

Informe N°2

Implementación de Microservicios PerfuSmart

**Grupo 1**

Gonzalo Navarrete

Carla Prado

Fernando Zárate

# Índice

[**Índice**](#_oaj3wgs38f2z) **2**

[**1. Introducción 2**](#_pchxvxk06jl)

[**2. Diagrama de arquitectura de microservicios 3**](#_bb16dueut928)

[**3. Estructura del proyecto 4**](#_1vfsb79vnyas)

[**4. Base de datos: MySQL 5**](#_tjpzlmb1ckgx)

[**5. Implementación de los servicios 6**](#_jsqpm74cabua)

[**6. Control de versiones Git – Github 8**](#_s4l8j1bh1n8)

[**7. Conclusión. 9**](#_9owy1fa7ra4s)

# Introducción

El presente informe describe parte del desarrollo y fase de pruebas de un proyecto backend construido con el framework Spring Boot, con el lenguaje Java, con la meta de aprendizaje de buenas prácticas en el desarrollo web y de APIs RESTful.

Este tipo de enfoque es clave dentro del desarrollo FullStack, ya que cubre la lógica del servidor y la integración con bases de datos, que son componentes esenciales en cualquier aplicación web completa.

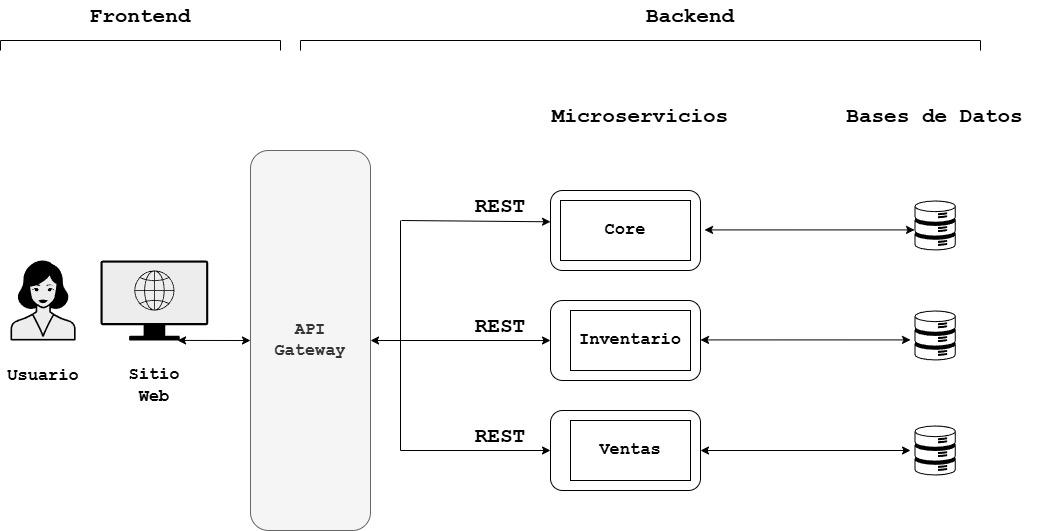
El entorno de trabajo utilizado consiste en herramientas muy usadas por empresas para construir aplicaciones web y sistemas backend de forma rápida y estructurada. Permiten crear servicios que se conectan a bases de datos, procesan información y responden a peticiones desde el navegador o apps móviles. Se usa ampliamente porque ahorra tiempo, reduce errores y facilita el mantenimiento del código.

Como parte de la evaluación, se trabajaron tres entidades independientes —Producto, Cliente y Ventas— cada una representada por su propia tabla en una base de datos MySQL. Las entidades se desarrollaron de forma aislada en un mismo archivo de proyecto, simulando una arquitectura basada en microservicios desacoplados, conforme a las instrucciones del docente.

El proyecto alcanzó la fase de testing y depuración de las API REST, durante la cual se utilizaron herramientas como Postman para verificar el correcto funcionamiento de los endpoints CRUD de cada entidad, que representan, en teoría, la funcionalidad principal de cada microservicio definido en el informe anterior. Además, se integraron prácticas esenciales de desarrollo backend como el uso de controladores, servicios, repositorios, integración con base de datos relacional, y control de versiones a través de Git y GitHub.

# Diagrama de arquitectura de microservicios

El siguiente diagrama se diseñó pensando en el estándar en una arquitectura de microservicios. Siguiendo el principio de modularidad, cada microservicio tiene su propia base de datos y pueden ser desarrollados por separado, sin afectar a los otros.

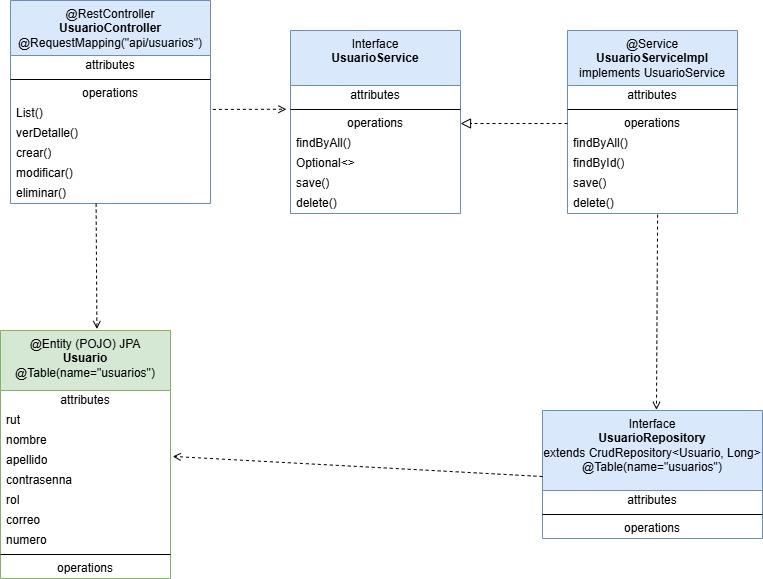


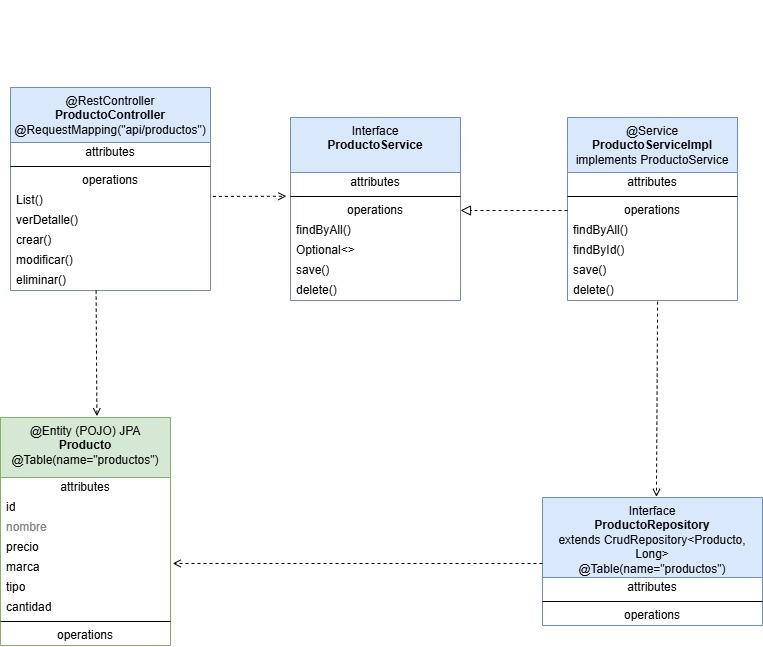
# Estructura del proyecto

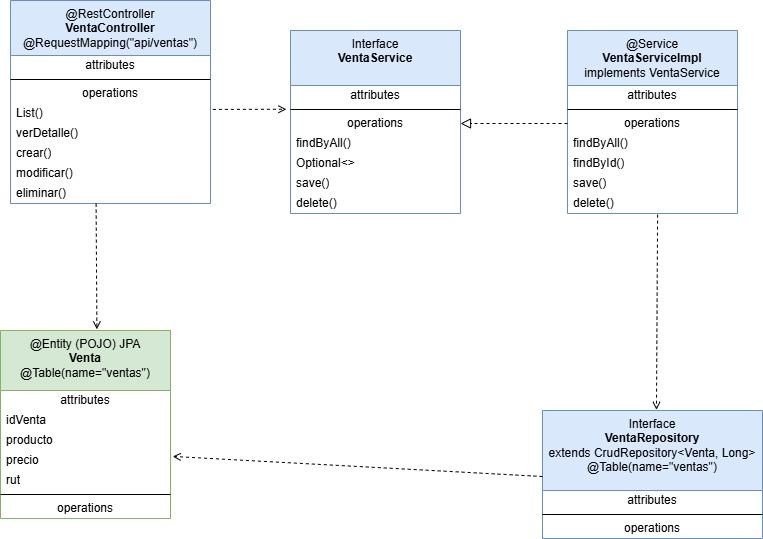
El trabajo de esta unidad se enfocó en la fase de testing de las APIs REST, verificando la correcta comunicación HTTP entre las entidades Spring Boot y la base de datos con sus endpoints asignados. En esta instancia sólo se trabajó con una base de datos, en la que se simula cada microservicio como una entidad y una tabla en dicha base.

A continuación se muestran diagramas de clases actualizados, teniendo en cuenta lo anterior. Cada parte del sistema tiene las siguientes clases:

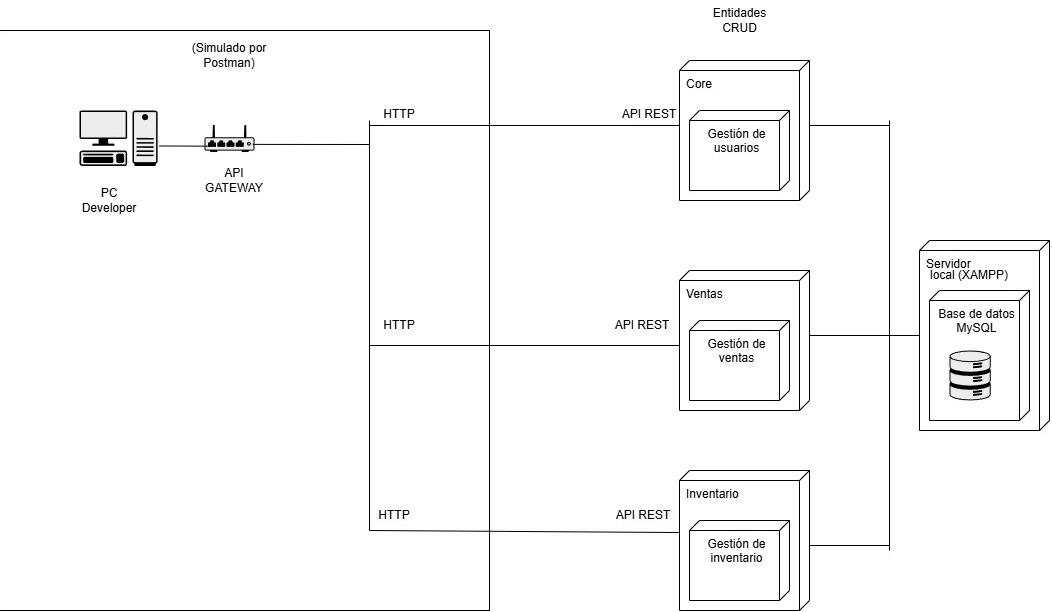
* Entidad
* Controller
* Repository
* Service
* ServiceImpl







La gestión del flujo de solicitudes HTTP, normalmente realizada a través de una API gateway que actúa como puente entre el frontend y el backend, se llevó a cabo por Postman, lo que se explica en mayor detalle en la sección 5 (Implementación de los servicios), en un servidor local (a través de XAMPP). El siguiente diagrama de despliegue representa este entorno.



**Dependencias Maven**

En un proyecto Spring Boot con Maven, las dependencias son bibliotecas externas que la aplicación necesita para funcionar correctamente. Maven se encarga de descargarlas automáticamente desde un repositorio central, gestionar sus versiones y añadirlas al proyecto sin intervención manual. Estas dependencias se declaran en el archivo pom.xml.

Una vez agregadas, se pueden utilizar directamente en el código, generalmente a través de anotaciones y clases que proveen funcionalidades específicas.

Las dependencias utilizadas en este proyecto son:

1. **Spring Web:** Permite crear servicios REST (conjuntos de funciones accesibles desde otros sistemas a través de internet, como consultar información, guardar datos, actualizar registros o eliminarlos), y manejar peticiones HTTP (solicitudes que se envían desde un navegador o cliente para acceder a recursos de un servidor), facilitando el desarrollo de APIs web (interfaces que permiten que dos aplicaciones se comuniquen a través de la web).
2. **Spring Data JPA:** Facilita el acceso a bases de datos usando **JPA** (Java Persistence API, para mapear objetos Java a tablas de una base de datos) y **Hibernate** (una herramienta que implementa JPA y se encarga de traducir automáticamente entre clases Java y datos en la base de datos), permitiendo realizar operaciones CRUD con poco código y de manera eficiente.

* Mapear: relacionar o vincular dos estructuras distintas entre sí, de manera que puedan trabajar juntas de forma coherente.

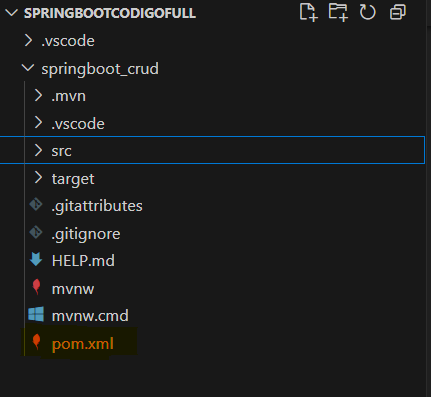
1. **Springboot Dev Tools:** Mejora la experiencia de desarrollo recargando automáticamente la aplicación al detectar cambios en el código.

1. **MySqlDriver:** Es el conector necesario para establecer la conexión entre la aplicación y una base de datos MySQL.

**Archivo pom.xml de Maven**

* **Ubicación**

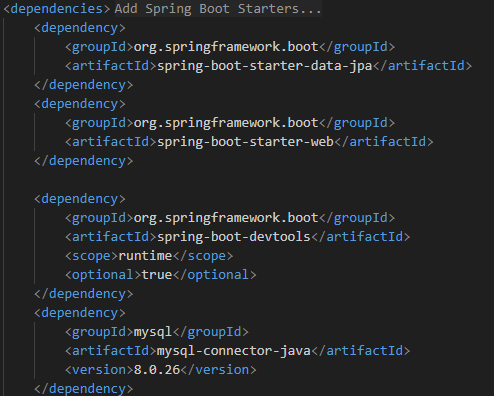
El archivo pom.xml del proyecto es un archivo de configuración usado por Maven para gestionar las dependencias, el ciclo de vida de construcción y otros aspectos esenciales, como herramientas de gestión y construcción de proyectos. Se encuentra en el directorio principal.



* **Especificando Dependencias**

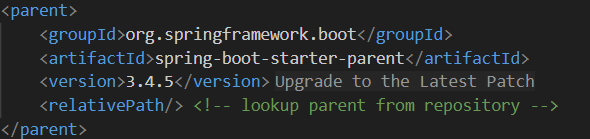
En esta sección se definen las dependencias utilizadas entre los tags <dependency>, que contienen identificadores Maven, tales como:

El tag <artifactId> define el nombre único del artefacto (es decir, del proyecto o dependencia) a utilizar, que puede ser un archivo .jar o .war. Se usa junto con <groupId> y <version> para ubicar y descargar dependencias desde un repositorio Maven.



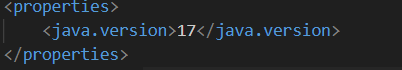
* **Configuraciones parent**

El tag <parent> en el archivo pom.xml se usa para heredar configuraciones comunes desde otro proyecto o POM base. De no utilizarse, habría que definir manualmente todas las versiones de dependencias, configuraciones de plugins y propiedades, lo cual es más propenso a errores y menos mantenible.



* **Versión de Java**

En <properties>, está la versión de Java utilizada. Esto especifica que el proyecto usa Java 17.



**Componentes**

Los componentes son clases que representan partes funcionales de la aplicación y que Spring detecta y administra automáticamente para que puedan ser utilizadas (inyectadas) donde se necesiten. Las clases son marcadas con anotaciones con el siguiente formato: @Component.

1. **Anotaciones de componente o anotaciones de propósito** (Indican el rol de cada clase dentro de la arquitectura Spring Boot)
2. **@SpringBootApplication:** Es la anotación principal de arranque. Le dice a Spring Boot que cargue y configure automáticamente todos los componentes del proyecto (controladores, servicios, repositorios, etc.).
3. **@RestController:** Indica que la clase se encarga de recibir peticiones HTTP del cliente (como GET, POST, PUT, DELETE) y devolver respuestas en formato JSON. Es la forma de crear APIs web.

* **GET**: Solicita datos del servidor (con las bases de datos).
* **POST**: Envía datos al servidor para crear un nuevo recurso.
* **PUT**: Envía datos para modificar un recurso existente.
* **DELETE**: Solicita eliminar un recurso del servidor.

1. **@Repository:** Marca una clase que accede a la base de datos, y se encarga de operaciones como guardar, buscar, actualizar o eliminar datos (CRUD).
2. **@Service:** Define una clase que contiene la lógica de negocio. Actúa como intermediario entre el controlador (@RestController) y el repositorio (@Repository).
3. **@Entity:** Indica que la clase representa una tabla en la base de datos. Cada atributo de la clase corresponde a una columna.
4. **Anotaciones y funciones por capa.**

En este proyecto, se utilizan diversas anotaciones que permiten un desarrollo eficiente en las funcionalidades de los microservicios, las anotaciones utilizadas son:

1.-**@RequestMapping:** define la ruta base que un controlador o método debe manejar.

2.-**@GetMapping:** se utiliza para obtener datos del servidor sin hacer ninguna modificación.

3.-**@PostMapping**: se utiliza para ingresar datos al servidor.

4.-**@PutMapping**: Se utiliza para modificar datos del servidor.

5.-**@DeleteMapping:**Se utiliza para eliminar datos del servidor.

6.-**@Autowired:** se utiliza para inyectar automáticamente dependencias en los componentes de la aplicación.

7.-**@Table**:se utiliza para especificar el nombre de la tabla de base de datos a la que estará vinculada la entidad.

8.-**@Id:** define el atributo como identificador primario.

9.-**@GeneratedValue:** indica que el atributo tendrá un valor generado automáticamente.

10.-**@Column:**indica el nombre que tendrá el atributo en la tabla de la base de datos.

11.-**@Override:**se utiliza para indicar que se está sobrescribiendo un método definido previamente en otra clase o interfaz  
12.-**@Transactional:** gestiona las transacciones con la base de datos, si todo sale bien genera un commit, si algo falla, hace un rollback para mantener la integridad de los datos.

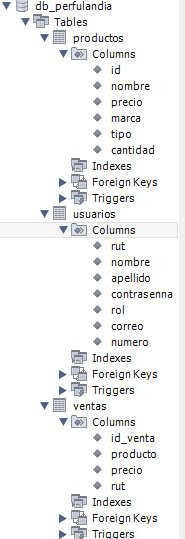
13.-**@PathVariable:** en SpringBoot sirve para extraer un valor de la URL y pasarlo como argumento al método del controlador.

14.-**@RequestBody:** recibir datos en formato JSON desde el cuerpo de una solicitud HTTP (por postman en este caso)y convertirlos automáticamente en un objeto Java.

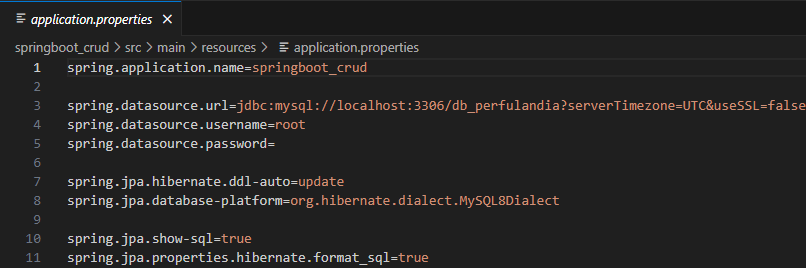
# Base de datos:

En este proyecto se utiliza el motor de base de datos **MySQL**, el cual es gestionado y visualizado mediante **MySQL Workbench**. El proyecto maneja tres entidades principales que representan tablas en la base de datos: **Usuario**, **Producto** y **Venta**. Cada una de estas entidades está mapeada con CRUD (Create, Read, Update, and Delete) .

**Estructura de Tablas**

**\*\*\*\*\*\*\*\*\*\*\*\*\***

1. **Usuario** Contiene campos como **rut**, **nombre**, **apellido** y **contraseña, rol, correo y número de teléfono**. Representa a los usuarios del sistema.
2. **Producto** Incluye campos como **id**, **nombre**, **precio**, **marca, tipo** y **cantidad**. Esta tabla almacena los productos que se gestionan dentro de la aplicación.
3. **Venta** Posee campos como **id**, **producto**, **precio** y **rut**. Se usa para registrar las transacciones realizadas, enlazando usuarios y productos.

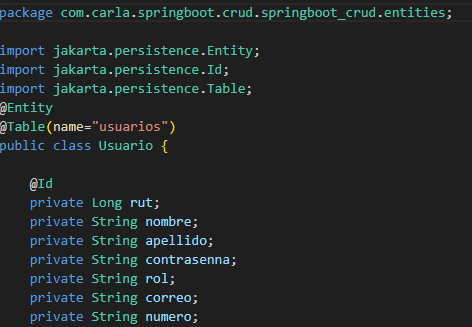


Visualización del archivo application.properties en el cual se presenta la conexión con la base de datos utilizada.

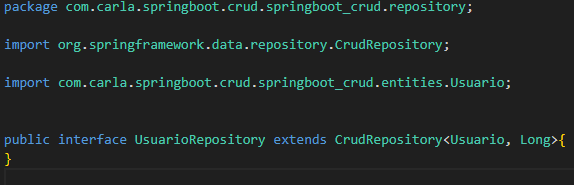
# Implementación de los servicios

En este proyecto se encuentran tres microservicios desarrollados, siendo estos el microservicio de usuarios, productos y ventas.

**Entidad: Usuario.java** Clase anotada con **@Entity**, representa un Usuario con **rut, nombre, apellido,contraseña, rol, correo y numero.**

****

**Repositorio: UsuarioRepository.java**Interfaz que extiende **CrudRepository<Usuario, Long>**, permitiendo heredar métodos CRUD

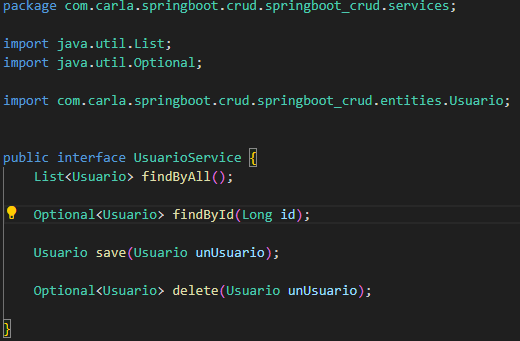


**Servicios:**.

**UsuarioService.java**

Es una interfaz que define los métodos que deben implementarse para gestionar usuarios, como buscar, guardar o eliminar.

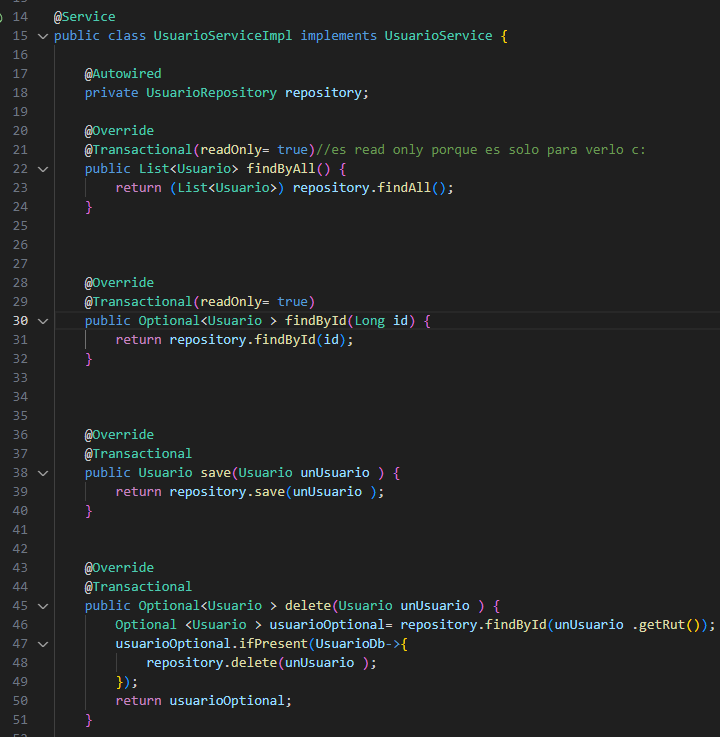
Sirve como base para la clase **UsuarioServiceImpl.java**

****

**UsuarioServiceImpl.java**

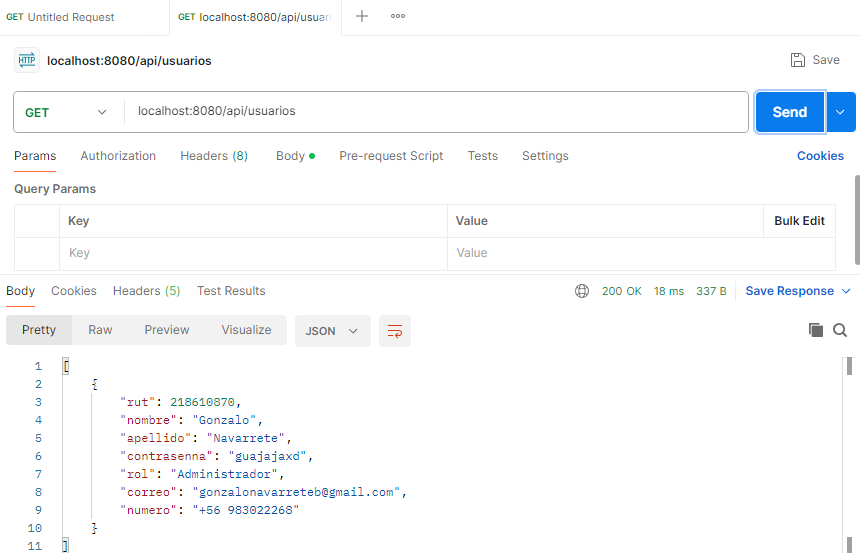
Es una clase de servicio, anotada con **@Service**, que implementa la interfaz **UsuarioService.java**

Se encarga de realizar las operaciones reales usando el repositorio **UsuarioRepository.java.**

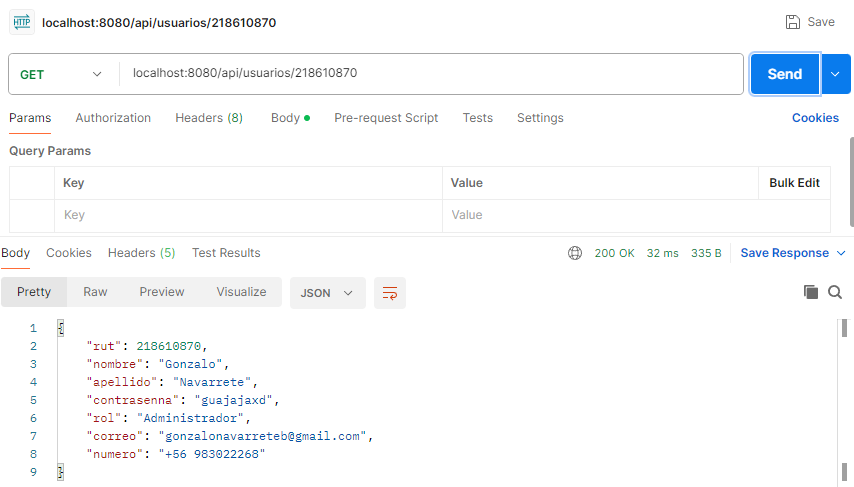


**Controlador: UsuarioController.java** Expone los siguientes métodos:

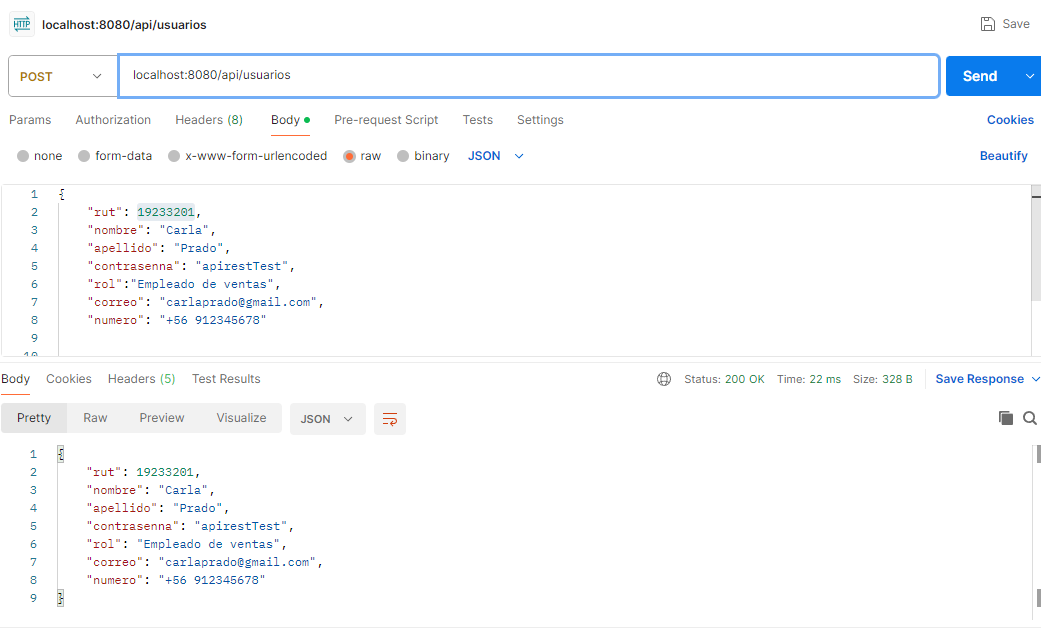
**GET api/Usuarios** → Lista todos los productos



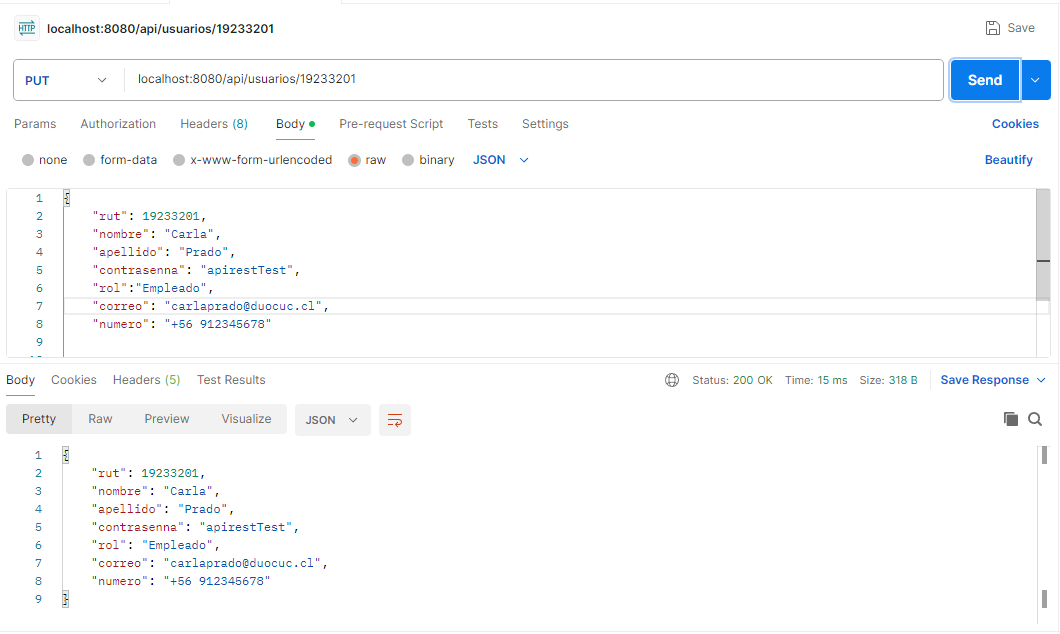
**GET api/Usuario/{rut}** → Busca un producto por su rut



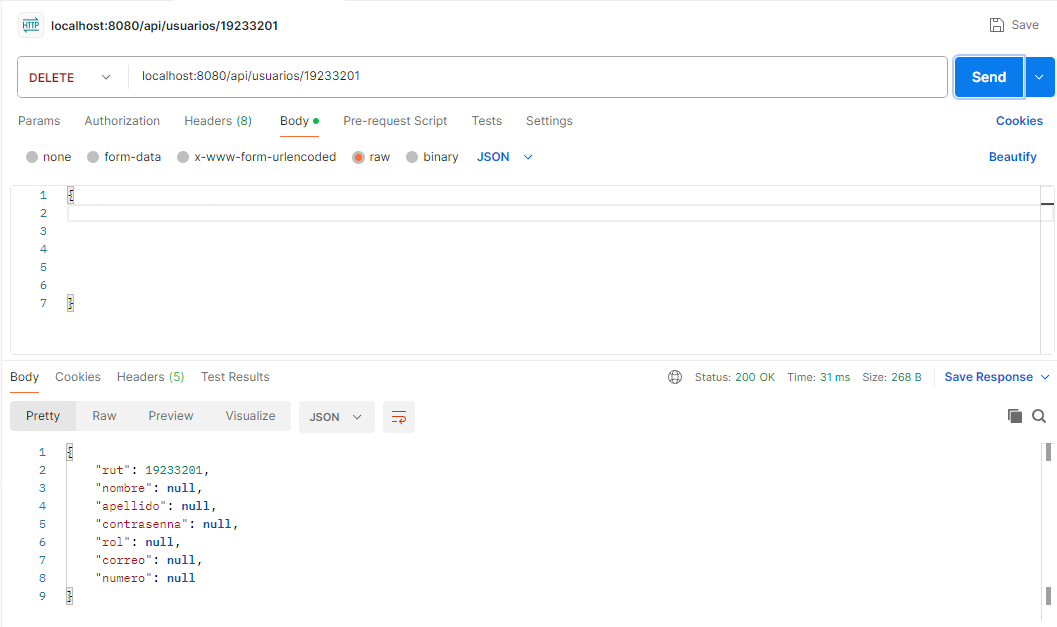
**POST api/Usuarios** → Crea un nuevo Usuario

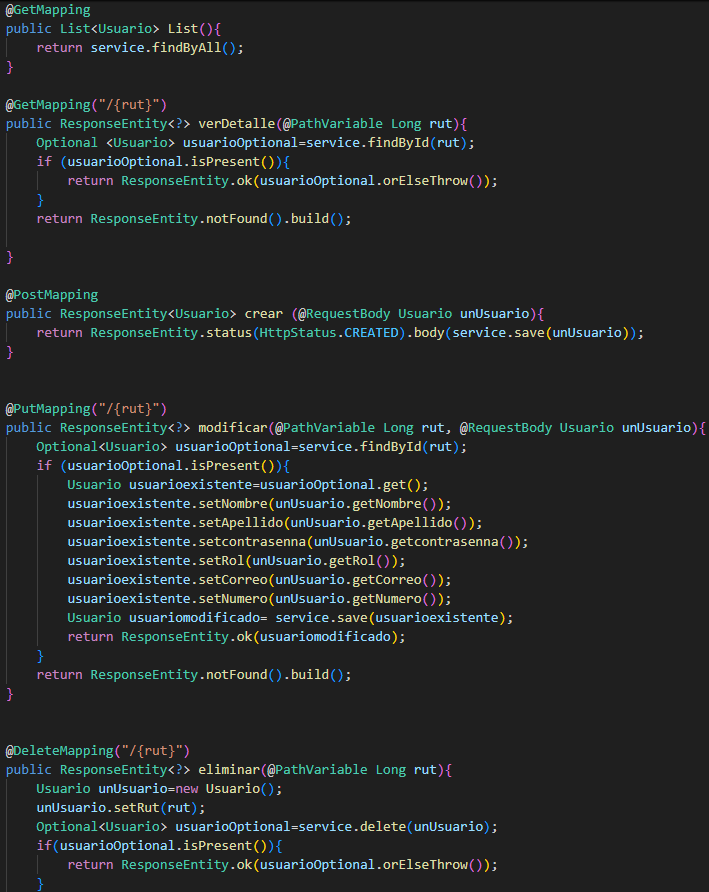


**PUT api/Usuario/{rut}** → Actualiza un Usuario

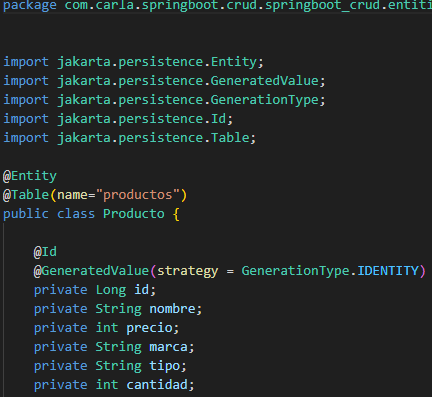


**DELETE api/Usuario/{rut}** → Elimina un Usuario

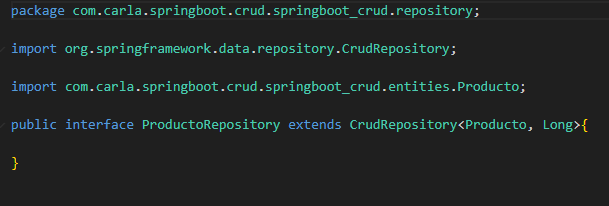


****

**Entidad: Producto.java** Clase anotada con **@Entity**, representa un producto con **id**, **nombre, precio,** **marca, tipo y cantidad**.



**Repositorio: ProductoRepository.java** Extiende **CrudRepository<Producto,** **Long>,** permitiendo heredar métodos CRUD.

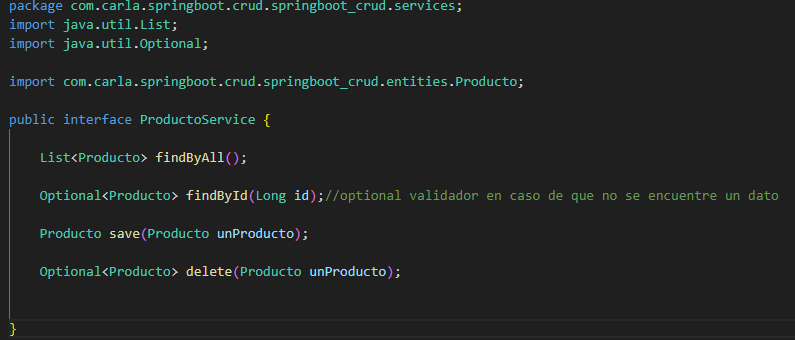


**Servicios:**.

**ProductoService.java**

Es una interfaz que define los métodos que deben implementarse para gestionar productos, como buscar, guardar o eliminar.

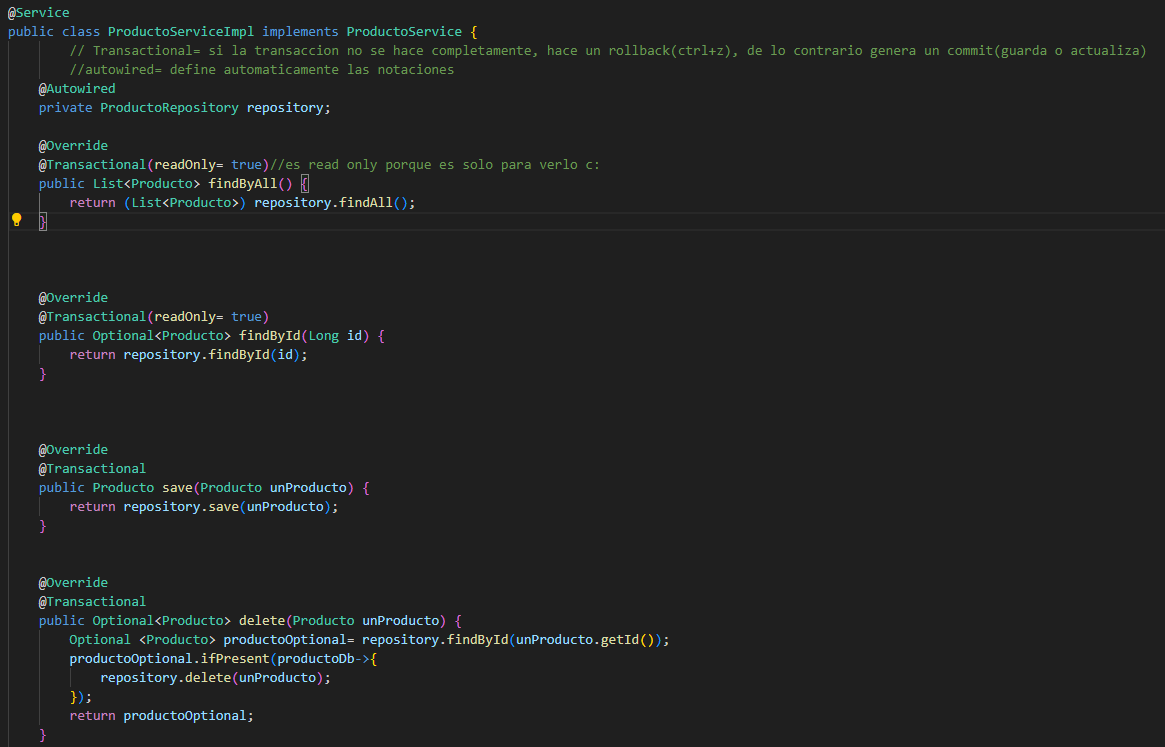
Sirve como base para la clase **ProductoServiceImpl.java.**

****

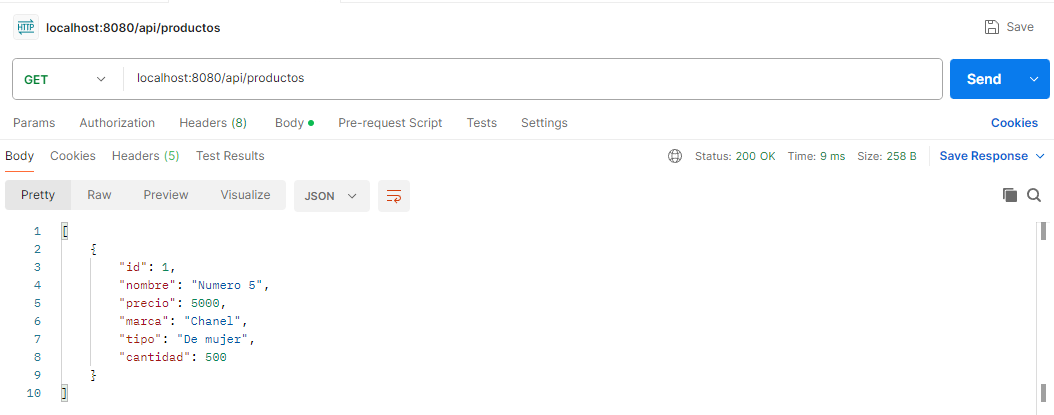
**ProductoServiceImpl.java**

Es una clase de servicio, anotada con **@Service**, que implementa la interfaz **ProductoService.java.**

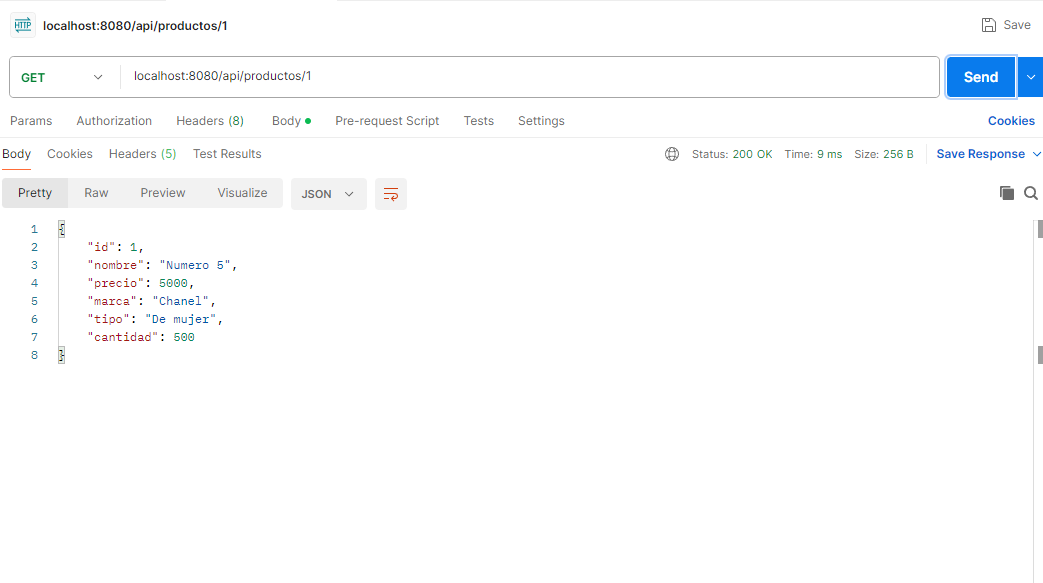
Se encarga de realizar las operaciones reales usando el repositorio **ProductoRepository.java.**

****

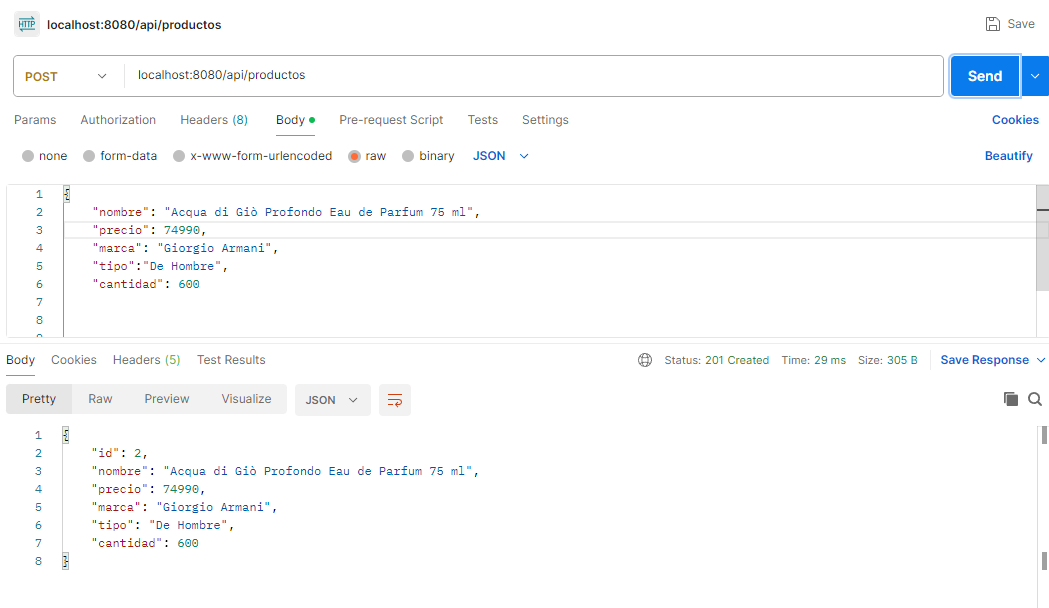
**Controlador: ProductoController.java** Expone los siguientes método http:

**GET /productos** → Lista todos los productos   


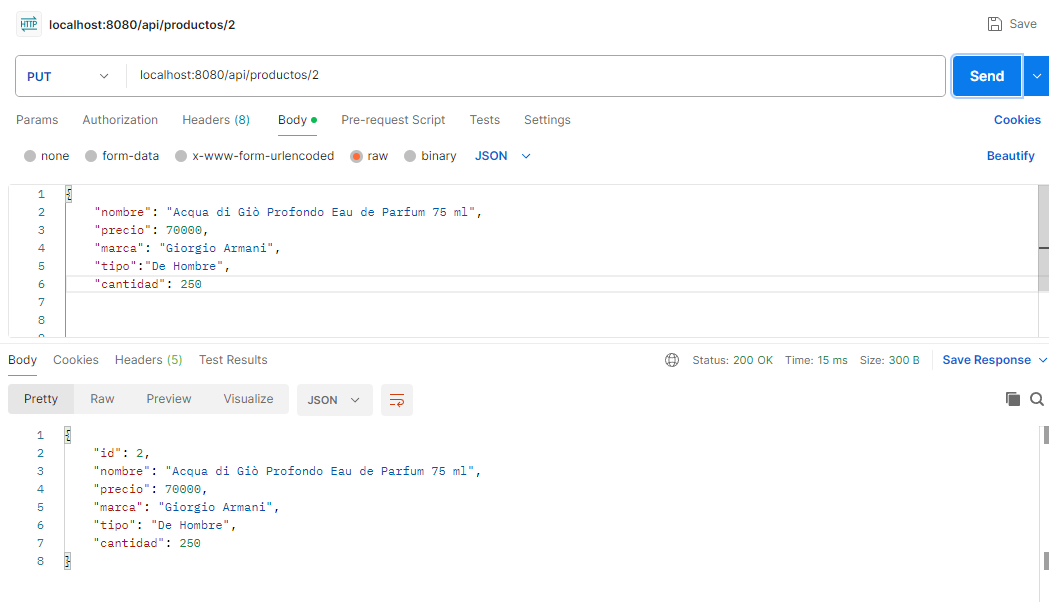
**GET /productos/{id}** → Busca un producto por ID



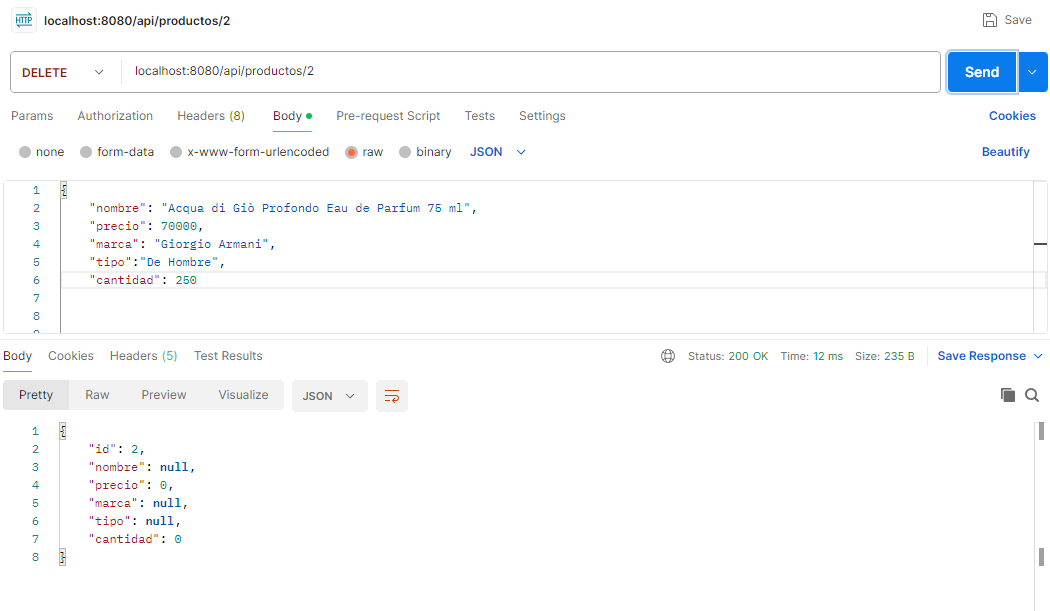
**POST /productos** → Crea un nuevo producto

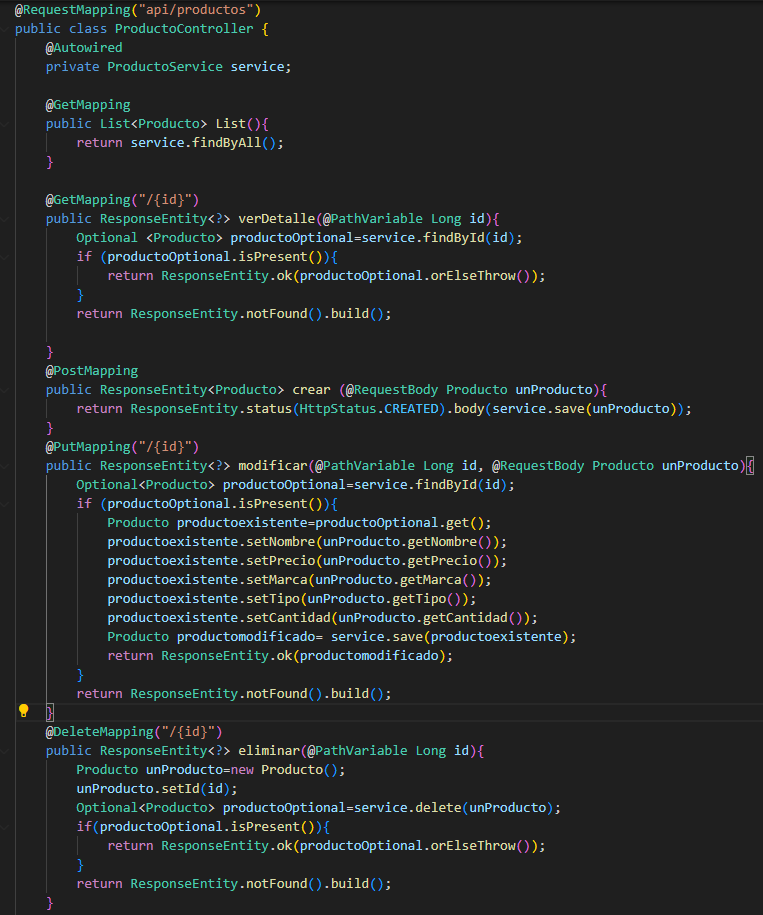


**PUT /productos/{id}** → Actualiza un producto

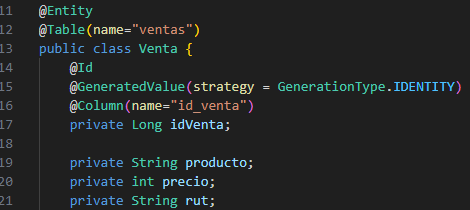


**DELETE /productos/{id}** → Elimina un producto

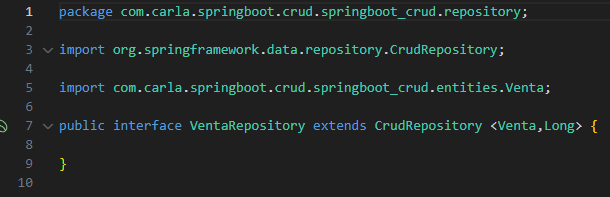




**Entidad: Venta.java** Anotada con **@Entity**, representa una venta con **id,nombre del producto,precio, rut del comprador**

****

**Repositorio: VentaRepository.java** Extiende **CrudRepository<Venta, Long>**, permitiendo heredar métodos CRUD.



**Servicios:**.

**VentaService.java**

Es una interfaz que define los métodos que deben implementarse para gestionar ventas, como buscar, guardar o eliminar.

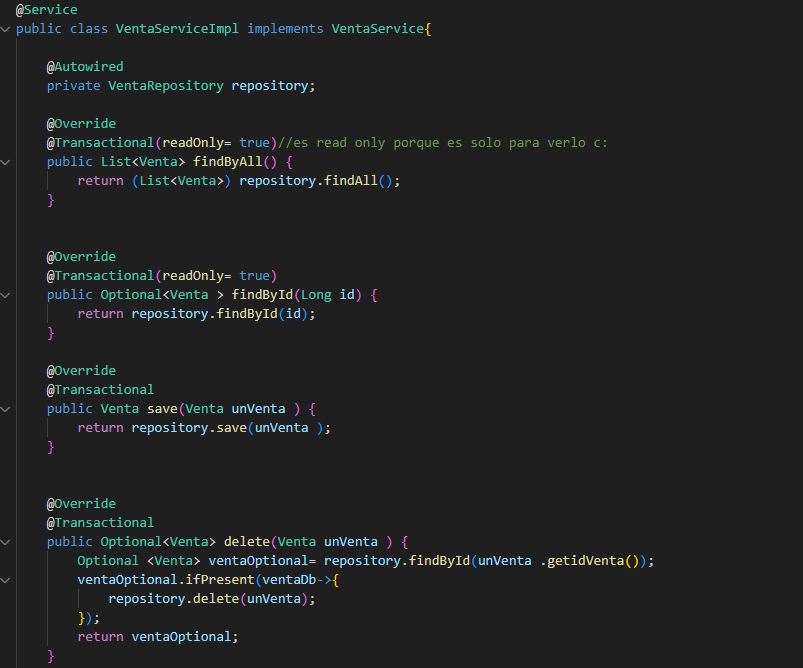
Sirve como base para la clase **VentaServiceImpl.java.**

****

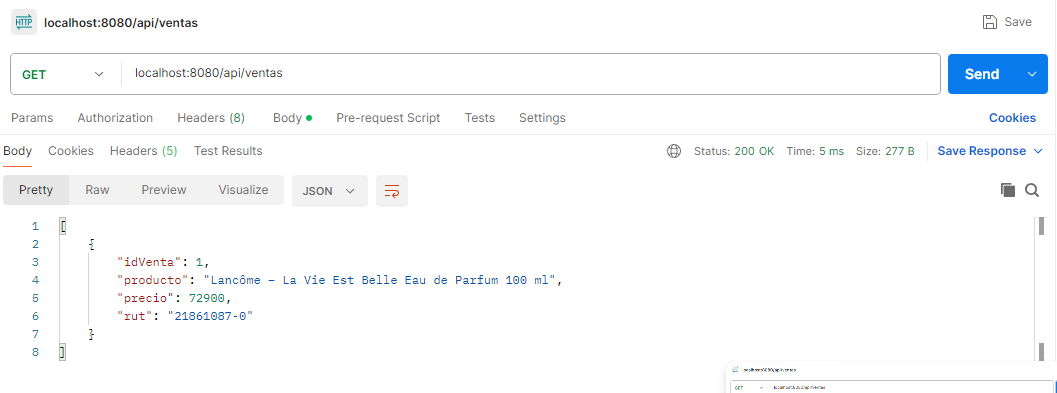
**VentaServiceImpl.java**

Es una clase de servicio, anotada con **@Service**, que implementa la interfaz **VentaService.java.**

Se encarga de realizar las operaciones reales usando el repositorio **VentaRepository.java.**

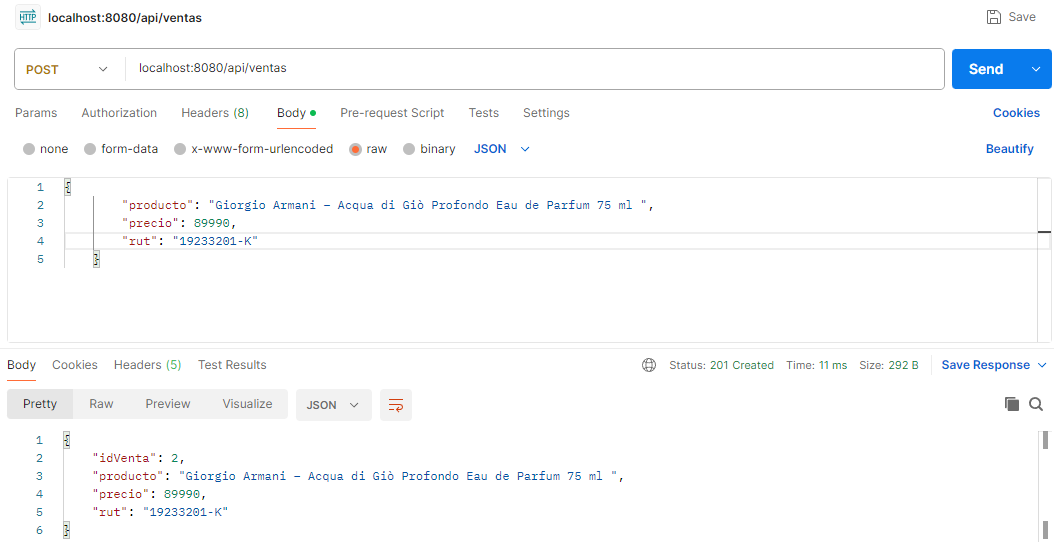
****

**Controlador: VentaController.java** Expone los siguientes endpoints:

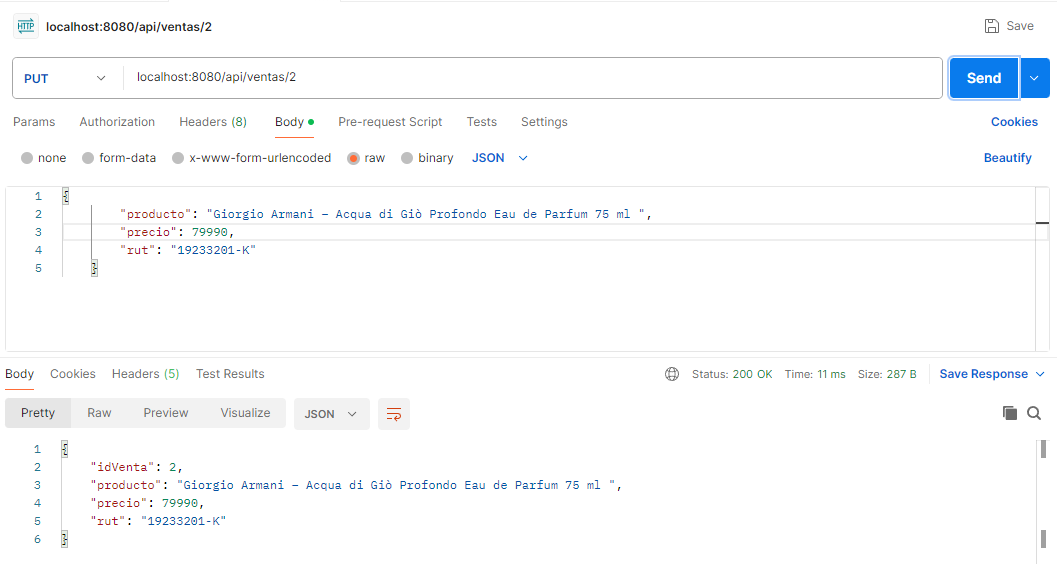
**GET api/ventas** → Lista todas las ventas  


**GET api/ventas/{id}** → Busca un venta por ID  

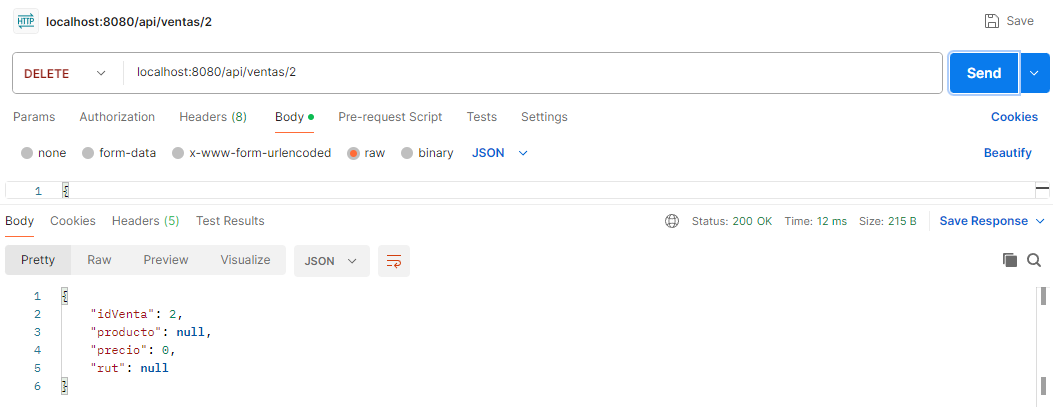

**POST api/ventas** → Crea una nueva venta

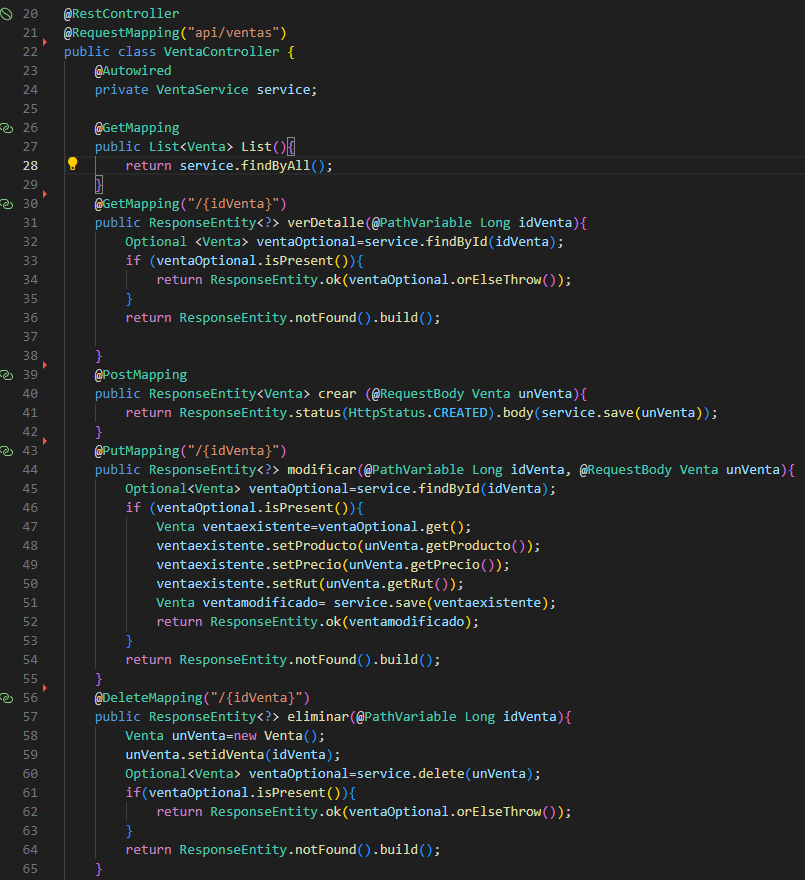


**PUT api/ventas/{id}** → Actualiza un proceso de venta



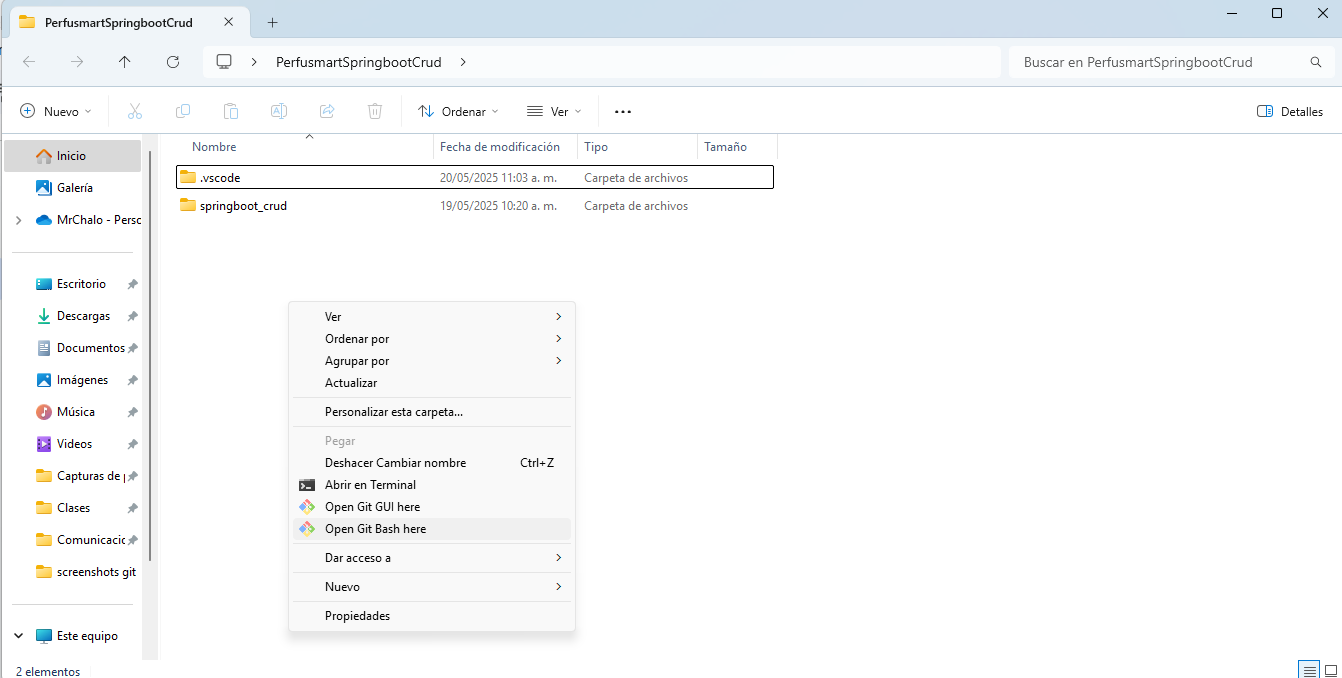
**DELETE api/ventas/{id}** → Elimina una venta

\_



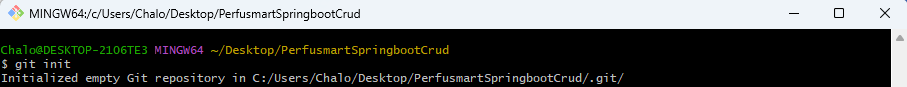
# Control de versiones Git – Github

Para comenzar, se abre el gitBash desde la carpeta en donde se localiza nuestro proyecto:





Desde la terminal de git, inicializamos el repositorio:



Configuramos el repositorio con nuestras credenciales:





Posteriormente, generamos un token en nuestro github, con el scope “repo”, para luego utilizarlo como password:



Vinculamos la carpeta local de nuestro proyecto, con nuestro repositorio remoto en github:



Luego preparamos los archivos locales de nuestro proyecto para ser subidos al repositorio remoto:

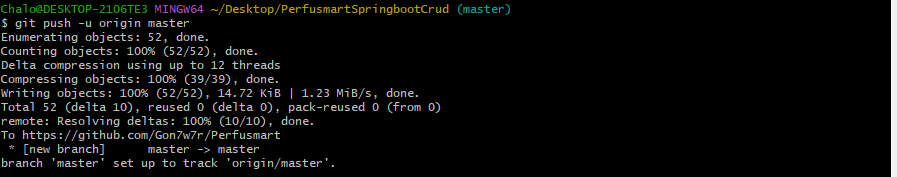


Realizamos control de versión:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Subimos los archivos al repositorio:



Podemos visualizar el repositorio actualizado con los archivos de nuestro proyecto llamado Perfusmart.

# Conclusión.

El desarrollo de este proyecto con Spring Boot permitió aplicar de forma práctica conceptos clave del backend moderno, como la creación de APIs RESTful, la separación por capas (controlador, servicio y repositorio), y la integración con una base de datos relacional mediante JPA. Aunque el enfoque propuesto simulaba una arquitectura basada en microservicios, el trabajo se ejecutó de manera local sobre una sola base de datos, lo cual facilitó el entendimiento de los componentes sin complejidad adicional.

Además, el uso de herramientas como Postman, Git y MySQL Workbench reforzó habilidades fundamentales para el desarrollo web y la depuración de servicios. Este tipo de proyectos representa una base sólida para entender la lógica del lado servidor en aplicaciones Full Stack, y constituye un paso importante para abordar sistemas más complejos en entornos reales.