tennislab: mongodb reactivo

Mongo Atlas – KtorFit - Caché

IES Luis Vives

AD – Mario Resa y Sebastián Mendoza

Contenido

[Introducción 1](#_Toc126166030)

[Diseño 2](#_Toc126166031)

[Lenguaje y tecnologías 2](#_Toc126166032)

[Diagrama 3](#_Toc126166033)

[Usuario, pedido, turno y raqueta 3](#_Toc126166034)

[Producto y adquisición 5](#_Toc126166035)

[Tarea, pedido, raqueta, encordado, personalizado y adquisición 6](#_Toc126166036)

[Máquina, encordadora, personalizadora, turno 8](#_Toc126166037)

[Enums 9](#_Toc126166038)

[Estructura de proyecto 10](#_Toc126166039)

[Gradle 10](#_Toc126166040)

[Modelos 12](#_Toc126166041)

# 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]](#footnote-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.

# 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]](#footnote-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]](#footnote-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.

## 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.

Diagrama, Dibujo de ingeniería

Descripción generada automáticamente

### 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.

Diagrama

Descripción generada automáticamente

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.

1. Usuario – Turno.

Diagrama

Descripción generada automáticamente

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*.

1. Usuario – Raqueta.

Diagrama

Descripción generada automáticamente

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*.

Diagrama

Descripción generada automáticamente

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

Diagrama

Descripción generada automáticamente

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.

1. Tarea – Raqueta

Diagrama

Descripción generada automáticamente

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.

1. Tarea – Pedido

Diagrama

Descripción generada automáticamente

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:

Diagrama

Descripción generada automáticamente

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:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

## Estructura de proyecto

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

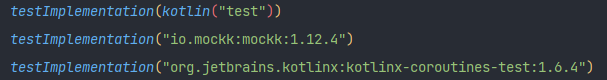
En este apartado iremos desglosando poco a poco como se han ido configuración las clases en sus diferentes carpetas. Hay que señalar que, aunque en el diagrama anterior no se comentaban las variables que tenían cada clase, es ahora en la siguiente sección donde repasaremos más detenidamente en la elección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



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

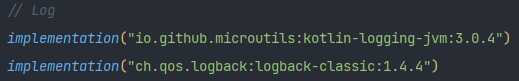
Texto

Descripción generada automáticamente

1. Coroutines. Para usar coroutines en Kotlin



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



1. Serializador. Serializador específico para Kotlin.



1. BCrypt. Encriptador de contraseñas.

Logotipo

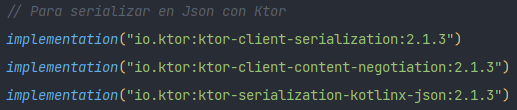
Descripción generada automáticamente

1. KtorFit. Retrofit pero para Kotlin.

Texto

Descripción generada automáticamente

1. Serializador en Ktor.



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



1. Documentación para Kotlin.



1. Koin, inyección de dependencias.

Texto

Descripción generada automáticamente

### 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

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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

#### Personalizar

Texto

Descripción generada automáticamente

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

#### Tarea

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

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:

Texto

Descripción generada automáticamente

#### Response

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

#### Producto

Texto

Descripción generada automáticamente

### 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:

Texto

Descripción generada automáticamente

Archivo *Properties*:

Texto

Descripción generada automáticamente

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

Texto

Descripción generada automáticamente

### 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.

Texto

Descripción generada automáticamente

#### ProductoService

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

Texto

Descripción generada automáticamente

### 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:

Texto

Descripción generada automáticamente

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:

Texto

Descripción generada automáticamente

### 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

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

#### UsuarioRepository

Logotipo

Descripción generada automáticamente

#### UsuarioMongoRepositoryImpl

Texto

Descripción generada automáticamente

#### 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.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

#### UsuariosKtorFitRepositoryImpl

Texto

Descripción generada automáticamente

#### TareasKtorFitRepository

Texto

Descripción generada automáticamente

### 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*.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

1. API REST - https://www.ibm.com/es-es/cloud/learn/rest-apis [↑](#footnote-ref-1)
2. KtorFit - https://foso.github.io/Ktorfit/ [↑](#footnote-ref-2)
3. Cache4K - https://reactivecircus.github.io/cache4k/ [↑](#footnote-ref-3)