Pestaña 1

**Transformación digital de Perfulandia SPA**

**Integrantes:**

**Rocío Trujillo**

**Gabriel Olguin**

**Matías Olivera**

**Índice**

1. **Mejora de la Evaluación 1.**

1.1 Qué falló.

1.2 Que mejoramos.

1. **Mejora de la Evaluación 2.**

2.1 Qué falló.

2.2 Que mejoramos.

1. **Diagrama de arquitectura de microservicios.**
2. **Plan de pruebas.**

4.1 Pruebas Unitarias.

4.2 Pruebas de integración.

1. **Ejecución de pruebas**.

5.1 Pruebas Unitarias.

5.2 Pruebas de integración.

1. **Git-GitHub.**
2. **Conclusión.**

# **1. Mejoras Evaluación 1**

**1.1.- Qué falló.**

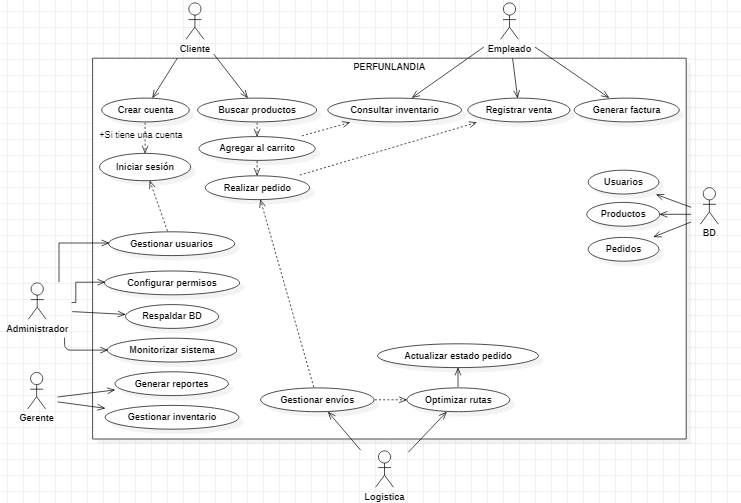
Revisando y ayudándonos de la guía del profesor nos dimos cuenta de que tuvimos los siguientes fallos en la evaluación 1.

**1.2.- Qué Mejoramos.**

Revisando y ayudándonos de la guía del profesor nos dimos cuenta de que tuvimos los siguientes fallos en la evaluación 1.

**1- Diagramas incompletos**

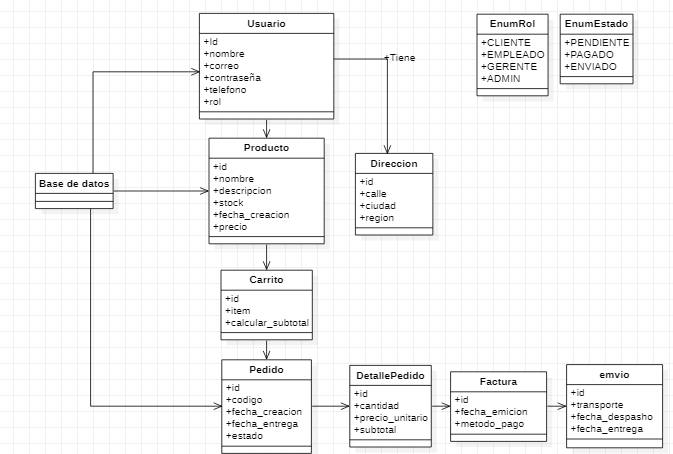
Fallo principalmente la arquitectura de los diagramas y también que agregamos diagramas que no se pidieron cuando solo se pedía el diagrama de caso de uso, diagrama de clases y diagrama de despliegue.

**1.1.-Diagrama de Casos de Uso:**

Este diagrama representa las **interacciones entre los distintos actores del sistema y las funcionalidades clave** de la plataforma Perfulandia.

* **Clientes** pueden crear cuentas, iniciar sesión, buscar productos, agregar al carrito y realizar pedidos.
* **Empleados** pueden consultar el inventario, registrar ventas y generar facturas.
* **Administradores** gestionan usuarios, permisos, respaldos, monitoreo del sistema y configuración general.
* **Gerentes** tienen funciones de control más estratégicas, como generar reportes y gestionar inventario.
* **Logística** se encarga de gestionar envíos, actualizar estados de pedidos y optimizar rutas de entrega.
* Las entidades como **Usuarios**, **Productos** y **Pedidos** interactúan con la **Base de Datos (BD)**.

**1.2 Diagrama de clases**

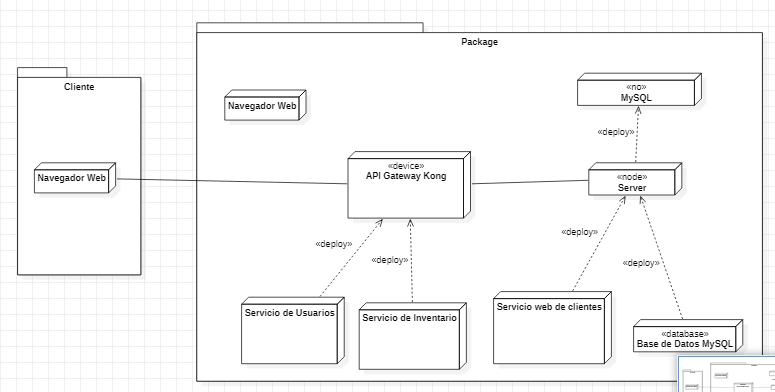


El diagrama de clases representa la estructura lógica del sistema y sus entidades principales. Cada clase modela un componente del negocio de Perfulandia SPA:

* **Usuario**: Contiene los datos personales y de autenticación del usuario, incluyendo su rol (Cliente, Empleado, Gerente o Admin).
* **Producto**: Almacena información de los productos disponibles en la tienda, como nombre, precio y stock.
* **Carrito**: Asocia productos seleccionados por el usuario antes de realizar un pedido. Tiene lógica para calcular subtotales.
* **Pedido**: Representa una orden de compra. Incluye fecha de creación, entrega y estado del pedido.
* **DetallePedido**: Desglosa los productos incluidos en un pedido, indicando cantidad y precios unitarios.
* **Factura**: Registra la emisión de facturas y el método de pago correspondiente a un pedido.
* **Envío**: Registra datos logísticos como transporte, fechas de despacho y entrega.
* **Dirección**: Relacionada al usuario, permite asociar direcciones de envío o facturación.
* **Enumeraciones (EnumRol y EnumEstado)**: Definen roles de usuario y estados posibles de un pedido.

Este modelo permite organizar claramente los datos del sistema, facilitando su implementación en la base de datos y en la lógica de los microservicios.

**1.3 Diagrama de despliegue**



El diagrama muestra cómo se distribuyen los componentes del sistema de Perfulandia SPA en el entorno de ejecución.

* Los **clientes** acceden al sistema desde un **navegador web**.
* Todas las peticiones pasan por el **API Gateway Kong**, que en ruta las solicitudes a los microservicios correspondientes.
* Se despliegan tres servicios principales:  
  + **Servicio de Usuarios**
  + **Servicio de Inventario**
  + **Servicio Web de Clientes**
* Cada servicio se comunica con una **base de datos MySQL**, que está desplegada en un **servidor dedicado**.
* El nodo Server representa la infraestructura donde corren los servicios y la base de datos.

Este despliegue asegura modularidad, facilita la escalabilidad y permite manejar cada servicio de forma independiente.

**2- Herramientas de trabajo colaborativo**

Mejoramos el uso de herramientas como miro, trello o github las cuales estamos usando de forma que nos facilite el trabajo en equipo y el desarrollo de este proyecto.

**3- Documentación técnica**

mejoramos en el apartado de subir de mejor maneja los archivos al github para un mejor orden y una entrega más limpia.

# 

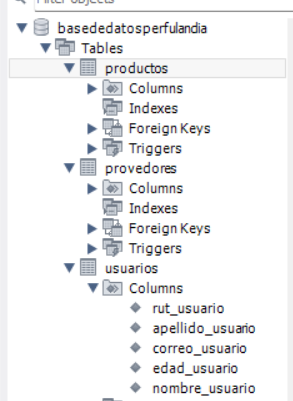
# **2. Mejoras Evaluacion 2**

**2.1.** **Qué falló.**

Al momento de construir el código, dentro de main, entidad de usuario se codificó como auto incremental el atributo id para identificar a los usuarios, cuando no era necesario al momento de presentarlo se nos dio el feedback para solucionarlo.

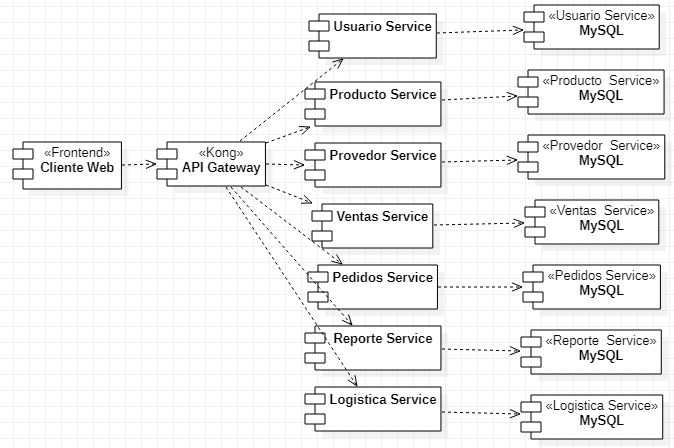
**2.2. Que mejoramos.**

Se realizó un cambio dentro de la entidad para dejar como identificador único, el rut de los usuarios para poder buscarlos y modificarlos dentro de la base de datos.



### 

# **3. Diagrama de arquitectura de microservicios**



* **Usuario Service**Usa API REST para registrar, autenticar y gestionar usuarios. Se elige REST por su simplicidad y compatibilidad con JWT para autenticación segura.
* **Product Service**Usa API REST para gestionar productos. Es ideal para operaciones CRUD rápidas y consultas desde el frontend.
* **Proveedor Service**Expone API REST para el mantenimiento de proveedores. REST es suficiente ya que son operaciones poco frecuentes.
* **Venta Service**API REST para registrar ventas y generar facturas. Se requiere respuesta inmediata al usuario, por eso se usa REST.
* **Pedido Service**Combina API REST (para crear pedidos) y eventos asincrónicos con RabbitMQ (para procesarlos sin bloquear al usuario). Esto mejora el rendimiento y escalabilidad.
* **Logística Service**Recibe pedidos por mensajería asincrónica (RabbitMQ) y expone una API REST para consultar el estado de envíos. Así se separan procesamiento y consultas.
* **Reporte Service**Usa API REST solo para lectura. Permite generar reportes desde interfaces administrativas sin afectar otros servicios.
* **Atencion Cliente Service**REST para crear y gestionar tickets, y WebSocket para integrar chat en tiempo real en etapas futuras. Se adapta según el canal de atención al cliente.

## 

# **4. Plan de pruebas**

## **Herramientas y Frameworks Utilizados**

Para la implementación de las pruebas, se utilizaron las siguientes herramientas:

1. **JUnit 5**: Framework principal para la creación de pruebas unitarias en Java.
2. **Mockito**: Para simular dependencias externas (como repositorios o servicios) y permitir pruebas aisladas.
3. **Spring Boot Test**: Proporciona un entorno de pruebas de integración usando contextos de Spring.
4. **Postman**: Para ejecutar y validar pruebas de integración sobre los endpoints REST.
5. **Maven**: Para la gestión de dependencias y ejecución de pruebas mediante el plugin mvn test.

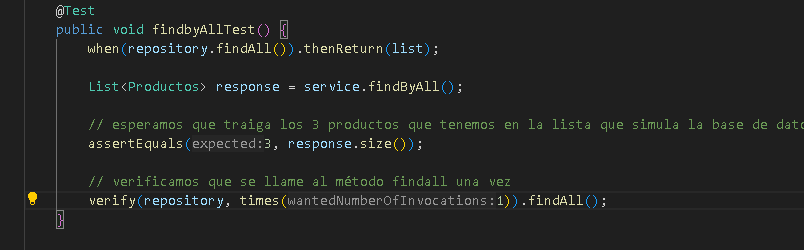
# 

# **4.1 Pruebas Unitarias**

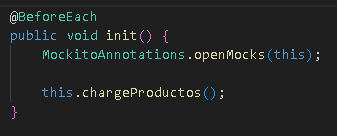
### **Codificación**

Se implementaron las mismas pruebas unitarias para los servicios de **UsuarioServiceImpl**, **ProductoServiceImpl** y **ProveedorServiceImpl**.

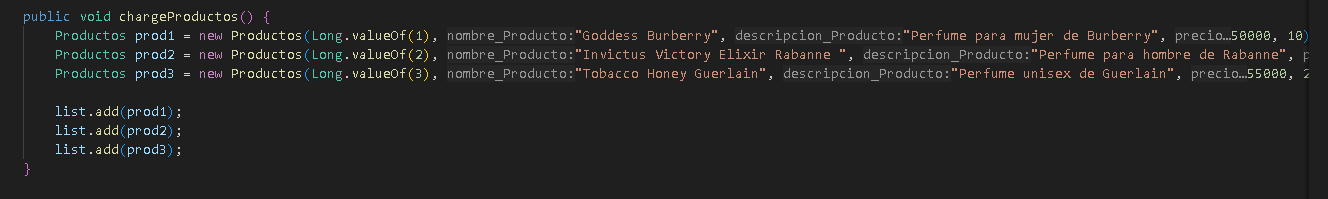
La siguiente prueba verifica que el método findByAll() de ProductosServicesImpl devuelva la cantidad correcta de productos simulados y que invoque al repositorio una sola vez.



@BeforeEach hace que lo que esta dentro se repita cada vez que se hace el test



Que en este caso es una lista para simular datos



# **4.2 Pruebas de integración**

Se implementaron las mismas pruebas de integración para los servicios de **ProductosRestController**, **UsuariosRestController** y **ProveedoresRestController**.

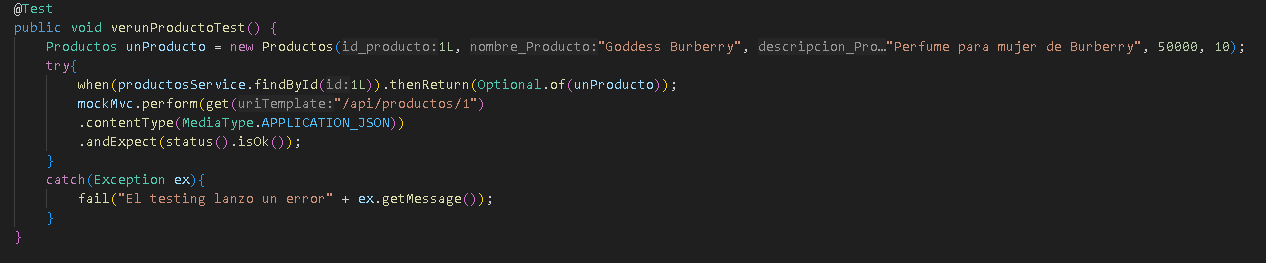
**verProductosTest()**: Esta prueba verifica que el endpoint GET /api/productos funcione correctamente.

* Se simula que el método productosService.findByAll() devuelve una lista de productos.
* Luego, se realiza una petición GET usando MockMvc.
* Se valida que la respuesta tenga código **200 OK**, indicando que se obtuvieron los productos exitosamente.



### **verunProductoTest():** Esta prueba comprueba que al consultar un producto existente mediante GET /api/productos/{id}, se retorne una respuesta satisfactoria:

* Se simula que el producto con ID 1 existe, retornando un Optional con datos.
* Se realiza la petición GET al endpoint.
* Se espera una respuesta con código 200 OK



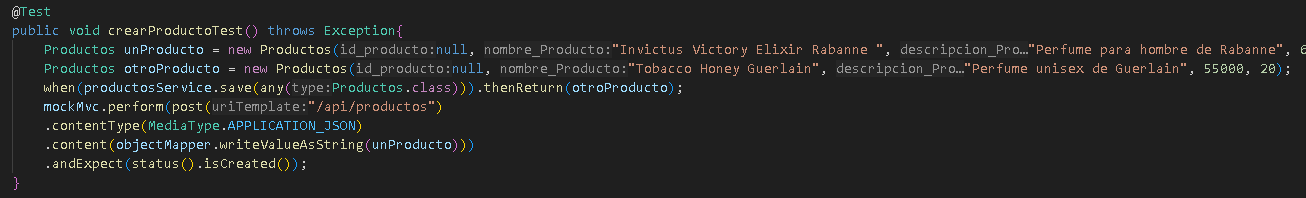
### **productoNoExisteTest():** Verifica el comportamiento del sistema al intentar obtener un producto que **no existe**:

* Se simula que el servicio devuelve Optional.empty() para el ID 10.
* Al hacer la petición GET, el sistema debe retornar un **404 Not Found**, indicando que no se encontró el producto solicitado.



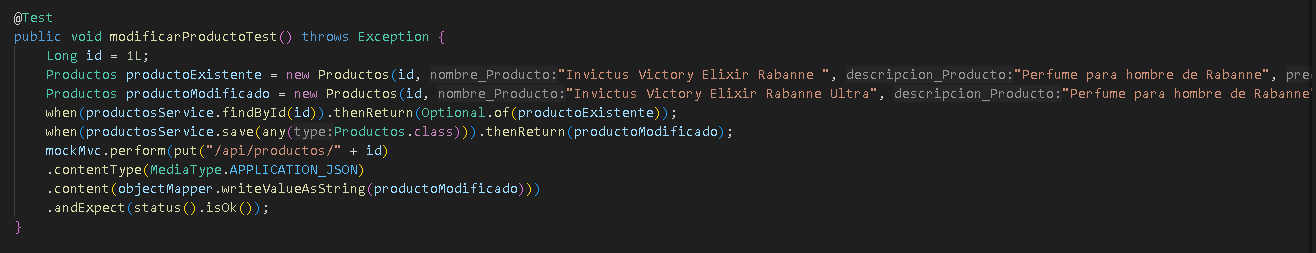
### **crearProductoTest():** Esta prueba simula la creación de un nuevo producto usando POST /api/productos.

* Se define un producto a crear.
* Se simula la respuesta del servicio usando when(...).thenReturn(...).
* La petición POST incluye el cuerpo en formato JSON usando objectMapper.
* Se valida que la respuesta tenga el código **201 Created**, indicando que el producto fue creado exitosamente.



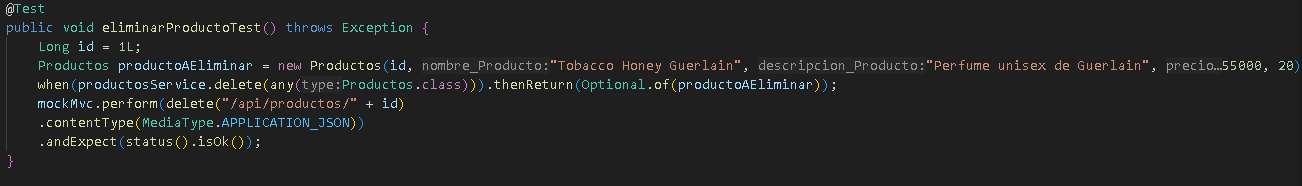
### **modificarProductoTest()**:Evalúa el correcto funcionamiento del endpoint PUT /api/productos/{id} para modificar un producto existente:

* Se simula que el producto con ID 1 existe.
* Se envía una petición con los nuevos datos.
* Se valida que el servicio guarde los cambios y que la respuesta tenga código 200 OK.



### **eliminarProductoTest():** Comprueba que se pueda eliminar correctamente un producto mediante DELETE /api/productos/{id}:

* Se simula que el producto a eliminar existe.
* Al realizar la petición DELETE, se espera recibir un código **200 OK** como confirmación de que fue eliminado correctamente.



# **5. Ejecución de pruebas**

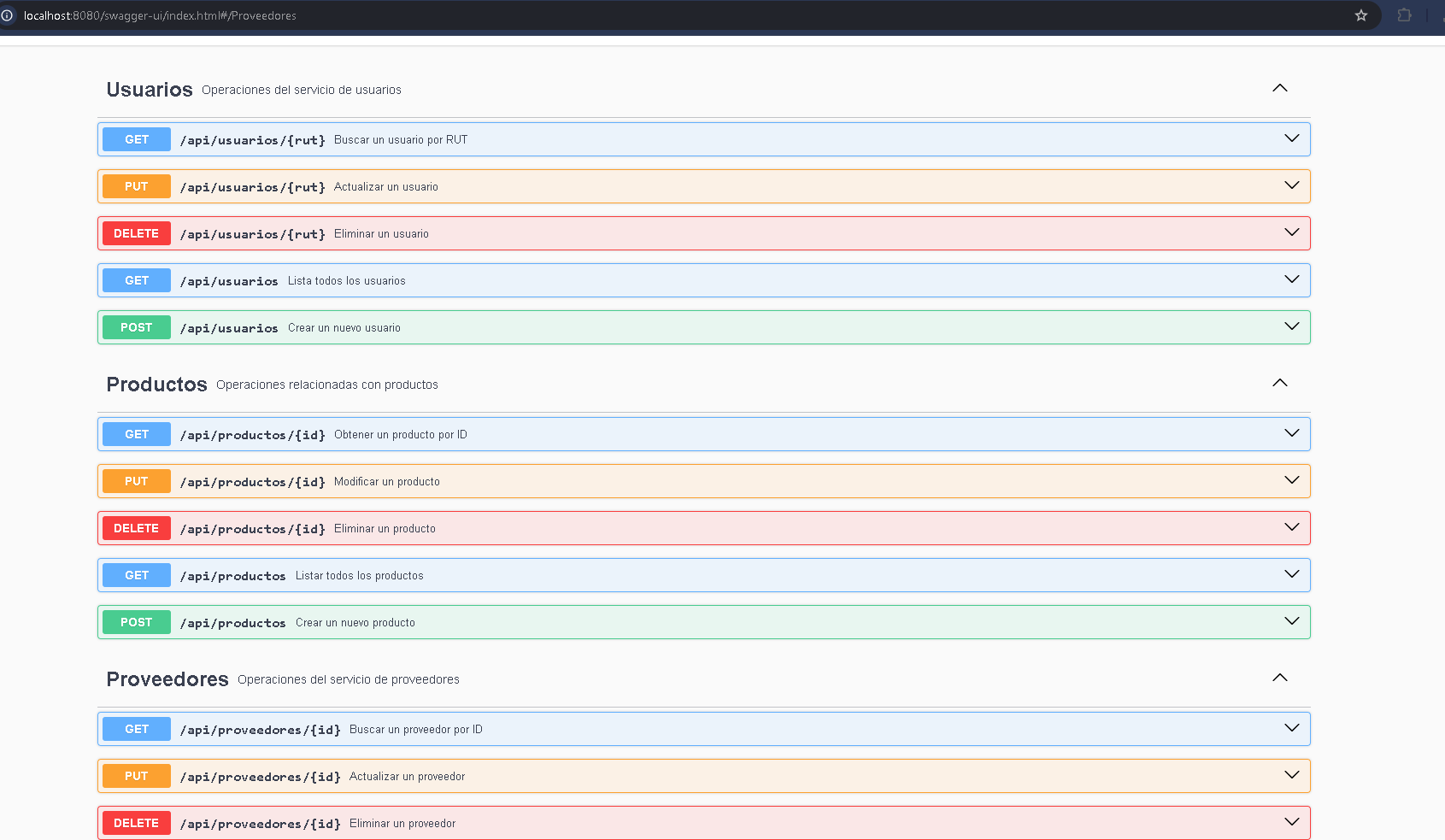
.

5.1 **Documentación de las pruebas ejecutadas**

Para facilitar la documentación, visualización y ejecución de pruebas manuales sobre los endpoints REST del sistema, se integró **Swagger UI**, una implementación de la especificación **OpenAPI**. Esta herramienta permite:

* Explorar todos los endpoints disponibles.
* Ver los métodos HTTP permitidos por cada recurso.
* Probar peticiones directamente desde el navegador sin necesidad de usar Postman.
* Observar las respuestas estructuradas (JSON) y los posibles códigos de estado.

Swagger fue accedido a través de la URL:  
<http://localhost:8080/swagger-ui.html>



El controlador ProductosRestController expone operaciones REST relacionadas con los productos del sistema. Está documentado utilizando **Swagger/OpenAPI** para describir cada endpoint y sus posibles respuestas, y además incorpora **HATEOAS** para enriquecer las respuestas con enlaces de navegación.

### **Anotaciones clave utilizadas**

* @RestController: Define esta clase como un controlador REST en Spring.
* @RequestMapping("api/productos"): Establece el prefijo de las rutas (/api/productos).
* @Autowired: Inyecta el servicio ProductosServices que contiene la lógica de negocio.

Se utilizaron las siguientes anotaciones para documentar la API:

* @Tag: Define una etiqueta de agrupación para Swagger (en este caso: **Productos**).
* @Operation: Describe de forma resumida y detallada lo que hace cada endpoint.
* @ApiResponse y @ApiResponses: Documentan los posibles códigos de respuesta (200, 404, 201, etc.).
* @ExampleObject: Muestra ejemplos en formato JSON que Swagger despliega automáticamente.

**Captura recomendada:** Mostrar Swagger UI con los endpoints expandidos y ejemplos visibles.

### 

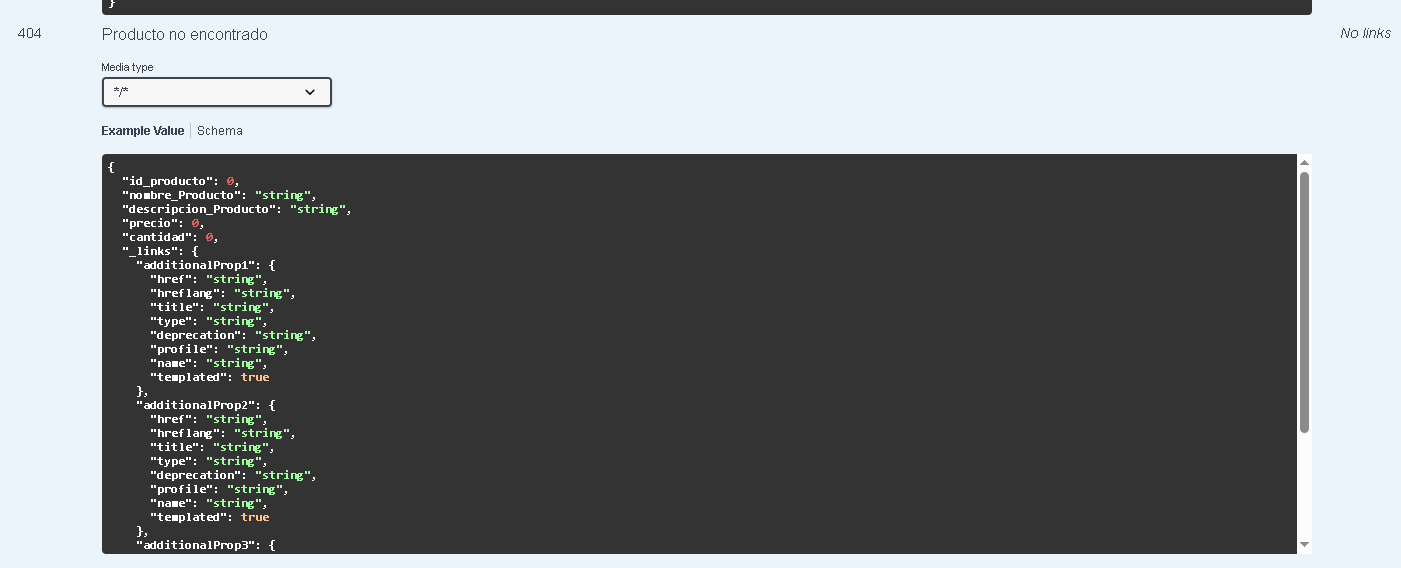
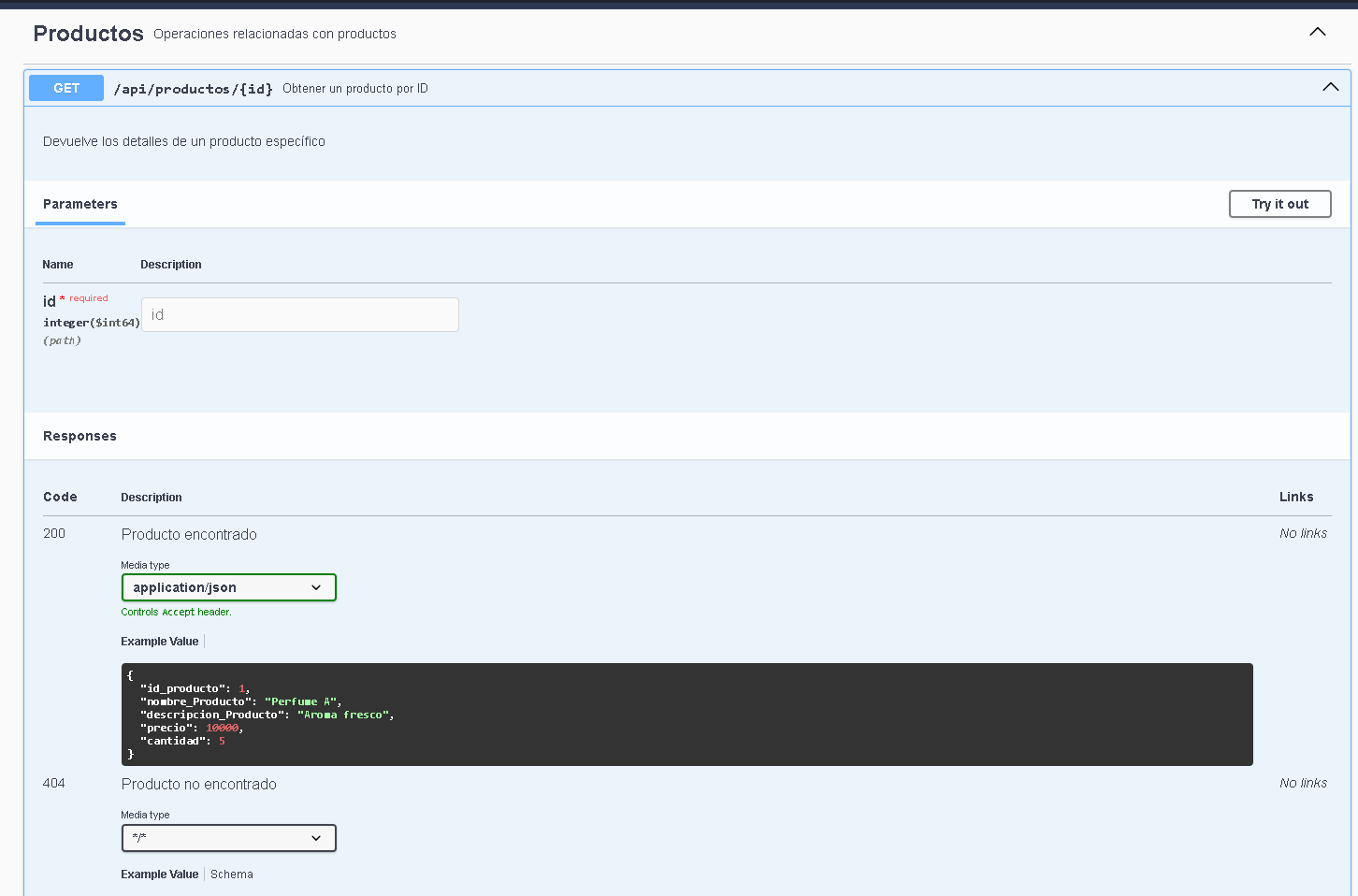
### **2. Funcionalidad de los Endpoints**

#### **GET /api/productos**

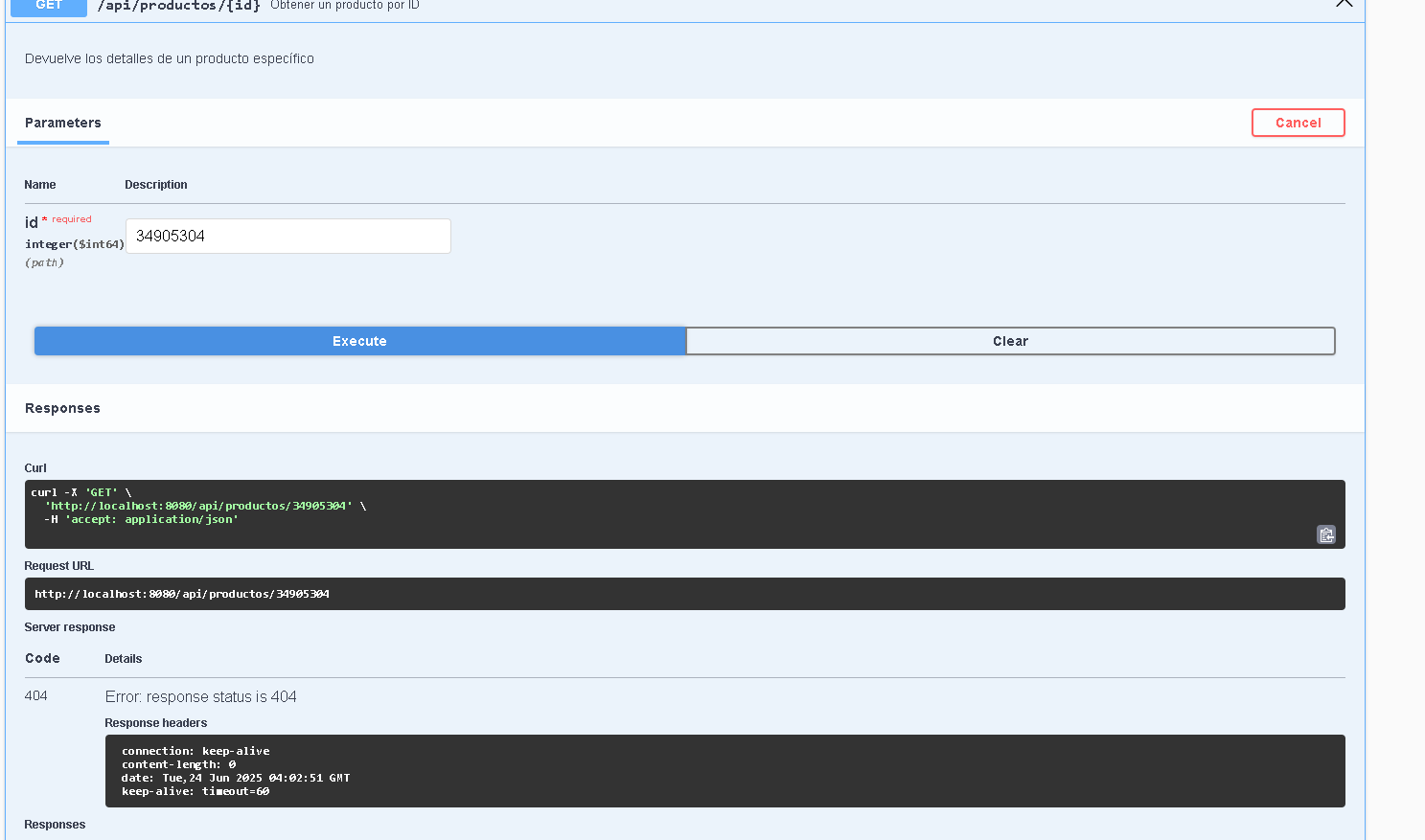
* **Método:** listar()
* **Descripción:** Devuelve una lista con todos los productos registrados.
* **Respuesta esperada:** HTTP 200 OK con un array de productos en JSON.

#### **GET /api/productos/{id}**

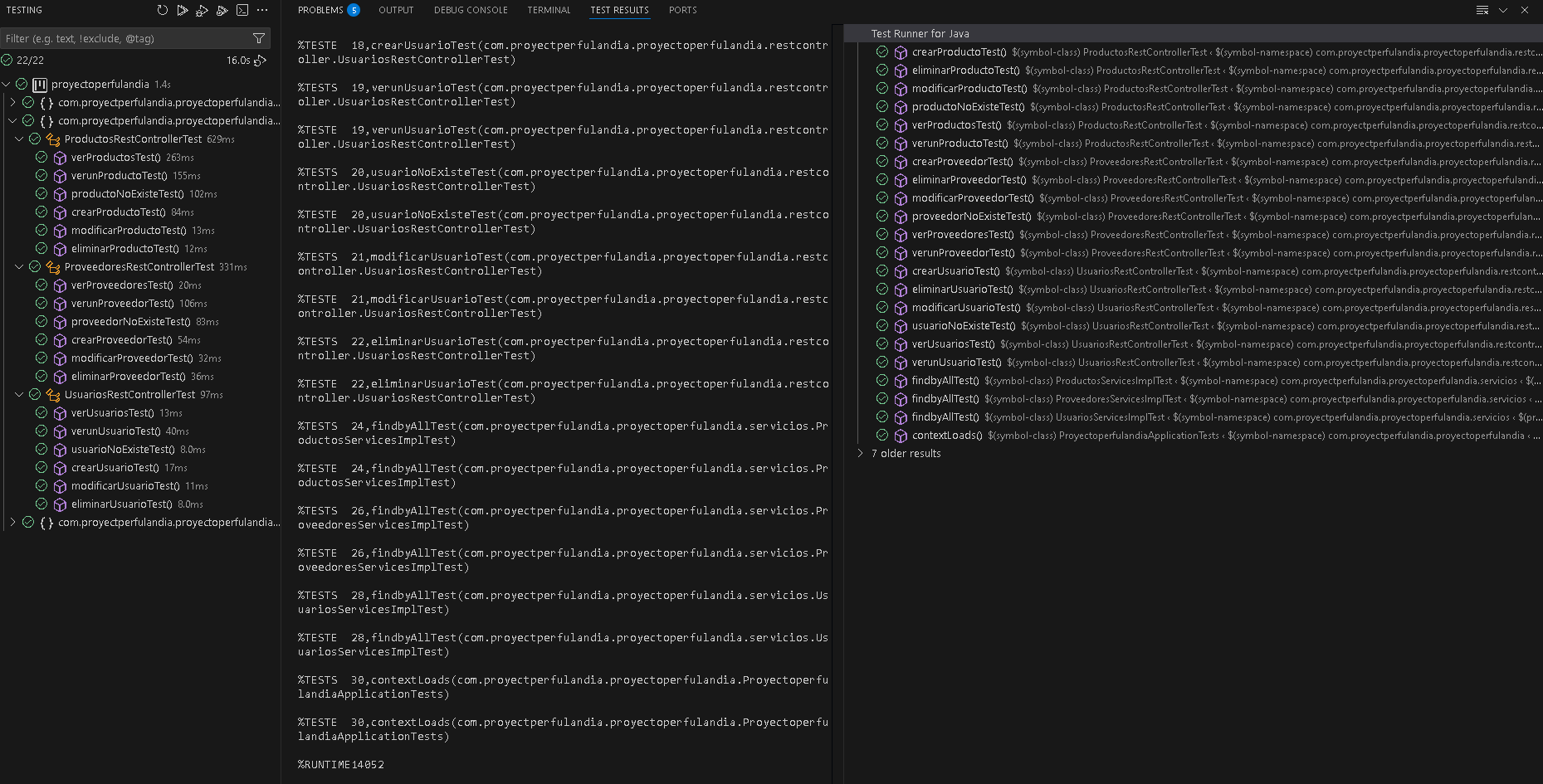
* **Método:** verDetalle(Long id)
* **Descripción:** Busca un producto por ID. Si existe, lo devuelve envuelto en un objeto **HATEOAS**.
* **HATEOAS agregado:**
  + self: enlace a sí mismo.
  + todos\_productos: enlace para volver a la lista general.
  + eliminar\_producto: enlace para eliminar ese producto.

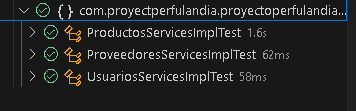


**Prueba ejecutada (404 no existe):**

****

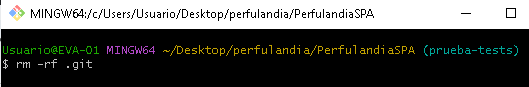
**5.2** **Ejecución de pruebas**



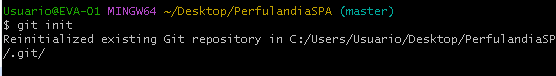


# **6. Git-GitHub**

primero un comando para limpiar los rastros del repositorio antiguo



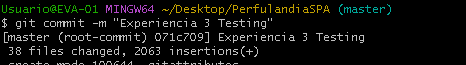
inicializamos el nuevo repositorio en la carpeta del proyecto para versionar los archivos localmente



se añaden los archivos



guarda los archivos en el repositorio local

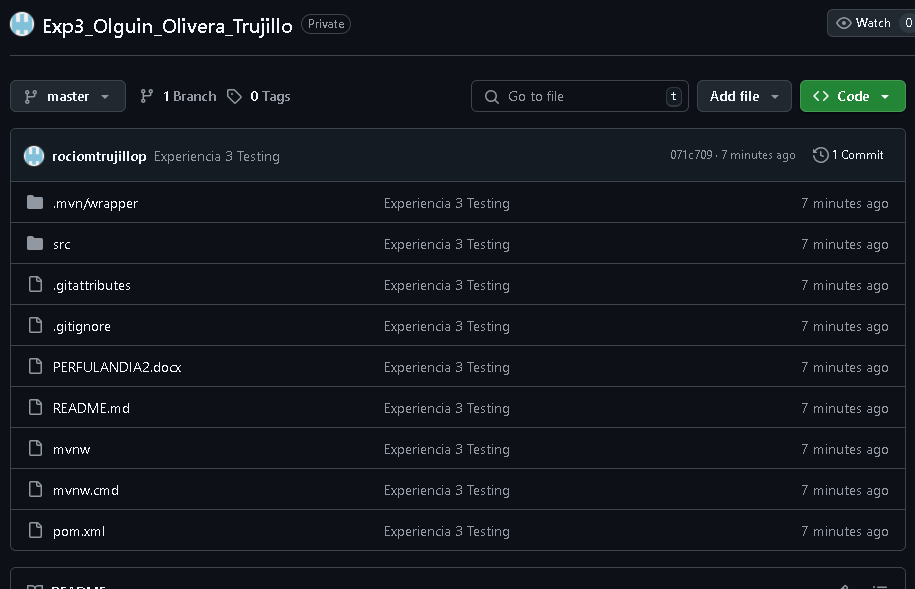


se conecta con el repositorio anteriormente creado



y se suben los archivos





ahora para dividir las experiencias en carpetas

# 

# **7. Conclusión**

Este proyecto representó un gran desafío técnico y de equipo que superamos con dedicación y aprendizaje continuo. Desde corregir diagramas UML hasta implementar una arquitectura de microservicios funcional, cada etapa nos dejó valiosas lecciones. Aprendimos que la atención a los detalles marca la diferencia, que las pruebas rigurosas son indispensables, y que herramientas como Trello y Miro son clave para la organización. Los errores iniciales se convirtieron en oportunidades para mejorar, demostrando nuestra capacidad de adaptación y trabajo colaborativo.

Lo Que Realmente Logramos:

1. De los Errores a las Soluciones:

Recuerdo cuando nos devolvieron los primeros diagramas por estar incompletos. Nos sentamos toda una tarde con pizzas y café para entender bien UML. Hoy esos mismos diagramas son nuestro orgullo, claros y profesionales.

Lo del ID autoincremental fue una lección dura. Aprendimos que hasta los pequeños detalles importan, y ahora el RUT como identificador funciona perfectamente.

1. Trabajo en Equipo que Dio Resultados:

Las tardes de pair programming con VSCode Live Share nos unieron como equipo. ¡Cuántos bugs resolvimos juntos!

1. Lo Que Aprendimos en el Proceso:

Que la documentación no es un trámite, sino un salvavidas para cuando algo falla.

Que discutir un diagrama en Miro puede ahorrar días de desarrollo.

Que equivocarse es parte del proceso, lo importante es mejorar.

Para el Futuro:

Queremos implementar Kubernetes y seguir mejorando. Pero hoy, estamos orgullosos de lo logrado. Este proyecto nos demostró que cuando trabajamos unidos y con pasión, los resultados llegan.