TENISLAB NOSQL

ALEJANDRO LÓPEZ ABAD

RUBEN GARCÍA-REDONDO MARÍN

Contenido

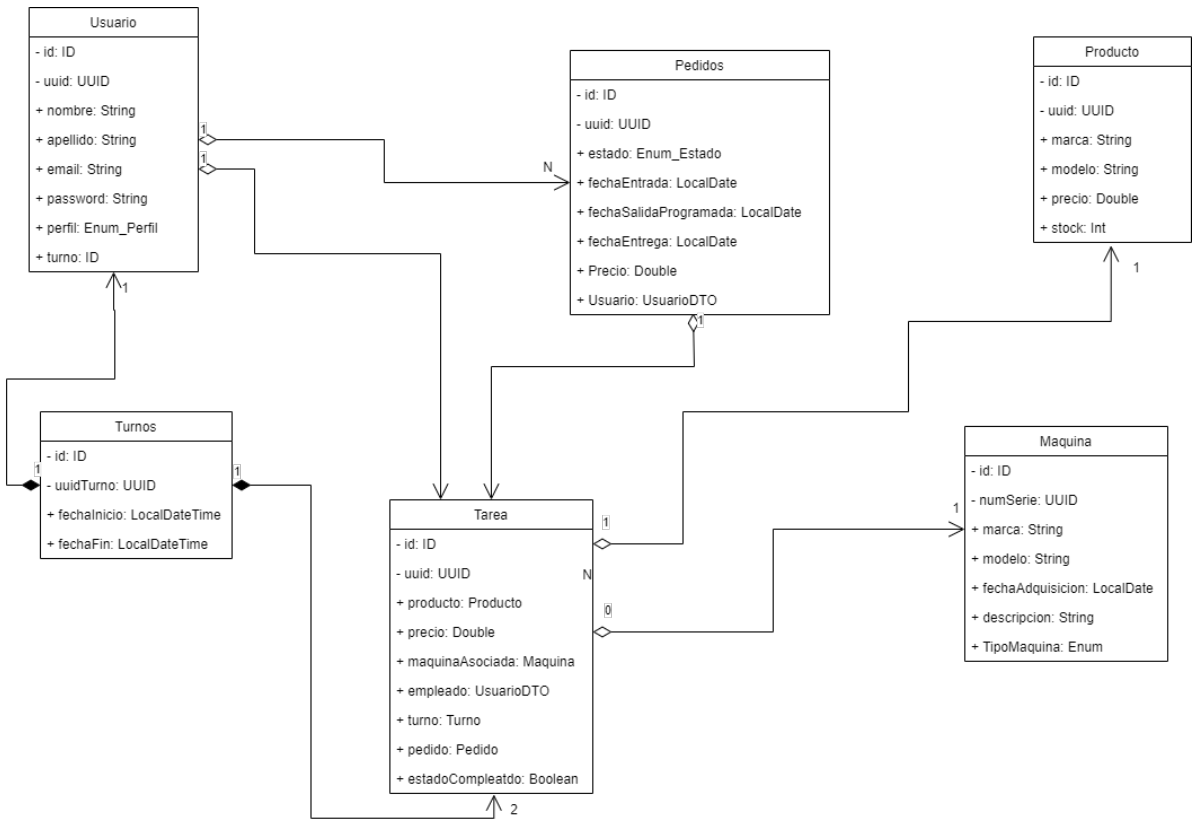
1. Introducción

Tenemos el mismo problema que en la práctica anterior, se necesitaba una aplicación para gestionar los pedidos del torneo Conde de Godó. Para ello tendremos que diseñar con una serie de restricciones y utilizando esta vez una base de datos NoSQL (No relacional).

2. Requisitos de información

Para poder realizar esta práctica hemos estado informándonos de los ciertos tipos de productos, máquinas y demás utilidades para poder realizarlo. También tuvimos que informarnos de cómo encriptar la contraseña de los usuarios, que, más adelante veremos qué tecnologías hemos utilizado.

3. Diagrama de clases



Aquí podemos observar el diagrama de clases, donde tenemos 6 clases (Usuario, Turno, Tarea, Pedido, Producto y Máquina). Los principales cambios que tenemos son que ahora solo tendremos una clase de máquina, que, podremos ver su tipo con una variable “TipoMaquina”, así podremos diferenciarlas y, por otro lado, todos los campos que teníamos en máquina los hemos reunido en un solo campo llamado “Descripción”. Tendremos una

tarea que será la que tendremos más generalizada, ya que tiene un producto, una máquina, el empleado, el turno en el cual se ha realizado esa tarea y el pedido al cual pertenece. Tenemos que tener bien en cuenta que un empleado no puede realizar tareas si no existe un turno al cual asignarle, al igual que, una tarea no puede realizarse si no existe un turno, ya que se necesita estar en un turno para poder cumplir la siguiente restricción: “Un encordador no puede realizar 2 tareas en un mismo turno”. Entendemos que un encordador no puede realizar más de 2 tareas simultáneamente que estén “PENDIENTES”. Dentro de los pedidos tendremos una fecha de cuando se realizó el pedido, una fecha de salida programada, y una fecha de entrega final. Dentro de los productos, tenemos tanto nombre, como marca y modelo del producto, además de un stock restante. Todas las clases tendrán un identificador de ID para buscar dentro de la base de datos, y un UUID para buscar desde fuera a la base de datos.

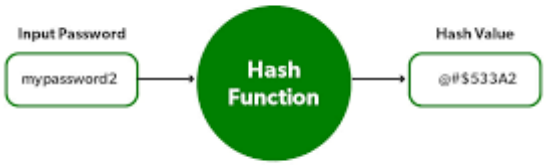
4. Tecnologías utilizadas

Para realizar la siguiente práctica hemos utilizado varías tecnologías. Explicaremos las más importantes y novedosas en cuanto a la práctica anterior:

- MongoDb

Esta tecnología será la principal para poder guardar los datos que utilicemos. Se trata de una base de datos no relacional donde guardaremos colecciones distintas. Nosotros para el almacenamiento utilizaremos “MongoDB Atlas”, para poder utilizar su tecnología de tiempo real.

- BCrypt



Este es un ejemplo de la tecnología que hemos utilizado para poder cifrar las contraseñas de los usuarios que tengamos registrados. Tenemos una cadena de texto con la contraseña y al pasarla por BCrypt tendremos un valor hash que guardaremos para que así esté cifrada y protegida.

- SqlDelight

Hemos utilizado esta tecnología para la parte de solo Mongo. Sirve para realizar una caché local para no tener que estar haciendo llamadas a la base de datos todo el tiempo.

- Ktorfit

Hemos decidido utilizar esta tecnología ya que está diseñada para poderse utilizar con Kotlin. Utilizaremos Ktorfit para poder hacer llamadas a la api que se nos pide en los requisitos.

- Koin

Para utilizar una inyección de dependencias en la parte de Mongo, hemos decidido utilizar Koin, ya que nos proporciona de una manera simple y con módulos las dependencias que tengamos.

- JUnit 5

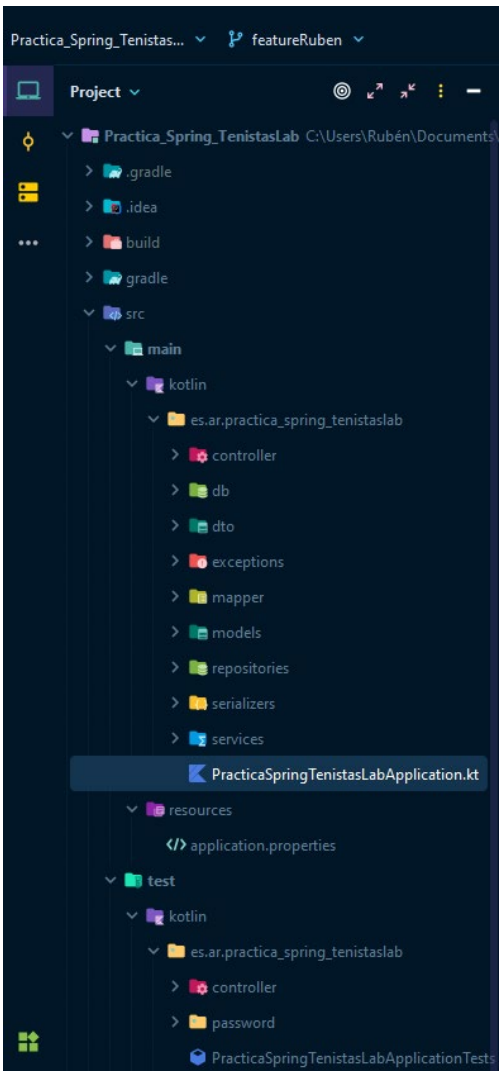
Para realizar los test, hemos utilizado JUnit, junto a test asíncronos.

- Spring Boot

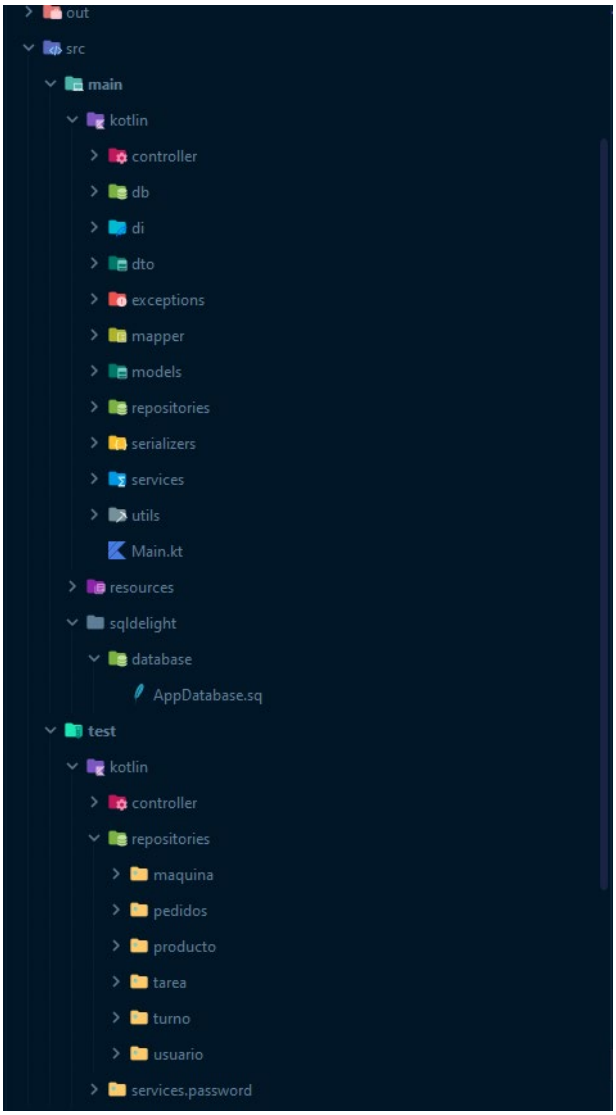
Para la parte de la práctica donde nos piden utilizar Spring más su inyección propia de dependencias, hemos utilizado Spring Boot con sus repositorios que nos da la propia tecnología.

5. Arquitectura del sistema

Este proyecto está dividido en dos ya que está resuelto con MongoDB nativo y con Spring y Mongo, por lo que en cada proyecto tendremos una estructura, para Spring lo hemos ordenado de la siguiente manera:



Para Mongo nativo lo hemos ordenador de la siguiente manera:



Como podemos observar, la diferencia entre un proyecto y otro, es la inyección de dependencias, ya que para Koin necesitábamos diseñar una clase que tuviese un módulo dentro con las dependencias necesarias, y para Spring, utilizamos su propia inyección de dependencias que trae. Además, otra diferencia es que en el de Mongo, tenemos una carpeta para SQL Delight para decirle las sentencias SQL que debe realizar, y la estructura de la tabla.

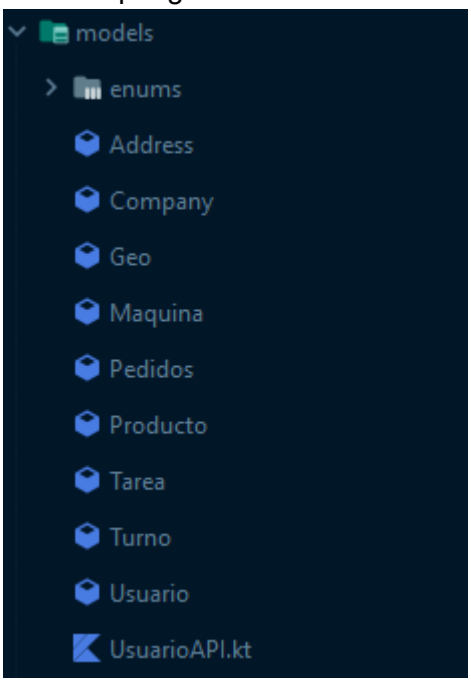
- Inyección de dependencias en Mongo (Utilizando Koin):

```
@Module
@ComponentScan("koin")
class DiAnnotationModule {
    val DiDslModule = module { this: Module
        // StringFormat
        single<StringFormat>(named( name: "StringFormatJson")) { Json { prettyPrint = true } }

        // Repository
        single<MaquinaRepository> { MaquinaRepositoryImpl() }
        single<PedidosRepository> { PedidosRepositoryImpl() }
        single<ProductoRepository> { ProductoRepositoryImpl() }
        single<TareasRepository> { TareasRepositoryImpl() }
        single<TareasRepositoryKtorfit> { TareasRepositoryKtorfit() }
        single<TurnoRepository> { TurnoRepositoryImpl() }
        single<UsuarioRepositoryKtorfit> { UsuarioRepositoryKtorfit() }
        single<UsuarioRepository> { UsuarioRepositoryImpl() }
        single<RemoteCachedRepositoryUsuario> { RemoteCachedRepositoryUsuario() }
        single<UsuariosService> { UsuariosService() }


        // Controlador
        single(named( name: "ControladorTenistas")) { Controlador(get(), get(),
            get(), get(), get(), get(),
            get(), get(), get()) }

    }
}
```

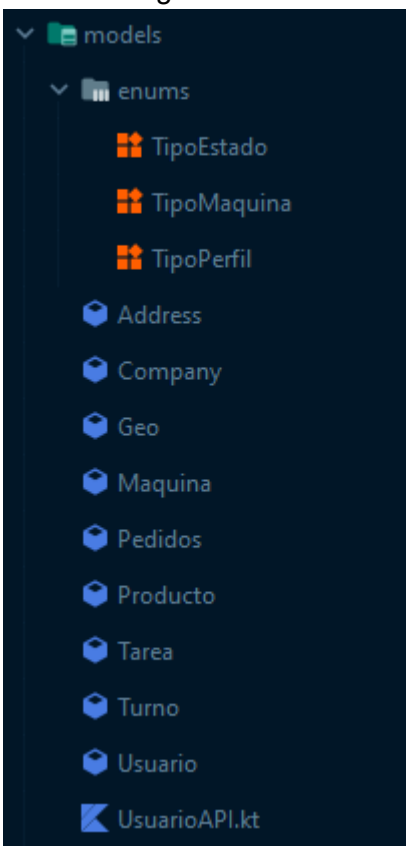
Para utilizar Koin debemos primero crearnos una clase con las anotaciones @Module y @ComponenScan(“koin”), para decirle a Koin que esta es la clase donde cogerá las dependencias que necesitemos. Crearemos un módulo con “module” Utilizaremos el single y el tipo de repositorio y después le diremos que implementación deberá coger.

5.1. Modelos

- Spring:



- Mongo:



- Modelo Máquina:

```kotlin
@Serializable
data class Maquina(
        @BsonId @Contextual
        val id: Id<Maquina> = newId<Maquina>(),
        val numSerie: String = UUID.randomUUID().toString(),
        val marca: String,
        val modelo: String,
        @Serializable(LocalDateSerializer::class)
        val fechaAdquisicion: LocalDate,
        val descripcion: String,
        val tipo: TipoMaquina
    )
```

Aquí podemos observar el modelo máquina de Mongo. Como comentamos antes, ahora solo tenemos una clase máquina, donde en este caso, nuestro id será el generado por mongo. Para ello le tendremos que poner la anotación @BsonId. Para serializar en json hemos metido una dependencia para que el serialización de kotlin pueda serializar, aunque habrá que ponerle la anotación @Contextual para decirle que podrá encontrar esa serialización en tiempo de ejecución.

```kotlin
@Serializable
@Document("Maquina")
data class Maquina(
    @Id @Serializable(ObjectIdSerializer::class)
    val id: ObjectId = ObjectId.get(),
    val numSerie:String = UUID.randomUUID().toString(),
    val marca: String,
    val modelo: String,
    @Serializable(LocalDateSerializer::class)
    val fechaAdquisicion: LocalDate,
    val descripcion: String,
    val tipo: TipoMaquina
) {
}
```

Este es el modelo máquina de Spring. En este caso para decir que esta clase se utilice en una colección le pondremos la anotación de @Document así le diremos con qué nombre queremos que la genere. Como spring trabaja con ObjectId, solo con ponerle la anotación @Id nos servirá. Para poder utilizar la serialización de Kotlin, hemos tenido que crear un serializador. Tenemos también el tipo máquina que es un enum con 2 tipos, Personalización y Encordar.

- Modelo Pedidos:

```
@Serializable
data class Pedidos(
    @BsonId @Contextual
    val id: Id<Pedidos> = newId<Pedidos>(),
    val uuidPedidos: String = UUID.randomUUID().toString(),
    val estado: TipoEstado,
    @Serializable(LocalDateSerializer::class)
    val fechaEntrada: LocalDate,
    @Serializable(LocalDateSerializer::class)
    val fechaSalidaProgramada: LocalDate,
    @Serializable(LocalDateSerializer::class)
    val fechaEntrega: LocalDate?,
    val precio: Double,
    @Contextual
    val usuario: Id<Usuario>
):
```

Igual que el modelo maquinas en Mongo, tenemos un Id generado por Mongo, además podemos ver una relación ya con Usuarios.

```
@Serializable
@Document("pedidos")
data class Pedidos(
    @Id @Serializable(ObjectIdSerializer::class)
    val id: ObjectId = ObjectId.get(),
    val uuidPedidos: String = UUID.randomUUID().toString(),
    val estado: TipoEstado,
    @Serializable(LocalDateSerializer::class)
    val fechaEntrada: LocalDate,
    @Serializable(LocalDateSerializer::class)
    val fechaSalidaProgramada: LocalDate,
    @Serializable(LocalDateSerializer::class)
    val fechaEntrega: LocalDate?,
    val precio: Double,
    val usuario: UsuarioDTO,
) {
```

Aquí tenemos el modelo de pedidos en Spring, que igual que antes ponemos las anotaciones y como Id el ObjectId.

- Modelo Producto:

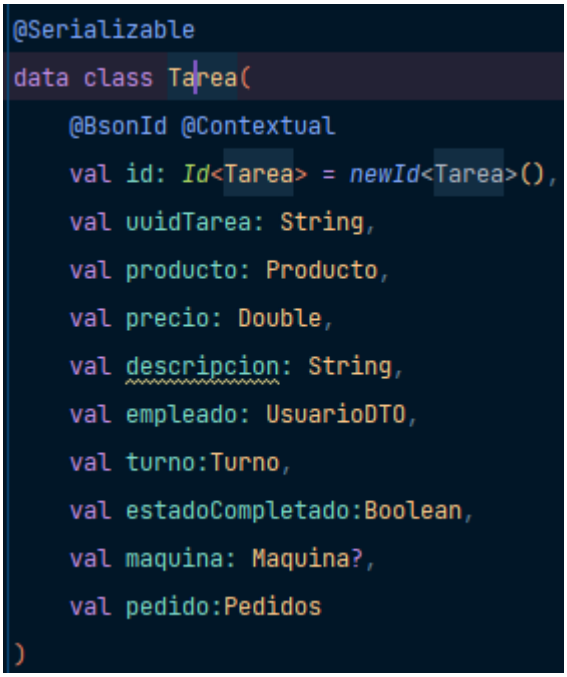
```
@Serializable
data class Producto(
    @BsonId @Contextual
    val id: Id<Producto> = newId<Producto>(),
    val uuidProducto: String = UUID.randomUUID().toString(),
    val marca: String,
    val modelo: String,
    val precio: Double,
    val stock: Int
)
```

Aquí tenemos el modelo de producto en Mongo.

```
@Serializable
@Document("producto")
data class Producto(
    @Id @Serializable(ObjectIdSerializer::class)
    val id: ObjectId= ObjectId.get(),
    val uuidProducto:  String = UUID.randomUUID().toString(),
    val marca: String,
    val modelo: String,
    val precio: Double,
    val stock: Int
)
```

Este es el modelo de producto en Spring.

- Modelo tarea:



```
@Serializable
data class Tarea(
    @BsonId @Contextual
    val id: Id<Tarea> = newId<Tarea>(),
    val uuidTarea: String,
    val producto: Producto,
    val precio: Double,
    val descripcion: String,
    val empleado: UsuarioDTO,
    val turno:Turno,
    val estadoCompletado:Boolean,
    val maquina: Maquina?,
    val pedido:Pedidos
)
```

Este es nuestro modelo de tarea en Mongo. Es el que más relaciones tiene ya que tiene un producto, un encordador, una máquina, un turno y un pedido asignado.

```kotlin
@Serializable
@Document("tarea")
data class Tarea(
    @Id @Serializable(ObjectIdSerializer::class)
    val id: ObjectId = ObjectId.get(),
    val uuidTarea: String = UUID.randomUUID().toString(),
    val producto: Producto,
    val precio: Double,
    val descripcion: String,
    val empleado:UsuarioDTO,
    val turno:Turno,
    val estadoCompletado:Boolean,
    val maquina: Maquina?,
    val pedido:Pedidos
) {
```

Este sería el modelo de Tarea en Spring, con sus anotaciones de @Document

- Modelo Turno:

```kotlin
@Serializable
data class Turno(
    @BsonId @Contextual
    val id : Id<Turno> = newId<Turno>(),
    val uuidTurno: String = UUID.randomUUID().toString(),
    @Serializable(LocalDateTimeSerializer::class)
    val fechaInicio: LocalDateTime,
    @Serializable(LocalDateTimeSerializer::class)
    val fechaFin: LocalDateTime
)
```

Este es el modelo turno en Mongo. Tenemos una fecha de inicio del turno y una fecha de final del turno.

```kotlin
@Serializable
@Document("turno")
data class Turno(
    @Id @Serializable(ObjectIdSerializer::class)
    val id: ObjectId = ObjectId.get(),
    val uuidTurno: String = UUID.randomUUID().toString(),
    @Serializable(LocalDateTimeSerializer::class)
    val fechaInicio: LocalDateTime,
    @Serializable(LocalDateTimeSerializer::class)
    val fechaFin: LocalDateTime
) {
```

Este sería el modelo de turno en Spring.

- Modelo Usuario:
  El último modelo que hemos utilizado en esta práctica sería el de Usuario:
  De este modelo se puede mencionar que tiene un Tipo de Perfil ya que el usuario puede ser un cliente/tenista, un encordador/empleado o el Admin el cual tendrá permisos para todo, hicimos un enum con estos tipos para una vez que nos llega la petición el controlador sabrá si tiene permisos para realizar la petición que estamos pidiendo o no.

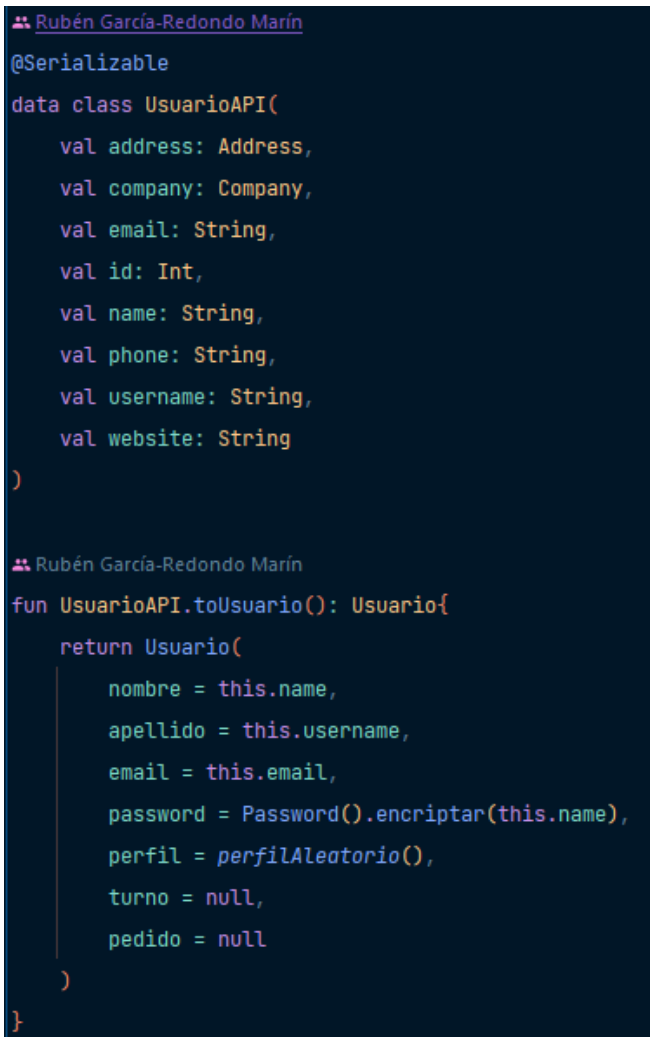
```
@Serializable
data class Usuario(
    @BsonId @Contextual
    val id: Id<Usuario> = newId(),
    val uuidUsuario: String = UUID.randomUUID().toString(),
    val nombre: String,
    val apellido: String,
    val email: String,
    val password: ByteArray,
    val perfil: TipoPerfil,
    @Contextual
    val turno: Id<Turno>?,
    @Contextual
    val pedido: List<Pedidos>?
) {
```

El usuario tendrá un turno que si es un cliente se quedará a nulo y si es un empleado estará asignado en el turno en el este haciendo las tareas.
Su diseño en mongo sería el siguiente:

```
@Serializable
@Document("usuario")
data class Usuario(
    @Id @Serializable(ObjectIdSerializer::class)
    val id: ObjectId = ObjectId.get(),
    val uuidUsuario:String = UUID.randomUUID().toString(),
    val nombre: String,
    val apellido: String,
    val email: String,
    val password: ByteArray,
    val perfil: TipoPerfil,
    val turno: Turno?,
){
```

En Spring sería así su diseño:

- Modelo UsuarioApi:



```
Rubén García-Redondo Marín
@Serializable
data class UsuarioAPI(
    val address: Address,
    val company: Company,
    val email: String,
    val id: Int,
    val name: String,
    val phone: String,
    val username: String,
    val website: String
)

Rubén García-Redondo Marín
fun UsuarioAPI.toUsuario(): Usuario{
    return Usuario(
        nombre = this.name,
        apellido = this.username,
        email = this.email,
        password = Password().encriptar(this.name),
        perfil = perfilAleatorio(),
        turno = null,
        pedido = null
    )
}
```

En los 2 casos hemos utilizado este modelo para poder consultar los usuarios que teníamos en nuestra api.

5.2. Repositorios:

Para llevar a cabo los siguientes repositorios en la práctica de mongo nos crearemos una interfaz CrudRepository

```kotlin
/**
 * Crud repository
 *
 */
interface CrudRepository<T, ID> {
    /**
     * Find all
     *
     * @return
     */
    fun findAll(): Flow<T>

    /**
     * Find by id
     *
     * @param id
     * @return
     */
    suspend fun findById(id: ID): T?

    /**
     * Find by u u i d
     *
     * @param uuid
     * @return
     */
    suspend fun findByUUID(uuid: String): T?

    /**
     * Save
     *
     * @param entity
     * @return
     */
    suspend fun save(entity: T): T?

    /**
     * Delete
     *
     * @param entity
     * @return
     */
    suspend fun delete(entity: T): Boolean
}
```

Como se puede ver las funciones son suspendidas ya que estamos trabajando con corrutinas, a continuación, mostrare el repositorio de Usuario en el cual utilizamos nuestra clase MongoDbManager la cual nos permite trabajar con las colecciones y hacer las operaciones CRUD

```
 *
 */
object MongoDbManager {
    val logger = KotlinLogging.logger {}
    val properties = PropertiesReader( fileName: "application.properties")
    private lateinit var mongoClient: CoroutineClient
    lateinit var database: CoroutineDatabase

    // Para Mongo Atlas
    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"

    init {
        logger.debug( msg: "Inicializando conexion a MongoDB")
        println("Inicializando conexion a MongoDB -> $MONGO_URI$OPTIONS")
        mongoClient =
            KMongo.createClient( connectionString: "$MONGO_URI$OPTIONS")
                .coroutine
        database = mongoClient.getDatabase( name: "tenistas")
    }
}
```

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

@Single
@Named("UsuarioRepositoryImpl")
class UsuarioRepositoryImpl:UsuarioRepository {

    val cache = SqlDeLightClient.queries
    val ktorfit = UsuarioRepositoryKtorfit()
    override fun findAll(): Flow<Usuario> {
        logger.debug { "findAll($)" }
        return MongoDbManager.database.getCollection<Usuario>()
            .find().publisher.asFlow()
    }

    override suspend fun findById(id: Id<Usuario>): Usuario? {
        logger.debug { "findById($id)" }
        return MongoDbManager.database.getCollection<Usuario>()
            .findOneById(id) ?: throw UsuarioException("No existe el Usuario con id $id")
    }

    override suspend fun findByUUID(uuid: String): Usuario? {
        logger.debug { "findByUUID($uuid)" }
        return MongoDbManager.database.getCollection<Usuario>().findOne( filter: Usuario::uuidUsuario eq uuid)?: throw UsuarioException("No existe el Usuario con el uuid $uuid")
    }
```

```
override suspend fun save(entity: Usuario): Usuario? {
    logger.debug { "save($entity)" }
    cache.insertUser(
        entity.id.toString(),
        entity.uuidUsuario,
        entity.nombre,
        entity.apellido,
        entity.email,
        entity.password.toString(),
        entity.perfil.name,
        entity.turno.toString(),
        entity.pedido.toString())
    return MongoDbManager.database.getCollection<Usuario>()
        .save(entity).let { entity }
}

private suspend fun insert(entity: Usuario): Usuario {
    logger.debug { "save($entity) - creando" }
    return MongoDbManager.database.getCollection<Usuario>()
        .save(entity).let { entity }
}

private suspend fun update(entity: Usuario): Usuario {
    logger.debug { "save($entity) - actualizando" }
    return MongoDbManager.database.getCollection<Usuario>()
        .save(entity).let { entity }
}

override suspend fun delete(entity: Usuario): Boolean {
    logger.debug { "delete($entity)" }
    cache.delete(entity.id.toString())
    return MongoDbManager.database.getCollection<Usuario>()
        .deleteOneById(entity.id).let { true }
}
```

Los otros dos repositorios nuevos que hay que mencionar es el de Ktorfit y el repositorio que se encarga de la cache.
Para Ktorfit tendremos primero que crear el cliente de la siguiente manera->

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

object KtorFitClient {
    private const val API_URL = "https://jsonplaceholder.typicode.com/"


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


    val instance by lazy {

        ktorfit.create<KtorFitRest>()
    }
```

Esta clase esta implementada en los dos proyectos ya que las consultas a la api de los dos proyectos están realizadas con Ktorfit. Necesitaremos una interfaz para el funcionamiento de Ktorfit donde introduciremos los endpoints y con las anotaciones de Ktorfit indicaremos que tipo de consulta se va a realizar a la API

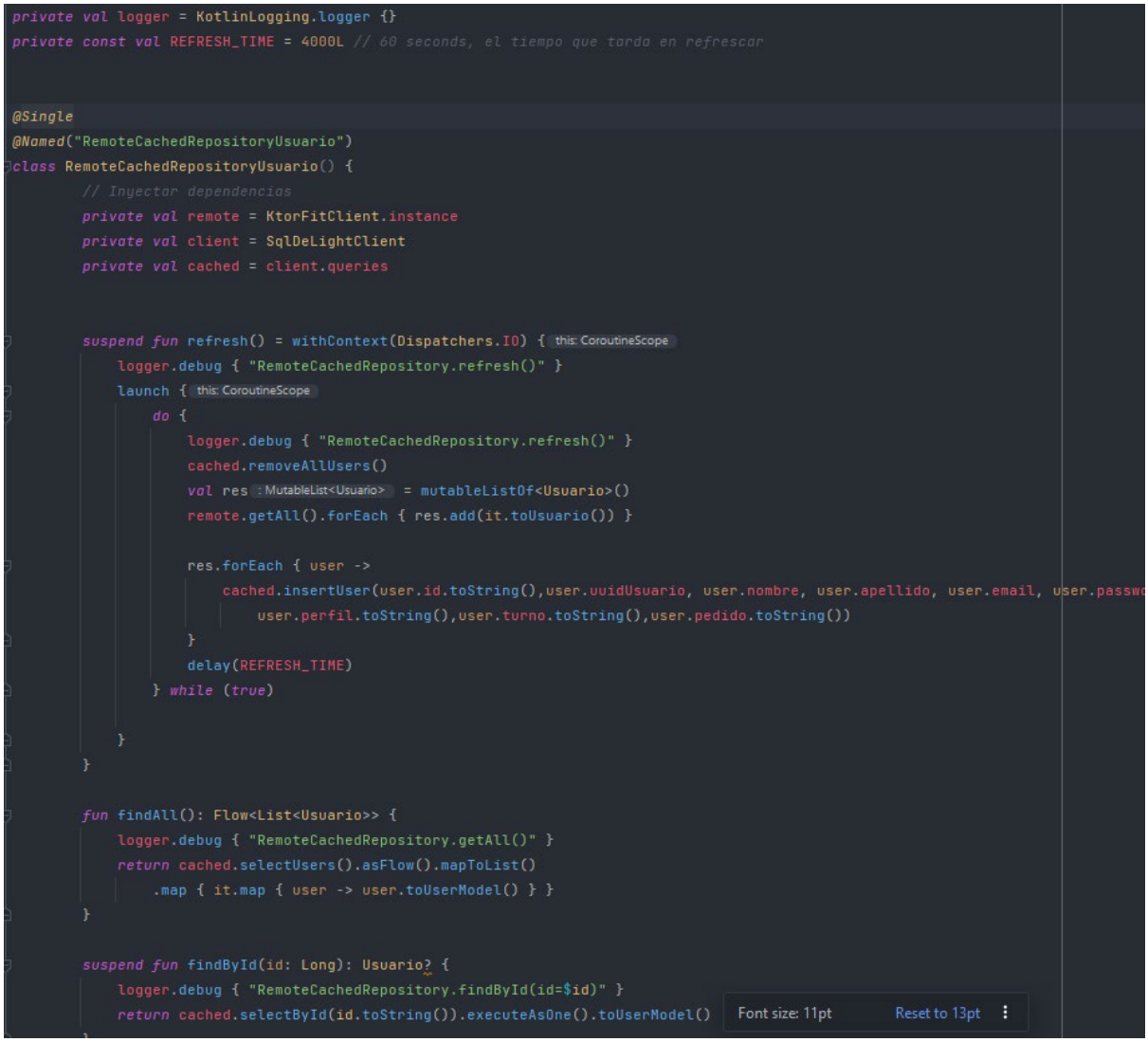
```
interface KtorFitRest {

    @GET("users")
    suspend fun getAll(): List<UsuarioAPI>

    @GET("users/{id}")
    suspend fun getById(@Path("id") id: Int): UsuarioAPI?

    @POST("users")
    suspend fun create(@Body user: UsuarioDTO): UsuarioDTO

    @PUT("users/{id}")
    suspend fun update(@Path("id") id: String, @Body user: UsuarioDTO): UsuarioDTO

    @PATCH("users/{id}")
    suspend fun upgrade(@Path("id") id: Long, @Body user: UsuarioDTO): UsuarioDTO

    @DELETE("users/{id}")
    suspend fun delete(@Path("id") id: String): Unit

    @POST("todos")
    suspend fun createTareas(@Body tarea: TareaDto): TareaDto

}
```

Vamos a fijarnos en la anotación PUT, la función debajo de la anotación se encarga de buscar en la api un usuario con el id que le pasemos, con un body que sería la entidad usuario, en este caso nuestro usuarioDTO y devuelve un usuarioDTO.
Para llevar a cabo esta implementación creamos un repositorio específico para Ktorfit ->

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


@Single
@Named("UsuarioRepositoryKtorfit")
class UsuarioRepositoryKtorfit {

    private val client by lazy { KtorFitClient.instance }

    suspend fun findAll(): Flow<Usuario> = withContext(Dispatchers.IO) { this: CoroutineScope
        logger.debug { "findAll()" }
        val call : List<UsuarioAPI> = client.getAll()
        try {
            logger.debug { "findAll() - OK" }
            val res : MutableList<Usuario> = mutableListOf<Usuario>()
            call.forEach { it: UsuarioAPI
                res.add(it.toUsuario())
            }
            return@withContext res.asFlow()
        } catch (e: Exception) {
            logger.error { "findAll() - ERROR" }
            throw RestException("Error al obtener los usuarios: ${e.message}")
        }

    }

    suspend fun findById(id: Int): Usuario? {
        logger.debug { "finById(id=$id)" }
        val call : UsuarioAPI? = client.getById(id)
        try {
            logger.debug { "findById(id=$id) - OK" }
            return call?.toUsuario()
        } catch (e: Exception) {
            logger.error { "findById(id=$id) - ERROR" }
            throw RestException("Error al obtener el usuario con id $id o no existe: ${e.message}")
        }
    }
```

```
suspend fun save(entity: Usuario): Usuario {
    logger.debug { "save(entity=$entity)" }
    try {
        val dto :UsuarioDTO = entity.toUsuarioDto()
        val res :UsuarioDTO = client.create(dto)
        logger.debug { "save(entity=$entity) - OK" }
        return entity
    } catch (e: Exception) {
        logger.error { "save(entity=$entity) - ERROR" }
        throw RestException("Error al crear el usuario: ${e.message}")
    }

}

suspend fun update(entity: Usuario): Usuario {
    logger.debug { "update(entity=$entity)" }
    try {
        val dto :UsuarioDTO = entity.toUsuarioDto()
        val res :UsuarioDTO = client.update(entity.id.toString(), dto)
        logger.debug { "update(entity=$entity) - OK" }
        return entity
    } catch (e: RestException) {
        logger.error { "update(entity=$entity) - ERROR" }
        throw RestException("Error al actualizar el usuario con ${entity.id}: ${e.message}")
    }
}

suspend fun delete(entity: Usuario): Usuario {
    logger.debug { "delete(entity=$entity)" }
    try {
        client.delete(entity.id.toString())
        logger.debug { "delete(entity=$entity) - OK" }
        return entity
    } catch (e: Exception) {
        logger.error { "delete(entity=$entity) - ERROR" }
        throw RestException("Error al eliminar el usuario con ${entity.id}: ${e.message}")
    }
}
```

Y, por último, el repositorio de la cache el cual también es nuevo, cabe destacar que los repositorios normales que se utilizan para guardar la información en la base de datos de mongo son todos iguales, pero con sus respectivas entidades, ahora mostraremos nuestro repositorio que trabajara con la cache,
RepositorioCached Mongo ->

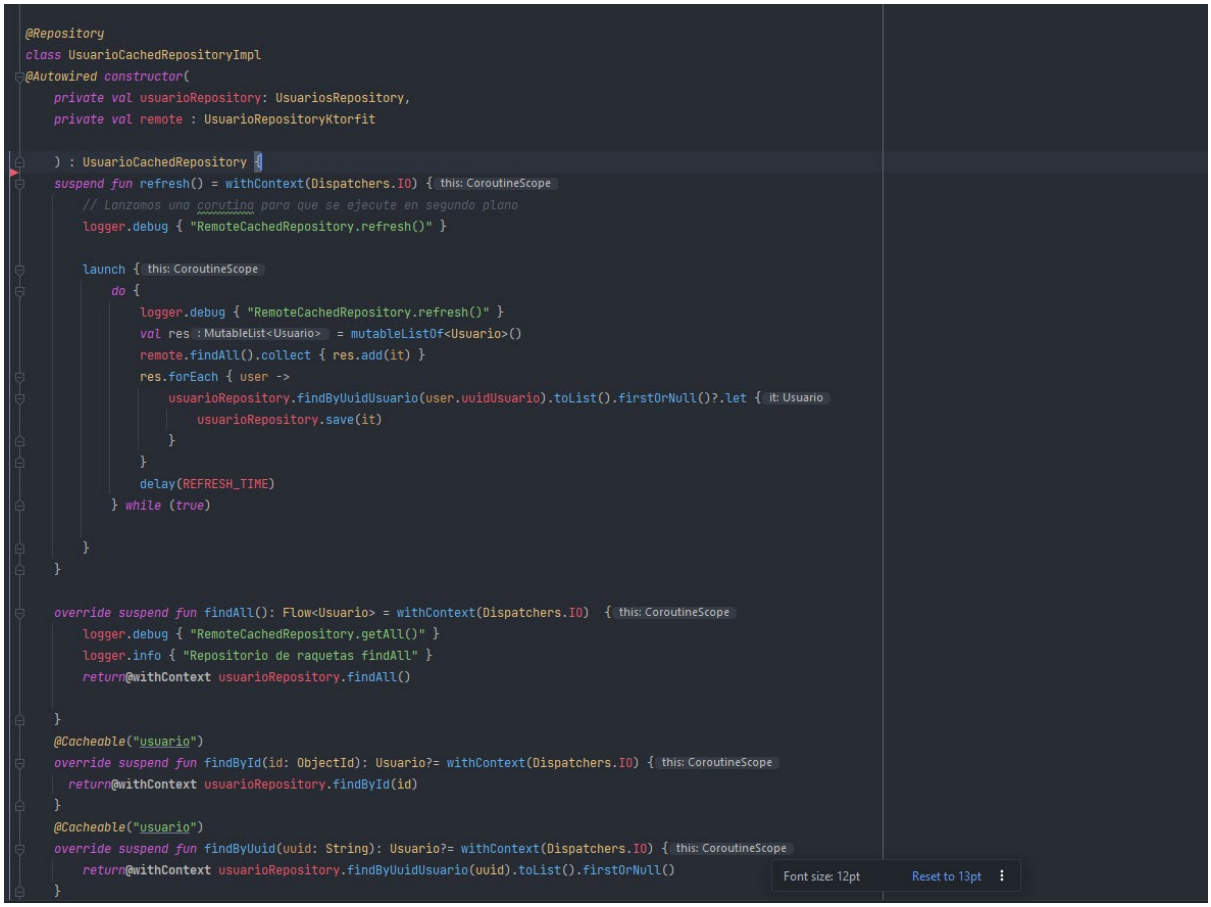


```kotlin
private val logger = KotlinLogging.logger {}
private const val REFRESH_TIME = 4000L // 60 seconds, el tiempo que tarda en refrescar


@Single
@Named("RemoteCachedRepositoryUsuario")
class RemoteCachedRepositoryUsuario() {
        // Inyectar dependencias
        private val remote = KtorFitClient.instance
        private val client = SqlDeLightClient
        private val cached = client.queries


        suspend fun refresh() = withContext(Dispatchers.IO) { this: CoroutineScope
            logger.debug { "RemoteCachedRepository.refresh()" }
            launch { this: CoroutineScope
                do {
                    logger.debug { "RemoteCachedRepository.refresh()" }
                    cached.removeAllUsers()
                    val res : MutableList<Usuario> = mutableListOf<Usuario>()
                    remote.getAll().forEach { res.add(it.toUsuario()) }

                    res.forEach { user ->
                        cached.insertUser(user.id.toString(),user.uuidUsuario, user.nombre, user.apellido, user.email, user.passwo
                            user.perfil.toString(),user.turno.toString(),user.pedido.toString())
                    }
                    delay(REFRESH_TIME)
                } while (true)

            }
        }

        fun findAll(): Flow<List<Usuario>> {
            logger.debug { "RemoteCachedRepository.getAll()" }
            return cached.selectUsers().asFlow().mapToList()
                .map { it.map { user -> user.toUserModel() } }
        }

        suspend fun findById(id: Long): Usuario? {
            logger.debug { "RemoteCachedRepository.findById(id=$id)" }
            return cached.selectById(id.toString()).executeAsOne().toUserModel()
        }
```

```kotlin
suspend fun save(entity: Usuario): Usuario {
    logger.debug { "RemoteCachedRepository.save(entity=$entity)" }
    val remote :UsuarioDTO = remote.create(entity.toUsuarioDto())
    val dto :Usuario = entity.toUserEntity()
    cached.insertUser(
        dto.id, dto.uuidUsuario, dto.nombre, dto.apellido, dto.email, dto.password, dto.perfil,
        dto.turno, dto.pedido
    )
    return cached.selectLastUser().executeAsOne().toUserModel()
}

suspend fun update(entity: Usuario): Usuario {
    // actualizamos localmente y remotamente
    logger.debug { "RemoteCachedRepository.update(entity=$entity)" }
    cached.update(
        id = entity.id.toString(),
        uuidUsuario = entity.uuidUsuario,
        nombre = entity.nombre,
        apellido = entity.apellido,
        email = entity.email,
        password = entity.password.toString(),
        perfil = entity.perfil.name,
        turno = entity.turno.toString(),
        pedido = entity.pedido.toString()
    )
    val dto :UsuarioDTO = remote.update(entity.id.toString(), entity.toUsuarioDto())
    return entity
}


suspend fun delete(entity: Usuario): Boolean {
    // borramos localmente y remotamente
    logger.debug { "RemoteCachedRepository.delete(entity=$entity)" }
    cached.delete(entity.id.toString())
    remote.delete(entity.id.toString())
    return true
}
```

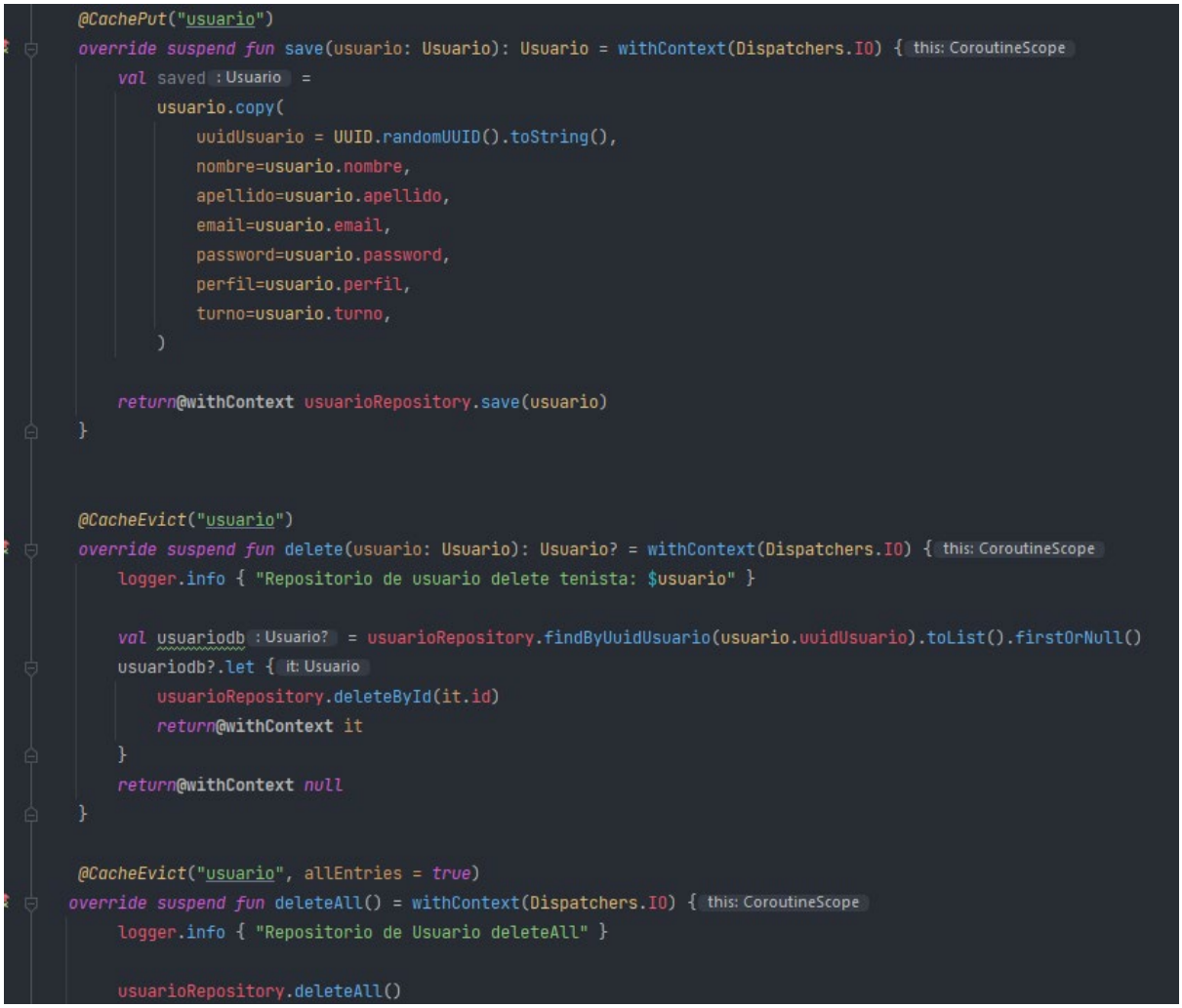
Una de las funciones a destacar en este repositorio seria la función de refresh la cual se encarga de refrescar la cache cada 60 s, en la práctica hemos puesto 4s para que se pudiera ver de forma más fácil en la consola, esto consiste en borrar todo lo que hay dentro de la cache y después hacer la consulta a la api para volver a guardar los usuarios

Comparada con la cache de Spring se puede ver fácilmente las diferencias, en spring hemos utilizado la propia cache de spring la cual es utiliza con anotaciones como voy a mostrar a continuación



```kotlin
@Repository
class UsuarioCachedRepositoryImpl
@Autowired constructor(
    private val usuarioRepository: UsuariosRepository,
    private val remote : UsuarioRepositoryKtorfit

    ) : UsuarioCachedRepository {
    suspend fun refresh() = withContext(Dispatchers.IO) { this: CoroutineScope
        // Lanzamos una coruting para que se ejecute en segundo plano
        logger.debug { "RemoteCachedRepository.refresh()" }

        launch { this: CoroutineScope
            do {
                logger.debug { "RemoteCachedRepository.refresh()" }
                val res : MutableList<Usuario> = mutableListOf<Usuario>()
                remote.findAll().collect { res.add(it) }
                res.forEach { user ->
                    usuarioRepository.findByUuidUsuario(user.uuidUsuario).toList().firstOrNull()?.let { it: Usuario
                        usuarioRepository.save(it)
                    }
                }
                delay(REFRESH_TIME)
            } while (true)

        }
    }

    override suspend fun findAll(): Flow<Usuario> = withContext(Dispatchers.IO)  { this: CoroutineScope
        logger.debug { "RemoteCachedRepository.getAll()" }
        logger.info { "Repositorio de raquetas findAll" }
        return@withContext usuarioRepository.findAll()

    }
    @Cacheable("usuario")
    override suspend fun findById(id: ObjectId): Usuario?= withContext(Dispatchers.IO) { this: CoroutineScope
      return@withContext usuarioRepository.findById(id)
    }
    @Cacheable("usuario")
    override suspend fun findByUuid(uuid: String): Usuario?= withContext(Dispatchers.IO) { this: CoroutineScope
        return@withContext usuarioRepository.findByUuidUsuario(uuid).toList().firstOrNull()
    }
```

Font size: 12pt Reset to 13pt



```
@CachePut("usuario")
override suspend fun save(usuario: Usuario): Usuario = withContext(Dispatchers.IO) { this: CoroutineScope
    val saved : Usuario =
        usuario.copy(
            uuidUsuario = UUID.randomUUID().toString(),
            nombre=usuario.nombre,
            apellido=usuario.apellido,
            email=usuario.email,
            password=usuario.password,
            perfil=usuario.perfil,
            turno=usuario.turno,
        )

    return@withContext usuarioRepository.save(usuario)
}


@CacheEvict("usuario")
override suspend fun delete(usuario: Usuario): Usuario? = withContext(Dispatchers.IO) { this: CoroutineScope
    logger.info { "Repositorio de usuario delete tenista: $usuario" }

    val usuariodb : Usuario? = usuarioRepository.findByUuidUsuario(usuario.uuidUsuario).toList().firstOrNull()
    usuariodb?.let { it: Usuario
        usuarioRepository.deleteById(it.id)
        return@withContext it
    }
    return@withContext null
}

@CacheEvict("usuario", allEntries = true)
override suspend fun deleteAll() = withContext(Dispatchers.IO) { this: CoroutineScope
    logger.info { "Repositorio de Usuario deleteAll" }

    usuarioRepository.deleteAll()
```

Del repositorio de spring podemos remarcar las anotaciones para que sea cacheable y también el cómo le inyectamos las dependencias con la anotación @ autowired

5.3. Servicios

Tenemos dos servicios a destacar uno es el de password el cual se encarga de cifrar la contraseña del usuario con BCrypt

```
 */
class Password {
    /**
     * Encripta la cadena con hash 12 rondas
     *
     * @param originalString Cadena a encriptar
     * @return la cadena encriptada
     */
    fun encriptar(originalString: String): ByteArray {
        return Bcrypt.hash(originalString, saltRounds: 12)
    }

    /**
     * Verifica que la cadena coincide con su hash
     *
     * @param originalString Cadena original
     * @param byteHash hash que se esperaba
     * @return true si la cadena coincide
     */
    fun verificar(originalString: String, byteHash: ByteArray): Boolean{
        return Bcrypt.verify(originalString, byteHash)
    }
}
```

y el segundo servicio es un watcher que se encarga de ver los cambios en tiempo real que ocurran en usuario

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

@Single
@Named("UsuarioService")
class UsuariosService {
    fun watch(): ChangeStreamPublisher<Usuario> {
        logger.debug { "watch()" }
        return MongoDbManager.database.getCollection<Usuario>() CoroutineCollection<Usuario>
            .watch<Usuario>() CoroutineChangeStreamPublisher<Usuario>
            .publisher
    }
}
```

5.4. Controlador:

En Mongo con inyección de dependencias de Koin:

```kotlin
@Single
@Named("ControladorTenistas")
class Controlador(
    @Named("MaquinaRepositoryImpl") private val maquinaRepositoryImpl: MaquinaRepository,
    @Named("PedidosRepositoryImpl") private val pedidosRepositoryImpl: PedidosRepository,
    @Named("ProductoRepositoryImpl") private val productoRepositoryImpl: ProductoRepository,
    @Named("TareasRepositoryImpl") private val tareasRepositoryImpl: TareasRepository,
    @Named("UsuarioRepositoryImpl") private val usuarioRepositoryImpl: UsuarioRepository,
    @Named("TurnoRepositoryImpl") private val turnoRepositoryImpl: TurnoRepository,
    @Named("UsuarioRepositoryKtorfit") private val ktorFitUsuario: UsuarioRepositoryKtorfit,
    @Named("RemoteCachedRepositoryUsuario") private val cacheRepositoryImpl: RemoteCachedRepositoryUsuario,
    @Named("UsuarioService") private val usuarioService: UsuariosService,
    var usuarioActual: Usuario? = null
) {
```

Para inyectar dependencias en el controlador hay que utilizar en nuestro caso @Named y dentro escribirle el nombre que le hayamos puesto en nuestro módulo.

Igualmente le ponemos el Single y el Named para luego referirnos en el Main.

Para Spring, utilizaremos este controlador:

```
@Controller
class Controlador
    @Autowired constructor(
     private val maquinaRepositoryImpl: MaquinaRepository,
     private val pedidosRepositoryImpl: PedidosRepository,
     private val productoRepositoryImpl: ProductoRepository,
     private val tareasRepositoryImpl: TareaRepository,
     private val usuarioRepositoryImpl: UsuariosRepository,
     private val turnoRepositoryImpl: TurnoRepository,
     private val ktorFitUsuario: UsuarioRepositoryKtorfit,
     private val cacheRepositoryImpl: UsuarioCachedRepositoryImpl,
     private val usuarioService: UsuariosService,

) {
    var usuarioActual: Usuario? = null
```

Utilizaremos el @Controller para decir a Spring que esta clase va a ser un controlador, y le pondremos el @Autowired para inyectar las dependencias que necesitamos.

Ahora veremos unos métodos del controlador que tienen especial interés ya que ahí miraremos las restricciones:

```
/**
 * Borrar maquina
 *
 * @param maquina
 * @return devuelve true si borra la maquina
 */
Rubén García-Redondo Marín
suspend fun borrarMaquina(maquina: Maquina): Boolean {
    val temp :Flow<Tarea> = listarTareas().filter { !it.estadoCompletado }
    return if(temp.count{ it.maquina?.numSerie == maquina.numSerie} == 0){
        if(usuarioActual!!.perfil == TipoPerfil.ADMINISTRADOR) {
            maquinaRepositoryImpl.delete(maquina)
            true
        }else{
            logger.error { "Solo un administrador puede borrar una máquina" }
            false
        }
    }else{
        logger.error { "No se puede borrar ya que esta máquina tiene asignadas tareas" }
        false
    }
}
```

Como podemos observar, en este método intentamos borrar una máquina que tengamos. Pero para ello antes debemos buscar si esa máquina está en alguna

tarea que aún no esté completada. En el caso de que no esté en ninguna tarea, entonces iremos a la siguiente restricción, y ver si nuestro usuario actual tiene el tipo de usuario “ADMINISTRADOR”, si es así, entonces procederemos a borrar dicha máquina.

```
/**
 * Guardar tarea
 *
 * @param tarea
 * @return guarda una Tarea
 *
 *En caso de que la tarea no este en un turno se podrá añadir ese turno
 * ,si no, no podrá añadirsea otro turno.
 */
Rubén García-Redondo Marín +1
suspend fun guardarTarea(tarea: Tarea): Tarea? {
    val temp :Flow<Tarea> = listarTareas()
    val turnoActual :Turno? = encontrarTurnoUUID(tarea.turno.uuidTurno)?.toList()?.firstOrNull()
    val empleado :Usuario? = encontrarUsuarioUUID(tarea.empleado.uuidUsuario)?.toList()?.firstOrNull()
    return if(usuarioActual!!.perfil == TipoPerfil.USUARIO) {
        if (turnoActual != null && empleado != null) {
            val veces :Int = temp
                .filter { !it.estadoCompletado }
                .filter { it.turno.uuidTurno == turnoActual.uuidTurno }
                .count { it.empleado.uuidUsuario == empleado.uuidUsuario }
            if (veces < 2) {
                tareasRepositoryImpl.save(tarea)
            } else {
                logger.error { "No puede tener 2 tareas en el mismo turno a la vez el empleado con uuid: ${empleado.uuidUsuario}" }
                null
            }
        } else {
            logger.error { "No existe el empleado: ${empleado!!.uuidUsuario}" }
            null
        }
    }else{
        logger.error{"No tienes permiso para guardar tareas"}
        null
    }

}
```

Aquí podemos observar la función de guardar tarea. Primero miraremos si el usuario es un trabajador o administrador, después miraremos si el empleado al cual le asignamos esa tarea existe, si es así, entonces miraremos cuantas tareas tiene “pendiente” ese empleado, si tiene 2, no dejaremos añadirle esa tarea, en caso contrario, guardaremos la tarea.

```kotlin
/**
 * Borrar usuario
 *
 * @param usuario
 * @return devuelve true si borra al usuario
 */
Rubén García-Redondo Marín
suspend fun borrarUsuario(usuario: Usuario): Boolean {
    return if(usuarioActual!!.perfil == TipoPerfil.ADMINISTRADOR) {
        if (usuario.perfil == TipoPerfil.ENCORDADOR) {
            val temp :Int = listarTareas().filter { !it.estadoCompletado }
                .count { it.empleado.uuidUsuario == usuario.uuidUsuario }
            if (temp == 0) {
                usuarioRepositoryImpl.delete(usuario)
                true
            } else {
                false
            }
        } else {
            usuarioRepositoryImpl.delete(usuario)
            true
        }
    }else{
        logger.error{"No tienes permisos para borrar usuarios"}
        false
    }
}
```

En esta función borraremos un usuario. Miraremos que el usuario actual que está intentando borrar ese usuario es de tipo “ADMINISTRADOR”. Si cumple esa restricción veremos si el usuario que queremos borrar tiene alguna tarea “PENDIENTE”. en el caso de que no tenga ninguna tarea pendiente, entonces le borraremos.