

# Unidad 3 REST API con SPRING 5

## Indice

1. REST.....	1
Características de REST.....	1
2. El protocolo HTTP.....	2
2.1 Métodos.....	2
2.2 Códigos de respuesta.....	3
3. Instalación de JRE y del JDK OpenJDK 8.....	4
4. Administración de Java.....	4
5. Uso de lombok.....	4
6. Ejecución de un proyecto en SPRING.....	5
7. Uso de H2.....	5
7.2 Carga de datos con H2.....	5
8. Creación API REST con Spring Boot version1.....	8
8.1 Crear API_REST_Version1.....	8
8.2 Prueba.....	12
8.3 Crear API_REST_Version2.....	13
8.4 Crear API_REST_Version3.....	13

## 1. REST

REST es cualquier interfaz entre sistemas que use HTTP para obtener datos o generar operaciones sobre esos datos en todos los formatos posibles, como XML y JSON.

Twitter, YouTube, los sistemas de identificación con Facebook... hay cientos de empresas que generan negocio gracias a REST y las APIs REST. Sin ellas, todo el crecimiento en horizontal sería prácticamente imposible. Esto es así porque REST es el estándar más lógico, eficiente y habitual en la creación de APIs para servicios de Internet.

### Características de REST

- **Protocolo cliente/servidor sin estado:** cada petición HTTP contiene toda la información necesaria para ejecutarla, lo que permite que ni cliente ni servidor necesiten recordar ningún estado previo para satisfacerla. Aunque esto es así, algunas aplicaciones HTTP incorporan memoria caché.
- Las operaciones más importantes relacionadas con los datos en cualquier sistema REST y la especificación HTTP son cuatro: **POST** (crear), **GET** (leer y consultar), **PUT** (editar) y **DELETE** (eliminar).
- **Los objetos en REST siempre se manipulan a partir de la URI.** Es la URI y ningún otro elemento el identificador único de cada recurso de ese sistema REST. La URI nos facilita acceder a la información para su modificación o borrado, o, por ejemplo, para compartir su ubicación exacta con terceros.

- **Interfaz uniforme:** para la transferencia de datos en un sistema REST, este aplica acciones concretas (POST, GET, PUT y DELETE) sobre los recursos, siempre y cuando estén identificados con una URI. Esto facilita la existencia de una interfaz uniforme que sistematiza el proceso con la información.
- **Sistema de capas:** arquitectura jerárquica entre los componentes. Cada una de estas capas lleva a cabo una funcionalidad dentro del sistema REST.
- **Uso de hipermedios:** hipermedia es un término acuñado por [Ted Nelson](#) en 1965 y que es una extensión del concepto de hipertexto. Ese concepto llevado al desarrollo de páginas web es lo que permite que el usuario puede navegar por el conjunto de objetos a través de enlaces HTML. En el caso de una API REST, el concepto de hipermedia explica la capacidad de una interfaz de desarrollo de aplicaciones de proporcionar al cliente y al usuario los enlaces adecuados para ejecutar acciones concretas sobre los datos.

REST está orientado al concepto de recurso

1. Cada recurso debe ser accesible a través de una URI
2. El servidor puede ofrecer diferentes representaciones de un mismo recurso (por ejemplo en XML, JSON o HTML).

Ventajas del uso de REST

- Separación cliente - servidor
- Visibilidad, fiabilidad y escalabilidad
- Heterogeneidad
- Variedad de formatos: JSON, XML, ...
- En general, es más rápido y utiliza menos ancho de banda.

## 2. El protocolo HTTP

Petición / respuesta

- El cliente (agente de usuario-user agent) realiza una petición enviando un mensaje al servidor.
- El servidor recibe la petición, y envía un mensaje de respuesta al cliente.

Formato del mensaje

- Válido tanto para request como response.
- Solo cambia la línea inicial.
- Estructura

### 2.1 Métodos

- Tipos de peticiones diferentes: Indica el tipo de acción a realizar sobre el recurso indicado.
- Los más conocidos

- **GET:** Solicita un recurso al servidor. Solo deben recuperar datos, y no deben tener otro efecto.

- **POST:** Envía datos para que sean procesados. Los datos se incluyen en el cuerpo de la petición.
- **PUT:** Suele utilizarse en operaciones de actualización completa.
- **DELETE:** Borra el recurso especificado.
- **HEAD:** Idéntico a GET, pero la respuesta no devuelve el cuerpo. Útil para obtener metadatos
- **OPTIONS:** Devuelve la lista de métodos HTTP que soporta un recurso.
- **PATCH:** Actualización parcial de un recurso

## 2.2 Códigos de respuesta

- El servidor debe incluirlos en la primera línea.
- Indica qué ha pasado con la petición.
- Cada código tiene un significado concreto.
- Código numérico de 3 cifras

Códigos de respuesta más usados

### -2XX. Respuestas correctas

- 200 OK. Respuesta estándar de peticiones correctas.
- 201 Created. Se ha creado un recurso.
- 204 No Content. Petición con éxito que no devuelve contenido en la respuesta.

### -4XX. Errores de cliente

- 400 Bad request. Solicitud con sintaxis errónea. El servidor no procesará la solicitud porque no puede, o no debe, debido a un error del cliente.
- 401 Unauthorized. La autenticación ha fallado.
- 403 Forbidden. Solicitud legal, pero no se tienen privilegios para realizarla.
- 404 Not Found. Recurso no encontrado.
- 405 Method not allowed. La URI es correcta, pero no soporta el verbo utilizado.
- 415 Unsupported Media Type. La petición tiene un formato que no entiende el servidor y por eso no se procesa.
- 429 Too many requests. Indica que el usuario ha enviado demasiadas solicitudes en un período de tiempo determinado.

### 5XX. Errores de servidor

- 500 Internal Server Error. El servidor tiene un error ajeno a la naturaleza del servidor web.
- 502 Bad Gateway. El servidor actúa de proxy o gateway con otro servidor, y ha recibido una respuesta inválida del otro servidor.
- 503 Service unavailable. El servidor no puede responder a la petición por estar congestionado o en modo mantenimiento.

### 3. Instalación de JRE y del JDK OpenJDK 8

En ciertos entornos se exige la presencia del JRE de Java 8, ya que es un estándar muy sólido al que se apegan aun a día de hoy muchos desarrollos y aplicaciones, por lo que será necesario instalar Java OpenJDK 8 en Ubuntu 20.04 para dar soporte a estas aplicaciones.

En el caso del entorno de ejecución o JRE de OpenJDK 8 para Ubuntu 20.04 instalamos el paquete `openjdk-8-jre`:

```
~$ sudo apt install -y openjdk-8-jre
```

Y para el kit de desarrollo JDK de OpenJDK 8 seleccionaríamos el paquete `openjdk-8-jdk`:

```
~$ sudo apt install -y openjdk-8-jdk
```

### 4. Administración de Java

Puede tener varias instalaciones de Java en un servidor. Puede configurar la versión predeterminada que se utilizará en la línea de comandos con el comando `update-alternatives`.

```
~$ sudo update-alternatives --config java
```

```
azul@azul:~$ sudo update-alternatives --config java
```

Existen 2 opciones para la alternativa java (que provee `/usr/bin/java`).

Selección Ruta Prioridad Estado

```
-----
* 0 /usr/lib/jvm/java-11-openjdk-amd64/bin/java 1111 modo automático
```

```
1 /usr/lib/jvm/java-11-openjdk-amd64/bin/java 1111 modo manual
```

```
2 /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java 1081 modo manual
```

Pulse <Intro> para mantener el valor por omisión [\*] o pulse un número de selección: 2

`update-alternatives`: utilizando `/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java` para proveer `/usr/bin/java` (java) en modo manual

```
azul@azul:~$ java -version
```

```
openjdk version "1.8.0_265"
```

```
OpenJDK Runtime Environment (build 1.8.0_265-8u265-b01-0ubuntu2~20.04-b01)
```

```
OpenJDK 64-Bit Server VM (build 25.265-b01, mixed mode)
```

```
/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java
```

Puede hacer esto para otros comandos de Java, como el compilador (`javac`):

```
sudo update-alternatives --config javac
```

### 5. Uso de lombok

1. Descargamos la librería y la copiamos en el Escritorio.
2. Vamos a instalar el complemento de lombok en Spring.

Desde un terminal:

```
azul@azul:~$ cd Escritorio/
```

```
azul@azul:~/Escritorio$ java -jar lombok.jar
```

localizamos el IDE (en nuestro caso spring suite)

3. abrimos Spring Suite y new project-Spring Starter Project. Y localizamos lombok.
4. Ya podemos hacer uso de las anotaciones de lombok para ahorrar código.

## 6. Ejecución de un proyecto en SPRING

Desde la Suite: Run as - SPRING Boot App

Desde consola: tenemos que estar en el directorio raíz de la aplicación (recordamos que la aplicación tiene que estar parada en la Suite).

```
azul@azul:~/DATOS/TRABAJO/SPRING5/workspace22/ApiRest_version01$ mvn clean
azul@azul:~/DATOS/TRABAJO/SPRING5/workspace22/ApiRest_version01$ mvn install
azul@azul:~/DATOS/TRABAJO/SPRING5/workspace22/ApiRest_version01$ mvn spring-boot:run
```

## 7. Uso de H2

<https://howtodoinjava.com/spring-boot2/h2-database-example/>

```
CREATE MEMORY TABLE "PUBLIC"."PRODUCTO" (
    "ID" BIGINT NOT NULL,
    "NOMBRE" VARCHAR(255),
    "PRECIO" FLOAT NOT NULL
)
```

### 7.2 Carga de datos con H2

**H2 puede cargar los datos automáticamente desde un Script en el momento que la aplicación se levanta.** Para que funcione solo debe de ser colocado en los **resources** con el nombre **data.sql**

**H2 también nos brinda una consola** con la cual podemos visualizar la información como lo haríamos con cualquier otro cliente de base de datos.

Para acceder a esta consola escribimos en el navegador la siguiente dirección <http://localhost:8080/h2/>.

Viene de application.properties

```
# Custom H2 Console URL
spring.h2.console.path=/h2
```



Español ▼ Preferencias Tools Ayuda

Registrar

Configuraciones guardadas:

Generic H2 (Embedded) ▼

Nombre de la configuración:

Generic H2 (Embedded)

Guardar

Eliminar

---

Controlador:

org.h2.Driver

URL JDBC:

jdbc:h2:mem:testdb

Nombre de usuario:

usuario

Contraseña:

.....

Conectar

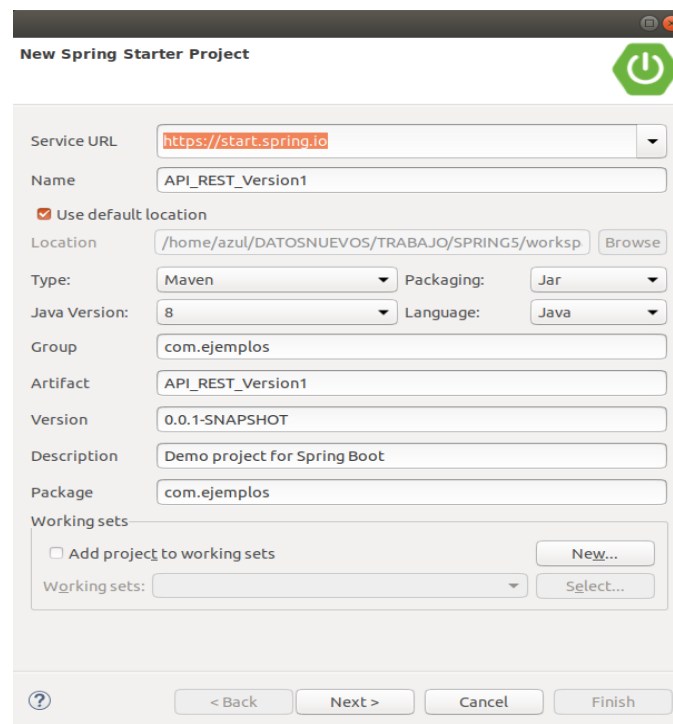
Probar la conexión

## 8. Creación API REST con Spring Boot version1

### 8.1 Crear API\_REST\_Version1

Vamos a crear un servicio que acceda a una tabla de productos que se encuentra en una base de datos H2.

- Creamos un proyecto nuevo: File-New- Spring Starter Project



- añadir dependencias (Lombok, Spring Web, H2Database, Spring Data JPA)
- SpringBootApplication es la clase principal de la aplicación. Sobre la anotación hacer ctrl+clic

- Dentro de la carpeta src/main/resources tenemos :
  - application.properties que permite aplicar configuraciones varias.  
Server.port=8090      cuidado con los espacios en blanco
  - static: recursos estáticos de la aplicación imágenes, JS, hojas de estilos.
  - templates: plantillas vistas de los controladores
- Dentro de la carpeta src/main/java tenemos : el código fuente y la carpeta base es en nuestro caso: com.ejemplos ( de aquí parten el resto de carpetas o paquetes). Recordemos que todas nuestras clases de Spring tiene que estar dentro de la raíz donde se encuentra esta clase principal.
- Carpeta target: al publicar aquí se crea el empaquetado.



- Añadir datos de BD a application.properties (src/main/resources)

```
# Enabling H2 Console
spring.h2.console.enabled=true

# Custom H2 Console URL
spring.h2.console.path=/h2

spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=usuario
spring.datasource.password=usuario
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
# temporary data storage
# spring.datasource.url = jdbc:h2:file:/data/sample
```

- Crear en src/main/resources data.sql
- para acceder a la consola de H2 se escribe en el navegador la url http://localhost:8080/h2
- Crear el modelo y el EJB de Producto (facade de acceso a datos)

package com.ejemplos.modelo;

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data @NoArgsConstructor @AllArgsConstructor
@Entity
public class Producto {
    @Id @GeneratedValue
    private Long id;

    private String nombre;

    private float precio;
}
```

```
public interface ProductoRepositorio extends JpaRepository<Producto, Long> {
}
```

Por si tenemos problemas al crear el proyecto:

- copiar proyecto ApiRest\_version01\_copia (es el proyecto vacío, sólo contiene el fichero applications.properties).
- Crea Producto.java y ProductoRepositorio.java (puedes ejecutar el proyecto correctamente en Boot Dashboard- start)
- Cerramos el proyecto y añadimos data.sql a la carpeta resources (Volvemos a ejecutar para probar y vuelve a ejecutar correctamente).

Al crear el modelo, Spring conecta con JPA y crea la tabla producto correspondiente. Luego con data.sql sólo se crean las filas de la misma.

- Crear el controlador: en la carpeta base com.ejemplos.controller

- En una aplicación web con Spring usamos la anotación `@Controller` : componente de Spring. Básicamente es un objeto del tipo controlador que va a ser manejado de forma automática dentro del contenedor de Spring. En nuestra API REST hacemos uso de `@RestController` controlador de tipo REST manejado por el contenedor de Spring.
- Un controlador va a tener métodos de acción que van a manejar una petición HTTP del usuario. Por ejemplo para mostrar un formulario, para cargar datos, un listado, consultas, guardar, insertar , eliminar.
- Cada método Handler o de acción representa una página web que hace algo y que trabaja con los datos de nuestra aplicación y estos datos se van a mostrar en una vista dentro de la respuesta o se envían normalmente en formato JSON a otra aplicación.
- El controlador se encarga de manejar las peticiones del usuario, mostrar la página, la respuesta con los datos que el usuario ha solicitado.
- Un controlador puede tener varios métodos y cada método va a manejar una página o una petición HTTP distinta.
- Lo que haremos es mapear o relacionar un método a una ruta URL, de tal forma que cuando un cliente (el usuario) acceda a dicha URL, automáticamente se invoque este método y devuelva la información solicitada.

Entonces para mapear podemos usar las siguientes notaciones (todas son equivalentes):

- ◆ `@RequestMapping` (la importamos con control para espacio).

**`@RequestMapping(value="/index")`**

Vamos a indicar en el value o en el path. ( value es un alias del path).

value es un String siempre comienza con / y la ruta ( nombre también alfanumérico simple sin caracteres especiales, sin espacio, sin acentos, sin ñs). Por ejemplo

“/index” entonces cada vez que escribamos en el navegador local

<http://localhost:8080/index> se ejecuta este método.

`RequestMapping` por defecto es del tipo request o el método de petición GET.

Otra forma:

**`@RequestMapping(value="/index",method=RequestMethod.GET)`**

si se omite es tipo GET , también podemos especificar POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE.

- ◆ `@GetMapping` (alias).

**`@GetMapping(value="/index")`**

Por cada tipo de petición tenemos su correspondiente mapping. Puedo omitir el value sino hay más parámetros @GetMapping("/index")

Un método puede estar mapeado a más de una ruta o URL.

**@GetMapping(value={"/index", "/", "/home"})**

## ProductoController.java

```
@RestController
@RequiredArgsConstructor //lombok crea el constructor
public class ProductoController {

    private final ProductoRepositorio productoRepositorio; // se declara como final pq no se va a modificar este repositorio
    //se inyecta solo al crear el bean controlador
    //Dentro de la carpeta resources está data.sql
    /**
     * Obtenemos todos los productos
     * @return
     */
    @GetMapping("/producto")
    public List<Producto> obtenerTodos() {
        return productoRepositorio.findAll();
    }

    /**
     * Obtenemos un producto en base a su ID
     * @param id
     * @return Null si no encuentra el producto
     */
    //@PathVariable : permite inyectar un fragmento de la URL en una variable, es decir, pasa el valor
    //del id de la URL al método como parámetro donde esté @PathVariable
    @GetMapping("/producto/{id}")
    public Producto obtenerUno(@PathVariable Long id) {
        return productoRepositorio.findById(id).orElse(null); //devuelve un Optional
        //no manejamos todavía los errores y devolvemos un null
    }

    /**
     * Insertamos un nuevo producto
     * @param nuevo
     * @return producto insertado
     */
    //@RequestBody Permite inyectar el cuerpo de la petición en un objeto
    @PostMapping("/producto")
    public Producto nuevoProducto(@RequestBody Producto nuevo) {
        return productoRepositorio.save(nuevo);
    }

    /**
     * @param editar
     * @param id
     * @return
     */
    @PutMapping("/producto/{id}")
    public Producto editarProducto(@RequestBody Producto editar, @PathVariable Long id) {
        if (productoRepositorio.existsById(id)) {
            editar.setId(id);
            return productoRepositorio.save(editar);
        } else {
            return null;
        }
    }
}
```

```

/**
 * Borra un producto del catálogo en base a su id
 * @param id
 * @return
 */
@DeleteMapping("/producto/{id}")
public Producto borrarProducto(@PathVariable Long id) {
    if (productoRepositorio.existsById(id)) {
        Producto result = productoRepositorio.findById(id).get();
        productoRepositorio.deleteById(id);
        return result;
    } else
        return null;
}
}

```

Anotaciones que usamos:

#### @RequestBody

Permite inyectar el cuerpo de la petición en un objeto

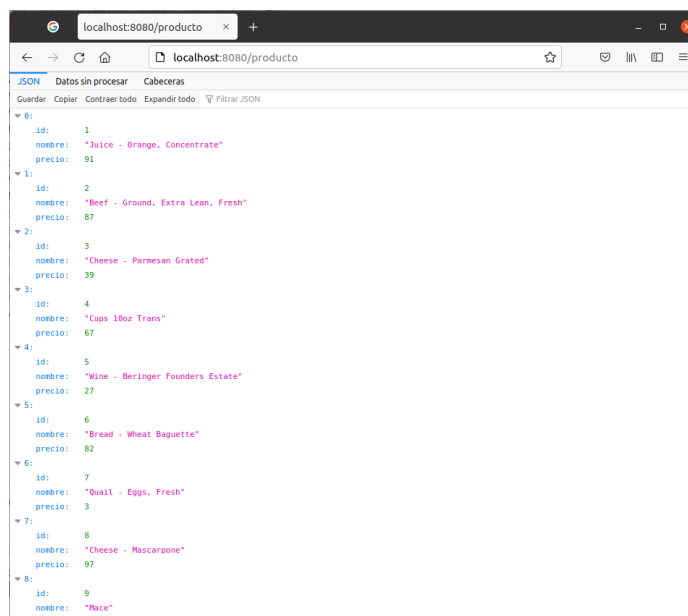
#### @PathVariable

Nos permite inyectar un fragmento de la URL en una variable

## 8.2 Prueba

Levantamos el servicio y accedemos al navegador a la siguiente URL

<http://localhost:8080/producto> y deberá mostrarnos algo similar a esto.



## 8.3 Crear API\_REST\_Version2

Se añade al controlador de productos la gestión de los mensajes de error del servidor.

### Clase `ResponseEntity <T>`

- [java.lang.Object](#)
  - [org.springframework.http.ResponseEntity <T>](#)
    - [org.springframework.http.ResponseEntity <T>](#)

Sirve para agregar un código de estado [HttpStatus](#) a la respuesta del servidor.

## 8.4 Crear API\_REST\_Version3

```
#sirve para desactivar los mensajes de traza del servidor en properties
server.error.include-stacktrace=never
```

Hacemos uso de objetos de tipo DTO (Diferentes vistas de un mismo modelo).

Un **objeto de transferencia de datos** ( *data transfer object*, abreviado **DTO**) es un objeto que transporta datos entre procesos. La motivación de su uso tiene relación con el hecho que la comunicación entre procesos se realiza generalmente mediante interfaces remotas (por ejemplo, servicios web), donde cada llamada es una operación costosa. Como la mayor parte del costo de cada llamada está relacionado con la comunicación de ida y vuelta entre el cliente y servidor, una forma de reducir el número de llamadas es usando un objeto (el DTO) que agrega los datos que habrían sido transferidos por cada llamada, pero que son entregados en una sola llamada.

La diferencia entre un objeto de transferencia de datos y un [objeto de negocio](#) (*business object*) o un [objeto de acceso a datos](#) (*data access object*, DAO) es que un DTO no tiene más comportamiento que almacenar y entregar sus propios datos ( setters y getters).

Los DTO son objetos simples que no deben contener lógica de negocio que requiera pruebas generales.

- Añadimos nueva dependencia al pom.xml para poder usar modelmapper

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.3.5</version>
</dependency>
```

- ¿Para qué sirve modelmapper? Las aplicaciones requieren objetos que aunque son muy similares en cuanto a estructura/necesidades tienen entre ellos leves modificaciones. El mapeo tiene la finalidad de facilitar la conversión, el traspaso de unos datos de un objeto entity a otra entity.

El objetivo de ModeMapper es hacernos el mapeo entre objetos más sencillo, para ello, tenemos que definir el modelo (en este caso, el objeto inicial) que se mapeará y el modelo final (en este caso, el objeto final) sobre el que volcará los valores mapeados.

### **BENEFICIOS DE MODELMAPER**

Algunos de los beneficios de utilizar ModelMapper son:

- **Basado en convenciones e inteligente:** no necesita un mapeo manual (a excepción de los campos cuyos nombres no coincidan tanto en nombre como en tipo de dato).
- **API simple y segura:** al no picar nosotros el código manualmente y hacerse automáticamente por detrás evitamos la posibilidad de que se produzcan posibles fallos.
- **Extensible:** admite diferentes formatos de modelos de datos. JavaBeans, JSON, BBDD...

URL Web Oficial: <http://modelmapper.org/>

- Se crea una clase Configuracion.java en el paquete `com.ejemplos.configuracion` y dentro creamos un bean de tipo ModelMapper para toda nuestra aplicación.

```
@Configuration
public class Configuracion {

    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }
}
```

- Vamos a relacionar los productos con las categorías.

1. Ampliamos data.sql.
2. Creamos en el modelo: Categoría y su facade del repositorio.
3. Cuando realicemos una petición de consulta de los productos: aparecen los datos de los productos con los datos de la categoría completa a la que pertenecen.

```
{
    "id": 4,
    "nombre": "Juice - Orange, Concentrate",
    "precio": 91.0,
    "categoria": {
        "id": 2,
        "nombre": "Bebida"
    }
}
```

```

    },
    {
        "id": 5,
        "nombre": "Beef - Ground, Extra Lean, Fresh",
        "precio": 87.0,
        "categoria": {
            "id": 1,
            "nombre": "Comida"
        }
    }
}

```

4. Queremos sin embargo mostrar los datos de producto pero que incluya sólo el nombre de la categoria. Podemos crear otra vista del modelo Producto llamada ProductoDTO

```

public class ProductoDTO {
    private long id;
    private String nombre;
    private String categoriaNombre;
}

```

5. Necesitamos convertir un objeto de tipo Producto a ProductoDTO cuando obtengamos el listado, para ello se usa

@Component

@RequiredArgsConstructor

```

public class ProductoDTOConverter {
    private final ModelMapper modelMapper;

    public ProductoDTO convertirADto(Producto producto) {
        return modelMapper.map(producto, ProductoDTO.class);
    }
}

```

6. Desde ProductoController se usa la conversión.

## Nota JAVA 8

### stream():

¿Qué es Stream? Es una **secuencia de elementos**. Con este método podremos **transformar una colección de objetos** (arrays, lists,...) **en una sucesión de objetos**.

La interfaz **List** hereda de **Collection** por lo que cualquier clase que la implemente también podrá hacer uso de su método **stream**.

### map():

**map** aplicará una función que llamaremos **F** sobre cada uno de los elementos de la sucesión y devolverá otra sucesión de elementos ya modificados.

¿Y esa función es...?

Cualquiera que necesitemos aplicar, eso sí cumpliendo unas restricciones.

\*El método `trim()` elimina los espacios en blanco en ambos extremos del string. Los espacios en blanco en este contexto, son todos los caracteres sin contenido (espacio, tabulación, etc.) y todos los caracteres de nuevas líneas (LF,CR,etc.).

Por ejemplo, podemos decir que queremos que haga un `*trim()` a cada elemento de la sucesión de tipo `String` de la sucesión del primer elemento. Para ello podemos escribirlo así en Java 8:

```
streamCadenas.map(s->s.trim());
```

En esta opción hacemos uso de una **variable s**, cuyo ámbito es el del método (fuera de ese `map` no existirá y no será necesario declararla previamente).

**Estamos indicando que cada elemento de esa sucesión lo vamos a guardar en una variable s**, del mismo tipo del elemento, y **vamos a aplicarle la función trim()**.

¿Por qué no hemos puesto `String s->s.trim()`? Porque **Java 8 reconoce perfectamente el tipo de la variable s** al conocer el tipo de los elementos de la sucesión (`Stream`). Si hubiésemos utilizado el `Stream` `enterosStream` `s` sería del tipo `Integer` y tampoco haría falta declararlo.

Otra forma de hacer lo mismo sería la siguiente:

```
streamCadenas.map(String :: trim).
```

La diferencia con la anterior es que **nos ahorramos la variable**, esta nomenclatura no es válida siempre. Debemos asegurarnos de que el método no tiene parámetros, si los tuviera no podría usarse de este modo aunque sí en su primera versión

### **filter():**

**filter** recibe una sucesión de elementos y devuelve aquellos que cumplan con el patrón buscado (predicate).

Empezamos mostrando un ejemplo básico.

Tenemos un listado de cadenas en las que guardamos tipos de vehículos y vamos a obtener de ellos los que no sean “motorbike”

```
List vehicles = Arrays.asList("car", "motorbike", "bus");
```

Antes haríamos algo de este tipo:

```
List filteredVehicles = new ArrayList();
for(String vehicle: vehicles){
    if(!"motorbike".equals(vehicle)){
        filteredVehicles.add(vehicle);
    }
}
```

Con Java 8 sería tal que así:

```
List filteredVehicles = vehicles.stream()
    .filter(v -> !"motorbike".equals(v))
    .collect(Collectors.toList());
```