

Servicios REST

Programación de Servicios y Procesos

Israel Serrano

Pablo Baldazo

Jorge Buendía

Programación de Servicios

y Procesos

2ºDAM



Información del documento

Nombre del documento	Servicios REST
Autor del documento	Israel Serrano, Pablo Baldazo, Jorge Buendía
Versión del proyecto	v1.0
Fecha de entrega	21.11.2022
Repositorio GitHub	https://github.com/PIJ-Group/Servicios_REST.git

Distribución y autores del documento final

Nombre	Organización/Puesto
Israel Serrano	Todos los integrantes del equipo, hemos trabajado de manera conjunta durante la realización de este proyecto. Hemos dado importancia a la funcionalidad principal del programa, la cual, hemos desarrollado en conjunto. Posteriormente, cada miembro del equipo se centró en resolver distintos problemas que nos iban surgiendo durante el desarrollo del proyecto. Finalmente, pusimos en común todas las soluciones, para decidir cuál era la mejor y también conocer los errores cometidos.
Pablo Baldazo	
Jorge Buendía	



Contenido

1	Introducción	4
1.1	Objetivo	4
2	Requerimientos.....	5
3	Explicación de la actividad	7
3.1	Paquete servicioest	7
3.2	Paquete servicioest.modelo.entidad	8
3.3	Paquete servicioest.modelo.persistencia.....	10
3.4	Paquete servicioest.controlador.....	13
3.5	Paquete servicioest.cliente	17
3.6	Paquete servicioest.cliente.entidad	21
3.7	Paquete servicioest.cliente.servicio.....	23
4	Resultados	27
4.1	Funcionalidad del Servicio REST Servidor con Postman.....	27
4.2	Funcionalidad a través del Servicio REST Cliente	35
5	Conclusiones	41



1 Introducción

1.1 Objetivo

El objetivo de este proyecto es el de crear una aplicación REST que administre una serie de videojuegos, simulando un videoclub virtual que será gestionada a través de dos máquinas cliente-servidor realizadas mediante el framework Spring.



2 Requerimientos

Requerimiento 1

Se pide hacer un servicio REST, dicho servicio gestionará una serie de Videojuegos.

Los videojuegos tendrán un ID, un nombre, una compañía y una nota. Los videojuegos se encontrarán alojados en el servidor REST. Dicho servidor cuando arranque tendrá 5 videojuegos preestablecidos con todos los datos rellenos. Los videojuegos se guardarán en memoria en cualquier tipo de estructura de datos (como puede ser una lista).

El servicio REST proporcionará un servicio CRUD completo, y podrá ser consumido mediante un cliente como **Postman** y todo el intercambio de mensajes se hará a través de JSON.

Además, se pide a través de **Postman** realizar las siguientes tareas y apreciar los resultados:

- Dar de alta un videojuego
- Dar de baja un videojuego por ID
- Modificar un videojuego por ID
- Obtener un videojuego por ID
- Listar todos los videojuegos

Requerimiento 2

Se pide que no pueda haber dos videojuegos con id o nombre repetido



Requerimiento 3

Se pide crear una aplicación java que sea capaz de trabajar con el servidor REST de videojuegos. La aplicación mostrara un menú que sea capaz de realizar todas las operaciones del servidor, como puede ser:

- Dar de alta un videojuego
- Dar de baja un videojuego por ID
- Modificar un videojuego por ID
- Obtener un videojuego por ID
- Listar todos los videojuegos
- Salir

Para cada opción, se tendrá que pedir los datos necesarios al usuario y/o mostrar los resultados pertinentes. La aplicación se ejecutará hasta que se pulse la opción de “salir”.

Consideraciones

Para toda la actividad se valorará la claridad de código, la modularidad, la eficiencia de los algoritmos empleados y comentarios explicativos sobre los puntos clave de la aplicación (JavaDoc o normales, lo que se considere más apropiado).



3 Explicación de la actividad

SERVIDOR

Para llevar a cabo una correcta realización de la actividad, hemos creado dos proyectos **Spring**. El primero de ellos contiene toda la información y funcionalidad relativa al *Servidor*, mientras que el segundo hace referencia al *Cliente*, el cual consumirá la aplicación servidora.

En primer lugar, nos centraremos en el proyecto que contiene el servidor, al cual hemos denominado  `Servicios_REST_Servidor [boot] [Servicios_REST main]`

Ahora, iremos explicando brevemente la estructura de paquetes y clases más importantes del mismo:

3.1 Paquete `servicioest`

3.1.1. Clase `ServiciosRestApplication`

```
1 package servicioest;
2
3 import org.springframework.boot.SpringApplication;
4
5
6 @SpringBootApplication
7 public class ServiciosRestApplication {
8
9     public static void main(String[] args) {
10         System.out.println("Servicio Rest -> Cargando el contexto de Spring");
11
12         SpringApplication.run(ServiciosRestApplication.class, args);
13
14         System.out.println("Servicio Rest -> Contexto de Spring cargado");
15     }
16 }
17
18 }
```

Esta clase será el punto de arranque de nuestra aplicación. Utilizaremos la anotación **@SpringBootApplication** para que Spring inyecte dependencias y pueda dar de alta distintos objetos utilizando anotaciones en las clases, las cuales



estarán en este mismo paquete o en paquetes hijos, para poder ser localizadas por Spring.

Además, hemos añadido dos trazas que nos indican cuando empieza a cargarse el contexto de Spring y cuando está totalmente cargado.

3.1.2. Clase *ServletInitializer*

```
1 package servicioest;
2
3 import org.springframework.boot.builder.SpringApplicationBuilder;
4
5
6 public class ServletInitializer extends SpringBootServletInitializer {
7
8     @Override
9     protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
10         return application.sources(Application.class);
11     }
12
13 }
14
```

La siguiente clase se crea por defecto cuando desplegamos un proyecto Spring Boot, no es más que un **WebApplicationInitializer** que ofrece Spring Boot para configurar la aplicación web (es la encargada de crear el **WebApplicationContext** y su configuración).

3.2 Paquete `servicioest.modelo.entidad`

3.2.1. Clase *Videojuego*

```
1 package servicioest.modelo.entidad;
2
3 import java.util.Objects;
4
5 public class Videojuego {
6     private int id;
7     private String nombre;
8     private String compañía;
9     private double nota;
10
11     public Videojuego() {
12         super();
13     }
14
15     public Videojuego(int id, String nombre, String compañía, Double nota) {
16         this.id = id;
17         this.nombre = nombre;
18         this.compañía = compañía;
19         this.nota = nota;
20     }
21 }
```

Esta clase será la entidad con la que vamos a trabajar en nuestro servicio REST.



```
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getNombre() {
    return nombre;
}
public void setNombre(String nombre) {
    this.nombre = nombre;
}
public String getCompañia() {
    return compañia;
}
public void setCompañia(String compañia) {
    this.compañia = compañia;
}
public double getNota() {
    return nota;
}
public void setNota(double nota) {
    this.nota = nota;
}
}
```

La clase está compuesta por varios atributos, así como por los constructores y también los getters y setters.

```
@Override
public String toString() {
    return "Videojuegos [id=" + id + ", nombre=" + nombre + ", compañia=" + compañia + ", nota=" + nota + "]";
}

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

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

Para terminar, además de introducir el método `toString`, también hemos realizado los métodos `hashCode` y `equals`, los cuales necesitaremos más adelante para restringir la acción de añadir un nuevo videojuego con ID o nombres repetidos.



3.3 Paquete servicioest.modelo.persistencia

3.3.1. Clase DaoVideojuego

```
@Component
public class DaoVideojuego {
    private List<Videojuego> listaVideojuegos;

    public DaoVideojuego () {
        super();
        System.out.println("DaoVideojuego -> Cargando lista de videojuegos...");

        listaVideojuegos = new ArrayList<Videojuego>();

        Videojuego v1 = new Videojuego(657, "Tetris", "Spectrum Holobyte", 9.0);
        Videojuego v2 = new Videojuego(852, "DOOM 64", "Bethesda", 9.8);
        Videojuego v3 = new Videojuego(951, "The Legend Of Zelda: The Ocarina of Time", "Nintendo", 9.3);
        Videojuego v4 = new Videojuego(753, "The Last Of Us", "Naughty Dog", 8.9);
        Videojuego v5 = new Videojuego(654, "Arkanoid", "Taito Corporation", 6.4);

        listaVideojuegos.add(v1);
        listaVideojuegos.add(v2);
        listaVideojuegos.add(v3);
        listaVideojuegos.add(v4);
        listaVideojuegos.add(v5);
    }
}
```

Esta clase se rige por el patrón DAO (Data Access Object), objeto que se encarga de hacer las diferentes consultas. En este caso vamos a simular una pequeña base de datos que almacena videojuegos, trabajando con una lista de objetos cargados en memoria.

Antes de continuar, debemos escribir la anotación **@Component**, encima de la clase DaoVideojuego. Esto significa que estamos dando de alta un único objeto de esta clase dentro del contexto de Spring, cuyo nombre, será el nombre de dicha clase en notación lowerCamelCase.

Una vez escrita la anotación, creamos una variable de tipo List <Videojuego>, donde cargaremos posteriormente 5 videojuegos predefinidos.

Una vez cargados los videojuegos, nos centraremos en los distintos métodos y su funcionalidad:

```
//AÑADIR
public Videojuego añadirVideojuego(Videojuego v) {

    int cont = 0;

    if(listaVideojuegos.contains(v)) {
        System.out.println("Añadir => Videojuego en lista");
        return null;
    }else {

        listaVideojuegos.add(v);
        cont = listaVideojuegos.size() -1;
        System.out.println("Añadido el videojuego => " + listaVideojuegos.get(cont) + " a la lista");
        return listaVideojuegos.get(cont);
    }
}
```



- **Añadir:** Este método añade un videojuego a la lista, estableciendo como parámetro del mismo, un videojuego completo. Utilizaremos el método `.contains`, el cual se apoya directamente en los métodos `equals` y `hashCode`. Por lo tanto, si el juego a añadir contiene un ID o nombre que coincida con los que ya están en lista, nos saldrá la traza en el servidor de "Videojuego en lista", diciendo que el videojuego no puede ser añadido. En caso contrario, añadiremos el videojuego a la lista y guardaremos la última posición del array en una variable auxiliar creada al inicio del método, para así poder devolver el nuevo videojuego que se ha añadido en la última posición del array.

```
//BORRAR
public Videojuego borrarVideojuego(int id) {

    try {

        for(Videojuego v : listaVideojuegos) {

            if(v.getId() == id){

                int vAux = listaVideojuegos.indexOf(v);
                System.out.println(vAux);
                System.out.println("Borrar => Videojuego " + v + "");
                return listaVideojuegos.remove(vAux);

            }

        }

    } catch (UnsupportedOperationException e) {

        System.out.println("Borrar => El videojuego no se encuentra en la lista ");
        return null ;

    }

    return null;

}
```

- **Borrar:** El siguiente método, borra un videojuego a través pasándole un ID, acto seguido, recorreremos nuestra lista de videojuegos con un *for each* y vamos comparando el ID que nos pasan con los de los videojuegos almacenados.
Creamos una variable auxiliar donde igualamos la posición de la variable que itera el `arrayList` y en la cual ha coincidido el ID. Para terminar, trazamos el videojuego que vamos a eliminar y finalmente lo eliminamos.



```
//ACTUALIZAR
public Videojuego actualizar(Videojuego v) {

    try {
        for(Videojuego vid : listaVideojuegos) {
            if(vid.getId() == v.getId()) {
                int vAux = listaVideojuegos.indexOf(vid);
                Videojuego vix = listaVideojuegos.get(vAux);
                vix.setNombre(v.getNombre());
                vix.setCompañia(v.getCompañia());
                vix.setNota(v.getNota());
                return vix;
            }
        }

    } catch (IndexOutOfBoundsException e) {
        System.out.println(" Actualizar => El videojuego no está en la lista");
        return null;
    }
    return null;
}
```

- Actualizar: Método al que le pasamos un Videojuego completo, iteramos nuevamente la lista buscando el ID del videojuego que queramos modificar.

Creamos una variable auxiliar donde guardaremos la posición del ID que haya coincidido y posteriormente la añadiremos en su respectiva posición a una variable de clase Videojuego.

Después establecemos un nombre, una compañía y una nota y los recogemos dentro de ese objeto Videojuego que nos habían pasado en primera instancia. Finalmente retornamos el videojuego.

```
//BUSCAR
public Videojuego buscarVideojuego(int id) {
    for(Videojuego v : listaVideojuegos) {
        if(v.getId() == id){
            return v;
        }
    }
    return null;
}

//LISTAR
public List<Videojuego> listarVideojuegos(){
    return listaVideojuegos;
}
```

- Buscar: El siguiente método devuelve un Videojuego, pasándole un ID, volvemos a recorrer la lista y establecemos la condición de que cuando coincida dicho ID, retornemos el videojuego.



- Listar: Retornamos un arrayList de videojuegos.

Destacar únicamente, que en algunos de los métodos hemos introducido bloques Try-Catch, con las excepciones `IndexOutOfBoundsException` y `UnsupportedOperationException` para controlar y capturar todos aquellos errores que puedan afectar al flujo normal del programa.

3.4 Paquete `serviciorest.controlador`

3.3.2. Clase `ControladorVideojuegos`

En esta clase, vamos a realizar un CRUD completo contra la entidad `Videojuego`. Lo primero de todo, será dar de alta el objeto `ControladorVideojuegos` con la anotación **`@RestController`**, que a su vez necesita de un objeto de tipo `DaoPersona`, que ya dimos de alta anteriormente con la anotación **`@Component`**.

Después utilizamos la anotación **`@Autowired`**, que realiza una inyección de dependencias entre objetos dados de alta en el contexto de Spring. Con esta acción le estamos diciendo que inyecte un objeto de tipo `DaoVideojuego` a la referencia `daoVideojuego`.

Ahora nos centraremos en explicar los diferentes puntos de acceso:

```
@RestController
public class ControladorVideojuegos {

    @Autowired
    private DaoVideojuego daoVideojuego;

    //AÑADIR
    @PostMapping(path = "videojuegos", consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Videojuego> agregarVideojuego(@RequestBody Videojuego v){
        System.out.println("Agregar => Intentando dar de alta el videojuego: " + v);
        daoVideojuego.añadirVideojuego(v);
        if (v != null)
            return new ResponseEntity<Videojuego>(v, HttpStatus.CREATED); //201 videojuego creado
        else
            return new ResponseEntity<Videojuego>(HttpStatus.I_AM_A_TEAPOT); //418 soy una tetera
    }
}
```


- Añadir: En este caso, vamos a dar de alta a una persona, utilizaremos el método `Post` a través de la anotación **`@PostMapping`**. Nuestro `path` (ruta) será “videojuegos” e irá acompañado, en este caso de dos atributos claves:



1. **Produces**, que se encargará de serializar automáticamente el resultado a JSON.
2. **Consumes**, que será el encargado de enviar el formato en el que el cliente nos enviará la información.

El siguiente paso será el de responder adecuadamente, utilizando la clase `ResponseEntity`. Esta clase encapsula tanto el resultado en JSON como el código de respuesta del protocolo HTTP.

Finalmente obtenemos el videojuego que nos envíe el cliente a través de la anotación **@RequestBody**, con el que Spring deserializará automáticamente el JSON.

Acto seguido trazaremos una ruta de control y retornaremos el videojuego, así como el código de respuesta correspondiente si se ha creado satisfactoriamente `201 (CREATED)` o el código `418 (I_AM_A_TEAPOT)`, si no hemos podido añadir el videojuego, gastando una pequeña broma al usuario .

```
//BORRAR
@DeleteMapping(path = "videojuegos/{id}")
public ResponseEntity<Videojuego> borrarVideojuego(@PathVariable("id") int id){
    System.out.println("Borrar => Borrando videojuego con id : " + id);
    Videojuego v = daoVideojuego.borrarVideojuego(id);
    if(v != null)
        return new ResponseEntity<Videojuego>(HttpStatus.OK); //200 videojuego borrado
    else
        return new ResponseEntity<Videojuego>(HttpStatus.NOT_FOUND); //404 videojuego no encontrado
}
```

- **Borrar**: En esta ocasión, vamos a borrar un videojuego y realmente tenemos una estructura similar al caso anterior, con la diferencia de que en esta ocasión utilizaremos el verbo `Delete` con la anotación **@DeleteMapping** y en el *path*, tenemos que introducir el `{ID}` que podemos obtener usando la anotación **@PathVariable("id")** dentro de un parámetro de entrada. El "id" se corresponde con el "{ID}", es decir, deben de llamarse igual.



Finalmente, si la variable *v*, es distinta de null, devolvemos el videojuego con el código de respuesta *200 (OK)*, en caso contrario, devolveremos el código *404 (NOT FOUND)*, videojuego no encontrado.

```
//MODIFICAR
@PutMapping(path = "videojuegos/{id}", consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Videojuego> modificarVideojuego(@PathVariable("id") int id, @RequestBody Videojuego v){
    System.out.println("Modificar => Modificando videojuego con ID : " + id);
    Videojuego vid = daoVideojuego.actualizar(v);
    if(vid != null) {
        System.out.println("Modificar => Videojuego modificado : " + v);
        return new ResponseEntity<Videojuego>(HttpStatus.OK); //200 videojuego modificado
    }else
        return new ResponseEntity<Videojuego>(HttpStatus.NOT_FOUND); //404 videojuego no encontrado
}
```

- **Modificar:** En este punto de acceso vamos a modificar un videojuego, en esta ocasión utilizaremos el verbo Put a través de la anotación **@PutMapping**.

Nuestro *path*, volverá a ser “videojuegos/{id}”, puesto que buscaremos el videojuego a modificar introduciendo el ID del mismo y con el atributo **consumes** seguido de la clase **MediaType** estableceremos la serialización a JSON nuevamente. Obtenemos el ID con la anotación **@PathVariable** y la respuesta del usuario con la anotación **@RequestBody**.

Realizamos una pequeña traza y volvemos a dar las mismas respuestas, *200 (OK)* en caso de modificarlo correctamente o *404 (NOT FOUND)* en caso de no haber podido encontrarlo.

```
//BUSCAR
@GetMapping(path = "videojuegos/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Videojuego> obtenerVideojuego(@PathVariable("id")int id){
    System.out.println("Obtener => Buscando videojuego con el id:" +id);
    Videojuego v = daoVideojuego.buscarVideojuego(id);
    if(v != null) {
        return new ResponseEntity<Videojuego> (v,HttpStatus.OK);//200 OK
    }else
        return new ResponseEntity<Videojuego> (HttpStatus.NOT_FOUND);//404 NOT FOUND
}
```

- **Buscar:** Ahora vamos a buscar un videojuego por su ID, en este caso, podemos observar que tenemos prácticamente las mismas sentencias que el método anterior.



La única diferencia es que aquí no recogemos ninguna respuesta del usuario y utilizamos el verbo *Get*, seguido de la anotación **@GetMapping**. Además, no hay una interacción más allá de digitar el ID del videojuego que se desea buscar y finalmente devolvemos los mismos códigos de respuesta correspondientes.

```
//LISTAR
@GetMapping(path="videojuegos/lista",produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<List<Videojuego>> listarVideojuegos(){
    System.out.println("Listar => Mostrando la lista de videojuegos");
    List <Videojuego> v = daoVideojuego.listarVideojuegos();
    return new ResponseEntity <List<Videojuego>> (v,HttpStatus.OK);
}
```

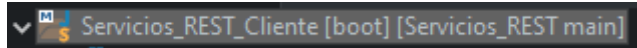
- **Listar:** El último caso es ligeramente diferente, puesto que cambia nuestro *path* a “videojuegos/lista”. También será diferente nuestra respuesta, puesto que será **<List<Videojuego>>**, ya que vamos a producir la lista completa de videojuegos y la serializaremos a JSON.
En este caso no habrá una condición que evaluar, únicamente retornaremos la lista junto con el código de respuesta 200 (OK).



CLIENTE

El segundo proyecto hace referencia al *Cliente*, el cual consumirá la aplicación servidora.

Dicho proyecto lo hemos denominado de la siguiente manera:



En los siguientes puntos explicaremos la estructura de paquetes y clases que contiene:

3.5 Paquete `servicioest.cliente`

3.5.1. Clase *ServiciosRestClienteApplication*

Esta clase será el punto de arranque de nuestra aplicación cliente. Utilizaremos la anotación **SpringBootApplication** para que Spring inyecte dependencias y pueda dar de alta distintos objetos utilizando anotaciones en las clases, las cuales estarán en este mismo paquete o en paquetes hijos, para poder ser localizadas por Spring.

En este caso, a diferencia de la misma clase en la aplicación servidor, contiene la lógica del menú y las opciones que podrá seleccionar el cliente, las cuáles pasamos a detallar a continuación:

```
@SpringBootApplication
public class ServiciosRestClienteApplication implements CommandLineRunner {

    Scanner sc = new Scanner(System.in);

    @Autowired
    private ServicioProxyVideojuego spv;

    @Autowired
    private ApplicationContext context;

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

    public static void main(String[] args) {

        System.out.println("Cliente => Cargando el contexto de Spring");
        SpringApplication.run(ServiciosRestClienteApplication.class, args);
    }
}
```



La clase implementa la interfaz **CommandLineRunner**, la cual nos obliga a implementar un método `run`, ya que, al tener las variables y objetos dinámicos, no podríamos acceder desde el `main` a ninguna de ellas al tratarse de un objeto estático, por lo que delegamos la función del `main` al método `run` que obliga a implementar.

Con la anotación **@Autowired** inyectamos la dependencia de las clases **ServicioProxyVideojuego** y **ApplicationContext**, para así poder acceder a todos sus métodos y en el caso del contexto de Spring, con dicha variable podremos cerrar el servidor del cliente en el caso de pulsar cierta opción del menú.

Además, con la anotación **@Bean**, creamos un objeto `RestTemplate`, mediante un método, que será el que use el objeto `ServicioProxyVideojuego` para poder realizar las peticiones HTTP a nuestro servidor Rest.

Por último, en el `main` de la clase creamos una traza que diga que se está cargando el contexto de Spring y el método `run` de la aplicación que ejecuta la clase.

```
@Override
public void run(String... args) throws Exception {

    System.out.println("Cliente => Contexto de Spring cargado");

    //Inicialización de variables
    int opcion;
    Videojuego vAux;
    List <Videojuego> listaVideojuegos = null;
    int idAux;

    do {
        opcion = menu();

        switch(opcion) {

            case 1:
                vAux = new Videojuego();

                System.out.println("Añade el ID: ");
                vAux.setId(sc.nextInt());
                sc.nextLine();

                System.out.println("Añade el nombre");
                vAux.setNombre(sc.nextLine());

                System.out.println("Añade la compañía");
                vAux.setCompañia(sc.nextLine());

                System.out.println("Añade la nota");
                vAux.setNota(sc.nextDouble());

                spv.añadirVideojuego(vAux);
                break;
        }
    } while (opcion != 0);
}
```



```
case 2:
    System.out.println("Escribe el ID del videojuego a borrar: ");
    idAux = sc.nextInt();

    spv.borrarVideojuego(idAux);
    break;

case 3:
    vAux = new Videojuego();

    System.out.println("Escribe el ID del videojuego a modificar: ");
    vAux.setId(sc.nextInt());
    sc.nextLine();

    System.out.println("Escribe el nombre a modificar: ");
    vAux.setNombre(sc.nextLine());

    System.out.println("Escribe la compañía a modificar: ");
    vAux.setCompañia(sc.nextLine());

    System.out.println("Escribe la nota a modificar: ");
    vAux.setNota(sc.nextDouble());

    spv.actualizarVideojuego(vAux);
    break;

case 4:
    System.out.println("Escribe el ID del videojuego a buscar: ");
    idAux = sc.nextInt();
    vAux = spv.buscarVideojuego(idAux);
    System.out.println(vAux);
    break;

case 5:
    listaVideojuegos = spv.listarVideojuego();

    for(Videojuego v : listaVideojuegos)
        System.out.println(v);
    break;

case 6:
    salir();
    break;

default:
    System.out.println("Elige una opción correcta por favor");
    break;
}
}while(opcion != 6);
}
```

En el método run implementado crearemos un switch del menú del cliente, en el cual interactuamos por consola con el cliente pidiendo una serie de datos, que serán pasados por argumento a los métodos de la variable ServicioProxyVideojuego, que será la encargada de la función de conexión con el servidor.

Inicializamos unas variables auxiliares que nos ayudarán a guardar los datos recogidos por el cliente y utilizarlos en sus respectivos métodos.



Como norma general, en todos los casos del switch se pedirán los datos correspondientes a cada método por consola, depende del caso, creamos un objeto Videojuego auxiliar, en otros una variable para guardar el ID del videojuego e incluso una lista auxiliar para guardar la lista que devuelve el servidor con todos los videojuegos cargados.

Sin embargo, el caso 6, precisa de especial atención porque será el encargado de cerrar la aplicación cliente en caso de que este lo desee, en el cual ejecutamos el método salir () (que describiremos a continuación).

```
public int menu() {  
    System.out.println("\n ----- VIDEOCLUB PIJ ----- \n");  
    System.out.println("1. Dar de alta un videojuego");  
    System.out.println("2. Dar de baja un videojuego por ID");  
    System.out.println("3. Modificar un videojuego por ID");  
    System.out.println("4. Obtener un videojuego por ID");  
    System.out.println("5. Listar todos los videojuegos");  
    System.out.println("6. Salir de la aplicación");  
    int option = sc.nextInt();  
    System.out.println("Has elegido la opción: " + option);  
    return option;  
}  
  
public void salir() {  
    SpringApplication.exit(context, () ->0);  
    System.out.println("El Cliente ha cerrado la conexión");  
}
```

La función menú devuelve el menú con el que el cliente interactúa para poder solicitar al servidor una función concreta.

Por último, la función salir () hace que, al seleccionar el cliente la opción 6 del menú, pueda cerrar la aplicación, en cuya lógica utilizamos el contexto creado al inicio de la clase, apoyándonos de una función anónima para ahorrar código.



3.5.2. Clase ServletInitializer

```
1 package servicioest.cliente;
2
3 import org.springframework.boot.builder.SpringApplicationBuilder;
4
5
6 public class ServletInitializer extends SpringBootServletInitializer {
7
8     @Override
9     protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
10         return application.sources(ServiciosRestClienteApplication.class);
11     }
12
13 }
14 }
```

La siguiente clase se crea por defecto cuando desplegamos un proyecto Spring Boot, no es más que un **WebApplicationInitializer** que ofrece Spring Boot para configurar la aplicación web (es la encargada de crear el **WebApplicationContext** y su configuración).

3.6 Paquete servicioest.cliente.entidad

3.6.1. Clase Videojuego

Esta clase será la entidad de la aplicación cliente, es la misma que la clase Videojuego del proyecto Servidor.

```
package servicioest.cliente.entidad;

import java.util.Objects;

public class Videojuego {
    private int id;
    private String nombre;
    private String compañía;
    private double nota;

    public Videojuego() {
        super();
    }

    public Videojuego(int id, String nombre, String compañía, Double nota) {
        this.id = id;
        this.nombre = nombre;
        this.compañía = compañía;
        this.nota = nota;
    }
}
```

La clase consta de los atributos de la imagen, con un constructor que contiene todos los atributos disponibles.



```
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getNombre() {
    return nombre;
}
public void setNombre(String nombre) {
    this.nombre = nombre;
}
public String getCompañia() {
    return compañia;
}
public void setCompañia(String compañia) {
    this.compañia = compañia;
}
public double getNota() {
    return nota;
}
public void setNota(double nota) {
    this.nota = nota;
}
```

Tiene una serie de getters y setters para poder acceder a sus atributos.

```
@Override
public String toString() {
    return "Videojuegos [id=" + id + ", nombre=" + nombre + ", compañia=" + compañia + "]";
}

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

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

Y, por último, para poder cumplir con el requerimiento de la no repetición del ID y nombre, se crean los métodos hashCode () y equals (), en los que pasamos dichos atributos para restringirlos.



3.7 Paquete `serviciorest.cliente.servicio`

3.7.1. Clase *ServicioProxyVideojuego*

Esta clase será la encargada de la conexión y protocolo de envío/recepción de datos con el servidor.

```
package serviciorest.cliente.servicio;

import java.util.Arrays;

@Service
public class ServicioProxyVideojuego {
    public static final String URL = "http://localhost:8080/videojuegos/";

    @Autowired
    private RestTemplate restTemplate;
```

Para ello utilizamos la anotación **@Service**, que construirá la clase de Servicio que conectará con el servidor.

Creamos una variable para acumular la dirección en la que estará alojado nuestro servidor, en nuestro caso <http://localhost:8080/videojuegos/>, en la que terminaremos con el símbolo "/" para poder concatenar a continuación el resto del path dependiendo del método.

Además, con la etiqueta **@Autowired** inyectamos en esta clase el objeto **RestTemplate** creado en la anterior clase *ServiciosRestClienteApplication*, como explicamos anteriormente en dicha clase, nos servirá para realizar las conexiones con el servidor.

A continuación, pasamos a detallar los métodos de conexión del CRUD realizado:

```
//AÑADIR
public Videojuego añadirVideojuego(Videojuego v) {
    try {
        ResponseEntity <Videojuego> re = restTemplate.postForEntity(URL, v, Videojuego.class);
        System.out.println("Alta Videojuego => Código de respuesta " + re.getStatusCode());
        return re.getBody();
    } catch (HttpClientErrorException e) {
        System.out.println("Alta Videojuego => El videojuego no se puede dar de alta");
        System.out.println(e.getStatusCode());
        return null;
    }
}
```

- **Método añadir:** Creamos una variable apoyándonos en la clase `ResponseEntity` para poder recibir los datos del body de la petición HTTP y su código de estado.



Dicha variable la igualamos a la variable `restTemplate` creada en la clase `ServicioRestClienteApplication` con el fin de crear el protocolo HTTP, siendo en el caso del método añadir, el método **`postForEntity`**.

Para poder acceder a la URL, por parámetro le pasamos la URL creada como variable al inicio de la clase.

Además, también le incluimos como argumento del método el parámetro de la función y el casteo a la clase `Videojuego`.

Por último, terminamos el bloque Try devolviendo el body del objeto `ResponseEntity`.

El bloque catch contará con las excepciones necesarias a controlar, en este caso `HttpClientErrorException` y devolverá null.

```
//BORRAR
public boolean borrarVideojuego(int id) {
    try {
        restTemplate.delete(URL + id);
        System.out.println("Videojuego con ID " + id + " ha sido borrado correctamente.");
        return true;
    } catch (HttpClientErrorException e) {
        System.out.println("Borrar Videojuego => El videojuego con ID " + id + " no se ha podido borrar");
        System.out.println(e.getStatusCode());
        return false;
    }
}
```

- **Método borrar:** En este caso, hemos decidido crear un método booleano, ya que, al borrar el videojuego, no habría forma de pasarle que Videojuego se ha borrado de la base de datos, por lo que confirmaremos con un true o false.

Utilizaremos el método delete del objeto `restTemplate`, al cual le pasamos la variable URL a la que sumamos el parámetro ID añadido, para poder completar la dirección completa de esa operación del servidor, en este caso <http://localhost:8080/videojuegos/id>.

Para acabar con el Try, devolvemos el valor true.

El bloque catch contará con las excepciones necesarias a controlar, en este caso `HttpClientErrorException` y devolverá un false.



```
//ACTUALIZAR
public boolean actualizarVideojuego(Videojuego v) {
    try {
        restTemplate.put(URL + v.getId(), v, Videojuego.class);
        System.out.println("El videojuego con ID " + v.getId() + " se ha modificado correctamente");
        return true;
    } catch (HttpClientErrorException e) {
        System.out.println("Actualizar Videojuego => El videojuego con ID " + v.getId() + " no se ha podido actualizar");
        System.out.println(e.getStatusCode());
        return false;
    }
}
```

- **Método actualizar:** Igual que en el método anterior, se crea un método booleano al cual le pasamos como parámetro un Videojuego. En este caso, utilizamos el método put de restTemplate, al que le pasamos por parámetro la URL, le sumamos el ID del objeto introducido por parámetro (así conseguimos la dirección del ejemplo anterior), el videojuego introducido por parámetro y el casteo a Videojuego.

Terminamos el bloque Try devolviendo un true, mientras que el bloque catch contará con las excepciones necesarias a controlar, en este caso HttpClientErrorException y devolverá un false.

```
//BUSCAR
public Videojuego buscarVideojuego(int id) {
    try {
        ResponseEntity <Videojuego> re = restTemplate.getForEntity(URL + id, Videojuego.class);
        System.out.println("Buscar Videojuego => Código de respuesta " + re.getStatusCode());
        return re.getBody();
    } catch (HttpClientErrorException e) {
        System.out.println("Buscar Videojuego => El videojuego con ID " + id + " no esta en la lista");
        System.out.println(e.getStatusCode());
        return null;
    }
}
```

- **Método buscar:** Método por el cual devolvemos un objeto que recogeremos el body de la petición con el método getBody (). Como en el primer método, nos apoyamos en crear un objeto ResponseEntity de Videojuego para poder acceder a las trazas y al body de la petición.

La dirección de la petición la conseguiremos utilizando el método getForEntity de restTemplate, al que pasamos como parámetro la URL y le sumamos el ID que añadimos como parámetro del método buscarVideojuego (<http://localhost:8080/videojuegos/id>), por último, realizamos el casteo a Videojuego.



Al final del bloque Try, devolvemos el body nombrado al principio del método, mientras que el bloque catch contará con las excepciones necesarias a controlar, en este caso `HttpClientErrorException` y devolverá `null`.

```
//LISTAR
public List<Videojuego> listarVideojuego(){
    try {
        ResponseEntity <Videojuego[]> re = restTemplate.getForEntity(URL + "lista", Videojuego[].class);
        Videojuego[] listAux = re.getBody();
        return Arrays.asList(listAux);
    } catch (HttpClientErrorException e) {
        System.out.println("Listar Videojuegos => Error al listar los videojuegos");
        System.out.println(e.getStatusCode());
        return null;
    }
}
```

- **Método Listar:** El último método devolverá una lista de todos los videojuegos del servidor.

Nos apoyamos en un objeto de `ResponseEntity` para poder obtener el body del mensaje HTTP, el cual cargaremos en un array auxiliar de la clase `Videojuego`.

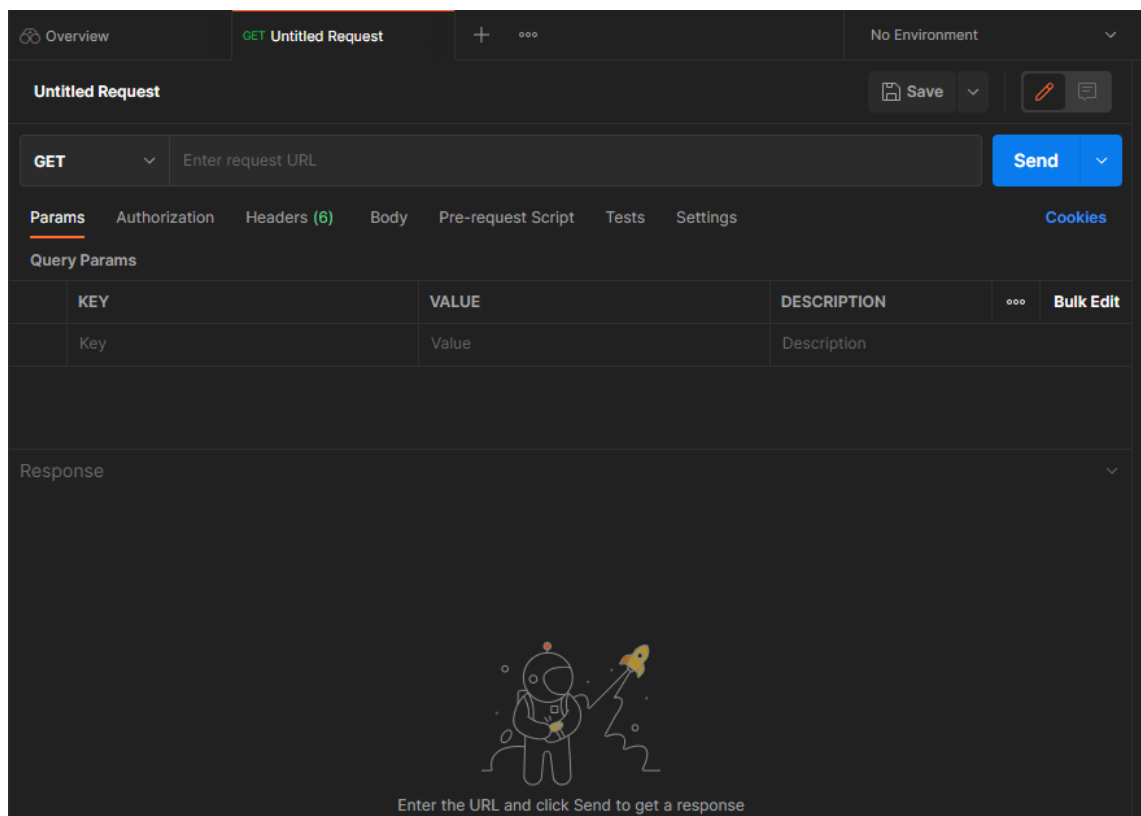
Para mostrarlo, lo devolvemos utilizando el método **`asList`** de la clase `Arrays`, para poder transformarlo de array simple a `ArrayList`.

Para poder obtener la dirección del recurso, utilizamos el método `getForEntity` de `restTemplate` al cual le pasamos como argumento la URL y sumamos una cadena “lista”, para completar el path de dicho recurso (<http://localhost:8080/videojuegos/listar>). Por último, se hará un casteo a un array de tipo `Videojuego`.

Creamos una lista auxiliar en la cual cargamos el body de la petición.

Terminamos el bloque Try con la devolución del array anteriormente descrito.

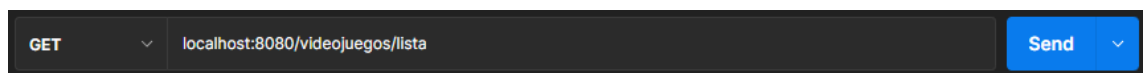
El bloque catch contará con las excepciones necesarias a controlar, en este caso `HttpClientErrorException` y devolverá `null`.



- Dar de alta un videojuego

Para mostrar como añadimos un videojuego, primero vamos a mostrar la lista de los videojuegos que tenemos ahora mismo.

Para ello introducimos la URL de la solicitud (URL request) **localhost:8080/videojuegos/lista**, tal y como vemos en la imagen y pulsamos enviar.





Obteniendo el siguiente resultado.

```
1  {
2    {
3      "id": 657,
4      "nombre": "Tetris",
5      "compañia": "Spectrum Holobyte",
6      "nota": 9.0
7    },
8    {
9      "id": 852,
10     "nombre": "DOOM 64",
11     "compañia": "Bethesda",
12     "nota": 9.8
13   },
14   {
15     "id": 951,
16     "nombre": "The Legend Of Zelda: The Ocarina of Time",
17     "compañia": "Nintendo",
18     "nota": 9.3
19   },
20   {
21     "id": 753,
22     "nombre": "The Last Of Us",
23     "compañia": "Naughty Dog",
24     "nota": 8.9
25   },
26   {
27     "id": 654,
28     "nombre": "Arkanoid",
29     "compañia": "Taito Corporation",
30     "nota": 6.4
31   }
32 }
```

Ahora que tenemos la lista de los videojuegos vamos a añadir uno nuevo. Para ello, en la URL request, escribimos la URL localhost:8080/videojuegos, seteando en el desplegable el verbo **POST** como vemos en la imagen siguiente.



Justo debajo de la barra de la URL request, elegimos la opción Body y seleccionamos raw, en el desplegable de la derecha ponemos JSON, puesto que es el formato en el que vamos a enviar y recibir la información, y en el cuadro de texto introducimos en dicho formato, el videojuego que pretendemos añadir.



```
1 {
2   ...."id": 993,
3   ...."nombre": "League Of Legends",
4   ...."compañia": "Riot Games",
5   ...."nota": 11
6 }
7
8
```

Pulsamos en el botón **Send**, y obtenemos el código de estado 201 Created.

```
201 Created 64 ms 245 B
```

Para comprobar que se añadió en nuestra lista de videojuegos, volvemos a mostrarla como hicimos en el primer paso, comprobando que el videojuego se añadió correctamente al final del listado.

```
25 },
26 {
27   "id": 654,
28   "nombre": "Arkanoid",
29   "compañia": "Taito Corporation",
30   "nota": 6.4
31 },
32 {
33   "id": 993,
34   "nombre": "League Of Legends",
35   "compañia": "Riot Games",
36   "nota": 11.0
37 }
38
```

Mientras hemos estado haciendo todas estas consultas a través del cliente **Postman**, en la consola de nuestro Servicio REST han ido apareciendo todos los mensajes que hemos considerado poner para trazar la funcionalidad del servicio.

```
Listar => Mostrando la lista de videojuegos
Agregar => Intentando dar de alta el videojuego: Videojuegos [id=993, nombre=League Of Legends, compañía=Riot Games, nota=11.0]
Añadido el videojuego => Videojuegos [id=993, nombre=League Of Legends, compañía=Riot Games, nota=11.0] a la lista
Listar => Mostrando la lista de videojuegos
```

En el caso de ir a introducir un videojuego con un ID o un Nombre que ya esté en la lista, no lo añade y nos saca un mensaje de error en la consola, de que el videojuego ya está en la lista.



```
Params  Authorization  Headers (8)  Body  Pre-request Script  Tests  Settings
none  form-data  x-www-form-urlencoded  raw  binary  GraphQL  JSON
1  {
2    "id": 852,
3    "nombre": "DOOM 64",
4    "compañia": "Bethesda",
5    "nota": 9.8
6  }
```

```
Agregar => Intentando dar de alta el videojuego: Videojuegos [id=852, nombre=DOOM 64, compañía=Bethesda, nota=9.8]
Añadir => Videojuego en lista
```


- Dar de baja un videojuego por ID

La siguiente funcionalidad que vamos a mostrar es la de borrar un videojuego de la lista.

Como ya hemos mostrado anteriormente los videojuegos que tenemos, elegimos uno que vamos a dar de baja.

Para ello en la barra de URL request escribimos **localhost:8080/videojuegos/753**, donde 753 es el ID del videojuego que queremos dar de baja, y seleccionamos el verbo **DELETE**.

```
DELETE  localhost:8080/videojuegos/753  Send
```

Pulsamos en **Send** y obtenemos el código de estado 200 OK  200 OK 17 ms 123 B

Volvemos a mostrar los videojuegos, en donde comprobamos que el videojuego con ID 753, ya no existe.



```
1  [
2    {
3      "id": 657,
4      "nombre": "Tetris",
5      "compañia": "Spectrum Holobyte",
6      "nota": 9.0
7    },
8    {
9      "id": 852,
10     "nombre": "DOOM 64",
11     "compañia": "Bethesda",
12     "nota": 9.8
13   },
14   {
15     "id": 951,
16     "nombre": "The Legend Of Zelda: The Ocarina of Time",
17     "compañia": "Nintendo",
18     "nota": 9.3
19   },
20   {
21     "id": 654,
22     "nombre": "Arkanoid",
23     "compañia": "Taito Corporation",
24     "nota": 6.4
25   },
26   {
27     "id": 993,
28     "nombre": "League Of Legends",
29     "compañia": "Riot Games",
30     "nota": 11.0
31   }
32 ]
```

Mientras tanto en la consola del Servicio REST se va trazando todo lo que hacemos.

```
Borrar => Borrando videojuego con id : 753
3
Borrar => Videojuego Videojuegos [id=753, nombre=The Last Of Us, compañia=Naughty Dog, nota=8.9]
Listar => Mostrando la lista de videojuegos
```

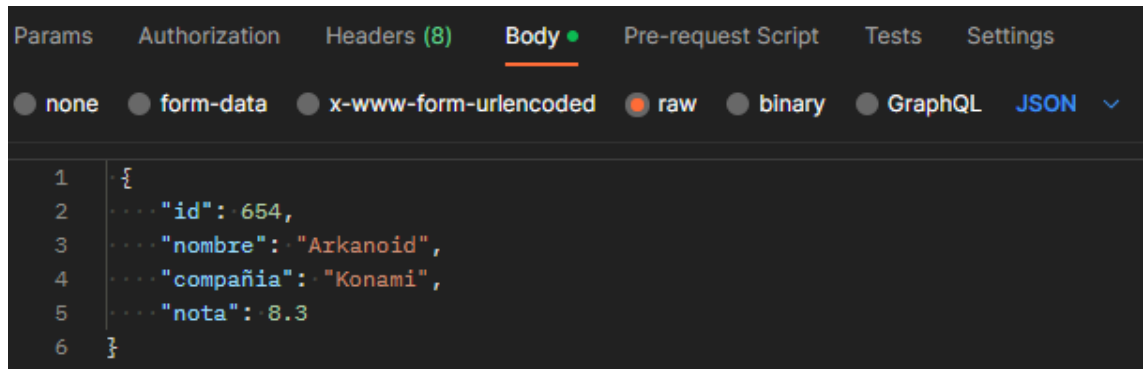
- Modificar un videojuego por ID

Es este caso vamos a usar el verbo **PUT** e introducimos en la URL request **localhost:8080/videojuegos/654** donde 654 es el ID del videojuego que vamos a modificar,

```
PUT    localhost:8080/videojuegos/654    Send
```



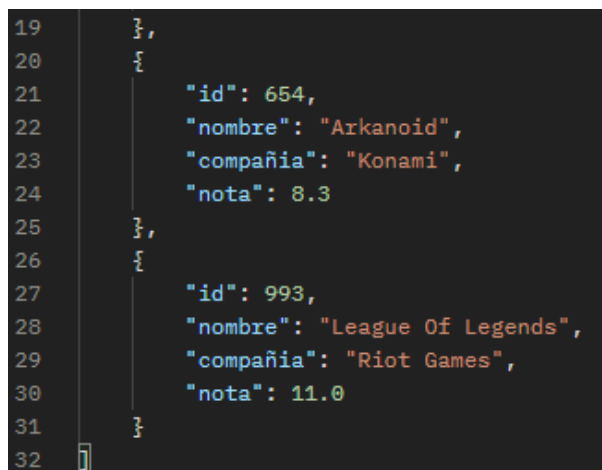

En **Body**, igual que al añadir, seleccionamos el formato Json e introducimos los datos que modificaremos, en este ejemplo le cambiamos la compañía y la nota.



Pulsamos en **Send** y obteniendo el código de estado 200 OK

200 OK 21 ms 123 B

Procedemos a mostrar los videojuegos como hicimos las veces anteriores, para comprobar que se han modificado los datos.



Como en las funcionalidades anteriores, en la consola se muestran las trazas de las consultas que vamos haciendo con sus resultados.

```
Modificar => Modificando videojuego con ID : 654
Modificar => Videojuego modificado : Videojuegos [id=654, nombre=Arkanoid, compañía=Konami, nota=8.3]
Listar => Mostrando la lista de videojuegos
```




- Obtener un videojuego por ID

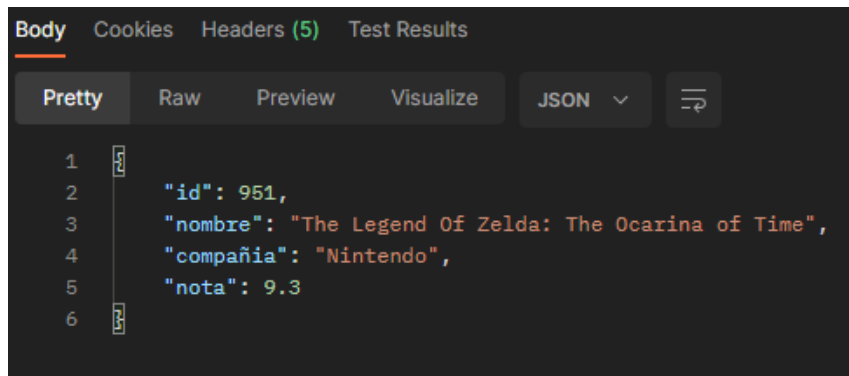
Para realizar esta funcionalidad usamos el verbo **GET** y en el URL request ponemos **localhost:8080/videojuegos/951**, donde 951 es el ID del videojuego que queremos obtener.



Pulsamos el botón **Send** obteniendo el código de estado 200 OK

 200 OK 20 ms 260 B

Y nos devuelve el videojuego que hemos solicitado.



Y de nuevo obtenemos la traza de la consulta en la consola.

```
Obtener => Buscando videojuego con el id:951
```

- Listar todos los videojuegos

En el primer punto ya utilizamos esta funcionalidad para demostrar el funcionamiento de dar de alta un videojuego, pero repetimos la operación y obtenemos nuestra lista de videojuegos con todas las variaciones que hemos ido haciendo durante la demostración de funcionamiento del Servicio REST. Introducimos la URL request **localhost:8080/videojuegos/lista**, poniendo el verbo **GET** tal y como vemos en la imagen y pulsamos **Send**.





Obteniendo el siguiente resultado.

```
Body Cookies Headers (5) Test Results
Pretty Raw Preview Visualize JSON
1  [
2    {
3      "id": 657,
4      "nombre": "Tetris",
5      "compañia": "Spectrum Holobyte",
6      "nota": 9.0
7    },
8    {
9      "id": 852,
10     "nombre": "DOOM 64",
11     "compañia": "Bethesda",
12     "nota": 9.8
13   },
14   {
15     "id": 951,
16     "nombre": "The Legend Of Zelda: The Ocarina of Time",
17     "compañia": "Nintendo",
18     "nota": 9.3
19   },
20   {
21     "id": 654,
22     "nombre": "Arkanoid",
23     "compañia": "Konami",
24     "nota": 8.3
25   },
26   {
27     "id": 993,
28     "nombre": "League Of Legends",
29     "compañia": "Riot Games",
30     "nota": 11.0
31   }
32 ]
```

4.2 Funcionalidad a través del Servicio REST Cliente

Vamos a mostrar cómo funciona el Servicio REST Servidor, a través de un Servicio REST Cliente.

Lo primero es arrancar el Servicio REST Servidor en Eclipse ejecutando como Spring Boot App nuestro ServiciosRestApplication situado en el paquete serviciorest, y a continuación arrancamos de la misma manera el Servicio REST Cliente.

36



Ahora procederemos a mostrar todas las funcionalidades requeridas

- Dar de alta un videojuego

Para mostrar esta funcionalidad, elegimos del menú la opción 1 y vamos introduciendo los datos según nos lo va indicando, obteniendo como resultado que el videojuego ha sido dado de alta con el código de estado 201.

```
1
Has elegido la opción: 1
Añade el ID:
112
Añade el nombre
Gradius
Añade la compañía
Konami
Añade la nota
7
Alta Videojuego => Código de respuesta 201 CREATED
```

En la consola del Servicio REST Servidor, nos aparece la traza de la consulta realizada, en este caso, añadir un videojuego.

```
Agregar => Intentando dar de alta el videojuego: Videojuegos [id=112, nombre=Gradius, compañía=Konami, nota=7.0]
Añadido el videojuego => Videojuegos [id=112, nombre=Gradius, compañía=Konami, nota=7.0] a la lista
```

Si intentamos agregarlo de nuevo, puesto que ya está dado de alta nos sale este mensaje de que ya está en la lista en la consola del Servicio REST Servidor.

```
Agregar => Intentando dar de alta el videojuego: Videojuegos [id=112, nombre=Gradius, compañía=Konami, nota=7.0]
Añadir => Videojuego en lista
```

- Dar de baja un videojuego por ID

En este caso, seleccionamos la opción 2 del menú, nos solicita el ID del videojuego que queremos dar de baja, y nos devuelve que el videojuego con ese ID ha sido borrado.

```
----- VIDEOCLUB PIJ -----

1. Dar de alta un videojuego
2. Dar de baja un videojuego por ID
3. Modificar un videojuego por ID
4. Obtener un videojuego por ID
5. Listar todos los videojuegos
6. Salir de la aplicación
2
Has elegido la opción: 2
Escribe el ID del videojuego a borrar:
654
Videojuego con ID 654 ha sido borrado correctamente.
```



En la consola del Servicio REST Servidor, obtenemos la traza de la consulta de borrado del videojuego con ID 654

```
Borrar => Borrando videojuego con id : 654
4
Borrar => Videojuego Videojuegos [id=654, nombre=Arkanoid, compañía=Taito Corporation, nota=6.4]
```

Si intentamos borrar un videojuego que no está en la lista, nos genera un código de estado 404 Not Found

```
2
Has elegido la opción: 2
Escribe el ID del videojuego a borrar:
9632
Borrar Videojuego => El videojuego con ID 9632 no se ha podido borrar
404 NOT_FOUND
```

- Modificar un videojuego por ID

Ahora vamos a elegir la opción 3 para modificar los datos de un videojuego, introducimos el ID del videojuego que queremos modificar y luego los datos a modificar según nos los van pidiendo.

```
----- VIDEOCLUB PIJ -----

1. Dar de alta un videojuego
2. Dar de baja un videojuego por ID
3. Modificar un videojuego por ID
4. Obtener un videojuego por ID
5. Listar todos los videojuegos
6. Salir de la aplicación
3
Has elegido la opción: 3
Escribe el ID del videojuego a modificar:
657
Escribe el nombre a modificar:
Tetris 99
Escribe la compañía a modificar:
Nintendo
Escribe la nota a modificar:
8
El videojuego con ID 657 se ha modificado correctamente
```

En la consola del Servicio REST Servidor, como en las demás ocasiones, nos saca la traza de la consulta realizada.

```
Modificar => Modificando videojuego con ID : 657
Modificar => Videojuego modificado : Videojuegos [id=657, nombre=Tetris 99, compañía=Nintendo, nota=8.0]
```



- Obtener un videojuego por ID

En el menú, ésta es la opción 4, la escribimos e introducimos el ID del videojuego que queremos obtener.

```
----- VIDEOCLUB PIJ -----  
1. Dar de alta un videojuego  
2. Dar de baja un videojuego por ID  
3. Modificar un videojuego por ID  
4. Obtener un videojuego por ID  
5. Listar todos los videojuegos  
6. Salir de la aplicación  
4  
Has elegido la opción: 4  
Escribe el ID del videojuego a buscar:  
852  
Buscar Videojuego => Código de respuesta 200 OK  
Videojuegos [id=852, nombre=DOOM 64, compañía=Bethesda, nota=9.8]
```

En el Servicio REST Servidor nos saca la traza.

```
Obtener => Buscando videojuego con el id:852
```

- Listar todos los videojuegos

La siguiente opción que tenemos es la 5, la cual nos muestra todos los videojuegos que tenemos en la lista después de todas las modificaciones que hemos realizado en la misma durante la muestra de la funcionalidad.

```
----- VIDEOCLUB PIJ -----  
1. Dar de alta un videojuego  
2. Dar de baja un videojuego por ID  
3. Modificar un videojuego por ID  
4. Obtener un videojuego por ID  
5. Listar todos los videojuegos  
6. Salir de la aplicación  
5  
Has elegido la opción: 5  
Videojuegos [id=657, nombre=Tetris 99, compañía=Nintendo, nota=8.0]  
Videojuegos [id=852, nombre=DOOM 64, compañía=Bethesda, nota=9.8]  
Videojuegos [id=951, nombre=The Legend Of Zelda: The Ocarina of Time, compañía=Nintendo, nota=9.3]  
Videojuegos [id=753, nombre=The Last Of Us, compañía=Naughty Dog, nota=8.9]  
Videojuegos [id=112, nombre=Gradius, compañía=Konami, nota=7.0]
```

Y como en todas las ocasiones anteriores, en la consola del Servicio REST Servidor nos sale la traza de la consulta.

```
Listar => Mostrando la lista de videojuegos
```



- Salir

Por último, solo nos queda mostrar la opción salir de la aplicación, la cual cerrará el Servicio REST Cliente.

```
----- VIDEOCLUB PIJ -----
1. Dar de alta un videojuego
2. Dar de baja un videojuego por ID
3. Modificar un videojuego por ID
4. Obtener un videojuego por ID
5. Listar todos los videojuegos
6. Salir de la aplicación
6
Has elegido la opción: 6
2022-11-19 01:49:16.681 INFO 30724 --- [main] o.apache.catalina.core.StandardService : Stopping service [Tomcat]
El Cliente ha cerrado la conexión
```




5 Conclusiones

Este trabajo en conjunto, nos ha parecido muy interesante, puesto que hemos podido aprender y aplicar la funcionalidad del framework Spring en nuestros programas java, entendiendo cómo funcionan los servicios REST y su forma de intercambio de datos mediante protocolo HTTP.

Además, hemos podido repasar todos los conocimientos adquiridos durante el curso anterior en lo referente al lenguaje de Java, lo que nos ha permitido afianzarlos y reforzar aquellos que ya conocíamos.

Asimismo, nos hemos enfrentado a varios retos mientras avanzando en nuestro proyecto, lo que también nos aporta una experiencia de cara a posibles problemas en un futuro al realizar este tipo de acciones. Ha sido interesante que cada integrante del grupo, aportara sus propias pruebas y argumentos de todos y cada uno de los problemas que se iban presentado durante la realización de la activad, con el fin de comparar y poder debatir acerca de cuál sería la forma más adecuada de realizar ciertas acciones.

Algunas de las decisiones más importantes que hemos tomado en conjunto, han sido acerca de la presentación del proyecto, utilizando en todo momento buenas prácticas y que todo quedará de la forma más clara y ordenada. También hemos debatido acerca de cómo implementar ciertos métodos, así como la funcionalidad esperada de los mismos. Aunque algunas decisiones tuvieran más peso que otras, al final ha sido fundamental ponernos de acuerdo, escuchando a todas las partes para sacar adelante un buen proyecto.

Al final esta metodología nos ayuda a crecer, en lo referente al trabajo en equipo, recordándonos lo importante que resulta la puesta en común de todas las ideas, puesto que, esta estrategia de trabajo, enriquece a todos los integrantes del equipo y genera que el proyecto a realizar sea mucho más dinámico e interactivo.

