Super Technology

Proyecto de desarrollo multiplataforma

IES Luis vives

Programación de Servicios y Procesos

Acceso a Datos

Equipo y funciones

El presente apartado tiene como objetivo presentar a los miembros del equipo de proyecto de desarrollo. Este equipo ha trabajado arduamente para alcanzar los objetivos y metas propuestas.

Se presentarán brevemente los integrantes del equipo y sus respectivas áreas de especialización.

Equipo de Proyecto:

Mario González Resa: Representante de Equipo

Daniel Rodríguez

Roberto Blázquez Martín

Iván Azagra Troya

Sebastián Mendoza Acosta

Azahara Blanco Rodríguez

Alfredo Maldonado

El equipo de proyecto de desarrollo se encuentra conformado por miembros altamente capacitados y especializados en distintas áreas.

Cada integrante ha aportado valiosas habilidades y conocimientos al proyecto para lograr los objetivos propuestos. Con el trabajo en equipo y el esfuerzo conjunto, se espera cumplir satisfactoriamente con los requisitos del proyecto y satisfacer las necesidades del cliente.

Para mayor informacion de los miembros del equipo y distintos proyectos realizados detallamos los git hub de cada uno de sus componentes:

Mario Resa

https://github.com/Mario999X

Daniel Rodríguez

https://github.com/Idliketobealoli

Sebastián Mendoza

https://github.com/SebsMendoza

Alfredo Rafael Maldonado]

https://github.com/reyalfre

Azahara Blanco

https://github.com/Azaharabl

Roberto Blazquez

https://github.com/xBaank

Iván Azagra

https://github.com/IvanAzagraTroya

Resumen de Proyecto: proyecto y los objetivos que se buscan alcanzar para la empresa.

El proyecto consiste en desarrollar una arquitectura basada en microservicios con APIs para una empresa de piezas, aparatos electrónicos y servicios de estos.

Se implementarán tres microservicios con bases de datos independientes para la gestión de:

Usuarios.

Productos.

Pedidos.

El objetivo del proyecto es mejorar la eficiencia en la gestión de la empresa y optimizar la experiencia del usuario al utilizar las APIs para interactuar con los servicios de la empresa. Con la implementación de los microservicios, se espera que se reduzcan los tiempos de respuesta y mejore la escalabilidad del sistema, ayude a la seguridad y persistencia de los datos, lo que permitirá un crecimiento futuro de la empresa.

Introducción: proyecto y los desafíos que se buscan resolver.

El presente proyecto ha sido desarrollado por un equipo de estudiantes en el ies Luis vives en el curso de Desarrollo de aplicaciones multiplataforma, centrándonos en los módulos de la especialidad de Programación de servicios y procesos, y Acceso a datos.

En este proyecto se busca aplicar los conocimientos adquiridos en su formación académica reciente. El proyecto consiste en implementar una arquitectura basada en microservicios utilizando tecnologías como BBDD reactivas, Ktor, MariaDB, Postgres, Spring, SSL y Token entre otras de testeo, documentación etc.

A pesar de que el equipo no cuenta con una gran experiencia, han decidido asumir el desafío de desarrollar un proyecto innovador que pueda resolver problemas reales. La idea es utilizar las herramientas y tecnologías mencionadas para desarrollar una solución escalable y eficiente que permita una gestión efectiva de datos en una empresa de piezas, servicios y aparatos electrónicos.

El proyecto se enfrenta a varios desafíos, como, por ejemplo, la complejidad del diseño arquitectónico, la implementación de tecnologías avanzadas y la integración de diferentes sistemas y servicios.

Además, el equipo deberá trabajar en equipo y coordinar sus esfuerzos para garantizar el éxito del proyecto.

En resumen, el equipo de estudiantes se ha propuesto enfrentar estos desafíos con dedicación y esfuerzo para alcanzar los objetivos del proyecto y lograr una solución eficiente y escalable para la gestión de datos en una empresa de piezas y aparatos electrónicos.

Métodos de organización de equipo.

El siguiente punto de documentación se refiere a los métodos utilizados para organizar el equipo durante el proyecto. En este caso, se han empleado varios recursos para asegurar la correcta coordinación y colaboración del equipo.

Uno de los recursos clave ha sido la plataforma de Git Hub, que ha permitido a los miembros del equipo trabajar juntos en un entorno en línea seguro y eficiente. A través de Git Hub, se ha podido compartir el código y realizar el seguimiento de los cambios realizados en el proyecto, lo que ha facilitado la identificación de problemas y la toma de decisiones en equipo.

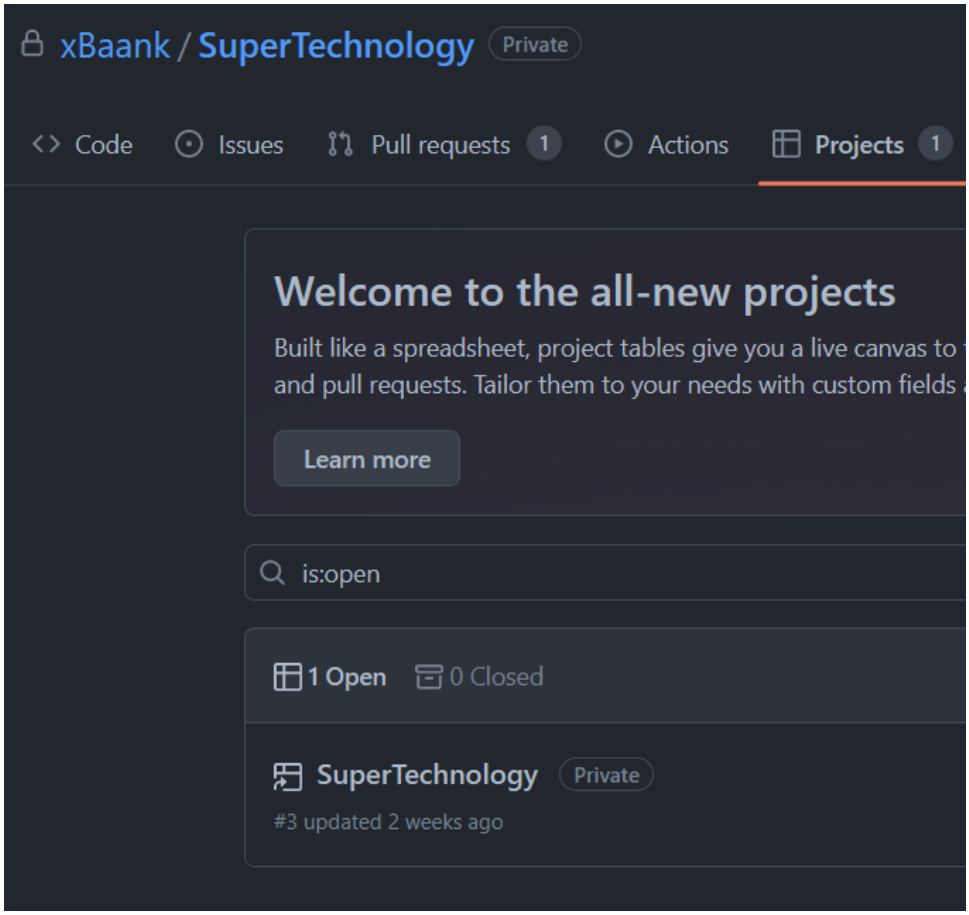
Además, también se ha utilizado la función de proyectos de Git Hub para mantener una visión general del estado del proyecto y asignar tareas específicas a los miembros del equipo. Esto ha permitido asegurar que todos los aspectos del proyecto estén en curso y se avance de manera coordinada.

Otro recurso clave ha sido la organización de reuniones semanales. Estas reuniones han permitido al equipo mantenerse en contacto y compartir actualizaciones sobre el progreso del proyecto, identificar problemas y soluciones en conjunto, y asegurar que los objetivos del proyecto se estén cumpliendo de manera efectiva.

En resumen, los métodos de organización de equipo utilizados en este proyecto han incluido el uso de Git Hub con Git flow en la realización de ramas.

Con esto compartir y realizar seguimiento de los cambios en el código, la función de proyectos de Git Hub para mantener una visión general del proyecto, y las reuniones semanales para mantener la coordinación y comunicación efectiva entre los miembros del equipo.

Captura del proyecto y el Project para la organización:


xBaank / SuperTechnology
Private
Code
Issues
Pull requests 1
Actions
Projects 1
Welcome to the all-new projects
Built like a spreadsheet, project tables give you a live canvas to
and pull requests. Tailor them to your needs with custom fields
Learn more
is:open
1 Open
0 Closed
SuperTechnology
Private
#3 updated 2 weeks ago

Justificación del diseño y metodologías usadas en el proyecto

Las metodologías ágiles, en particular la metodología Scrum que se basan en la división del proyecto en ciclos cortos llamados "sprints". Cada sprint es una fase de planificación, desarrollo, prueba y revisión que culmina con la entrega de un incremento de producto funcional.

En un proyecto de un mes, la implementación de sprints semanales puede ofrecer varios beneficios, como los siguientes:

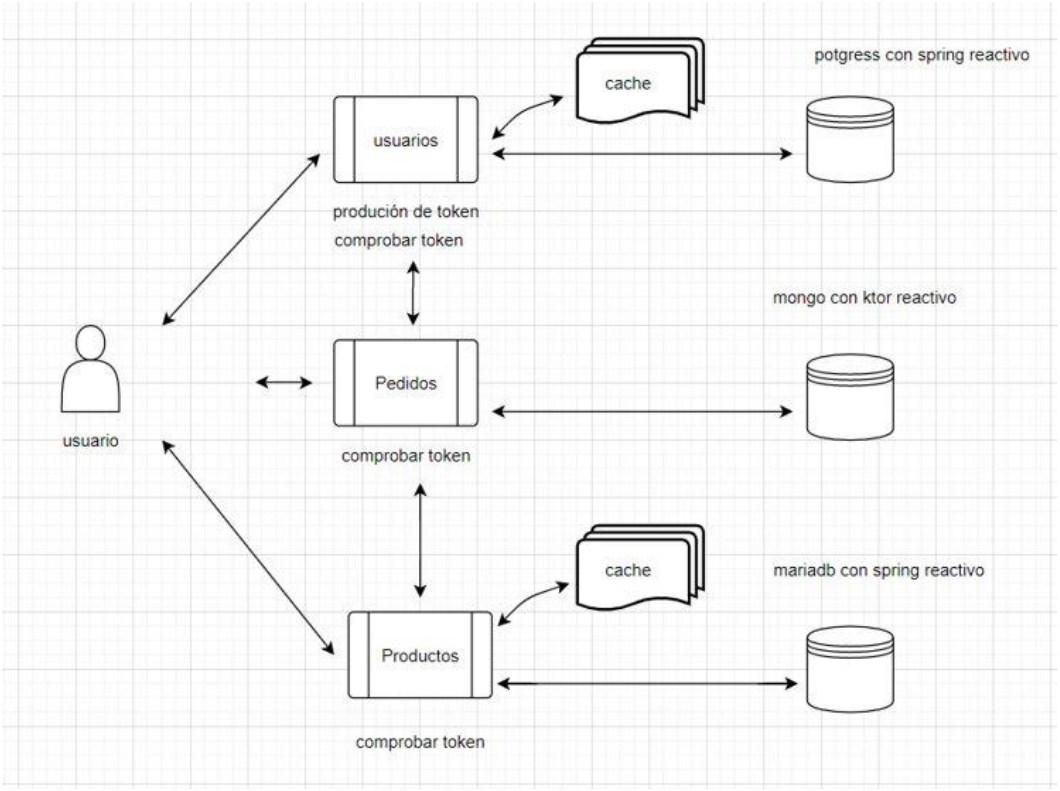
1. Flexibilidad y adaptabilidad: al utilizar metodologías ágiles, el equipo puede adaptarse rápidamente a los cambios y ajustes necesarios en el proyecto, ya que se trabaja con ciclos cortos que permiten ajustar el rumbo con facilidad. Esto evita que el equipo se vea atrapado en planes rígidos y permite que el proyecto evolucione en función de las necesidades reales.
2. Enfoque en entregables: los sprints semanales ofrecen una forma clara y tangible de medir el progreso del proyecto, ya que al final de cada sprint se espera que se entregue un incremento de producto funcional. Esto ayuda a mantener el enfoque en los resultados concretos y a garantizar que el equipo se mantenga en el camino correcto.
3. Mejora de la calidad: al dividir el proyecto en sprints semanales, se pueden realizar pruebas de forma continua, lo que ayuda a detectar errores y problemas de forma temprana. Esto permite que el equipo pueda tomar medidas inmediatas para corregir los errores y mejorar la calidad del producto final.
4. Mayor involucramiento del cliente: en las metodologías ágiles, el cliente es parte integral del equipo y se espera que esté involucrado de forma continua en el proceso de desarrollo. Al entregar un incremento de producto funcional al final de cada sprint, el cliente tiene la oportunidad de ver y evaluar el progreso del proyecto de forma continua, lo que mejora la comunicación y la colaboración entre el equipo y el cliente.

En general, la implementación de sprints semanales en un proyecto de un mes permite al equipo trabajar de forma más eficiente, flexible y centrada en los resultados. Al utilizar metodologías ágiles, se fomenta la colaboración, la comunicación y la mejora continua, lo que ayuda a garantizar que el proyecto cumpla con los requisitos y expectativas del cliente.

Marco teórico: tecnologías usadas y por qué de su elección entre otras tecnologías.

Existen muchas tecnologías que se pueden utilizar para crear APIs en un proyecto de desarrollo, y la elección de la tecnología depende de muchos factores como las necesidades del proyecto, las habilidades del equipo de desarrollo, el presupuesto, entre otros.

En este proyecto se tomó la decisión de hacer la elección de tecnologías del siguiente modo:



En el caso de una API de productos:

Respecto a la utilización de Spring con MariaDB Reactivo para desarrollar la API de productos, hay varias razones por las que hemos optado por esta elección:

1. Spring Framework es uno de los frameworks más populares y completos para el desarrollo de aplicaciones empresariales en Java. Proporciona un conjunto de herramientas y librerías que permiten desarrollar aplicaciones escalables y robustas.

2. MariaDB es una base de datos de código abierto, que se deriva de MySQL y tiene una gran compatibilidad con la mayoría de las aplicaciones y herramientas que utilizan MySQL. Al ser reactivo, permite procesar varias solicitudes simultáneamente, lo que lo hace ideal para aplicaciones de alta concurrencia.

3. La programación reactiva es una metodología que se enfoca en la eficiencia y el rendimiento, permitiendo el manejo de grandes volúmenes de datos con un bajo consumo de recursos. En aplicaciones donde la velocidad es un factor clave, como las APIs, la programación reactiva puede ser muy útil.

En cuanto a por qué utilizar Spring con MariaDB Reactivo en lugar de otras tecnologías, algunos factores que influllen en esta elección son la facilidad de integración, el soporte de la comunidad, la escalabilidad, la eficiencia en el manejo de grandes volúmenes de datos y la experiencia del equipo de desarrollo en estas tecnologías. En general, Spring con MariaDB Reactivo puede ser una buena elección para proyectos de alta concurrencia que requieren una respuesta rápida y eficiente en la gestión de grandes volúmenes de datos.

En el caso de una API de pedidos:

En cuanto a la elección de MongoDB con Ktor como tecnología para el proyecto en la parte de pedidos, existen las siguientes ventajas justifican su uso frente a otras tecnologías:

MongoDB es una base de datos NoSQL que permite una fácil escalabilidad horizontal y una buena capacidad para almacenar grandes cantidades de datos de forma eficiente.

1. Ktor es un framework web de Kotlin que se caracteriza por ser ligero, flexible y escalable, lo que lo hace una buena opción para la implementación de APIs.

2. La combinación de MongoDB con Ktor permite desarrollar aplicaciones web modernas, escalables y fáciles de mantener.

3. Ktor cuenta con soporte para el uso de coroutines, lo que permite desarrollar aplicaciones asíncronas y escalables que aprovechan mejor los recursos del sistema.

En resumen, la elección de MongoDB con Ktor para el proyecto de una API de pedidos que conecta con dos sistemas existentes puede ofrecer ventajas en términos de escalabilidad, eficiencia, flexibilidad y mantenibilidad, lo que justifica su uso frente a otras tecnologías.

En el caso de una API de Usuarios:

En el caso de la API de usuarios, es el api principal que crea tokens para la validación y autentificación de los usuarios en las demás Apis, la utilización de las tecnologías de cache y Postgres con Spring Reactivo puede ofrecer varias ventajas que justifican su uso:

1. Cache: la utilización de una capa de cache puede mejorar significativamente el rendimiento de la aplicación al reducir la necesidad de hacer múltiples consultas a la base de datos. Almacenar los tokens en una capa de cache puede permitir a la aplicación servirlos de forma rápida sin tener que acceder a la base de datos cada vez que se necesita un token. Esto puede reducir la latencia y mejorar la escalabilidad de la aplicación.

2. Postgres: la elección de Postgres como base de datos puede ofrecer varias ventajas. En primer lugar, Postgres es una base de datos relacional madura y muy utilizada, lo que significa que cuenta con una gran cantidad de recursos y herramientas disponibles para su uso. En segundo lugar, Postgres cuenta con características avanzadas como soporte para transacciones, integridad referencial, índices avanzados y soporte para JSON y otras estructuras de datos no relacionales, lo que lo hace una buena opción para una aplicación de usuarios que requiere una alta consistencia y flexibilidad en la gestión de datos.

3. Spring Reactivo: la utilización de Spring Reactivo puede mejorar la escalabilidad y la eficiencia de la aplicación al permitir el manejo asíncrono de peticiones HTTP y el procesamiento de múltiples solicitudes simultáneamente. Esto puede mejorar la capacidad de la aplicación para manejar un alto volumen de tráfico y reducir la latencia en la respuesta de la aplicación.

En conclusión, la utilización de una capa de cache y Postgres con Spring Reactivo en un proyecto basado en una API de usuarios que crea tokens y los consume puede mejorar significativamente la eficiencia, escalabilidad y rendimiento de la aplicación, lo que justifica su uso en este tipo de proyectos.

Análisis de requerimientos:

Para explicar en profundidad los requerimientos del proyecto en un principio explicaremos los distintos usuarios que pueden acceder a las apis.

Todos los usuarios tendrán las siguientes características:

Nombre

Email

Teléfono

Rol: Que puede ser User, Admin, o Superadmin.

Avatar

Activo: característica que indica si sigue logado en el sistema.

Fecha de creación: fecha del día y hora del logado

Fecha de actualización: fecha del día y hora de la última actualización.

Fecha de desactivación: fecha del día y hora del deslogado.

Id: clave única que representa a este usuario en el sistema

Dirección / Direcciones

Como podemos ver en nuestra aplicación usaremos para la Autentificación y la Autorización también un campo clave, que, por supuesto estará cifrado desde el momento que el usuario la introduzca, en este caso optamos por usar Bcrypt.

Pueden ver en más detalle la justificación de la encriptación y seguridad de bcrypt en el apartado cifrado de contraseñas.

Análisis de los diferentes tipos de usuarios o clientes que utilizarán las APIs.

Tras la descripción de los datos necesarios para todos los usuarios como podemos ver tenemos tres tipos de clientes, podremos diferenciarlos a través de su campo “Rol “. Que nos servirá para la autorización en las diferentes apis que utiliza nuestro programa.

Los tres tipos de rol especificados son:

User, Admin, o Superadmin

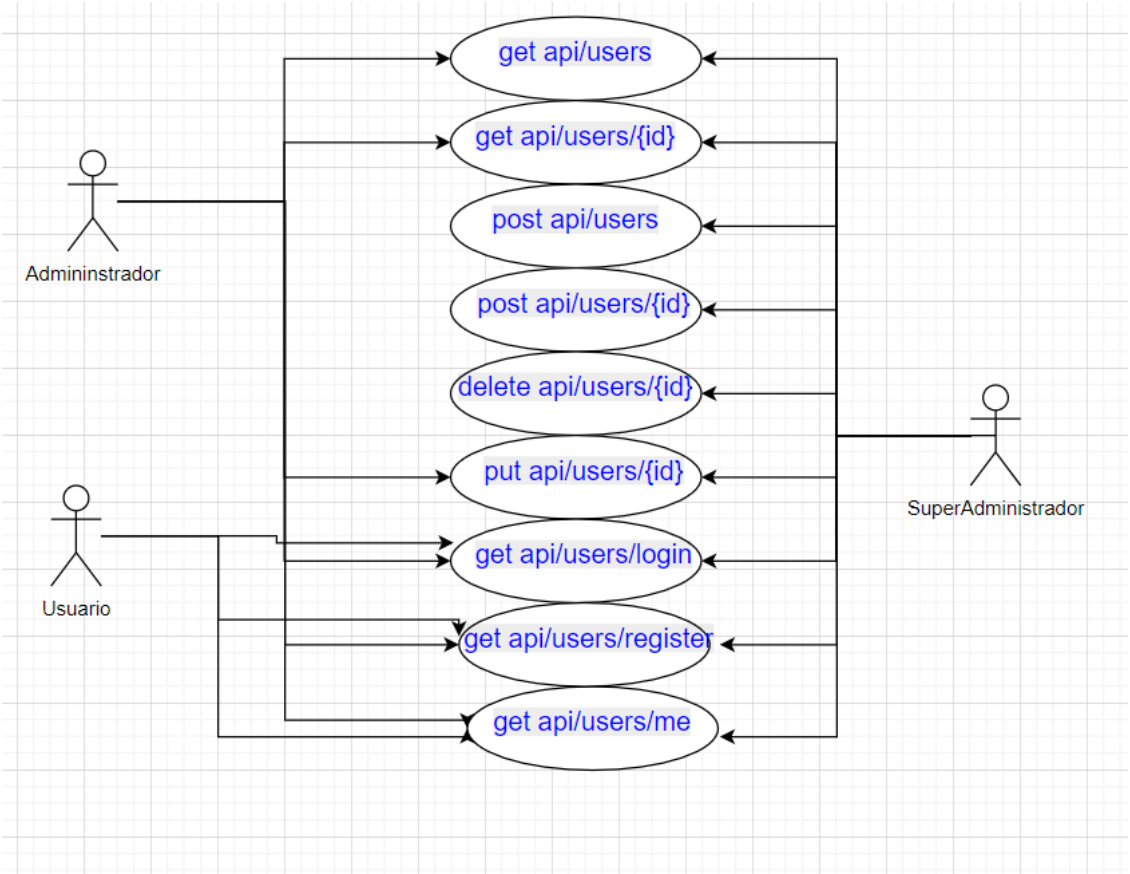
1. User: cliente logado en el sistema que puede tener asociada unas compras a su nombre.

2. Admin: Usuario trabajador que puede gestionar los distintos pedidos y la modificación parcial de productos.

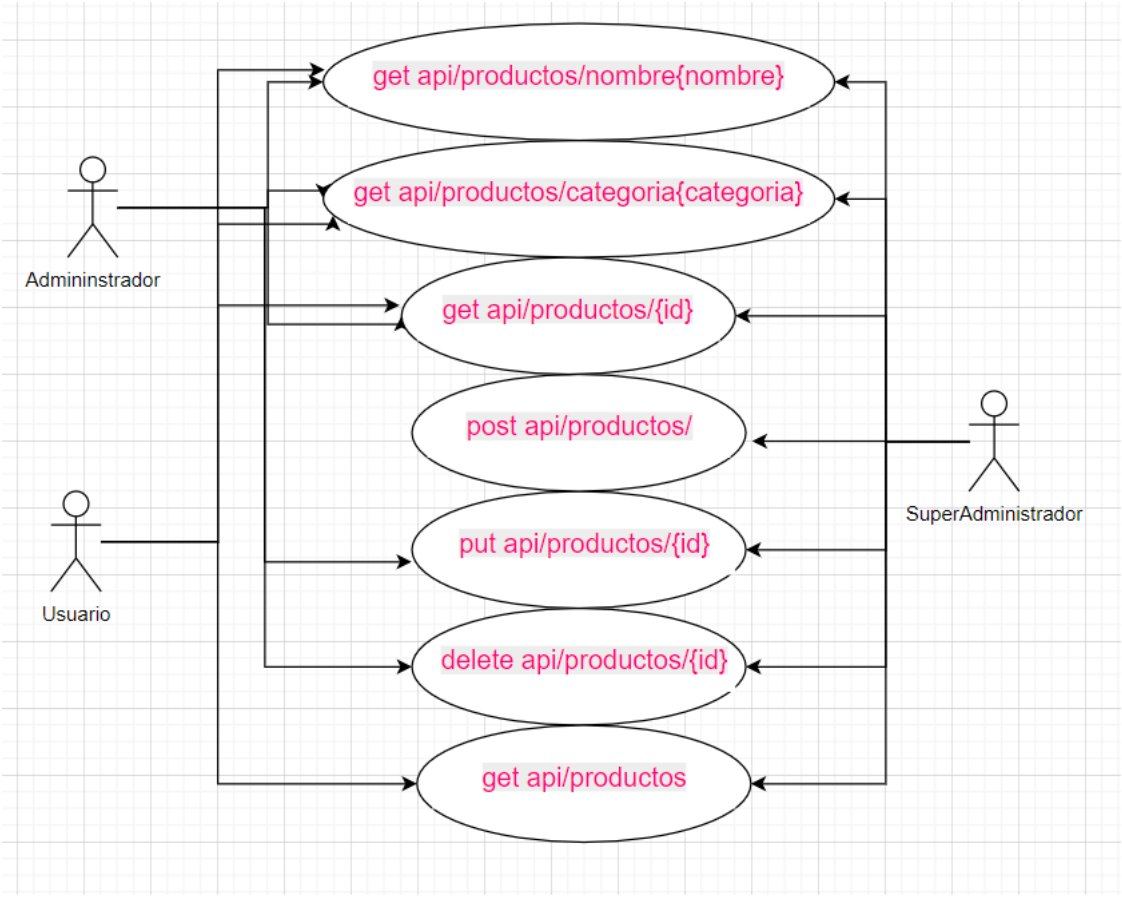
3. Superadmin: Usuario trabajador que puede gestionar todos los apartados de nuestro programa y de todas sus apis asociadas exceptuando las contraseñas de otros usuarios.

Teniendo en cuenta estos tres roles hemos dividido los recursos permitidos en la aplicación, y para mayor comprensión hemos subdividido estos en tres campos:

Relacionados con Usuarios:

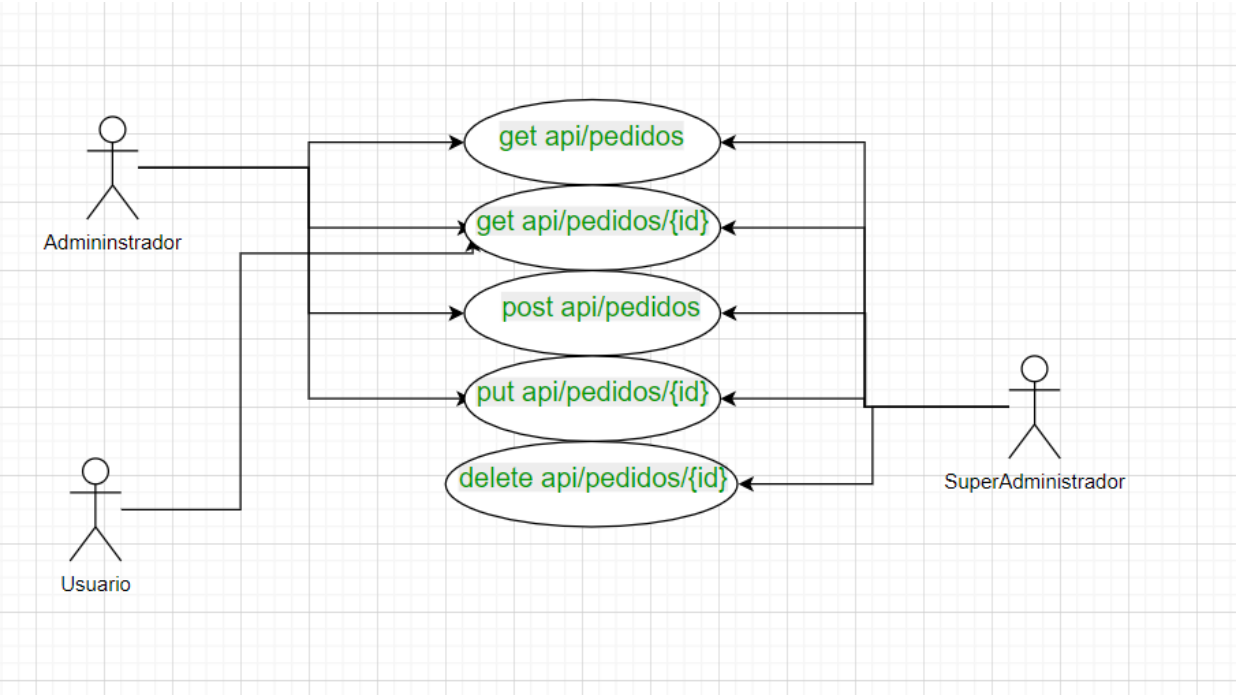


Relacionados con Productos:



Haciendo en estas distinciones de endpoint entre la obtención de productos Usuarios o Trabajadores, ya que teniendo en cuenta los requisitos funcionales del proyecto, no es de buena práctica que los clientes puedan tener acceso a los detalles de creación, modificación o activación de los productos.

Relacionados con Pedidos:


get api/pedidos
get api/pedidos/{id}
post api/pedidos
put api/pedidos/{id}
delete api/pedidos/{id}
Admininstrador
Usuario
SuperAdministrador

Descripción de los requisitos del proyecto en términos de funcionalidad.

1. Gestión de usuarios: la API debe permitir la gestión de usuarios, incluyendo la creación, actualización, eliminación y autenticación de usuarios.

5. Gestión de pedidos: la API debe permitir la gestión de pedidos, incluyendo la creación, actualización, eliminación y consulta de pedidos.

6. Gestión de productos: la API debe permitir la gestión de productos, incluyendo la creación, actualización, eliminación y consulta de productos.

7. Autenticación y autorización: la API debe incluir mecanismos de autenticación y autorización para garantizar que solo los usuarios autorizados puedan acceder a las funcionalidades de la API.

8. Validación de datos: la API debe incluir mecanismos para validar los datos de entrada y salida para asegurar que cumplan con los requisitos de formato y contenido.

9. Seguridad: la API debe incluir medidas de seguridad para proteger los datos de los usuarios y prevenir ataques malintencionados.

10. Integración con las otras Apis del proyecto de forma eficiente: la API debe permitir la integración con otras aplicaciones y sistemas para intercambiar información y compartir datos.

11. Escalabilidad: la API debe ser escalable para soportar un alto volumen de solicitudes y transacciones.

12. Rendimiento: la API debe ser rápida y eficiente para garantizar una experiencia de usuario fluida y sin interrupciones.

13. Documentación: la API debe contar con una documentación clara y detallada para ayudar a los desarrolladores a entender cómo utilizar la API y cómo interactuar con ella.

14. Testeo: la API debe contar con unos test unitarios y de integración válidos y de calidad para probar el buen funcionamiento.

Descripción de los requisitos del proyecto en términos de rendimiento.

1. Tiempo de respuesta: la API debe ser rápida y eficiente para garantizar un tiempo de respuesta rápido a las solicitudes de los usuarios.

2. Escalabilidad: la API debe ser capaz de manejar un alto volumen de solicitudes y transacciones sin comprometer el rendimiento.

3. Carga de trabajo: la API debe ser capaz de manejar una carga de trabajo variable y adaptarse a los cambios en la demanda de los usuarios.

4. Rendimiento de la base de datos: la API debe estar diseñada para maximizar el rendimiento de la base de datos y minimizar el tiempo de espera.

5. Ancho de banda: la API debe ser capaz de manejar una gran cantidad de datos y de transferirlos de manera eficiente a través de la red.

6. Optimización de consultas: la API debe estar diseñada para optimizar las consultas a la base de datos y minimizar el tiempo de espera.

7. Cache de datos: la API puede incluir un sistema de cache de datos para reducir la carga en la base de datos y mejorar el tiempo de respuesta.

8. Escalabilidad horizontal: la API debe estar diseñada para ser escalable horizontalmente, es decir, ser capaz de agregar más recursos de hardware para aumentar la capacidad de la aplicación.

Descripción de los requisitos del proyecto en términos de seguridad.

1. Autenticación y autorización: Todos los usuarios deben autenticarse antes de acceder a cualquier recurso protegido. Para ello, se debe utilizar tokens de acceso, que se pueden generar mediante el uso de Spring Security. Además, es necesario garantizar que los usuarios solo tengan acceso a los recursos para los que tienen permiso.

2. Protección de datos: Es necesario garantizar que los datos de los usuarios y los pedidos se protejan adecuadamente contra accesos no autorizados. Para ello, se pueden utilizar técnicas de encriptación y protección de datos, como SSL.

3. Manejo de errores: El proyecto debe estar diseñado para manejar errores de seguridad de manera adecuada. Esto incluye errores de autenticación, intentos de acceso no autorizados y otros problemas relacionados con la seguridad.

4. Protección contra ataques de fuerza bruta: Las APIs deben estar diseñadas para protegerse contra ataques de fuerza bruta, como intentos de adivinar contraseñas y tokens de acceso, con tecnologías tales como Bcrypt.

En general, los requisitos de seguridad para un proyecto de APIs de usuarios, pedidos y productos deben ser rigurosos para garantizar la protección adecuada de los datos del usuario y prevenir cualquier vulnerabilidad de seguridad.

Explicación de elección de Bcrypt en el apartado cifrado de contraseñas.

La utilización de algoritmos de encriptación como Bcrypt para proteger las contraseñas es importante por varias razones:

1. Seguridad: Bcrypt es un algoritmo de encriptación de contraseñas seguro y robusto que utiliza una función hash y un salto (salt) aleatorio para generar una cadena de texto encriptada. Esto significa que las contraseñas almacenadas en la base de datos están encriptadas y no pueden ser leídas fácilmente por alguien que no tenga acceso a la clave de encriptación.

2. Resistencia a ataques de fuerza bruta: Bcrypt es resistente a ataques de fuerza bruta porque es muy lento y requiere mucha CPU para calcular la función hash. Esto hace que sea muy difícil para los atacantes descifrar la contraseña encriptada mediante la prueba de múltiples contraseñas hasta encontrar la correcta.

3. Protección contra ataques de diccionario: Bcrypt también es resistente a ataques de diccionario porque utiliza un salto (salt) aleatorio que cambia cada vez que se genera una contraseña encriptada. Esto significa que incluso si un atacante tiene acceso a una lista de contraseñas encriptadas, no podrá utilizarlas para descifrar fácilmente otras contraseñas.

En resumen, utilizar Bcrypt para encriptar las contraseñas en un programa de desarrollo puede mejorar significativamente la seguridad y protección de los datos de los usuarios. Esto es especialmente importante en aplicaciones que manejan información sensible como datos personales o de empresa.

Por ello hemos optado para este proyecto con la intención de desarrollar con la seguridad necesaria para un sistema de calidad.

Lenguaje seleccionado:

Hemos decidido utilizar el lenguaje de programación Kotlin para la realización de este proyecto por cualidades como la concisión del código, cosa que nos permite realizar clases con un contenido más legible y sencillo de entender que lo que podría ser en Java como alternativa dentro de la JVM conocida por el equipo de desarrollo.

Kotlin nos permite tener un mejor control de la nulabilidad de los campos con lo que podemos reducir la cantidad de errores por campos nulos que pudiesen darse en las distintas apis del servicio, también elimina las clases estáticas aportando un código más limpio y mejora la interoperabilidad con código Java en caso de ser necesario, esto facilita también la escritura de pruebas unitarias al evitar el anclaje a un contexto estático con lo que nos permite mockear o remplazar implementaciones durante las pruebas.

Kotlin nos permite trabajar tanto con Ktor como con Spring, frameworks requeridos por el cliente para el desarrollo de nuestra aplicación.

Desarrollo e implementación:

Descripción detallada del proceso de desarrollo de los microservicios y las APIs, incluyendo las tecnologías utilizadas, metodología de desarrollo y explicación de las clases.

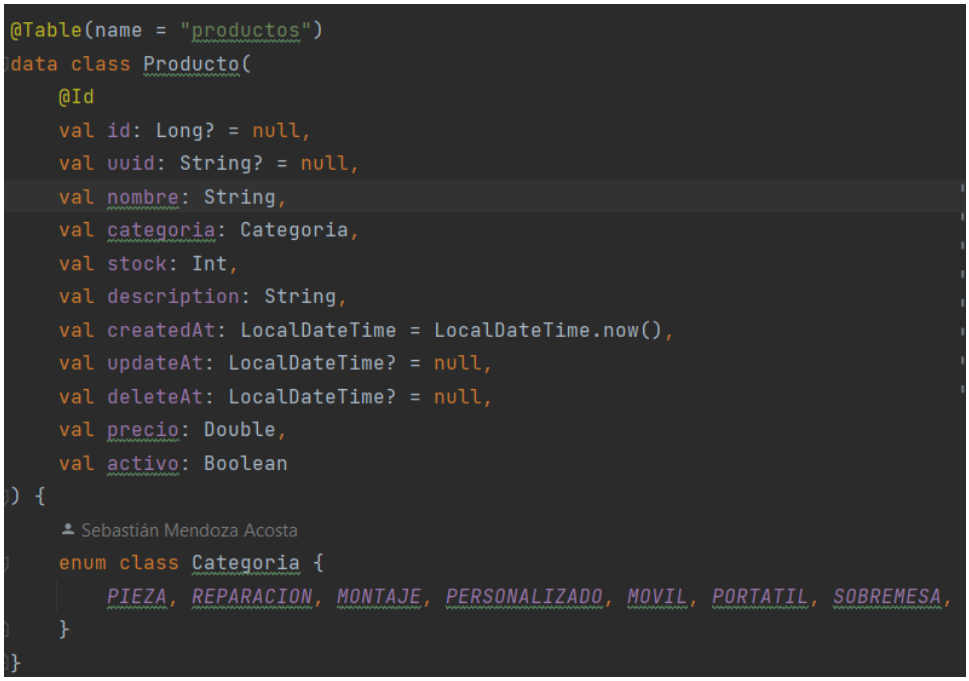
En cada una de las Apis al haber desarrollado tecnologías diferentes tenemos una metodología de implantación similar pero un código distinto, por lo que para mejor explicación iremos detallando el código por los siguientes paquetes:

1. Modelos.
2. Controladores.
3. Excepciones.
4. Configuración.
5. Cache.
6. Repositorios.
7. Autorización y Autentificación.

Además, explicaremos las especificaciones de cada api que, aunque individuales se unen en un mismo proyecto de microservicios.

Modelos Producto:

Clase que representa una entidad de producto que es utilizada en la aplicación que se comunica con una base de datos de productos y con la de pedidos.



Especificaciones sobre la clase:

`@Table(name = "productos")`: esto es una anotación que indica que esta clase se mapeará a una tabla llamada "productos" en la base de datos.

`data class Producto`: esto es una clase de datos de Kotlin, lo que significa que se proporcionan automáticamente algunos métodos comunes para trabajar con esta clase, como **`toString()`**, **`equals()`**, **`hashCode()`** y otros.

`@Id`: esto es otra anotación que indica que este campo representa la clave primaria en la tabla de la base de datos. La clave primaria es un identificador único que se utiliza para identificar un registro específico en la tabla.

`val id: Long? = null`: esto es un campo de datos que representa la clave primaria del producto. El valor puede ser **`null`** cuando se crea un nuevo producto en la base de datos, ya que la clave primaria se generará automáticamente por el sistema de la base de datos.

`val uuid: String? = null`: esto es otro campo de datos que representa un identificador único universal (UUID) para el producto. Al igual que la clave primaria, este valor puede ser **`null`** al crear un nuevo producto, y se generará automáticamente.

`val nombre: String`: esto es un campo de datos que representa el nombre del producto.

`val categoria: Categoria`: esto es un campo de datos que representa la categoría del producto. La categoría es una enumeración definida dentro de la clase.

`val stock: Int`: esto es un campo de datos que representa la cantidad de stock disponible para este producto.

`val description: String`: esto es un campo de datos que representa la descripción del producto.

`val createdAt: LocalDateTime = LocalDateTime.now()`: esto es un campo de datos que representa la fecha y hora en que se creó este producto. El valor predeterminado es la fecha y hora actual.

`val updateAt: LocalDateTime? = null`: esto es un campo de datos que representa la fecha y hora en que se actualizó este producto por última vez. El valor predeterminado es **`null`**.

`val deleteAt: LocalDateTime? = null`: esto es un campo de datos que representa la fecha y hora en que se eliminó este producto de la base de datos. El valor predeterminado es **`null`**.

`val precio: Double`: esto es un campo de datos que representa el precio del producto.

`val activo: Boolean`: esto es un campo de datos que representa si el producto está activo o no.

`enum class Categoria`: esto es una enumeración que define las diferentes categorías posibles para un producto. Estas categorías son "PIEZA", "REPARACION", "MONTAJE", "PERSONALIZADO", "MOVIL", "PORTATIL", "SOBREMESA" y "TABLET".

Modelos Pedido:

Clase que representa una entidad de Pedido que es utilizado en la aplicación que se comunica con una base de datos de usuarios y con la de pedidos y productos.

```
Bank +1
data class Pedido(
    val _id: Id<Pedido> = newId(),
    val usuario: UsuarioDto,
    val tareas: List<Tarea>,
    val iva: Double,
    val estado: EstadoPedido,
    val createdAt: Long
)

Bank
val Pedido.total: Double get() = tareas.sumOf(Tarea::precio)

Bank
enum class EstadoPedido {
    ENTREGADO,
    EN_PROCESO,
    CANCELADO
}
```

_id: es una propiedad de tipo **Id<Pedido>**. El valor predeterminado es una nueva instancia de **Id<Pedido>**.

usuario: es una propiedad de tipo **UsuarioDto**, que es un objeto que contiene información del usuario que ha hecho el pedido.

tareas: es una propiedad de tipo **List<Tarea>**, que es una lista de tareas relacionadas con el pedido.

iva: es una propiedad de tipo **Double** que representa el impuesto sobre el valor añadido (IVA) que se aplica al pedido.

estado: es una propiedad de tipo **EstadoPedido** que representa el estado actual del pedido. Puede ser **ENTREGADO**, **EN_PROCESO** o **CANCELADO**.

createdAt: es una propiedad de tipo **Long** que representa la fecha y hora en que se creó el pedido.

Además, el objeto **Pedido** tiene una propiedad calculada llamada **total**, que devuelve la suma de precios de todas las tareas en el pedido.

Modelos Tarea:

Clase que representa una entidad de Tarea que es utilizado en la aplicación que se comunica con una base de datos de usuarios y con la de pedidos y productos.

```
data class Tarea(
    val _id: Id<Tarea> = newId(),
    val producto: ProductoDto,
    val empleado: UsuarioDto,
    val createdAt: Long
)

Bank
val Tarea.precio: Double get() = producto.precio
```

_id: es una propiedad de tipo **Id<Tarea>**. El valor predeterminado es una nueva instancia de **Id<Tarea>**.

producto: es una propiedad de tipo **ProductoDto**, que es un objeto que contiene información sobre el producto relacionado con la tarea.

empleado: es una propiedad de tipo **UsuarioDto**, que es ser un objeto que contiene información sobre el empleado que realizará la tarea.

createdAt: es una propiedad de tipo **Long** que representa la fecha y hora en que se creó la tarea.

Modelos Usuario / Users:

Clase que representa una entidad de Usuario que es utilizado en la aplicación que se comunica con una base de datos de usuarios y con la de pedidos y productos.

```kotlin
@Table(name = "users")
data class User(
    @Id
    val id: UUID? = null,
    @get:JvmName("userName")
    val username: String,
    val email: String,
    @get:JvmName("userPassword")
    val password: String,
    val phone: String,
    val avatar: String = "",
    val role: UserRole,
    @Column("created_at")
    val createdAt: LocalDate = LocalDate.now(),
    val active: Boolean
) : UserDetails {

    /**
     * Clase enum usada para los distintos roles de los usuarios
     * @author Mario Gonzalez, Daniel Rodriguez, Joan Sebastian Mendoza,
     * Alfredo Rafael Maldonado, Azahara Blanco, Ivan Azagra, Roberto Blazquez
     */
    Mario Resa
    enum class UserRole {
        USER, ADMIN, SUPER_ADMIN
    }
```

@Table(name = "productos"): esto es una anotación que indica que esta clase se mapeará a una tabla llamada "productos" en la base de datos.

data class Producto: esto es una clase de datos de Kotlin, lo que significa que se proporcionan automáticamente algunos métodos comunes para trabajar con esta clase, como **toString()**, **equals()**, **hashCode()** y otros.

@Id: esto es otra anotación que indica que este campo representa la clave primaria en la tabla de la base de datos. La clave primaria es un identificador único que se utiliza para identificar un registro específico en la tabla.

val id: Long? = null: esto es un campo de datos que representa la clave primaria del producto. El valor puede ser **null** cuando se crea un nuevo producto en la base de datos, ya que la clave primaria se generará automáticamente por el sistema de la base de datos.

val uuid: String? = null: esto es otro campo de datos que representa un identificador único universal (UUID) para el producto. Al igual que la clave primaria, este valor puede ser **null** al crear un nuevo producto, y se generará automáticamente.

val nombre: String: esto es un campo de datos que representa el nombre del producto.

val categoria: Categoria: esto es un campo de datos que representa la categoría del producto. La categoría es una enumeración definida dentro de la clase.

val stock: Int: esto es un campo de datos que representa la cantidad de stock disponible para este producto.

val description: String: esto es un campo de datos que representa la descripción del producto.

val createdAt: LocalDateTime = LocalDateTime.now(): esto es un campo de datos que representa la fecha y hora en que se creó este producto. El valor predeterminado es la fecha y hora actual.

val updateAt: LocalDateTime? = null: esto es un campo de datos que representa la fecha y hora en que se actualizó este producto por última vez. El valor predeterminado es **null**.

val deleteAt: LocalDateTime? = null: esto es un campo de datos que representa la fecha y hora en que se eliminó este producto de la base de datos. El valor predeterminado es **null**.

val precio: Double: esto es un campo de datos que representa el precio del producto.

val activo: Boolean: esto es un campo de datos que representa si el producto está activo o no.

enum class Categoria: esto es una enumeración que define las diferentes categorías posibles para un producto. Estas categorías son "PIEZA", "REPARACION", "MONTAJE", "PERSONALIZADO", "MOVIL", "PORTATIL", "SOBREMESA" y "TABLET".

Controladores.

EL controlador es responsable de recibir las solicitudes del usuario, de tomar decisiones y coordinar acciones en función de esas solicitudes, y de generar la respuesta correspondiente. Es una parte clave de la lógica de una aplicación y su correcto diseño es esencial para lograr una aplicación robusta y escalable.

En el caso de nuestro sistema reciben las peticiones de usuarios y toman acciones con la autentificación las caches, repositorios y todas las partes del programa para dar una respuesta correcta.

Tenemos los siguientes controladores que describimos a continuación:

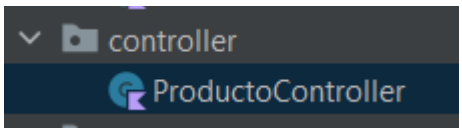
Controladores con Spring con Productos.

Controladores con Spring con Usuarios.

Controladores con Ktor en Pedidos.

Controladores con Spring con Productos:

Este es un controlador en una aplicación Spring que maneja las solicitudes HTTP para los recursos relacionados con los productos. La anotación @RestController se utiliza para marcar la clase como un controlador y la anotación @RequestMapping especifica la ruta base para todas las solicitudes HTTP manejadas por este controlador.



El constructor de la clase se inyecta con una instancia de ProductoCachedRepositoryImpl mediante la anotación @Autowired, lo que indica que Spring debe proporcionar una instancia de ProductoCachedRepositoryImpl al crear una instancia de ProductoController.

El método findAllProductos() es un controlador de solicitud HTTP GET que se asocia con la ruta de solicitud "/productos". La anotación @GetMapping se utiliza para especificar la ruta de solicitud y el método HTTP asociado. Dentro del método, se llama a la función findAll() del repositorio para obtener todos los productos y se convierten a objetos ProductoDto. Finalmente, se devuelve una ResponseEntity que contiene un flujo de objetos ProductoDto. La anotación @Operation y la anotación @ApiResponse se utilizan para proporcionar

información adicional sobre la operación en la documentación de la API y la anotación @PreAuthorize se utiliza para especificar los roles que tienen acceso a esta operación.

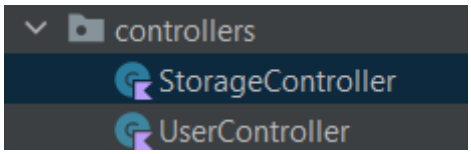
Ejemplo de un método en la clase controlador:

```
@RestController
@RequestMapping(APIConfig.API_PATH + "/productos")
class ProductoController
@Autowired constructor(
    private val repository: ProductoCachedRepositoryImpl
) {
    /**
     * Find all productos: Find all the products.
     *
     * @return Return all the products.
     */
    Sebastián Mendoza Acosta
    @Operation(summary = "Get all productos", description = "Get the products list", tags = ["Products"])
    @ApiResponse(responseCode = "200", description = "Lista de Producto")
    @PreAuthorize("hasAnyRole('USER','ADMIN', 'SUPER_ADMIN')")
    @GetMapping("")
    suspend fun findAllProductos(): ResponseEntity<Flow<ProductoDto>> =
        withContext(Dispatchers.IO) { this: CoroutineScope
            logger.info { "Get productos" }
            val res = repository.findAll().map { it.toDto() }
            return@withContext ResponseEntity.ok(res)
        }
```

Controladores con Spring con Usuarios:

En este api tenemos dos controladores ya que uno controlará las imágenes y ficheros que quiera subir el usuario, normalmente una imagen para su logado.

El otro controlador será para controlar la creación borrado y modificación de los usuarios propiamente dichos.



Este controlador es muy parecido al anteriormente mencionado de productos, como podemos ver también tiene las anotaciones que vemos en productos ya que hemos querido seguir con la misma línea de programación.

está anotada con varias anotaciones, como @Operation, @Parameter, @ApiResponse, @PreAuthorize y @PostMapping, que se utilizan en combinación con el framework Spring para implementar un endpoint HTTP para la creación de usuarios.

La anotación @Operation proporciona una descripción general de la operación y se utiliza para documentar el endpoint con OpenAPI.

La anotación @Parameter se utiliza para describir los parámetros de entrada del endpoint. En este caso, hay dos parámetros: userDTOcreate y user. userDTOcreate es un objeto que contiene los datos necesarios para crear un nuevo usuario y user es el objeto que contiene información de autenticación que se utiliza para autorizar la creación del usuario.

La anotación @ApiResponse se utiliza para describir las respuestas posibles del endpoint. En este caso, la respuesta exitosa es un ResponseEntity que contiene un objeto UserDTOwithToken que contiene información del usuario recién creado y un token JWT generado para el usuario.

La anotación @PreAuthorize se utiliza para restringir el acceso a este endpoint a los roles de "ADMIN" o "SUPER_ADMIN".

La anotación @PostMapping indica que este endpoint maneja solicitudes HTTP POST.

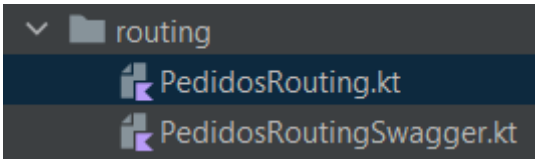
En resumen, este controlador implementa una función para crear usuarios a través de un endpoint HTTP en una aplicación web utilizando Spring y Kotlin.

Ejemplo del User Controller:

```
Mario Resa +1
@Operation(summary = "Create", description = "Endpoint for creating users that can only be accessed by administrators."
@Parameter(name = "userDTOcreate", description = "Valid DTO for creation.", required = true)
@Parameter(name = "user", description = "Token for authentication.", required = true)
@ApiResponse(responseCode = "200", description = "Response Entity with the DTO for visualization from the created user,
@ApiResponse(responseCode = "400", description = "If it cannot create successfully.")
@PreAuthorize("hasAnyRole('ADMIN', 'SUPER_ADMIN')")
@PostMapping("/create")
suspend fun create(
    @Valid @RequestBody userDTOcreate: UserDTOcreate,
    @AuthenticationPrincipal user: User
): ResponseEntity<UserDTOwithToken> =
    withContext(Dispatchers.IO) { this: CoroutineScope
        log.info { "Creando usuario por parte de un administrador" }

        userDTOcreate.validate()
        val userSaved = service.create(userDTOcreate)
        val addresses = userSaved.id?.let { service.findAllFromUserId(it).toSet() } ?: setOf()

        ResponseEntity(
            UserDTOwithToken(userSaved.toDTO(addresses), jwtTokenUtils.create(userSaved)),
            HttpStatus.CREATED
        ) ^withContext
    }
```

Controladores con Ktor en Pedidos:

En Ktor los distintos endpoints se manejan con Routing , por lo que en este caso lo introduciremos como controlador.



En estas clases hacemos el routing en Ktor para la entidad Pedido.

Primero, se define la ruta base para todos los endpoints relacionados con Pedido usando la función route("/pedidos").

Luego, se inyecta un PedidosRepository en la variable repository para poder utilizarlo en las rutas.

Se utilizan dos autenticaciones diferentes para los endpoints. El endpoint /usuario/me solo puede ser accedido por usuarios autenticados con rol "user", mientras que el endpoint /usuario/{id} solo puede ser accedido por usuarios autenticados con rol "admin". Para realizar la autenticación se utiliza la función authenticate de Ktor.

Dentro de cada endpoint se realiza una consulta al repositorio, donde se obtiene información relacionada con los Pedido. En el caso de /usuario/me, se obtiene el ID del usuario autenticado a través de su token JWT. Luego se utiliza este ID para buscar todos los Pedido asociados a este usuario.

Para varios endpoints se define un paginado por defecto, donde se utiliza DEFAULT_PAGE y DEFAULT_SIZE si no se especifican valores.

Finalmente, se utiliza la función call.respond para devolver la respuesta en formato JSON y call.handleError para manejar los errores.

Ejemplo de la clase:

```
fun Routing.pedidosRouting() = route("/pedidos") {
    val repository by inject<PedidosRepository>()
    authenticate("user") {
        get("/usuario/me", builder = OpenApiRoute::getByUsuarioMe) {
            val page = call.request.queryParameters["page"]?.toIntOrNull() ?: DEFAULT_PAGE
            val size = call.request.queryParameters["size"]?.toIntOrNull() ?: DEFAULT_SIZE
            val userId = call.principal<JWTPrincipal>()?.getClaim("id", String::class) ?: "

            repository.getByUserId(userId, page, size)
                .map { buildPagedPedidoDto(it) }
                .onLeft { call.handleError(it) }
                .onRight { call.respond(it) } ^get
        }
    }
    authenticate("admin") {
        get("/usuario/{id}", builder = OpenApiRoute::getByUsuarioId) {
            val page = call.request.queryParameters["page"]?.toIntOrNull() ?: DEFAULT_PAGE
            val size = call.request.queryParameters["size"]?.toIntOrNull() ?: DEFAULT_SIZE
            val usuarioId = call.parameters.getOrFail("id")

            repository.getByUserId(usuarioId, page, size)
                .map { buildPagedPedidoDto(it) }
                .onLeft { call.handleError(it) }
                .onRight { call.respond(it) } ^get
        }
```

Excepciones.

El control de excepciones es una parte crucial en el desarrollo de un sistema robusto y confiable. El uso de “try-catch” nos permite capturar y manejar de forma adecuada errores y situaciones imprevistas que puedan ocurrir durante la ejecución de nuestro código.

Es importante lanzar nuestras propias excepciones para que el usuario o el equipo de desarrollo reciba los mensajes de error que queremos, en lugar de mensajes genéricos o poco informativos. Para ello, debemos crear nuestras propias clases de excepción que extiendan de “Exception”, “RuntimeException” u otra clase de excepción de acuerdo a nuestras necesidades.

Podemos lanzar estas excepciones utilizando la sentencia “throw” dentro del bloque try-catch, especificando el mensaje de error que queremos que se muestre. También podemos utilizar la sentencia “finally “para realizar tareas de limpieza o cierre de recursos independientemente de si se produjo o no una excepción.

Aquí ponemos algunos ejemplos de control de excepciones:

```
@GetMapping("/{id}")
suspend fun findProductById(
    @PathVariable id: String
): ResponseEntity<ProductoDto> =
    withContext(Dispatchers.IO) { this: CoroutineScope
        logger.info { "Obteniendo producto por id" }
        try {
            val res = repository.findById(id)?.toDto()
                ?: throw ProductoNotFoundException("Producto no encontrado con id: $id .")
            return@withContext ResponseEntity.ok(res)
        } catch (e: ProductoNotFoundException) {
            throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
        } ^withContext
    }
```

Finalmente exponemos todas las excepciones que pueden ocasionar nuestro código:

Relacionados con los productos:

```
sealed class ProductoException(mensaje : String): RuntimeException(mensaje)
```

```
@ResponseStatus(HttpStatus.NOT_FOUND)
class ProductoNotFoundException(mensaje: String): ProductoException(mensaje)
```

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
class ProductoBadRequestException(mensaje: String): ProductoException(mensaje)
```

```
@ResponseStatus(HttpStatus.CONFLICT)
class  ProductoConflictIntegrityException(mensaje: String): ProductoException(mensaje)
```

```
@ResponseStatus(HttpStatus.CONFLICT)
class  ProductoConflictIntegrityException(mensaje: String): ProductoException(mensaje)
```

Relacionados con los tokens:

```
@ResponseStatus(HttpStatus.UNAUTHORIZED)
class TokenInvalidException(message: String) : TokenException(message)
```

Relacionados con los Usuarios:

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
class UserExceptionBadRequest(message: String?) : UserException(message)
```

```
@ResponseStatus(HttpStatus.NOT_FOUND)
class UserExceptionNotFound(message: String?) : UserException(message)
```

Relacionados con los Direcciones:

```
sealed class AddressException(message: String?) : RuntimeException(message)
```

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
class AddressExceptionBadRequest(message: String?) : UserException(message)
```

```
@ResponseStatus(HttpStatus.NOT_FOUND)
class AddressExceptionNotFound(message: String?) : AddressException(message)
```

Relacionados con los Pedidos:

```
sealed class DomainError : Exception()
```

```
class PedidoNotFound(override val message: String) : PedidoError()
```

```
class PedidoSaveError(override val message: String) : PedidoError()
```

```
class InvalidPedidoId(override val message: String) : PedidoError()
```

```
class InvalidPedidoPage(override val message: String) : PedidoError()
```

```
class InvalidPedidoFormat(override val message: String) : PedidoError()
```

```
class MissingPedidoId(override val message: String) : PedidoError()
```

Relacionados con los Storage:

```
sealed class StorageException : RuntimeException {
    Mario Resa
    constructor(message: String?) : super(message)
    Mario Resa
    constructor(message: String?, cause: Throwable?) : super(message, cause)
}
```

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
class StorageExceptionBadRequest : StorageException {
    Mario Resa
    constructor(message: String?) : super(message)
    Mario Resa
    constructor(message: String?, cause: Throwable?) : super(message, cause)
}
```

```
@ResponseStatus(HttpStatus.NOT_FOUND)
class StorageExceptionNotFound : StorageException {
    Mario Resa
    constructor(message: String?) : super(message)
    Mario Resa
    constructor(message: String?, cause: Throwable?) : super(message, cause)
}
```

Configuración.

Para la configuración tenemos varios ficheros importantes:

Api de mariaDB con spring:

Este fichero es un archivo de configuración para la aplicación Spring Boot para el api de Productos.

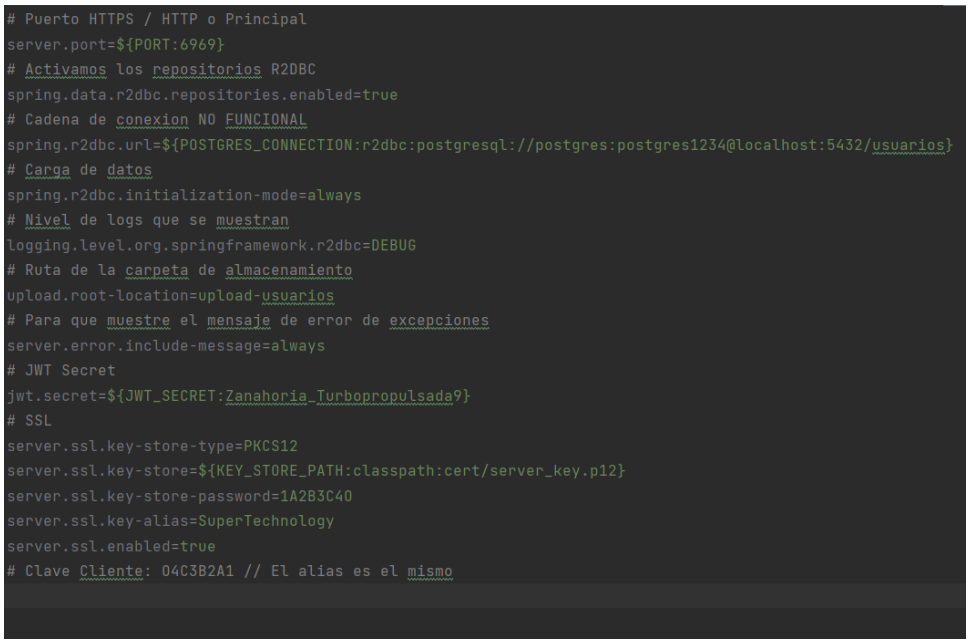
```
server.port=${PORT:6963}
spring.r2dbc.url=r2dbc:mariadb://mariadbadmin:mariadbpass@localhost:3306/productosTest
spring.data.r2dbc.repositories.enabled=true
spring.r2dbc.initialization-mode=always
logging.level.org.springframework.r2dbc=DEBUG
server.error.include-message=always
jwt.secret=Zanahoria_Turbopropulsada9
pagination.init=0
pagination.size=10
```

Contiene varias propiedades que se utilizan para configurar la aplicación:

1. server.port=${PORT:6963}: Establece el puerto del servidor. Utiliza la variable de entorno PORT si está definida, en caso contrario utiliza el valor predeterminado 6963.
2. Spring.r2dbc.url=r2dbc:mariadb://mariadbadmin:mariadbpass@localhost:3306/productosTest: Establece la URL de conexión a la base de datos MariaDB. Especifica el nombre de usuario, la contraseña, el host, el puerto y el nombre de la base de datos.
3. spring.data.r2dbc.repositories.enabled=true: Habilita la funcionalidad de Spring Data R2DBC para acceder a la base de datos.
4. spring.r2dbc.initialization-mode=always: Especifica que se debe inicializar la base de datos al arrancar la aplicación.
5. logging.level.org.springframework.r2dbc=DEBUG: Establece el nivel de registro para la librería Spring R2DBC.
6. server.error.include-message=always: Agrega el mensaje de error en la respuesta HTTP.
7. jwt.secret=Zanahoria_Turbopropulsada9: Especifica la clave secreta utilizada para generar tokens JWT.
8. pagination.init=0: Establece el índice inicial para la paginación.
9. pagination.size=10: Establece el tamaño de página para la paginación.

Api de Postges con Spring:

Este fichero es otro archivo de configuración para la aplicación Spring Boot, esta vezpara el api de usuarios.



```
# Puerto HTTPS / HTTP o Principal
server.port=${PORT:6969}
# Activamos los repositorios R2DBC
spring.data.r2dbc.repositories.enabled=true
# Cadena de conexion NO FUNCIONAL
spring.r2dbc.url=${POSTGRES_CONNECTION:r2dbc:postgresql://postgres:postgres1234@localhost:5432/usuarios}
# Carga de datos
spring.r2dbc.initialization-mode=always
# Nivel de logs que se muestran
logging.level.org.springframework.r2dbc=DEBUG
# Ruta de la carpeta de almacenamiento
upload.root-location=upload-usuarios
# Para que muestre el mensaje de error de excepciones
server.error.include-message=always
# JWT Secret
jwt.secret=${JWT_SECRET:Zanahoria_Turbopropulsada9}
# SSL
server.ssl.key-store-type=PKCS12
server.ssl.key-store=${KEY_STORE_PATH:classpath:cert/server_key.p12}
server.ssl.key-store-password=1A2B3C4O
server.ssl.key-alias=SuperTechnology
server.ssl.enabled=true
# Clave Cliente: O4C3B2A1 // El alias es el mismo
```

Contiene varias propiedades que se utilizan para configurar la aplicación:

1. server.port=${PORT:6969}: Establece el puerto del servidor. Utiliza la variable de entorno PORT si está definida, en caso contrario utiliza el valor predeterminado 6969.
2. spring.data.r2dbc.repositories.enabled=true: Habilita la funcionalidad de Spring Data R2DBC para acceder a la base de datos.
3. spring.r2dbc.url=${POSTGRES_CONNECTION:r2dbc:postgresql://postgres:postgres1234@localhost:5432/usuarios}: Especifica la cadena de conexión para la base de datos Postgres. Incluye el nombre de usuario, la contraseña, el host, el puerto y el nombre de la base de datos.
4. spring.r2dbc.initialization-mode=always: Especifica que se debe inicializar la base de datos al arrancar la aplicación.
5. logging.level.org.springframework.r2dbc=DEBUG: Establece el nivel de registro para la librería Spring R2DBC.
6. upload.root-location=upload-usuarios: Especifica la ruta de la carpeta de almacenamiento.
7. server.error.include-message=always: Agrega el mensaje de error en la respuesta HTTP.
8. jwt.secret=${JWT_SECRET:Zanahoria_Turbopropulsada9}: Especifica la clave secreta utilizada para generar tokens JWT.
9. server.ssl.key-store-type=PKCS12: Especifica el tipo de keystore para SSL.

10. server.ssl.key-store=${KEY_STORE_PATH:classpath:cert/server_key.p12}: Especifica la ubicación del keystore para SSL. Utiliza la variable de entorno KEY_STORE_PATH si está definida, en caso contrario utiliza el valor predeterminado classpath:cert/server_key.p12.
11. server.ssl.key-store-password=1A2B3C4O: Especifica la contraseña del keystore para SSL.
12. server.ssl.key-alias=SuperTechnology: Especifica el alias del keystore para SSL.
13. server.ssl.enabled=true: Habilita el uso de SSL.

Api de Ktor con Mongo:

Este fichero es un fichero de configuración de ktor:

```
ktor {
  deployment {
    port = 8080
    port = ${?PORT}
    sslPort = 8443
    sslPort = ${?SSL_PORT}
  }
  security {
    ssl {
      keyStore = .cert/server_key.p12
      keyStore = ${?KEYSTORE}
      keyAlias = SuperTechnology
      keyAlias = ${?KEY_ALIAS}
      keyStorePassword = 1A2B3C4O
      keyStorePassword = ${?KEYSTORE_PASSWORD}
      privateKeyPassword = 1A2B3C4O
      privateKeyPassword = ${?PRIVATE_KEY_PASSWORD}
    }
  }
  application {
    modules = [pedidosApi.MainKt.module]
  }
}
```

```
jwt {
  secret = "Zanahoria_Turbopropulsada9"
  secret = ${?JWT_SECRET}
}

mongo {
  connectionString = "mongodb://root:example@localhost:27017"
  connectionString = ${?MONGO_CONNECTION_STRING}
  database = "pedidos"
  database = ${?MONGO_DATABASE}
}

// Configuración de los microservicios
usuarios {
  url = "http://localhost:8081"
  url = ${?USUARIOS_URL}
}

productos {
  url = "http://localhost:8082"
  url = ${?PRODUCTOS_URL}
}
```

En el encontramos:

1. Configuración de despliegue de la aplicación, incluyendo el puerto en el que la aplicación escuchará y el puerto SSL.
2. Configuración de seguridad SSL, incluyendo el almacén de claves, alias, contraseña, y contraseña de la clave privada.
3. Configuración de la aplicación, que incluye la lista de módulos utilizados.
4. Configuración de autenticación con JWT, con la clave secreta utilizada para firmar y verificar tokens JWT
5. Configuración de la base de datos MongoDB, incluyendo la cadena de conexión y la base de datos utilizada.
6. Configuración de las URLs de los microservicios de usuarios y productos, que se utilizarán en la aplicación para conectarse a estos servicios.

Cache.

La caché es una técnica utilizada en la programación de APIs para almacenar temporalmente los resultados de operaciones costosas en recursos como bases de datos, sistemas de archivos, redes, etc. y así evitar realizar operaciones repetitivas y lentas en la aplicación.

Cuando se utiliza una caché, los resultados de las operaciones que se almacenan se pueden devolver rápidamente en lugar de tener que ejecutar la operación completa. Esto puede mejorar significativamente el rendimiento de la aplicación y reducir la carga en los recursos subyacentes.

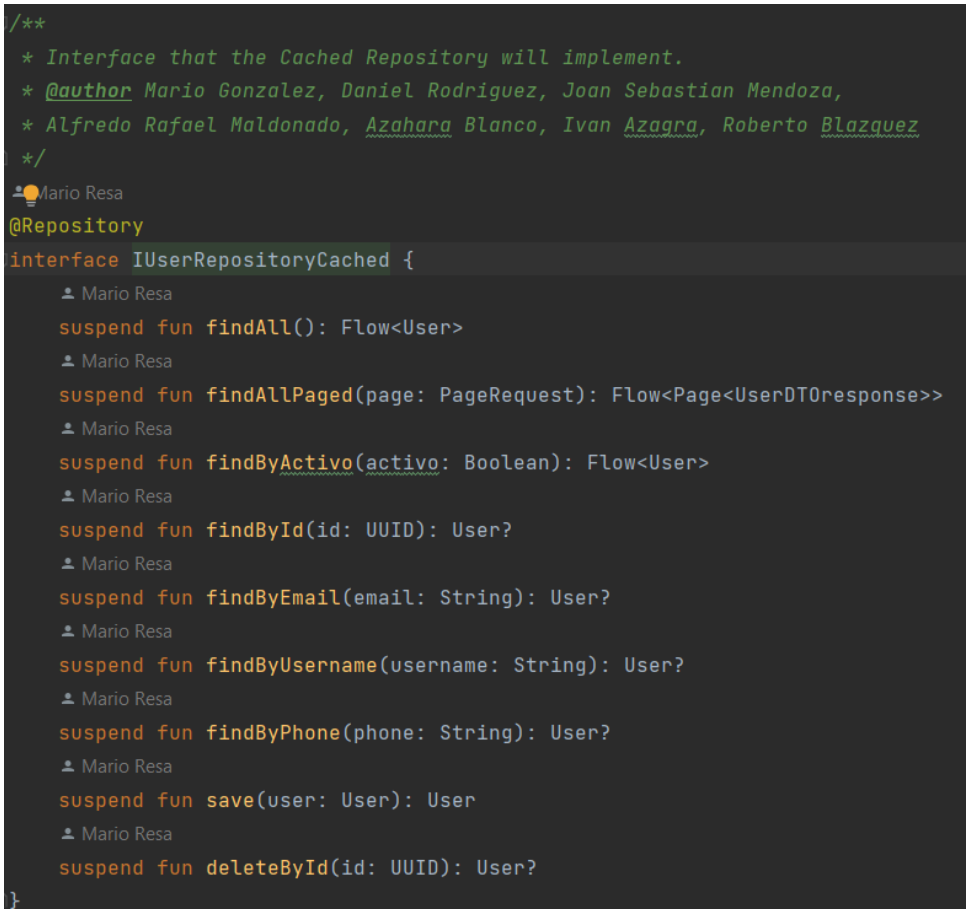
En nuestra aplicación hemos valorado el uso de cache y realizaremos la implementación de la cache en dos de nuestras tres apis, la de usuarios y la de productos.

Ambas se consultarán a través del controlador.

Productos “interfaz de la cache”:

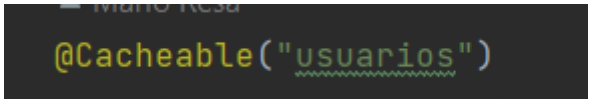
```
interface ProductoCacheRepository {
    Sebastián Mendoza Acosta
    suspend fun findAll(): Flow<Producto>
    Sebastián Mendoza Acosta
    suspend fun findById(uuid: String): Producto?
    Sebastián Mendoza Acosta
    suspend fun findByCategoria(categoria: String): Flow<Producto>
    Sebastián Mendoza Acosta
    suspend fun findByNombre(nombre: String): Flow<Producto>
    Sebastián Mendoza Acosta
    suspend fun save(producto: Producto): Producto
    Sebastián Mendoza Acosta
    suspend fun update(uuid: String, producto: Producto): Producto?
    Sebastián Mendoza Acosta
    suspend fun delete(uuid: String): Producto?
    Sebastián Mendoza Acosta
    suspend fun findAllPage(pageRequest: PageRequest): Flow<Page<Producto>>
```

Usuarios “Interfaz de cache”:



```
/**
 * Interface that the Cached Repository will implement.
 * @author Mario Gonzalez, Daniel Rodriguez, Joan Sebastian Mendoza,
 * Alfredo Rafael Maldonado, Azahara Blanco, Ivan Azagra, Roberto Blazquez
 */
Mario Resa
@Repository
interface IUserRepositoryCached {
    Mario Resa
    suspend fun findAll(): Flow<User>
    Mario Resa
    suspend fun findAllPaged(page: PageRequest): Flow<Page<UserDTOresponse>>
    Mario Resa
    suspend fun findByActivo(activo: Boolean): Flow<User>
    Mario Resa
    suspend fun findById(id: UUID): User?
    Mario Resa
    suspend fun findByEmail(email: String): User?
    Mario Resa
    suspend fun findByUsername(username: String): User?
    Mario Resa
    suspend fun findByPhone(phone: String): User?
    Mario Resa
    suspend fun save(user: User): User
    Mario Resa
    suspend fun deleteById(id: UUID): User?
}
```

Estas interfaces se aplican en los proyectos de Springboot a traves de anotaciones en la claese repositorio como son:



```
@Cacheable("usuarios")
```

La anotación "@Cacheable" se utiliza para cachear los resultados de los métodos de un componente o servicio en la memoria caché.

Esta anotación permite especificar el nombre de la caché a utilizar en la que se almacenará el resultado del método, en este caso "usuarios". Si se llama al mismo método con los mismos argumentos, el resultado se devolverá desde la caché en lugar de ejecutar nuevamente el método.

```
@CachePut("productos")
```

La anotación "@CachePut" se utiliza para actualizar o agregar un valor en la caché. Al contrario que "@Cacheable", "@CachePut" siempre ejecuta el método y luego actualiza la caché con el resultado. Si la clave ya existe en la caché, el valor se actualiza, y si no existe, se crea uno nuevo.

La anotación "@CachePut" se utiliza en un método que modifica el estado del objeto de negocio y, por lo tanto, también modifica el valor almacenado en la caché. Al utilizar esta anotación, se asegura de que la caché siempre esté actualizada con el último estado del objeto.

Repositorios.

En Spring, los repositorios son una capa de abstracción que se utiliza para interactuar con bases de datos o sistemas de almacenamiento de datos. Los repositorios se definen como interfaces que extienden una interfaz genérica llamada "CrudRepository" o en nuestro caso para que sea reactivo, “CoroutinecrudRepository” proporcionada por Spring Data.

Una vez definido el repositorio, es necesario indicar a Spring cómo implementar el acceso a los datos. Para ello, utilizamos @Repository".

Repository de Productos:

```
@Repository
interface ProductosRepository : CoroutineCrudRepository<Producto, String> {
    Sebastián Mendoza Acosta
    fun findByNombre(nombre: String): Flow<Producto>
    Sebastián Mendoza Acosta
    fun findByCategoria(categoria: String): Flow<Producto>
    Sebastián Mendoza Acosta
    fun findByUuid(id: String): Flow<Producto>
    Sebastián Mendoza Acosta
    fun findAllBy(pageable: Pageable?): Flow<Producto>
}
```

Repository de Usuarios:

```
@Repository
interface UserRepository : CoroutineCrudRepository<User, UUID> {
    Mario Resa
    fun findFirstByEmail(email: String): Flow<User>
    Mario Resa
    fun findByUsername(username: String): Flow<User>
    Mario Resa
    fun findFirstByPhone(phone: String): Flow<User>
    Mario Resa
    fun findAllByActiveOrderByCreatedAt(active: Boolean): Flow<User>
    Mario Resa
    fun findAllBy(page: Pageable?): Flow<User>
```

Repository de Pedidos en el caso de ktor es algo diferente:

Esta es parte de la implementación del repositorio en Ktor utilizando MongoDB como base de datos. El repositorio se encarga de realizar operaciones CRUD en la colección de Pedidos de la base de datos.

```kotlin
const val MAX_SIZE = 500

Roberto Blázquez
class PagedFlow<T>(val page: Int, val size: Int, results: Flow<T>) : Flow<T> by results

Roberto Blázquez +2
class PedidosRepository(private val collection: CoroutineCollection<Pedido>) {

    Bank +1
    suspend fun getByPage(page: Int, size: Int): Either<PedidoError, PagedFlow<Pedido>> = either {
        validatePage(page, size).bind()

        val flow = collection.find().skip(page * size).limit(size).toFlow()
        PagedFlow(page, size, flow) ^either
    }

    Roberto Blázquez +2
    suspend fun getById(id: String): Either<PedidoError, Pedido> {
        val _id = id.toObjectIdOrNull()?.toId<Pedido>()
            ?: return PedidoError.InvalidPedidoId("id : '$id' format is incorrect").left()

        return collection.find(Pedido::_id eq _id).first()?.right()
            ?: PedidoError.PedidoNotFound("Pedido with with id : '${id}' not found").left()
    }
```

Autorización y Autentificación.

La autenticación y la autorización son dos conceptos relacionados pero distintos en la seguridad de una aplicación. La autenticación se refiere a la verificación de la identidad de un usuario, mientras que la autorización se refiere a si ese usuario tiene permiso para realizar ciertas acciones en la aplicación.

Los tokens son la forma que hemos usado en nuestro proyecto para implementar autenticación y autorización en aplicaciones entre las distintas Apis.

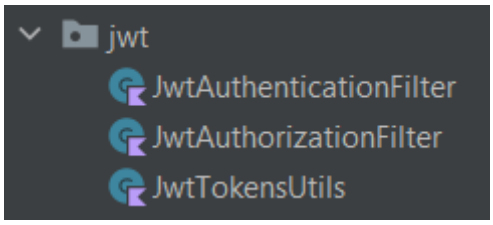
Un token es una cadena de caracteres que representa la identidad de un usuario y su estado de autenticación y autorización.

En el proceso un usuario ingresa sus credenciales (nombre de usuario y contraseña etc.) en un formulario de inicio de sesión, este pasará en formato JSON a nuestro api de Usuarios. Si las credenciales son válidas, el servidor genera un token único que se devuelve al cliente. Este token se incluye en las solicitudes posteriores del usuario a esta y las otras apis del sistema para que estas puedan verificar la identidad del usuario y sus permisos.

La ventaja de utilizar tokens es que son portátiles y escalables, lo que significa que el usuario puede utilizar el mismo token para autenticarse y autorizarse en diferentes partes de la aplicación o incluso en diferentes aplicaciones. Además, los tokens pueden ser válidos por un período de tiempo limitado, lo que mejora la seguridad de la aplicación, ya que los tokens expirados no se pueden utilizar para acceder a recursos de la aplicación.

Autorización y Autentificación en Usuarios.

Es la única Api de nuestro programa que crea los tokens y loga a los usuarios.



Por ello necesita las siguientes tres clases:

1. JwtAuthenticationFilter

La clase JwtAuthenticationFilter es un filtro de autenticación de Spring Security que se encarga de validar las credenciales del usuario (nombre de usuario y contraseña etc) y generar un token JWT si son correctas. El filtro extiende la clase

UsernamePasswordAuthenticationFilter que es una clase de Spring Security para procesar solicitudes de autenticación basadas en nombre de usuario y contraseña.

La clase tiene tres métodos importantes:

El método attemptAuthentication() se encarga de intentar autenticar al usuario utilizando las credenciales proporcionadas en la solicitud. Si las credenciales son correctas, se llama al método successfulAuthentication().

El método successfulAuthentication() se llama cuando el usuario se ha autenticado correctamente. Este método genera un token JWT utilizando un objeto JwtTokenUtils y lo añade a la respuesta.

El método unsuccessfulAuthentication() se llama cuando la autenticación ha fallado. En este caso, se devuelve un objeto BadCredentialsError con un mensaje de error y un estado HTTP 401 (No autorizado).

2. JwtAuthorizationFilter

Esta clase extiende la clase BasicAuthenticationFilter, lo que significa que intercepta todas las solicitudes que pasan por el filtro y verifica si el usuario que las realiza está autorizado para realizar la acción solicitada.

El método doFilterInternal es el método principal de la clase, y se ejecuta cada vez que se recibe una solicitud HTTP que pasa por el filtro. El método verifica si el encabezado Authorization de la solicitud contiene un token JWT válido. Si no lo es, la solicitud se envía a través de la cadena de filtros sin hacer nada más. Si es válido, el método llama al método getAuthentication, que se encarga de decodificar el token y cargar la información del usuario correspondiente desde la base de datos.

3. JwtTokensUtils

Esta clase es una utilidad para la creación y decodificación de tokens JWT (JSON Web Tokens).

Pruebas y verificación: descripción de las pruebas realizadas para validar el correcto funcionamiento de los microservicios y las APIs, incluyendo pruebas unitarias, pruebas de integración, pruebas de rendimiento, pruebas de seguridad.

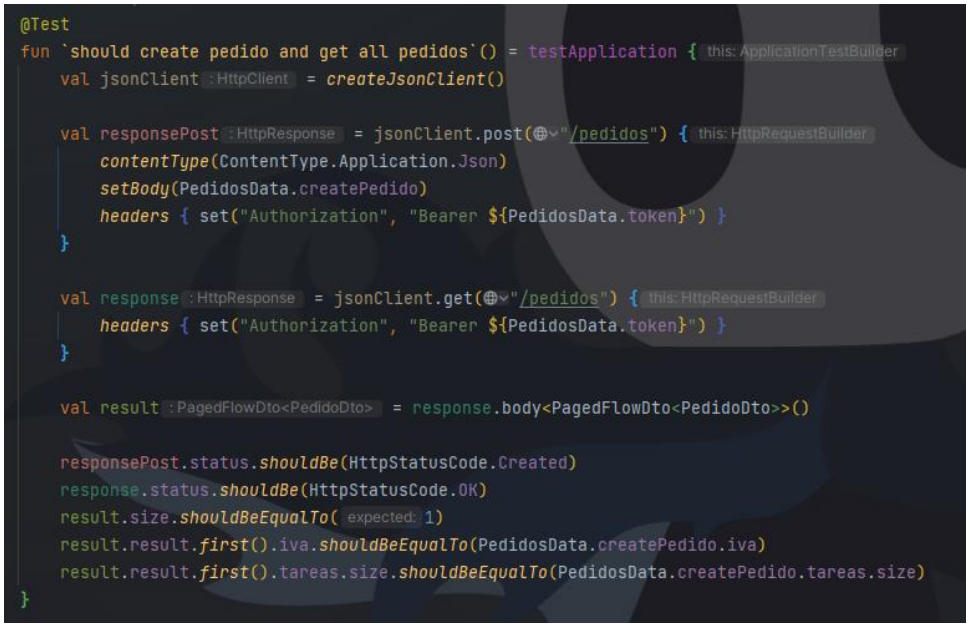
Hemos realizado distintas pruebas para confirmar el correcto funcionamiento de los servicios, se han realizado pruebas de integración, unitarias y test mockeados utilizando MockK en el caso de Ktor y springmockk dentro de las apis que utilizan spring.

Los test de integración son pruebas que comprueban que los distintos componentes de la aplicación trabajan juntos correctamente y cumplen los requerimientos de la especificación, estos test son útiles para encontrar fallos que no se detectarían en los test de pruebas unitarias.

Los test unitarios son pruebas automatizadas para corroborar el funcionamiento esperado, estas pruebas se ejecutan de manera aislada del resto de componentes, por lo que se prueban sin dependencias externas, estas pruebas se realizan antes de integrar componentes para detectar posibles errores en una etapa temprana.

Por último, hemos realizado test mockeados que son aquellos test que simulan el comportamiento de la aplicación, son pruebas automatizadas para aislar el comportamiento y probarlo de manera independiente al resto del sistema, en estas pruebas se utilizan objetos simulados, también conocidos como mocks, los cuales imitan los comportamientos de las dependencias reales, esto nos sirve para simular las condiciones que pueden ocurrir durante la ejecución.

Test de integración en Ktor:

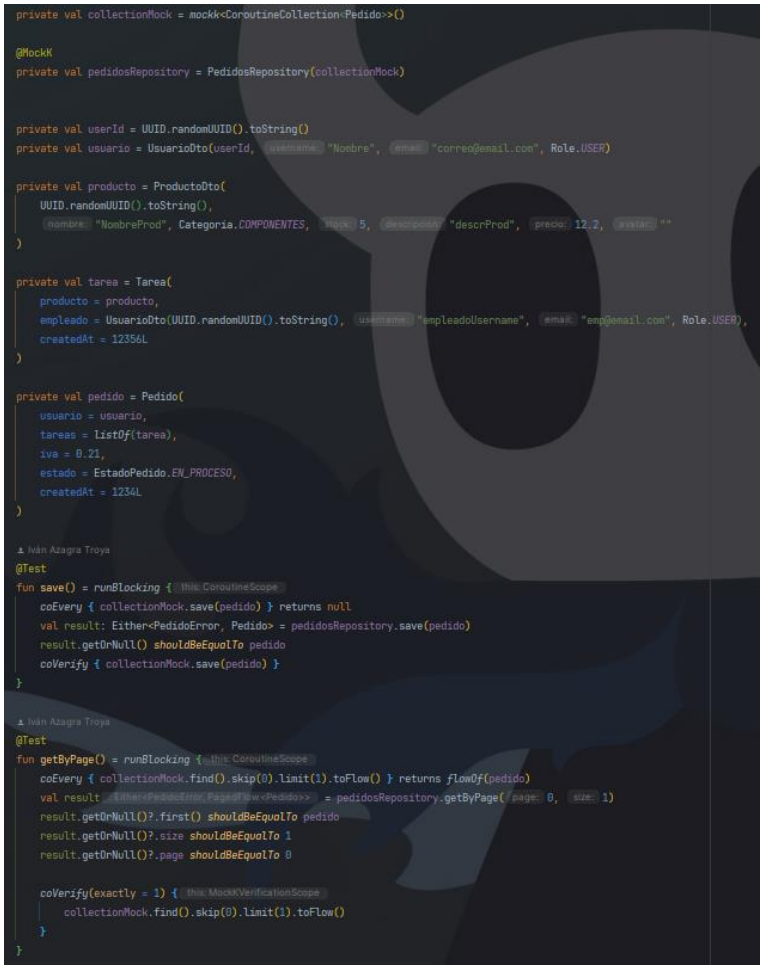


```kotlin
@Test
fun `should create pedido and get all pedidos`() = testApplication { this: ApplicationTestBuilder
    val jsonClient :HttpClient = createJsonClient()

    val responsePost :HttpResponse = jsonClient.post("/pedidos") { this: HttpRequestBuilder
        contentType(ContentType.Application.Json)
        setBody(PedidosData.createPedido)
        headers { set("Authorization", "Bearer ${PedidosData.token}") }
    }

    val response :HttpResponse = jsonClient.get("/pedidos") { this: HttpRequestBuilder
        headers { set("Authorization", "Bearer ${PedidosData.token}") }
    }

    val result :PagedFlowDto<PedidoDto> = response.body<PagedFlowDto<PedidoDto>>()

    responsePost.status.shouldBe(HttpStatusCode.Created)
    response.status.shouldBe(HttpStatusCode.OK)
    result.size.shouldBeEqualTo( expected: 1)
    result.result.first().iva.shouldBeEqualTo(PedidosData.createPedido.iva)
    result.result.first().tareas.size.shouldBeEqualTo(PedidosData.createPedido.tareas.size)
}
```

Se ha utilizado Kluent para una definición de test más legibles con nombres sin estar en “formato Kotlin”. Creamos un contenedor para las pruebas con cada ejecución para que esté vacío siempre, en el caso de Ktor utilizamos una base de datos de Mongo.

En estos test hacemos una petición al api, para ello creamos un cliente el cual realiza la petición, después comprobamos la respuesta o respuestas comparándolo con los resultados esperados por un funcionamiento correcto. En el caso de los Flows comprobamos el primer resultado.

Test mockeados en Ktor:



```
private val collectionMock = mockk<CoroutineCollection<Pedido>>()

@MockK
private val pedidosRepository = PedidosRepository(collectionMock)


private val userId = UUID.randomUUID().toString()
private val usuario = UsuarioDto(userId,  username: "Nombre",  email: "correo@email.com", Role.USER)

private val producto = ProductoDto(
    UUID.randomUUID().toString(),
     nombre: "NombreProd", Categoria.COMPONENTES,  stock: 5,  descripcion: "descrProd",  precio: 12.2,  avatar: ""
)

private val tarea = Tarea(
    producto = producto,
    empleado = UsuarioDto(UUID.randomUUID().toString(),  username: "empleadoUsername",  email: "emp@email.com", Role.USER),
    createdAt = 12356L
)

private val pedido = Pedido(
    usuario = usuario,
    tareas = listOf(tarea),
    iva = 0.21,
    estado = EstadoPedido.EN_PROCESO,
    createdAt = 1234L
)

 Iván Azagra Troya
@Test
fun save() = runBlocking {  this: CoroutineScope
    coEvery { collectionMock.save(pedido) } returns null
    val result: Either<PedidoError, Pedido> = pedidosRepository.save(pedido)
    result.getOrNull() shouldBeEqualTo pedido
    coVerify { collectionMock.save(pedido) }
}

 Iván Azagra Troya
@Test
fun getByPage() = runBlocking {  this: CoroutineScope
    coEvery { collectionMock.find().skip(0).limit(1).toFlow() } returns flowOf(pedido)
    val result  : Either<PedidoError, PageOf<Pedido>> = pedidosRepository.getByPage( page: 0,  size: 1)
    result.getOrNull()?.first() shouldBeEqualTo pedido
    result.getOrNull()?.size shouldBeEqualTo 1
    result.getOrNull()?.page shouldBeEqualTo 0

    coVerify(exactly = 1) {  this: MockKVerificationScope
        collectionMock.find().skip(0).limit(1).toFlow()
    }
}
```

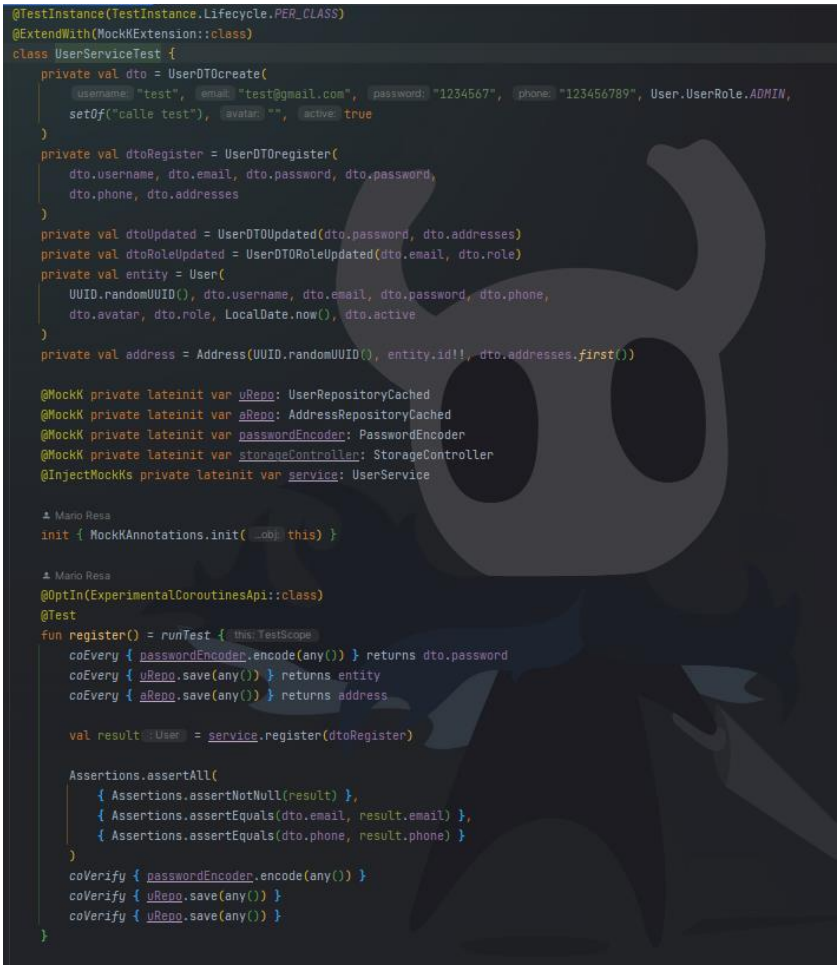
Tenemos que tener ciertos datos de prueba para comprobar el funcionamiento, para ello mockeamos un CoroutineCollection para simular el funcionamiento de la base de datos que es utilizado desde el repositorio.

En el repositorio trabajamos con Either, de la libería Arrow de Kotlin, por lo que este tipo encapsula dos resultados, estos se guardan uno a la izquierda y otro a la derecha, en la izquierda se guardan los resultados de error y a la derecha los resultados esperados en caso de funcionar correctamente, por esto al realizar el test del método mockeamos la función del repositorio y después recogemos el resultado del Either con el que comprobamos el resultado almacenado con un getOrNull ya que en caso de ser error recibiremos como respuesta un nulo y nos interesa el resultado correcto, después de esto se verifica la respuesta mockeada.

Test mockeados en Springboot

Se preparan los datos para la comprobación del funcionamiento de los métodos del repositorio, estos test funcionan como los dichos anteriormente, pero sin el uso de Eithers, mockeamos el repositorio y llamamos al método a testear, este nos devuelve un flow con los productos, este resultado lo pasamos a lista para después comprobarlo dentro de los asserts, comprobando que no hay resultados nulos, los códigos y el número de resultados como si es el resultado esperado.

```kotlin
@ExtendWith(MockKExtension::class)
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class ProductoControllerTest {

    @MockK
    private lateinit var repository: ProductoCachedRepositoryImpl

    @InjectMockKs
    lateinit var controller: ProductoController

    Sebastián Mendoza Acosta
    init {
        MockKAnnotations.init( obj: this)
    }

    private var producto = Producto(
        id = 0L,
        uuid = "a765df90-e4aa-4306-b93e-20600accf8f7",
        nombre = "Teclado",
        categoria = Producto.Categoria.PIEZA,
        stock = 3,
        description = "Teclado para ordenador de sobremesa",
        precio = 15.50,
        activo = true
    )

    val productoDto = producto.toDto()

    val productoCreateDto = ProductoCreateDto(
        nombre = "Movil",
        categoria = Producto.Categoria.PIEZA.name,
        stock = 3,
        description = "Teclado para ordenador de sobremesa",
        precio = 15.50,
        activo = true.toString()
    )

    val superAdmin = User(
        username = "superadmin",
        email = "superadmin@admin.com",
        password = "super1234",
        role = User.UserRole.SUPER_ADMIN,
        active = true
    )

    Sebastián Mendoza Acosta
    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun findAllProductos() = runTest { this: TestScope
        coEvery { repository.findAll() } returns flowOf(producto)
        val result: ResponseEntity<Flow<ProductoDto>> = controller.findAllProductos()
        val res: List<ProductoDto> = result.body!!.toList()

        assertAll(
            { assertNotNull(result) },
            { assertNotNull(res) },
            { assertEquals(result.statusCode, HttpStatus.OK) },
            { assertEquals( expected: 1, res.count()) },
            { assertEquals(productoDto.nombre, res[0].nombre) }
        )

        coVerify { repository.findAll() }
    }
```



```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@ExtendWith(MockKExtension::class)
class UserServiceTest {
    private val dto = UserDTOcreate(
        username: "test", email: "test@gmail.com", password: "1234567", phone: "123456789", User.UserRole.ADMIN,
        setOf("calle test"), avatar: "", active: true
    )
    private val dtoRegister = UserDTOregister(
        dto.username, dto.email, dto.password, dto.password,
        dto.phone, dto.addresses
    )
    private val dtoUpdated = UserDTOUpdated(dto.password, dto.addresses)
    private val dtoRoleUpdated = UserDTORoleUpdated(dto.email, dto.role)
    private val entity = User(
        UUID.randomUUID(), dto.username, dto.email, dto.password, dto.phone,
        dto.avatar, dto.role, LocalDate.now(), dto.active
    )
    private val address = Address(UUID.randomUUID(), entity.id!!, dto.addresses.first())

    @MockK private lateinit var uRepo: UserRepositoryCached
    @MockK private lateinit var aRepo: AddressRepositoryCached
    @MockK private lateinit var passwordEncoder: PasswordEncoder
    @MockK private lateinit var storageController: StorageController
    @InjectMockKs private lateinit var service: UserService

    Mario Resa
    init { MockKAnnotations.init( obj: this) }

    Mario Resa
    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun register() = runTest { this: TestScope
        coEvery { passwordEncoder.encode(any()) } returns dto.password
        coEvery { uRepo.save(any()) } returns entity
        coEvery { aRepo.save(any()) } returns address

        val result : User = service.register(dtoRegister)

        Assertions.assertAll(
            { Assertions.assertNotNull(result) },
            { Assertions.assertEquals(dto.email, result.email) },
            { Assertions.assertEquals(dto.phone, result.phone) }
        )
        coVerify { passwordEncoder.encode(any()) }
        coVerify { uRepo.save(any()) }
        coVerify { uRepo.save(any()) }
    }
```

La clase UserServiceTest mockea los repositorios utilizados para su funcionamiento, también mockea clases que aportan funcionalidades como el codificador de contraseñas y luego los inyecta a User service, después se realizan las comprobaciones de la clase haciendo uso de los distintos repositorios y comprobando los resultados esperados.

Recomendaciones para futuros proyectos o para la mejora de este proyecto:

Comunicación más fluida

Mejorar la comunicación efectiva con todos los compañeros de equipo para una solución de problemas más rápida y para el mejor mantenimiento del código realizado por otros compañeros.

Para ello:

Algunas recomendaciones podrían ser:

1. Establecer reuniones regulares con más antelación para asegurar que todos los miembros del equipo estén al tanto de los avances y problemas del proyecto y asistan a las reuniones.

2. Usar herramientas de comunicación que todos tengan en rápido acceso para mantener una comunicación más fluida entre los miembros del equipo.

3. Identificar posibles obstáculos de comunicación, como barreras culturales o de idioma, y encontrar soluciones para superarlos.
4.

En general, la idea es promover una cultura de comunicación abierta y efectiva dentro del equipo, ya que esto ya se ha realizado durante el proceso de realización, pero seguir con esta cultura para que todos puedan trabajar juntos de manera más eficiente y productiva.

Añadirle aluna interfaz y un Gateway

Añadirle aluna interfaz y un Gateway para hacer del usuario una experiencia personalizada y por lo tanto dar mejor rendimiento a un cliente especifico.

En este caso no hemos realizado estas funciones para que nuestro código sea más adaptable a distintas aplicaciones donde solo sean necesarios algunos módulos de este, pero para la experiencia total de un único cliente especializado pensamos que sería mejor una interfaz y un Gateway. Que si es posible realizaremos más adelante.

Bibliografía: lista de las fuentes de información utilizadas para la elaboración de la memoria del proyecto.

- Spring Data - https://docs.spring.io/spring-data/r2dbc/docs/current/reference/html/

- Spring Boot - https://spring.io/projects/spring-boot

- Corrutinas Kotlin - https://kotlinlang.org/docs/coroutines-overview.html

- MariaDB - https://mariadb.org/

- PostgreSQL - https://www.postgresql.org/

- R2DBC - https://r2dbc.io/

- Mockk - https://mockk.io/

- Dokka - https://github.com/Kotlin/dokka

- Auth0 JWT - https://auth0.com/docs/secure/tokens/json-web-tokens

- Swagger-SpringDoc-OpenAPI - https://springdoc.org/v2/

- JUnit5 - https://junit.org/junit5/

- Postman - https://www.postman.com/

- VSC - https://code.visualstudio.com/

- Thunder Client - https://www.thunderclient.com