion": "Pelicula"
}
```




```
{
  "id": "7",
  "nombre": "Las Hurdes",
  "director": "Luis Buñuel",
  "clasificacion": "Documental"
},
{
  "id": "8",
  "nombre": "La sirenita",
  "director": "Rob Marshall",
  "clasificacion": "Drama"
}
}
```

- PUT de una entidad que no exista.

Si hago un PUT a una entidad que no exista no lo encuentra y por lo tanto no actualiza nada:

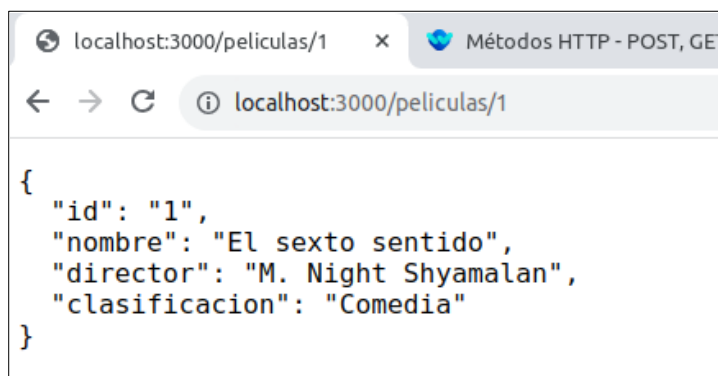
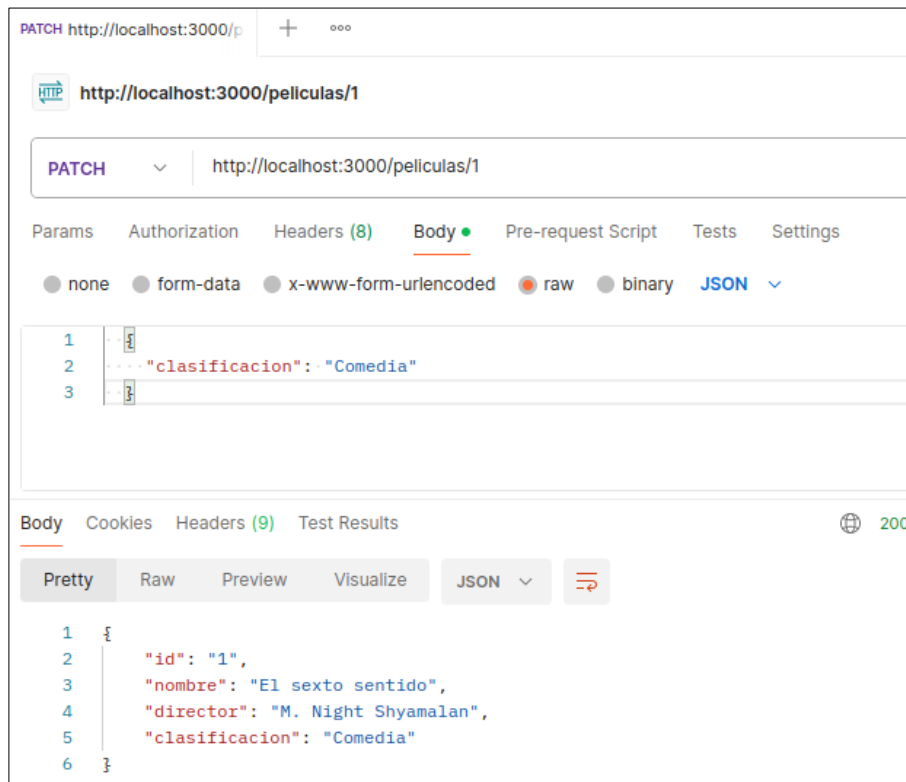


- PATCH de una entidad existente.

Partiendo de una pelicula con clasificacion Drama la actualizar a Comedia.

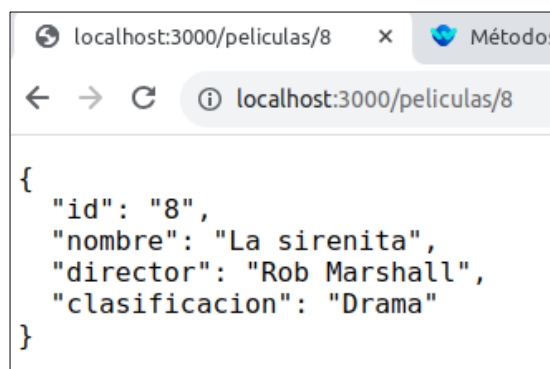
The screenshot shows a web browser window with the URL `localhost:3000/peliculas/1`. The response is a JSON object: `{ "id": "1", "nombre": "El sexto sentido", "director": "M. Night Shyamalan", "clasificacion": "Drama" }`.

El valor se actualizo correctamente:

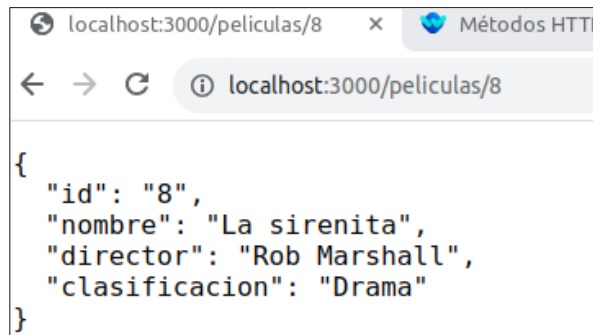
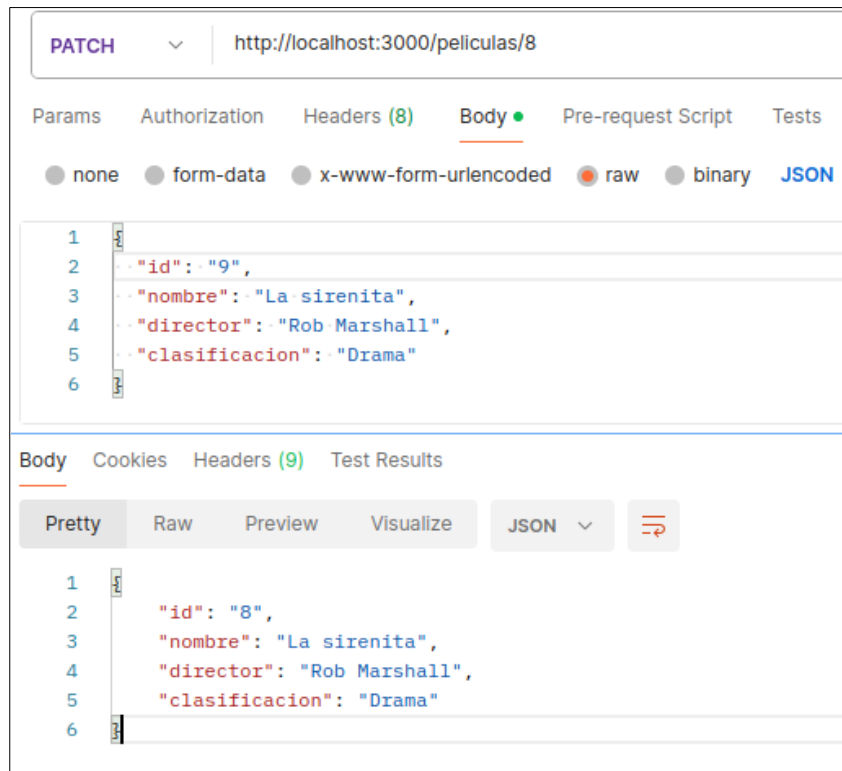


- PATCH de una entidad existente, pero provocando incoherencia entre el id del path y el pasado en json.

Partiendo de la sirenita con id 8 haremos un cambio a ver qué sucede:



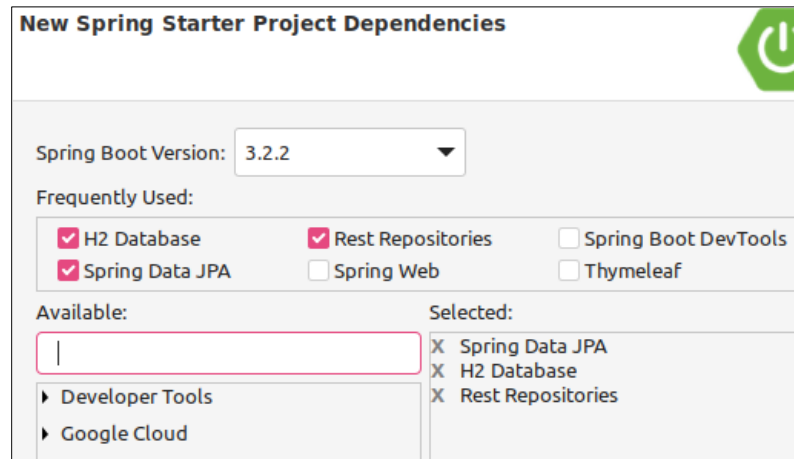
Lo que sucede es que si cambio el id de 8 a 9 . me dice que a actualizado correctamente, sin embargo, no modifica el id lo deja con el id anterior en este caso sigue siendo el id:8.



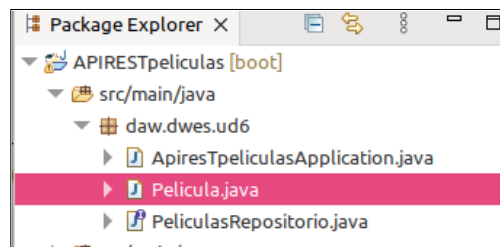
Apartado 2.- API con REST Repository.

Sigue este otro tutorial <https://spring.io/guides/gs/accessing-data-rest/> para implementar el mismo API del apartado anterior pero esta vez con SpringBoot. Tendrás que crear una clase para la entidad (ej: Pelicula.java) un interfaz repositorio y teniendo activas las dependencias JPA DATA y H2 automáticamente creará una base de datos en memoria.

Creo el proyecto con las dependencias correspondientes:



Creo la interface PeliculasRepositorio y la clase Peliculas.java:



```

1 package daw.dwes.ud6;
2
3 import jakarta.persistence.Entity;
4
5 @Entity
6 public class Pelicula {
7     @Id
8     @GeneratedValue(strategy = GenerationType.IDENTITY)
9     private Long id;
10    private String nombre;
11    private String director;
12    private String clasificacion;
13
14    public String getNombre() {
15        return nombre;
16    }
17
18    public void setNombre(String nombre) {
19        this.nombre = nombre;
20    }
21
22    public String getDirector() {
23        return director;
24    }
25
26    public void setDirector(String director) {
27        this.director = director;
28    }
29
30    public String getClasificacion() {
31        return clasificacion;
32    }
33
34    public void setClasificacion(String clasificacion) {
35        this.clasificacion = clasificacion;
36    }
37 }

```

```

ApiesTpeliculasApplication.java PeliculasRepositorio.java X
1 package daw.dwes.ud6;
2
3 import org.springframework.data.repository.PagingAndSortingRepository;
10 @RepositoryRestResource(collectionResourceRel = "peliculas", path = "peliculas")
11 public interface PeliculasRepositorio extends PagingAndSortingRepository<Pelicula,Long>, CrudRepository<Pelicula,Long> {
12
13     List<Pelicula> findByNombre(@Param("nombre") String nombre);
14
15 }
16

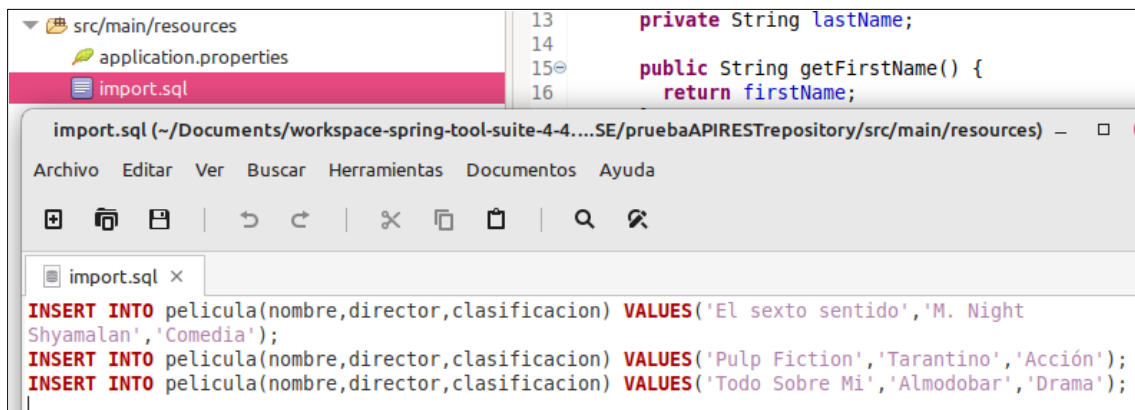
```

Puedes cargar la base de datos inicialmente creando un archivo import.sql en la carpeta resources con órdenes INSERT de la entidad (todo minúsculas y si en Java hay cambios a mayúscula en SQL se traducen a _) Ejemplo para entidad Persona con atributos firstName y lastName:

INSERT INTO person(first_name,last_name,id) VALUES('Javier','Puche',2);

INSERT INTO person(first_name,last_name,id) VALUES('Agustín','Aguilera',1);

podemos quitar los id y que los autogenera la base de datos si marcamos en Persona id como: @GeneratedValue(strategy = GenerationType.IDENTITY)



The screenshot shows an IDE with two tabs. The top tab is a Java class named `PeliculasRepositorio.java` with the following code:

```

13 private String lastName;
14
15 public String getFirstName() {
16     return firstName;

```

The bottom tab is an SQL file named `import.sql` located at `~/Documents/workspace-spring-tool-suite-4-4...SE/pruebaAPIRESTrepositorio/src/main/resources`. It contains three INSERT statements:

```

INSERT INTO pelicula(nombre,director,clasificacion) VALUES('El sexto sentido','M. Night Shyamalan','Comedia');
INSERT INTO pelicula(nombre,director,clasificacion) VALUES('Pulp Fiction','Tarantino','Acción');
INSERT INTO pelicula(nombre,director,clasificacion) VALUES('Todo Sobre Mi','Almodobar','Drama');

```

Realiza las mismas pruebas que en el apartado anterior y las de búsqueda y filtrado del tutorial.

- GET de la lista de entidades.

```

4      {
5        "nombre": "El sexto sentido",
6        "director": "M. Night Shyamalan",
7        "clasificacion": "Comedia",
8        "_links": {
9          "self": {
10             "href": "http://localhost:8080/peliculas/1"
11           },
12          "pelicula": {
13             "href": "http://localhost:8080/peliculas/1"
14           }
15        }
16      },
17      {
18        "nombre": "Pulp Fiction",
19        "director": "Tarantino",
20        "clasificacion": "Acción",
21        "_links": {
22          "self": {
23             "href": "http://localhost:8080/peliculas/2"
24           },
25          "pelicula": {
26             "href": "http://localhost:8080/peliculas/2"
27           }
28        }
29      }
30    ]
  
```

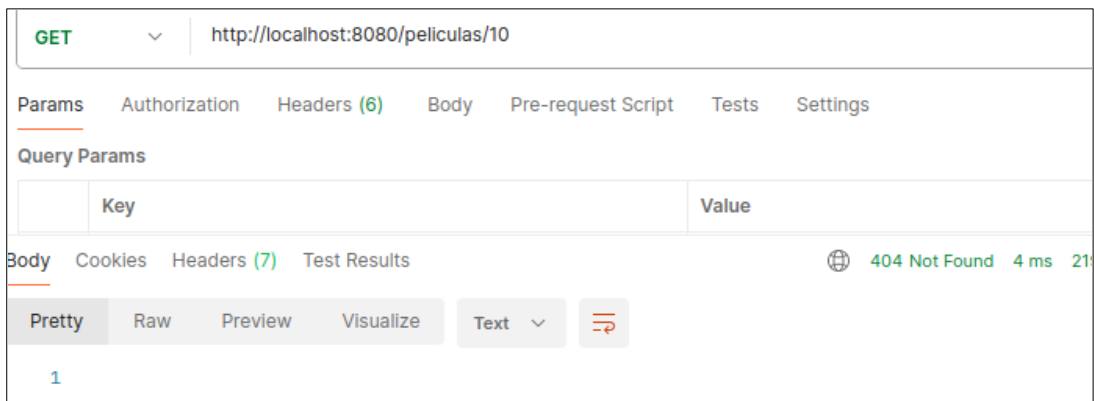
- GET de una entidad existente.

```

1      {
2        "nombre": "El sexto sentido",
3        "director": "M. Night Shyamalan",
4        "clasificacion": "Comedia",
5        "_links": {
6          "self": {
7             "href": "http://localhost:8080/peliculas/1"
8           },
9          "pelicula": {
10             "href": "http://localhost:8080/peliculas/1"
11           }
12        }
13      }
  
```

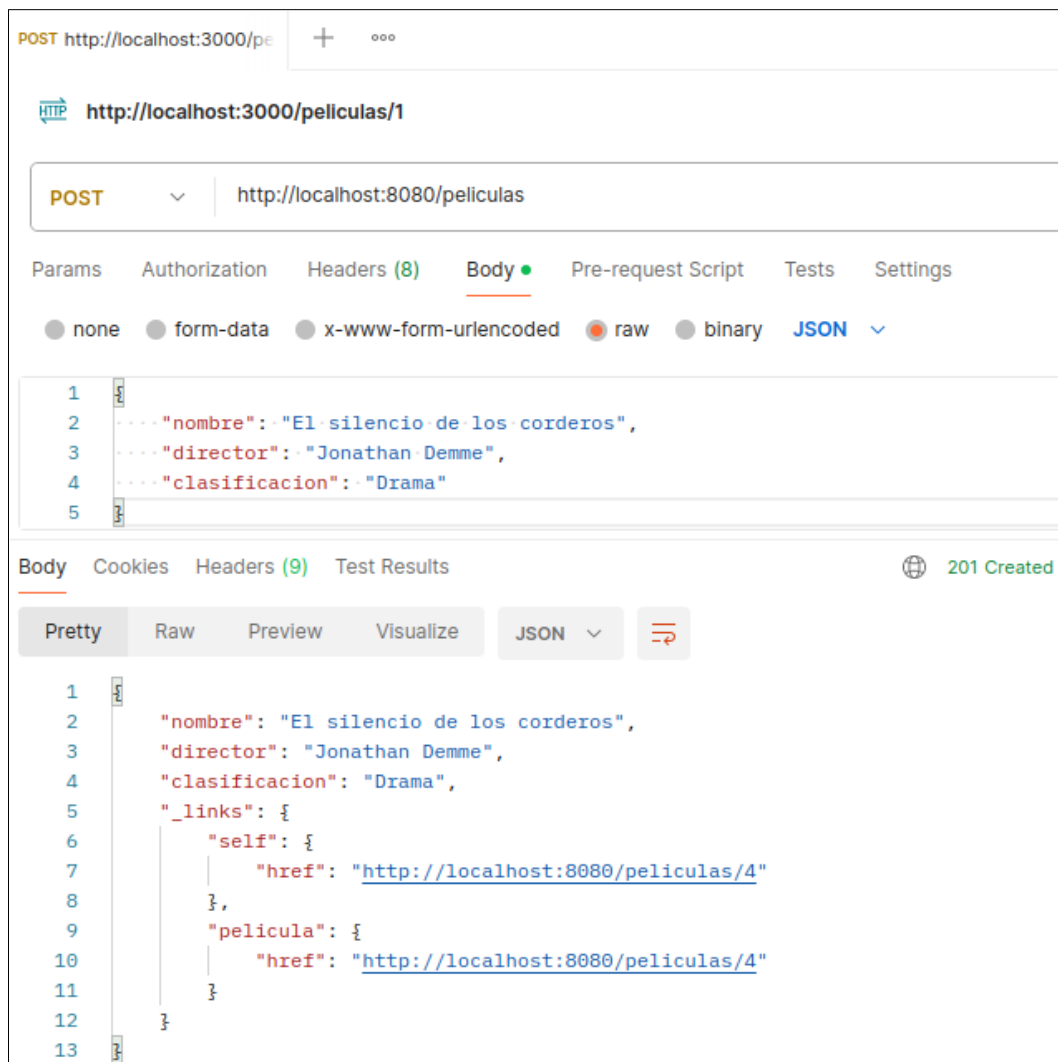
- GET de una entidad que no exista (id inválido, por ejemplo)

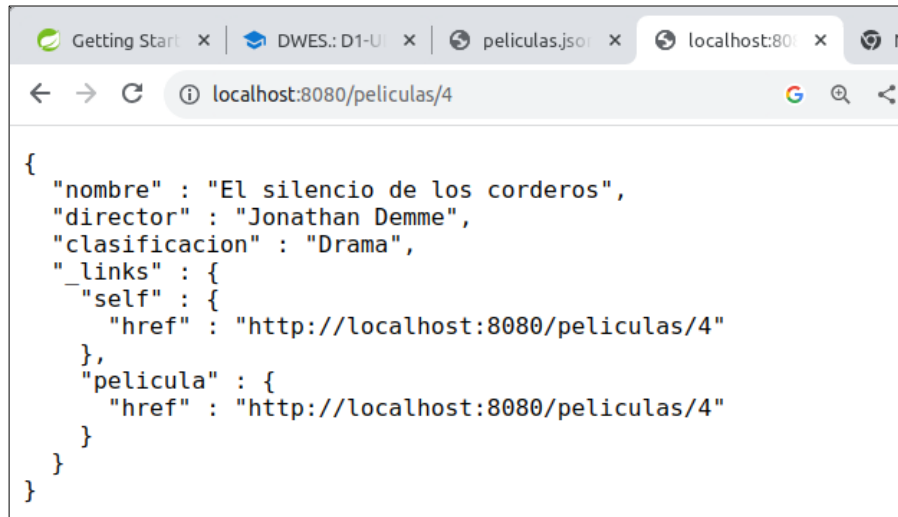
Me aparece el error 404 not found pero ningun mensaje en el body:



- POST de una entidad nueva.

Se crea correctamente:





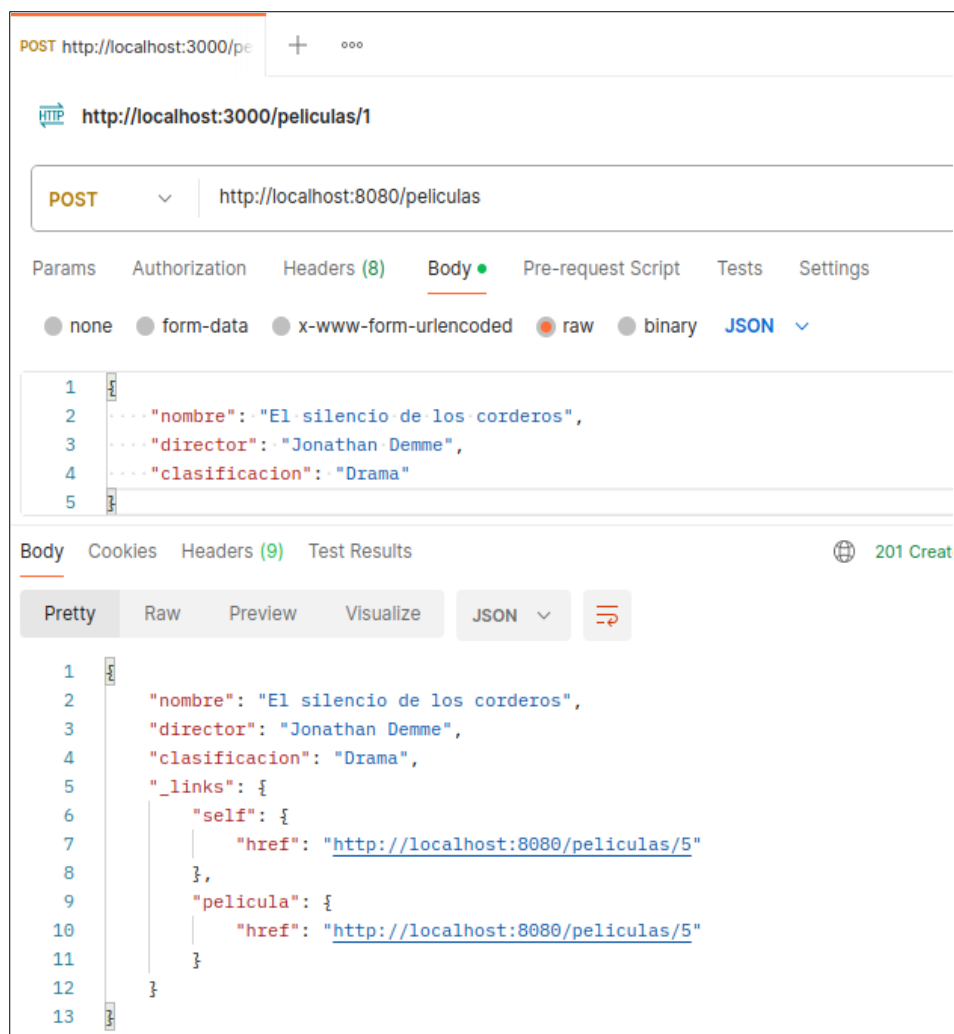
```

{
  "nombre" : "El silencio de los corderos",
  "director" : "Jonathan Demme",
  "clasificacion" : "Drama",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/peliculas/4"
    },
    "pelicula" : {
      "href" : "http://localhost:8080/peliculas/4"
    }
  }
}

```

- POST de una entidad ya existente.

si volvemos a meter la misma pelicula se crea y se duplica con otra id en este caso es la id:5



POST http://localhost:3000/pe + ...

HTTP http://localhost:3000/peliculas/1

POST http://localhost:8080/peliculas

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

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

```

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

```

Body Cookies Headers (9) Test Results 201 Creat

Pretty Raw Preview Visualize JSON

```

1 {
2   "nombre": "El silencio de los corderos",
3   "director": "Jonathan Demme",
4   "clasificacion": "Drama",
5   "_links": {
6     "self": {
7       "href": "http://localhost:8080/peliculas/5"
8     },
9     "pelicula": {
10      "href": "http://localhost:8080/peliculas/5"
11    }
12  }
13 }

```



```

    },
    "pelicula" : {
      "href" : "http://localhost:8080/peliculas/3"
    }
  }, {
    "nombre" : "El silencio de los corderos",
    "director" : "Jonathan Demme",
    "clasificacion" : "Drama",
    "_links" : {
      "self" : {
        "href" : "http://localhost:8080/peliculas/4"
      },
      "pelicula" : {
        "href" : "http://localhost:8080/peliculas/4"
      }
    }
  }, {
    "nombre" : "El silencio de los corderos",
    "director" : "Jonathan Demme",
    "clasificacion" : "Drama",
    "_links" : {
      "self" : {
        "href" : "http://localhost:8080/peliculas/5"
      },
      "pelicula" : {
        "href" : "http://localhost:8080/peliculas/5"
      }
    }
  }
}
]

```

- DEL de una entidad existente.

la película con id 5 ha sido eliminada correctamente:

DEL http://localhost:3000/peli + ...

HTTP http://localhost:3000/peliculas/1

DELETE http://localhost:8080/peliculas/5

Params Authorization Headers (6) **Body** Pre-request Script Tests Settings

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

1

Body Cookies Headers (8) Test Results 200 OK 12 ms

Pretty Raw Preview Visualize JSON

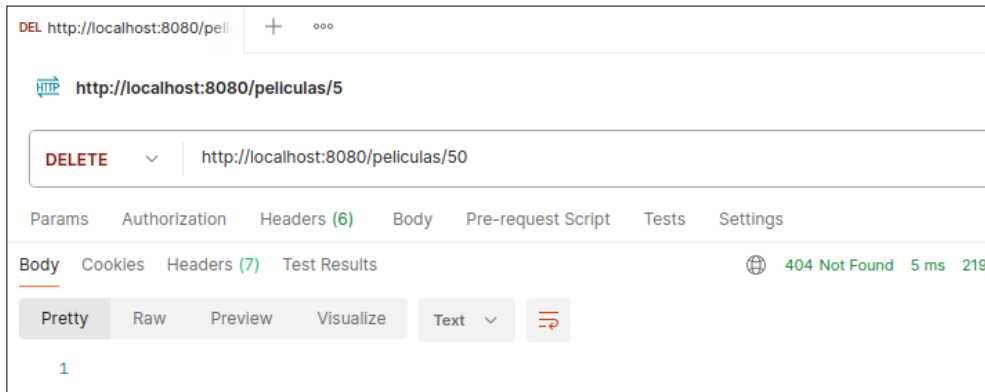
```

1
2  "nombre": "El silencio de los corderos",
3  "director": "Jonathan Demme",
4  "clasificacion": "Drama",
5  "_links": {
6    "self": {
7      "href": "http://localhost:8080/peliculas/5"
8    },
9    "pelicula": {
10     "href": "http://localhost:8080/peliculas/5"
11   }
12 }
13

```

- DEL de una entidad que no exista.

Me dice que no la encuentra:



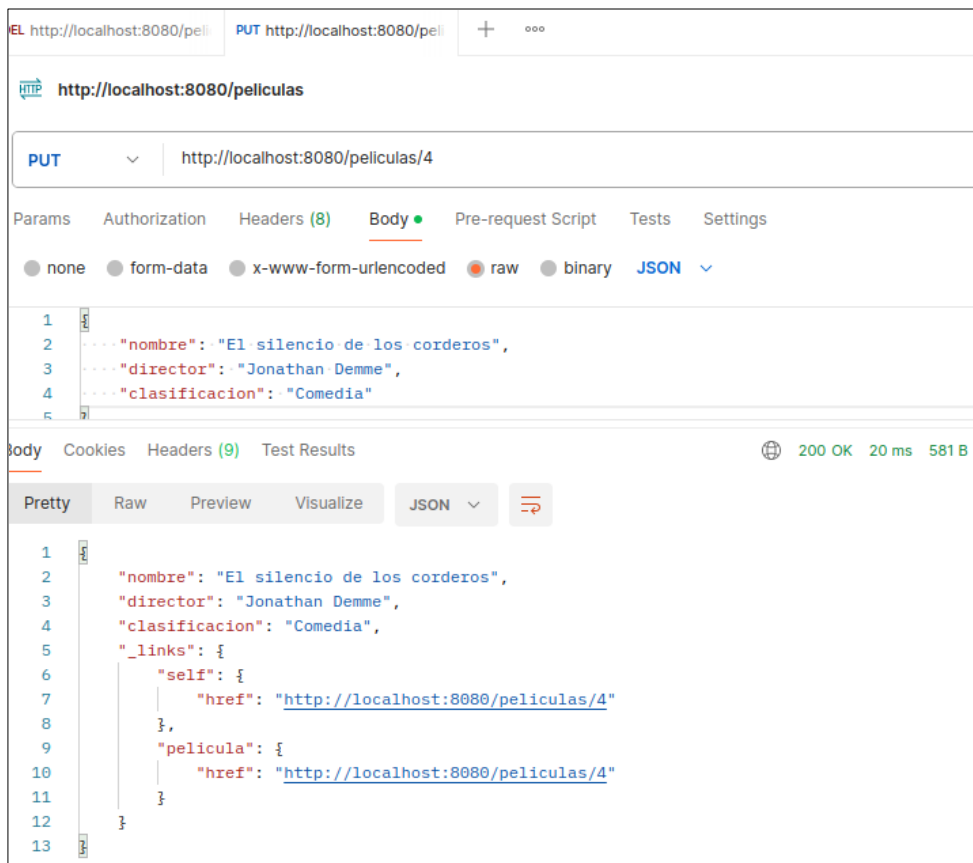
- PUT de una entidad existente.

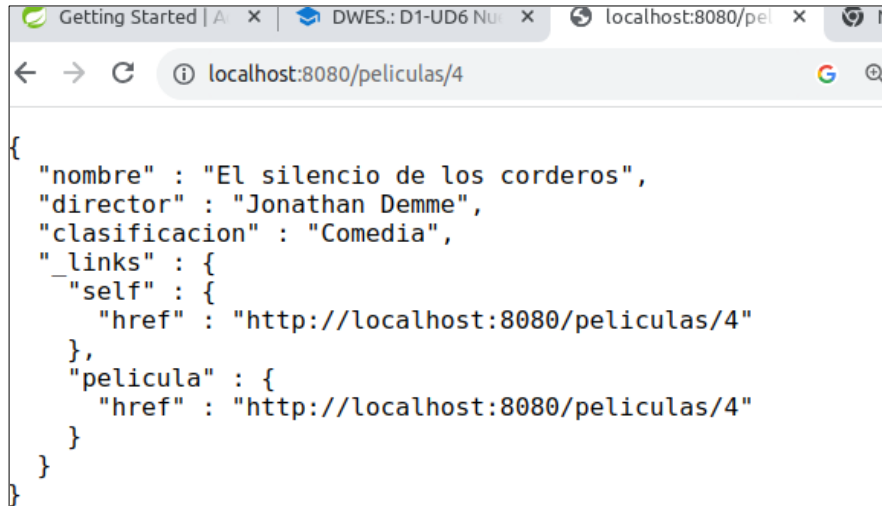
Me realiza el cambio correctamente he cambiado de drama a comedia:

```

{
  "nombre": "El silencio de los corderos",
  "director": "Jonathan Demme",
  "clasificacion": "Drama",
  "_links": {
    "self": {
      "href": "http://localhost:8080/peliculas/4"
    },
    "pelicula": {
      "href": "http://localhost:8080/peliculas/4"
    }
  }
}

```

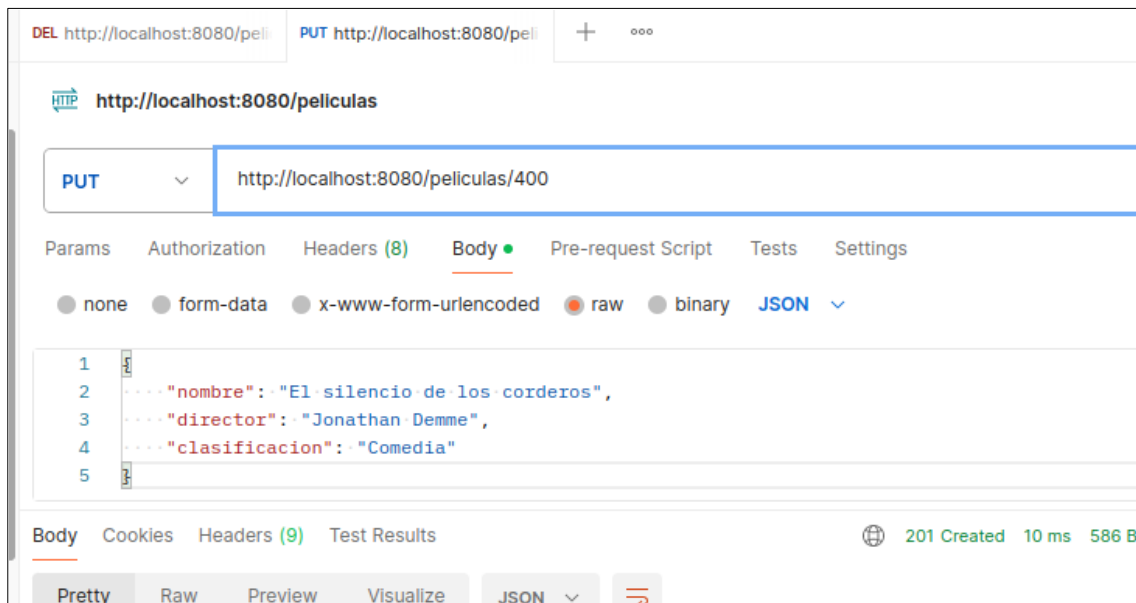




```
{
  "nombre" : "El silencio de los corderos",
  "director" : "Jonathan Demme",
  "clasificacion" : "Comedia",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/peliculas/4"
    },
    "pelicula" : {
      "href" : "http://localhost:8080/peliculas/4"
    }
  }
}
```

- PUT de una entidad que no exista.

Si hago la actualización de una entidad que no exista en vez de modificar me crea un nuevo objeto con los datos introducidos:



```
{
  "nombre" : "El silencio de los corderos",
  "director" : "Jonathan Demme",
  "clasificacion" : "Comedia",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/peliculas/6"
    },
    "pelicula" : {
      "href" : "http://localhost:8080/peliculas/6"
    }
  }
}
```

- PATCH de una entidad existente.

Cambiare datos de la película id:1

```

← → ↻ ⓘ localhost:8080/peliculas/1

{
  "nombre": "El sexto sentido",
  "director": "M. Night Shyamalan",
  "clasificacion": "Drama",
  "_links": {
    "self": {
      "href": "http://localhost:8080/peliculas/1"
    },
    "pelicula": {
      "href": "http://localhost:8080/peliculas/1"
    }
  }
}

```

Se actualiza correctamente:

PATCH

http://localhost:8080/peliculas/1

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

```

1 {
2   "nombre": "El quinto sentido",
3   "director": "M. Night Shyamalan",
4   "clasificacion": "Comedia"
5 }

```

body

Cookies

Headers (8)

Test Results

200 OK

90 ms

542 B

Pretty

Raw

Preview

Visualize

JSON

```

1 {
2   "nombre": "El quinto sentido",
3   "director": "M. Night Shyamalan",
4   "clasificacion": "Comedia",
5   "_links": {
6     "self": {
7       "href": "http://localhost:8080/peliculas/1"
8     },
9     "pelicula": {
10      "href": "http://localhost:8080/peliculas/1"
11    }
12  }
13 }

```

← → ↻ ⓘ localhost:8080/peliculas/1

```

{
  "nombre": "El quinto sentido",
  "director": "M. Night Shyamalan",
  "clasificacion": "Comedia",
  "_links": {
    "self": {
      "href": "http://localhost:8080/peliculas/1"
    },
    "pelicula": {
      "href": "http://localhost:8080/peliculas/1"
    }
  }
}

```

- PATCH de una entidad existente, pero provocando incoherencia entre el id del path y el pasado en json.

Se muestra que la solicitud se ha hecho correctamente, sin embargo, no se modifica el id. En este caso cambie a id:2 pero no se cambió nada.

Overview | petición: GET New Re • GET https:// • GET https:// • PATCH http:// • + ▾

HTTP http://localhost:8080/peliculas/1

PATCH ▾ http://localhost:8080/peliculas/1

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

● none ● form-data ● x-www-form-urlencoded ● **raw** ● binary ● GraphQL **JSON** ▾

```

1 {
2   "nombre": "El quinto sentido",
3   "director": "M. Night Shyamalan",
4   "clasificacion": "Comedia",
5   "_links": {
6     "self": {
7       "href": "http://localhost:8080/peliculas/2"
8     },
9     "pelicula": {
10      "href": "http://localhost:8080/peliculas/2"
11    }
12  }
13 }

```

Body Cookies Headers (8) Test Results **200 OK** 14 m

Pretty Raw Preview Visualize **JSON** ▾

```

1 {
2   "nombre": "El quinto sentido",
3   "director": "M. Night Shyamalan",
4   "clasificacion": "Comedia",
5   "_links": {
6     "self": {
7       "href": "http://localhost:8080/peliculas/1"
8     },
9     "pelicula": {
10      "href": "http://localhost:8080/peliculas/1"
11    }
12  }
13 }

```

← → ↻ ⓘ localhost:8080/peliculas/1

```

{
  "nombre" : "El quinto sentido",
  "director" : "M. Night Shyamalan",
  "clasificacion" : "Comedia",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/peliculas/1"
    },
    "pelicula" : {
      "href" : "http://localhost:8080/peliculas/1"
    }
  }
}

```

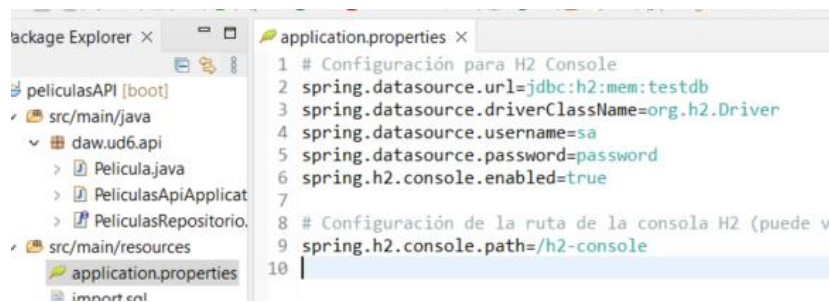
En los logs iniciales de arranque de SpringBoot puedes ver el acceso a la consola H2 para inspeccionar la base de datos en memoria. En el próximo tema veremos cómo activar que muestre todas las queries que realiza la aplicación para depuración.

```
Starting PeliculasApiApplication using Java 17.0.9 with PID 19436 (C:\Users\CAROLINA\
No active profile set, falling back to 1 default profile: "default"
Bootstrapping Spring Data JPA repositories in DEFAULT mode.
Finished Spring Data repository scanning in 32 ms. Found 1 JPA repository interface
Tomcat initialized with port 8080 (http)
Starting service [Tomcat]
Starting Servlet engine: [Apache Tomcat/10.1.18]
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 893 ms
HikariPool-1 - Starting...
HikariPool-1 - Added connection conn0: url=jdbc:h2:mem:testdb user=SA
HikariPool-1 - Start completed.
H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:testdb'
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 en
Initialized JPA EntityManagerFactory for persistence unit 'default'
spring.jpa.open-in-view is enabled by default. Therefore, database queries may be pe
Tomcat started on port 8080 (http) with context path ''
Started PeliculasApiApplication in 3.086 seconds (process running for 3.655)
```

Para poder conectarme a la consola H2, en la ruta:

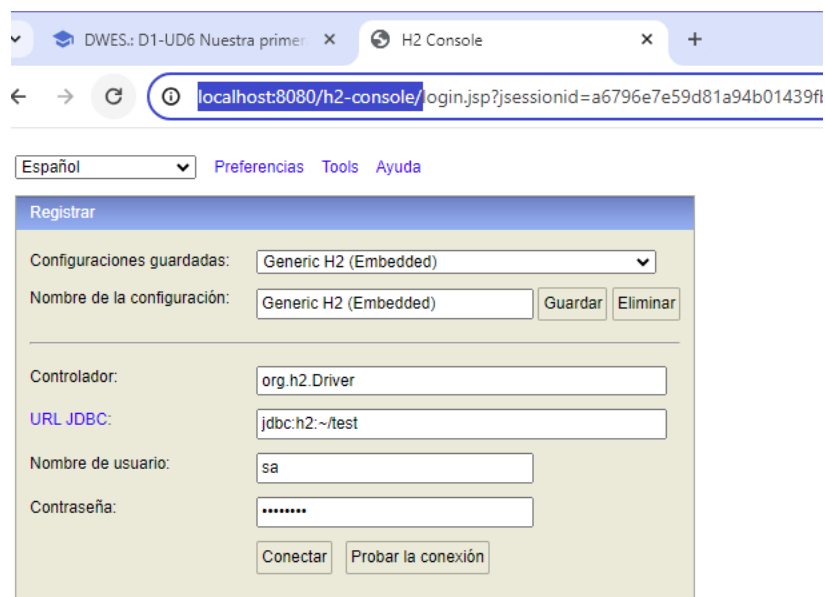
<http://localhost:8080/h2-console/>

tuve que agregar el siguiente código en application.properties:

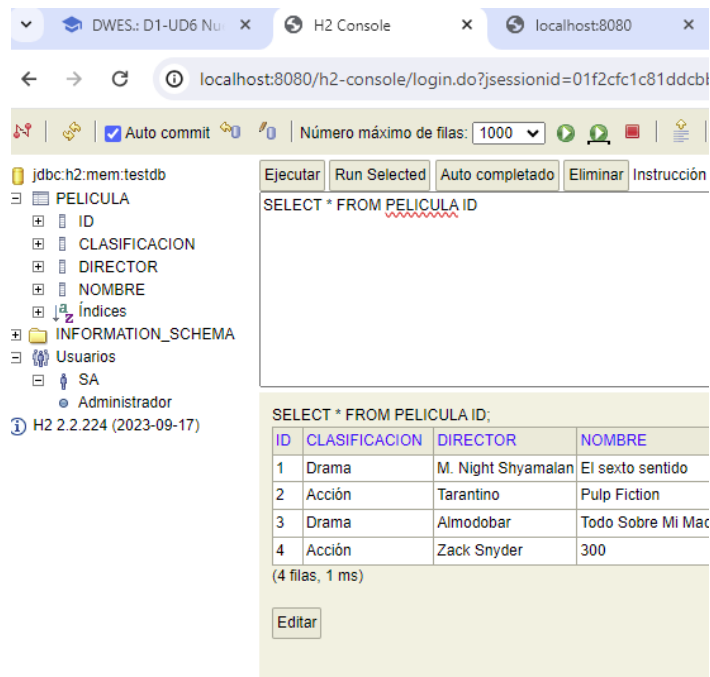


```
1 # Configuración para H2 Console
2 spring.datasource.url=jdbc:h2:mem:testdb
3 spring.datasource.driverClassName=org.h2.Driver
4 spring.datasource.username=sa
5 spring.datasource.password=password
6 spring.h2.console.enabled=true
7
8 # Configuración de la ruta de la consola H2 (puede v
9 spring.h2.console.path=/h2-console
10
```

con ello conseguí entrar a la url y me apareció la consola:



en url JDBC modifique con la ruta: jdbc:h2:mem:testdb y contraseña: password



Compara las pruebas del apartado 1 con las de este apartado ¿devuelven los mismos códigos en los mismos casos, el mismo cuerpo de mensaje?

Ejercicio 1:

Operación	Resultado Esperado 1	Resultado Esperado 2	Resultado Esperado 3
GET	200 OK	200 OK	404 Not Found
POST	201 Created	Se duplica	
DELETE	200 OK	404 Not Found	
PUT	200 OK	404 Not Found	
PATCH	200 OK	200 OK pero no modifica id	

Ejercicio 2:

Operación	Resultado Esperado 1	Resultado Esperado 2	Resultado Esperado 3
GET	200 OK	200 OK	404 Not Found
POST	201 Created	Se duplica	
DELETE	200 OK	404 Not Found	
PUT	200 OK	201 Created Crea nuevo objeto	
PATCH	200 OK	200 OK pero no modifica id	

Ambas nos dan resultados iguales según los códigos de errores o verificado, pero, solo en el PUT en el apartado 2 de hacer un PUT de una entidad no existente dan resultados diferentes.

Mientras que en el ejercicio 1 nos dice que no encuentra la entidad en el ejercicio 2 nos crea un nuevo objeto con los valores json introducidos.

Otra diferencia es que cuando en el ejercicio 1 se encuentra con errores 404 Not found nos muestra en el body un mensaje “not found” mientras que en el ejercicio 2 cuando nos aparece un error 404 no muestra nada.

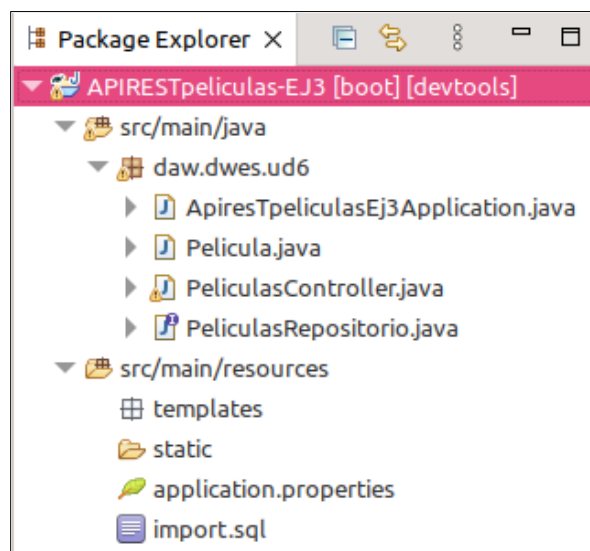
Apartado 3. (extra)API REST con @RestController (web) pero sin RestRepository.

Vamos a reimplementar la misma API REST del apartado anterior, pero sin la dependencia REST Repository. Tendremos que crear nosotros los puntos de acceso con un @RestController (dependencia Spring Web) Puedes basarte en el controlador de esta API: <https://github.com/joseluisgs/springboot-profesores-madrid-2022-2023/tree/main/05-SpringWeb> En clase explicaré el porqué de tantos paquetes distintos, para nuestra primera API no nos harán falta. Fíjate que las respuestas son de tipo ResponseEntity para poder encapsular tanto el código HTTP de respuestas como el contenido JSON de la misma, algo que no pasaría si solo devolvemos un objeto (el JSON del mismo)

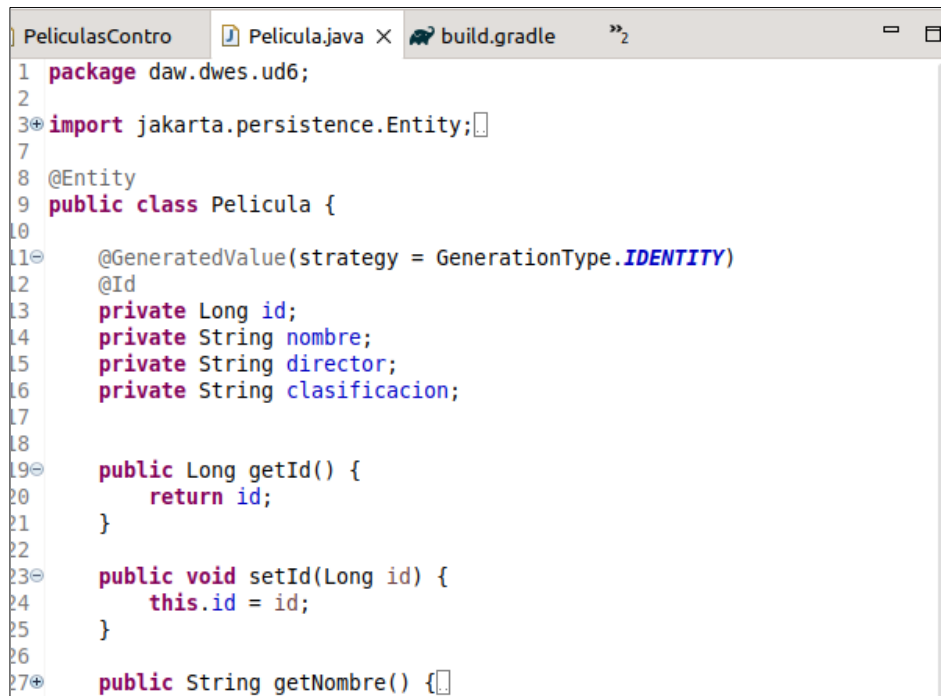
Fíjate que este ejemplo crea un repositorio basado en una colección. Si prefieres usar la dependencia H2 + JPA y simplemente declarar la interfaz del repositorio, es lo que haremos intensivamente en el tema 5.

Se trata de realizar nosotros el código de capturar las peticiones y elaborar las respuestas para tener un mayor control en caso de que tuviéramos que hacer más operaciones y no solamente el simple mantenimiento del repositorio.

Comencé creando el proyecto APIRESTpelículas-EJ3 con las dependencias JPA, H2, Spring web. presenta la siguiente estructura:



Recupere el código del objeto "Película" película.java del apartado anterior:

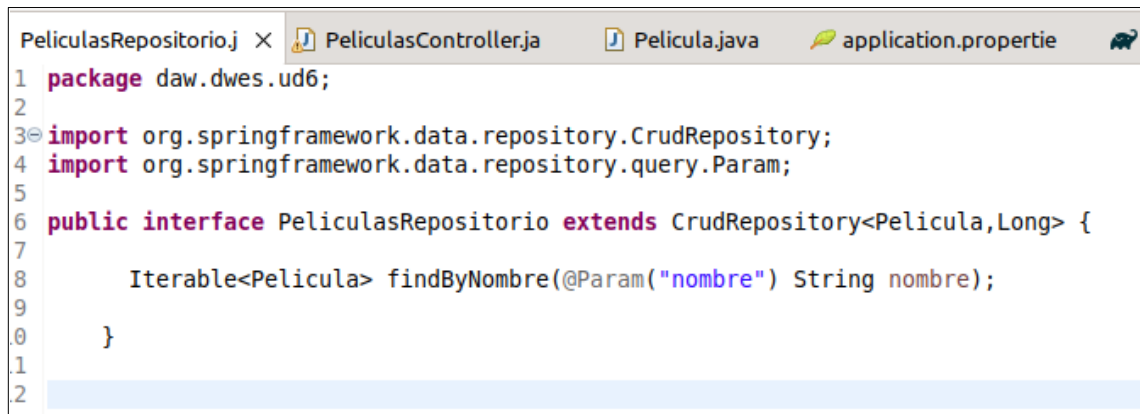


```

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

```

En la interface PeliculasRepositorio.java se hicieron las siguientes modificaciones:



```

1 package daw.dwes.ud6;
2
3 import org.springframework.data.repository.CrudRepository;
4 import org.springframework.data.repository.query.Param;
5
6 public interface PeliculasRepositorio extends CrudRepository<Película, Long> {
7
8     Iterable<Película> findByNombre(@Param("nombre") String nombre);
9
10 }

```

esta interfaz define un método personalizado (findByNombre) que extiende la funcionalidad básica proporcionada por CrudRepository. Este método se utiliza para realizar consultas específicas en la base de datos para recuperar películas por su nombre. El uso de @Param("nombre") ayuda a vincular el parámetro proporcionado al valor que se utilizará en la consulta

Las operaciones básicas que CrudRepository proporciona son las siguientes:

Guardar (save), Buscar por ID (findById), Eliminar por ID (deleteById), Obtener todos (findAll), Contar (count), Eliminar entidad (delete), Eliminar todos (deleteAll).

Cree un controlador "PelículasController":



```

1 package daw.dwes.ud6;
2
3 import org.springframework.beans.factory.annotation.Autowired;
14
15 @RestController
16 @RequestMapping(value = "/películas")
17 public class PelículasController {
18
19     private final PelículasRepositorio películasRepositorio;
20
21     @Autowired
22     public PelículasController(PelículasRepositorio películasRepositorio) {
23         this.películasRepositorio = películasRepositorio;
24     }
25
26     @GetMapping
27     public ResponseEntity<Iterable<Película>> getPelículas() {
28         return ResponseEntity.ok(películasRepositorio.findAll());
29     }
30
31     @GetMapping("/{id}")
32     public ResponseEntity<Película> getPelícula(@PathVariable(name = "id") Long id) {
33         return ResponseEntity.ok(películasRepositorio.findById(id).get());
34     }
35
36     @PostMapping
37     public ResponseEntity<Película> createPelícula(@RequestBody Película película) {
38         return ResponseEntity.status(HttpStatus.CREATED).body(películasRepositorio.save(película));
39     }
40
41     @PutMapping("/{id}")
42     public ResponseEntity<Película> updatePelícula(@PathVariable(name = "id") Long id, Película película) {
43         return ResponseEntity.ok(películasRepositorio.save(película));
44     }
45
46     @DeleteMapping("/{id}")
47     public ResponseEntity<Void> deletePelícula(@PathVariable(name = "id") Long id) {
48         películasRepositorio.deleteById(id);
49         return ResponseEntity.noContent().build();
50     }
51 }

```

Este código define un controlador (**PelículasController**) en Spring Boot para gestionar operaciones CRUD (Create, Read, Update, Delete) relacionadas con entidades **Película**. Aquí hay una explicación concisa de cada parte:

- **@RestController**: Anotación que indica que esta clase es un controlador de Spring MVC y que cada método de la clase devuelve directamente el objeto a ser serializado en el cuerpo de la respuesta HTTP.
- **@RequestMapping(value = "/películas")**: Mapea todas las solicitudes que comienzan con "/películas" a este controlador.
- **@Autowired**: Anotación para la inyección de dependencias, utilizada en el constructor para inyectar una instancia de **PelículasRepositorio**.
- **@GetMapping**: Anotación para mapear solicitudes HTTP GET.
- **@PostMapping**: Anotación para mapear solicitudes HTTP POST.

- **@PutMapping**: Anotación para mapear solicitudes HTTP PUT.
- **@DeleteMapping**: Anotación para mapear solicitudes HTTP DELETE.
- **@PatchMapping**: Anotación para mapear solicitudes HTTP PATCH.

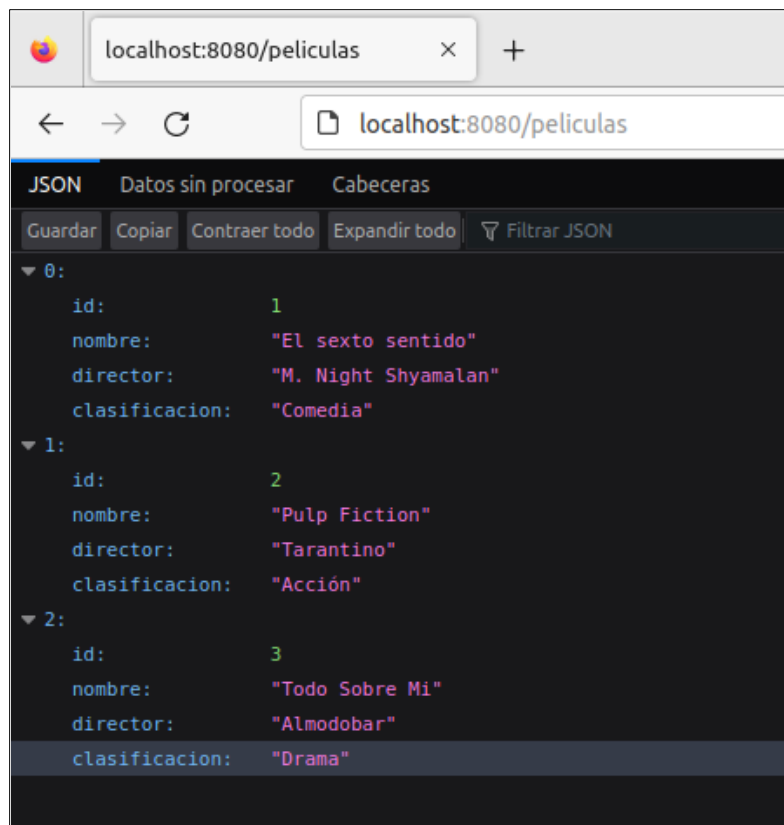
En cuanto a los métodos:

- **getPelículas**: Maneja las solicitudes GET para recuperar todas las películas.
- **getPelícula**: Maneja las solicitudes GET para recuperar una película por su identificador.
- **createPelícula**: Maneja las solicitudes POST para crear una nueva película.
- **updatePelícula**: Maneja las solicitudes PUT para actualizar una película existente.
- **deletePelícula**: Maneja las solicitudes DELETE para eliminar una película por su identificador.
- **patchPelícula**: Maneja las solicitudes PATCH para realizar actualizaciones parciales en una película existente.

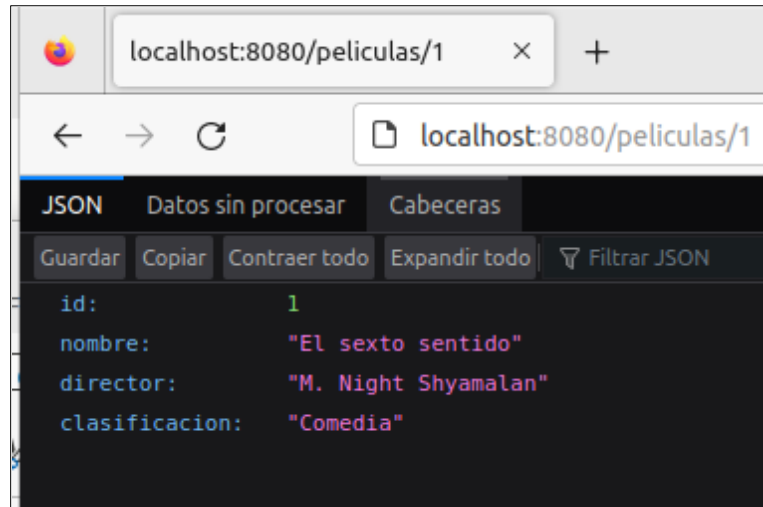
Cada método utiliza el repositorio (**películasRepositorio**) para interactuar con la base de datos y devuelve respuestas HTTP adecuadas, como **ResponseEntity**, con los resultados de las operaciones. Además, se utiliza **@RequestBody** para mapear los datos del cuerpo de la solicitud a objetos **Película**, y **@PathVariable** para obtener valores de la URL.

Pruebas:

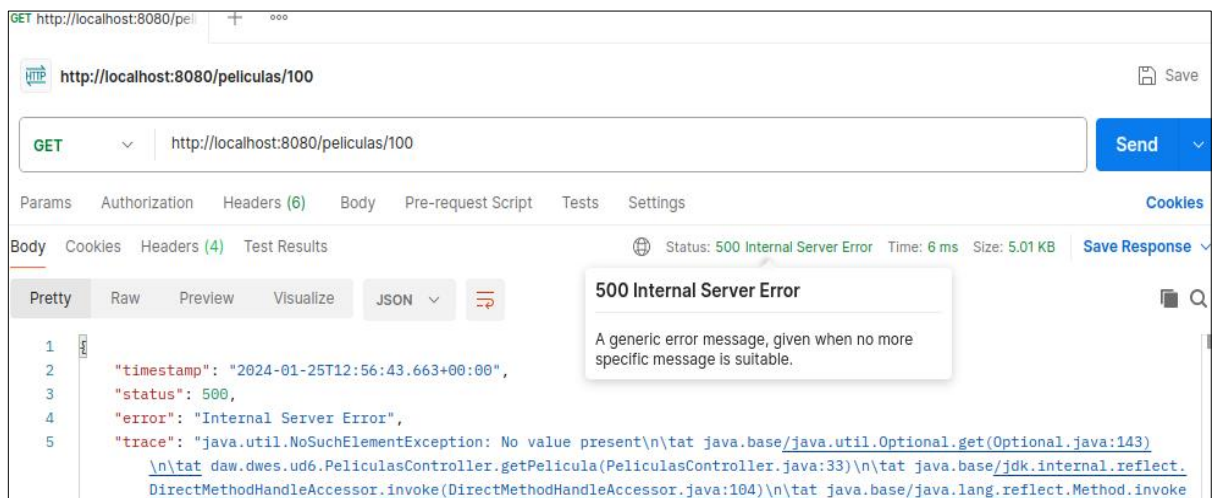
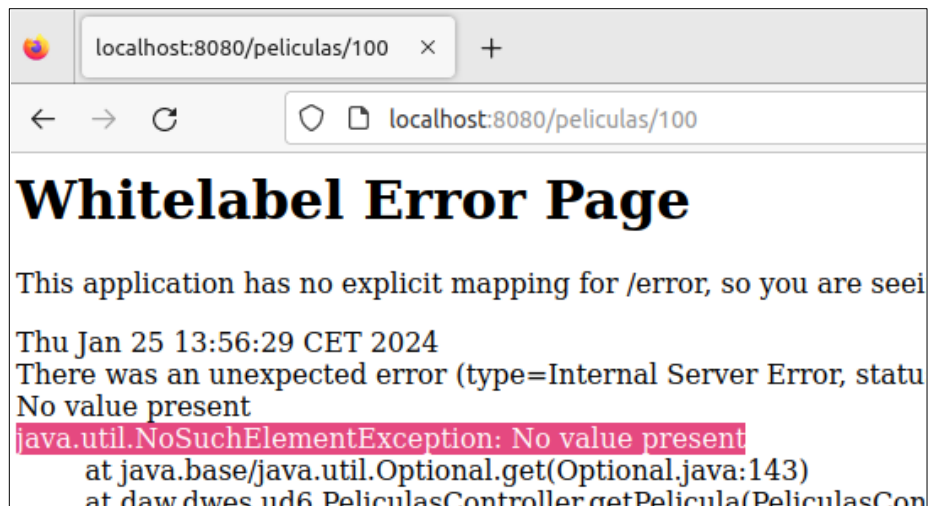
- GET de la lista de entidades.



- GET de una entidad existente.

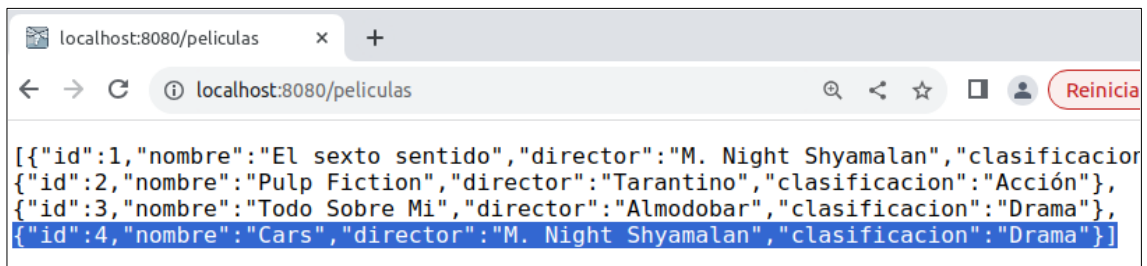
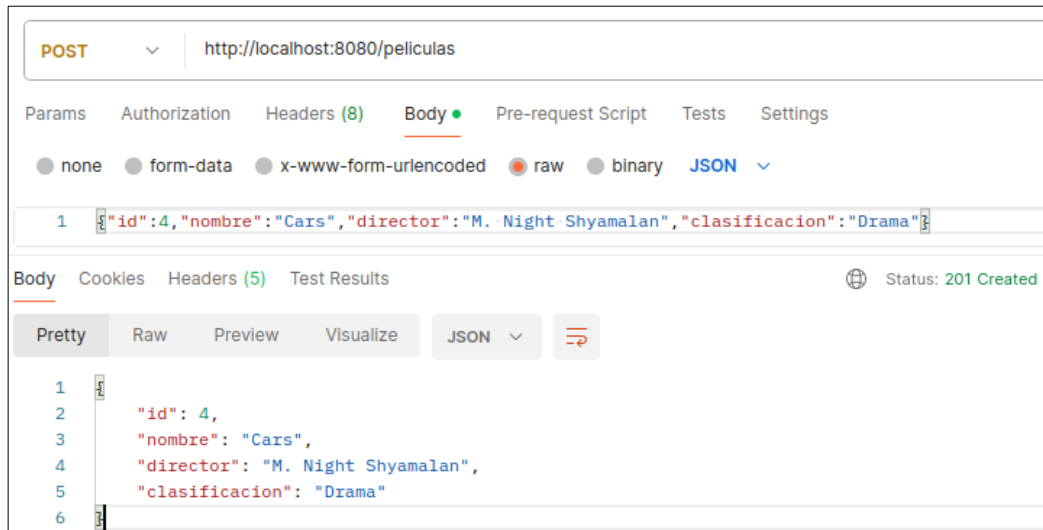


- GET de una entidad que no exista (id inválido, por ejemplo)
Nos da error 505.



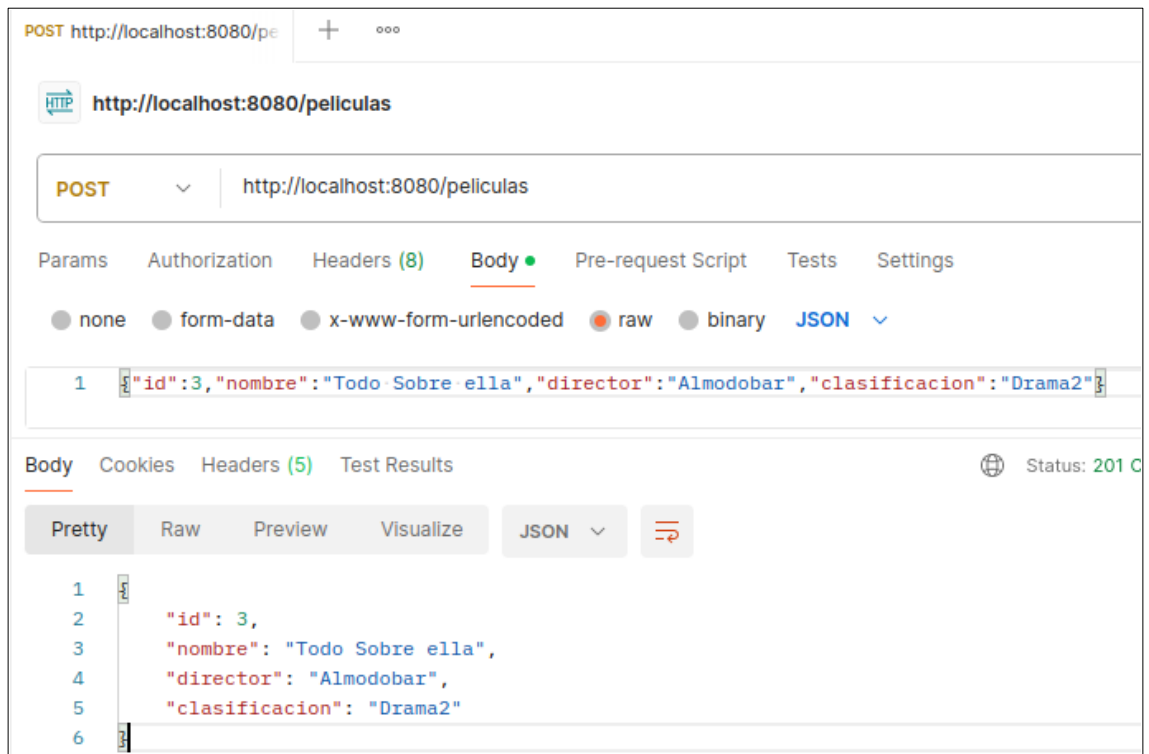
- POST de una entidad nueva.

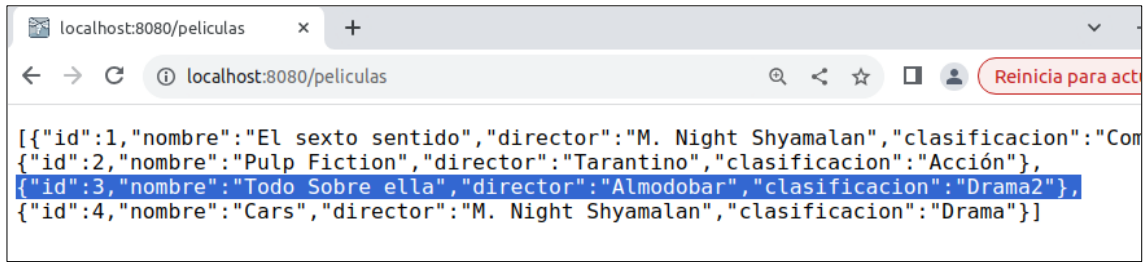
Crea la entidad:



- POST de una entidad ya existente.

Si hacemos el post de la id:3 a otros valores , lo actualiza :

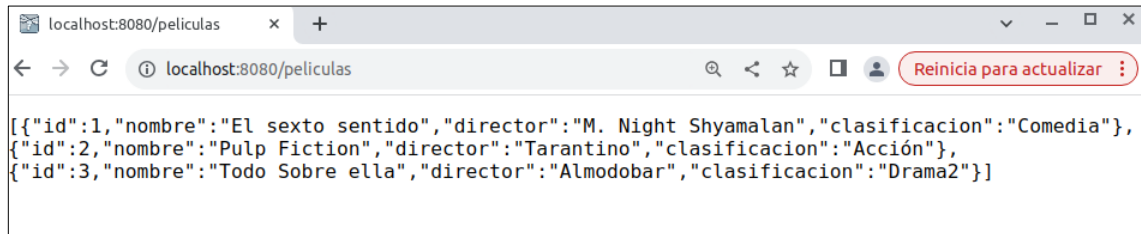
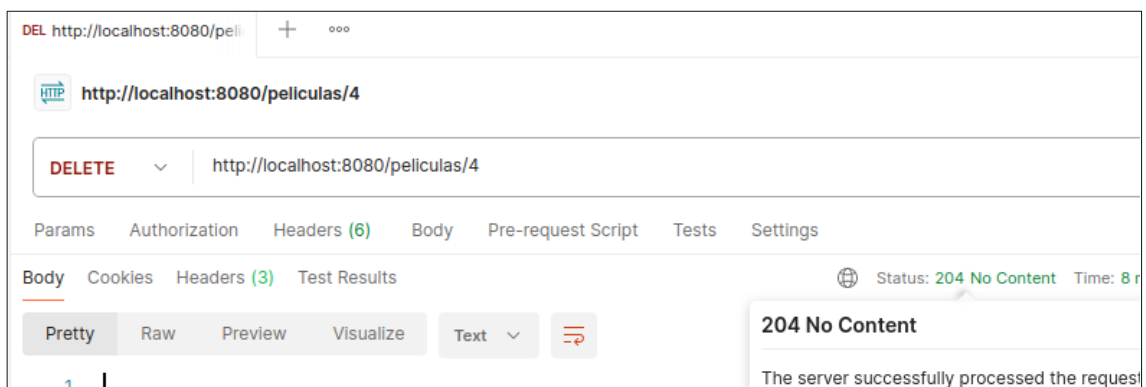




```
[{"id":1,"nombre":"El sexto sentido","director":"M. Night Shyamalan","clasificacion":"Comedia"}, {"id":2,"nombre":"Pulp Fiction","director":"Tarantino","clasificacion":"Acción"}, {"id":3,"nombre":"Todo Sobre ella","director":"Almodobar","clasificacion":"Drama2"}, {"id":4,"nombre":"Cars","director":"M. Night Shyamalan","clasificacion":"Drama"}]
```

- DEL de una entidad existente.

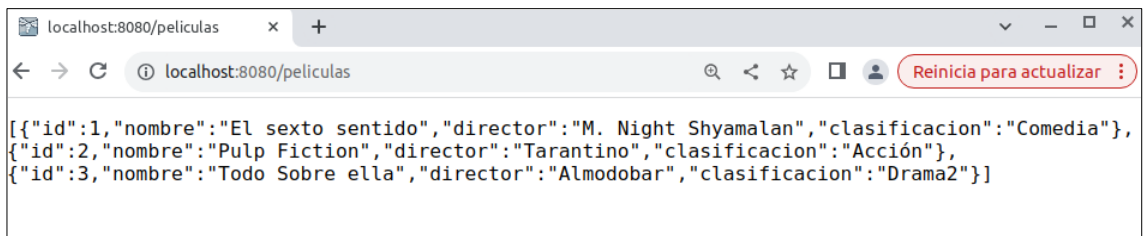
Nos da el código de estado 204 que la petición ha sido procesada correctamente pero no devuelve algún contenido.



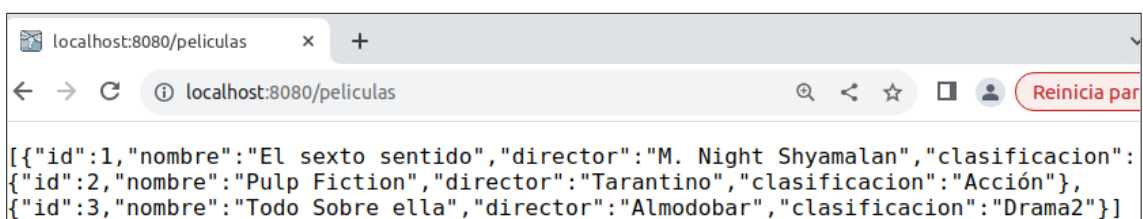
```
[{"id":1,"nombre":"El sexto sentido","director":"M. Night Shyamalan","clasificacion":"Comedia"}, {"id":2,"nombre":"Pulp Fiction","director":"Tarantino","clasificacion":"Acción"}, {"id":3,"nombre":"Todo Sobre ella","director":"Almodobar","clasificacion":"Drama2"}]
```

- DEL de una entidad que no exista.

Nos dice que ha realizado la solicitud, pero no efectúa ningún cambio:



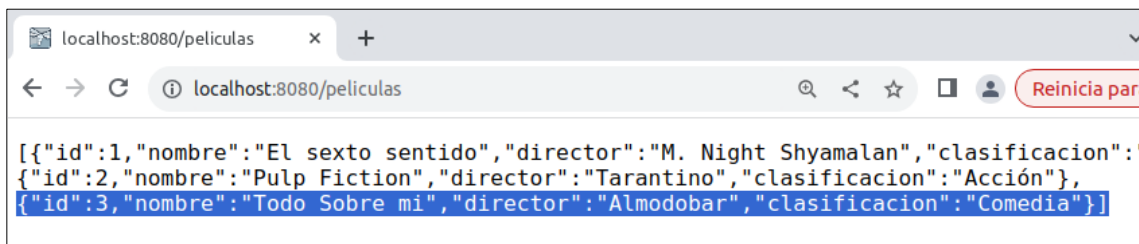
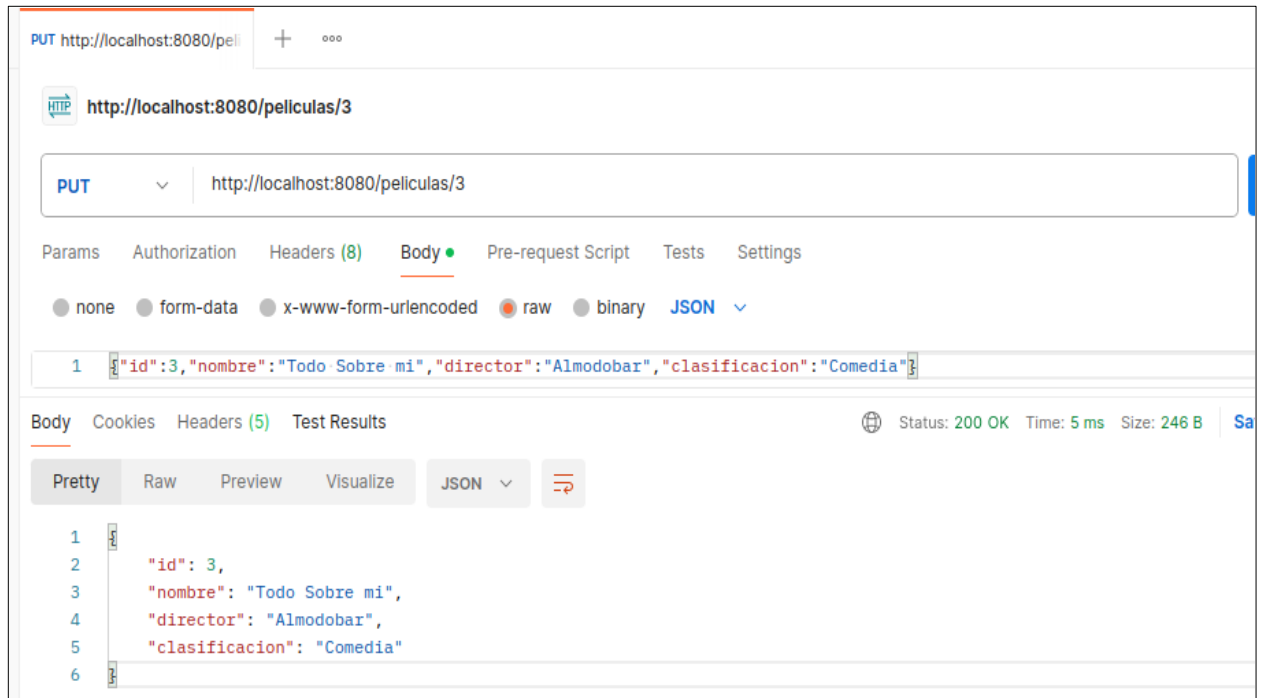
```
[{"id":1,"nombre":"El sexto sentido","director":"M. Night Shyamalan","clasificacion":"Comedia"}, {"id":2,"nombre":"Pulp Fiction","director":"Tarantino","clasificacion":"Acción"}, {"id":3,"nombre":"Todo Sobre ella","director":"Almodobar","clasificacion":"Drama2"}]
```



```
[{"id":1,"nombre":"El sexto sentido","director":"M. Night Shyamalan","clasificacion":"Comedia"}, {"id":2,"nombre":"Pulp Fiction","director":"Tarantino","clasificacion":"Acción"}, {"id":3,"nombre":"Todo Sobre ella","director":"Almodobar","clasificacion":"Drama2"}]
```

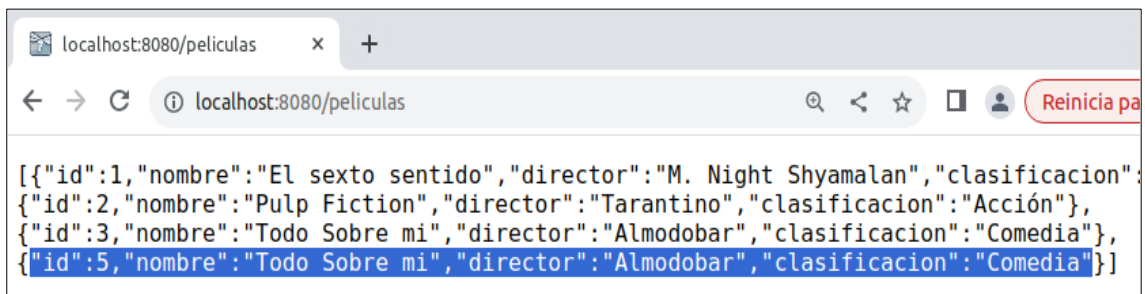
- PUT de una entidad existente.

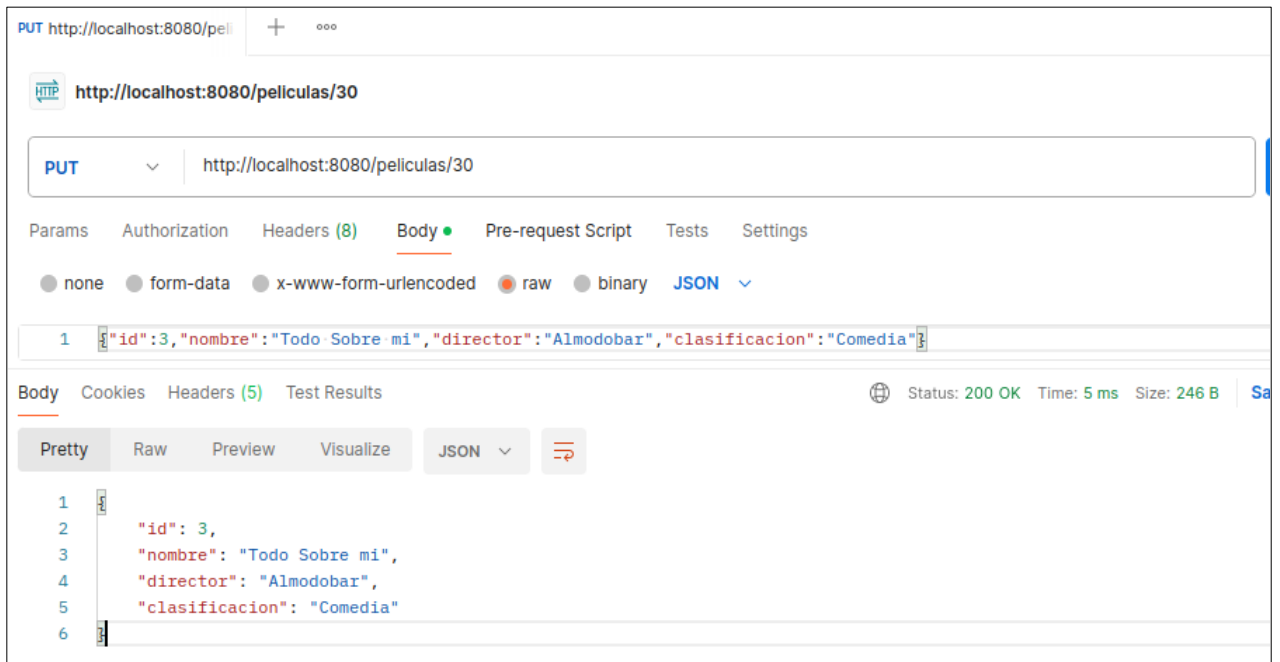
Actualiza correctamente la entidad:



- PUT de una entidad que no exista.

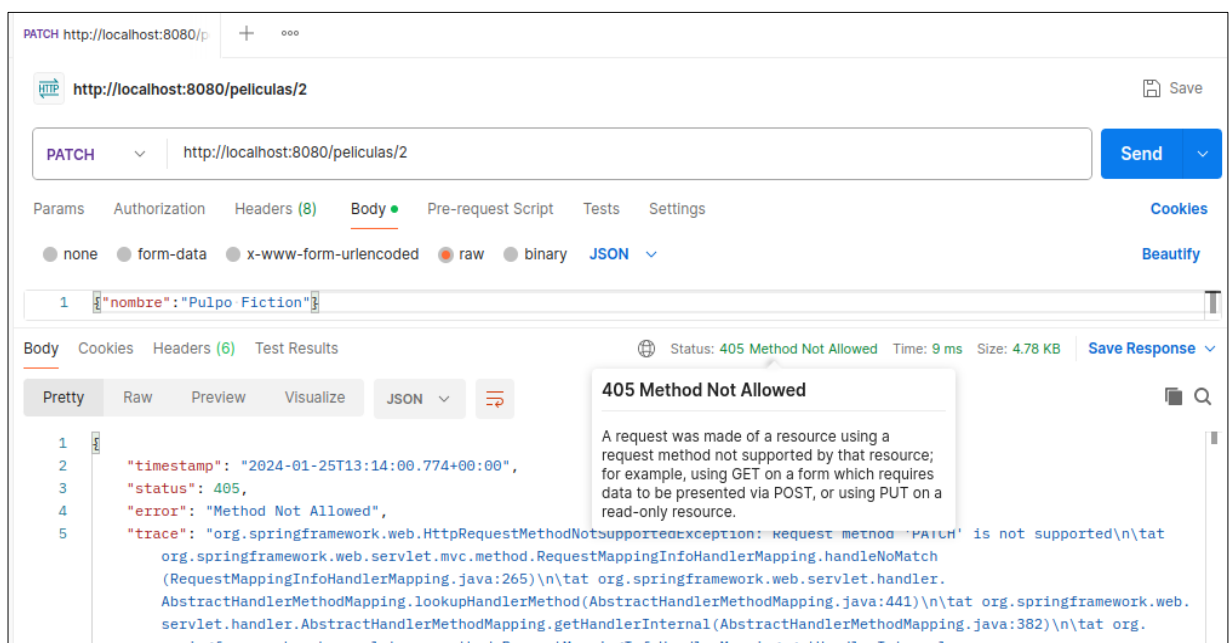
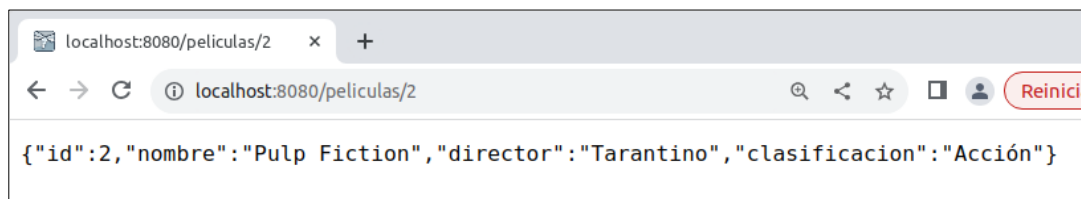
Crea una nueva entidad seguida de la ultima existente.





- PATCH de una entidad existente.

Con el código actual me dice que el método no está permitido seguramente es porque no he creado un método que haga solicitudes PATCH



He agregado el siguiente código en PeliculasController para que pueda realizar una solicitud PATCH:

```

49 peliulasRepositorio.deleteById(id);
50 return ResponseEntity.noContent().build();
51 }
52
53 @PatchMapping("/{id}")
54 public ResponseEntity<Pelicula> patchPelicula(@PathVariable(name = "id") Long id, @Request
55 Pelicula peliculaExistente = peliulasRepositorio.findById(id).orElse(null);
56
57 if (peliculaExistente == null) {
58     return ResponseEntity.notFound().build();
59 }
60
61 if (pelicula.getNombre() != null) {
62     peliculaExistente.setNombre(pelicula.getNombre());
63 }
64
65 if (pelicula.getDirector() != null) {
66     peliculaExistente.setDirector(pelicula.getDirector());
67 }
68
69 if (pelicula.getClasificacion() != null) {
70     peliculaExistente.setClasificacion(pelicula.getClasificacion());
71 }
72 Pelicula peliculaActualizada = peliulasRepositorio.save(peliculaExistente);
73
74 return ResponseEntity.ok(peliculaActualizada);
75 }
76 }
77

```

Con esta modificación ya puedo realizar un PATCH exitoso, en este caso lo hice de la id:3 cambié el director “almodobar” a “Pedro Almodobar”:

The screenshot shows a REST client interface with the following details:

- Method:** PATCH
- URL:** http://localhost:8080/peliculas/3
- Body:**

```

1 {
2   "director": "Pedro Almodobar"
3 }

```
- Status:** 200 OK, 11 ms, 250 B
- Response Body:**

```

1 {
2   "id": 3,
3   "nombre": "Todo Sobre Mi",
4   "director": "Pedro Almodobar",
5   "clasificacion": "Drama"
6 }

```
- Browser View:** The browser shows the JSON array of movies, with the third movie (id: 3) highlighted, showing the updated director: "Pedro Almodobar".

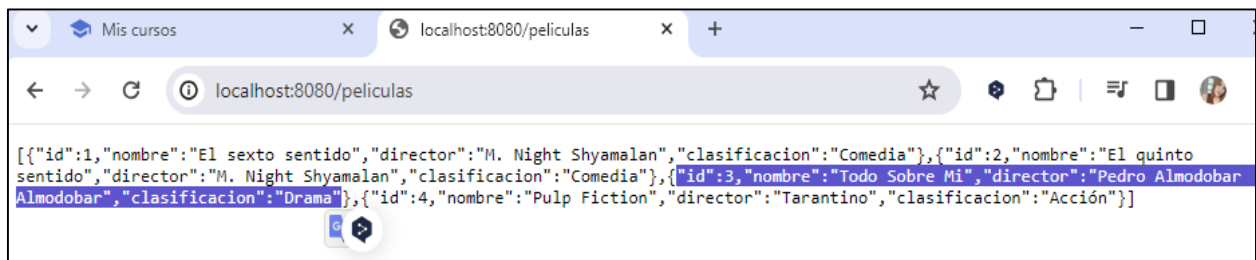
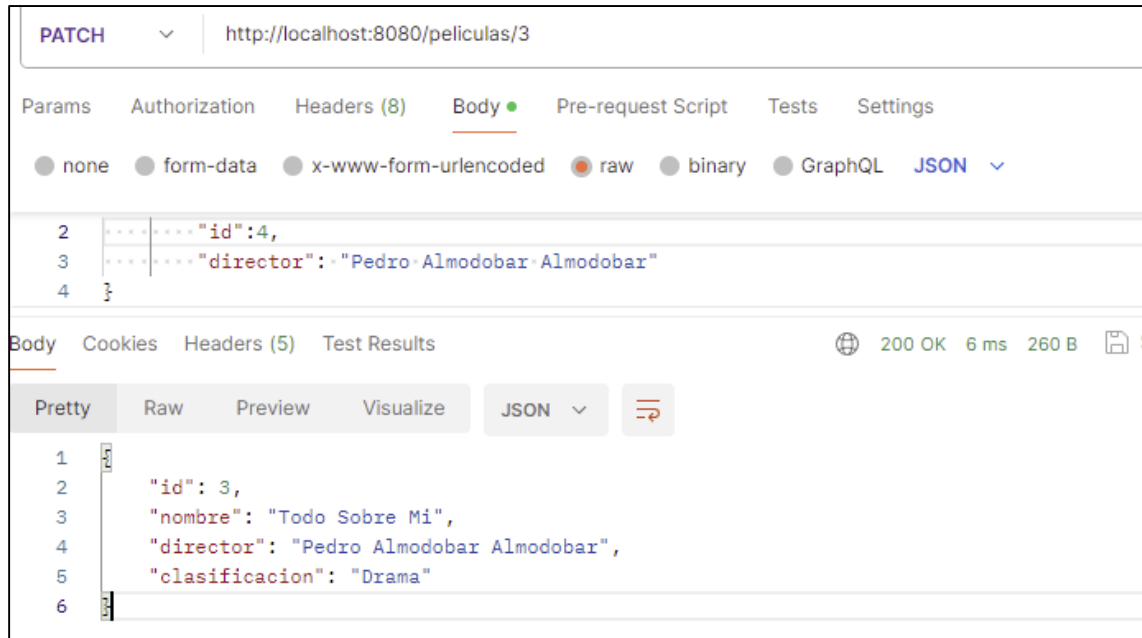

```

[{"id":1,"nombre":"El sexto sentido","director":"M. Night Shyamalan","clasificacion":"Comedia"}, {"id":2,"nombre":"El quinto sentido","director":"M. Night Shyamalan","clasificacion":"Comedia"}, {"id":3,"nombre":"Todo Sobre Mi","director":"Pedro Almodobar","clasificacion":"Drama"}, {"id":4,"nombre":"Pulp Fiction","director":"Tarantino","clasificacion":"Acción"}]

```

- PATCH de una entidad existente, pero provocando incoherencia entre el id del path y el pasado en json.

Si lo que quiero es cambiar de una existente, pero provocando un error en el id en el pasado json, me hace el cambio, pero sin importar el id que he introducido, sino que efectúa el cambio en la id del Path que es la id:3



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

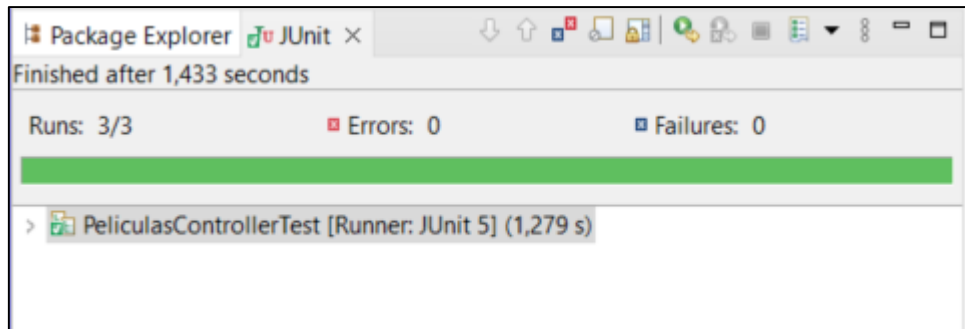
Bastaría con adaptar los siguientes ejemplos: <https://github.com/joseluisgs/springboot-profesores-madrid-2022-2023/tree/main/07-Testing> a cualquiera de las APIs de los apartados anteriores.

- **Test unitarios:** estos test se encargan de probar una unidad de código (una clase, un método, etc.) de forma aislada. Para ello se suelen utilizar mocks para aislar la unidad de código que estamos probando de las dependencias que tiene.

```

PelículasControllerTest.java X
18
19 import static org.junit.jupiter.api.Assertions.*;
20
21 class PelículasControllerTest {
22     @Mock
23     private PelículasRepositorio películasRepositorio;
24
25     @InjectMocks
26     private PelículasController películasController;
27
28     @SuppressWarnings("deprecation")
29     @BeforeEach
30     void setUp() {
31         MockitoAnnotations.initMocks(this);
32     }
33
34     @Test
35     void getPelículas() {
36         // Configuración del mock
37         List<Película> películasMock = Arrays.asList(
38             new Película(1L, "Película1", "Director1", "Clasificación1"),
39             new Película(2L, "Película2", "Director2", "Clasificación2")
40         );
41         when(películasRepositorio.findAll()).thenReturn(películasMock);
42
43         // Test
44         ResponseEntity<Iterable<Película>> response = películasController.getPelículas();
45
46         // Verificaciones
47         verify(películasRepositorio, times(1)).findAll();
48         assertEquals(HttpStatus.OK, response.getStatusCode());
49         assertNotNull(response.getBody());
50         assertEquals(películasMock, response.getBody());
51     }
52
53     @Test
54     void getPelícula() {
55         // Configuración del mock
56         Película películaMock = new Película(1L, "Película1", "Director1", "Clasificación1");
57         when(películasRepositorio.findById(1L)).thenReturn(Optional.of(películaMock));
58
59         // Test
60         ResponseEntity<Película> response = películasController.getPelícula(1L);
61
62         // Verificaciones
63         verify(películasRepositorio, times(1)).findById(1L);
64         assertEquals(HttpStatus.OK, response.getStatusCode());
65         assertNotNull(response.getBody());
66         assertEquals(películaMock, response.getBody());
67     }
68
69     @Test
70     void createPelícula() {
71         // Configuración del mock
72         Película películaMock = new Película(1L, "Película1", "Director1", "Clasificación1");
73         when(películasRepositorio.save(any(Película.class))).thenReturn(películaMock);
74
75         // Test
76         ResponseEntity<Película> response = películasController.createPelícula(películaMock);
77
78         // Verificaciones
79         verify(películasRepositorio, times(1)).save(películaMock);
80         assertEquals(HttpStatus.CREATED, response.getStatusCode());
81         assertNotNull(response.getBody());
82         assertEquals(películaMock, response.getBody());
83     }
84 }

```



▪ **Anotaciones:**

- **@Mock:** Anota un campo para indicar que es un mock. En este caso, se está utilizando para **PeliculasRepositorio**, lo que significa que se creará un mock para esta interfaz.
- **@InjectMocks:** Anota un campo para indicar que se deben inyectar los mocks en esta instancia. En este caso, se está utilizando para **PeliculasController**, lo que significa que los mocks se inyectarán automáticamente en las dependencias de la clase **PeliculasController**.
- **@BeforeEach:** Este método se ejecuta antes de cada prueba y se utiliza para inicializar los mocks. **MockitoAnnotations.initMocks(this)** inicializa todos los campos anotados con **@Mock** y **@InjectMocks** en la clase actual.

▪ **Prueba getPeliculas():**

- Configura el comportamiento del mock **peliculasRepositorio** utilizando **when** para que, cuando se llame a **findAll()**, devuelva una lista de películas simulada (**peliculasMock**).
- Ejecuta la prueba llamando al método **getPeliculas()** de la instancia **peliculasController**.
- Verifica que el método **findAll()** del mock se llamó exactamente una vez.
- Verifica que la respuesta tiene un código de estado OK, no es nula y es igual a la lista simulada **peliculasMock**.

▪ **Prueba getPelicula():**

- Configura el comportamiento del mock **peliculasRepositorio** utilizando **when** para que, cuando se llame a **findById(1L)**, devuelva una película simulada (**peliculaMock**).
- Ejecuta la prueba llamando al método **getPelicula(1L)** de la instancia **peliculasController**.
- Verifica que el método **findById(1L)** del mock se llamó exactamente una vez.

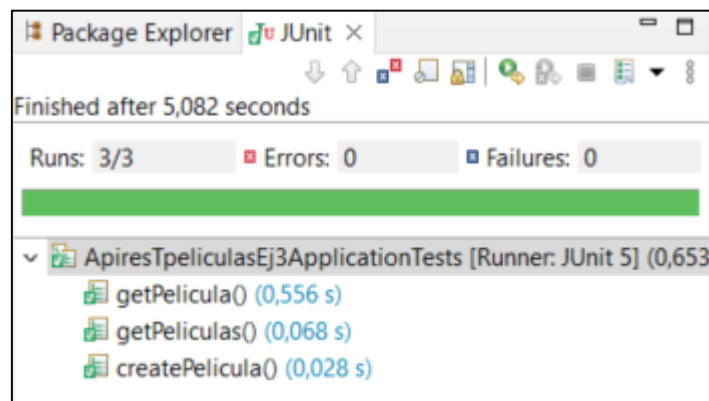
- Verifica que la respuesta tiene un código de estado OK, no es nula y es igual a la película simulada **peliculaMock**.
- **Prueba createPelicula():**
 - Configura el comportamiento del mock **peliculasRepositorio** utilizando **when** para que, cuando se llame a **save(any(Pelicula.class))**, devuelva la película simulada (**peliculaMock**).
 - Ejecuta la prueba llamando al método **createPelicula(peliculaMock)** de la instancia **peliculasController**.
 - Verifica que el método **save(peliculaMock)** del mock se llamó exactamente una vez.
 - Verifica que la respuesta tiene un código de estado CREATED, no es nula y es igual a la película simulada **peliculaMock**.

Estas pruebas están diseñadas para verificar el comportamiento del controlador **PeliculasController** en diferentes escenarios, utilizando mocks para simular el comportamiento del repositorio (**peliculasRepositorio**). La idea es asegurarse de que el controlador interactúe correctamente con su dependencia (**peliculasRepositorio**) y que devuelva las respuestas esperadas.

- **Test de integración:** estos test 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.
- **Anotaciones:**
 - **@SpringBootTest:** Indica que este es un test de integración de Spring Boot. Carga la configuración de la aplicación y permite probarla como si estuviera en ejecución.
- **Inyección de Dependencia:**
 - **@Autowired private PeliculasController peliculasController;;** Inyecta una instancia de PeliculasController en la prueba. Esto se hace para poder llamar a los métodos de PeliculasController y verificar sus resultados.
- **Método getPeliculas():**
 - **void getPeliculas():** Este método prueba el método getPeliculas de PeliculasController.
 - **ResponseEntity<Iterable<Pelicula>> response = peliculasController.getPeliculas();** Llama al método getPeliculas y obtiene la respuesta.
 - **assertEquals(HttpStatus.OK, response.getStatusCode());** Verifica que el código de estado de la respuesta sea OK (200).

- `assertNotNull(response.getBody());`: Verifica que el cuerpo de la respuesta no sea nulo.
- **Método `getPelicula()`:**
- `void getPelicula()`: Este método prueba el método `getPelicula` de `PeliculasController`.
 - `Long id = 1L`; Define un identificador para la prueba.
 - `ResponseEntity<Pelicula> response = peliculasController.getPelicula(id)`; Llama al método `getPelicula` con el identificador y obtiene la respuesta.
 - `assertEquals(HttpStatus.OK, response.getStatusCode());`: Verifica que el código de estado de la respuesta sea OK (200).
 - `assertNotNull(response.getBody());`: Verifica que el cuerpo de la respuesta no sea nulo.
- **Método `createPelicula()`:**
- `void createPelicula()`: Este método prueba el método `createPelicula` de `PeliculasController`.
 - `Pelicula nuevaPelicula = new Pelicula(1L, "Nueva Pelicula", "Director", "Clasificacion")`; Crea una nueva instancia de `Pelicula`.
 - `ResponseEntity<Pelicula> response = peliculasController.createPelicula(nuevaPelicula)`; Llama al método `createPelicula` con la nueva película y obtiene la respuesta.
 - `assertEquals(HttpStatus.CREATED, response.getStatusCode());`: Verifica que el código de estado de la respuesta sea CREATED (201).
 - `assertNotNull(response.getBody());`: Verifica que el cuerpo de la respuesta no sea nulo.

En resumen, estos tests aseguran que los métodos de `PeliculasController` funcionen correctamente.




```

1  package daw.dwes.ud6;
2
3  *import static org.junit.jupiter.api.Assertions.*;
11
12 //TEST DE INTEGRACION
13 @SpringBootTest
14 class ApiresTpeliculasEj3ApplicationTests {
15
16     @Autowired
17     private PeliculasController peliculasController;
18
19     @Test
20     void getPeliculas() {
21         // Test
22         ResponseEntity<Iterable<Pelicula>> response = peliculasController.getPeliculas();
23         // Verificaciones
24         assertEquals(HttpStatus.OK, response.getStatusCode());
25         assertNotNull(response.getBody());
26     }
27
28
29     @Test
30     void getPelicula() {
31         // Configuración del entorno de prueba si es necesario
32         Long id = 1L;
33
34         // Test
35         ResponseEntity<Pelicula> response = peliculasController.getPelicula(id);
36
37         // Verificaciones
38         assertEquals(HttpStatus.OK, response.getStatusCode());
39         assertNotNull(response.getBody());
40     }
41
42     @Test
43     void createPelicula() {
44         // Configuración del entorno de prueba
45         Pelicula nuevaPelicula = new Pelicula(1L, "Nueva Pelicula", "Director", "Clasificacion");
46
47         // Test
48         ResponseEntity<Pelicula> response = peliculasController.createPelicula(nuevaPelicula);
49
50         // Verificaciones
51         assertEquals(HttpStatus.CREATED, response.getStatusCode());
52         assertNotNull(response.getBody());
53     }
54
55 }
56

```

- **Test End-to-End:** estos test se encargan de probar que todo el sistema funciona correctamente. Para ello se suelen utilizar herramientas que simulan un navegador web y que permiten simular las acciones que haría un usuario en la aplicación. Por ejemplo, cuando usamos Postman.

```
PeliculasControllerMvcIntegrationTest.java ×
1 package daw.dwes.ud6;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 @SpringBootTest
6 // Configuramos el cliente MVC
7 @AutoConfigureMockMvc
8 // Configuramos los testers de JSON
9 @AutoConfigureJsonTesters
10 // Limpiamos el contexto antes de cada test
11 @DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)
12 // Ordenamos los test por orden de ejecución
13 @TestMethodOrder(MethodOrderer.OrderAnnotation.class)
14 class PeliculasControllerMvcIntegrationTest {
15
16     @Autowired
17     private MockMvc mockMvc;
18
19     @Autowired
20     private JacksonTester<List<Película>> jsonPelículaList;
21
22     @Autowired
23     private JacksonTester<Película> jsonPelícula;
24
25     @MockBean
26     private PeliculasRepositorio peliculasRepositorio; // Simulamos el repositorio
27
28     @Test
29     @Order(1)
30     void getPelículasTest() throws Exception {
31         // Configuramos el comportamiento del repositorio simulado
32         when(peliculasRepositorio.findAll()).thenReturn(Arrays.asList(
33             new Película(1L, "El sexto sentido", "M. Night Shyamalan", "Comedia"),
34             new Película(2L, "Pulp Fiction", "Tarantino", "Acción"),
35             new Película(3L, "Todo Sobre Mi", "Almodobar", "Drama")
36         ));
37
38         MockHttpServletResponse response = mockMvc.perform(
39             MockMvcRequestBuilders.get("/películas")
40                 .accept(MediaType.APPLICATION_JSON))
41             .andReturn().getResponse();
42
43         List<Película> peliculasResponse = jsonPelículaList.parse(response.getContentAsString()).getObject();
44
45         assertEquals(HttpStatus.OK.value(), response.getStatus());
46         assertNotNull(peliculasResponse);
47         assertEquals(3, peliculasResponse.size());
48
49         Película pelicula1 = peliculasResponse.get(0);
50         assertEquals("El sexto sentido", pelicula1.getNombre());
51         assertEquals("M. Night Shyamalan", pelicula1.getDirector());
52         assertEquals("Comedia", pelicula1.getClasificacion());
53
54         // Verificamos que se ha llamado al método del repositorio con el id correcto
55         verify(peliculasRepositorio, times(1)).findById(1L);
56     }
57 }
```



```

@Test
@Order(2)
void getPeliculaTest() throws Exception {
    // Guardamos una película en el repositorio simulado
    when(peliculasRepositorio.findById(1L)).thenReturn(Optional.of(
        new Pelicula(1L, "El sexto sentido", "M. Night Shyamalan", "Comedia")
    ));

    MockHttpServletResponse response = mockMvc.perform(
        MockMvcRequestBuilders.get("/peliculas/1")
            .accept(MediaType.APPLICATION_JSON)
            .andReturn().getResponse();

    Pelicula peliculaResponse = jsonPelicula.parse(response.getContentAsString()).getObject();

    assertEquals(HttpStatus.OK.value(), response.getStatus());
    assertNotNull(peliculaResponse);
    assertEquals(1L, peliculaResponse.getId());
    assertEquals("El sexto sentido", peliculaResponse.getNombre());
    assertEquals("M. Night Shyamalan", peliculaResponse.getDirector());
    assertEquals("Comedia", peliculaResponse.getClasificacion());

    // Verificamos que se ha llamado al método del repositorio con el id correcto
    verify(peliculasRepositorio, times(1)).findById(1L);
}

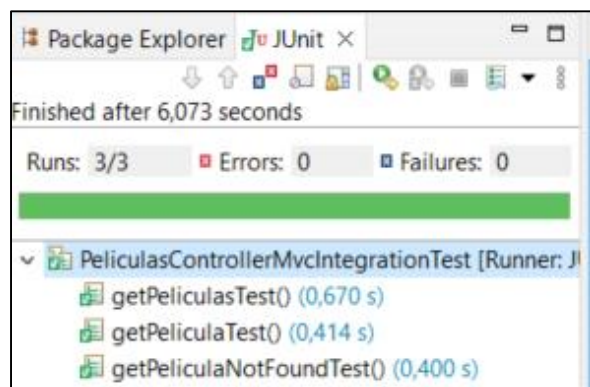
@Test
@Order(3)
void getPeliculaNotFoundTest() throws Exception {
    // Configuramos el comportamiento del repositorio simulado
    when(peliculasRepositorio.findById(100L)).thenReturn(Optional.empty());

    MockHttpServletResponse response = mockMvc.perform(
        MockMvcRequestBuilders.get("/peliculas/100")
            .accept(MediaType.APPLICATION_JSON)
            .andReturn().getResponse();

    assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatus());

    // Verificamos que se ha llamado al método del repositorio
    verify(peliculasRepositorio, times(1)).findById(100L);
}
}

```



Configuración y Anotaciones:

- Se usan diversas anotaciones de Spring Boot (@SpringBootTest, @AutoConfigureMockMvc, @AutoConfigureJsonTesters, @DirtiesContext, @TestMethodOrder) para configurar el entorno de prueba y el contexto de la aplicación.

Inicialización de Componentes:

- Se inyecta un cliente MockMvc (mockMvc) para realizar solicitudes HTTP simuladas.
- Se utilizan objetos JacksonTester para facilitar la conversión entre objetos Java y representaciones JSON.

Simulación del Repositorio:

- Se utiliza @MockBean para simular el repositorio de películas (películasRepositorio) en el contexto de prueba.

Prueba getPelículasTest:

- Se configura el comportamiento simulado del repositorio para que devuelva una lista de películas cuando se llame al método findAll.
- Se realiza una solicitud GET a "/películas" y se verifica que la respuesta tenga el código de estado HTTP 200 (OK).
- Se verifica que la respuesta JSON contiene la lista esperada de películas.
- Se verifica que el método findAll del repositorio se llamó una vez.

Prueba getPelículaTest:

- Se configura el comportamiento simulado del repositorio para que devuelva una película específica cuando se llame al método findById.
- Se realiza una solicitud GET a "/películas/1" y se verifica que la respuesta tenga el código de estado HTTP 200 (OK).
- Se verifica que la respuesta JSON contiene la película esperada.
- Se verifica que el método findById del repositorio se llamó una vez con el ID correcto.

Prueba getPelículaNotFoundTest:

- Se configura el comportamiento simulado del repositorio para que devuelva un Optional vacío cuando se llame al método findById.
- Se realiza una solicitud GET a "/películas/100" y se verifica que la respuesta tenga el código de estado HTTP 404 (Not Found).
- Se verifica que el método findById del repositorio se llamó una vez con el ID correcto.

Estas pruebas garantizan que el controlador funcione correctamente al interactuar con el repositorio de películas y que las respuestas se generen según lo esperado.

CONCLUSIÓN

En resumen, durante esta práctica he aprendido desde la configuración inicial con Json-server hasta la implementación con Spring Boot y REST Repository de una API REST. Exploré diversos enfoques y utilicé varias herramientas, lo que me permitió comprender tanto la creación de servicios web sencillos con Json-server como la implementación más estructurada que ofrece Spring Boot.

La inclusión de pruebas automatizadas, tanto unitarias como de integración, resaltó la importancia de garantizar el correcto funcionamiento de la API y asegurar la calidad del código que desarrollé. Además, pude destacar las diferencias y similitudes entre los enfoques utilizados, proporcionándome una visión más clara al momento de elegir tecnologías y estrategias para futuros proyectos.