tennislab: mongodb reactivo

Mongo Atlas – KtorFit - Caché

IES Luis Vives

AD – Mario Resa y Sebastián Mendoza

Contenido

[Introducción 1](#_Toc126091738)

[Diseño 2](#_Toc126091739)

[Lenguaje y tecnologías 2](#_Toc126091740)

[Diagrama 3](#_Toc126091741)

[Usuario, pedido, turno y raqueta 3](#_Toc126091742)

[Producto y adquisición 5](#_Toc126091743)

[Tarea, pedido, raqueta, encordado, personalizado y adquisición 6](#_Toc126091744)

[Máquina, encordadora, personalizadora, turno 8](#_Toc126091745)

[Enums 9](#_Toc126091746)

# 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. 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

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)