

Informe N°3

Implementación de Microservicios PerfuSmart

**Grupo 1**

Gonzalo Navarrete

Carla Prado

Fernando Zárate

# Índice

[**Índice 2**](#_heading=h.qpjh05lfj5go)

[**1. Diagrama de arquitectura de microservicios 3**](#_heading=h.tqw5yodplef7)

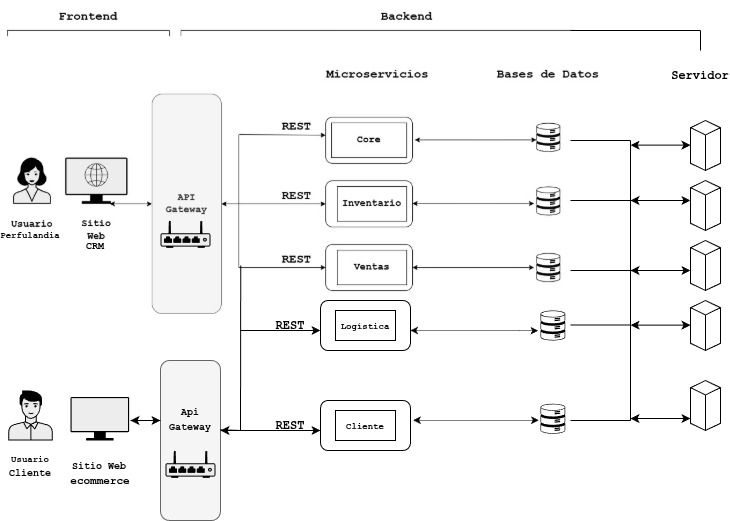
[**2. Plan de pruebas 6**](#_heading=h.y90ynbd8umvm)

[**3. Ejecución de pruebas 22**](#_heading=h.uewm8aux54zg)

[**4. Git-GitHub 29**](#_heading=h.1q731jtme61t)

# Diagrama de arquitectura de microservicios

1. Diagrama de despliegue de arquitectura de microservicios.



**API Gateway:** Es una puerta de entrada única (un punto centralizado) para todas las llamadas a las APIs de un sistema. En vez de que un cliente (por ejemplo, una app móvil o web) llame directamente a cada microservicio, todas las solicitudes pasan primero por la API Gateway.

**Funcionalidades:**

Enrutamiento: Redirige las peticiones entrantes a los microservicios correspondientes según la URL, método HTTP, o reglas configuradas.

Autenticación y Autorización: Controla quién puede acceder a qué recursos, validando tokens o credenciales antes de pasar la petición.

Agregación de respuestas: Puede combinar datos de varios microservicios y devolver una única respuesta al cliente.

Transformación: Modifica las solicitudes o respuestas, como cambiar formatos (por ejemplo, XML a JSON), agregar o eliminar datos.

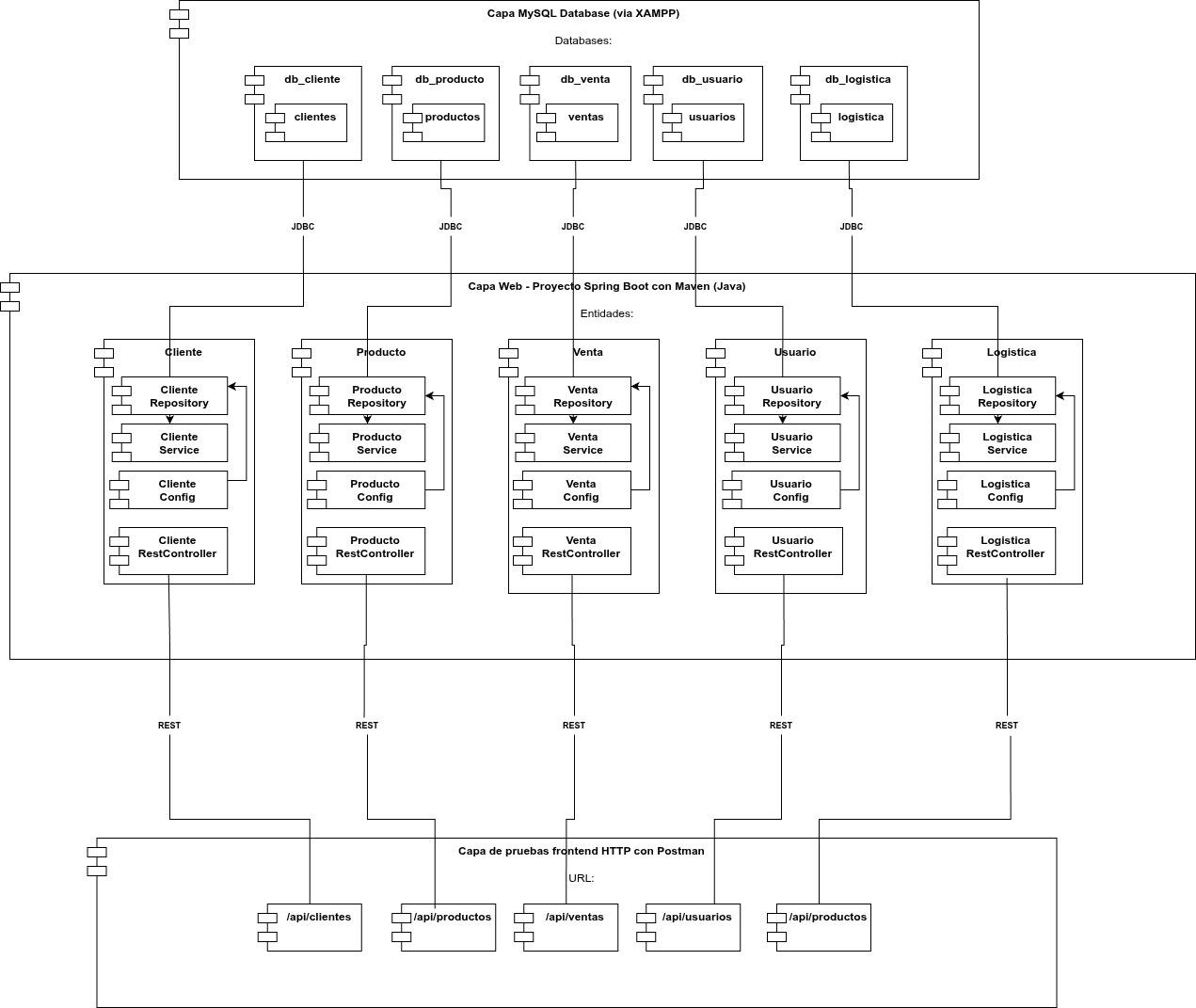
Limitación y control de tráfico (rate limiting): Evita abusos limitando la cantidad de solicitudes de un usuario o cliente en un período.

Registro y monitoreo: Centraliza el logging de peticiones para facilitar el seguimiento y diagnóstico.

Manejo de errores: Controla errores y los presenta de forma consistente al cliente.

Caching: Puede almacenar respuestas frecuentes para mejorar la velocidad y reducir carga a los microservicios.

1. Diagrama de componentes con capa Spring Boot, capa Base de Datos y capa pruebas frontend con Postman (Trabajo realizado en Experiencia anterior).



**Clase Config:**

La clase Config en un proyecto Spring Boot sirve para configurar manualmente el acceso a la base de datos, especialmente cuando se necesita controlar detalles avanzados. Se usa cuando no basta la configuración automática de Spring Boot o cuando quieres definir varias conexiones de base de datos.

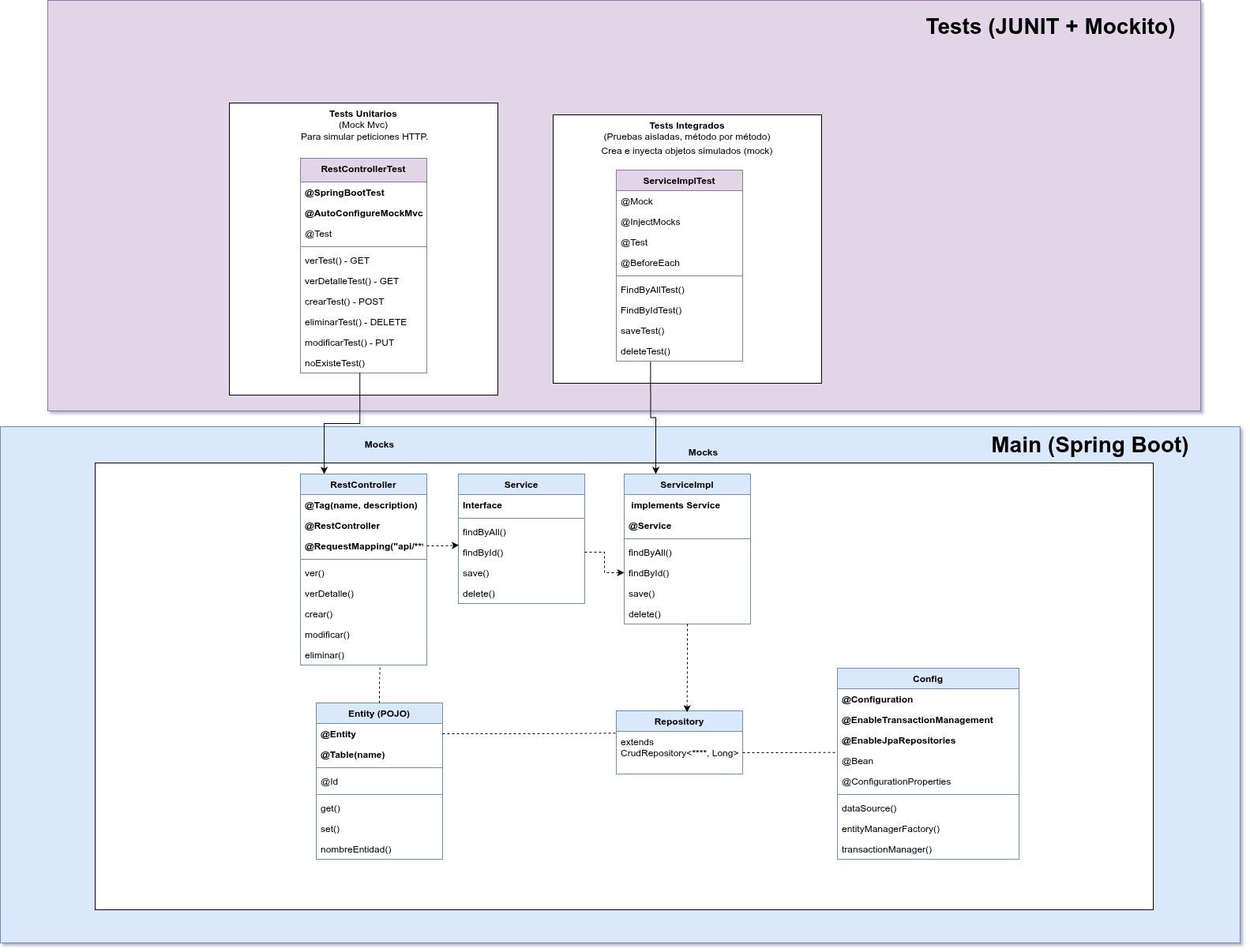
Cuando se trabaja con más de una base de datos, se necesita:

* Definir múltiples DataSource (Origen de datos, una para cada base de datos).
* Separar entidades y repositorios por base de datos.
* Asignar transacciones a la conexión correcta.
* Separar repositorios por paquete, para que Spring sepa qué repositorio usar con qué conexión.

Métodos relevantes:

* DataSource(): Múltiples, para conectar varias bases de datos
* EntityManagerFactory(): para separar el manejo de entidades entre bases
* TransactionManager(): Para que cada base tenga su propia gestión de transacciones

# Plan de pruebas



Para asegurar la correcta funcionalidad de los servicios y controladores REST en este proyecto, se implementaron pruebas unitarias y de integración utilizando herramientas y frameworks especializados.

1. **Frameworks de pruebas**
2. **JUnit 5**

Framework para escribir y ejecutar pruebas. Verifica que los métodos devuelvan lo esperado.

Se utilizó JUnit 5 como framework principal de pruebas. Este permitió definir métodos de prueba estructurados, aserciones claras (assertEquals, assertTrue, assertNotNull, etc.) y establecer rutinas de preparación con @BeforeEach. JUnit también facilitó la integración con entornos de desarrollo (IDE) y la automatización de pruebas.

1. **Mockito**

Framework que permite simular objetos y su comportamiento. Se usa para aislar dependencias (como base de datos).

Se usó Mockito para crear objetos simulados (mocks) de dependencias como LogisticaRepository y LogisticaServiceImpl. Esto permitió realizar pruebas unitarias de manera aislada, evitando la necesidad de una base de datos real y asegurando que cada componente se pruebe de forma independiente.

1. **Pruebas desarrolladas**
2. **Unitarias**

Las pruebas unitarias son aquellas que se enfocan en **verificar el comportamiento aislado de una unidad de código**, normalmente una función o método dentro de una clase. Su objetivo es asegurarse de que cada parte del sistema funciona correctamente de forma independiente, sin depender de otros componentes del sistema, como bases de datos, servidores web o clases externas.

En estas pruebas, se usan herramientas como Mockito para simular (mockear) dependencias externas, de modo que se controle el entorno y se validen únicamente los resultados del método en prueba.

Ejemplo: probar que el método save() de una clase de servicio devuelve un objeto correctamente guardado, simulando (mockeando) el repositorio para no acceder a una base real.

* Son rápidas de ejecutar.
* Fáciles de mantener si el código está bien estructurado.
* Ayudan a encontrar errores de lógica muy específicos.
* No dependen de la base de datos ni del contexto completo de Spring.

1. **De Integración**

Las pruebas de integración validan que **varias partes** del sistema **trabajen correctamente** **juntas**. A diferencia de las pruebas unitarias, estas no aíslan los componentes, sino que verifican que, por ejemplo, un controlador REST se comunique bien con un servicio, y este a su vez con el repositorio y la base de datos.

Se usan para detectar errores en la interacción entre componentes reales y en la configuración del sistema, como problemas con transacciones, persistencia, o conversiones de datos.

Ejemplo: realizar una llamada HTTP simulada (usando MockMvc) a un endpoint /clientes, y verificar que la respuesta sea correcta y los datos vengan de la base de datos esperada.

* Detectan errores de configuración o integración que las unitarias no ven.
* Simulan escenarios más cercanos al comportamiento real del sistema.
* Verifican que las capas (controlador, servicio, repositorio) estén correctamente conectadas.

1. **Tipos de clases de Prueba en este proyecto**
2. **ServiceImpl**Test

Esta clase se encarga de realizar pruebas unitarias sobre la lógica del servicio ServiceImpl. El objetivo es verificar que los métodos como findByAll(), findById(), save() y delete() funcionen correctamente de forma aislada, sin acceder a la base de datos ni al controlador.

Estas pruebas aseguran que la lógica interna del servicio responde correctamente a distintos escenarios, por ejemplo:

* Qué pasa si se encuentra o no un cliente por su ID.
* Si se guarda correctamente un nuevo cliente.
* Si se elimina un cliente solo si existe.

1. **RestController**Test

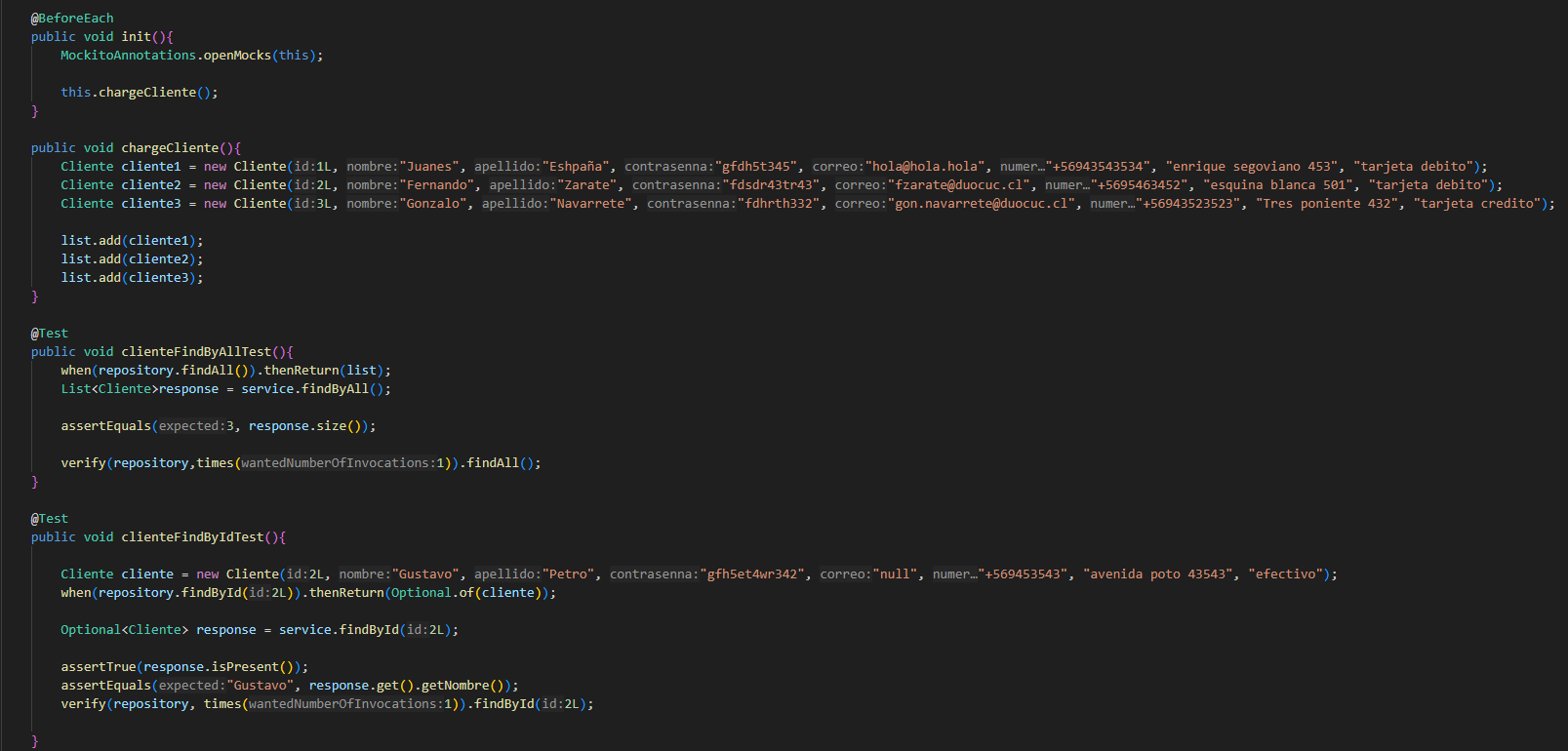
Esta clase se enfoca en probar el comportamiento del controlador RestController a través de peticiones simuladas HTTP. Aunque no se conecta a una base de datos real, sí prueba la integración entre el controlador y el servicio, usando MockMvc para simular llamadas reales como GET, POST, PUT y DELETE.

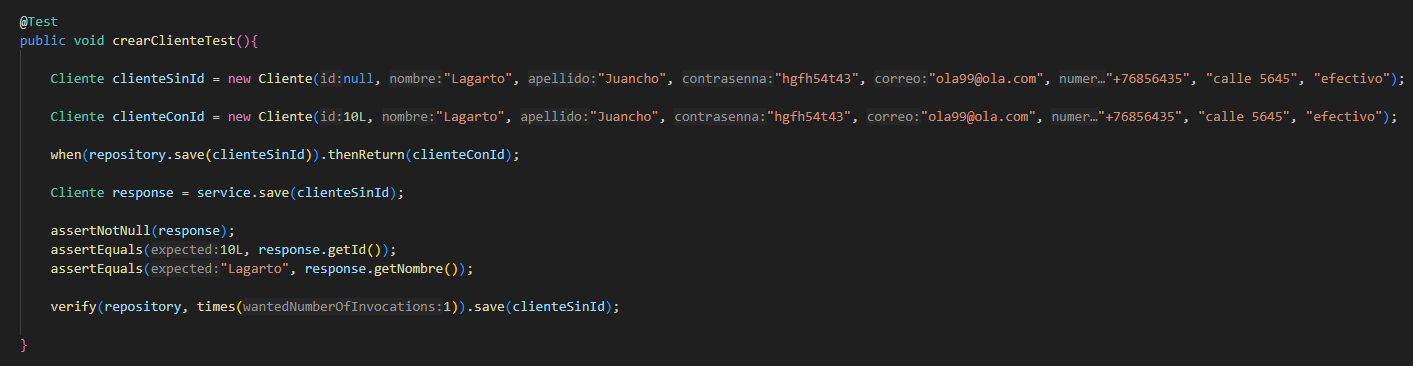
Este tipo de prueba comprueba que:

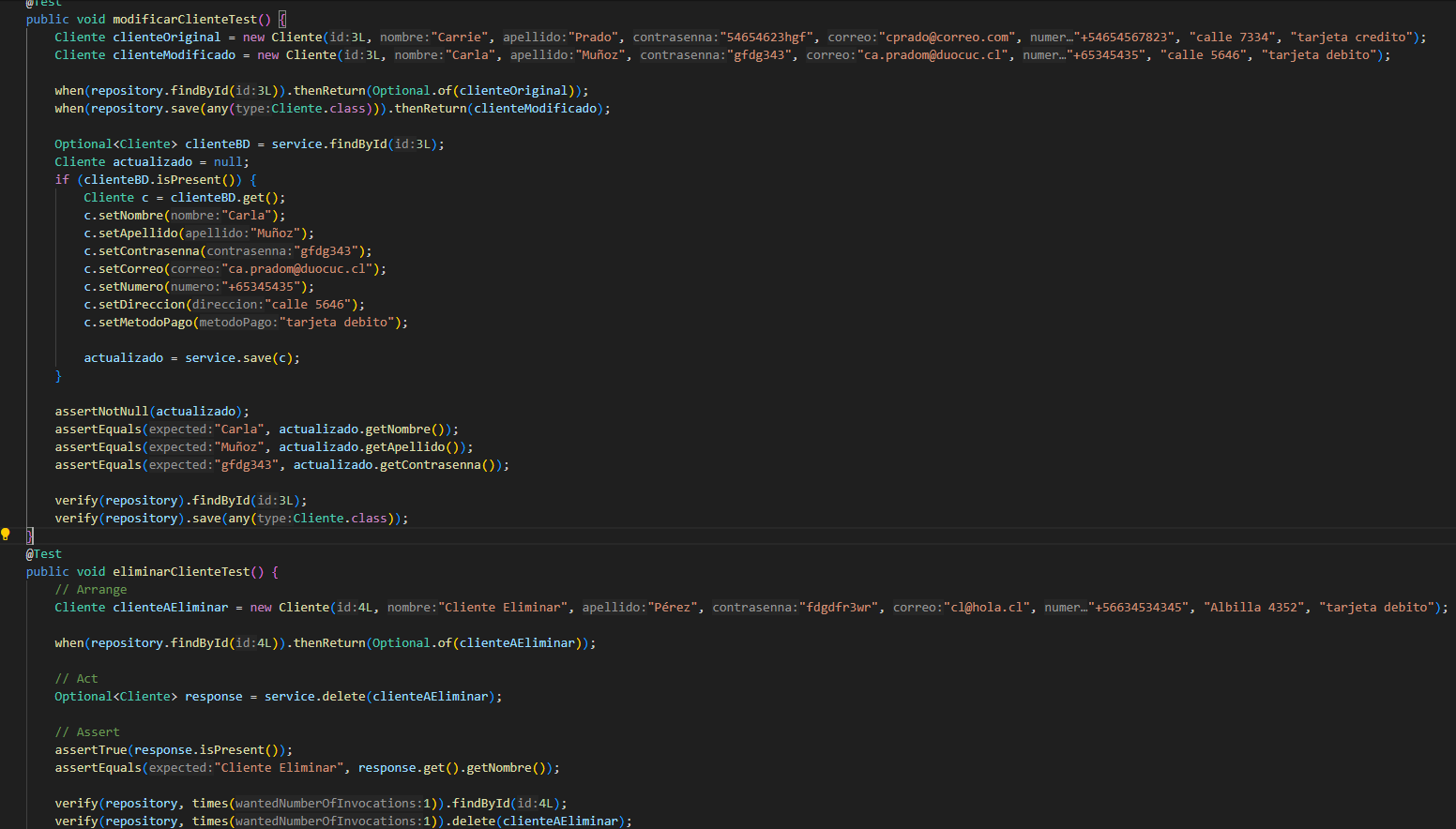
* Las rutas HTTP funcionan correctamente (/api/clientes, /api/clientes/{id}, etc.).
* El controlador responde con los códigos de estado esperados (200 OK, 201 Created, 404 Not Found, etc.).
* Los datos enviados o recibidos se manejan adecuadamente.

1. **Anotaciones principales**
2. **@SpringBootTest (Spring Boot):** Carga el contexto completo de la aplicación Spring Boot, incluyendo todas las configuraciones, beans, servicios, controladores y componentes necesarios. Se utiliza para pruebas de integración, ya que permite probar cómo interactúan los componentes en un entorno muy similar al real.
3. **@AutoConfigureMockMvc (Spring Boot):** Activa e inyecta automáticamente un objeto MockMvc, el cual permite simular peticiones HTTP (GET, POST, PUT, DELETE) sin necesidad de levantar un servidor web real. Sirve para probar controladores REST de forma realista y rápida, sin desplegar toda la aplicación. Más rápido que levantar un servidor real. Permite probar endpoints completos, encabezados, estados HTTP, etc.
4. **@Mock (Mockito):** Crea un objeto simulado (mock), es decir, una falsa implementación que no ejecuta la lógica real de la clase, pero puede comportarse según lo que se le configure. Se usa para reemplazar dependencias reales (como un repositorio o servicio) por versiones simuladas que permiten controlar el comportamiento del entorno durante una prueba unitaria. Evita usar bases de datos o conexiones reales. Permite probar una clase de forma aislada.
5. **@InjectMocks (Mockito):** Inyecta automáticamente los objetos simulados (@Mock) en la clase bajo prueba.
6. **@Test (JUnit):** Marca un método como prueba.
7. **@BeforeEach (JUnit):** Ejecuta un método antes de cada prueba.
8. **Pruebas unitarias**

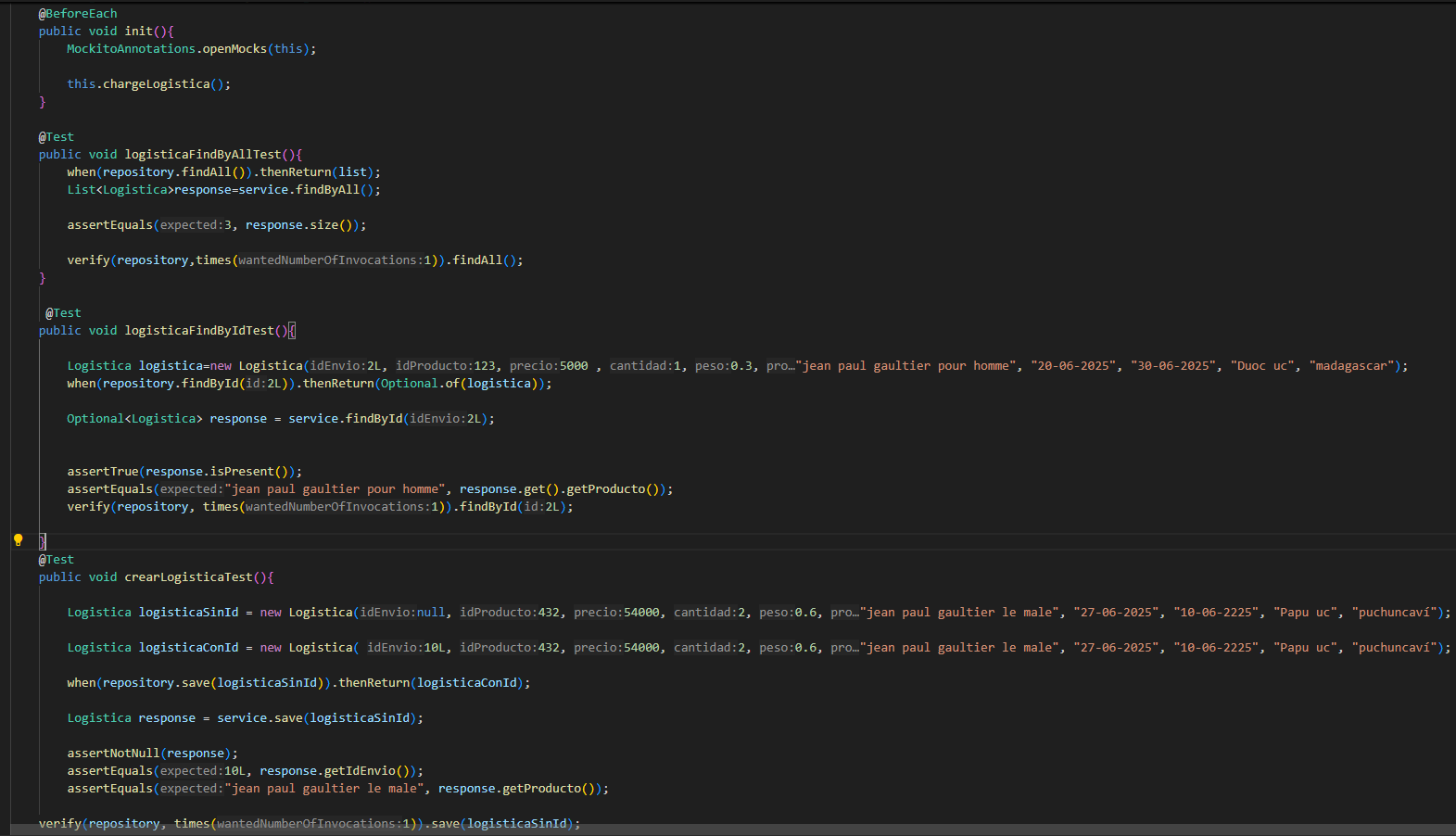
* ClienteServiceImplTest





****

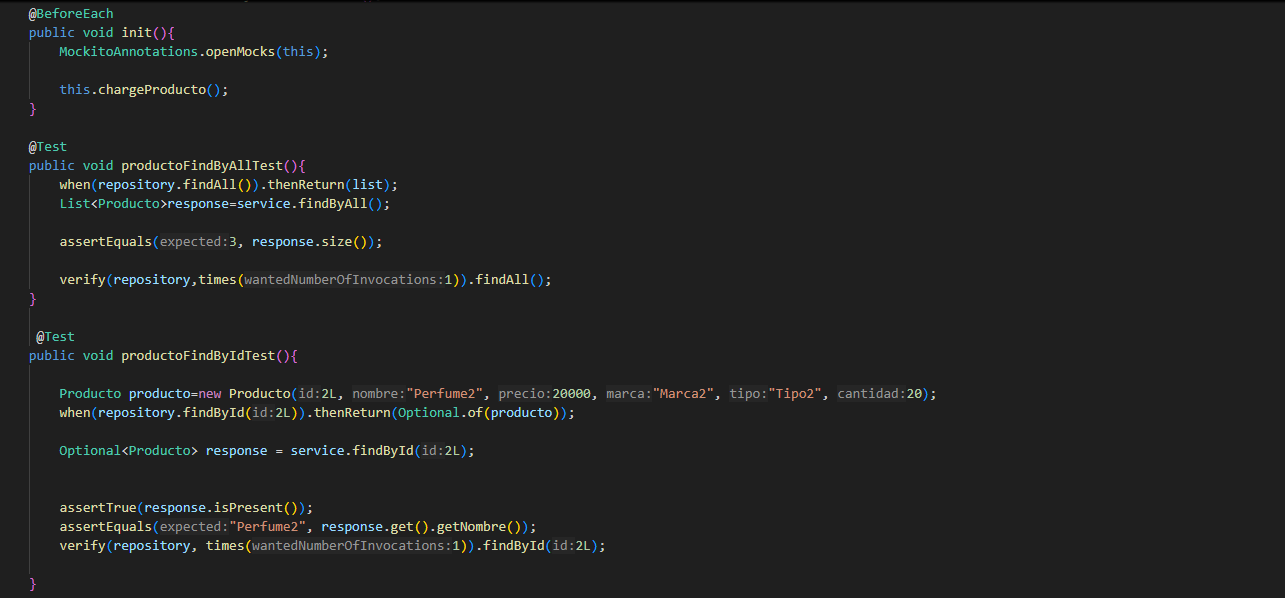
* LogisticaServiceImplTest

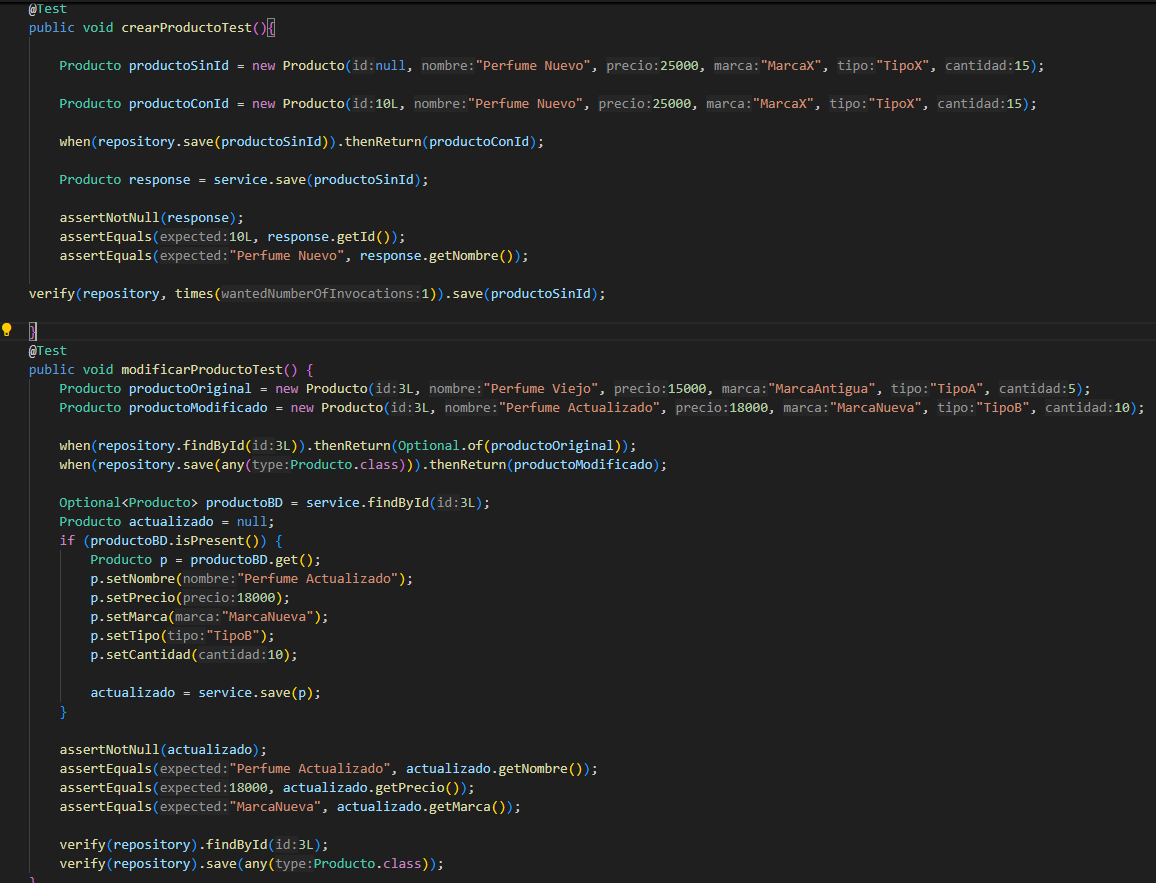






* ProductoServiceImplTest

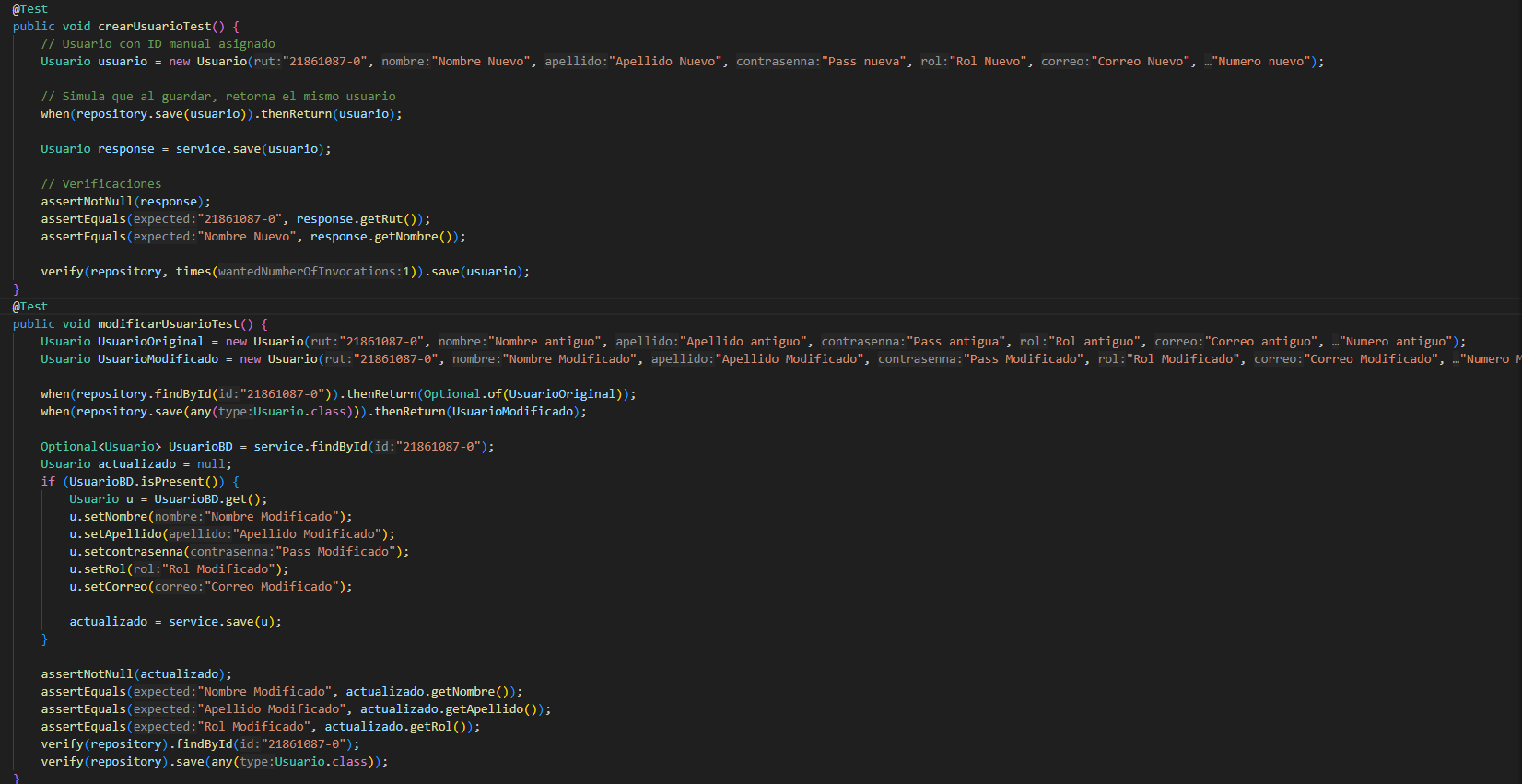






* UsuarioServiceImplTest

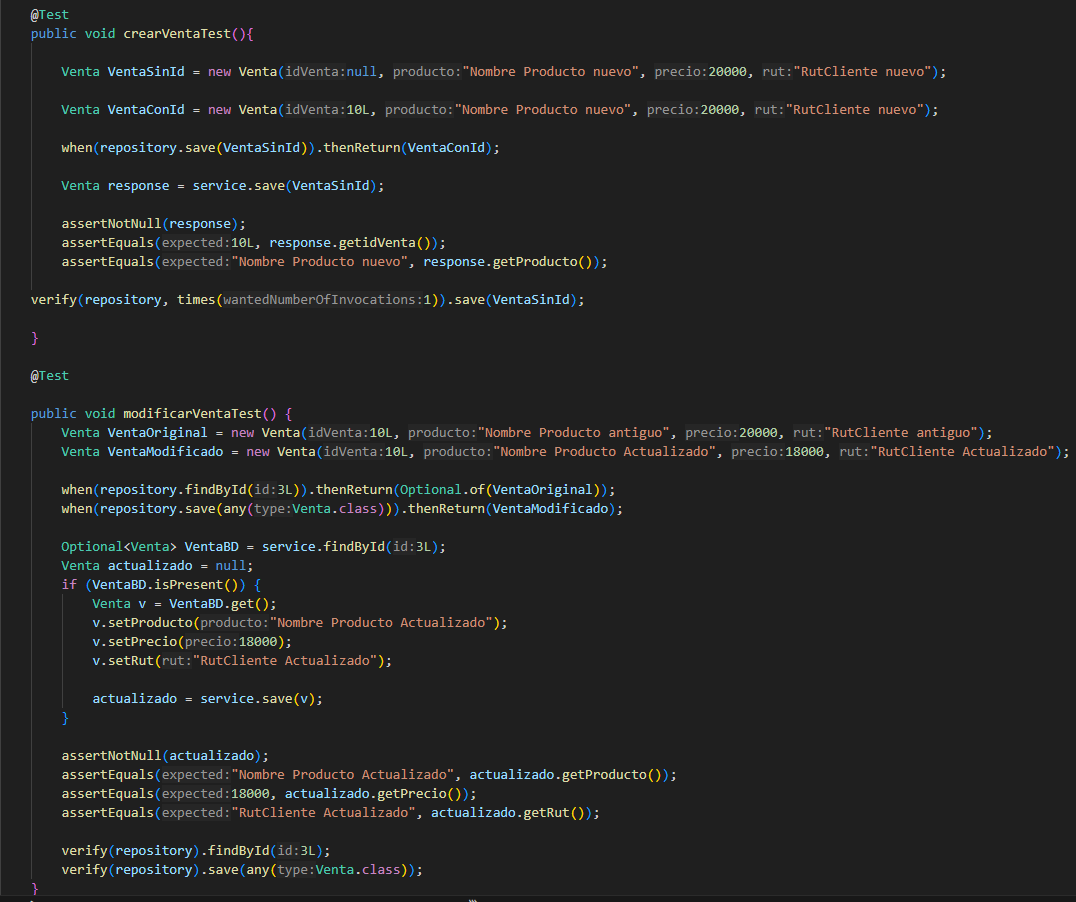






* VentaServiceImplTest



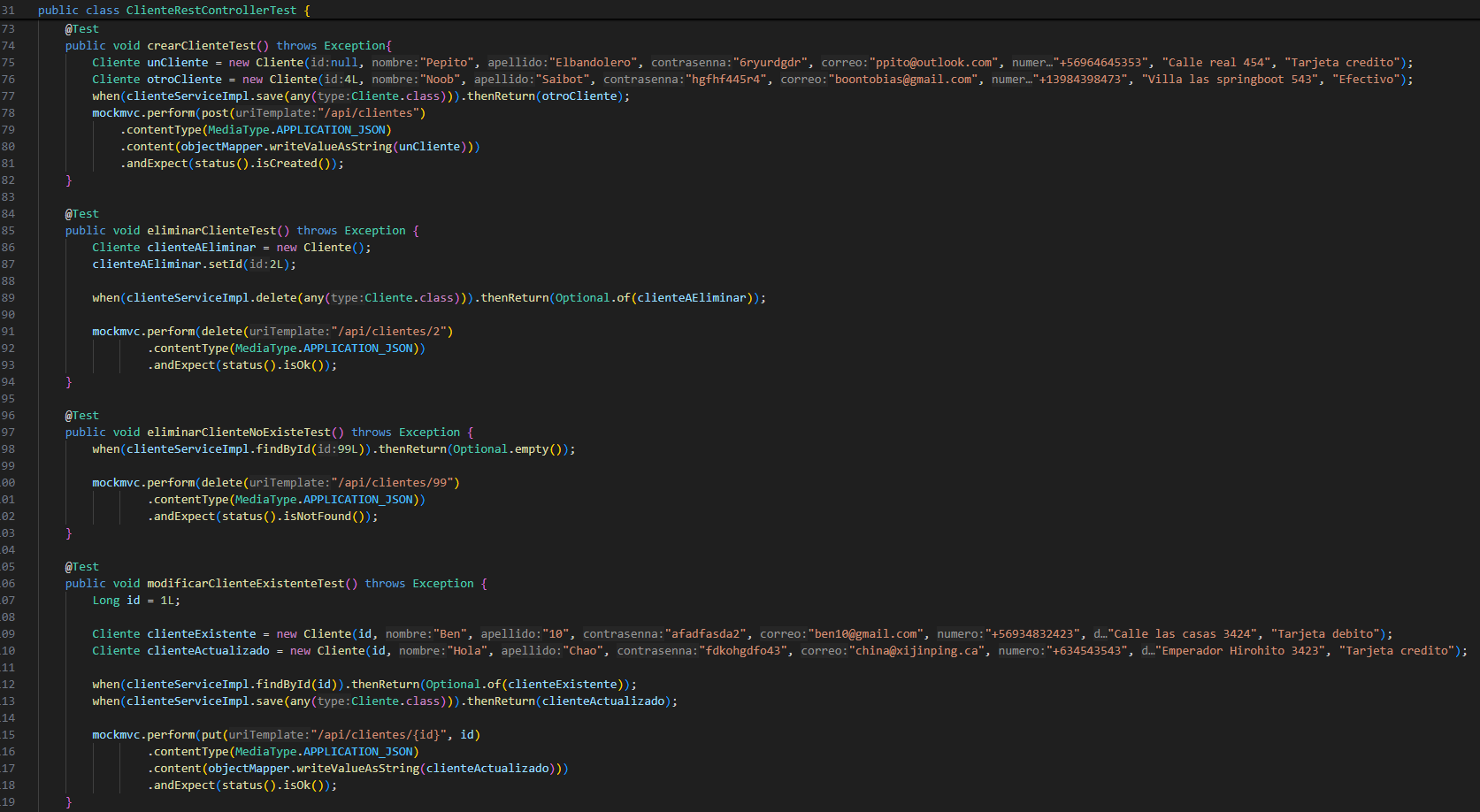




1. **Pruebas de integración**

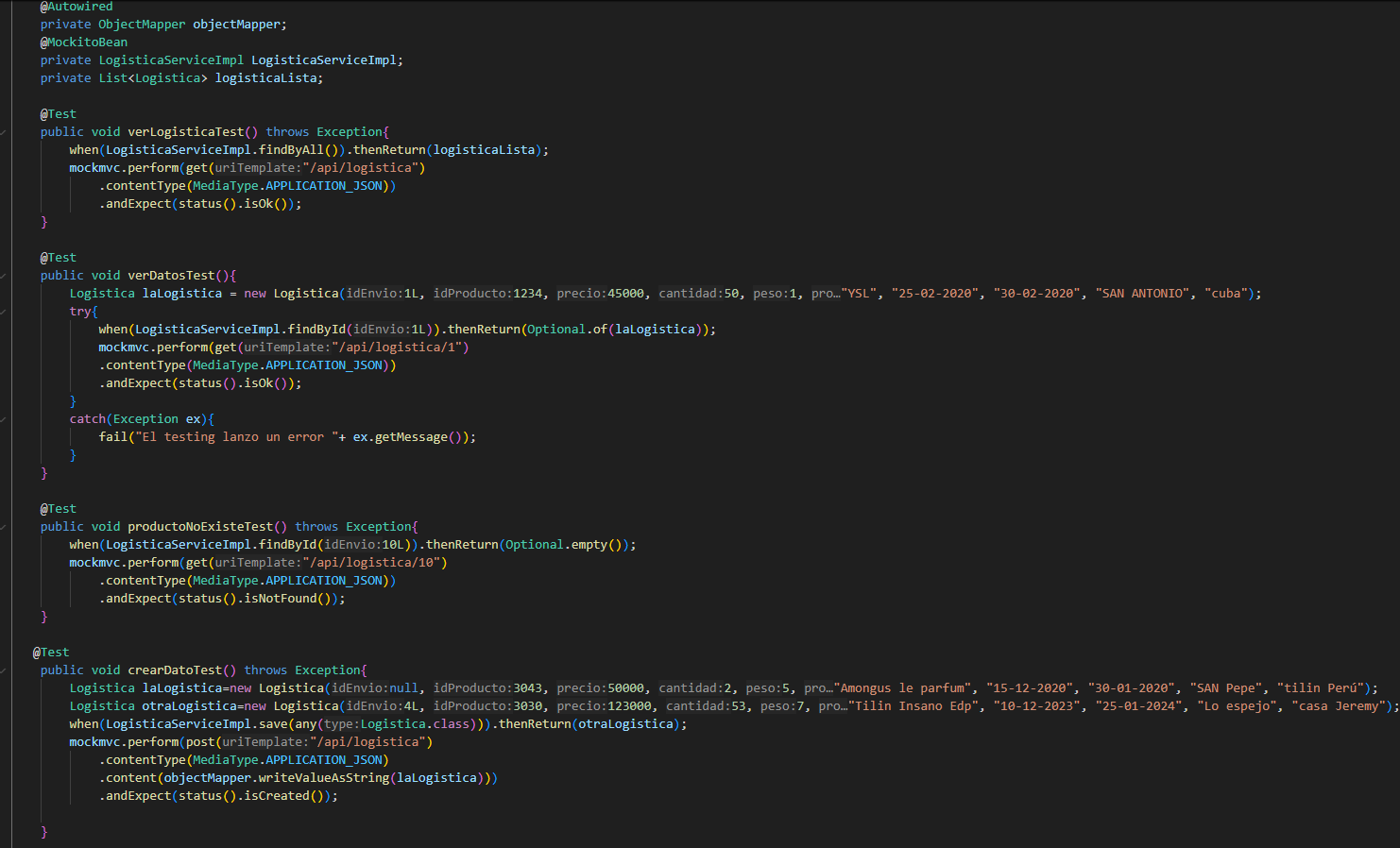
* ClienteRestControllerTest

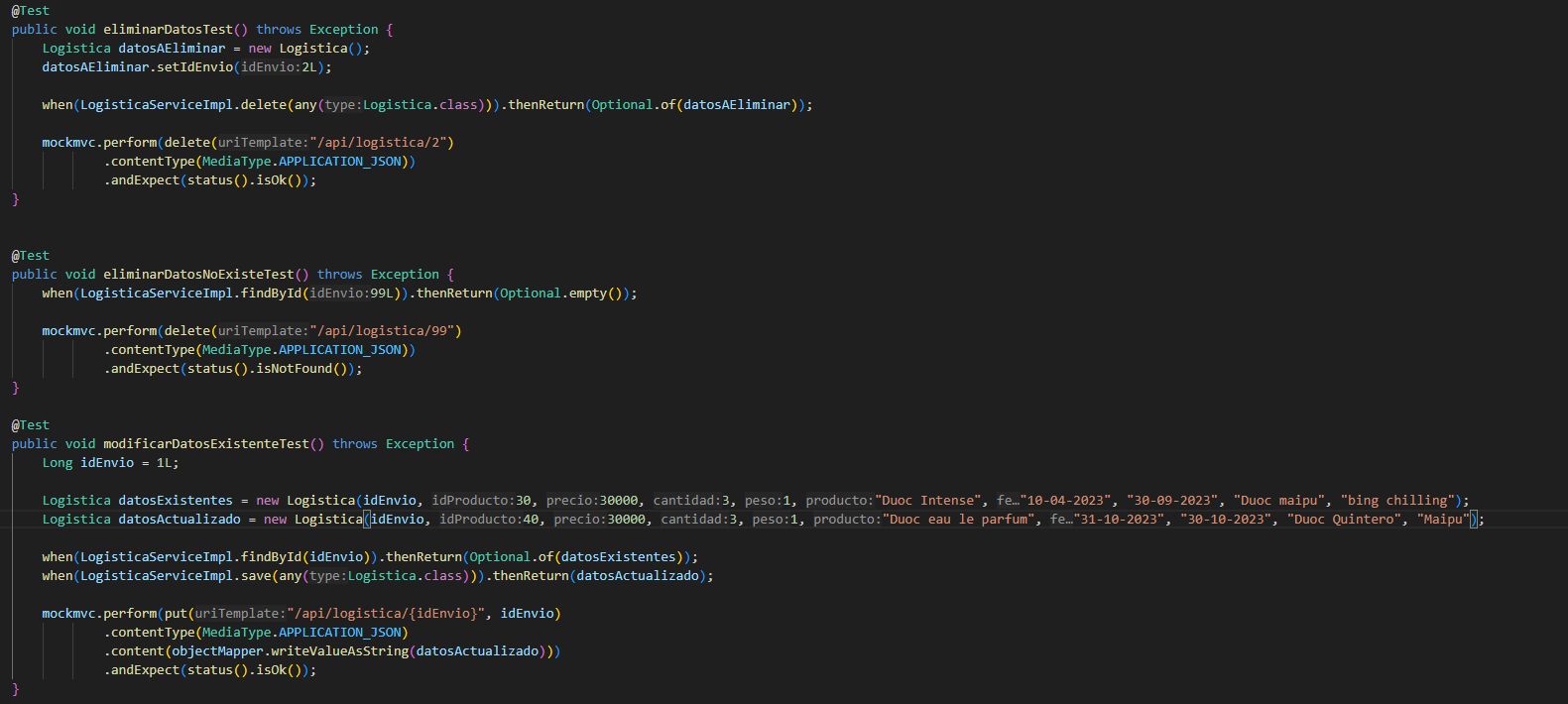






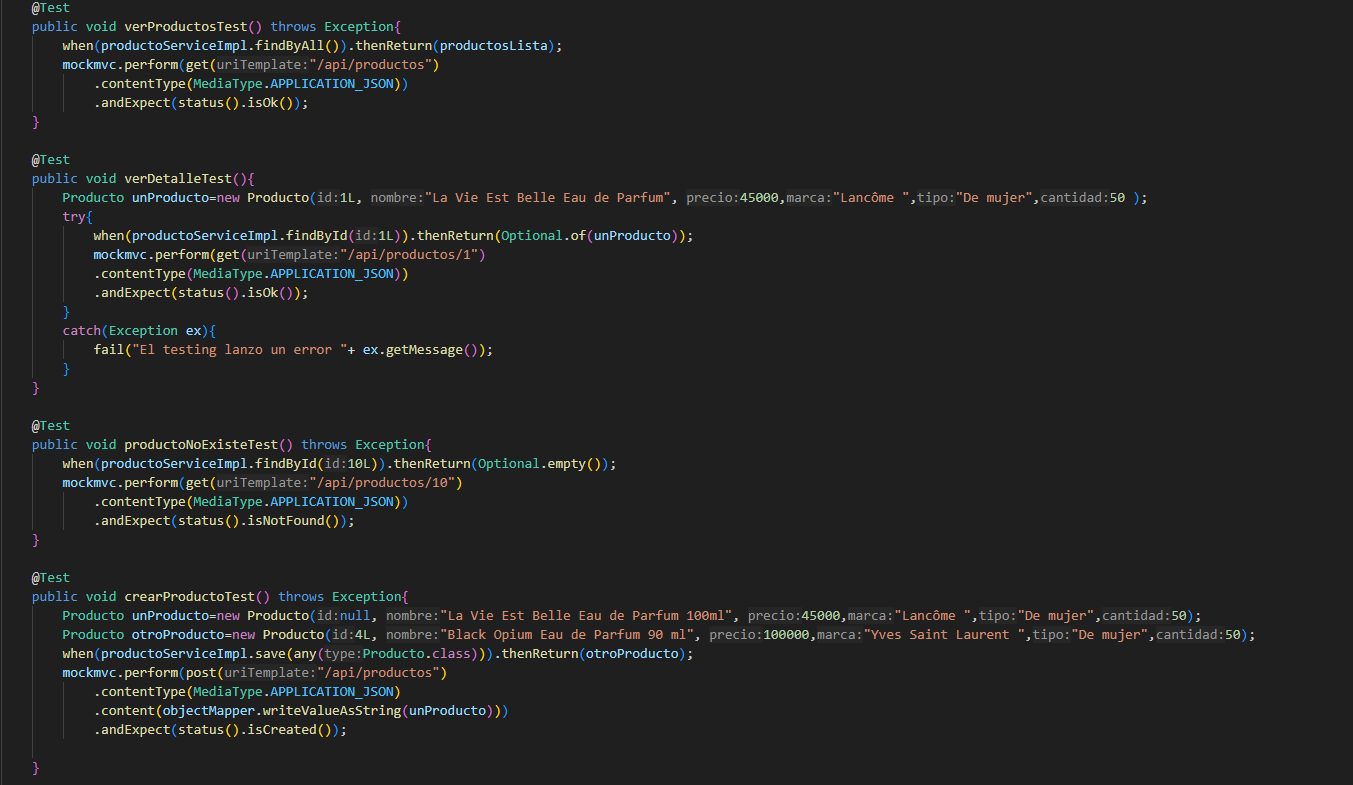
* LogisticaRestController



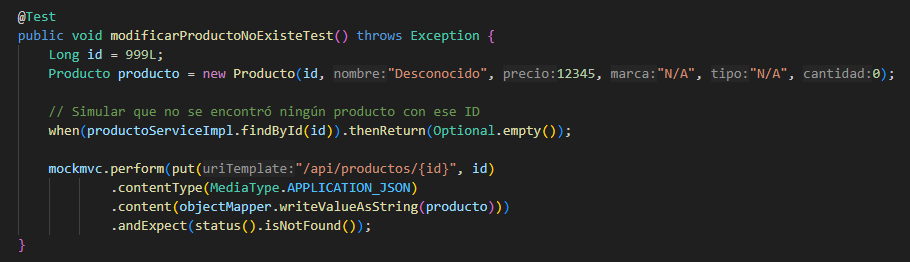




* ProductoRestControllerTest

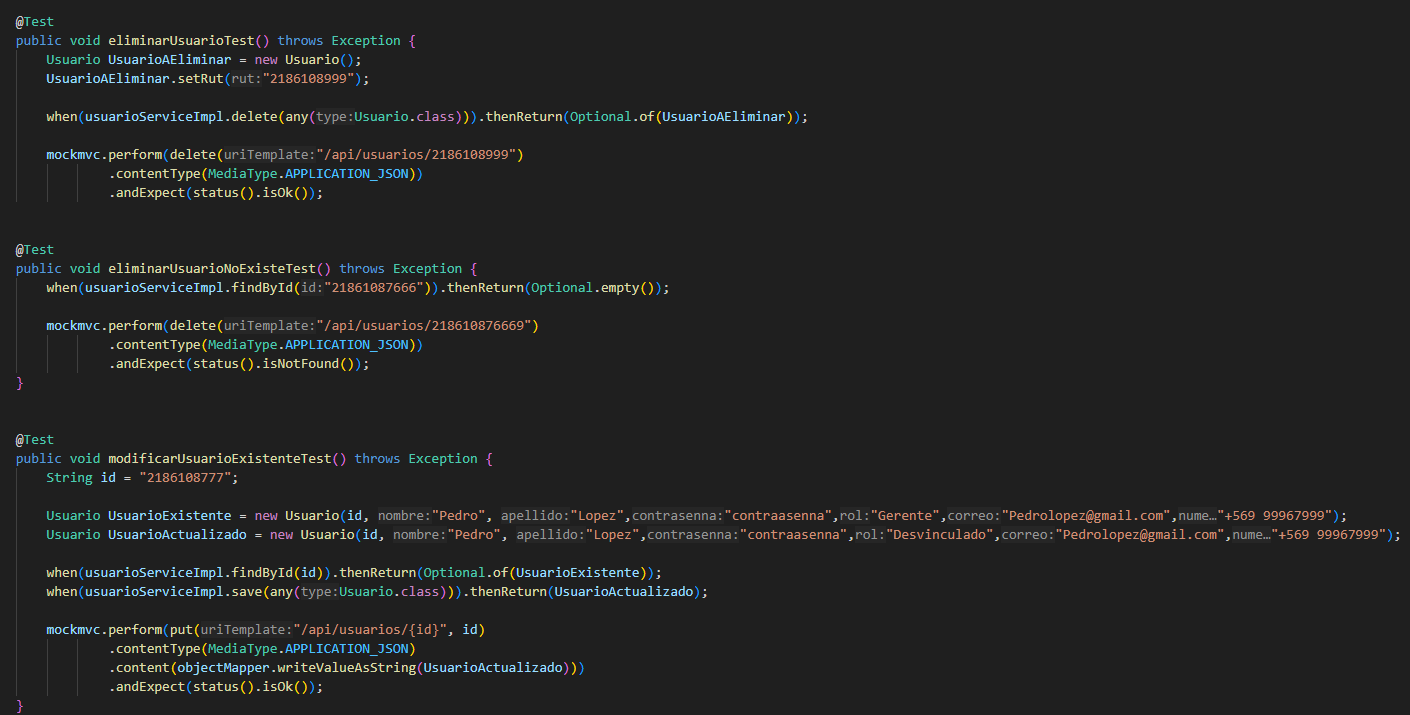


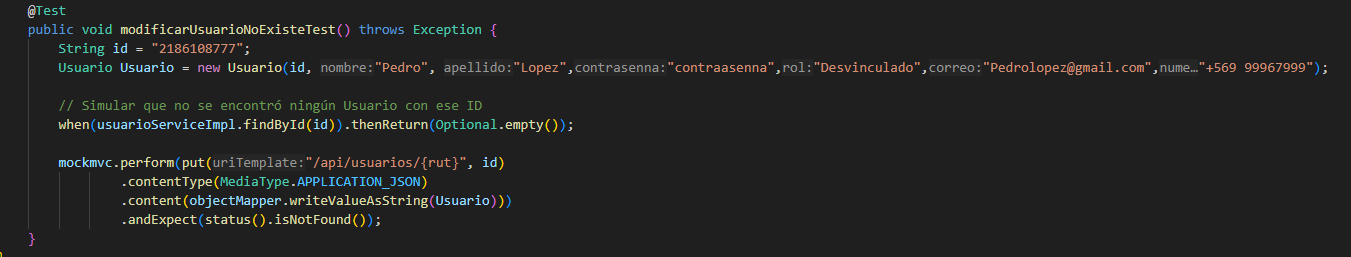




* UsuarioRestControllerTest



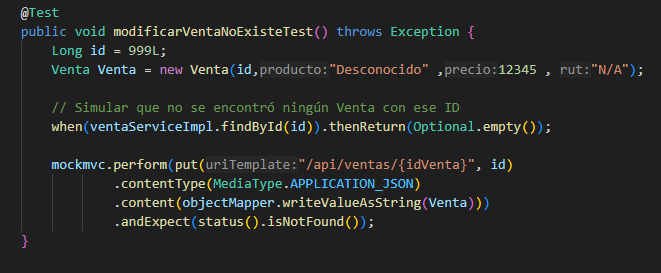




* VentaRestControllerTest



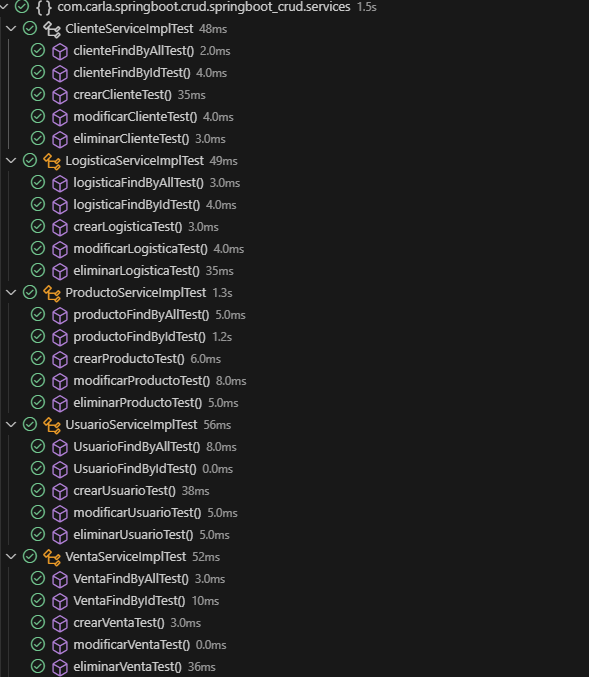




# Ejecución de pruebas

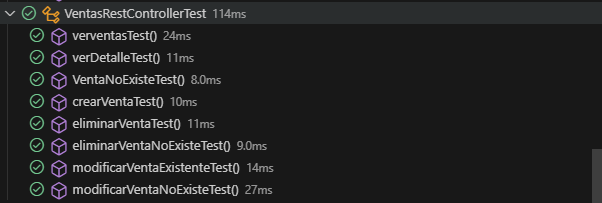
Las capturas de Visual Studio Code evidencian la ejecución exitosa de las pruebas unitarias y de integración desarrolladas en el proyecto. Estas pruebas permiten verificar tanto la lógica interna de los servicios como el correcto funcionamiento de los controladores REST en conjunto con sus dependencias simuladas. Todas las pruebas se ejecutaron sin errores, y el comportamiento del sistema es el esperado antes de su despliegue.

1. **Ejecución pruebas Unitarias**

****

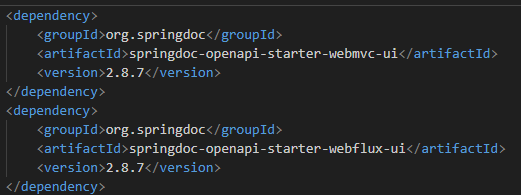
1. **Ejecución pruebas de Integración**





1. **Documentación**

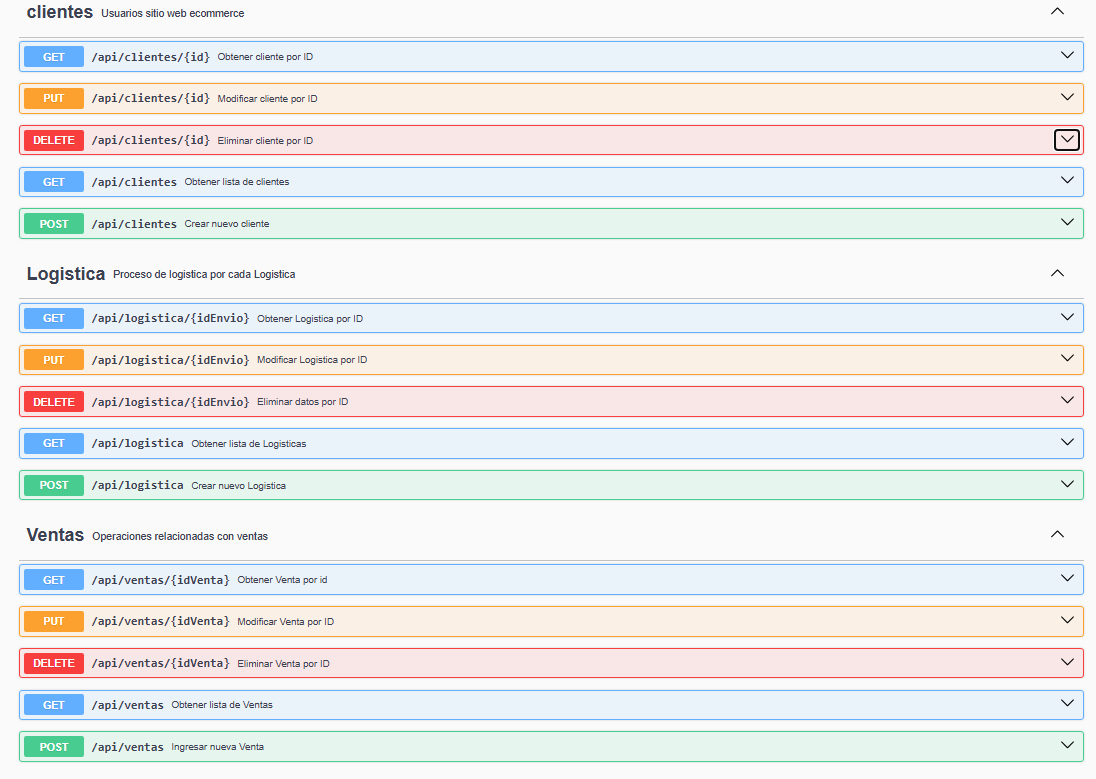
Para documentar los endpoints desarrollados en los microservicios, se utilizó Swagger mediante la especificación OpenAPI, integrando la dependencia Springdoc OpenAPI. Esta herramienta permite generar automáticamente una interfaz visual e interactiva para explorar y probar los endpoints REST expuestos por la aplicación. En particular, se empleó la siguiente dependencia:

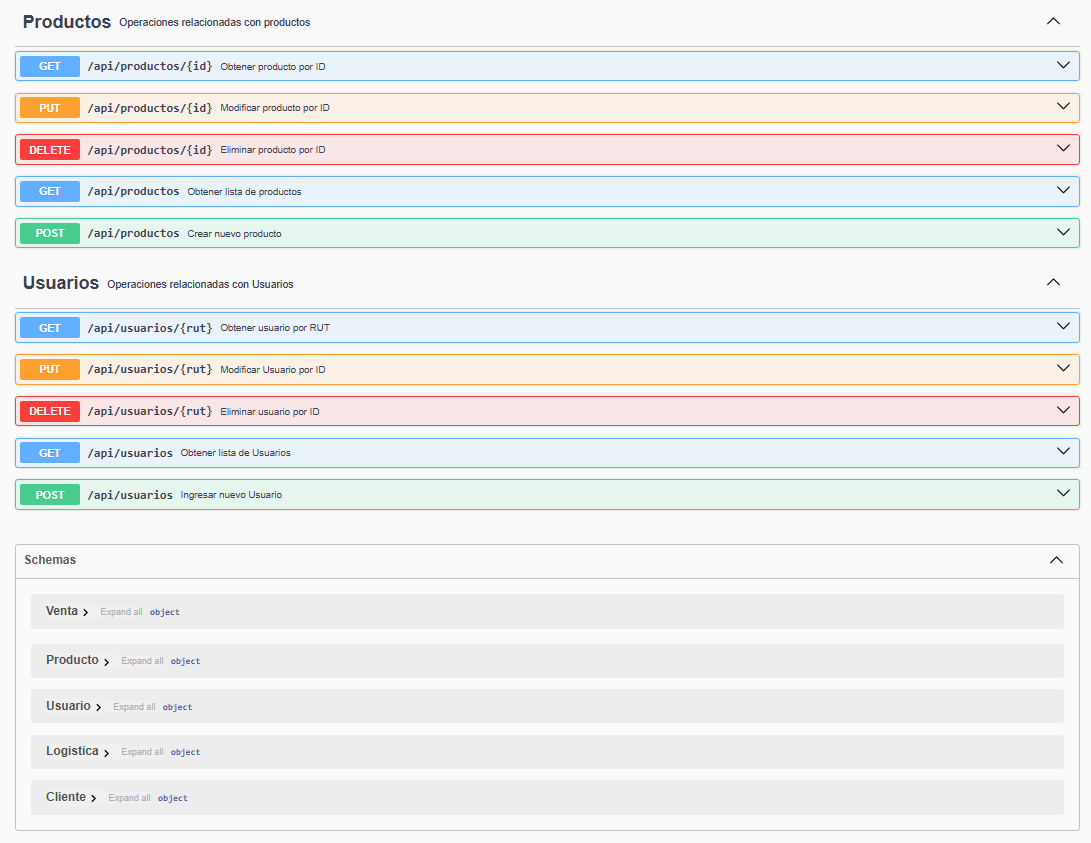
****

A continuación, se presentan capturas de pantalla de la interfaz visual generada automáticamente por Swagger UI gracias a la integración con Springdoc OpenAPI. Esta interfaz permite visualizar, probar y documentar de manera interactiva todos los endpoints REST disponibles en la aplicación. Cada operación incluye información detallada como:

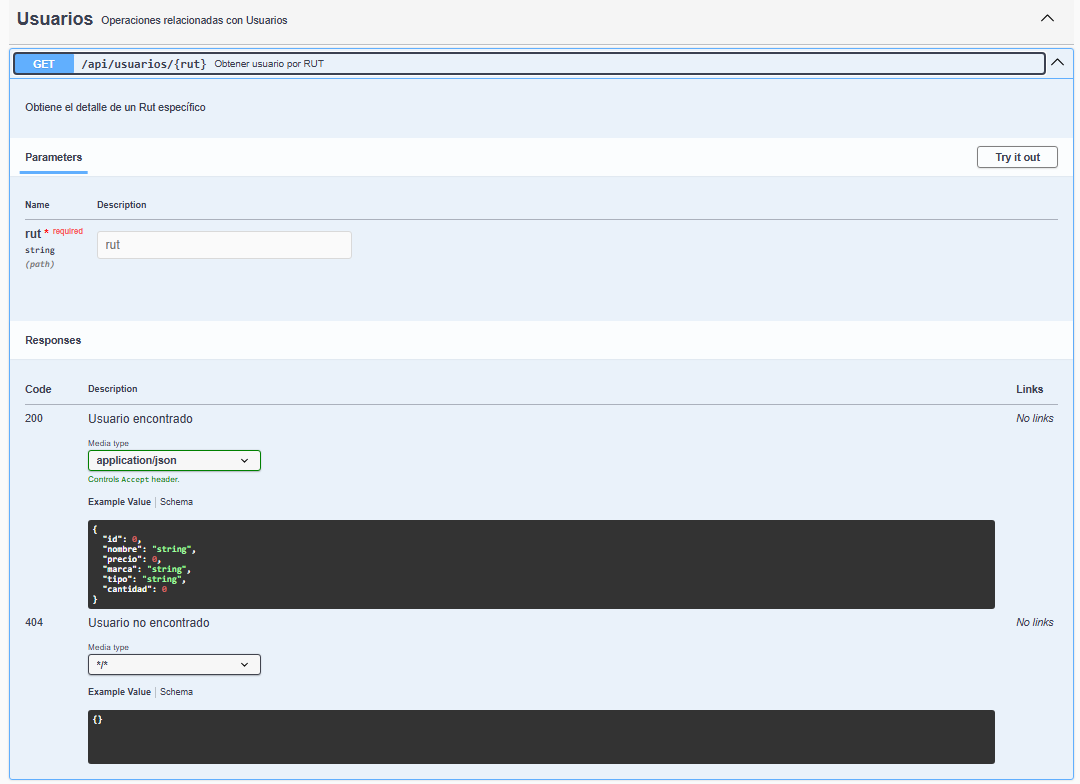
* La ruta del endpoint (GET, POST, PUT, DELETE, etc.).
* La descripción de lo que hace cada operación.
* Los códigos de respuesta esperados.
* Los parámetros y el formato del cuerpo (body) requerido para cada petición.
* La posibilidad de probar los endpoints en tiempo real, directamente desde el navegador.

Esta documentación es especialmente útil tanto para desarrolladores como para testers o equipos externos que necesitan conocer cómo interactuar con el API.

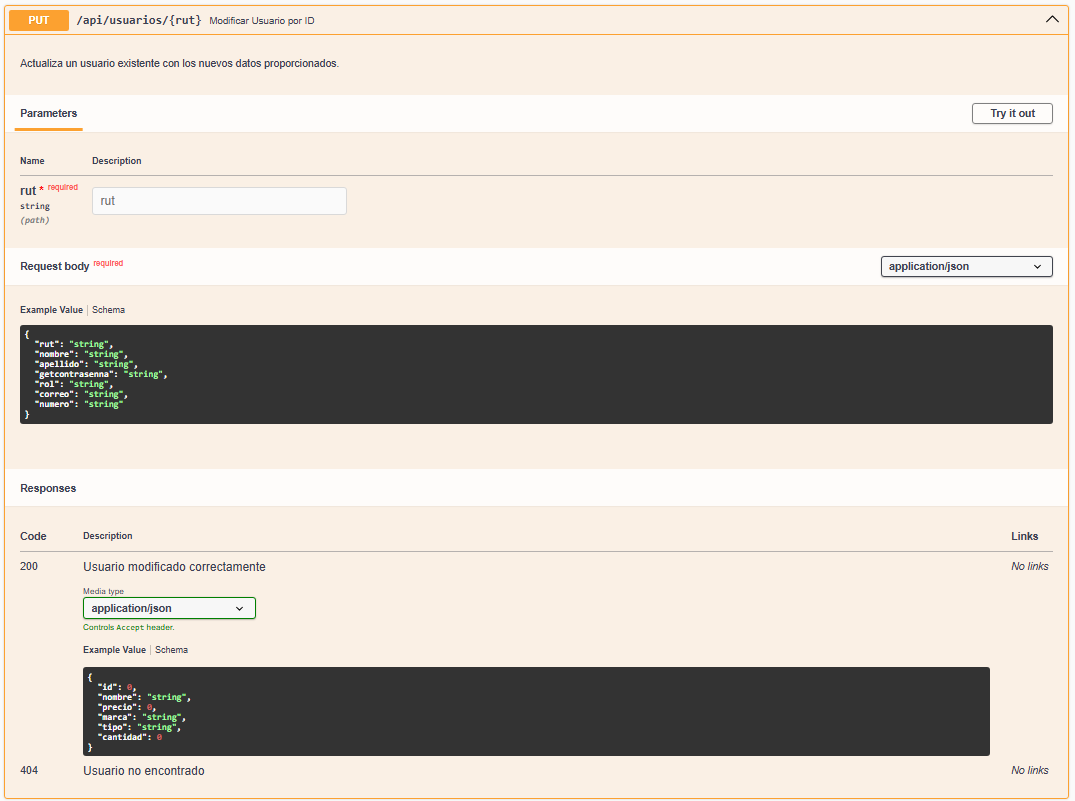
****

****

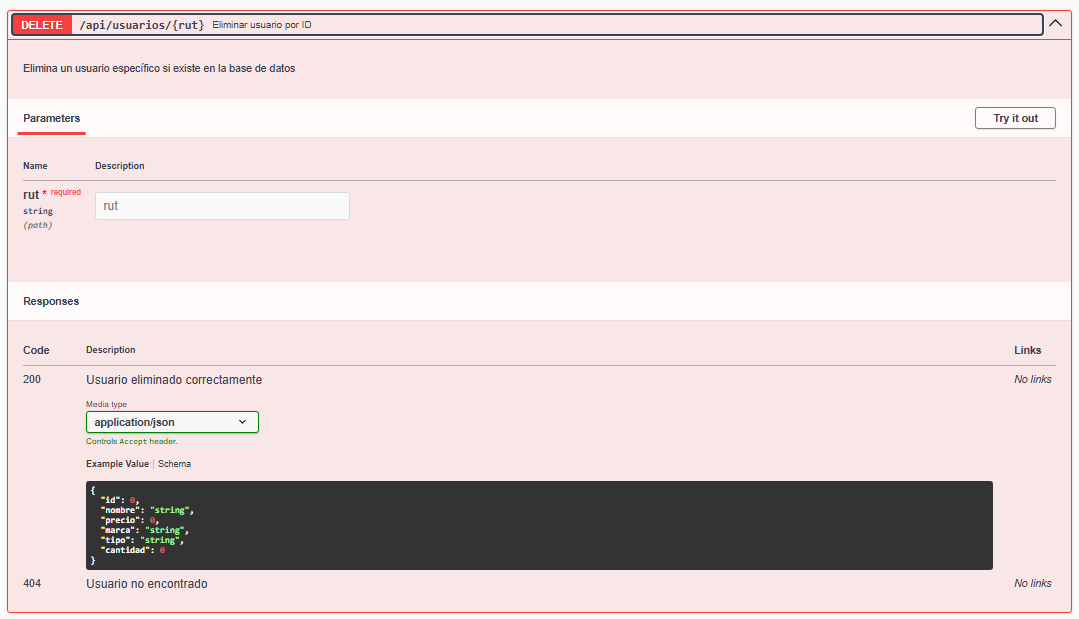
1. **Obtener usuario por RUT específico**

****

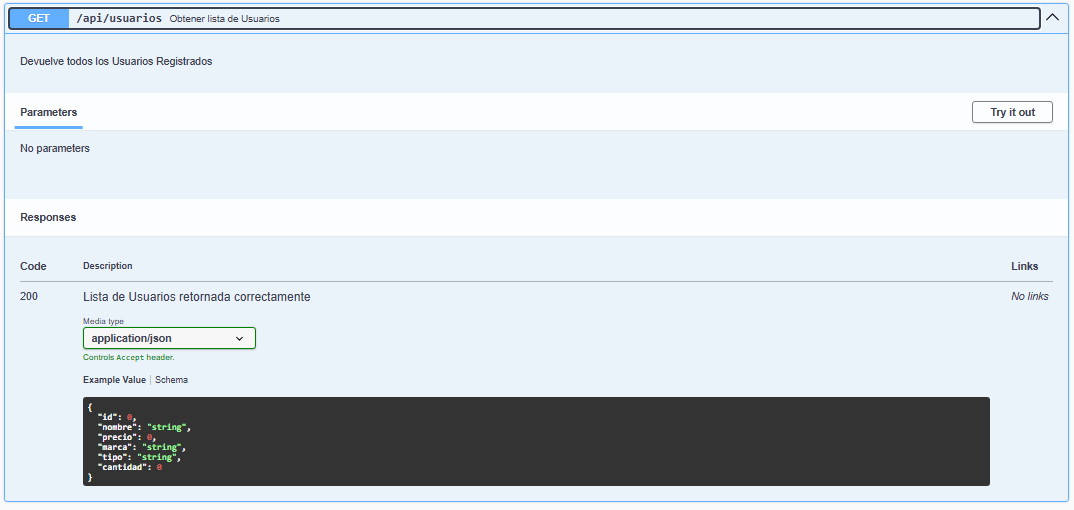
1. **Modificar Usuario**

****

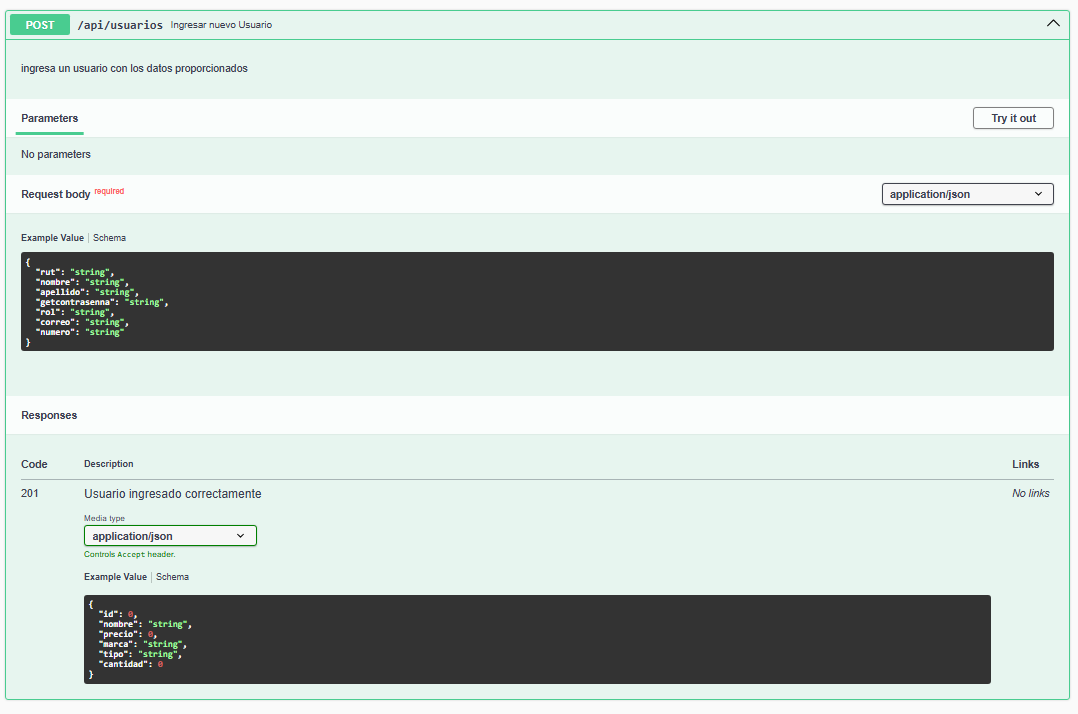
1. **Eliminar Usuario**

****

1. **Obtener Lista de Usuarios**

****

1. **Ingresar Usuario**

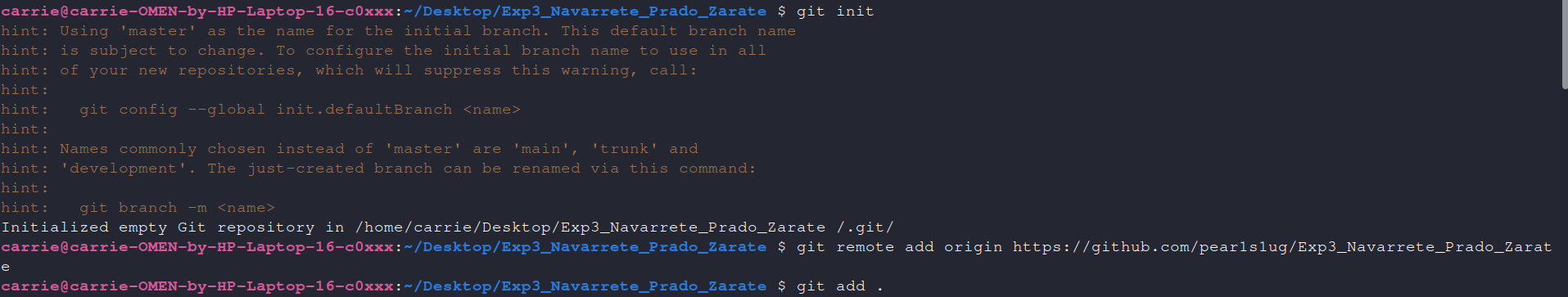
****

# Git-GitHub

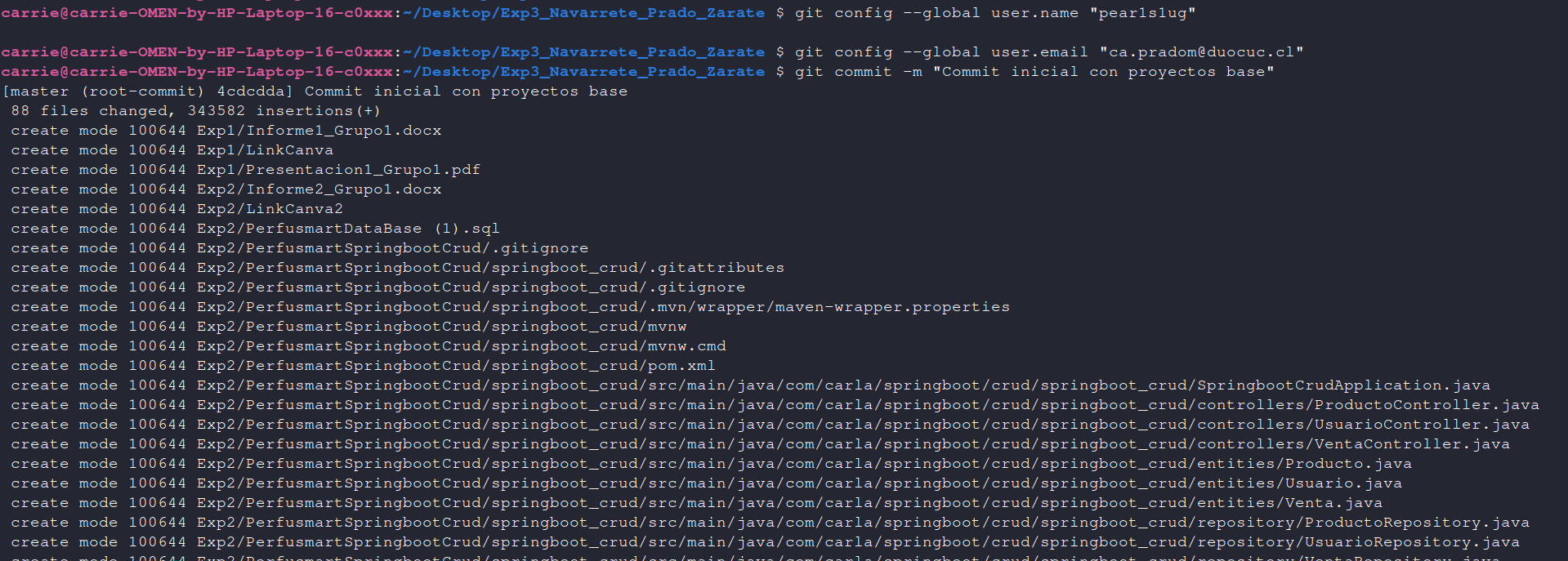
Se presentan capturas de pantalla que muestran el uso de comandos en Git Bash y terminal Linux para la creación y gestión de ramas (branches) en un entorno de trabajo colaborativo. El flujo seguido permite organizar el trabajo de cada integrante del equipo de forma ordenada, asegurando una integración controlada y sin conflictos.

El proceso incluyó los siguientes pasos clave:

1. Definir branch master tras haber creado un repositorio vacío.
2. Inicializar git, en este caso en la consola por defecto de Linux, y agregar la carpeta local que contiene el proyecto inicial.



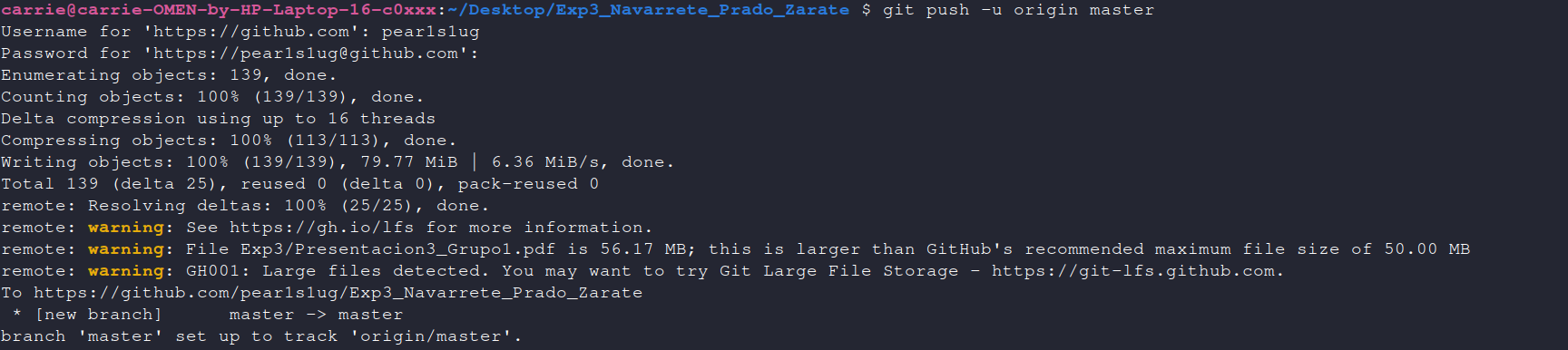
1. Definir datos del dueño del repositorio y realizar commit (guardado local).



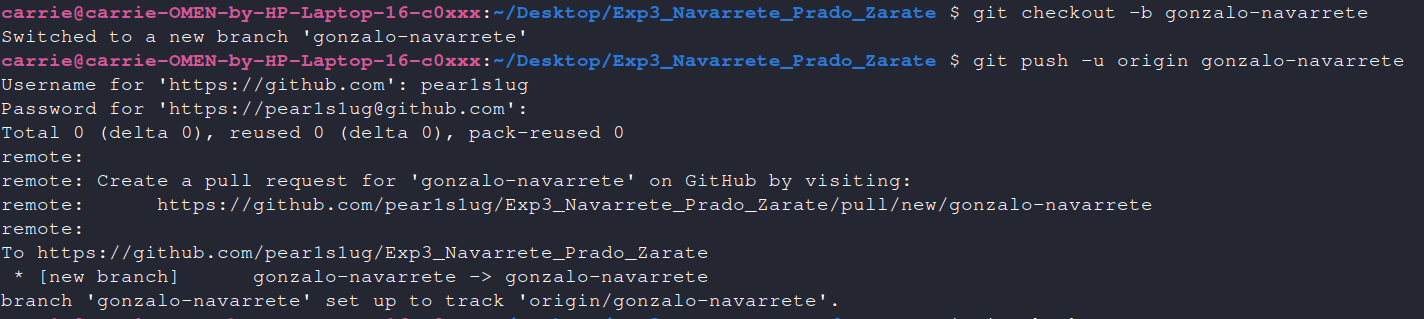
1. Renombrar branch “main” a “master”.

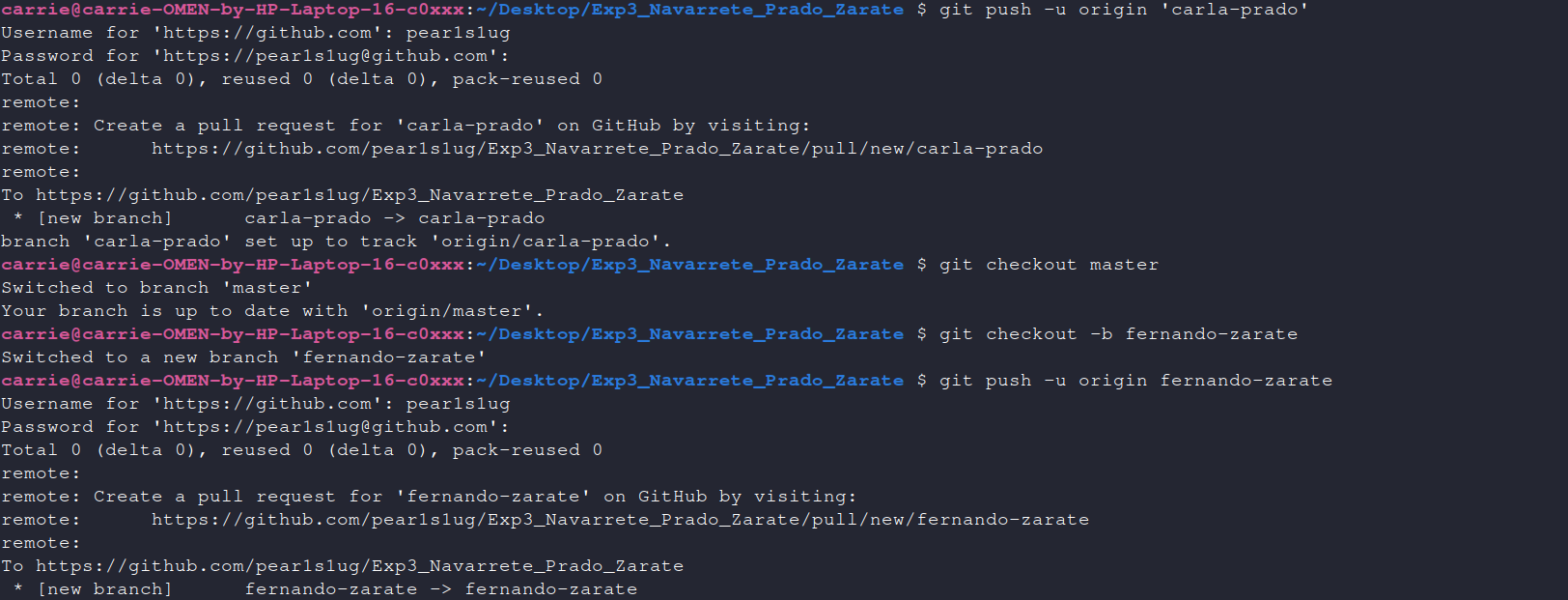


1. Realizar push (envío al repositorio remoto en GitHub).

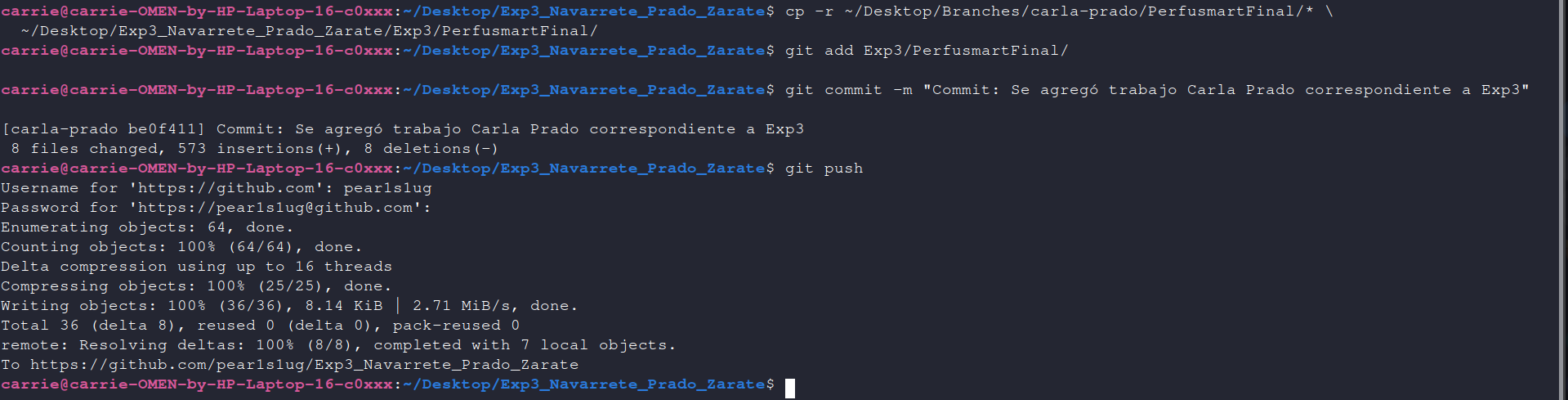


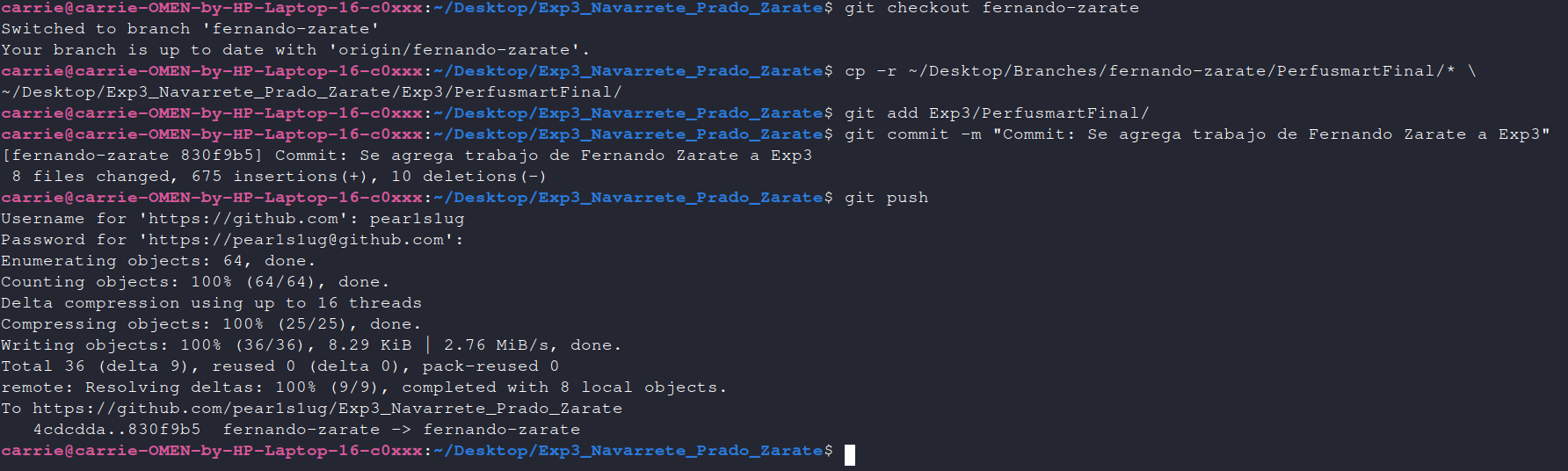
1. Crear branches con los nombres de los integrantes.



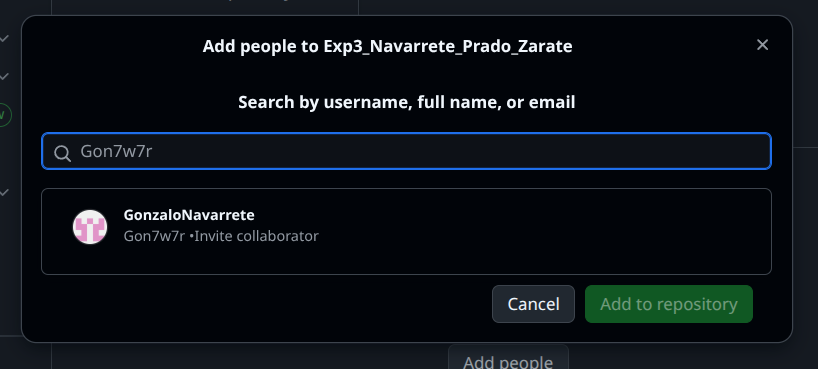


1. Subir trabajo de cada integrante a la branch que le corresponda.

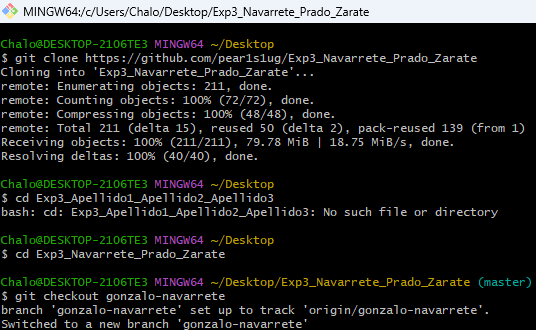




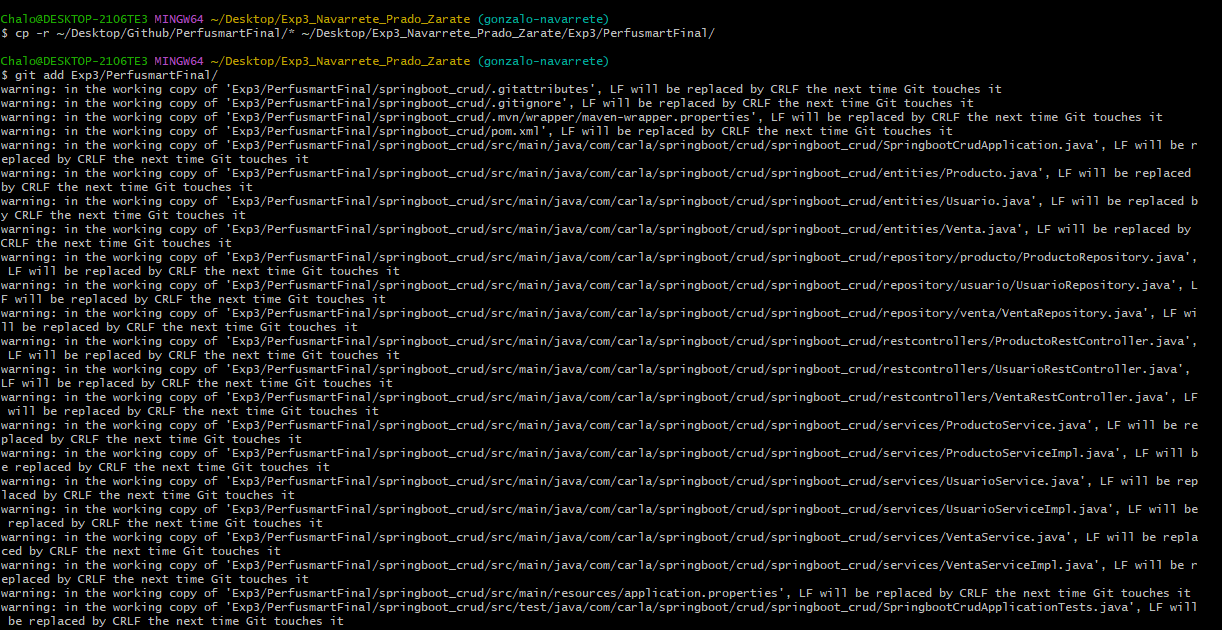
1. Subir trabajo desde otro computador
2. Dar permisos de colaborador al otro integrante



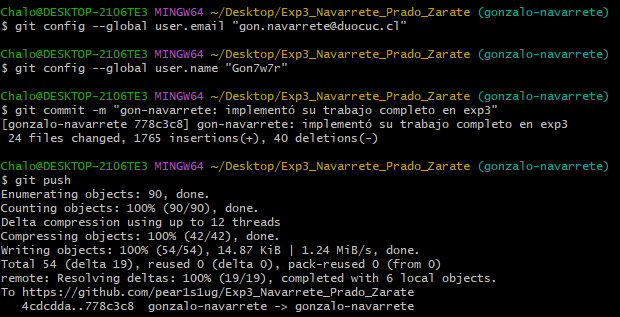
1. Desde Git Bash en Windows, clonar repositorio y ubicarse en carpeta clonada. Luego, ir a la branch correspondiente.



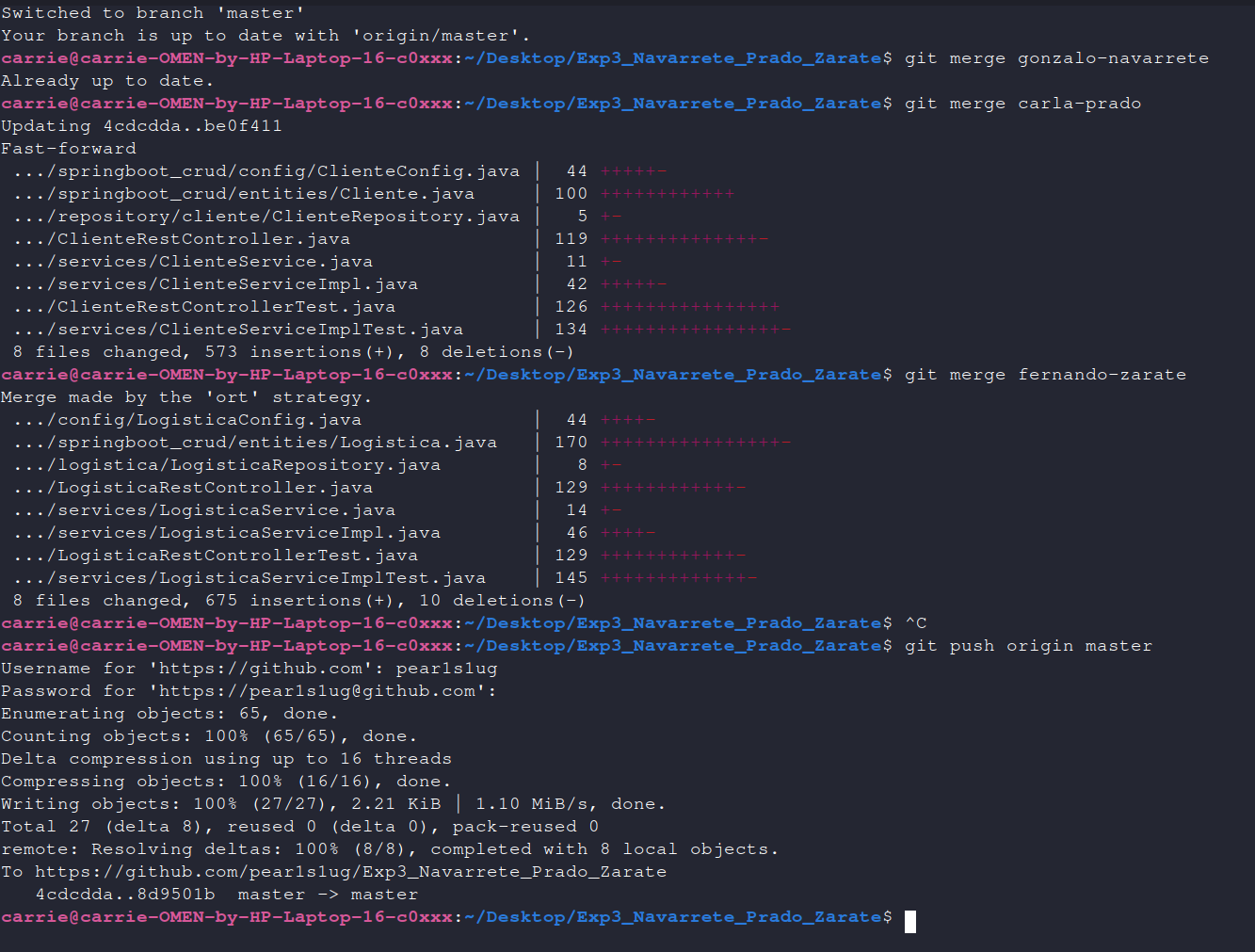
1. Copiar y pegar el archivo con el trabajo del integrante en la carpeta de la Experiencia 3, sólo dentro de su branch.

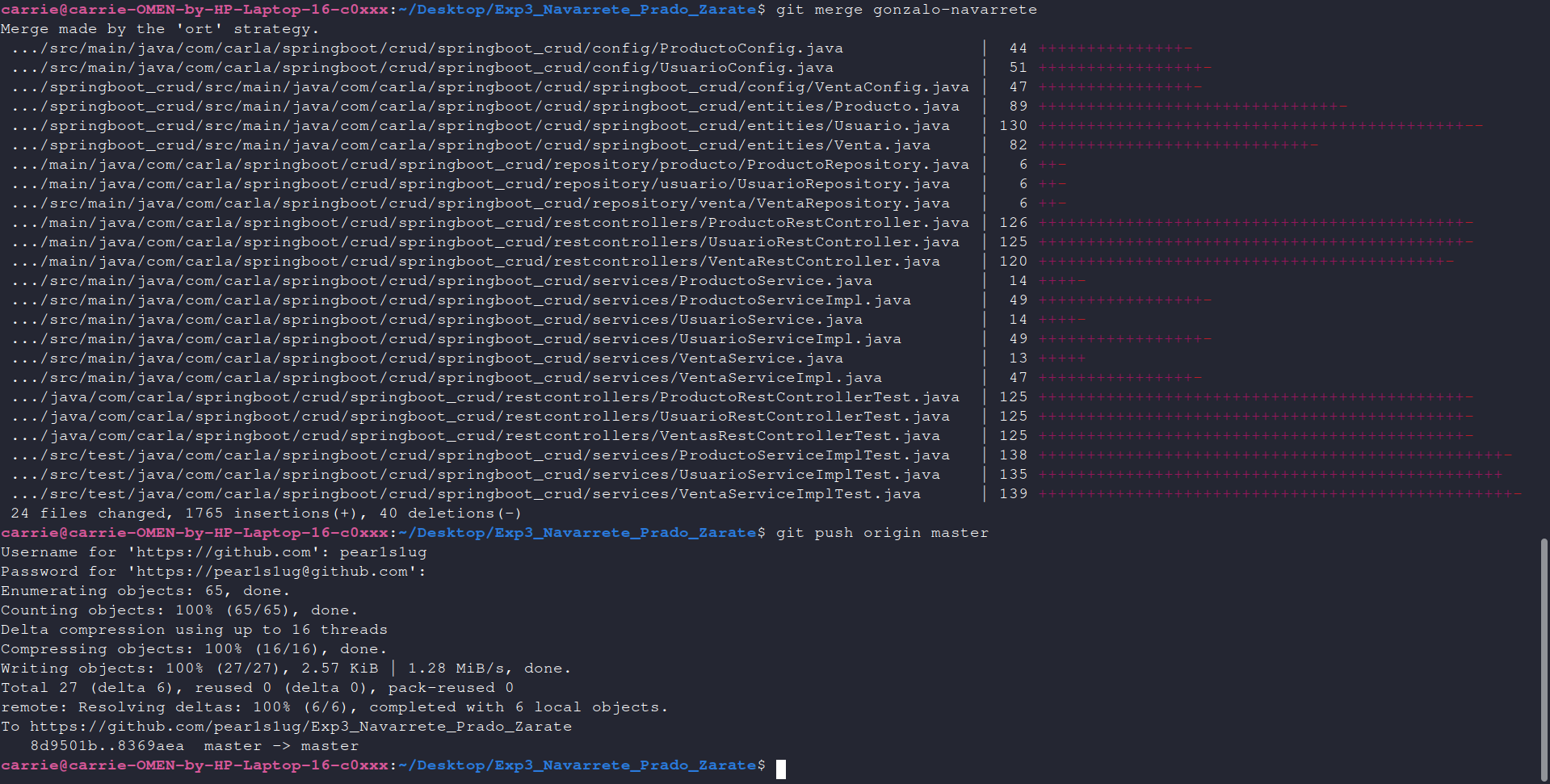


1. Iniciar sesión con sus credenciales y realizar commit y push.



1. Realizar merge entre todas las branches con la master.





1. Conclusión

Esta experiencia permitió aplicar de manera concreta los conocimientos adquiridos en torno a microservicios, pruebas y buenas prácticas de documentación. La implementación de pruebas unitarias e integración utilizando JUnit, Mockito y MockMvc no solo fortaleció la calidad del código, sino que también ayudó a comprender la importancia de aislar responsabilidades y asegurar que cada componente funcione correctamente, tanto de forma individual como en conjunto.

Además, el uso de herramientas como Swagger y Springdoc OpenAPI permitió generar documentación clara, estandarizada y accesible, lo cual es clave en un entorno de desarrollo profesional donde la colaboración y la comprensión del sistema por parte de otros equipos son fundamentales. También se destaca la gestión del proyecto en GitHub, donde se aprendió a trabajar en ramas, colaborar en equipo y subir avances de forma organizada.

En resumen, esta experiencia no solo reforzó aspectos técnicos como la codificación y el testeo, sino también habilidades transversales como el trabajo colaborativo, la organización de versiones y la entrega profesional de proyectos.