Arquitectura de Software II

Trabajo Práctico №2 - 2025s1

Maria Laura Medici
Natalia Stefania Rodriguez

Índice:

Introducción

Vamos a diseñar una plataforma de compra y venta de productos. Usuarios compradores y vendedores se van a comunicar exclusivamente mediante la plataforma Atlas, que permitirá la creación de esos usuarios, la creación de productos (por parte de los vendedores) y la compra de los mismos.

Requerimientos funcionales

Se va a tener dos tipos de usuarios en el sistema: Usuario comprador y vendedor.

- Creación y eliminación de usuarios vendedores.
- Creación, modificación y eliminación de usuarios compradores.
- Los usuarios compradores pueden ver todos los productos que tiene el sistema
- Los usuarios vendedores van a ser capaces de crear, actualizar o eliminar un producto
- Los usuarios vendedores van a ser capaces de ver que persona compro sus productos
- Los usuarios compradores van a poder realizar una compra sobre el producto que elijan
- Se va a poder realizar distintos tipos de búsqueda sobre el producto, entre las que podemos mencionar dado un rango de precios, dada la categoría o dado un nombre
- Se van a enviar notificaciones al usuario cuando compre un producto, y a la vez se va a enviar notificación al vendedor de que un usuario compró su producto.

Casos de uso

Usuario

- Crear

Nombre	Crear Usuario
Descripción	Permite crear un nuevo usuario en el sistema
Precondición	Ninguna
Flujo Principal	1. El cliente envía una solicitud POST a ***/usuario*** con los datos del nuevo usuario 2. El sistema valida los datos 3. El sistema crea el usuario en la base de datos 4. El sistema devuelve una respuesta de éxito
Flujo Alternativo	● Si los datos son inválidos o se intenta duplicar un

	usuario existente activo, el sistema devuelve un error ● Si el usuario existe y está eliminado, se vuelve a poner como activo al usuario
Postcondición	Un usuario es creado en la base de datos

- Modificar

Nombre	Modificar Usuario
Descripción	Permite modificar los datos de un usuario existente
Precondición	El usuario debe de existir en el sistema
Flujo Principal	1. El cliente envía una solicitud PUT a ***/usuario/*** con los nuevos datos del usuario 2. El sistema valida los datos 3. El sistema modifica el usuario en la base de datos 4. El sistema devuelve una respuesta de éxito
Flujo Alternativo	● Si el usuario no existe o figura como eliminado, el sistema devuelve un error de no encontrado. ● Si no se ingresan todos los campos obligatorios, el sistema devuelve mensaje de error
Postcondición	Los datos del usuario son actualizados en el sistema

- Eliminar

Nombre	Eliminar Usuario
Descripción	Permite eliminar un usuario existente del sistema
Precondición	El usuario debe de existir en el sistema
Flujo Principal	1. El cliente envía una solicitud DELETE a ***/usuario/{mail}*** 2. El sistema verifica la existencia del usuario activo 3. El sistema realiza un eliminado lógico en la base de datos 4. El sistema devuelve una respuesta de éxito.
Flujo Alternativo	Si el usuario no existe, el sistema devuelve un error de no encontrado
Postcondición	Se ha eliminado al usuario del sistema

Vendedor

- Crear

Nombre	Crear Vendedor
Descripción	Permite crear un nuevo vendedor en el sistema.
Precondición	-
Flujo Principal	1. El cliente envía una solicitud POST a ***/vendedor*** con los datos del nuevo vendedor 2. El sistema valida los datos 3. El sistema crea el vendedor en la base de datos 4. El sistema devuelve una respuesta de éxito
Flujo Alternativo	Si los datos son inválidos o se intenta duplicar un vendedor existente, el sistema devuelve un error
Postcondición	Un nuevo vendedor es creado en el sistema

- Modificar

Nombre	Modificar Vendedor
Descripción	Permite modificar los datos de un vendedor existente
Precondición	El vendedor debe de existir en el sistema
Flujo Principal	1. El cliente envía una solicitud PUT a ***/vendedor*** con los nuevos datos del vendedor 2. El sistema valida que el email exista 3. El sistema modifica el vendedor en la base de datos 4. El sistema devuelve una respuesta de éxito
Flujo Alternativo	Si el vendedor no existe, el sistema devuelve un error de no encontrado
Postcondición	Los datos del vendedor son actualizados en el sistema

- Eliminar

Nombre	Eliminar Vendedor
Descripción	Permite eliminar un vendedor existente del sistema, junto con todos sus productos

Precondición	El vendedor debe de existir en el sistema
Flujo Principal	1. El cliente envía una solicitud DELETE a ***/vendedor/{mail}*** 2. El sistema verifica la existencia del vendedor 3. El sistema realiza un borrado lógico en la base de datos 4. El sistema devuelve una respuesta de éxito.
Flujo Alternativo	Si el vendedor no existe, el sistema devuelve un error de no encontrado
Postcondición	Se ha eliminado al vendedor

- Buscar por Id

Nombre	Obtener Detalle del Vendedor
Descripción	Permite obtener la información de un vendedor específico
Precondición	Ninguna
Flujo Principal	1. El cliente envía una solicitud GET a ***/vendedor/{mail}*** 2. El sistema verifica la existencia del vendedor 3. El sistema devuelve la información del vendedor solicitado
Flujo Alternativo	Si el vendedor no existe, el sistema devuelve un error
Postcondición	Se obtiene la información del vendedor específico

Producto

- Crear

Nombre	Crear Producto
Descripción	Permite crear un nuevo producto en el sistema
Precondición	Ninguna
Flujo Principal	5. El cliente envía una solicitud POST a ***/producto*** con los datos del nuevo producto 6. El sistema valida los datos 7. El sistema crea el producto en la base de datos o actualiza los datos del mismo

	8. El sistema devuelve una respuesta de éxito
Flujo Alternativo	Si los datos son inválidos o el vendedor no existe, el sistema devuelve un error
Postcondición	Un nuevo producto es creado en el sistema

- Modificar

Nombre	Modificar Producto
Descripción	Permite modificar los datos de un producto existente
Precondición	El producto debe de existir en el sistema
Flujo Principal	1. El cliente envía una solicitud PUT a ***/producto/*** con los nuevos datos del producto 2. El sistema valida los datos 3. El sistema actualiza el producto en la base de datos 4. El sistema devuelve una respuesta de éxito
Flujo Alternativo	Si el producto no fue encontrado en el sistema o hay algun error con el vendedor, el sistema devuelve un error de no encontrado
Postcondición	Los datos del producto son actualizados en el sistema

- Eliminar

Nombre	Eliminar Producto
Descripción	Permite eliminar un producto existente del sistema
Precondición	El producto debe de existir en el sistema
Flujo Principal	1. El cliente envía una solicitud DELETE a ***/producto/{id}/{mail}*** 2. El sistema verifica la existencia del producto 3. El sistema verifica que el producto sea del vendedor 4. El sistema hace un eliminado lógico en la base de datos 5. El sistema devuelve una respuesta de éxito.
Flujo Alternativo	● Si el producto no existe, el sistema devuelve un error de no encontrado. ● Si el producto es de otro vendedor, el sistema devuelve error

Postcondición	Se ha eliminado al producto del sistema

- Buscar por vendedor

Nombre	Obtener Detalle de los productos de un vendedor
Descripción	Permite obtener la información de todos los productos creados por un vendedor
Precondición	Ninguna
Flujo Principal	1. El cliente envía una solicitud GET a ***/producto/{mailVendedor}*** 2. El sistema devuelve la información de los productos solicitados
Flujo Alternativo	Si el vendedor no existe o el producto fue , el sistema devuelve una lista vacía
Postcondición	Se obtiene la información de los productos creados por el vendedor

- Obtener productos filtrados

Nombre	Obtener todos los productos que cumplan con el filtro
Descripción	Permite obtener la información de todos los productos cuyo filtro pueden ser: ● Precio mayor a ● Precio menor a ● Rango de precios - entre mayor y menor a ● Categoría ● Descripción
Precondición	Ninguna
Flujo Principal	1. El cliente envía una solicitud GET a ***/producto/disponibles/{filtro}*** 2. El sistema devuelve la información de los productos solicitados
Flujo Alternativo	Si no hay productos que superen alguno de esos filtros, devuelve lista vacía
Postcondición	Se obtiene la información de los productos cuya información se encuentre comprendida con la aplicación del filtro

Venta

- Registrar venta

Nombre	Registrar venta
Descripción	Permite registrar la venta de un producto existente
Precondición	El producto y el usuario a comprar deben de existir en el sistema
Flujo Principal	1. El cliente envía una solicitud PUT a ***/venta/*** con el id del producto, el email del usuario y la cantidad que desea comprar 2. El sistema valida los datos 3. El sistema actualiza la cantidad de stock del producto en la base de datos 4. El sistema envía notificaciones al usuario comprador que se realizó la compra correctamente 5. El sistema envía notificaciones al usuario vendedor de que se ha comprado un producto 6. El sistema devuelve una respuesta de éxito
Flujo Alternativo	● Si el producto o el usuario no son encontrados en el sistema, el sistema devuelve un error de no encontrado ● Si la cantidad que se desea comprar es mayor a la cantidad de stock del producto, el sistema devuelve un mensaje de error ● Si la cantidad que se desea comprar es menor a 1, el sistema devuelve un mensaje de error
Postcondición	Los datos del producto son actualizados en el sistema y se envían las notificaciones correspondientes

Atributos de calidad

Se enumeran algunos de los atributos y sus escenarios, importantes para esta aplicación, con una breve descripción:

1. Escalabilidad

➢ **Fuente del estímulo (Source):** Administrador del sistema
➢ **Estímulo (Stimulus):** Agregar nuevas funcionalidades

- ➢ **Artefacto (Artifact):** Código e interfaz de usuario
- ➢ **Entorno (Environment):** Desarrollo
- ➢ **Respuesta (Response):** La nueva funcionalidad se agrega, se testea y se despliega
- ➢ **Medición de la respuesta (Response Measure):** El esfuerzo de no menos de 2 personas durante un mes
- ➢ **Justificación:** La escalabilidad es crucial para permitir el crecimiento del sistema, ya que anticipa la necesidad de agregar nuevas funcionalidades y características a medida que el negocio crece.

2. Desplegabilidad

- ➢ **Fuente del estímulo (Source):** Administrador del sistema
- ➢ **Estímulo (Stimulus):** Nueva versión del sistema a ser desplegada, incluyendo la funcionalidad de envío de notificaciones
- ➢ **Artefacto (Artifact):** Actualización de funcionalidades existentes y nuevas, incluyendo el sistema de notificaciones
- ➢ **Entorno (Environment):** Despliegue completo
- ➢ **Respuesta (Response):** Incorporar la nueva funcionalidad de notificaciones, y separación de componentes, sin interrumpir el servicio
- ➢ **Medición de la respuesta (Response Measure):** Tiempo total del despliegue y efectividad del sistema de notificaciones
- ➢ **Justificación:** Un sistema de CI/CD bien implementado es esencial para facilitar el despliegue de nuevas versiones y actualizaciones, minimizando el tiempo de inactividad y asegurando una transición fluida. La incorporación de un sistema de notificaciones también debe ser ágil y eficiente, garantizando que los usuarios reciban información relevante en tiempo real, lo que mejora la experiencia del usuario y la interacción con la plataforma.

3. Performance

- ➢ **Fuente del estímulo (Source):** Usuario final
- ➢ **Estímulo (Stimulus):** Búsqueda y filtrado de productos
- ➢ **Artefacto (Artifact):** Sistema completo
- ➢ **Entorno (Environment):** Normal
- ➢ **Respuesta (Response):** El sistema retorna resultados de búsqueda
- ➢ **Medición de la respuesta (Response Measure):** Latencia
- ➢ **Justificación:** La performance es crítica, ya que los usuarios esperan respuestas rápidas al buscar y filtrar productos, lo que impacta directamente en la experiencia del usuario y en la tasa de conversión de ventas.

4. Usabilidad

- ➢ **Fuente del estímulo (Source):** Usuario final

- ➢ **Estímulo (Stimulus):** El usuario entiende cómo usar el sistema
- ➢ **Artefacto (Artifact):** Sistema completo
- ➢ **Entorno (Environment):** Tiempo de ejecución
- ➢ **Respuesta (Response):** El sistema es fácil de usar, con ayudas y tutoriales cuando sea necesario
- ➢ **Medición de la respuesta (Response Measure):** Cantidad de errores, satisfacción del usuario, tasa de retención de usuarios
- ➢ **Justificación:** La usabilidad es fundamental para garantizar que tanto los compradores como los vendedores puedan interactuar con la plataforma de manera efectiva, lo que fomenta un uso continuo y una mayor satisfacción.

5. Interoperabilidad

- ➢ **Fuente del estímulo (Source):** Usuario final
- ➢ **Estímulo (Stimulus):** Procesar el pago de un producto
- ➢ **Artefacto (Artifact):** Sistema de pagos de la aplicación
- ➢ **Entorno (Environment):** Tiempo de ejecución
- ➢ **Respuesta (Response):** Realizar el pago correctamente
- ➢ **Medición de la respuesta (Response Measure):** Cantidad de transacciones exitosas y errores en pagos
- ➢ **Justificación:** A pesar de que el sistema no cuenta con un sistema de pagos, la interoperabilidad es esencial para asegurar que el sistema de pagos de una aplicación de compras funcione sin problemas con diferentes métodos de pago y plataformas, garantizando una experiencia de compra fluida y confiable.

6. Seguridad

- ➢ **Fuente del estímulo (Source):** Usuario final
- ➢ **Estímulo (Stimulus):** Acceso a datos sensibles
- ➢ **Artefacto (Artifact):** Base de datos y sistema de autenticación
- ➢ **Entorno (Environment):** Tiempo de ejecución
- ➢ **Respuesta (Response):** Proteger la información sensible y garantizar que solo los usuarios autorizados tengan acceso
- ➢ **Medición de la respuesta (Response Measure):** Número de brechas de seguridad y accesos no autorizados
- ➢ **Justificación:** En cualquier sistema, la seguridad es crítica para proteger la información personal y financiera de los usuarios (en este caso compradores o vendedores), lo que a su vez genera confianza en la plataforma.

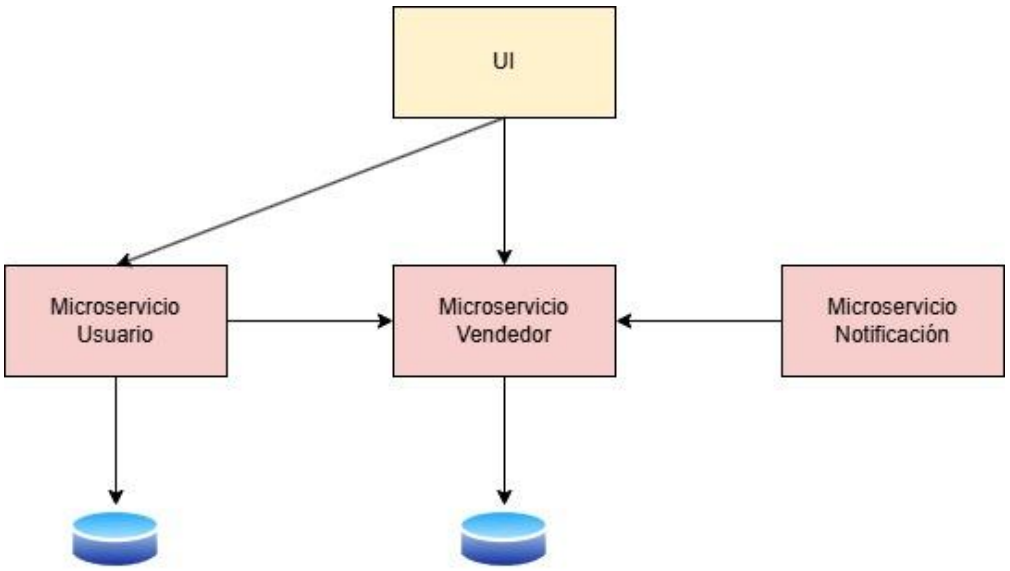
Tipo de arquitectura: Microservicios

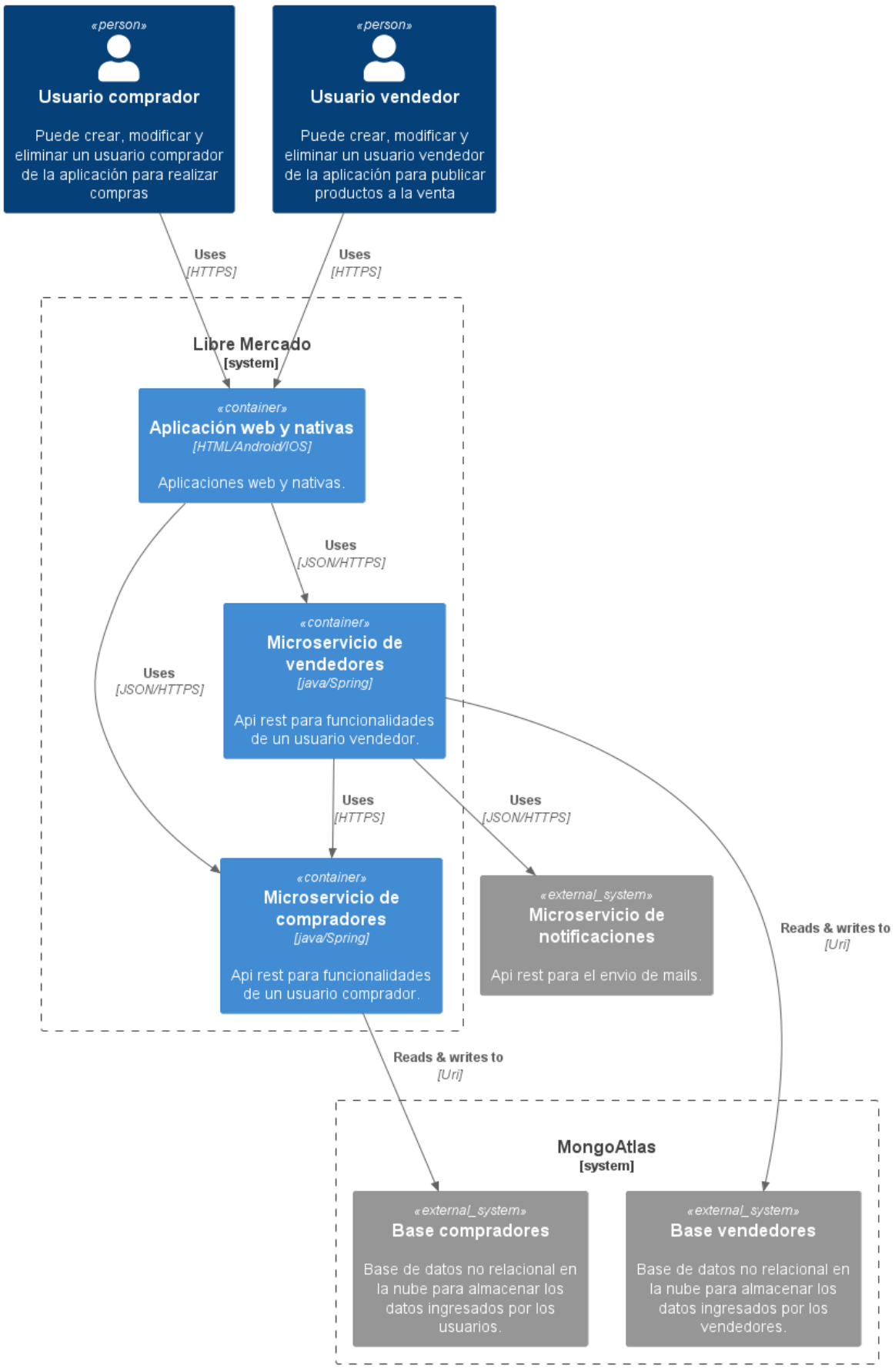
En un principio, la arquitectura de nuestro sistema era Hexagonal. Ya que este enfoque nos permitió crear un modelo de negocios desacoplado entre las capas de dominio, adaptadores

e infraestructura. Sin embargo, a medida que el negocio iba creciendo nos encontramos con limitaciones significativas:

Escalabilidad: escalar una parte específica de la aplicación, sin escalar toda la aplicación se vuelve un desafío, lo que resultaba un uso ineficiente de nuestros recursos. Cuando la compra tenía una alta demanda, y necesitábamos hacer un cambio con respecto a los usuarios compradores, nos veíamos obligados a escalar toda la aplicación.
Resiliencia y Mantenibilidad: Un fallo en un componente podía potencialmente afectar a toda la aplicación, comprometiendo la resiliencia del sistema. Además de que el mantenimiento de una base de código grande y densamente acoplada, a pesar de su modularidad hexagonal, se volvió más complejo y propenso a errores.

Por estos motivos, se tomó la decisión de migrar hacia una **arquitectura de microservicios**. Este enfoque fue elegido debido a la necesidad de desarrollar un sistema flexible, escalable y robusto, capaz de adaptarse a un entorno tecnológico con cambios constantes. En este contexto, la evolución continua nos permite adaptarnos rápidamente a las necesidades del mercado, mejorando la calidad de nuestro software e innovando constantemente. Los microservicios nos ofrecen una escalabilidad granular, mayor resiliencia y una independencia total para los equipos, acelerando los ciclos de desarrollo y despliegue. Este nuevo enfoque introduce nuevas complejidades en la gestión y el monitoreo, pero los beneficios a largo plazo relacionados a la agilidad y la escalabilidad del negocio son innegables.




«person»
Usuario comprador
Puede crear, modificar y eliminar un usuario comprador de la aplicación para realizar compras
«person»
Usuario vendedor
Puede crear, modificar y eliminar un usuario vendedor de la aplicación para publicar productos a la venta
Uses
[HTTPS]
Uses
[HTTPS]
Libre Mercado
[system]
«container»
Aplicación web y nativas
[HTML/Android/IOS]
Aplicaciones web y nativas.
Uses
[JSON/HTTPS]
Uses
[JSON/HTTPS]
«container»
Microservicio de vendedores
[java/Spring]
Api rest para funcionalidades de un usuario vendedor.
Uses
[HTTPS]
Uses
[JSON/HTTPS]
«container»
Microservicio de compradores
[java/Spring]
Api rest para funcionalidades de un usuario comprador.
«external_system»
Microservicio de notificaciones
Api rest para el envio de mails.
Reads & writes to
[Uri]
Reads & writes to
[Uri]
MongoAtlas
[system]
«external_system»
Base compradores
Base de datos no relacional en la nube para almacenar los datos ingresados por los usuarios.
«external_system»
Base vendedores
Base de datos no relacional en la nube para almacenar los datos ingresados por los vendedores.

Justificación de la división del servicio

Drivers desintegradores de servicio

- **Alcance y funcionalidad:** En la arquitectura hexagonal original teníamos un único servicio que abarcaba las funcionalidades para administrar usuarios (compradores y vendedores), productos y el procesamiento de ventas. Al observar que las responsabilidades de un usuario comprador son diferentes a las de un usuario vendedor, decidimos dividir los servicios en dos, teniendo en cuenta los siguientes contextos:
  Gestión de Usuarios: Existe una necesidad clara de gestionar el perfil de los usuarios compradores y el de los vendedores. Aunque ambos son "usuarios", sus roles y las operaciones básicas son distintas e independientes. Por este motivo, se crearon microservicios dedicados a la gestión de usuarios compradores y otra para la de usuarios vendedores. Esta división promueve el principio de Responsabilidad Única a nivel de entidad principal y permite la evolución independiente de los perfiles.
  Compra y Venta de Productos: Por otro lado, las funcionalidades relacionadas con la búsqueda de producto y la venta de los mismos, están fuertemente acopladas. Estas operaciones tienen una alta interdependencia lógica y un ciclo de vida que a menudo requiere consistencia transaccional (como la gestión de stock durante una venta). Para gestionar esta complejidad y asegurar la atomicidad donde es crítica, se decidió que estas funcionalidades residan dentro del microservicio de Vendedores. Aunque el nombre del servicio es "Vendedores", este encapsula el dominio completo de la oferta y la transacción, desde la perspectiva del proceso de venta.

- **Volatilidad del código:** Otro factor determinante para la división de servicios fue la diferencia en la frecuencia de cambio y evolución del código entre ambos dominios. El servicio responsable de las funcionalidades del usuario comprador presenta una volatilidad considerablemente baja. Esto se basa en que las reglas de negocio del comprador son relativamente estables y maduras, con menos requerimientos de cambios o adición de nuevas funcionalidades a mediano o largo plazo, lo que contrarresta con las operaciones de venta y producto. Al aislar este dominio en un servicio independiente, logramos los siguientes beneficios:
  Reducción de despliegues: La baja volatilidad del microservicio de compradores implica que requiere menos modificaciones y, por ende, menos despliegues. Reduciendo de esta manera, el riesgo asociado a las puestas en producción y minimizando la interrupción en el resto del sistema.
  Mejora de la Estabilidad: Un servicio con pocas modificaciones tiende a ser más estable y predecible, contribuyendo a la resiliencia general de la aplicación.
  Por el contrario, el microservicio relacionado al vendedor, los productos y las funcionalidades de venta fueron considerados dominios de alta volatilidad, debido a los cambios frecuentes a corto plazo.

- **Escalabilidad y rendimiento:** La separación de estos servicios es altamente beneficiosa para la escalabilidad y el rendimiento, debido a que el servicio para vendedores va a presentar una carga de trabajo mayor y más volátil. Al aislarlo,

podemos **escalar de forma independiente** este componente de mayor demanda (mediante la adición de más servidores, memoria y CPU), optimizando el uso de recursos y asegurando el rendimiento crítico, sin necesidad de escalar todo el sistema.

- **Tolerancia a fallos:** La división de los servicios mejora la tolerancia a fallos. Debido a que el código de compradores presenta una menor volatilidad, por lo que la probabilidad de fallos es inferior que el de productos y vendedores. Al mantener independientes los servicios, logramos el aislamiento de dominios. Lo que conlleva a que si existe un fallo en el servicio de vendedores no afectará la disponibilidad ni la funcionalidad del servicio de usuarios compradores, aumentando la resiliencia global del sistema.

- **Seguridad:** La seguridad es otro concepto importante para la separación de servicios. El servicio para usuarios vendedores requiere mayor nivel de seguridad debido a que maneja transacciones críticas y de información sensible relacionada a la compra de los productos con algún medio de pago. Todo esto nos permite implementar controles más robustos de seguridad específicos con las necesidades propias del servicio, optimizando los esfuerzos de seguridad sin sobrecargar el servicio de compradores con medidas innecesarias.

- **Modificabilidad:** El desacoplamiento de los servicios permite que cada servicio pueda evolucionar y recibir nuevas funcionalidades de manera independiente. Si surgen nuevos requerimientos para el comprador, se implementarán en el servicio correspondiente, lo mismo si son para las operaciones de ventas o la gestión del vendedor, se añadirán al suyo. Eliminando la necesidad de desplegar el sistema completo por cada cambio, reduciendo significativamente el número de despliegues generales.

Drivers integradores de servicio

- **Transacciones de base de datos:** Decidimos unificar los módulos de vendedor, productos y ventas en un único microservicio debido a la intensa relación que hay entre un vendedor y sus productos. También necesitábamos que al realizarse una venta, quede actualizado el stock de un producto y que se registre la venta, pero si la segunda fallaba necesitábamos poder volver atrás la modificación del producto y su stock.

- **Workflow y coreografía:** Tener los módulos de vendedor, productos y ventas en un único microservicio nos permite que la creación de un producto sea inmediata ya que debe validar que el vendedor exista y asociarlo.
  Pasa lo mismo al momento de registrar una venta, donde necesitamos saber si un producto está disponible y con stock para su compra, ya que luego de hacer esta verificación la venta se registra. Si la registración de la venta fallase, habría que hacer rollback del stock del producto y por eso es necesario tener estos módulos juntos.

- **Relación entre datos:** El módulo de producto se dejó junto al módulo de ventas debido a que para crear un producto o para buscar alguno se debe verificar que el estado del vendedor sea ACTIVO. Y como se dijo anteriormente, los módulos de venta y producto están juntos para facilitar el rollback de una venta de ser necesaria.

Test de integración

Los test de integración son cruciales en una arquitectura de microservicios, ya que los mismos verifican que los distintos componentes de la aplicación trabajan conjuntamente de la manera en que se espera que lo hagan, garantizando la funcionalidad global.
En esencia, los mismos se centran en verificar que la comunicación y cooperación sea fluida entre las distintas partes de un sistema de software. Este tipo de pruebas se realizan después de los tests unitarios, en las que los componentes individuales se prueban de forma aislada, y antes de las pruebas del sistema, que evalúan toda la aplicación de software en su conjunto.

Importancia de los tests de integración

- **Detección temprana de errores de integración:** los test de integración ayudan a la identificación y resolución de errores encontrados cuando los diferentes componentes del sistema interactúan. Esta detección temprana evita que los mismos errores se vuelvan más complejos y costosos con el paso de sprints de desarrollo.
- **Mayor fiabilidad del sistema:** al verificar que todos los componentes trabajan armoniosamente, las pruebas de integración mejoran la fiabilidad del sistema. Lo que resulta en un sistema mucho más estable y fiable.
- **Mejora en la localización de errores:** cuando surgen los errores en los tests de integración, podemos localizar el origen de los mismos. La localización precisa de los mismos agiliza la depuración de la aplicación, facilitando y acelerando la resolución de los errores.
- **Validación de Requisitos Funcionales:** Aseguran que el sistema cumple con los requisitos funcionales establecidos, verificando que las funcionalidades se implementen correctamente en conjunto.
- **Pruebas de Escenarios Reales:** Simulan escenarios del mundo real, ayudando a garantizar que el sistema se comportará como se espera en condiciones normales.
- **Facilitación de Cambios:** Con tests de integración bien definidos, los cambios en el código pueden realizarse con mayor confianza, ya que se puede verificar rápidamente que las interacciones entre componentes siguen funcionando.

Justificación técnica sobre el porqué nuestro TP emplea test de integración entre los 2 componentes solicitados

1) **¿Que es un componente en arquitectura de microservicios?**

En una arquitectura de microservicios, cada microservicio es un **componente** responsable de una funcionalidad específica del sistema.
Cada componente es autónomo y se puede desplegar independientemente.
Se pueden desarrollar en diferentes lenguajes de programación y se escalan individualmente.
Por lo tanto, podemos decir que nuestros microservicios para el TP-2 son componentes de un sistema de compra/venta programado en lenguaje Java.

Microservicio de vendedores https://github.com/marilaumedici/arqSoft2ModuloVendedor
Microservicio de compradores https://github.com/marilaumedici/arqSoft2ModuloUsuario
Microservicio de notificaciones https://github.com/natirodriguez/arqSoft2ModuloNotificacion

Dentro de cada microservicio también encontramos componentes, denominados subcomponentes, como los controllers (que manejan las peticiones http), los servicios (que implementan la lógica del negocio), los repositorios (que administran los accesos a la base de datos), DTOs, entidades, etc.

2) **¿Que es un test de integración?**

Un **test de integración** es una prueba que verifica que los componentes reales del sistema funcionan correctamente cuando se integran entre sí. A diferencia de los tests unitarios, que prueban unidades aisladas (como una sola clase o método), los tests de integración aseguran que las piezas trabajen bien juntas sin el uso de mocks ya que no necesitan simular comportamiento. Utilizan implementaciones reales de servicios, repositorios, bases de datos, APIs, etc.

3) **¿ Cómo sabemos que los test presentados en clase son de integración por lo dicho anteriormente?**

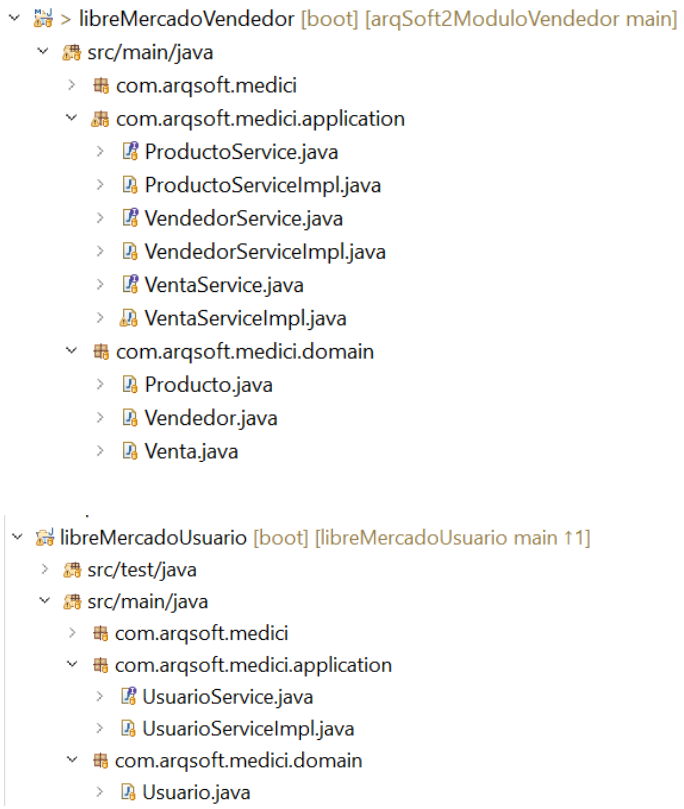
Para poder hacer un test de integración del sistema de compra/venta, es necesario probar el método de venta, ya que:

- Por lo dicho anteriormente, vamos a considerar como componentes a los 3 microservicios
- En el único lugar donde hay comunicación entre componentes es en la venta de un producto

Como estamos probando la venta de un artículo, **el test de integración se va a definir en el microservicio de vendedores**, en la clase VentaServiceIntegracionTest**,** ya que en este componente, solo existen los módulos de vendedores (módulo que solo expone endpoints

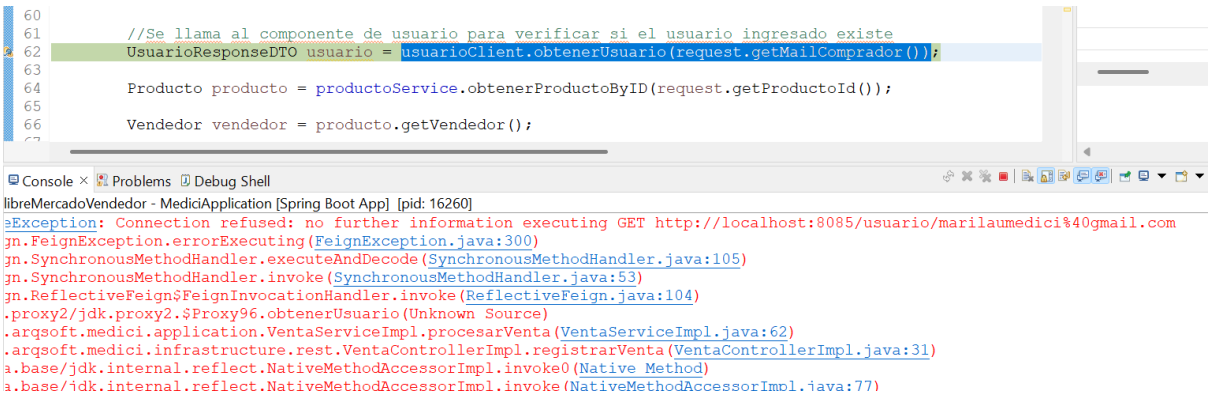
de ABM de vendedores y para poder realizar ventas) y de productos (módulo que le brinda a un vendedor métodos para poder administrar sus productos).

Al momento de realizar una venta, es necesario que validemos si el usuario que está comprando el producto existe, pero como **en el microservicio de vendedores no está implementada la lógica para poder verificar la existencia de un comprador**, es necesario recurrir al microservicio de compradores mediante peticiones http/rest.



Otra forma de validar que el test es de integración es que si el microservicio de compradores está caído o no está arriba, el test va a fallar. No solo el test va a fallar, sino que va a obtener una respuesta http al momento de tratar de validar la existencia del comprador. Esto para demostrar que se están comunicando por apis rest.

Se adjunta captura donde claramente se quiere hacer un GET, pero como el servicio de compradores está caído, da error de conexión.



Esto es debido a que el componente de vendedores quiere comunicarse con el componente de compradores y no puede.
Y lo mismo pasa al querer enviar notificaciones, ya que si el microservicio de notificaciones esta caido, tambien dara error de conexión.

El test de integracion de realizar una venta solo va a funcionar si todos los componentes están arriba.

4) ¿Cómo sabemos que en efecto se están haciendo llamados Rest entre los componentes?

En un principio, podemos decir que la clase Java VentaServiceImpl, definida en el microservicio de vendedores donde está implementado el método de venta de un producto, tiene un atributo de interfaz UsuarioCliente, denominado usuarioCliente. Cuando se realiza una venta, se llama a esa variable de instancia para verificar si el comprador existe en la base de datos o no.

Las interfaces en Java solo son plantillas que definen contratos para clases, especificando qué métodos deben implementar, pero no cómo. Por lo tanto, podemos decir que en la interfaz UsuarioCliente no está implementada la verificación de la existencia del usuario comprador. ¿Pero entonces quién lo hace?

Imagen de la interfaz UsuarioCliente y los métodos que define:

```java
@FeignClient(name = "usuario-client", url = "http://localhost:8085")
public interface UsuarioCliente {

    @GetMapping(path = "/usuario/{email}",
    produces = MediaType.APPLICATION_JSON_VALUE)
    public UsuarioResponseDTO obtenerUsuario(@PathVariable(value = "email") String mail);

    @PostMapping(path = "/usuario/",
    consumes = MediaType.APPLICATION_JSON_VALUE,
    produces = MediaType.APPLICATION_JSON_VALUE)
    public void crearUsuario(@RequestBody UsuarioDTO request);

    @DeleteMapping(path = "/usuario/{email}", produces = MediaType.APPLICATION_JSON_VALUE)
    public void eliminarUsuario(@PathVariable(value = "email") String mail);
```

Esta interfaz tiene el annotation @FeignClient para poder utilizar Feign, un cliente HTTP. Su propósito es facilitar la comunicación entre servicios en una arquitectura de microservicios, permitiendo hacer llamadas HTTP a otros servicios como si se estuvieran llamando a métodos locales. Cuando una interfaz está anotada con @FeignClient, Spring crea automáticamente una implementación del cliente HTTP basado en esa interfaz. Esta implementación usará HTTP (normalmente REST) para comunicarse con otro servicio.

En esta annotation:

- name = "usuario-client" Nombre del servicio
- url = "http://localhost:8085" Dirección del servicio al que se harán las llamadas http. Es quien implementa realmente el método obtenerUsuario

Cuando en el código se llama a usuarioClient.obtenerUsuario(request.getMailComprador()), Feing internamente:

1. Crea una solicitud http GET a http://localhost:8085/usuario/medici@gmail.com
2. Envía una solicitud
3. Recibe una respuesta JSON
4. Convierte esa respuesta en un objeto UsuarioResponseDTO

El servicio que corre en http://localhost:8085 (el microservicio de compradores) tiene un controlador real que implementa la lógica del endpoint.

Lo mismo pasa en el test de integración que cuando se inicia, el servicio remoto (http://localhost:8085 en este caso) tiene que estar levantado y accesible u obtendremos errores de conexión, como demostramos en el punto 3.

5) ¿Qué pasa si uno de los componentes está desarrollado en otra tecnología que no sea Java Spring Boot? ¿Cómo sería la comunicación con ese componente?

Si el microservicio de compradores no estuviera hecho en Java y estuviera hecho en otra tecnología no pasaría nada. Lo único que importa es que sigan respetando los contratos definidos por la interfaz UsuarioCliente y que sea un servicio REST.

El microservicio hecho en Java no sabrá ni le importará en qué código está programado el servicio que atiende ese endpoint. Mientras devuelva un JSON válido y mientras que el formato coincida con el que espera la clase UsuarioResponseDTO, todo va funcionar igual.

Como dijimos anteriormente, Feing solo envía peticiones http y espera recibir un JSON de respuesta. Mientras el servicio que esté del otro lado cumpla con esos requisitos no habrá problema.

Modelo de datos

A raíz de nuestra decisión de cambiar de una arquitectura hexagonal a un enfoque de microservicios, se hizo evidente la necesidad de replantear nuestro modelo de datos. En la arquitectura hexagonal original, usábamos una base de datos no relacional, específicamente MongoDB Atlas, centralizada y compartida. Este diseño, aunque funcional para nuestra aplicación inicial, presentaba desafíos que contradecían los principios fundamentales de los microservicios:
Acoplamiento no deseado: tener una base de datos compartida generaba un fuerte acoplamiento entre servicios. Cualquier cambio en la estructura de nuestras colecciones impactaba a múltiples partes de nuestra aplicación, dificultando de esta manera los despliegues y aumentaba el riesgo de tener que realizar regresiones.
Falta de flexibilidad tecnológica: Estábamos limitados a una única tecnología de base de datos para todo el sistema.

La elección de utilizar bases de datos no relacionales en nuestros microservicios, nos ofrece la flexibilidad, escalabilidad y rendimiento necesarios para adaptarnos a los cambios en los requisitos del negocio y a las necesidades de nuestros usuarios. Mongodb tiene la capacidad de escalar y manejar grandes volúmenes de datos, lo que la convierte en una solución ideal para soportar el crecimiento y la evolución de nuestra plataforma.

Para no saturar la base de datos, se decidió dividirla en dos, una para el microservicio de usuarios compradores y otra para el microservicio de vendedores; adoptando de esta manera el patrón “Base de datos por Servicio”. Para el microservicio de compradores, sus datos residen en una base de datos específica, ya que no necesita información directa del servicio de vendedores. Mientras que el servicio de vendedores también poseerá su propia base de datos, pero **solo consultará la existencia de un usuario comprador** al momento de la venta a través de la API del servicio de compradores. Al estar las entidades de venta y comprador fuertemente relacionadas, y para asegurarnos la transaccionalidad en el proceso de ventas, se decidió tener una base de datos propia por cada microservicio.

Tener dividido en dos las bases también nos permite asegurarnos que algunos microservicios seguirán funcionando en caso de que alguna falle.

Para asegurar la transaccionalidad atómica de las ventas, se decidió mantener en una misma base las colecciones de producto y venta. Si el registro de una venta fallara, necesitaríamos poder devolver el stock al producto de manera fiable, algo que sería extremadamente complejo o inviable de garantizar con **transacciones distribuidas** entre bases de datos completamente separadas.

Esta estrategia de fragmentación de bases de datos, aunque introduce cierta complejidad, nos permite maximizar la **autonomía de los servicios**, la **escalabilidad independiente** y la **resiliencia** general del sistema.

Forma de visualización de los datos creados en Mongodb:

- **producto:**

```
_id: ObjectId('6814c148cba7d2529a984fe4')
nombre : "shampoo"
descripcion : "descripcion"
precio : 200.5
stock : 100
categoria : "BAZAR"
estado : "DISPONIBLE"
vendedor : DBRef('vendedor', 'nati@gmail.com')
_class : "com.arqsoft.medici.domain.Producto"
```

- **usuario:**

```
_id: "nati@gmail.com"
nombre : "Nati"
apellido : "Rodriguez"
estado : "ACTIVO"
_class : "com.arqsoft.medici.domain.Usuario"
```

- **vendedor:**

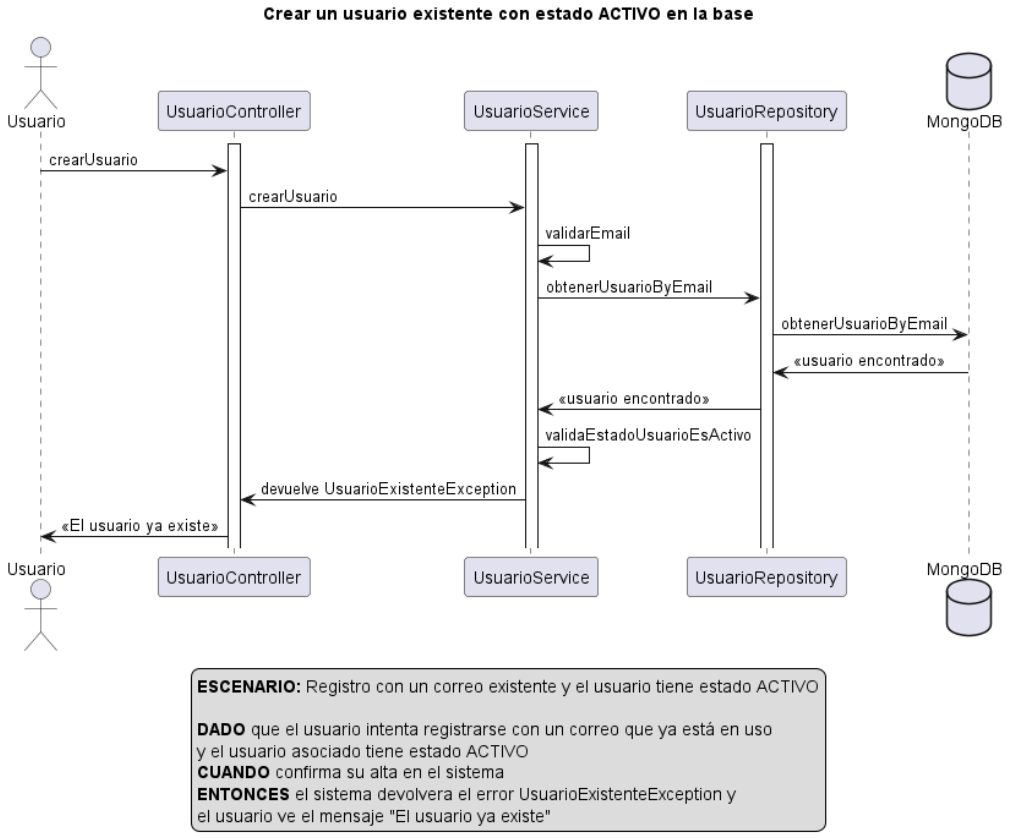
```
_id: "nati@gmail.com"
razonSocial : "Jar"
estado : "ACTIVO"
_class : "com.arqsoft.medici.domain.Vendedor"
```

- **venta:**

```
_id: ObjectId('681fd89df2974c2da09b76c1')
productoId : "681fd4da893c652c769f6e5b"
mailVendedor : "juarezconfort@gmail.com"
mailComprador : "marilaumedici@gmail.com"
fechaVenta : 2025-05-10T22:50:37.402+00:00
precioIndividual : 250000
precioFinal : 250000
cantidadComprada : 1
_class : "com.arqsoft.medici.domain.Venta"
```

Diagrama de secuencia

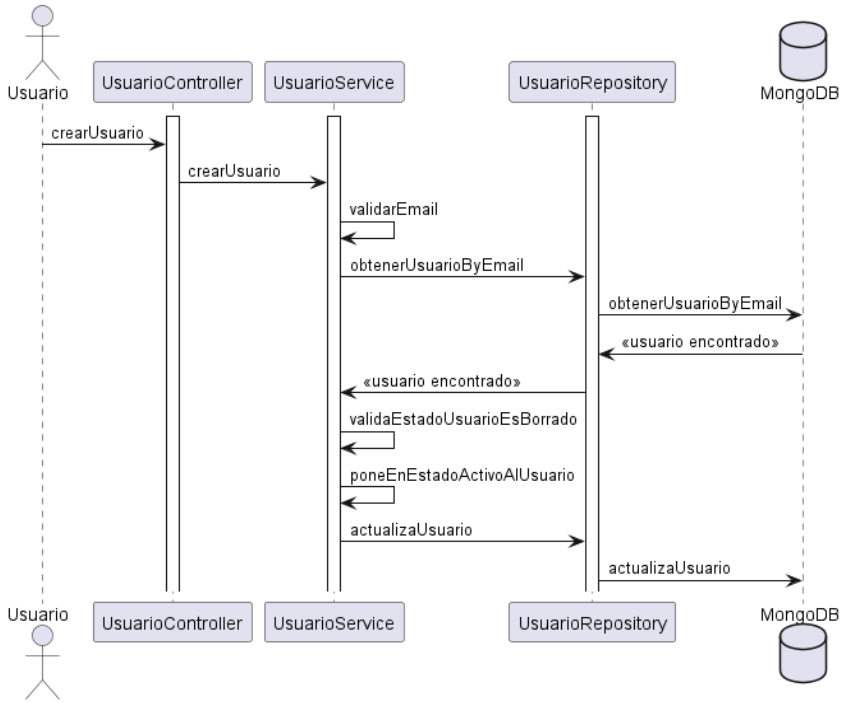
Usuario



ESCENARIO: Registro con un correo existente y el usuario tiene estado ACTIVO

DADO que el usuario intenta registrarse con un correo que ya está en uso
y el usuario asociado tiene estado ACTIVO
CUANDO confirma su alta en el sistema
ENTONCES el sistema devolvera el error UsuarioExistenteException y
el usuario ve el mensaje "El usuario ya existe"

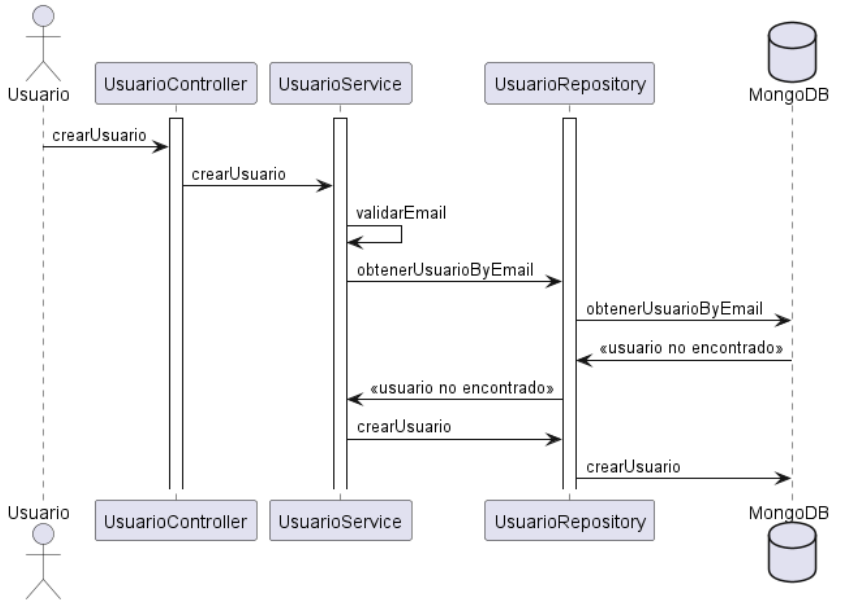
Crear un usuario existente con estado BORRADO en la base



ESCENARIO: Registro con un correo existente pero el usuario tiene estado BORRADO

DADO que el usuario intenta registrarse con un correo que ya está en uso
pero el usuario asociado tiene estado BORRADO
CUANDO confirma su alta en el sistema
ENTONCES el usuario podrá registrarse y pasará a tener estado ACTIVO

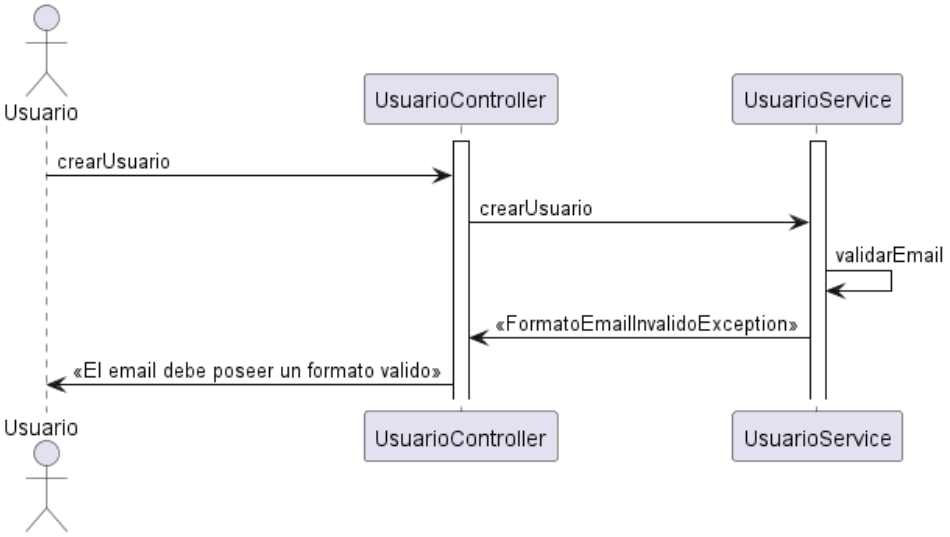
Crear un usuario inexistente en la base



ESCENARIO: Registro con un correo inexistente

COMO usuario de Libre Mercado
QUIERO poder enrolarme a la aplicacion
PARA poder comprar articulos a la venta

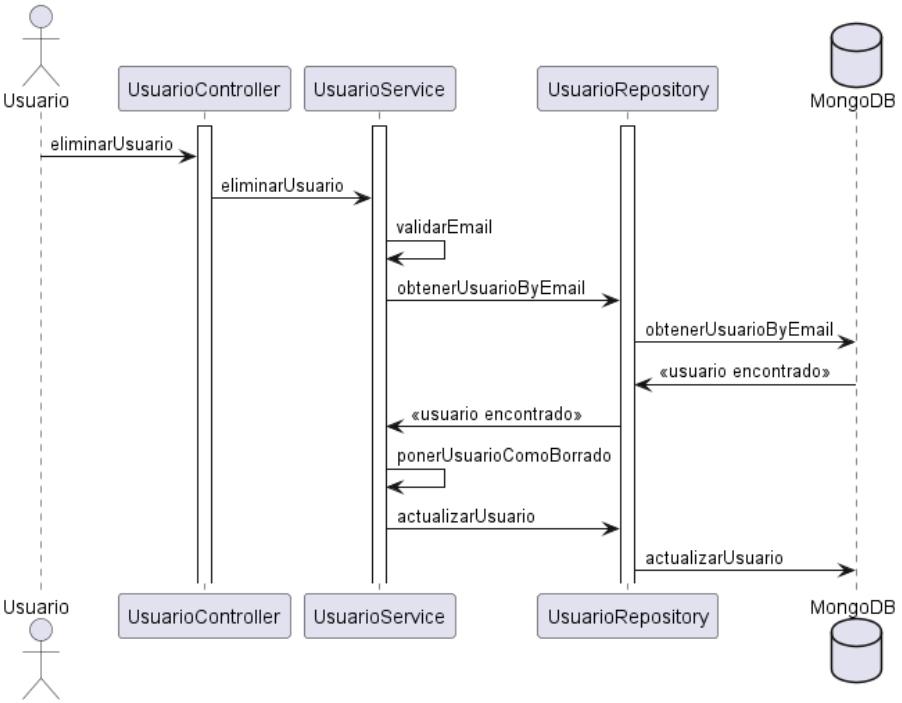
Crear un usuario con un email invalido



ESCENARIO: Registro con un correo que es vacio o con un formato no valido

DADO que el usuario intenta registrarse con un correo invalido
CUANDO confirma su alta en el sistema
ENTONCES el sistema devolvera el error FormatoEmailInvalidoException
Y el usuario vera el mensaje de error "El email debe poseer un formato valido"

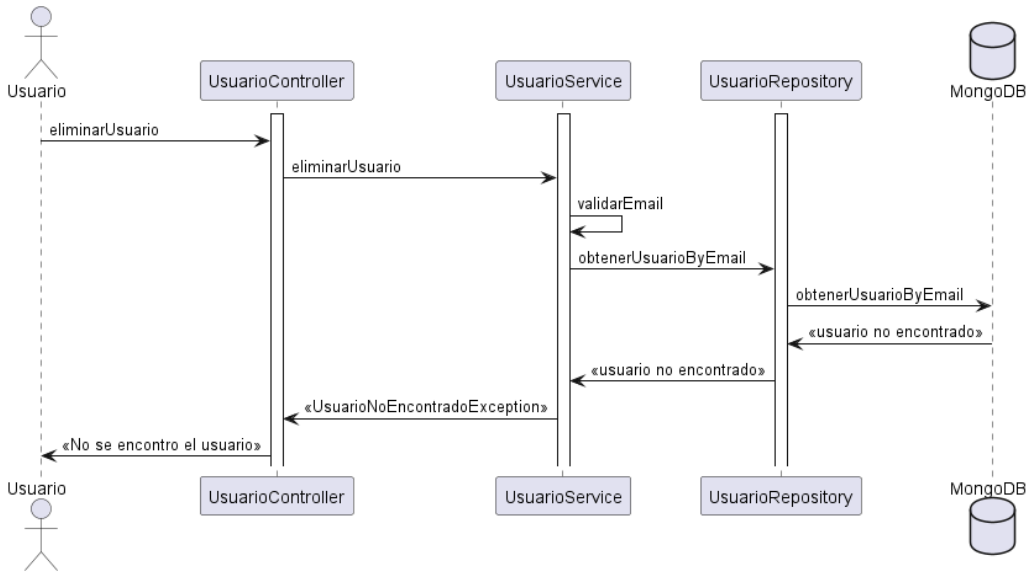
Eliminar un usuario existente en la base



ESCENARIO: Borrado logico de un usuario existente

COMO usuario de Libre Mercado
QUIERO poder desenrolarme de la aplicacion
PARA que ya no se pueda transaccionar con mi usuario

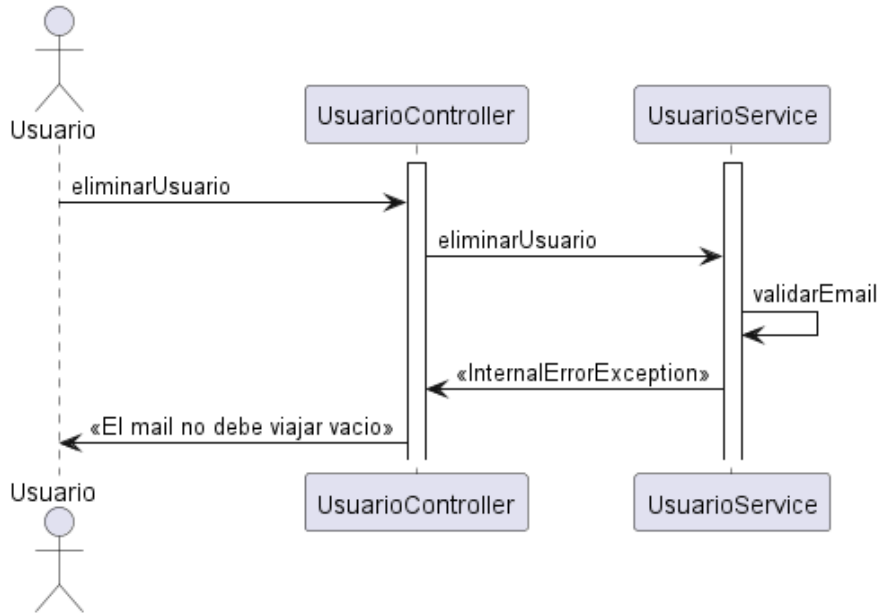
Eliminar un usuario inxistente en la base



ESCENARIO: Borrado logico de un usuario inexistente

DADO que el mail no tiene un usuario asociado en la base
CUANDO confirma su eliminacion
ENTONCES el sistema devolvera el error "No se encontro el usuario"

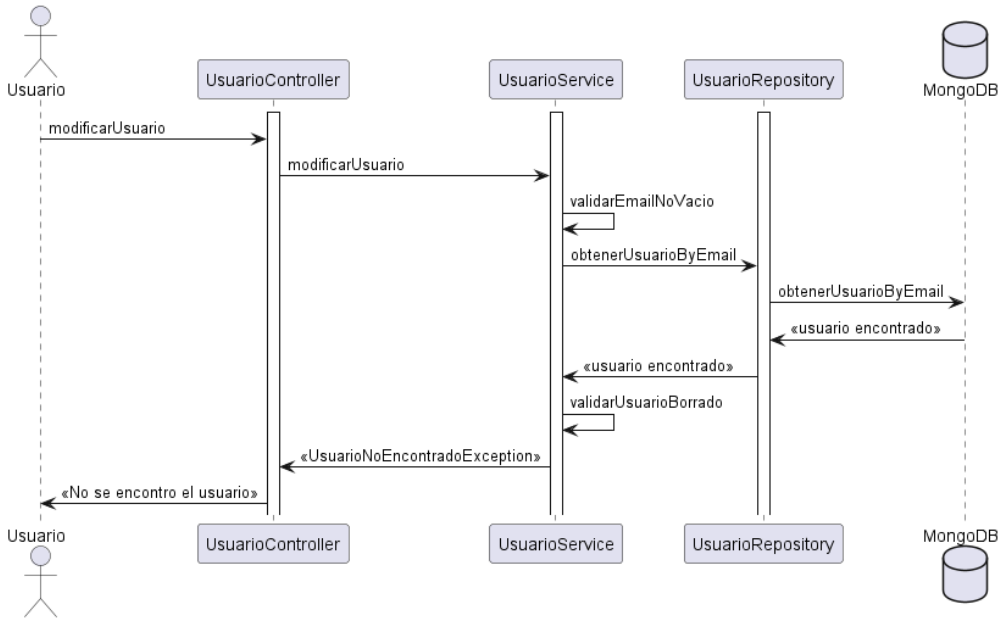
Eliminar un usuario sin pasar un mail



ESCENARIO: Borrado logico de un usuario sin pasar un mail

DADO que no se pasa un mail al servicio
CUANDO confirma su eliminacion
ENTONCES el sistema devolvera el error "El mail no debe viajar vacio"

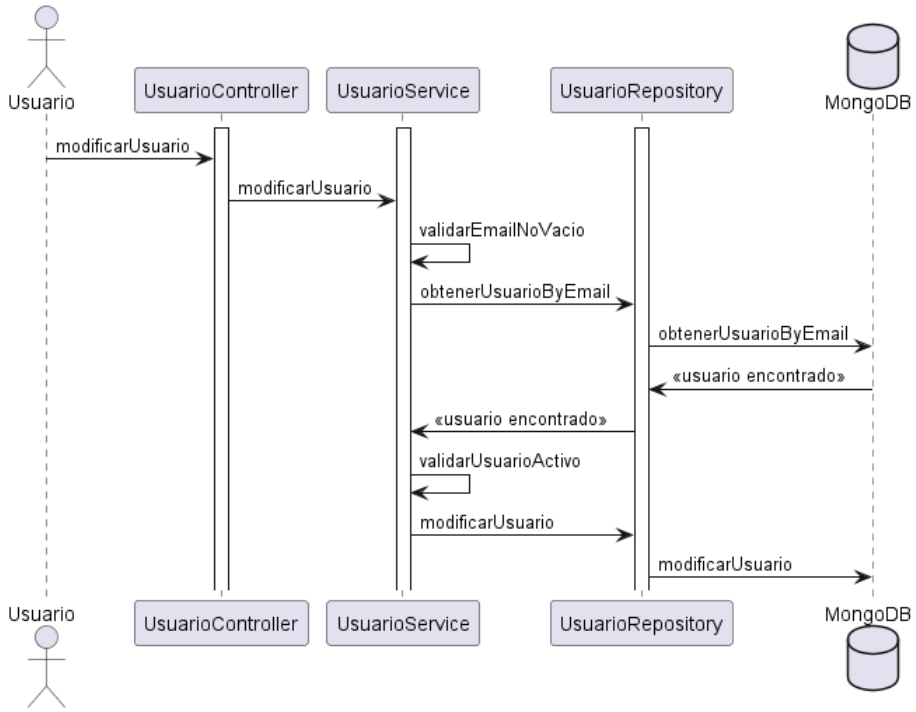
Modificar un usuario existente en la base con estado BORRADO



ESCENARIO: Modificacion de usuario existente BORRADO

DADO que el usuario intenta modificar sus datos pero tiene estado BORRADO
CUANDO confirma la modificacion
ENTONCES el sistema devolvera el error "No se encontro el usuario"

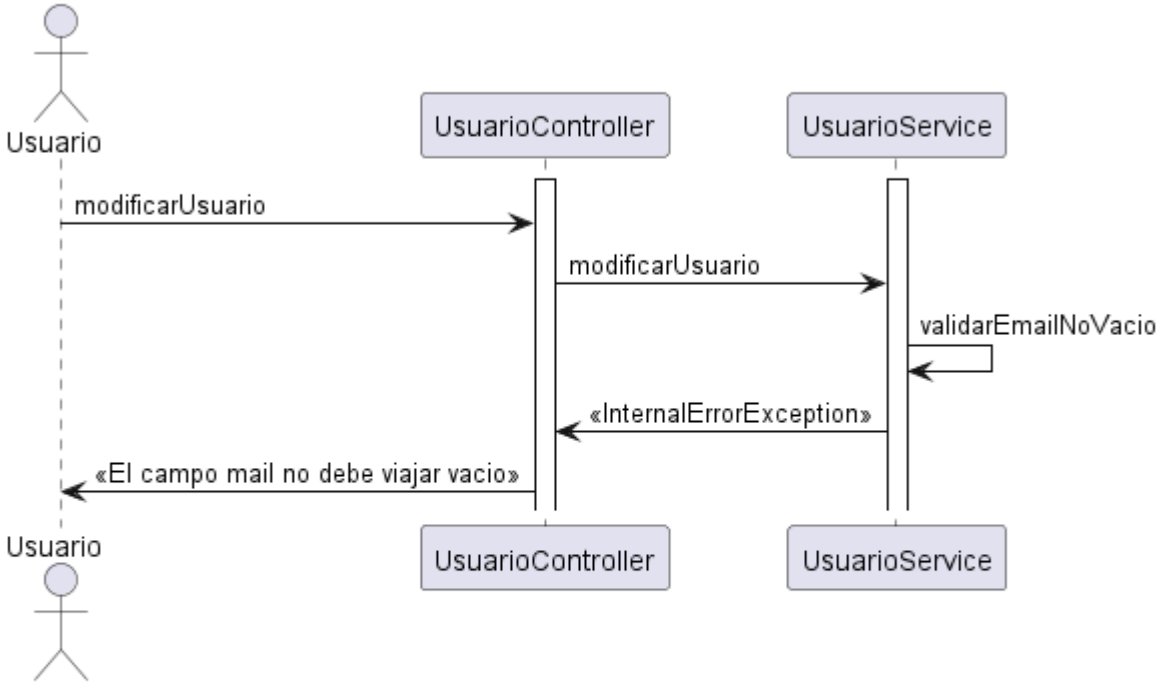
Modificar un usuario existente en la base



ESCENARIO: Modificacion de un usuario ACTIVO existente

COMO usuario de Libre Mercado
QUIERO poder modificar los datos de mi usuario
PARA poder tener actualizado mis datos en el sistema

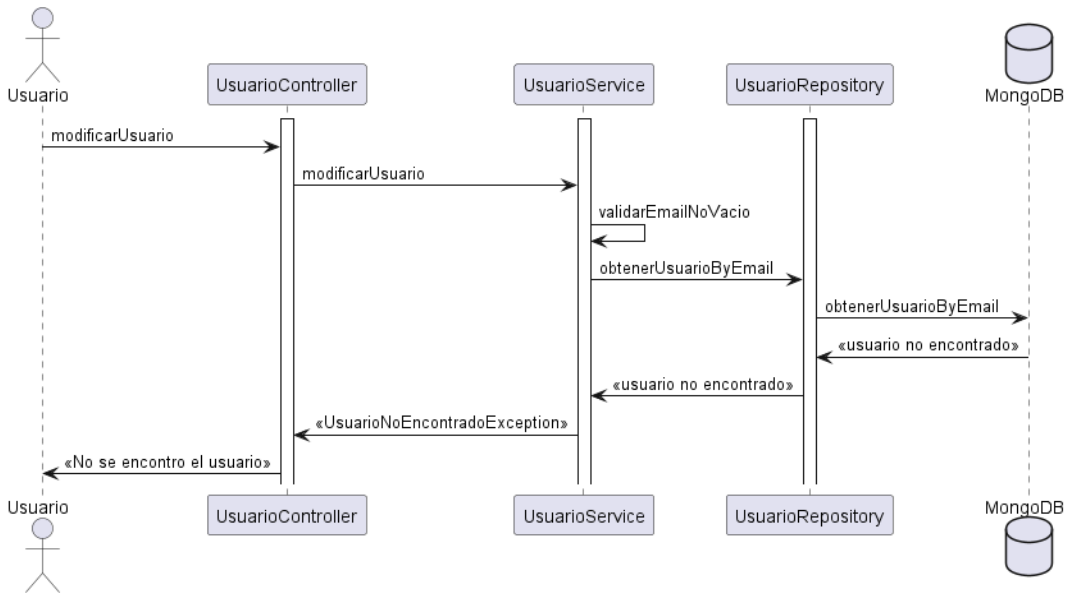
Modificar un usuario sin pasar un mail



ESCENARIO: No se brinda un mail

DADO que el email es vacio
CUANDO confirma la modificacion
ENTONCES el sistema devolvera el error "El campo mail no debe viajar vacio"

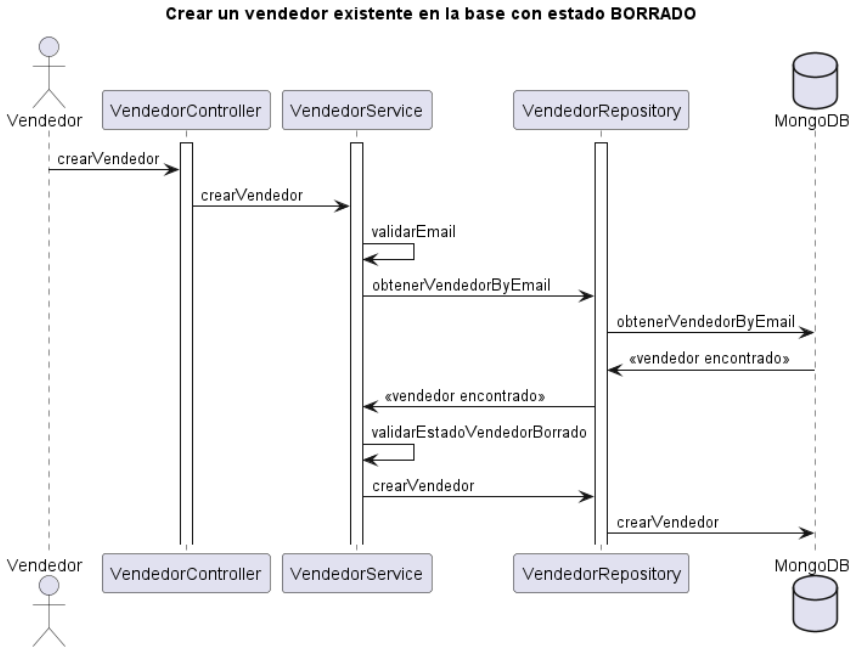
Modificar un usuario inexistente en la base



ESCENARIO: Modificacion de usuario inexistente

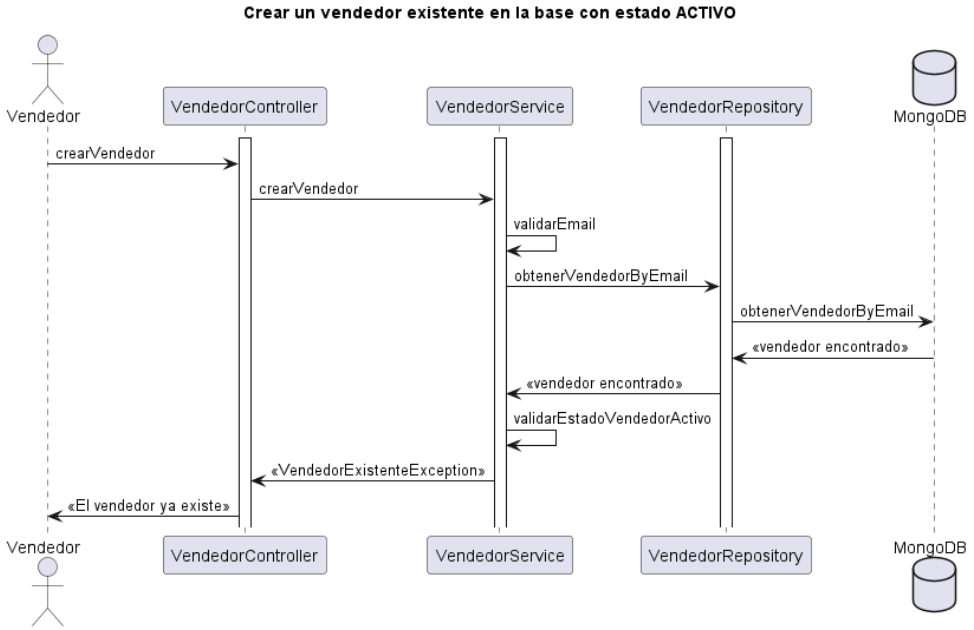
DADO que el email no esta asociado a ningun usuario en la base
CUANDO confirma la modificacion
ENTONCES el sistema devolvera el error "No se encontro el usuario"

Vendedor



ESCENARIO: Registro con un correo existente y vendedor con estado BORRADO

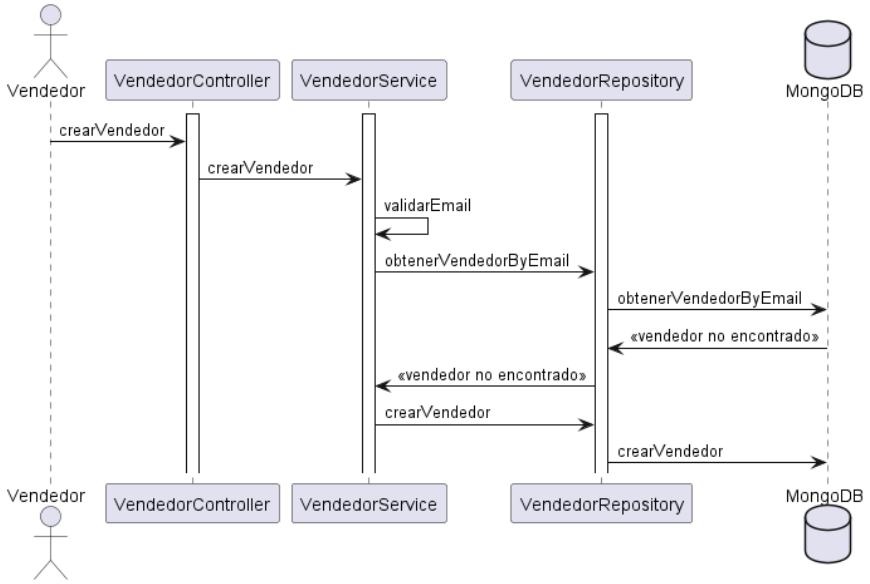
COMO vendedor de Libre Mercado
QUIERO poder re enrolarme a la aplicacion
PARA poder vender articulos a los usuarios de la aplicacion



ESCENARIO: Registro con un correo existente y vendedor con estado ACTIVO

DADO que el vendedor intenta registrarse con un correo que ya está en uso
y el vendedor asociado tiene estado ACTIVO
CUANDO confirma su alta en el sistema
ENTONCES el sistema devolvera el error VendedorExistenteException y
el vendedor ve el mensaje "El vendedor ya existe"

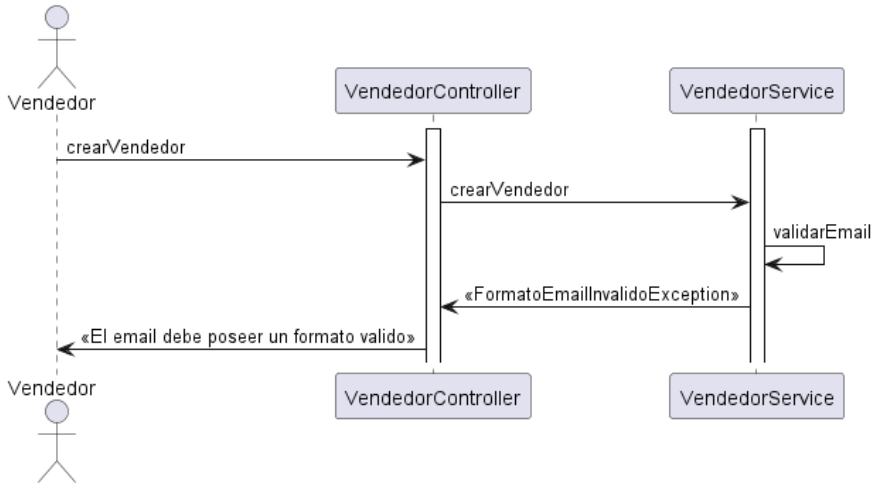
Crear un vendedor inexistente en la base



ESCENARIO: Registro con un correo inexistente

COMO vendedor de Libre Mercado
QUIERO poder enrolarme a la aplicacion
PARA poder vender articulos a los usuarios de la aplicacion

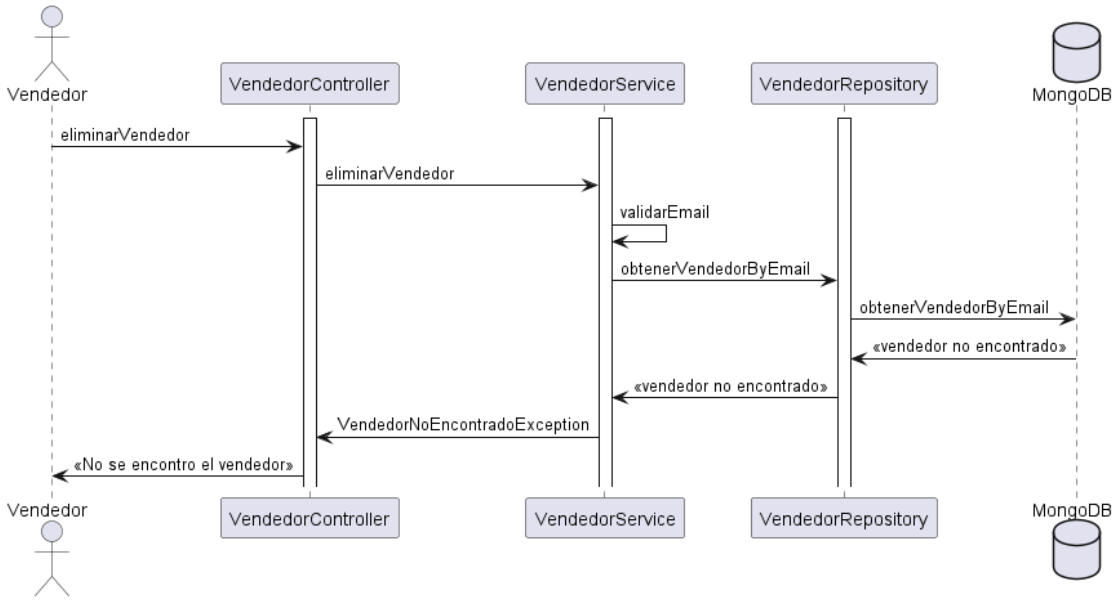
Crear un vendedor con un mail vacio o invalido



ESCENARIO: Registro con un correo invalido

DADO que el vendedor intenta registrarse con un correo invalido
CUANDO confirma su alta en el sistema
ENTONCES el sistema devolvera el error FormatoEmailInvalidoException y el vendedor vera el mensaje "El email debe poseer un formato valido"

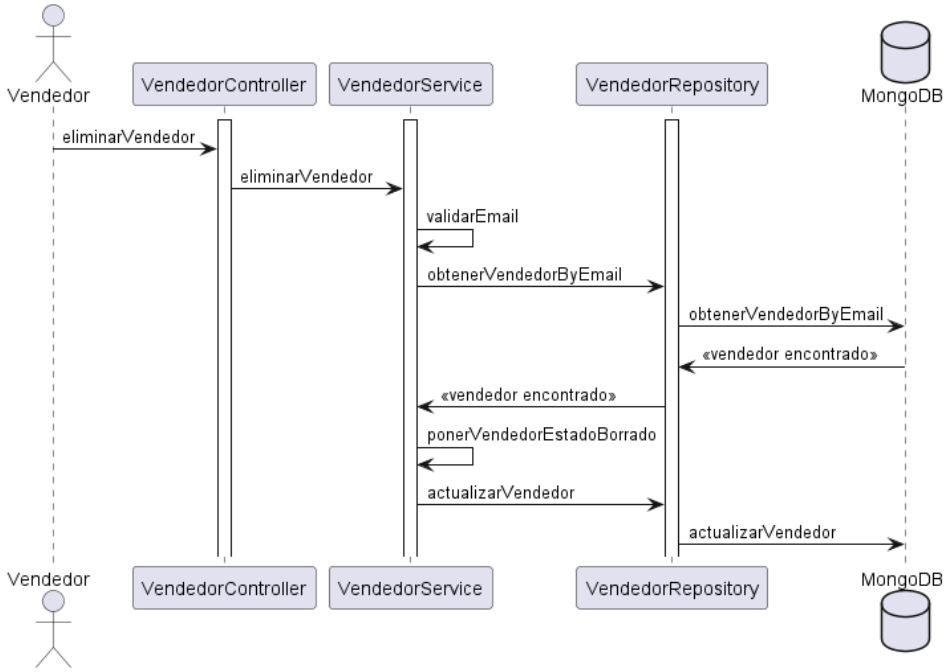
Eliminar un vendedor inexistente en la base



ESCENARIO: Borrar un vendedor inexistente en la base

DADO el mail no esta asociado a ningun vendedor en la base
CUANDO confirma la eliminacion
ENTONCES el sistema devolvera el error VendedorNoEncontradoException
Y el usuario vera el mensaje de error "No se encontro el vendedor"

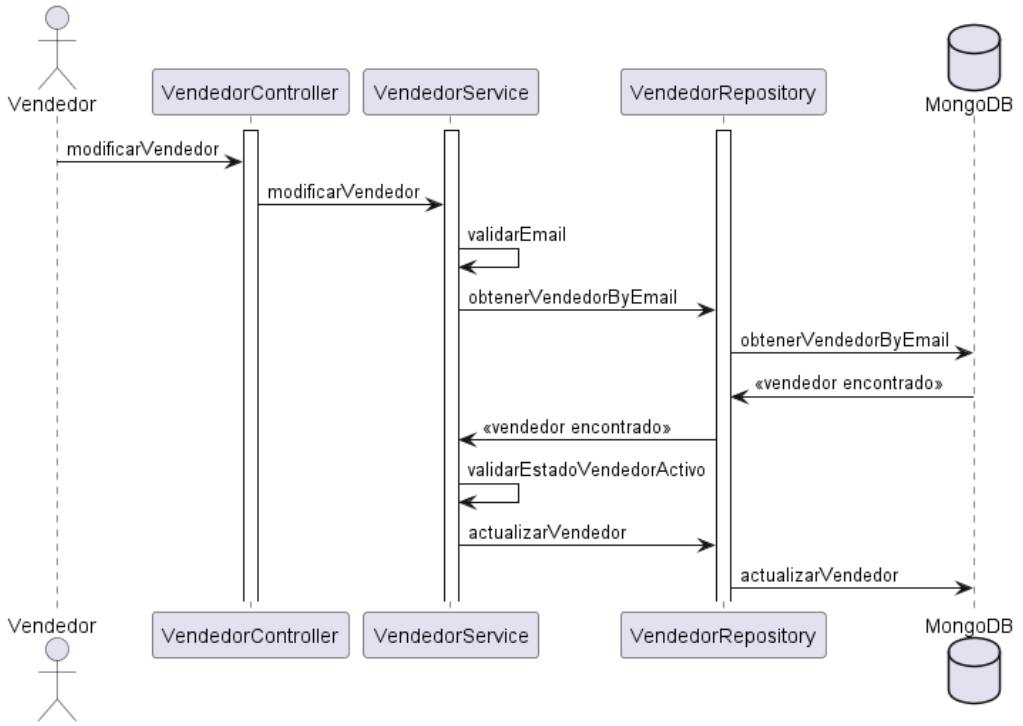
Eliminar un vendedor existente en la base



ESCENARIO: Borrar un vendedor existente en la base

COMO vendedor de Libre Mercado
QUIERO poder borrar mi usuario de la aplicacion
PARA que ya no se pueda transaccionar desde mi cuenta

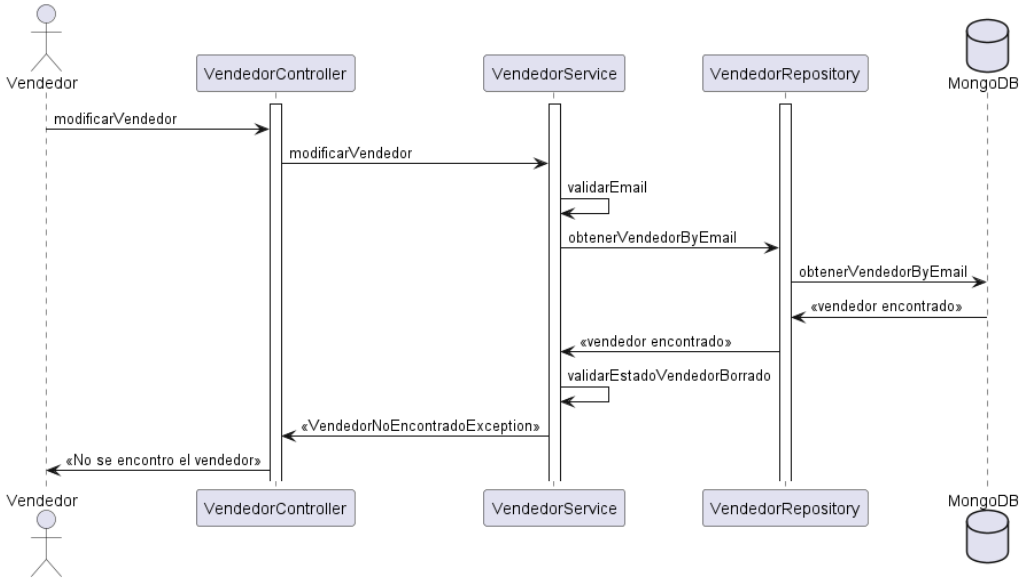
Modificar un vendedor existente en la base



ESCENARIO: Modificar vendedor ACTIVO

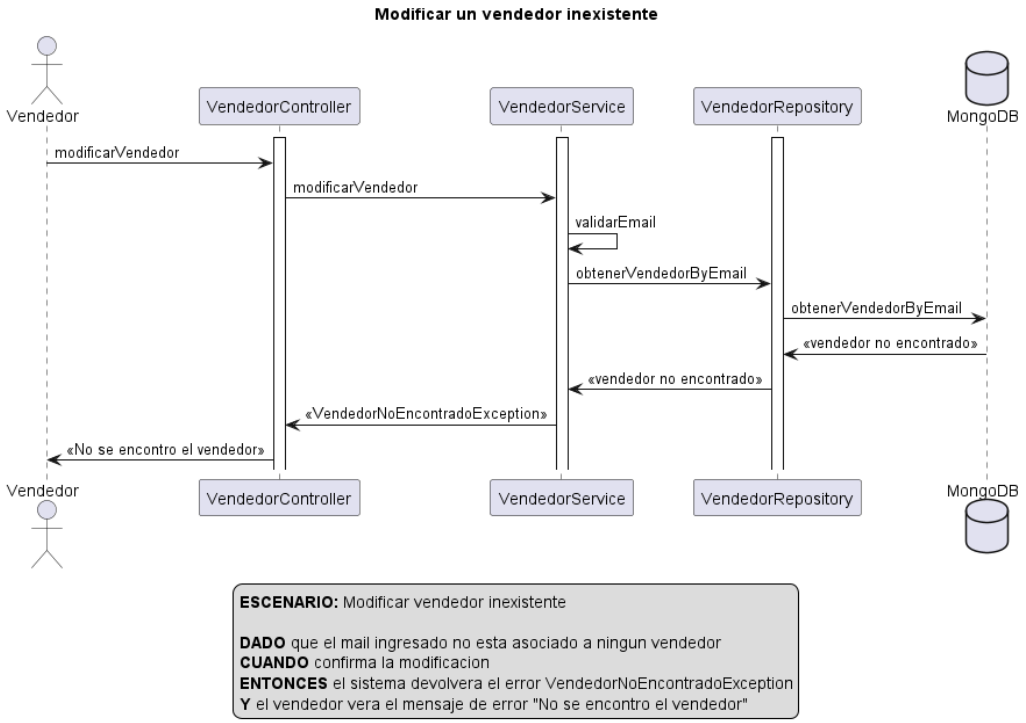
COMO vendedor de Libre Mercado
QUIERO poder modificar mis datos
PARA poder tener mis datos de contacto actualizados

Modificar un vendedor existente con estado BORRADO



ESCENARIO: Modificar vendedor BORRADO

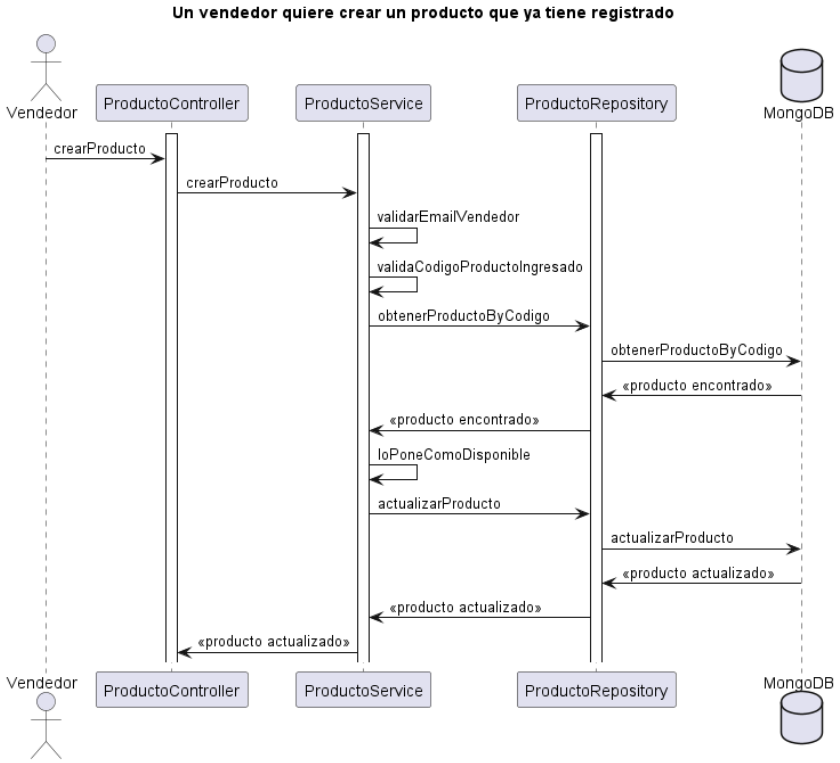
DADO que el vendedor intenta modificar los datos de un usuario borrado
CUANDO confirma la modificacion
ENTONCES el sistema devolvera el error VendedorNoEncontradoException
Y el vendedor vera el mensaje de error "No se encontro el vendedor"



ESCENARIO: Modificar vendedor inexistente

DADO que el mail ingresado no esta asociado a ningun vendedor
CUANDO confirma la modificacion
ENTONCES el sistema devolvera el error VendedorNoEncontradoException
Y el vendedor vera el mensaje de error "No se encontro el vendedor"

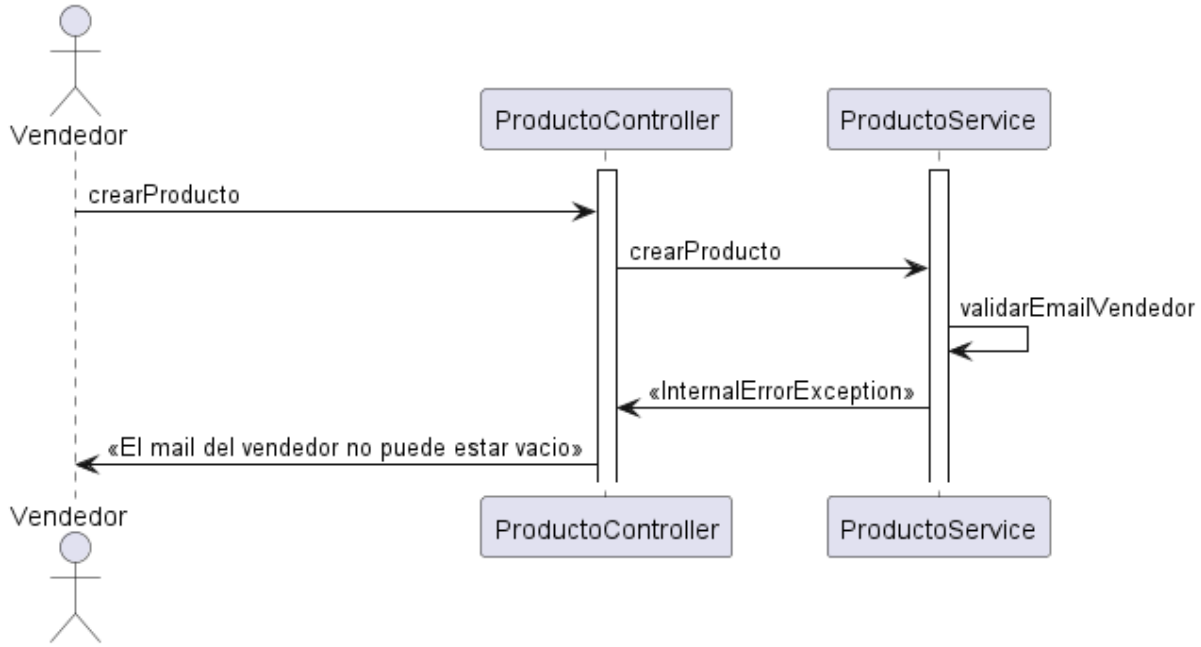
Producto



ESCENARIO: Vendedor crea un producto que ya tiene registrado

COMO vendedor de Libre Mercado
QUIERO poder crear un producto
PARA que algun usuario de la aplicacion pueda comprar de manera online lo que vendo

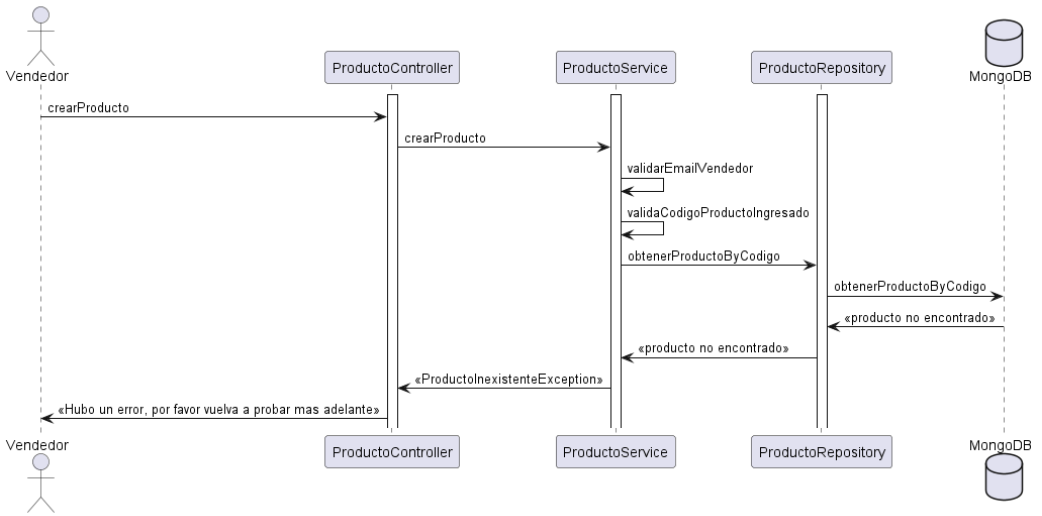
Un vendedor crea un producto pero no pasa su email



ESCENARIO: Vendedor crea un producto que no esta en base

DADO que el vendedor no pasa un mail de vendedor
CUANDO confirma el alta en el sistema
ENTONCES el sistema devolvera el error InternalErrorException
Y el vendedor vera el mensaje de error "El mail del vendedor no puede estar vacio"

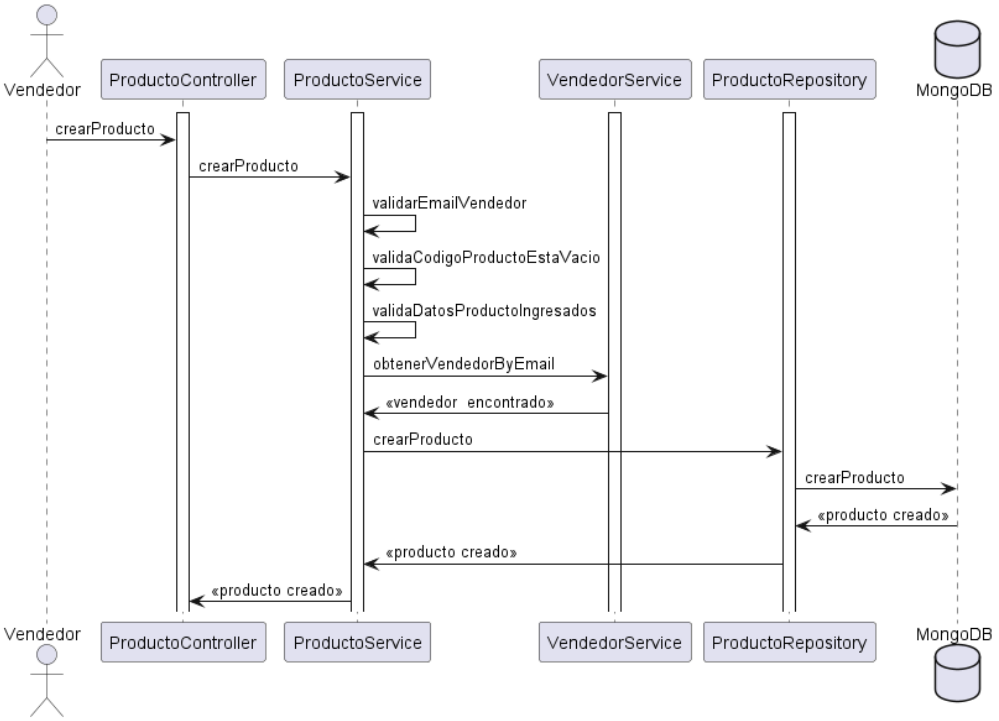
Un vendedor quiere actualizar un producto que no esta en base



ESCENARIO: Vendedor crea un producto que no esta en base

DADO que el vendedor pasa un codigo de producto que no existe
CUANDO confirma el alta en el sistema
ENTONCES el sistema devolvera el error ProductoInexistenteException
Y el vendedor vera el mensaje de error "Hubo un error, por favor vuelva a probar mas adelante"

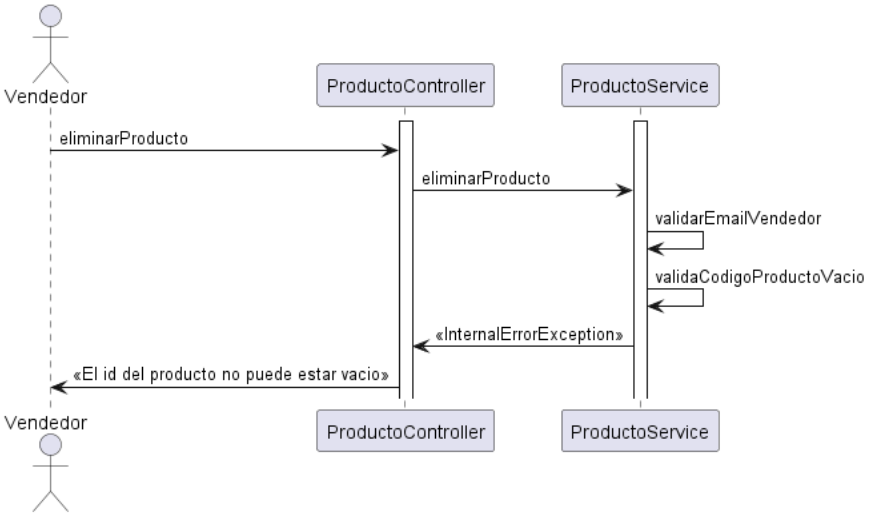
Un vendedor crea un nuevo producto en la base



ESCENARIO: Vendedor crea un nuevo producto

COMO vendedor de Libre Mercado
QUIERO poder crear un producto
PARA que algun usuario de la aplicacion pueda comprar de manera online lo que vendo

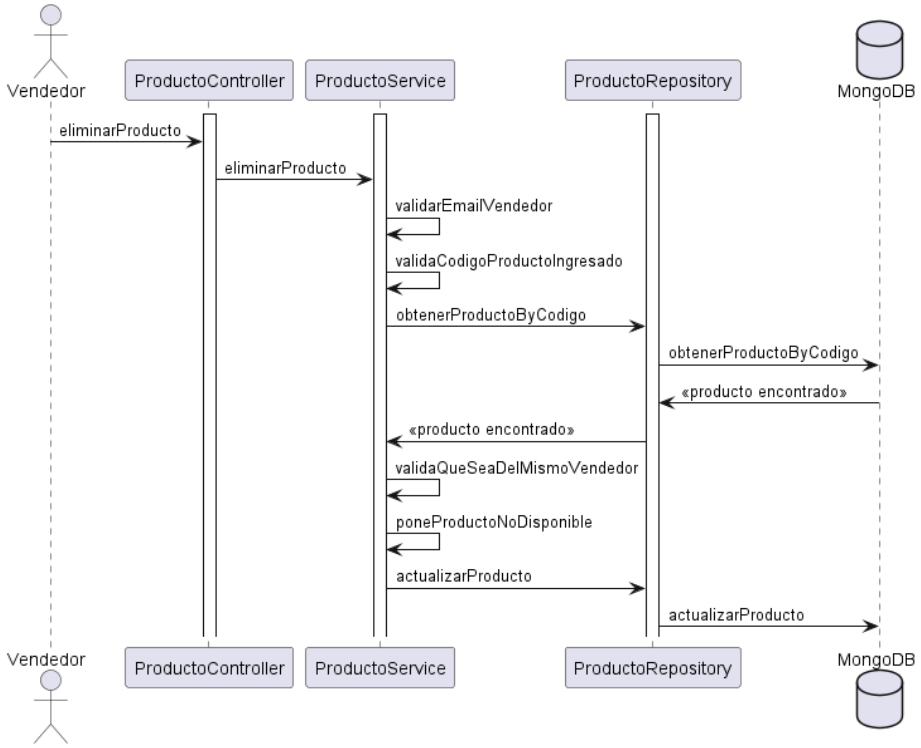
Un vendedor quiere eliminar un producto pero no pasa el codigo de producto



ESCENARIO: Vendedor trata de eliminar un producto y no pasa el codigo

DADO que un vendedor trata de eliminar un producto y no pasa el codigo
CUANDO confirma la eliminacion en el sistema
ENTONCES el sistema devolvera el error InternalErrorException
Y el vendedor vera el mensaje de error "El id del producto no puede estar vacio"

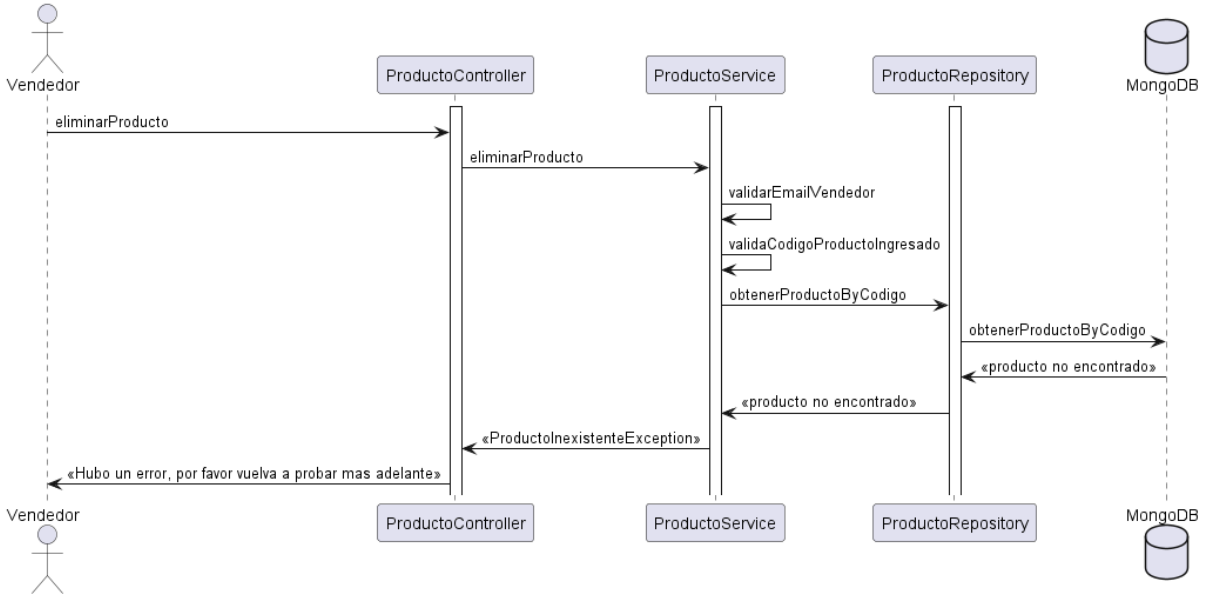
Un vendedor quiere eliminar un producto que ya tiene registrado



ESCENARIO: Vendedor elimina un producto que ya tiene registrado

COMO vendedor de Libre Mercado
QUIERO poder eliminar un producto
PARA que algun usuario de la aplicacion no compre algun producto que ya no tengo

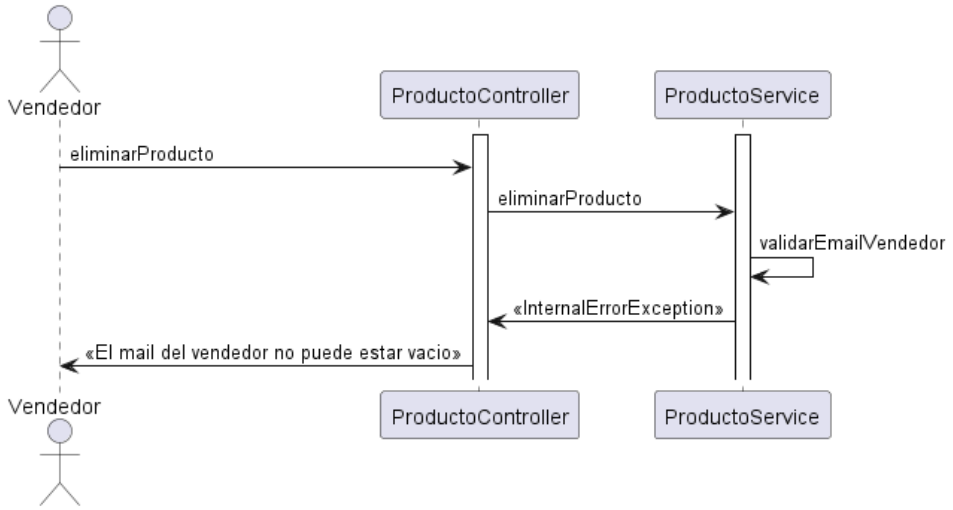
Un vendedor quiere eliminar un producto que no tiene registro asociado al codigo de producto



ESCENARIO: Vendedor elimina un producto que no existe en base

DADO que se trata de eliminar un producto que no existe en base
CUANDO confirma la eliminacion en el sistema
ENTONCES el sistema devolvera el error ProductoInexistenteException
Y el vendedor vera el mensaje de error "Hubo un error, por favor vuelva a probar mas adelante"

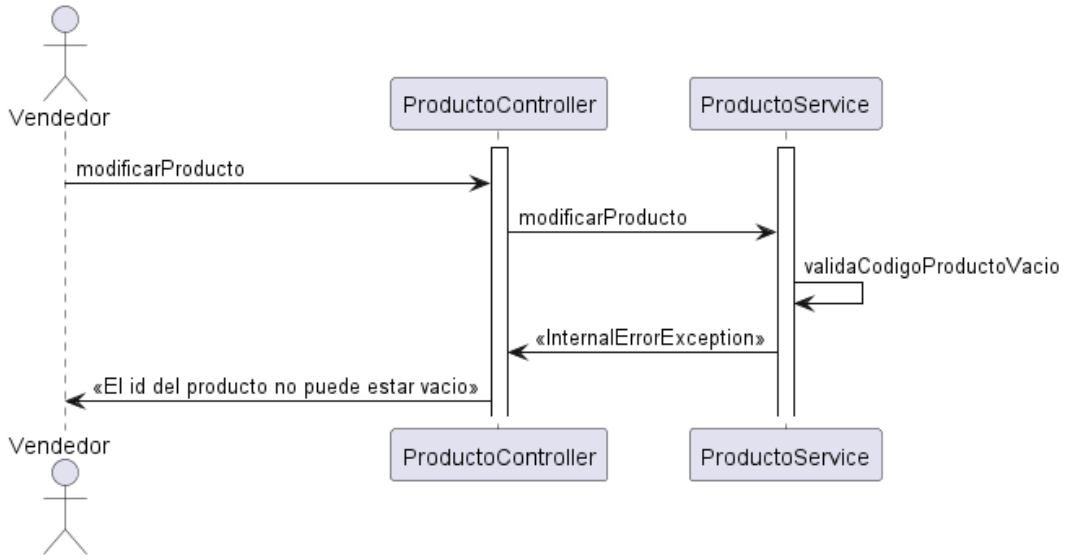
Un vendedor quiere eliminar un producto pero no pasa el mail del vendedor



ESCENARIO: Vendedor trata de eliminar un producto y no pasa el mail del vendedor

DADO que un vendedor trata de eliminar un producto y no pasa el mail del vendedor
CUANDO confirma la eliminacion en el sistema
ENTONCES el sistema devolvera el error InternalErrorException
Y el vendedor vera el mensaje de error "El mail del vendedor no puede estar vacio"

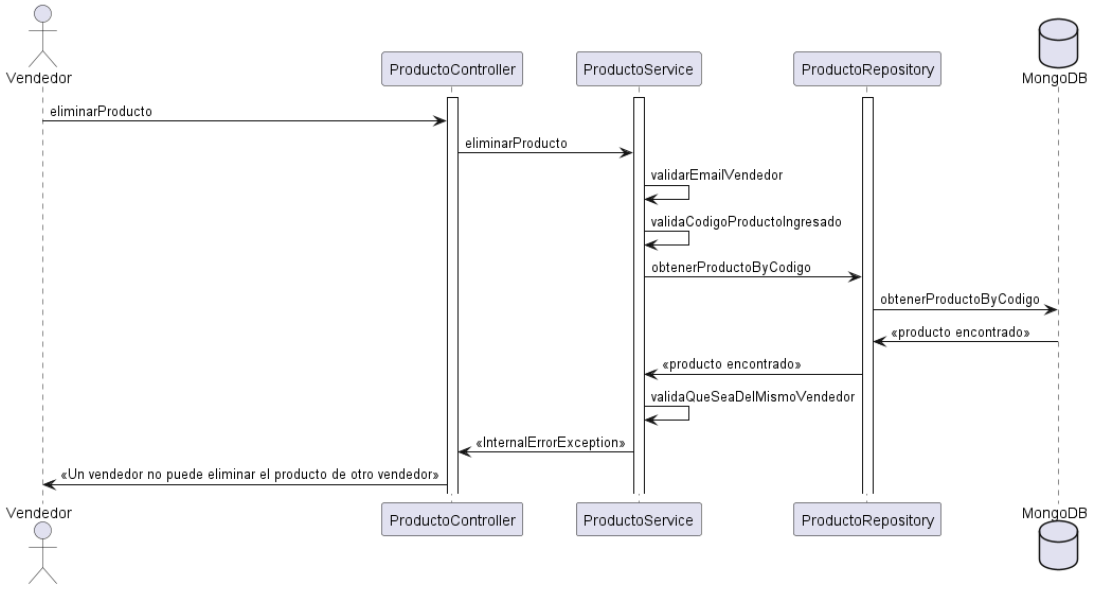
Un vendedor quiere modificar un producto pero no pasa el codigo del producto



ESCENARIO: Vendedor modifica un producto sin pasar el codigo de cual

DADO el codigo de producto es vacio
CUANDO confirma la modificacion del producto en el sistema
ENTONCES el sistema devolvera el error InternalErrorException
Y el vendedor vera el mensaje de error "El id del producto no puede estar vacio"

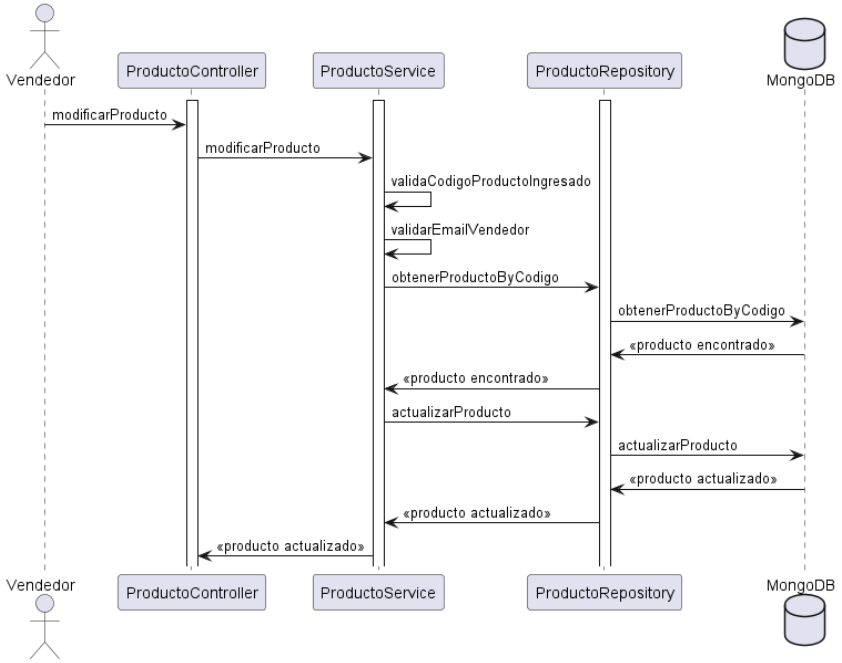
Un vendedor quiere eliminar un producto que no le pertenece



ESCENARIO: Vendedor trata de eliminar un producto que no le pertenece

DADO que un vendedor trata de eliminar un producto que no le pertenece
CUANDO confirma la eliminacion en el sistema
ENTONCES el sistema devolvera el error InternalErrorException
Y el vendedor vera el mensaje de error "Un vendedor no puede eliminar el producto de otro vendedor"

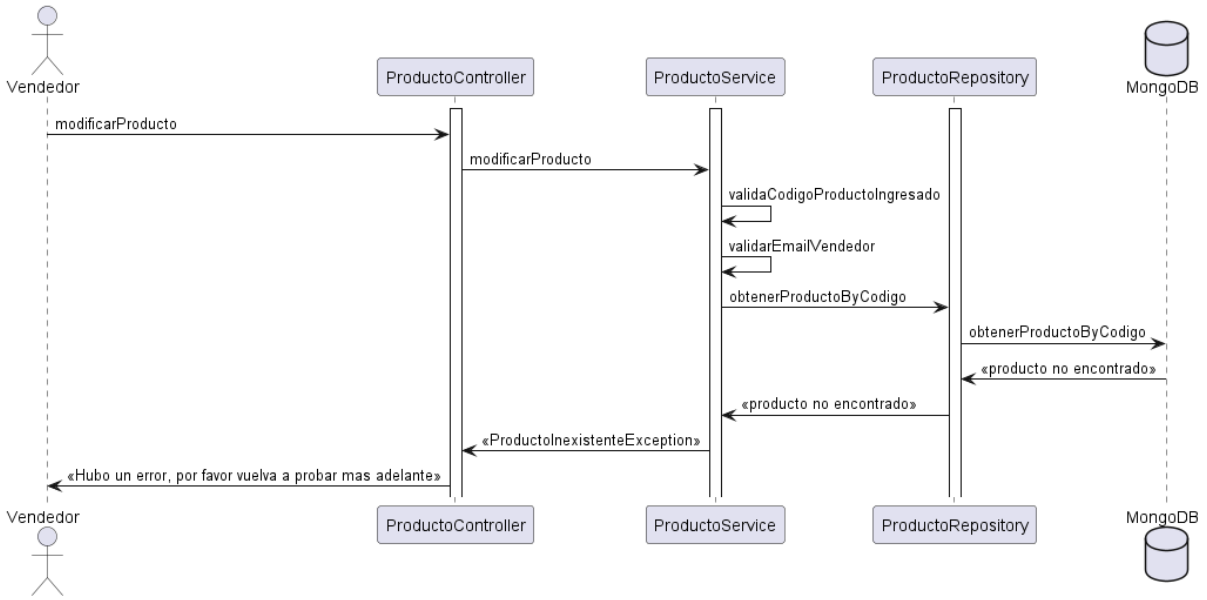
Un vendedor quiere modificar un producto que ya tiene registrado



ESCENARIO: Vendedor modifica un producto que ya tiene registrado

COMO vendedor de Libre Mercado
QUIERO poder actualizar los datos de un producto
PARA que algun usuario de la aplicacion pueda visualizar los datos actualizados del producto

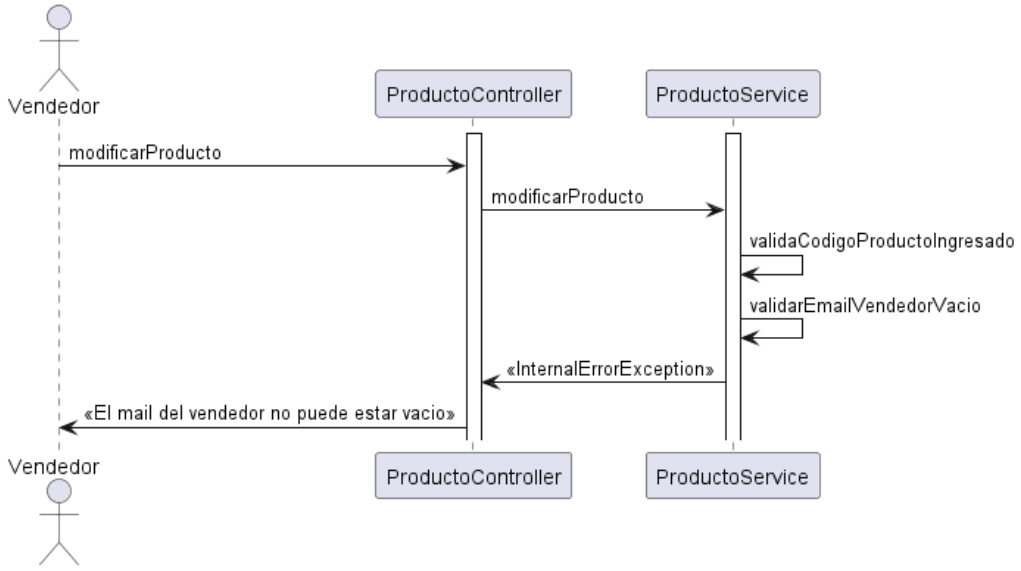
Un vendedor quiere modificar un producto cuyo codigo no esta en base



ESCENARIO: Vendedor modifica un producto que no existe en base

DADO que el codigo de producto ingresado no existe en la base
CUANDO confirma la modificacion del producto en el sistema
ENTONCES el sistema devolvera el error ProductoInexistenteException
Y el vendedor vera el mensaje de error "Hubo un error, por favor vuelva a probar mas adelante"

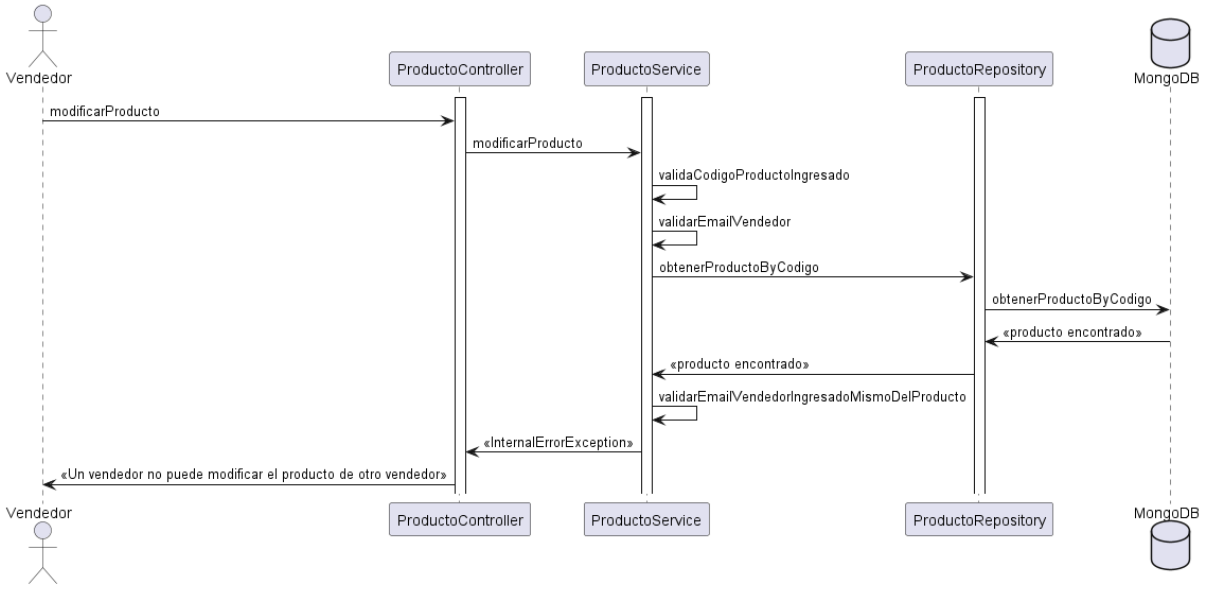
Un vendedor quiere modificar un producto pero no pasa el mail del vendedor



ESCENARIO: Vendedor modifica un producto sin pasar el codigo de cual

DADO el mail del vendedor es vacio
CUANDO confirma la modificacion del producto en el sistema
ENTONCES el sistema devolvera el error InternalErrorException
Y el vendedor vera el mensaje de error "El mail del vendedor no puede estar vacio"

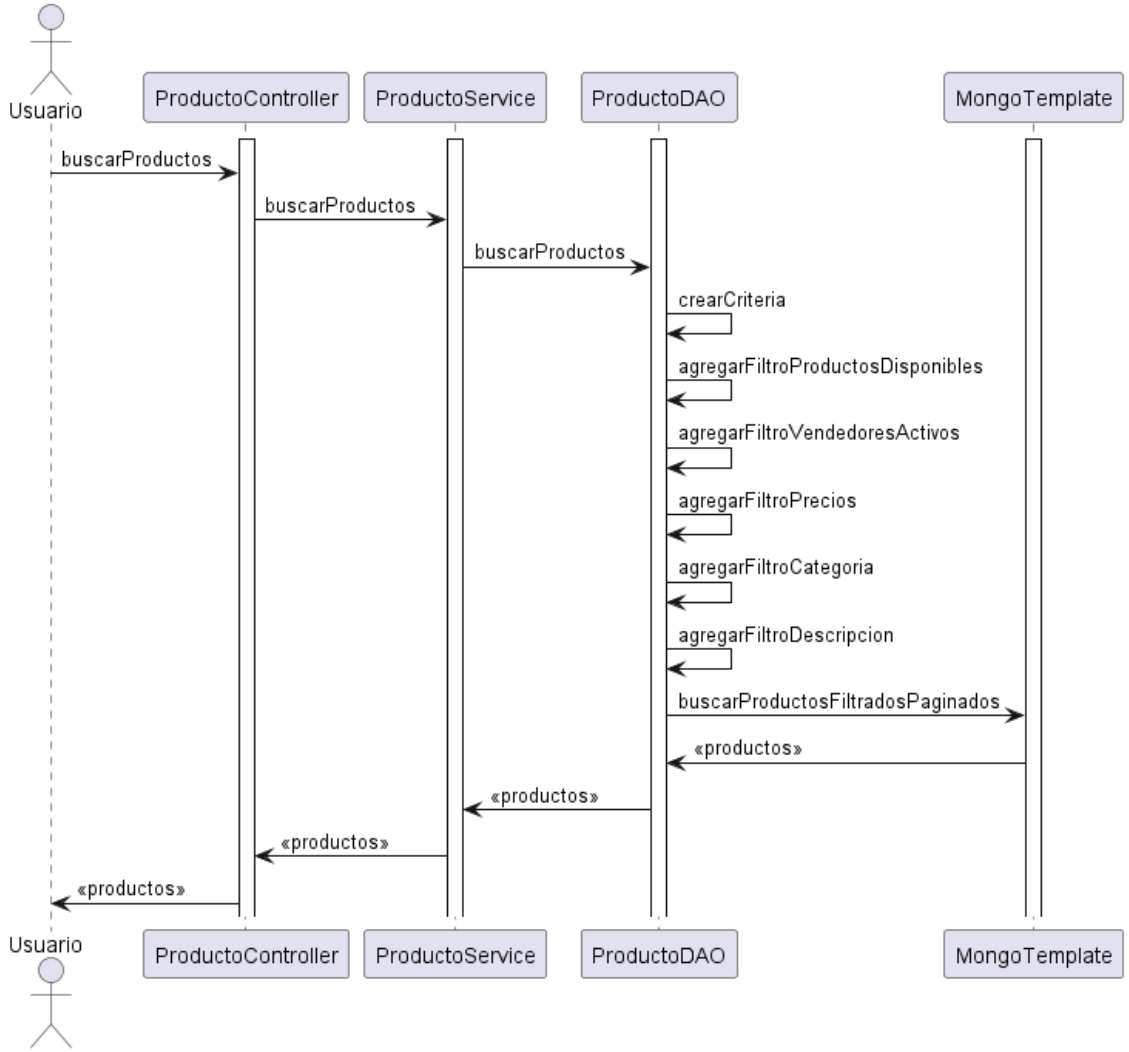
Un vendedor quiere modificar un producto de otro vendedor



ESCENARIO: Vendedor modifica un producto de otro vendedor

DADO que el mail vendedor del producto a modificar es diferente al mail vendedor ingresado
CUANDO confirma la modificacion del producto en el sistema
ENTONCES el sistema devolvera el error InternalErrorException
Y el vendedor vera el mensaje de error "Un vendedor no puede modificar el producto de otro vendedor"

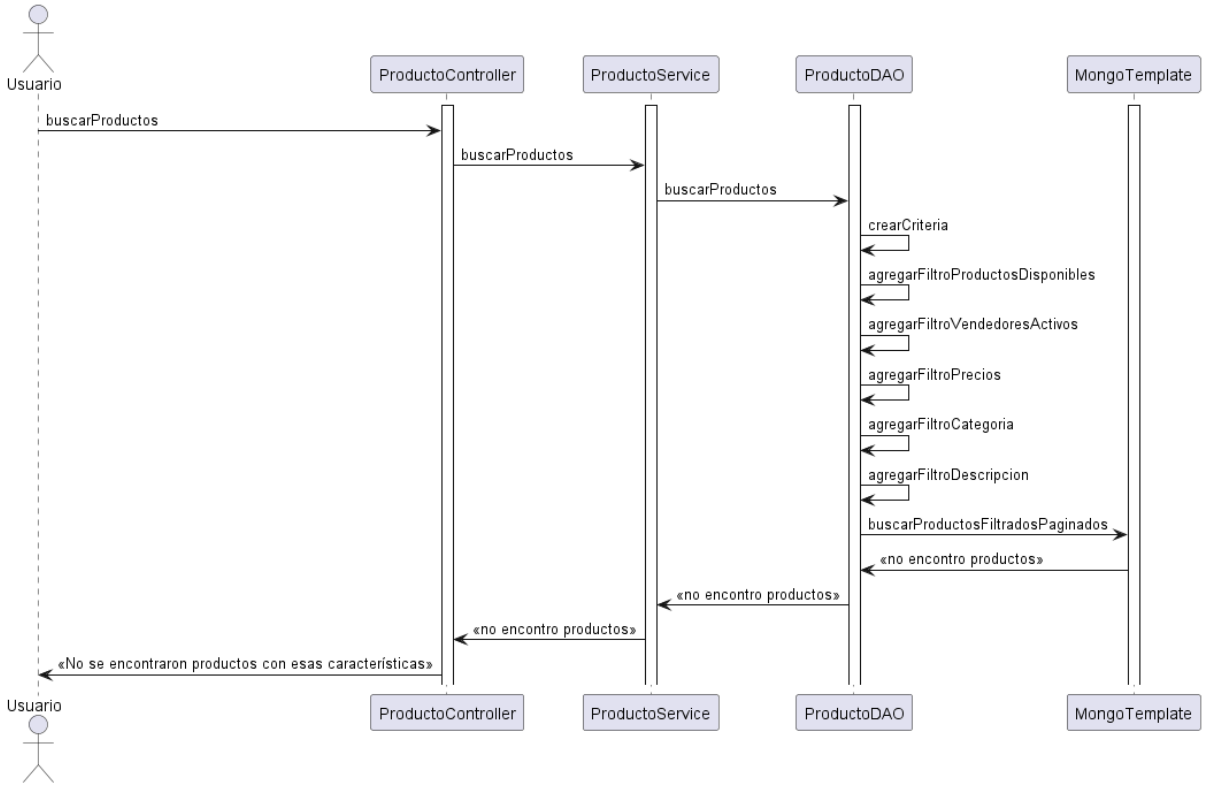
Un usuario realiza la busqueda de productos



ESCENARIO: Usuario quiere ver productos para su compra

COMO usuario de Libre Mercado
QUIERO poder buscar varios productos
PARA poder elegir de entre ellos uno poder comprarlo

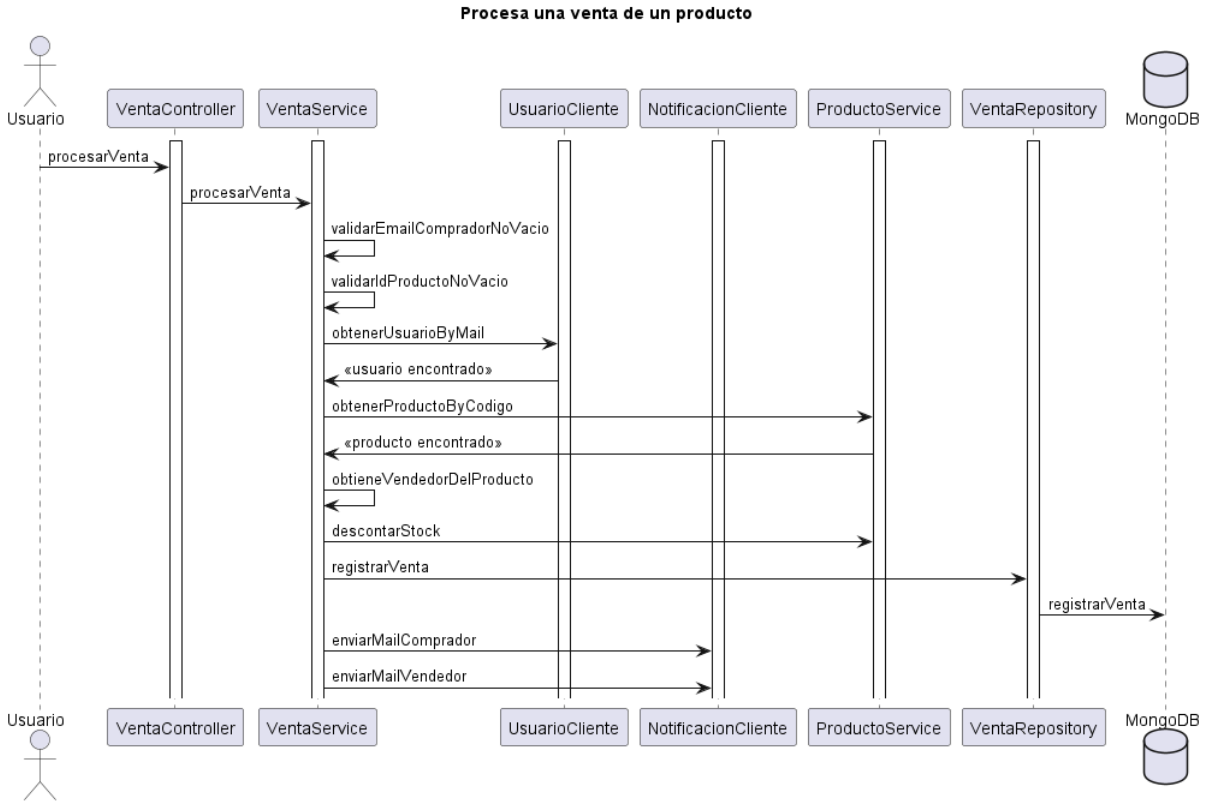
Un usuario realiza la busqueda de productos con filtros pero no trae ningun resultado



ESCENARIO: Usuario quiere ver productos para su compra pero no los haya

DADO que el usuario pone alguna caracteristica que ningun producto existente en la base posee
CUANDO confirma la busqueda en el sistema
ENTONCES el sistema devolvera una lista vacia sin productos
Y el usuario vera el mensaje de error "No se encontraron productos con esas características"

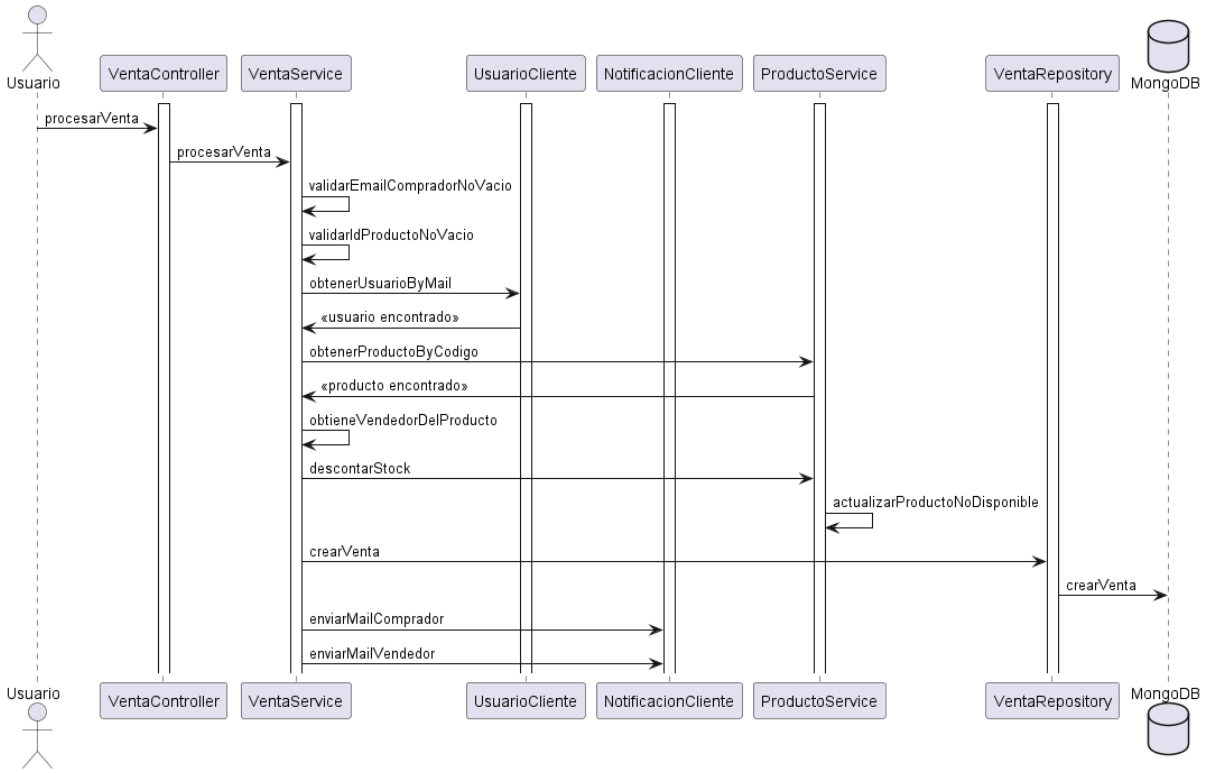
Venta



ESCENARIO: Generar una venta
PRECONDICION: El vendedor esta ACTIVO

COMO usuario de Libre Mercado
QUIERO poder realizar la compra de un producto
PARA poder adquirir un producto

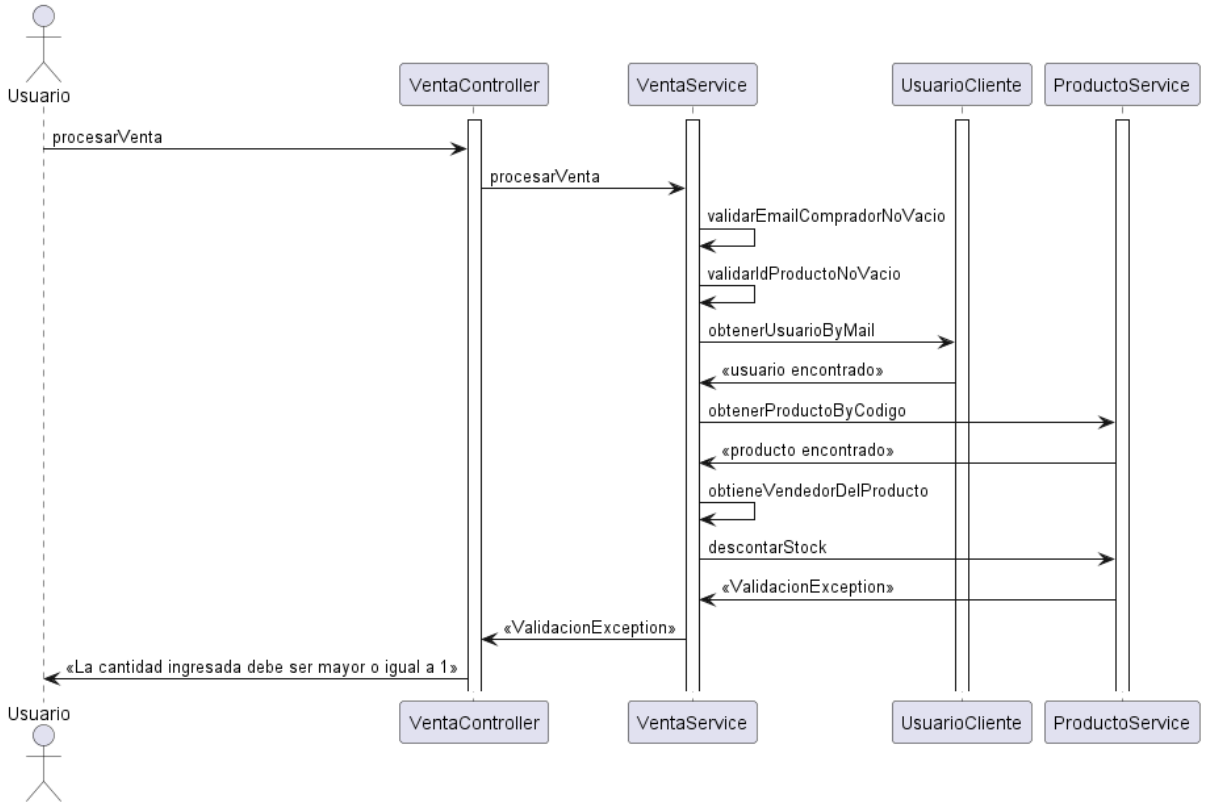
Procesa una venta de un producto que queda sin stock



ESCENARIO: Generar una venta
PRECONDICION: El vendedor esta ACTIVO

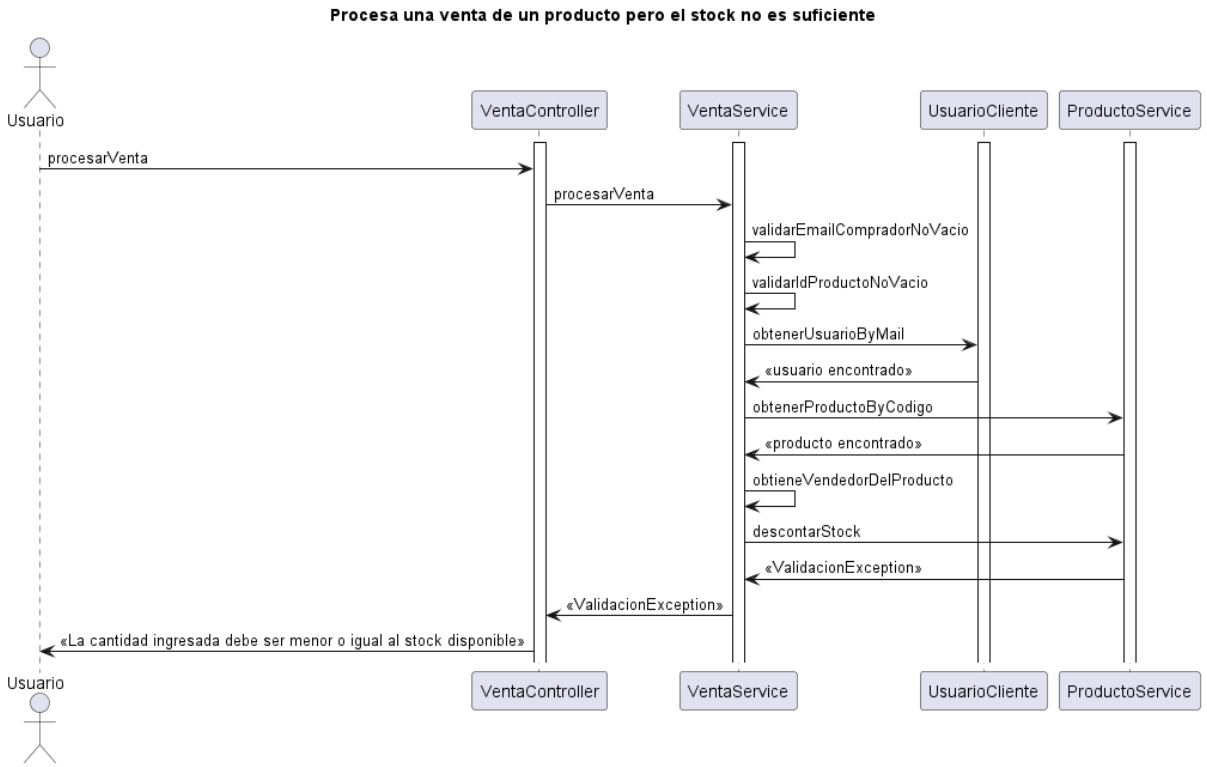
DADO que el usuario va a comprar toda la cantidad del producto
CUANDO confirma la compra en el sistema
ENTONCES se realizaraa la compra
Y el producto quedara en estado NO DISPONIBLE

Procesa una venta de un producto pero el stock no es suficiente



ESCENARIO: Generar una venta
PRECONDICION: El vendedor esta ACTIVO

DADO que el usuario ingreso una cantidad menor igual a 0
CUANDO confirma la compra en el sistema
ENTONCES el sistema devolvera el error ValidacionException
Y el usuario vera el mensaje de error "La cantidad ingresada debe ser mayor o igual a 1"



ESCENARIO: Generar una venta
PRECONDICION: El vendedor esta ACTIVO

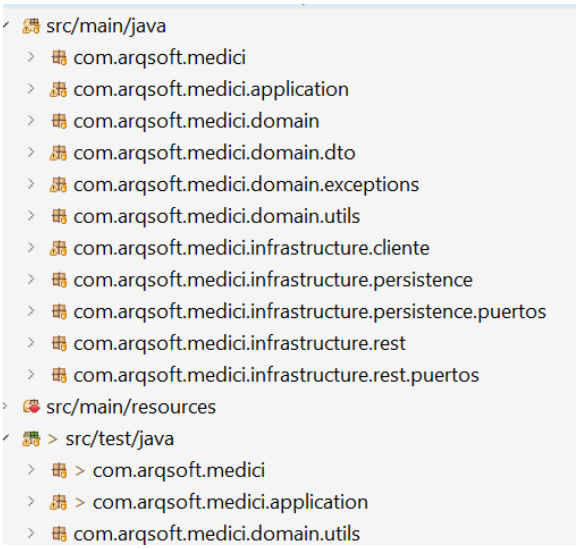
DADO que el usuario ingreso una cantidad mayor al que hay
CUANDO confirma la compra en el sistema
ENTONCES el sistema devolvera el error ValidacionException
Y el usuario vera el mensaje de error "La cantidad ingresada debe ser menor o igual al stock disponible"

Especificaciones de las APIs

Para asegurarnos que nuestros microservicios sean interoperables, fáciles de consumir y consistentes, hemos adoptado los principios de diseño RESTful (*Representational State Transfer*) para la exposición de nuestras APIs. Estos principios ayudan a crear APIs escalables, eficientes y fáciles de utilizar.

- **Cliente-servidor:** El cliente y el servidor están completamente separados y son independientes. En nuestro sistema la interfaz de usuario (cliente) y el backend (servidor) están desacoplados, por lo que ambos se comunican mediante solicitudes y respuestas.
- **Stateless:** La comunicación entre el cliente y el servidor debe de ser sin estado, lo que implica que no se almacenan ni se comparte información entre peticiones. Siendo las mismas independientes y contienen sólo la información que es necesaria para procesarlas. En nuestra aplicación, en las solicitudes mandamos solamente la información necesaria, y no dependemos de datos almacenados en el servidor.
- **Cacheable:** cuando sea necesario, es posible almacenar los datos de nuestras entidades en caché, ya sea del lado cliente o del lado del servidor, para mejorar el rendimiento de nuestra aplicación.

- **Sistema en capas:** nuestra arquitectura presenta múltiples capas, donde cada capa tiene una función específica y puede interactuar con otras capas sin que el cliente conozca la estructura interna. En la siguiente imagen se puede observar como es la división en capas de cada uno de nuestros servicios.



- **Identificador Único:** cada recurso tiene un identificador único e irrepetible, no pueden existir más de un recurso con el mismo identificador en la red.
- **Uso correcto de HTTP:** Rest debe de respetar los verbos y códigos de estado para cada operación. En nuestra aplicación se podrán encontrar los siguientes métodos:
    - **GET:** Se utiliza para recuperar los datos de un recurso en específico.
    - **POST:** Se utiliza para crear un nuevo recurso.
    - **PUT:** Se utiliza para actualizar un recurso ya existente. En nuestra aplicación, si el recurso no existe, tira una excepción.
    - **DELETE:** Se utiliza para eliminar un recurso ya existente.

Listado de endpoints

1. Servicio: Usuario (/usuario)
   Descripción: Gestión de usuarios.
   Métodos soportados: Post, Put, Delete
   Códigos de respuesta:
   - ➢ 200 (Ok)
   - ➢ 400 (Bad Request)
2. Servicio: Vendedor (/vendedor)
   Descripción: Gestión de vendedores.
   Métodos soportados: Get, Post, Put, Delete
   Códigos de respuesta:
   - ➢ 200 (Ok)
   - ➢ 400 (Bad Request)
3. Servicio: Producto (/producto)
   Descripción: Gestión de productos.
   Métodos soportados: Get, Post, Put, Delete
   Códigos de respuesta:
   - ➢ 200 (Ok)

- ➢ 400 (Bad Request)

4. Servicio: Venta (/venta)
   Descripción: Registro de la venta realizada por un usuario comprador.
   Métodos soportados: Post
   Códigos de respuesta:
   - ➢ 200 (Ok)
   - ➢ 400 (Bad Request)