



ACCESO A DATOS Y PROGRAMACIÓN DE SERVICIOS Y PROCESOS

Realizado por:

Alejandro Sánchez Monzón
Daniel Rodríguez Fernández
Mireya Sánchez Pinzón
Jorge Sánchez Berrocoso
Ruben García-Redondo Marín
Sergio Pérez Fernández

07/03/2023

Proyecto	BiquesDAM		
Entregable	Especificación de Requisitos		
Autor	Daniel Rodríguez Fernández – Jorge Sánchez Berrocoso – Alejandro Sánchez Monzón - Mireya Sánchez Pinzón - Rubén García Redondo Marín - Sergio Pérez Fernández		
Versión/Edición	V1.1	Fecha Versión	06/03/2023
Aprobado por		Fecha Aprobación	08/03/2023
		Nº Total de Páginas	139

PROJECT MANAGER

Ruben García-Redondo Marín

PRODUCT OWNER

Daniel Rodríguez Fernández

PROGRAMMERS

Alejandro Sánchez Monzón

Jorge Sánchez Berrocoso

Mireya Sánchez Pinzón

Sergio Pérez Fernández

1.	Introducción.....	6
2.	Descripción del problema propuesto.....	7
3.	Diagrama & Estructura.....	8
3.1.	Diagrama de Clase.....	8
3.2.	Estructura de Endpoints y Seguridad.....	9
4.	Requisitos.....	11
4.1.	Requisitos Funcionales.....	11
4.2.	Requisitos No Funcionales	12
4.3.	Requisitos de Información	13
5.	Evaluación y Análisis	15
5.1.	Microservicio A ()	15
5.1.1.	DTO.....	15
5.1.2.	Config	20
5.1.3.	Exceptions	20
5.1.4.	Plugins	21
5.1.5.	Repositories	23
5.1.6.	Routes	26
5.1.7.	Services	36
5.1.8.	Tests	38
5.2.	Microservicio B ()	41
5.2.1.	Modelos	41
5.2.2.	DB.....	43
5.2.3.	DTO.....	43
5.2.4.	Mappers	44
5.2.5.	Repository	45
5.2.6.	Services	48
5.2.7.	Controllers.....	52
5.2.8.	Utils	56
5.2.9.	Validators	57
5.2.10.	Config	57
5.2.11.	Tests	61
5.3.	72
5.3.1.	Modelos	72
5.3.1.1.	Modelo Appointment.....	72
5.3.1.2.	Interfaz OnSale.....	72
5.3.1.3.	Modelo Product	73
5.3.1.4.	Modelo Service	74
5.3.2.	Mappers	75

5.3.2.1.	Appointment Mapper	75
5.3.3.2	Product Mapper	76
5.3.2.2.	Service Mapper	77
5.3.4.1.	AppointmentDTO	77
5.3.4.2.	OnSaleDTO	78
5.3.4.3.	ProductDTO.....	79
5.3.4.4.	ServiceDTO.....	80
5.3.5.	Repositorios	80
5.3.5.1.	AppointmentsCachedRepository, IAppointmentsCachedRepository, AppointmentsRepository	80
5.3.5.2.	ProductsCachedRepository, IProductsCachedRepository, ProductsRepository.....	82
5.3.5.3.	ServiceRepository.....	84
5.3.6.	Services	85
5.3.6.1.	AppointmentsService, IAppointmentsService	85
5.3.7.	ProductsService, IProductsService.....	86
5.3.8.	ServicesService, IServicesService	88
5.3.9.	StorageService, IStorageService.....	89
5.3.10.	Validators	93
5.3.10.1.	AppointmentValidator	93
5.3.10.2.	ProductValidator	93
5.3.10.3.	ServiceValidator	94
5.3.11.	Serializers	94
5.3.11.1.1.	LocalDateSerializer.....	94
5.3.11.1.2.	UUIDSerializer	94
5.3.12.	Exceptions	95
5.3.12.1.	AppointmentException	95
5.3.12.2.	OnSaleException	95
5.3.12.3.	ProductException.....	95
5.3.12.4.	ServiceException	96
5.3.12.5.	StorageException	96
5.3.13.	Utils: PropertiesReader	97
5.3.14.	Controllers.....	98
5.3.14.1.	ProductsServicesController.....	98
5.3.14.2.	StorageController.....	102
5.3.15.	La clase principal: MicroservicioProductoServiciosApplication	103
5.3.16.	Tests	103
5.3.16.1.	La clase principal de los tests: MicroservicioProductoServiciosApplicationTests	103
5.3.16.2.	ProductsServicesControllerTest.....	104
5.3.16.3.	AppointmentsCachedRepositoryTest	105

5.3.16.4.	ProductsCachedRepositoryTest	107
5.3.16.5.	AppointmentServiceTest.....	108
5.3.16.6.	ProductsServiceTest.....	109
5.3.16.7.	ServicesServiceTest.....	110
5.4.	Microservicio D ()	111
5.4.1.	MODELOS	111
5.4.2.	MAPPER.....	113
5.4.3.	DTO.....	115
5.4.4.	DB.....	116
5.4.5.	REPOSITORIES	116
5.4.6.	SERVICES	118
5.4.7.	VALIDATOR.....	120
5.4.8.	CONFIG.....	121
5.4.9.	PLUGINS	122
5.4.10.	EXCEPTIONS	126
5.4.11.	SERIALIZERS.....	127
5.4.12.	UTILS.....	128
5.4.13.	APPLICATION CONF.....	129
5.4.14.	TEST	130
6.	Enlace proyecto.....	136
7.	¡Error! Marcador no definido.

1. Introducción

Se ha creado un grupo de 6 alumnos/as de 2DAM para resolver una práctica compuesta para el módulo de Acceso a Datos y Programación de Servicios y Procesos. Como perfiles importantes necesitamos un Project Manager que represente al grupo y un Product Owner que entienda y conozca el producto a desarrollar.

El profesor de ambos módulos nos ha dado los requisitos mínimos que tiene que cumplir dicha práctica y se tratan de puntos importantes a evaluar.

Algunos de estos requisitos mínimos son los siguientes: utilizar tres gestores de bases de datos diferentes, utilizar tanto Ktor como Springboot para crear los microservicios, tener diferentes microservicios, cada microservicio implementa sus propias operaciones CRUD, todos los microservicios deben estar protegidos así como sus rutas, se debe desplegar todas las bases de datos a través de docker, alguno de los modelos debe estar cacheados.

El tiempo estimado para realizar dicha práctica será de 30 días y se entregará y defenderá en clase el día 08 de marzo de 2023.

2. Descripción del problema propuesto

Necesitamos resolver un sistema que nos permita gestionar los productos, servicios y pedidos de una tienda de bicicletas de Leganés llamada Burning Bikes. El problema propuesto consiste en la integración y comunicación entre tres microservicios: B, C y D. El microservicio B es responsable de manejar los usuarios con sus respectivos roles, utilizando Spring Boot y Maria DB como base de datos. El microservicio C gestiona los servicios y productos utilizando Spring Boot y H2 como base de datos, mientras que el microservicio D gestiona los pedidos y las líneas de pedidos utilizando Ktor y MongoDB como base de datos no relacional.

El principal problema es garantizar que los tres microservicios puedan comunicarse y compartir información de manera eficiente y segura. Esto implica diseñar una arquitectura de microservicios robusta que permita una fácil integración y escalabilidad.

Además, es importante garantizar la seguridad de los datos y la autenticación de los usuarios en todos los microservicios, especialmente en el microservicio B, que es responsable de manejar la información de los usuarios y sus roles.

En el problema propuesto, la comunicación y unificación de los tres microservicios (B, C y D) se llevará a cabo en el microservicio A, utilizando el framework Ktor.

El microservicio A será responsable de integrar y coordinar la comunicación entre los otros tres microservicios, utilizando las interfaces de programación de aplicaciones (APIs) proporcionadas por cada uno de ellos. Además, el microservicio A también manejará la lógica de negocio que involucra a los tres microservicios, proporcionando una vista unificada y coherente de los datos.

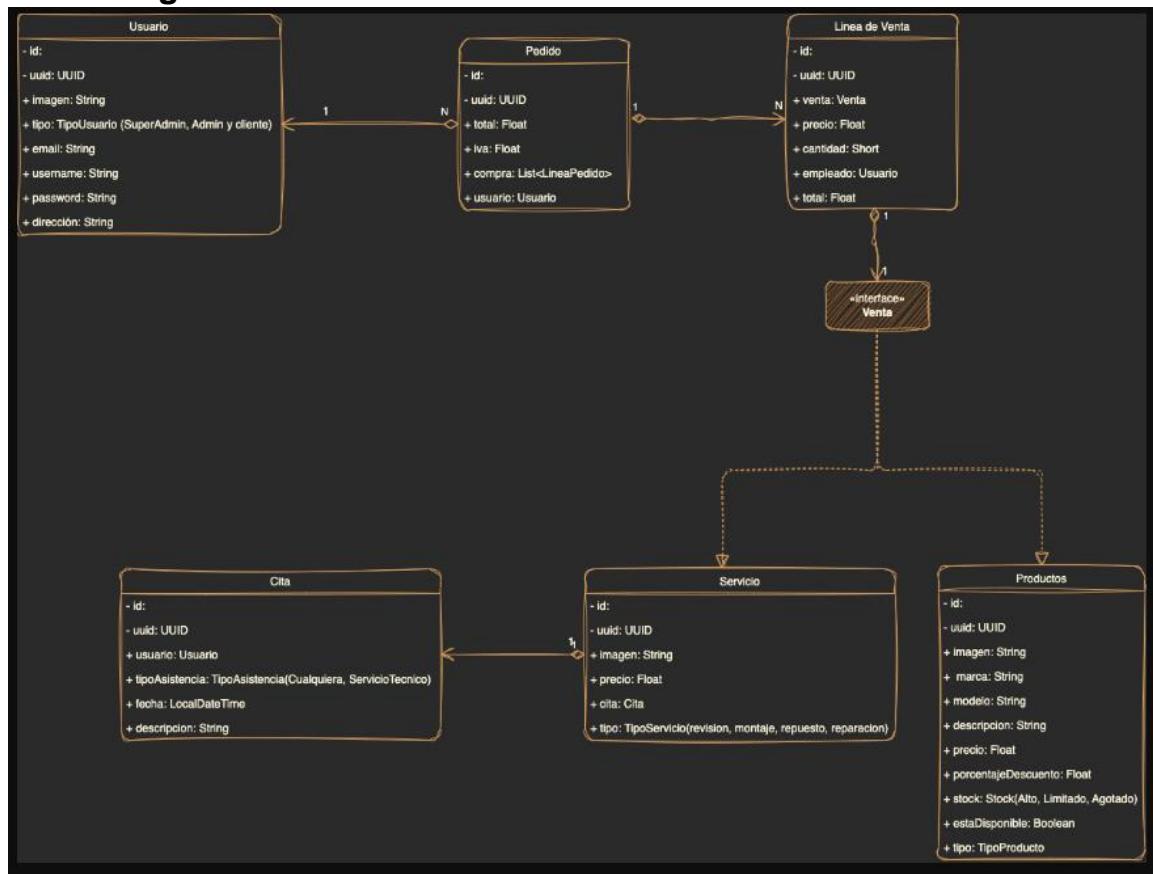
Para lograr esto, se necesitará un diseño cuidadoso de la arquitectura de microservicios, así como la implementación de mecanismos de comunicación y coordinación, como la mensajería asincrónica y la propagación de eventos.

Además, se deberá establecer un sistema de autenticación y autorización centralizado en el microservicio A, que permita a los usuarios acceder a los diferentes microservicios según su rol y permisos asignados.

En resumen, el problema propuesto implica la integración y coordinación de tres microservicios a través de un microservicio centralizado, utilizando Ktor como herramienta de comunicación y unificación. La solución requerirá un diseño cuidadoso de la arquitectura de microservicios, así como la implementación de mecanismos de seguridad y coordinación para garantizar una comunicación eficiente y segura entre los diferentes componentes del sistema.

3. Diagrama & Estructura

3.1. Diagrama de Clase



La clase Usuario tiene los atributos id, UUID, avatar, tipo de usuario (que puede ser cliente, admin o superadmin), email, username, password (cifrado con Bcrypt) y dirección. La clase Usuario tiene una relación uno a muchos con la clase Pedido, que tiene los atributos id, uuid, total, iva y usuario que realizó el pedido.

Además, la clase Pedido tiene una relación muchos a uno con la clase LineaDeVenta, que tiene los atributos id, UUID, venta, precio, cantidad, usuario y total. La clase LineaDeVenta tiene una relación uno a uno con la interfaz Venta, que es implementada por las clases Servicio y Producto.

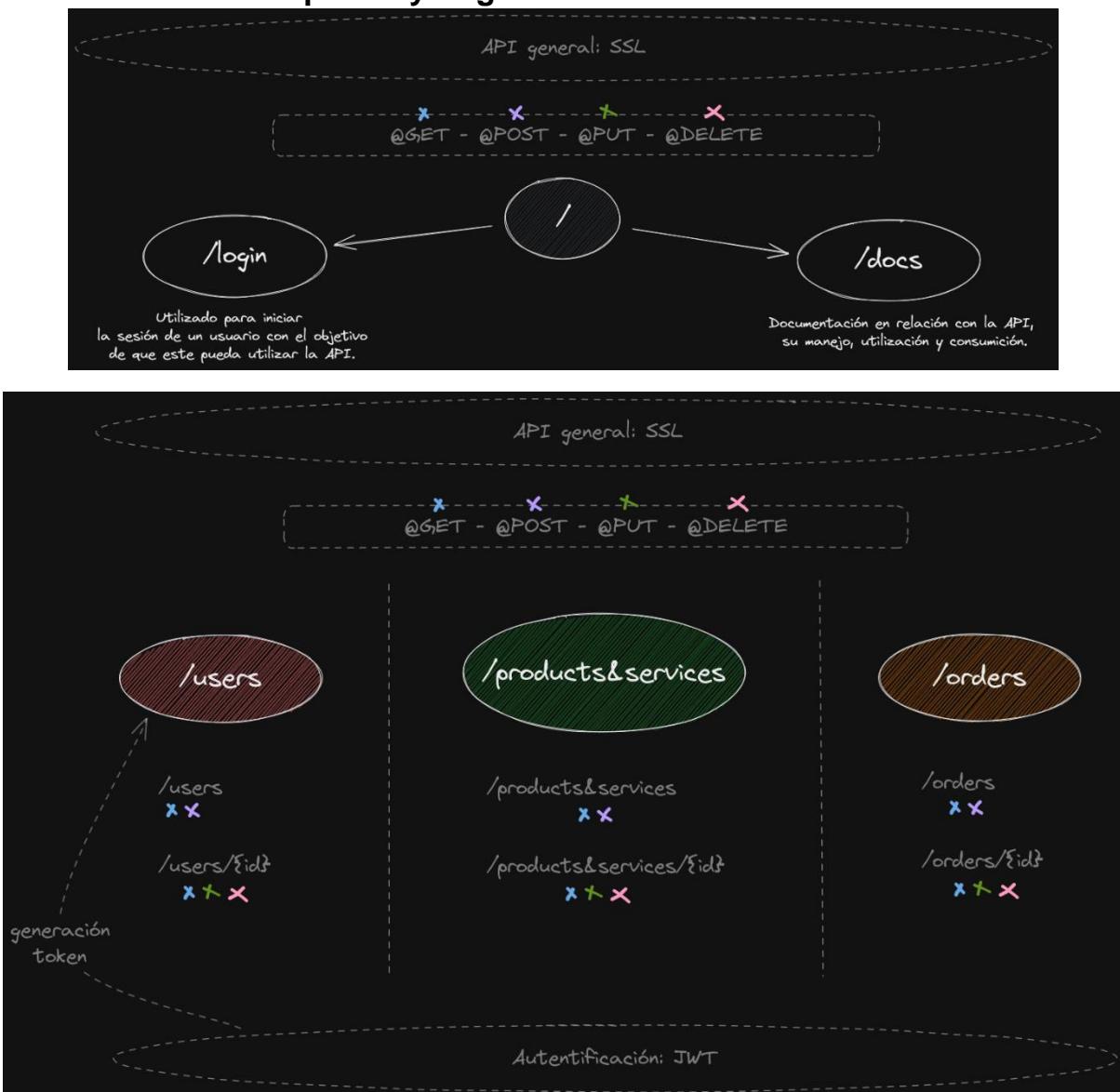
La clase Producto tiene los atributos id, UUID, imagen, marca, modelo, descripción, precio, porcentajeDescuento, stock, disponible (un valor booleano que indica si el producto está disponible o no) y tipo de producto. La clase Servicio implementa la interfaz Venta y tiene los atributos id, UUID, imagen, precio, cita y tipo de servicio, que puede ser revisión, montaje, repuesto o reparación.

Por último, la clase Servicio tiene una relación uno a uno con la clase Cita, que tiene los atributos id, UUID, usuario, tipo de asistencia, fecha y descripción.

En resumen, la clase Usuario representa a los usuarios del sistema, la clase Pedido registra los pedidos realizados por los usuarios y la clase LineaDeVenta registra las ventas de los productos y servicios.

La interfaz Venta define los atributos que tienen tanto los productos como los servicios, que son implementados por las clases Producto y Servicio. La clase Servicio también tiene una relación con la clase Cita, que representa las citas programadas para los servicios.

3.2. Estructura de Endpoints y Seguridad



La estructura de endpoints que hemos organizado es la siguiente:

Comenzando por los microservicios no visibles:

- Usuarios:

Su endpoint terminará en “/users” y al recurso que accede son los usuarios registrados en el sistema.

Se encargará de generar el token para cada nuevo usuario permitiendo así navegar a través de los demás recursos haciendo uso de este token para autenticar.

- Productos/Servicios:

Su endpoint terminará en “/products&services” y al recurso que accede son todos los productos y servicios que la empresa tiene datos de alta. Como aclaración, este recurso incluye todos los productos, estén o no clasificados como disponible o en los que se especifique que haya o no stock. Será responsabilidad del software que lo use hacerse cargo de lo que se va a mostrar a cada tipo de usuario.

Se encargará de validar el token que le llegue para poder realizar las operaciones que se soliciten. Teniendo en cuenta tanto la autenticación del token como que el usuario esté autorizado.

- Pedidos:

Su endpoint terminará en “/orders” y al recurso que accede son los pedidos registrados en el sistema. Este recurso incluye todos los pedidos que se han realizado independientemente de su estado, es decir, si se acaban de crear, si están en proceso, o bien estén finalizados.

A partir del propio resumen del pedido seremos capaces de generar una “factura.”

Se encargará de validar el token que le llegue para poder realizar las operaciones que se soliciten. Teniendo en cuenta tanto la autenticación del token como que el usuario esté autorizado.

Siguiendo por la API REST visible:

Tendremos dos endpoints adicionales, además de los de los microservicios.

- Login:

Tendremos un recurso que permitirá, a usuarios que ya existan en el sistema, iniciar sesión para poder acceder, dependiendo de sus permisos, a unos recursos u otros.

- Docs:

Tendremos un recurso que permitirá, acceder a la documentación que iremos ampliando a lo largo de la práctica.

4. Requisitos

4.1. Requisitos Funcionales

Un requisito funcional define una función del sistema de software o sus componentes.

Cod	Requisito Funcional	Check
RF01	Registro de usuarios: El sistema debe permitir la creación de nuevos usuarios con los siguientes datos obligatorios: email, username, password y tipo de usuario.	
RF02	Gestión de usuarios: El sistema debe permitir la edición y eliminación de usuarios existentes. Los usuarios deben poder actualizar su propia información de perfil.	
RF03	Registro de pedidos: El sistema debe permitir la creación de nuevos pedidos, incluyendo la información del usuario que realizó el pedido y el total de la compra.	
RF04	Gestión de pedidos: El sistema debe permitir la búsqueda y visualización de todos los pedidos realizados. Los administradores deben poder actualizar y eliminar pedidos.	
RF05	Registro de ventas: El sistema debe permitir el registro de ventas de productos y servicios en los pedidos.	
RF06	Gestión de ventas: El sistema debe permitir la búsqueda y visualización de todas las ventas registradas. Los administradores deben poder actualizar y eliminar ventas.	
RF07	Gestión de productos: El sistema debe permitir la gestión de los productos disponibles para la venta, incluyendo la creación, edición y eliminación de productos.	
RF08	Gestión de servicios: El sistema debe permitir la gestión de los servicios disponibles para la venta, incluyendo la creación, edición y eliminación de servicios.	
RF09	Registro de citas: El sistema debe permitir la creación de citas para los servicios, incluyendo la información del usuario que solicitó la cita, el tipo de asistencia y la fecha de la cita.	
RF10	Gestión de citas: El sistema debe permitir la búsqueda y visualización de todas las citas registradas. Los administradores deben poder actualizar y eliminar citas.	
RF11	Autenticación y autorización: El sistema debe requerir autenticación para acceder a las funciones protegidas y verificar la autorización para realizar ciertas acciones basadas en el tipo de usuario.	
RF12	Seguridad: El sistema debe garantizar la seguridad de los datos del usuario, incluyendo la protección de contraseñas y la encriptación de datos sensibles.	

4.2. Requisitos No Funcionales

Los Requisitos No Funcionales se refieren a todos los requisitos que describen características de funcionamiento.

Se suelen clasificar en:

Requisitos de calidad de ejecución, que incluyen seguridad, usabilidad y otros medibles en tiempo de ejecución.

Requisitos de calidad de evolución, como testeabilidad, extensibilidad o escalabilidad, que se evalúan en los elementos estáticos del sistema software.

Cod	Requisito No Funcional	Check
RNF1	Despliegue de bases de datos en docker	
RNF2	Uso de MongoDB como base de datos	
RNF3	Uso de MariaDB como base de datos	
RNF4	Uso de H2 como base de datos	
RNF5	Usar en al menos un microservicio Springboot	
RNF6	Usar en al menos un microservicio Ktor	
RNF7	Usar Koin como inyector de dependencias	
RNF8	Comentar la API con Swagger	
RNF9	Protección de rutas	
RNF10	Uso de Token para la autentificación de usuarios	
RNF11	Usar Koin como inyector de dependencias	
RNF12	Tener testeados repositorios	
RNF13	Tener testeados controladores	
RNF14	Tener testeados rutas	
RNF15	Tener testeados servicios	
RNF16	Tener cacheado al menos un modelo	
RNF17	Protección de las rutas	
RNF18	Tener encriptada la contraseña de los usuarios	

4.3. Requisitos de Información

Cod	Requisito de Información	Check
RI1	Productos:	
	<ul style="list-style-type: none">- id: ObjectId- uuid: UUID- imagen: String- modelo: String- descripcion: String- precio: Float- porcentajeDescuento: Float- stock: Enum- estaDisponible: Boolean- tipo: Enum	
RI2	Servicios:	
	<ul style="list-style-type: none">- id: ObjectId- uuid: UUID- imagen: String- precio: Float- cita: Cita- tipo: Enum	
RI3	Cita:	
	<ul style="list-style-type: none">- id: ObjectId- uuid: UUID- estado: Enum- total: Float- iva: Float- compra: List<LineaPedido>	
RI4	Pedido:	
	<ul style="list-style-type: none">- id: ObjectId- uuid: UUID- estado: Enum- total: Float- iva: Float- compra: List<LineaPedido>- cliente: Usuario	
RI5	Línea de pedido:	
	<ul style="list-style-type: none">- id: ObjectId- uuid: UUID- productoServicio: Producto/Servicio- cantidad: Int- precio: Double- total: Double	

- empleado: Usuario

RI6 Usuario:

- id: ObjectId
- uuid: UUID
- imagen: String
- tipo: Enum
- email: String
- username: String
- password: String
- direccion: String

5. Evaluación y Análisis

5.1. Microservicio A ()

5.1.1. DTO

La primera clase es AppointmentDTO, que representa una cita y tiene propiedades como el ID, el identificador único (UUID), el usuario que la reserva, la asistencia requerida, la fecha y una descripción.

La segunda clase es AppointmentCreateDTO, que representa la creación de una nueva cita de servicios y tiene propiedades como el ID del usuario que la reserva, la cita solicitada, la fecha y una descripción.

Ambas clases están marcadas con la anotación @Serializable para indicar que se pueden serializar en formato JSON.

```
@Serializable
data class AppointmentDTO (
    val id: Long?,
    val uuid: String,
    val user: String,
    val assistance : String,
    val date: String,
    val description: String
)

@Serializable
data class AppointmentCreateDTO (
    val userId: String,
    val assistance : String,
    val date: String,
    val description: String
)
```

Este código define una clase de datos en Kotlin llamada FinalOrderDTO que representa una orden finalizada de un cliente y tiene varias propiedades como:

- uuid: un identificador único para la orden.
- status: el estado actual de la orden.
- total: el costo total de la orden.
- iva: el impuesto al valor agregado aplicado a la orden.
- orderLine: una lista de objetos OrderLineDTO que representan los detalles de la orden.
- cliente: un objeto UserResponseDTO que representa al cliente que realizó la orden.

La clase está marcada con la anotación @Serializable, lo que significa que se puede serializar en formato JSON utilizando la biblioteca serialization.

```
@Serializable
data class FinalOrderDTO (
    val uuid: String,
    val status: String,
    val total: Double,
    val iva: Double,
    val orderLine: List<OrderLineDTO>,
    val cliente: UserResponseDTO
)
```

La primera clase es OrderDTOCreate, que representa una nueva orden de compra que se va a crear y tiene propiedades como el estado, el costo total, el impuesto al valor agregado, una lista de objetos OrderLineCreateDTO que representan los detalles de la orden y el ID del cliente que realizó la orden.

Luego, hay tres clases más: OrderSaveDTO, OrderDTOUpdate y OrderDTO, que representan una orden de compra que ya existe y se van a guardar o actualizar. Estas clases tienen propiedades similares a OrderDTOCreate, excepto que la lista de OrderLineCreateDTO se reemplaza por una lista de cadenas (List<String>) que representan las ID de las líneas de la orden.

Finalmente, está la clase OrderAllDTO que simplemente contiene una lista de objetos OrderDTO y se utiliza para enviar una lista completa de órdenes a través de la API.

```
@Serializable
data class OrderDTOCreate(
    val status: String,
    val total: Double,
    val iva: Double,
    val orderLine: List<OrderLineCreateDTO>,
    val cliente: Long
)

@Serializable
data class OrderSaveDTO(
    val status: String,
    val total: Double,
    val iva: Double,
    val orderLine: List<String>,
    val cliente: Long
)

@Serializable
data class OrderDTOUpdate(
    val status: String,
    val total: Double,
    val iva: Double,
    val orderLine: List<String>,
    val cliente: Long
)

@Serializable
data class OrderDTO(
    val id: String,
    val uuid: String,
    val status: String,
    val total: Double,
    val iva: Double,
    var orderLine: List<String>,
    val cliente: Long
)

data class OrderAllDTO(
    val data: List<OrderDTO>,
)
```

La primera clase es OrderLineDTO, que representa una línea de una orden de compra y tiene propiedades como el ID, el identificador único (UUID), la venta a la que pertenece, la cantidad, el precio, el costo total y el empleado que procesó la línea.

Luego, hay dos clases más: OrderLineCreateDTO y OrderLineUpdateDTO, que representan la creación o actualización de una línea de orden de compra. Estas clases tienen propiedades similares a OrderLineDTO, excepto que las propiedades ID y UUID son opcionales (pueden ser nulas) ya que se generan automáticamente en la base de datos.

Todas las clases están marcadas con la anotación @Serializable para indicar que se pueden serializar en formato JSON.

```
@Serializable
data class OrderLineDTO(
    val id: String?,
    val uuid: String?,
    val sale: String?,
    val amount: Int?,
    val price: Double?,
    val total: Double?,
    val employee: String?
)

@Serializable
data class OrderLineCreateDTO(
    val sale: String,
    val amount: Int,
    val price: Double,
    val total: Double,
    val employee: String
)

@Serializable
data class OrderLineUpdateDTO(
    val sale: String,
    val amount: Int,
    val price: Double,
    val total: Double,
    val employee: String
)
```

La primera clase es ProductDTO, que representa un producto y tiene propiedades como el ID, el identificador único (UUID), la imagen del producto, la marca, el modelo, la descripción, el precio, el porcentaje de descuento, el inventario, si está disponible y el tipo de producto.

Luego, hay otra clase llamada ProductCreateDTO, que se utiliza para crear un nuevo producto. Esta clase tiene propiedades similares a ProductDTO, excepto que el ID es opcional (puede ser nulo) ya que se genera automáticamente en la base de datos.

Ambas clases están marcadas con la anotación @Serializable para indicar que se pueden serializar en formato JSON.

```
@Serializable
data class ProductDTO(
    val id: Long?,
    val uuid: String,
    val image: String,
    val brand: String,
    val model: String,
    val description: String,
    val price: Float,
    val discountPercentage: Float,
    val stock : String,
    val isAvailable: Boolean,
    val type : String
)

@Serializable
data class ProductCreateDTO(
    val image: String,
    val brand: String,
    val model: String,
    val description: String,
```

```

    val price: Float,
    val discountPercentage: Float,
    val stock : String,
    val isAvailable: Boolean,
    val type : String
)

```

La clase SaleDTO representa una venta y tiene tres propiedades: productEntity, que es una instancia de ProductDTO y representa un producto vendido; serviceEntity, que es una instancia de ServiceDTO y representa un servicio vendido; y type, que es una cadena de texto que indica el tipo de venta.

Luego, hay dos clases más: SaleCreateDTO y FinalSaleDTO, que se utilizan para crear una nueva venta y para representar la venta final respectivamente. Estas clases tienen propiedades similares a SaleDTO, pero la propiedad productEntity es opcional y puede ser una instancia de ProductCreateDTO en SaleCreateDTO, mientras que en FinalSaleDTO es una instancia de ProductDTO. La propiedad serviceEntity también cambia de una instancia de ServiceCreateDTO a una instancia de FinalServiceDTO.

Finalmente, la clase AllSaleDTO es una lista de instancias de SaleDTO y se utiliza para representar todas las ventas.

```

@Serializable
data class SaleDTO(
    val productEntity: ProductDTO?,
    val serviceEntity: ServiceDTO?,
    val type: String?
)

@Serializable
data class SaleCreateDTO(
    val productEntity: ProductCreateDTO?,
    val serviceEntity: ServiceCreateDTO?,
    val type: String?
)

@Serializable
data class FinalSaleDTO(
    val productEntity: ProductDTO?,
    val serviceEntity: FinalServiceDTO?,
    val type: String?
)

data class AllSaleDTO(
    val data: List<SaleDTO>
)

```

La clase ServiceDTO representa un servicio y tiene cinco propiedades: id, que es el identificador del servicio; uuid, que es un identificador único universal en formato de cadena; image, que es la imagen del servicio; price, que es el precio del servicio; appointment, que es el identificador de la cita asociada al servicio; y type, que es el tipo de servicio.

Luego, hay dos clases más: ServiceCreateDTO y FinalServiceDTO, que se utilizan para crear un nuevo servicio y para representar el servicio final respectivamente. Estas clases tienen propiedades similares a ServiceDTO, pero la propiedad id y uuid son opcionales en ServiceCreateDTO, mientras que en FinalServiceDTO son obligatorias. La propiedad appointment cambia de un identificador de cadena en ServiceDTO y ServiceCreateDTO a una instancia de AppointmentDTO en FinalServiceDTO.

```

@Serializable
data class ServiceDTO(
    val id: Long?,

```

```

    val uuid: String,
    val image: String,
    val price: Float,
    val appointment : String,
    val type : String
)

@Serializable
data class FinalServiceDTO(
    val id: Long?,
    val uuid: String,
    val image: String,
    val price: Float,
    val appointment : AppointmentDTO,
    val type : String
)

@Serializable
data class ServiceCreateDTO(
    val image: String,
    val price: Float,
    val appointment : String,
    val type : String
)

```

UserRegisterDTO: DTO utilizado para registrar un nuevo usuario. Contiene información como la imagen, el rol, el correo electrónico, el nombre de usuario, la contraseña y la dirección.

UserUpdateDTO: DTO utilizado para actualizar la información de un usuario existente. Tiene los mismos campos que UserRegisterDTO.

UserLoginDTO: DTO utilizado para iniciar sesión en la aplicación. Contiene el nombre de usuario y la contraseña.

UserResponseDTO: DTO utilizado para devolver información de usuario en respuestas de la API. Contiene el ID del usuario, el UUID (identificador único universal), la imagen, el rol, el correo electrónico, el nombre de usuario y la dirección.

UserTokenDTO: DTO utilizado para devolver un token de autenticación junto con la información de usuario. Contiene un objeto UserResponseDTO y un token.

UserDataDTO: DTO utilizado para devolver una lista de objetos UserResponseDTO.

```

@kotlinx.serialization.Serializable
data class UserRegisterDTO(
    val image: String?,
    val rol: Set<String>,
    val email: String,
    val username: String,
    val password: String,
    val address: String
)

@kotlinx.serialization.Serializable
data class UserUpdateDTO(
    val image: String?,
    val rol: Set<String>,
    val email: String,
    val username: String,
    val password: String,
    val address: String
)

@kotlinx.serialization.Serializable

```

```

data class UserLoginDTO(
    val username: String,
    val password: String
)

@kotlinx.serialization.Serializable
data class UserResponseDTO(
    val id: Long?,
    val uuid: String?,
    val image: String?,
    val rol: Set<String>?,
    val email: String?,
    val username: String?,
    val address: String?
)

@kotlinx.serialization.Serializable
data class UserTokenDTO(
    val user: UserResponseDTO?,
    val token: String?
)

data class UserDataDTO(
    val data: List<UserResponseDTO>?
)

```

5.1.2. Config

La clase tiene un constructor con un parámetro config anotado con @InjectedParam, que es un mapa de propiedades de configuración que se proporciona a través de la inyección de dependencias. La clase define varias propiedades de solo lectura que se inicializan con los valores correspondientes del mapa de configuración.

Por ejemplo, audience es una cadena que representa la audiencia del token (es decir, a quién está dirigido el token), secret es la clave secreta utilizada para firmar y verificar el token, issuer es la entidad que emite el token y realm es el ámbito de protección del token.

En resumen, la clase TokenConfig se utiliza para centralizar la configuración relacionada con los tokens de autenticación y autorización en una aplicación y proporcionar un acceso fácil a esta configuración en todo el código de la aplicación.

```

@Single
data class TokenConfig(
    @InjectedParam private val config: Map<String, String>
) {
    val audience = config["audience"].toString()
    val secret = config["secret"].toString()
    val issuer = config["issuer"].toString()
    val realm = config["realm"].toString()
}

```

5.1.3. Exceptions

Explicamos una de las clases de excepciones ya que el resto de clases de excepciones siguen la misma filosofía, al igual que en los próximos microservicios.

En este caso, la clase sellada se llama AppointmentException, lo que significa que esta clase puede tener subclases que se utilizan para representar excepciones específicas relacionadas con las citas. Las tres subclases definidas son AppointmentNotFoundException, AppointmentBadRequestException y AppointmentConflictIntegrityException.

Cada una de estas subclases hereda de AppointmentException y proporciona un mensaje de error personalizado a través de su constructor. Estas subclases se utilizan para representar diferentes tipos de excepciones que pueden surgir en el contexto de una aplicación que maneja

citas, como una cita no encontrada, una solicitud incorrecta de cita o una violación de integridad en la base de datos de citas.

```
sealed class AppointmentException(message: String): RuntimeException(message)

class AppointmentNotFoundException(message: String) : AppointmentException(message)

class AppointmentBadRequestException(message: String) :
AppointmentException(message)

class AppointmentConflictIntegrityException(message: String) :
AppointmentException(message)
```

5.1.4. Plugins

A continuación explicamos el plugin de seguridad:

Primero, se define un conjunto de parámetros para la configuración del token, los cuales se leen desde la configuración de la aplicación. Luego se crea una instancia de la clase TokenConfig, la cual encapsula los valores de configuración del token.

A continuación, se define una instancia del servicio TokensService, el cual es utilizado para verificar y generar tokens JWT.

Después, se configura el esquema de autenticación de la aplicación utilizando el proveedor de autenticación jwt(). Se establece el validador del token, el cual utiliza el servicio TokensService para verificar la autenticidad del token. También se define el "realm", que es el ámbito de protección para la autenticación.

```
fun Application.configureSecurity() {
    val tokenConfigParams = mapOf<String, String>(
        "audience" to environment.config.property("jwt.audience").getString(),
        "secret" to environment.config.property("jwt.secret").getString(),
        "issuer" to environment.config.property("jwt.issuer").getString(),
        "realm" to environment.config.property("jwt.realm").getString()
    )

    val tokenConfig: TokenConfig = get { parametersOf(tokenConfigParams) }

    val jwtService: TokensService by inject()

    authentication {
        jwt {
            verifier(jwtService.verifyJWT())
            realm = tokenConfig.realm
            validate { credential ->
                JWTPrincipal(credential.payload)
            }

            challenge { defaultScheme, realm ->
                call.respond(HttpStatusCode.Unauthorized, "Invalid token")
            }
        }
    }
}
```

Ahora explicamos el plugin de Socket:

Primero llamamos a la función configureSockets() para configurar el servidor WebSocket. En esta función, se instala el módulo WebSockets y se establecen algunos valores de configuración para el servidor WebSocket, como el período de ping, el tiempo de espera, el tamaño máximo del marco y si se utiliza máscara.

Luego definimos una ruta para el WebSocket con el path /ws. Dentro de esta ruta, se define el comportamiento del servidor WebSocket en función de los mensajes entrantes. En este caso, si el mensaje entrante es de tipo texto, el servidor envía una respuesta de tipo texto con el mismo mensaje recibido. Si el mensaje es "bye", el servidor cierra la conexión.

Después de definir el comportamiento del WebSocket, se damos dos funciones adicionales para iniciar el servidor y el cliente. La función Server.main inicia un servidor Echo que espera y acepta conexiones entrantes en el puerto 9002. La función Client.main inicia un cliente Echo que se conecta al servidor Echo y espera la entrada del usuario para enviar mensajes al servidor.

```
fun Application.configureSockets() {  
  
    install(WebSocketSockets) {  
        pingPeriod = Duration.ofSeconds(15)  
        timeout = Duration.ofSeconds(15)  
        maxFrameSize = Long.MAX_VALUE  
        masking = false  
    }  
    routing {  
        webSocket("/ws") { // websocketSession  
            for (frame in incoming) {  
                if (frame is Frame.Text) {  
                    val text = frame.readText()  
                    outgoing.send(Frame.Text("YOU SAID: $text"))  
                    if (text.equals("bye", ignoreCase = true)) {  
                        close(CloseReason(CloseReason.Codes.NORMAL, "Client said  
BYE"))  
                    }  
                }  
            }  
        }  
    }  
}  
  
object EchoApp {  
    val selectorManager = ActorSelectorManager(Dispatchers.IO)  
    val DefaultPort = 9002  
  
    object Server {  
        @JvmStatic  
        fun main(args: Array<String>) {  
            runBlocking {  
                val serverSocket = aSocket(selectorManager).tcp().bind(port =  
DefaultPort)  
                println("Echo Server listening at ${serverSocket.localAddress}")  
                while (true) {  
                    val socket = serverSocket.accept()  
                    println("Accepted $socket")  
                    launch {  
                        val read = socket.openReadChannel()  
                        val write = socket.openWriteChannel(autoFlush = true)  
                        try {  
                            while (true) {  
                                val line = read.readUTF8Line()  
                                write.writeStringUtf8("$line\n")  
                            }  
                        } catch (e: Throwable) {  
                            socket.close()  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```

        }
    }

object Client {
    @JvmStatic
    fun main(args: Array<String>) {
        runBlocking {
            val socket = aSocket(selectorManager).tcp().connect("127.0.0.1",
port = DefaultPort)
            val read = socket.openReadChannel()
            val write = socket.openWriteChannel(autoFlush = true)

            launch(Dispatchers.IO) {
                while (true) {
                    val line = read.readUTF8Line()
                    println("server: $line")
                }
            }

            for (line in System.`in`.lines()) {
                println("client: $line")
                write.writeStringUtf8("$line\n")
            }
        }
    }

    private fun InputStream.lines() = Scanner(this).lines()

    private fun Scanner.lines() = sequence {
        while (hasNext()) {
            yield(readLine())
        }
    }
}

object TlsRawSocket {
    @JvmStatic
    fun main(args: Array<String>) {
        runBlocking {
            val selectorManager = ActorSelectorManager(Dispatchers.IO)
            val socket = aSocket(selectorManager).tcp().connect("www.google.com",
port = 443)
                .tls(coroutineContext = coroutineContext)
            val write = socket.openWriteChannel()
            val EOL = "\r\n"
            write.writeStringUtf8("GET / HTTP/1.1${EOL}Host:
www.google.com${EOL}Connection: close${EOL}${EOL}")
            write.flush()

            println(socket.openReadChannel().readRemaining().readBytes().toString(Charsets.UTF_8))
        }
    }
}

```

5.1.5. Repositories

Definimos una interfaz llamada `IAppointmentRepository` que establece un conjunto de métodos que deben ser implementados por cualquier clase que desee actuar como un repositorio para las citas (`Appointment`).

Los métodos definidos en esta interfaz son los siguientes:

`findAll(token: String): Flow<AppointmentDTO>`: Devuelve un Flow de objetos AppointmentDTO, que representan todas las citas en el repositorio.

`findById(token: String, id: UUID): AppointmentDTO`: Devuelve un objeto AppointmentDTO que representa la cita con el ID especificado.

`save(token: String, entity: AppointmentCreateDTO): AppointmentDTO`: Guarda una nueva cita en el repositorio y devuelve un objeto AppointmentDTO que representa la cita guardada.

`update(token: String, id: UUID, entity: AppointmentCreateDTO): AppointmentDTO`: Actualiza una cita existente con el ID especificado en el repositorio y devuelve un objeto AppointmentDTO que representa la cita actualizada.

`delete(token: String, id: UUID)`: Elimina la cita con el ID especificado del repositorio.

Cada uno de estos métodos recibe un parámetro token de tipo String, que se utiliza para autenticar la solicitud y asegurar que el usuario tiene los permisos necesarios para realizar la operación correspondiente.

```
interface IAppointmentRepository {
    suspend fun findAll(token: String): Flow<AppointmentDTO>
    suspend fun findById(token: String, id: UUID): AppointmentDTO
    suspend fun save(token: String, entity: AppointmentCreateDTO): AppointmentDTO
    suspend fun update(token: String, id: UUID, entity: AppointmentCreateDTO): AppointmentDTO
    suspend fun delete(token: String, id: UUID)
}
```

Aquí podemos ver la implementación de la interfaz IAppointmentRepository, que define los métodos que se deben implementar para acceder a los datos de las citas de servicio de una aplicación.

La clase KtorFitRepositoryAppointment es una implementación de IAppointmentRepository. Vemos que implementamos la anotación @Single. Esta anotación es una forma de indicar que esta clase es un objeto singleton, lo que significa que solo habrá una instancia de esta clase en toda la aplicación. Esta anotación se utiliza comúnmente en frameworks de inyección de dependencias.

Definimos que esta clase se identificará por el nombre "KtorFitRepositoryAppointment" al inyectarla en otras clases. Esto se hace mediante la anotación @Named.

Cada método implementado utiliza la biblioteca KtorFit para hacer una llamada a una API que proporciona datos sobre las citas de servicios. Por ejemplo, en la implementación del método `findAll()`, se hace una llamada asíncrona a la API con el token de autenticación del usuario, y se devuelve un flujo de objetos AppointmentDTO. En caso de que ocurra una excepción durante la llamada, se lanza una excepción personalizada que indica el problema.

Cada método también se ejecuta dentro del contexto de un hilo específico utilizando la función `withContext(Dispatchers.IO)`, lo que indica que se ejecutará en un hilo de entrada-salida. Esto se hace para asegurarse de que la aplicación no se bloquee mientras espera una respuesta de la API.

```
@Single
@Named("KtorFitRepositoryAppointment")
class KtorFitRepositoryAppointment : IAppointmentRepository {
    private val client by lazy { KtorFitClientSales.instance }

    override suspend fun findAll(token: String): Flow<AppointmentDTO> =
        withContext(Dispatchers.IO) {
```

```

    val call = async {
        client.getAllAppointments(token).asFlow()
    }

    try {
        return@withContext call.await()
    } catch (e: Exception) {
        throw AppointmentNotFoundException("Error getting appointments:
${e.message}")
    }
}

override suspend fun delete(token: String, id: UUID) =
withContext(Dispatchers.IO) {
    val call = async {
        client.deleteAppointment(token, id.toString())
    }

    try {
        return@withContext call.await()
    } catch (e: Exception) {
        throw AppointmentNotFoundException("Error deleting appointent with id
$id : ${e.message}")
    }
}

override suspend fun update(token: String, id: UUID, entity:
AppointmentCreateDTO): AppointmentDTO = withContext(Dispatchers.IO){
    val call = async {
        client.updateAppointment(token, id.toString(), entity)
    }

    try {
        return@withContext call.await()
    } catch (e: Exception) {
        throw AppointmentNotFoundException("Error updating appointment with id
$id : ${e.message}")
    }
}

override suspend fun save(token: String, entity: AppointmentCreateDTO):
AppointmentDTO = withContext(Dispatchers.IO) {
    val call = async {
        client.createAppointments(token, entity)
    }
    try {
        return@withContext call.await()
    } catch (e: Exception) {
        throw AppointmentConflictIntegrityException("Error saving appointment
$entity : ${e.message}")
    }
}

override suspend fun findById(token: String, id: UUID): AppointmentDTO =
withContext(Dispatchers.IO) {
    val call = async {
        client.getAppointmentById(token, id.toString())
    }
    try {
        return@withContext call.await()
    } catch (e: Exception) {
        throw AppointmentNotFoundException("Error getting appointment with id
$id : ${e.message}")
    }
}

```

```
    }  
}
```

5.1.6. Routes

La función Application.ordersRoutes() define las rutas para los endpoints relacionados con los pedidos. En primer lugar, se definen tres repositorios que se utilizarán para interactuar con la base de datos: orderRepository, orderLineRepository y userRepository. Estos repositorios se obtienen mediante la inyección de dependencias (inject) y se identifican mediante un nombre específico (named) que se pasa como argumento.

La ruta principal para los endpoints de pedidos es route("/\$ENDPOINT"), donde \$ENDPOINT es un valor que se define en otra parte de la aplicación. Dentro de esta ruta, se utiliza la función authenticate para requerir autenticación para todas las solicitudes de esta ruta.

Dentro de la ruta, se definen varias solicitudes HTTP (get, post, put y delete) que corresponden a las operaciones CRUD básicas (Crear, Leer, Actualizar, Eliminar) para los pedidos.

Para cada solicitud, se manejan los posibles errores que puedan surgir al realizar la operación correspondiente. En algunos casos, se utilizan las funciones async y await para realizar varias operaciones en paralelo y esperar a que todas las operaciones hayan finalizado antes de continuar.

```
fun Application.ordersRoutes() {  
    val orderRepository by  
inject<KtorFitRepositoryOrders>(named("KtorFitRepositoryOrders"))  
    val orderLineRepository by  
inject<KtorFitRepositoryOrdersLine>(named("KtorFitRepositoryOrdersLine"))  
    val userRepopsitory by  
inject<KtorFitRepositoryUsers>(named("KtorFitRepositoryUsers"))  
    val tokenService by inject<TokensService>()  
  
    routing {  
        route("/$ENDPOINT") {  
            authenticate {  
                // ORDERS  
                get {  
                    try {  
                        val originalToken = call.principal<JWTPrincipal>()!!  
                        val token = tokenService.generateToken(originalToken)  
  
//if(originalToken.payload.getClaim("rol").split(",").toSet().toString())  
  
                        val orderDTO = async {  
                            orderRepository.findAll("Bearer $token").toList()  
                        }  
  
                        call.respond(HttpStatusCode.OK, orderDTO.await())  
                    } catch (e: OrderNotFoundException) {  
                        call.respond(HttpStatusCode.NotFound, e.message.toString())  
                    }  
                }  
  
                get("/{id}") {  
                    try {  
                        val token = tokenService.generateToken(call.principal()!!)  
                        val id = UUID.fromString(call.parameters["id"])!  
  
                        val orderDTO = async {  
                            orderRepository.findById("Bearer $token", id)  
                        }.await()  
  
                        val res = mutableListOf<OrderLineDTO>()  
                    }  
                }  
            }  
        }  
    }  
}
```

```

        orderDTO.orderLine.forEach {
            res.add(
                async {
                    orderLineRepository.findById("Bearer $token",
                    UUID.fromString(it))
                }.await()
            )
        }
        val cliente = userRepopsitory.findById("Bearer $token",
        orderDTO.cliente.toLong())

        val finalOrder = FinalOrderDTO(
            orderDTO.uuid,
            orderDTO.status,
            orderDTO.total,
            orderDTO.iva,
            res,
            cliente
        )
        call.respond(HttpStatusCode.OK, finalOrder)
    } catch (e: OrderNotFoundException) {
        call.respond(HttpStatusCode.NotFound, e.message.toString())
    }
}

post {
    try {
        val originalToken = call.principal<JWTPrincipal>()!!
        val token = tokenService.generateToken(originalToken)

        if
(originalToken.payload.getClaim("rol").toString().contains("[ADMIN]") ||
originalToken.payload.getClaim("rol").toString().contains("SUPERADMIN"))
        ) {
            val orderCreate = call.receive<OrderDTOCreate>()
            val res = mutableListOf<String>()

            orderCreate.orderLine.forEach {
                res.add(
                    async {
                        orderLineRepository.save("Bearer $token",
                        it).uuid
                    }.await() !!
                )
            }

            val saveOrder = OrderSaveDTO(
                orderCreate.status,
                orderCreate.total,
                orderCreate.iva,
                res,
                orderCreate.cliente
            )

            val resOrder = async {
                orderRepository.save("Bearer $token", saveOrder)
            }.await()

            call.respond(HttpStatusCode.Created, resOrder)
        } else {
            call.respond(HttpStatusCode.Unauthorized, "You are not")
        }
    }
}

```

```

authorized")
    }

} catch (e: OrderBadRequestException) {
    call.respond(HttpStatusCode.BadRequest,
e.message.toString())
}
}

put("/{id}") {
    try {
        val originalToken = call.principal<JWTPrincipal>()!!
        val token = tokenService.generateToken(originalToken)

        if
(originalToken.payload.getClaim("rol").toString().contains("[ADMIN]") ||
originalToken.payload.getClaim("rol").toString().contains("SUPERADMIN"))
        ) {
            val id = UUID.fromString(call.parameters["id"]!!)
            val orderUpdate = call.receive<OrderDTOUpdate>()
            val res = mutableListOf<String>()

            orderUpdate.orderLine.forEach {
                res.add(
                    async {
                        orderLineRepository.findById("Bearer
$token", UUID.fromString(it)).uuid
                    }.await()!!
                )
            }
        }

        val saveOrder = OrderDTOUpdate(
            orderUpdate.status,
            orderUpdate.total,
            orderUpdate.iva,
            res,
            orderUpdate.cliente
        )

        val resOrder = async {
            orderRepository.update("Bearer $token", id,
saveOrder)
        }.await()

        call.respond(HttpStatusCode.OK, resOrder)
    } else {
        call.respond(HttpStatusCode.Unauthorized, "You are not
authorized")
    }
}

} catch (e: OrderNotFoundException) {
    call.respond(HttpStatusCode.NotFound, e.message.toString())
} catch (e: OrderBadRequestException) {
    call.respond(HttpStatusCode.BadRequest,
e.message.toString())
}
}

delete("/{id}") {
    try {
        val originalToken = call.principal<JWTPrincipal>()!!
        val token = tokenService.generateToken(originalToken)

        if

```

```
(originalToken.payload.getClaim("rol").toString().contains("SUPERADMIN")) {
    val id = UUID.fromString(call.parameters["id"])!!

    val order = async {
        orderRepository.findById("Bearer $token", id)
    }.await()

    order.orderLine.forEach {
        async {
            orderLineRepository.delete("Bearer $token",
                UUID.fromString(it))
        }.await()
    }

    async {
        orderRepository.delete("Bearer $token",
            UUID.fromString(order.uuid))
    }.await()

    call.respond(HttpStatusCode.NoContent)
} else {
    call.respond(HttpStatusCode.Unauthorized, "You are not
authorized")
}
} catch (e: OrderNotFoundException) {
    call.respond(HttpStatusCode.NotFound, e.message.toString())
}
}
}
}
```

La función `salesRoutes()` define rutas HTTP relacionadas con ventas de productos o servicios. Esta función se llama en algún lugar de la aplicación para registrar estas rutas.

La constante `ENDPOINT` se utiliza para definir la ruta principal del recurso de venta de la aplicación.

Después de inyectar algunos servicios necesarios, la función define tres rutas:

GET /sales: Esta ruta responde con una lista de ventas. La función busca todas las ventas en el repositorio y las convierte en una lista de objetos FinalSaleDTO. Si una venta es de tipo "SERVICE", se crea un objeto FinalServiceDTO que contiene información adicional sobre el servicio vendido. Si es de tipo "PRODUCT", solo se incluye información básica del producto vendido. Finalmente, la lista se envía en la respuesta HTTP.

GET /sales/product/{id}: Esta ruta responde con la información detallada de un producto vendido en función de su ID.

GET /sales/service/{id}: Esta ruta responde con la información detallada de un servicio vendido en función de su ID.

POST /sales: Esta ruta crea una nueva venta. Se verifica si el usuario que realiza la solicitud tiene permisos de administrador o superadministrador antes de crear la venta. La solicitud debe incluir información sobre el producto o servicio que se está vendiendo. La respuesta HTTP incluye información detallada sobre la venta creada.

PUT /sales/{id}: Esta ruta actualiza una venta existente en función de su ID. De nuevo, se verifica si el usuario que realiza la solicitud tiene permisos de administrador o superadministrador antes de actualizar la venta. La solicitud debe incluir información actualizada sobre el producto o servicio que se está vendiendo. La respuesta HTTP incluye información detallada sobre la venta actualizada.

```
private const val ENDPOINT = "sales"

fun Application.salesRoutes() {
    val salesRepository by
inject<KtorFitRepositorySales>(named("KtorFitRepositorySales"))
    val appointmentRepository by
inject<KtorFitRepositoryAppointment>(named("KtorFitRepositoryAppointment"))
    val tokenService by inject<TokensService>()

routing {
    route("/{$ENDPOINT}") {
        authenticate {
            get {
                try {
                    val originalToken = call.principal<JWTPrincipal>()!!
                    val token = tokenService.generateToken(originalToken)

                    val list = async {
                        salesRepository.findAll("Bearer $token").toList()
                    }

                    val result: MutableList<FinalSaleDTO> = mutableListOf()
                    list.await().forEach {
                        if (it.type == "SERVICE") {
                            val service = FinalSaleDTO(
                                productEntity = null,
                                serviceEntity = FinalServiceDTO(
                                    id = it.serviceEntity!!.id,
                                    uuid = it.serviceEntity.uuid,
                                    image = it.serviceEntity.image,
                                    price = it.serviceEntity.price,
                                    appointment = async {
                                        appointmentRepository.findById(
                                            "Bearer $token",
                                            UUID.fromString(it.serviceEntity.appointment)
                                        )
                                    }.await(),
                                    type = it.serviceEntity.type
                                ),
                                type = "SERVICE"
                            )
                            result.add(service)
                        } else {
                            val product = FinalSaleDTO(
                                productEntity = it.productEntity,
                                serviceEntity = null,
                                type = "PRODUCT"
                            )
                            result.add(product)
                        }
                    }
                    call.respond(HttpStatusCode.OK, result)
                } catch (e: SaleNotFoundException) {
                    call.respond(HttpStatusCode.NotFound, e.message.toString())
                }
            }
        }
    }
}
```

```

        }

    }

    get("/product/{id}") {
        try {
            val token = tokenService.generateToken(call.principal()!!)
            val id = call.parameters["id"]!!

            val list = async {
                salesRepository.findById("Bearer $token", id)
            }

            list.await().forEach {
                if (it.type == "PRODUCT") {
                    call.respond(HttpStatusCode.OK, it.productEntity!!)
                }
            }

            call.respond(HttpStatusCode.NotFound, "Product not found
with uuid: $id")
        } catch (e: SaleNotFoundException) {
            call.respond(HttpStatusCode.NotFound, e.message.toString())
        }
    }

    get("/service/{id}") {
        try {
            val token = tokenService.generateToken(call.principal()!!)
            val id = call.parameters["id"]!!

            val list = async {
                salesRepository.findById("Bearer $token", id)
            }

            list.await().forEach {
                if (it.type == "SERVICE") {
                    val result = FinalServiceDTO(
                        id = it.serviceEntity!.id,
                        uuid = it.serviceEntity.uuid,
                        image = it.serviceEntity.image,
                        price = it.serviceEntity.price,
                        appointment = async {
                            appointmentRepository.findById(
                                "Bearer $token",
                                UUID.fromString(it.serviceEntity.appointment)
                            )
                        }.await(),
                        type = it.serviceEntity.type
                    )

                    call.respond(HttpStatusCode.OK, result)
                }
            }

            call.respond(HttpStatusCode.NotFound, "Service not found
with uuid: $id")
        } catch (e: SaleNotFoundException) {
            call.respond(HttpStatusCode.NotFound, e.message.toString())
        }
    }
}

```

```

post {
    try {
        val originalToken = call.principal<JWTPrincipal>()!!
        val token = tokenService.generateToken(originalToken)

        if
(originalToken.payload.getClaim("rol").toString().contains("[ADMIN]") ||
originalToken.payload.getClaim("rol").toString().contains("SUPERADMIN"))
        ) {

            val dto = call.receive<SaleCreateDTO>()

            val result = async {
                salesRepository.save("Bearer $token", dto)
            }

            val res = result.await()

            if (res.type == "SERVICE") {
                val res = FinalServiceDTO(
                    id = res.serviceEntity!!.id,
                    uuid = res.serviceEntity.uuid,
                    image = res.serviceEntity.image,
                    price = res.serviceEntity.price,
                    appointment = async {
                        appointmentRepository.findById(
                            "Bearer $token",
                            UUID.fromString(res.serviceEntity.appointment)
                                )
                    }.await(),
                    type = res.serviceEntity.type
                )

                call.respond(HttpStatusCode.Created, res)
            } else {
                call.respond(HttpStatusCode.Created, res)
            }
        } else {
            call.respond(HttpStatusCode.Unauthorized, "You are not
authorized")
        }
    }

    } catch (e: SaleNotFoundException) {
        call.respond(HttpStatusCode.NotFound, e.message.toString())
    }
}

put("/{id}") {
    try {
        val originalToken = call.principal<JWTPrincipal>()!!
        val token = tokenService.generateToken(originalToken)

        if
(originalToken.payload.getClaim("rol").toString().contains("[ADMIN]") ||
originalToken.payload.getClaim("rol").toString().contains("SUPERADMIN"))
        ) {

            val id = call.parameters["id"]
            val dto = call.receive<SaleCreateDTO>()

```

```

        val result = async {
            salesRepository.update("Bearer $token",
UUID.fromString(id), dto)
        }

        val res = result.await()

        if (res.type == "SERVICE") {
            val res = FinalServiceDTO(
                id = res.serviceEntity!!.id,
                uuid = res.serviceEntity.uuid,
                image = res.serviceEntity.image,
                price = res.serviceEntity.price,
                appointment = async {
                    appointmentRepository.findById(
                        "Bearer $token",
                        UUID.fromString(res.serviceEntity.appointment)
                    )
                }.await(),
                type = res.serviceEntity.type
            )
            call.respond(HttpStatusCode.OK, res)
        } else {
            call.respond(HttpStatusCode.OK, res)
        }
    } else {
        call.respond(HttpStatusCode.Unauthorized, "You are not
authorized")
    }
}

} catch (e: SaleNotFoundException) {
    call.respond(HttpStatusCode.NotFound, e.message.toString())
}

}

delete("/{id}") {
    try {
        val originalToken = call.principal<JWTPrincipal>()!!
        val token = tokenService.generateToken(originalToken)

        if
(originalToken.payload.getClaim("rol").toString().contains("[ADMIN]")) {
            val id = call.parameters["id"]

            val result = async {
                salesRepository.delete("Bearer $token",
UUID.fromString(id))
            }

            call.respond(HttpStatusCode.NoContent)
        } else {
            call.respond(HttpStatusCode.Unauthorized, "You are not
authorized")
        }
    }

    } catch (e: SaleNotFoundException) {
        call.respond(HttpStatusCode.NotFound, e.message.toString())
    }
}
}

```

```
        }
    }
}
```

Primero, definimos una constante llamada "ENDPOINT" con valor "users" que utilizamos en las rutas para formar la URL base. Luego, la inyectamos a un objeto de la clase "KtorFitRepositoryUsers" y otro de la clase "TokensService" utilizando la función "inject" de Koin, un contenedor de inyección de dependencias.

Después utilizamos la función "routing" de Ktor para definir las rutas y sus respectivas operaciones HTTP. Para cada ruta, se define una función lambda que contiene la lógica de la operación correspondiente.

La ruta "/login" permite iniciar sesión enviando un objeto JSON con el correo electrónico y contraseña del usuario. Se llama al método "login" del objeto "userRepository" utilizando la información proporcionada y se devuelve la respuesta al cliente.

La ruta "/register" permite registrar un nuevo usuario enviando un objeto JSON con la información del usuario. Se llama al método "register" del objeto "userRepository" utilizando la información proporcionada y se devuelve la respuesta al cliente.

Las rutas que empiezan con "authenticate" (por ejemplo, la ruta "/users") requieren autenticación mediante un token de acceso JWT. Si el token es válido, se llama a los métodos correspondientes de "userRepository" para obtener información de usuarios, actualizar información o eliminar usuarios, y se devuelve la respuesta al cliente.

```
private const val ENDPOINT = "users"

fun Application.usersRoutes() {
    val userRepository by
    inject<KtorFitRepositoryUsers>(named("KtorFitRepositoryUsers"))
    //val userRepository = KtorFitRepositoryUsers()
    val tokenService by inject<TokensService>()

    routing {
        route("/{$ENDPOINT}") {
            post("/login") {
                logger.debug { "API Gateway -> /login" }

                try {
                    val login = call.receive<UserLoginDTO>()

                    val user = async {
                        userRepository.login(login)
                    }

                    call.respond(HttpStatusCode.OK, user.await())
                } catch (e: UserBadRequestException) {
                    call.respond(HttpStatusCode.BadRequest, e.message.toString())
                }
            }

            post("/register") {
                logger.debug { "API Gateway -> /register" }

                try {
                    val register = call.receive<UserRegisterDTO>()

                    val user = async {
                        userRepository.register(register)
                    }
                }
            }
        }
    }
}
```

```

        call.respond(HttpStatusCode.Created, user.await())

    } catch (e: UserBadRequestException) {
        call.respond(HttpStatusCode.BadRequest, e.message.toString())
    }
}

authenticate {
    get {
        logger.debug { "API Gateway -> /users" }
        try {
            val token = tokenService.generateToken(call.principal()!!)
            val res = async {
                userRepository.findAll("Bearer $token")
            }
            val users = res.await()

            call.respond(HttpStatusCode.OK, users)

        } catch (e: UserNotFoundException) {
            call.respond(HttpStatusCode.NotFound, e.message.toString())
        }
    }

    get("/{id}") {
        logger.debug { "API Gateway -> /users/id" }

        try {
            val token = tokenService.generateToken(call.principal()!!)
            val id = call.parameters["id"]

            val user = async {
                userRepository.findById("Bearer $token", id!!.toLong())
            }

            call.respond(HttpStatusCode.OK, user.await())

        } catch (e: UserNotFoundException) {
            call.respond(HttpStatusCode.NotFound, e.message.toString())
        }
    }

    put("/{id}") {
        logger.debug { "API Gateway -> /users/id" }

        try {
            val token = tokenService.generateToken(call.principal()!!)
            val id = call.parameters["id"]
            val user = call.receive<UserUpdateDTO>()

            val updatedUser = async {
                userRepository.update("Bearer $token", id!!.toLong(),
user)
            }

            call.respond(HttpStatusCode.OK, updatedUser.await())

        } catch (e: UserNotFoundException) {
            call.respond(HttpStatusCode.NotFound, e.message.toString())
        } catch (e: UserBadRequestException) {
            call.respond(HttpStatusCode.BadRequest,
e.message.toString())
        }
    }
}

```

```

        }

        delete("/{id}") {
            logger.debug { "API Gateway -> /users/{id}" }

            try {
                val token = tokenService.generateToken(call.principal()!!)
                val id = call.parameters["id"]

                userRepository.delete("Bearer $token", id!!.toLong())

                call.respond(HttpStatusCode.NoContent)
            } catch (e: UserNotFoundException) {
                call.respond(HttpStatusCode.NotFound, e.message.toString())
            }
        }
    }
}

```

5.1.7. Services

A continuación explicamos como implementamos cada uno de los servicios que utilizamos, como en general todos funcionan de la misma forma, vamos a explicar por ejemplo el de usuarios.

Primero, se define una constante llamada "ENDPOINT" con valor "users" que se utilizará en las rutas para formar la URL base. Luego, se inyecta un objeto de la clase "KtorFitRepositoryUsers" y otro de la clase "TokensService" utilizando la función "inject" de Koin, un contenedor de inyección de dependencias.

Después, se utiliza la función "routing" de Ktor para definir las rutas y sus respectivas operaciones HTTP. Para cada ruta, se define una función lambda que contiene la lógica de la operación correspondiente.

La ruta "/login" permite iniciar sesión enviando un objeto JSON con el correo electrónico y contraseña del usuario. Se llama al método "login" del objeto "userRepository" utilizando la información proporcionada y se devuelve la respuesta al cliente.

La ruta "/register" permite registrar un nuevo usuario enviando un objeto JSON con la información del usuario. Se llama al método "register" del objeto "userRepository" utilizando la información proporcionada y se devuelve la respuesta al cliente.

Las rutas que empiezan con "authenticate" (por ejemplo, la ruta "/users") requieren autenticación mediante un token de acceso JWT. Si el token es válido, se llama a los métodos correspondientes de "userRepository" para obtener información de usuarios, actualizar información o eliminar usuarios, y se devuelve la respuesta al cliente.

```

interface KtorFitRestUsers {
    @GET("users")
    suspend fun findAll(
        @Header("Authorization") token: String
    ): UserDataDTO

    @GET("users/{id}")
    suspend fun findById(
        @Header("Authorization") token: String,
        @Path("id") id: Long
    ): UserResponseDTO

    @POST("users/login")
}

```

```

        suspend fun login(
            @Body user: UserLoginDTO
        ): UserTokenDTO

        @POST("users/register")
        suspend fun register(
            @Body user: UserRegisterDTO
        ): UserTokenDTO

        @PUT("users/{id}")
        suspend fun update(
            @Header("Authorization") token: String,
            @Path("id") id: Long, @Body user: UserUpdateDTO
        ): UserResponseDTO

        @DELETE("users/{id}")
        suspend fun delete(
            @Header("Authorization") token: String,
            @Path("id") id: Long
        )
    }
}

```

KtorFitClientUsers tiene dos propiedades:

API_URL: una constante que especifica la URL base del servicio de la API de usuarios.

instance: una propiedad de solo lectura que se inicializa de forma perezosa (lazy) y devuelve una instancia de la interfaz KtorFitRestUsers que se crea mediante la biblioteca Ktorfit. La instancia de la interfaz se crea utilizando el método create de Ktorfit, que utiliza la URL base y la definición de la interfaz para crear un cliente HTTP que puede interactuar con la API.

La biblioteca Ktorfit proporciona una forma sencilla de definir una interfaz que define la API y de crear un cliente HTTP que pueda interactuar con la API. En este caso, KtorFitRestUsers es una interfaz que define los métodos que se utilizan para interactuar con la API de usuarios. Luego, KtorFitClientUsers utiliza Ktorfit para crear un cliente HTTP que implementa la interfaz y que se utiliza para interactuar con la API de usuarios.

```

object KtorFitClientUsers {
    private const val API_URL = "http://localhost:8383/"

    private val ktorfit by lazy {
        Ktorfit.Builder()
            .httpClient {
                install(ContentNegotiation) {
                    gson()
                }
                install(DefaultRequest) {
                    header(HttpHeaders.ContentType, ContentType.Application.Json)
                }
            }
            .baseUrl(API_URL)
            .build()
    }

    val instance by lazy {
        ktorfit.create<KtorFitRestUsers>()
    }
}

```

El constructor de la clase acepta un objeto TokenConfig que contiene la configuración necesaria para generar y verificar tokens JWT. Los métodos son los siguientes:

verifyJWT(): Este método devuelve un objeto JWTVerifier que se utiliza para verificar la validez de un token JWT. Se crea utilizando la librería com.auth0:java-jwt.

`generateToken(token: JWTPrincipal)`: Este método genera un nuevo token JWT basado en la información proporcionada por el objeto `JWTPrincipal` que se le pasa como argumento. El token generado tiene un tiempo de expiración, issuer, subject y los claims "username" y "rol". Se utiliza también la librería `com.auth0:java-jwt` para generar el token, y se utiliza la clave secreta definida en el método `sign()` para firmar el token.

La anotación `@Single` es una anotación personalizada que indica que esta clase es una clase singleton, es decir, solo existe una instancia de esta clase en la aplicación.

```
@Single
class TokensService {
    private val tokenConfig: TokenConfig
) {
    fun verifyJWT(): JWTVerifier {
        return JWT.require(Algorithm.HMAC512(tokenConfig.secret))
            //.withAudience(tokenConfig.audience)
            .withIssuer(tokenConfig.issuer)
            .build()
    }

    fun generateToken(token: JWTPrincipal): String{
        return JWT.create()
            .withIssuer(token.payload.issuer)
            .withSubject(token.payload.subject)
            .withClaim("username",
token.payload.getClaim("username").toString().replace("\"", ""))
            .withClaim("rol",
token.payload.getClaim("rol").toString().replace("\"", ""))
            .withExpiresAt(token.payload.expiresAt)
            .sign(Algorithm.HMAC512("BiquesDAM"))
    }
}
```

5.1.8. Tests

Esta es una clase de pruebas unitarias (JUnit) para probar las rutas de la aplicación relacionadas con la entidad "User". La anotación `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` indica que se debe utilizar una instancia de la clase de prueba para todos los métodos de prueba. La anotación `@TestMethodOrder(MethodOrderer.OrderAnnotation::class)` indica que los métodos de prueba se ejecutarán en un orden específico, determinado por la anotación `@Order`.

La clase contiene los siguientes campos:

`config`: una instancia de la clase `ApplicationConfig` que se utiliza para configurar la aplicación en el contexto de pruebas.

`responseDTO`: una instancia de la clase `UserResponseDTO` que contiene datos de usuario simulados para usar en las pruebas.

`updateDTO`: una instancia de la clase `UserUpdateDTO` que contiene datos de usuario simulados para usar en las pruebas.

`registerDTO`: una instancia de la clase `UserRegisterDTO` que contiene datos de usuario simulados para usar en las pruebas.

`loginDTO`: una instancia de la clase `UserLoginDTO` que contiene datos de inicio de sesión simulados para usar en las pruebas.

La clase también contiene varios métodos de prueba que se ejecutan en el orden especificado por la anotación `@Order`. Cada método de prueba utiliza el método `testApplication` para crear y configurar una instancia de la aplicación en el contexto de pruebas, y luego realiza una solicitud

HTTP simulada al servidor de pruebas utilizando el cliente HTTP incluido en la biblioteca de pruebas. Cada método de prueba utiliza los datos de usuario simulados para realizar operaciones CRUD (crear, leer, actualizar y eliminar) en la base de datos simulada de la aplicación y realiza aserciones sobre los resultados esperados de estas operaciones.

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@TestMethodOrder(MethodOrderer.OrderAnnotation::class)
class UserRoutesTest {
    private val config = ApplicationConfig("application.conf")

    val responseDTO = UserResponseDTO(
        id = 1,
        uuid = "41152394-fb3f-44ee-bb6c-be4fc4af8bf1",
        image = "Test",
        email = "Test",
        username = "Test",
        rol = setOf("ADMIN"),
        address = "address"
    )

    val updateDTO = UserUpdateDTO(
        image = "Test",
        email = "Test",
        username = "Test",
        password = "Test",
        rol = setOf("ADMIN"),
        address = "address"
    )

    val registerDTO = UserRegisterDTO(
        image = "Test",
        email = "Test",
        username = "Test",
        password = "Test",
        rol = setOf("ADMIN"),
        address = "address"
    )

    val loginDTO = UserLoginDTO(
        username = "admin",
        password = "admin1234"
    )

    @Test
    @Order(1)
    fun register() = testApplication {
        environment { config }

        val client = createClient {
            install(ContentNegotiation) {
                json()
            }
        }

        val response = client.post("/users/register") {
            contentType(ContentType.Application.Json)
            setBody(registerDTO)
        }

        assertEquals(response.status, HttpStatusCode.Created)
    }

    @Test
    @Order(2)
```

```

fun login() = testApplication {
    environment { config }

    val client = createClient {
        install(ContentNegotiation) {
            json()
        }
    }

    val response = client.post("/users/login") {
        contentType(ContentType.Application.Json)
        setBody(loginDTO)
    }

    assertEquals(response.status, HttpStatusCode.OK)
}

@OptIn(InternalAPI::class)
@Test
@Order(3)
fun findAll() = testApplication {
    environment { config }

    val client = createClient {
        install(ContentNegotiation) {
            json()
        }
    }

    val login = client.post("/users/login") {
        contentType(ContentType.Application.Json)
        setBody(loginDTO)
    }.content.readRemaining().readBytes().decodeToString()

    KotlinLogging.logger{}.error { login }

    val response = client.get("/users")
    assertEquals(HttpStatusCode.OK, response.status)
}

@Test
@Order(4)
fun findById() = testApplication {
    environment { config }

    val client = createClient {
        install(ContentNegotiation) {
            json()
        }
    }

    val response = client.get("/users/1")
    assertEquals(response.status, HttpStatusCode.OK)
}

@Test
@Order(5)
fun update() = testApplication {
    environment { config }

    val client = createClient {
        install(ContentNegotiation) {

```

```

        json()
    }

    val response = client.put("/users/1") {
        contentType(ContentType.Application.Json)
        setBody(updateDTO)
    }

    assertEquals(response.status, HttpStatusCode.OK)
}

@Test
@Order(6)
fun delete() = testApplication {
    environment { config }

    val client = createClient {
        install(ContentNegotiation) {
            json()
        }
    }

    val response = client.delete("/users/1")

    assertEquals(response.status, HttpStatusCode.NoContent)
}
}

```

5.2. Microservicio B ()

5.2.1. Modelos

Esta clase representa a un usuario de la API, utilizamos las anotaciones de JPA para mapear objetos a tablas en una base de datos relacional.

La anotación `@Table(name = "USERS")` especifica que esta clase representa a la tabla USERS en la base de datos.

La anotación `@Serializable` indica que la clase es serializable, lo que significa que se puede convertir en una secuencia de bytes y almacenarla en algún lugar (como en la memoria o en un archivo).

La clase tiene varias propiedades, como `id`, `uuid`, `image`, `rol`, `email`, `username`, `password` y `address`, que representan diferentes atributos del usuario, como su identificación, correo electrónico, contraseña y dirección. Las propiedades tienen diferentes tipos, como `Long`, `String` y `LocalDateTime`.

La anotación `@Id` indica que la propiedad `id` es la clave primaria de la tabla.

La anotación `@Column("rol")` indica que la propiedad `rol` se mapea a la columna "rol" en la tabla.

La clase también tiene tres propiedades de fecha (`createdAt`, `updatedAt` y `lastPasswordChangeAt`) que se utilizan para mantener un registro de cuándo se creó, actualizó y cambió por última vez la contraseña del usuario.

La clase implementa la interfaz `UserDetails`, que es una interfaz de Spring Security que se utiliza para representar los detalles de un usuario que se utilizan para autenticarlo y autorizarlo en una aplicación. La clase implementa los métodos necesarios de la interfaz, como `getAuthorities()`, `getPassword()`, `getUsername()`, `isAccountNonExpired()`, `isAccountNonLocked()`, `isCredentialsNonExpired()` y `isEnabled()`.

Por último, la clase tiene un enum llamado TipoUsuario que enumera los diferentes roles que un usuario puede tener en la aplicación, como "SUPERADMIN", "ADMIN" y "CLIENT".

```
@Table(name = "USERS")
@Serializable
data class User(
    @Id
    val id: Long? = null,
    // @Serializable(with = UUIDSerializer::class)
    val uuid: String = UUID.randomUUID().toString(),
    val image: String? = null,
    @Column("rol")
    val rol: String = User.TipoUsuario.CLIENT.name,
    val email: String,
    @get:JvmName("userName")
    val username: String,
    @get:JvmName("userPassword")
    val password: String,
    val address: String,

    // Históricos.
    @Column("created_at")
    @Serializable(with = LocalDateTimeSerializer::class)
    val createdAt: LocalDateTime = LocalDateTime.now(),
    @Column("updated_at")
    @Serializable(with = LocalDateTimeSerializer::class)
    val updatedAt: LocalDateTime = LocalDateTime.now(),
    @Column("last_password_changed_at")
    @Serializable(with = LocalDateTimeSerializer::class)
    val lastPasswordChangeAt: LocalDateTime = LocalDateTime.now()
) : UserDetails {
    override fun getAuthorities(): MutableCollection<out GrantedAuthority> {
        return rol.split(",") .map { SimpleGrantedAuthority("ROLE_${it.trim() }") }
    }.toMutableList()
}

    override fun getPassword(): String {
        return password
    }

    override fun getUsername(): String {
        return username
    }

    override fun isAccountNonExpired(): Boolean {
        return true
    }

    override fun isAccountNonLocked(): Boolean {
        return true
    }

    override fun isCredentialsNonExpired(): Boolean {
        return true
    }

    override fun isEnabled(): Boolean {
        return true
    }

    enum class TipoUsuario {
        SUPERADMIN, ADMIN, CLIENT
    }
}
```

5.2.2. DB

El método `getUsersInit()`, que devuelve una lista de usuarios (`User`) inicializados con algunos valores.

Cada objeto de `User` se inicializa con valores específicos para sus propiedades. Por ejemplo, el primer objeto de `User` tiene un `username` de "alejandro", un `email` de "alejandro@correo.com", una `password` de "alejandro1234", una `image` de "https://upload.wikimedia.org/wikipedia/commons/f/f4/User_Avatar_2.png", una `address` de "Aranjuez" y un rol de "ADMIN".

```
fun getUsersInit() = listOf(
    User(
        username = "alejandro",
        email = "alejandro@correo.com",
        password = "alejandro1234",
        image =
"https://upload.wikimedia.org/wikipedia/commons/f/f4/User_Avatar_2.png",
        address = "Aranjuez",
        rol = User.TipoUsuario.ADMIN.name
    ),
    User(
        username = "jorge",
        email = "jorge@correo.com",
        password = "jorge1234",
        image =
"https://upload.wikimedia.org/wikipedia/commons/f/f4/User_Avatar_2.png",
        address = "Leganés",
        rol = User.TipoUsuario.CLIENT.name
    )
)
```

5.2.3. DTO

Cada clase de DTO se define con la anotación `@Serializable`, lo que significa que los objetos de estas clases se pueden serializar (convertir en un formato de bytes) y deserializar (convertir de un formato de bytes a objetos de la clase).

La clase `UserDTO` representa un objeto de usuario completo con su información, incluyendo un `id`, `uuid`, `image`, `rol`, `email`, `username`, `address`, y `metadata`. La clase `Metadata` es una clase anidada que se utiliza para representar los metadatos relacionados con el objeto de usuario, incluyendo su fecha de creación y actualización.

Las clases `UserRegisterDTO`, `UserUpdateDTO` y `UserLoginDTO` representan objetos de usuario que se utilizan en diferentes operaciones en la aplicación. `UserRegisterDTO` se utiliza para representar los datos de registro de un nuevo usuario, mientras que `UserUpdateDTO` se utiliza para representar los datos necesarios para actualizar la información del usuario. `UserLoginDTO` se utiliza para representar los datos necesarios para autenticar al usuario en la aplicación.

La clase `UserResponseDTO` se utiliza para representar la información básica del usuario, que se devuelve en las respuestas de la API, sin incluir datos sensibles como la contraseña. La clase `UserTokenDTO` se utiliza para representar la información del usuario junto con su token de autenticación. La clase `UserDataDTO` se utiliza para representar una lista de objetos de usuario.

```
@Serializable
data class UserDTO(
    val id: Long?,
    val uuid: String,
    val image: String?,
    val rol: Set<String>,
```

```

    val email: String,
    val username: String,
    val address: String,
    val metadata: Metadata
) {
    @Serializable
    data class Metadata(
        @Serializable(with = LocalDateTimeSerializer::class)
        val createdAt: LocalDateTime? = LocalDateTime.now(),
        @Serializable(with = LocalDateTimeSerializer::class)
        val updatedAt: LocalDateTime? = LocalDateTime.now()
    )
}

@Serializable
data class UserRegisterDTO(
    val image: String?,
    val rol: Set<String>,
    val email: String,
    val username: String,
    val password: String,
    val address: String
)

@Serializable
data class UserUpdateDTO(
    val image: String?,
    val rol: Set<String>,
    val email: String,
    val username: String,
    val password: String,
    val address: String
)

@Serializable
data class UserLoginDTO(
    val username: String,
    val password: String
)

@Serializable
data class UserResponseDTO(
    val id: Long?,
    val uuid: String,
    val image: String?,
    val rol: Set<String>,
    val email: String,
    val username: String,
    val address: String
)

@Serializable
data class UserTokenDTO(
    val user: UserResponseDTO,
    val token: String
)

@Serializable
data class UserDataDTO(
    val data: List<UserResponseDTO>
)

```

5.2.4. Mappers

Estas son funciones de extensión que se definen para la clase User, la clase UserRegisterDTO y la clase UserUpdateDTO.

La función User.toDTO() convierte un objeto User en un objeto UserResponseDTO. En este proceso, algunos de los atributos del objeto User se asignan a los atributos correspondientes del objeto UserResponseDTO.

La función UserRegisterDTO.toModel() convierte un objeto UserRegisterDTO en un objeto User. En este proceso, algunos de los atributos del objeto UserRegisterDTO se asignan a los atributos correspondientes del objeto User.

La función UserUpdateDTO.toModelFromUpdated() convierte un objeto UserUpdateDTO en un objeto User. En este proceso, algunos de los atributos del objeto UserUpdateDTO se asignan a los atributos correspondientes del objeto User.

```
fun User.toDTO(): UserResponseDTO {
    return UserResponseDTO(
        id = id,
        uuid = uuid,
        image = image,
        rol = rol.split(",") .map { it.trim() } .toSet(),
        email = email,
        username = username,
        address = address
    )
}

fun UserRegisterDTO.toModel(): User {
    return User(
        image = image,
        rol = rol.joinToString(", ") { it.uppercase().trim() },
        email = email,
        username = username,
        password = password,
        address = address
    )
}

fun UserUpdateDTO.toModelFromUpdated(): User {
    return User(
        image = image,
        rol = rol.joinToString(", ") { it.uppercase().trim() },
        email = email,
        username = username,
        password = password,
        address = address
    )
}
```

5.2.5. Repository

Se trata de la interfaz de repositorio para manejar operaciones de CRUD (Crear, Leer, Actualizar, Eliminar) en una base de datos utilizando Spring Data y Kotlin.

La anotación @Repository indica que esta interfaz es un componente de Spring que maneja el acceso a datos y la persistencia. La interfaz extiende la interfaz CoroutineCrudRepository que proporciona los métodos CRUD básicos de Spring Data para trabajar con entidades, como save, findById, findAll, delete, etc.

Además de los métodos CRUD básicos, esta interfaz también define tres métodos personalizados que devuelven un flujo (Flow) de objetos User que coinciden con una consulta específica:

findUserByEmail: busca un usuario por correo electrónico.

`findUserByUuid`: busca un usuario por UUID (identificador único universal).

`findUserByUsername`: busca un usuario por nombre de usuario.

```
@Repository
interface UsersRepository : CoroutineCrudRepository<User, Long> {
    fun findUserByEmail(email: String): Flow<User>
    fun findUserByUuid(uuid: String): Flow<User>
    fun findUserByUsername(username: String): Flow<User>
}
```

Los métodos `findAll()`, `findById()`, `findByEmail()` y `findByUUID()` se utilizan para buscar usuarios en la caché. Todos ellos devuelven un objeto `User` o una secuencia de objetos `User` envuelto en un objeto `Flow`, que permite obtener resultados de manera asíncrona y con una API reactiva.

El método `save()` se utiliza para almacenar un usuario en la caché y devuelve el mismo usuario con su identificador actualizado. El método `update()` actualiza un usuario existente con los datos del usuario proporcionado y devuelve el usuario actualizado o nulo si el usuario no existe. Finalmente, el método `deleteById()` se utiliza para eliminar un usuario de la caché y devuelve el usuario eliminado o nulo si no existe.

```
interface IUserCachedRepository {
    suspend fun findAll(): Flow<User>
    suspend fun findById(id: Long): User?
    suspend fun findByEmail(email: String): User?
    suspend fun findByUUID(uuid: String): User?
    suspend fun save(user: User): User
    suspend fun update(id: Long, user: User): User?
    suspend fun deleteById(id: Long): User?
}
```

Este código define una implementación de la interfaz `IUserCachedRepository`, que se utiliza para acceder a los datos de los usuarios. La implementación utiliza una caché para mejorar el rendimiento de las operaciones de lectura y escritura en la base de datos.

La anotación `@Repository` indica que la clase es un componente de Spring que maneja el acceso a la base de datos y la caché. La clase se inyecta con una instancia de la interfaz `UsersRepository`, que se utiliza para acceder a los datos de la base de datos.

Los métodos definidos en la interfaz `IUserCachedRepository` se implementan en esta clase. Cada método realiza una operación en la base de datos y/o en la caché. Los métodos anotados con `@Cacheable` consultan la caché antes de realizar una operación en la base de datos, y los métodos anotados con `@CachePut` y `@CacheEvict` actualizan y eliminan datos de la caché, respectivamente.

En resumen, esta clase implementa una capa de acceso a datos para los usuarios, que utiliza una caché para mejorar el rendimiento de las operaciones de lectura y escritura en la base de datos.

```
@Repository
class UserCachedRepository
@Autowired constructor(
    private val usersRepository: UsersRepository
) : IUserCachedRepository {
    override suspend fun findAll(): Flow<User> = withContext(Dispatchers.IO) {
        logger.info { "findAll()" }

        return@withContext usersRepository.findAll()
    }
}
```

```

    @Cacheable("USERS")
    override suspend fun findById(id: Long): User? = withContext(Dispatchers.IO) {
        logger.info { "findById($id)" }

        return@withContext usersRepository.findById(id)
    }

    @Cacheable("USERS")
    override suspend fun findByEmail(email: String): User? =
    withContext(Dispatchers.IO) {
        logger.info { "findByEmail($email)" }

        return@withContext usersRepository.findUserByEmail(email).firstOrNull()
    }

    @Cacheable("USERS")
    override suspend fun findByUUID(uuid: String): User? =
    withContext(Dispatchers.IO) {
        logger.info { "findByUUID($uuid)" }

        return@withContext usersRepository.findUserByUuid(uuid).firstOrNull()
    }

    @CachePut("USERS")
    override suspend fun save(user: User): User = withContext(Dispatchers.IO) {
        logger.info { "save($user)" }

        val saved = user.copy(
            id = user.id,
            uuid = user.uuid,
            createdAt = LocalDateTime.now(),
            updatedAt = LocalDateTime.now()
        )

        return@withContext usersRepository.save(user)
    }

    @CachePut("USERS")
    override suspend fun update(id: Long, user: User): User? =
    withContext(Dispatchers.IO) {
        logger.info { "Updating user: $user" }

        val userDB =
        usersRepository.findUserByUsername(user.username).firstOrNull()

        userDB?.let {
            val updtatedUser = user.copy(
                id = user.id,
                updatedAt = LocalDateTime.now()
            )

            return@withContext usersRepository.save(updtatedUser)
        }

        return@withContext null
    }

    @CacheEvict("USERS")
    override suspend fun deleteById(id: Long): User? = withContext(Dispatchers.IO)
{
    logger.info { "deleteById($id)" }

    val user = findById(id)
    user?.let {
        usersRepository.deleteById(id)
    }
}

```

```

        return@withContext user
    }

    return@withContext null
}
}

```

En este caso, se definen tres clases selladas: StorageException, TokenException y UserExceptions. Cada una de ellas tiene dos subclases definidas:

StorageNotFoundException y StorageBadRequestException son subclases de StorageException, y representan las excepciones que pueden ocurrir al realizar operaciones en un almacenamiento de datos.

TokenInvalidException es una subclase de TokenException, y representa una excepción que ocurre cuando se encuentra un token inválido.

UserNotFoundException y UserBadRequestException son subclases de UserExceptions, y representan las excepciones que pueden ocurrir al realizar operaciones relacionadas con los usuarios.

Cada una de estas clases selladas tiene anotaciones de @ResponseStatus con un valor de estado HTTP correspondiente para cada una de las subclases. Esto significa que cuando se arroja una excepción, se puede personalizar la respuesta HTTP que se envía de vuelta al cliente, en lugar de tener que manejar la excepción en el controlador y construir la respuesta HTTP manualmente.

```

sealed class StorageException(message: String) : RuntimeException(message)

@ResponseStatus(HttpStatus.NOT_FOUND)
class StorageNotFoundException(message: String) : StorageException(message)

@ResponseStatus(HttpStatus.BAD_REQUEST)
class StorageBadRequestException(message: String) : StorageException(message)

sealed class TokenException(message: String) : RuntimeException(message)

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

sealed class UserExceptions(message: String) : RuntimeException(message)

@ResponseStatus(HttpStatus.NOT_FOUND)
class UserNotFoundException(message: String) : RuntimeException(message)

@ResponseStatus(HttpStatus.BAD_REQUEST)
class UserBadRequestException(message: String) : RuntimeException(message)

```

5.2.6. Services

Esta es una interfaz que define los métodos que debe implementar cualquier servicio de almacenamiento que se quiera utilizar en una aplicación. Los métodos definidos en esta interfaz permiten inicializar el servicio de almacenamiento, cargar todos los archivos almacenados en el servicio, cargar un archivo específico por nombre, almacenar un archivo, y eliminar un archivo por su nombre.

Aquí está la descripción de los métodos definidos en esta interfaz:

`initStorageService()`: Este método se encarga de inicializar el servicio de almacenamiento. La implementación de este método dependerá del servicio de almacenamiento específico que se esté utilizando.

`loadAll()`: Este método devuelve un Stream de Path que contiene todos los archivos almacenados en el servicio de almacenamiento. La implementación de este método dependerá del servicio de almacenamiento específico que se esté utilizando.

`loadFile(fileName: String)`: Este método carga un archivo específico del servicio de almacenamiento por su nombre y devuelve un objeto Path que representa ese archivo. Si el archivo no existe, lanzará una excepción `StorageNotFoundException`.

`storeFile(file: MultipartFile)`: Este método almacena un archivo en el servicio de almacenamiento y devuelve el nombre del archivo almacenado. El parámetro file es un objeto `MultipartFile` que representa el archivo que se quiere almacenar.

`deleteFile(fileName: String)`: Este método elimina un archivo del servicio de almacenamiento por su nombre. Si el archivo no existe, lanzará una excepción `StorageNotFoundException`.

```
interface IStorageService {  
    fun initStorageService()  
    fun loadAll(): Stream<Path>  
    fun loadFile(fileName: String): Path  
    fun storeFile(file: MultipartFile): String  
    fun deleteFile(fileName: String)  
}
```

Este es un ejemplo de implementación de una interfaz `IStorageService` que define los métodos necesarios para manejar un servicio de almacenamiento de archivos. La clase `StorageService` es anotada con `@Service`, lo que significa que es un componente que forma parte del sistema y está disponible para ser injectado en otras clases.

El constructor de `StorageService` toma un parámetro `path` que es una ruta de directorio donde se almacenarán los archivos. Dentro del constructor, se inicializa la propiedad `ruta` con el valor de la ruta proporcionada y se llama al método `initStorageService()` para verificar si la carpeta de almacenamiento existe y crearla si es necesario.

Los métodos `loadAll()`, `loadFile()`, `storeFile()` y `deleteFile()` implementan la funcionalidad definida en la interfaz `IStorageService`. Por ejemplo, `loadAll()` devuelve un `Stream<Path>` que contiene todas las rutas de archivos dentro del directorio de almacenamiento. Mientras tanto, `storeFile()` guarda el archivo proporcionado en la ubicación del directorio de almacenamiento y devuelve el nombre del archivo guardado.

Cada método dentro de la clase incluye un bloque try-catch que maneja excepciones y lanza excepciones personalizadas `StorageBadRequestException` o `StorageNotFoundException` si es necesario. Además, cada método registra mensajes informativos utilizando un logger `logger.info()`.

En resumen, `StorageService` es una clase que implementa la funcionalidad necesaria para manejar un servicio de almacenamiento de archivos y proporciona métodos para inicializar el servicio, cargar archivos y guardar archivos. También maneja excepciones para proporcionar mensajes de error personalizados.

```
@Service  
class StorageService(  
    @Value("\${upload.root-location}") path: String,  
) : IStorageService {  
    private val ruta: Path
```

```

init {
    logger.info { "Initializing service." }

    ruta = Paths.get(path)
    this.initStorageService()
}

override fun initStorageService() {
    try {
        if (!Files.exists(ruta))
            Files.createDirectory(ruta)
    } catch (e: IOException) {
        throw StorageBadRequestException("Unable to initialize storage service -> ${e.message}")
    }
}

override fun loadAll(): Stream<Path> {
    logger.info { "Loading storage..." }

    return try {
        Files.walk(ruta, 1)
            .filter { path -> !path.equals(ruta) }
            .map(ruta::relativize)
    } catch (e: IOException) {
        throw StorageBadRequestException("Error reading storage service -> ${e.message}")
    }
}

override fun loadFile(fileName: String): Path {
    logger.info { "Loading $fileName..." }

    return ruta.resolve(fileName)
}

override fun storeFile(file: MultipartFile): String {
    logger.info { "Storing $file..." }

    val fileName = StringUtils.cleanPath(file.originalFilename.toString())
    val extension = StringUtils.getFilenameExtension(fileName).toString()
    val saved = UUID.randomUUID().toString() + "." + extension

    try {
        if (file.isEmpty) {
            throw StorageBadRequestException("Error storing: $fileName")
        }
        if (fileName.contains("..")) {
            throw StorageBadRequestException("Security path error storing: $fileName")
        }
        file.inputStream.use { inputStream ->
            Files.copy(
                inputStream, ruta.resolve(saved),
                StandardCopyOption.REPLACE_EXISTING
            )
            return saved
        }
    } catch (e: IOException) {
        throw StorageBadRequestException("Error storing: $fileName -> ${e.message}")
    }
}

```

```

override fun deleteFile(fileName: String) {
    logger.info { "Deleting $fileName..." }

    try {
        val file = loadFile(fileName)
        Files.deleteIfExists(file)

        if (!Files.exists(file)) {
            throw StorageNotFoundException("File $fileName not found.")
        } else {
            Files.delete(file)
        }
    } catch (e: IOException) {
        throw StorageBadRequestException("Error deleting: $fileName -> ${e.message}")
    }
}

```

Esta clase UserService es una implementación de la interfaz UserDetailsService de Spring Security, que proporciona un método para cargar los detalles del usuario por nombre de usuario. Además, la clase UserService tiene otros métodos que se utilizan para buscar, guardar, actualizar y eliminar usuarios.

En los parámetros del constructor de UserService, se utilizan las anotaciones @Autowired para inyectar instancias de los repositorios UsersRepository y UserCachedRepository, y el objeto passwordEncoder.

El UserService también utiliza la anotación @Cacheable para cachear los resultados del método findUserById(), lo que puede mejorar el rendimiento de la aplicación al reducir la cantidad de llamadas a la base de datos.

```

@Service
class UserService
@Autowired constructor(
    private val usersRepository: UsersRepository,
    private val userCachedRepository: UserCachedRepository,
    private val passwordEncoder: PasswordEncoder
) : UserDetailsService {
    override fun loadUserByUsername(username: String?): UserDetails = runBlocking {
        return@runBlocking
        usersRepository.findUserByUsername(username!!).firstOrNull() ?: throw UserNotFoundException("User doesn't found with username: $username")
    }

    suspend fun findAll() = withContext(Dispatchers.IO) {
        logger.info { "findAll()" }

        return@withContext userCachedRepository.findAll()
    }

    @Cacheable("USERS")
    suspend fun findUserById(id: Long) = withContext(Dispatchers.IO) {
        return@withContext userCachedRepository.findById(id) ?: throw UserNotFoundException("User with id $id not found.")
    }

    suspend fun save(user: User, isAdmin: Boolean = false): User =
        withContext(Dispatchers.IO) {
            if (usersRepository.findUserByUsername(user.username).firstOrNull() != null)

```

```

    ) {
        logger.info { "User already exists." }
        throw UserBadRequestException("Username already exists.")
    }
    if (usersRepository.findUserByEmail(user.email)
        .firstOrNull() != null
    ) {
        logger.info { "Email already exists." }
        throw UserBadRequestException("Email already exists.")
    }

    val saved = user.copy(
        password = passwordEncoder.encode(user.password),
        createdAt = LocalDateTime.now(),
        updatedAt = LocalDateTime.now()
    )

    try {
        return@withContext userCachedRepository.save(saved)
    } catch (e: Exception) {
        throw UserBadRequestException("Error creating the user. ->
${e.message}")
    }
}

suspend fun update(id: Long, user: User): User? = withContext(Dispatchers.IO) {
    try {
        return@withContext userCachedRepository.update(id, user)
    } catch (e: UserBadRequestException) {
        throw UserBadRequestException("Error updating the user.")
    } catch (ex: UserNotFoundException) {
        throw UserNotFoundException("User with id $id not found.")
    }
}

suspend fun deleteById(id: Long): User? = withContext(Dispatchers.IO) {
    return@withContext userCachedRepository.deleteById(id)
    ?: throw UserNotFoundException("User with id $id not found.")
}
}

```

5.2.7. Controllers

La anotación "@RequestMapping("/users")" establece la ruta base para todos los puntos finales dentro de esta clase.

La clase UsersController tiene varios métodos que corresponden a diferentes puntos finales REST. Por ejemplo, el método "login" es un punto final REST para autenticar y generar un token JWT. El método "register" es un punto final REST para registrar nuevos usuarios.

Los otros métodos son para realizar diferentes operaciones CRUD en los usuarios, como buscar, actualizar y eliminar usuarios. Estos métodos están anotados con "@PreAuthorize" para garantizar que solo los usuarios con los roles apropiados puedan acceder a ellos.

```

@RestController
@RequestMapping("/users")
class UsersController @Autowired constructor(
    private val userService: UserService,
    private val authenticationManager: AuthenticationManager,
    private val jwtTokenUtils: JwtTokenUtils
) {
    @PostMapping("/login")
    fun login(@RequestBody loggingDto: UserLoginDTO): ResponseEntity<UserTokenDTO> {
        try {
            logger.info { "User login: ${loggingDto.username}" }
        }
    }
}

```

```

        val authentication: Authentication =
authenticationManager.authenticate(
            UsernamePasswordAuthenticationToken(
                loggingDto.username,
                loggingDto.password
            )
        )

        SecurityContextHolder.getContext().authentication = authentication

        val user = authentication.principal as User

        val jwtToken: String = jwtTokenUtils.generateToken(user)
        logger.info { "Token de usuario: $jwtToken" }

        return ResponseEntity.ok(UserTokenDTO(user.toDTO(), jwtToken))
    } catch (e: Exception) {
        throw ResponseStatusException(HttpStatus.UNAUTHORIZED)
    }
}

@PostMapping("/register")
suspend fun register(@RequestBody usuarioDto: UserRegisterDTO): ResponseEntity<UserTokenDTO> {
    logger.info { "User register: ${usuarioDto.username}" }

    try {
        val user = usuarioDto.validate().toModel()
        user.role.forEach { println(it) }
        val userInsert = userService.save(user)
        val jwtToken: String = jwtTokenUtils.generateToken(userInsert)

        logger.info { "Token de usuario: $jwtToken" }
        return ResponseEntity.ok(UserTokenDTO(userInsert.toDTO(), jwtToken))
    } catch (e: UserBadRequestException) {
        throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
    }
}

@PreAuthorize("hasAnyRole('ADMIN', 'SUPERADMIN')")
@GetMapping("")
suspend fun findAll(@AuthenticationPrincipal user: User): ResponseEntity<UserDataDTO> {
    logger.info { "API -> findAll()" }

    val res = userService.findAll().toList().map { it.toDTO() }
    val res2 = UserDataDTO(res)
    return ResponseEntity.ok(res2)
}

@PreAuthorize("hasAnyRole('ADMIN', 'SUPERADMIN')")
@GetMapping("/{id}")
suspend fun findById(@PathVariable id: String): ResponseEntity<UserResponseDTO> {
    logger.info { "API -> findById($id)" }

    try {
        val res = userService.findUserById(id.toLong()).toDTO()
        return ResponseEntity.ok(res)
    } catch (e: UserNotFoundException) {
        throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
    }
}

```

```

    @PreAuthorize("hasAnyRole('ADMIN', 'SUPERADMIN')")
    @PutMapping("/{id}")
    suspend fun update(
        @AuthenticationPrincipal user: User,
        @PathVariable id: String, @RequestBody userDTO: UserUpdateDTO
    ): ResponseEntity<UserResponseDTO> {
        logger.info { "API -> update($id)" }

        try {
            val rep = userDTO.validate()
            val updated = user.copy(
                image = userDTO.image,
                address = userDTO.address
            )
            val res = userService.update(id.toLong(), updated)

            return ResponseEntity.status(HttpStatus.OK).body(res?.toDTO())
        } catch (e: UserNotFoundException) {
            throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
        } catch (e: UserBadRequestException) {
            throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
        }
    }

    @PreAuthorize("hasAnyRole('SUPERADMIN')")
    @DeleteMapping("/{id}")
    suspend fun delete(@PathVariable id: String): ResponseEntity<UserDTO> {
        logger.info { "API -> delete($id)" }

        try {
            userService.deleteById(id.toLong())
            return ResponseEntity.noContent().build()
        } catch (e: UserNotFoundException) {
            throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
        } catch (e: UserBadRequestException) {
            throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
        }
    }
}

```

La anotación `@RestController` se usa para marcar la clase como un controlador de Spring que maneja solicitudes HTTP. La anotación `@RequestMapping("/users/storage")` especifica la URL base para las solicitudes que maneja este controlador.

La clase tiene tres métodos para manejar solicitudes HTTP GET, POST y DELETE.

El método `load()` se usa para manejar solicitudes GET. Recupera el archivo solicitado del servicio de almacenamiento y lo devuelve en la respuesta HTTP. El nombre del archivo se especifica como una variable de ruta en la URL.

El método `upload()` maneja solicitudes POST que contienen un archivo en formato multipart/form-data. Almacena el archivo en el servicio de almacenamiento y devuelve una respuesta HTTP que incluye una URL para recuperar el archivo.

El método `delete()` maneja solicitudes DELETE y borra el archivo especificado en la variable de ruta de la URL.

En resumen, la clase `StorageController` es un controlador de Spring que maneja solicitudes HTTP relacionadas con la carga, descarga y eliminación de archivos en un servicio de almacenamiento.

```

@RestController
@RequestMapping("/users/storage")

```

```

class StorageController {
    @Autowired constructor(
        private val storageService: StorageService
    ) {
        @GetMapping(value = ["{filename:.+}"])
        @ResponseBody
        fun load(@PathVariable filename: String?, request: HttpServletRequest): ResponseEntity<Resource> =
            runBlocking {
                try {
                    val scope = CoroutineScope(Dispatchers.IO)

                    val file = storageService.loadFile(filename!!)

                    val fileResource: Resource = scope.async {
                        UrlResource(file.toUri())
                    }.await()

                    var contentType = try {

request.servletContext.getMimeType(fileResource.file.absolutePath)
                    } catch (e: IOException) {
                        throw StorageBadRequestException("Can not determinate content
type -> ${e.message}")
                    }

                    if (contentType == null) {
                        contentType = "application/octet-stream"
                    }

                    return@runBlocking ResponseEntity.ok()
                        .contentType(MediaType.parseMediaType(contentType))
                        .body<Resource?>(fileResource)
                } catch (e: Exception) {
                    throw StorageNotFoundException("Unable action -> ${e.message}")
                }
            }

        @PostMapping(value = [""], consumes = [MediaType.MULTIPART_FORM_DATA_VALUE])
        fun upload(@RequestPart("file") file: MultipartFile): ResponseEntity<Map<String, String>> = runBlocking {
            return@runBlocking try {
                if (!file.isEmpty) {
                    val scope = CoroutineScope(Dispatchers.IO)
                    val stored = scope.async { storageService.storeFile(file) }.await()

                    val url = MvcUriComponentsBuilder
                        .fromMethodName(StorageController::class.java, "load", stored,
null)
                        .build().toUriString()

                    val response =
                        mapOf("url" to url, "name" to stored, "created_at" to
LocalDateTime.now().toString())
                        ResponseEntity.status(HttpStatus.CREATED).body(response)
                } else {
                    throw StorageBadRequestException("Can not load file.")
                }

            } catch (e: StorageException) {
                throw StorageBadRequestException(e.message.toString())
            }
        }

        @DeleteMapping(value = ["{filename:.+}"])
    }
}

```

```

@ResponseBody
fun delete(@PathVariable filename: String?, request: HttpServletRequest): ResponseEntity<Resource> = runBlocking {
    try {
        val scope = CoroutineScope(Dispatchers.IO)
        scope.launch {
            storageService.deleteFile(filename!!)
        }.join()

        return@runBlocking ResponseEntity.ok().build()
    } catch (e: StorageException) {
        throw StorageBadRequestException(e.message.toString())
    }
}

```

5.2.8. Utils

La interfaz KSerializer es parte de la biblioteca de serialización de Kotlin y se utiliza para serializar y deserializar objetos en diferentes formatos. En este caso, el formato utilizado es una cadena de texto.

El objeto LocalDateTimeSerializer implementa dos funciones: deserialize() y serialize(). La función deserialize() se utiliza para convertir una cadena de texto en un objeto LocalDateTime, mientras que la función serialize() se utiliza para convertir un objeto LocalDateTime en una cadena de texto.

La función descriptor devuelve un descriptor de serialización primitivo que describe la estructura de los objetos que se van a serializar. En este caso, el descriptor tiene un nombre "LocalDateTime" y un tipo de dato primitivo "STRING".

```

object LocalDateTimeSerializer : KSerializer<LocalDateTime> {
    override val descriptor = PrimitiveSerialDescriptor("LocalDateTime",
PrimitiveKind.STRING)

    override fun deserialize(decoder: Decoder): LocalDateTime {
        return LocalDateTime.parse(decoder.decodeString())
    }

    override fun serialize(encoder: Encoder, value: LocalDateTime) {
        encoder.encodeString(value.toString())
    }
}

```

El objeto define dos funciones principales: "deserialize" y "serialize". La función "deserialize" toma una representación de cadena de un UUID y devuelve un objeto UUID. La función "serialize" toma un objeto UUID y lo convierte en una representación de cadena para que pueda ser almacenado o enviado en un formato serializado. Además, el objeto también define un descriptor que describe el tipo de datos que está siendo serializado/deserializado, que en este caso es un UUID.

```

object UUIDSerializer : KSerializer<UUID> {
    override val descriptor = PrimitiveSerialDescriptor("UUID",
PrimitiveKind.STRING)

    override fun deserialize(decoder: Decoder): UUID {
        return UUID.fromString(decoder.decodeString())
    }

    override fun serialize(encoder: Encoder, value: UUID) {
        encoder.encodeString(value.toString())
    }
}

```

5.2.9. Validators

La función UserRegisterDTO.validate() valida si el objeto UserRegisterDTO tiene un username no vacío, un email en el formato correcto y no vacío, un password no vacío y de al menos 5 caracteres, una address no vacía y un rol no vacío. Si alguna de estas condiciones no se cumple, se lanza una excepción UserBadRequestException con un mensaje de error apropiado. Si todas las condiciones se cumplen, la función devuelve el objeto UserRegisterDTO.

La función UserUpdateDTO.validate() valida si el objeto UserUpdateDTO tiene un username, un email en el formato correcto, un password de al menos 5 caracteres, una address y un rol, todos ellos no vacíos. Si alguna de estas condiciones no se cumple, se lanza una excepción UserBadRequestException con un mensaje de error apropiado. Si todas las condiciones se cumplen, la función devuelve el objeto UserUpdateDTO.

```
fun UserRegisterDTO.validate(): UserRegisterDTO {
    if (this.username.isNotBlank()) {
        if (this.email.isNotBlank() && this.email.matches(Regex("^[A-Za-z0-9+_.-]+@[.]+\$"))) {
            if (this.password.isBlank() || this.password.length < 5)
                throw UserBadRequestException("The password cannot be empty or less than 5 characters")
            else if (this.address.isBlank())
                throw UserBadRequestException("The address cannot be empty")
            else if (this.rol.isEmpty())
                throw UserBadRequestException("The rol cannot be empty")
        } else
            throw UserBadRequestException("The email cannot be empty or does not have the correct format")
    } else {
        throw UserBadRequestException("The username cannot be empty")
    }

    return this
}

fun UserUpdateDTO.validate(): UserUpdateDTO {
    if (this.username.isBlank()) {
        throw UserBadRequestException("The username cannot be empty")
    } else if (this.email.isBlank() || !this.email.matches(Regex("^[A-Za-z0-9+_.-]+@[.]+\$")))
        throw UserBadRequestException("The email cannot be empty or does not have the correct format")
    else if (this.password.isBlank() || this.password.length < 5)
        throw UserBadRequestException("The password cannot be empty or less than 5 characters")
    else if (this.address.isBlank())
        throw UserBadRequestException("The address cannot be empty")
    else if (this.rol.isEmpty())
        throw UserBadRequestException("The rol cannot be empty")

    return this
}
```

5.2.10. Config

La clase JWTAuthenticationFilter tiene tres propiedades privadas, jwtTokenUtil que es una instancia de la clase JWTTokensUtils, authenticationManager que es una instancia de la clase AuthenticationManager y logger que se utiliza para imprimir información de depuración.

La clase sobrescribe tres métodos de la clase UsernamePasswordAuthenticationFilter: attemptAuthentication, successfulAuthentication y unsuccessfulAuthentication.

El método attemptAuthentication es utilizado para intentar la autenticación del usuario. En este método, se utiliza la biblioteca de serialización de Kotlin para decodificar un objeto UserLoginDTO desde la entrada del servidor. Este objeto contiene las credenciales del usuario, como el nombre de usuario y la contraseña. A continuación, se crea un objeto UsernamePasswordAuthenticationToken con estas credenciales y se utiliza el objeto authenticationManager para autenticar al usuario.

Si la autenticación es exitosa, el método successfulAuthentication se llama, que establece una respuesta HTTP con un encabezado de autorización que contiene el token JWT generado para el usuario autenticado.

Si la autenticación falla, el método unsuccessfulAuthentication se llama, que establece una respuesta HTTP con un mensaje de error que indica que las credenciales proporcionadas son incorrectas. En este método, se utiliza una clase interna llamada BadCredentialsError que se utiliza para generar un objeto JSON que contiene información del error, como la marca de tiempo, el código de estado y el mensaje de error.

```
class JWTAuthenticationFilter(
    private val jwtTokenUtil: JWTTokenUtils,
    private val authenticationManager: AuthenticationManager
) : UsernamePasswordAuthenticationFilter() {

    @OptIn(ExperimentalSerializationApi::class)
    override fun attemptAuthentication(req: HttpServletRequest, response: HttpServletResponse): Authentication {
        logger.info { "Trying to authenticate..." }

        val credentials = Json.decodeFromStream<UserLoginDTO>(req.inputStream)
        val auth = UsernamePasswordAuthenticationToken(
            credentials.username,
            credentials.password,
        )
        return authenticationManager.authenticate(auth)
    }

    override fun successfulAuthentication(
        req: HttpServletRequest?, res: HttpServletResponse, chain: FilterChain?,
        auth: Authentication
    ) {
        logger.info { "Authentication is correct!" }

        val user = auth.principal as User
        val token: String = jwtTokenUtil.generateToken(user)
        res.addHeader("Authorization", token)
    }

    override fun unsuccessfulAuthentication(
        request: HttpServletRequest,
        response: HttpServletResponse,
        failed: AuthenticationException
    ) {
        logger.info { "Authentication incorrect" }

        val error = BadCredentialsError()
        response.status = error.status
        response.contentType = "application/json"
        response.writer.append(error.toString())
    }
}

private data class BadCredentialsError()
```

```

        val timestamp: Long = Date().time,
        val status: Int = 401,
        val message: String = "User or password incorrect.",
    ) {
    override fun toString(): String {
        return ObjectMapper().writeValueAsString(this)
    }
}

```

Este código define una clase JWTAuthorizationFilter que se utiliza para filtrar las solicitudes HTTP entrantes y comprobar si están autorizadas. Esta clase hereda de BasicAuthenticationFilter, que es un filtro de seguridad que se encarga de verificar la autenticación básica HTTP en la solicitud.

El constructor de la clase toma una instancia de JWTTokenUtils que se utiliza para verificar y decodificar el token JWT, una instancia de UserService que se utiliza para cargar el usuario de la base de datos, y una instancia de AuthenticationManager que se utiliza para autenticar la solicitud.

La función doFilterInternal() es la que realiza la comprobación de autorización. Primero, comprueba si la solicitud tiene un encabezado Authorization. Si no lo tiene, pasa la solicitud al siguiente filtro. Si tiene un encabezado de autorización, obtiene el token de autorización de la solicitud y utiliza el método getAuthentication() para obtener una instancia de UsernamePasswordAuthenticationToken que se utiliza para autenticar la solicitud.

La función getAuthentication() utiliza el método verify() de la instancia de JWTTokenUtils para verificar el token JWT y decodificarlo. Luego, utiliza el nombre de usuario codificado en el token para cargar el usuario de la base de datos utilizando el método loadUserByUsername() de la instancia de UserService. Finalmente, crea una instancia de UsernamePasswordAuthenticationToken utilizando el usuario cargado y las autoridades del usuario, y devuelve esta instancia.

```

class JWTAuthorizationFilter(
    private val jwtTokenUtil: JWTTokenUtils,
    private val service: UserService,
    authManager: AuthenticationManager,
) : BasicAuthenticationFilter(authManager) {

    @Throws(IOException::class, ServletException::class)
    override fun doFilterInternal(
        req: HttpServletRequest,
        res: HttpServletResponse,
        chain: FilterChain
    ) {
        logger.info { "Filtrando" }
        val header = req.getHeader(AUTHORIZATION.toString())
        if (header == null) {
            chain.doFilter(req, res)
            return
        }
        getAuthentication(header.substring(7))?.also {
            SecurityContextHolder.getContext().authentication = it
        }
        chain.doFilter(req, res)
    }

    private fun getAuthentication(token: String): UsernamePasswordAuthenticationToken? = runBlocking {
        logger.info { "Obteniendo autenticación" }
        val tokenDecoded = jwtTokenUtil.verify(token) ?: return@runBlocking null
    }
}

```

```

        val username = tokenDecoded.getClaim("username").toString().replace("\\"", "
"")

        val user = service.loadUserByUsername(username)

        return@runBlocking UsernamePasswordAuthenticationToken(
            user,
            null,
            user.authorities
        )
    }
}

```

generateToken: Este método recibe un objeto User y genera un token JWT utilizando la librería java-jwt. El token incluye el identificador del usuario, el emisor del token, la fecha de expiración y algunas reclamaciones (claims) personalizadas, como el nombre de usuario y el rol del usuario.

verify: Este método recibe un token JWT y verifica su validez utilizando la misma clave secreta utilizada para generar el token. Si el token es válido, devuelve un objeto DecodedJWT, que contiene la información del token, como las reclamaciones personalizadas. Si el token es inválido o ha expirado, devuelve null.

```

@Component
class JwtTokenUtils {
    fun generateToken(user: User): String {
        logger.info { "Generate token for user: ${user.username}" }

        return JWT.create()
            .withSubject(user.uuid.toString())
            .withIssuer("BiquesUsuarios")
            .withExpiresAt(Date(System.currentTimeMillis() + (60 * 60 * 1000)))
            .withClaim("username", user.username)
            .withClaim("rol", user.rol.split(",").toSet().toString())
            .sign(Algorithm.HMAC512("BiquesDAM"))
    }

    fun verify(authToken: String): DecodedJWT? {
        logger.info { "Validating the token: $authToken" }

        return try {
            JWT.require(Algorithm.HMAC512("BiquesDAM")).build().verify(authToken)
        } catch (e: Exception) {
            null
        }
    }
}

```

Esto es una clase de configuración en Spring llamada EncoderConfig, que define un bean llamado passwordEncoder() que retorna una instancia de BCryptPasswordEncoder(), una implementación de la interfaz PasswordEncoder proporcionada por Spring Security que utiliza el algoritmo de hash de contraseñas bcrypt para codificar las contraseñas de usuario. Esto permite que Spring Security pueda usar la codificación bcrypt para almacenar y verificar las contraseñas de usuario en la base de datos de la aplicación.

```

@Configuration
class EncoderConfig {
    @Bean
    fun passwordEncoder(): PasswordEncoder {
        return BCryptPasswordEncoder()
    }
}

```

La clase se inicializa con una instancia de UserService y JWTTokenUtils, y define dos métodos authManager y filterChain. El método authManager crea un objeto AuthenticationManager y lo configura para usar el servicio de usuario. El método filterChain configura la seguridad HTTP para permitir ciertas rutas a todos los usuarios y otras rutas solo a usuarios autenticados y con roles específicos. También agrega dos filtros de seguridad JWT (JWTAuthenticationFilter y JWTAuthorizationFilter) a la cadena de filtros de seguridad.

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true, prePostEnabled = true)
class SecurityConfig {
    @Autowired constructor(
        private val userService: UserService,
        private val jwtTokenUtils: JWTTokenUtils
    ) {
        @Bean
        fun authManager(http: HttpSecurity): AuthenticationManager {
            val authenticationManagerBuilder = http.getSharedObject(
                AuthenticationManagerBuilder::class.java
            )

            authenticationManagerBuilder.userDetailsService(userService)
            return authenticationManagerBuilder.build()
        }

        @Bean
        fun filterChain(http: HttpSecurity): SecurityFilterChain {
            val authenticationManager = authManager(http)

            http
                .csrf()
                .disable()
                .exceptionHandling()
                .and()

                .authenticationManager(authenticationManager)

            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)

                .and()
                .authorizeHttpRequests()
                .requestMatchers("/error/**").permitAll()
                .requestMatchers("/users/login", "/users/register").permitAll()
                .requestMatchers("/**").permitAll()
                .requestMatchers("/users", "/users{id}").hasAnyRole("ADMIN",
"SUPERADMIN")
                    .anyRequest().authenticated()

                .and()
                .addFilter(JWTAuthenticationFilter(jwtTokenUtils,
authenticationManager))
                .addFilter(JWTAuthorizationFilter(jwtTokenUtils, userService,
authenticationManager))

            return http.build()
        }
    }
}
```

5.2.11. Tests

La prueba tiene varios casos de prueba que cubren diferentes métodos del controlador:

findAll(): prueba que se obtiene una lista de usuarios correctamente.

findById(): prueba que se obtiene un usuario por su ID correctamente.

`findByIdNotFound()`: prueba que se maneja correctamente una excepción `UserNotFoundException` cuando se intenta obtener un usuario que no existe.

`update()`: prueba que se actualiza un usuario correctamente.

`updateNotFound()`: prueba que se maneja correctamente una excepción `UserNotFoundException` cuando se intenta actualizar un usuario que no existe.

`delete()`: prueba que se elimina un usuario correctamente.

`deleteNotFound()`: prueba que se maneja correctamente una excepción `UserNotFoundException` cuando se intenta eliminar un usuario que no existe.

En cada caso de prueba, se utilizan los mocks de `UserService`, `AuthenticationManager` y `JWTTokensUtils` para simular el comportamiento de estas dependencias en respuesta a las llamadas de los métodos del controlador. Luego se verifica que el resultado de la prueba sea el esperado utilizando las funciones de aserción de JUnit5 (`assertAll`, `assertNotNull`, `assertEquals`, etc.).

La prueba también utiliza `runTest` y `ExperimentalCoroutinesApi` para permitir que los métodos asíncronos (que devuelven flujos) se prueben de forma sincrónica y fácil de leer.

```
@ExtendWith(MockKExtension::class)
@SpringBootTest
class UsersControllerTest {
    @MockK
    private lateinit var userService: UserService

    @MockK
    private lateinit var authenticationManager: AuthenticationManager

    @MockK
    private lateinit var jwtUtils: JWTTokensUtils

    @InjectMocks
    lateinit var usersController: UsersController

    private final val user = User(
        id = 12L,
        uuid = "91e0c247-c611-4ed2-8db8-a495f1f16fee",
        username = "TestService",
        email = "testService@gmail.com",
        password = "test1234",
        image =
        "https://upload.wikimedia.org/wikipedia/commons/f/f4/User_Avatar_2.png",
        address = "Test Services",
        rol = User.TipoUsuario.ADMIN.name
    )

    val userResponseDTO = user.toDTO()

    val userRegisterDTO = UserRegisterDTO(
        username = "TestService",
        email = "testService@gmail.com",
        password = "test1234",
        image =
        "https://upload.wikimedia.org/wikipedia/commons/f/f4/User_Avatar_2.png",
        address = "Test Services",
        rol = setOf(User.TipoUsuario.ADMIN.name)
    )

    val userUpdateDTO = UserUpdateDTO(
```

```

        username = "TestService",
        email = "testService@gmail.com",
        password = "test1234",
        image =
"https://upload.wikimedia.org/wikipedia/commons/f/f4/User_Avatar_2.png",
        address = "Test Services",
        rol = setOf(User.TipoUsuario.ADMIN.name)
    )

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun findAll() = runTest {
    coEvery { userService.findAll() } returns flowOf(user)

    val result = usersController.findAll(user)
    val dtoBody = result.body!!.data

    assertAll(
        { assertNotNull(result) },
        { assertNotNull(dtoBody) },
        { assertEquals(1, dtoBody.size) },
        { assertEquals(userResponseDTO.id, dtoBody[0].id) },
        { assertEquals(userResponseDTO.username, dtoBody[0].username) },
        { assertEquals(userResponseDTO.email, dtoBody[0].email) },
        { assertEquals(result.statusCode, HttpStatusCode.OK) },
    )
}

coVerify(exactly = 1) { userService.findAll() }
}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun findById() = runTest {
    coEvery { userService.findUserById(user.id!!) } returns user

    val result = usersController.findById(user.id.toString())
    val responseDTO = result.body!!

    assertAll(
        { assertNotNull(result) },
        { assertNotNull(responseDTO) },
        { assertEquals(userResponseDTO.id, responseDTO.id) },
        { assertEquals(userResponseDTO.username, responseDTO.username) },
        { assertEquals(userResponseDTO.email, responseDTO.email) },
        { assertEquals(result.statusCode, HttpStatusCode.OK) },
    )
}

coVerify(exactly = 1) { userService.findUserById(userResponseDTO.id!!) }
}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun findByIdNotFound() = runTest {
    coEvery { userService.findUserById(user.id!!) } throws
UserNotFoundException("User with id ${user.id} not found.")

    val exception = assertThrows<ResponseStatusException> {
        usersController.findById(user.id.toString())
    }

    assertEquals("""404 NOT_FOUND "User with id ${user.id} not found."""", exception.message)
}

coVerify(exactly = 1) { userService.findUserById(user.id!!) }
}

```

```

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun update() = runTest {
    coEvery { userService.update(any(), any()) } returns user

    val result = usersController.update(user, user.id.toString(), userUpdateDTO)
    val responseDTO = result.body!!

    assertAll(
        { assertNotNull(result) },
        { assertNotNull(responseDTO) },
        { assertEquals(userResponseDTO.id, responseDTO.id) },
        { assertEquals(userResponseDTO.username, responseDTO.username) },
        { assertEquals(userResponseDTO.email, responseDTO.email) },
        { assertEquals(result.statusCode, HttpStatusCode.OK) },
    )
}

coVerify(exactly = 1) { userService.update(any(), any()) }

}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun updateNotFound() = runTest {
    coEvery { userService.update(any(), any()) } throws
UserNotFoundException("User with id ${user.id} not found.")

    val exception = assertThrows<ResponseStatusException> {
        usersController.update(user, user.id.toString(), userUpdateDTO)
    }

    assertEquals("""404 NOT_FOUND "User with id ${user.id} not found."""", exception.message)

    coVerify(exactly = 1) { userService.update(any(), any()) }
}

}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun delete() = runTest {
    coEvery { userService.deleteById(user.id!!) } returns user

    val result = usersController.delete(user.id.toString())

    assertAll(
        { assertNotNull(result) },
        { assertEquals(result.statusCode, HttpStatusCode.NO_CONTENT) },
    )

    coVerify(exactly = 1) { userService.deleteById(user.id!!) }
}

}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun deleteNotFound() = runTest {
    coEvery { userService.deleteById(user.id!!) } throws
UserNotFoundException("User with id ${user.id} not found.")

    val exception = assertThrows<ResponseStatusException> {
        usersController.delete(user.id.toString())
    }

    assertEquals("""404 NOT_FOUND "User with id ${user.id} not found."""", exception.message)
}

```

```
}
```

Este es un archivo de prueba para una clase llamada UsersCachedRepository. El código hace uso del framework MockK para crear objetos simulados para las pruebas. También utiliza la anotación @SpringBootTest para realizar pruebas de integración con Spring Boot.

La clase UsersCachedRepositoryTest contiene varios métodos que prueban los métodos de la clase UsersCachedRepository como findAll(), findById(), findByEmail(), findByUuid(), save(), update() y deleteById().

En cada método de prueba, se simula el comportamiento del objeto de repositorio real para probar el comportamiento de la clase UsersCachedRepository. Cada prueba verifica que la funcionalidad de la clase de repositorio se comporta de la manera esperada.

```
@ExtendWith(MockKExtension::class)
@SpringBootTest
class UsersCachedRepositoryTest {

    private val user = User(
        id = 12L,
        uuid = "91e0c247-c611-4ed2-8db8-a495f1f16fee",
        username = "Test",
        email = "test@gmail.com",
        password = "test1234",
        image =
"https://upload.wikimedia.org/wikipedia/commons/f/f4/User_Avatar_2.png",
        address = "Test Services",
        rol = User.TipoUsuario.CLIENT.name
    )

    @MockK
    lateinit var repository: UsersRepository

    @InjectMocks
    lateinit var repositoryCached: UserCachedRepository

    init {
        MockKAnnotations.init(this)
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun findAll() = runTest {
        coEvery { repository.findAll() } returns flowOf(user)

        val result = repositoryCached.findAll().toList()
        assertEquals(1, result.size)
        assertEquals(user, result[0])
    }

    coVerify(exactly = 1) { repository.findAll() }
}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun findById() = runTest {
    coEvery { repository.findById(user.id!!) } returns user

    val result = repositoryCached.findById(12L)
```

```

        assertAll(
            { assertEquals(user.email, result!!.email) },
            { assertEquals(user.username, result!!.username) },
        )

        coVerify { repository.findById(any()) }
    }

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun findByIdNoEncontrado() = runTest {
    coEvery { repository.findById(user.id!!) } returns null

    val result = repositoryCached.findById(12L)
    assertNull(result)

    coVerify { repository.findById(any()) }
}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun findByEmail() = runTest {
    coEvery { repository.findUserByEmail(any()) } returns flowOf(user)

    val result = repositoryCached.findByEmail("test@gmail.com")
    assertAll(
        { assertEquals(user.email, result!!.email) },
        { assertEquals(user.username, result!!.username) },
    )

    coVerify { repository.findUserByEmail(any()) }
}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun findByEmailNoEncontrado() = runTest {
    coEvery { repository.findUserByEmail(any()) } returns flowOf()

    val result = repositoryCached.findByEmail("sara@gmail.com")
    assertNull(result)

    coVerify { repository.findUserByEmail(any()) }
}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun findByUuid() = runTest {
    coEvery { repository.findUserByUuid(user.uuid) } returns flowOf(user)

    val result = repositoryCached.findByUUID(user.uuid) !!

    assertAll(
        { assertEquals(user.email, result.email) },
        { assertEquals(user.username, result.username) },
    )
}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun findByUuidNoEncontrado() = runTest {
    coEvery { repository.findUserByUuid(user.uuid) } returns flowOf()

    val result = repositoryCached.findByUUID(user.uuid)
}

```

```

        assertNull(result)
    }

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun save() = runTest {
    coEvery { repository.save(user) } returns user

    val result = repositoryCached.save(user)

    assertAll(
        { assertEquals(user.email, result.email) },
        { assertEquals(user.username, result.username) },
    )

    coVerify { repository.save(user) }
}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun update() = runTest {
    coEvery { repository.findUserByUsername(any()) } returns flowOf(user)
    coEvery { repository.save(any()) } returns user

    val result = repositoryCached.update(user.id!!, user)!!

    assertAll(
        { assertEquals(user.email, result.email) },
        { assertEquals(user.username, result.username) },
    )

    coVerify { repository.save(any()) }
}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun updateNoEncontrado() = runTest {
    coEvery { repository.findUserByUsername(any()) } returns flowOf()

    val result = repositoryCached.update(user.id!!, user)

    assertNull(result)
}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun deleteById() = runTest {
    coEvery { repository.findById(any()) } returns user
    coEvery { repository.deleteById(any()) } returns Unit

    repositoryCached.deleteById(user.id!!)

    coVerify { repository.findById(any()) }
    coVerify { repository.deleteById(any()) }
}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun deleteByIdNoEncontrado() = runTest {
    coEvery { repository.findById(any()) } returns null

    repositoryCached.deleteById(user.id!!)
}

```

```

        coVerify { repository.findById(any()) }
    }
}

```

Este código es una suite de pruebas unitarias para la clase UserService, que es responsable de la lógica de negocio relacionada con los usuarios de una aplicación. Las pruebas están escritas usando el marco de pruebas de JUnit 5. La clase UserService tiene tres dependencias: UsersRepository, UserCachedRepository y PasswordEncoder. Se usan objetos simulados (mocks) para estas dependencias en las pruebas. Las pruebas cubren la funcionalidad de UserService, como cargar usuarios por nombre de usuario, buscar usuarios por ID, guardar usuarios y lanzar excepciones si se intenta guardar usuarios con nombres de usuario o correos electrónicos duplicados. Se usa la biblioteca MockK para crear objetos simulados y se usan funciones coroutine para hacer pruebas asíncronas.

```

@ExtendWith(MockKExtension::class)
@SpringBootTest
class UserServiceTest {

    private val user = User(
        id = 12L,
        uuid = "91e0c247-c611-4ed2-8db8-a495f1f16fee",
        username = "Test",
        email = "test@gmail.com",
        password = "test1234",
        image =
"https://upload.wikimedia.org/wikipedia/commons/f/f4/User_Avatar_2.png",
        address = "Test Services",
        rol = User.TipoUsuario.ADMIN.name
    )

    @MockK
    lateinit var repository: UsersRepository

    @MockK
    lateinit var repositoryCached: UserCachedRepository

    @MockK
    lateinit var passwordEncoder: PasswordEncoder

    @InjectMocks
    lateinit var userService: UserService

    init {
        MockKAnnotations.init(this)
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun loadByUsername() = runTest {
        coEvery { repository.findUserByUsername(any()) } returns flowOf(user)

        val result = userService.loadUserByUsername(user.username)

        assertAll(
            { assertEquals(user.username, result.username) },
        )
    }
}

```

```

        coVerify { repository.findUserByUsername(any()) }

    }

    @Test
    fun loadByUsernameNoEncontrado() = runTest {
        coEvery { repository.findUserByUsername(any()) } returns flowOf(user)

        val result = userService.loadUserByUsername(user.username)

        assertAll(
            { assertEquals(user.username, result.username) },
        )

        coVerify { repository.findUserByUsername(any()) }
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun findAll() = runTest {
        coEvery { repositoryCached.findAll() } returns flowOf(user)

        val result = userService.findAll().toList()

        assertAll(
            { assertEquals(1, result.size) },
            { assertEquals(user, result[0]) }
        )

        coVerify(exactly = 1) { repositoryCached.findAll() }
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun findUserById() = runTest {
        coEvery { repositoryCached.findById(any()) } returns user

        val result = userService.findUserById(user.id!!)

        assertAll(
            { assertEquals(user.username, result.username) },
            { assertEquals(user.email, result.email) },
        )

        coVerify { repositoryCached.findById(any()) }
    }

    @Test
    fun findUserByIdNoEncontrado() = runTest {
        coEvery { repositoryCached.findById(any()) } returns null

        val res = assertThrows<UserNotFoundException> {
            userService.findUserById(user.id!!)
        }

        assertEquals("User with id ${user.id} not found.", res.message)

        coVerify { repositoryCached.findById(any()) }
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    @Test

```

```

fun save() = runTest {
    coEvery { repository.findUserByUsername(any()) } returns flowOf()
    coEvery { repository.findUserByEmail(any()) } returns flowOf()
    coEvery { passwordEncoder.encode(any()) } returns "encoded_password"
    coEvery { repositoryCached.save(any()) } returns user

    val result = userService.save(user, true)

    assertAll(
        { assertEquals(user.username, result.username) },
        { assertEquals(user.email, result.email) },
    )

    coVerify(exactly = 1) { repository.findUserByUsername(any()) }
    coVerify(exactly = 1) { repository.findUserByEmail(any()) }
    coVerify(exactly = 1) { passwordEncoder.encode(any()) }
    coVerify(exactly = 1) { repositoryCached.save(any()) }

}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun saveNoEncontradoUsername() = runTest {
    coEvery { repository.findUserByUsername(any()) } returns flowOf(user)

    assertThrows<UserBadRequestException> { userService.save(user) }

    coVerify(exactly = 1) { repository.findUserByUsername(any()) }
    coVerify(exactly = 0) { repository.findUserByEmail(any()) }
    coVerify(exactly = 0) { passwordEncoder.encode(any()) }
    coVerify(exactly = 0) { repositoryCached.save(any()) }

}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun saveNoEncontradoEmail() = runTest {
    coEvery { repository.findUserByUsername(any()) } returns flowOf()
    coEvery { repository.findUserByEmail(any()) } returns flowOf(user)

    assertThrows<UserBadRequestException> { userService.save(user) }

    coVerify(exactly = 1) { repository.findUserByUsername(any()) }
    coVerify(exactly = 1) { repository.findUserByEmail(any()) }
    coVerify(exactly = 0) { passwordEncoder.encode(any()) }
    coVerify(exactly = 0) { repositoryCached.save(any()) }

}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun update() = runTest {
    coEvery { repositoryCached.findByUUID(any()) } returns user
    coEvery { repositoryCached.update(any(), any()) } returns user

    val result = userService.update(user.id!!, user)

    assertAll(
        { assertEquals(user.username, result?.username) },
        { assertEquals(user.email, result?.email) },
    )

    coVerify { repositoryCached.update(any(), any()) }

}

```

```

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun updateNoEncontrado() = runTest {
    coEvery {
        repositoryCached.update(
            any(),
            any()
        )
    } throws UserNotFoundException("User with id ${user.id} not found")

    val result = assertThrows<UserNotFoundException> {
        userService.update(user.id!!, user)
    }

    assertEquals("User with id ${user.id} not found.", result.message)

    coVerify(exactly = 1) { repositoryCached.update(any(), any()) }

}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun deleteById() = runTest {
    coEvery { repositoryCached.findUUID(any()) } returns user
    coEvery { repositoryCached.deleteById(any()) } returns user

    val result = userService.deleteById(user.id!!)

    assertAll(
        { assertEquals(user.username, result?.username) },
        { assertEquals(user.email, result?.email) },
    )

    coVerify { repositoryCached.deleteById(any()) }

}

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun deleteIdNoEncontrado() = runTest {
    coEvery { repositoryCached.deleteById(any()) } throws
UserNotFoundException("User with id ${user.id} not found.")

    val res = assertThrows<UserNotFoundException> {
        userService.deleteById(user.id!!)
    }

    assertEquals("User with id ${user.id} not found.", res.message)

    coVerify(exactly = 1) { repositoryCached.deleteById(any()) }

}

}

```

5.3. Microservicio C

5.3.1. Modelos

5.3.1.1. Modelo Appointment

```
@Serializable
@Table("Appointments")
data class Appointment{

    @Id
    val id: Long? = null,

    @Serializable(with = UUIDSerializer::class)
    val uuid: UUID,

    @Serializable(with = UUIDSerializer::class)
    val userId: UUID,

    @Contextual
    val assistance : AssistanceType,

    @Serializable(with = LocalDateSerializer::class)
    val date: LocalDate,

    val description: String
}

enum class AssistanceType(val value: String){
    ANY(value: "ANY"),
    TECHNICAL(value: "TECHNICAL");
    companion object {
        fun from(tipo: String): AssistanceType {
            return when (tipo.uppercase()){
                "ANY" -> ANY
                "TECHNICAL" -> TECHNICAL
                else -> throw IllegalArgumentException("Assistance type unknown.")
            }
        }
    }
}
```

El modelo Appointment representa las citas que puede realizar un usuario. Es serializable y tiene los siguientes atributos:

- id: Introducido con la anotación de Spring “@Id” es de tipo Long y es generado automáticamente.
- UUID y userId: Son identificadores únicos manejados por la clase UUIDSerializer.
- assistance: enum que dicta el tipo de asistencia que solicita el usuario.
- date: marca la fecha de la cita. Es manejada por la clase LocalDateSerializer.
- description: para la descripción de la cita.

5.3.1.2. Interfaz OnSale

```
interface OnSale{
    val id: Long?
    val uuid: UUID
}
```

La interfaz OnSale representa los elementos que se pueden comprar en la tienda.

- id: es el identificador del elemento.
- UUID: es el identificador único del elemento.

La interfaz OnSale representa los elementos que se pueden comprar en la tienda.

5.3.1.3. Modelo Product

```

@Serializable
@Table("Products")
data class Product{

    @Id
    override val id: Long? = null,
    @Serializable(with = UUIDSerializer::class)
    override val uuid: UUID,
    val image: String,
    val brand: String,
    val model: String,
    val description: String,
    val price: Float,
    val discountPercentage: Float,
    @Contextual
    val stock : StockType,
    val isAvailable: Boolean,
    @Contextual
    val type : ProductType,
): OnSale
}

enum class StockType(val value: String){
    HIGH( value: "HIGH"),
    LIMITED( value: "LIMITED"),
    SOLDOUT( value: "SOLDOUT");
    companion object {
        fun from(tipo: String): StockType {
            return when (tipo.uppercase()) {
                "HIGH" -> HIGH
                "LIMITED" -> LIMITED
                "SOLDOUT" -> SOLDOUT
                else -> throw IllegalArgumentException("Stock type unknown.")
            }
        }
    }
}

enum class ProductType(val value: String){
    BIKES( value: "BIKES"),
    E_BIKES( value: "E_BIKES"),
    COMPONENTS( value: "COMPONENTS"),
    EQUIPMENT( value: "EQUIPMENT"),
    ACCESORIES( value: "ACCESORIES");
    companion object {
        fun from(tipo: String): ProductType {
            return when (tipo.uppercase()) {
                "BIKES" -> BIKES
                "E_BIKES" -> E_BIKES
                "COMPONENTS" -> COMPONENTS
                "EQUIPMENT" -> EQUIPMENT
                "ACCESORIES" -> ACCESORIES
                else -> throw IllegalArgumentException("Product type unknown.")
            }
        }
    }
}

```

El modelo Product representa los productos que se pueden comprar en la tienda.

- Id: es el identificador del producto. Utiliza la anotación de Spring “@Id” y es generado automáticamente.
- UUID: es el identificador único del producto y es manejado por la clase UUIDSerializer
- Image: es la imagen del producto.
- Brand: es la marca del producto.
- Model: es el modelo del producto.
- Description: es la descripción del producto.
- Price: es el precio del producto en formato Float.
- DiscountPercentage: es el porcentaje de descuento del producto en formato Float.
- Stock: designa el Stock del producto mediante el enum StockType.
- IsAvailable: boolean que indica si el producto está disponible.
- Type: designa el Tipo de producto mediante el enum ProductType.

5.3.1.4. Modelo Service

```
@Serializable  
@Table("Services")  
data class Service(  
  
    @Id  
    override val id: Long? = null,  
  
    @Serializable(with = UuidSerializer::class)  
    override val uuid: UUID,  
  
    val image: String,  
  
    val price: Float = 0.0f,  
  
    @Serializable(with = UuidSerializer::class)  
    val appointment: UUID,  
  
    @Contextual  
    val type: ServiceType  
  
) : OnSale
```

```
enum class ServiceType(val type: String) {  
    REVISION(type: "REVISION"),  
    ASSEMBLY(type: "ASSEMBLY"),  
    REPLACEMENT(type: "REPLACEMENT"),  
    REPAIR(type: "REPAIR");  
    companion object {  
  
        /** Returns the ServiceType from a string. */  
        fun from(tipo: String): ServiceType {  
            return when (tipo.uppercase()) {  
                "REVISION" -> REVISION  
                "ASSEMBLY" -> ASSEMBLY  
                "REPLACEMENT" -> REPLACEMENT  
                "REPAIR" -> REPAIR  
                else -> throw IllegalArgumentException("Service unknown.")  
            }  
        }  
    }  
}
```

El modelo Service representa los servicios que están a la venta.

- Id: es el identificador del producto. Utiliza la anotación de Spring “@Id” y es generado automáticamente.
- UUID: es el identificador único del producto y es manejado por la clase UuidSerializer
- Image: es la imagen del producto.
- Price: es el precio del servicio en formato Float.
- Appointment: identificador de la cita asociada.
- Type: designa el tipo de servicio mediante el enum ServiceType

5.3.2. Mappers

5.3.2.1. Appointment Mapper

```
fun Appointment.toDTO(): AppointmentDTO{
    return AppointmentDTO(
        id = id,
        uuid = uuid.toString(),
        user = userId.toString(),
        assistance = assistance.type,
        date = date.toString(),
        description = description
    )
}

fun AppointmentCreateDTO.toModel(uuid: UUID): Appointment{
    return Appointment(
        uuid = uuid,
        userId = UUID.fromString(userId),
        assistance = AssistanceType.from(assistance),
        date = LocalDate.parse(date),
        description = this.description
    )
}
```

Este mapper contiene los métodos para convertir un Appointment en AppointmentDTO y para convertir un AppointmentCreateDTO en Appointment.

5.3.3.2 Product Mapper

```
fun ProductDTO.toOnSaleDTO (): OnSaleDTO {
    return OnSaleDTO(
        productEntity = ProductDTO(
            id = id,
            uuid = uuid,
            image = image,
            brand = brand,
            model = model,
            description = description,
            price = price,
            discountPercentage = discountPercentage,
            stock = stock,
            isAvailable = isAvailable,
            type = type
        ),
        serviceEntity = null,
        type = OnSaleType.PRODUCT
    )
}

fun ProductCreateDTO.toModel(uuid : UUID): Product{
    return Product(
        uuid = uuid,
        image = image,
        brand = brand,
        model = model,
        description = description,
        price = price,
        discountPercentage = discountPercentage,
        stock = StockType.from(stock),
        isAvailable = isAvailable,
        type = ProductType.from(type)
    )
}

fun Product.toDTO(): ProductDTO{
    return ProductDTO(
        id = id,
        uuid = uuid.toString(),
        image = image,
        brand = brand,
        model = model,
        description = description,
        price = price,
        discountPercentage = discountPercentage,
        stock = stock.value,
        isAvailable = isAvailable,
        type = type.value
    )
}

fun ProductCreateDTO.toOnSaleCreateDTO (): OnSaleCreateDTO {
    return OnSaleCreateDTO(
        productEntity = ProductCreateDTO(
            image = image,
            brand = brand,
            model = model,
            description = description,
            price = price,
            discountPercentage = discountPercentage,
            stock = stock,
            isAvailable = isAvailable,
            type = type
        ),
        serviceEntity = null,
        type = OnSaleType.PRODUCT
    )
}
```

Este mapper contiene los métodos para convertir un Product en ProductDTO, un productCreateDTO en Product, un ProductDTO en OnSaleDTO y un ProductCreateDTO en OnSaleCreateDTO.

5.3.2.2. Service Mapper

```
fun ServiceDTO.toOnSaleDTO(): OnSaleDTO {
    return OnSaleDTO(
        productEntity = null,
        serviceEntity = ServiceDTO(
            id = id,
            uuid = uuid,
            image = image,
            price = price,
            appointment = appointment,
            type = type ),
        type = OnSaleType.SERVICE
    )
}

fun Service.toDTO(): ServiceDTO{
    return ServiceDTO(
        id = id,
        uuid = uuid.toString(),
        image = image,
        price = price,
        appointment = appointment.toString(),
        type = type.type
    )
}

fun ServiceCreateDTO.toModel(uuid: UUID): Service {
    return Service(
        uuid = uuid,
        image = image,
        price = price,
        appointment = UUID.fromString(appointment),
        type = ServiceType.from(type)
    )
}

fun ServiceCreateDTO.toOnSaleCreateDTO(): OnSaleCreateDTO {
    return OnSaleCreateDTO(
        productEntity = null,
        serviceEntity = ServiceCreateDTO(
            image = image,
            price = price,
            appointment = appointment,
            type = type ),
        type = OnSaleType.SERVICE
    )
}
```

Este mapper contiene los métodos para convertir un Service en ServiceDTO, un ServiceCreateDTO en Service, un serviceDTO en OnSaleDTO y un ServiceCreateDTO en OnSaleCreateDTO.

5.3.4.1. AppointmentDTO

```
data class AppointmentDTO(
    val id: Long?,
    val uuid: String,
    val user: String,
    val assistance : String,
    val date: String,
    val description: String
)

data class AppointmentCreateDTO(
    val userId: String,
    val assistance : String,
    val date: String,
    val description: String
)
```

Contiene las clases de tipo DTO (Data Transfer Object) para appointment.

5.3.4.2. OnSaleDTO

```
data class OnSaleDTO(  
    val productEntity : ProductDTO?,  
    val serviceEntity : ServiceDTO?,  
    val type : OnSaleType  
)  
  
data class OnSaleCreateDTO(  
    val productEntity : ProductCreateDTO?,  
    val serviceEntity : ServiceCreateDTO?,  
    val type : OnSaleType  
)  
  
enum class OnSaleType(val value:String) {  
    PRODUCT(value: "PRODUCT"),  
    SERVICE(value: "SERVICE")  
}
```

Contiene las clases de tipo DTO (Data Transfer Object) para la interfaz OnSale.

5.3.4.3. ProductDTO

```
data class ProductDTO(  
    val id: Long?,  
    val uuid: String,  
    val image: String,  
    val brand: String,  
    val model: String,  
    val description: String,  
    val price: Float,  
    val discountPercentage: Float,  
    val stock : String,  
    val isAvailable: Boolean,  
    val type : String  
)  
  
data class ProductCreateDTO(  
    val image: String,  
    val brand: String,  
    val model: String,  
    val description: String,  
    val price: Float,  
    val discountPercentage: Float,  
    val stock : String,  
    val isAvailable: Boolean,  
    val type : String  
)
```

Contiene las clases de tipo DTO (Data Transfer Object) para Product

5.3.4.4. ServiceDTO

```
data class ServiceDTO(  
    val id: Long?,  
    val uuid: String,  
    val image: String,  
    val price: Float,  
    val appointment : String,  
    val type : String  
)  
  
data class ServiceCreateDTO(  
    val image: String,  
    val price: Float,  
    val appointment : String,  
    val type : String  
)
```

Contiene las clases de tipo DTO (Data Transfer Object) para Service

5.3.5. Repositorios

```
interface CRUDRepository<T, ID> {  
  
    suspend fun findAll(): Flow<T>  
    suspend fun findById(id: ID): T?  
    suspend fun save(entity: T): T  
    suspend fun update(entity: T): T?  
    suspend fun delete(entity: T): Boolean  
  
}
```

Todos los repositorios parten de esta interfaz CRUDRepository

5.3.5.1. AppointmentsCachedRepository, IAppointmentsCachedRepository, AppointmentsRepository

```
@Repository  
interface AppointmentsRepository : CoroutineCrudRepository<Appointment, Long> {  
  
    suspend fun findByUuid(uuid: UUID): Flow<Appointment>  
}  
  
interface IAppointmentsCachedRepository : CRUDRepository<Appointment, Long>
```

```

private val logger = KotlinLogging.logger {}

@Repository
class AppointmentsCachedRepository
    @Autowired constructor(
        private val appointmentsRepository: AppointmentsRepository
    ) : IAppointmentsCachedRepository {

    override suspend fun findAll(): Flow<Appointment> = withContext(Dispatchers.IO) { thisCoroutineScope

        logger.info { "Cached appointment - findAll()" }
        return@withContext appointmentsRepository.findAll()

    }

    @Cacheable("appointments")
    override suspend fun findById(id: Long): Appointment? = withContext(Dispatchers.IO) { thisCoroutineScope

        logger.info { "Cached appointments - findById() with id: $id" }
        return@withContext appointmentsRepository.findById(id)

    }

    @Cacheable("appointments")
    suspend fun findByUuid(uuid: UUID): Appointment? = withContext(Dispatchers.IO) { thisCoroutineScope

        logger.info { "Cached appointments - findByUuid() with id: $uuid" }
        return@withContext appointmentsRepository.findByUuid(uuid).firstOrNull()

    }

    @CachePut("appointments")
    override suspend fun save(entity: Appointment): Appointment = withContext(Dispatchers.IO) { thisCoroutineScope

        logger.info { "Cached appointments - save() appointment: $entity" }
        return@withContext appointmentsRepository.save(entity)

    }

    @CachePut("appointments")
    override suspend fun update(entity: Appointment): Appointment? = withContext(Dispatchers.IO) { thisCoroutineScope

        logger.info { "Cached appointments - update() appointment: $entity" }
        val appointment = appointmentsRepository.findByUuid(entity.uuid).firstOrNull()
        appointment?.let{ itAppointment {
            val updated = it.copy(
                uuid = entity.uuid,
                userId = entity.userId,
                assistance = entity.assistance,
                date = entity.date,
                description = entity.description,
            )
            return@withContext appointmentsRepository.save(updated)
        }}
        return@withContext null
    }

    @CacheEvict("appointments")
    override suspend fun delete(entity: Appointment): Boolean = withContext(Dispatchers.IO) { thisCoroutineScope

        logger.info { "Cached appointment - delete() product: $entity" }
        val appointment = entity.id?.let { appointmentsRepository.findById(it) }
        try {
            appointment?.let { itAppointment {
                appointmentsRepository.deleteById(it.id!!.toString().toLong())
                return@withContext true
            }}
        } catch (e: Exception) {
            throw AppointmentConflictIntegrityException("No se puede borrar el appointment: $entity")
        }
        return@withContext false
    }

}

```

Este es un repositorio que implementa a la interfaz AppointmentsRepository que utiliza Spring para realizar operaciones de CRUD (Crear, Leer, Actualizar y Eliminar) para objetos Appointment. Además, este repositorio utiliza las anotaciones para implementar la caché en Spring

Las funciones en este repositorio son las siguientes:

- **findAll**: devuelve un flow de Appointments
- **findById**: recupera un Appointment según su identificador
- **findByUUID**: recupera un Appointment según su UUID
- **save**: guarda un Appointment
- **update**: actualiza un Appointment.
- **delete**: elimina un Appointment.

5.3.5.2. ProductsCachedRepository, IProductsCachedRepository, ProductsRepository

```
@Repository
interface ProductsRepository : CoroutineCrudRepository<Product, Long> {
    suspend fun findByUuid(uuid: UUID): Flow<Product>
}

interface IProductsCachedRepository : CRUDRepository<Product, Long>

@Repository
class ProductsRepository @Autowired constructor(
    private val productsRepository: ProductsRepository
) : IProductsCachedRepository {

    override suspend fun findAll(): Flow<Product> = withContext(Dispatchers.IO) { thisCoroutineScope {
        logger.info { "Cached products - findAll()" }
        return@withContext productsRepository.findAll()
    }

    @Cacheable("products")
    override suspend fun findById(id: Long): Product? = withContext(Dispatchers.IO) { thisCoroutineScope {
        logger.info { "Cached products - findById() with id: $id" }
        return@withContext productsRepository.findById(id)
    }

    @Cacheable("products")
    suspend fun findByUuid(uuid: UUID): Product? = withContext(Dispatchers.IO) { thisCoroutineScope {
        logger.info { "Cached products - findByUuid() with id: $uuid" }
        return@withContext productsRepository.findByUuid(uuid).firstOrNull()
    }
}
}
}
}
```

```

@CachePut("products")
override suspend fun save(entity: Product) = withContext(Dispatchers.IO) { thisCoroutineScope

    logger.info { "Cached products - save() product: $entity" }
    return@withContext productsRepository.save(entity)
}

@CachePut("products")
override suspend fun update(entity: Product): Product? = withContext(Dispatchers.IO) { thisCoroutineScope
    logger.info { "Cached products - update() product: $entity" }
    val product = productsRepository.findByUuid(entity.uuid).firstOrNull()
    product?.let { itProduct ->
        val updated = it.copy(
            uuid = entity.uuid,
            image = entity.image,
            brand = entity.brand,
            model = entity.model,
            description = entity.description,
            price = entity.price,
            discountPercentage = entity.discountPercentage,
            stock = entity.stock,
            isAvailable = entity.isAvailable,
            type = entity.type
        )
        return@withContext productsRepository.save(updated)
    }
    return@withContext null
}

@CacheEvict("products")
override suspend fun delete(entity: Product): Boolean = withContext(Dispatchers.IO) { thisCoroutineScope

    logger.info { "Cached products - delete() product: $entity" }

    val product = productsRepository.findById(entity.id!!.toString().toLong())

    try {
        product?.let { itProduct ->
            productsRepository.deleteById(it.id!!.toString().toLong())
            return@withContext true
        }
    } catch (e: Exception) {
        throw ProductConflictIntegrityException("No se puede borrar el producto: $entity")
    }

    return@withContext false
}

```

Este es un repositorio que implementa a la interfaz ProductsRepository que utiliza Spring para realizar operaciones de CRUD (Crear, Leer, Actualizar y Eliminar) para objetos Product. Además, este repositorio utiliza las anotaciones para implementar la caché en Spring.

Las funciones en este repositorio son las siguientes:

- **FindAll()**: devuelve un flow de Product
- **FindById()**: recupera un Product según su identificador
- **FindByUUID()**: recupera un Product según su UUID
- **Save()**: guarda un Product
- **Update()**: actualiza un Product.
- **Delete()**: elimina un Product.

5.3.5.3. ServiceRepository

```
@Repository
interface ServiceRepository : CoroutineCrudRepository<Service, Long>{
    suspend fun findByUuid(uuid: UUID): Flow<Service>
}
```

Este es el repositorio para Service

5.3.6. Services

5.3.6.1. AppointmentsService, IAppointmentsService

```
interface IAppointmentsService {

    suspend fun findAll(): Flow<Appointment>
    suspend fun findById(id: Long): Appointment
    suspend fun save(appointment: Appointment): Appointment
    suspend fun update(appointment: Appointment): Appointment
    suspend fun delete(appointment: Appointment): Appointment

}

@Service
class AppointmentService
    @Autowired constructor(
        private val appointmentsRepository : AppointmentsCachedRepository
    ): IAppointmentsService {

    override suspend fun findAll(): Flow<Appointment> {
        logger.info { "Service appointment - findAll()" }
        return appointmentsRepository.findAll()
    }

    suspend fun findByUuid(uuid: UUID): Appointment = withContext(Dispatchers.IO) { thisCoroutineScope {
        logger.info { "Service appointments - findByUuid() with uuid: $uuid" }
        return@withContext appointmentsRepository.findByUuid(uuid)
            ?: throw AppointmentNotFoundException("Appointment not found with uuid: $uuid")
    }
}

    override suspend fun findById(id: Long): Appointment {
        logger.info { "Service appointments - findById() with id: $id" }
        return appointmentsRepository.findById(id)
            ?: throw AppointmentNotFoundException("Appointment not found with id: $id")
    }

    override suspend fun save(appointment: Appointment): Appointment {
        logger.info { "Service appointments - save() appointment: $appointment" }
        return appointmentsRepository.save(appointment)
    }

    override suspend fun update(appointment: Appointment): Appointment {
        logger.info { "Service appointments - update() appointment: $appointment" }

        val found = appointmentsRepository.findByUuid(appointment.uuid)

        return found?.let { itAppointment
            appointmentsRepository.update(appointment)
        } ?: throw AppointmentNotFoundException("Appointment not found with uuid: ${appointment.uuid}")
    }

    override suspend fun delete(appointment: Appointment): Appointment {
        logger.info { "Service appointments - delete() product: $appointment" }

        val found = appointmentsRepository.findByUuid(appointment.uuid)

        found?.let { itAppointment
            appointmentsRepository.delete(found)
            return appointment
        } ?: throw AppointmentNotFoundException("Appointment not found with uuid: ${appointment.id}")
    }
}
```

La clase AppointmentService es un servicio que se encarga de manejar las operaciones relacionadas con la entidad Appointment.

Tiene una dependencia de IAppointmentsService que le proporciona los métodos que implementa y está etiquetada como componente de Spring “@Service”.

Los métodos disponibles en este servicio son:

- **findAll()**: devuelve un Flow de todos los Appointment almacenados en la base de datos.
- **findById(UUID)**: devuelve el Appointment correspondiente al UUID proporcionado.
- **findById(id: Long)**: devuelve el Appointment correspondiente al id proporcionado.
- **save(appointment: Appointment)**: guarda el Appointment.
- **update(appointment: Appointment)**: actualiza un Appointment existente en la base de datos y lo devuelve.
- **delete(appointment: Appointment)**: elimina un Appointment existente en la base de datos y devuelve un valor booleano que indica si la eliminación fue exitosa o no.

5.3.7. ProductsService, IProductsService

```
@Service
class ProductsService {
    @Autowired constructor(
        private val productsRepository: ProductsCachedRepository
    ): IProductsService{
        override suspend fun findAll(): Flow<Product> {

            logger.info { "Service products - findAll()" }
            return productsRepository.findAll()
        }

        suspend fun findByUuid(uuid: UUID): Product = withContext(Dispatchers.IO) { thisCoroutineScope {

            logger.info { "Service products - findByUuid() with uuid: $uuid" }

            return@withContext productsRepository.findByUuid(uuid)
                ?: throw ProductNotFoundException("Not found with uuid: $uuid")
        }
    }

        override suspend fun findById(id: Long): Product {

            logger.info { "Service products - findById() with id: $id" }
            return productsRepository.findById(id)
                ?: throw ProductNotFoundException("Not found with id: $id")
        }
    }

    interface IProductsService {

        suspend fun findAll(): Flow<Product>
        suspend fun findById(id: Long): Product
        suspend fun save(product: Product): Product
        suspend fun update(product: Product): Product
        suspend fun delete(product: Product): Product
    }
}
```

```

override suspend fun save(product: Product): Product {
    logger.info { "Service products - save() product: $product" }
    return productsRepository.save(product)
}

override suspend fun update(product: Product): Product {
    logger.info { "Service products - update() product: $product" }

    val found = productsRepository.findByUuid(product.uuid)

    return found?.let { itProduct
        productsRepository.update(product)
    } ?: throw ProductNotFoundException("Not found with uuid: ${product.uuid}")
}

override suspend fun delete(product: Product): Product {
    logger.info { "Service appointments - delete() product: $product" }

    val found = productsRepository.findByUuid(product.uuid)

    found?.let { itProduct
        productsRepository.delete(found)
        return product
    } ?: throw ProductNotFoundException("Product not found with uuid: ${product.uuid}")
}

```

La clase **ProductsService** es un servicio que se encarga de manejar las operaciones relacionadas con la entidad **Products**.

Tiene una dependencia de **IProductsService** que le proporciona los métodos que implementa y está etiquetada como componente de Spring “`@Service`”.

Los métodos disponibles en este servicio son:

- **findAll()**: devuelve un **Flow** de todos los Products almacenados en la base de datos.
- **findById(uuid: UUID)**: devuelve el Product correspondiente al UUID proporcionado.
- **findById(id: Long)**: devuelve el Product correspondiente al id proporcionado.
- **save(product: Product)**: guarda el Product.
- **update(product: Product)**: actualiza un Product existente en la base de datos y lo devuelve.
- **delete(product: Product)**: elimina un Product existente en la base de datos y devuelve un valor booleano que indica si la eliminación fue exitosa o no.

5.3.8. ServicesService, IServicesService

```
interface IServicesService {  
  
    suspend fun findAll(): Flow<Service>  
    suspend fun findById(id: Long): Service  
    suspend fun save(service: Service): Service  
    suspend fun update(service: Service): Service  
    suspend fun delete(service: Service): Service  
    suspend fun findAppointment(uuid: UUID): Appointment?  
  
}
```

```
@org.springframework.stereotype.Service  
class ServicesService  
    @Autowired  
    constructor(  
        private val serviceRepository: ServiceRepository,  
        private val appointmentRepository: AppointmentsCachedRepository  
    ): IServicesService {  
  
    override suspend fun findAll(): Flow<Service> {  
  
        logger.info { "Service service - findAll()" }  
        return serviceRepository.findAll()  
    }  
  
    suspend fun findByUuid(uuid: UUID): Service = withContext(Dispatchers.IO) { thisCoroutineScope  
        logger.info { "Service service - findByUuid() with uuid: $uuid" }  
        return@withContext serviceRepository.findByUuid(uuid).firstOrNull()  
            ?: throw ServiceNotFoundException("Not found with uuid: $uuid")  
    }  
  
    override suspend fun findById(id: Long): Service {  
  
        logger.info { "Service service - findById() with id: $id" }  
        return serviceRepository.findById(id)  
            ?: throw ServiceNotFoundException("Not found with id: $id")  
    }  
  
}
```

```
override suspend fun update(service: Service): Service {  
  
    logger.info { "Service service - update() service: $service" }  
    val found = serviceRepository.findByUuid(service.uuid).firstOrNull()  
  
    if(found!=null){  
        val appointment = findAppointment(service.appointment)  
        if(appointment != null) {  
            return serviceRepository.save(found)  
        }else{  
            throw AppointmentNotFoundException("Appointment not found with uuid: ${service.appointment}")  
        }  
    }else{  
        throw ServiceNotFoundException("Not found with uuid: ${service.uuid}")  
    }  
}  
  
override suspend fun delete(service: Service): Service {  
  
    logger.info { "Service service - delete() service: $service" }  
  
    val found = serviceRepository.findByUuid(service.uuid).firstOrNull()  
  
    found?.let { itService  
        serviceRepository.delete(found)  
        return service  
    } ?: throw ServiceNotFoundException("Service not found with uuid: ${service.uuid}")  
}
```

La clase **ServicesService** es un servicio que se encarga de manejar las operaciones relacionadas con la entidad **Services**.

Tiene una dependencia de **IServicesService** que le proporciona los métodos que implementa y está etiquetada como componente de Spring “@Service”.

Los métodos disponibles en este servicio son:

- **findAll()**: devuelve un **Flow** de todos los Services almacenados en la base de datos.

- **findById(uuid: UUID)**: devuelve el Service correspondiente al UUID proporcionado.
- **findById(id: Long)**: devuelve el Service correspondiente al id proporcionado.
- **save(service: Service)**: guarda el Service.
- **update(service: Service)**: actualiza un Services existente en la base de datos y lo devuelve.
- **delete(service: Service)**: elimina un Services existente en la base de datos y devuelve un valor booleano que indica si la eliminación fue exitosa o no.

5.3.9. StorageService, IStorageService

```
interface IStorageService {

    fun init()
    fun store(file: MultipartFile): String
    fun loadAll(): Stream<Path>
    fun load(filename: String): Path
    fun loadAsResource(filename: String): Resource
    fun delete(filename: String)
    fun deleteAll()
    fun getUrl(filename: String): String
    fun store(file: MultipartFile, filenameFromUser: String): String

}

@Service
class StorageService(
    @Value("\${upload.root-location}") path: String,
): IStorageService {

    private val rootLocation: Path = Paths.get(path)

    init {
        logger.info { "File storage is starting" }
        initStorageDirectory()
    }

    private final fun initStorageDirectory() {
        if (!Files.exists(rootLocation)) {
            logger.info { "Creating storage directory: $rootLocation" }
            Files.createDirectory(rootLocation)
        } else {
            logger.info { "The directory for the storage already exists; data will be deleted" }
            deleteAll()
            Files.createDirectory(rootLocation)
        }
    }
}
```

```

override fun store(file: MultipartFile): String {

    logger.info { "Saving file: ${file.originalFilename}" }

    val filename = StringUtils.cleanPath(file.originalFilename.toString())
    val extension = StringUtils.getFilenameExtension(filename).toString()
    val storedFilename = UUID.randomUUID().toString() + "." + extension
    try {
        if (file.isEmpty) {
            throw StorageBadRequestException("Empty file $filename")
        }
        if (filename.contains( other: "..")) {
            throw StorageBadRequestException("Non authorised path $filename")
        }
        file.getInputStream.use { inputStream ->
            Files.copy(
                inputStream, rootLocation.resolve(storedFilename),
                StandardCopyOption.REPLACE_EXISTING
            )

            return storedFilename
        }
    } catch (e: IOException) {
        throw StorageBadRequestException("Error while saving file: $filename", e)
    }
}

override fun store(file: MultipartFile, filenameFromUser: String): String {

    logger.info { "Saving file: ${file.originalFilename}" }

    val filename = StringUtils.cleanPath(file.originalFilename.toString())
    val extension = StringUtils.getFilenameExtension(filename).toString()
    val storedFilename = "$filenameFromUser.$extension"
    try {
        if (file.isEmpty) {
            throw StorageBadRequestException("Empty file $filename")
        }
        if (filename.contains( other: "..")) {
            throw StorageBadRequestException("Non authorised path $filename")
        }
        file.getInputStream.use { inputStream ->
            Files.copy(
                inputStream, rootLocation.resolve(storedFilename),
                StandardCopyOption.REPLACE_EXISTING
            )

            return storedFilename
        }
    } catch (e: IOException) {
        throw StorageBadRequestException("Error while saving file: $filename", e)
    }
}

```

```
override fun loadAll(): Stream<Path> {
    logger.info { "Loading all files" }

    return try {
        Files.walk(rootLocation, maxDepth: 1)
            .filter { path -> !path.equals(rootLocation) }
            .map(rootLocation::relativize)
    } catch (e: IOException) {
        throw StorageBadRequestException("Error while reading stored files", e)
    }
}

override fun load(filename: String): Path {
    logger.info { "Loading file: $filename" }

    return rootLocation.resolve(filename)
}

override fun loadAsResource(filename: String): Resource {
    logger.info { "Loading file as resource: $filename" }

    return try {
        val file = load(filename)
        val resource = UrlResource(file.toUri())
        if (resource.exists() || resource.isReadable) {
            resource
        } else {
            throw StorageFileNotFoundException(
                "Cannot read file: $filename"
            )
        }
    } catch (e: MalformedURLException) {
        throw StorageFileNotFoundException("Cannot read file: $filename", e)
    }
}

override fun deleteAll() {
    logger.info { "Deleting all files" }

    FileSystemUtils.deleteRecursively(rootLocation.toFile())
}
```

```

override fun init() {
    logger.info { "Initializing storage file directory" }

    try {
        if (!Files.exists(rootLocation))
            Files.createDirectory(rootLocation)
    } catch (e: IOException) {
        throw StorageBadRequestException("Couldn't initialize the storage service", e)
    }
}

override fun delete(filename: String) {

    logger.info { "Deleting file: $filename" }

    val justFilename: String = StringUtils.getFilename(filename).toString()
    try {
        val file = load(justFilename)
        if (!Files.exists(file)) {
            throw StorageFileNotFoundException("File $filename does not exist")
        } else {
            Files.delete(file)
        }
    } catch (e: IOException) {
        throw StorageBadRequestException("An error occurred while deleting a file", e)
    }
}

override fun getUrl(filename: String): String {
    logger.info { "Obtaining file: $filename" }

    return MvcUriComponentsBuilder
        .fromMethodName(StorageController::class.java, methodName: "serverFile", filename, null)
        .build().toUriString()
}

```

La clase StorageService ofrece métodos para manejar el almacenamiento de archivos en una ubicación específica en el sistema de archivos local. Implementa la interfaz IStorageService y está etiquetada como componente de Spring "@Service".

Los métodos disponibles en este servicio son:

- **initStorageDirectory()**: El cual crea el directorio donde se guardarán los archivos.
- **Store(file: MultipartFile, filenameFromUser: String)**: toma un objeto MultipartFile y lo guarda. Si se proporciona un nombre de archivo personalizado, lo usará en lugar del nombre de archivo original del objeto MultipartFile.
- **loadAll()**: devuelve un Stream<Path> de todos los archivos almacenados en la ubicación de almacenamiento.
- **Load(filename: String)**: carga un archivo de la ubicación de almacenamiento y devuelve su ruta.
- **loadAsResource(filename: String)**: carga un archivo y devuelve un objeto Resource que puede ser utilizado para acceder a los datos del archivo.
- **deleteAll()**: elimina todos los archivos de la ubicación de almacenamiento.
- **Delete (filename: String)**: elimina un archivo específico de la ubicación de almacenamiento.
- **getUrl(filename: String)**: devuelve una URL con la que se puede acceder al archivo especificado.

5.3.10. Validators

Son funciones que sirven para validar que los campos de ciertos DTO sean correctos.

5.3.10.1. AppointmentValidator

```
fun AppointmentCreateDTO.validate(): AppointmentCreateDTO {  
  
    if (this.userId.isBlank())  
        throw ProductBadRequestException("The user cannot be empty.")  
    if (this.assistance.isBlank())  
        throw ProductBadRequestException("Assistance type cannot be empty.")  
    if (this.date.isBlank())  
        throw ProductBadRequestException("The date field cannot be empty.")  
  
    return this  
}
```

Este método sirve para validar que los campos de AppointmentCreateDTO son todos válidos, verificando que el userId, el tipo de asistencia y date no son campos vacíos. En caso de que no sea validado, se devuelve el método ValidationResult.Invalid.

5.3.10.2. ProductValidator

```
fun ProductCreateDTO.validate(): ProductCreateDTO {  
  
    if (this.image.isBlank())  
        throw ProductBadRequestException("The image cannot be empty.")  
    if (this.brand.isBlank())  
        throw ProductBadRequestException("The brand cannot be empty.")  
    if (this.model.isBlank())  
        throw ProductBadRequestException("The model cannot be empty.")  
    if (this.description.isBlank())  
        throw ProductBadRequestException("The description cannot be empty.")  
    if (this.price < 0.0)  
        throw ProductBadRequestException("The percentage cannot be negative.")  
    if (this.stock.isBlank())  
        throw ProductBadRequestException("Stock cannot be empty.")  
    if (this.isAvailable.toString().isBlank())  
        throw ProductBadRequestException("Availability cannot be empty.")  
    if (this.type.isBlank())  
        throw ProductBadRequestException("The Product Type cannot be empty.")  
  
    return this  
}
```

Este método sirve para validar que los campos de ProductCreateDTO son todos válidos, verificando que image, brand, model, description, stock, la disponibilidad y el tipo no son campos vacíos y que el precio es mayor a 0. En caso de que no sea validado, se devuelve el método ValidationResult.Invalid.

5.3.10.3. ServiceValidator

```
fun ServiceCreateDTO.validate(): ServiceCreateDTO {  
  
    if (this.image.isBlank()) {  
        throw ServiceBadRequestException("The image cannot be empty.")  
    } else if (this.price <= 0) {  
        throw ServiceBadRequestException("The price cannot be negative.")  
    } else if (this.type.isBlank()) {  
        throw ServiceBadRequestException("The type cannot be empty.")  
    }  
  
    return this  
}
```

Este método sirve para validar que los campos de ServiceCreateDTO son todos válidos, verificando que imagen y el tipo no son campos vacíos y que el precio es mayor a 0. En caso de que no sea validado, se devuelve el método ValidationResult.Invalid.

5.3.11. Serializers

5.3.11.1. LocalDateSerializer

```
object LocalDateSerializer : KSerializer<LocalDate> {  
  
    override val descriptor = PrimitiveSerialDescriptor(serialName: "LocalDate", PrimitiveKind.STRING)  
  
    override fun deserialize(decoder: Decoder): LocalDate {  
        return LocalDate.parse(decoder.decodeString())  
    }  
  
    override fun serialize(encoder: Encoder, value: LocalDate) {  
        encoder.encodeString(value.toString())  
    }  
}
```

LocalDateSerializer es un serializador Kotlin que convierte un LocalDate en su representación como String y viceversa. Deserialize() convierte en LocalDate, mientras que serialize() convierte en String.

5.3.11.1.2. UUIDSerializer

```
object UUIDSerializer : KSerializer<UUID> {  
    override val descriptor = PrimitiveSerialDescriptor(serialName: "UUID", PrimitiveKind.STRING)  
  
    override fun deserialize(decoder: Decoder): UUID {  
        return UUID.fromString(decoder.decodeString())  
    }  
  
    override fun serialize(encoder: Encoder, value: UUID) {  
        encoder.encodeString(value.toString())  
    }  
}
```

UUIDSerializer es un serializador Kotlin que convierte un UUID en su representación como String y viceversa. Deserialize() convierte en UUID, mientras que serialize() convierte en String.

5.3.12. Exceptions

Hemos hecho una serie de excepciones personalizadas para las respuestas Http de Spring utilizando la anotación “@ResponseStatus”

5.3.12.1. AppointmentException

```
sealed class AppointmentException(message: String): RuntimeException(message)

@ResponseStatus(HttpStatus.NOT_FOUND)
class AppointmentNotFoundException(message: String) : AppointmentException(message)

@ResponseStatus(HttpStatus.BAD_REQUEST)
class AppointmentBadRequestException(message: String) : AppointmentException(message)

@ResponseStatus(HttpStatus.BAD_REQUEST)
class AppointmentConflictIntegrityException(message: String) : AppointmentException(message)
```

La clase sellada AppointmentException contiene las subclases AppointmentNotFoundException, la cual se utiliza cuando no se encuentra el Appointment solicitado y establece el código HTTP 404 Not Found; AppointmentBadRequestException, se utiliza cuando se proporciona una solicitud incorrecta y establece el código HTTP 400 Bad Request; y AppointmentConflictIntegrityException, la cual se utiliza cuando se produce un conflicto de integridad en la base de datos y establece el código de estado HTTP 409 Conflict.

5.3.12.2. OnSaleException

```
sealed class OnSaleException(message: String): RuntimeException(message)

@ResponseStatus(HttpStatus.NOT_FOUND)
class OnSaleNotFoundException(message: String) : OnSaleException(message)

@ResponseStatus(HttpStatus.BAD_REQUEST)
class OnSaleBadRequestException(message: String) : OnSaleException(message)

@ResponseStatus(HttpStatus.BAD_REQUEST)
class OnSaleConflictIntegrityException(message: String) : OnSaleException(message)
```

La clase sellada OnSaleException contiene las subclases OnSaleNotFoundException, la cual se utiliza cuando no se encuentra el OnSale solicitado y establece el código HTTP 404 Not Found; OnSaleBadRequestException, se utiliza cuando se proporciona una solicitud incorrecta y establece el código HTTP 400 Bad Request; y OnSaleConflictIntegrityException, la cual se utiliza cuando se produce un conflicto de integridad en la base de datos y establece el código de estado HTTP 409 Conflict.

5.3.12.3. ProductException

```
sealed class ProductException(message: String): RuntimeException(message)

@ResponseStatus(HttpStatus.NOT_FOUND)
class ProductNotFoundException(message: String) : ProductException(message)

@ResponseStatus(HttpStatus.BAD_REQUEST)
class ProductBadRequestException(message: String) : ProductException(message)

@ResponseStatus(HttpStatus.BAD_REQUEST)
class ProductConflictIntegrityException(message: String) : ProductException(message)
```

La clase sellada ProductException contiene las subclases ProductNotFoundException, la cual se utiliza cuando no se encuentra el Product solicitado y establece el código HTTP 404 Not Found; ProductRequestException, se utiliza cuando se proporciona una solicitud incorrecta y establece el código HTTP 400 Bad Request; y ProductConflictIntegrityException, la cual se utiliza cuando se produce un conflicto de integridad en la base de datos y establece el código de estado HTTP 409 Conflict.

5.3.12.4. ServiceException

```
sealed class ServiceException(message: String): RuntimeException(message)

@ResponseStatus(HttpStatus.NOT_FOUND)
class ServiceNotFoundException(message: String) : ServiceException(message)

@ResponseStatus(HttpStatus.BAD_REQUEST)
class ServiceBadRequestException(message: String) : ServiceException(message)

@ResponseStatus(HttpStatus.BAD_REQUEST)
class ServiceConflictIntegrityException(message: String) : ServiceException(message)
```

La clase sellada ServiceException contiene las subclases ServiceNotFoundException, la cual se utiliza cuando no se encuentra el Service solicitado y establece el código HTTP 404 Not Found; ServiceRequestException, se utiliza cuando se proporciona una solicitud incorrecta y establece el código HTTP 400 Bad Request; y ServiceConflictIntegrityException, la cual se utiliza cuando se produce un conflicto de integridad en la base de datos y establece el código de estado HTTP 409 Conflict.

5.3.12.5. StorageException

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

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

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

La clase sellada StorageException contiene las subclases StorageNotFoundException, la cual se utiliza cuando no se encuentra elStorage solicitado y establece el código HTTP 404 Not Found; StorageRequestException, se utiliza cuando se proporciona una solicitud incorrecta y establece el código HTTP 400 Bad Request.

5.3.13. Utils: PropertiesReader

```
class PropertiesReader(private val fileName: String) {  
    private val properties = Properties()  
  
    init {  
        val file = this::class.java.classLoader.getResourceAsStream(fileName)  
        if (file != null) {  
            properties.load(file)  
        } else {  
            throw FileNotFoundException("File: $fileName not found")  
        }  
    }  
  
    fun getProperty(key: String): String {  
        val value = properties.getProperty(key)  
        if (value != null) {  
            return value  
        } else {  
            throw FileNotFoundException("Property: $key not found in file: $fileName")  
        }  
    }  
}
```

La clase PropertiesReader carga un archivo de propiedades y lo almacena en un objeto Properties. Si el archivo no se puede encontrar, se lanza una excepción FileNotFoundException. La función getProperty() toma una clave de propiedad como parámetro y devuelve el valor correspondiente. Si la propiedad no se encuentra en el archivo, se lanza una excepción FileNotFoundException.

5.3.14. Controllers

5.3.14.1. ProductsServicesController

```
@RestController
@RequestMapping("/products&services")
class ProductsServicesController {
    @Autowired constructor(
        private val productService: ProductsService,
        private val appointmentService: AppointmentService,
        private val servicesService: ServicesService
    ) {

        @GetMapping("/list")
        suspend fun findAll(): ResponseEntity<MutableList<OnSaleDTO>> {
            logger.info {"On sale controller - findAll() "}

            val resProducts = productService.findAll().toList()
                .map { it.toDTO().toOnSaleDTO() }

            val resServices = servicesService.findAll().toList()
                .map { it.toDTO().toOnSaleDTO() }

            val res : MutableList<OnSaleDTO> = mutableListOf()

            resProducts.forEach { it.OnSaleDTO
                res.add(it)
            }

            resServices.forEach { it.OnSaleDTO
                res.add(it)
            }

            return ResponseEntity.ok(res)
        }

        @GetMapping("/appointments")
        suspend fun findAllAppointments(): ResponseEntity<List<AppointmentDTO>> {
            logger.info {"On sale controller - findAll appointments() "}

            val res = appointmentService.findAll().toList()
                .map { it.toDTO() }

            return ResponseEntity.ok(res)
        }
    }
}
```

```

@GetMapping("/{id}")
suspend fun findById(@PathVariable id: Long): ResponseEntity<MutableList<OnSaleDTO>> {
    logger.info { "On sale controller - findById(): $id" }

    val res : MutableList<OnSaleDTO> = mutableListOf()

    val product: OnSaleDTO? = try {
        productsService.findById(id).toDTO().toOnSaleDTO()
    } catch (e: ProductNotFoundException) {
        null
    }

    if (product != null) { res.add(product) }

    val service: OnSaleDTO? = try {
        servicesService.findById(id).toDTO().toOnSaleDTO()
    } catch (e: OnSaleNotFoundException) {
        null
    }

    if (service != null) { res.add(service) }
    if (res.isEmpty()) {
        throw ResponseStatusException(HttpStatus.NOT_FOUND, "Nothing found for id: $id")
    }
}

return ResponseEntity.ok(res)
}

@GetMapping("/appointments/{id}")
suspend fun findAppointmentById(@PathVariable id: Long): ResponseEntity<AppointmentDTO> {

    logger.info { "On sale controller - findAppointmentById(): $id" }

    try {
        val entity = appointmentService.findById(id)
        val result = entity.toDTO()
        return ResponseEntity.ok(result)
    } catch (e: AppointmentNotFoundException) {
        throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
    }
}

@PostMapping("")
suspend fun create(@Valid @RequestBody entityDto: OnSaleCreateDTO): ResponseEntity<OnSaleDTO> {

    logger.info { "On sale controller - OnSale create()"}

    if(entityDto.type.value == "PRODUCT"){
        try{
            val rep = entityDto.productEntity?.validate()?.toModel(UUID.randomUUID())
            val res = rep?.let { productsService.save(it).toDTO().toOnSaleDTO() }
            return ResponseEntity.status(HttpStatus.CREATED).body(res)

        } catch (e: ProductBadRequestException) {
            throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
        }
    }else{
        try {
            val rep = entityDto.serviceEntity?.validate()?.toModel(UUID.randomUUID())
            val res = rep?.let { servicesService.save(it).toDTO().toOnSaleDTO() }

            return ResponseEntity.status(HttpStatus.CREATED).body(res)
        } catch (e: ServiceBadRequestException) {
            throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
        }
    }
}

```

```

@PostMapping("appointments")
suspend fun create(@Valid @RequestBody entityDto: AppointmentCreateDTO): ResponseEntity<AppointmentDTO> {
    logger.info { "On sale controller - appointment create()" }

    try{
        val rep = entityDto.validate().toModel(UUID.randomUUID())
        val res = appointmentService.save(rep).toDTO()

        return ResponseEntity.status(HttpStatus.CREATED).body(res)

    } catch (e: ProductBadRequestException) {
        throw ResponseStatusException(HttpStatus.BAD_REQUEST, e.message)
    }
}

@PutMapping("{uuid}")
suspend fun update(@PathVariable uuid: UUID, @Valid @RequestBody entityDto: OnSaleCreateDTO): ResponseEntity<OnSaleDTO> {

    logger.info { "On sale controller - OnSale update() : $uuid" }

    if(entityDto.type.value == "PRODUCT"){
        try {
            val rep = entityDto.productEntity?.validate()?.toModel(uuid)
            val res = rep?.let { productsService.update(it).toDTO().toOnSaleDTO() }

            return ResponseEntity.status(HttpStatus.OK).body(res)

        } catch (e: ProductNotFoundException) {
            throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
        }
    }else{
        try {
            val rep = entityDto.serviceEntity?.validate()?.toModel(uuid)
            val res = rep?.let { servicesService.update(it).toDTO().toOnSaleDTO() }

            return ResponseEntity.status(HttpStatus.OK).body(res)

        } catch (e: ServiceNotFoundException) {
            throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
        }
    }
}

@PutMapping("appointments/{uuid}")
suspend fun update(
    @PathVariable uuid: UUID,
    @Valid @RequestBody entityDTO: AppointmentCreateDTO): ResponseEntity<AppointmentDTO> {

    logger.info { "On sale controller - appointment update() : $uuid" }

    try {
        val rep = entityDTO.validate().toModel(uuid)
        val res = appointmentService.update(rep).toDTO()

        return ResponseEntity.status(HttpStatus.OK).body(res)

    } catch (e: ServiceNotFoundException) {
        throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
    }
}

```

```

@DeleteMapping("/{uuid}")
suspend fun delete(@PathVariable uuid: UUID): ResponseEntity<OnSaleDTO> {

    logger.info { "On sale controller - delete() : $uuid" }

    try {
        productService.delete(productService.findByUuid(uuid).toDTO().toOnSaleDTO())
    } catch (e: ProductNotFoundException) {
        try {
            servicesService.delete(servicesService.findByUuid(uuid).toDTO().toOnSaleDTO())
        } catch (e: ServiceNotFoundException) {
            throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
        }
    }

    return ResponseEntity.noContent().build()
}

@DeleteMapping("/appointments/{uuid}")
suspend fun deleteAppointment(@PathVariable uuid: UUID): ResponseEntity<AppointmentDTO> {

    logger.info { "On sale controller - deleteAppointment() : $uuid" }

    try {
        appointmentService.delete(appointmentService.findByUuid(uuid))
        return ResponseEntity.noContent().build()
    } catch (e: AppointmentNotFoundException) {
        throw ResponseStatusException(HttpStatus.NOT_FOUND, e.message)
    }
}

```

Esta clase es un controlador de Spring ya que se usa la anotación “@RestController”, que maneja las solicitudes Http relacionadas con la gestión de productos y servicios mediante las rutas “@Mapping”. Los métodos que contiene son findAll(), el cual devuelve una lista con todos los productos y servicios; findAllAppointments(), el cual devuelve una lista con todos los Appointments; findById() sirve para obtener un producto o servicio mediante su ID; findAppointmentsById(), con lo que se obtiene un Appointment mediante su id; create() sirve para crear un nuevo producto, servicio y appointment; update() maneja una solicitud para actualizar un producto, servicio o appointment existente; y para finalizar, delete(), el cual elimina un producto, servicio o appointment.

5.3.14.2. StorageController

```
@RestController
@RequestMapping(value = ["/products&services/storage"])
class StorageController {
    @Autowired constructor(private val storageService: StorageService) {
        @GetMapping(value = ["/{filename:.+}"])
        @ResponseBody
        fun serverFile(@PathVariable filename: String?, request: HttpServletRequest): ResponseEntity<Resource> = runBlocking {
            logger.info { "Searching for file: $filename" }

            val coScope = CoroutineScope(Dispatchers.IO)

            val file: Resource =
                withContext(coScope.coroutineContext) { storageService.loadAsResource(filename.toString()) }

            var contentType: String? = try {
                request.servletContext.getMimeType(file.file.getAbsolutePath)
            } catch (ex: IOException) {
                throw StorageBadRequestException("Incorrect file extension.", ex)
            }

            if (contentType == null) {
                contentType = "application/octet-stream"
            }

            return@runBlocking ResponseEntity.ok()
                .contentType(MediaType.parseMediaType(contentType))
                .body<Resource?>(file)
        }
    }

    @PostMapping(value = [""], consumes = [MediaType.MULTIPART_FORM_DATA_VALUE])
    fun uploadFile(@RequestPart("file") file: MultipartFile): ResponseEntity<Map<String, String>> = runBlocking {
        logger.info { "Saving file: ${file.originalFilename}" }

        return@runBlocking try {
            if (!file.isEmpty) {
                val coScope = CoroutineScope(Dispatchers.IO)

                val fileStored = withContext(coScope.coroutineContext) { thisCoroutineScope
                    storageService.store(file)
                }

                val urlStored = storageService.getUrl(fileStored)
                val response = mapOf("url" to urlStored, "name" to fileStored, "created_at" to LocalDateTime.now().toString())
                ResponseEntity.status(HttpStatus.CREATED).body(response)
            } else {
                throw StorageBadRequestException("Cannot save an empty file")
            }
        } catch (e: StorageException) {
            throw StorageBadRequestException(e.message.toString())
        }
    }

    @DeleteMapping(value = ["/{filename:.+}"])
    @ResponseBody
    fun deleteFile(@PathVariable filename: String?): ResponseEntity<Resource> = runBlocking {
        logger.info { "Deleting file: $filename" }

        try {
            CoroutineScope(Dispatchers.IO).launch { thisCoroutineScope
                storageService.delete(filename.toString())
            }.join()

            return@runBlocking ResponseEntity.ok().build()
        } catch (e: StorageException) {
            throw StorageBadRequestException(e.message.toString())
        }
    }
}
```

Esta clase es un controlador de Spring, ya que se usa la anotación “@RestController”, que maneja las solicitudes Http relacionadas con la gestión de productos y servicios mediante las rutas “@Mapping”. Los métodos que contiene son serverFile(), con el cual se coge un nombre de archivo y se devuelve un resource con el nombre del archivo; uploadFile(), con el que se puede cargar un archivo en el servidor; y deleteFile(), con lo que se puede eliminar un archivo del servidor tomando el nombre.

5.3.15. La clase principal: MicroservicioProductoServiciosApplication

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cache.annotation.EnableCaching

@SpringBootApplication
@EnableCaching
class MicroservicioProductoServiciosApplication

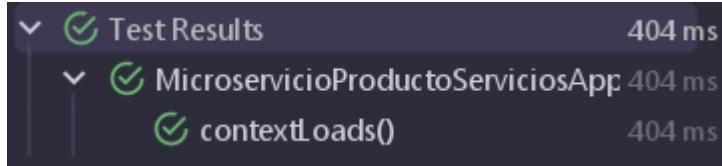
fun main(args: Array<String>) {
    runApplication<MicroservicioProductoServiciosApplication>(*args)
}
```

Aquí se añaden las anotaciones de Spring “@SpringBootApplication”, indicando que la clase es la clase principal de la aplicación; y “@EnableCaching”, para habilitar el soporte de caché de Spring. La función runApplication es la función que permite cargar y ejecutar la aplicación.

5.3.16. Tests

5.3.16.1. La clase principal de los tests:

MicroservicioProductoServiciosApplicationTests



```
@SpringBootTest
class MicroservicioProductoServiciosApplicationTests {

    @Test
    fun contextLoads() {
    }

}
```

En esta clase, añadiendo las anotaciones “@SpringBootTest” designamos que los tests usarán un contexto de Spring para su ejecución. La función contextLoads() se utiliza para comprobar que Spring se carga correctamente.

5.3.16.2. ProductsServicesControllerTest

ProductsServicesControllerTest	917 ms
updateProduct()	542 ms
deleteAppointment()	22 ms
findAppointmentById()	21 ms
findAll()	116 ms
createAppointment()	13 ms
findById()	18 ms
deleteProduct()	11 ms
findByIdNoExiste()	20 ms
createService()	12 ms
deleteProductNoExiste()	12 ms
deleteAppointmentNoExiste()	16 ms
updateService()	15 ms
updateAppointmentNoExiste()	11 ms
updateNoExiste()	17 ms
deleteServiceNoExiste()	13 ms
updateAppointment()	8 ms
deleteService()	15 ms
createProduct()	9 ms
findAllAppointments()	13 ms
findAppointmentByIdNoExiste()	13 ms

```
@SpringBootTest
internal class ProductsServicesControllerTest {

    private val product = Product(
        id = 1L,
        uuid = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d12"),
        image = "test",
        brand = "test",
        model = "test",
        description = "test",
        price = 0.0f,
        discountPercentage = 0.0f,
        stock = StockType.SOLDOUT,
        isAvailable = false,
        type = ProductType.ACCESSIONS
    )

    private val appointment = Appointment(
        id = 1L,
        uuid = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d14"),
        userId = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d13"),
        assistance = AssistanceType.ANY,
        date = LocalDate.parse("2023-03-03"),
        description = "test"
    )
}
```

```
private val service = Service(
    id = 1L,
    uuid = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d15"),
    image = "test",
    price = 0.0f,
    appointment = appointment.uuid,
    type = ServiceType.REVISION
)

@MockK
private lateinit var productService: ProductsService

@MockK
private lateinit var appointmentService: AppointmentService

@MockK
private lateinit var serviceService: ServicesService

@InjectMocks
lateinit var controller: ProductsServicesController

init {
    MockKAnnotations.init(...obj: this)
}
```

Esta clase contiene los tests del controller ProductsServicesController utilizando MockK. Se comienza creando un Appointment, Service y Product simulados con datos de ejemplo y añadiendo las dependencias de los servicios, los cuales también se simulan. El método init inicializa los objetos simulados para que en los tests se utilicen esos en vez de los reales.

Después de esto comienzan las funciones de los tests (omitidos en las capturas porque llegan a las 600 líneas de código), las cuales implementan el uso de coEvery, donde se especifica el comportamiento de los servicios simulados; assertAll, para asegurarnos de que

el resultado es correcto; y coVerify, para verificar que se invocaron los servicios simulados especificados.

- **findAll()**: en los coEvery se devuelve un flow con los Product encontrados, otro para Service y un objeto Appointment.
- **findAllAppointments()**: utilizando un coEvery se devuelve un flow de Appointment.
- **findById()**: en los coEvery se devuelve el Product y Service encontrado
- **findByIdNoExiste()**: en los coEvery se busca un Id que no existe y se comprueba que lo que se devuelve es un ProductNotFoundException y un ServiceNotFoundException
- **findAppointmentById()**: en el coEvery se devuelve el Appointment encontrado
- **findAppointmentByIdNoExiste()**: en el coEvery se busca un Id que no existe y se comprueba que lo que devuelve es un AppointmentNotFoundException
- **createProduct()**, y **createService()**: en los coEvery se utiliza una función save() donde se devuelve un product y service, respectivamente. Utilizando el método create() se crea un ProductCreateDTO y un ServiceCreateDTO, convirtiéndolos en OnSaleCreateDTO
- **createAppointment()**: en el coEvery se utiliza la función save() donde se devuelve un Appointment. Utilizando el método create() se crea un AppointmentCreateDTO
- **updateProduct()** y **updateService()**: los coEvery buscan por uuid productos y servicios y los actualizan. Luego, se crea un ServiceCreateDTO y un ProductCreateDTO, los cuales se convierten en OnSaleCreateDTO.
- **updateAppointment()**: en el coEvery se actualiza un Appointment y se crea un AppointmentCreateDTO.
- **updateNoExiste()**: en los coEvery se intenta buscar por Id un Product y un Service y se verifica que se lanza un ProductNotFoundException y un ServiceNotFoundException. Se crea un ProductCreateDTO y se convierte en un OnSaleCreateDTO.
- **updateAppointmentNoExiste()**: en el coEvery se intenta actualizar un Appointment y se verifica que se lanza un AppointmentNotFoundException.
- **deleteProduct()**, **deleteService()** y **deleteAppointment()**: se busca por uuid su respectivo objeto y se borra. Luego, se confirma que el resultado devuelve un NO_CONTENT
- deleteProductNoExiste(), deleteServiceNoExiste(),
- **deleteAppointmentNoExiste()**, **deleteServiceNoExiste()**, **deleteProductNoExiste()**: se busca por uuid cada objeto y se comprueba que se lanza un AppointmentNotFoundException, ServiceNotFoundException y ProductNotFoundException.

5.3.16.3. AppointmentsCachedRepositoryTest

AppointmentsCachedRepositoryTest	
updateNotFound()	414 ms
delete()	112 ms
findAll()	35 ms
update()	16 ms
findById()	17 ms
findByUuid()	17 ms
save()	16 ms
findByIdNotFound()	13 ms
findByUuidNotFound()	11 ms

```

@ExtendWith(MockKExtension::class)
@SpringBootTest
internal class AppointmentsCachedRepositoryTest{

    private val appointment = Appointment(
        id = 1L,
        uuid = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d12"),
        userId = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d13"),
        assistance = AssistanceType.ANY,
        date = LocalDate.parse("2023-03-03"),
        description = "test"
    )

    @MockK
    lateinit var repository: AppointmentsRepository

    @InjectMocks
    lateinit var cachedRepository: AppointmentsCachedRepository

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

```

Esta clase contiene los tests de AppointmentsCachedRepositoryTest utilizando MockK. Se comienza creando un Appointment simulado con datos de ejemplo y añadiendo las dependencias de los repositorios, los cuales también se simulan.

Después de esto comienzan las funciones de los tests , las cuales implementan el uso de coEvery, donde se especifica el comportamiento de los servicios simulados; assertAll, para asegurarnos de que el resultado es correcto; y coVerify, para verificar que se invocaron los servicios simulados especificados.

- **findAll()**: en el coEvery devuelve un flow de Appointment con todos los encontrados.
- **findById()**: se devuelve el appointment encontrado con ese Id
- **findByIdNotFound()**: se comprueba que devuelve null
- **findByUuid()**: busca en el repositorio cacheado el uuid del appointment
- **findByUuidNotFound()**: comprueba que al buscar por uuid el resultado es null
- **save()**: comprueba que en el repositorio cacheado se guarda el appointment
- **update()**: busca en el repositorio la uuid del appointment y luego comprueba que se actualiza
- **updateNotFound()**: comprueba que al buscar por uuid e intentar actualizar el resultado es null
- **delete()**: se busca por id el appointment y comprueba que se devuelve unit al borrarlo

5.3.16.4. ProductsCachedRepositoryTest

ProductsCachedRepositoryTest	708 ms
updateNotFound()	449 ms
delete()	144 ms
findAll()	32 ms
update()	19 ms
findById()	17 ms
findByUuid()	17 ms
save()	10 ms
findByIdNotFound()	10 ms
findByUuidNotFound()	10 ms

```
@ExtendWith(MockKExtension::class)
@SpringBootTest
internal class ProductsCachedRepositoryTest {

    private val product = Product(
        id = 1L,
        uuid = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d12"),
        image = "test",
        brand = "test",
        model = "test",
        description = "test",
        price = 0.0f,
        discountPercentage = 0.0f,
        stock = StockType.SOLDOUT,
        isAvailable = false,
        type = ProductType.ACCESSORIES
    )

    @MockK
    lateinit var repository: ProductsRepository

    @InjectMocks
    lateinit var cachedRepository: ProductsCachedRepository

    init {
        MockKAnnotations.init(this)
    }
}
```

Esta clase contiene los tests de ProductsCachedRepositoryTest utilizando MockK. Se comienza creando un Products simulado con datos de ejemplo y añadiendo las dependencias de los repositorios, los cuales también se simulan.

Después de esto comienzan las funciones de los tests, las cuales implementan el uso de coEvery, donde se especifica el comportamiento de los servicios simulados; assertAll, para asegurarnos de que el resultado es correcto; y coVerify, para verificar que se invocaron los servicios simulados especificados.

- **findAll()**: en el coEvery devuelve un flow de Products con todos los encontrados.
- **findById()**: se devuelve el Product encontrado con ese Id

- **findByIdNotFound()**: se comprueba que devuelve null
- **findByIdUuid()**: busca en el repositorio cacheado el uuid del Product
- **findByIdUuidNotFound()**: comprueba que al buscar por uuid el resultado es null
- **save()**: comprueba que en el repositorio cacheado se guarda el Product
- **update()**: busca en el repositorio la uuid del Product y luego comprueba que se actualiza
- **updateNotFound()**: comprueba que al buscar por uuid e intentar actualizar el resultado es null
- **delete()**: se busca por id el Product y comprueba que se devuelve unit al borrarlo

5.3.16.5. AppointmentServiceTest

 Tests passed: 9 of 9 tests – 674 ms

```
@ExtendWith(MockKExtension::class)
@SpringBootTest
internal class AppointmentServiceTest {

    private val appointment = Appointment(
        id = 1L,
        uuid = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d12"),
        userId = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d13"),
        assistance = AssistanceType.ANY,
        date = LocalDate.parse("2023-03-03"),
        description = "test"
    )

    @MockK
    lateinit var repository: AppointmentsCachedRepository

    @InjectMocks
    lateinit var service: AppointmentService

    init {
        MockKAnnotations.init(this)
    }
}
```

Esta clase contiene los tests de AppointmentServiceTest utilizando MockK. Se comienza creando un Appointment simulado con datos de ejemplo y añadiendo las dependencias del servicio AppointmentService, el cual se simula.

Después de esto comienzan las funciones de los tests, las cuales implementan el uso de coEvery, donde se especifica el comportamiento de los servicios simulados; assertAll, para asegurarnos de que el resultado es correcto; y coVerify, para verificar que se invocaron los servicios simulados especificados.

- **findAll()**: en el coEvery devuelve un flow de Appointments con todos los encontrados.
- **findById()**: se devuelve el Appointment encontrado con ese Id
- **findByIdNotFound()**: se comprueba que se lanza un AppointmentNotFoundException
- **findByIdUuid()**: busca en el servicio el uuid del Appointment
- **findByIdUuidNotFound()**: comprueba que al buscar por uuid se lanza un AppointmentNotFoundException
- **save()**: comprueba que en el servicio se guarda el Appointment
- **update()**: busca en el repositorio la uuid del Appointment y luego comprueba que se actualiza
- **updateNotFound()**: comprueba que al buscar por uuid e intentar actualizar se lanza un AppointmentNotFoundException.

- **delete()**: se busca por id el Appointment y comprueba que se devuelve false al borrarlo

5.3.16.6. ProductsServiceTest

ProductsServiceTest	717 ms
updateNotFound()	483 ms
delete()	25 ms
findAll()	110 ms
update()	11 ms
findById()	12 ms
findByUuid()	37 ms
save()	13 ms
findByIdNotFound()	12 ms
findByUuidNotFound()	14 ms

```
@ExtendWith(MockKExtension::class)
@SpringBootTest
internal class ProductsServiceTest {

    private val product = Product(
        id = 1L,
        uuid = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d12"),
        image = "test",
        brand = "test",
        model = "test",
        description = "test",
        price = 0.0f,
        discountPercentage = 0.0f,
        stock = StockType.SOLDOUT,
        isAvailable = false,
        type = ProductType.ACCESESORIES
    )

    @MockK
    lateinit var repository: ProductsCachedRepository

    @InjectMocks
    lateinit var service: ProductsService

    init {
        MockKAnnotations.init(...obj: this)
    }
}
```

Esta clase contiene los tests de ProductsServiceTest utilizando MockK. Se comienza creando un Appointment simulado con datos de ejemplo y añadiendo las dependencias del servicio ProductsService, el cual se simula.

Después de esto comienzan las funciones de los tests, las cuales implementan el uso de coEvery, donde se especifica el comportamiento de los servicios simulados; assertAll, para asegurarnos de que el resultado es correcto; y coVerify, para verificar que se invocaron los servicios simulados especificados.

- **findAll()**: en el coEvery devuelve un flow de Product con todos los encontrados.
- **findById()**: se devuelve el Product encontrado con ese Id
- **findByIdNotFound()**: se comprueba que se lanza un ProductNotFoundException
- **findByUuid()**: busca en el servicio el uuid del Product
- **findByUuidNotFound()**: comprueba que al buscar por uuid se lanza un ProductNotFoundException
- **save()**: comprueba que en el servicio se guarda el Product
- **update()**: busca en el repositorio la uuid del Product y luego comprueba que se actualiza
- **updateNotFound()**: comprueba que al buscar por uuid e intentar actualizar se lanza un ProductNotFoundException.
- **delete()**: se busca por id el Product y comprueba que se devuelve false al borrarlo

5.3.16.7. ServicesServiceTest

	ServicesServiceTest	640 ms
	updateNotFound()	376 ms
	delete()	56 ms
	findAll()	12 ms
	update()	83 ms
	findById()	14 ms
	findByUuid()	38 ms
	save()	16 ms
	findByIdNotFound()	20 ms
	saveAppointmentNotExist()	12 ms
	findByUuidNotFound()	13 ms

```
@SpringBootTest
internal class ServicesServiceTest {

    private val appointment = Appointment(
        id = 1L,
        uuid = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d12"),
        userId = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d13"),
        assistance = AssistanceType.ANY,
        date = LocalDate.parse("2023-03-03"),
        description = "test"
    )

    private val entity = Service(
        id = 1L,
        uuid = UUID.fromString("e2613745-ad51-460f-8aa2-733379be7d12"),
        image = "test",
        price = 0.0f,
        appointment = appointment.uuid,
        type = ServiceType.REVISION
    )

    @MockK
    lateinit var repository: ServiceRepository

    @MockK
    lateinit var appointmentRepository: AppointmentsCachedRepository

    @InjectMockKs
    lateinit var service: ServicesService

    init {
        MockAnnotations.init(this)
    }
}
```

Esta clase contiene los tests de ServicesServiceTest utilizando MockK. Se comienza creando un Appointment y un Service (llamado entity para evitar confusiones) simulado con

datos de ejemplo y añadiendo las dependencias del servicio ServicesService, ServiceRepository y AppointmentsCachedRepository, los cuales se simulan.

Después de esto comienzan las funciones de los tests, las cuales implementan el uso de coEvery, donde se especifica el comportamiento de los servicios simulados; assertAll, para asegurarnos de que el resultado es correcto; y coVerify, para verificar que se invocaron los servicios simulados especificados.

- **findAll()**: en el coEvery devuelve un flow de entity con todos los encontrados.
- **findById()**: se devuelve el entity encontrado con ese Id
- **findByIdNotFound()**: se comprueba que se lanza un ServiceNotFoundException
- **findByUuid()**: busca en el servicio el uuid del entity y lo devuelve como flow
- **findByUuidNotFound()**: comprueba que al buscar por uuid se lanza un ServiceNotFoundException
- **save()**: comprueba que en el servicio se guarda el entity
- **saveAppointmentNotExist()**: comprueba que se lanza un AppointmentNotFoundException al intentar guardar un Appointment que no existe
- **update()**: busca en el repositorio la uuid del Appointment y luego comprueba que se actualiza el entity
- **updateNotFound()**: comprueba que al buscar por uuid e intentar actualizar se lanza un ServiceNotFoundException.
- **delete()**: se busca por uuid el entity y comprueba que se devuelve Unit al borrarlo

5.4. Microservicio D ()

5.4.1. MODELOS

```
@Serializable
data class Order(
    @BsonId @Contextual
    val id: Id<Order> = newId(),
    @Serializable(with = UuidSerializer::class)
    val uuid: UUID = UUID.randomUUID(),
    @Contextual
    val status: StatusOrder,
    val total: Double,
    val iva: Double,
    val orderLine: List<String>,
    @Serializable(with = UuidSerializer::class)
    val cliente: UUID
) {
    enum class StatusOrder(statusOrder: String) {
        IN_PROGRESS(statusOrder: "IN_PROGRESS"),
        FINISHED(statusOrder: "FINISHED"),
        DELIVERED(statusOrder: "DELIVERED");

        companion object {
            fun from(statusOrder: String): StatusOrder {
                return when (statusOrder.uppercase()) {
                    "IN_PROGRESS" -> IN_PROGRESS
                    "FINISHED" -> FINISHED
                    "DELIVERED" -> DELIVERED
                    else -> throw IllegalArgumentException("Nodos no válidos")
                }
            }
        }
    }
}
```

El modelo Order representa los pedidos en el sistema. Tiene los siguientes atributos:

- id: un identificador único generado por el sistema.
- uuid: un identificador único generado mediante el uso de la clase UUID.
- status: el estado actual del pedido, representado por la enumeración StatusOrder.
- total: el total del pedido, en formato Double.
- iva: el impuesto al valor agregado aplicado al pedido, en formato Double.
- orderLine: una lista de identificadores de línea de pedido asociados a este pedido.
- cliente: el identificador único del cliente que realizó el pedido, generado mediante el uso de la clase UUID.

La clase también tiene una enumeración StatusOrder que define los posibles estados que puede tener un pedido (IN_PROGRESS, FINISHED y DELIVERED).

Este modelo se ha implementado utilizando la biblioteca de serialización de Kotlin.

```
@Serializable  
data class OrderLine(  
    @BsonId @Contextual  
    val id: Id<OrderLine> = newId(),  
    @Serializable(with = UUIDSerializer::class)  
    val uuid: UUID = UUID.randomUUID(),  
    @Serializable(with = UUIDSerializer::class)  
    val product: UUID,  
    val amount: Int,  
    val price: Double,  
    val total: Double,  
    @Serializable(with = UUIDSerializer::class)  
    val employee: UUID  
)
```

Este modelo representa una línea de venta de un pedido, donde se registran los detalles de un producto o servicio que fue vendido.

- id: identificador único de la línea de venta, generado automáticamente.
- uuid: identificador único de la línea de venta, generado aleatoriamente.
- product: identificador único del producto o servicio que fue vendido.
- amount: cantidad de productos o servicios vendidos en esta línea de venta.
- price: precio unitario del producto o servicio.
- total: precio total de la línea de venta (cantidad x precio unitario).
- employee: identificador único del empleado que realizó la venta.

Este modelo se utiliza en conjunto con el modelo Order, ya que un pedido puede tener una o varias líneas de venta, donde cada línea representa un producto o servicio vendido.

5.4.2. MAPPER

```
fun OrderDTO.toEntity(): Order{
    return Order(
        id = ObjectId(this.id).toId(),
        uuid = UUID.fromString(this.uuid),
        status = Order.StatusOrder.from(status),
        total = this.total,
        iva = this.iva,
        orderLine = this.orderLine,
        cliente = UUID.fromString(this.cliente),
    )
}

fun Order.toDTO(): OrderDTO{
    return OrderDTO(
        id = id.toString(),
        uuid = uuid.toString(),
        status = status.name,
        total = this.total,
        iva = this.iva,
        orderLine = this.orderLine,
        cliente = this.cliente.toString()
    )
}
```

Son funciones de mapeo (mapper) que convierten objetos del tipo OrderDTO a Order y viceversa.

La función OrderDTO.toEntity() recibe un objeto OrderDTO y lo convierte en un objeto Order, asignando los valores correspondientes a cada uno de los atributos.

La función Order.toDTO() realiza la operación inversa, convirtiendo un objeto Order en un objeto OrderDTO.

```
fun OrderDTO.toEntity(): Order{
    return Order(
        id = ObjectId(this.id).toId(),
        uuid = UUID.fromString(this.uuid),
        status = Order.StatusOrder.from(status),
        total = this.total,
        iva = this.iva,
        orderLine = this.orderLine,
        cliente = UUID.fromString(this.cliente),
    )
}

fun Order.toDTO(): OrderDTO{
    return OrderDTO(
        id = id.toString(),
        uuid = uuid.toString(),
        status = status.name,
        total = this.total,
        iva = this.iva,
        orderLine = this.orderLine,
        cliente = this.cliente.toString()
    )
}
```

Son funciones de mapeo (mapper) que convierten objetos del tipo OrderLineDTO a OrderLine y viceversa.

La función OrderLineDTO.toEntity() recibe un objeto OrderLineDTO y lo convierte en un objeto OrderLine, asignando los valores correspondientes a cada uno de los atributos.

La función OrderLine.toDTO() realiza la operación inversa, convirtiendo un objeto OrderLine en un objeto OrderLineDTO.

5.4.3. DTO

```
@Serializable
data class OrderDTO(
    val id: String? = newId<Order>().toString(),
    val uuid: String = UUID.randomUUID().toString(),
    val status: String,
    val total: Double,
    val iva:Double,
    val orderLine: List<String>,
    val cliente: Long
)

data class OrderUpdateDTO(
    val id: String,
    val uuid: String ,
    val status: String,
    val total: Double,
    val iva:Double,
    val orderLine: List<String>,
    val cliente: Long
)

@Serializable
data class OrderAllDTO(
    val data: List<OrderDTO>
)
```

```
@Serializable
data class OrderLineDTO(
    val id: String? = newId<OrderLine>().toString(),
    val uuid: String? = UUID.randomUUID().toString(),
    val product: String,
    val amount: Int,
    val price: Double,
    val total: Double,
    val employee: String
) { }
```

5.4.4. DB

```
object MongoDBManager {
    val properties = PropertiesReader(fileName: "application.properties")
    private lateinit var mongoDbClient: CoroutineClient
    lateinit var database: CoroutineDatabase

    private val STRING_CONNECTION = properties.getProperty("string_connection")
    private val STRING_CONNECTION_TEST = properties.getProperty("string_connection_test")

    init {
        val clientSettings = MongoClientSettings.builder()
            .applyConnectionString(ConnectionString(STRING_CONNECTION))
            .uvidRepresentation(UuidRepresentation.JAVA_LEGACY).build()
        mongoDbClient = KMongo.createClient(clientSettings).coroutine
        database = mongoDbClient.getDatabase(name: "order")
    }
}
```

Esta clase es un objeto singleton llamado MongoDBManager que se utiliza para manejar la conexión a la base de datos MongoDB.

Se hace uso de la biblioteca KMongo para crear un cliente MongoDB y obtener una referencia a la base de datos "order". La conexión a la base de datos se establece a través de una cadena de conexión proporcionada en el archivo application.properties.

5.4.5. REPOSITORIES

```
class OrderRepositoryImpl: OrderRepository {
    private val db = MongoDBManager.database
    override suspend fun findByUUID(uuid: UVID): Order {
        return db.getCollection<Order>().find().publisher.asFlow().toList().firstOrNull { it.uuid == uuid } ?: throw OrderNotFoundException("The order with uuid $uuid does not exist")
    }

    override fun findAll(): Flow<Order> {
        return db.getCollection<Order>().find().publisher.asFlow()
    }

    override suspend fun findById(id: Id<Order>): Order {
        return db.getCollection<Order>().findOneById(id) ?: throw OrderNotFoundException("The order with id $id does not exist")
    }

    override suspend fun save(entity: Order): Order {
        return db.getCollection<Order>().save(entity).let { entity }
    }

    override suspend fun delete(entity: Order): Boolean {
        val res = db.getCollection<Order>().deleteOneById(entity.id)
        if (res.deletedCount.toInt() == 1){
            return true
        }else{
            throw OrderNotFoundException("The order with id ${entity.id} does not exist")
        }
    }

    override suspend fun update(entity: Order): Order {
        val res = db.getCollection<Order>().updateOne(entity)
        if (res.modifiedCount.toInt() > 0){
            return entity
        }else{
            throw OrderNotFoundException("The order with id ${entity.id} does not exist")
        }
    }
}
```

Este es un repositorio que implementa a la interfaz OrderRepository que utiliza una base de datos MongoDB para realizar operaciones de CRUD (Crear, Leer, Actualizar y Eliminar) en

objetos de tipo Order. La implementación utiliza el cliente de MongoDB para Kotlin y ofrece una serie de funciones que permiten trabajar con la base de datos MongoDB.

Las funciones en este repositorio son las siguientes:

- findByUUID: devuelve un objeto Order según su identificador único UUID.
- findAll: devuelve todos los objetos Order de la base de datos.
- findById: recupera un objeto Order según su identificador Id.
- save: inserta un objeto Order en la base de datos.
- delete: elimina un objeto Order de la base de datos.
- update: actualiza un objeto Order en la base de datos.

```
class OrderLineRepositoryImpl: OrderLineRepository {  
    private val db = MongoDBManager.database  
    override suspend fun findByUUID(uuid: UUID): OrderLine {  
        return db.getCollection<OrderLine>().find().publisher.asFlow().toList().firstOrNull { it.uuid == uuid } ?: throw OrderLineNotFoundException("The order line with uid $uuid does not exist")  
    }  
  
    override fun findAll(): Flow<OrderLine> {  
        return db.getCollection<OrderLine>().find().publisher.asFlow()  
    }  
  
    override suspend fun findById(id: Id<OrderLine>): OrderLine {  
        return db.getCollection<OrderLine>().findOneById(id) ?: throw OrderLineNotFoundException("The order line with id $id does not exist")  
    }  
  
    override suspend fun save(entity: OrderLine): OrderLine {  
        return db.getCollection<OrderLine>().save(entity).let { entity }  
    }  
  
    override suspend fun delete(entity: OrderLine): Boolean {  
        val res = db.getCollection<OrderLine>().deleteOneById(entity.id)  
        if (res.deletedCount.toInt() == 1){  
            return true  
        }else{  
            throw OrderLineNotFoundException("The order line with id ${entity.id} does not exist")  
        }  
    }  
  
    override suspend fun update(entity: OrderLine): OrderLine {  
        val res = db.getCollection<OrderLine>().updateOne(entity)  
        if (res.modifiedCount.toInt() == 1){  
            return entity  
        }else{  
            throw OrderLineNotFoundException("The order line with id ${entity.id} does not exist")  
        }  
    }  
}
```

Este es otro repositorio que se encarga de manejar las operaciones CRUD (crear, leer, actualizar, eliminar) para la entidad OrderLine. Al igual que OrderRepositoryImpl, implementa la interfaz OrderLineRepository.

La implementación de los métodos es similar a la anterior, con la diferencia de que aquí se trabaja con la colección OrderLine en la base de datos. Por ejemplo, en el método findByUUID, se busca un objeto OrderLine en la base de datos a través del valor de su campo uuid. Si no se encuentra, se lanza una excepción personalizada OrderLineNotFoundException.

5.4.6. SERVICES

```
class OrderService {
    val orderRepository: OrderRepository
    ) {
        fun getAllOrder(): Flow<Order> {
            println("getAllOrder")
            return orderRepository.findAll()
        }

        suspend fun getOrderByUUID(uuid: UUID): Order {
            return orderRepository.findByUUID(uuid)
        }

        suspend fun saveOrder(order: Order): Order {
            return orderRepository.save(order)
        }

        suspend fun updateOrder(order: Order): Order {
            return orderRepository.update(order)
        }

        suspend fun deleteOrder(order: Order): Boolean {
            return orderRepository.delete(order)
        }
    }
}
```

La clase OrderService es un servicio que se encarga de manejar las operaciones relacionadas con la entidad Order.

Tiene una dependencia de OrderRepository que le proporciona los métodos para interactuar con la base de datos.

Los métodos disponibles en este servicio son:

- getAllOrder(): devuelve un Flow de todos los objetos Order almacenados en la base de datos.
- getOrderByUUID(uuid: UUID): devuelve un Order correspondiente al UUID proporcionado.
- saveOrder(order: Order): inserta un nuevo Order en la base de datos y devuelve el mismo Order con su ID asignado.
- updateOrder(order: Order): actualiza un Order existente en la base de datos y devuelve la orden actualizada.
- deleteOrder(order: Order): elimina un Order existente en la base de datos y devuelve un valor booleano que indica si la eliminación fue exitosa o no.

```
class OrderLineService{
    private val orderLineRepository: OrderLineRepository,
    ) {
        fun getAllOrderLine(): Flow<OrderLine> {
            return orderLineRepository.findAll()
        }

        suspend fun getOrderLineByUUID(uuid: UUID): OrderLine {
            return orderLineRepository.findByUUID(uuid)
        }

        suspend fun saveOrderLine(orderLine: OrderLine): OrderLine {
            return orderLineRepository.save(orderLine)
        }

        suspend fun updateOrderLine(orderLine: OrderLine): OrderLine {
            return orderLineRepository.update(orderLine)
        }

        suspend fun deleteOrderLine(orderLine: OrderLine): Boolean {
            return orderLineRepository.delete(orderLine)
        }
}
```

La clase OrderLineService es un servicio que se encarga de manejar las operaciones relacionadas con la entidad Order.

Tiene una dependencia de OrderLineRepository que le proporciona los métodos para interactuar con la base de datos.

Los métodos disponibles en este servicio son:

- getAllOrderLine(): devuelve un Flow de todos los objetos OrderLine almacenados en la base de datos.
- getOrderLineByUUID(uuid: UUID): devuelve un OrderLine correspondiente al UUID proporcionado.
- saveOrderLine(orderLine: OrderLine): inserta un nuevo OrderLine en la base de datos y devuelve el mismo OrderLine con su ID asignado.
- updateOrderLine(orderLine: OrderLine): actualiza un OrderLine existente en la base de datos y devuelve la orden actualizada.
- deleteOrderLine(orderLine: OrderLine): elimina un OrderLine existente en la base de datos y devuelve un valor booleano que indica si la eliminación fue exitosa o no.

```

@Single
class TokensService(
    private val tokenConfig: TokenConfig
) {
    fun verifyToken(): JWTVerifier {
        return JWT.require(Algorithm.HMAC512(tokenConfig.secret))
            .withAudience(tokenConfig.audience)
            .withIssuer(tokenConfig.issuer)
            .acceptExpiresAt(tokenConfig.expiration.toLong())
            .build()
    }
}

```

Este servicio es una clase Kotlin con una anotación @Single que define una instancia única de la clase TokensService. El constructor de la clase toma un objeto TokenConfig como parámetro, que contiene la configuración necesaria para generar y verificar los tokens JWT.

El método verifyToken() es el que se encarga de verificar la validez de un token JWT, utilizando el algoritmo HMAC512 para la firma, y el tiempo de expiración.

Este servicio es útil en un sistema que requiere autenticación y autorización basada en tokens, ya que permite verificar la validez de los tokens que son enviados en las solicitudes HTTP, y así garantizar que sólo los usuarios autenticados y autorizados tengan acceso a los recursos protegidos del sistema.

5.4.7. VALIDATOR

```

fun RequestValidationConfig.orderValidation(){
    validate<OrderDTO>{order ->
        if(order.orderLine.isEmpty()){
            ValidationResult.Invalid(reason: "The order line cannot be empty")
        } else if (order.iva < 0){
            ValidationResult.Invalid(reason: "The IVA cannot be negative")
        } else if (order.status.isEmpty()){
            ValidationResult.Invalid(reason: "The status cannot be empty")
        } else if (order.total < 0){
            ValidationResult.Invalid(reason: "The total cannot be negative")
        }else {
            ValidationResult.Valid
        }
    }
}

```

La clase Order Validator define una serie de validaciones que deben ser realizadas en las solicitudes que se reciben en una aplicación. En este caso, la función orderValidation se encarga de validar los datos de OrderDTO.

La función de validación recibe un objeto de tipo OrderDTO y retorna un objeto de tipo ValidationResult, que indica si la validación fue exitosa o si se encontró algún error. Si se encontró un error, se proporciona un mensaje que describe el problema.

La validación comprueba que el objeto OrderDTO tenga al menos una línea de orden, que el IVA sea un número no negativo, que el estado no esté vacío y que el total no sea un número negativo. Si todas las validaciones pasan, se devuelve un objeto

ValidationResult.Valid para indicar que la solicitud es válida. Si alguna validación falla, se devuelve un objeto ValidationResult.Invalid con un mensaje de error correspondiente.

```
fun RequestValidationConfig.orderLineValidation(){
    validate<OrderLineDTO>{ orderLine ->
        if(orderLine.amount < 0){
            ValidationResult.Invalid( reason: "The amount cannot be less than 0")
        } else if (orderLine.price < 0){
            ValidationResult.Invalid( reason: "The price cannot be 0 or negative")
        } else if (orderLine.total < 0){
            ValidationResult.Invalid( reason: "The total cannot be 0 or negative")
        } else{
            ValidationResult.Valid
        }
    }
}
```

La clase RequestValidationConfig es una clase que proporciona una funcionalidad de validación de datos en una aplicación. En este caso, se está definiendo una función de extensión orderLineValidation() que se utiliza para validar objetos de tipo OrderLineDTO.

La función de validación toma un objeto OrderLineDTO y verifica si el amount, price y total son válidos. Si alguno de estos valores es menor que cero o si el precio o el total son iguales a cero o negativos, la función devuelve un objeto ValidationResult.Invalid con un mensaje de error correspondiente. Si todos los valores son válidos, la función devuelve un objeto ValidationResult.Valid.

El propósito de esta función de validación es garantizar que los objetos OrderLineDTO que se utilizan en la aplicación cumplan con ciertos criterios de validación para evitar errores o comportamientos inesperados en la aplicación.

5.4.8. CONFIG

```
@Single
data class TokenConfig(
    @InjectedParam private val config: Map<String, String>
) {
    val audience = config["audience"].toString()
    val secret = config["secret"].toString()
    val issuer = config["issuer"].toString()
    val expiration = config["expiration"].toString()
    val realm = config["realm"].toString()
}
```

Esta clase TokenConfig es una clase de datos que se utiliza para configurar los tokens JWT que se utilizan para autenticar y autorizar solicitudes en una aplicación.

El constructor de esta clase toma un parámetro injectado, config, que es un mapa de cadenas clave-valor que contiene la configuración necesaria para los tokens JWT. En particular, el mapa debe contener las claves "audience", "secret", "issuer" y "expiration", que se utilizan para configurar el algoritmo, la audiencia, el emisor y el tiempo de expiración de los tokens JWT.

Estos valores se utilizan más tarde para construir el JWTVerifier en el servicio TokensService que se usa para verificar la validez de los tokens JWT.

5.4.9. PLUGINS

```
fun Application.configureRouting() {
    val orderService = OrderService(OrderRepositoryImpl())
    val orderLineService = OrderLineService(OrderLineRepositoryImpl())

    routing { this: Routing -

        get("orderline") { this: PipelineContext<Unit, ApplicationCall>
            val res = mutableListOf<OrderLineDTO>()
            orderLineService.getAllOrderLine().collect { orderLine ->
                res.add(orderLine.toDto())
            }
            call.respond(HttpStatusCode.OK, res)
        }

        get("order") { this: PipelineContext<Unit, ApplicationCall>
            val res = mutableListOf<OrderDTO>()
            orderService.getAllOrder().collect { order ->
                res.add(order.toDTO())
            }
            call.respond(
                HttpStatusCode.OK, res
            )
        }

        get("order/{id}") { this: PipelineContext<Unit, ApplicationCall>
            try {
                val id = call.parameters["id"]?.toUUID()!!
                val res = orderService.getOrderByUUID(id).toDTO()
                call.respond(HttpStatusCode.OK, res)
            } catch (e: UUIDException) {
                call.respond(HttpStatusCode.BadRequest, e.message.toString())
            }
        }
    }
}
```

```
get("orderline/{id}") { this: PipelineContext<Unit, ApplicationCall>
    try {
        val id = call.parameters["id"]?.toUUID()!!
        val res = orderLineService.getOrderLineByUUID(id).toDto()
        call.respond(HttpStatusCode.OK, res)
    } catch (e: UUIDException) {
        call.respond(HttpStatusCode.BadRequest, e.message.toString())
    }
}

post("order") { this: PipelineContext<Unit, ApplicationCall>
    try {
        val dto = call.receive<OrderDTO>()
        val order = dto.toEntity()
        val res = orderService.saveOrder(order)
        call.respond(HttpStatusCode.Created, res.toDTO())
    } catch (e: OrderNotFoundException) {
        call.respond(HttpStatusCode.NotFound, e.message.toString())
    } catch (e: RequestValidationException) {
        call.respond(HttpStatusCode.BadRequest, e.reasons)
    }
}

post("orderline") { this: PipelineContext<Unit, ApplicationCall>
    try {
        val dto = call.receive<OrderLineDTO>()
        val orderLine = dto.toEntity()
        val res = orderLineService.saveOrderLine(orderLine)
        call.respond(HttpStatusCode.Created, res.toDto())
    } catch (e: OrderLineNotFoundException){
        call.respond(HttpStatusCode.NotFound, e.message.toString())
    } catch (e: RequestValidationException){
        call.respond(HttpStatusCode.BadRequest, e.reasons)
    }
}
```

```

put("order/{id}") { this: PipelineContext<Unit, ApplicationCall>
    try {
        val id = call.parameters["id"]?.toUUID()!!
        val dto = call.receive<OrderDTO>()
        val oldOrder = orderService.getAllOrder().toList().firstOrNull { it.uuid == id }
        val orderUpdateDto = OrderUpdateDTO(
            oldOrder!!.id.toString(),
            oldOrder.uuid.toString(),
            dto.status,
            dto.total,
            dto.iva,
            dto.orderLine,
            dto.cliente
        )
        val orderUpdate = orderUpdateDto.toEntity()
        val res = orderService.updateOrder(orderUpdate)
        call.respond(HttpStatusCode.OK, res.toDTO())
    } catch (e: OrderNotFoundException){
        call.respond(HttpStatusCode.NotFound, e.message.toString())
    } catch (e: RequestValidationException){
        call.respond(HttpStatusCode.BadRequest, e.reasons)
    }
}

put("orderline/{id}") { this: PipelineContext<Unit, ApplicationCall>
    try{
        val id = call.parameters["id"]?.toUUID()!!
        val dto = call.receive<OrderLineDTO>()
        val orderLine = dto.toEntity()
        val res = orderLineService.updateOrderLine(orderLine)
        call.respond(HttpStatusCode.OK, res.toDTO())
    }catch (e: OrderLineNotFoundException){
        call.respond(HttpStatusCode.NotFound, e.message.toString())
    } catch (e: RequestValidationException){
        call.respond(HttpStatusCode.BadRequest, e.reasons)
    }
}
}

```

```

delete("order/{id}") { this: PipelineContext<Unit, ApplicationCall>
    try {
        val id = call.parameters["id"]?.toUUID()!!
        val order = orderService.getOrderByUUID(id)
        val res = orderService.deleteOrder(order)
        call.respond(HttpStatusCode.NoContent)
    } catch (e: OrderNotFoundException){
        call.respond(HttpStatusCode.NotFound, e.message.toString())
    } catch (e: RequestValidationException){
        call.respond(HttpStatusCode.BadRequest, e.reasons)
    }
}

delete("orderline/{id}") { this: PipelineContext<Unit, ApplicationCall>
    try {
        val id = call.parameters["id"]?.toUUID()!!
        val orderLine = orderLineService.getOrderByUUID(id)
        val res = orderLineService.deleteOrderLine(orderLine)
        call.respond(HttpStatusCode.NoContent)
    } catch (e: OrderLineNotFoundException){
        call.respond(HttpStatusCode.NotFound, e.message.toString())
    } catch (e: RequestValidationException){
        call.respond(HttpStatusCode.BadRequest, e.reasons)
    }
}
}

```

En esta clase, se utiliza la función routing para definir todas las rutas disponibles para la aplicación web, que son las siguientes:

- GET /orderline: Devuelve una lista de todas las líneas de pedido en formato OrderLineDTO.
- GET /order: Devuelve una lista de todos los pedidos en formato OrderDTO.

- GET /order/{id}: Devuelve el pedido con el id especificado en formato OrderDTO.
- GET /orderline/{id}: Devuelve la línea de pedido con el id especificado en formato OrderLineDTO.
- POST /order: Crea un nuevo pedido a partir de un objeto OrderDTO y devuelve el pedido creado en formato OrderDTO.
- POST /orderline: Crea una nueva línea de pedido a partir de un objeto OrderLineDTO y devuelve la línea de pedido creada en formato OrderLineDTO.
- PUT /order/{id}: Actualiza el pedido con el id especificado a partir de un objeto OrderDTO y devuelve el pedido actualizado en formato OrderDTO.
- PUT /orderline/{id}: Actualiza la línea de pedido con el id especificado a partir de un objeto OrderLineDTO y devuelve la línea de pedido actualizada en formato OrderLineDTO.
- DELETE /order/{id}: Elimina el pedido con el id especificado.
- DELETE /orderline/{id}: Elimina la línea de pedido con el id especificado.

En la función configureRouting se crean dos servicios (orderService y orderLineService) que se utilizan para manejar las operaciones relacionadas con los pedidos y las líneas de pedido. Además, se utilizan diferentes métodos respond para enviar respuestas al cliente según el resultado de las operaciones.

En las rutas que requieren un id específico (GET, PUT y DELETE), se utiliza un bloque try-catch para manejar la posible excepción UUIDException que puede ocurrir si el id especificado no es válido. En caso de que se lance esta excepción, se devuelve un mensaje de error con un código de estado BadRequest.

En las rutas POST y PUT, se utiliza también un bloque try-catch para manejar la posible excepción RequestValidationException que puede ocurrir si el objeto recibido no cumple con las reglas de validación especificadas en las clases OrderValidationConfig y OrderLineValidationConfig. En caso de que se lance esta excepción, se devuelve un mensaje de error con un código de estado BadRequest.

```

fun Application.configureSecurity() {

    val tokenConfigParams = mapOf<String, String>(
        "audience" to environment.config.property("jwt.audience").getString(),
        "secret" to environment.config.property("jwt.secret").getString(),
        "issuer" to environment.config.property("jwt.issuer").getString(),
        "realm" to environment.config.property("jwt.realm").getString()
    )

    val tokenConfig: TokenConfig = get { parametersOf(tokenConfigParams) }

    val jwtService: TokenService by inject()

    authentication { this: AuthenticationConfig -
        jwt { this: JWTAuthenticationProvider.Config -
            verifier(jwtService.verifyToken())
            realm = tokenConfig.realm
            validate { this: ApplicationCall ->
                JWTPrincipal(credential.payload)
            }

            challenge { this: JWTChallengeContext ->
                defaultScheme, realm ->
                call.respond(HttpStatusCode.Unauthorized, message = "Invalid token")
            }
        }
    }
}

```

Esta clase es una función de configuración de seguridad, en la que se utiliza JWT para autenticar y autorizar a los usuarios.

El método **configureSecurity()** es utilizado para configurar la seguridad de la aplicación utilizando autenticación basada en JWT. En este caso, se utiliza el servicio **TokensService** para verificar los tokens JWT y se valida si la audiencia y el nombre de usuario están presentes en el payload del token. Si la validación es exitosa, se crea un **JWTPrincipal**. Si la validación falla, se envía un desafío de autenticación HTTP no autorizada con el mensaje “Token no valido o expirado”.

```

fun Application.configureSerialization() {
    install(ContentNegotiation) { this: ContentNegotiationConfig -
        json()
    }

    routing { this: Routing -
        get("/json/kotlinx-serialization") { this: PipelineContext<Unit, ApplicationCall> -
            call.respond(mapOf("hello" to "world"))
        }
    }
}

```

```
fun Application.configureValidation(){
    install(RequestValidation) { this: RequestValidationConfig -
        orderValidation()
        orderLineValidation()
    }
}
```

Utilizamos el módulo **RequestValidation** para agregar validadores personalizados a las solicitudes. Estamos agregando dos validadores personalizados llamados **orderValidation** y **orderLineValidation** para validar los datos de las solicitudes de pedido y línea de venta.

Al agregar esta configuración, cualquier solicitud que no cumpla con las validaciones especificadas en los validadores personalizados se rechazará y se enviará una respuesta de error correspondiente al cliente.

5.4.10. EXCEPTIONS

```
class OrderLineException(message: String) : RuntimeException(message) {
}
```

```
class OrderException(message: String) : RuntimeException(message) {
}
```

```
class OrderLineNotFoundException(message: String): RuntimeException(message) {
}
```

```
class OrderNotFoundException(message: String) : RuntimeException(message) {
}
```

```
class UUIDException(message: String) : RuntimeException(message) {
}
```

En este caso, se definen cuatro clases: **OrderException**, **OrderLineException** y **OrderNotFoundException** y **UUIDException**. Cada una de ellas representan las excepciones que pueden ocurrir al realizar operaciones en un almacenamiento de datos.

5.4.11. SERIALIZERS

```
object UUIDSerializer : KSerializer<UUID> {
    override val descriptor = PrimitiveSerialDescriptor("UUID", PrimitiveKind.STRING)

    override fun deserialize(decoder: Decoder): UUID {
        return UUID.fromString(decoder.decodeString())
    }

    override fun serialize(encoder: Encoder, value: UUID) {
        encoder.encodeString(value.toString())
    }
}
```

En este código se implementa de una clase KSerializer que se utiliza para serializar y deserializar objetos de la clase UUID. La clase UUID representa un identificador único universal que se utiliza a menudo en aplicaciones de software.

La clase UUIDSerializer también implementa los métodos deserialize y serialize.

```
object LocalDateTimeSerializer : KSerializer<LocalDateTime> {
    override val descriptor = PrimitiveSerialDescriptor("LocalDateTime", PrimitiveKind.STRING)

    override fun deserialize(decoder: Decoder): LocalDateTime {
        return LocalDateTime.parse(decoder.decodeString())
    }

    override fun serialize(encoder: Encoder, value: LocalDateTime) {
        encoder.encodeString(value.toString())
    }
}
```

Esta es una implementación de una clase KSerializer que se utiliza para serializar y deserializar objetos de la clase LocalDateTime.

La clase LocalDateTimeSerializer implementa dos métodos, deserialize y serialize, que convierten objetos de LocalDateTime.

```
object LocalDateSerializer : KSerializer<LocalDate> {
    override val descriptor = PrimitiveSerialDescriptor("LocalDate", PrimitiveKind.STRING)

    override fun deserialize(decoder: Decoder): LocalDate {
        return LocalDate.parse(decoder.decodeString())
    }

    override fun serialize(encoder: Encoder, value: LocalDate) {
        encoder.encodeString(value.toString())
    }
}
```

Esta es una implementación de una clase KSerializer que se utiliza para serializar y deserializar objetos de la clase LocalDate.

La clase LocalDateSerializer implementa dos métodos, deserialize y serialize, que convierten objetos de LocalDate.

5.4.12. UTILS

```
fun String.toUUID(): UUID{
    return try {
        UUID.fromString( name: this)
    } catch (e: IllegalArgumentException) {
        throw UUIDException("The id is invalid or not in the UUID format")
    }
}
```

Esta es una función de extensión en Kotlin que convierte una cadena en un objeto UUID. Si la cadena no se puede convertir, se lanza una excepción personalizada que indica que la cadena no está en el formato UUID válido.

```
class PropertiesReader(private val fileName: String) {
    private val properties = Properties()

    init {
        val file = this::class.java.classLoader.getResourceAsStream(fileName)
        if (file != null) {
            properties.load(file)
        } else {
            throw FileNotFoundException("File not found: $fileName")
        }
    }

    fun getProperty(key: String): String {
        val value = properties.getProperty(key)
        if (value != null) {
            return value
        } else {
            throw FileNotFoundException("Property $key not found in file: $fileName")
        }
    }
}
```

Esta clase en resumen se encarga de leer un archivo de propiedades y permitir el acceso a los valores de las propiedades mediante el uso de claves. Si la propiedad no se puede encontrar en el archivo, lanza una excepción para indicar el error.

5.4.13. APPLICATION CONF

```
ktor {  
    deployment {  
        port = 7878  
        port = ${?PORT}  
    }  
    application {  
        modules = [ biques.dam.es.ApplicationKt.module ]  
    }  
  
    jwt {  
        secret = "BiquesDAM"  
        issuer = "BiquesUsuario"  
        audience = "DAM"  
    }  
}
```

La propiedad "deployment" establece el puerto en el que se ejecutará la aplicación, en este caso, en el puerto 7878.

La propiedad "jwt" establece la configuración para el uso de tokens JWT. Se define el secreto compartido entre el servidor y los clientes, el emisor del token (issuer) y la audiencia del token (audience). Estos parámetros se utilizan para verificar la autenticidad del token JWT en el lado del servidor.

5.4.14. TEST

The screenshot shows the IntelliJ IDEA interface with the code editor and a tool window below it.

Code Editor Content:

```
24
25 class OrderRepositoryImplTest {
26     private val db = MongoDbManager.database.database
27     val delete = db.getCollection<Order>().drop()
28     private val orderRepository = OrderRepositoryImpl()
29     private val order = Order(
30         ObjectId(hexString = "223456789912345678901232").toId(),
31         UUID.fromString(name = "4d83b14d-bdae-422a-9e41-87fbdaef9cff"),
32         Order.StatusOrder.DELIVERED,
33         total = 12.0,
34         iva = 12.0,
35         listOf(
36             "ed6f7d0a-7f7a-45bf-8b63-a1aa21383271",
37             "e213f434-4c2b-4a28-953f-3981b1ff7e00",
38         ),
39         cliente = 1
40     )
41
42     @BeforeEach
43     fun setUp(): Unit = runBlocking { thisCoroutineScope {
44         db.getCollection<Order>().drop()
45         orderRepository.save(order)
46     } }
47 }
```

Tool Window Content (Run Tab):

Test	Time	Notes
OrderRepositoryImplTest	3 sec 96 ms	> Configure project : ksp-1.8.0-1.0.8 is too old for k... ksp-1.8.0-1.0.8 is too old for k... ksp-1.8.0-1.0.8 is too old for k... ksp-1.8.0-1.0.8 is too old for k... > Task :kspKotlin > Task :processResources > Task :compileKotlin 'compileJava' task (current target) By default will become an error s... Consider using JVM toolchain: http://
updateNotExist()	1 sec 791 ms	
delete()	145 ms	
findAll()	285 ms	
update()	163 ms	
findByIdNotExist()	97 ms	
findById()	164 ms	
deleteNotExist()	88 ms	
findByIdUUID()	112 ms	
save()	156 ms	
findByIdNotExist()	95 ms	

Estos son los test de OrderRepository, estamos utilizando JUnit 5 y MongoDB como base de datos. Antes de cada prueba, se configura el entorno llamando al método `setUp()`, que borra cualquier colección existente de la base de datos y crea una instancia de `OrderRepositoryImpl` para realizar las pruebas.

Cada prueba comprueba una funcionalidad específica de la implementación de `OrderRepository`, las pruebas incluyen casos como buscar un pedido por su id, buscar todos los pedidos almacenados en el repositorio, actualizar un pedido y verificar si el pedido fue actualizado correctamente, y guardar un nuevo pedido en el repositorio y verificar que se haya guardado correctamente. También se incluyen pruebas para verificar que se manejen adecuadamente los errores cuando no se encuentra un pedido específico en el repositorio.

The screenshot shows an IDE interface with two main panes. The top pane displays the source code for `OrderLineRepositoryImplTest`. The bottom pane shows the test results for the same class.

```
26 class OrderLineRepositoryImplTest {
27     private val db = MongoDbManager.database.database
28     private val orderLineRepository = OrderLineRepositoryImpl()
29     private val linea_pedido = OrderLine(
30         ObjectId(hexString: "123456789912345678901232").toId(),
31         UUID.fromString(name: "4d83b14d-bdae-422a-9e41-87fbdaef9cff"),
32         UUID.fromString(name: "0c34f158-5b8e-4193-9ad3-f00e7be93352"),
33         amount: 3,
34         price: 5.0,
35         total: 15.0,
36         employee: 1
37     )
38
39     @BeforeEach
40     fun setUp(): Unit = runBlocking { thisCoroutineScope
41         db.getCollection<OrderLine>().drop()
42         orderLineRepository.save(linea_pedido)
43     }
44
45     @OptIn(ExperimentalCoroutinesApi::class)
46     @Test
47     fun test() {
48         // Test implementation
49     }
50 }
```

Test Results:

Test Method	Time (ms)
updateNotExist()	1 sec 408 ms
delete()	107 ms
findAll()	360 ms
update()	138 ms
findByIdNotExist()	94 ms
findById()	158 ms
deleteNotExist()	98 ms
findById()	103 ms
save()	89 ms
findByIdNotExist()	77 ms

Output of the test execution:

```
> Configure project :
ksp-1.8.0-1.0.8 is too old for
> Task :kspKotlin UP-TO-DATE
> Task :compileKotlin UP-TO-DA
> Task :compileJava NO-SOURCE
> Task :processResources UP-TO
> Task :classes UP-TO-DATE
> Task :jar UP-TO-DATE
```

Estos son los test de `OrderLineRepository` que se encarga de gestionar la línea de venta, estamos utilizando JUnit 5 y MongoDB como base de datos. Antes de cada prueba, se configura el entorno llamando al método `setUp()`, que borra cualquier colección existente de la base de datos y crea una instancia de `OrderLineRepositoryImpl` para realizar las pruebas.

Cada prueba comprueba una funcionalidad específica de la implementación de `OrderLineRepository`, las pruebas incluyen casos como buscar un línea de venta por su id, buscar todas las líneas de venta almacenadas en el repositorio, actualizar una línea de venta y verificar si la línea de venta fue actualizada correctamente, y guardar una nueva línea de venta en el repositorio y verificar que se haya guardado correctamente. También se incluyen pruebas para verificar que se manejen adecuadamente los errores cuando no se encuentra una línea de venta específico en el repositorio.

The screenshot shows the Android Studio IDE. The top part displays the `OrderServiceTest` file with code for testing the `OrderService`. The bottom part shows the `Test Results` window with a summary of 5 tests passed in 557 ms. The details of each test are listed, along with build logs for tasks like `:kspKotlin UP-TO-DATE` and `:jar UP-TO-DATE`.

```
26     @TestInstance(TestInstance.Lifecycle.PER_CLASS)
27     @ExtendWith(MockKExtension::class)
28     class OrderServiceTest {
29         private val order = Order(
30             ObjectId(hexString: "223456789912345678901232").toId(),
31             UUID.fromString(name: "4d83b14d-bdae-422a-9e41-87fbdaef9cff"),
32             Order.StatusOrder.DELIVERED,
33             total: 12.0,
34             iva: 12.0,
35             listOf(
36                 "ed0f7d0a-7f7a-45bf-8b63-a1aa21383271",
37                 "e213f434-4c2b-4a28-953f-3981b1ff7e00",
38             ),
39             cliente: 1
40         )
41
42         @MockK
43         lateinit var orderRepository: OrderRepository
44
45         @InjectMockKs
46         lateinit var service: OrderService
47
48         @OptIn(ExperimentalCoroutinesApi::class)
49         @Test
50         fun findAll() = runTest { ... }
```

Esta clase `OrderServiceTest` contiene varias funciones de prueba que prueban el comportamiento de varias funciones en la clase `OrderService`:

`findAll()`: prueba que la función `findAll()` devuelve una lista de todos los pedidos guardados en el repositorio.

`findById()`: prueba que la función `findById()` devuelve el pedido correctamente cuando se le da un UUID válido.

`save()`: prueba que la función `save()` guarda correctamente una orden en el repositorio y devuelve la orden guardada.

`update()`: prueba que la función `update()` actualiza correctamente un pedido existente en el repositorio y devuelve el pedido actualizado.

`delete()`: prueba que la función `delete()` elimina correctamente un pedido del repositorio y devuelve true, y que se lanza una excepción si se intenta acceder a la orden eliminada.

Las pruebas están implementadas utilizando JUnit 5 y las corutinas de Kotlin. Utilizan la función `runTest()` de la biblioteca `kotlinx.coroutines.test` para ejecutar las funciones de prueba en un contexto de corutina. Además, se usan las anotaciones de MockK para crear objetos falsos y las funciones de prueba `coEvery()` y `coVerify()` de MockK para verificar el comportamiento esperado de las funciones de la clase `OrderService`.

```
25     @TestInstance(TestInstance.Lifecycle.PER_CLASS)
26     @ExtendWith(MockKExtension::class)
27     class OrderLineServiceTest {
28         private val orderLine = OrderLine(
29             ObjectId(hexString = "123456789912345678901232").toId(),
30             UUID.fromString(name = "4d83b14d-bdae-422a-9e41-87fbdaef9cff"),
31             UUID.fromString(name = "0c34f158-5b8e-4193-9ad3-f00e7be93352"),
32             amount = 3,
33             price = 5.0,
34             total = 15.0,
35             employee = 1
36         )
37         @MockK
38         lateinit var orderLineRepository: OrderLineRepository
39         @InjectMockks
40         lateinit var service: OrderLineService
41
42         @Test
43         fun findAll() = runTest { this@TestScope
44             coEvery { orderLineRepository.findAll() } returns flowOf(orderLine)
45             val result = service.getAllOrderLine().toList()
46             assertEquals(1, result.size)
47         }
48     }
49
50     Run: OrderLineServiceTest x
51
52     Tests Results
53     └─ OrderLineServiceTest
54         └─ findAll() (607 ms)
55             > Configure project : 607 ms
56             ksp-1.8.0-1.0.8 is too old for kotlin-1.8.20
57             ksp-1.8.0-1.0.8 is too old for kotlin-1.8.20
58             ksp-1.8.0-1.0.8 is too old for kotlin-1.8.20
59             ksp-1.8.0-1.0.8 is too old for kotlin-1.8.20
60             > Task :kspKotlin UP-TO-DATE
```

Esta clase OrderLineServiceTest contiene varias funciones de prueba que prueban el comportamiento de varias funciones en la clase OrderLineService:

findAll(): prueba que la función findAll() devuelve una lista de todas las líneas de ventas guardados en el repositorio.

findByUuid(): prueba que la función findByUuid() devuelve la línea de venta correctamente cuando se le da un UUID válido.

save(): prueba que la función save() guarda correctamente la línea de venta en el repositorio y devuelve la orden guardada.

update(): prueba que la función update() actualiza correctamente una línea de venta existente en el repositorio y devuelve el pedido actualizado.

delete(): prueba que la función delete() elimina correctamente una línea de venta del repositorio y devuelve true, y que se lanza una excepción si se intenta acceder a la orden eliminada.

Las pruebas están implementadas utilizando JUnit 5 y las corutinas de Kotlin. Utilizan la función runTest() de la biblioteca kotlincoroutines.test para ejecutar las funciones de prueba en un contexto de corutina. Además, se usan las anotaciones de MockK para crear objetos falsos y las funciones de prueba coEvery() y coVerify() de MockK para verificar el comportamiento esperado de las funciones de la clase OrderLineService.

The screenshot shows the Android Studio interface. At the top is the code editor with the file `OrderRouter.kt`. The code defines a class `OrderRouter` with methods for creating, reading, updating, and deleting orders. Below the code editor is the `Run` toolbar. The bottom half of the screen is the `Test Results` window, which displays the execution of the `OrderRouter` test suite. It shows four tests passed: `delete()`, `getAll()`, `update()`, and `post()`, each taking approximately 3 seconds and 630ms. The results are color-coded with green checkmarks. A log message at the bottom of the results window indicates that the project configuration is up-to-date.

```
18
19 class OrderRouter {
20     private val order = Order(
21         ObjectId( hexString: "223456789912345678901232" ).toId(),
22         Uuid.fromString( name: "4d83b14d-bdae-422a-9e41-87fbdaef9cff" ),
23         Order.StatusOrder.DELIVERED,
24         total: 12.0,
25         ivar: 12.0,
26         listOf(
27             "ed6f7d0a-7f7a-45bf-8b63-alaa21383271",
28             "e213f434-4c2b-4a28-953f-3981b1ff7e00",
29         ),
30         cliente: 1
31     )
32
33     private val orderDto = order.toDTO()
34
35     //FINDALL
36     @Test
37     fun getAll() = testApplication { this: ApplicationTestBuilder<Order>
38         environment { config }
39         val response = client.get("/order")
40     }
41 }
```

La clase `OrderRouter` está probando las rutas de la API relacionada con el manejo de pedidos. La clase `OrderRouter` tiene 4 métodos de prueba que prueban las rutas `getAll`, `post`, `update` y `delete`.

En el método de prueba `getAll`, se envía una solicitud GET al punto final `"/order"` para recuperar todos los pedidos almacenados en la base de datos. El resultado esperado es un código de estado HTTP 200 (OK).

En el método de prueba `post`, se envía una solicitud POST al punto final `"/order"` con un pedido JSON en el cuerpo de la solicitud para crear un nuevo pedido en la base de datos. El resultado esperado es un código de estado HTTP 201 (Created).

En el método de prueba `update`, primero se crea un pedido en la base de datos utilizando el punto final POST. Luego, se actualiza este pedido modificando algunos de sus campos y enviando una solicitud PUT al punto final `"/order/{id}"` donde el `{id}` es el ID del pedido que se actualiza. El resultado esperado es un código de estado HTTP 200 (OK).

En el método de prueba `delete`, primero se crea un pedido en la base de datos utilizando el punto final POST. Luego, se envía una solicitud DELETE al punto final `"/order/{id}"` donde el `{id}` es el ID del pedido que se desea eliminar. El resultado esperado es un código de estado HTTP 204 (No Content).

```

class OrderLineRoute {
    private val orderLine = OrderLine(
        ObjectId(hexString: "123456789912345678901232").toId(),
        UUID.fromString(name: "4d83b14d-bdae-422a-9e41-87fbdaef9cff"),
        UUID.fromString(name: "0c34f158-5b8e-4193-9ad3-f00e7be93352"),
        amount: 3,
        price: 5.0,
        total: 15.0,
        employee: 1
    )
    private val orderLineDto = orderLine.toDto()
}

@Test
fun getAll() = testApplication { this: ApplicationTestBuilder<OrderLine>
    environment { config }
    val response = client.get(OrderLineRoute::class.java, "/orderline")
    assertEquals(HttpStatusCode.OK, response.status)
}

//POST
@Test

```

Tests passed: 4 of 4 tests – 3 sec 53 ms

Test	Time
OrderLineRoute.getAll()	3 sec 53 ms
OrderLineRoute.delete()	2 sec 778 ms
OrderLineRoute.getAll()	92 ms
OrderLineRoute.update()	108 ms
OrderLineRoute.post()	75 ms

> Configure project :
ksp-1.8.0-1.0.8 is too old for ksp
> Task :kspKotlin UP-TO-DATE
> Task :compileKotlin UP-TO-DATE
> Task :compileJava NO-SOURCE

La clase OrderLineRoute está probando las rutas de la API relacionada con el manejo de líneas de venta. La clase OrderLineRoute tiene 4 métodos de prueba que prueban las rutas getAll, post, update y delete.

La prueba getAll() realiza una solicitud HTTP GET a la ruta /orderline y verifica que la respuesta tenga un estado HTTP 200 (OK).

La prueba post() realiza una solicitud HTTP POST a la ruta /orderline con una línea de venta JSON en el cuerpo de la solicitud para crear un nuevo pedido en la base de datos. El resultado esperado es un código de estado HTTP 201 (Created).

La prueba delete() primero realiza una solicitud HTTP POST para crear una instancia de OrderLine, y luego realiza una solicitud HTTP DELETE a la ruta /orderline/4d83b14d-bdae-422a-9e41-87fbdaef9cff (un ID de ejemplo). Verifica que la respuesta tenga un estado HTTP 204 (NO CONTENT).

La prueba update() primero realiza una solicitud HTTP POST para crear una instancia de OrderLine, y luego realiza una solicitud HTTP PUT a la ruta /orderline/4d83b14d-bdae-422a-9e41-87fbdaef9cff (un ID de ejemplo) con un cuerpo JSON. Verifica que la respuesta tenga un estado HTTP 200 (OK).

6. Enlace proyecto

<https://github.com/idanirf/BiquesDAM>

Alejandro Sánchez Monzón,
Daniel Rodríguez Fernández,
Mireya Sánchez Pinzón,
Jorge Sánchez Berrocoso,
Ruben García-Redondo Marín y
Sergio Pérez Fernández

Damos por finalizada y entregada la práctica en,
Madrid a 07 de Marzo de 2023 a las 00:00:00 horas