

TennisLab 2

Práctica Acceso a Datos 03



Autores: Mohamed Asidah & Rocío Palao

Introducción	3
Requisitos	3
Requisitos de información	3
Clientes	3
Empleados	4
Pedidos	4
Tareas	4
Productos	5
Raquetas	6
Máquinas personalizadas	6
Máquinas de encordado	7
Requisitos funcionales	7
Diagramas	8
Diagrama de Clases	8
Casos de uso	9
Casos de uso del administrados	9
Caso de uso del trabajador	10
Caso de uso del cliente	11
Arquitectura	12
Guía de uso	13
Configuración inicial	13
Tecnologías	14
MongoDB	14
Spring Data	14
KMongo	14
KOIN	14
Kotlin Coroutines	14
Kotlin Serialization	14
Guava	14
Cache4K	14
Mordant	15
KtorFit	15
MockK	15
GitHub	15
Control de versiones	15
Testing	16

Introducción

En esta práctica vamos a realizar la implementación de la tienda de productos y servicios relacionados con el tenis de la práctica anterior esta vez usando una base de datos NoSql con MongoDB.

Para la implementación usaremos principalmente Kotlin como lenguaje de programación y KMongo y Spring Data como frameworks con los que acceder a la base de datos.

La base de datos de Mongo se encuentra alojada en Mongo Atlas pero también es accesible desde un docker local incluido en el proyecto.

Requisitos

Requisitos de información

Todos nuestros elementos tienen en común los datos de Id que será generado automáticamente por MongoDB y un UUID que se generará aleatoriamente al crear el elemento. A parte de esos datos cada una de nuestras clases se diferenciará en los siguientes datos:

Clientes

Nuestros clientes van a ir definidos por un nombre, un nombre de usuario, un email y una contraseña. Estos serán los datos que el cliente proporcionará al sistema.

Aparte de estos datos, el cliente también llevará una lista de pedidos y una lista de raquetas que se irán actualizando según se incluyan estos. Estas listas están guardadas por referencia a estos objetos.

	data	Customer
Ⓟ	name	String
Ⓟ	password	String
Ⓟ	NId	Int
Ⓟ	available	Boolean
Ⓟ	email	String
Ⓟ	username	String
Ⓟ	orderList	List<String>?
Ⓟ	tennisRacketsList	List<Racket>?
Ⓟ	id	String
Ⓟ	uuid	UUID

Empleados

Los empleados además de los datos comunes también cuentan con un tiempo de entrada y salida que sirven para controlar los turnos y se crean automáticamente al iniciar sesión y cerrar sesión respectivamente, una lista de tareas con las tareas asignadas a ese empleado y guardadas de forma embebida para así poder acceder a ellas desde el empleado sin tener que recuperarlo. También tendrá la máquina que tiene asignada guardada como referencia y que se liberará al acabar su turno (cerrar sesión), y, el listado de pedidos que está preparando guardados como referencias.

	data	Employee
(P)	available	Boolean
(P)	entryTime	LocalDateTime?
(P)	surname	String
(P)	id	String
(P)	departureTime	LocalDateTime?
(P)	uuid	UUID
(P)	name	String
(P)	password	String
(P)	tasks	List<Task>?
(P)	stringings	List<Stringing>?
(P)	email	String
(P)	admin	Boolean
(P)	orderList	List<String>?
(P)	machine	String?

Pedidos

Los pedidos los guardaremos con su fecha de entrada, una fecha de salida que se mantendrá como nula hasta que el pedido se complete, fecha máxima de preparación, y la fecha final de preparación. También tendremos el precio total que será la suma de todas las tareas que forman ese pedido, el cliente que realizó el pedido guardado como embebido, la lista de tareas que forman el pedido y por último, el estado del pedido (recibido, terminado o en progreso).

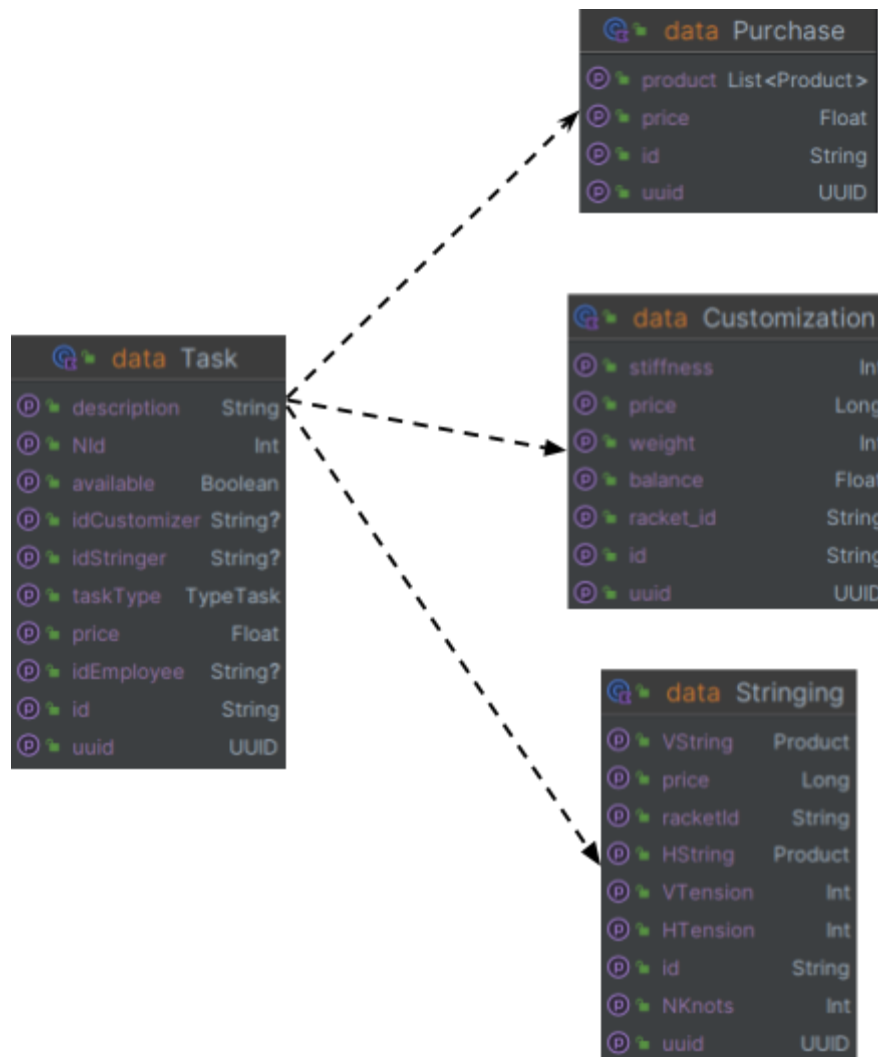
	data	Order
(P)	state	Status
(P)	entryDate	LocalDate?
(P)	exitDate	LocalDate?
(P)	tasks	List<Task>?
(P)	client	Customer
(P)	maxDate	LocalDate?
(P)	finalDate	LocalDate?
(P)	totalPrice	Float
(P)	id	String
(P)	uuid	UUID

la

Tareas

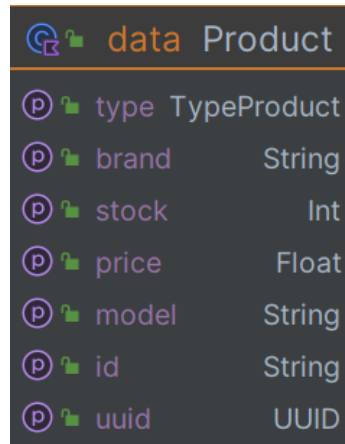
Las tareas vamos a tenerlas guardadas con el id del empleado que realizará la tarea, el id de la máquina que utiliza para completarlo, el tipo de tarea (encordado, personalizado y adquisición), el precio de la tarea y la descripción de la tarea que se realiza. Esta tarea puede ser:

- Encordado: recibirá los datos desde el usuario y serán las tensiones verticales y horizontales, el número de nudos (2 o 4), y el cordaje que se usará para el encordado vertical y el cordaje horizontal. También recibirá el id de la tarea que se va encordar.
- Personalizado: el cliente le dará los datos de equilibrio, peso y rigidez, así como el id de la raqueta que se va a personalizar.
- Adquisición: las tareas de adquisición contará con la lista de productos que se van a comprar y el precio total.



Productos

Los productos estarán identificados por un tipo (cordaje, raqueta o complemento), la marca, el modelo, el precio y el stock.

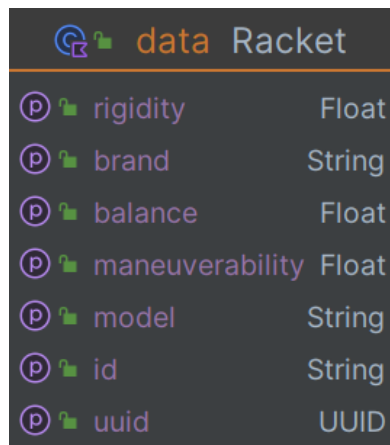


A screenshot of a GraphQL IDE showing the schema for a `Product` type. The schema is defined within a `data` namespace. The `Product` type has the following fields: `type` (enum `TypeProduct`), `brand` (String), `stock` (Int), `price` (Float), `model` (String), `id` (String), and `uuid` (UUID). Each field is preceded by a purple circle icon containing a white 'P'.

data Product	
type	TypeProduct
brand	String
stock	Int
price	Float
model	String
id	String
uuid	UUID

Raquetas

Nuestras raquetas estarán formadas por una marca, un modelo, el valor de equilibrio, el peso y la rigidez.



A screenshot of a GraphQL IDE showing the schema for a `Racket` type. The schema is defined within a `data` namespace. The `Racket` type has the following fields: `rigidity` (Float), `brand` (String), `balance` (Float), `maneuverability` (Float), `model` (String), `id` (String), and `uuid` (UUID). Each field is preceded by a purple circle icon containing a white 'P'.

data Racket	
rigidity	Float
brand	String
balance	Float
maneuverability	Float
model	String
id	String
uuid	UUID

Máquinas personalizadas

Las máquinas para el personalizado estarán formadas por una marca y un modelo, sus ids, la fecha de adquisición y si puede modificar el peso o no, si puede modificar el equilibrio o no, y, si puede modificar la rigidez o no.

	data	Customizer
(P)	brand	String
(P)	available	Boolean
(P)	balance	Boolean
(P)	acquisitionDate	LocalDate
(P)	rigidity	Boolean
(P)	maneuverability	Boolean
(P)	model	String
(P)	id	String
(P)	uuid	UUID

Máquinas de encordado

Las máquinas de encordado estarán compuestas por una marca, un modelo, si es automática o no, la fecha de adquisición, la tensión máxima de encordado y la tensión mínima de encordado.

	data	Stringer
(P)	brand	String
(P)	available	Boolean
(P)	automatic	TypeMachine
(P)	acquisitionDate	LocalDate
(P)	minimumTension	Int
(P)	maximumTension	Int
(P)	model	String
(P)	id	String
(P)	uuid	UUID

Requisitos funcionales

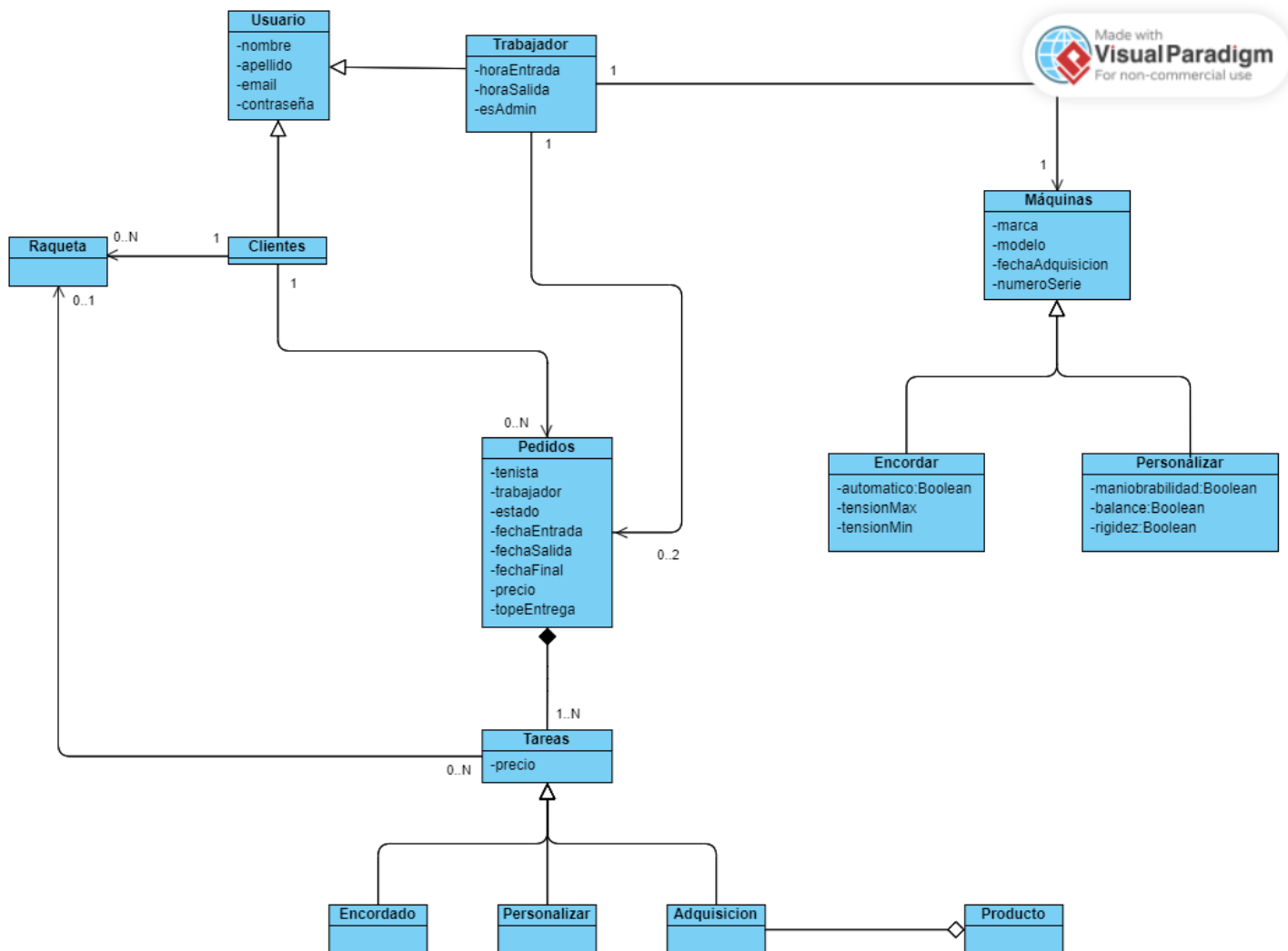
Nuestros requisitos funcionales serán los siguientes:

- Se exige que la aplicación funcione recuperando datos de usuarios desde una API (<https://jsonplaceholder.typicode.com/users>).
- Mantener un histórico de las tareas que se vayan creando en otro endpoint de la misma API (<https://jsonplaceholder.typicode.com/todos>).
- Los trabajadores no podrán tener más de dos pedidos asignados por turno.
- Un trabajador solo puede tener una máquina asignada por turno y una máquina solo puede estar asignada a un trabajador por turno.
- Tener una caché de usuarios que se actualiza cada 60 segundos.
- Guardar los datos en una base de datos noSql MongoDB tanto en remoto (Mongo Atlas) como en local (Docker).
- Aplicar reactividad y recibir notificaciones en tiempo real sobre los cambios en los pedidos.

Diagramas

Diagrama de Clases

Para la realización de este diagrama de clases, nos hemos basado en el de nuestra anterior práctica. Pero en este caso hemos decidido cambiar algunas relaciones

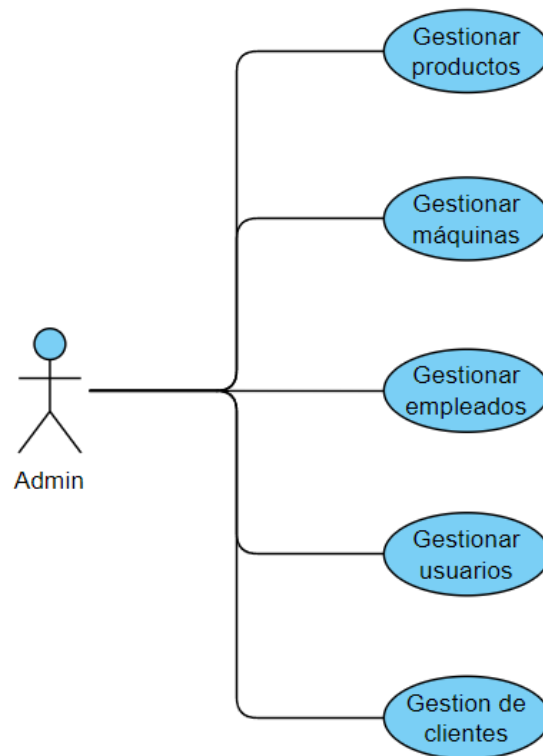


- Hemos eliminado al administrador ya que nos dimos cuenta de que no era necesario teniendo a los trabajadores y añadiendo el atributo de `esAdmin`.
- La relación entre los clientes y las raquetas se ha creado ya que pensamos que un cliente puede tener muchas raquetas por lo que era mejor que cada cliente tenga sus raquetas.

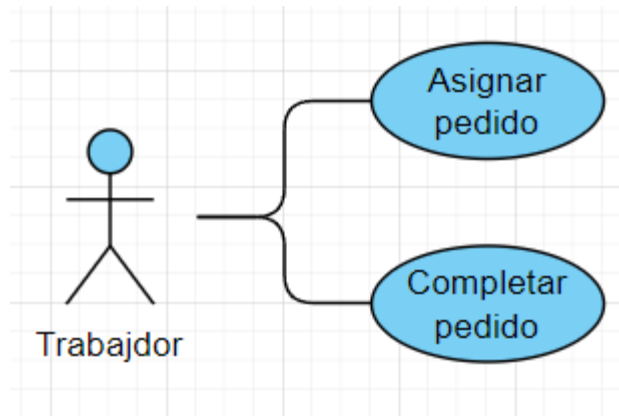
- La relación entre los trabajadores y las máquinas se ha creado ya que en un turno un trabajador debe de tener asignada una máquina.

Casos de uso

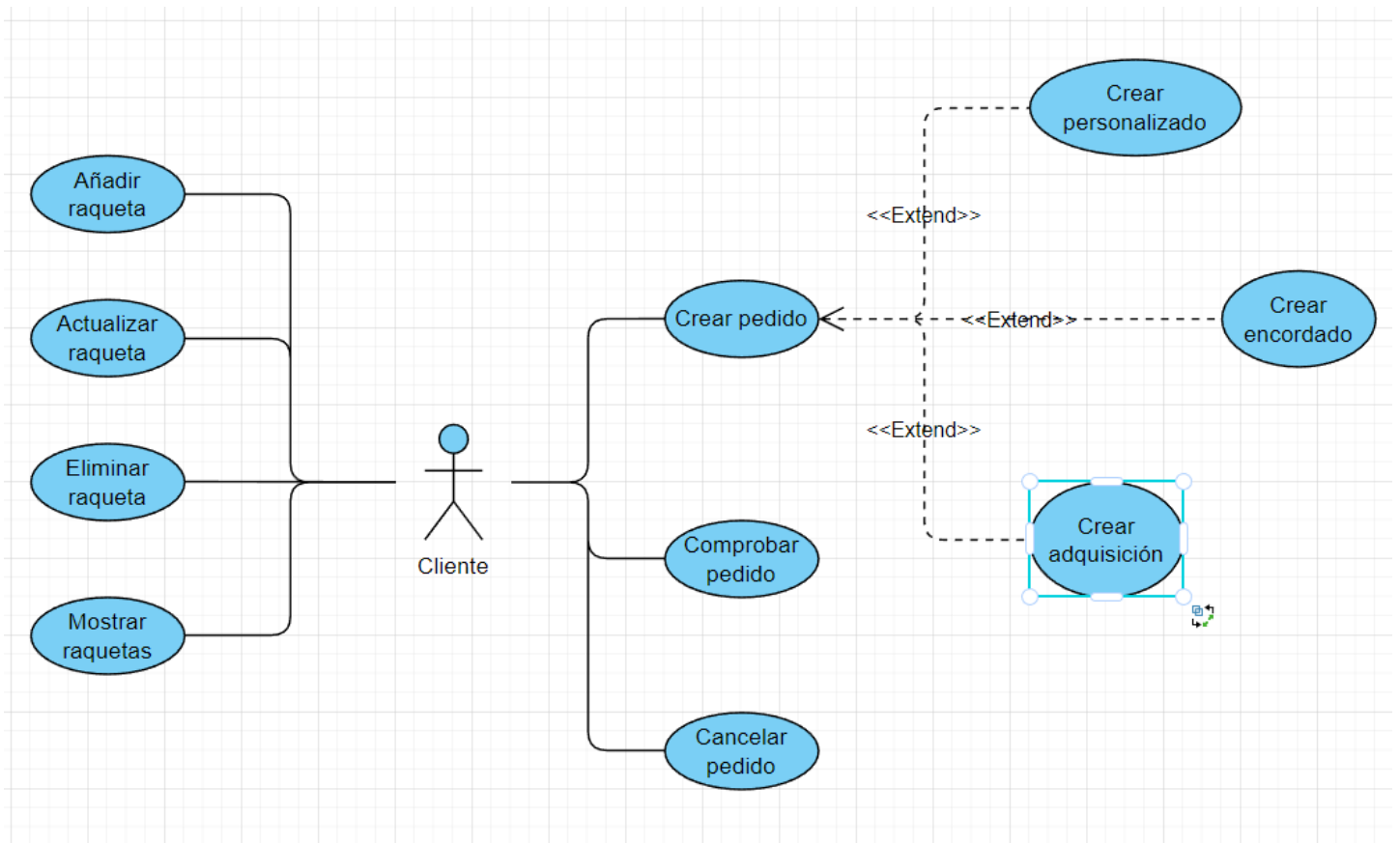
Casos de uso del administrados



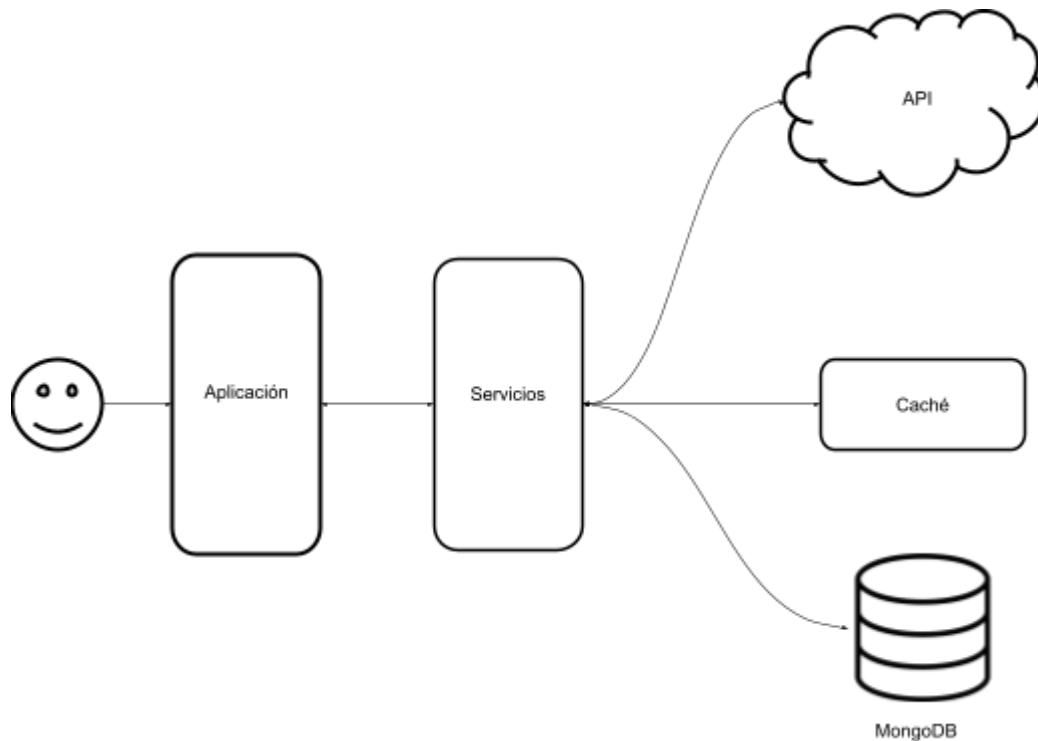
Caso de uso del trabajador



Caso de uso del cliente



Arquitectura



Vamos a utilizar una arquitectura Modelo Vista Controlador (MVC) en la que tendremos una aplicación (la vista), unos modelos de información, una capa de servicio que conectará el servicio de Api, el de la caché y el de MongoDB, y un controlador que mediará entre todos.

La forma en la que esto funcionará será la siguiente:

1. El usuario realizará una petición.
2. El controlador comprobará si se puede satisfacer esa petición desde la caché.
 - a. Si no se puede desde la caché, pasaremos a MongoDB.
 - i. Si no se puede desde MongoDB iremos a la API-
3. El controlador le devolverá al usuario los datos que ha solicitado o completará la tarea.

Guía de uso

Configuración inicial

Se pone a disposición del usuario un archivo `application.properties` que se puede usar para seleccionar cómo funcionará la aplicación una vez arranquemos:

- `mongo.conection`: contiene la cadena de conexión a Mongo Atlas para realizar las consultas. En los campos de `USER` y `PASSWORD` habrá que poner un usuario y contraseña válidos para realizar acceder.

`mongo.conection` = `mongodb+srv://USER:PASSWORD@tennislabcluster.jgjryxb.mongodb.net/`

- `mongo.conection.local`: indicará la cadena de conexión en caso de que queramos usar MongoDB de forma local con un despliegue en docker.

`mongo.conection.local` = `mongodb://mongoadmin:mongopass@localhost/tennislab?authSource=admin`

- `mongo.local`: tiene un valor de `true` o `false` y le indicará al programa si vamos a ejecutarlo de forma local (`true`) o de forma remota (`false`).

`mongo.local` = `true`

- `database.clear`: valor booleano que indicará si queremos borrar todos los datos de la base de datos (`true`) o mantenerlos (`false`).

`database.clear` = `false`

- `database.sample`: valor boolean que indicará si queremos iniciar el programa con un conjunto de datos de prueba (`true`).

`database.sample` = `false`

Los usuarios disponibles para usar Mongo Atlas son (usuario : contraseña):

- moha : 1234
- rocio : 1234
- joseluis : 1234

Tecnologías

Las tecnologías usadas para implementar este problema han sido:

MongoDB

MongoDB es una base de datos de documentos que ofrece una gran escalabilidad y flexibilidad, y un modelo de consultas e indexación avanzado.

Spring Data

Framework de Spring que hemos utilizado para implementar la base de datos asíncrona de MongoDB.

KMongo

Framework para Kotlin que hemos utilizado para implementar la base de datos asíncrona de MongoDB.

KOIN

Koin es un framework inteligente para Kotlin usado para gestionar la inyección de dependencias enfocado a aplicaciones. Lo hemos implementado con anotaciones.

Kotlin Coroutines

Soporte de kotlin para implementar corrutinas y así programar de forma concurrente, no bloqueante y thread safe.

Kotlin Serialization

Conjunto de librerías de Kotlin para la JVM que utilizaremos para serializar JSON.

Guava

Conjunto de librerías para Java de Google que utilizaremos para hashear las contraseñas de los usuarios en SHA-512.

Cache4K

Librería para Kotlin que ofrece una forma sencilla para manejar una caché en forma de mapa con clave y valor.

Mordant

Librería para dar formato a la terminal y hacer que la aplicación sea más visual y legible.

KtorFit

Ktorfit es un procesador de cliente HTTP/Kotlin para Kotlin multiplataforma usando KSP y el cliente KTOR. Inspirado por Retrofit.

MockK

Librería para realizar mocking de elementos durante el testeo y así simular su comportamiento.

GitHub

Para el control de versiones hemos utilizado un repositorio en GitHub.

Control de versiones

Para el control de versiones hemos utilizado GitHub (Git).

Hemos creado un repositorio en el que ambos hemos trabajado como colaboradores. Luego realizando un fork de ese repositorio y aplicando GitFlow cada uno ha utilizado una rama feature y develop donde se realizaban los cambios para luego realizar un pull request al repositorio principal.

Testing

El testeo de la aplicación se ha realizado con JUnit 5 y MockK.

Esta vez todos nuestros test eran con aparición de las corrutinas de kotlin, por lo que hemos tenido que utilizar la dependencia de kotlin para el testeo de corrutinas.

```
testImplementation("org.jetbrains.kotlinx:kotlinx-coroutines-test:1.6.4")
```