ACCESO A DATOS

TENIS LAB BARCELONA

Realizado por:

Daniel Rodríguez Fernández
Jorge Sánchez Berrocoso
Alfredo Maldonado Pertuz

13/02/2023

Proyecto	TennisLab BARCELONA - AD2		
Entregable	Documentación completada		
Autor	Daniel Rodríguez Fernández Jorge Sánchez Berrocoso Alfredo Maldonado		
Versión/Edición	V1.1	**Fecha Versión**	13/02/2023
Aprobado por	Equipo desarrollo	**Fecha Aprobación**	18/02/2023
		Nº Total de Páginas	52

PROJECT MANAGER
Daniel Rodríguez Fernández Jorge Sánchez Berrocoso Alfredo Maldonado Pertuz

Tabla de contenido

1. Introducción

Se ha planteado un nuevo problema en el módulo de acceso a datos de segundo de desarrollo de aplicaciones multiplataforma. En general hay que crear una solución para una tienda de material de tenis, esta solución nos debe permitir gestionar la personalización de material de los clientes, adquisición de nuevo material, gestionar los usuarios, gestionar los turnos de nuestros empleados así como las tareas que tienen asignadas.
Para resolver este problema vamos a implementar nuevas tecnologías que no habíamos implementado anteriormente, como es Hibernate y JPA.

2. Descripción del problema propuesto

En nuestra aplicación se conectan distintos usuarios, con su nombre, apellido, email y password (siempre codificado en la base de datos, usando sha512). Además, sabemos que existe el perfil de administrador (encargado o jefe), encordador (trabajador) y tenista (cliente). Trabajamos con varias máquinas, que son de encordar o de personalización. Para cada máquina, nos interesa saber su marca, modelo, fecha de adquisición y número de serie. Si la máquina es de encordar, debemos saber si es manual o automática, tensión máxima y tensión mínima de trabajo. Si es de personalizar, debemos saber si puede o no medir maniobrabilidad (swingweight), balance (equilibrio) y rigidez (resilencia). Los pedidos pueden estar formados por varias tareas o partes de trabajo a realizar que recibimos de un tenista y son asignados a un encordador y tiene un estado: recibido, en proceso o terminado y la máquina asociada si se necesita. Tenemos un tope de entrega marcado por una fecha. Los pedidos tienen una fecha de entrada, una fecha de salida programada y de salida final y un precio asociado que es la suma de todas las acciones. La fecha de salida final será inicialmente la programada, luego se actualizará a la real. Para cada tarea/acción a realizar necesitamos saber la raqueta o raquetas (una acción por raqueta) con la que trabajar si se necesita. Si es encordado necesitamos tensión de cuerdas horizontales y cordaje, tensión de cuerdas verticales y cordaje y si queremos dos o cuatro nudos. El precio será 15€ más el precio del producto o productos a usar. Si es personalización necesitamos saber peso en gramos, balance y rigidez. Su precioseráde60€. Por ejemplo, Rafa Nadal encuerda su Raqueta Babolat Pure Aero Rafa a 25Kg tanto horizontal es como verticales en 4 nudos usando Babolat RPM Blast como cordaje. Si es adquisición sumamos el precio del producto adquirido (comprado). Podemos tener en cuenta que podemos tener distintas acciones para un mismo tenista en un pedido, por ejemplo 3 encordados, un equilibrado y cuatro complementos que pueden ser adquiridos. Debemos tener en cuenta que un encordador no puede tener más de dos pedidos activos por turno. Del turno nos interesa saber comienzo y fin del mismo. Un encordador no puede usar otra máquina si ya tiene asignada una en un turno determinado. Además, como vendemos distintos productos del tipo: raquetas, cordajes, y complementos como overgrips, grips, anti vibradores, fundas, etc. Necesitamos saber el tipo, marca, modelo, precio y stock del mismo. Ten en cuenta que solo podrá realizar operaciones CRUD el encargado del sistema, pero la asignación de pedidos la puede hacer también los encordadores. Por otro lado, nos interesa mantener el histórico de los elementos del sistema y

- CRUD completo de los elementos que consideres necesarios.

- Sistema de errores y excepciones personalizados.

- Información completa en JSON de un pedido.

- Listado de pedidos pendientes en JSON.

- Listado de pedidos completados en JSON.

- Listado de productos y servicios que ofrecemos en JSON.

- Listado de asignaciones para los encordadores por fecha en JSON. 1. Completar la información que te falta hasta tener los requisitos de información completos. 2. Realizar el Diagrama de Clases asociado, mostrando la semántica, navegabilidad y cardinalidad de las relaciones, justificando la respuesta con el máximo detalle. Crear las Clases del modelo asociadas así como las tablas para el almacenamiento en una base de datos relacional.

3. Implementación y test de repositorios y controladores de las operaciones CRUD y otras operaciones relevantes aplicando las restricciones indicadas usando Exposed.

4. Implementación y test de repositorios y controladores de las operaciones CRUD y otras operaciones relevantes aplicando las restricciones indicadas usando JPA.

Nuestro programa debe llamarse con un JAR de la siguiente manera: java -jar tennislab.jar. Se debe entregar:

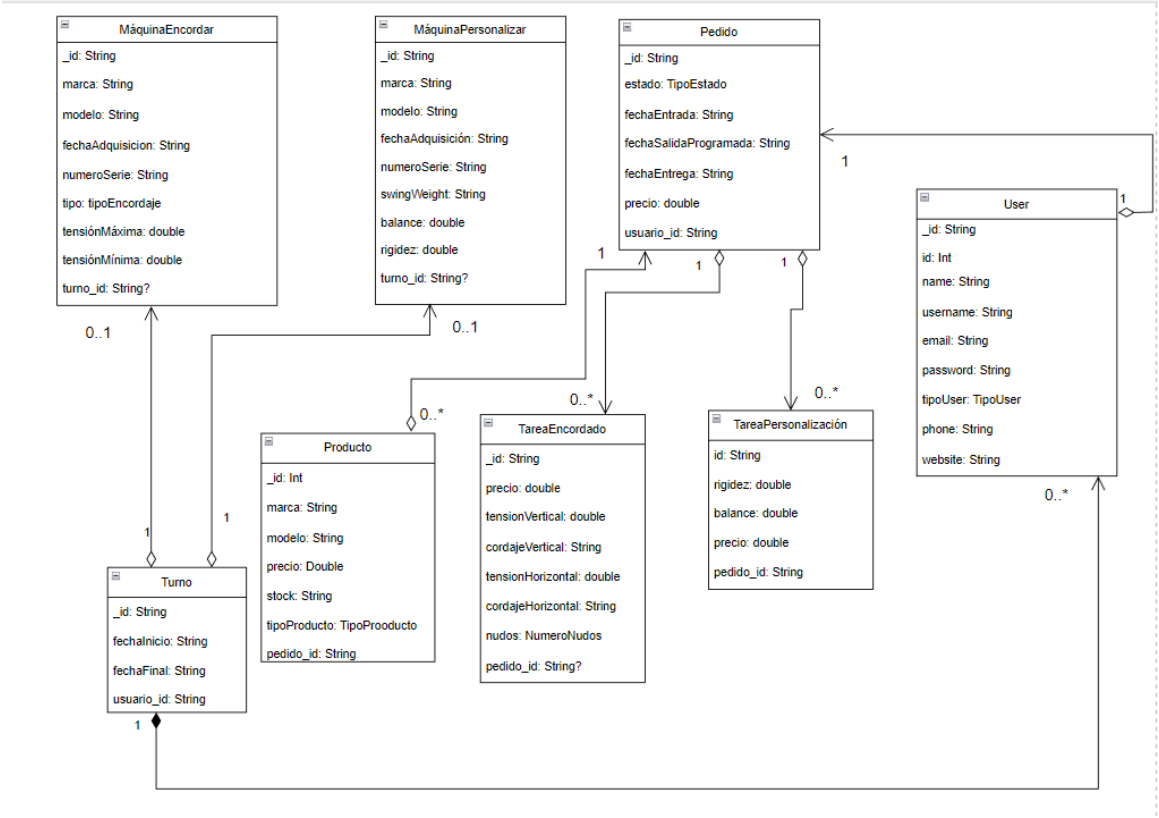
- Repositorio GitHub Personal y el de entrega con la solución en el que incluyas:

o Readme explicando el proyecto y el nombre de los integrantes. Usa Markdown y mejor a su estilo. Si no perderás puntos por la presentación. o Código fuente comentado y perfectamente estructurado con JDoc/KDoc. Además de los gitignore adecuados y que siga el flujo de trabajo GitFlow. o No se deben incluir los ejecutables si no se deben poder crear los jar desde el propio proyecto. Asegúrate que se puede crear y que los ficheros de configuración de la base de datos, así como datos de ejemplo están en directorios que se pueden ejecutar o se pueden leer desde resources. o Documentación en PDF donde expliques el diseño y propuesta de solución, así como clases y elementos usados haciendo especial énfasis en:

Requisitos de Información. Diagrama de clases y justificación de este. Arquitectura del sistema y patrones usados. Explicación de forma de acceso a los datos. La no entrega de este fichero invalidará la práctica. La aplicación no debe fallar y debe reaccionar antes posibles fallos asegurando la consistencia y calidad de esta. o Enlace en el readme al vídeo en YouTube donde se explique las partes más relevantes de la práctica y se muestre su ejecución. La duración del vídeo debe ser unos 30 minutos. La no entrega de este vídeo y donde se vea su ejecución anulará el resultado de la práctica. o Repositorio oficial de la entrega Enlace de entrega: https://classroom.github.com/a/TkCVuv1P. La subirán los dos miembros del equipo, sino está en este repositorio se invalidará la práctica no pudiéndose entregar por otros medios.

3. Diagramas

3.1. Diagrama de Clase



A continuación explicamos la implementación del diagrama de clase que hemos desarrollado a partir de las indicaciones y requisitos del cliente.
Primero de todo vamos a identificar todas las clases que hemos implementado:

- User
- Turno
- TareaEncordado
- TareaPersonalizacion
- MaquinaEncordar
- MaquinaPersonalizar
- Pedido
- Producto

Entre estas clases se han creado relaciones para formar la arquitectura del software, más adelante explicaremos cada relación que se produce en el diagrama de clase.
Una de las primeras justificaciones que queremos dar es que hemos roto todas las herencias que se podían producir en el modelo de esta forma nos ha parecido más rápido a la hora de gestionar las relaciones entre las clases. Tecnológicamente también aportamos una explicación de esta forma a la hora por ejemplo de realizar una consulta la hacemos directamente sobre la clase a la hora de buscar será más rápido de la otra forma tendríamos que buscar la clase no heredada y de ahí saber qué tipo es en que hereda y ejecutar la consulta por lo tanto el tiempo sería mayor.
Vamos a comentar cada una de las clases y qué relación tiene con cada una de las clases del modelo.

- Users: esta clase hace referencia a los usuarios registrados de nuestro sistema, de cada usuario recogeremos algunos datos como puede ser su contraseña que estará encriptada con sha256 o el tipo de usuario que es que le dará más o menos privilegios de realizar operaciones en el sistema.
- Turno: con la clase turno registramos cada vez que un usuario que es de tipo empleado ha comenzado una nuevo turno, registrado su hora de inicio y su hora

de fin hay que tener en cuenta que esta clase tiene una relación con Users de forma referenciada, ya que cada vez que registramos un turno a ese turno le asignamos el id del usuario que registra ese turno.

- Pedido: los pedido registran las solicitudes que realizan los usuarios, hay que destacar que los pedidos están relacionados con las tareas y los usuarios, primero explicamos la relación que tiene con los usuarios.

Un pedido esta realizado por un usuario por lo tanto a ese pedido se le asigna el usuario que lo realiza, para hacer esta asignación lo hacemos por referencia al pedido le asignamos el id del usuario que realiza el pedido.

Ahora explicamos la relación que tiene los pedidos con las tareas de personalización y de encordar. Un pedido está formado por tareas, puede tener una o varias tareas pero nunca podrá ser null.

- TareaEncordado: es una clase que hace la composición al pedido, contiene su id, y el resto de los campos propios de la clase, además tiene un campo que es el número de pedido nunca puede ser null, una tarea siempre tiene que estar asociada a un pedido, lo hacemos de forma referenciada.
- TareaPersonalizacion: es otra clase que hace la composición del pedido, contiene su id, y el resto de los campos que son propios de la clase, además tiene un campo que hace referencia el número de pedido que nunca podrá ser null, es decir que una tarea siempre tiene que estar asociada a un pedido, lo hacemos de forma referenciada a la clase pedido, de esta forma es mucho más rápido ya que no es necesario embeber todo solo aquello que es necesario.
- MaquinaEncordado: es una clase que utilizamos para registrar todas las máquinas de encordado existentes, hay que decir que comparte campos con las máquinas de personalización pero ya que hemos decidido que no exista herencia entre clases, cada clase de maquina tendrá sus propios campos.

Además cada máquina de encordado tendrá asociado un turno de esta forma podemos filtrar y limitar el número de máquinas que puede utilizar un usuario en cada turno.

- MaquinaPersonalizacion: es una clase que utilizamos para registrar todas las máquinas de personalización existentes, hay que decir que comparte campos con las máquinas de encordado pero ya que hemos decidido que no exista herencia entre clases, cada clase de maquina tendrá sus propios campos.

Además cada máquina de encordado tendrá asociado un turno de esta forma podemos filtrar y limitar el número de máquinas que puede utilizar un usuario en cada turno.

- Producto: son productos o servicios que podemos ofrecer y pueden obtener los clientes, tiene atributos uno de ellos es el stock que es la cantidad de elementos existentes de ese producto.

Tiene una relación con los pedido, la relación se hace de forma referenciada, para asignar un producto a un pedido hay que introducir el número de pedido que tenía asignado.

Relaciones:

Pedido-User: Es una relación 1-1 porque se ha decido para que el usuario esté dado de alta en la plataforma tiene que ser con un pedido, y por otro lado solo puede hacer un pedido en el momento.

Pedido-Producto: Es una relación 1-1* porque al hacer un pedido obligatoriamente tiene que haber un producto en el pedido y un máximo indefinido.

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

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

Turno-MaquinaPersonalizar: Al igual que el anterior, se trata de una relación 0..N-1 porque que la máquinaPersonalizar siempre va a existir aunque no se esté realizando ningún turno en ese momento, mientras que puede haber varios turnos que hagan uso de máquinas personalizadoras.

4. Requisitos

4.1. Requisitos Funcionales

Un requisito funcional define una función del sistema de software o sus componentes.

Los requisitos funcionales son: cálculos, detalles técnicos, manipulación de datos y otras funcionalidades específicas que nuestro sistema debe cumplir.

Cod	Requisito Funcional	Check
001	Al iniciar el programa pedirá los datos de inicio de sesión	✓
002	Al meter los datos de inicio de sesión el programa comprobará que está sesión es correcta.	✓
003	Al iniciar una sesión de Gerente correcta nos mostrará el menú de inicio de gerente.	✓
004	Al iniciar una sesión de Gerente correcta nos mostrará el menú de inicio de gerente.	✓
005	Al meter los datos de inicio de sesión erróneos el programa informará de ello y volverá al menú de inicio de sesión	✓
006	Al seleccionar el gerente cambio de contraseña en el menú el programa le mostrará las opciones del menú de cambio de contraseña	✓
007	Al seleccionar el empleado cambio de contraseña en el menú el programa pedirá su antigua contraseña y la nueva	✓
008	El sistema cuando el usuario inserte los datos correspondientes correctos en la opción de “cambiar contraseña” este comprobará los datos y cambiará está confirmará la acción.	✓
009	Al seleccionar el Gerente cambio de contraseña en el menú el programa le mostrará las opciones de cambio de contraseña.	✓
010	Al seleccionar el Gerente cambio de contraseña propia en el menú de cambio de contraseña le mostrará los empleados disponibles para el cambio.	✓
011	Al seleccionar el Gerente “Modificar Usuario”, el sistema le mostrará todos los usuarios del sistema; y pedirá que seleccione uno.	✓
012	Al seleccionar el Gerente “Modificar Usuario” y elegir un usuario, el sistema le mostrará todos datos de este y pedirá los datos nuevos.	✓
013	Al seleccionar el Gerente “Modificar Usuario”, elegir un usuario e introducir los datos, comprobará estos y mostrará un mensaje de confirmación u error dependiendo de si los datos son correctos y se han guardado correctamente.	✓
014	Al seleccionar el gerente “crear usuario” en el menú el programa le pedirá los datos para crear un usuario nuevo.	✓
015	Al seleccionar el Gerente “crear usuario” e introducir los datos, comprobará estos y mostrará un mensaje de	✓

	confirmación u error dependiendo de si los datos son correctos y se han guardado correctamente.	
016	Al seleccionar el gerente “Crear Pedido” en el menú el programa le mostrará un mensaje con el pedido a crear y opciones para añadir tarea, u cambiar datos de pedido.	✓
017	Al seleccionar el gerente “Crear Pedido”, y seleccionar la opción “modificar Datos” el programa pedirá los nuevos datos el pedido.	✓
018	Al seleccionar el gerente “Crear Pedido” seleccionar la opción “modificar Datos” e introducir los datos el programa comprobará los datos, mostrará la confirmación u error de los campos y volverá el menú de pedido.	✓
019	Al seleccionar el gerente “Crear Pedido”, y seleccionar la opción “añadir tarea” el programa pedirá los datos de la tarea, comprobará los datos, mostrará un mensaje de error o confirmación de estos y volverá al menú de pedido.	✓
020	Al seleccionar el gerente “Crear Pedido”, y seleccionar la opción “modificar tarea” el programa pedirá seleccionar la tarea a modificar, comprobará los datos, mostrará un mensaje de error o confirmación de estos e irá al menú correspondiente.	✓
021	Tras seleccionar la opción de crear pedido, si el cliente confirma la creación el programa guardará los datos correspondientes y mostrará un message de confirmación.	✓
022	Al seleccionar el gerente “Crear turno” el programa mostrará el menú correspondiente y pedirá los datos para la creación.	✓
023	Al seleccionar el gerente “Crear turno” e introducir los datos correspondientes, el programa comprobará los datos, mostrará un mensaje de confirmación u error y devolverá al menú anterior.	✓
024	Al seleccionar el gerente “modificar turno” e introducir los datos correspondientes, el programa comprobará los datos, mostrará un mensaje de confirmación u error y devolverá al menú anterior.	✓
025	Al seleccionar el gerente “modificar turno” e introducir los datos correspondientes, el programa comprobará los datos, mostrará un mensaje de confirmación u error y devolverá al menú anterior.	✓
026	Al seleccionar el gerente “asignar tareas” en el menú el programa le mostrará las tareas y empleados del sistema, y pedirá los datos para la asignación.	✓
027	Al seleccionar el gerente “asignar tareas” e introducir los datos para la asignación, el programa comprobará los datos y mandará un mensaje de confirmación u error la introducción de datos en el sistema.	✓

028	Al seleccionar el gerente “Modificar tarea” en el menú el programa le mostrará las tareas del sistema, y pedirá los datos para la asignación.	✓
029	Al seleccionar el gerente “Modificar tareas” e introducir los datos para la tarea, el programa comprobará los datos y mandará un mensaje de confirmación u error la introducción de datos en el sistema.	✓
030	Al seleccionar el gerente “crear Maquina” en el menú el programa le pedirá los datos de la maquina nueva.	✓
031	Al seleccionar el gerente “crear Maquina” e introducir los datos de la maquina nueva, el sistema comprobará que los datos son correctos y mostrará un mensaje de error o confirmación de la introducción e los datos en el sistema.	✓
032	Al seleccionar el gerente “Modificar Maquina” en el menú el programa le pedirá los datos de la maquina nueva,	✓
033	Al seleccionar el gerente “Modificar Maquina” e introducir los datos de la máquina, el sistema comprobará que los datos son correctos y mostrará un mensaje de error o confirmación de la introducción e los datos en el sistema.	✓
034	Al seleccionar el gerente “Crear producto” en el menú el programa le pedirá los datos correspondientes a la creación.	✓
035	Al seleccionar el gerente “Crear Raqueta” en el menú el programa le pedirá los datos correspondientes a la creación.	✓
036	Al seleccionar el gerente “Crear Raqueta” e introducir los datos el sistema comprobará que los datos son correctos y mostrará un mensaje de error o confirmación de la introducción de los datos en el sistema.	✓
037	Al seleccionar el gerente “Crear Producto” e introducir los datos el sistema comprobará que los datos son correctos y mostrará un mensaje de error o confirmación de la introducción de los datos en el sistema.	✓
038	Al seleccionar el Empleado “Modificar tarea” en el menú el programa le mostrará las tareas del sistema, y pedirá los datos para la asignación.	✓
039	Al seleccionar el Empleado “Modificar tareas” e introducir los datos para la tarea, el programa comprobará los datos y mandará un mensaje de confirmación u error la introducción de datos en el sistema.	✓
040	Al seleccionar el Empleado “modificar turno” e introducir los datos correspondientes, el programa comprobará los	✓

	datos, mostrará un mensaje de confirmación u error y devolverá al menú anterior.	
041	Al seleccionar el Empleado “modificar turno” e introducir los datos correspondientes, el programa comprobará los datos, mostrará un mensaje de confirmación u error y devolverá al menú anterior.	✓
042	Al seleccionar el Empleado “modificar turno” e introducir los datos correspondientes, el programa comprobará los datos, mostrará un mensaje de confirmación u error y devolverá al menú anterior.	✓
043	Al seleccionar el Empleado “asignar tareas” en el menú el programa le mostrará las tareas y empleados del sistema, y pedirá los datos para la asignación.	✓
044	Al seleccionar el Empleado “asignar tareas” e introducir los datos para la asignación, el programa comprobará los datos y mandará un mensaje de confirmación u error la introducción de datos en el sistema.	✓
045	Al seleccionar el Empleado “Modificar tarea” en el menú el programa le mostrará las tareas del sistema, y pedirá los datos para la asignación.	✓
046	Al seleccionar el Empleado “Modificar tareas” e introducir los datos para la tarea, el programa comprobará los datos y mandará un mensaje de confirmación u error la introducción de datos en el sistema.	✓
047	Al seleccionar el Empleado “Modificar tarea” en el menú el programa le mostrará las tareas del sistema, y pedirá los datos para la asignación.	✓
048	Al seleccionar el Empleado “Modificar tareas” e introducir los datos para la tarea, el programa comprobará los datos y mandará un mensaje de confirmación u error la introducción de datos en el sistema.	✓
049	Al seleccionar el Usuario “Volver” el programa iniciará el menú de sesión.	✓
050	Al seleccionar el Usuario “Salir” el programa se realizará una copia de seguridad y se cerrará.	✓
051	Al seleccionar el gerente “Imprimir Pedido” el sistema mostrará un listado de los pedidos en el sistema.	✓
052	Al seleccionar el gerente “Imprimir Pedido” e introducir pedido que se quiera, el sistema comprobará los datos, mostrará un mensaje de confirmación u error y en caso de confirmación creará fichero json en una carpeta del sistema.	✓
053	Al seleccionar el gerente “Imprimir Pedidos Pendientes” el sistema creará un listado de los pedidos pendientes en el sistema en Json.	✓
054	Al seleccionar el gerente “Imprimir Pedidos Completos” el sistema creará un listado de los pedidos completos en el sistema en Json.	✓

055	Al seleccionar el gerente “Imprimir Listado productos y servicios” el sistema creará un listado de los productos y servicios en el sistema en Json.	✓
056	Al seleccionar el gerente “Imprimir Listado asignaciones” el sistema creará un listado desde las asignaciones ordenado por fecha en el sistema en Json.	✓

4.2. Requisitos No Funcionales

Los Requisitos No Funcionales se refieren a todos los requisitos que describen características de funcionamiento.

Se suelen clasificar en:

Requisitos de calidad de ejecución, que incluyen seguridad, usabilidad y otros medibles en tiempo de ejecución.

Requisitos de calidad de evolución, como testeabilidad, extensibilidad o escalabilidad, que se evalúan en los elementos estáticos del sistema software.

Cod	Requisito No Funcional	Check
RNF1	Serialization: Kotlin	✓
RNF2	Hibernate como opción.	✓
RNF3	Exposed como opción 2.	✓
RNF4	Ejecución del programa resultante a través de un JAR	✓
RNF5	Testeado con Junit	✓
RNF6	Testear controladores con Mockk	✓
RNF7	Código comentado con Kdoc	✓
RNF8	Uso de Bases de datos relaciones como H2, SQLite, MySQL, MariaDB.	✓
RNF9	Exportar informes en ficheros de tipo JSON	✓

4.3. Requisitos de Información

Cod	Requisito de Información	Check
RI1	Users: • _id: String • name: String • username: String • email: String • password: String • tipoUser: String • phone: String • website: String	✓
RI2	Turno: • _id: String • fechaInicio: LocalDate • fechaFinal: LocalDate • usuario_id: String	✓
RI3	MaquinaEncordar: • _id: String • marcaModelo: String • modelo: String • fechaAdquisición: LocalDate • numeroSerie: String	✓

	• tipo: TipoEncordaje • tensiónMáxima: double • tensiónMínima: double • turno_id: String?	
RI4	MaquinaPersonalizar: • _id: String • marca: String • modelo: String • fechaAdquisición: LocalDate • numeroSerie: String • swingweight: String • balance: double • rigidez: double • turno_id: string?	✓
RI5	Pedido: • _id: String • estado: String • fechaEntrada: LocalDate • fechaSalidaProgramada: LocalDate • fechaEntrega: LocalDate • precio: double • usuario_id: String	✓
RI6	Producto: • _id: String • marca: String • modelo: String • precio: double • stock: string • tipoProducto: TipoProducto • pedido_id: String	✓
RI7	TareaEncordado: • _id: String • precio: String • tensionVertical: String • cordajeVertical: double • tensionHorizontal: string • cordajeHorizontal: TipoProducto • nudos: NumeroNudos • pedido_id: String?	✓
RI8	TareaPersonalización: • _id: String • rigidez: String • peso: String • balance: double • precio: string • pedido_id: String	✓

5. Evaluación y Análisis

5.1. JPA

5.1.1. Paquete db.

Podemos encontrar en primer lugar el objeto HibernateManager, lo utilizamos para realizar la conexión a la base de datos a través de Hibernate, con este manager podemos gestionar las transacciones que se realizan entre el código y la base de datos.

```
object HibernateManager : Closeable {
    private var entityManagerFactory =
Persistence.createEntityManagerFactory("TennisLab")
    lateinit var manager: EntityManager
    private lateinit var transaction: EntityTransaction

    init {
        entityManagerFactory =
Persistence.createEntityManagerFactory("TennisLab")
        manager = entityManagerFactory.createEntityManager()
        transaction = manager.transaction
    }
    fun open() {
        logger.debug { "Iniciando EntityManagerFactory" }
        manager = entityManagerFactory.createEntityManager()
        transaction = manager.transaction
    }
    override fun close() {
        logger.debug { "Cerrando EntityManager" }
        manager.close()
    }
    fun query(operations: () -> Unit) {
        open()
        try {
            operations()
        } catch (e: SQLException) {
            logger.error { "Error en la consulta: ${e.message}" }
        } finally {
            close()
        }
    }
    fun transaction(operations: () -> Unit) {
        open()
        try {
            logger.debug { "Transaction iniciada" }
            transaction.begin()
            operations()
            transaction.commit()
            logger.debug { "Transaction finalizada" }
        } catch (e: SQLException) {
            transaction.rollback()
            logger.error { "Error en la transacción: ${e.message}" }
            throw SQLException(e)
        } finally {
            close()
        }
    }
}
```

En el fichero data podemos encontrar varios métodos, tantos como modelos hemos creado en nuestro proyecto, en cada método estamos creando varios modelos por ejemplo de máquinas, turnos, usuarios, tareas, pedidos … Para que cuando iniciemos la

base de datos ya tengamos unos datos cargados de ejemplo y podamos operar y ver el comportamiento.

```
fun getUsuariosInit() = listOf(
    Usuario(
        id = 0,
        uuid = UUID.randomUUID(),
        nombre = "Alfonso",
        apellido = "Cabello",
        correo = "alfonso@cabello.com",
        password = "1234",
        tipoUsuario = TipoUsuario.USUARIO
    ),
    Usuario(
        id = 1,
        uuid = UUID.randomUUID(),
        nombre = "Marcelo",
        apellido = "Alvarez",
        correo = "marcelo@alvarez.com",
        password = "1234",
        tipoUsuario = TipoUsuario.ADMINISTRADOR
    ),
    Usuario(
        id = 2,
        uuid = UUID.randomUUID(),
        nombre = "Fernando",
        apellido = "Alonso",
        correo = "fernado@alonso.com",
        password = "1234",
        tipoUsuario = TipoUsuario.ENCORDADOR
    ),
    Usuario(
        id = 3,
        uuid = UUID.randomUUID(),
        nombre = "Cristiano",
        apellido = "Ronaldo",
        correo = "cristiano@ronaldo.com",
        password = "1234",
        tipoUsuario = TipoUsuario.USUARIO
    ),
    Usuario(
        id = 4,
        uuid = UUID.randomUUID(),
        nombre = "Rafael",
        apellido = "Nadal",
        correo = "rafael@nadal.com",
        password = "1234",
        tipoUsuario = TipoUsuario.ADMINISTRADOR
    )
)

fun getProductosInit() = listOf(
    Producto(
        id = 0,
        uuid = UUID.randomUUID(),
        marca = "Nike",
        modelo = "H10",
        precio = 50.45,
        stock = 120,
        tipoProducto = TipoProducto.CORDAJE,
        pedido = getPedidosInit()[0]
    ),
    Producto(
        id = 1,
        uuid = UUID.randomUUID(),
```

```
        marca = "Nike",
        modelo = "H10",
        precio = 50.45,
        stock = 120,
        tipoProducto = TipoProducto.CORDAJE,
        pedido = getPedidosInit()[1]
    ),
    Producto(
        id = 2,
        uuid = UUID.randomUUID(),
        marca = "Nike",
        modelo = "H10",
        precio = 50.45,
        stock = 120,
        tipoProducto = TipoProducto.CORDAJE,
        pedido = getPedidosInit()[2]
    ),
    Producto(
        id = 3,
        uuid = UUID.randomUUID(),
        marca = "Nike",
        modelo = "H10",
        precio = 50.45,
        stock = 120,
        tipoProducto = TipoProducto.CORDAJE,
        pedido = getPedidosInit()[3]
    ),
    Producto(
        id = 4,
        uuid = UUID.randomUUID(),
        marca = "Nike",
        modelo = "H10",
        precio = 50.45,
        stock = 120,
        tipoProducto = TipoProducto.CORDAJE,
        pedido = getPedidosInit()[4]
    ),
)

fun getPedidosInit() = listOf(
    Pedido(
        id = 0,
        uuid = UUID.randomUUID(),
        estado = TipoEstado.TERMINADO,
        fechaEntrada = LocalDate.of(2022, 3, 15),
        fechaSalidaProgramada = LocalDate.of(2022, 5, 10),
        fechaEntrega = LocalDate.of(2022, 6, 7),
        precio = 10.20,
        usuario = getUsuariosInit()[0]
    ),
    Pedido(
        id = 1,
        uuid = UUID.randomUUID(),
        estado = TipoEstado.TERMINADO,
        fechaEntrada = LocalDate.of(2022, 2, 15),
        fechaSalidaProgramada = LocalDate.of(2022, 6, 10),
        fechaEntrega = LocalDate.of(2022, 7, 10),
        precio = 10.20,
        usuario = getUsuariosInit()[1]
    ),

    Pedido(
        id = 2,
        uuid = UUID.randomUUID(),
        estado = TipoEstado.TERMINADO,
```

```
        fechaEntrada = LocalDate.of(2022, 4, 15),
        fechaSalidaProgramada = LocalDate.of(2022, 9, 10),
        fechaEntrega = LocalDate.of(2022, 10, 10),
        precio = 10.20,
        usuario = getUsuariosInit()[2]
    ),
    Pedido(
        id = 3,
        uuid = UUID.randomUUID(),
        estado = TipoEstado.TERMINADO,
        fechaEntrada = LocalDate.of(2022, 6, 15),
        fechaSalidaProgramada = LocalDate.of(2022, 6, 20),
        fechaEntrega = LocalDate.of(2022, 7, 10),
        precio = 10.20,
        usuario = getUsuariosInit()[3]
    ),
    Pedido(
        id = 4,
        uuid = UUID.randomUUID(),
        estado = TipoEstado.TERMINADO,
        fechaEntrada = LocalDate.of(2022, 6, 15),
        fechaSalidaProgramada = LocalDate.of(2022, 7, 10),
        fechaEntrega = LocalDate.of(2022, 8, 10),
        precio = 10.20,
        usuario = getUsuariosInit()[4]
    ),
)

fun getTareasEncoradadoInit() = listOf(
    TareaEncordado(
        id = 0,
        uuid = UUID.randomUUID(),
        precio = 100.50,
        tensionVertical = 22.50,
        cordajeVertical = "Babolat",
        tensionHorizontal = 26.10,
        cordajeHorizontal = "Babolat",
        nudos = NumeroNudos.DOS,
        pedido = getPedidosInit()[0],
    ),
    TareaEncordado(
        id = 1,
        uuid = UUID.randomUUID(),
        precio = 100.50,
        tensionVertical = 22.50,
        cordajeVertical = "Wilson",
        tensionHorizontal = 26.10,
        cordajeHorizontal = "Wilson",
        nudos = NumeroNudos.DOS,
        pedido = getPedidosInit()[1],
    ),
)

fun getTareasPersonalizacion() = listOf(
    TareaPersonalizacion(
        id = 0,
        uuid = UUID.randomUUID(),
        rigidez = 69.69,
        peso = 0.20,
        balance = 327.10,
        precio = 99.99,
        pedido = getPedidosInit()[2]
    ),
    TareaPersonalizacion(
        id = 1,
```

```
        uuid = UUID.randomUUID(),
        rigidez = 79.69,
        peso = 0.27,
        balance = 326.10,
        precio = 89.99,
        pedido = getPedidosInit()[3]
    ),
    TareaPersonalizacion(
        id = 2,
        uuid = UUID.randomUUID(),
        rigidez = 59.69,
        peso = 0.25,
        balance = 327.10,
        precio = 79.99,
        pedido = getPedidosInit()[4]
    ),
)

fun getMaquinasEncordar() = listOf(
    MaquinaEncordar(
        id = 0,
        uuid = UUID.randomUUID(),
        marca = "HEAD",
        modelo = "2020",
        fechaAdquisicion = LocalDate.of(2022, 3, 5),
        numeroSerie = 20,
        tipo = TipoEncordaje.MANUAL,
        tensionMaxima = 25.10,
        tensionMinima = 22.60,
        turno = getTurnoInit()[0]
    ),
    MaquinaEncordar(
        id = 1,
        uuid = UUID.randomUUID(),
        marca = "HEAD",
        modelo = "2020",
        fechaAdquisicion = LocalDate.of(2022, 5, 5),
        numeroSerie = 20,
        tipo = TipoEncordaje.MANUAL,
        tensionMaxima = 26.10,
        tensionMinima = 23.60,
        turno = getTurnoInit()[0]
    ),
)

fun getMaquinasPersonalizar() = listOf(
    MaquinaPersonalizar(
        id = 0,
        uuid = UUID.randomUUID(),
        marca = "Signum Pro",
        modelo = "30",
        fechaAdquisicion = LocalDate.of(2022, 4, 15),
        numeroSerie = 7,
        swingweight = true,
        balance = 300.80,
        rigidez = 100.60,
        turno = getTurnoInit()[0],
    ),
    MaquinaPersonalizar(
        id = 1,
        uuid = UUID.randomUUID(),
        marca = "Solinco",
        modelo = "50",
        fechaAdquisicion = LocalDate.of(2022, 5, 15),
        numeroSerie = 9,
```

```
        swingweight = false,
        balance = 310.80,
        rigidez = 120.60,
        turno = getTurnoInit()[1],
    ),
)

fun getTurnoInit() = listOf(
    Turno(
        id = 0,
        uuid = UUID.randomUUID(),
        fechaInicio = LocalDateTime.of(2022, 10, 14, 9, 10),
        fechaFinal = LocalDateTime.of(2022, 11, 20, 10, 20),
        usuario = getUsuariosInit()[0]
    ),
    Turno(
        id = 1,
        uuid = UUID.randomUUID(),
        fechaInicio = LocalDateTime.of(2022, 8, 14, 9, 10),
        fechaFinal = LocalDateTime.of(2022, 9, 20, 10, 20),
        usuario = getUsuariosInit()[1]
    ),
    Turno(
        id = 2,
        uuid = UUID.randomUUID(),
        fechaInicio = LocalDateTime.of(2022, 7, 14, 9, 10),
        fechaFinal = LocalDateTime.of(2022, 8, 20, 10, 20),
        usuario = getUsuariosInit()[2]
    ),
)
```

5.1.2. Extensions

Paquete que contiene las extensiones necesarias para complementar funcionalidades de nuestro programa.

Aquí podemos encontrar el fichero Locale, que lo utilizamos para mostrar los datos en el formato que deseemos dependiendo del país en el que nos encontremos, en nuestro caso la moneda y la fecha va en formato español.

```
fun Double.toDinero(): String {
    return NumberFormat.getCurrencyInstance(Locale("es", "ES")).format(this)
}

fun LocalDate.toLocalDate(): String {
    return this.format(
        DateTimeFormatter
            .ofLocalizedDate(FormatStyle.MEDIUM).withLocale(Locale("es",
"ES"))
    )
}
```

5.1.3. Models

Aquí Podemos encontrar todas las clases que utilizamos en nuestro programa además de un paquete de enumns. Como todos los modelos son similares, solo cambian los tipos de datos que almacenamos, vamos a explicar con detalle uno de ellos.

En este caso vamos a explicar el modelo Usuario, utilizamos varias anotaciones que son de Hibernate, en primer lugar usamos @Entity que con ella establecemos la entidad, usamos @Table con esta anotación indicamos el nombre de la tabla en la bbdd, también implementamos @NamedQuery con esta anotación creamos una query sql de la clase que acabamos de crear, usamos @Id para indicar cual es la primary key de la clase y la

anotación @Column para indicar otra referencia de nuestra clase que podamos consultar con ella más adelante.

```
@Entity
@Table(name = "Usuario")
@NamedQuery(name = "Usuario.findAll", query = "SELECT t FROM Usuario t")
data class Usuario(
    @Id
    val id:Int,
    @Column(name="UUID_Usuario")
    @Type(type = "uuid-char")
    val uuid: UUID,
    val nombre: String,
    val apellido: String,
    val correo: String,
    val password: String,
    val tipoUsuario: TipoUsuario,
)
```

Aquí también podemos ver como hemos creado uno de los enumns en este caso el de TipoEstado, este enum lo utilizamos para clasificar el estado en el que se encuentra un pedido.

```
enum class TipoEstado(val est: String) {
    RECIBIDO("RECIBIDO"),
    EN_PROCESO("EN PROCESO"),
    TERMINADO("TERMINADO");

    companion object {
        fun from(estado: String): TipoEstado {
            return when (estado.uppercase()) {
                "RECIBIDO" -> RECIBIDO
                "EN_PROCESO" -> EN_PROCESO
                "TERMINADO" -> TERMINADO
                else -> throw IllegalArgumentException("Estado no
reconocido.")
            }
        }
    }
}
```

5.1.4. Exceptions

En el paquete de excepciones podemos encontrar cada una de las excepciones que van a utilizar los repositorios y controladores del proyecto. De esta forma podemos sacar un mensaje personalizado dependiendo de la excepción que se genere. A continuación podemos ver un ejemplo de excepción, el resto de las excepciones son creadas de la misma forma.

```
class MaquinaEncordadoraException(message: String) :
RuntimeException(message) {
}
```

5.1.5. Repositories

En el paquete repositories podemos encontrar cada uno de los repositorios que tiene un modelo asociado. Hemos implementado SOLID, a la hora de crear el diseño y arquitectura de los repositorios, en este caso el Principio de Responsabilidad Única.

Para ello hemos creado una interfaz genérica que comparten todos los repositorios.

```
interface ICRUDRepository<T,ID> {
    fun findAll(): List<T>
    fun findById(id: ID): T?
    fun save(entity: T): T
    fun delete(entity: T): Boolean
```

Esta interfaz se la hemos añadido a cada interfaz de repositorio de cada modelo, sí algún modelo necesita añadir algún método diferente lo podrá hacer aquí de forma que no afecte al resto de repositorios.

```
interface IUsuarioRepository: ICRUDRepository<Usuario, Int> {
}
```

A continuación podemos ver la implementación del repositorio de usuarios en este caso si analizamos en detalle el código podemos ver que implementamos la interfaz de usuariosRepository y de esta forma sus métodos. Los métodos como podemos comprobar son los de un CRUD, en cada método del repositorio llamamos a nuestro manager que se encarga de gestionar la transacción de datos con la base de datos.

```
class UsuarioRepositoryImplement: IUsuarioRepository {
    override fun findAll(): List<Usuario> {
        var usuarios = mutableListOf<Usuario>()
        HibernateManager.query {
            val query: TypedQuery<Usuario> =
manager.createNamedQuery("Usuario.findAll", Usuario::class.java)
            usuarios = query.resultList
        }
        return usuarios

    }

    override fun findById(id: Int): Usuario? {
        var usuario: Usuario? = null
        HibernateManager.query {
            usuario = manager.find(Usuario::class.java, id)
        }
        return usuario
    }

    override fun save(entity: Usuario): Usuario {
        HibernateManager.transaction {
            manager.merge(entity)
        }
        return entity
    }

    override fun delete(entity: Usuario): Boolean {
        var res = false
        HibernateManager.transaction {
            val usuario = manager.find(Usuario::class.java, entity.id)
            if (usuario != null) {
                manager.remove(entity.id)
                res = true
            }
        }
        return res
    }
}
```

5.1.6. Dto

A continuación explicamos la funcionalidad de este paquete y los ficheros que contiene que funcionalidad tienen. En este paquete almacenamos todos los ficheros necesarios y relacionados con los dto de cada modelo, para poder serializar y convertir cada modelo en un JSON, XML, String, es necesario utilizar estás clases que nos permitan convertir una clase de Kotlin a un JSON por ejemplo. Para ello utilizamos la anotación @Serializable que es de Kotlin y nos permite convertir una clase normal en un JSON por ejemplo.

```
@Serializable
data class EncordadoDTO(
    val id: Int,
    val uuid: String,
    val precio: Double,
    val pedido: String,
    val tensionVertical: Double,
    val cordajeVertical: String,
    val tensionHorizontal: Double,
    val cordajeHorizontal: String,
    val nudos: String
)


fun TareaEncordado.toDTO(): EncordadoDTO {
    return EncordadoDTO(
        id = id,
        uuid = uuid.toString(),
        precio = precio,
        tensionVertical = tensionVertical,
        cordajeVertical = cordajeVertical,
        tensionHorizontal = tensionHorizontal,
        cordajeHorizontal = cordajeHorizontal,
        nudos = nudos.toString(),
        pedido = pedido.toString(),

        )
}
```

5.1.7. Controllers

Los controladores se encargan de gestionar cada una de las acciones que tiene que hacer nuestro programa, nosotros hemos decidido crear un controlador por modelo, en vez de tenerlos todos en uno, así de esta forma nos podemos asegura que cada controlador solo controla su modelo correspondiente.

En este caso vemos el controlador de Usuarios, podemos ver que gestiona las principales operaciones CRUD del repositorio.

```
class UsuarioController(
    private val usuarioRepository: IUsuarioRepository =
UsuarioRepositoryImplement()
) {
    fun findAll(): List<Usuario> {
        return usuarioRepository.findAll()
    }

    fun findById(id: Int): Usuario? {
        return usuarioRepository.findById(id)
    }

    fun save(usuario: Usuario): Usuario {
        return usuarioRepository.save(usuario)
    }
```

```
    fun delete(usuario: Usuario): Boolean {
        return usuarioRepository.delete(usuario)
    }
}
```

5.1.8. Serializers

En este paquete podemos encontrar un fichero que contiene todos los métodos para serializar aquellas clases que nos interesen. En este caso serializamos para obtener los datos en un archivo JSON.

```
class JSON {

    fun turnoJSON( turnoDTO: List<TurnoDTO>) {
        val directorio = System.getProperty("user.dir") +
                File.separator + "src" +
                File.separator + "main" +
                File.separator + "resources"
        val fichero = File(directorio + File.separator + "turnosDTO.json")
        val json = Json { prettyPrint = true }
        fichero.writeText(json.encodeToString(turnoDTO))
    }

    fun pedidoJSON( pedidoDTO: List<PedidoDTO>) {
        val directorio = System.getProperty("user.dir") +
                File.separator + "src" +
                File.separator + "main" +
                File.separator + "resources"
        val fichero = File(directorio + File.separator + "pedidosJSON.json")
        val json = Json { prettyPrint = true }
        fichero.writeText(json.encodeToString(pedidoDTO))
    }

    fun productoJSON(productoDTO: List<ProductoDTO>) {
        val directorio = System.getProperty("user.dir") +
                File.separator + "src" +
                File.separator + "main" +
                File.separator + "resources"
        val fichero = File(directorio + File.separator +
"productosJSON.json")
        val json = Json { prettyPrint = true }
        fichero.writeText(json.encodeToString(productoDTO))
    }
}
```

5.1.9. Services

En el paquete Services podemos encontrar una clase que se encarga de encriptar la contraseña de los usuarios implementando el algoritmo de cifrado sha256.

```
class EncriptarPassword {
    fun encriptar(contra: String): String {
        return Hashing.sha256()
            .hashString(contra, StandardCharsets.UTF_8)
            .toString()
    }
}
```

5.1.10. Utils

Podemos encontrar la clase de propiedades de nuestro proyecto que se encarga de realizar la lectura del fichero de propiedades que tenemos en resources.

```
class ApplicationProperties {
    private val properties: Properties = Properties()
```

```
    init {
        try {

properties.load(javaClass.classLoader.getResourceAsStream("application.prop
erties"))
        } catch (ex: IOException) {
            System.err.println("IOException Ocurrido al leer el fichero de
propiedades: " + ex.message)
            logger.error { "IOException Ocurrido al leer el fichero de
propiedades: " + ex.message }
        }
    }

    fun readProperty(keyName: String?): String {
        return properties.getProperty(keyName, "No existe esa clave en el
fichero de propiedades")
    }
}
```

Si nos vamos a nuestro application properties vemos el siguiente contenido que hace referencia a la configuración de propiedades de nuestro proyecto.

```
app.title=TennisLab Dam
app.curso=Acceso a Datos
app.version=v1.0
database.server.url=jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;
database.server.port=3306
database.name=tennislab
database.username=sa
database.password=
database.jdbc.driver=org.h2.Driver
database.init=true
```

5.1.11. Main

En nuestro main podemos ver todo el código relacionado con la ejecución y lógica del programa, como podemos ver cargamos datos, nos loggeamos, y llamando a los controladores podemos realizar cada una de las operaciones CRUD de cada uno de los modelos que hemos creado.

```
fun main(){
        initDatabase()
    val maquinaController=MaquinaController()
    val pedidoController=PedidoController()
    val productoController=ProductoController()
    val tareaController=TareaController()
    val turnoController=TurnoController()
    val usuarioController=UsuarioController()
    val serviceJSON= JSON()

    getUsuariosInit().forEach{usuario -> usuarioController.save(usuario)   }
    getTurnoInit().forEach { turno ->turnoController.save(turno)   }
    getPedidosInit().forEach{pedido ->pedidoController.save(pedido)   }
    getProductosInit().forEach{producto ->productoController.save(producto)
}
    getTareasEncoradadoInit().forEach{tarea -
>tareaController.saveTareaEncordado(tarea)}
    getTareasPersonalizacion().forEach { tarea -
>tareaController.saveTareaPersonalizacion(tarea) }
    getMaquinasEncordar().forEach { maquina ->
maquinaController.saveMaquinaEncordar(maquina) }
    getMaquinasPersonalizar().forEach { maquina ->
maquinaController.saveMaquinaPersonalizar(maquina) }
```

```
    val pedido = pedidoController.findById(2)?.toDTO()
    serviceJSON.pedidoJSON(listOf(pedido!!))

    val pedidosPendientes = pedidoController
        .findAll()
        .filter { it.estado == TipoEstado.EN_PROCESO }
        .map { it.toDTO() }
    serviceJSON.pedidoJSON(pedidosPendientes)

    val productos = productoController
        .findAll()
        .map { it.toDTO() }
    serviceJSON.productoJSON( productos)

    val asignacion = turnoController
        .findAll()
        .sortedBy { it.fechaInicio }
        .map { it.toDTO() }
    serviceJSON.turnoJSON(asignacion)

}


fun initDatabase() {
    val properties = ApplicationProperties()
    logger.debug { "Leyendo fichero de configuración..." +
properties.readProperty("app.title") }
    HibernateManager.open()
    HibernateManager.close()
}
```

5.1.12. Test Repositorios

Para hacer los test de los repositorios hemos utilizado JUNIT de esta forma podemos comprobar que cada uno de los métodos que hemos implementado en el repositorio funciona tal y como nos piden las especificaciones, por cada repositorio hacemos su correspondiente test llamando al repositorio y cargando unos datos de prueba.

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
internal class UsuarioRepositoryImplementTest {
    private val usuarioRepositoryImplement = UsuarioRepositoryImplement()

    private val usuario = Usuario(
        id = 0,
        uuid = UUID.randomUUID(),
        nombre = "Alfonso",
        apellido = "Cabello",
        correo = "alfonso@cabello.com",
        password = "1234",
        tipoUsuario = TipoUsuario.USUARIO
    )
    @AfterEach
    fun tearDown(){
        HibernateManager.transaction {
            val query= HibernateManager.manager.createNativeQuery("DELETE
FROM Usuario")
            query.executeUpdate()
        }
    }
    @BeforeEach
    fun beforeEach(){
        HibernateManager.transaction {
            val query = HibernateManager.manager.createNativeQuery("DELETE
FROM Usuario")
```

```
            query.executeUpdate()
        }
        HibernateManager.open()
        HibernateManager.close()
    }
    @Test
    fun findAll(){
        val res = usuarioRepositoryImplement.findAll()

        assert(res.isEmpty())
    }
    //Da fallo
    @Test
    fun findById(){
        usuarioRepositoryImplement.save(usuario)
        val res = usuarioRepositoryImplement.findById(usuario.id)
        assert(res == usuario)
    }
    @Test
    fun findByIdNoExiste(){
        val res = usuarioRepositoryImplement.findById(-5)
        assert(res==null)
    }
    //Da fallo
    @Test
    fun saveInsert(){
        val res = usuarioRepositoryImplement.save(usuario)

        assertAll(
            { assertEquals(res.id, usuario.id) },
            { assertEquals(res.uuid, usuario.uuid) },
            { assertEquals(res.nombre, usuario.nombre) },
            { assertEquals(res.apellido, usuario.apellido) },
            { assertEquals(res.correo, usuario.correo) },
            { assertEquals(res.password, usuario.password) },
            { assertEquals(res.tipoUsuario, usuario.tipoUsuario) }
        )
    }
    @Test
    fun delete(){
        usuarioRepositoryImplement.save(usuario)
        val res = usuarioRepositoryImplement.delete(usuario)
        assert(res)
    }
    @Test
    fun deleteNoExiste(){
        val res = usuarioRepositoryImplement.delete(usuario)
        assert(!res)
    }
}
```

5.1.13. Test controladores

Para hacer los test de los repositorios hemos utilizado MockK para Kotlin de esta forma podemos comprobar que cada uno de los métodos que hemos implementado en el controlador funciona tal y como nos piden las especificaciones, por cada controlador hacemos su correspondiente test llamando al controlador y cargando unos datos de prueba.

```
internal class UsuarioControllerTest {
    @MockK
    lateinit var usuarioRepositoryImplement: UsuarioRepositoryImplement
    @InjectMockKs
    lateinit var usuarioController: UsuarioController

    private val usuario= Usuario(
        id = 1,
        uuid = UUID.randomUUID(),
        nombre = "Marcelo",
        apellido = "Alvarez",
        correo = "marcelo@alvarez.com",
        password = "1234",
        tipoUsuario = TipoUsuario.ADMINISTRADOR
    )
    init {
        MockKAnnotations.init(this)
    }
    @Test
    fun findAllUsuarios(){
        every{usuarioRepositoryImplement.findAll()} returns listOf(usuario)
        val res = usuarioController.findAll()
        assert(res == listOf(usuario))
        verify(exactly = 1) {usuarioRepositoryImplement.findAll()  }
    }
    @Test
    fun findByIdUsuario(){
        every { usuarioRepositoryImplement.findById(usuario.id) }returns
usuario
        val res = usuarioController.findById(usuario.id)
        assert(res==usuario)
        verify(exactly = 1)
{usuarioRepositoryImplement.findById(usuario.id)  }
    }
    @Test
    fun findByIdNoExisteUsuario(){
        every { usuarioRepositoryImplement.findById(usuario.id) }returns
null
        val res = usuarioController.findById(usuario.id)
        assert(res == null)
        verify(exactly = 1)
{usuarioRepositoryImplement.findById(usuario.id)  }
    }
    @Test
    fun saveUsuario(){
        every { usuarioRepositoryImplement.save(usuario) }returns usuario
        val res = usuarioController.save(usuario)
        assert(res == usuario)
        verify(exactly = 1) {usuarioRepositoryImplement.save(usuario)  }
    }
    @Test
    fun deleteUsuario(){
        every { usuarioRepositoryImplement.delete(usuario) }returns true
        val res = usuarioController.delete(usuario)
        assert(res)
        verify(exactly = 1) {usuarioRepositoryImplement.delete(usuario)  }
    }
    @Test
    fun deleteNoExisteUsuario(){
        every { usuarioRepositoryImplement.delete(usuario) }returns false
        val res = usuarioController.delete(usuario)
        assert(!res)
        verify(exactly = 1) {usuarioRepositoryImplement.delete(usuario)  }
    }
}
```

5.2. Exposed

5.2.1. Modelos

En el paquete de modelos podemos encontrar todas las clases con las que trabajamos en la aplicación, dentro de este paquete también podemos encontrar un paquete de enumns que son aquellos para tipificar datos dentro de una clase. A continuación podemos ver la explicación y código de un modelo y un enum, ya que todos son iguales.

A continuación podemos ver la clase Usuario, para crear la clase la creamos con un data class y los datos correspondientes a la clase.

```
data class Usuario(
    val id:Int,
    val uuid: UUID,
    val nombre: String,
    val apellido: String,
    val correo: String,
    val password: String,
    val tipoUsuario: TipoUsuario
)
```

Ahora comprobamos el código de un enum sí lo comprobamos con los enumns que creamos en la práctica con JPA es el mismo código no hay variaciones.

```
enum class TipoUsuario(valor: String){
    USUARIO("USUARIO"),
    ADMINISTRADOR("ADMINISTRADOR"),
    ENCORDADOR("ENCORDADOR");
    companion object {
        fun from(tipoUsuario: String): TipoUsuario {
            return when (tipoUsuario.uppercase()) {
                "USUARIO" -> USUARIO
                "ADMINISTRADOR" -> ADMINISTRADOR
                "ENCORDADOR" -> ENCORDADOR
                else -> throw IllegalArgumentException("Usuario no
reconocido")
            }
        }
    }
}
```

5.2.2. DTO

En este paquete almacenamos todos los ficheros necesarios y relacionados con los dto de cada modelo, para poder serializar y convertir cada modelo en un JSON, XML, String, es necesario utilizar estás clases que nos permitan convertir una clase de Kotlin a un JSON por ejemplo. Para ello utilizamos la anotación @Serializable que es de Kotlin y nos permite convertir una clase normal en un JSON por ejemplo. Y también la anotación @SerialName con ella indicamos el nombre del objeto que estamos serializando y que da como resultado a la serialización.

```
@Serializable
@SerialName("Pedido")
data class PedidoDTO(
    val id: Int,
    val uuid: String,
    val estado: String,
    val fechaEntrada: String,
    val fechaSalidaProgramada: String,
    val fechaEntrega: String,
    val precio: Double,
```

```
    val usuario: String,
)

/**
 * To d t o
 *
 * @return Objeto en dto.
 */
fun Pedido.toDTO(): PedidoDTO {
    return PedidoDTO(
        id = id,
        uuid=uuid.toString(),
        estado=estado.toString(),
        fechaEntrada=fechaEntrada.toString(),
        fechaSalidaProgramada=fechaSalidaProgramada.toString(),
        fechaEntrega=fechaEntrega.toString(),
        precio=precio,
        usuario = usuario.toString()
    )
}
```

5.2.3. DB

Encontramos en primer lugar el fichero data donde tenemos varios métodos, tantos como modelos hemos creado en nuestro proyecto, en cada método estamos creando varios modelos por ejemplo de máquinas, turnos, usuarios, tareas, pedidos … Para que cuando iniciemos la base de datos ya tengamos unos datos cargados de ejemplo y podamos operar y ver el comportamiento.

```
fun getUsuariosInit() = listOf(
    Usuario(
        id = 0,
        uuid = UUID.randomUUID(),
        nombre = "Alfonso",
        apellido = "Cabello",
        correo = "alfonso@cabello.com",
        password = "1234",
        tipoUsuario = TipoUsuario.USUARIO
    ),
    Usuario(
        id = 1,
        uuid = UUID.randomUUID(),
        nombre = "Marcelo",
        apellido = "Alvarez",
        correo = "marcelo@alvarez.com",
        password = "1234",
        tipoUsuario = TipoUsuario.ADMINISTRADOR
    ),
    Usuario(
        id = 2,
        uuid = UUID.randomUUID(),
        nombre = "Fernando",
        apellido = "Alonso",
        correo = "fernado@alonso.com",
        password = "1234",
        tipoUsuario = TipoUsuario.ENCORDADOR
    ),
    Usuario(
        id = 3,
        uuid = UUID.randomUUID(),
        nombre = "Cristiano",
        apellido = "Ronaldo",
        correo = "cristiano@ronaldo.com",
        password = "1234",
        tipoUsuario = TipoUsuario.USUARIO
```

```
    ),
    Usuario(
        id = 4,
        uuid = UUID.randomUUID(),
        nombre = "Rafael",
        apellido = "Nadal",
        correo = "rafael@nadal.com",
        password = "1234",
        tipoUsuario = TipoUsuario.ADMINISTRADOR
    )
)

/**
 * Get productos init
 *
 */
fun getProductosInit() = listOf(
    Producto(
        id = 0,
        uuid = UUID.randomUUID(),
        marca = "Nike",
        modelo = "H10",
        precio = 50.45,
        stock = 120,
        tipoProducto = TipoProducto.CORDAJE,
        pedido = getPedidosInit()[0]
    ),
    Producto(
        id = 1,
        uuid = UUID.randomUUID(),
        marca = "Nike",
        modelo = "H10",
        precio = 50.45,
        stock = 120,
        tipoProducto = TipoProducto.CORDAJE,
        pedido = getPedidosInit()[1]
    ),
    Producto(
        id = 2,
        uuid = UUID.randomUUID(),
        marca = "Nike",
        modelo = "H10",
        precio = 50.45,
        stock = 120,
        tipoProducto = TipoProducto.CORDAJE,
        pedido = getPedidosInit()[2]
    ),
    Producto(
        id = 3,
        uuid = UUID.randomUUID(),
        marca = "Nike",
        modelo = "H10",
        precio = 50.45,
        stock = 120,
        tipoProducto = TipoProducto.CORDAJE,
        pedido = getPedidosInit()[3]
    ),
    Producto(
        id = 4,
        uuid = UUID.randomUUID(),
        marca = "Nike",
        modelo = "H10",
        precio = 50.45,
        stock = 120,
        tipoProducto = TipoProducto.CORDAJE,
```

```
        pedido = getPedidosInit()[4]
    ),
)

/**
 * Get pedidos init
 *
 */
fun getPedidosInit() = listOf(
    Pedido(
        id = 0,
        uuid = UUID.randomUUID(),
        estado = TipoEstado.EN_PROCESO,
        fechaEntrada = LocalDate.of(2022, 3, 15),
        fechaSalidaProgramada = LocalDate.of(2022, 5, 10),
        fechaEntrega = LocalDate.of(2022, 6, 7),
        precio = 10.20,
        usuario = getUsuariosInit()[0]
    ),
    Pedido(
        id = 1,
        uuid = UUID.randomUUID(),
        estado = TipoEstado.RECIBIDO,
        fechaEntrada = LocalDate.of(2022, 2, 15),
        fechaSalidaProgramada = LocalDate.of(2022, 6, 10),
        fechaEntrega = LocalDate.of(2022, 7, 10),
        precio = 10.20,
        usuario = getUsuariosInit()[1]
    ),

    Pedido(
        id = 2,
        uuid = UUID.randomUUID(),
        estado = TipoEstado.TERMINADO,
        fechaEntrada = LocalDate.of(2022, 4, 15),
        fechaSalidaProgramada = LocalDate.of(2022, 9, 10),
        fechaEntrega = LocalDate.of(2022, 10, 10),
        precio = 10.20,
        usuario = getUsuariosInit()[2]
    ),
    Pedido(
        id = 3,
        uuid = UUID.randomUUID(),
        estado = TipoEstado.TERMINADO,
        fechaEntrada = LocalDate.of(2022, 6, 15),
        fechaSalidaProgramada = LocalDate.of(2022, 6, 20),
        fechaEntrega = LocalDate.of(2022, 7, 10),
        precio = 10.20,
        usuario = getUsuariosInit()[3]
    ),
    Pedido(
        id = 4,
        uuid = UUID.randomUUID(),
        estado = TipoEstado.TERMINADO,
        fechaEntrada = LocalDate.of(2022, 6, 15),
        fechaSalidaProgramada = LocalDate.of(2022, 7, 10),
        fechaEntrega = LocalDate.of(2022, 8, 10),
        precio = 10.20,
        usuario = getUsuariosInit()[4]
    ),
)

/**
 * Get tareas encoradado init
 *
```

```
 */
fun getTareasEncoradadoInit() = listOf(
    TareaEncordado(
        id = 0,
        uuid = UUID.randomUUID(),
        precio = 100.50,
        tensionVertical = 22.50,
        cordajeVertical = "Babolat",
        tensionHorizontal = 26.10,
        cordajeHorizontal = "Babolat",
        nudos = NumeroNudos.DOS,
        pedido = getPedidosInit()[0],
    ),
    TareaEncordado(
        id = 1,
        uuid = UUID.randomUUID(),
        precio = 100.50,
        tensionVertical = 22.50,
        cordajeVertical = "Wilson",
        tensionHorizontal = 26.10,
        cordajeHorizontal = "Wilson",
        nudos = NumeroNudos.DOS,
        pedido = getPedidosInit()[1],
    ),
)

/**
 * Get tareas personalizacion
 *
 */
fun getTareasPersonalizacion() = listOf(
    TareaPersonalizacion(
        id = 0,
        uuid = UUID.randomUUID(),
        rigidez = 69.69,
        peso = 0.20,
        balance = 327.10,
        precio = 99.99,
        pedido = getPedidosInit()[2]
    ),
    TareaPersonalizacion(
        id = 1,
        uuid = UUID.randomUUID(),
        rigidez = 79.69,
        peso = 0.27,
        balance = 326.10,
        precio = 89.99,
        pedido = getPedidosInit()[3]
    ),
    TareaPersonalizacion(
        id = 2,
        uuid = UUID.randomUUID(),
        rigidez = 59.69,
        peso = 0.25,
        balance = 327.10,
        precio = 79.99,
        pedido = getPedidosInit()[4]
    ),
)

/**
 * Get maquinas encordar
 *
 */
fun getMaquinasEncordar() = listOf(
```

```
    MaquinaEncordar(
        id = 0,
        uuid = UUID.randomUUID(),
        marca = "HEAD",
        modelo = "2020",
        fechaAdquisicion = LocalDate.of(2022, 3, 5),
        numeroSerie = 20,
        tipo = TipoEncordaje.MANUAL,
        tensionMaxima = 25.10,
        tensionMinima = 22.60,
        turno = getTurnoInit()[0]
    ),
    MaquinaEncordar(
        id = 1,
        uuid = UUID.randomUUID(),
        marca = "HEAD",
        modelo = "2020",
        fechaAdquisicion = LocalDate.of(2022, 5, 5),
        numeroSerie = 20,
        tipo = TipoEncordaje.MANUAL,
        tensionMaxima = 26.10,
        tensionMinima = 23.60,
        turno = getTurnoInit()[0]
    ),
)

/**
 * Get maquinas personalizar
 *
 */
fun getMaquinasPersonalizar() = listOf(
    MaquinaPersonalizar(
        id = 0,
        uuid = UUID.randomUUID(),
        marca = "Signum Pro",
        modelo = "30",
        fechaAdquisicion = LocalDate.of(2022, 4, 15),
        numeroSerie = 7,
        swingweight = true,
        balance = 300.80,
        rigidez = 100.60,
        turno = getTurnoInit()[0],
    ),
    MaquinaPersonalizar(
        id = 1,
        uuid = UUID.randomUUID(),
        marca = "Solinco",
        modelo = "50",
        fechaAdquisicion = LocalDate.of(2022, 5, 15),
        numeroSerie = 9,
        swingweight = false,
        balance = 310.80,
        rigidez = 120.60,
        turno = getTurnoInit()[1],
    ),
)

/**
 * Get turno init
 *
 */
fun getTurnoInit() = listOf(
    Turno(
        id = 0,
        uuid = UUID.randomUUID(),
```

```
        fechaInicio = LocalDateTime.of(2022, 10, 14, 9, 10),
        fechaFinal = LocalDateTime.of(2022, 11, 20, 10, 20),
        usuario = getUsuariosInit()[0]
    ),
    Turno(
        id = 1,
        uuid = UUID.randomUUID(),
        fechaInicio = LocalDateTime.of(2022, 8, 14, 9, 10),
        fechaFinal = LocalDateTime.of(2022, 9, 20, 10, 20),
        usuario = getUsuariosInit()[1]
    ),
    Turno(
        id = 2,
        uuid = UUID.randomUUID(),
        fechaInicio = LocalDateTime.of(2022, 7, 14, 9, 10),
        fechaFinal = LocalDateTime.of(2022, 8, 20, 10, 20),
        usuario = getUsuariosInit()[2]
    ),
)
```

También podemos encontrar el fichero DataBaseManager que se encarga de almacenar la configuración del comportamiento entre la base de datos y el código del programa, donde también inicializamos las tablas que deseamos tener en nuestro programa. Además desde este fichero podemos borrar las tablas por si usamos alguna de test.

```
object DataBaseManager {
    lateinit var appConfig: AppConfig
    fun init(appConfig: AppConfig) {
        this.appConfig = appConfig
        logger.debug("Initializing database")
        Database.connect(
            url = appConfig.jdbcUrl,
            driver = appConfig.jdbcDriverClassName,
            user = appConfig.jdbcUserName,
            password = appConfig.jdbcPassword
        )

        logger.debug("Database initialized successfully")

        if (appConfig.jdbcCreateTables) {
            createTables()
        }
    }

    private fun createTables() = transaction {
        logger.debug("Creating tables")

        if (appConfig.jdbcshowSQL)
            addLogger(StdOutSqlLogger)

        SchemaUtils.create(
            TablaMaquinaEncordar,
            TablaMaquinaPersonalizar,
            TablaPedido,
            TablaProducto,
            TablaTareaEncordado,
            TablaTareaPersonalizacion,
            TablaTurno,
            TablaUsuario
        )
        logger.debug("Tables created")
    }

    fun dropTables() = transaction {
        logger.debug { "Eliminando tablas de la base de datos" }
```

```
        if (appConfig.jdbcshowSQL)
            addLogger(StdOutSqlLogger)

        val tables = arrayOf(
            TablaMaquinaEncordar,
            TablaMaquinaPersonalizar,
            TablaPedido,
            TablaProducto,
            TablaTareaEncordado,
            TablaTareaPersonalizacion,
            TablaTurno,
            TablaUsuario
        )

        SchemaUtils.drop(*tables)
        logger.debug { "Tablas eliminadas (${tables.size}):
${tables.joinToString { it.tableName }}" }
    }

    fun clearTables() = transaction {
        logger.debug { "clear tables" }

        if (appConfig.jdbcshowSQL)
            addLogger(StdOutSqlLogger)

        val tables = arrayOf(
            TablaMaquinaEncordar,
            TablaMaquinaPersonalizar,
            TablaPedido,
            TablaProducto,
            TablaTareaEncordado,
            TablaTareaPersonalizacion,
            TablaTurno,
            TablaUsuario
        )

        tables.forEach {
            it.deleteAll()
        }
        logger.debug { "Limpiar tablas (${tables.size}):
${tables.joinToString { it.tableName }}" }
    }
}
```

5.2.4. Exceptions

En el paquete de excepciones podemos encontrar cada una de las excepciones que van a utilizar los repositorios y controladores del proyecto. De esta forma podemos sacar un mensaje personalizado dependiendo de la excepción que se genere. A continuación podemos ver un ejemplo de excepción, el resto de las excepciones son creadas de la misma forma.

```
class MaquinaEncordarException(message: String) : RuntimeException(message)
```

5.2.5. Config

Contiene la configuración inicial para la conexión de la base de datos con el proyecto, donde recogemos datos como usuario, contraseña, url, versión, creación de tablas, para establecer la conexión con la base de datos.

```
/**
 * App config: Clase que enlaza con el config.properties y con el
DataBaseManager
```

```
 *
 * @property nombre
 * @property version
 * @property jdbcUrl
 * @property jdbcUserName
 * @property jdbcPassword
 * @property jdbcDriverClassName
 * @property jdbcMaximumPoolSize
 * @property jdbcCreateTables
 * @property jdbcshowSQL
 * @constructor Create empty App config
 */
data class AppConfig(
    val nombre: String,
    val version: String,
    val jdbcUrl: String,
    val jdbcUserName: String,
    val jdbcPassword: String,
    val jdbcDriverClassName: String,
    val jdbcMaximumPoolSize: Int = 10,
    val jdbcCreateTables: Boolean = true,
    val jdbcshowSQL: Boolean = true,
) {
    companion object {
        val DEFAULT = AppConfig(
            nombre = "app",
            version = "1.0.0",
            jdbcUrl = "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;",
            jdbcUserName = "tenista",
            jdbcPassword = "tenista",
            jdbcDriverClassName = "org.h2.Driver",
            jdbcMaximumPoolSize = 10,
            jdbcCreateTables = true,
            jdbcshowSQL = true,
        )

        /**
         * From properties file: Método que obtiene los datos de la
configuración desde el config.properties
         *
         * @param fileName
         * @return AppConfig
         */
        fun fromPropertiesFile(fileName: String): AppConfig {
            logger.debug { "Loading properties from file: $fileName" }
            val properties = Properties()
            properties.load(FileInputStream(fileName))
            return AppConfig(
                nombre = properties.getProperty("nombre"),
                version = properties.getProperty("version"),
                jdbcUrl = properties.getProperty("jdbc.url"),
                jdbcUserName = properties.getProperty("jdbc.username"),
                jdbcPassword = properties.getProperty("jdbc.password"),
                jdbcDriverClassName =
properties.getProperty("jdbc.driverClassName"),
                jdbcMaximumPoolSize =
properties.getProperty("jdbc.maximumPoolSize").toInt(),
                jdbcCreateTables =
properties.getProperty("jdbc.createTables").toBoolean(),
                jdbcshowSQL =
properties.getProperty("jdbc.showSQL").toBoolean(),
            )
        }
    }
}
```

5.2.6. Repositories

En el paquete repositories podemos encontrar cada uno de los repositorios que tiene un modelo asociado. Hemos implementado SOLID, a la hora de crear el diseño y arquitectura de los repositorios, en este caso el Principio de Responsabilidad Única.
Para ello hemos creado una interfaz genérica que comparten todos los repositorios.

```
/**
 * I c r u d repository
 *
 * @param T
 * @param ID
 * @constructor Create empty I c r u d repository
 */
interface ICRUDRepository<T, ID> {
    /**
     * Find all
     *
     * @return
     */
    fun findAll(): List<T>

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

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

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

Esta interfaz se la hemos añadido a cada interfaz de repositorio de cada modelo, sí algún modelo necesita añadir algún método diferente lo podrá hacer aquí de forma que no afecte al resto de repositorios.

```
/**
 * I usuario repository
 *
 * @constructor Create empty I usuario repository
 */
interface IUsuarioRepository : ICRUDRepository<Usuario, Int> {
}
```

A continuación podemos ver la implementación del repositorio de usuarios en este caso si analizamos en detalle el código podemos ver que implementamos la interfaz de usuariosRepository y de esta forma sus métodos. Los métodos como podemos comprobar son los de un CRUD, en cada método del repositorio llamamos a una transacción que realizamos desde el DAO de cada entidad.

```
/**
 * Usuario repository implement
 *
 * @property usuariosDAO
 * @constructor Create empty Usuario repository implement
 */
class UsuarioRepositoryImplement (
    private val usuariosDAO: IntEntityClass<UsuariosDAO>
) : IUsuarioRepository {

    private val logger = KotlinLogging.logger {}

    override fun findAll(): List<Usuario> = transaction {
        logger.debug { "findAll() - buscando todos " }
        usuariosDAO.all().map { it.fromUsuarioDAOToUsuario() }
    }


    override fun findById(id: Int): Usuario = transaction {
        logger.debug { "findById($id) - buscando $id" }
        usuariosDAO.findById(id)
            ?.fromUsuarioDAOToUsuario() ?: throw UsuarioException("Usuario no
encontrado con id: $id")
    }



    override fun save(entity: Usuario): Usuario = transaction {
        val existe = usuariosDAO.findById(entity.id)

        existe?.let {
            update(entity, existe)
        } ?: run {
            insert(entity)
        }
    }


    override fun delete(entity: Usuario): Boolean = transaction {
        val existe = usuariosDAO.findById(entity.id) ?: return@transaction false
        logger.debug { "delete($entity) - borrando" }
        existe.delete()
        true
    }


    private fun insert(entity: Usuario): Usuario {
        logger.debug { "save($entity) - creando" }
        return usuariosDAO.new(entity.id) {
            uuid = entity.uuid
            nombre = entity.nombre
            apellido = entity.apellido
            correo = entity.correo
            password = encriptar(entity.password)
            tipoUsuario = entity.tipoUsuario
        }.fromUsuarioDAOToUsuario()
    }
```

```
    private fun update(entity: Usuario, existe: UsuariosDAO): Usuario {
        logger.debug { "save($entity) - actualizando" }
        return existe.apply {
            uuid = entity.uuid
            nombre = entity.nombre
            apellido = entity.apellido
            correo = entity.correo
            password = entity.password
            tipoUsuario = entity.tipoUsuario
        }.fromUsuarioDAOToUsuario()
    }
}
```

5.2.7. Entities/DAO

Utilizamos este paquete para almacenar cada uno de los ficheros correspondiente a las entidades asociadas a cada modelo de nuestra aplicación. Debido a que la BBDD no reconoce algún tipo de dato que sí tiene nuestro modelo, creamos estos objetos intermedios entre el modelo y la base de datos que nos permiten convertir una clase en un objeto reconocible por la base de datos y un objeto de la base de datos en una clase. Esta clase ya la hemos utilizado en los repositorios para pode almacenar los datos de las operaciones CRUD.

```
object TablaUsuario : IntIdTable("USUARIO") {
    val uuid = uuid("uuid").uniqueIndex()
    val nombre = varchar("nombre", 50)
    val apellido = varchar("apellido", 50)
    val correo = varchar("correo", 50)
    val password = varchar("contrasena", 255)
    val tipoUsuario = enumeration<TipoUsuario>("perfil")
}

/**
 * Usuarios d a o
 *
 * @constructor
 *
 * @param id
 */
class UsuariosDAO(id: EntityID<Int>) : IntEntity(id) {
    companion object : IntEntityClass<UsuariosDAO>(TablaUsuario)

    var uuid by TablaUsuario.uuid
    var nombre by TablaUsuario.nombre
    var apellido by TablaUsuario.apellido
    var correo by TablaUsuario.correo
    var password by TablaUsuario.password
    var tipoUsuario by TablaUsuario.tipoUsuario
}
```

5.2.8. Controllers

Los controladores se encargan de gestionar cada una de las acciones que tiene que hacer nuestro programa, nosotros hemos decidido crear un controlador por modelo, en vez de tenerlos todos en uno, así de esta forma nos podemos asegura que cada controlador solo controla su modelo correspondiente.

En este caso vemos el controlador de Usuarios, podemos ver que gestiona las principales operaciones CRUD del repositorio. Ya que el controlador se encarga de la lógica tendrá que llevar a cabo las conversiones de DAO y Objetos llamando a cada DAO que se encargue de su asociado.

```
class UsuariosController (
    private val usuarioRepositoryImplement: UsuarioRepositoryImplement =
```

```
UsuarioRepositoryImplement(UsuariosDAO)
) {

    /**
     * Find all
     *
     * @return
     */
    fun findAll(): List<Usuario> {
        return usuarioRepositoryImplement.findAll()
    }

    /**
     * Find by id
     *
     * @param id
     * @return
     */
    fun findById(id: Int): Usuario {
        return usuarioRepositoryImplement.findById(id)
    }

    /**
     * Save
     *
     * @param usuario
     * @return
     */
    fun save(usuario: Usuario): Usuario {
        return usuarioRepositoryImplement.save(usuario)
    }

    /**
     * Delete
     *
     * @param usuario
     * @return
     */
    fun delete(usuario: Usuario): Boolean {
        return usuarioRepositoryImplement.delete(usuario)
    }
}
```

5.2.9. Mappers

Los mapas se encargan de mapear cada una de las clases que tenemos y de esta forma podemos pasar de DAO→Object y de Object→DAO, recorre cada una de las clases leyendo los datos para poder ejecutar su DAO correspondiente.

```
/**
 * From usuario d a o to usuario
 *
 * @return El objeto ya en modelo.
 */
fun UsuariosDAO.fromUsuarioDAOToUsuario(): Usuario {
    return Usuario(
        id = id.value,
        uuid = uuid,
        nombre = nombre,
        apellido = apellido,
        correo = correo,
        password = password,
        tipoUsuario = tipoUsuario,
    )
}
```

5.2.10. Serializers

El paquete de Serializers se encarga de almacenar aquellos datos que nos ha solicitado el cliente en un fichero JSON, de esta forma exportamos los datos almacenados en la base de datos en un fichero JSON que por ejemplo en un futuro podríamos crear la funcionalidad de leer ficheros JSON y que fuese otra forma de almacenar datos en la base de datos.

```
class StorageJSON {

    /**
     * Turno j s o n: Este método sirve para pasar los datos de turno en un
fichero json y se almacenará en resources.
     *
     * @param nombreArchivo
     * @param turnoDTO
     */
    fun turnoJSON(nombreArchivo: String, turnoDTO: List<TurnoDTO>) {
        logger.info("Write Json turno.")
        val directorio = System.getProperty("user.dir") +
                File.separator + "src" +
                File.separator + "main" +
                File.separator + "resources"
        val fichero = File(directorio + File.separator +
"$nombreArchivo.json")
        val json = Json { prettyPrint = true }
        fichero.writeText(json.encodeToString(turnoDTO))
    }

    /**
     * Pedido j s o n: Este método sirve para pasar los datos de pedido en un
fichero json y se almacenará en resources.
     *
     * @param nombreArchivo
     * @param pedidoDTO
     */
    fun pedidoJSON(nombreArchivo: String, pedidoDTO: List<PedidoDTO>) {
        logger.info("Write Json Pedido.")
        val directorio = System.getProperty("user.dir") +
                File.separator + "src" +
                File.separator + "main" +
                File.separator + "resources"
        val fichero = File(directorio + File.separator +
"$nombreArchivo.json")
        val json = Json { prettyPrint = true }
        fichero.writeText(json.encodeToString(pedidoDTO))
    }

    /**
     * Producto json: Este método sirve para pasar los datos de producto en
un fichero json y se almacenará en resources.
     *
     * @param nombreArchivo
     * @param productoDTO
     */
    fun productoJson(nombreArchivo: String, productoDTO: List<ProductoDTO>) {
        logger.info("Write Json Producto.")
        val directorio = System.getProperty("user.dir") +
                File.separator + "src" +
                File.separator + "main" +
                File.separator + "resources"
        val fichero = File(directorio + File.separator +
"$nombreArchivo.json")
        val json = Json { prettyPrint = true }
```

```
            fichero.writeText(json.encodeToString(productoDTO))
        }
    }
```

5.2.11. Services

En el paquete Services podemos encontrar una clase que se encarga de encriptar la contraseña de los usuarios implementando el algoritmo de cifrado sha256.

```
class EncriptarPassword {
    fun encriptar(contra: String): String {
        return Hashing.sha256()
            .hashString(contra, StandardCharsets.UTF_8)
            .toString()
    }
}
```

5.2.12. Main

En nuestro main podemos ver todo el código relacionado con la ejecución y lógica del programa, como podemos ver cargamos datos, nos loggeamos, y llamando a los controladores podemos realizar cada una de las operaciones CRUD de cada uno de los modelos que hemos creado.

```
fun main() {
    println("TennisLab")
    initDataBase()
    val maquinasControllers = MaquinasController()
    val pedidosControllers = PedidosController()
    val productosController = ProductosController()
    val tareasController = TareasController()
    val turnosController = TurnosController()
    val usuariosController = UsuariosController()
    val serviceJSON = StorageJSON()

    getUsuariosInit().forEach { usuario -> usuariosController.save(usuario) }
    getTurnoInit().forEach { turno -> turnosController.save(turno) }
    getPedidosInit().forEach { pedido -> pedidosControllers.save(pedido) }
    getProductosInit().forEach { producto -> productosController.save(producto) }
    getTareasEncoradadoInit().forEach { tarea ->
tareasController.saveTareaEncordado(tarea) }
    getTareasPersonalizacion().forEach { tarea ->
tareasController.saveTareaPersonalizacion(tarea) }
    getMaquinasEncordar().forEach { maquina ->
maquinasControllers.saveMaquinaEncordar(maquina) }
    getMaquinasPersonalizar().forEach { maquina ->
maquinasControllers.saveMaquinaPersonalizar(maquina) }
    val listAsignacion = turnosController
        .findAll()
        .sortedBy { it.fechaInicio }
        .map { it.toDTO() }
    serviceJSON.turnoJSON("UserAsignaciónByDate", listAsignacion)
    val pedido = pedidosControllers.findByID(2).toDTO()
    serviceJSON.pedidoJSON("AllPedidoData", listOf(pedido))
    val allProductos = productosController
        .findAll()
        .map { it.toDTO() }
    serviceJSON.productoJson("productos", allProductos)
    val pedidosSinFinalizar = pedidosControllers
        .findAll()
        .filter { it.estado == TipoEstado.EN_PROCESO }
        .map { it.toDTO() }
    serviceJSON.pedidoJSON("pedidosSinFinalizar", pedidosSinFinalizar)

    val pedidosFinalizados = pedidosControllers
        .findAll()
        .stream()
```

```
        .filter { it.estado == TipoEstado.TERMINADO }
        .map { it.toDTO() }.toList()
    serviceJSON.pedidoJSON("pedidosFinalizados", pedidosFinalizados)




}
fun initDataBase() {
    val appConfig =
AppConfig.fromPropertiesFile("src/main/resources/config.properties")
    println("Configuración: $appConfig")

    DataBaseManager.init(appConfig)

}
```

5.2.13. Resources/ Config.properties

En este fichero podemos encontrar todos los string de conexión con la base de datos que después serán leídos por el programa a través de los ficheros que hemos creado anteriormente desde el Config o desde db.

```
nombre=TennisLab Dam
version=1.0
jdbc.url=jdbc:h2:mem:tenistas;DB_CLOSE_DELAY=-1;
jdbc.driverClassName=org.h2.Driver
jdbc.username=tenista
jdbc.password=tenista
jdbc.maximumPoolSize=10
jdbc.createTables=true
jdbc.showSQL=false
```

5.2.14. Test Controllers

Para hacer los test de los repositorios hemos utilizado Mockk para Kotlin de esta forma podemos comprobar que cada uno de los métodos que hemos implementado en el controlador funciona tal y como nos piden las especificaciones, por cada controlador hacemos su correspondiente test llamando al controlador y cargando unos datos de prueba.

```
@ExtendWith(MockKExtension::class)
class UsuariosControllerTest {

    @MockK
    lateinit var usuarioRepositoryImplement: UsuarioRepositoryImplement

    @InjectMockKs
    lateinit var usuariosController: UsuariosController

    private val usuario= Usuario(
        id = 1,
        uuid = UUID.randomUUID(),
        nombre = "Marcelo",
        apellido = "Alvarez",
        correo = "marcelo@alvarez.com",
        password = "1234",
        tipoUsuario = TipoUsuario.ADMINISTRADOR
    )

    init {
        MockKAnnotations.init(this)
    }

    @Test
    fun findAllUsuarios() {
```

```
        every { usuarioRepositoryImplement.findAll() } returns listOf(usuario)
        val res = usuariosController.findAll()
        assert(res == listOf(usuario))
        verify(exactly = 1) { usuarioRepositoryImplement.findAll() }
    }

    @Test
    fun findByIdUsuario() {
        every { usuarioRepositoryImplement.findById(usuario.id) } returns usuario
        val res = usuariosController.findById(usuario.id)
        assert(res == usuario)
        verify(exactly = 1) { usuarioRepositoryImplement.findById(usuario.id) }

    }

    @Test
    fun findByIdNoExisteUsuario() {
        every { usuarioRepositoryImplement.findById(usuario.id) } throws
UsuarioException("Error: No encontrado usuario con id: ${usuario.id}")
        val res = assertThrows<UsuarioException> {
            usuariosController.findById(usuario.id)
        }
        assert(res.message == "Error: No encontrado usuario con id:
${usuario.id}")
        verify(exactly = 1) { usuarioRepositoryImplement.findById(usuario.id) }
    }

    @Test
    fun saveUsuario() {
        every { usuarioRepositoryImplement.save(usuario) } returns usuario
        val res = usuariosController.save(usuario)
        assert(res == usuario)
        verify(exactly = 1) { usuarioRepositoryImplement.save(usuario) }
    }

    @Test
    fun deleteUsuario() {
        every { usuarioRepositoryImplement.delete(usuario) } returns true
        val res = usuariosController.delete(usuario)
        assert(res)
        verify(exactly = 1) { usuarioRepositoryImplement.delete(usuario) }
    }

    @Test
    fun deleteNoExiste() {
        every { usuarioRepositoryImplement.delete(usuario) } throws
UsuarioException("Error: No encontrado usuario con id: ${usuario.id}")
        val res = assertThrows<UsuarioException> {
usuariosController.delete(usuario) }
        assert(res.message == "Error: No encontrado usuario con id:
${usuario.id}")
        verify(exactly = 1) { usuarioRepositoryImplement.delete(usuario) }
    }

}
```

5.2.15. Test Repositories

Para hacer los test de los repositorios hemos utilizado JUNIT de esta forma podemos comprobar que cada uno de los métodos que hemos implementado en el repositorio funciona tal y como nos piden las especificaciones, por cada repositorio hacemos su correspondiente test llamando al repositorio y cargando unos datos de prueba.

```kotlin
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class UsuarioRepositoryImplementTest {
    private val usuariosRepository = UsuarioRepositoryImplement(UsuariosDAO)

    private val usuario = Usuario(
        id = 0,
        uuid = UUID.randomUUID(),
        nombre = "Alfonso",
        apellido = "Cabello",
        correo = "alfonso@cabello.com",
        password = "1234",
        tipoUsuario = TipoUsuario.USUARIO
    )

    @BeforeAll
    fun setUp() {
        DataBaseManager.init(AppConfig.DEFAULT)
    }

    @AfterAll
    fun tearDown() {
        DataBaseManager.dropTables()
    }

    @BeforeEach
    fun beforeEach() {
        DataBaseManager.clearTables()
    }

    @Test
    fun findAll() {
        val res = usuariosRepository.findAll()

        assert(res.isEmpty())
    }

    @Test
    fun findById() = transaction {
        UsuariosDAO.new(usuario.id) {
            uuid = usuario.uuid
            nombre = usuario.nombre
            apellido = usuario.apellido
            correo = usuario.correo
            password = usuario.password
            tipoUsuario = usuario.tipoUsuario

        }

        val res = usuariosRepository.findById(usuario.id)

        assert(res == usuario)
    }

    @Test
    fun findByIdNoExiste() {
        assertThrows<UsuarioException> {
            val res = usuariosRepository.findById(-5)
        }

    }

    @Test
    fun saveInsert() {
        val res = usuariosRepository.save(usuario)
```

```
        assertAll(
            { assertEquals(res.id, usuario.id) },
            { assertEquals(res.uuid, usuario.uuid) },
            { assertEquals(res.nombre, usuario.nombre) },
            { assertEquals(res.apellido, usuario.apellido) },
            { assertEquals(res.correo, usuario.correo) },
            { assertEquals(res.tipoUsuario, usuario.tipoUsuario) },
        )
    }

    @Test
    fun saveUpdate() = transaction {
        UsuariosDAO.new(usuario.id) {
            uuid = usuario.uuid
            nombre = usuario.nombre
            apellido = usuario.apellido
            correo = usuario.correo
            password = usuario.password
            tipoUsuario = usuario.tipoUsuario
        }

        val res = usuariosRepository.save(usuario)

        assert(res == usuario)
    }

    @Test
    fun delete() = transaction {
        UsuariosDAO.new(usuario.id) {
            uuid = usuario.uuid
            nombre = usuario.nombre
            apellido = usuario.apellido
            correo = usuario.correo
            password = usuario.password
            tipoUsuario = usuario.tipoUsuario
        }

        val res = usuariosRepository.delete(usuario)

        assert(res)
    }

    @Test
    fun deleteNoExiste() {
        val res = usuariosRepository.delete(usuario)

        assert(!res)
    }
}
```

6. Enlace al vídeo y Proyecto

Enlace al proyecto →

https://github.com/idanirf/AD-P2-jsanchez-drodriguez-amaldonado

Enlace al video → **https://www.youtube.com/watch?v=0_qt9_vKdUE**

7. Referencias y librerías utilizadas

- MockK
  https://mockk.io/
- Serialization
  https://kotlinlang.org/docs/serialization.html
- H2
  https://mvnrepository.com/artifact/com.h2database/h2
- Guava
  https://github.com/google/guava

Jorge Sánchez Berrocoso
Daniel Rodriguez Fernandez
Alfredo Maldonado Pertuz

Damos por finalizada y entregada la práctica 02 de Acceso a datos en,
Madrid a 18 de Febrero de 2023 a las 16:30:00.