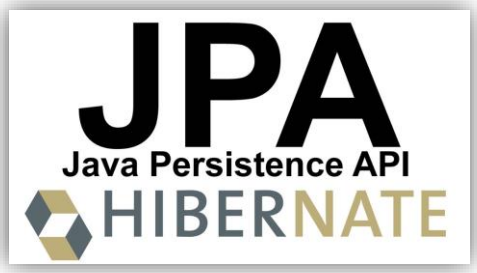
AD-P02-TennisLab
Memoria del proyecto

Diseño y propuesta de solución.

Para este proyecto se nos requería del uso de tecnologías como **Exposed** e **Hibernate JPA.**



Estas tecnologías/ librerías son herramientas que nos ofrecen la posibilidad de seguir un flujo de información más ordenado y eficiente dentro de la base de datos.

Mediante el uso de anotaciones, clases “DAO” e interfaces de repositorios basados en los famosos CRUD. Las tecnologías escogidas nos ayudan a gestionar las entidades y relaciones entre estas dentro de la base de datos.

Comúnmente estábamos acostumbrados a utilizar herramientas más “manuales” como puede ser el uso de bases de datos basadas en **JDBC,** donde uno era el encargado de crear (mediante scripts SQL o uso de managers) cada uno de los elementos de la base de datos, y su posterior utilización, algo que puede llegar a ser más tedioso y difícil de manejar.

Hemos basado nuestra práctica en dos proyectos, uno en el cual hemos utilizado las librerías necesarias para la implementación y testeo de **Exposed**, y otras donde hemos utilizado las dedicadas para **JPA** en **Hibernate**. En ambos casos hemos tenido como base las bases de dato de **H2,** que son las que hemos trabajado en clase.

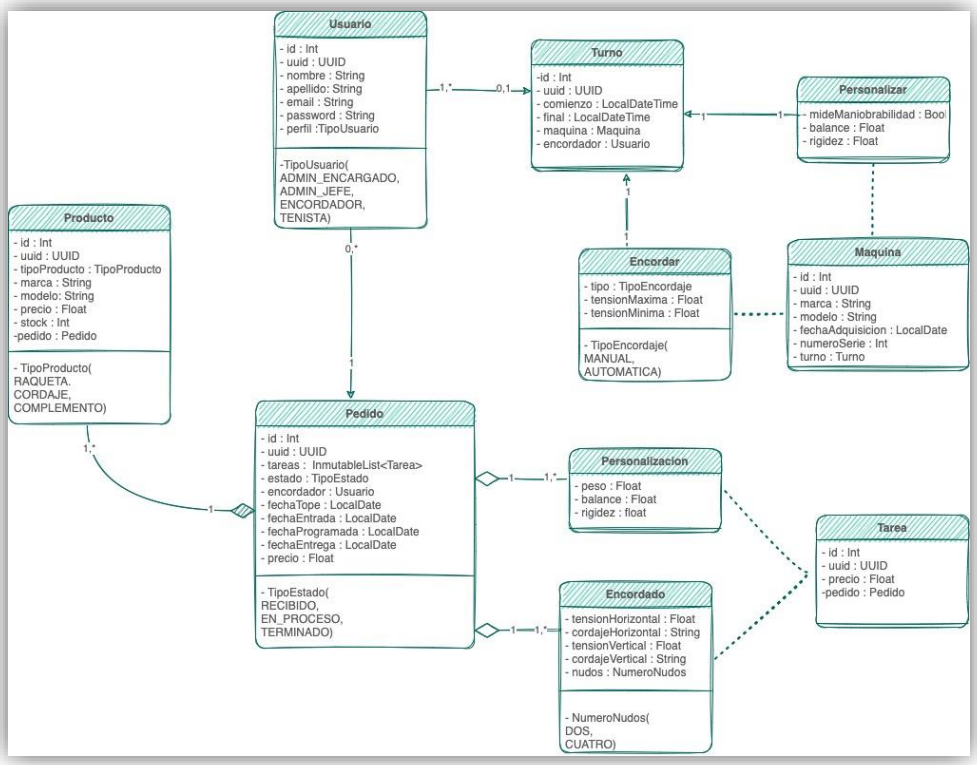
Para **Exposed** ha sido necesario el uso de entidades, las cuales trabajaban con tecnología DAO. Estas entidades se ocupan de "traducir" la información de los modelos a tablas y clases que **Exposed** utiliza para el seguimiento de la información y gestión de las relaciones. A la hora de trabajar con DTOs para la salida de dicha información por JSON, todo funciona al igual que siempre, pero, en casos como el uso de repositorios y controladores, será con estas nuevas entidades con las que trabajáremos.

En el caso de **JPA**, la cosa no es tan compleja. Si utilizamos herramientas como **Hibernate**, bastará con seguir nuestro diagrama entidad-relación o diagrama de clases para saber que anotaciones utilizar y en qué momento. Estas anotaciones complementarán a los modelos, y a la hora de realizar las operaciones CRUD de la base de datos, simplemente deberemos tener un manager que efectúe el uso de dichas operaciones.

Cambiar de **JPA** a **Exposed** y viceversa es algo muy sencillo, siempre y cuando se haya realizado correctamente uno de los dos y el diagrama de clases tenga coherencia. En caso de los controladores, es prácticamente idéntico, comparten DTOs y utilidades. Las variables de los modelos son las mismas, solo que las anotaciones de **JPA** sustituyen a las entidades de **Exposed**. Los repositorios sí son algo más distintos, su base es idéntica (interfaz CRUD) pero las operaciones a realizar dentro difieren un poco dependiendo del manager que utilicemos.

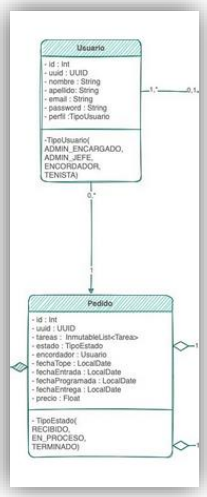
A continuación, vamos a proceder a explicar detenidamente nuestra interpretación del enunciado plasmado en el diagrama de clases.

Diagrama de clases y justificación de este.

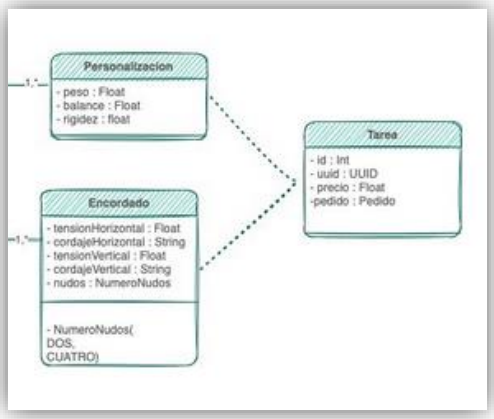


Nuestro sistema consiste en la existencia de un **Usuario**, este usuario tiene diferentes datos como nombre, email, password, y un tipo. Este tipo es importante, el usuario puede ser un trabajador (de tipo encargado, jefe o encordador) o, por el contrario, un cliente (tipo tenista).

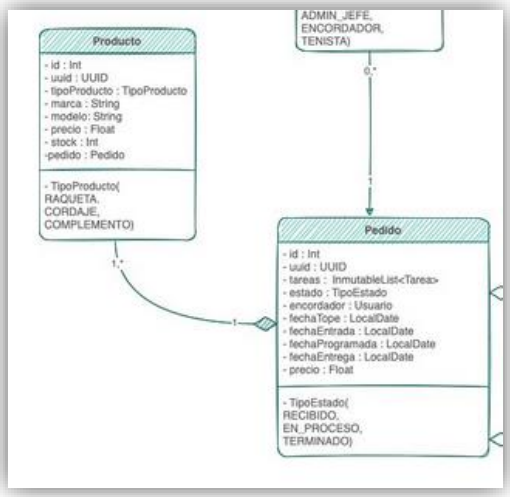
El **usuario** de tipo **encordador** puede estar asociado a 0 (ninguno) o muchos (más de uno, pero por restricciones, solo 2) **pedidos**, que son en los cuales él está trabajando. Por otro lado, un **pedido** podrá tener asignado a sí mismo únicamente un **encordador**.



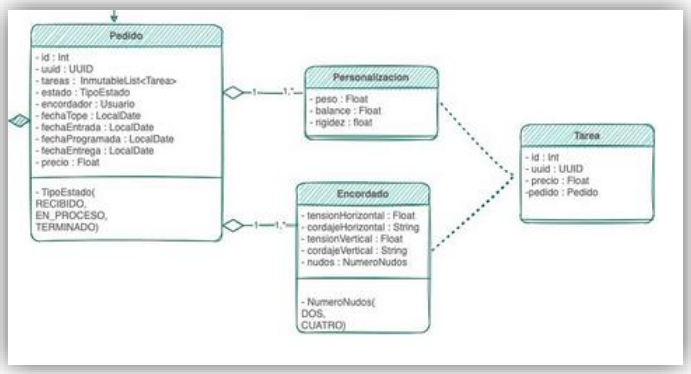
Estos **pedidos** son una composición de **productos** (ya sea raquetas, cordajes, complementos) y diferentes **tareas** (pueden ser tareas de **encordado** o tareas de **personalización**). Estas tareas comparten entre ellas algunos datos como el precio, el pedido al que van asignadas, el identificador y el identificador alfanumérico. Es por este motivo que hemos decidido crear una **interfaz** que tipifica a tareas de encordado y tareas de personalización en una sola: las **tareas**.



Por parte de los **productos**, estos solo podrán estar en un **pedido**. Inicialmente diseñamos el sistema para que los productos fuesen un almacén de todo el "**inventario**" que se gestiona en la tienda, **pero** finalmente, creyendo que, para este caso, describía mejor el uso de la información, hemos diseñado una relación en la cual los **productos** (todos) tendrán un identificador asociado que irá ligado (será el mismo) al **pedido** en el que están contenidos. Por el otro lado, en un **pedido** podrá haber siempre muchos **productos**, o simplemente uno, mientras que un **producto** solo podrá estar asignado a un **pedido**. Y no hablamos del tipo de **producto**, sino esa **unidad**.

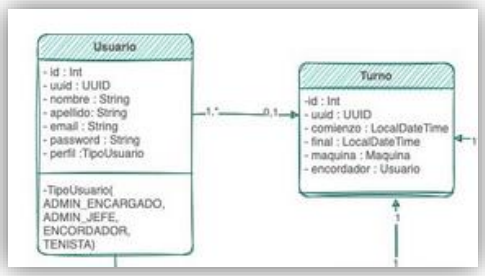


Para el caso de las **tareas**, tanto de encordado como de personalización, la relación es idéntica. Una tarea solo podrá estar contenida en un único **pedido**, ya que hablamos de esa **tarea** en concreto, y no del tipo de **tarea** (recordemos que esta tiene un identificador que la hace única). Por el otro lado, los **pedidos** podrán tener varias **tareas**, ya sean el mismo tipo o de tipos distintos, pero siempre podrán tener varias, o al menos una.



Por el otro lado del **diagrama**, tenemos una relación entre **turnos** y **usuarios**, concretamente los **usuarios** que tienen un tipo relacionado con ser **trabajadores** de la empresa.

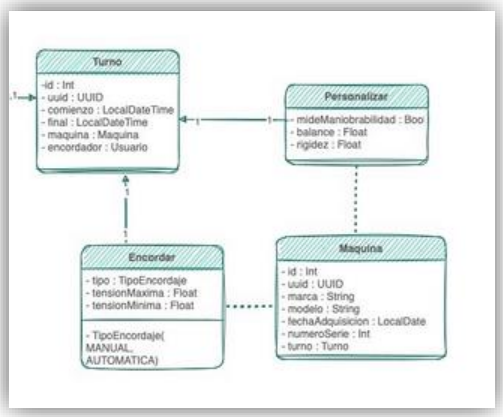
Los usuarios trabajarán siempre en un **único turno**, o en ninguno, mientras que los **turnos** siempre tendrán asignado al menos a un **usuario**. Esta interpretación se debe a que **no** hemos entendido turno como un turno de trabajo (tarde o noche).



Para nosotros, un **turno** dentro de este sistema hace referencia al periodo de tiempo en el cual el **trabajador** está realizando su tarea.

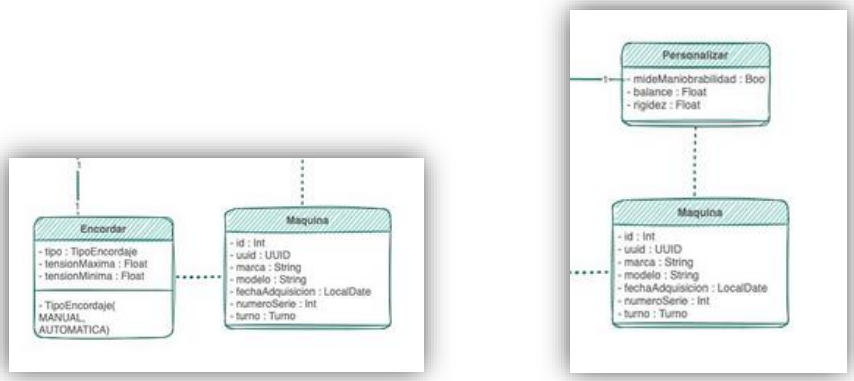
Por ejemplo: *en un McDonald, una persona calienta la carne y otra monta la hamburguesa. Ambos están en un mismo turno de trabajo (tarde, noche...) pero están realizando acciones distintas dentro de su trabajo.*

Es en base a esta idea que entendemos entonces que cada **turno** tendrá asociado a sí mismo una **máquina**, con la cual se realizan las acciones. Estas **máquinas** tendrán una relación de a uno con **turnos**. Es decir, una misma **máquina** solo podrá ser empleada en un **turno** de forma simultánea, ya que esta no puede ser usada a la misma vez en **turnos** distintos (siendo la misma **máquina**).



Como pasaba con **tareas**, las maquinas pueden ser de **encordado**, o de **personalización**. Pero, ambas tenían variables compartidas como pueden ser el identificador, el identificador alfanumérico, el modelo, el número de serie...

Es por esto que hemos decidido también crear una **interfaz** llamada "**Maquina**" que tipifica a las otras dos y las otorga un contexto.



Es en base a esta interpretación del sistema, que hemos procedido a montar nuestro proyecto.

Requisitos de Información.

Para completar los sistemas de información y la buena estructuración del diagrama del proyecto, ha sido necesaria realizar un análisis y estudio de la información requerida, y, a su vez, completar la misma con las fuentes disponibles.

Por una parte, se nos proporcionaron diferentes enlaces a páginas de información y compraventa de equipamiento deportivo dentro del entorno del tenis tales como:

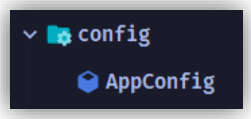
- ***https://www.tenniswarehouse-europe.com/lc/RacquetStringTerms.html?lang=es***
- ***https://www.tenniswarehouse-europe.com/?lang=es***
- ***https://www.tennis-point.es/***

Por otra parte, tras leer el enunciado, nos dimos cuenta de que había datos que nos parecían importantes para poder continuar el desarrollo de nuestro sistema, es por esto que, fue necesario realizar entrevistas con el cliente para conocer sus necesidades (muchas veces gracias a la información dada en clase) y otras veces vía "Discord" gracias a las preguntas de otros compañeros.

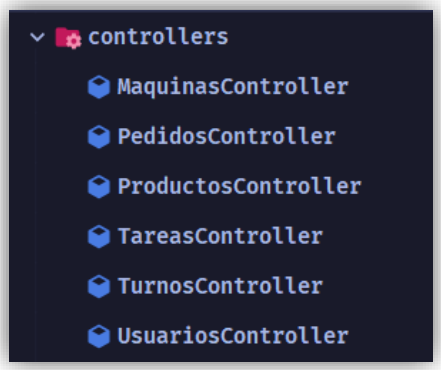
Arquitectura del sistema y patrones usados.

Para ambos proyectos hemos seguido una **arquitectura** por **capas**, en las que los **repositorios**, los cuales son los que tratan de forma directa con los datos, nos sean llamados de forma directa en la función principal, sino que utilicen unos **controladores** asignados los cuales, mediante **inyección** de **dependencias**, hacen el llamado a dichos métodos. Esto sirve para poner una capa más de "seguridad" a la estructura del proyecto, y **protegiendo** aún más los **datos**.

Exposed



El paquete de "**config**" hace referencia a todas las clases y archivos Kotlin que sirven para la propia configuración de la base de datos a la hora de trabajar con ella dentro del proyecto. En el interior de la clase la cual observamos tenemos diferentes valores de configuración, los cuales serán llamados al método principal para la ejecución del proyecto.

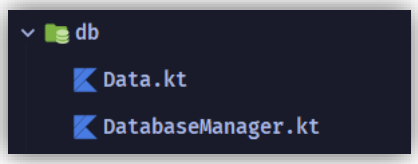


El paquete de “**controllers**" alberga los controladores de todos los repositorios del programa, en el interior de cada uno de ellos, tenemos una referencia al repositorio, esta es utilizada para crear funciones las cuales retornan el resultado de cada una de las operaciones del repositorio:

```
Mireya Sánchez Pinzón
class MaquinasController(
    private val maquinasEncordarRepository: MaquinasEncordarRepository = MaquinasEncordarRepository(
        MaquinasEncordarDAO,
        TurnosDAO
    ),
    private val maquinasPersonalizarRepository: MaquinasPersonalizarRepository = MaquinasPersonalizarRepository(
        MaquinasPersonalizarDAO,
        TurnosDAO
    )

){

    Mireya Sánchez Pinzón
    fun getMaquinasEncordar(): List<MaquinaEncordar> {
        return maquinasEncordarRepository.findAll()
    }
```



El paquete de “**db**” hace referencia a todo lo que tiene que ver con los datos y operaciones de datos. Dentro del fichero Kotlin de “Data” generamos datos al azar de todos los tipos de modelo para poner en marcha nuestra aplicación. Por otor lado, en el fichero “DatabaseManager” llamamos a las operaciones de la base de datos tanto como para su manipulación como para su creación y conexión.

El paquete de "**dto**" contiene todos los ficheros y clases de Kotlin los cuales cumplen dos funciones.

Por un lado, la declaración de data clases las cuales definen las variables de los modelos a tipos de datos simples tales como Int, String, etc... Por otro lado, también realizan la función de mapeo de dichos modelos a su DTO, realizando la transformación de dichos datos:

```kotlin
@Serializable
@SerialName("Pedido")
data class PedidoDTO(
    val id: Int,
    val uuid: String,
    val estado: String,
    val encordador: String,
    val fechaTope: String,
    val fechaEntrada: String,
    val fechaProgramada: String,
    val fechaEntrega: String,
    val precio: Float
)

Mireya Sánchez Pinzón
fun Pedido.toDTO(): PedidoDTO {
    return PedidoDTO(
        id = id,
        uuid = uuid.toString(),
        estado = estado.toString(),
        encordador = encordador.toString(),
        fechaTope = fechaTope.toString(),
        fechaEntrega = fechaEntrega.toString(),
        fechaProgramada = fechaProgramada.toString(),
        fechaEntrada = fechaEntrada.toString(),
        precio = precio
```

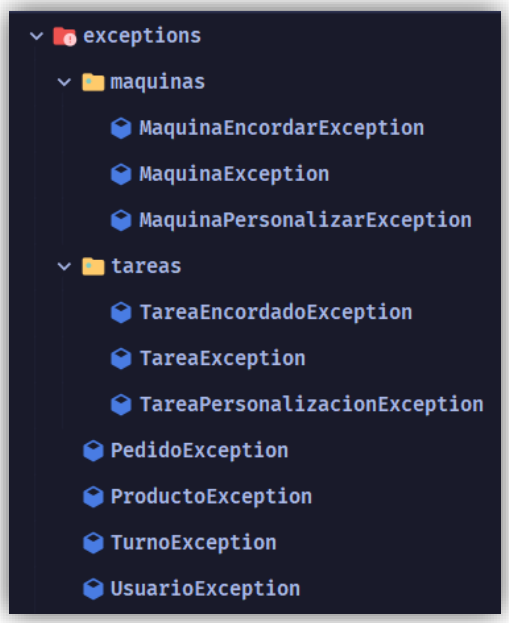
Como estos DTO luego serán las clases que serializaremos para crear los ficheros JSON, les añadimos la anotación de @Serializable.

El paquete de "**entities**" contiene las entidades, DAOs y tablas requeridas para la utilización de **Exposed** en base a los modelos originales. En el contenido de cada uno de los ficheros JSON viene declarada la tabla a la que hacen referencia, con las declaraciones de sus columnas, la declaración de sus relaciones entre entidades, y la creación de los DAO:

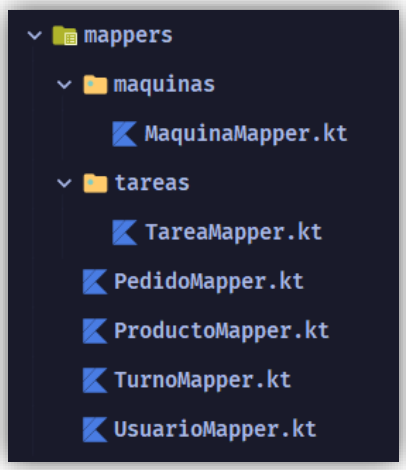
```
object UsuariosTable : IntIdTable( name: "USUARIOS") {
    val uuid = uuid( name: "uuid").uniqueIndex()
    val nombre = varchar( name: "nombre",  length: 50)
    val apellido = varchar( name: "apellido",  length: 50)
    val email = varchar( name: "email",  length: 50)
    val contrasena = varchar( name: "contrasena",  length: 255)
    val perfil = enumeration<TipoUsuario>( name: "perfil")
}

Mireya Sánchez Pinzón +1
class UsuariosDAO(id: EntityID<Int>) : IntEntity(id) {
    Alejandro Sánchez Monzón
    companion object : IntEntityClass<UsuariosDAO>(UsuariosTable)

    var uuid by UsuariosTable.uuid
    var nombre by UsuariosTable.nombre
    var apellido by UsuariosTable.apellido
    var email by UsuariosTable.email
    var contrasena by UsuariosTable.contrasena
    var perfil by UsuariosTable.perfil
}
```



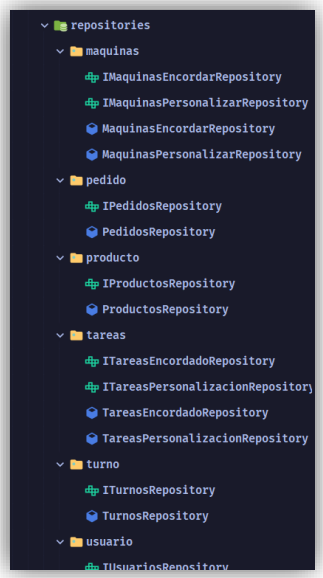
En el paquete de "**exceptions**" vienen creadas todas las clases que contienen las excepciones personalizadas utilizadas para gestionar los errores del programa con un mensaje personalizado.



El paquete "**mappers**" transforma los objetos declarados como DAOs a modelos normales para trabajar con ellos fuera del entorno de la base de datos. Su empelo es parecido a los mappers de DTO a objetos de tipos complejos, reasignado el valor uno a una de las variables de los mismos:

```kotlin
Mireya Sánchez Pinzón +1
fun UsuariosDAO.fromUsuarioDAOToUsuario(): Usuario {
    return Usuario(
        id = id.value,
        uuid = uuid,
        nombre = nombre,
        apellido = apellido,
        email = email,
        contrasena = contrasena,
        perfil = perfil,
    )
}
```

En el paquete “**models**” encontramos las clases que hacen referencia a la base de la información del proyecto. Aquí encontramos la declaración de las variables con las que luego trabajaremos, su tipificación y su representación a nivel programática del diagrama de clases.



El paquete de "**repositories**" es el que organiza los repositorios de cada una de las entidades de nuestro programa, en el interior de cada una de las carpetas encontramos una interfaz, mediante la cual, en base al patrón de Segregación de Interfaces, asignamos una interfaz que recibe los métodos de la interfaz CRUD general y que se implementa en cada uno de los repositorios para mayor maniobrabilidad.

Dentro de estas clases encontramos siempre los mismos métodos, pero trabajan de forma distinta enfocadas cada una en la entidad que gestionan.

Hibernate JPA

A la hora de trabajar con **JPA**, como hemos indicado anteriormente, la arquitectura es realmente parecida, pero se deben tener en cuenta dos cuestiones importantes.

La primera es que las anotaciones necesarias para JPA irán ligadas a los modelos, y no será necesario el uso de unas "entidades" como en **Exposed**.

Lo segundo es que los repositorios, a pesar de ser organizados igual, no necesitan pasar por parámetros ningún DAO, ya que aquí no usamos los DAO de forma explícita. Es por esto que tampoco necesitamos mappers de DAO a modelos ni viceversa.

Ejemplo de modelos con anotaciones:

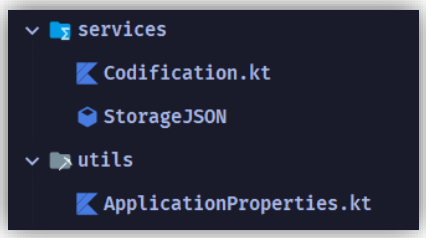
```kotlin
Alejandro Sánchez Monzón +1
@Entity
@Table(name = "TURNOS")
@NamedQuery(name = "Turno.findAll", query = "SELECT t FROM Turno t")
data class Turno(
    @Id
    val id: Int,

    @Column(name = "uuid")
    @Type(type = "uuid-char")
    val uuid: UUID,

    @Type(type = "org.hibernate.type.LocalDateTimeType")
    val comienzo: LocalDateTime,

    @Type(type = "org.hibernate.type.LocalDateTimeType")
    val final: LocalDateTime,

    @OneToOne
    @JoinColumn(name = "usuario_id", referencedColumnName = "id", nullable = false)
    val encordador: Usuario
)
```



Por último, en relación a la arquitectura, destacar algunos paquetes más, “**utils**” y “**services**”.

Ambos paquetes son necesarios para el buen funcionamiento el programa, dese gestionar los métodos que imprimirán en JSON los datos, hasta codificar contraseñas, etc...

Explicación de forma de acceso a los datos.

Para este proyecto usamos bases de datos de tipo H2. Empleamos el uso de bases en local, las cuales se crean y se tiran cada vez que ejecutamos el programa.

Dependiendo del uso de librerías y tecnologías para la realización de la práctica, el acceso a los datos es distinto es por ello que vamos a explicar cada uno por separado utilizando un repositorio de ejemplo.

Exposed

Utilizamos la clase “**DatabaseManager**” para llamar a las operaciones de la base de datos.

Todos los métodos dentro de este entorno serán iguales a transacciones.

Para llamar a todas las entidades de un tipo utilizamos los métodos .**all**() y .**map**{} donde cada valor es mapeado de su DAO al modelo original.

```
usuariosDAO.all().map { it.fromUsuarioDAOToUsuario() }
```

Para llamar a una entidad en concreto utilizamos el método directo .**findById**(id).

```
usuariosDAO.findById(id)
    ?.fromUsuarioDAOToUsuario()
```

Para insertar y actualizar entidades, controlando si esta existe o no, simplemente reasignamos los valores o creamos un DAO nuevo y lo mapeamos a su modelo original.

```
private fun insert(entity: Usuario): Usuario {
    logger.debug { "save($entity) - creando" }
    return usuariosDAO.new(entity.id) { this: UsuariosDAO
        uuid = entity.uuid
        nombre = entity.nombre
        apellido = entity.apellido
        email = entity.email
        contrasena = passwordCodification(entity.contrasena)
        perfil = entity.perfil
    }.fromUsuarioDAOToUsuario()
}

Mireya Sánchez Pinzón +1
private fun update(entity: Usuario, existe: UsuariosDAO): Usuario {
    logger.debug { "save($entity) - actualizando" }
    return existe.apply { this: UsuariosDAO
        uuid = entity.uuid
        nombre = entity.nombre
        apellido = entity.apellido
        email = entity.email
        contrasena = entity.contrasena
        perfil = entity.perfil
    }.fromUsuarioDAOToUsuario()
}
```

Para eliminar entidades llamamos al método directo .**delete**().

```
existe.delete()
```

Hibernate JPA

Es importante declarar de forma correcta la unidad de persistencia, esta es la base principal de nuestro trabajo con JPA.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
    <persistence-unit name="default">
        <description>TennisLab</description>
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <class>models.Usuario</class>
        <class>models.Turno</class>
        <class>models.Pedido</class>
        <class>models.Producto</class>
        <class>models.tareas.TareaEncordado</class>
        <class>models.tareas.TareaPersonalizacion</class>
        <class>models.maquinas.MaquinaEncordar</class>
        <class>models.maquinas.MaquinaPersonalizar</class>

        <properties>
            <property name="hibernate.connection.url" value="jdbc:h2:mem:tennislab;DB_CLOSE_DELAY=-1;"/>
            <property name="hibernate.connection.driver_class" value="org.h2.Driver"/>
            <property name="hibernate.connection.user" value="sa"/>
            <property name="hibernate.connection.password" value=""/>

            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
        </properties>
    </persistence-unit>
</persistence>
```

Utilizamos la clase “**HibernateManager**” para llamar a las operaciones de la base de datos.

El método **createnamedQuery**() dentro de .**query**{} ejecuta una sentencia anteriormente declarada que se puede almacenar en una variable para registrar los resultados.

```
HibernateManager.query {
    val query: TypedQuery<Usuario> = manager.createNamedQuery( name: "Usuario.findAll", Usuario::class.java)
    usuarios = query.resultList
}
```

El método **find**() dentro de .**query**{} permite acceder a la variable “id” (que pasamos por parámetros) de ese modelo y así acceder a los datos como resultado.

```
HibernateManager.query {
    usuario = manager.find(Usuario::class.java, id)
}
```

El método .**merge**() dentro de .**transaction**{} se ocupa de insertar la entidad que nosotros pasamos por parámetros o, de lo contrario, si esta ya existe, actualizarla con sus datos.

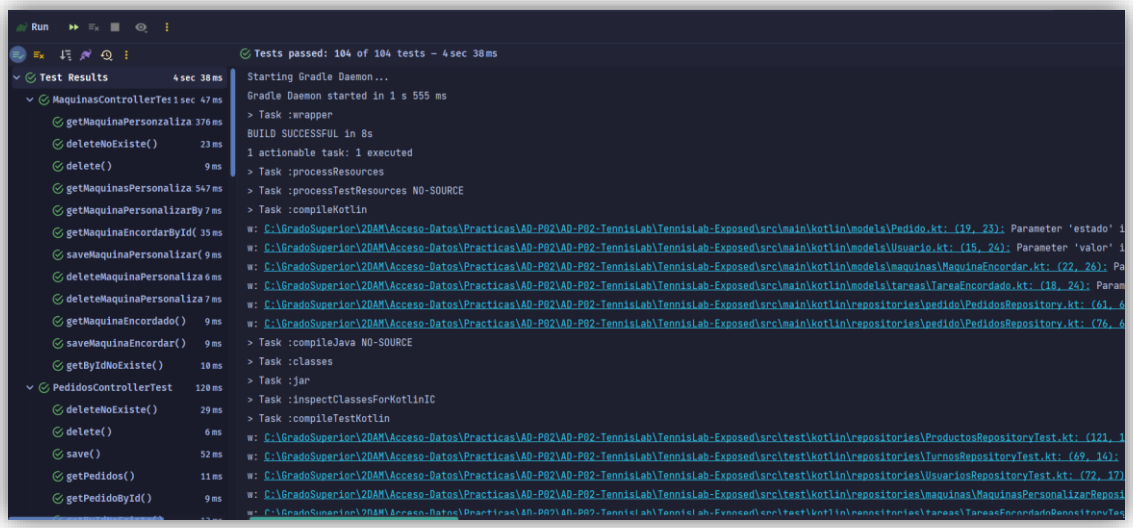
```
HibernateManager.transaction {
    manager.merge(entity)
}
```

El método .**remove**(), con ayuda del .**find**(), dentro de .**transaction**{} nos permite eliminar una entidad específica de la base de datos.

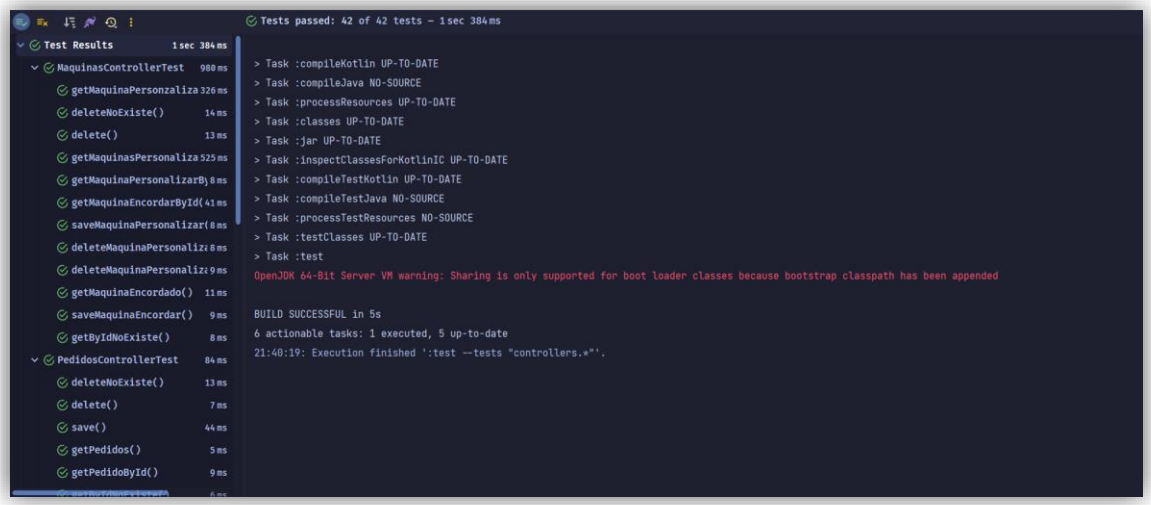
```kotlin
HibernateManager.transaction {
    val usuario : Usuario!  = manager.find(Usuario::class.java, entity.id)
    usuario?.let { it: Usuario
        manager.remove(it)
        result = true
    }
}
```

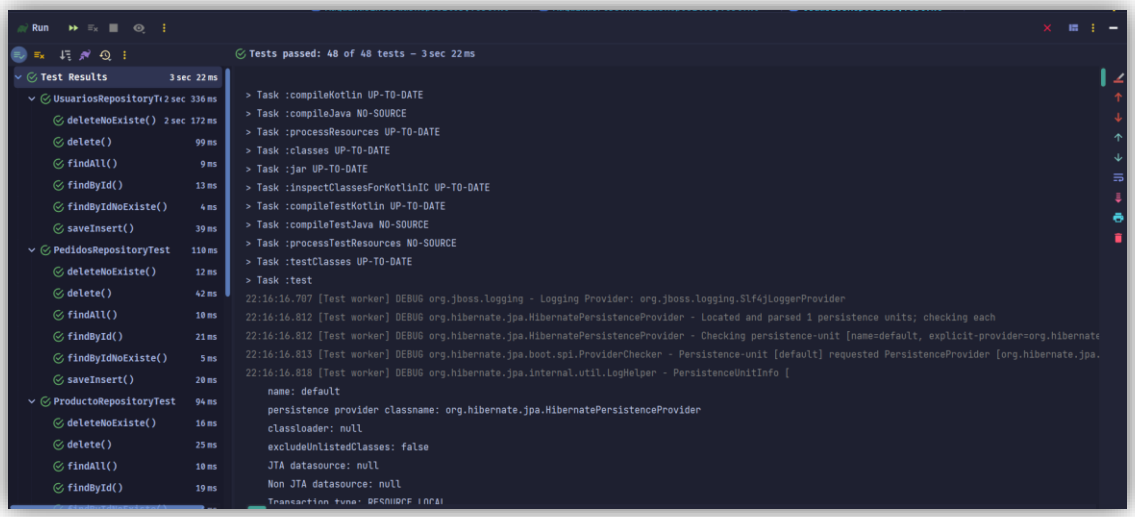
Tests

Exposed



Hibernate JPA


Tests passed: 42 of 42 tests – 1 sec 384 ms
Test Results 1 sec 384 ms
MaquinasControllerTest 980 ms
getMaquinaPersonaliza 326 ms
deleteNoExiste() 14 ms
delete() 13 ms
getMaquinasPersonaliza 525 ms
getMaquinaPersonalizarBy 8 ms
getMaquinaEncordarById(41 ms
saveMaquinaPersonalizar(8 ms
deleteMaquinaPersonaliz 8 ms
deleteMaquinaPersonaliz 9 ms
getMaquinaEncordado() 11 ms
saveMaquinaEncordar() 9 ms
getByIdNoExiste() 8 ms
PedidosControllerTest 84 ms
deleteNoExiste() 13 ms
delete() 7 ms
save() 44 ms
getPedidos() 5 ms
getPedidoById() 9 ms
> Task :compileKotlin UP-TO-DATE
> Task :compileJava NO-SOURCE
> Task :processResources UP-TO-DATE
> Task :classes UP-TO-DATE
> Task :jar UP-TO-DATE
> Task :inspectClassesForKotlinIC UP-TO-DATE
> Task :compileTestKotlin UP-TO-DATE
> Task :compileTestJava NO-SOURCE
> Task :processTestResources NO-SOURCE
> Task :testClasses UP-TO-DATE
> Task :test
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
BUILD SUCCESSFUL in 5s
6 actionable tasks: 1 executed, 5 up-to-date
21:40:19: Execution finished ':test --tests "controllers.*"'.


Run
Tests passed: 48 of 48 tests – 3 sec 22 ms
Test Results 3 sec 22 ms
UsuariosRepositoryTe 2 sec 336 ms
deleteNoExiste() 2 sec 172 ms
delete() 99 ms
findAll() 9 ms
findById() 13 ms
findByIdNoExiste() 4 ms
saveInsert() 39 ms
PedidosRepositoryTest 110 ms
deleteNoExiste() 12 ms
delete() 42 ms
findAll() 10 ms
findById() 21 ms
findByIdNoExiste() 5 ms
saveInsert() 20 ms
ProductoRepositoryTest 94 ms
deleteNoExiste() 16 ms
delete() 25 ms
findAll() 10 ms
findById() 19 ms
> Task :compileKotlin UP-TO-DATE
> Task :compileJava NO-SOURCE
> Task :processResources UP-TO-DATE
> Task :classes UP-TO-DATE
> Task :jar UP-TO-DATE
> Task :inspectClassesForKotlinIC UP-TO-DATE
> Task :compileTestKotlin UP-TO-DATE
> Task :compileTestJava NO-SOURCE
> Task :processTestResources NO-SOURCE
> Task :testClasses UP-TO-DATE
> Task :test
22:16:16.707 [Test worker] DEBUG org.jboss.logging - Logging Provider: org.jboss.logging.Slf4jLoggerProvider
22:16:16.812 [Test worker] DEBUG org.hibernate.jpa.HibernatePersistenceProvider - Located and parsed 1 persistence units; checking each
22:16:16.812 [Test worker] DEBUG org.hibernate.jpa.HibernatePersistenceProvider - Checking persistence-unit [name=default, explicit-provider=org.hibernate
22:16:16.813 [Test worker] DEBUG org.hibernate.jpa.boot.spi.ProviderChecker - Persistence-unit [default] requested PersistenceProvider [org.hibernate.jpa.
22:16:16.818 [Test worker] DEBUG org.hibernate.jpa.internal.util.LogHelper - PersistenceUnitInfo [
name: default
persistence provider classname: org.hibernate.jpa.HibernatePersistenceProvider
classloader: null
excludeUnlistedClasses: false
JTA datasource: null
Non JTA datasource: null