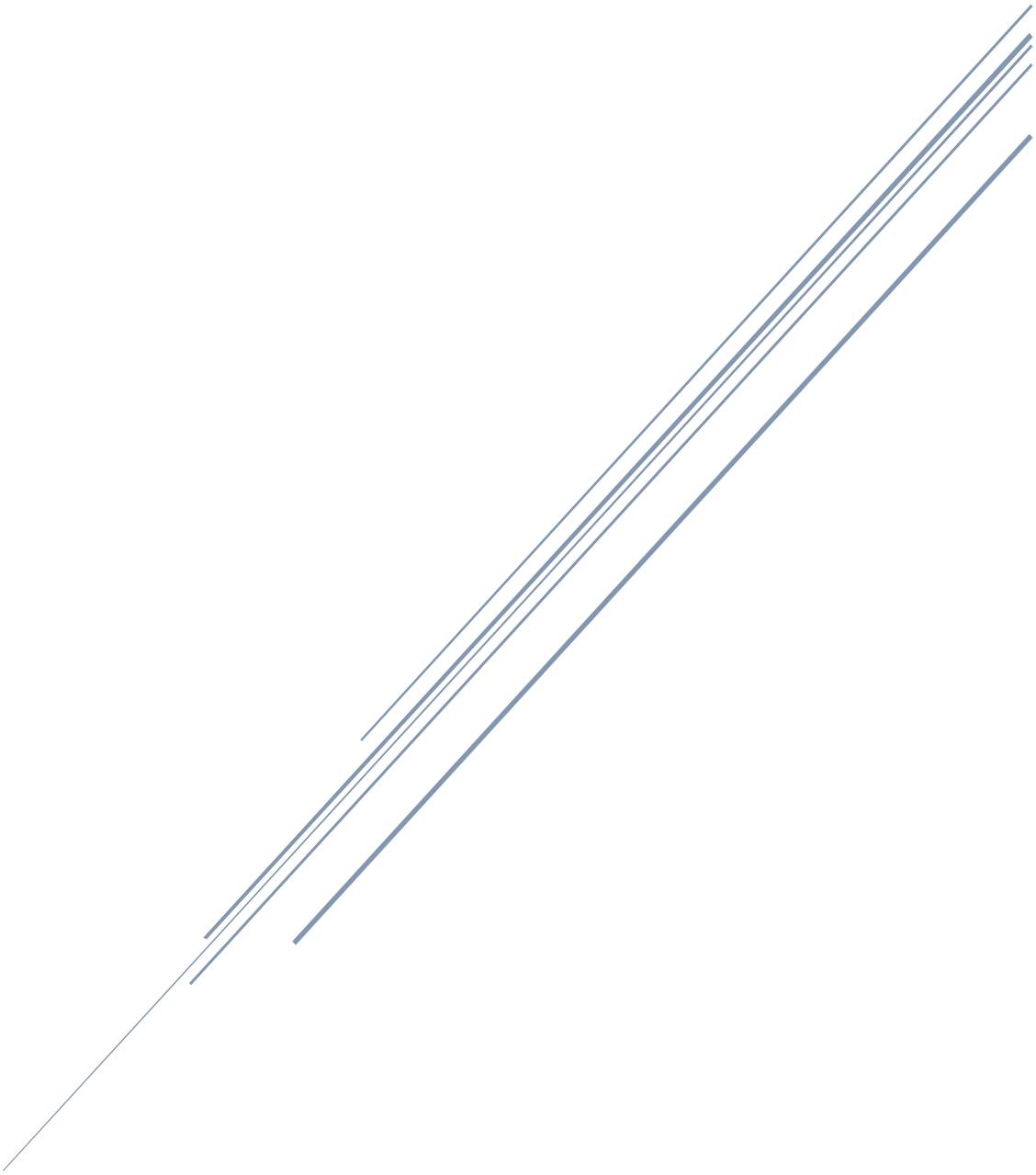
TENNISLAB: MONGODB REACTIVO

Mongo Atlas – KtorFit - Caché



IES Luis Vives
AD – Mario Resa y Sebastián Mendoza

Contenido

Introducción

La forma en la que accedemos a la información ha ido cambiando a lo largo de los años, tanto que es posible ver que, de un año para otro, la forma en la que lo hacemos va difiriendo según las necesidades volátiles que pide el mundo. Es por ello por lo que el ideal de un buen programador es estar en un continuo aprendizaje, una constante actualización, para estar siempre a la vanguardia de la tecnología y la información. Este hecho nos trae precisamente aquí, a este proyecto. Vistas las tecnologías y el diseño de BBDD (Bases de Datos) enfocadas a la estructura relacional, este proyecto trae a primera fila las BBDD NoSQL, siglas de ***Not Only SQL***, es decir no solo SQL (no relacionales).

Las BBDD no relacionales se caracterizan en que no se utilizan estructuras tan sencillas como las tablas, ni se almacena sus datos en forma de registros o campos. Esto nos indica que hay cierta flexibilidad a la hora de almacenar información y que es muy fácil adaptarse a las necesidades de cualquier proyecto a desarrollar.

A parte de acceder a la información en una BBDD de forma local, tenemos la opción de hacerlo de forma remota, las API. Una API, o *interfaz de programación de aplicaciones*, es un conjunto de reglas que definen cómo pueden las aplicaciones o los dispositivos conectarse y comunicarse entre sí. Una API REST es una API que cumple los principios de diseño del estilo de arquitectura REST o *transferencia de estado representacional*. Por este motivo, las API REST a veces se conocen como API RESTful.[1]

Y siguiendo en la línea del tipo de acceso a la información, también nos encontramos con las cachés, información guardada de forma local y temporal de cierta cantidad de información.

La unión de estas tres tecnologías da origen a este proyecto, una pequeña ampliación al anterior y que nos enseña que siempre hay diferentes formas de resolver un problema.

[1] API REST - https://www.ibm.com/es-es/cloud/learn/rest-apis

Diseño

Lenguaje y tecnologías

Siguiendo la línea de los anteriores proyectos, en este también se ha optado por **Kotlin** como lenguaje de programación. A la hora de usar una BBDD se ha utilizado MongoDB reactivo, más específicamente **Mongo Atlas**. La decisión de esto es para evitar de primeras los contenedores en Docker, además de ser una alternativa bastante factible y diferente a la que no estamos acostumbrados de ver.

Como el proyecto necesita de acceso a una API Rest, se ha optado por utilizar **KtorFit**: es un cliente HTTP / procesador de símbolos Kotlin para Kotlin multiplataforma (Js, Jvm, Android, iOS, Linux) que utiliza clientes KSP y Ktor inspirados en Retrofit[2].

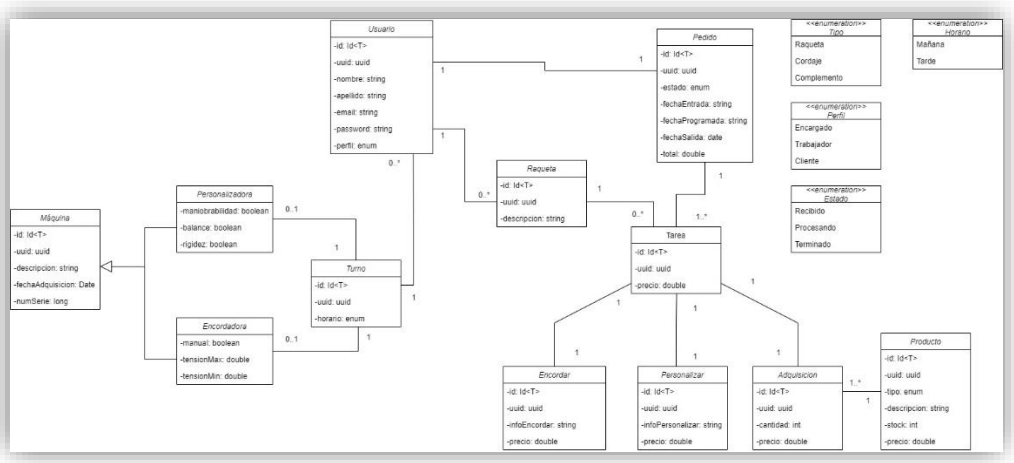
Para terminar, y no menos importante, la caché. En esto caso se ha utilizado **Cache4K**: cache4k proporciona una caché de clave-valor en memoria simple para Kotlin Multiplatform, con soporte para desalojos basados en el tiempo (expiración) y basados en el tamaño.[3]

Mongo Atlas se utilizará para guardar todo el contenido de la aplicación, KtorFit se usará para acceder a la API Rest dada y Cache4K para tener los datos especificados guardados en memoria de forma temporal.

[2] KtorFit - https://foso.github.io/Ktorfit/
[3] Cache4K - https://reactivecircus.github.io/cache4k/

Diagrama

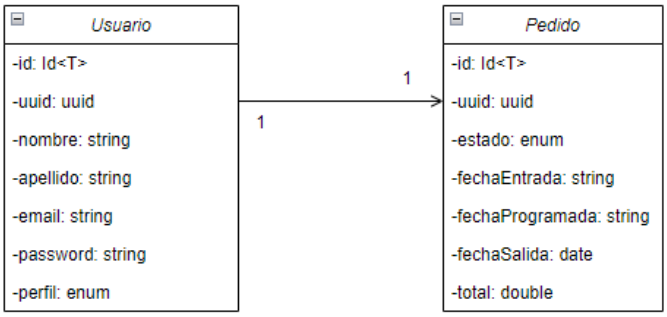
Antes de empezar a picar código, una de las partes más importantes del desarrollo de un proyecto es la creación del diagrama de clases. Este diagrama define las relaciones que van a tener las clases entre sí dando un panorama de cómo va a funcionar el proyecto. Hay diversos diagramas que pueden explicar un proyecto, pero en nuestro caso se ha optado por el de clases ya que nos facilita enseñar las propiedades que tienen cada clase, además de poder ver la cardinalidad entre las mismas.



Usuario, pedido, turno y raqueta

Una de las relaciones más importante que tiene este proyecto es la que tiene *Usuario* con *pedido, turno* y *raqueta*. Vamos por partes:

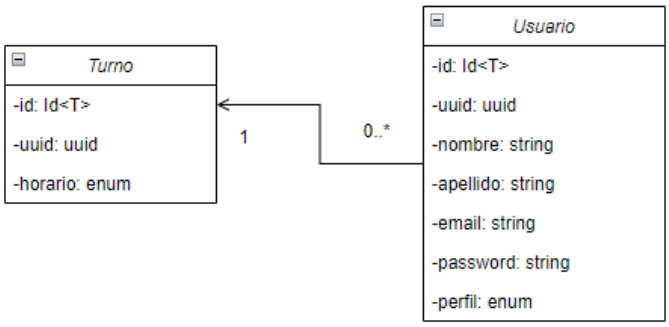
1. Usuario – Pedido.



La relación entre estas dos clases de uno a uno, es decir, un usuario tiene un pedido, y un pedido pertenece a un usuario. La elección de esta relación se ha determinado que sea unidireccional, es decir, una de las clases estará en la otra, pero no viceversa. En este caso, se ha optado a que

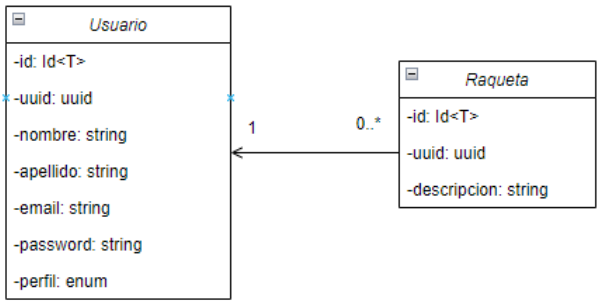
Pedido tenga un *Usuario*, pero que no suceda en el otro sentido. El motivo de esta elección es que nos parecía mas sencillo consultar el pedido de un usuario con la identificación de *Usuario* que tuviera el pedido. Otro motivo ha sido la recursividad, ya que, si el usuario tuviera una lista de pedidos, los pedidos también tendrían un usuario y así hasta el infinito. Lo más seguro es que haya una solución más factible en la que funcione la bidireccionalidad, pero nosotros nos decantamos por esta opción por encima de las demás.

2. Usuario – Turno.



La relación entre *Usuario* y *Turno* se da en los casos en la que el usuario es un trabajador. Como se puede ver en la imagen, esta también es una relación unidireccional de uno a muchos: un usuario tiene un turno de trabajo, y un turno es asignado a uno o varios usuarios. Al igual que en el caso anterior, *Turno* tendrá la información de *Usuario* para evitar recursividad, además de que nos parecía más cómodo consultar los turnos de los usuarios desde el propio turno, evitando también que los perfiles de cliente aparecieran un *Turno* en nulo cuando se consultara por la información del *Usuario*.

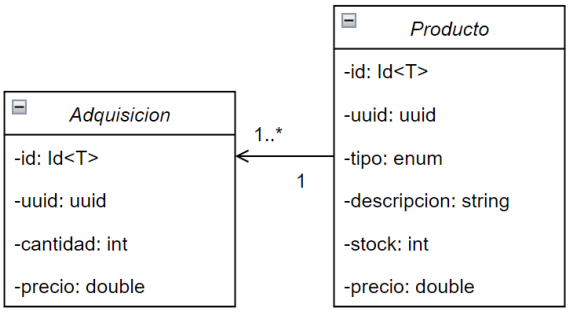
3. Usuario – Raqueta.



En el caso de la relación entre *Usuario* y *Raqueta* es más sencilla que los anteriores. Aquí hemos optado a embeber *Raqueta* en *Usuario*, ya que no necesariamente debe haber una colección con la información de las raquetas de los clientes. Como podemos ver en la cardinalidad, la relación sería de uno a muchos, es decir, un usuario tendría una lista de raquetas, y una (o muchas) raquetas tendrían un usuario específico. Aunque ya hemos señalado que esta relación se embebe, hemos creído oportuno dejarlo como está ya que sería factible crear una tabla independiente de raquetas, y que los usuarios (clientes) tengan una lista de ellas.

Producto y adquisición

El segundo elemento más importante de nuestra aplicación es el *Producto*. Ya que nuestra aplicación emula una tienda, lo suyo es que en dicha tienda haya una serie de productos para comprar. Es por ello por lo que empezaremos explicando la relación existente entre *Producto* y *Adquisición*.

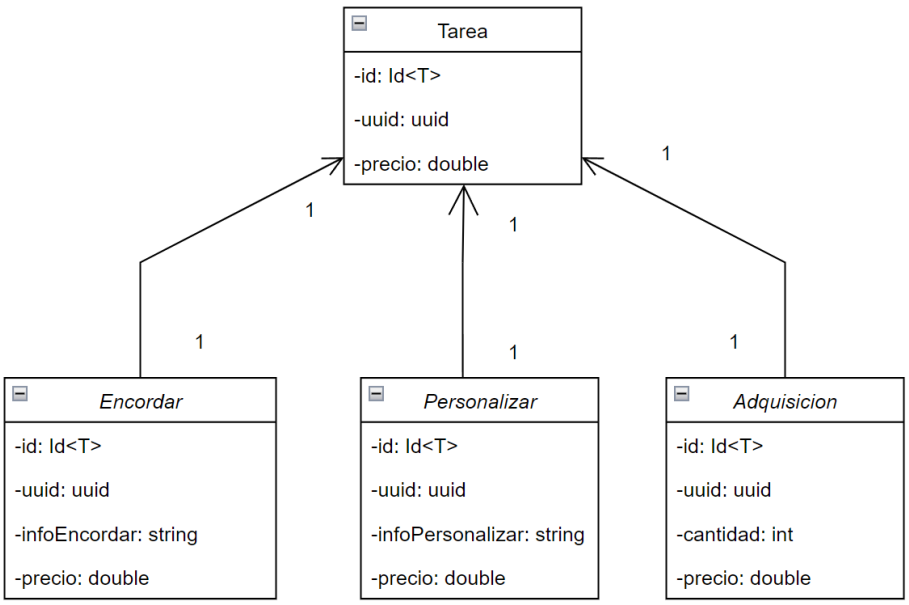


Como podemos observar en la imagen anterior, tenemos una relación de uno a muchos, es decir, un producto puede ser adquirido una o varias veces, y una adquisición está compuesta por un único producto. En este caso, la unidireccionalidad será en el sentido de la adquisición, es decir, *Producto* estará en *Adquisición*, y no viceversa. Al igual que en las anteriores explicaciones, se ha hecho así para evitar recursividad, además de que nos parecía lógico que los productos estuvieran en las adquisiciones, y que no tenía sentido meter la información de las adquisiciones dentro de productos.

Tarea, pedido, raqueta, encordado, personalizado y adquisición

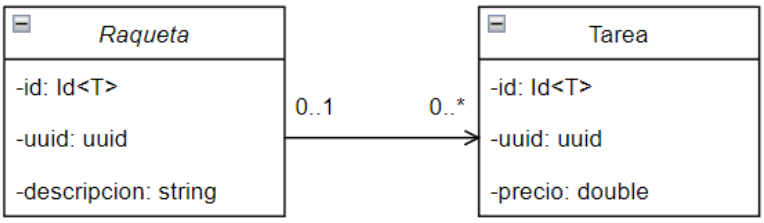
Una de las partes centrales del proyecto son las tareas, ya que ésta tiene una relación muy estrecha con varias clases del proyecto. Esto se debe a que las tareas comprenden gran parte de la lógica de la aplicación con respecto a lo que se pide en el proyecto. Según este, las tareas están compuestas por una raqueta (siempre que fuera necesario), un encordado de raqueta, una personalización de esta o una adquisición. Esto quiere decir que, según el servicio que pida un cliente, se creará una tarea específica. Vamos a explicarlo con más detalle:

1. Tarea – encordar – personalizar – adquisición



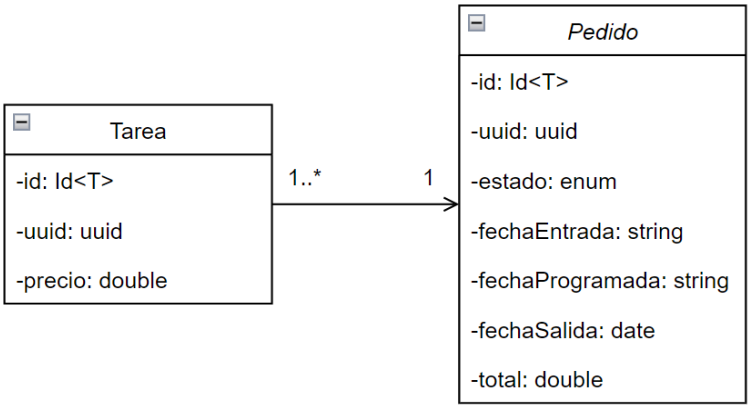
Una de las primeras cosas a destacar es que las clases *Encordar*, *Personalizar* y *Adquisición* se encontrarán unidireccionalmente en *Tarea*, esto quiere decir que no tendremos la clase *Tarea* en las otras tres, puesto que nuestra lógica nos dictó que no merecía la pena agregar la información de *Tarea* en las demás clases con la que se relaciona. En el caso de que se quiera consultar una adquisición, por ejemplo, iríamos directamente a *Tarea* y veríamos la información allí, y lo mismo sucede con las demás. Por otro lado, podemos ver que las relaciones de cardinalidad nos indica que existe una relación de uno a uno entre las clases, siendo este el resultado de las condiciones que pide el proyecto, una tarea por cada acción.

2. Tarea – Raqueta



Una de las condiciones del proyecto es que, si fuera necesario, las tareas tendrían una raqueta de un cliente, ya sea para hacerle una personalización o un nuevo encordado. En la imagen adjunta, vemos que tenemos una relación de uno a muchos, es decir, *Raqueta* puede estar en ninguna o varias *Tareas*, y éstas pueden tener una raqueta (si llegara a necesitarla). En este caso, hemos decidido que *Raqueta* esté unidireccionalmente en *Tarea*, pues no nos hace falta tener la información de la tarea en la raqueta seleccionada.

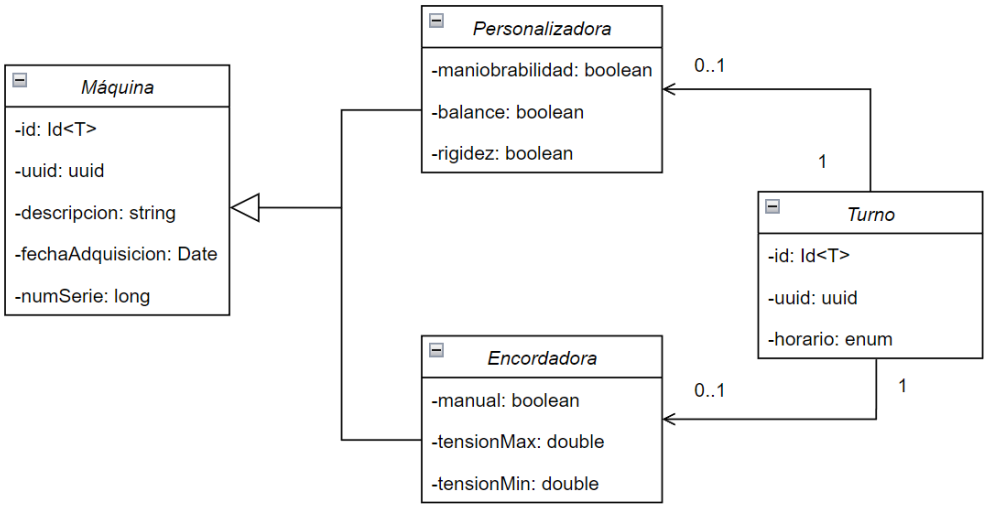
3. Tarea – Pedido



Por último, en esta sección, y no menos importante, la relación *Tarea* y *Pedido*. Tal y como se especifica en las condiciones del proyecto, los pedidos se componen de una lista de tareas, es decir, existe una relación de uno a muchos: un pedido tiene una o varias tareas, y las tareas están asociadas a un pedido. En nuestro proyecto se decidió que *Tarea* estuviera dentro de *Pedido* (condiciones del proyecto), pero no quisimos tener la información de los pedidos en las tareas. Nos parecía redundante tener tanta información en ambos lados, puesto que, si quisiéramos saber las tareas de un pedido, consultaríamos el pedido en sí, y es muy poco probable que se consulte una tarea individual.

Máquina, encordadora, personalizadora, turno

Una parte muy importante de nuestro proyecto son las máquinas que se encargan de personalizar o encordar las raquetas. Según el proyecto, dichas máquinas tendrán asociados un turno en el que se pueden usar y en los que no. Es por ello por lo que la disposición de estas clases en el diagrama ha quedado así:

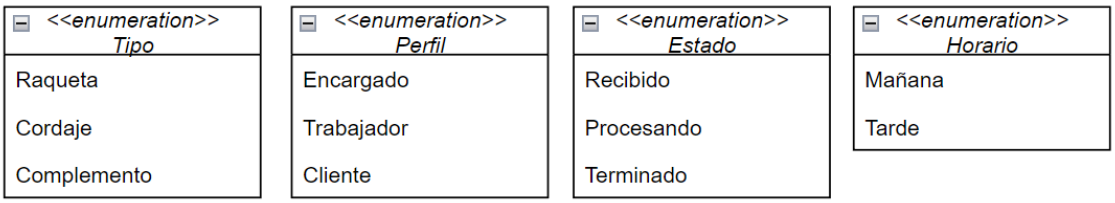


Por un lado, tenemos una herencia de *Máquina* a las clases *Personalizadora* y *Encordadora*. Teníamos la elección de hacer dos máquinas por separado, pero quisimos ver si podíamos trabajar con herencia en el proyecto. Teniendo dichas máquinas por separado, estas se relacionan cada una con la clase *Turno*. En ambas relaciones tenemos uno a muchos, es decir, las maquinas tienen un turno, y los turnos pueden o no tener máquinas asociadas.

En esta ocasión, se ha elegido que los turnos estén en las máquinas, y no viceversa, es decir, embebemos *Turno* en las máquinas. Nos pareció más cómodo consultar en las propias máquinas si están asociadas a un turno, en vez de hacerlo mediante la clase *Turno*.

Enums

Para terminar esta sección, dejamos señalados cómo se han configurado los enums en el diagrama. Aunque no hacen parte directa de cómo se relacionan las clases del proyecto, es importante señal que sí hacen parte en la configuración de un diagrama de clase. Así pues, este es el resultado:



Estructura de proyecto

Teniendo el diagrama, el lenguaje y las tecnologías, podemos empezar a estructurar nuestro proyecto para llevar a cabo la práctica.

En este apartado iremos desglosando poco a poco como se han ido configuración las clases en sus diferentes carpetas. Hay que señalar que, aunque en el diagrama anterior no se comentaban las variables que tenían cada clase, es ahora en la siguiente sección donde repasaremos más detenidamente en la elección de las variables en cada clase.

Gradle

Ya que nuestro proyecto necesita de diferentes tecnologías para funcionar, lo primero que debemos hacer es configurar el Gradle, es decir, introducir las dependencias que harán funcionar correctamente la aplicación.

Dependencias

1. Test. Dependencias para realizar test en este proyecto.

```
testImplementation(kotlin("test"))
testImplementation("io.mockk:mockk:1.12.4")
testImplementation("org.jetbrains.kotlinx:kotlinx-coroutines-test:1.6.4")
```

2. KMongo. Dependencia para poder usar correctamente MongoDB con Kotlin.

```
// Kmongo
implementation("org.litote.kmongo:kmongo-async:4.7.2")
implementation("org.litote.kmongo:kmongo-coroutine:4.7.2")
```

3. Coroutines. Para usar coroutines en Kotlin

```
// Corrutinas
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4")
```

4. Logs. Dependencia espacial para imprimir por consola mejor la información.

```
// Log
implementation("io.github.microutils:kotlin-logging-jvm:3.0.4")
implementation("ch.qos.logback:logback-classic:1.4.4")
```

5. Serializador. Serializador específico para Kotlin.

```
// Serializar Json
implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.4.1")
```

6. BCrypt. Encriptador de contraseñas.

```
// BCrypt
implementation("com.ToxicBakery.library.bcrypt:bcrypt:1.0.9")
```

7. KtorFit. Retrofit pero para Kotlin.

```
// KtorFit
ksp("de.jensklingenberg.ktorfit:ktorfit-ksp:1.0.0-beta16")
implementation("de.jensklingenberg.ktorfit:ktorfit-lib:1.0.0-beta16")
```

8. Serializador en Ktor.

```
// Para serializar en Json con Ktor
implementation("io.ktor:ktor-client-serialization:2.1.3")
implementation("io.ktor:ktor-client-content-negotiation:2.1.3")
implementation("io.ktor:ktor-serialization-kotlinx-json:2.1.3")
```

9. Caché. Para guardar cierta información en caché de forma temporal.

```
// Cache
implementation("io.github.reactivecircus.cache4k:cache4k:0.9.0")
```

10. Documentación para Kotlin.

```
//Dokka Documentación Kotlin
dokkaHtmlPlugin("org.jetbrains.dokka:kotlin-as-java-plugin:1.7.20")
```

11. Koin, inyección de dependencias.

```
//Koin
implementation("io.insert-koin:koin-core:3.2.2")
implementation("io.insert-koin:koin-annotations:1.0.3")
ksp("io.insert-koin:koin-ksp-compiler:1.0.3")
```

Modelos

Trece son las clases modelo que comprenden nuestro proyecto. Para que la explicación de esta sección no sea repetitiva, solamente enseñaremos cómo están diseñadas las clases y puntualizaremos en aquellas que tengan algún elemento especial.

Producto

```
Mario +1
@Serializable
class Producto(
    @BsonId @Contextual
    val id: Id<Producto> = newId(),
    @Contextual
    val uuid: UUID = UUID.randomUUID(),
    val tipo: Tipo,
    val descripcion: String,
    val stock: Int,
    var precio: Double
) {
    Mario
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}

Mario
enum class Tipo { RAQUETA, CORDAJE, COMPLEMENTO }
```

Tres elementos importantes por destacar es que cada clase modelo tendrá las siguientes etiquetas: @Serializable, @BsonId y @Contextual. Estas etiquetas nos sirven para que la serialización de los objetos creados se haga de forma correcta, y así no tener ningún problema a la hora de utilizar Json para el manejo de la información. Las últimas dos etiquetas nos ayudan a no tener problemas en la obtención de UUID y el Id de Kmongo.

Adquisición

```
Mario +1
@Serializable
class Adquisicion(
    @BsonId @Contextual
    val id: Id<Adquisicion> = newId(),
    @Contextual
    val uuid: UUID = UUID.randomUUID(),
    var cantidad: Int,
    val producto: Producto,
    val precio: Double = CalculoPrecioTarea.calculatePrecio(producto.precio, data2: null, data3: null) * cantidad

) {
    Sebastián Mendoza Acosta +1
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}
```

Clase igual a la anterior, matizando en dos puntos: 1. Tenemos la clase externa *Producto* y 2. Un método genérico para obtener el precio del producto adquirido, multiplicado por la cantidad del mismo (dicho método se explicará más adelante).

Encordar

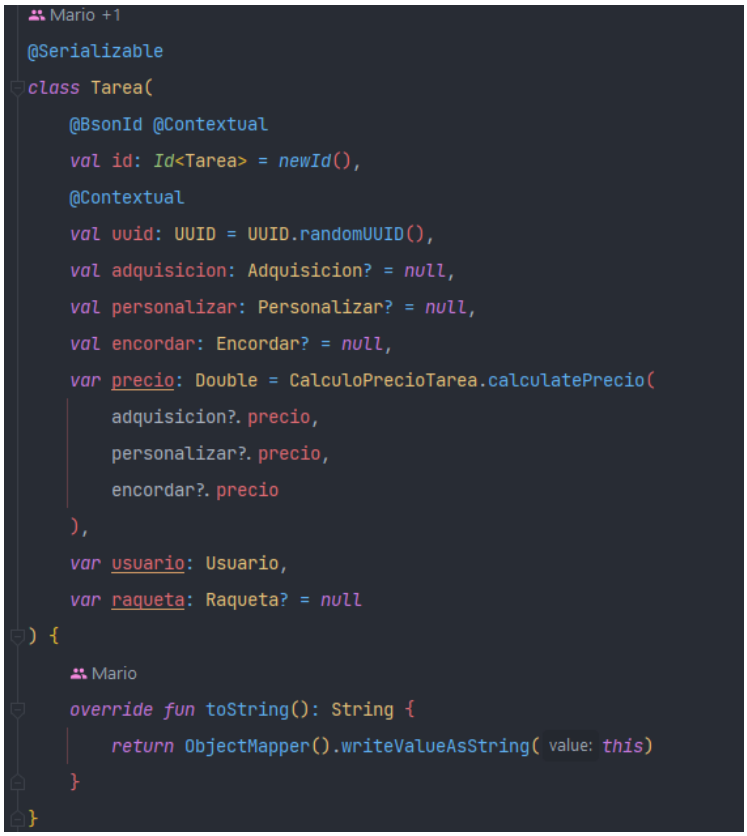
```
Mario +1
@Serializable
class Encordar(
    @BsonId @Contextual
    val id: Id<Encordar> = newId(),
    @Contextual
    val uuid: UUID = UUID.randomUUID(),
    var informacionEndordado: String,
    val precio: Double = 15.0
) {
    Mario
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}
```

En el caso de encordar, se “settea” el precio a 15 ya que es uno de los parámetros estipulados en la práctica.

Personalizar

Ídem a la clase anterior, siendo en este caso el precio de una personalización de 60.

Tarea



En la clase *Tarea* introducimos las clases *Adquisición, Personalizar, Encordar, Usuario y Raqueta*. Las tres primeras se inicializan a *null* ya que una tarea se compone exclusivamente de uno de

estos elementos a la vez por condiciones del proyecto. El usuario en este caso sería un trabajador asignado a la tarea, y la raqueta por si necesariamente se vaya a utilizar una en dicha acción.

Pedido

En el caso de *Pedido*, tres variables aquí son importantes: *Cliente, Tareas y Precio*. Para nuestro proyecto, una de las elecciones a la hora de hacer esta clase era que los pedidos iban a tener un propietario, es decir, el cliente (además de que así se planteó en el diagrama antes explicado). Por otro lado, tendría también una lista de tareas, la cual modificaría el valor del precio. Es por ello que, en la variable *Precio*, lo que hacemos es un *sumOf* en la lista, cogiendo el precio. Esto nos devolvería el valor total del pedido.

Usuario

```
Mario +1
@Serializable
class Usuario(
    @BsonId @Contextual
    val id: Id<Usuario> = newId(),
    @Contextual
    val uuid: UUID = UUID.randomUUID(),
    var name: String,
    var email: String,
    val password: ByteArray,
    var raqueta: List<Raqueta>? = null,
    var perfil: Perfil
) {
    Mario
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}

Mario
enum class Perfil { ADMIN, ENCORDADOR, CLIENTE }
```

A pesar de que la clase *Usuario* es muy importante en nuestro proyecto, se esperaría suponer que en la misma hubiera más clases externas, ya que hace parte no solo de los pedidos, sino también en las raquetas, e incluso en la clase *Turno* (la veremos más adelante). Así pues, lo único que tenemos en *Usuario* es una lista de raquetas, y esto se debe a que puede surgir en alguna ocasión que un cliente quiera personalizar o "re-encordar" una raqueta que ya tenga en propiedad.

Raqueta

```
Mario +1
@Serializable
class Raqueta(
    @BsonId @Contextual
    val id: Id<Raqueta> = newId(),
    @Contextual
    val uuid: UUID = UUID.randomUUID(),
    val descripcion: String
)
{
    Mario
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}
```

Como podemos ver, la clase *Raqueta* no goza de múltiples variables, tan solo de sus identificadores y de una descripción en la que se indica la marca y el modelo de dicha raqueta. Al estar embebida en el usuario, nos pareció importante acotar la información que tendría la raqueta para que no abultara mucho en la clase *Usuario*.

Turno

```
Mario
@Serializable
class Turno(
    @BsonId @Contextual
    val id: Id<Turno> = newId(),
    @Contextual
    val uuid: UUID = UUID.randomUUID(),
    var horario: TipoHorario,
    var trabajador: Usuario
)
{
    Mario
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}

Mario
enum class TipoHorario { TEMPRANO, TARDE }
```

Como ya se explicó en el diagrama, *Turno* estará embebida en las máquinas. De esta manera nos era más cómodo consultar los turnos de los trabajadores en la colección *Turno* que desde *Usuario*. En este caso, los turnos estarían compuestos por un horario (mañana o tarde) y de un usuario.

Máquina

```
Mario +1
@Serializable
open class Maquina(
    @BsonId @Contextual
    val id: Id<Maquina>,
    @Contextual
    val uuid: UUID,
    val descripcion: String,
    var fechaAdquisicion: String,
    var numSerie: Long,
    var turno: Turno?

) {
    Sebastián Mendoza Acosta +1
    override fun toString(): String {
        return "Maquina(id=$id, uuid=$uuid, descripcion='$descripcion', fechaAdquisicion='$fechaAdquisicion', numSerie=$numSerie, turno=$turno)"
    }
}
```

La clase *Máquina* es la única que hace herencia en todo el proyecto. Siguiendo la línea del proyecto anterior, esta clase tendrá las variables que salen en la imagen. Éstas, al final, serán heredadas en las clases *Encordadora* y *Personalizadora*.

Encordadora y Personalizadora

```
Mario +1
class Encordadora(
    id: Id<Maquina> = newId(),
    uuid: UUID = UUID.randomUUID(),
    descripcion: String,
    fechaAdquisicion: String,
    numSerie: Long,
    turno: Turno? = null,
    var isManual: Boolean,
    var tensionMax: Double,
    var tensionMin: Double,
) : Maquina(id, uuid, descripcion, fechaAdquisicion, numSerie, turno) {
    Sebastián Mendoza Acosta +1
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}
```

Como se mencionó en el apartado anterior, una de las clases que iba heredar de *Máquina* sería la clase *Encordadora*. Esta clase, además de tener las de la herencia, tiene otras tres que definirían el tipo de encordadora tiene la tienda (proyecto). Lo mismo sucede con la clase *Personalizadora* que veremos a continuación:

```
Mario +1
class Personalizadora(
    id: Id<Maquina> = newId(),
    uuid: UUID = UUID.randomUUID(),
    descripcion: String,
    fechaAdquisicion: String,
    numSerie: Long,
    turno: Turno? = null,
    var maniobrabilidad: Boolean,
    var balance: Boolean,
    var rigidez: Boolean
) : Maquina(id, uuid, descripcion, fechaAdquisicion, numSerie, turno) {
    Sebastián Mendoza Acosta +1
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}
```

Response

```
Mario
@Serializable
sealed class Response<T>

Mario
@Serializable
class ResponseSuccess<T : Any>(val code: Int, val data: T) : Response<T>()

Mario
@Serializable
class ResponseFailure(val code: Int, val message: String) : Response<@Contextual Nothing>()
```

Para terminar esta sección, la clase *Response*. Esta clase se ocupa de devolver un código y un mensaje según la ejecución de una función.

DTO

En este proyecto también hemos contado con DTO(*Data Transfer Object*) a la hora de devolver por la consola cierta información. Comparten cierta similitud con las clases modelos equivalentes, por lo que solo se mostrarán un par de ejemplos para no ser repetitivos. Hay que añadir que cada una de estas nuevas clases cuentan con un método que convierte los objetos del modelo a éstos nuevos DTO. El resultado ha sido el siguiente:

Usuario

```
Mario +1
@Serializable
data class UsuarioDto(
    val id: String,
    var name: String,
    var email: String,
)

Mario
fun UsuarioDto.toUsuario(pass: String): Usuario {
    return Usuario(
        id = TransformIDs.generateIdUsers(id),
        uuid = UUID.randomUUID(),
        name = name,
        email = email,
        password = Cifrador.codifyPassword(pass),
        perfil = Perfil.CLIENTE
    )
}

Mario
fun Usuario.toUsuarioDto(): UsuarioDto {
    return UsuarioDto(
        id = id.toString(),
        name, email
    )
}
```

Producto

```
Mario *
@Serializable
data class ProductoDto(
    val tipo: String,
    val descripcion: String,
    val stock: String,
    val precio: String
) {
}

Mario *
fun Producto.toProductoDto(): ProductoDto {
    return ProductoDto(
        tipo = tipo.name,
        descripcion,
        stock = stock.toString(),
        precio = precio.toString()
    )
}
```

Database

Como nuestra aplicación tiene que guardar la información en MongoDB, elegimos que ésta fuera mediante Mongo Atlas. Para ello, se tuvo que configurar mediante un archivo *Properties* las diferentes variables que componían nuestra conexión a Mongo Atlas. La lectura de las variables se ha hecho mediante un método al que se le indica dónde están dichas propiedades (la carpeta) y el nombre. El método es el siguiente:

```
fun propertiesReader(propertiesFile: String): Properties {
    val file =
        FileInputStream( name: "." + File.separator + "src" + File.separator + "main" + File.separator +
         "resources" + File.separator + propertiesFile)
    val prop = Properties()
    prop.load(file)
    return prop
}
```

Archivo *Properties*:

```
mongo.type=mongodb+srv://
host=dam.ahgscx9.mongodb.net
database=tennislab
username=tennislab
password=mongoreactivo
options=?authSource=admin&retryWrites=true&w=majority
```

Teniendo las propiedades, ya podemos configurar la clase que maneja MongoDB:

```
Sebastián Mendoza Acosta +1
object MongoDbManager {

    private val properties = Propiedades.propertiesReader( propertiesFile: "config.properties")
    private var mongoClient: CoroutineClient
    var database: CoroutineDatabase

    private val MONGO_TYPE = properties.getProperty("mongo.type")
    private val HOST = properties.getProperty("host")
    private val DATABASE = properties.getProperty("database")
    private const val USERNAME = "tennisLab"
    private const val PASSWORD = "mongoreactivo"
    private val OPTIONS = properties.getProperty("options")

    private val MONGO_URI =
        "$MONGO_TYPE$USERNAME:$PASSWORD@$HOST/$DATABASE"

    Sebastián Mendoza Acosta
    init {
        log.debug( msg: "Inicializando conexión a MongoDB")

        println("Inicializando conexión a MongoDB → $MONGO_URI$OPTIONS")
        mongoClient =
            KMongo.createClient( connectionString: "$MONGO_URI$OPTIONS")
                .coroutine
        database = mongoClient.getDatabase( name: "tennisLab")
    }
}
```

Servicios

En la sección de *Servicios* tenemos dos clases muy importantes: la configuración de la caché y el método que comprueba los cambios en tiempo real de los *Productos*.

Cache

Para poder cachear información con *Cache4K*, hay que configurar algunos parámetros. Este proyecto pedía que se hiciera cada sesenta segundos, por ejemplo.

```
Mario +1
@Single
@Named("UsuariosCache")
class UsuariosCache {

    val refreshTime = 60000 // 1 minuto

    val cache = Cache.Builder()
        .expireAfterAccess(5.minutes)
        .build<Id<Usuario>, Usuario>()
}
```

ProductoService

La clase *ProductoService* contiene una función que "observa" los cambios producidos en la colección de datos de *Producto*.

```
@Single
@Named("ProductoService")
class ProductoService {
    Sebastián Mendoza Acosta
    fun watch(): ChangeStreamPublisher<Producto> {
        logger.debug { "Watch()" }
        return MongoDbManager.database.getCollection<Producto>().watch<Producto>()
            .publisher
    }
}
```

KtorFit

Una de las indicaciones para la realización de la práctica es acceder a una API Rest externa para obtener la información de los usuarios que pueden interactuar con la aplicación. El primero paso es conocer la *url* y la configuración de los *endpoints.*

En el primer caso la url es “https://jsonplaceholder.typicode.com/” y los endpoints los siguientes:

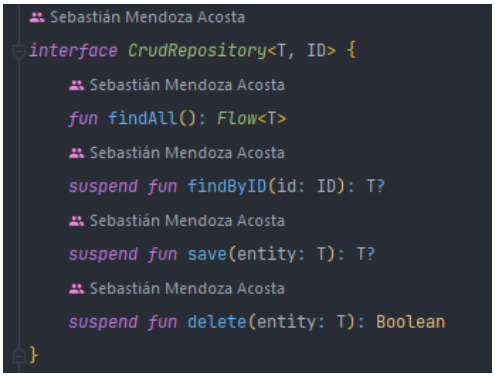
Podemos ver en esta imagen que solo tenemos dos, el de *users* y el de *todos*, el primero para obtener la lista de todos los usuarios y el segundo para agregar *Tareas* a la API, y cada uno con sus respectivos DTO.

Teniendo el cliente, ya podemos configurar el cliente de KtorFit:

Repositorios

La carpeta *Repositories* alberga la configuración de cómo se harán los CRUD (*Create, Read, Update and Delete*) en nuestro proyecto. Es por ello por lo que se compone de tres elementos esenciales: una interfaz general que tiene todos los métodos que implementarán los demás repositorios, una interfaz específica para cada clase con su clase modelo y el tipo de dato que manejarán, y la implementación del repositorio donde se desarrollarán la configuración de los CRUD de cada clase. Para esta ocasión, enseñaremos un ejemplo para una colección, y todos los que necesita *Usuario* para que pueda funcionar con la API Rest, la caché y MongoDB, y también de *Tarea*, ya que hay que subir a la API las tareas que tiene un cliente.

CrudRepository



UsuarioRepository

UsuarioMongoRepositoryImpl

```
Mario +1
class UsuariosMongoRepositoryImpl : UsuarioRepository {

    Mario
    override fun findAll(): Flow<Usuario> {
        println("\tfindAllMongo")
        return MongoDbManager.database.getCollection<Usuario>().find().toFlow()
    }

    Mario
    override suspend fun findByID(id: Id<Usuario>): Usuario? {
        println("\tfindByIDMongo")
        return MongoDbManager.database.getCollection<Usuario>().findOneById(id)
    }

    Mario
    override suspend fun delete(entity: Usuario): Boolean {
        println("\tdeleteMongo")
        var existe = false
        if (MongoDbManager.database.getCollection<Usuario>().deleteOneById(entity.id).equals(entity)) existe = true

        return existe
    }

    Mario
    override suspend fun save(entity: Usuario): Usuario {
        println("\tsaveMongo")
        MongoDbManager.database.getCollection<Usuario>().save(entity)
        return entity
    }

}
```

UsuarioCacheRepositoryImpl

Esta clase en particular es bastante extensa, por lo que se mostrará en varias partes. Por lo pronto, hemos de explicar que implementa la interfaz de *UsuarioRepository*. Dentro de la clase también usamos la caché, un método que, según la configuración de ésta, nos indica los usuarios que están en la BBDD.

```
Mario +1 *
class UsuariosCacheRepositoryImpl : UsuarioRepository {

    private val cacheUsuarios = UsuariosCache()
    private var refreshJob: Job? = null

    private var listaBusquedas = mutableListOf<Usuario>()

    Mario
    init {
        refreshCache()
    }
```

```kotlin
Mario
override fun findAll(): Flow<Usuario> {
    println("\tfindAllCache")
    return cacheUsuarios.cache.asMap().values.asFlow()
}

Mario
override suspend fun delete(entity: Usuario): Boolean {
    println("\tdeleteCache")
    var existe = false
    val usuario = cacheUsuarios.cache.asMap()[entity.id]
    if (usuario != null) {
        listaBusquedas.removeIf { it.id == usuario.id }
        cacheUsuarios.cache.invalidate(entity.id)
        existe = true
    }
    return existe
}

Mario
override suspend fun save(entity: Usuario): Usuario {
    println("\tsaveCache")
    listaBusquedas.add(entity)
    return entity
}
```

```kotlin
override suspend fun findByID(id: Id<Usuario>): Usuario? {
    println("\tfindByIDCache")
    var usuario: Usuario? = null

    cacheUsuarios.cache.asMap().forEach { it: Map.Entry<Any?, Usuario>
        if (it.key == id) {
            usuario = it.value
        }
    }
    return usuario
}

Mario
fun refreshCache() {
    if (refreshJob != null) refreshJob?.cancel()

    refreshJob = CoroutineScope(Dispatchers.IO).launch { this: CoroutineScope
        while (true) {
            println("Refrescando cache usuarios")
            if (listaBusquedas.isNotEmpty()) {
                listaBusquedas.forEach { it: Usuario
                    val user = it
                    cacheUsuarios.cache.put(user.id, user)
                }

                listaBusquedas.clear()

                println("Cache actualizada: ${cacheUsuarios.cache.asMap().size}")
            }
            delay(cacheUsuarios.refreshTime.toLong())
        }
    }
}
```

UsuariosKtorFitRepositoryImpl

```
Mario +1
class UsuariosKtorFitRepositoryImpl {

    private val client by lazy { KtorFitClient.instance }

    Mario
    suspend fun findAll(): Flow<UsuarioDto> = withContext(Dispatchers.IO) { this: CoroutineScope
        val call = client.getAll()

        return@withContext call.asFlow()
    }

}
```

TareasKtorFitRepository

```
class TareasKtorFitRepository : TareaRepository {

    private val client by lazy { KtorFitClient.instance }

    Mario
    suspend fun uploadTarea(entity: Tarea): Tarea = withContext(Dispatchers.IO) { this: CoroutineScope
        println("\tuploadAdquisicion")

        client.createTarea(entity.toTareaDto())
        return@withContext entity
    }

    Mario
    override fun findAll(): Flow<Tarea> {
        logger.debug { "findAll()" }
        return MongoDbManager.database.getCollection<Tarea>().find().publisher.asFlow()
    }

    Mario
    override suspend fun findByID(id: Id<Tarea>): Tarea? {
        logger.debug { "findByID($id)" }
        return MongoDbManager.database.getCollection<Tarea>().findOneById(id)
    }

    Mario
    override suspend fun save(entity: Tarea): Tarea {
        logger.debug { "save($entity)" }
        return MongoDbManager.database.getCollection<Tarea>().save(entity).let { entity }
    }

    Mario
    override suspend fun delete(entity: Tarea): Boolean {
        logger.debug { "delete($entity)" }
        return MongoDbManager.database.getCollection<Tarea>().deleteOneById(entity.id).let { true }
    }
```

Controladores

Teniendo los repositorios configurados, es hora de diseñar los controladores. Estos se encargarán de manejar los CRUD más eficientemente que si se hiciera directamente con los repositorios. En este caso, enseñaremos el controlador *APIController* ya que usa el repositorio que guarda la información del usuario en cache, en mongo y en la API.

APIController

Esta clase comprende el uso de cuatro repositorios diferentes: los que utiliza *Usuario* y *Tarea*.

```
Mario +1
@Single
class APIController(
    private val usuariosCacheRepositoryImpl: UsuariosCacheRepositoryImpl,
    private val usuariosMongoRepositoryImpl: UsuariosMongoRepositoryImpl,
    private val usuariosKtorFitRepositoryImpl: UsuariosKtorFitRepositoryImpl,
    private val tareasKtorFitRepository: TareasKtorFitRepository
) {
```

```
Mario +1
suspend fun getAllUsuariosApi(): Flow<Usuario> = withContext(Dispatchers.IO) { this: CoroutineScope
    val listado = mutableListOf<Usuario>()
    usuariosKtorFitRepositoryImpl.findAll().collect { listado.add(it.toUsuario( pass: "Hola1")) }

    println(Json.encodeToString(ResponseSuccess( code: 200, listado.toString())))
    return@withContext listado.asFlow()
}

Mario
suspend fun getAllUsuariosMongo(): Flow<Usuario> = withContext(Dispatchers.IO) { this: CoroutineScope
    val response = usuariosMongoRepositoryImpl.findAll()

    println(Json.encodeToString(ResponseSuccess( code: 200, response.toString())))
    return@withContext response
}

Mario
suspend fun getAllUsuariosCache(): Flow<Usuario> = withContext(Dispatchers.IO) { this: CoroutineScope
    val response = usuariosCacheRepositoryImpl.findAll()

    println(Json.encodeToString(ResponseSuccess( code: 200, response.toString())))
    return@withContext response
}
```

```
Mario
suspend fun saveUsuario(entity: Usuario): Usuario = withContext(Dispatchers.IO) { this: CoroutineScope
    launch { this: CoroutineScope
        val response = usuariosCacheRepositoryImpl.save(entity)
        println(Json.encodeToString(ResponseSuccess( code: 201, response.toUsuarioDto())))
    }

    launch { this: CoroutineScope
        val response = usuariosMongoRepositoryImpl.save(entity)
        println(Json.encodeToString(ResponseSuccess( code: 201, response.toUsuarioDto())))
    }
    joinAll()

    return@withContext entity
}

Mario
suspend fun getUsuarioById(id: Id<Usuario>): Usuario? {
    var userSearch = usuariosCacheRepositoryImpl.findByID(id)
    if (userSearch == null) {
        println(Json.encodeToString(ResponseSuccess( code: 200, userSearch.toUsuarioDto())))

    } else {
        userSearch = usuariosMongoRepositoryImpl.findByID(id)
        if (userSearch == null) {
            println(Json.encodeToString(ResponseSuccess( code: 201, userSearch.toUsuarioDto())))
        } else System.err.println(Json.encodeToString(ResponseFailure( code: 404,  message: "User not found")))
    }

    return userSearch
}
```

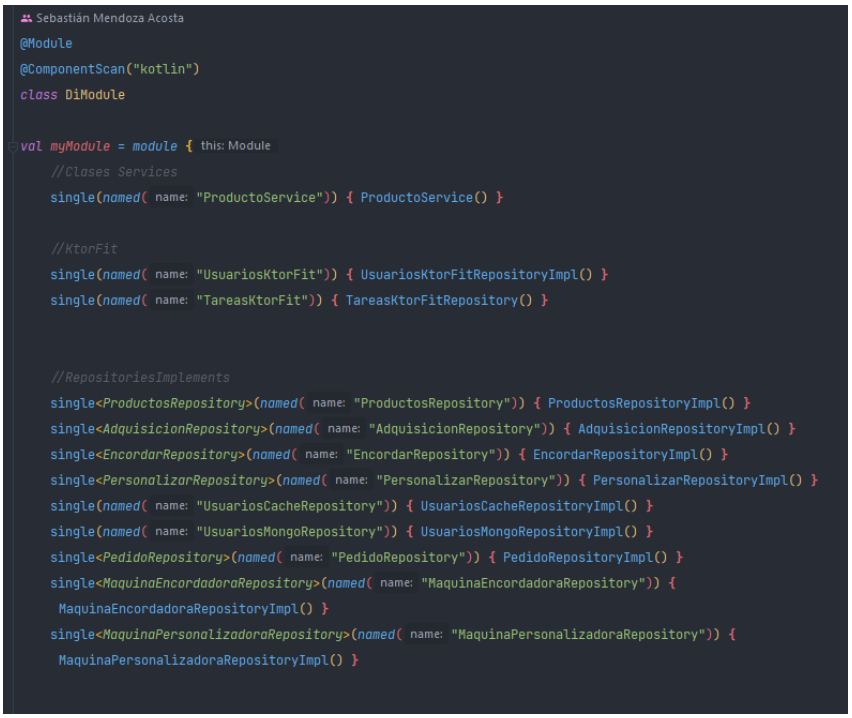
```
Mario
suspend fun deleteUsuario(entity: Usuario) = withContext(Dispatchers.IO) { this: CoroutineScope
    launch { this: CoroutineScope
        usuariosCacheRepositoryImpl.delete(entity)
        println(Json.encodeToString(ResponseSuccess( code: 200, entity.toUsuarioDto())))
    }

    launch { this: CoroutineScope
        usuariosMongoRepositoryImpl.delete(entity)
        println(Json.encodeToString(ResponseSuccess( code: 200, entity.toUsuarioDto())))
    }

    joinAll()
}

// TAREAS
Mario
suspend fun getAllTareas(): Flow<Tarea> = withContext(Dispatchers.IO) { this: CoroutineScope
    val response = tareasKtorFitRepository.findAll()

    println(Json.encodeToString(ResponseSuccess( code: 200, response.toString())))
    return@withContext response
}
```

```
Mario +1
suspend fun saveTarea(entity: Tarea): Tarea = withContext(Dispatchers.IO) { this: CoroutineScope
    if (entity.usuario.perfil == Perfil.ENCORDADOR) {
        launch { this: CoroutineScope
            tareasKtorFitRepository.save(entity)
            println(Json.encodeToString(ResponseSuccess( code: 201, entity.toTareaDto())))
        }

        launch { this: CoroutineScope
            tareasKtorFitRepository.uploadTarea(entity)
            println(Json.encodeToString(ResponseSuccess( code: 201, entity.toTareaDto())))
        }

        joinAll()
    } else System.err.println(
        Json.encodeToString(
            ResponseFailure(
                code: 400,
                message: "No ha sido posible almacenar $entity || El usuario debe de ser de tipo ${Perfil
                .ENCORDADOR.name}"
            )
        )
    )
    return@withContext entity
}
```

```
Mario
suspend fun getTareaById(id: Id<Tarea>): Tarea? {
    val response = tareasKtorFitRepository.findByID(id)

    if (response == null) {
        System.err.println(Json.encodeToString(ResponseFailure( code: 404,  message: "Tarea not found")))
    } else println(Json.encodeToString(ResponseSuccess( code: 200, response.toTareaDto())))

    return response
}

Mario +1
suspend fun deleteTarea(entity: Tarea): Boolean {

    println(Json.encodeToString(ResponseSuccess( code: 200, entity.toTareaDto())))
    return tareasKtorFitRepository.delete(entity)
}
}
```

Koin

En algunas clases anteriores se ha podido observar que tenían unas etiquetas especiales, tales como @Single. Estas etiquetas pertenecen a la tecnología de inyección de dependencias Koin. Para que las dependencias funcionen, se debe crear una clase especial en la que se indica qué clases se inyectan y dónde.



App

Para que Koin nos funcione correctamente en nuestra aplicación, se ha creado una clase en la que se implementa un componente de Koin para poder inyectar los controladores y darnos la posibilidad de hacer el CRUD para nuestro proyecto.

La clase es muy extensa, por lo que se enseñarán los elementos fundamentales para poder hacer el CRUD.

```
Sebastián Mendoza Acosta +1
fun run(): Unit = runBlocking { this: CoroutineScope
    val limpiar = launch { this: CoroutineScope
        limpiarDatos()
    }
    limpiar.join()
```

```
Sebastián Mendoza Acosta +1
suspend fun limpiarDatos() = withContext(Dispatchers.IO) { this: CoroutineScope
    logger.debug { "Borrando datos de la base de datos" }
    if (MongoDbManager.database.getCollection<Producto>().countDocuments() > 0) {
        MongoDbManager.database.getCollection<Producto>().drop()
    }
    if (MongoDbManager.database.getCollection<Adquisicion>().countDocuments() > 0) {
        MongoDbManager.database.getCollection<Adquisicion>().drop()
    }
    if (MongoDbManager.database.getCollection<Encordar>().countDocuments() > 0) {
        MongoDbManager.database.getCollection<Encordar>().drop()
    }
    if (MongoDbManager.database.getCollection<Personalizar>().countDocuments() > 0) {
        MongoDbManager.database.getCollection<Personalizar>().drop()
    }
    if (MongoDbManager.database.getCollection<Encordadora>().countDocuments() > 0) {
        MongoDbManager.database.getCollection<Encordadora>().drop()
    }
    if (MongoDbManager.database.getCollection<Personalizadora>().countDocuments() > 0) {
        MongoDbManager.database.getCollection<Personalizadora>().drop()
    }
    if (MongoDbManager.database.getCollection<Usuario>().countDocuments() > 0) {
        MongoDbManager.database.getCollection<Usuario>().drop()
    }
    if (MongoDbManager.database.getCollection<Tarea>().countDocuments() > 0) {
        MongoDbManager.database.getCollection<Tarea>().drop()
    }
    if (MongoDbManager.database.getCollection<Pedido>().countDocuments() > 0) {
        MongoDbManager.database.getCollection<Pedido>().drop()
    }

}
```

```
//Controladores
val productoController: ProductoController by inject()
val adquisicionController: AdquisicionController by inject()
val encordarController: EncordarController by inject()
val personalizarController: PersonalizarController by inject()
val maquinaEncordadoraController: MaquinaEncordadoraController by inject()
val maquinaPersonalizadoraController: MaquinaPersonalizadoraController by inject()
val apiController: APIController by inject()
val pedidoController: PedidoController by inject()
```

```
//Listas
val productosList = mutableListOf<Producto>()
val adquisicionList = mutableListOf<Adquisicion>()
val encordadosList = mutableListOf<Encordar>()
val personalizacionesList = mutableListOf<Personalizar>()
val maquinaEncordadoraList = mutableListOf<Encordadora>()
val maquinaPersonalizadoraList = mutableListOf<Personalizadora>()
```

```
//Usuarios
val listadoUsers = apiController.getAllUsuariosApi().toList().toMutableList()
listadoUsers[0].raqueta = getRaquetasInit()
listadoUsers[9].perfil = Perfil.ENCORDADOR
listadoUsers[8].perfil = Perfil.ADMIN

// Create CACHE-MONGO
listadoUsers.forEach { it: Usuario
    apiController.saveUsuario(it)
    println(it)
}
// FindAll CACHE-MONGO
apiController.getAllUsuariosCache().collect { it: Usuario
    println(it)
}
apiController.getAllUsuariosMongo().collect { it: Usuario
    println(it)
}
```

```
// FindById → Cache → Mongo
val userId = apiController.getUsuarioById(listadoUsers[1].id)
val userId2 = apiController.getUsuarioById(listadoUsers[2].id)
println(userId)
println(userId2)
// Update → Cache && Mongo
userId?.let { it: Usuario
    it.name = "Solid Snake"
    apiController.saveUsuario(it) ^let
}
// Delete → Cache && Mongo
userId2?.let { it: Usuario
    apiController.deleteUsuario(it)
}
```

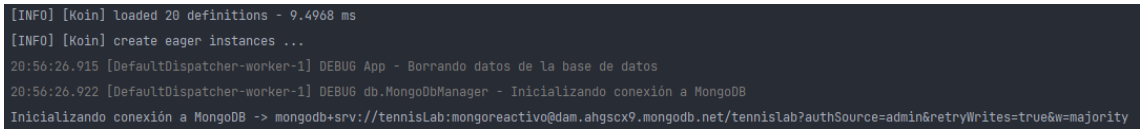
Main

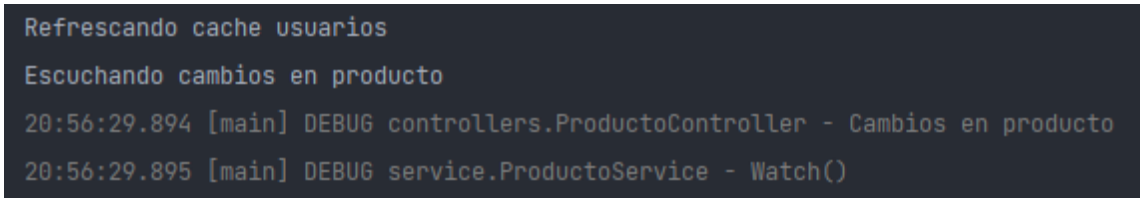
Para que la inyección de dependencias funcione, el *Main* tiene un configuración especial. En este caso, necesitamos implementar el módulo donde se definen las inyecciones de dependencia, e incluir la clase donde tenemos la lógica del CRUD de nuestro proyecto.

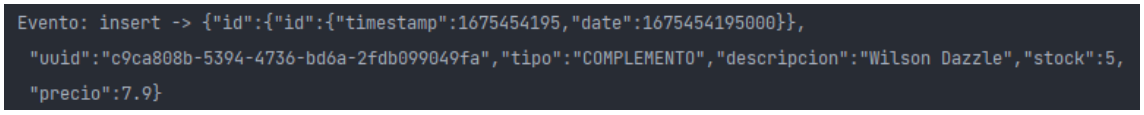
```
Sebastián Mendoza Acosta
fun main() {
    startKoin { this: KoinApplication
        printLogger()
        modules(
            DiModule().run { myModule }
        )
    }
    AppMongo().run()
}
```

Ejecución de proyecto

Ahora enseñaremos parte de la ejecución para enseñar el funcionamiento de la aplicación.







Test

Una de las partes más importantes a la hora de realizar un proyecto es la comprobación de su funcionamiento, es decir, “testear” las funciones de la aplicación. Para este proyecto se ha usado Mock. Enseñaremos algunas tests:

```
Mario
@ExtendWith(MockKExtension::class)
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
internal class AdquisicionRepositoryImplTest {
    private val adquisicion = Adquisicion(
        cantidad = 1,
        producto = Producto(
            tipo = Tipo.RAQUETA,
            descripcion = "Babolat Pure Air",
            stock = 3,
            precio = 345.95
        ),
    )

    @InjectMockKs
    private lateinit var adquisicionRepository: AdquisicionRepositoryImpl

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

```
    Mario
    @OptIn(ExperimentalCoroutinesApi::class)
    @BeforeAll
    fun setUp() = runTest { this: TestScope
        MongoDbManager.database.getCollection<Adquisicion>().drop()
    }

    Mario
    @OptIn(ExperimentalCoroutinesApi::class)
    @AfterAll
    fun tearDown() = runTest { this: TestScope
        MongoDbManager.database.getCollection<Adquisicion>().drop()
    }

    Mario
    @OptIn(ExperimentalCoroutinesApi::class)
    @BeforeEach
    fun beforeEach() = runTest { this: TestScope
        adquisicionRepository.save(adquisicion)
    }

    Mario
    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun findAll() = runTest { this: TestScope
        val res = adquisicionRepository.findAll().toList()

        assertAll(
            { assertEquals( expected: 1, res.size) }
        )
    }
```

Ejecución de test