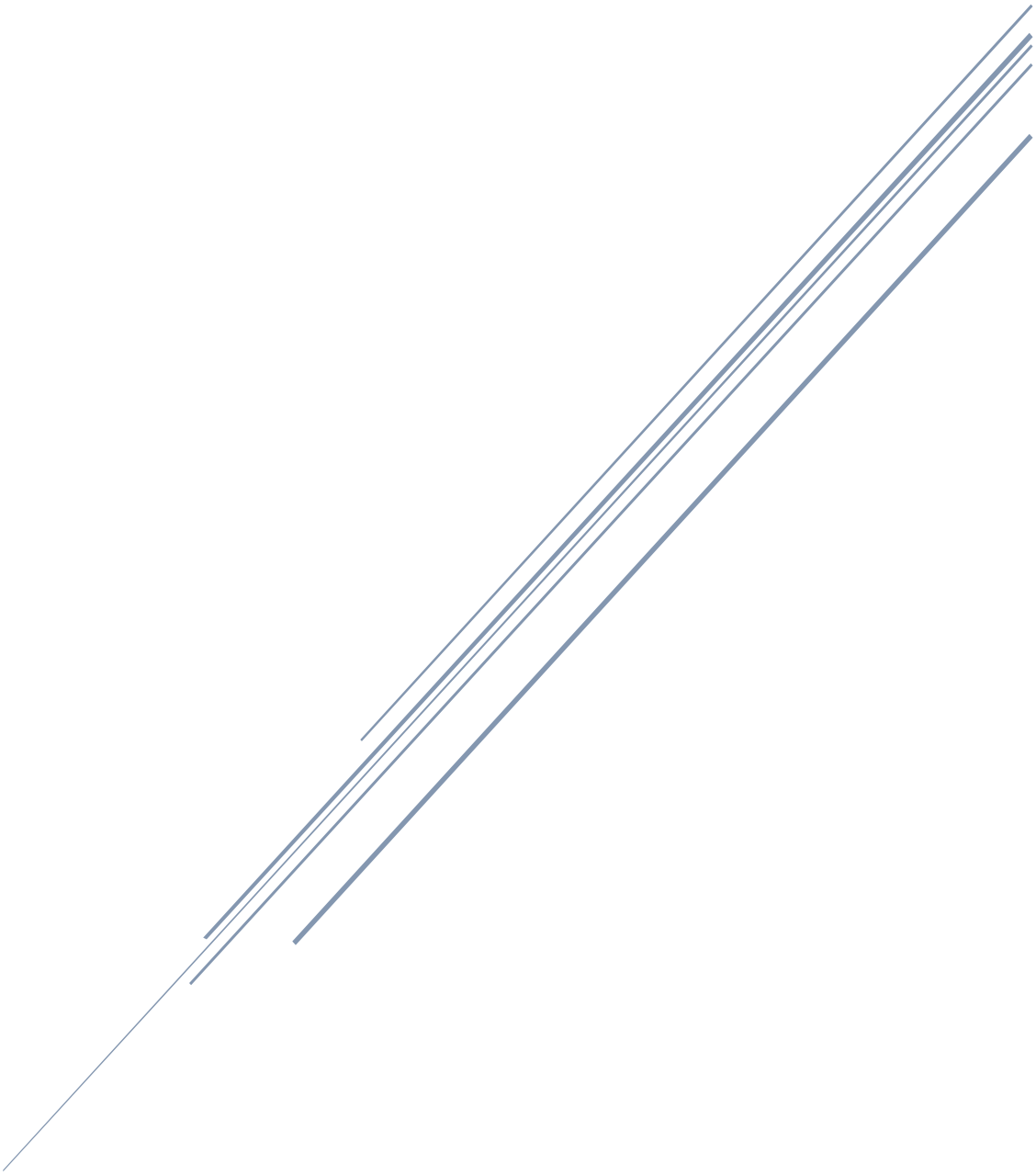
TENNISLAB: MONGODB + SPRING DATA REACTIVO

Mongo Atlas – KtorFit – Caché – Spring Data



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.

Finalmente, aplicamos un conocido framework en el mundo de Java, *Spring,* y para ser más exactos, *Spring Data*, que nos permitirá trabajar de una forma específica.

La unión de estas 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].

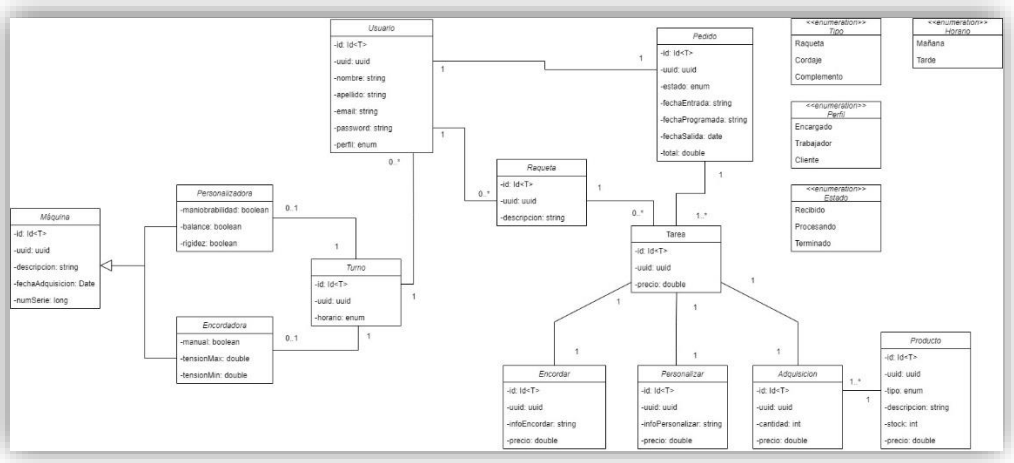
Para terminar, y no menos importante, la caché. En esto caso se ha utilizado **Cache4K**: cache4k proporciona una caché de clave-valor en memoria simple para Kotlin Multiplatform, con soporte para desalojos basados en el tiempo (expiración) y basados en el tamaño.[3]

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

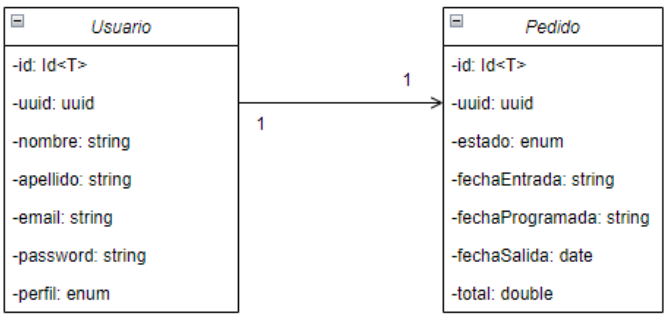
Antes de empezar a picar código, una de las partes más importantes del desarrollo de un proyecto es la creación del diagrama de clases. Este diagrama define las relaciones que van a tener las clases entre sí dando un panorama de cómo va a funcionar el proyecto. Hay diversos diagramas que pueden explicar 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:

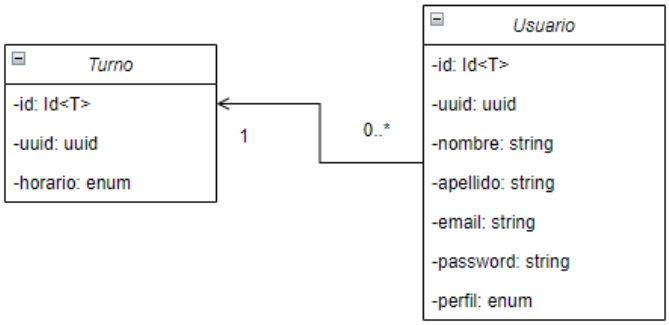
1. Usuario – Pedido.



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,

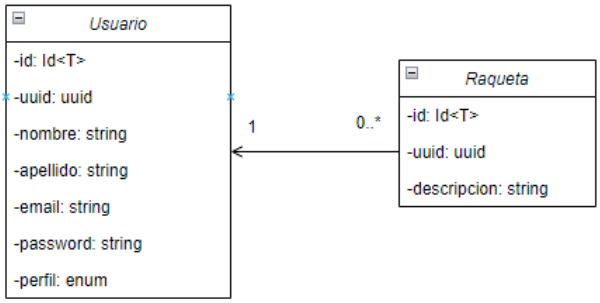
es decir, una de las clases estará en la otra, pero no viceversa. En este caso, se ha optado a que *Pedido* tenga un *Usuario*, pero que no suceda en el otro sentido. El motivo de esta elección es que nos parecía más sencillo consultar el pedido de un usuario con la identificación de *Usuario* que tuviera 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*.

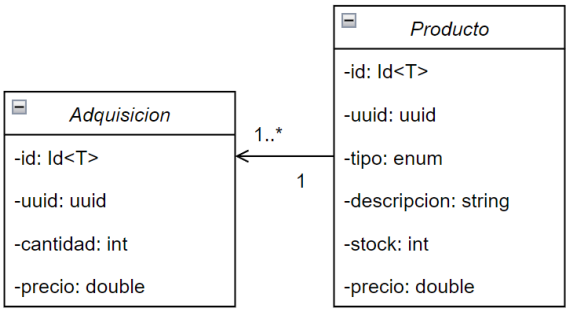
3. Usuario – Raqueta.



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*.

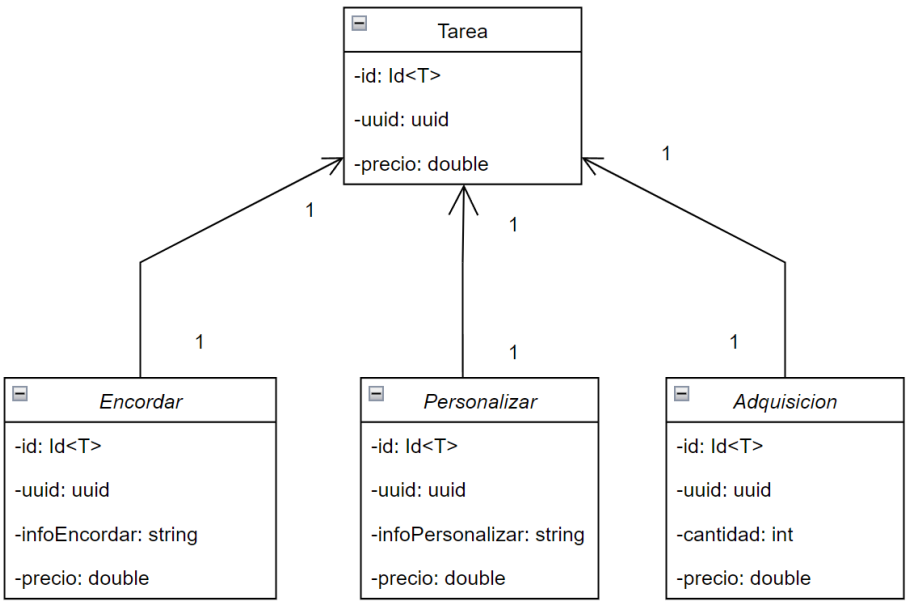


Como podemos observar en la imagen anterior, tenemos una relación de uno a muchos, es decir, un producto puede ser adquirido una o varias veces, y una adquisición está compuesta por un único producto. En este caso, la unidireccionalidad será en el sentido de la adquisición, es decir, *Producto* estará 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

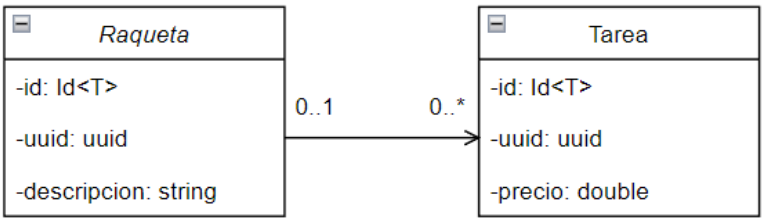
Una de las partes centrales del proyecto son las tareas, ya que ésta tiene una relación muy estrecha con varias clases del proyecto. Esto se debe a que las tareas comprenden gran parte de la lógica de la aplicación con respecto a lo que se pide en el proyecto. Según este, las tareas están compuestas 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



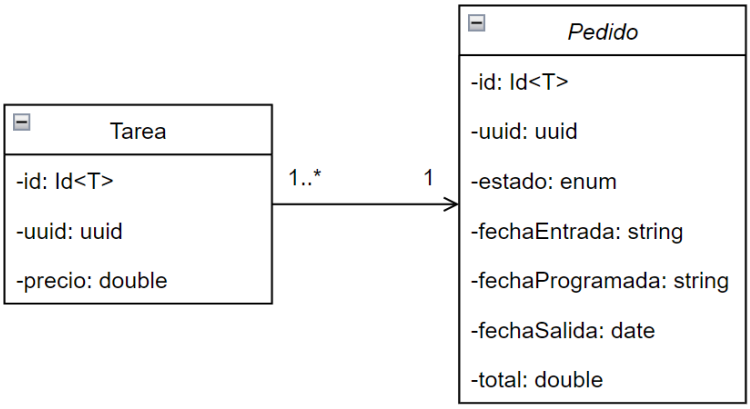
Una de las primeras cosas a destacar es que las clases *Encordar*, *Personalizar* y *Adquisición* se encontrarán unidireccionalmente en *Tarea*, esto quiere decir que no tendremos la clase *Tarea* en las otras tres, puesto que nuestra lógica nos dictó que no merecía la pena agregar la información de *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.

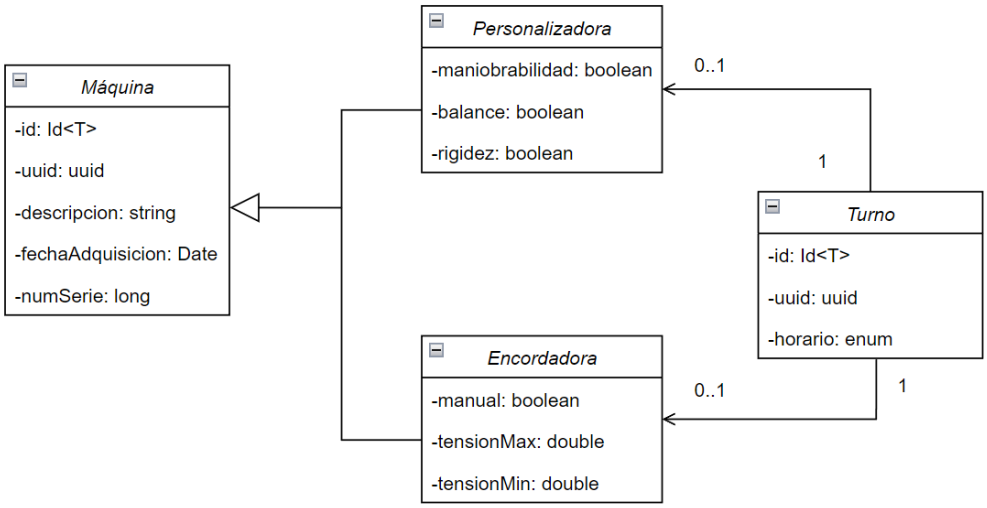
3. Tarea – Pedido



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:

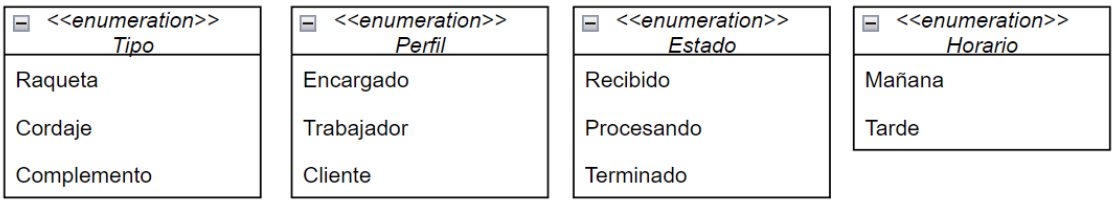


Por un lado, tenemos una herencia de *Máquina* a las clases *Personalizadora* y *Encordadora*. Teníamos la elección de hacer dos máquinas por separado, pero quisimos ver si podíamos trabajar con herencia en el proyecto. Teniendo dichas máquinas por separado, estas se relacionan cada una con la clase *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. Spring Data. Dependencias base al usar Spring Initializr y elegir la dependencia *Spring Data Reactive MongoDB*

```
dependencies { this: DependencyHandlerScope
    implementation("org.springframework.boot:spring-boot-starter-data-mongodb-reactive")
    implementation("io.projectreactor.kotlin:reactor-kotlin-extensions")
    implementation("org.jetbrains.kotlin:kotlin-reflect")
    implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-reactor")
```

2. Test. Dependencia de Spring para realizar pruebas.

```
    testImplementation("org.springframework.boot:spring-boot-starter-test")
    testImplementation("io.projectreactor:reactor-test")
```

3. Serialization-Json. Serializador específico para Kotlin.

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

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

```
// Log
implementation("io.github.microutils:kotlin-logging-jvm:3.0.4")
```

5. BCrypt. Encriptador de contraseñas.

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

6. 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")
```

7. 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")
```

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

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

9. Jackson.

```
// Jackson
implementation("com.fasterxml.jackson.module:jackson-module-kotlin:2.14.+")
```

10. Documentación para Kotlin.

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

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

```
Sebastián Mendoza Acosta +1
@Document("productos")
data class Producto(
    @Id
    val id: ObjectId = ObjectId.get(),
    val uuid: UUID = UUID.randomUUID(),
    val tipo: Tipo,
    val descripcion: String,
    val stock: Int,
    var precio: Double
){
    Sebastián Mendoza Acosta +1
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}
Sebastián Mendoza Acosta
enum class Tipo { RAQUETA, CORDAJE, COMPLEMENTO }
```

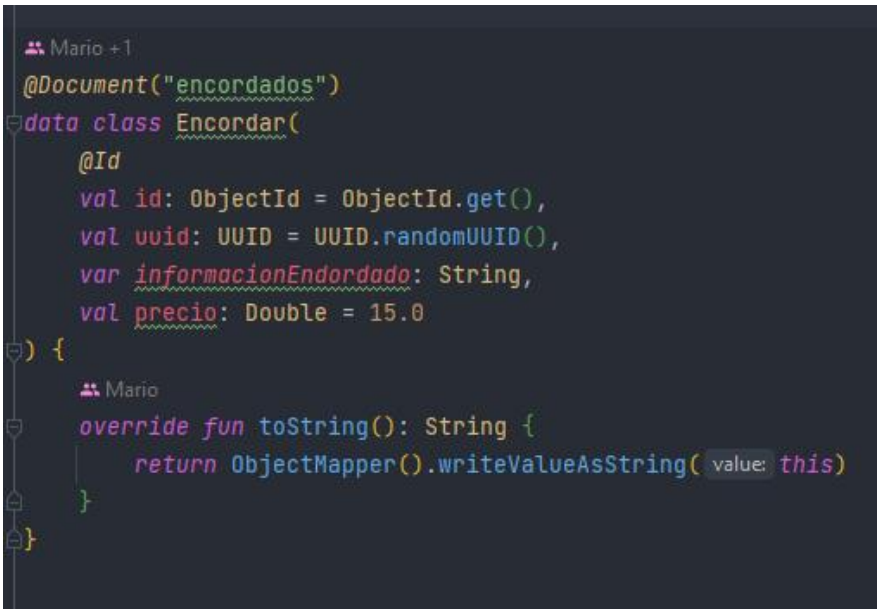
A comparación de la versión sin Spring, aquí usaremos dos etiquetas principalmente: @Document, @Id. La falta de uso de @Serializable se debe a que su uso es trasladado a un uso exclusivo en los *DTO* de cada modelo.

Adquisición

```
Mario
@Document("adquisiciones")
data class Adquisicion(
    @Id
    val id: ObjectId = ObjectId.get(),
    val uuid: UUID = UUID.randomUUID(),
    var cantidad: Int,
    val producto: Producto,
    var precio: Double = CalculoPrecioTarea.calculatePrecio(producto.precio, data2: null, data3: null) * cantidad
) {
    Mario
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
```

Lo más importante a mencionar es el uso de una clase objeto con la función *calculatePrecio,* que, usando la cantidad del propio modelo, devuelve el total de la adquisición.

Encordar



El precio siempre deberá de ser el mismo, 15.0

Personalizar

Igual que en el caso anterior, el precio será siempre de 60.0

Tarea

```
Sebastián Mendoza Acosta
@Document("tareas")
data class Tarea(
    val id: ObjectId = ObjectId.get(),
    val uuid: UUID = UUID.randomUUID(),
    val adquisicion: Adquisicion? = null,
    val personalizar: Personalizar? = null,
    val encordar: Encordar? = null,
    var precio: Double = CalculoPrecioTarea.calculatePrecio(
        adquisicion?.precio,
        personalizar?.precio,
        encordar?.precio
    ),
    var usuario: Usuario,
    var raqueta: Raqueta? = null
) {
    Sebastián Mendoza Acosta
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}
```

Una tarea puede estar compuesta de una adquisición, una personalización y de un encordado, el precio, usando un método que se mencionó anteriormente, *calculatePrecio,* calcularemos el precio de la tarea al completo; esta necesita un usuario (trabajador) y una posible raqueta.

Pedido

```
Sebastián Mendoza Acosta
@Document
data class Pedido(
    @Id
    val id: ObjectId = ObjectId.get(),
    val uuid: UUID = UUID.randomUUID(),
    var estadoPedido: EstadoPedido,
    val fechaEntrada: String,
    val fechaProgramada: String,
    var fechaSalida: String? = null,
    var cliente: Usuario,
    var tareas: List<Tarea>,
    val precio: Double = tareas.sumOf { it.precio }
) {
    Sebastián Mendoza Acosta
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}

Sebastián Mendoza Acosta
enum class EstadoPedido { RECIBIDO, PROCESANDO, TERMINADO }
```

Contamos con un enum class a usar para el estado del pedido, una serie de fechas; la única preparada a Null será la fecha de Salida, que será actualizada cuando el pedido se envié. Tendrá asociado un usuario (cualquier tipo), y tendrá una lista de tareas.

Usuario

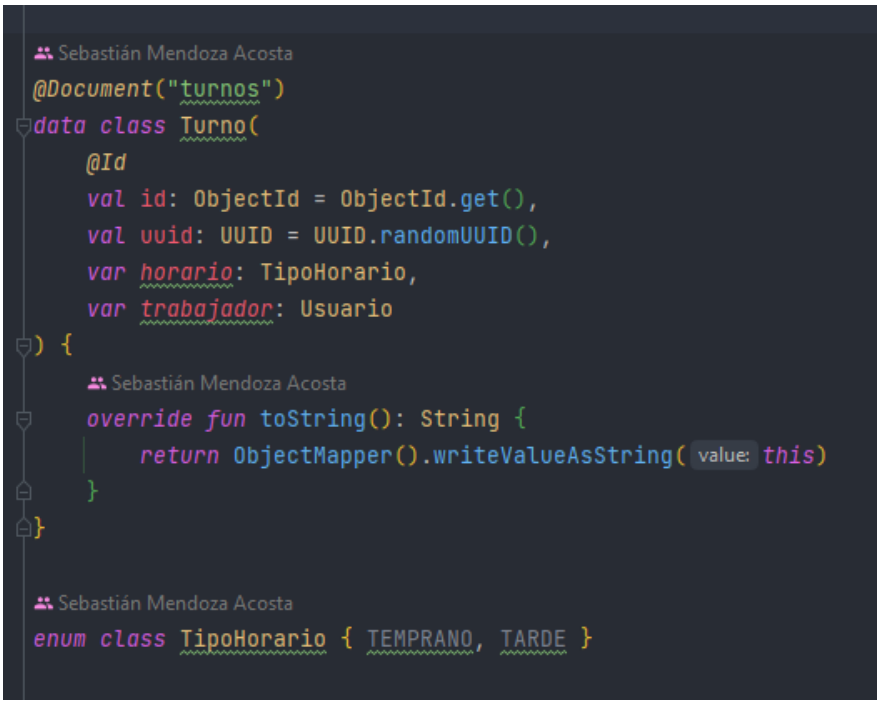
A diferencia de la práctica anterior, esta vez decidimos juntar a todos los usuarios en una sola clase sin herencia, y hacer uso de un enum class para diferenciar entre los tres tipos de usuario que existen: administrador, encordador / trabajador y cliente. Esta decisión fue tomada debido a la falta de características distintas entre los usuarios; no había nada que los diferenciara, a excepción de los permisos que podrían tener en el sistema, por ejemplo, no es posible que un cliente forme parte de una tarea o de un turno.

La contraseña ha sido almacenada usando Bcrypt, así permanece protegida en la base de datos.

Raqueta

En este caso, decidimos simplificar la raqueta por la decisión de diseño de embeberla en el propio usuario, así que solo cuenta con un ID, un UUID y una simple descripción.

Turno



Es un caso parecido al de las raquetas, esta vez, los turnos se han embebido en las propias máquinas, en estas se indicará el horario que tendrán y el trabajador, además de los identificadores habituales.

Máquina

Este caso es uno de los pocos que se ha quedado prácticamente igual, contamos con un modelo base que todas las máquinas existentes heredaran y que luego implantaran sus propias variables únicas.

Encordadora y Personalizadora

```
Sebastián Mendoza Acosta
@Document("encordadoras")
class Encordadora(
    id: ObjectId = ObjectId.get(),
    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
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}
```

```
Sebastián Mendoza Acosta
@Document
class Personalizadora(
    id: ObjectId = ObjectId.get(),
    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
    override fun toString(): String {
        return ObjectMapper().writeValueAsString( value: this)
    }
}
```

Ambos modelos cuentan con la misma base, pero tienen campos específicos.

La decisión de poner el turno a null de base es para poder mostrar que la máquina esta disponible para su uso.

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>()
```

Esta clase será usada en los controladores, y servirá para mostrar un mensaje con un código durante la ejecución del programa.

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:

UsuarioDto

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

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

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

"Por una decisión de diseño, cuando se conecte un usuario este será por defecto un cliente".

ProductoDto

```
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

Nuestra aplicación debe de guardar la información en MongoDB, finalmente elegimos que esta fuera mediante Mongo Atlas. Spring se encarga de realizar la conexión, mientras se le proporcione la información de forma correcta.

Archivo Properties:

```
application.properties
1  spring.data.mongodb.uri=mongodb+srv://tennisLab:mongoreactivo@dam.ahgscx9.mongodb.net/tennisLabSpring
2
```

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 realizar la cache sobre usuarios, se eligió usar *Cache4k.*

```kotlin
/**
 * Clase donde se configura las características de la caché
 */
Mario
class UsuariosCache {

    val refreshTime = 60000 // 1 minuto

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

ProductoService

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

```kotlin
private val logger = KotlinLogging.logger { }

Sebastián Mendoza Acosta
@Service
class ProductoService
@Autowired constructor(
    private val reactiveMongoTemplate: ReactiveMongoTemplate
) {
    Sebastián Mendoza Acosta
    fun watch(): Flow<ChangeStreamEvent<Producto>> {
        logger.info { "watch()" }
        return reactiveMongoTemplate.changeStream(
            collectionName: "productos",
            ChangeStreamOptions.empty(),
            Producto::class.java
        ).asFlow()
    }
}
```

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

El paquete *Repositories* alberga la configuración de cómo se harán los CRUD (*Create, Read, Update and Delete*) en nuestro proyecto. Al usar Spring, esta parte se encuentra tremendamente simplificada, a excepción de los repositorios que hagan uso de KtorFit o de la cache. En esta ocasión, se mostrarán un par de ejemplos de repositorios normales de Spring, y los específicos que hemos tenido que diseñar a mano.

AdquisicionRepository

```
Mario
interface AdquisicionRepository : CoroutineCrudRepository<Adquisicion, ObjectId> {
}
```

PedidoRepository

```
Sebastián Mendoza Acosta
interface PedidoRepository : CoroutineCrudRepository<Pedido, ObjectId> {
}
```

UsuariosCacheRepository

```
Mario
@Repository
class UsuariosCacheRepository {

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

    private var listaBusquedas = mutableListOf<Usuario>()

    Mario
    init {
        refreshCache()
    }

    Mario
    suspend fun findAll(): Flow<Usuario> {
        println("\tFindALL CACHE")
        return cacheUsuarios.cache.asMap().values.asFlow()
    }

    Mario
    suspend fun delete(entity: Usuario): Boolean {
        println("\tDelete CACHE")
        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
suspend fun save(entity: Usuario): Usuario {
    println("\tSave CACHE")
    listaBusquedas.add(entity)
    return entity
}

Mario
suspend fun findById(id: ObjectId): Usuario? {
    println("\tfindBYID CACHE")
    var usuario: Usuario? = null

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

Mario
private final 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())
        }
    }
}
```

En este repositorio es donde se mantendrá actualizada la cache de usuarios con el uso de una corrutina. También, preparamos los métodos que usaremos en el controlador para realizar las operaciones CRUD necesarias.

UsuariosKtorFitRepositoy

```
@Repository
class UsuariosKtorFitRepository {

    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()
    }
}
```

Aquí es donde obtendremos a los usuarios del servicio externo, haciendo uso de KtorFit

TareasKtorFitRepository

```
Mario
@Repository
class TareasKtorFitRepository {

    private val client by lazy { KtorFitClient.instance }

    Mario
    suspend fun uploadTarea(entity: Tarea): Tarea = withContext(Dispatchers.IO) { this: CoroutineScope
        logger.info { "Subiendo tarea $entity al historico" }

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

Mismo caso que en el anterior, a excepción de la operación, en este caso, se trata de subir una tarea (TareaDto) al servicio externo.

Controladores

Los controladores se encargan de realizar la función que sea necesaria del repositorio / repositorios que tengan inyectados, usando *@Autowired*, así como de comprobar que X valor sea correcto para la acción a ejecutar. Aquí haremos uso de la clase modelo "*Response*".

En casos específicos, como el de usuarios o tareas, tendremos la opción de hacer varias operaciones a la vez dentro de un mismo método; cuando guardemos una tarea en MongoDb, a su vez podremos almacenarla en el servicio externo.

Mostraremos dos ejemplos:

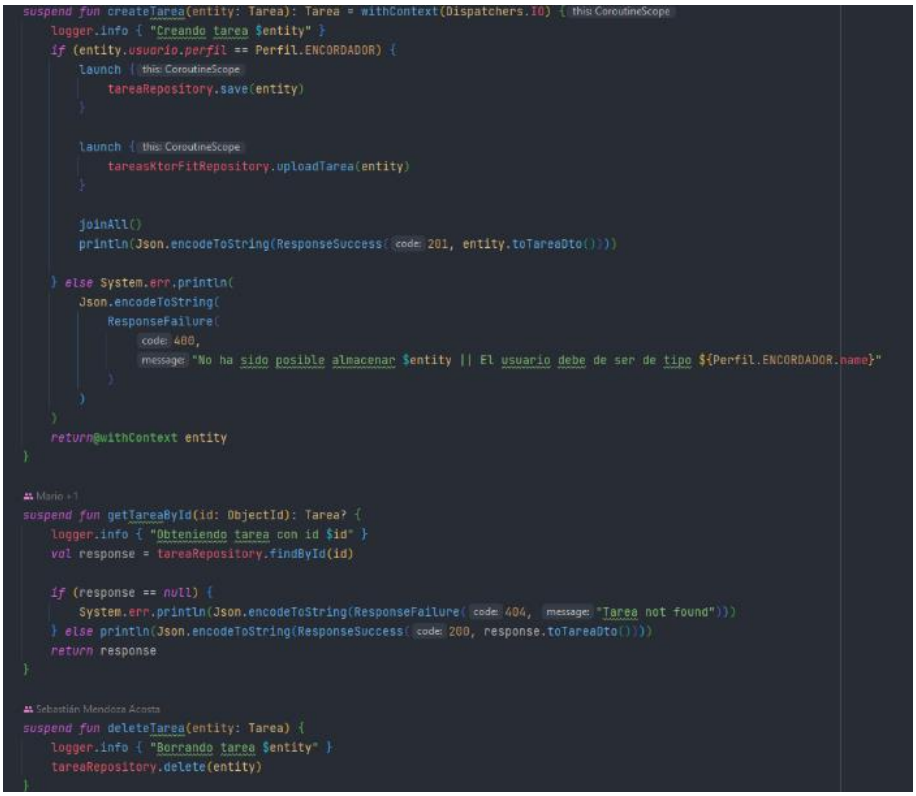
PedidoController

```
@Controller
class PedidoController
@Autowired constructor(
    private val pedidoRepository: PedidoRepository
) {
    Mario +1
    fun getPedidos(): Flow<Pedido> {
        logger.info { "Obteniendo pedidos" }
        val response = pedidoRepository.findAll()

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

    Mario +1
    suspend fun createPedido(entity: Pedido): Pedido {
        logger.info { "Creando pedido $entity" }
        val response = pedidoRepository.save(entity)

        println(Json.encodeToString(ResponseSuccess( code: 201, response.toPedidoDto())))
        return response
    }

    Mario +1
    suspend fun getPedidoById(id: ObjectId): Pedido? {
        logger.info { "Obteniendo pedido con id $id" }
        val response = pedidoRepository.findById(id)

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

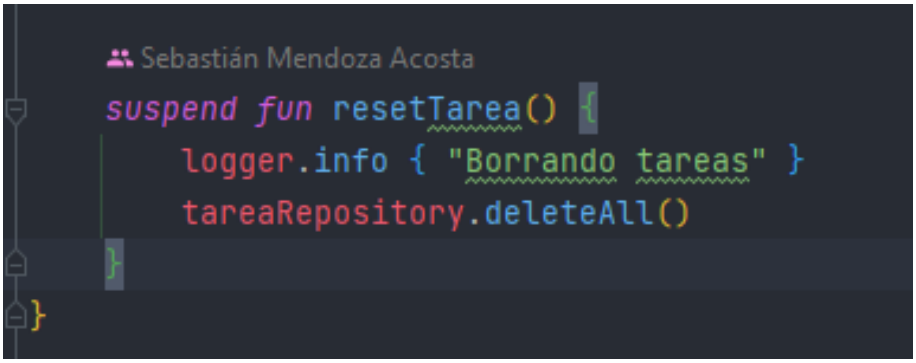
```
Sebastián Mendoza Acosta +1
suspend fun deletePedido(entity: Pedido) {
    logger.info { "Borrando pedido $entity" }

    println(Json.encodeToString(ResponseSuccess( code: 200, entity.toPedidoDto())))
    pedidoRepository.delete(entity)
}

Sebastián Mendoza Acosta
suspend fun resetPedidos() {
    logger.info { "Borrando pedidos" }
    pedidoRepository.deleteAll()
}
```

TareaController

```
/**
 * Controlador al que se le pasan dos repositorios, uno se encarga de realizar CRUD de MongoDB y el otro, a subir las tareas
 * a un servicio externo.
 *
 * @property tareaRepository
 * @property tareasKtorFitRepository
 */
Mario +1
@Controller
class TareaController
@Autowired constructor(
    private val tareaRepository: TareaRepository,
    private val tareasKtorFitRepository: TareasKtorFitRepository
) {
    Mario +1
    fun getTareas(): Flow<Tarea> {
        logger.info { "Obteniendo tareas" }
        val response = tareaRepository.findAll()

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



*"Se puede observar el uso de las corrutinas en el método **createTarea**"*

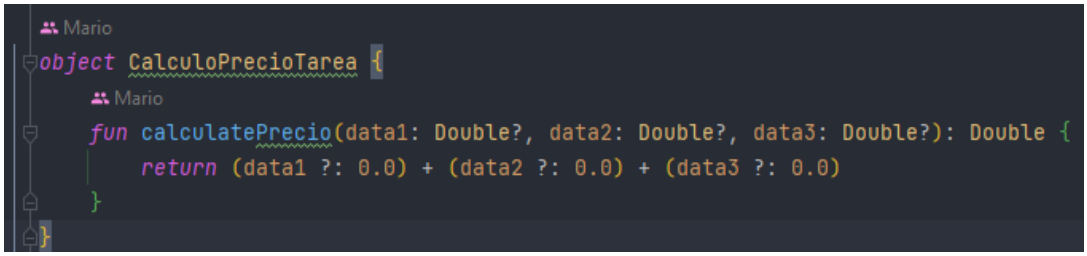


Utils

CalculoPrecioTarea

Esta clase es usada para el calculo del precio en dos modelos: Tarea y Adquisición

Esta función recibe por parámetros hasta tres Double.



Cifrador

Clase usada para cifrar la contraseña de los usuarios, esta función recibe por parámetro un String.

```
Mario
object Cifrador {
    Mario
    fun codifyPassword(password: String): ByteArray {
        return Bcrypt.hash(password, saltRounds: 12)
    }
}
```

Ejecución

En la clase main, prepararemos todos los controladores y realizaremos las operaciones CRUD de cada repositorio a través de su controlador; además, y en la mayoría de los casos, haremos uso de un archivo *"Data"* que contendrá valores de prueba para las operaciones.

Además, limpiaremos la base de datos, siempre, antes de la ejecución, para tener únicamente los datos actualizados.

Data

```
Sebastián Mendoza Acosta
fun getProductoInit() = listOf(
    Producto(
        tipo = Tipo.RAQUETA,
        descripcion = "Babolat Pure Air",
        stock = 3,
        precio = 279.95
    ),
    Producto(
        tipo = Tipo.COMPLEMENTO,
        descripcion = "Wilson Dazzle",
        stock = 5,
        precio = 7.90
    )
)

Mario
fun getAdquisicionInit() = listOf(
    Adquisicion(
        cantidad = 1,
        producto = getProductoInit()[0],
    ),
    Adquisicion(
        cantidad = 2,
        producto = getProductoInit()[1],
    ),
)

Mario
fun getEncordaciones() = listOf(
    Encordar(
        informacionEndordado = "Dato1"
    ),
    Encordar(
        informacionEndordado = "Dato2"
    )
)
```

Main

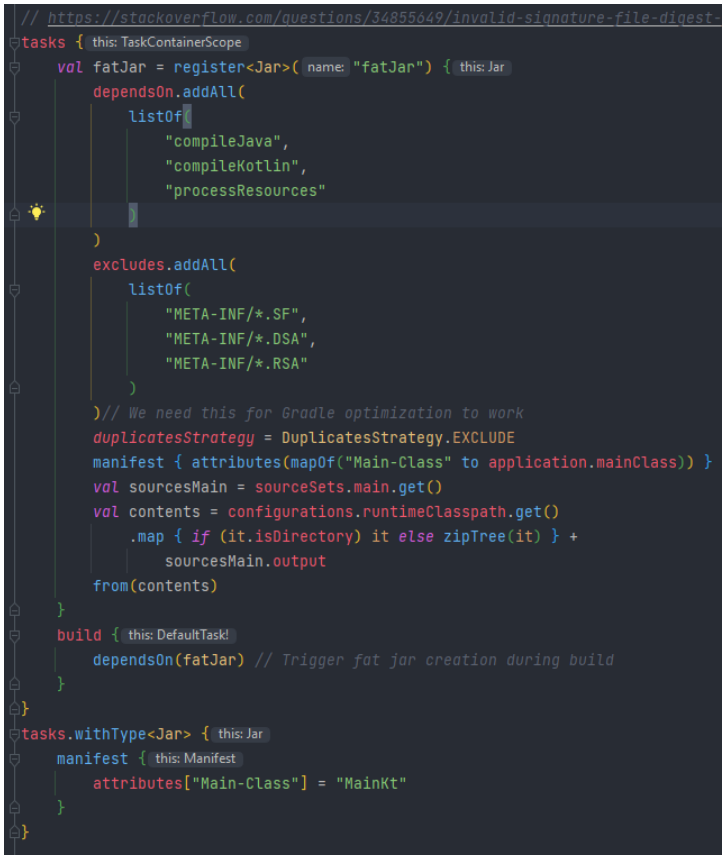
```
Sebastián Mendoza Acosta +1
@SpringBootApplication
class MongoDbSpringDataReactivoApplication
@Autowired constructor(
    private val productoController: ProductoController,
    private val usuarioController: UsuarioController,
    private val encordarController: EncordarController,
    private val personalizarController: PersonalizarController,
    private val adquisicionController: AdquisicionController,
    private val tareaController: TareaController,
    private val pedidoController: PedidoController,
    private val encordadoraController: MaquinaEncordadoraController,
    private val personalizadoraController: MaquinaPersonalizadoraController
) : CommandLineRunner {
    Sebastián Mendoza Acosta +1
    override fun run(vararg args: String?): Unit = runBlocking { this: CoroutineScope
        // Listas
        val productosList = mutableListOf<Producto>()
        val encordacionesList = mutableListOf<Encordar>()
        val personalizacionesList = mutableListOf<Personalizar>()
        val adquisicionesList = mutableListOf<Adquisicion>()
        val maquinaEncordadoraList = mutableListOf<Encordadora>()
        val maquinaPersonalizadoraList = mutableListOf<Personalizadora>()

        // Obtención de datos
        val productosInit = getProductoInit()
        val encordacionesInit = getEncordaciones()
        val personalizacionesInit = getPersonalizaciones()
        val adquisicionInit = getAdquisicionInit()
        val encordadoraInit = getEncordadorasInit()
        val personalizadoraInit = getPersonalizadorasInit()

        val clear = launch { this: CoroutineScope
            productoController.resetProductos()
            usuarioController.resetUsuariosMongo()
            encordarController.resetEncordaciones()
            personalizarController.resetPersonalizaciones()
            adquisicionController.resetAdquisiciones()
            tareaController.resetTarea()
```

```
    val productosListener = launch { this: CoroutineScope
        productoController.watchProductos()
            .onStart { println("Escuchando cambios en Producto...") }
            .collect { println("Evento: ${it.operationType?.value} -> Producto: ${it.body}") }
    }

    // Usuarios
    println("Usuarios")
    val usuariosList = usuarioController.getAllUsuariosApi().toList().toMutableList()
    usuariosList[0].raqueta = getRaquetasInit()
    usuariosList[9].perfil = Perfil.ENCORDADOR
    usuariosList[8].perfil = Perfil.ADMIN

    // Create CACHE-MONGO
    usuariosList.forEach { it: Usuario
        usuarioController.createUsuario(it)
        println(it)
    }
    // FindAll CACHE-MONGO
    usuarioController.getAllUsuariosCache().collect { it: Usuario
        println(it)
    }
    usuarioController.getAllUsuariosMongo().collect { it: Usuario
        println(it)
    }
    // FindById -> Cache -> Mongo
    val userId = usuarioController.getUsuarioById(usuariosList[1].id)
    userId?.let { println(it) }
    // Update -> Cache && Mongo
    userId?.let { it: Usuario
        it.name = "Solid Snake"
        usuarioController.createUsuario(it) ^let
    }
    // Delete -> Cache && Mongo
    val userDelete = usuarioController.getUsuarioById(usuariosList[2].id)
    userDelete?.let { it: Usuario
        usuarioController.deleteUsuario(it)
    }
```

Jar

Hemos configurado el proyecto para que sea posible la creación de un archivo de extensión *jar*.

"Para el correcto funcionamiento del jar, este deberá ser desplazado a la raíz del programa."



"Mención a la solución encontrada por un fallo al tratar de ejecutar por primera vez: "

https://stackoverflow.com/questions/34855649/invalid-signature-file-digest-for-manifest-main-attributes-exception-while-tryin

Test

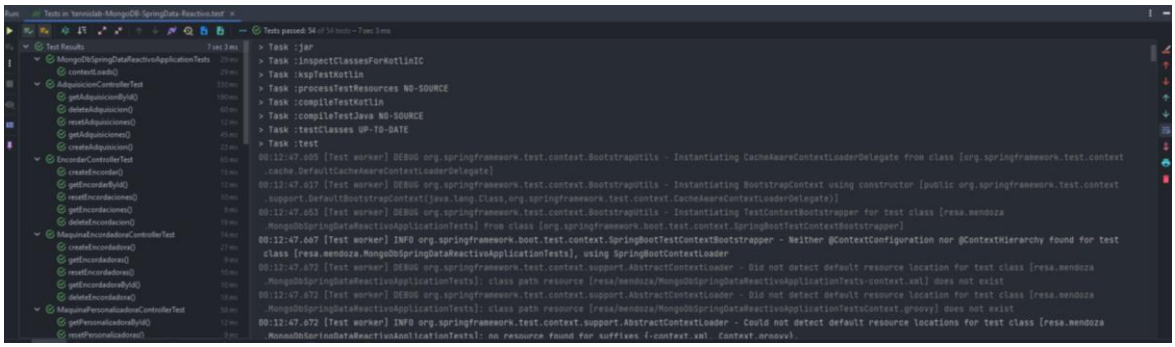
Para realizar las pruebas hemos usado herramientas propias de Spring, Junit5 y Mockk.

Adjuntamos ejemplo y la muestra de todos los test en funcionamiento:

MaquinaPersonalizadoraControllerTest

```kotlin
@ExtendWith(MockKExtension::class)
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@SpringBootTest
internal class MaquinaPersonalizadoraControllerTest {

    @MockK
    private lateinit var maquinaPersonalizadoraRepository: MaquinaPersonalizadoraRepository

    @InjectMockKs
    private lateinit var maquinaPersonalizadoraController: MaquinaPersonalizadoraController

    private val personalizadora = Personalizadora(
        descripcion = "Toshiba ABC",
        fechaAdquisicion = LocalDate.now().toString(),
        numSerie = 540L,
        maniobrabilidad = true,
        balance = false,
        rigidez = false
    )

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

    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun getPersonalizadoras() = runTest { this: TestScope
        every { maquinaPersonalizadoraRepository.findAll() } returns flowOf(personalizadora)
        val res = maquinaPersonalizadoraController.getPersonalizadoras().toList()
        assertAll(
            { assertEquals( expected: 1, res.size) }
        )
        verify(exactly = 1) { maquinaPersonalizadoraRepository.findAll() }
    }
```

```kotlin
    fun createPersonalizadora() = runTest { this: TestScope
        coEvery { maquinaPersonalizadoraRepository.save(any()) } returns personalizadora
        val res = maquinaPersonalizadoraController.createPersonalizadora(personalizadora)
        assertAll(
            { assertEquals(res.balance, personalizadora.balance) }
        )
        coVerify(exactly = 1) { maquinaPersonalizadoraRepository.save(any()) }
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun getPersonalizadoraById() = runTest { this: TestScope
        coEvery { maquinaPersonalizadoraRepository.findById(any()) } returns personalizadora
        val res = maquinaPersonalizadoraController.getPersonalizadoraById(personalizadora.id)
        assertAll(
            { assertEquals(res!!.balance, personalizadora.balance) }
        )
        coVerify(exactly = 1) { maquinaPersonalizadoraRepository.findById(any()) }
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun deletePersonalizadora() = runTest { this: TestScope
        coEvery { maquinaPersonalizadoraRepository.delete(any()) } returns Unit

        val res = maquinaPersonalizadoraController.deletePersonalizadora(personalizadora)

        assertEquals(res, Unit)
        coVerify(exactly = 1) { maquinaPersonalizadoraRepository.delete(any()) }
    }

    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun resetPersonalizadoras() = runTest { this: TestScope
        coEvery { maquinaPersonalizadoraRepository.deleteAll() } returns Unit

        val res = maquinaPersonalizadoraController.resetPersonalizadoras()

        assertEquals(res, Unit)
        coVerify(exactly = 1) { maquinaPersonalizadoraRepository.deleteAll() }
    }
```


Tests in 'tennislab-MongoDB-SpringData-Reactivo.test'
Tests passed: 54 of 54 tests – 7 sec 3 ms
Test Results 7 sec 3 ms
MongoDbSpringDataReactivoApplicationTests 29 ms
contextLoads() 29 ms
AdquisicionControllerTest 330 ms
getAdquisicionById() 180 ms
deleteAdquisicion() 60 ms
resetAdquisiciones() 12 ms
getAdquisiciones() 45 ms
createAdquisicion() 23 ms
EncontrarControllerTest 65 ms
createEncontrar() 15 ms
getEncontrarById() 12 ms
resetEncontraciones() 10 ms
getEncontraciones() 8 ms
deleteEncontracion() 10 ms
MaquinaEncontradoraControllerTest 54 ms
createEncontradora() 27 ms
getEncontradoras() 8 ms
resetEncontradoras() 10 ms
getEncontradoraById() 10 ms
deleteEncontradora() 10 ms
MaquinaPersonalizadoraControllerTest 58 ms
getPersonalizadoraById() 12 ms
resetPersonalizadoras() 3 ms
> Task :jar
> Task :inspectClassesForKotlinIC
> Task :kspTestKotlin
> Task :processTestResources NO-SOURCE
> Task :compileTestKotlin
> Task :compileTestJava NO-SOURCE
> Task :testClasses UP-TO-DATE
> Task :test
00:12:47.605 [Test worker] DEBUG org.springframework.test.context.BootstrapUtils - Instantiating CacheAwareContextLoaderDelegate from class [org.springframework.test.context
.cache.DefaultCacheAwareContextLoaderDelegate]
00:12:47.617 [Test worker] DEBUG org.springframework.test.context.BootstrapUtils - Instantiating BootstrapContext using constructor [public org.springframework.test.context
.support.DefaultBootstrapContext(java.lang.Class,org.springframework.test.context.CacheAwareContextLoaderDelegate)]
00:12:47.653 [Test worker] DEBUG org.springframework.test.context.BootstrapUtils - Instantiating TestContextBootstrapper for test class [resa.mendoza
.MongoDbSpringDataReactivoApplicationTests] from class [org.springframework.boot.test.context.SpringBootTestContextBootstrapper]
00:12:47.667 [Test worker] INFO org.springframework.boot.test.context.SpringBootTestContextBootstrapper - Neither @ContextConfiguration nor @ContextHierarchy found for test
class [resa.mendoza.MongoDbSpringDataReactivoApplicationTests], using SpringBootContextLoader
00:12:47.672 [Test worker] DEBUG org.springframework.test.context.support.AbstractContextLoader - Did not detect default resource location for test class [resa.mendoza
.MongoDbSpringDataReactivoApplicationTests]: class path resource [resa/mendoza/MongoDbSpringDataReactivoApplicationTests-context.xml] does not exist
00:12:47.672 [Test worker] DEBUG org.springframework.test.context.support.AbstractContextLoader - Did not detect default resource location for test class [resa.mendoza
.MongoDbSpringDataReactivoApplicationTests]: class path resource [resa/mendoza/MongoDbSpringDataReactivoApplicationTestsContext.groovy] does not exist
00:12:47.672 [Test worker] INFO org.springframework.test.context.support.AbstractContextLoader - Could not detect default resource locations for test class [resa.mendoza
.MongoDbSpringDataReactivoApplicationTests]: no resource found for suffixes {-context.xml, Context.groovy}.