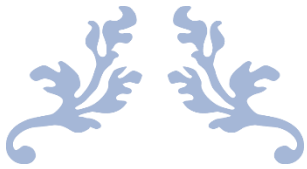


AD-P03-TennislabNoSQL

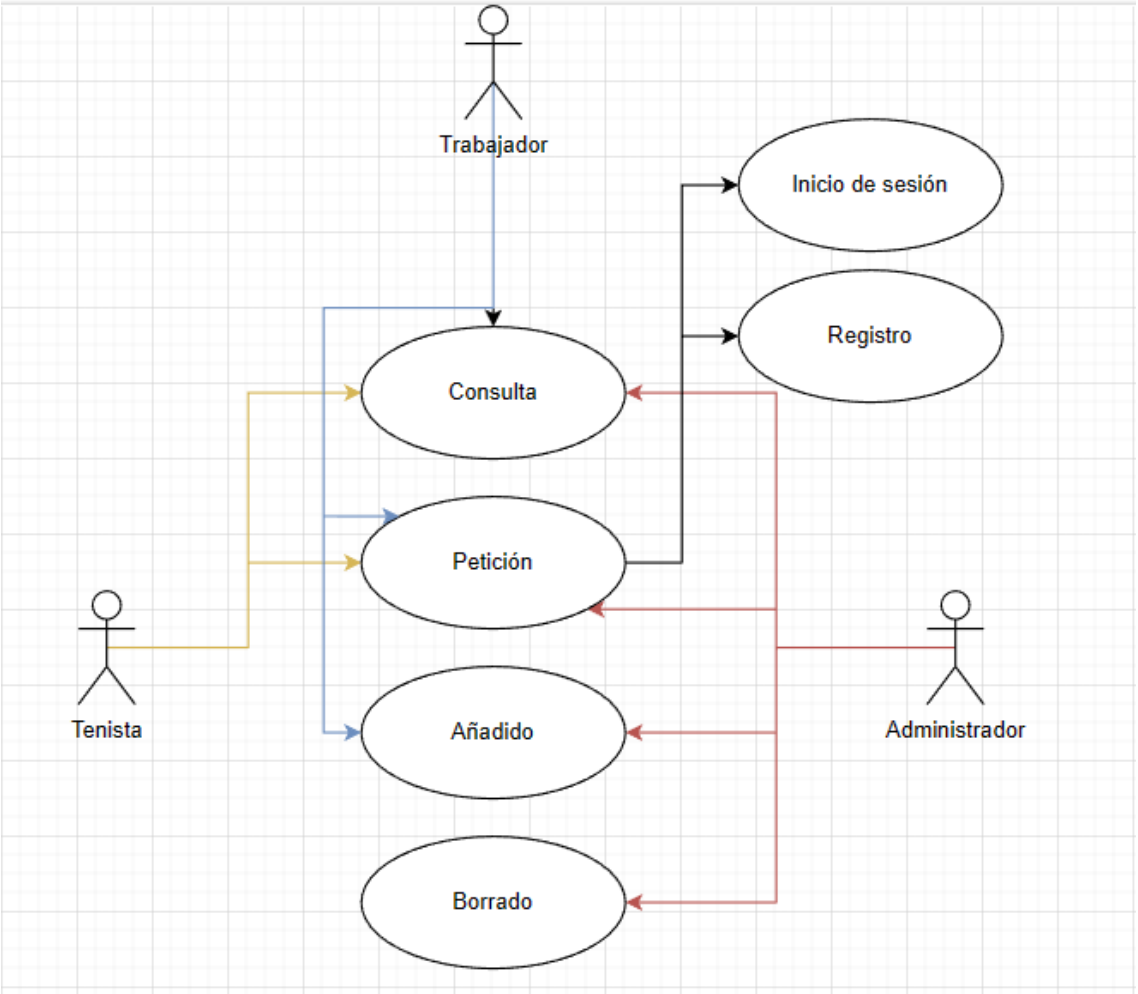
Documentación del proyecto

5 DE FEBRERO DE 2023

IVÁN AZAGRA Y DANIEL RODRÍGUEZ
IES Luis Vives

Contenido

Diagrama de uso:

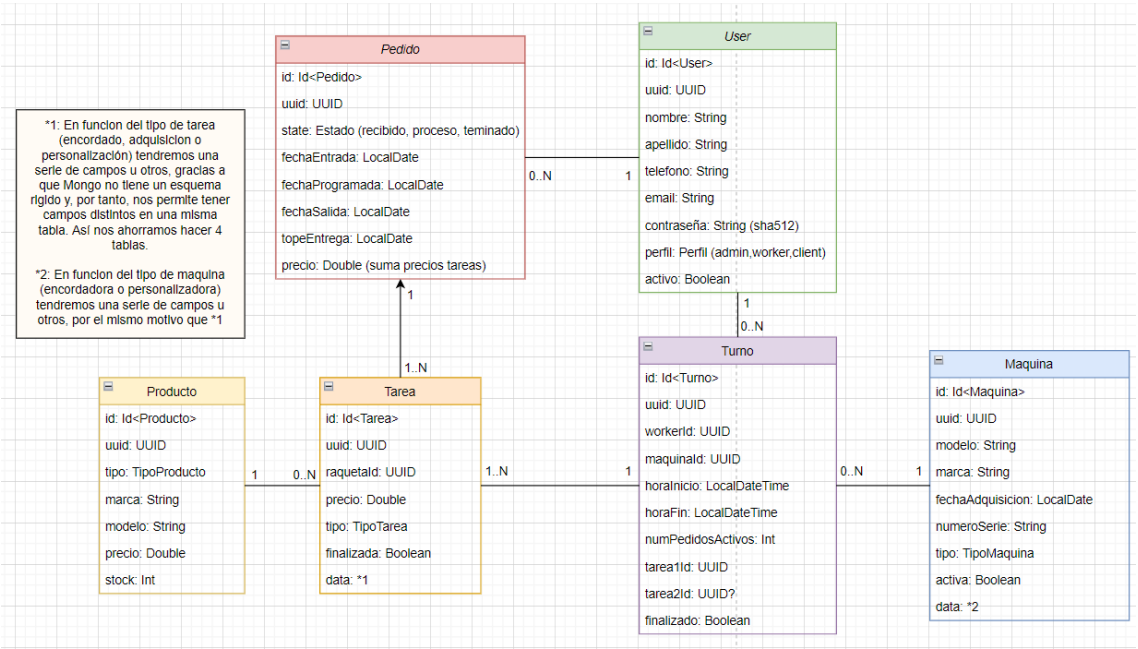


El tenista puede realizar consultas y peticiones para realizar el inicio de sesión o el registro en caso de no tener cuenta en nuestro sistema.

El trabajador puede realizar consultas, peticiones de inicio de sesión o registro y de añadido limitado únicamente a los turnos.

El administrador puede realizar consultas, peticiones de inicio de sesión o registro, añadido de cualquier entidad del sistema y borrado de las mismas, es el único con la capacidad de hacer esta última acción.

Diagrama de clases:



Para el proyecto hemos tenido la necesidad de crear los modelos de Producto, Tarea, Pedido, Usuario, Turno y Máquina, ya que los necesitamos para mostrar qué vende la aplicación, que tareas tienen los empleados que usuario usa la aplicación, ya sea cliente, empleado o administrador, en que turno se realizan las tareas y qué máquinas han sido utilizadas para según que tareas se estén realizando, por ello definimos unas relaciones que explicaré después de definir los modelos utilizados.

Pedido: Este objeto está compuesto con un identificador, los estados que puede tener el pedido y las fechas requeridas, estas son la fecha de entrada del pedido, la fecha programada de salida, la fecha real de salida, y la fecha tope de entrega, este planteamiento viene dado para tener un registro de cuándo se recibe el pedido, cuándo es la estimación en la que se piensa que se entregará para dar una fecha al cliente cuando se recibe el pedido, esta fecha después se puede actualizar a la fecha de salida real, que podría no ser la misma y por último tendremos la fecha de entrega, que podría variar en unos pocos días, ya que cuando sale el pedido de las instalaciones no tiene por qué ser el mismo día que el pedido llegue al cliente, o que el mismo lo recoja.

Tarea: Este objeto está compuesto por un identificador propio, un identificador de la raqueta, precio que va a tener, el tipo de tarea a realizar, un atributo para marcar si está finalizada o no, el id del pedido al que pertenece la tarea, el id del producto adquirido que puede ser nulo porque puede no ser una tarea de tipo adquisición sino una tarea de tipo personalización o de encordaje, en el caso de las personalizaciones tenermos los atributos de peso de tipo int nullable, balance de tipo doble nullable, rigidez de tipo int nullable por lo dicho anteriormente y por último los atributos de la tarea de encordaje que son la tensión horizontal, identificador del cordaje horizontal, tensión vertical, identificación del cordaje vertical y un atributo de tipo boolean para definir si tiene dos nudos o no, todos estos también nullables.

Producto: Este objeto está compuesto de un identificador, el tipo de producto que es, ya que podrían ser raquetas, cordajes, overgrips, grips, antivibradores y fundas, el modelo también

tiene registros de las marcas y modelos los cuáles son cadenas de texto, el precio del producto que se trata con un Double y el stock actual.

User: Este objeto será la representación de los usuarios del sistema, se compone de un identificador, el nombre, apellido, teléfono, email, la contraseña que será encriptada con sha512, el tipo de perfil que podrá ser admin, worker o client y por último un valor de si está activo o no para conservar el registro, pero sin poder acceder a la cuenta en caso de estar inactiva. Guardamos los datos, pero no se permite más el acceso si la cuenta está inactiva.

Turno: Este objeto representa la jornada con un identificador en la que se ha completado una o dos tareas, las cuales son referenciadas a través de su identificador, este modelo se compone por la referencia al identificador del trabajador, y al de la máquina, las horas de inicio y de fin que son dos LocalDateTime y el número de pedidos activos, solo es obligatoria una referencia a una tarea, ya que es necesario que mínimo haya una tarea para guardar el registro del turno, por ello la segunda es de tipo String? con lo que admitimos valores nullables en este.

Máquina: Se trata de la representación de la máquina que se haya utilizado para trabajar durante el turno, está compuesto por un identificador, modelo, marca y número de serie de tipo String, la fecha de adquisición de la máquina LocalDate, el tipo de la máquina que podrá ser encordadora o personalizadora, un atributo de tipo boolean para definir si la máquina está en activo o no y por último se encapsulan los datos en el valor data representado el cuál será los atributos de específicos del tipo de máquina del que se trata. Estos atributos encapsulados son en caso de la encordadora un atributo para saber si es manual de tipo boolean, tensión máxima y tensión mínima de tipo doble nullable, en el caso de las personalizadoras utilizarán los atributos que indican si mide la maniobravilidad, si mide la rigidez y si mide el balance todos estos de tipo boolean nullable. Estos últimos atributos son nullables porque dependiendo del tipo de máquina del que se trate utilizarán unos atributos u otros.

Relaciones:

Pedido-User: Se trata de una relación 0..N-1 ya que el usuario puede darse de alta en la plataforma pero no realizar ningún pedido, o por otro lado podría realizar múltiples, sin embargo para que haya un pedido debe de existir un usuario que lo realice.

Pedido-Tarea: Se trata de una relación 1-1..N ya que cada pedido está compuesto por mínimo una tarea y puede tener tantas tareas como requisitos establezca el cliente.

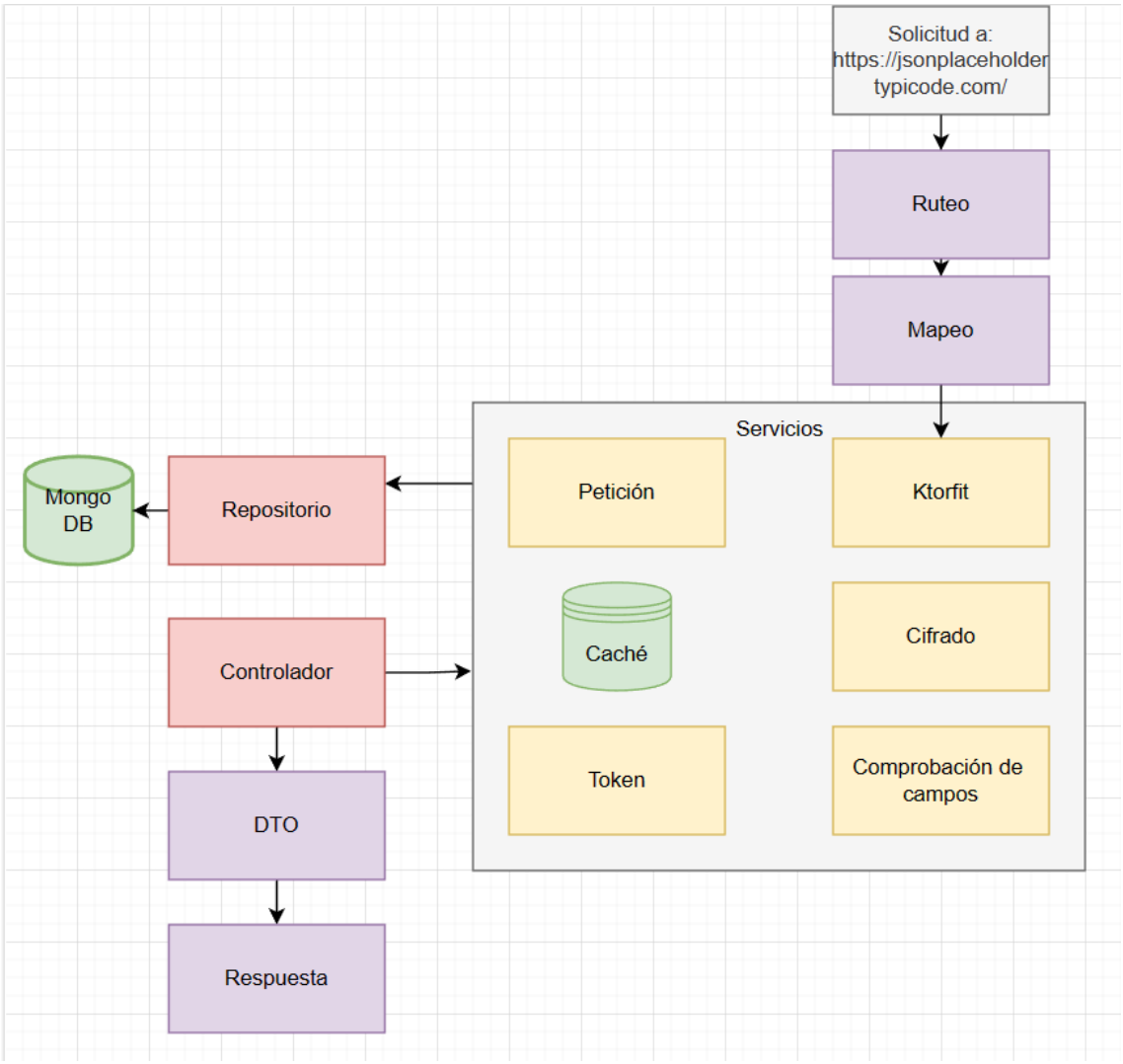
User-Turno: Se trata de una relación 1-0..N ya que tiene que haber un usuario de tipo worker para poder realizar el turno de trabajo, sin un trabajador disponible no puede haber un turno y por ello no hay una máquina en uso ni una o varias tareas que se vayan a cumplir durante ese turno.

Tarea-Producto: Se trata de una relación 0..N-1 ya que el producto existirá aunque no haya una tarea existente en ese momento, sin embargo también puede haber varias tareas que requieran de los productos.

Tarea-Turno: Se trata de una relación 1..N-1 ya que puede haber entre 1 y varias tareas durante el transcurso de un turno y por la existencia de las tareas habrá un turno en las que se irán cumpliendo.

Turno-Maquina: Se trata de una relación 0..N-1 ya que la máquina siempre va a existir aunque no se esté realizando ningún turno en ese momento, mientras que puede haber varios turnos que hagan uso de máquinas.

Arquitectura:



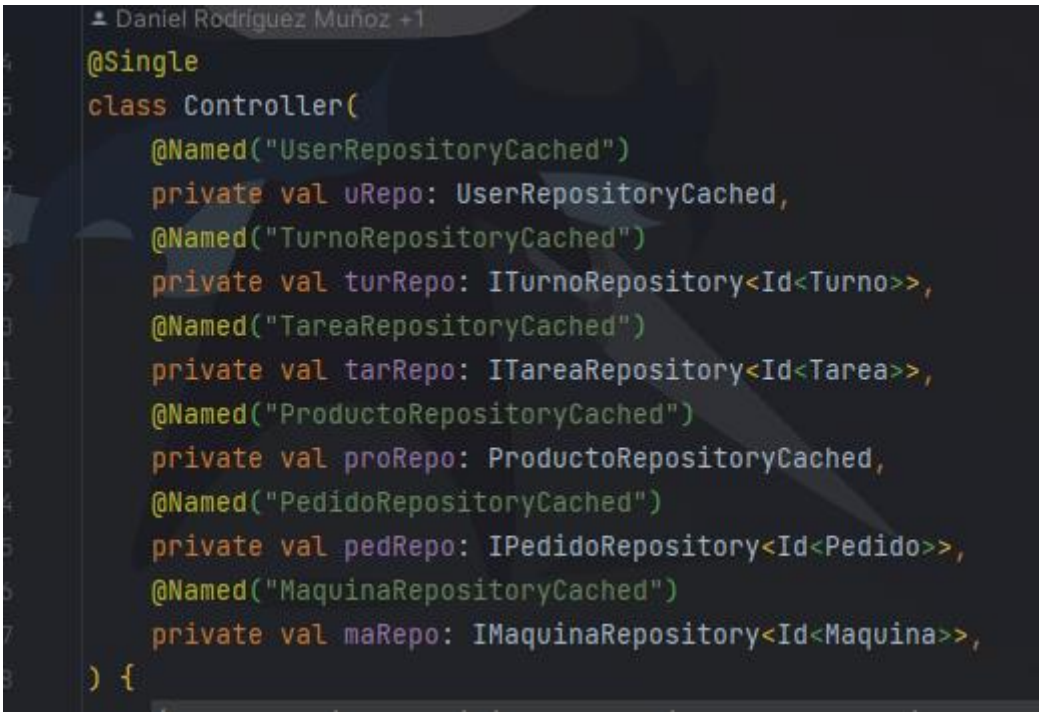
Comienza con el endpoint de consulta que es ruteado con ktorfit y mapeado a las diferentes entidades que utilizamos en el programa, el repositorio recoge los datos del endpoint consultado y los guarda en la base de datos de mongo y se utiliza para realizar diferentes consultas a esta base de datos, en nuestros servicios tenemos una caché que guardará los datos consultados para agilizar el proceso dentro del lado del cliente, esta misma caché se actualiza cada minuto para tener los datos más recientes.

El controlador utiliza estos servicios para realizar restricciones y comprobaciones, para ello utilizamos los tokens para limitar las funciones dependiendo del tipo de perfil que las quiera realizar, utiliza también un sistema de cifrado para el guardado seguro de las contraseñas, el controlador hace uso de los objetos de transferencia de datos (DTO) para generar las respuestas con los datos.

En cuanto a la organización de carpetas, aunque parecida hay ciertos cambios entre ambas versiones, comenzaré señalando aquellas pertenecientes al proyecto con mongo:

Para comenzar hemos tenido que crear una carpeta que englobe todo el proyecto, esto ha sido para aplicar la inyección de dependencias a través de “Koin” a todas las clases que se encuentran dentro de ella, aunque hay otras maneras de hacerlo hemos decidido hacerlo de esta manera ya que era una forma sencilla para aplicarlo sin tener que crear una clase que estar modificando con la adición de posibles nuevas clases en un futuro, de esta manera solamente tenemos que aplicar el decorador @Named y el nombre de la clase que queramos inyectar.

En cuanto a la organización las carpetas primordiales son la carpeta de los modelos, donde encontramos los modelos utilizados en la aplicación junto a sus enumeradores y una clase response que utilizamos para devolver los códigos de peticiones junto a el mensaje propuesto, la carpeta de los repositorios, donde podemos encontrar una carpeta por repositorio de cada modelo donde están las interfaces y repositorios normales y cacheados que se utilizan y por último la carpeta controllers donde tenemos un controlador singular para la aplicación, desde este se hace uso de todos los métodos de los repositorios, aquí se puede ver cómo se aplica la inyección de dependencias con Koin mencionada anteriormente a través del constructor.



```kotlin
/** @param id Identificador de tipo UUID del objeto User ...*/
Daniel Rodríguez Muñoz +1
suspend fun findUserByUuid(id: UUID) : String = withContext(Dispatchers.IO) {
    val user = uRepo.findByUUID(id)

    val res = if (user == null) ResponseError( code: 404,  message: "NOT FOUND: User with id $id not found.")
    else ResponseSuccess( code: 200, user.toDTO())

    json.encodeToString(res)
}

/** @param id Identificador de tipo int del objeto User ...*/
Daniel Rodríguez Muñoz
suspend fun findUserById(id: Int) : String = withContext(Dispatchers.IO) {
    val user = uRepo.findById(id)

    val res = if (user == null) ResponseError( code: 404,  message: "NOT FOUND: User with id $id not found.")
    else ResponseSuccess( code: 200, user.toDTO())

    json.encodeToString(res)
}

/** Este método devuelve todos los usuarios que se encuentren registrados en la base de datos ...*/
Daniel Rodríguez Muñoz
suspend fun findAllUsers() : String = withContext(Dispatchers.IO) {
    val users = uRepo.findAll().toList()

    val res = if (users.isEmpty()) ResponseError( code: 404,  message: "NOT FOUND: No users found.")
    else ResponseSuccess( code: 200, UserDTOvisualizeList(toDTO(users)))

    json.encodeToString(res)
}

/** Este método devuelve todos los usuarios que se encuentren activos o inactivos dependiendo del parámetro pasado
Daniel Rodríguez Muñoz
suspend fun findAllUsersWithActivity(active: Boolean) : String = withContext(Dispatchers.IO) {
    val users = uRepo.findAll().toList().filter { it.activo == active }

    val res = if (users.isEmpty()) ResponseError( code: 404,  message: "NOT FOUND: No users found.")
    else ResponseSuccess( code: 200, UserDTOvisualizeList(toDTO(users)))

    json.encodeToString(res)
```

```kotlin
import org.litote.kmongo.Id
import org.litote.kmongo.newId
import koin.serializers.LocalDateSerializer
import koin.serializers.UUIDSerializer
import java.time.LocalDate
import java.util.*

/**
 * @Author Daniel Rodriguez Muñoz
 * Clase POKO de las maquinas.
 * La primera parte de los parametros son obligatorios
 * los parametros opcionales son especificos para los
 */
Daniel Rodríguez Muñoz
@Serializable
data class Maquina(
    @BsonId @Contextual
    val id: Id<Maquina> = newId(),
    @Serializable(with = UUIDSerializer::class)
    val uuid: UUID = UUID.randomUUID(),
    val modelo: String,
    val marca: String,
    @Serializable(with = LocalDateSerializer::class)
    val fechaAdquisicion: LocalDate,
    val numeroSerie: String,
    val tipo: TipoMaquina,
    val activa: Boolean,

    // esto es data para encordadoras
    val isManual: Boolean?,
    val maxTension: Double?,
    val minTension: Double?,

    // esto es data para personalizadoras
    val measuresManeuverability: Boolean?,
    val measuresRigidity: Boolean?,
    val measuresBalance: Boolean?
)
```

```kotlin
package koin.models.maquina

/**
 * Tipos de máquina
 */
Daniel Rodríguez Muñoz
enum class TipoMaquina {
    ENCORDADORA,PERSONALIZADORA
}
```

```
override suspend fun findByUUID(id: UUID): Maquina? = withContext(Dispatchers.IO) { this: CoroutineScope
    var result: Maquina? = null

    cache.cache.asMap().forEach { if (it.key == id) result = it.value }
    if (result != null) {
        listSearches.add(result!!)
        return@withContext result
    }

    result = repo.findByUUID(id)
    if (result != null) listSearches.add(result!!)

    result ^withContext
}

/** Este método guarda la entidad pasada a listSearches y también en el repositorio ...*/
Daniel Rodríguez Muñoz
override suspend fun save(entity: Maquina): Maquina = withContext(Dispatchers.IO) { this: CoroutineScope
    listSearches.add(entity)
    repo.save(entity)
    entity ^withContext
}

/** Este método cambia el estado del maquina con el identificador pasado de activo a ...*/
Daniel Rodríguez Muñoz
override suspend fun setInactive(id: Id<Maquina>): Maquina? = withContext(Dispatchers.IO) { this: CoroutineScope
    val result :Maquina? = repo.setInactive(id)
    if (result != null) listSearches.add(result)
    result ^withContext
}

/** Este método borra el maquina del repositorio y de la caché en caso de que se encuentre ...*/
Daniel Rodríguez Muñoz
override suspend fun delete(id: Id<Maquina>): Maquina? = withContext(Dispatchers.IO) { this: CoroutineScope
    val entity :Maquina? = repo.delete(id)
    if (entity != null){
        listSearches.removeIf { it.uuid == entity.uuid }
        cache.cache.invalidate(entity.uuid)
    }
    entity ^withContext
}
```

Para el funcionamiento de la aplicación necesitamos otras carpetas aparte de las mencionadas anteriormente, que, aunque son el grueso de la aplicación hacen uso de distintas clases creadas para el tratado de datos, conexiones o utilidades necesarias.

Para la conexión de la base de datos utilizada en esta práctica hemos creado una carpeta db donde se encuentra el manager de la base de datos que lee el archivo de propiedades para obtener la cadena de conexión y crea el cliente para conectarse a la bbdd, también contiene una clase con datos “prefabricados” para la inyección de datos a la api.

```
object DBManager {
    private var mongoClient: CoroutineClient
    var database: CoroutineDatabase

    /**
     * Parámetros de la cadena de conexión
     */
    // configuraciones para mongoAtlas aqui
    val properties = readProperties()
    private val MONGO_TYPE = properties.getProperty("MONGO_TYPE")
    private val HOST = properties.getProperty("HOST")
    private val PORT = properties.getProperty("PORT")
    private val DATABASE = properties.getProperty("DATABASE")
    private val USERNAME = properties.getProperty("USERNAME")
    private val PASSWORD = properties.getProperty("PASSWORD")
    private val OPTIONS = properties.getProperty("OPTIONS")

    private val MONGO_URI = "$MONGO_TYPE$USERNAME:$PASSWORD@$HOST/$DATABASE"
    Daniel Rodríguez Muñoz
    init {
        logger.debug( msg: "Initializing connection to MongoDB")
        println("Initializing connection to MongoDB : $MONGO_URI$OPTIONS")
        mongoClient = KMongo.createClient( connectionString: "$MONGO_URI$OPTIONS").coroutine
        database = mongoClient.getDatabase(DATABASE)
    }
}

/**
 * Este método sirve para leer del archivo de propiedades necesario para establecer la conexión con mongo
 * @return Objeto Properties con los datos para la conexión
 */
Iván Azagra +1
fun readProperties(): Properties {
    val properties = Properties()
    try {
        properties.load(
            FileInputStream( name: ".${File.separator}" +
                    "TennisLab-Mongo${File.separator}src${File.separator}main${File.separator}" +
                    "resources${File.separator}config.properties")
        )

    }
```

Tenemos una carpeta “DTO” (Data Transfer Object) en la que hemos creado una carpeta por cada modelo y en la que hemos creado un DTO para cada caso que hemos observado. Estos casos son la creación de modelos con los DTOCreate, el caso de la visualización con los DTOVisualize y por último la visualización de una lista con el DTOListVisualize que utilizamos en el controlador para mostrar los datos devueltos desde el repositorio o para ejecutar las funciones dedicadas a crear datos.

```kotlin
@Serializable
data class ProductoDTOcreate(
    @Serializable(with = UUIDSerializer::class)
    val uuid: UUID = UUID.randomUUID(),
    val tipo: TipoProducto,
    val marca: String,
    val modelo: String,
    var precio: Double,
    val stock: Int = 0
)

Daniel Rodríguez Muñoz
@Serializable
@SerialName("ProductoDTOvisualize")
data class ProductoDTOvisualize(
    val tipo: TipoProducto,
    val marca: String,
    val modelo: String,
    var precio: Double,
    val stock: Int
)

Daniel Rodríguez Muñoz
@Serializable
@SerialName("ProductoDTOvisualizeList")
data class ProductoDTOvisualizeList(val productos: List<ProductoDTOvisualize>)
```

Tenemos una carpeta de mappers para trabajar con los DTOs anteriores, en esta carpeta tenemos de nuevo un mapper por tipo de modelo y se dedican a pasar los modelos a DTOVisualize o para pasa de DTOCreate a el modelo original para el guardado de los datos, aunque también hay casos como el del modelo tarea que tiene DTOs para trabajar con el API, estos mappers son toDTOapi que devuelve el DTO TareaDTOfromApi o lo mismo, pero casando los valores del DTOFromApi a el modelo original.

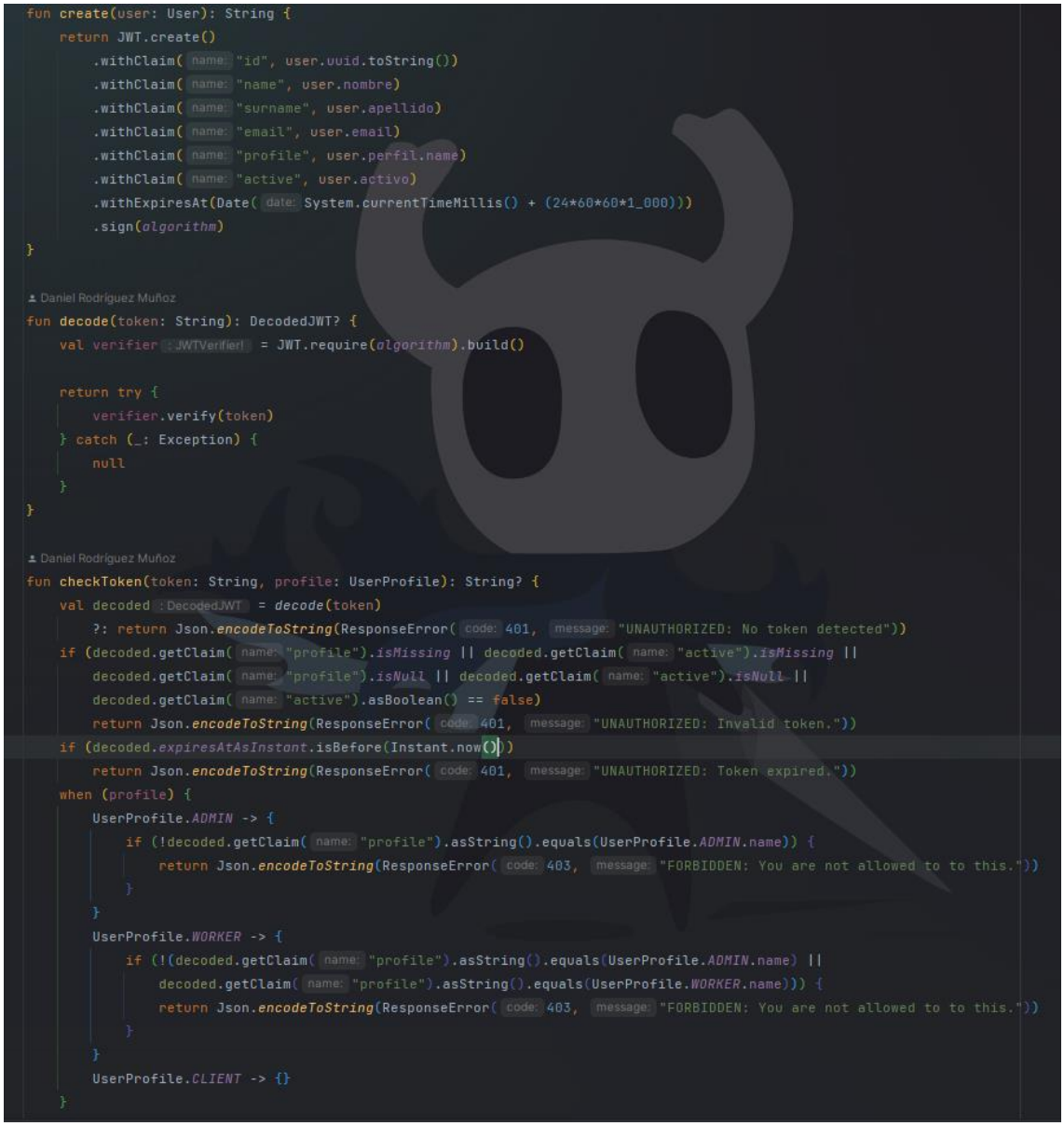
```kotlin
fun Tarea.toDTOapi() = TareaDTOFromApi(
    id = id.idValue.toString(),
    uuid = uuid.toString(),
    raquetaId = raquetaId.toString(),
    precio = precio,
    tipo = tipo,
    finalizada = finalizada,
    pedidoId = pedidoId.toString(),
    productoAdquiridoId = productoAdquiridoId.toString(),
    peso = peso,
    balance = balance,
    rigidez = rigidez,
    tensionHorizontal = tensionHorizontal,
    cordajeHorizontalId = cordajeHorizontalId.toString(),
    tensionVertical = tensionVertical,
    cordajeVerticalId = cordajeVerticalId.toString(),
    dosNudos = dosNudos
)

Daniel Rodríguez Muñoz
fun TareaDTOFromApi.fromDTO() = Tarea (
    id = id?.toId() ?: newId(),
    uuid = uuid?.let {UUID.fromString(it)}
        .run { UUID.fromString( name: "00000000-0000-0000-0000-000000000000") },
    raquetaId = raquetaId?.let { UUID.fromString(it) }
        .run {UUID.fromString( name: "00000000-0000-0000-0000-000000000001") },
    precio = precio ?: 0.0,
    tipo = tipo ?: TipoTarea.ADQUISICION,
    finalizada = finalizada ?: true,
    pedidoId = pedidoId?.let { UUID.fromString(it) }
        .run { UUID.fromString( name: "00000000-0000-0000-0000-000000000002") },
    productoAdquiridoId = productoAdquiridoId?.let { UUID.fromString(it) }
        .run { UUID.fromString( name: "00000000-0000-0000-0000-000000000002") },
    peso = peso,
    balance = balance,
    rigidez = rigidez,
    tensionHorizontal = tensionHorizontal,
    cordajeHorizontalId = cordajeHorizontalId?.let { UUID.fromString(it) }.run { null },
    tensionVertical = tensionVertical,
    cordajeVerticalId = cordajeVerticalId?.let { UUID.fromString(it) }.run { null },
    dosNudos = dosNudos
)
```

En la carpeta serializers tenemos tres serializadores utilizados para trabajar junto al serializador de kotlin, estas son el serializador de localDate, el serializador de LocalDateTime y por último el serializador de UUIDs ya que por defecto el serializador no trabaja con estos tipos de datos.

Por último, tenemos la carpeta services donde encontramos todos los servicios utilizados para la aplicación, aquí tenemos la carpeta cache donde se definen las caches que utilizarán todos los repositorios cacheados, por ello dentro de cache hay una división por carpeta de cada repositorio existente en la que hemos introducido el cacheado contando con una interfaz.

En la carpeta ktorfit creamos la implementación de Ktorfit con su interfaz y la definición del cliente que se utilizará para la conexión a el endpoint de consulta.

En el servicio de login tenemos la función de logado, registro y aparte un archivo para el servicio de tokens utilizados junto al logado, aquí se comprueban las contraseñas , se crean los usuarios y se crean los tokens del usuario, se comprueban y se codifican y decodifican.



```
fun create(user: User): String {
    return JWT.create()
        .withClaim( name: "id", user.uuid.toString())
        .withClaim( name: "name", user.nombre)
        .withClaim( name: "surname", user.apellido)
        .withClaim( name: "email", user.email)
        .withClaim( name: "profile", user.perfil.name)
        .withClaim( name: "active", user.activo)
        .withExpiresAt(Date( date: System.currentTimeMillis() + (24*60*60*1_000)))
        .sign(algorithm)
}

Daniel Rodríguez Muñoz
fun decode(token: String): DecodedJWT? {
    val verifier : JWTVerifier = JWT.require(algorithm).build()

    return try {
        verifier.verify(token)
    } catch (_: Exception) {
        null
    }
}

Daniel Rodríguez Muñoz
fun checkToken(token: String, profile: UserProfile): String? {
    val decoded : DecodedJWT = decode(token)
        ?: return Json.encodeToString(ResponseError( code: 401, message: "UNAUTHORIZED: No token detected"))
    if (decoded.getClaim( name: "profile").isMissing || decoded.getClaim( name: "active").isMissing ||
        decoded.getClaim( name: "profile").isNull || decoded.getClaim( name: "active").isNull ||
        decoded.getClaim( name: "active").asBoolean() == false)
        return Json.encodeToString(ResponseError( code: 401, message: "UNAUTHORIZED: Invalid token."))
    if (decoded.expiresAtAsInstant.isBefore(Instant.now()))
        return Json.encodeToString(ResponseError( code: 401, message: "UNAUTHORIZED: Token expired."))
    when (profile) {
        UserProfile.ADMIN -> {
            if (!decoded.getClaim( name: "profile").asString().equals(UserProfile.ADMIN.name)) {
                return Json.encodeToString(ResponseError( code: 403, message: "FORBIDDEN: You are not allowed to to this."))
            }
        }
        UserProfile.WORKER -> {
            if (!(decoded.getClaim( name: "profile").asString().equals(UserProfile.ADMIN.name) ||
                decoded.getClaim( name: "profile").asString().equals(UserProfile.WORKER.name))) {
                return Json.encodeToString(ResponseError( code: 403, message: "FORBIDDEN: You are not allowed to to this."))
            }
        }
        UserProfile.CLIENT -> {}
    }
```

```kotlin
suspend fun login(user: UserDTOLogin, repo: UserRepositoryCached): String? {
    val u :User = repo.findByEmail(user.email) ?: return null
    return if (!matches(user.password, u.password.encodeToByteArray())) null
    else create(u)
}

Daniel Rodríguez Muñoz
suspend fun register(user: UserDTORegister, repo: UserRepositoryCached): String? {
    val u :UserDTOvisualize? = createUserWithoutToken(user.fromDTO(), repo)
    return if (u != null) {
        val res :User? = repo.findByEmail(u.email)
        if (res == null) null
        else create(res)
    }
    else null
}

Daniel Rodríguez Muñoz
private suspend fun createUserWithoutToken(user: UserDTOcreate, repo: UserRepositoryCached): UserDTOvisualize? {
    if (fieldsAreIncorrect(user) || checkUserEmailAndPhone(user, repo))
        return null

    val res :User = repo.save(user.fromDTO())
    return res.toDTO()
}
```

```kotlin
object KtorFitClient {
    private const val URI = "https://jsonplaceholder.typicode.com/"

    private val ktorFit by lazy {
        Ktorfit.Builder().httpClient { this: HttpClientConfig<*>
            install(ContentNegotiation) { this: ContentNegotiation.Config
                json(
                    Json { this: JsonBuilder
                        isLenient = true
                        ignoreUnknownKeys = true
                        prettyPrint = true
                    }
                )
            }
            defaultRequest { this: DefaultRequest.DefaultRequestBuilder
                header(HttpHeaders.ContentType, ContentType.Application.Json)
            }
        }
            .baseUrl(URI)
            .build()
    }

    val instance by lazy {
        ktorFit.create<IKtorFit>()
    }
}
```

```kotlin
interface IKtorFit {
    Daniel Rodríguez Muñoz
    @GET("users")
    suspend fun getAll(): List<UserDTOfromAPI>

    Daniel Rodríguez Muñoz
    @GET("users/{id}")
    suspend fun getById(@Path("id") id: Int): UserDTOfromAPI?

    Daniel Rodríguez Muñoz
    @GET("todos")
    suspend fun getAllTareas(): List<TareaDTOFromApi>

    Daniel Rodríguez Muñoz
    @POST("todos")
    suspend fun saveTareas(@Body tarea: TareaDTOFromApi): TareaDTOFromApi
}
```

```kotlin
class _IKtorFitImpl(
  private val client: KtorfitClient,
) : IKtorFit {
  Daniel Rodríguez Muñoz *
  override suspend fun getAll(): List<UserDTOfromAPI> {
    val requestData = RequestData(method="GET",
        relativeUrl="users",
        returnTypeData=TypeData( qualifiedName: "kotlin.collections.List",listOf(TypeData( qualifiedName: "koin.dto.user.Use

    return client.suspendRequest<List<UserDTOfromAPI>, UserDTOfromAPI>(requestData)!!
  }

  Daniel Rodríguez Muñoz *
  override suspend fun getById(id: Int): UserDTOfromAPI? {
    val requestData = RequestData(method="GET",
        relativeUrl="users/{id}",
        returnTypeData=TypeData( qualifiedName: "UserDTOfromAPI?"),
        paths = listOf(PathData( key: "id", value: "$id", encoded: false)))

    return client.suspendRequest<UserDTOfromAPI?, UserDTOfromAPI?>(requestData)
  }

  Daniel Rodríguez Muñoz *
  override suspend fun getAllTareas(): List<TareaDTOFromApi> {
    val requestData = RequestData(method="GET",
        relativeUrl="todos",
        returnTypeData=TypeData( qualifiedName: "kotlin.collections.List",listOf(TypeData( qualifiedName: "koin.dto.tarea.Ta

    return client.suspendRequest<List<TareaDTOFromApi>, TareaDTOFromApi>(requestData)!!
  }

  Daniel Rodríguez Muñoz *
  override suspend fun saveTareas(tarea: TareaDTOFromApi): TareaDTOFromApi {
    val requestData = RequestData(method="POST",
        relativeUrl="todos",
        bodyData = tarea,
        returnTypeData=TypeData( qualifiedName: "koin.dto.tarea.TareaDTOFromApi"))
```

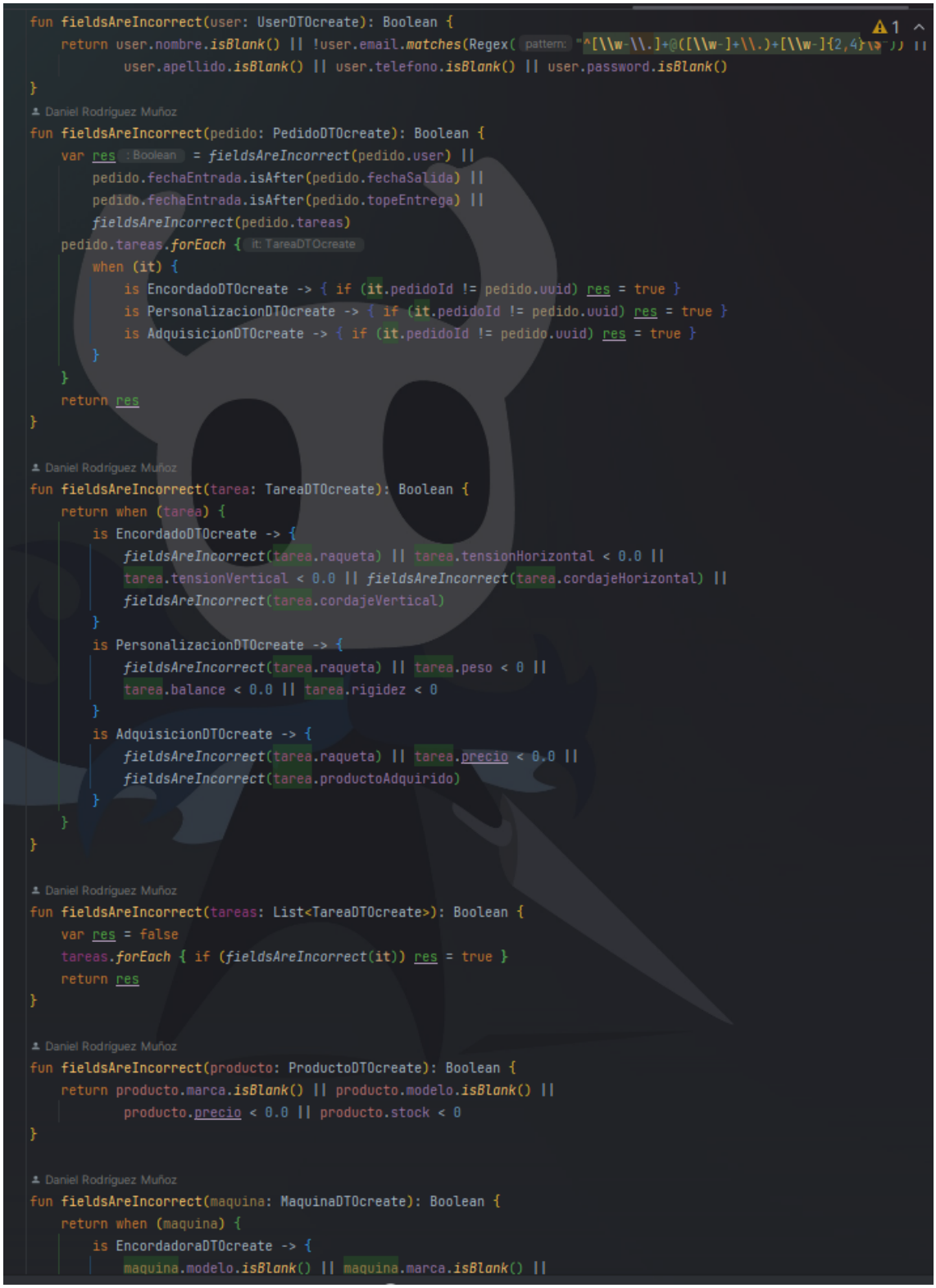
```kotlin
@Module
@ComponentScan("koin")
class KoinModule
```

```kotlin
@Single
class MaquinaCache : IMaquinaCache {
    override val hasRefreshAllCacheJob: Boolean = true
    override val refreshTime: Long = 60 * 1000L
    override val cache = Cache.Builder()
        .maximumCacheSize( size: 50)
        .expireAfterAccess(1.minutes)
        .build<UUID, Maquina>()

    Daniel Rodríguez Muñoz
    init { logger.debug { "Initializing MaquinaCache..." } }
}
```

Y por último en la carpeta utils tenemos las funcionalidades de cifrado, un comprobador de campos utilizado en el servicio de logado, un servicio de menus para interactuar con la aplicación y una función utils para pasar el uuid a cadena de texto.

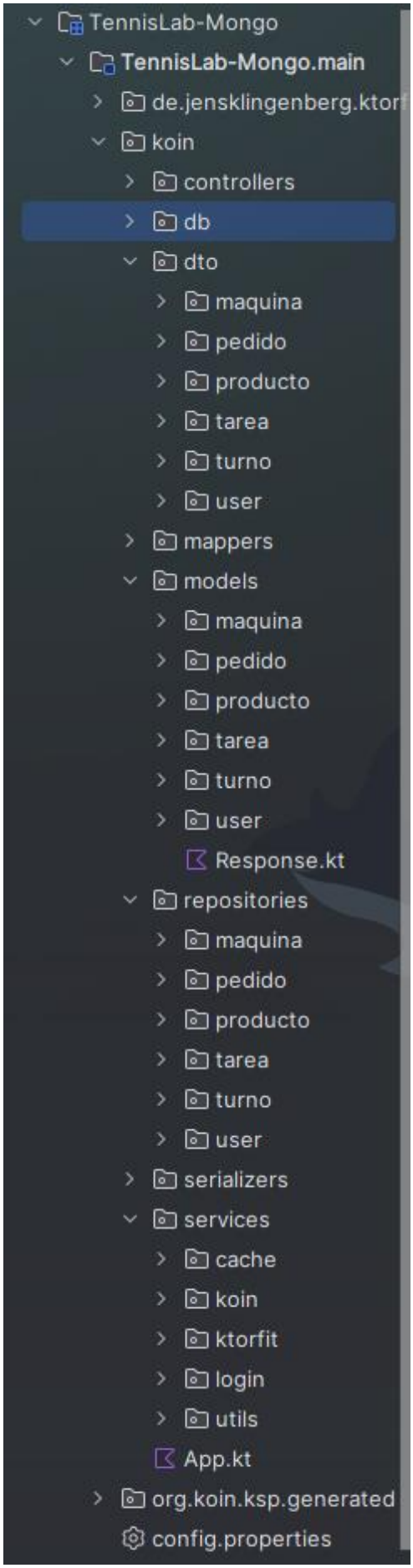
```kotlin
fun cipher(message: String) : String {
    return Bcrypt.hash(message,  saltRounds: 12).decodeToString()
}

Daniel Rodríguez Muñoz
fun matches(message: String, cipheredText: ByteArray) : Boolean {
    return Bcrypt.verify(message, cipheredText)
}
```

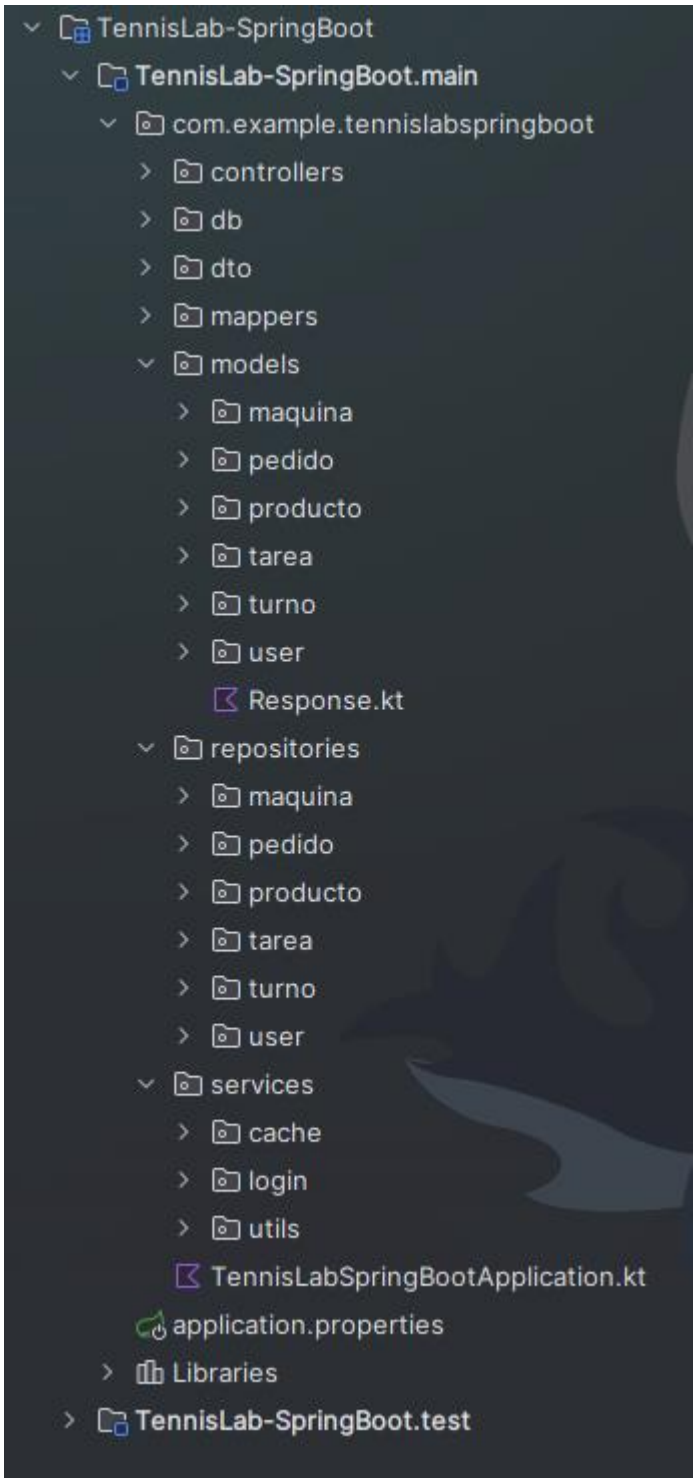
```
class _IKtorFitImpl(
  private val client: KtorfitClient,
) : IKtorFit {
  Daniel Rodríguez Muñoz *
  override suspend fun getAll(): List<UserDTOfromAPI> {
    val requestData = RequestData(method="GET",
        relativeUrl="users",
        returnTypeData=TypeData( qualifiedName: "kotlin.collections.List",listOf(TypeData( qualifiedName: "koin.dto.user.Use

    return client.suspendRequest<List<UserDTOfromAPI>, UserDTOfromAPI>(requestData)!!
  }

  Daniel Rodríguez Muñoz *
  override suspend fun getById(id: Int): UserDTOfromAPI? {
    val requestData = RequestData(method="GET",
        relativeUrl="users/{id}",
        returnTypeData=TypeData( qualifiedName: "UserDTOfromAPI?"),
        paths = listOf(PathData( key: "id", value: "$id", encoded: false)))

    return client.suspendRequest<UserDTOfromAPI?, UserDTOfromAPI?>(requestData)
  }

  Daniel Rodríguez Muñoz *
  override suspend fun getAllTareas(): List<TareaDTOFromApi> {
    val requestData = RequestData(method="GET",
        relativeUrl="todos",
        returnTypeData=TypeData( qualifiedName: "kotlin.collections.List",listOf(TypeData( qualifiedName: "koin.dto.tarea.Ta

    return client.suspendRequest<List<TareaDTOFromApi>, TareaDTOFromApi>(requestData)!!
  }

  Daniel Rodríguez Muñoz *
  override suspend fun saveTareas(tarea: TareaDTOFromApi): TareaDTOFromApi {
    val requestData = RequestData(method="POST",
        relativeUrl="todos",
        bodyData = tarea,
        returnTypeData=TypeData( qualifiedName: "koin.dto.tarea.TareaDTOFromApi"))
```



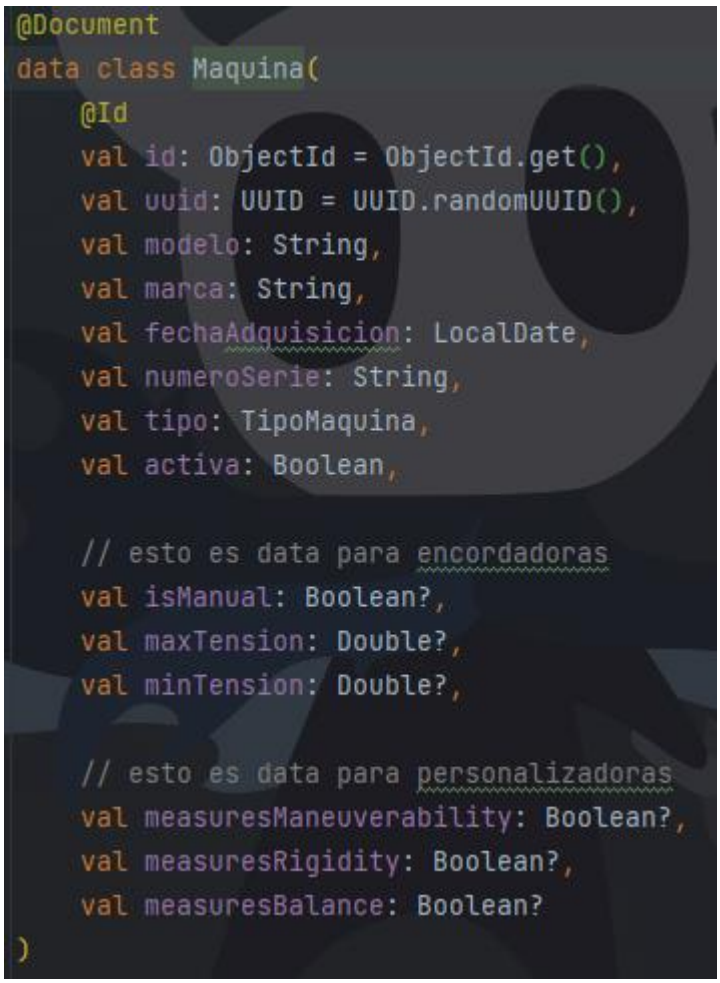
```kotlin
fun fieldsAreIncorrect(user: UserDTOcreate): Boolean {
    return user.nombre.isBlank() || !user.email.matches(Regex( pattern: "^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$")) ||
            user.apellido.isBlank() || user.telefono.isBlank() || user.password.isBlank()
}

Daniel Rodríguez Muñoz
fun fieldsAreIncorrect(pedido: PedidoDTOcreate): Boolean {
    var res : Boolean = fieldsAreIncorrect(pedido.user) ||
        pedido.fechaEntrada.isAfter(pedido.fechaSalida) ||
        pedido.fechaEntrada.isAfter(pedido.topeEntrega) ||
        fieldsAreIncorrect(pedido.tareas)
    pedido.tareas.forEach { it: TareaDTOcreate
        when (it) {
            is EncordadoDTOcreate -> { if (it.pedidoId != pedido.uuid) res = true }
            is PersonalizacionDTOcreate -> { if (it.pedidoId != pedido.uuid) res = true }
            is AdquisicionDTOcreate -> { if (it.pedidoId != pedido.uuid) res = true }
        }
    }
    return res
}

Daniel Rodríguez Muñoz
fun fieldsAreIncorrect(tarea: TareaDTOcreate): Boolean {
    return when (tarea) {
        is EncordadoDTOcreate -> {
            fieldsAreIncorrect(tarea.raqueta) || tarea.tensionHorizontal < 0.0 ||
            tarea.tensionVertical < 0.0 || fieldsAreIncorrect(tarea.cordajeHorizontal) ||
            fieldsAreIncorrect(tarea.cordajeVertical)
        }
        is PersonalizacionDTOcreate -> {
            fieldsAreIncorrect(tarea.raqueta) || tarea.peso < 0 ||
            tarea.balance < 0.0 || tarea.rigidez < 0
        }
        is AdquisicionDTOcreate -> {
            fieldsAreIncorrect(tarea.raqueta) || tarea.precio < 0.0 ||
            fieldsAreIncorrect(tarea.productoAdquirido)
        }
    }
}

Daniel Rodríguez Muñoz
fun fieldsAreIncorrect(tareas: List<TareaDTOcreate>): Boolean {
    var res = false
    tareas.forEach { if (fieldsAreIncorrect(it)) res = true }
    return res
}

Daniel Rodríguez Muñoz
fun fieldsAreIncorrect(producto: ProductoDTOcreate): Boolean {
    return producto.marca.isBlank() || producto.modelo.isBlank() ||
            producto.precio < 0.0 || producto.stock < 0
}

Daniel Rodríguez Muñoz
fun fieldsAreIncorrect(maquina: MaquinaDTOcreate): Boolean {
    return when (maquina) {
        is EncordadoraDTOcreate -> {
            maquina.modelo.isBlank() || maquina.marca.isBlank() ||
```


Daniel Rodríguez Muñoz
suspend fun menu(json: Json, token: String, controller: Controller): Boolean {
var userInput = 0
while (userInput < 1 || userInput > 7) {
println("""
Select something to do:
1. Users
2. Productos
3. Maquinas
4. Tareas
5. Pedidos
6. Turnos
7. Exit
""".trimIndent())
userInput = readln().toIntOrNull() ?: 0
}
return when (userInput) {
1 -> menuUsers(json, token, controller)
2 -> menuProductos(json, token, controller)
3 -> menuMaquinas(json, token, controller)
4 -> menuTareas(json, token, controller)
5 -> menuPedidos(json, token, controller)
6 -> menuTurnos(json, token, controller)
7 -> true
else -> true
}
}
Daniel Rodríguez Muñoz
suspend fun menuTurnos(json: Json, token: String, controller: Controller): Boolean {
var userInput = 0
while (userInput < 1 || userInput > 6) {
println("""
Select something to do:
1. Find All
2. Find by Id
3. Save
4. Delete (safe)


TennisLab-Mongo
TennisLab-Mongo.main
de.jensklingenberg.ktor
koin
controllers
db
dto
maquina
pedido
producto
tarea
turno
user
mappers
models
maquina
pedido
producto
tarea
turno
user
Response.kt
repositories
maquina
pedido
producto
tarea
turno
user
serializers
services
cache
koin
ktorfit
login
utils
App.kt
org.koin.ksp.generated
config.properties


TennisLab-SpringBoot
TennisLab-SpringBoot.main
com.example.tennislabspringboot
controllers
db
dto
mappers
models
maquina
pedido
producto
tarea
turno
user
Response.kt
repositories
maquina
pedido
producto
tarea
turno
user
services
cache
login
utils
TennisLabSpringBootApplication.kt
application.properties
Libraries
TennisLab-SpringBoot.test

A simple vista se puede observar que en spring no hemos necesitado una carpeta para koin que englobe toda la aplicación para la inyección de dependencias, ya que hemos utilizado la propia de spring para la realización de la aplicación con esta tecnología, se puede ver el uso de los decoradores necesarios para trabajar con spring como son @Controller, @Repository, @Autowired, @Document, @Id, @Service y @SpringBootApplication en el caso del main.



Usando spring solo hemos necesitado crear los repositorios cacheados, ya que los repositorios normales nos los da hecho al crear una interfaz que implemente en nuestro caso la interfaz CoroutineCrudRepository para la asincronía y añadirle el @Repository.

En el caso de los servicios hay que utilizar @Service en ellos en caso de que se trate de una clase como es el caso de los servicios de cacheado utilizados dentro de los repositorios de este tipo, pero por el resto el código no tiene ninguna modificación resaltable.

Requisitos funcionales:

Para los requisitos funcionales empezaré definiendo su significado, un requisito funcional es aquel que define el comportamiento interno del software, ya sean detalles técnicos, cómo manipula la información...

Entre los requisitos funcionales de este programa se encuentran los siguientes:

- El programa posee un sistema de autenticación que funciona con tokens.
- El programa cuenta con un catálogo de productos que puede ser consultado por cualquier tipo de usuario.
- Los empleados pueden consultar el catálogo y solo pueden hacer añadidos en el apartado de turnos.
- Los administradores pueden consultar el catálogo, hacer añadidos en cualquier ámbito (turnos, pedidos, usuarios…) y por último es el único capaz de hacer borrados definitivos en el sistema.
- El sistema cuenta con una función de inactividad o finalización con el que filtrar usuarios y pedidos o tareas para que no aparezcan en el catálogo general y mantener así un registro histórico, aquellas entidades con el atributo de finalización o inactividad en afirmativo no serán visibles más que por el administrador.

Requisitos no funcionales:

La definición de requisitos no funcionales es que son aquellos que no se refieren directamente a las funciones específicas suministradas por el sistema, sino a propiedades del sistema como rendimiento, seguridad o disponibilidad, hablan de cómo hace las cosas el sistema y no de qué hace.

Entre los requisitos no funcionales de este programa se encuentran los siguientes:

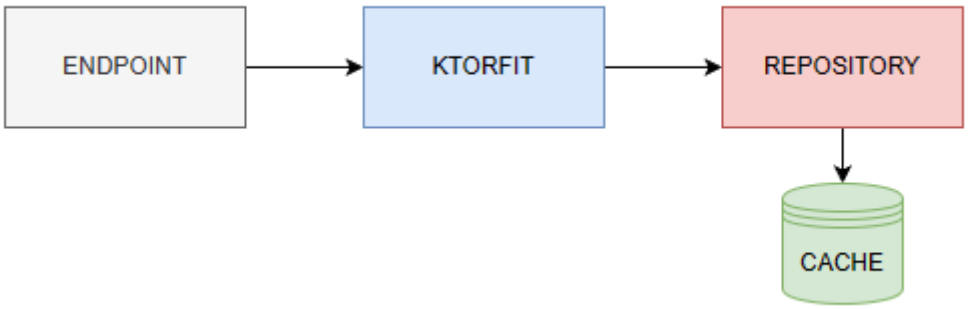
- Uso de tokens para identificar usuarios disponibles en nuestra base de datos y cuyo token se utiliza para validar el nivel de acceso a las funcionalidades.
- La aplicación cuenta con una caché para agilizar el proceso de consultas, esta caché se actualiza cada 60 segundos.
- Si se realiza una consulta y este dato no se encuentra en la caché se realizará una petición al repositorio para que lo llame desde la base de datos.
- Hace falta tener conexión a internet para realizar las consultas a la API.
- Este programa está escrito en Kotlin usando Kmongo para trabajar con la base de datos no relacional.
- Caché realizada con cache4k.
- Programa asíncrono con el uso de corrutinas y funciones suspendidas.
- Programa totalmente reactivo.
- Uso de Ktorfit para consultar la API y realizar los routeos.
- Se puede consultar sin conexión siempre que esté la información cacheada.
- La caché no puede actualizarse sin conexión.

Requisitos de información:

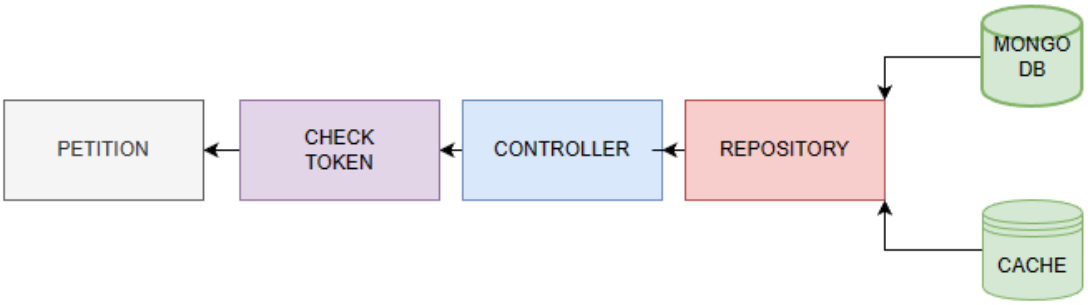
La especificación de estos requisitos es una descripción completa del comportamiento del sistema, este incluye los casos de uso que describen todas las interacciones de los usuarios, estos engloban los requisitos funcionales y no funcionales.

El programa consulta desde el endpoint **https://jsonplaceholder.typicode.com/** y guarda la información obtenida en nuestra base de datos no relacional a través del repositorio, el controlador nos permite comprobar entonces esa información haciendo consultas a nuestra base de datos en vez de a la API, las consultas ya realizadas se guardarán en la caché y esta se actualizará cada 60 segundos para tener la información más actualizada pero sin tener

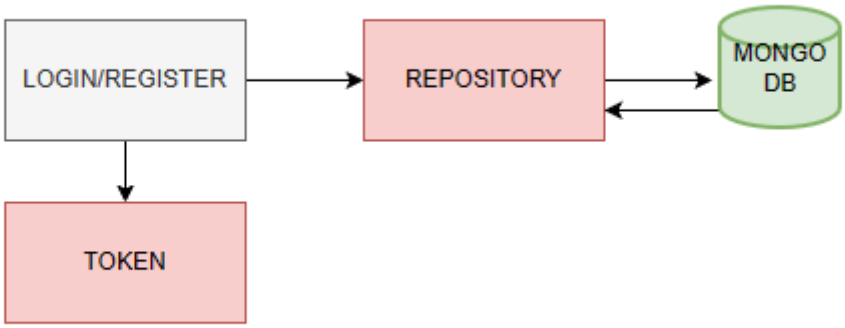
conectado el dispositivo todo el rato directamente a la base de datos realizando consultas de actualización en local.



Tenemos 3 tipos de usuarios, los usuarios normales que solo hacen consultas a el catálogo y no pueden hacer inserciones ni borrados en nuestro sistema pero si pueden realizar peticiones, los usuarios de tipo trabajador los cuales pueden consultar el catálogo y hacer inserciones solo en el apartado de turnos, también pueden realizar peticiones, por último tenemos a los usuarios de tipo administrador, estos pueden consultar el catálogo, hacer inserciones y borrados de todo tipo y realizar peticiones, estos últimos son los únicos con la capacidad de hacer borrados reales, puesto que cada entidad con las que trabajamos tiene un atributo de finalización o actividad que define la disponibilidad en nuestro sistema, si este atributo se encuentra en estado negativo no aparecerá en los registros accesibles para el resto de usuarios y sólo el administrador podrá verlos y realizar borrados reales sobre las entidades que considere.

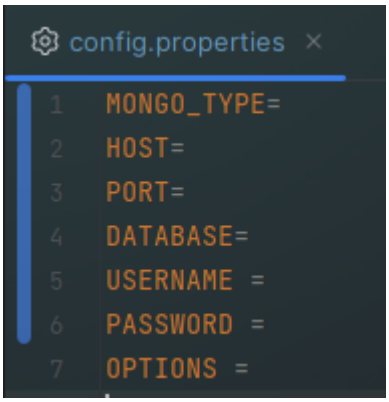


Para el funcionamiento de lo anterior tenemos un sistema de validación basado en tokens con el que dependiendo del nivel de acceso del token podrá realizar las acciones o las tendrá bloqueadas, para esto hemos usado JWT para la generación de esos tokens con los datos de nuestros usuarios, el token se genera cogiendo los datos de todos los parámetros del usuario y después codificándolos, de esta manera dentro del token figura el tipo de usuario del que se trata y podemos distinguir las funcionalidades que tiene disponibles, usamos una clave cifrada con HMAC256 para firmar estos tokens y confirmar su validez a la hora de decodificar los tokens en su posterior uso de confirmación del tipo de usuario.



Tenemos un sistema de cifrado en el que utilizamos Bcrypt, esto se utiliza para cifrar las contraseñas y guardarlas de esta manera para garantizar la seguridad de estas, usamos Bcrypt también para hacer la verificación a la hora de hacer los inicios de sesión para confirmar la autenticidad de las credenciales.

Uso de properties:



Mongo_Type: Sirve para distinguir el tipo de dirección de cadena que se utilizará para conectar con la base de datos de mongo, ya que se puede elegir entre la conexión con DNS SRV record que es lo que indica el “+srv” en la cadena de conexión, también puede ser una conexión con formato estándar de cadena de texto.

En la parte del host irá la dirección a la cual deberá de conectarse la aplicación y a continuación se especificaría el puerto de funcionamiento y la base de datos a la cual se va a conectar. Por último, se introduce el usuario y la contraseña que utilizará para conectarse, este tiene que ya estar dado de alta en la base de datos para que funcione.

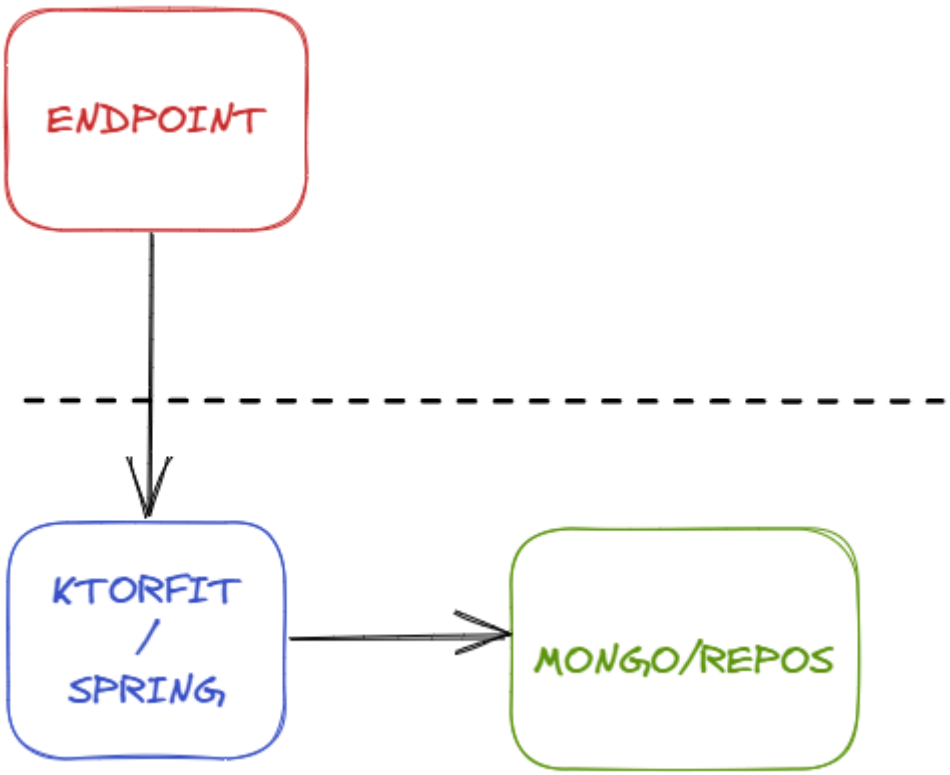
El apartado opciones en nuestro caso está vacío, pero aquí irían las variables que definen el modelo de autenticación, si usa TSL/SSL, si usa encriptado de mongo y si se conecta a un proxy entre otros.

```
MONGO_TYPE=mongodb+srv://
HOST=drmdam2023.7qtbnuz.mongodb.net
PORT=1707
DATABASE=practica03mongoIvanLoli
USERNAME = loli
PASSWORD = 1707
OPTIONS =
```

Explicación de tecnologías utilizadas:

Ktorfit/Spring:

Para la realización de esta aplicación hemos hecho dos versiones, una de ellas se conecta a mongo mediante el uso de ktorfit y trabaja directamente con ello, mientras que la segunda versión funciona a través de spring, utilizándolo para conectarse a el endpoint provisto para realizar la consulta y adquisición de los datos a almacenar.

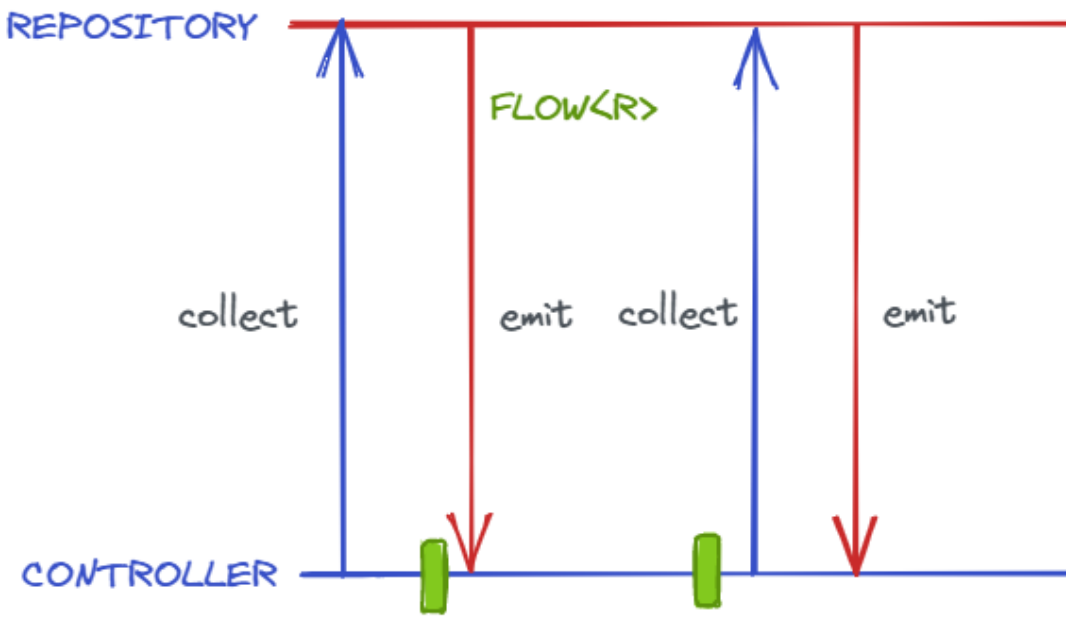


Corrutinas:

En ambas versiones hemos usado Kotlin con corrutinas y su librería de serialización para hacer el paso de datos, con las corrutinas conseguimos hacer que la aplicación funcione de manera asíncron. Esto lo hacemos para realizar aplicaciones no bloqueantes, las corrutinas son un conjunto de sentencias para realizar tareas específicas con la capacidad de suspender o resumir la ejecución sin bloquear hilos, se pueden ejecutar varias corrutinas en un mismo hilo, esto nos permite evitar la creación de una cantidad de hilos innecesaria cuidando los recursos

utilizados en la aplicación, facilita el recibir datos de una tarea asíncrona y facilita el intercambio de datos entre estas tareas. Trabajamos principalmente con Flows.

Para esto utilizamos esta línea para introducirlo al proyecto:
“implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.4.2")”



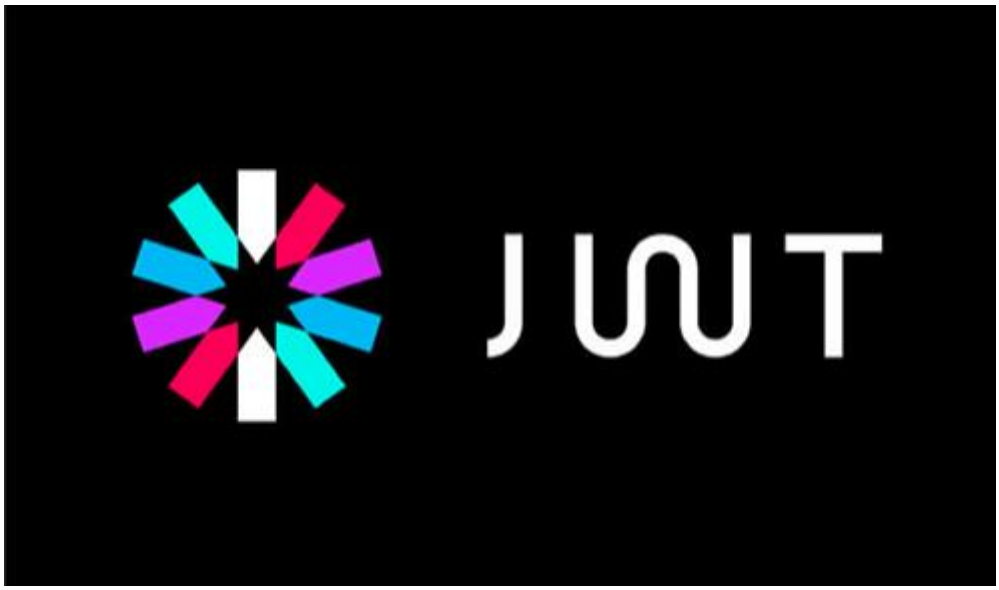
Gson:

Como trabajamos devolviendo JSON’s con los datos hemos utilizado la librería de Google “Gson” para devolver los datos utilizando pretty printing a su vez para una visualización de datos más legible y limpia. Gson es una librería de código abierto que nos permite la serialización y deserialización de objetos en la JVM aportándonos representación de los objetos con notación JSON. Esta librería nos permite la conversión entre objetos creados dentro del entorno de la jvm a notación json a través de los métodos toJson() y fromJson(), para ello tenemos que crear cada modelo implementando la interfaz serializable.

Cache4k:

Para agilizar el tiempo de respuesta hemos implementado un cacheado de la información utilizando para ello la librería cache4k, de esta manera no creamos una caché en una base de datos alternativa en el dispositivo, sino que creamos un guardado en memoria disponible mientras el dispositivo esté encendido, cache4k también soporta funciones como tiempo de expiración y restricciones de tamaño por lo que era una propuesta interesante a implementar, solo soporta el nuevo modelo de memoria nativo de kotlin, pero es multiplataforma. Se trata básicamente de un mapa guardado en memoria en

JWT:



Necesitábamos hacer uso de tokens para la verificación de usuarios, para ello hemos hecho uso de JWT para que generase los tokens con los datos de nuestros modelos y así tener los tokens de validación que poder consultar y comprobar durante la ejecución del programa. JSON Web Tokens es el estándar para las representaciones de peticiones.

Bcrypt:

Para el guardado de las contraseñas hemos usado Bcrypt, un cifrador de alta potencia que nos permite implementar una capa de seguridad al no dejar las contraseñas visibles en la base de datos. Se trata de un algoritmo de hash con varias ventajas, entre ellas se encuentra el uso aleatorio de secuencias añadidas a las contraseñas para hacer más difícil la decodificación a la fuerza, de esta manera previene las tablas de hash de palabras claves.