

---

# DESARROLLO, PRUEBAS Y DOCUMENTACIÓN DE UNA API REST

---

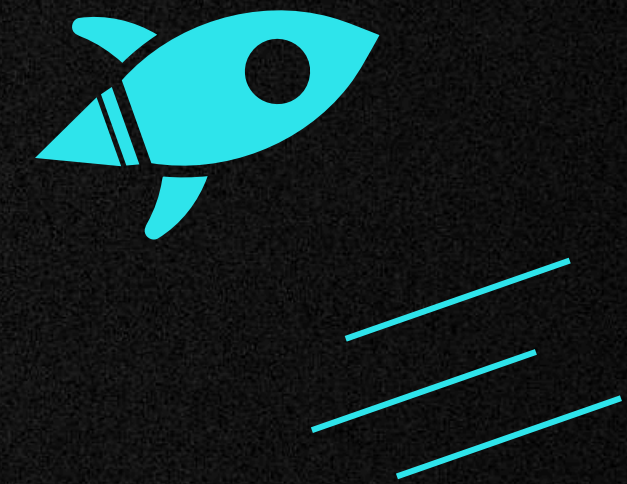
Lidia García Muñoz





# CONTENIDO

- 01 Contexto
- 02 Requisitos para el proyecto
- 03 ¿Qué es una API REST?
- 04 Implementación de un servicio REST
- 05 Postman: probando los web services
- 06 Operaciones CRUD: pruebas de funcionalidad
- 07 Comunicación del lado cliente (frontend) con la API
- 08 SpringBoot OpenAPI: documentando la API
- 09 Conclusiones





# 1.CONTEXTO

En este proyecto, se crea una API REST que permite gestionar productos (agregar, obtener, actualizar y eliminar). El enfoque principal de la presentación es explicar **cómo implementamos el servicio web REST utilizando Spring Boot, cómo se realizan las pruebas del servicio con Postman, y cómo se documenta la API utilizando SpringDoc OpenAPI.**

Además, se incluye una interfaz frontend en HTML y JavaScript para interactuar con la API.

Nombre	Precio	Acciones	
Placa pladur Phonetic	19.5 €	<a href="#">Editar</a>	<a href="#">Eliminar</a>
Placa pladur PPF	39.5 €	<a href="#">Editar</a>	<a href="#">Eliminar</a>
Placa pladur Standar	30 €	<a href="#">Editar</a>	<a href="#">Eliminar</a>
Montante Placo	1.75 €	<a href="#">Editar</a>	<a href="#">Eliminar</a>
Taladro eléctrico	200.5 €	<a href="#">Editar</a>	<a href="#">Eliminar</a>
Destornillador manual	2 €	<a href="#">Editar</a>	<a href="#">Eliminar</a>
Tornillos mm 9,5mm	1.5 €	<a href="#">Editar</a>	<a href="#">Eliminar</a>

Agregar Producto

Nombre  Precio  [Agregar](#)



## 2. REQUISITOS PARA EL PROYECTO

### Backend:

**Java y Spring Boot** es una de las tecnologías más utilizadas para desarrollar aplicaciones web y servicios backend.

Es un **framework** basado en Spring que permite desarrollar aplicaciones en Java de forma rápida y sencilla, eliminando configuraciones complejas.



### Interfaz Gráfica:

**Bootstrap** es un framework de CSS desarrollado por Twitter que facilita la creación de sitios web responsivos y estilizados de manera rápida. Proporciona una colección de clases predefinidas, componentes listos para usar y un sistema de rejilla (grid system) que permite estructurar el diseño de manera flexible y adaptable a distintos dispositivos.



### Base de Datos:

**MySQL** es un sistema de gestión de bases de datos relacional (RDBMS) de código abierto, ampliamente utilizado en aplicaciones web y empresariales.

**JPA (Java Persistence API)** es una especificación de Java que facilita la gestión de bases de datos relacionales a través de mapeo objeto-relacional (ORM). Permite trabajar con bases de datos mediante objetos Java en lugar de escribir consultas SQL manualmente.



### Gestor de Dependencias:

Herramienta de **gestión de proyectos en Java** que facilita la compilación, empaquetado y administración de dependencias. Se basa en un archivo de configuración (**pom.xml**) donde se definen las librerías y configuraciones del proyecto.





# 3. ¿QUÉ ES UNA API REST?

Una API REST (Representational State Transfer) es una **arquitectura para sistemas distribuidos**. **Utiliza el protocolo HTTP para realizar las operaciones CRUD sobre recursos**, los cuales son representados en formato JSON o XML.

Las características clave de una API REST incluyen:

- **Stateless**: Cada solicitud es independiente y contiene toda la información necesaria.
- **Interfaz basada en HTTP**: Usa métodos como GET, POST, PUT y DELETE para interactuar con los recursos.
- **Escalabilidad y simplicidad**: Facilita la integración y comunicación entre aplicaciones distribuidas.



## IMPLEMENTACIÓN:

- Se construye una API en el servidor.
- Se crean endpoints: *get/users*, *POST/login*, etc.
- Conecta con la base de datos (en este caso, con MySQL) y estructura la respuesta.



## 4. IMPLEMENTACIÓN DE UN SERVICIO REST

A continuación, **vamos a ver cómo implementar el servicio web REST** en nuestro código. Explicaremos cómo se estructuran los tres componentes principales:

- **ProductoController**: Contiene las rutas y *endpoints* de la API para las operaciones CRUD.
  - GET /productos: Devuelve todos los productos.
  - POST /productos: Crea un nuevo producto.
  - PUT /productos/{id}: Actualiza un producto existente.
  - DELETE /productos/{id}: Elimina un producto.
- **ProductoService**: Contiene la lógica de negocio y gestiona las operaciones sobre la base de datos a través del repositorio.
- **ProductoRepository**: Interfaz que extiende de JpaRepository para facilitar las operaciones de base de datos.



```

@RestController
@RequestMapping("/productos")
@CrossOrigin(origins = "*")
public class ProductoController {

    @Autowired
    private ProductoService service;

    @Operation(summary = "Obtener lista de productos")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Productos encontrados"),
        @ApiResponse(responseCode = "500", description = "Error en el servidor")
    })
    @GetMapping
    public List<Producto> obtenerProductos() { return service.obtenerProductos(); }

    @Operation(summary = "Agregar un nuevo producto")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "201", description = "Producto creado"),
        @ApiResponse(responseCode = "400", description = "Solicitud inválida")
    })
    @PostMapping
    public Producto agregarProducto(@RequestBody Producto producto) { return service.guardarProducto(producto); }

    @Operation(summary = "Actualizar un producto existente")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Producto actualizado"),
        @ApiResponse(responseCode = "404", description = "Producto no encontrado")
    })
    @PutMapping("/{id}")
    public Producto actualizarProducto(@PathVariable Long id, @RequestBody Producto producto) {
        return service.actualizarProducto(id, producto);
    }

    @Operation(summary = "Eliminar un producto")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "204", description = "Producto eliminado"),
        @ApiResponse(responseCode = "404", description = "Producto no encontrado")
    })
    @DeleteMapping("/{id}")
    public void eliminarProducto(@PathVariable Long id) { service.eliminarProducto(id); }
}

```

**ProductoController.java**

Este archivo define un controlador en *Spring Boot* para **manejar solicitudes HTTP relacionadas con productos**.

Esta API REST usa métodos HTTP estándar, URLs significativas y respuestas en formato JSON.

El controlador tiene varias **rutas HTTP** que interactúan con la lógica del negocio a través del *ProductoService*:

- `@RequestMapping("/productos")` → Define la ruta base para todos los endpoints de este controlador.
- `@CrossOrigin(origins = "*")` → Permite que esta API sea accesible desde cualquier origen (evita problemas de CORS).

**Endpoints** definidos:

- `@GetMapping` → Responde a solicitudes GET en `/productos`.
- `@PostMapping` → Responde a solicitudes POST en `/productos`.
- `@PutMapping("/{id}")` → Responde a solicitudes PUT en `/productos/{id}`.
- `@DeleteMapping("/{id}")` → Responde a solicitudes DELETE en `/productos/{id}`.



Este archivo define el servicio ProductoService, que **se encarga de la lógica de negocio** de la aplicación.

Es una capa intermedia entre el controlador (ProductoController) y el repositorio (ProductoRepository).

Se relaciona con la API REST porque **el controlador usa este servicio para manejar las solicitudes HTTP** (GET, POST, PUT, DELETE) y realizar operaciones en la base de datos.

- *@Autowired* → Spring inyecta automáticamente una instancia de ProductoRepository.
- *ProductoRepository* es el que se comunica con la base de datos.
- Contiene los métodos CRUD necesarios para la realización de operaciones con los productos (obtener, editar, eliminar...).

```
@Service
public class ProductoService {

    @Autowired
    private ProductoRepository repository;

    public List<Producto> obtenerProductos() {
        return repository.findAll();
    }

    public Producto guardarProducto(Producto producto) {
        return repository.save(producto);
    }

    public Producto actualizarProducto(Long id, Producto productoDetalles) {
        Optional<Producto> optionalProducto = repository.findById(id);
        if (optionalProducto.isPresent()) {
            Producto producto = optionalProducto.get();
            producto.setNombre(productoDetalles.getNombre());
            producto.setPrecio(productoDetalles.getPrecio());
            return repository.save(producto);
        } else {
            throw new RuntimeException("Producto no encontrado");
        }
    }

    public void eliminarProducto(Long id) {
        repository.deleteById(id);
    }
}
```

**ProductoService.java**



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

*ProductoRepository.java*

Este archivo **define la capa de acceso a datos en la aplicación**. Es el repositorio (*ProductoRepository*) que se encarga de interactuar con la base de datos.

Se relaciona con la API REST porque **el servicio (*ProductoService*) usa este repositorio para consultar, guardar, actualizar y eliminar productos en la base de datos**.

Aunque no lo ve os en el código, *Spring Boot* detecta automáticamente que esta interfaz es un repositorio porque **extiende de *JpaRepository***, esta le da funcionalidades automáticas sin tener que escribir código manualmente (*findAll()*; *findById(id)*, *save(producto)*, entre otros).



# 5. POSTMAN: PROBANDO LOS WEB SERVICES



Para verificar que la API REST funciona correctamente, se realizan las pruebas utilizando Postman. **Es una herramienta popular para desarrollar, probar y documentar APIs** de manera sencilla e intuitiva.

En las siguientes diapositivas veremos los pasos necesarios para hacer pruebas a nuestro API con Postman.

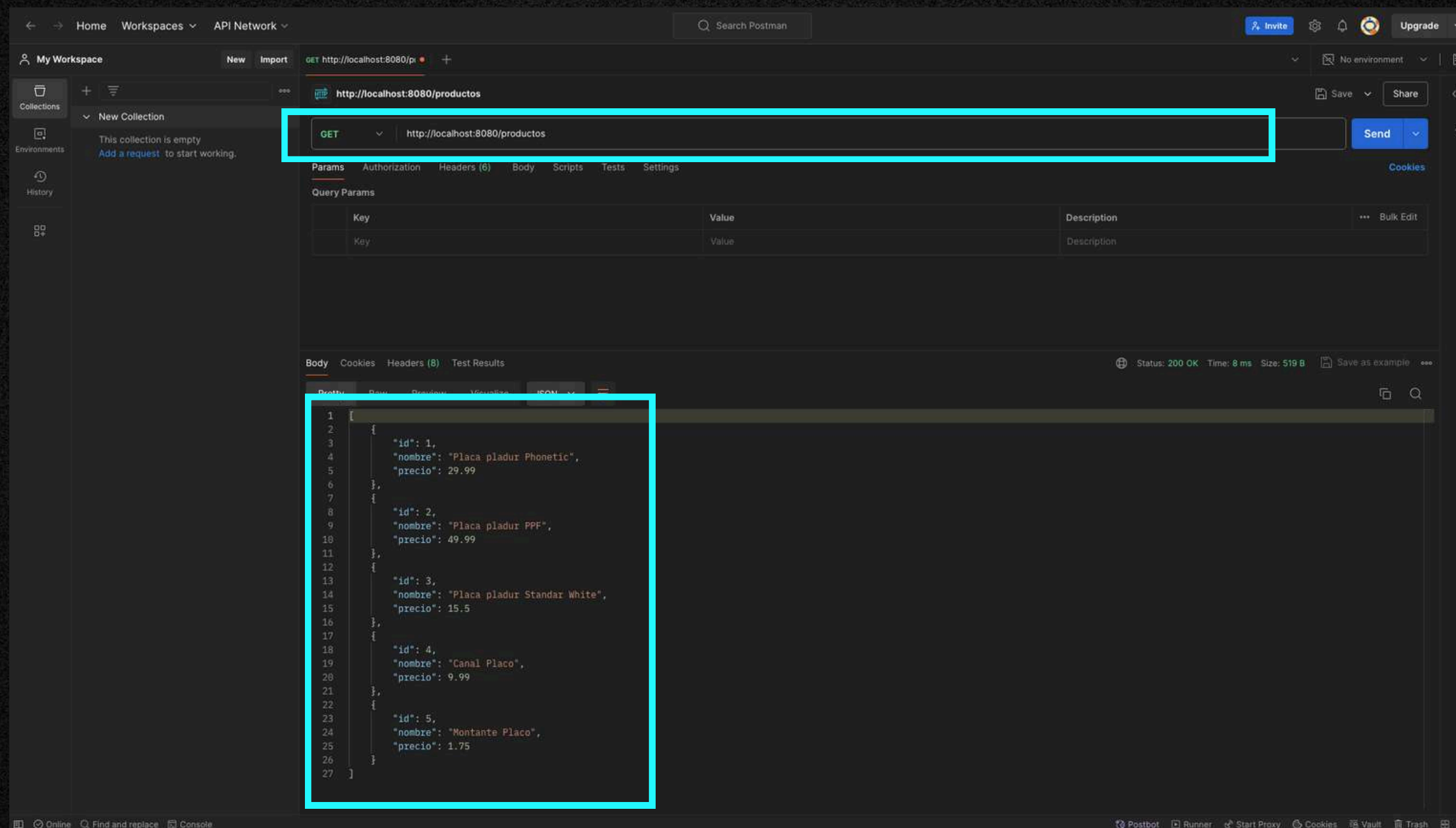
```
[{"id":1,"nombre":"Placa pladur Phonetic","precio":29.99}, {"id":2,"nombre":"Placa pladur PPF","precio":49.99}, {"id":3,"nombre":"Placa pladur Standar White","precio":15.5}, {"id":4,"nombre":"Canal Placo","precio":9.99}, {"id":5,"nombre":"Montante Placo","precio":1.75}]
```

*datos de nuestra API devueltos por el servidor*



## Pruebas de obtener productos:

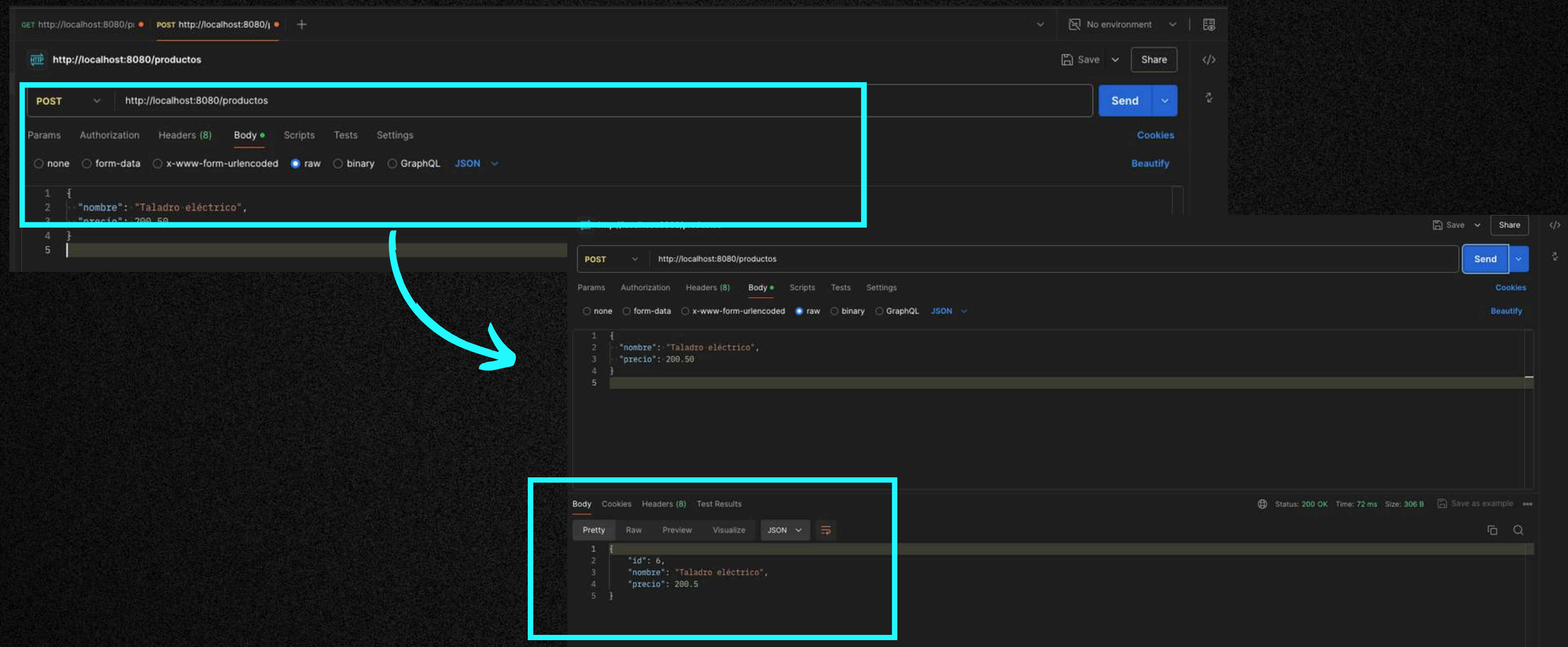
- Método: GET
- URL: http://localhost:8080/productos
- Respuesta esperada: Una lista de productos en formato JSON.





## Pruebas de agregar un producto:

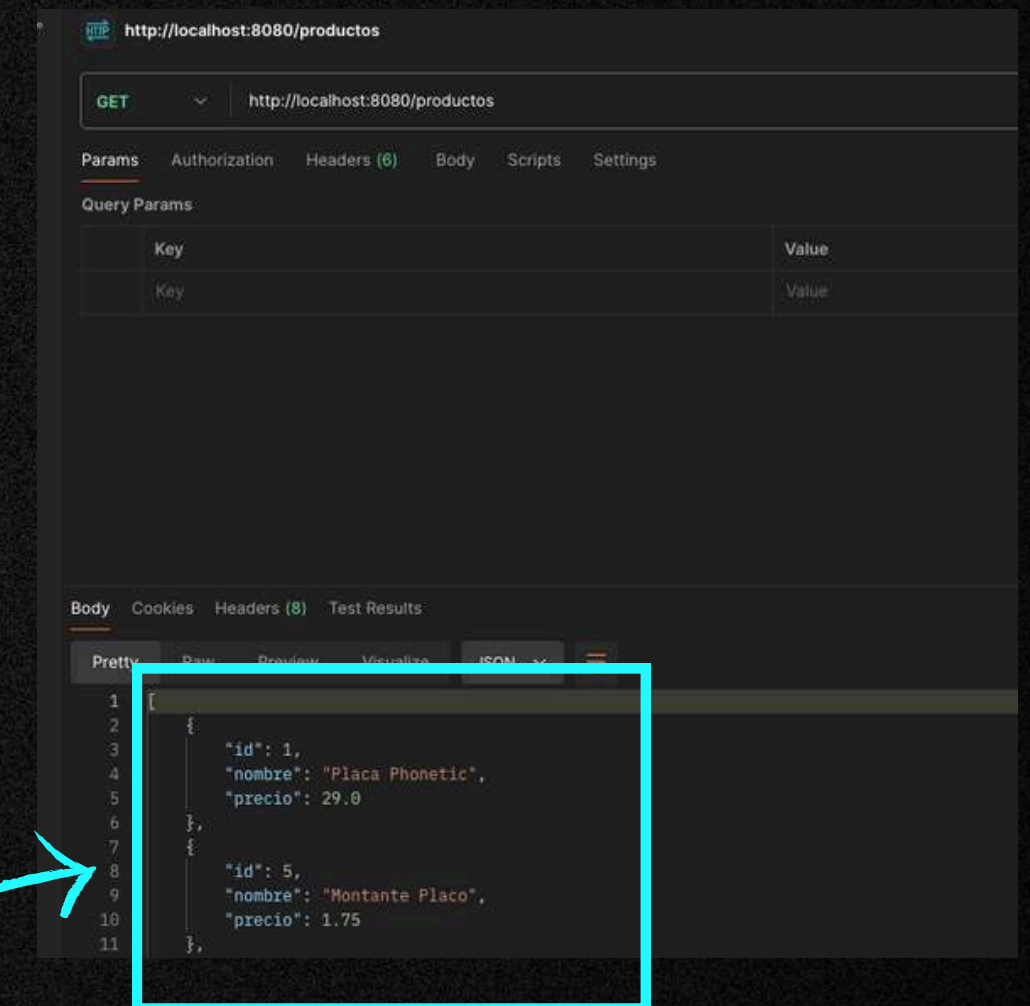
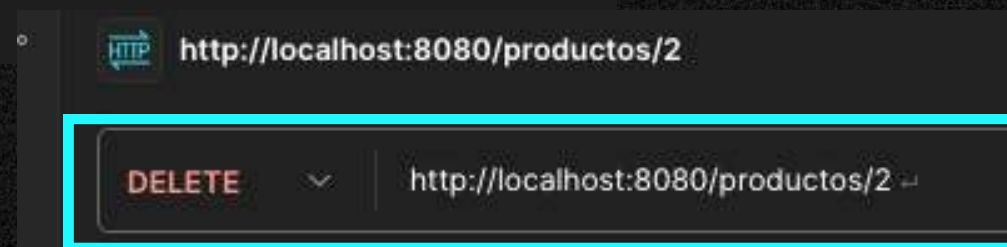
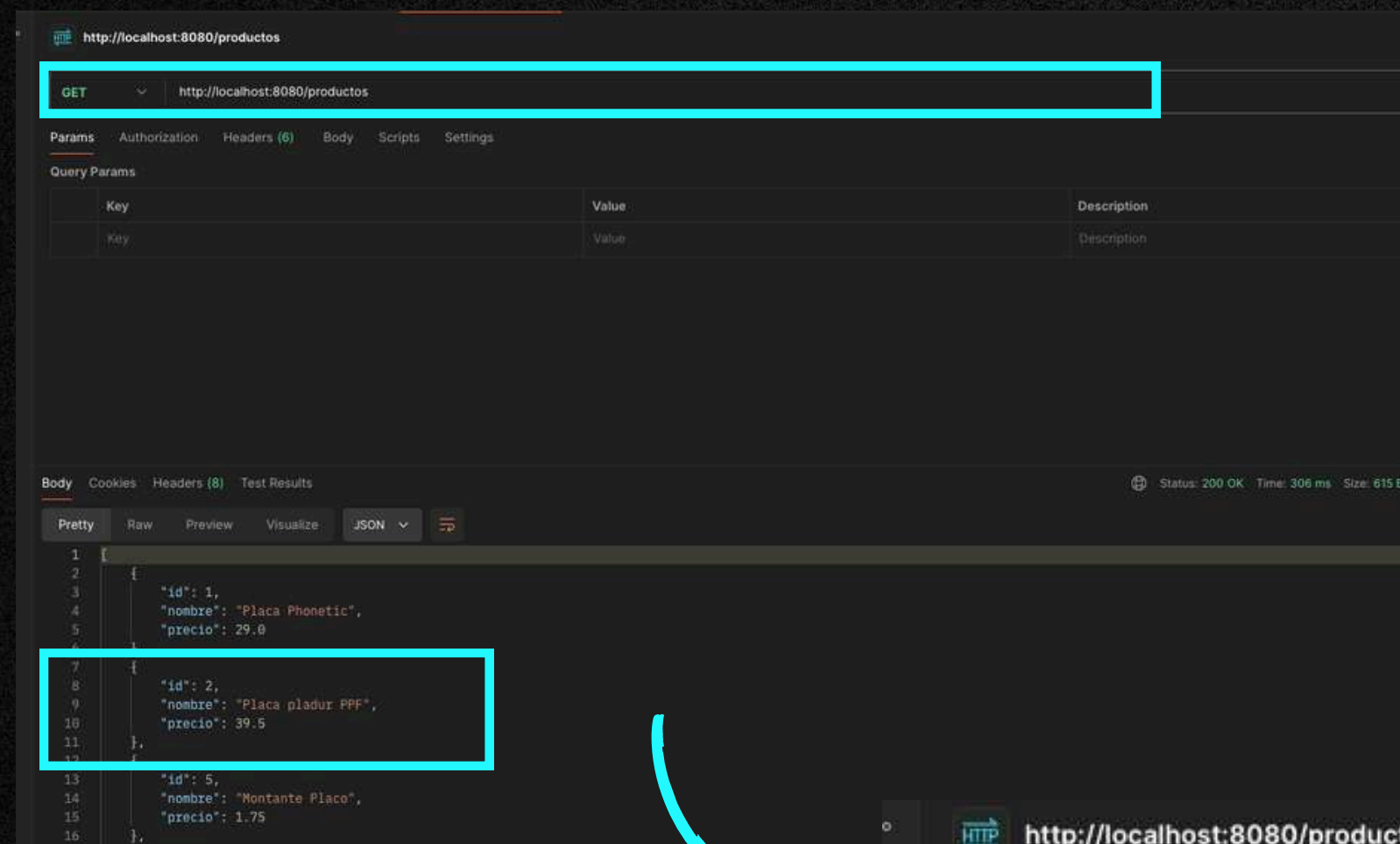
- Método: POST
- URL: http://localhost:8080/productos
- Cuerpo: escribimos en JSON el nombre y precio del producto que deseemos agregar.





## Pruebas de eliminar un producto:

- Método: DELETE
- URL: `http://localhost:8080/productos/{id}` (sustituir id por el número del id del producto que deseemos eliminar)





## 6. OPERACIONES CRUD: PRUEBAS DE FUNCIONALIDAD

**Las operaciones CRUD** (Create, Read, Update y Delete) son la base fundamental de cualquier aplicación que maneje datos. Estas operaciones permiten gestionar la información en una base de datos, proporcionando las acciones esenciales para registrar, consultar, modificar y eliminar registros.

En este caso, trabajamos con la entidad Producto, cuya información se almacena en una base de datos. Para interactuar con la base de datos de manera eficiente, utilizamos *productoRepository.java*, una clase que implementa las operaciones CRUD utilizando JPA.

A continuación, exploraremos cada operación CRUD con su respectiva implementación, incluyendo ejemplos de código y representaciones visuales para ilustrar su funcionamiento.



## **obtenerProductos()** → Obtener todos los productos


- Llama a **repository.findAll()** para obtener todos los productos desde la base de datos.
- Se usa en el *endpoint* GET /productos del controlador.

```
@GetMapping
public List<Producto> obtenerProductos() { return service.obtenerProductos(); }

@Operation(summary = "Agregar un nuevo producto")
@ApiResponses(value = {
    @ApiResponse(responseCode = "201", description = "Producto creado"),
    @ApiResponse(responseCode = "400", description = "Solicitud inválida")
})
```

Se usan códigos de estado HTTP para informar sobre el resultado:

- **201 Created**: Indica que el producto se creó exitosamente.
- **400 Bad Request**: Significa que hubo un error en la solicitud, como datos inválidos o incompletos.



The screenshot displays a web interface with a title 'Lista de Productos'. Below the title is a table with three columns: 'Nombre', 'Precio', and 'Acciones'. The table lists seven products: 'Placa pladur Phonetic' (19.5 €), 'Placa pladur PPF' (39.5 €), 'Placa pladur Standar' (30 €), 'Montante Placo' (1.75 €), 'Taladro eléctrico' (200.5 €), 'Destornillador manual' (2 €), and 'Tornillos mm 9,5mm' (1.5 €). Each product row has two buttons in the 'Acciones' column: a yellow 'Editar' button and a red 'Eliminar' button. Below the table is a section titled 'Agregar Producto' containing two input fields labeled 'Nombre' and 'Precio', and a green 'Agregar' button.

Nombre	Precio	Acciones
Placa pladur Phonetic	19.5 €	<button>Editar</button> <button>Eliminar</button>
Placa pladur PPF	39.5 €	<button>Editar</button> <button>Eliminar</button>
Placa pladur Standar	30 €	<button>Editar</button> <button>Eliminar</button>
Montante Placo	1.75 €	<button>Editar</button> <button>Eliminar</button>
Taladro eléctrico	200.5 €	<button>Editar</button> <button>Eliminar</button>
Destornillador manual	2 €	<button>Editar</button> <button>Eliminar</button>
Tornillos mm 9,5mm	1.5 €	<button>Editar</button> <button>Eliminar</button>

Agregar Producto

Nombre  Precio  Agregar

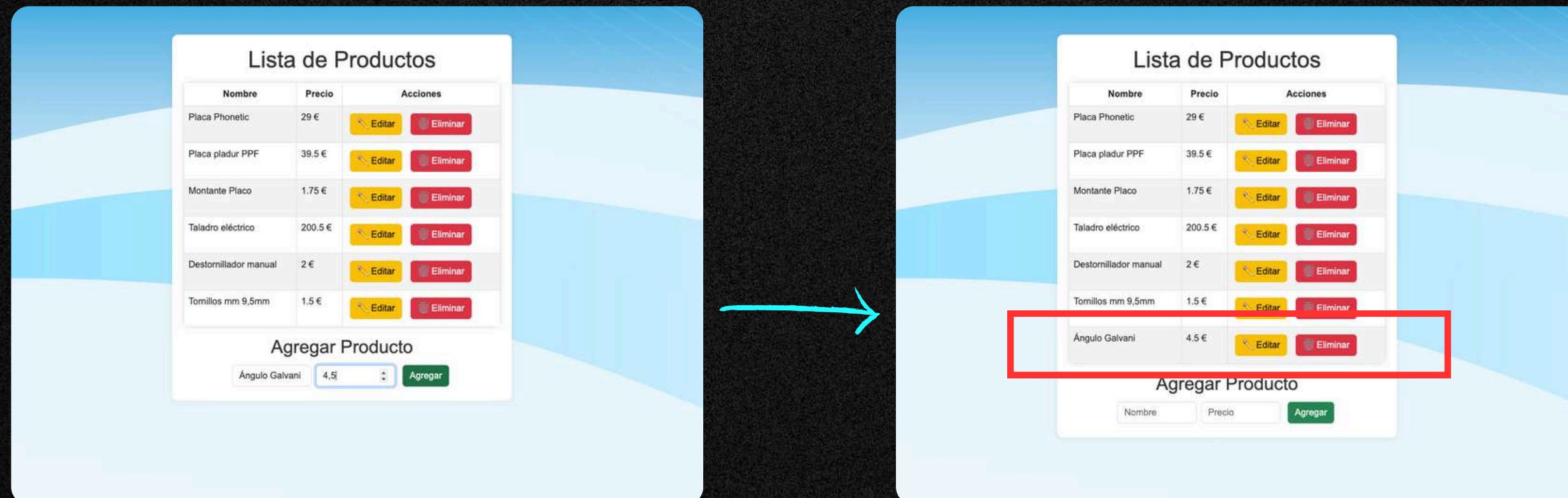


## guardarProducto() → Guardar un nuevo producto

- Llama a **repository.save(producto)**, que inserta un nuevo producto en la base de datos.
- Se usa en el *endpoint* POST /productos del controlador.

```
@PostMapping
public Producto agregarProducto(@RequestBody Producto producto) { return service.guardarProducto(producto); }

@Operation(summary = "Actualizar un producto existente")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200", description = "Producto actualizado"),
    @ApiResponse(responseCode = "404", description = "Producto no encontrado")
})
```

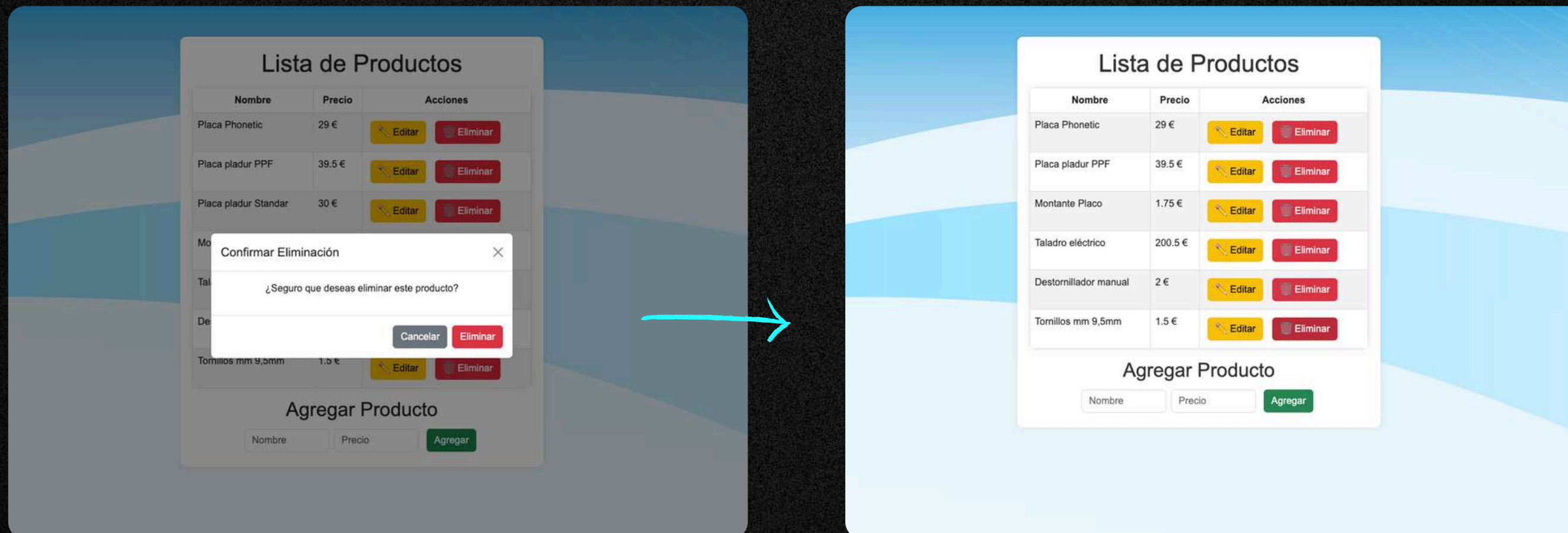




## **eliminarProducto()** → **Eliminar un producto por ID**

- Llama a **repository.deleteByld(id)**, que borra el producto de la base de datos.
- Se usa en el *endpoint* DELETE /productos/{id} del controlador.

```
@Operation(summary = "Eliminar un producto")
@ApiResponses(value = {
    @ApiResponse(responseCode = "204", description = "Producto eliminado"),
    @ApiResponse(responseCode = "404", description = "Producto no encontrado")
})
@DeleteMapping("/{id}")
public void eliminarProducto(@PathVariable Long id) { service.eliminarProducto(id); }
```

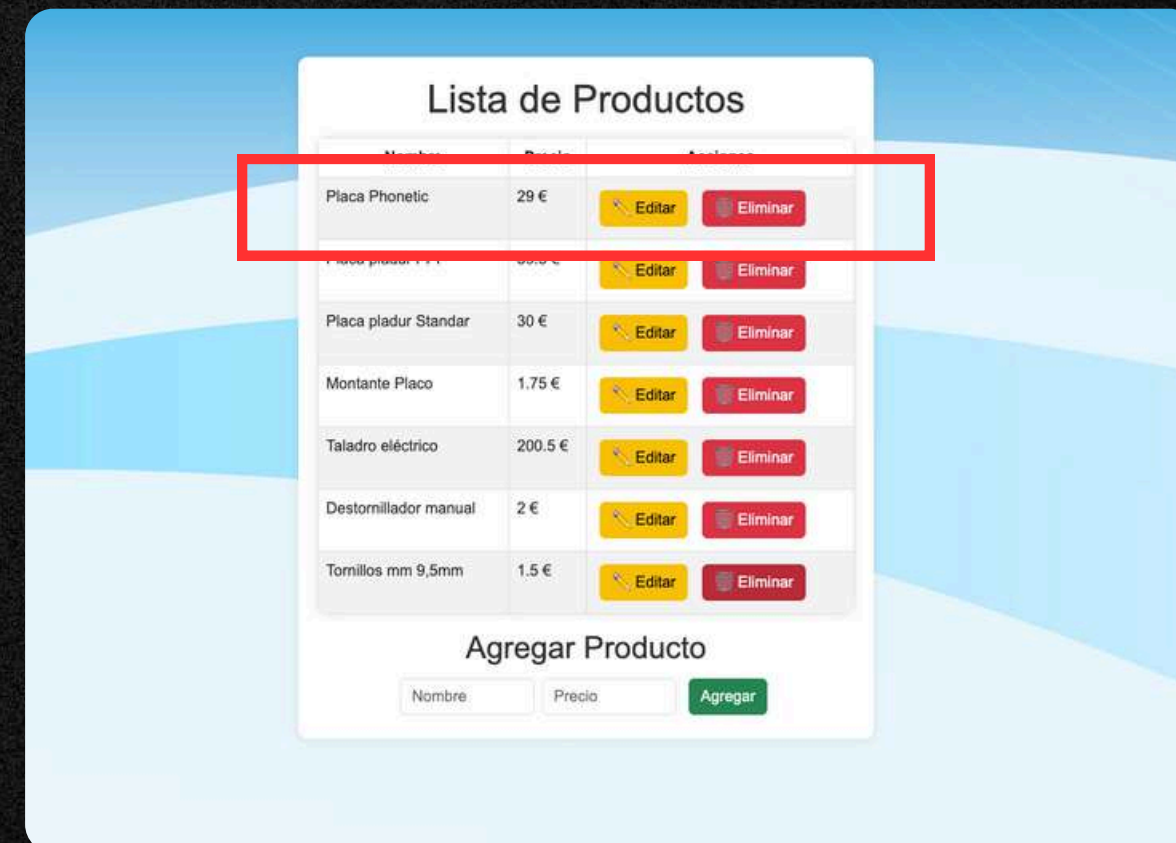
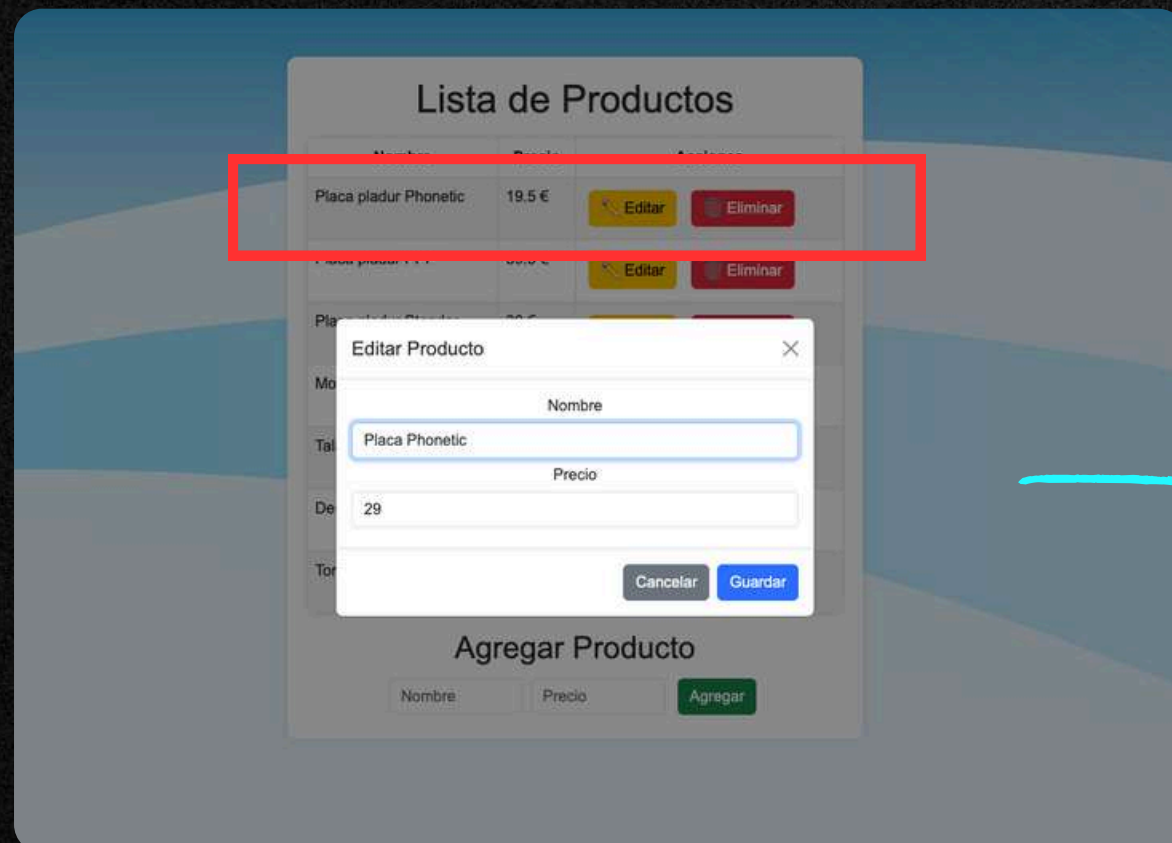




## actualizarProducto() → Editar un producto existente

- Busca el producto en la base de datos con **repository.findById(id)**.
- Si existe, actualiza su nombre y precio y lo guarda con **repository.save(producto)**.
- Se usa en el *endpoint* PUT /productos/{id} del controlador.

```
@Operation(summary = "Actualizar un producto existente")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200", description = "Producto actualizado"),
    @ApiResponse(responseCode = "404", description = "Producto no encontrado")
})
@PutMapping("/{id}")
public Producto actualizarProducto(@PathVariable Long id, @RequestBody Producto producto) {
    return service.actualizarProducto(id, producto);
}
```





## 7. COMUNICACIÓN DEL LADO CLIENTE (FRONTEND) CON LA API

El cliente (interfaz web) se comunica con la API REST utilizando *fetch()*, que permite hacer peticiones HTTP al servidor. Esta es la dirección donde se encuentra la API que maneja los productos > `const url = "http://localhost:8080/productos";`

A continuación , veámos algún ejemplo sobre cómo se aplica esto en nuestro código:

```
async function cargarProductos() {  
  const response = await fetch(url);  
  const productos = await response.json();  
}
```

*Se envía una petición GET a la API, que devuelve un JSON con los productos. Luego, se actualiza la tabla en el frontend.*

```
await fetch(`${url}/${id}`, { method: "DELETE" });
```

*Cuando el usuario confirma la eliminación, se envía una petición DELETE a la API para eliminar el producto.*



## 8. SPRINGBOOT OPENAPI: DOCUMENTANDO LA API

**SpringDoc OpenAPI** permite documentar la API REST de manera automática y sencilla.

En las siguientes diapositivas, explicamos cómo se implementa la documentación de la API en base al controlador *ProductoController*, utilizando anotaciones de **Swagger**.

Anotaciones clave:

- **@Operation**: Describe cada endpoint.
- **@ApiResponse**: Define las posibles respuestas para cada operación.

La documentación generada es accesible a través de una URL como *http://localhost:8080/swagger-ui.html*.



Lo primero que tenemos que hacer es **agregar la dependencia al archivo pom**, para poder trabajar con esta herramienta.

```
<!-- Dependencia para Springdoc OpenAPI -->
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.8.0</version>
</dependency>
```

Después crearemos una clase **SwaggerConfig.java** para documentar la API REST.

```
package com.example.demo;

import io.swagger.v3.oas.annotations.OpenAPIDefinition;

import io.swagger.v3.oas.annotations.info.Info;
import org.springframework.context.annotation.Configuration;

@Configuration
@OpenAPIDefinition(
    info = @Info(
        title = "API de Productos",
        description = "API para gestionar productos en una tienda",
        version = "v1.0"
    )
)

public class SwaggerConfig {
    // Esta clase configura el título, descripción y versión de tu API
}
```



Aquí tenemos nuestros **métodos CRUD documentados con SpringDoc** para darle mayor legibilidad al **código** con esta magnífica herramienta.

```
@Operation(summary = "Obtener lista de productos")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200", description = "Productos encontrados"),
    @ApiResponse(responseCode = "500", description = "Error en el servidor")
})
@GetMapping
public List<Producto> obtenerProductos() {
    return service.obtenerProductos();
}
```

```
@Operation(summary = "Agregar un nuevo producto")
@ApiResponses(value = {
    @ApiResponse(responseCode = "201", description = "Producto creado"),
    @ApiResponse(responseCode = "400", description = "Solicitud inválida")
})
@PostMapping
public Producto agregarProducto(@RequestBody Producto producto) {
    return service.guardarProducto(producto);
}
```

```
@Operation(summary = "Actualizar un producto existente")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200", description = "Producto actualizado"),
    @ApiResponse(responseCode = "404", description = "Producto no encontrado")
})
@PutMapping("/{id}")
public Producto actualizarProducto(@PathVariable Long id, @RequestBody Producto producto) {
    return service.actualizarProducto(id, producto);
}
```

```
@Operation(summary = "Eliminar un producto")
@ApiResponses(value = {
    @ApiResponse(responseCode = "204", description = "Producto eliminado"),
    @ApiResponse(responseCode = "404", description = "Producto no encontrado")
})
@DeleteMapping("/{id}")
public void eliminarProducto(@PathVariable Long id) {
    service.eliminarProducto(id);
}
```





## 9. CONCLUSIONES

En conclusión, el proyecto demuestra cómo crear una **API REST sencilla con Spring Boot para gestionar productos**, realizar pruebas unitarias y documentar la API utilizando herramientas como **SpringDoc OpenAPI**.

Al integrar tecnologías como **Postman** para las pruebas y Bootstrap para la interfaz, se facilita el desarrollo, prueba y despliegue de la aplicación.

Se destaca la importancia de seguir principios de diseño REST para asegurar que la API sea escalable, sencilla de consumir y mantenible. Además, la documentación automática con SpringDoc OpenAPI mejora la experiencia del desarrollador y facilita la integración con otros servicios.



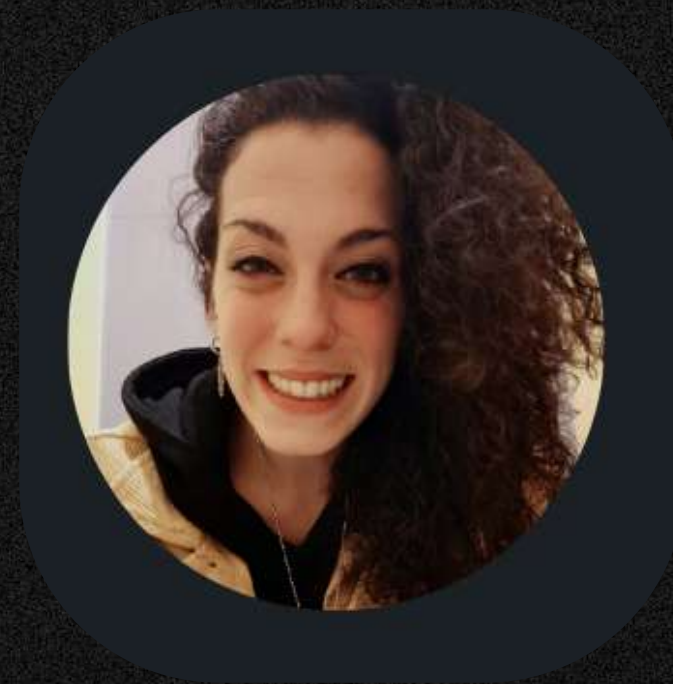


# BIBLIOGRAFÍA

- ✓ Temario Cesur, módulo Desarrollo Web Entorno Servidor, ud6 - *“Utilización de técnicas de acceso a datos”*.
- ✓ Spring Boot Official Documentation - *Focus on maximum service to retain existing customers*
- ✓ JDBC (Java Database Connectivity) Documentation - *Provide solutions to customers*
- ✓ Tech Viajero, Youtube - *“Cómo crear un CRUD #API con Spring Boot y MySQL en 40 minutos”*.



# ¡MUCHAS GRACIAS!



Lidia García Muñoz

