2023

AD-P03-TennisLab

ACCESO A DATOS

ALEJANDRO SÁNCHEZ MONZÓN

MIREYA SÁNCHEZ PINZÓN

Índice

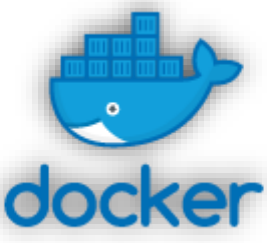
Introducción

La tercera práctica que hemos realizado reúne los conceptos trabajados en la anterior, siguiendo su modelo y contexto, pero esta vez basándonos en el modelo NoSQL, con implementaciones externas como son servicios a API REST o empleo de cachés.

La implementación de dicha aplicación cumple en nuestra práctica dos requisitos:

- Trabajo directo con un modelo **NoSQL** como son las bases de datos basadas en **MongoDB**.
- Trabajo con frameworks como **Spring Data**, desde el cual implementamos **MongoDB** de una forma semiautomática.

Por otra parte, hemos decidido montar un servidor local que sea el encargado de albergar la base de datos de MongoDB para la primera parte de la práctica, esta va ligada directamente a un servicio de **Docker**, el cual ejecutamos a placer mediante un docker-compose.



Dicho Docker determina a su vez las credenciales y puertos de entrada para los servicios necesarios para ejecutar nuestro servidor, viéndonos en la obligación de crear un manejador

para la base de datos, el cual, a nivel de programación, es el encargado de hacer funcionar la base de datos y conectar nuestro proyecto a la misma.

Todas las **credenciales** se leen desde un fichero **.properties**, el cual ha sido creado por nosotros.

A modo informativo, dejar constancia de que el lenguaje de programación base utilizado para este proyecto ha sido **Kotlin**, con el empleo de librerías nativas de Java y, a su vez, librerías externas tales como **Ktorfit**, **SqlDeLight**, **Mockito**...

Requisitos de información

Los requisitos de información para esta práctica compartían similitudes con la práctica anterior, una tienda de **productos** y **servicios** dentro del mundo del **tenis** que se ocupa de **gestionar** tanto la **compra** de recursos como el **tratado** de **raquetas** y **administración** de **producción**.

Esta vez, a diferencia de la anterior práctica, existían varios servicios a implementar, algunos de ellos, externalizados.

El objetivo ha sido crear un sistema basado en un **modelo NoSQL** en el cual tenemos un grupo de **usuarios** el cual **almacenamos en una API**, la cual ha sido estudiada para poder extraer sus datos y tratamiento de esta de la mejor forma posible.

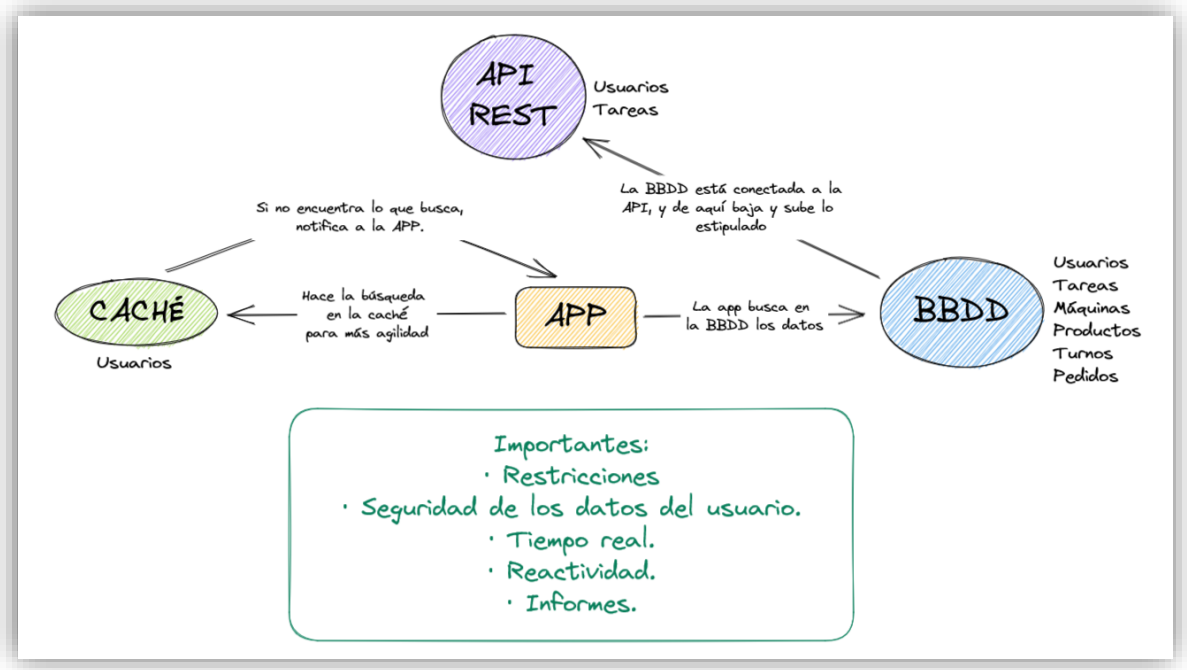
Por otro lado, estos **usuarios**, que pueden ser desde **clientes a trabajadores** (de varios tipos) tienen la capacidad de **realizar pedidos** o, por el contrario**, administrar la producción** interna de la tienda y su almacén.

Los usuarios clientes solo tienen la capacidad de realizar pedidos y conocer los productos y servicios (tareas) a su disposición. De estos elementos nos centramos en las **tareas**, las cuales están **ligadas a una maquina** con la cual el trabajador realiza su labor. Cada **máquina** ejecuta sus acciones **dentro de un turno**, el cual ha de ser respetado. Para finalizar, el programa ha de ser capaz de realizar diversos **informes** de una batería de datos seleccionados, para corroborar el buen funcionamiento del sistema.

Los usuarios no son los únicos ligados a un servicio externo, las **tareas** también están **conectadas a una API**, pero en este caso, con el objetivo de poder guardar un **histórico** en la **nube** de los datos sobre estas.

Por último, en relación con los servicios, los **usuarios** estarán conectados a una **caché** que se **refresca cada 60 segundos** para mantener una integridad de los datos y una fácil y eficaz consulta de estos, lo que **reduce los accesos a la base de datos,** haciendo el sistema más **rápido**.

A continuación, mostramos un breve esquema de **la estructura de información** del programa:



Del enunciado y los requisitos del cliente sacamos los siguientes requisitos:

- **Contraseñas cifradas** para fortalecer la seguridad del cliente.
- **Reactividad** en el sistema de base de datos para mejorar la eficacia de este.
- **Inyección de dependencias** a la hora de trabajar con la arquitectura del sistema.
- **Tipificación de resultados** y excepciones de forma personalizada.

Por otro lado, las restricciones en los requisitos de información son:

- Trabajadores (encordadores) con un **máximo de dos pedidos activos** por turno.

```
private suspend fun isPedidoOk(pedido: Pedido): Boolean {
    return if(pedido.tareas == null){
        true
    }else{
        val usuarios =
            listarPedidos()!!.toList().flatMap { it.tareas!! }.map { it.turno }
                .groupBy { it.encordador } Map<Usuario, List<Turno>>
                .filter { (_, turno) -> turno.size >= 2 }
                .map { it.key }

        pedido.tareas!!.any { it: Tarea
            usuarios.contains(it.turno.encordador)
        }
    }
}
```

A la hora de **guardar un nuevo pedido** en el sistema es importante verificar que los encordadores asignados a las tareas de dicho pedido, tengan por turno, como máximo, dos pedidos asignados.

Hemos resuelto esta restricción de la siguiente forma:

- Primero comprobamos que el pedido esté compuesto de **alguna tarea**, si no es así no habría ninguna restricción que resolver.

- En el caso de que, si la hubiese, resolveremos **esta restricción por partes**:

1. Hacemos una **búsqueda** de todos los **pedidos** que hay almacenados en nuestro sistema.

2. A continuación, nos centramos en las **tareas** que tiene asignada cada pedido, y de ellas, nos quedamos con los **turnos en las que son realizadas**.

3. A partir de ahí agrupamos en un **mapa** clave-valor los **encordadores** con los **turnos** que realizan y filtramos aquellos cuyos valores de turno **superen o sean igual a dos,** que, en ese caso, serán los que **no pueden hacerse cargo** de más tareas en sus turnos.

4. Por último, nos quedaremos con los **encordadores afectados** y comprobaremos que el encordador que se quiere asignar a la/s tarea/s del nuevo pedido **no esté entre los que ya no se pueden hacer cargo** de más tareas.

- Trabajadores (encordadores) con posibilidad de **utilizar solo una maquina** simultáneamente por **turno**.

```
suspend fun isTurnoOk(turno: Turno): Boolean {
    val turnosConElMismoEncordador = listarTurnos()?.filter { it.encordador == turno.encordador }
    val maquinaUtilizada = turnosConElMismoEncordador?.firstOrNull()?.maquina

    return turno.maquina == maquinaUtilizada
}
```

A la hora de guardar **un nuevo turno** es importante asegurarse que la **máquina** que va a utilizarse **coincida** con el **encordador** que la va a utilizar.

Hemos resuelto esta restricción de la siguiente forma:

- Primero hemos buscado todos los **turnos cuyo encordador** fuese el que se quiere asignar para el nuevo turno.
- A continuación, hemos encontrado la **máquina que utiliza en su turno**.
- Y, por último, hemos **verificado** que la máquina que se le quiere **asignar** en el nuevo turno es la que **utiliza** él.

- **Control de acceso a las operaciones CRUD** por tipo de trabajador (encargado, jefe, encordador).
- **Control de eliminaciones de datos** (la cuestión de los borrados en cascada).

```
require( value: listarTarea()?.filter { it.turno == turno }?.count() == 0)
{ "Antes de realizar la operación, elimine o actualice la tarea/s asociados a este turno. " }
```

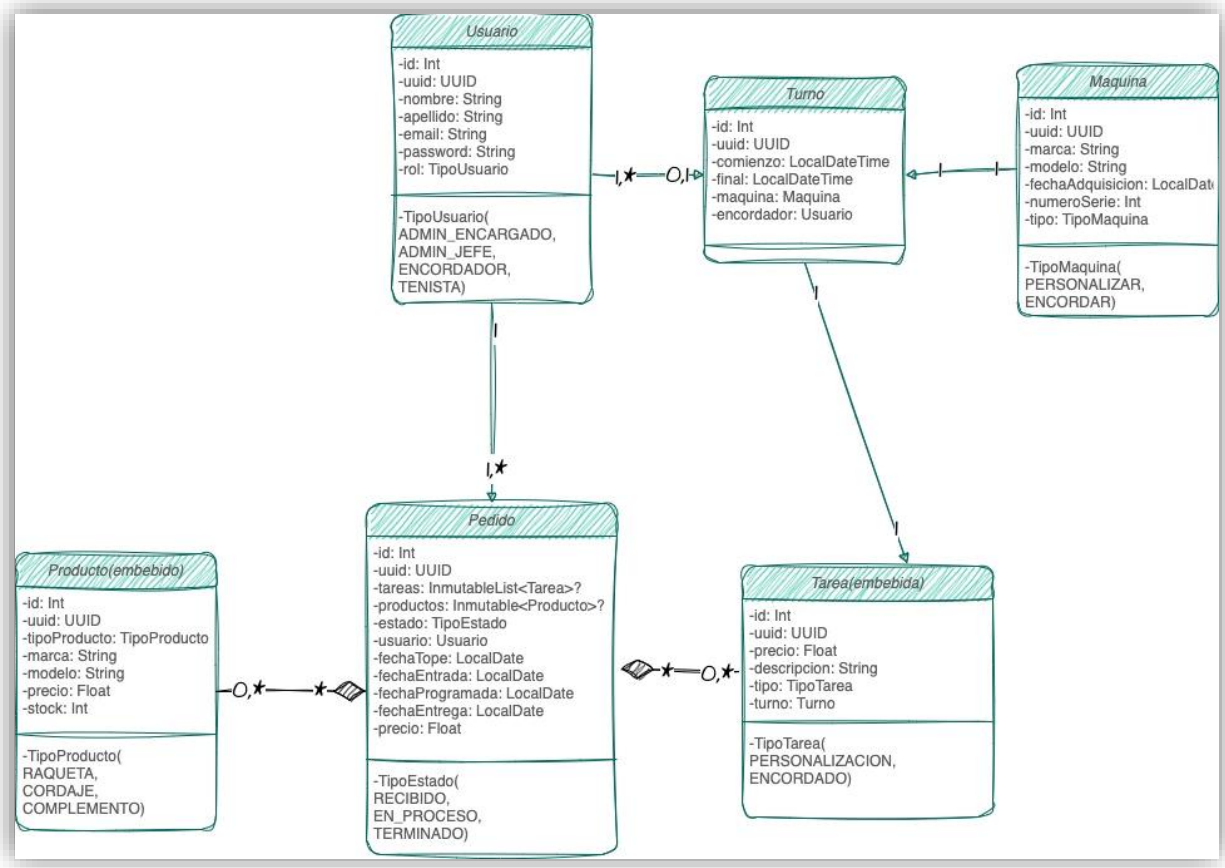
```
require( value: listarPedidos()?.filter { it.usuario == usuario }?.count() == 0)
{ "Antes de realizar la operación, elimine o actualice el pedido/s asociados a este usuario. " }
```

```
require( value: listarPedidos()?.map { it.tareas?.contains(tarea) }?.count() == 0)
{ "Antes de realizar la operación, elimine o actualice el pedido/s asociados a esta tarea." }
```

```
require( value: listarTurnos()?.filter { it.maquina == maquina }?.count() == 0)
{ "Antes de realizar la operación, elimine o actualice el turno/s asociados a esta máquina. " }
```

```
require( value: listarPedidos()?.map { it.productos?.contains(producto) }?.count() == 0)
{ "Antes de realizar la operación, elimine o actualice el pedido/s asociados a este producto." }
```

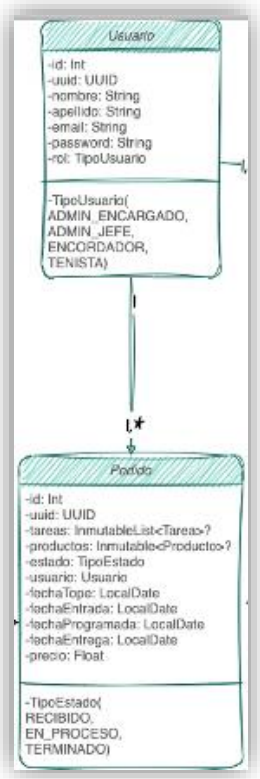
Diagrama de clases y justificación de este



Como hicimos en la práctica anterior, vamos a analizar la explicación del diagrama de clases parte por parte para así poder justificar todos sus aspectos y el por qué de esta interpretación del enunciado:

- Iniciamos el análisis del diagrama de clases por **Usuario**, los usuarios tendrán un id, uuid, nombre, apellido, email, password (cifrada) y un tipo, el cual puede ser encargado, jefe, encordador o tenista. Los usuarios tendrán relación directa con los pedidos y los turnos, comenzaremos por los pedidos.

    - Un **usuario** podrá realizar uno o **muchos pedidos**, mientras que cada uno de los **pedidos** podrán tener únicamente **un usuario asociado**.



- Los **pedidos**, los cuales tiene in id, un uuid, una lista de tareas posiblemente nula, una lista de productos posiblemente nula, un estado (recibido, en proceso o terminado), un usuario asociado, una fecha tope, una de entrada, una programada, una de entrega y un precio, se encontrará en situación de **composición** con **tareas** y **productos**, las cuales, en base al modelo NoSQL, irán **embebidas** a dicho modelo.

    - Un **pedido** es una **composición** de **productos**, es por esto por lo que incluimos en la clase dicho objeto. El producto a su vez tiene datos tales como id, uuid, tipo de producto, modelo, precio y stock.

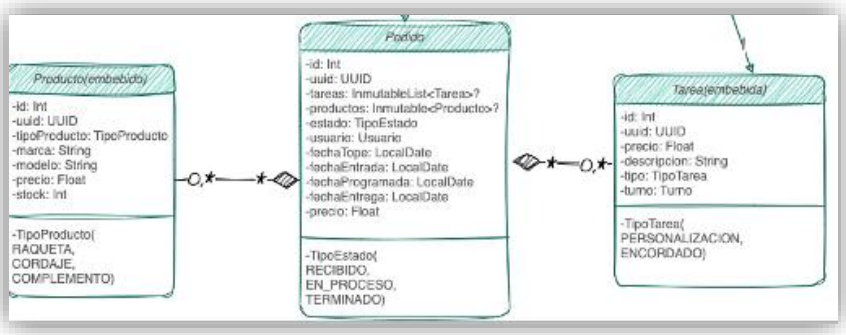
    - Una instancia concreta de Producto puede estar únicamente en un pedido, pero un pedido, puede tener varios productos o no tener ninguno.

    - Un **pedido** es, a su vez, una **composición** de **tareas**, las cuales poseen un id, un uuid un precio, una descripción, un tipo, y un turno.

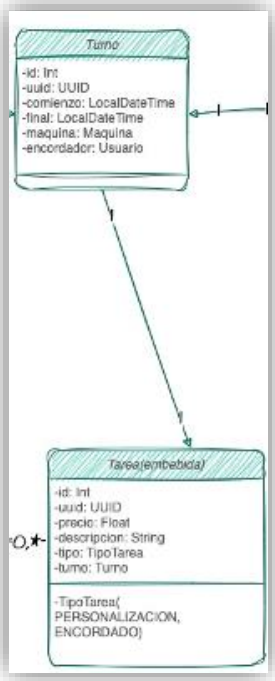
    - Una instancia concreta de Tarea puede estar únicamente en un pedido, pero un pedido, puede tener varias tareas o no tener ninguna.

El único requisito es que tiene que haber al menos uno de alguno de ellos para que el pedido pueda efectuarse. En la siguiente imagen se ve como lo hemos resuelto:

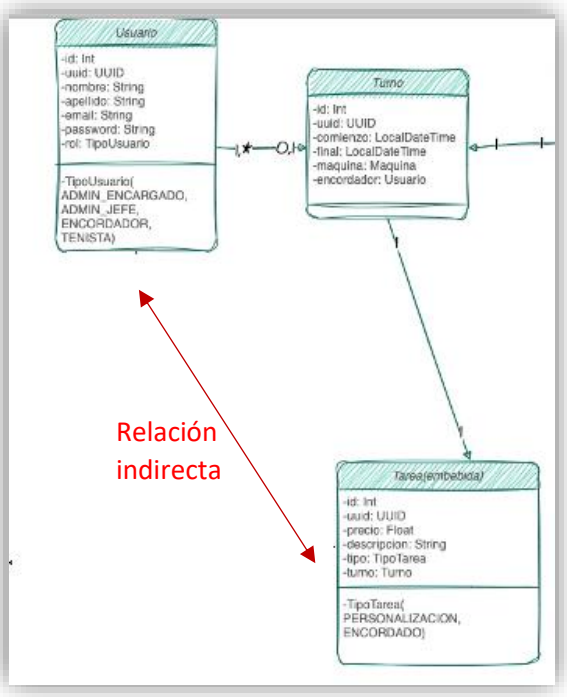
```
require( value: pedido.tareas ≠ null || pedido.productos ≠ null)
{ "El pedido no se puede realizar, su contenido está vacío." }
```



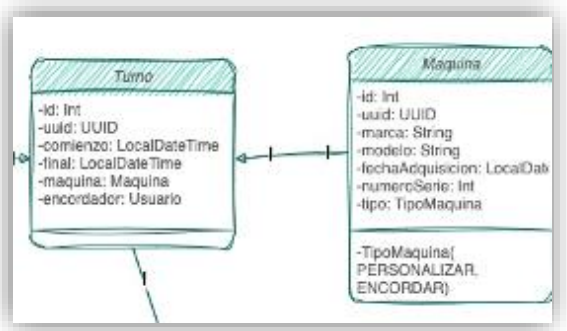
- Hablando de las **tareas** surge una nueva entidad en nuestro diagrama, los **turnos**, los cuales también se **asocian** con **usuarios**.
    - Una **tarea** tiene un **turno**, ese turno solo puede tener una tarea (recordamos que los turnos eran el momento en el que una acción específica se esta realizando)



- A su vez, los **turnos** tendrán un **usuario** de tipo NO tenista, el cual es el que ocupa ese **turno**. Es por esto por lo que podemos enlazar de forma **indirecta** que **usuario** realiza cada **tarea**: los que comparten cada turno.



- Los **turnos** tienen un id, un uuid, una fecha de comienzo y otra de final, una máquina y un usuario.
  - Esta **máquina**, la cual tiene un id, un uuid, un modelo, una fecha de adquisición, un número de serie y un tipo, solo podrá tener un **turno**, es decir, cada maquina trabaja en un turno y realiza una única tarea.



  - Es gracias a **turno** y a su relación con **tareas**, **maquinas** y **usuarios** que podemos **conectar todo el sistema** mediante una navegabilidad aparentemente sencilla. Ya que, si tenemos la información de este, tendremos también la información de los modelos relacionados. Todo esto empleando un sistema no relacional.

Alternativas y justificación detallada del modelo NoSQL

Entendemos que, en un proyecto como este, es fácil que surjan ideas e interpretaciones distintas del enunciado de este. Este tipo de situaciones se solucionan con un estudio/entrevista con el cliente para aclarar todo tipo de dudas, pero cuando hay cuestiones que pueden quedar sueltas, la decisión de cada grupo marca la diferencia dentro de la propuesta de solución.

A pesar de decantarnos por una solución concreta, parte del desarrollo del ejercicio ha sido barajar diferentes propuestas y elegir, mediante razonamiento, cual es el más recomendable.

- ### Embeber > Referenciar:

  A la hora de existir en nuestro sistema de BBDD **composiciones** entre las entidades (pedido compuesto de productos), la recomendación es **embeber** entre los modelos siguiendo la cardinalidad que se rija dentro de las relaciones. Si por el contrario tenemos **agregaciones** o la funcionalidad del sistema nos lo indica, trabajar con colecciones **referenciadas** puede ser una mejor solución.

  Esto es una cuestión que surge cuando incluimos al sistema el trabajo con un modelo NoSQL, visto esta que, en la práctica anterior, mediante modelos relacionales, este criterio se veía diferenciado.

- ### Generación de IDs:

  La cuestión de las IDs dentro de los modelos relacionales en comparación con los modelos NoSQL puede verse diferenciada. En bases de datos como **MongoDB**, es una buena práctica utilizar funcionalidades como **“newId()”** para generar identificadores de forma automática (como clave primaria). En **Spring Data** se puede recurrir a la utilización de **“ObjectId()”** con el mismo objetivo. Ambas variantes deberán tener su respectiva anotación para indicar que se trata realmente de un identificador (@Id, @BsonId).

  Esto no obliga a que la variable id deba ser de ese tipo, ya que se pueden transformar en otro tipo de datos como pueden ser los String, pero esto tienen sus consecuencias. La utilización de **datos primitivos** en base a tipos de generación de IDs pueden provocar un cambio en el funcionamiento del programa incluso en la integridad de este, llegando a obtener errores por problemas con **IDs duplicadas o no reconocidas.**

- ### Diferentes propuestas de diagrama de clases:

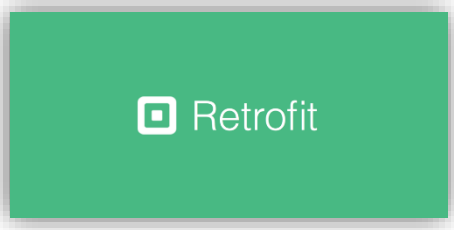
  Explicado en el punto anterior (justificación del diagrama de clases).

- Ktorfit > Retrofit:

  Dados los servicios externalizados conectados a la API que poseemos, es obligatorio decantarse por una tecnología para poder consumir dichas APIs para así extraer sus datos y trabajar con ellas. Dentro de todas las posibilidades, contemplamos dos de las trabajadas en clase, **Ktorfit** y **Retrofit**.

  Ambas opciones hubiesen cumplido de forma correcta la funcionalidad que necesitábamos implementar, pero lo que nos ha hecho decantarnos por Ktorfit es entre otras cosas:

  - Consideramos que al ser una librería dedicada para Kotlin, su implementación se nos haría más **familiar** y, a su vez, el trabajo con esta aún más llevadero.
  - Es una herramienta que hemos trabajado menos en lo que al curso se refiere, lo que nos **invita a conocer más** de la misma para proporcionarnos mayor versatilidad.
  - Ktorfit está, en gran parte, diseñada para poder trabajar con **corrutinas** de una forma más sencilla. Esto nos invita a emplear sus funcionabilidades en nuestro proyecto para mejorar la **eficiencia** de este.



- SqlDeLight:

  A la hora de implementar una caché, una de las tecnologías trabajadas en clase ha sido **SqlDeLIght**, hemos optado por esta opción debido a su fuerte conexión directa con **SQL** y su manejo de datos. Algunos de los puntos más destacables son:

  - Compilación > Ejecución: El código de Java o Kotlin generados por SqlDeLIght hacen que proporcione mayor seguridad antes que una metodología que ejecuta **consultas en tiempo de ejecución**. Siendo así capaz de **detectar errores** en la fase previa al funcionamiento directo del programa.
  - Mejora en el rendimiento de las consultas relacionado con su forma de trabajar (punto anterior).
  - **Claridad** a la hora de trabajar con dicha librería, debido a la posibilidad de tener las **consultas externamente declaradas** (fichero .sq).

  Independientemente de dichos puntos, existen otras alternativas que hemos estudiado como pueden ser librerías como **Cache4K**, entre otras.

Arquitectura del sistema y patrones usados

Para este proyecto, hemos seguido una arquitectura que se rige por una estructura de **Controlador - Repositorios - Servicios**, pero tratando de organizar bien las diferentes funciones dentro de **capas**.

Como se explica en el esquema del punto de requisitos de información, nuestro proyecto tiene su **núcleo en un controlador general**, el cual es el encargado de llamar a todos los métodos del programa, los cuales **encapsulan las llamadas a los repositorios**, cada uno con sus respectivas restricciones.

Controlador

Estructura del controlador:

En todo momento realizaremos la llamada a los mismos métodos, pero enfocado a un modelo distinto, a continuación, se muestra un ejemplo con el modelo Máquina, para que sean visibles sus restricciones y llamada a repositorios:

```
suspend fun listarMaquinas(): Flow<Maquina>? {
    return if (usuarioSesion?.rol == TipoUsuario.ADMIN_JEFE || usuarioSesion?.rol == TipoUsuario.ADMIN_EN
        logger.debug( msg: "Operación realizada con éxito")
        maquinasRepository.findAll()
    } else {
        logger.error( msg: "No está autorizado a realizar esta operación.")
        null
    }
}
```

```
suspend fun encontrarMaquina(id: String): Maquina? {
    return if (usuarioSesion?.rol == TipoUsuario.ADMIN_JEFE || usuarioSesion?.rol == TipoUsuario.ADMIN_EN
        logger.debug( msg: "Operación realizada con éxito")
        maquinasRepository.findById(id)
    } else {
        logger.error( msg: "No está autorizado a realizar esta operación.")
        null
    }
}
```

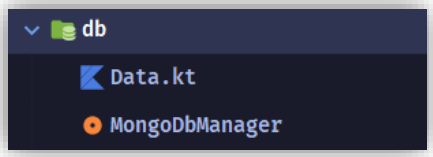
```
suspend fun guardarMaquina(maquina: Maquina) {
    if (usuarioSesion?.rol == TipoUsuario.ADMIN_JEFE || usuarioSesion?.rol == TipoUsuario.ADMIN_ENCARGAI
        maquinasRepository.save(maquina)
        logger.debug( msg: "Operación realizada con éxito")
    } else {
        logger.error( msg: "No está autorizado a realizar esta operación.")
    }
}
```

```
suspend fun actualizarMaquina(maquina: Maquina) {
    if (usuarioSesion?.rol == TipoUsuario.ADMIN_JEFE || usuarioSesion?.rol == TipoUsuario.ADMIN_ENCARGA
        maquinasRepository.update(maquina)
        logger.debug( msg: "Operación realizada con éxito")
    } else {
        logger.error( msg: "No está autorizado a realizar esta operación.")
    }
}
```

```
suspend fun borrarMaquina(maquina: Maquina) {
    if (usuarioSesion?.rol == TipoUsuario.ADMIN_JEFE || usuarioSesion?.rol == TipoUsuario.ADMIN_ENCARGADO
        require( value: listarTurnos()?.filter { it.maquina == maquina }?.count() == 0)
        { "Antes de realizar la operación, elimine o actualice el turno/s asociados a esta máquina. " }
        maquinasRepository.delete(maquina)
        logger.debug( msg: "Operación realizada con éxito")
    } else {
        logger.error( msg: "No está autorizado a realizar esta operación.")
    }
}
```

Base de datos

Por otro lado, tenemos el paquete de base de datos, dónde incluimos el **mánager de Mongo** y un fichero **Data.kt** donde insertamos **datos de prueba:**



Inyección de dependencias

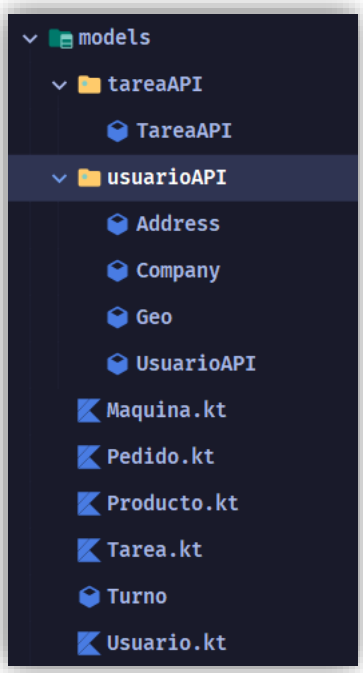
Con relación a la **inyección de dependencias** trabajando con **Koin**, en caso de emplear módulos manuales, es recomendable el uso de un paquete donde meter una clase que albergue todos estos módulos:

Modelos

El paso principal a la hora de trabajar con este tipo de arquitectura es declarar correctamente los modelos con los que vamos a trabajar. El objetivo es **seguir el enunciado** y hacer una extracción de la información necesaria para poder obtener los objetos con todos sus atributos y tipos.

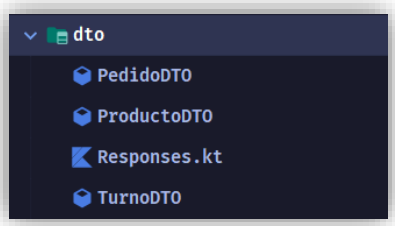
A la hora de trabajar con la API, también será necesario incluir unos modelos que alberguen los datos de esta, ya que en este caso son **distintos a los de nuestro sistema**.

El paquete de modelos se apreciaría tal que así:

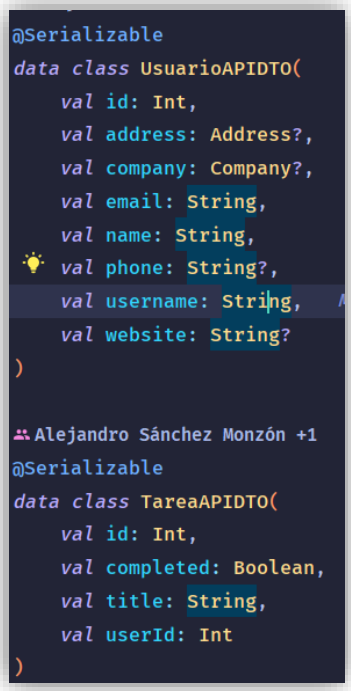


DTOs

Son necesarios también para trabajar con los repositorios y los servicios de la API un conjunto de **objetos de transferencia de datos (DTO)** los cuales estarán también declarados de forma manual en nuestro proyecto:



Los **DTO** individuales son empleados para facilitar la **serialización** a la hora de imprimir los **informes en JSON**, mientras que el fichero **Responses** agrupa todos los DTOs que empleamos **para trabajar con la API**.

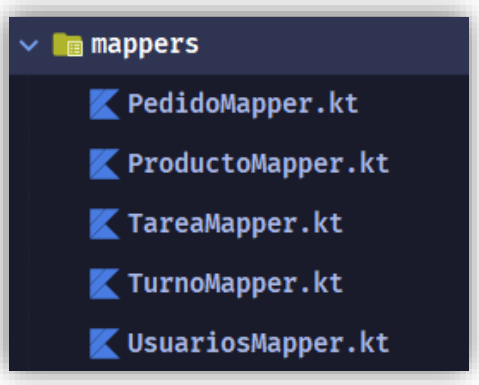


```kotlin
@Serializable
data class UsuarioAPIDTO(
    val id: Int,
    val address: Address?,
    val company: Company?,
    val email: String,
    val name: String,
    val phone: String?,
    val username: String,
    val website: String?
)

Alejandro Sánchez Monzón +1
@Serializable
data class TareaAPIDTO(
    val id: Int,
    val completed: Boolean,
    val title: String,
    val userId: Int
)
```

```kotlin
@Serializable
data class PedidoDTO(    Monzón, 3
    val id: String,
    val uuid: String,
    val tareas: String?,
    val productos: String?,
    val estado: String,
    val usuario: String,
    val fechaTope: String,
    val fechaEntrada: String,
    val fechaProgramada: String,
    val fechaEntrega: String,
    val precio: Float
)
```

Mapeadores

Para poder circular entre **DTOs** y **modelos** de una forma muy cómoda recurrimos al uso de **mappers**, los cuales **transforman** un Objeto, al Objeto deseado, la estructura de estos se vería tal que así:



Dentro encontramos algunas de sus funciones, en el caso de usuarios, tanto para **mapear** a **SQL** como a **DTO** para trabajo con la **API**, y viceversa.

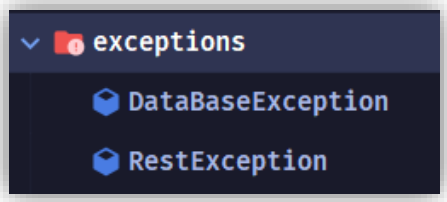
```kotlin
fun UsuarioAPIDTO.toModelUsuario(): Usuario {
    return Usuario(
        id = id.toString(),
        uuid = UUID.randomUUID(),
        nombre = name,
        apellido = username,
        email = email,
        password = cifrarPassword(username),
        rol = randomUserType()
    )
}
```

```kotlin
fun UsuarioModelo.toUsuarioAPIDTO(): UsuarioAPIDTO {
    return UsuarioAPIDTO(
        id = id.toInt(),
        address = null,
        company = null,
        name = nombre,
        username = apellido,
        email = email,
        phone = null,
        website = null
    )
}
```

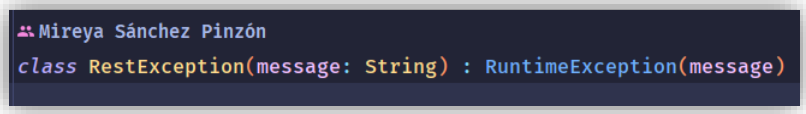
```kotlin
fun UsuarioSQL.toModel(): UsuarioModelo {
    return UsuarioModelo(
        id = id.toString(),
        uuid = UUID.fromString(uuid),
        nombre = nombre,
        apellido = apellido,
        email = email,
        password = password,
        rol = TipoUsuario.valueOf(rol)
    )
}
```

Excepciones

Por otro lado, tenemos el paquete de excepciones, en el cual hemos declarado nuestras **excepciones personalizadas** para trabajar con ellas:

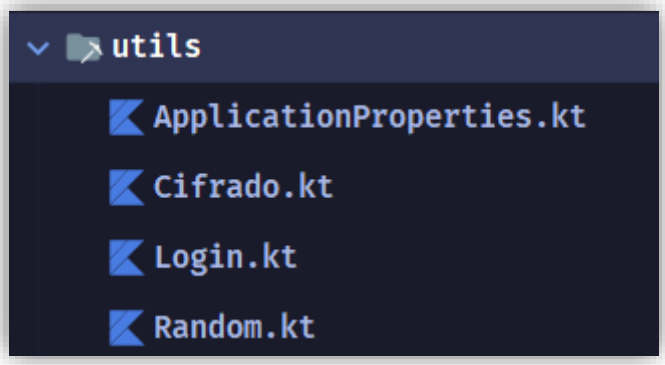


A continuación, un ejemplo de una de ellas:



Utilidades

En nuestra estructura de proyecto, también tenemos un paquete de **utilidades**, donde encontramos una variedad de **funciones** que nos sirven como **herramienta** a la hora de programar nuestro proyecto.



Repositorios

El siguiente paso es implementar los repositorios que albergaran la información y las **operaciones de la base de datos**, para ello trabajaremos empleando **interfaces**, las cuales **dosifican** y **clasifican** los métodos que emplearemos en cada uno de los repositorios.

De base, prácticamente todos los repositorios siguen la interfaz **CRUD**:

```
interface CRUDRepository<T, ID> {
    Mireya Sánchez Pinzón
    suspend fun findAll(): Flow<T>
    Mireya Sánchez Pinzón
    suspend fun findById(id: ID): T?
    Mireya Sánchez Pinzón
    suspend fun save(entity: T): T
    Mireya Sánchez Pinzón
    suspend fun update(entity: T): T
    Mireya Sánchez Pinzón
    suspend fun delete(entity: T): Boolean
}   Pinzón, 29/01/2023 17:32 • Repositorios
```

Para cada uno de los repositorios implementamos dichos métodos, a continuación, mostramos un ejemplo de cómo quedaría cada uno de ellos utilizando como muestra el modelo Usuario:

```
override suspend fun findAll(): Flow<Maquina> {
    logger.debug { "findAll()" }
    return MongoDbManager.database.getCollection<Maquina>().find().asFlow()
}
```

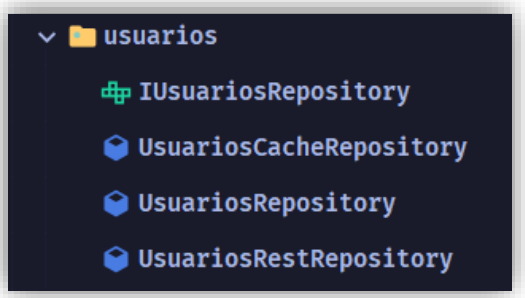
```
override suspend fun findById(id: String): Maquina? {
    logger.debug { "findById($id)" }
    return MongoDbManager.database.getCollection<Maquina>()
        .findOneById(id)
}
```

```
override suspend fun save(entity: Maquina): Maquina {
    logger.debug { "save($entity) - guardando" }
    return MongoDbManager.database.getCollection<Maquina>()
        .save(entity).let { entity }
}
```

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

```kotlin
override suspend fun delete(entity: Maquina): Boolean {
    logger.debug { "delete($entity) - borrando" }
    val encontrado : Maquina?  = findById(entity.id)
    return if (encontrado ≠ null) {
        MongoDbManager.database.getCollection<Maquina>()
            .deleteOneById(entity.id)
        true
    } else {
        logger.debug { "No se ha encontrado la máquina." }
        false
    }
}
```

Para la implementación del repositorio de la **caché** y de la **API REST**, también empleamos un **modelo parecido**, pero siguiendo la implementación necesaria para cada uno de ellos.



Todos los **repositorios** de cada uno de los modelos contienen una **interfaz específica** para poder ser aún más **concretos** en la **implementación de los métodos**.

Servicios

Dentro de los servicios destacamos **tres distintos**, el servicio para poder **imprimir los informes** en archivos **JSON**, el servicio de acceso a la **API REST**, y el servicio para trabajar con **SqlDeLight**:

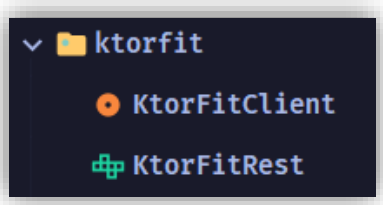
En el **servicio de JSON** tenemos funciones de **escrituras** de ficheros JSON, dado un objeto DTO. A continuación, mostramos un ejemplo de una de ellas:

```
fun writePedido(nombreArchivo: String, entityDTO: List<PedidoDTO>) {
    logger.info( msg: "Escribiendo JSON.")
    val directorio : String  = System.getProperty("user.dir") +
            File.separator + "src" +
            File.separator + "main" +
            File.separator + "resources"
    val fichero = File( pathname: directorio + File.separator + "$nombreArchivo.json")
    val json : Json  = Json { prettyPrint = true }
    fichero.writeText(json.encodeToString(entityDTO))
}
```

En el **servicio de SqlDeLight** tenemos una clase cliente que se ocupa de **conectar** con el **controlador de JDBC** y crear la base de datos que emplearemos para almacenar los datos.

```
class SqlDeLightClient {
    private val driver: SqlDriver = JdbcSqliteDriver(JdbcSqliteDriver.IN_MEMORY)

    Alejandro Sánchez Monzón
    init {
        AppDatabase.Schema.create(driver)
    }

    val queries = AppDatabase(driver).appDatabaseQueries

    Alejandro Sánchez Monzón +1
    fun removeAllData() {
        queries.transaction { this: TransactionWithoutReturn
            logger.debug { "Borrando datos de la cache ... " }
            queries.deleteAllUsuarios()
        }
    }
}
```

Para el **servicio de la API REST**, tendremos un objeto cliente que se ocupa **de conectar con la API de forma directa y construir el servicio de KtorFit**. Por otro lado, tendremos una **interfaz** en la cual indicaremos el **endpoint** en el cual cada una de las operaciones CRUD actuará y el DTO que devolverán.



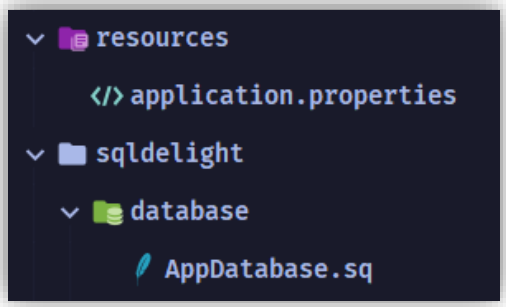
A continuación, mostramos algunos ejemplos del código de KtorFitClient y KtorFitRest:

```
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>()
    }
}
```

```kotlin
@GET("users")
suspend fun getAllUsuarios(): List<UsuarioAPIDTO>

/**
 * Método encargado de acceder a la API REST para encontrar una usuario cuyo
 *
 * @param id, El identidicador, tipo String, del usuario a encontrar.
 *
 * @return UsuarioAPIDTO, el usuario encontrado con los atributos que tiene
 */
Alejandro Sánchez Monzón +1
@GET("users/{id}")
suspend fun getUsuarioById(@Path("id") id: String): UsuarioAPIDTO

/**
 * Método encargado de acceder a la API REST para insertar un usuario dado p
 *
 * @param usuario, El usuario, tipo Usuario, a insertar.
 *
 * @return UsuarioAPIDTO, el usuario insertado con los atributos que tiene
 */
Alejandro Sánchez Monzón
@POST("users")
suspend fun createUsuario(@Body usuario: UsuarioAPIDTO): UsuarioAPIDTO
```

Resources y directorios

Dentro de la carpeta resources tenemos un archivo **.properties** en el cual indicamos diferentes **variables** que usamos para, por ejemplo, la conexión de la base de datos de mongo y el archivo .sq, el cual será la base para poder trabajar con SqlDeLight.



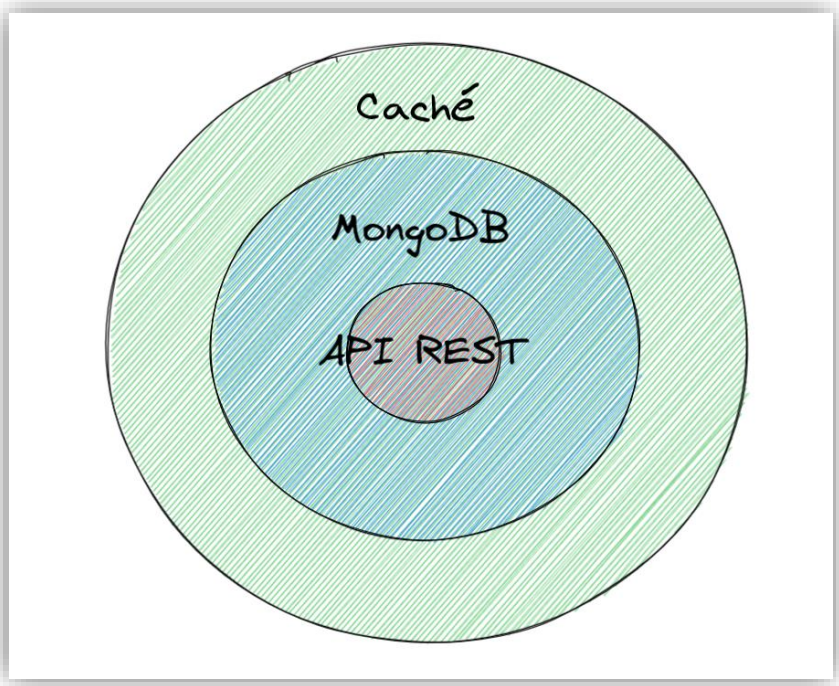

```
MONGO_INITDB_ROOT_USERNAME=mongoadmin
MONGO_INITDB_ROOT_PASSWORD=mongopass
MONGO_INITDB_DATABASE=test
MONGO_BBDD=tenistas
MONGO_OPTIONS=authSource=admin
MONGO_OPTIONS_WINDOWS=retryWrites=true&w=majority
MONGO_HOST=localhost   Monzón, 31/01/2023 20:03 • Mo
```

```
CREATE TABLE Usuario (
    id INTEGER PRIMARY KEY,
    uuid TEXT UNIQUE NOT NULL,
    nombre TEXT NOT NULL,
    apellido TEXT NOT NULL,
    email TEXT NOT NULL,
    password TEXT NOT NULL,
    rol TEXT NOT NULL
);

selectUsuarios:
SELECT * FROM Usuario;

selectUsuarioById:
SELECT * FROM Usuario WHERE id = ?;

createUsuario:
INSERT INTO Usuario(id, uuid, nombre, apellido, email, password, rol) VALUES ( id: ?,  uuid: ?,  nombre: ?,  apellic

updateUsuario:
UPDATE Usuario SET uuid = ?, nombre = ?, apellido = ?, email = ?, password = ?, rol = ? WHERE id = ?;

deleteUsuario:
DELETE FROM Usuario WHERE id = ?;

deleteAllUsuarios:
DELETE FROM Usuario;   Monzón, 29/01/2023 17:27 • Fichero .sq para poder trabajar con SqlDeLight
```

Explicación de forma de acceso a datos

Para trabajar con los datos de este proyecto, el primer paso es entender la **estructura de capas** en la cual organizamos el acceso a los datos.

Mediante el siguiente esquema, mostramos de forma visual cómo es la empleabilidad de la estructura del proyecto, mediante la cual mantenemos una **integridad**, **seguridad** y **eficacia** en relación con los datos:

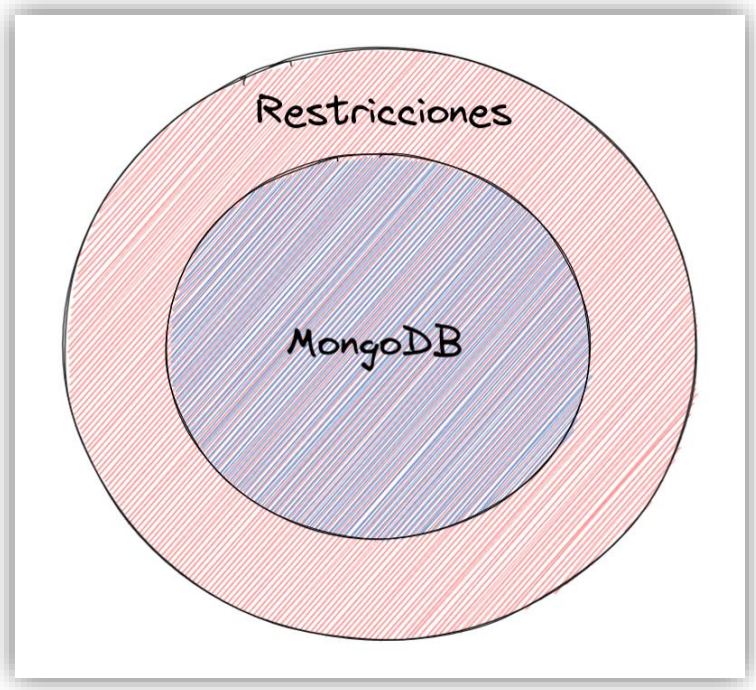


Cómo podemos observar, el **núcleo** general del sistema son esos **externalizados** los cuales, en el caso de necesitar acceder, antes deberán pasar por una serie de capas que **aseguran** las claves anteriormente mencionadas.

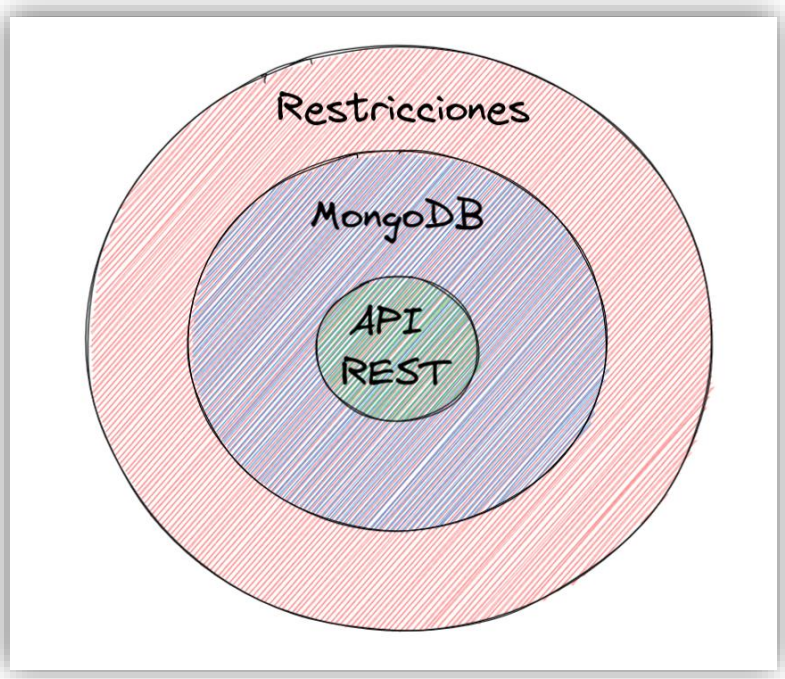
Si, por ejemplo, queremos consultar información sobre un Usuario, primero accederemos a la caché, de una forma rápida y eficaz, si esta acción no da sus frutos, será entonces cuando acudamos a la base de datos. De esta forma, ya estamos reduciendo el consumo de esta. Si a pesar de esto no podemos acceder a la información, será entonces cuando consultemos la API.

Es importante tener presente claves como la **INCONSISTENCIA EVENTUAL**, las consultas cumplen la misión de ser más rápidas y delegan la responsabilidad en el software que las usa.

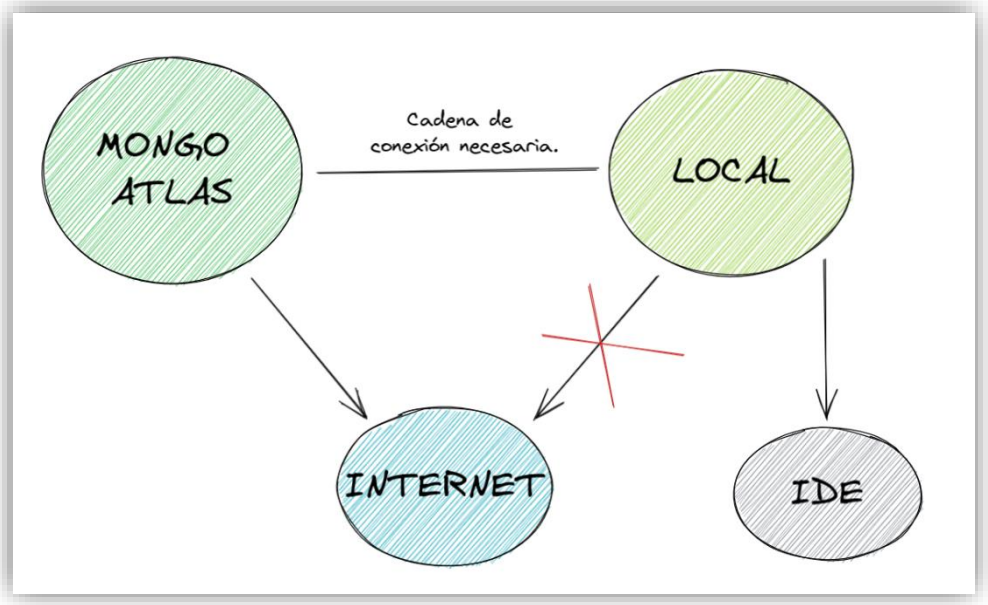
Para modelos de datos los cuales no cuentan con este servicio, su acceso directo será la **base de datos**, pero mediante el uso de **restricciones**:



Si el modelo de datos al que queremos acceder contiene un **servicio externalizado en la nube**, accederemos a este **encapsulando el acceso a la API**:



Variantes de control para empleo de bases de datos en MongoDB:



- Para la conservación de la información, han existido dos variantes para trabajar:
  - **MONGO ATLAS**: es una plataforma en la **nube** que ofrece una instancia **gestionada de MongoDB.**
  - **MONGODB LOCAL:** es una instalación autónoma en un **servidor o equipo local**.

Estas diferencias se diversifican dentro un conjunto de claves tales como la **escalabilidad**, la **ubicación**, el **costo**, la **seguridad** y el **mantenimiento**. Para una práctica de este calibre, a pesar de haber elegido una base de datos en local, la elección es insignificante.

Notas

Test

Se han testeado un total de más de 100 funcionalidades entre el proyecto de MongoDB base y Spring Data. Dentro de estas pruebas se encuentra la confirmación del correcto funcionamiento del controlador y los repositorios necesarios respectivamente.

Para la realización de dichas pruebas se han empleado tecnologías como **JUnit y Mockito.**

A continuación, dejamos unas capturas del repositorio de tareas como ejemplificación de ello:

Lo primero que hemos hecho ha sido crear unos métodos que se ejecutarán al principio y al final de cada uno de los test, de esta forma crearemos un entorno de prueba aislado (sandbox).

En esta primera imagen, preparamos lo necesario para que cada test se pueda limitar a probar una única acción.

Con este método conseguiremos dejar tal y como estaba el sistema antes de haber comenzado un test, esto es necesario para que no se queden los datos de prueba almacenados y que no interfieran en el siguiente test.

A continuación, empezaremos con los tests en sí mismos.

```kotlin
@Test
 fun findAll() = runTest{ this: TestScope
    val res = tareasRepository.findAll()

    assert(res.toList().isEmpty())
}
```

Para testear la búsqueda de todos los elementos de un repositorio lo que hemos hecho ha sido, aprovechar el método setUp(), que nos deja el repositorio a testear completamente vacío de datos. Afirmaremos que cuando el findAll() sea llamado devolverá un flujo, el cual convertido a una lista nos permitirá comprobar que está vacía.

```kotlin
@Test
 fun findById() = runTest{ this: TestScope
    val prueba = tareasRepository.save(tarea)

    val res = tareasRepository.findById(tarea.id)
    println(prueba)
    println(res)
    println(tarea)
    tareasRepository.findAll().toList().forEach { println(it) }
    assert(res == tarea)
}
```

Para testear la búsqueda de un dato en concreto, lo que hemos hecho ha sido guardar un dato en el repositorio, a continuación, buscarlo a través de su identificador y comprobar que, efectivamente lo que ha encontrado sea igual al dato que primeramente salvamos.

```kotlin
@Test
 fun findByIdNoExiste() = runTest{ this: TestScope
    val res = tareasRepository.findById( id: "-5")

    assert(res == null)
}
```

Además, este método, tiene la posibilidad de encontrarse con un fallo, buscar algo que no exista. Es por ello que comprobamos que, si no salvamos nada y aun así realizamos una búsqueda lo que devolverá en este caso será nulo.

```kotlin
@Test
fun save() = runTest{ this: TestScope
    val res = tareasRepository.save(tarea)

    assertAll(
        { assertEquals(res.id, tarea.id) },
        { assertEquals(res.uuid, tarea.uuid) },
        { assertEquals(res.precio, tarea.precio) },
        { assertEquals(res.tipo, tarea.tipo) },
        { assertEquals(res.descripcion, tarea.descripcion) },
        { assertEquals(res.turno, tarea.turno) }
    )
}
```

Para testear un salvado, lo que haremos será hacer un salvado y comprobar que la información que devuelve el método es exactamente la misma que la del dato que salvamos inicialmente.

```kotlin
@Test
fun update() = runTest{ this: TestScope
    tareasRepository.save(tarea)
    val operacion = tareasRepository.update(
        Tarea(
            id = "0",
            uuid = UUID.randomUUID(),
            precio = 100.0f,
            tipo = TipoTarea.ENCORDADO,
            descripcion = "actualizada",
            turno = turno
        )
    )
    val res = tareasRepository.findById(operacion.id)

    assertAll(
        { assertEquals(res?.descripcion, operacion.descripcion) }
    )
}
```

Para testear una actualización, lo que haremos será hacer un salvado de un dato, actualizar su campo “descripcion” y hacer una búsqueda del dato que hemos salvado. Finalmente comprobamos que lo que ha devuelto la búsqueda ha sido el dato inicial con su campo “descripcion” actualizado.

```
@Test
 fun delete() = runTest{ this: TestScope
    tareasRepository.save(tarea)

    val res = tareasRepository.delete(tarea)

    assert(res)
}
```

Para testear un borrado lo que hemos hecho es un salvado y posteriormente un borrado, si el borrado devuelve verdadero es que la operación ha resultado exitosa.

```
@Test
 fun deleteNoExiste() = runTest{ this: TestScope
    val res = tareasRepository.delete(tarea)

    assert(!res)
}
```

Además, este método, también tiene la posibilidad de encontrarse con un fallo, borrar algo que no exista. Es por ello que comprobamos que, si no salvamos nada y aun así realizamos el borrado lo que devolverá en este caso será falso.

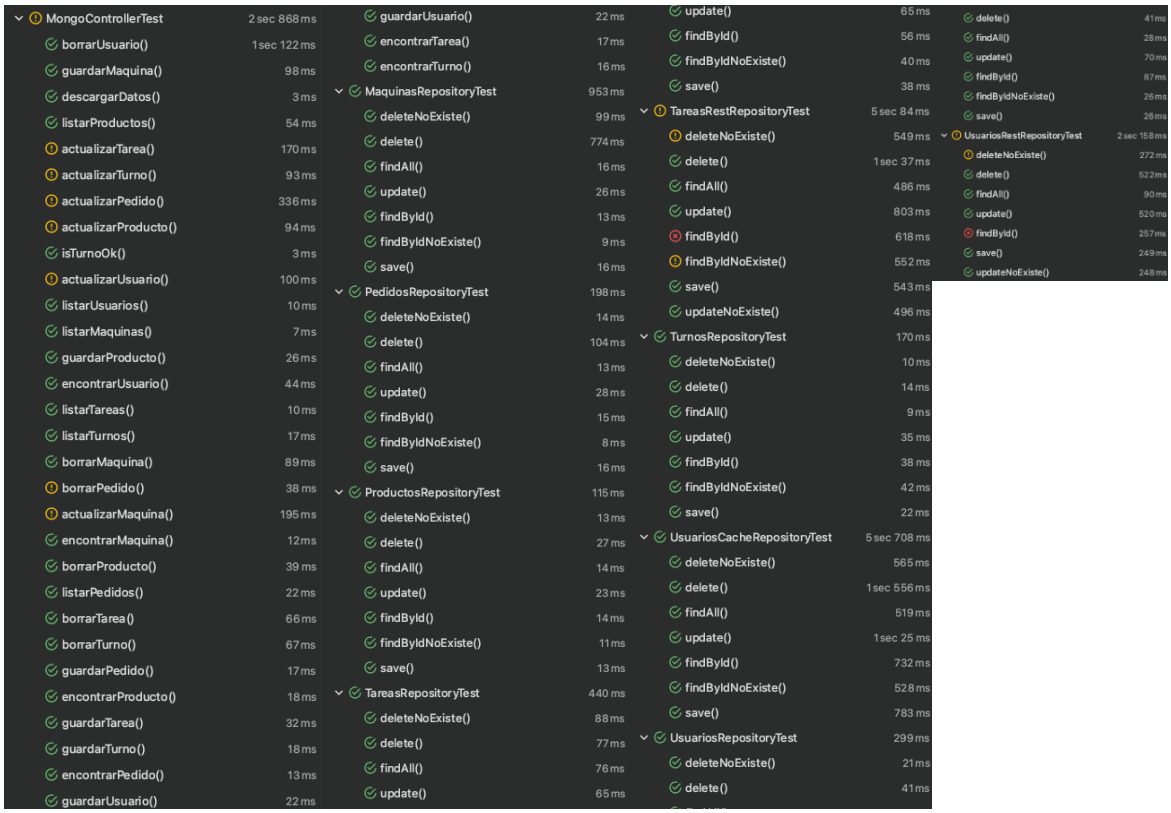
A parte de estos tests, es posible realizar pruebas para casos aún más específicos, como es el caso del controlador:

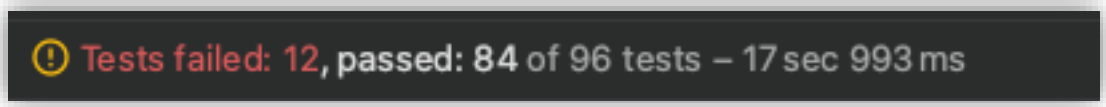
```
@Test
fun borrarMaquina() = runTest { this: TestScope
    coEvery { maquinasRepository.delete(maquina) }
    coEvery { maquinasRepository.save(maquina) } returns maquina
    coEvery { maquinasRepository.findAll() } returns flowOf(maquina)
    coEvery { turnosRepository.findAll() } returns flowOf(turno)

    usuarioSesion = usuarioDeSesion
    controller.guardarMaquina(maquina)
    val res = assertThrows<IllegalArgumentException> {
        controller.borrarMaquina(maquina)
    }

    assertEquals( expected: "Antes de realizar la operación, elimine o actualice el turno/s asociados a esta máquina. ", res.message)
}
```

Como se puede ver en este ejemplo, hemos decidido testear el caso en el que el borrado de una máquina no se ha podido realizar porque en ese momento tenga un turno asociado. También se podrían probar casos más específicos como un borrado exitoso, un borrado fallido porque el usuario de sesión no tuviese los permisos, o que al borrar la máquina esta no existiese.

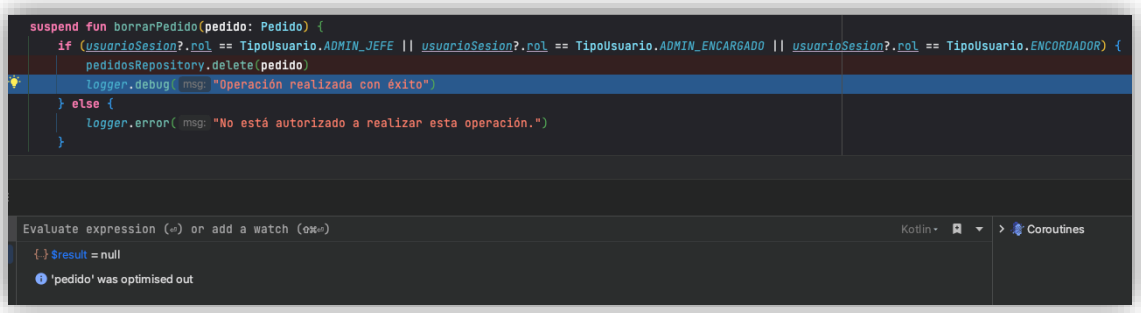
Sin embargo, en esta práctica hemos querido centrarnos en los más básicos.

Finalmente, estos son los resultados:





Como se puede apreciar, 12/96 tests han dado error, es algo que investigamos exhaustivamente, debido a que todos los tests están implementados bajo la misma idea para todos los repositorios/controlador. Pudiendo observar que algunas pruebas hechas exactamente bajo la misma metodología funcionaban y otros no, decidimos debuggearlos y nos dimos cuenta de que el test optimizaba la variable que necesitaba y hacía que el método no pudiese acceder a ella. Creemos que esto es lo que está provocando los fallos.



Consultas para JSON

Es importante recordar que es más **práctico** y **limpio, trabajar** y **depurar estructuras de JSON completas** antes que **realizar varias consultas** y accesos a la base de datos.

Conclusiones

En esta práctica hemos trabajado una nueva forma de crear sistemas de datos empleando modelos **NoSQL** como **MongoDB**, tanto de una forma básica como con un framework como **Spring Data**.

A su vez, hemos trabajado con una librería de consumo **de API REST** como es **Ktorfit**, y una tecnología como **SqlDeLight** para la implementación de la **caché**.

Se ha trabajado con ficheros .**properties** para una mejor organización de las diferentes variables y se ha seguido la **metodología de trabajo** basada en **GitFlow**.