

## **Tarea 0 – UD6**

### **Rompiendo el hielo con las APIs RESTful**

#### **1.- Leer los materiales y referencias del tema y realizar un resumen sobre qué es un API RESTful**

Una API RESTful (Interfaz de Programación de Aplicaciones basada en Transferencia de Estado Representacional) es una interfaz que permite a dos sistemas de computación intercambiar información de manera segura a través de Internet. Las API RESTful siguen principios clave, entre ellos:

1. **Stateless (Sin estado):** No mantienen ningún estado entre peticiones, lo que mejora la escalabilidad.
2. **Neutralidad tecnológica:** Admite prácticamente cualquier cliente y lenguaje de programación.
3. **URLs orientadas a recursos:** Cada URL gestiona todas las operaciones que un recurso específico soporta.
4. **Métodos HTTP:** Utiliza métodos estándar de HTTP, como GET, POST, PUT, PATCH y DELETE, para realizar acciones sobre los datos.
5. **Uniformidad:** Define una interfaz uniforme con restricciones que permiten una comunicación eficiente.

#### **Beneficios de las API RESTful:**

- **Escalabilidad:** Optimiza la interacción cliente-servidor, mejorando la escalabilidad.
- **Flexibilidad:** Permite la evolución independiente de componentes del servidor y cliente.
- **Independencia tecnológica:** Puede escribirse en diferentes lenguajes sin afectar el diseño de la API.

#### **Cómo Funcionan:**

- El cliente envía una solicitud al servidor con un identificador único de recurso, método HTTP, encabezados, datos y parámetros.
- El servidor autentica al cliente, procesa la solicitud y devuelve una respuesta con un código de estado, cuerpo del mensaje y encabezados.

#### **Métodos de Autenticación:**

- **HTTP básica:** Cliente envía nombre y contraseña codificados en base64 en el encabezado.
- **Autenticación del portador:** Utiliza un token cifrado enviado en los encabezados.
- **Claves de la API:** Asigna una clave única al cliente para su verificación.
- **OAuth:** Combina contraseñas y tokens para un acceso de inicio de sesión seguro.

#### **Componentes de la Respuesta del Servidor:**

- **Línea de estado:** Código de estado de tres dígitos que indica el resultado de la solicitud.
- **Cuerpo del mensaje:** Contiene la representación del recurso, seleccionada según los encabezados de la solicitud.
- **Encabezados:** Proporcionan metadatos sobre la respuesta, como fecha, tipo de contenido, etc.

**2.- Elige varias API REST de más abajo (o que busques) lee su documentación de acceso y haz pruebas desde el navegador, línea de comandos (curl) y/o Postman. ¿Permiten crear o modificar elementos?**

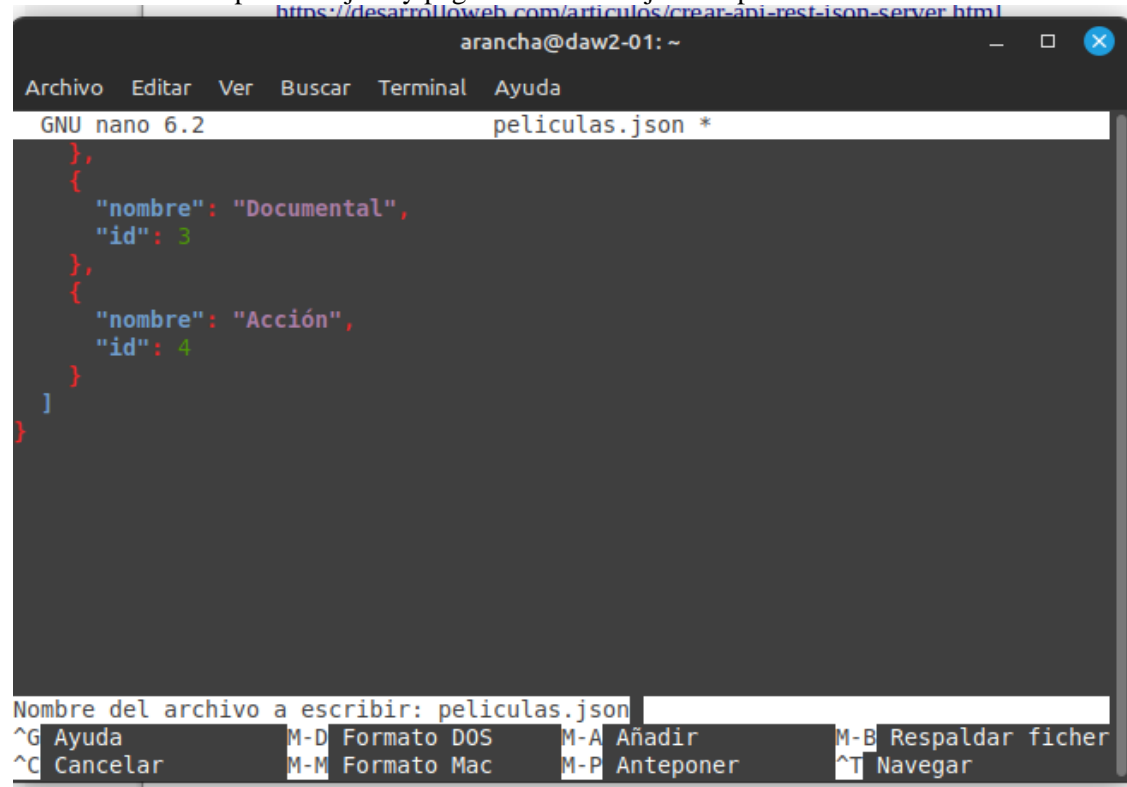
Tutorial:

<https://desarrolloweb.com/articulos/crear-api-rest-json-server.html>

Siguiendo el tutorial, instalo json-server:

```
arancha@daw2-01:~$ npm install -g json-server
added 54 packages in 9s
14 packages are looking for funding
  run `npm fund` for details
arancha@daw2-01:~$
```

Genero un archivo películas.json y pego ahí el listado json de películas:



```
arancha@daw2-01: ~
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
GNU nano 6.2 películas.json *
},
{
  "nombre": "Documental",
  "id": 3
},
{
  "nombre": "Acción",
  "id": 4
}
]
}

Nombre del archivo a escribir: películas.json
^G Ayuda      M-D Formato DOS  M-A Añadir      M-B Respalda fichero
^C Cancelar   M-M Formato Mac  M-P Anteponer   ^T Navegar
```

Inicio json-server con el archivo películas.json. Vemos que su puerto es el 3000, por lo que desde ahí podremos ver nuestra API:

```
arancha@daw2-01:~$ 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...

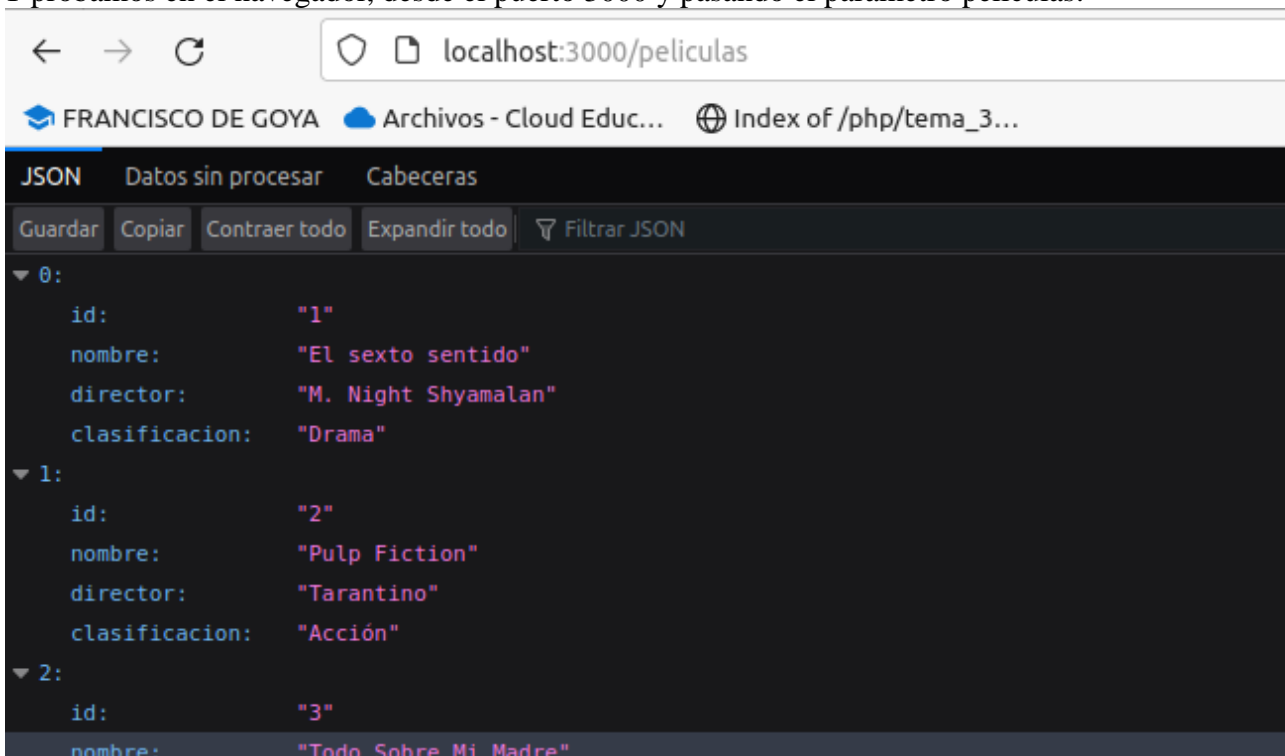
(.~ ~ ~.)

Index:
http://localhost:3000/

Static files:
Serving ./public directory if it exists

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

Y probamos en el navegador, desde el puerto 3000 y pasando el parámetro películas:



Ahora desde postman vamos a probar si podemos añadir un elemento.

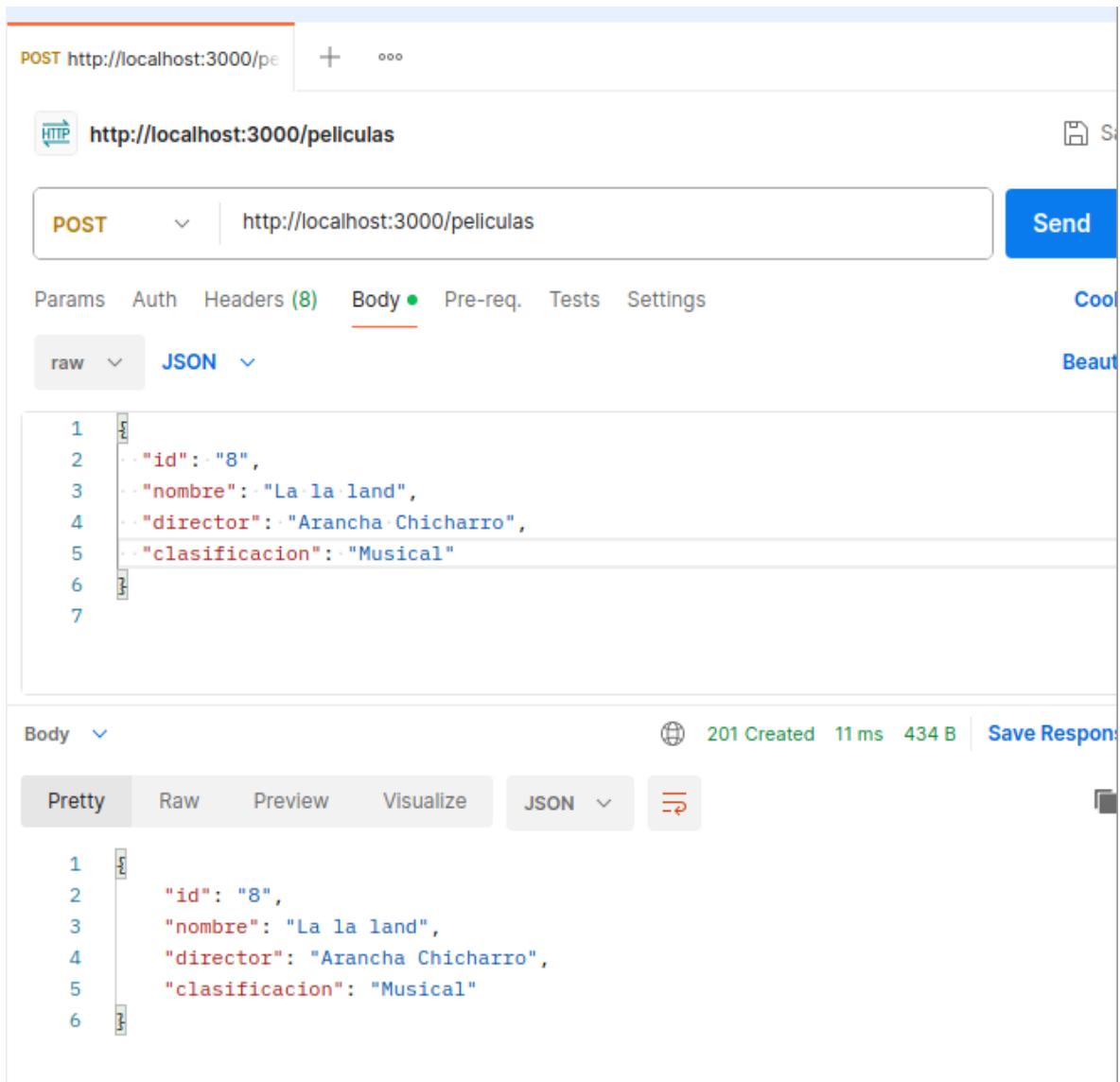
Introducimos nuestra url y seleccionamos POST.

Luego en el apartado Body, indicamos raw y JSON.

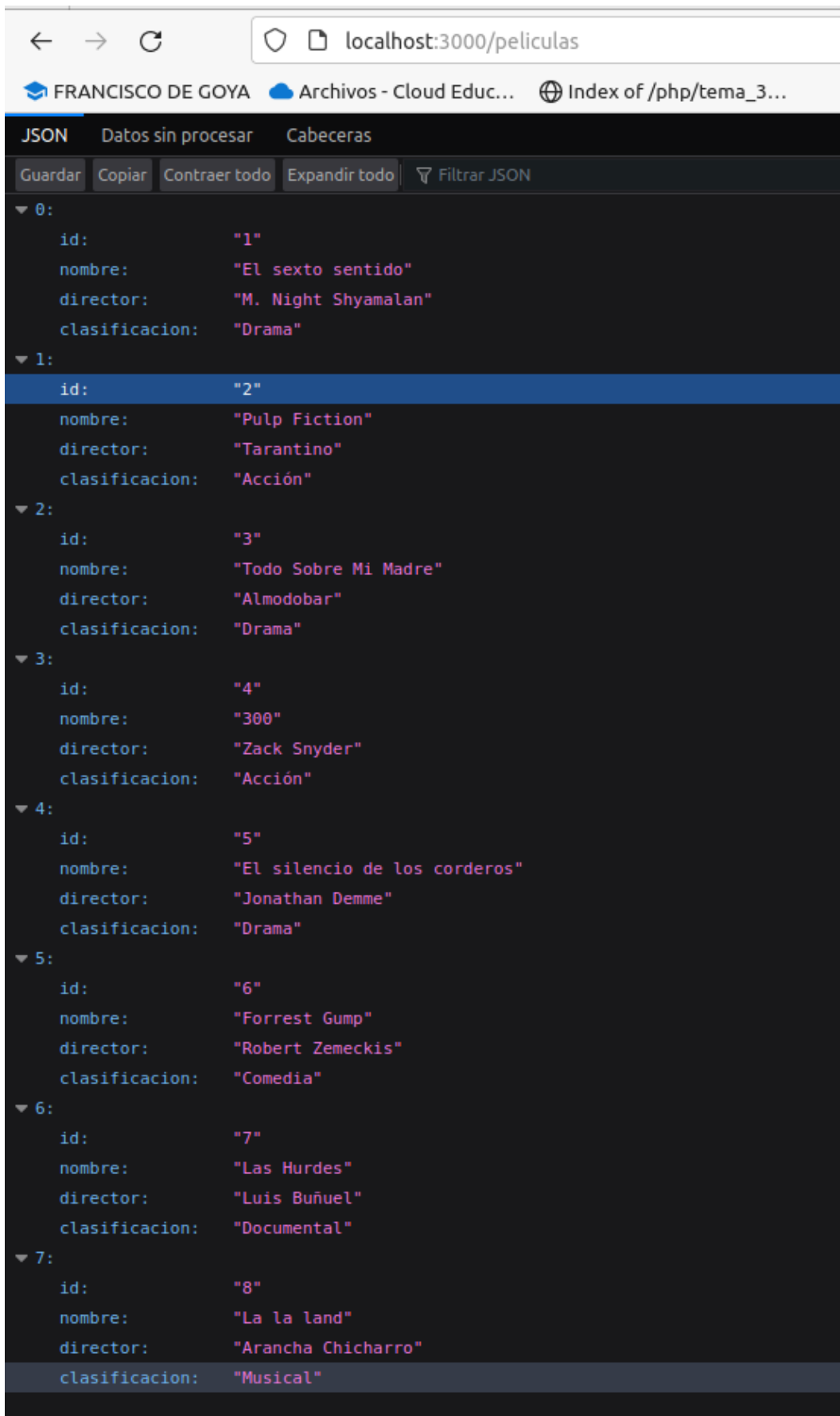
Escribimos un nuevo objeto JSON, en el mismo formato que los que ya están en archivo `peliculas.json`, en este caso pongo:

Id: 8  
Nombre: la la land  
Director: arancha chicharro  
Clasificación: Musical

Y damos a enviar:



Y probamos en el navegador, vemos que nuestro nuevo objeto se ha añadido al final, en la posición 8:



The screenshot shows a web browser at the URL `localhost:3000/peliculas`. The page displays a JSON array of movie data. The JSON is expanded to show 8 items, each with the following fields: `id`, `nombre`, `director`, and `clasificacion`.

| id | nombre                      | director           | clasificacion |
|----|-----------------------------|--------------------|---------------|
| 1  | El sexto sentido            | M. Night Shyamalan | Drama         |
| 2  | Pulp Fiction                | Tarantino          | Acción        |
| 3  | Todo Sobre Mi Madre         | Almodobar          | Drama         |
| 4  | 300                         | Zack Snyder        | Acción        |
| 5  | El silencio de los corderos | Jonathan Demme     | Drama         |
| 6  | Forrest Gump                | Robert Zemeckis    | Comedia       |
| 7  | Las Hurdes                  | Luis Buñuel        | Documental    |
| 8  | La la land                  | Arancha Chicharro  | Musical       |

### 3.- Sigue este tutorial para crear un servicio REST básico con SpringBoot:

<https://spring.io/guides/gs/rest-service/> Demasiado fácil ¿verdad? Solo implemente GET, veamos alguno más avanzado.

Creo proyecto llamada SaludoRestServices y añado dependencia SpringWeb.

Creo clase de representación llamada Saludo:

```
Saludo.java × *SaludoRestController.java
1 package com.example.demo;
2
3 public record Saludo(long id, String content) {
4
5 }
6
```

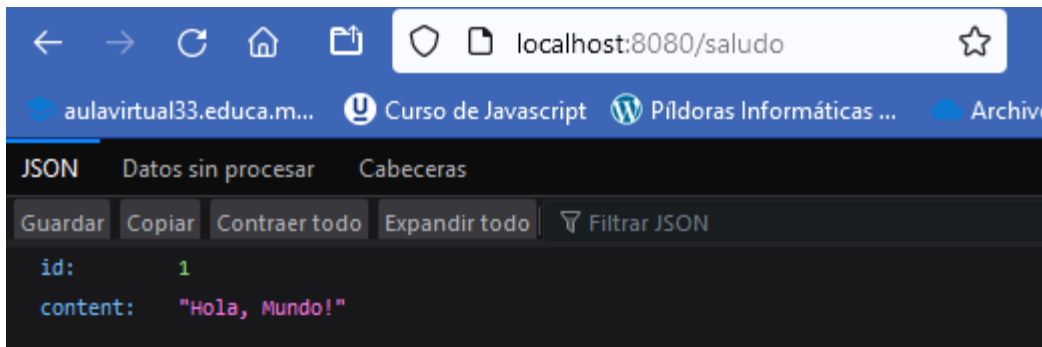
Creo clase controlador:

```
Saludo.java SaludoRestController.java ×
1 package com.example.demo;
2
3 import java.util.concurrent.atomic.AtomicLong;
4
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.RequestParam;
7
8 public class SaludoRestController {
9
10     private static final String template = "Hola, %s!";
11     private final AtomicLong counter = new AtomicLong();
12
13     @GetMapping("/saludo")
14     public Saludo saludo(
15         @RequestParam(value = "name", defaultValue = "Mundo")
16         String name) {
17         return new Saludo(counter.incrementAndGet(),
18             String.format(template, name));
19     }
20 }
```

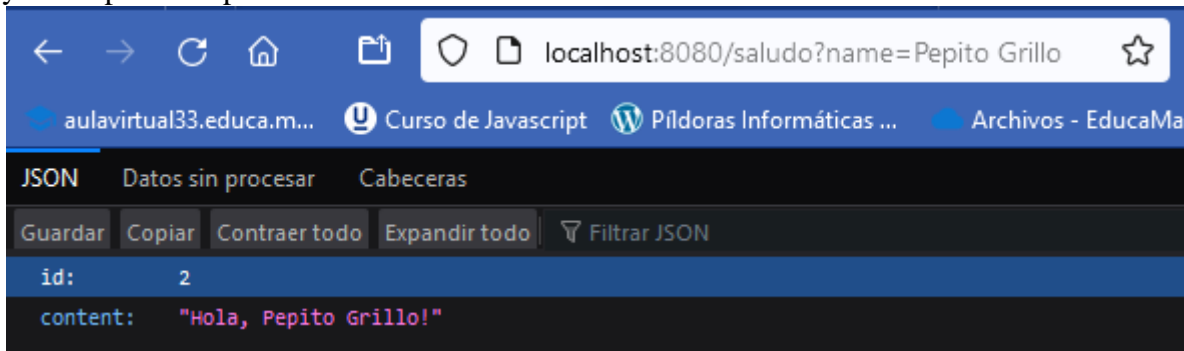
Una diferencia clave entre un controlador MVC tradicional y el controlador de servicios web RESTful es la forma en que se crea el cuerpo de la respuesta HTTP. En lugar de depender de una plantilla de visualización para mostrar los datos del saludo en HTML, este controlador de servicio web RESTful completa y devuelve un Saludo objeto. Los datos del objeto se escribirán directamente en la respuesta HTTP como JSON.

Ejecutamos aplicación.

Primero probamos con el método saludo sin parámetro:

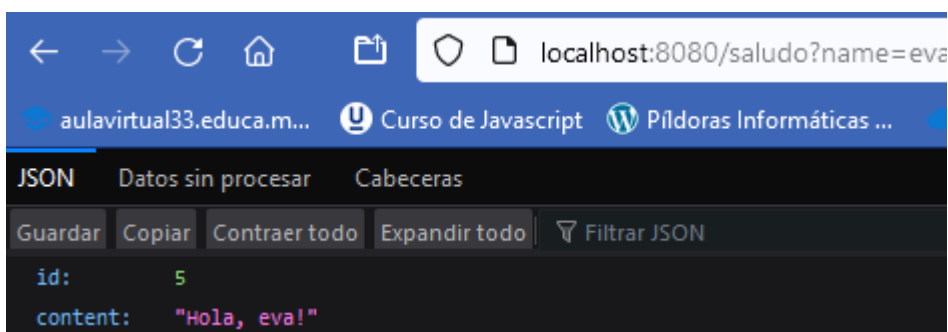
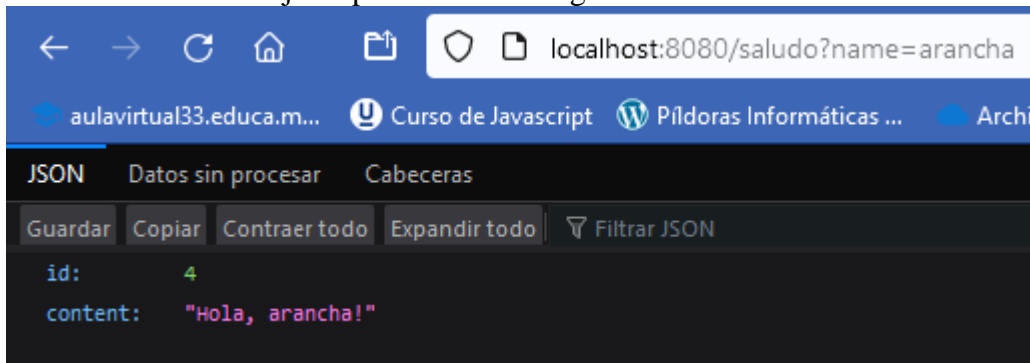


y ahora pasamos parámetro name:



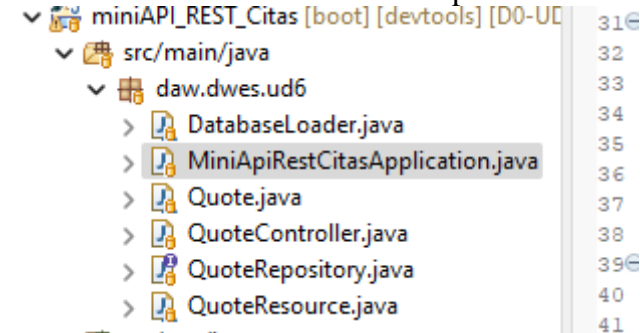
Podemos observar que el campo id, al que en la clase Controller identificamos con un counter (contador) y con la función counter.incrementAndGet() del método saludo, va en incremento.

Probamos con más objetos para estar más seguros del incremento del id:



4.- Clona estos ficheros mencionados en el siguiente tutorial para levantar nuestro propio servidor de “quotes” <https://github.com/spring-guides/quoters/tree/master/src/main/java/org/springframework/quoters> (ojo, `@PathVariable(name="id")` )

Clonamos todos los ficheros del repositorio:



Vamos a explicar un poco el código.

Esta aplicación de Spring es un ejemplo de un servicio web simple que gestiona y expone citas (quotes) mediante una API REST.

#### Quote.java:

Esta clase define "Quote" que, con la anotación `@Entity` almacena sus atributos a una tabla en la base de datos. Tiene un identificador único (**id**) generado automáticamente, y un campo de texto para la cita (**quote**). También tiene métodos getter, setter, y métodos override para equals, hashCode y toString.

#### QuoteRepository.java:

Esta interfaz extiende **JpaRepository**, para realizar operaciones CRUD en la base de datos. En este caso, se usa para realizar operaciones en la entidad **Quote**.

#### DatabaseLoader.java:

Es una clase de configuración que se ejecuta al iniciar la aplicación. Contiene un método **init** anotado con `@Bean` y **CommandLineRunner**, que carga algunas citas de ejemplo en la base de datos al iniciar la aplicación.

#### QuoteResource.java:

Esta clase es una versión simplificada de **Quote**, es cómo se mostrará en la API. Contiene un campo **type** para describir el estado de la solicitud y un campo **value** que contiene la cita (id y quote). También tiene los métodos getter, setter, equals, hashCode y toString.

#### QuoteController.java:

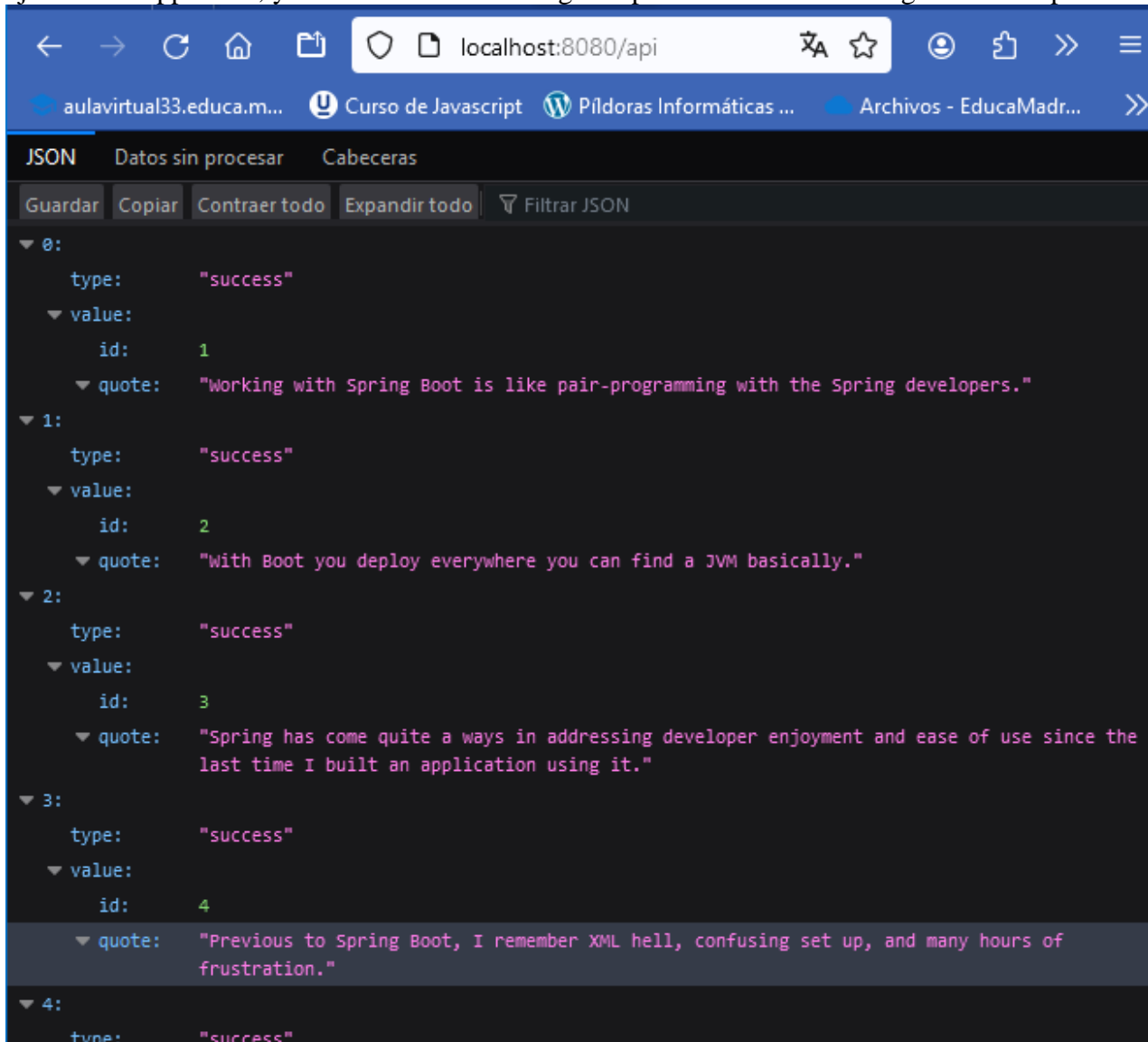
Este controlador define las rutas y métodos para gestionar las citas. Utiliza la anotación `@RestController` para indicar que este controlador responde a solicitudes HTTP.

- **getAll()**: Mapea la ruta `/api` y devuelve una lista de **QuoteResource** mostrando todas las citas almacenadas en la base de datos.
- **getOne(@PathVariable Long id)**: Mapea la ruta `/api/{id}` y devuelve una **QuoteResource** para la cita con el ID proporcionado. Si no existe, devuelve una respuesta indicando que la cita no existe.
- **getRandomOne()**: Mapea la ruta `/api/random` y devuelve una **QuoteResource** para una cita aleatoria de la base de datos.



Ahora vamos a probar ver las citas.

Ejecutamos app en sts, y accedemos en el navegador primero con el método getAll con /api:



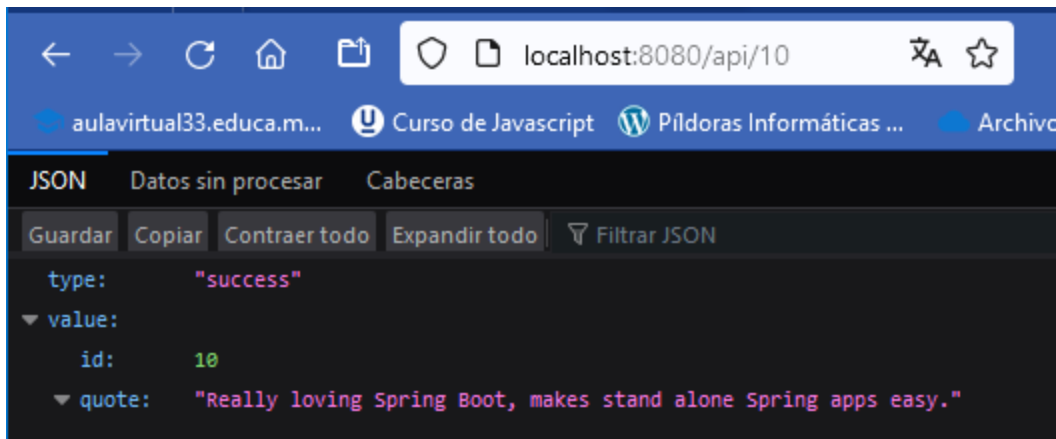
Analizamos:

El formato de cada elemento es type y value, atributos de la clase QuoteResource, y a su vez, el value se compone de id y quote, como los atributos de la clase Quote.

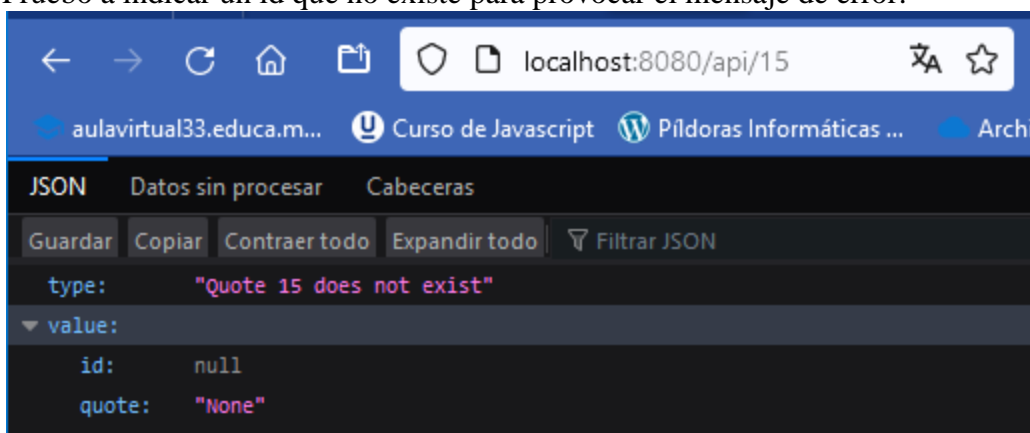
Ahora probamos con el método getOne. Da error, pero para corregirlo copiamos la anotación del profesor de añadir: @PathVariable(name="id"), para indicar que el la url va a mandar el valor del parámetro id.

```
@GetMapping("/api/{id}")
public QuoteResource getOne(@PathVariable(name="id") Long id) {

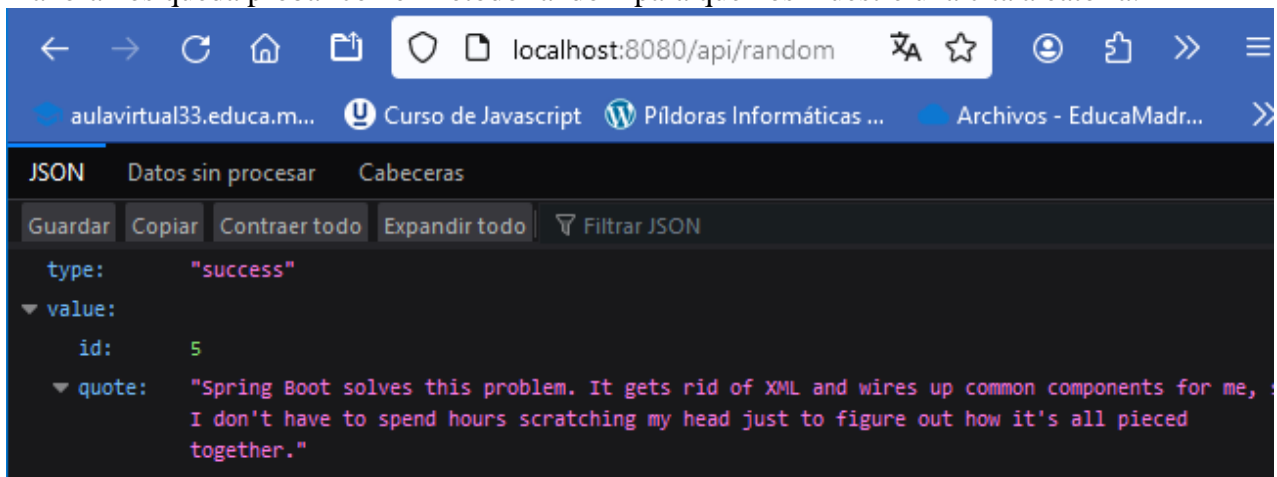
    return repository.findById(id)
        .map(quote -> new QuoteResource(quote, "success"))
        .orElse(new QuoteResource(NONE, "Quote " + id + " does not exist"))
}
```



Pruebo a indicar un id que no existe para provocar el mensaje de error:



Y ahora nos queda probar con el método random para que nos muestre una cita aleatoria:



5.- Sigue este tutorial para mostrar el consumo de una API REST en SpringBoot:

<https://spring.io/guides/gs/consuming-rest/> .

API elegida:

[http://official-joke-api.appspot.com/random\\_joke](http://official-joke-api.appspot.com/random_joke)

llamamos al proyecto ConsumoApiRest.

Creamos una clase de dominio donde ponemos los datos del objeto JSON. Con la anotación `@JsonIgnoreProperties` para indicar que cualquier propiedad que no esté vinculada se ignore. Para vincular directamente los datos a los tipos personalizados, se debe indicar que el nombre de la variable sea exactamente el mismo que la clave en el documento JSON devuelto por la API.

```
@JsonIgnoreProperties(ignoreUnknown = true)
public record Joke(
    String type,
    String setup,
    String punchline,
    Integer id) {
    //{"type":"general","setup":"What is red and
```

Y en la clase de application tenemos que añadir lo siguiente:

- Un logger, para enviar resultados al log.
- Un rest Template, que utiliza la biblioteca JSON para procesar los datos entrantes.
- Un CommandLineRunner, que ejecuta RestTemplate al inicio.

```
private static final Logger log = LoggerFactory.getLogger
    [(ConsumoApiRestApplication.class)];

public static void main(String[] args) {
    SpringApplication.run(ConsumoApiRestApplication.class, args);
}

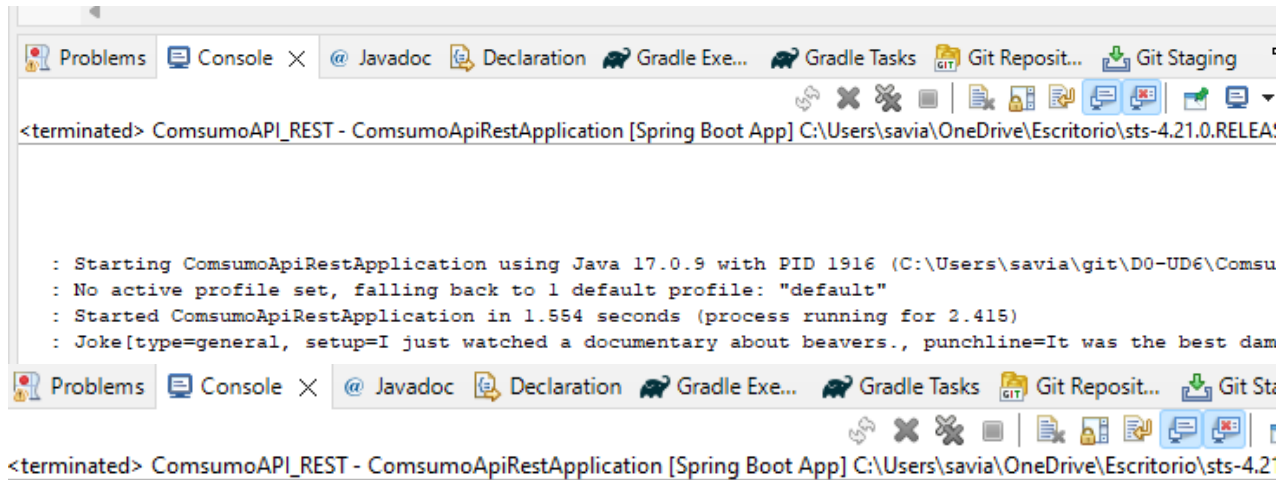
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder.build();
}

@Bean
@Profile("!test")
public CommandLineRunner run(RestTemplate restTemplate) throws Exception {
    return args -> {
        Joke joke = restTemplate.getForObject(
            "http://official-joke-api.appspot.com/random_joke", Joke.class);
        log.info(joke.toString());
    };
};
```

- Dentro del método **run**, se utiliza el **RestTemplate** para hacer una solicitud HTTP a un servicio externo que proporciona chistes aleatorios.
- La URL del servicio es "[http://official-joke-api.appspot.com/random\\_joke](http://official-joke-api.appspot.com/random_joke)".
- La respuesta se espera que sea un objeto de tipo **Joke** (puede ser una clase definida en tu código que representa un chiste).
- El chiste obtenido se registra utilizando **log.info(joke.toString())**.

Y ejecutamos:

Vemos en consola el chiste aleatorio:



```
<terminated> ComsumoAPI_REST - ComsumoApiRestApplication [Spring Boot App] C:\Users\savia\OneDrive\Escritorio\sts-4.21.0.RELEASE

: Starting ComsumoApiRestApplication using Java 17.0.9 with PID 1916 (C:\Users\savia\git\D0-UD6\ComsumoAPI_REST\bin\main started
: No active profile set, falling back to 1 default profile: "default"
: Started ComsumoApiRestApplication in 1.554 seconds (process running for 2.415)
: Joke[type=general, setup=I just watched a documentary about beavers., punchline=It was the best dam show I ever saw, id=41]
```

Y vemos los atributos que hemos puesto en la clase Joke: type, setup, punchline, id.