Documentación práctica de Acceso a Datos TennisLab:

Ivan Azagra Troya y Daniel Rodríguez Muñoz

Contenido

Dependencias:

- MockK: Hemos decidido utilizar esta dependencia para realizar los test unitarios de los controladores, mockeandolos para comprobar su correcta función.
- Kotlinx-coroutines-core: Esta dependencia es utilizada dentro del proyecto para poder realizar la aplicación con funciones concurrentes con la base de datos usando corrutinas para ello.
- Exposed-jdbc: Dependencia usada para la conexión con la base de datos
- Exposed-java-time: Dependencia para el uso de fechas para la base de datos
- H2database: Driver para el uso de la base de datos, estamos usando H2
- Gson: Dependencia utilizada para la creación de JSON
- HikariCP: Dependencia utilizada para poder trabajar concurrentemente
- Kotlin-logging: Dependencia utilizada para poder dejar mensajes debug en el proceso de desarrollo de la aplicación
- Logback-classic: Dependencia de logs que funciona junto a Kotlin-logging
- Junit-jupiter-api: Dependencia base para el desarrollo de test
- Junit-jupiter-engine: Dependencia para los test de integración.

Carpetas:

Config:

- Config.kt: Esta clase se encarga de pasar los datos necesarios para conectar y acceder a la base de datos usando para ellos los parámetros de nombre, versión, url, username, password, en la configuración también podemos establecer el máximo de conexiones, si puede crear nuevas tablas, el nombre del driver de la base de datos y si puede mostrar las queries.
  Tenemos una función dedicada a pasarle estas propiedades a través de una función que cargue el archivo indicado.

Controllers:

- AdquisicionController.kt: Esta clase sirve para realizar las acciones pasadas desde el repositorio de Adquisición devolviendo el valor que debe o un mensaje por defecto en caso de que no encuentre el resultado deseado, se compone de varias funciones: findAllAdquisiciones devuelve una cadena de texto con todas las adquisiciones a través de la clase AdquisicionesService.

```
suspend fun findAllAdquisiciones(): String {
    return service.getAllAdquisiciones().toString()
```

Con la función getAdquisicionById se le pasa por parámetro un uuid con el que en caso de existir se le pasará el valor devuelto por la clase service junto al repositorio de adquisiciones en el que se mapea en caso de existir a json y en caso de no existir se devuelve un mensaje por defecto.

```
suspend fun getAdquisicionById(id: UUID): String {
    return service.getAdquisicionById(id)?.toJSON() ?: "Adquisicion with id $id not found."
```

La función insertAdquisición recibe por parámetro un objeto dto que servirá para pasar los valores necesarios a service para la creación de la Adquisición a través del repositorio y después se usará para generar un json del valor añadido.

```
suspend fun insertAdquisicion(dto: AdquisicionDTO): String {
    return service.createAdquisicion(dto).toJSON()
```

deleteAdquisición recibe de nuevo un objeto dto con el que se pasan los valores que eliminarán de los datos guardados. Se hace una comprobación de si se ha borrado o no el dato, en caso afirmativo devuelve el objeto dto en JSON, en caso negativo devolverá un mensaje de error por defecto.

```
suspend fun deleteAdquisicion(dto: AdquisicionDTO): String {
    val result = service.deleteAdquisicion(dto)
    return if (result) dto.toJSON()
    else "Could not delete Adquisicion with id ${dto.id}"
```

- EncordadoController: Esta clase sirve para realizar las acciones junto al repositorio, se le pasa por inyección de dependencias.
  La función findAllEncordados devuelve en una cadena de texto los valores devueltos haciendo uso de la clase service inyectada que contiene todas las

funciones del repositorio mappeadas para devolver el valor esperado devolviendo una lista de encordados pasados a cadena de texto.

```
suspend fun findAllEncordados(): String {
    return service.getAllEncordados().toString()
```

La función getEncordadoById necesita que se le pase por parámetro un identificador, en el caso de esta aplicación es necesario un UUID, si el uuid introducido existe se devolverá a través del servicio el encordado en formato JSON y en caso contrario se devolverá un mensaje por defecto indicando el id introducido

```
suspend fun getEncordadoById(id: UUID): String {
    return service.getEncordadoById(id)?.toJSON() ?: "Encordado with id $id not found."
```

La función insertEncordado requiere que se le pasen los datos por parámetro en lo que será un dto que será convertido e insertado con la función del servicio designada para la creación de encordados y con la cual se obtiene de paso el objeto insertado en formato JSON.

```
suspend fun insertEncordado(dto: EncordadoDTO): String {
    return service.createEncordado(dto).toJSON()
```

La función deleteEncordado requiere que se le pase por parámetro el objeto dto que será eliminado, se comprueba si ha sido eliminado o no a través de service con su función deleteEncordado que devolverá verdadero o falso y se introducirá en la variable del resultado la cuál después es comprobada para realizar una acción u otra, en caso verdadero devolverá el objeto dto pasado por parámetro en formato JSON y en caso negativo devuelve un mensaje por defecto avisando que el objeto dto mostrando solo su id no ha podido ser eliminado.

```
suspend fun deleteEncordado(dto: EncordadoDTO): String {
    val result = service.deleteEncordado(dto)
    return if (result) dto.toJSON()
    else "Could not delete Encordado with id ${dto.id}"
```

- EncordadoraController:

  La función findAllEncordadoras devuelve en una cadena de texto los valores devueltos haciendo uso de la clase service inyectada que contiene todas las

funciones del repositorio mappeadas para devolver el valor esperado devolviendo una lista de encordadoras pasadas a cadena de texto.

```
suspend fun findAllEncordadoras(): String {
    return service.getAllEncordadoras().toString()
```

La función findAllManuales pide por parámetro un boolean que servirá para devolver las máquinas que sean manuales o no a través de la función del service.getAllEncordados y la utilización de un filter para devolver solo un tipo de máquina, estas máquinas serán las que tengan el parámetro isManual igual al parámetro.

```
suspend fun findAllManuales(bool: Boolean): String {
    return service.getAllEncordadoras().filter { it.isManual == bool }.toString()
```

La función getEncordadoraById necesita que se le pase por parámetro un identificador, en el caso de esta aplicación es necesario un UUID, si el uuid introducido existe se devolverá a través del servicio el encordado en formato JSON y en caso contrario se devolverá un mensaje por defecto indicando el id introducido

```
suspend fun getEncordadoraById(id: UUID): String {
    return service.getEncordadoraById(id)?.toJSON() ?: "Encordadora with id $id not found."
```

La función insertEncordadora requiere que se le pasen los datos por parámetro en lo que será un dto que será convertido e insertado con la función del servicio designada para la creación de encordados y con la cual se obtiene de paso el objeto insertado en formato JSON.

```
suspend fun insertEncordadora(dto: EncordadoraDTO): String {
    return service.createEncordadora(dto).toJSON()
```

La función deleteEncordadora requiere que se le pase por parámetro el objeto dto que será eliminado, se comprueba si ha sido eliminado o no a través de service con su función deleteEncordadora que devolverá verdadero o falso y se introducirá en la variable del resultado la cuál después es comprobada para realizar una acción u otra, en caso verdadero devolverá el objeto dto pasado por parámetro en formato JSON y en caso negativo devuelve un mensaje por defecto avisando que el objeto dto mostrando solo su id no ha podido ser eliminado.

```
suspend fun deleteEncordadora(dto: EncordadoraDTO): String {
    val result = service.deleteEncordadora(dto)
    return if (result) dto.toJSON()
    else "Could not delete Encordadora with id ${dto.id}"
```

- MaquinaController:

  Esta clase utiliza por inyección los servicios de las Personalizadoras y las Encordadoras.

  La función findAllMaquinas devuelve en una cadena de texto los valores devueltos haciendo uso de la clase service inyectada que contiene todas las funciones del repositorio mappeadas para devolver el valor esperado devolviendo una lista de máquinas pasadas a cadena de texto.

```
suspend fun findAllMaquinas(): String {
    val personalizadoras = pService.getAllPersonalizadoras()
    val encordadoras = eService.getAllEncordadoras()
    val maquinas: MutableList<MaquinaDTO> = mutableListOf()
    personalizadoras.forEach { maquinas.add(it) }
    encordadoras.forEach { maquinas.add(it) }
    return maquinas.toList().toString()
```

  La función getMaquinaById necesita que se le pase por parámetro un identificador, en el caso de esta aplicación es necesario un UUID, si el uuid introducido existe se devolverá a través del servicio el encordado en formato JSON y en caso contrario se devolverá un mensaje por defecto indicando el id introducido

```
suspend fun getMaquinaById(id: UUID): String {
    var busqueda: MaquinaDTO? = pService.getPersonalizadoraById(id)
    return if (busqueda != null) {
        PersonalizadoraController.getPersonalizadoraById(id)
    } else {
        busqueda = eService.getEncordadoraById(id)
        if (busqueda != null) {
            EncordadoraController.getEncordadoraById(id)
        } else {
            "Maquina with id $id not found."
```

  La función insertMaquina requiere que se le pasen los datos por parámetro en lo que será un dto que será convertido al tipo de Máquina que se haya pasado, sea esta Personalizadora o Encordadora, con lo que llama a el controlador de dicha Máquina y ejecuta la función insert de la máquina identificada, en caso de no ser reconocida devuelve un mensaje de error por defecto.

```
suspend fun insertMaquina(dto: MaquinaDTO): String {
    return when (dto) {
        is PersonalizadoraDTO -> PersonalizadoraController.insertPersonalizadora(dto)
        is EncordadoraDTO -> EncordadoraController.insertEncordadora(dto)
        else -> "Error at MaquinaController.insertMaquina: DTO not supported."
    }
}
```

La función deleteMaquina requiere que se le pase por parámetro el objeto dto que será eliminado se comprueba el tipo de máquina para ejecutar la función correspondiente al tipo de máquina que se haya pasado o en caso de que no sea reconocido el tipo de máquina devolverá un mensaje de error por defecto

```
suspend fun insertMaquina(dto: MaquinaDTO): String {
    return when (dto) {
        is PersonalizadoraDTO -> PersonalizadoraController.insertPersonalizadora(dto)
        is EncordadoraDTO -> EncordadoraController.insertEncordadora(dto)
        else -> "Error at MaquinaController.insertMaquina: DTO not supported."
    }
}
```

La función getMaquinaBySerialNumber requiere que se la pase por parámetro una cadena de texto, dentro de la función se crean 3 valores que serán obtenidos a través de el servicio junto a la función getAll(Maquina) con un filtrado de datos que comprueba si el valor de del atributo numero de serie es igual al parámetro pasado a la función. Por cada resultado en el valor se añade a una lista en la que después se filtra para devolver solo el primer valor de esta, de esta manera se evita realizar queries personalizadas para la base de datos, en caso de ser una lista vacía devuelve un error por defecto y en caso de haber un resultado se devuelve una cadena de texto en formato JSON con los valores de la máquina con el número de serie pasado.

```
suspend fun getMaquinaBySerialNumber(sNum: String): String {
    val personalizadoras = pService.getAllPersonalizadoras().filter { it.numeroSerie.contentEquals(sNum) }
    val encordadoras = eService.getAllEncordadoras().filter { it.numeroSerie.contentEquals(sNum) }
    val maquinas: MutableList<MaquinaDTO> = mutableListOf()
    personalizadoras.forEach { maquinas.add(it) }
    encordadoras.forEach { maquinas.add(it) }
    return maquinas.toList().firstOrNull()?.toString() ?: "Maquina with serial number $sNum not found."
}
```

La función getMaquinaBySerialNumberForCreation requiere que se le pase por parámetro una cadena de texto que será el número de serie. Dentro de la función se crean 3 valores que serán obtenidos a través del servicio junto a la

función getAll(Maquina) con un filtrado de datos que comprueba si el valor de del atributo número de serie es igual al parámetro pasado a la función. Por cada resultado en el valor se añade a una lista en la que después se filtra para devolver solo el primer valor de esta, con esto devolvemos una MaquinaDTO con los datos encontrados de la máquina con el número de serie introducido y en caso de no existir daría null.

```
suspend fun getMaquinaBySerialNumberForCreation(sNum: String): MaquinaDTO? {
    val encordadoras = eService.getAllEncordadoras().filter { it.numeroSerie == sNum }
    val personalizadoras = pService.getAllPersonalizadoras().filter { it.numeroSerie == sNum }
    val maquinas: MutableList<MaquinaDTO> = mutableListOf()
    personalizadoras.forEach { maquinas.add(it) }
    encordadoras.forEach { maquinas.add(it) }
    return maquinas.toList().firstOrNull()
}
```

La función getMaquinaByModel requiere que se le pase por parámetro una cadena de texto que será el modelo. Dentro de la función se crean 3 valores que serán obtenidos a través del servicio junto a la función getAll(Maquina) con un filtrado de datos que comprueba si el valor de del atributo número de serie es igual al parámetro pasado a la función. Por cada resultado en el valor se añade a una lista en la que después se filtra para devolver solo el primer valor de esta, con esto devolvemos una cadena de texto con formato JSON con los datos de las máquinas con ese modelo.

```
suspend fun getMaquinaByModel(model: String): String {
    val personalizadoras = pService.getAllPersonalizadoras().filter { it.modelo.contentEquals(model) }
    val encordadoras = eService.getAllEncordadoras().filter { it.modelo.contentEquals(model) }
    val maquinas: MutableList<MaquinaDTO> = mutableListOf()
    personalizadoras.forEach { maquinas.add(it) }
    encordadoras.forEach { maquinas.add(it) }
    return maquinas.toList().toString()
}
```

La función getMaquinaByBrand requiere que se le pase por parámetro una cadena de texto que será la marca. Dentro de la función se crean 3 valores que serán obtenidos a través del servicio junto a la función getAll(Maquina) con un filtrado de datos que comprueba si el valor de del atributo número de serie es igual al parámetro pasado a la función. Por cada resultado en el valor se añade a una lista en la que después se filtra para devolver solo el primer valor de esta,

con esto devolvemos una cadena de texto con formato JSON con los datos de las máquinas con esa marca.

```
suspend fun getMaquinaByBrand(brand: String): String {
    val personalizadoras = pService.getAllPersonalizadoras().filter { it.marca.contentEquals(brand) }
    val encordadoras = eService.getAllEncordadoras().filter { it.marca.contentEquals(brand) }
    val maquinas: MutableList<MaquinaDTO> = mutableListOf()
    personalizadoras.forEach { maquinas.add(it) }
    encordadoras.forEach { maquinas.add(it) }
    return maquinas.toList().toString()
}
```

La función findAllMaquinasByAcquisitionDate requiere que se le pase por parámetro la fecha deseada y el operador, siendo este el valor “>” o “<” y en caso de no ser ninguno de estos se devolverán listas vacías.

En caso de haber introducido los parámetros correctamente por cada maquina se añaden a la lista de maquinas y después se pasan a de mutableList a lista con un toList y justo después se pasa a cadena de texto con todos los valores que contenga.

```
suspend fun findAllMaquinasByAcquisitionDate(date: LocalDate, operador: String): String {
    lateinit var personalizadoras: List<PersonalizadoraDTO>
    lateinit var encordadoras: List<EncordadoraDTO>
    when (operador) {
        ">" -> {
            personalizadoras = pService.getAllPersonalizadoras().filter { it.fechaAdquisicion > date }
            encordadoras = eService.getAllEncordadoras().filter { it.fechaAdquisicion > date }
        }
        "<" -> {
            personalizadoras = pService.getAllPersonalizadoras().filter { it.fechaAdquisicion < date }
            encordadoras = eService.getAllEncordadoras().filter { it.fechaAdquisicion < date }
        }
        else -> {
            personalizadoras = listOf()
            encordadoras = listOf()
        }
    }
    val maquinas: MutableList<MaquinaDTO> = mutableListOf()
    personalizadoras.forEach { maquinas.add(it) }
    encordadoras.forEach { maquinas.add(it) }
    return maquinas.toList().toString()
}
```

- PedidoController:

  La función findAllPedidos devuelve en una cadena de texto los valores devueltos haciendo uso de la clase service inyectada que contiene todas las funciones del repositorio mappeadas para devolver el valor esperado devolviendo una lista de pedidos pasadas a cadena de texto con formato JSON.

```
suspend fun findAllPedidos(): String {
    return service.getAllPedidos().toString()
}
```

La función getPedidoById necesita que se le pase por parámetro un identificador, en el caso de esta aplicación es necesario un UUID, si el uuid introducido existe se devolverá a través del servicio el encordado en formato JSON y en caso contrario se devolverá un mensaje por defecto indicando el id introducido

```
suspend fun getPedidoById(id: UUID): String {
    return service.getPedidoById(id)?.toJSON() ?: "Pedido with id $id not found."
}
```

La función insertPedido requiere que se le pasen los datos por parámetro en lo que será un dto que será convertido e insertado con la función del servicio designada para la creación de encordados y con la cual se obtiene de paso el objeto insertado en formato JSON.

```
suspend fun insertPedido(dto: PedidoDTO): String {
    return service.createPedido(dto).toJSON()
}
```

La función deletePedido requiere que se le pase por parámetro el objeto dto que será eliminado, se comprueba si ha sido eliminado o no a través de service con su función deletePedido que devolverá verdadero o falso y se introducirá en la variable del resultado la cuál después es comprobada para realizar una acción u otra, en caso verdadero devolverá el objeto dto pasado por parámetro en formato JSON y en caso negativo devuelve un mensaje por defecto avisando que el objeto dto mostrando solo su id no ha podido ser eliminado.

```
suspend fun deletePedido(dto: PedidoDTO): String {
    val result = service.deletePedido(dto)
    return if (result) dto.toJSON()
    else "Could not delete Pedido with id ${dto.id}"
}
```

La función findAllPedidosFromUser requiere que se le pase por parámetro el objeto dto que será el usuario del que se quiere saber todos los pedidos, se devuelve el valor devuelto por la función de la clase service getAllPedidos donde se comprueba el id del cliente con el valor pasado por parámetro para devolver los pedidos de este usuario.

```
suspend fun findAllPedidosFromUser(user: UserDTO): String {
    return service.getAllPedidos().filter { it.client.id == user.id }.toString()
}
```

La función findAllPedidosWithEstado requiere de pasarle un estado por parámetro, con la función de la clase service getAllPedidos se aplica un filtrado donde el estado del pedido sea el mismo que el pasado por parámetro y se devuelve como una cadena de texto.

```
  suspend fun findAllPedidosWithEstado(state: PedidoEstado): String {
      return service.getAllPedidos().filter { it.state == state }.toString()
  }
```

- PersonalizacionController: Se le inserta por inyección la clase PersonalizacionService.

  La función findAllPersonalizaciones devuelve en una cadena de texto los valores devueltos haciendo uso de la clase service inyectada que contiene todas las funciones del repositorio mappeadas para devolver el valor esperado devolviendo una lista de personalizaciones pasadas a cadena de texto con formato JSON.

```
suspend fun findAllPersonalizaciones(): String {
    return service.getAllPersonalizaciones().toString()
}
```

La función getPersonalizacionById necesita que se le pase por parámetro un identificador, en el caso de esta aplicación es necesario un UUID, si el uuid introducido existe se devolverá a través del servicio el encordado en formato JSON y en caso contrario se devolverá un mensaje por defecto indicando el id introducido

```
suspend fun getPersonalizacionById(id: UUID): String {
    return service.getPersonalizacionById(id)?.toJSON() ?: "Personalizacion with id $id not found."
}
```

La función insertPersonalización requiere que se le pasen los datos por parámetro en lo que será un dto que será convertido e insertado con la función del servicio designada para la creación de encordados y con la cual se obtiene de paso el objeto insertado en formato JSON.

```
suspend fun insertPersonalizacion(dto: PersonalizacionDTO): String {
    return service.createPersonalizacion(dto).toJSON()
}
```

La función deletePersonalizacion requiere que se le pase por parámetro el objeto dto que será eliminado, se comprueba si ha sido eliminado o no a través de service con su función deletePersonalizacion que devolverá verdadero o falso y se introducirá en la variable del resultado la cuál después es comprobada para realizar una acción u otra, en caso verdadero devolverá el objeto dto pasado por parámetro en formato JSON y en caso negativo devuelve un mensaje por defecto avisando que el objeto dto mostrando solo su id no ha podido ser eliminado.

```
suspend fun deletePersonalizacion(dto: PersonalizacionDTO): String {
    val result = service.deletePersonalizacion(dto)
    return if (result) dto.toJSON()
    else "Could not delete Personalizacion with id ${dto.id}"
}
```

- PersonalizadoraController:

  La función findAllPersonalizadoras devuelve en una cadena de texto los valores devueltos haciendo uso de la clase service inyectada que contiene todas las funciones del repositorio mappeadas para devolver el valor esperado devolviendo una lista de personalizadoras pasadas a cadena de texto con formato JSON.

```
suspend fun findAllPersonalizadoras(): String {
    return service.getAllPersonalizadoras().toString()
}
```

  La función getPersonalizadoraById necesita que se le pase por parámetro un identificador, en el caso de esta aplicación es necesario un UUID, si el uuid introducido existe se devolverá a través del servicio el encordado en formato JSON y en caso contrario se devolverá un mensaje por defecto indicando el id introducido

```
suspend fun getPersonalizadoraById(id: UUID): String {
    return service.getPersonalizadoraById(id)?.toJSON() ?: "Personalizadora with id $id not found."
}
```

  La función insertPersonalización requiere que se le pasen los datos por parámetro en lo que será un dto que será convertido e insertado con la función del servicio designada para la creación de encordados y con la cual se obtiene de paso el objeto insertado en formato JSON.

```
suspend fun insertPersonalizadora(dto: PersonalizadoraDTO): String {
    return service.createPersonalizadora(dto).toJSON()
}
```

La función deletePersonalizadora requiere que se le pase por parámetro el objeto dto que será eliminado, se comprueba si ha sido eliminado o no a través de service con su función deletePersonalizadora que devolverá verdadero o falso y se introducirá en la variable del resultado la cuál después es comprobada para realizar una acción u otra, en caso verdadero devolverá el objeto dto pasado por parámetro en formato JSON y en caso negativo devuelve un mensaje por defecto avisando que el objeto dto mostrando solo su id no ha podido ser eliminado.

```
suspend fun deletePersonalizadora(dto: PersonalizadoraDTO): String {
    val result = service.deletePersonalizadora(dto)
    return if (result) dto.toJSON()
    else "Could not delete Personalizadora with id ${dto.id}"
}
```

La función findAllManeuverability requiere que se le pase por parámetro un boolean con el que se devolverá a través del service y la función getAllPersonalizadoras junto a un filtrado en el que se busca las personalizadoras que tengan el atributo igual al parámetro pasado y después de eso se pasa a string para devolver una cadena de texto con el resultado.

```
suspend fun findAllManeuverability(bool: Boolean): String {
    return service.getAllPersonalizadoras().filter { it.measuresManeuverability == bool }.toString()
}
```

La función findAllRigidity requiere que se le pase por parámetro un boolean con el valor que devolverá a través del service y la función getAllPersonalizadoras junto a un filtrado que compara el atributo de la personalizadora esperando que sea igual que el parámetro pasado, después de esto se pasa a string para devolver una cadena de texto con el resultado.

```
suspend fun findAllRigidity(bool: Boolean): String {
    return service.getAllPersonalizadoras().filter { it.measuresRigidity == bool }.toString()
}
```

La función findAllBalance requiere que se le pase por parámetro un boolean con el valor que devolverá a través del service y la función getAllPersonalizadoras junto a un filtrado que compara el atributo de la

personalizadora esperando que sea igual que el parámetro pasado, después de esto se pasa a string para devolver una cadena de texto con el resultado.

```
suspend fun findAllBalance(bool: Boolean): String {
    return service.getAllPersonalizadoras().filter { it.measuresBalance == bool }.toString()
}
```

- ProductoController:

  La función findAllProductos devuelve en una cadena de texto los valores devueltos haciendo uso de la clase service inyectada que contiene todas las funciones del repositorio mappeadas para devolver el valor esperado devolviendo una lista de personalizaciones pasadas a cadena de texto con formato JSON.

```
suspend fun findAllProductos(): String {
    return service.getAllProductos().toString()
}
```

  La función getProductoById necesita que se le pase por parámetro un identificador, en el caso de esta aplicación es necesario un UUID, si el uuid introducido existe se devolverá a través del servicio el encordado en formato JSON y en caso contrario se devolverá un mensaje por defecto indicando el id introducido.

```
suspend fun getProductoById(id: UUID): String {
    return service.getProductoById(id)?.toJSON() ?: "Producto with id $id not found."
}
```

  La función getProductosByTipo requiere que se le pase el tipo por parámetro, lo compara con el valor devuelto con la función del service que devuelve todos los productos mediante un filtrado en el que el tipo del producto en el atributo del objeto sea igual al pasado por parámetro, luego se pasa a cadena de texto para devolverse como tal.

```
suspend fun getProductosByTipo(tipo: TipoProducto): String {
    return service.getAllProductos().filter { it.tipoProducto == tipo }.toString()
}
```

  La función getProductosDisponibles devuelve todos los productos y con un filtrado devuelve solo los que tengan un stock mayor a 0 y se pasa a cadena de texto.

```
suspend fun findAllProductosDisponibles(): String {
    return service.getAllProductos().filter { it.stock > 0 }.toString()
}
```

La función insertProducto requiere que se le pasen los datos por parámetro en lo que será un dto que será convertido e insertado con la función del servicio designada para la creación de encordados y con la cual se obtiene de paso el objeto insertado en formato JSON.

```
suspend fun insertProducto(dto: ProductoDTO): String {
    return service.createProducto(dto).toJSON()
}
```

La función deleteProducto requiere que se le pase por parámetro el objeto dto que será eliminado, se comprueba si ha sido eliminado o no a través de service con su función deleteProducto que devolverá verdadero o falso y se introducirá en la variable del resultado la cuál después es comprobada para realizar una acción u otra, en caso verdadero devolverá el objeto dto pasado por parámetro en formato JSON y en caso negativo devuelve un mensaje por defecto avisando que el objeto dto mostrando solo su id no ha podido ser eliminado.

```
suspend fun deleteProducto(dto: ProductoDTO): String {
    val result = service.deleteProducto(dto)
    return if (result) dto.toJSON()
    else "Could not delete Producto with id ${dto.id}"
}
```

- TareaController:

  La función findAllTareas utiliza 3 valores en los que se obtienen todas las tareas usando los métodos getAll(tipo de tarea) que serán añadidas en una lista mutable que se devuelven en una cadena de texto después convertir la lista mutable a lista y de lista a cadena de texto.

```
suspend fun findAllTareas(): String {
    val adquisiciones = aService.getAllAdquisiciones()
    val encordados = eService.getAllEncordados()
    val personalizaciones = pService.getAllPersonalizaciones()
    val tareas: MutableList<TareaDTO> = mutableListOf()
    adquisiciones.forEach { tareas.add(it) }
    encordados.forEach { tareas.add(it) }
    personalizaciones.forEach { tareas.add(it) }
    return tareas.toList().toString()
}
```

La función getTareaById necesita que se le pase por parámetro un identificador, en el caso de esta aplicación es necesario un UUID, con la variable búsqueda se intenta obtener una adquisición con el id pasado por parámetro, en caso de obtener algo ejecutará la función de AdquisiciónController "getAdquisicionById" y en caso de que búsqueda sea nulo intentará encontrar un encordado con ese id y ejecutar el "getEncordadoById" y por último si en este caso también diese null intentará encontrar una personalización con el id pasado por parámetro y en caso de no encontrar ninguna tarea con ese id devolverá un mensaje de error por defecto.

```
suspend fun getTareaById(id: UUID): String {
    var busqueda: TareaDTO? = aService.getAdquisicionById(id)
    return if (busqueda != null) {
        AdquisicionController.getAdquisicionById(id)
    }
    else {
        busqueda = eService.getEncordadoById(id)
        if (busqueda != null) {
            EncordadoController.getEncordadoById(id)
        }
        else {
            busqueda = pService.getPersonalizacionById(id)
            if (busqueda != null) {
                PersonalizacionController.getPersonalizacionById(id)
            }
            else {
                "Tarea with id $id not found."
            }
        }
    }
}
```

La función insertTarea requiere que se le pasen los datos por parámetro en lo que será un dto que será comprobado para encontrar el tipo de Tarea que es y con eso realizar la ejecución de función correspondiente. Si es Adquisición ejecutará insertAdquisición de su controlador, en caso de ser Encordado ejecutará insertEncordado y si es Personalización ejecutará insertPersonalización.

En caso de no ser ninguna de estos dto devolverá un mensaje por defecto de error.

```
suspend fun insertTarea(dto: TareaDTO): String {
    return when (dto) {
        is AdquisicionDTO -> AdquisicionController.insertAdquisicion(dto)
        is EncordadoDTO -> EncordadoController.insertEncordado(dto)
        is PersonalizacionDTO -> PersonalizacionController.insertPersonalizacion(dto)
        else -> {
            "Error at TareaController.insertTarea: DTO not supported."
        }
    }
}
```

La función deleteTarea requiere que se le pase por parámetro el objeto dto que será eliminado, se comprueba el tipo de Tarea para ejecutar la función del delete del contolador correspondiente al tipo de tarea encontrado, en caso de no ser una tarea admitida devolverá un mensaje de error por defecto, se devolverá el dto eliminado en cadena de texto con formato json

```
suspend fun deleteTarea(dto: TareaDTO): String {
    return when (dto) {
        is AdquisicionDTO -> AdquisicionController.deleteAdquisicion(dto)
        is EncordadoDTO -> EncordadoController.deleteEncordado(dto)
        is PersonalizacionDTO -> PersonalizacionController.deletePersonalizacion(dto)
        else -> {
            "Error at TareaController.deleteTarea: DTO not supported."
        }
    }
}
```

- TurnoController:

  La función findAllTurnos devuelve en una cadena de texto los valores devueltos haciendo uso de la clase service inyectada que contiene todas las funciones del repositorio mappeadas para devolver el valor esperado devolviendo una lista de personalizaciones pasadas a cadena de texto con formato JSON.

```
suspend fun findAllTurnos(): String {
    return service.getAllTurnos().toString()
}
```

La función getTurnoById necesita que se le pase por parámetro un identificador, en el caso de esta aplicación es necesario un UUID, si el uuid introducido existe se devolverá a través del servicio el encordado en formato JSON y en caso contrario se devolverá un mensaje por defecto indicando el id introducido

```
suspend fun getTurnoById(id: UUID): String {
    return service.getTurnoById(id)?.toJSON() ?: "Turno with id $id not found."
}
```

La función insertTurno requiere que se le pasen los datos por parámetro en lo que será un dto que será convertido e insertado con la función del servicio designada para la creación de encordados y con la cual se obtiene de paso el objeto insertado en formato JSON.

```
suspend fun insertTurno(dto: TurnoDTO): String {
    return service.createTurno(dto).toJSON()
}
```

La función deleteTurno requiere que se le pase por parámetro el objeto dto que será eliminado, se comprueba si ha sido eliminado o no a través de service con su función deleteTurno que devolverá verdadero o falso y se introducirá en la variable del resultado la cuál después es comprobada para realizar una acción u otra, en caso verdadero devolverá el objeto dto pasado por parámetro en formato JSON y en caso negativo devuelve un mensaje por defecto avisando que el objeto dto mostrando solo su id no ha podido ser eliminado.

```
suspend fun deleteTurno(dto: TurnoDTO): String {
    val result = service.deleteTurno(dto)
    return if (result) dto.toJSON()
    else "Could not delete Turno with id ${dto.id}"
}
```

La función findAllTurnosSortedByFecha devuelve a través de la función del service getAllTurnos una lista de todos los turnos que después son ordenados con el uso de sortedBy con el atributo de hora de inicio, que luego es pasado a cadena de texto para devolverlo.

- UserController:

  La función findAllUsers devuelve en una cadena de texto los valores devueltos haciendo uso de la clase service inyectada que contiene todas las funciones del repositorio mappeadas para devolver el valor esperado devolviendo una lista de personalizaciones pasadas a cadena de texto con formato JSON.

```
suspend fun findAllUsers(): String {
    return service.getAllUsers().toString()
}
```

La función getUserById necesita que se le pase por parámetro un identificador, en el caso de esta aplicación es necesario un UUID, si el uuid introducido existe

se devolverá a través del servicio el encordado en formato JSON y en caso contrario se devolverá un mensaje por defecto indicando el id introducido

```
suspend fun getUserById(id: UUID): String {
    return service.getUserById(id)?.toJSON() ?: "User with id $id not found."
}
```

La función insertUser requiere que se le pasen los datos por parámetro en lo que será un dto que será convertido e insertado con la función del servicio designada para la creación de encordados y con la cual se obtiene de paso el objeto insertado en formato JSON.

```
suspend fun insertUser(dto: UserDTO): String {
    return service.createUser(dto).toJSON()
}
```

La función deleteUser requiere que se le pase por parámetro el objeto dto que será eliminado, se comprueba si ha sido eliminado o no a través de service con su función deleteUser que devolverá verdadero o falso y se introducirá en la variable del resultado la cuál después es comprobada para realizar una acción u otra, en caso verdadero devolverá el objeto dto pasado por parámetro en formato JSON y en caso negativo devuelve un mensaje por defecto avisando que el objeto dto mostrando solo su id no ha podido ser eliminado.

```
suspend fun deleteUser(dto: UserDTO): String {
    val result = service.deleteUser(dto)
    return if (result) dto.toJSON()
    else "Could not delete User with id ${dto.id}"
}
```

La función findAllUsersWitRole requiere que se le pase por parámetro un rol que será utilizado junto a la lista de usuarios obtenidas con la función getAllUsers del service que con un filtrado se comprueba que el parámetro perfil del objeto sea igual, esta lista se devuelve a una cadena de texto.

```
suspend fun findAllUsersWithRole(role: Profile): String {
    return service.getAllUsers().filter { it.perfil == role }.toString()
}
```

La función getUsersByEmail requiere que se le pase por parámetro una cadena de texto que será el email del usuario que se quiere encontrar llama a la función de service getUserByMail al que se le pasa el parámetro introducido a

esta función se devuelve como una cadena de texto con formato JSON y en caso de no existir este email devuelve un mensaje de error por defecto.

```
suspend fun getUserByEmail(email: String): String {
    return service.getUserByMail(email)?.toJSON() ?: "User with email $email not found."
}
```

La función getUserByEmailForLogin requiere una cadena de texto pasada por parámetro que servirá para pasárselo a la función de service getUserByMail para devolver un objeto DTO.

```
suspend fun getUserByEmailForLogin(email: String): UserDTO? {
    return service.getUserByMail(email)
}
```

La función getUserByPhone requiere que se le pase por parámetro una cadena de texto que será el teléfono que se utilizará en la función de service getUserByPhone que en caso de existir devolverá el usuario como cadena de texto en formato JSON, en caso de no existir devolverá un mensaje de error por defecto.

```
suspend fun getUserByPhone(phone: String): String {
    return service.getUserByPhone(phone)?.toJSON() ?: "User with phone $phone not found."
}
```

La función getUserByPhoneForLogin requiere que se le pase por parámetro una cadena de texto que será el teléfono que se utilizará en la función service getUserByPhone que en caso de existir devolverá el objeto DTO del usuario.

```
suspend fun getUserByPhoneForLogin(phone: String): UserDTO? {
    return service.getUserByPhone(phone)
}
```

Db:

- DataLoader: Clase utilizada para precargar datos de ejemplo de cada entidad utilizada en el proyecto. Hace uso de los mappers de cada tipo de entidad requerida para la inserción de entidades que se necesiten para las relaciones al necesitar pasar los datos de DTO a POKO.

```
class DataLoader {
    val pMapper = ProductoMapper()
    val uMapper = UserMapper()
    val mMapper = MaquinaMapper()
    val tMapper = TareaMapper()
    val turMapper = TurnoMapper()

    val raqueta = ProductoDTO(
        id = UUID.fromString("93a98d69-6da6-48a7-b34f-05b596ea83ba"),
        tipoProducto = TipoProducto.RAQUETAS,
        marca = "MarcaRaqueta",
        modelo = "ModeloRaqueta",
        precio = 150.5,
        stock = 3
    )
```

- DataBaseManager: Clase necesaria para el funcionamiento de la base de datos, esta clase hace uso de la clase AppConfig para recibir los parámetros necesarios para la conexión y funciones de la base de datos, tiene una función de inicialización para realizar la conexión y poder empezar con la gestión de datos. La función init recibe por parámetro la clase AppConfig para aplicar a la configuración de Hikari los parámetros que se encuentran en ella, para ello utiliza "HikariConfig().apply{}" y entre los corchetes asigna los valores a las variables.

  La función createTables es una transacción que se utiliza para crear los esquemas de las tablas necesarias que se encuentran en la carpeta de entities, toda clase de la que haya un DAO tiene su creación de tabla para la base de datos mediante código, esas son las tablas creadas a través de esta función.

- Dtos: Hay un dto por modelo realizado en el proyecto, todos tienen la misma función pasar los datos del modelo para poder crear los json que después son visualizados en ejecución. Utilizan la anotación @Expose para que los atributos sean visalizados en el JSON generado a través de la librería GSON, ya que sin esta anotación los datos no se serializarían y por ello no serían mostrados en consola. Esto ocurre por el funcionamiento de GsonBuilder y su función "excludeFieldsWithoutExposeAnnotation" con esto podemos evitar desvelar datos como la contraseña o teléfono de un usuario.
- Entities: Hay un archivo entity por cada modelo que requiere de tener una tabla propia en la base de datos, estos archivos se componen de un object ya que

solo puede estar en ejecución una vez durante el proceso para evitar errores, en este object creamos la Tabla una única vez, estas tablas utilizan como identificador UUID's haciendo uso de la función de exposed UUIDTable() que entre paréntesis lleva el nombre que deseamos que utilice en la base de datos, dentro de esta tabla se definen los atributos que contendrá y a qué hacen referencia o su tipo de dato, que pueden ser integer, double, boolean, etc…
En la clase Dao que se encuentra dentro de este fichero recibe el id haciendo uso de EntityID<UUID> donde le especificamos el tipo de id con el que estamos trabajando, después hacemos que extienda de UUIDEntity(id) y le pasamos por parámetro a este el id especificado a la clase DAO, con el companion object hacemos que extienda de la clase UUIDEntityClass<> con la clase DAO que se vaya a utilizar entre las flechas donde se asignan las variables del dao al campo de la tabla de la base de datos.

- Excepciones: Mapper exception utilizada para errores a la hora de mappear datos.

Login:

Contiene el archivo login compuesto por la función suspendida login donde se meten el email y contraseña a través de consola por el usuario, aquí se llama a los métodos que comprueban si el email es correcto y si la contraseña es correcta también, para ello se codifica a sha512 y se comprueba si es igual a la contraseña codificada en la base de datos, en caso de introducir las credenciales incorrectamente se mostrará un mensaje para preguntarle al usuario si desea salir o continuar en el programa, en caso afirmativo se cerrará con un exitProcess(0). Si se introducen los parámetros correctamente se mostrará un mensaje de registro correcto y devolverá el dto del usuario.
La función de registro lee por consola los parámetros del nombre, apellido, número de teléfono, email y contraseña, hay dos values dedicados con corrutinas a buscar si hay un usuario con ese teléfono o email para comprobar que no haya un usuario con ese email o teléfono ya registrado. En caso de ya existir devolverá un mensaje de parámetros incorrectos.

Mappers:

La clase Base Mapper funciona como una interfaz, es una clase abstracta que no se puede instanciar pero de la que se puede heredar para implementar sus funciones en otras clases, como hacemos en la práctica.

Los mappers comprueban el tipo de dato que se les haya pasado en caso de ser el de máquina o tareas y mappean los atributos de la clase DAO al Objeto POKO para recibir esos datos por consola.

Menu:

La clase Modo presentación inserta a la base de datos para mostrar el funcionamiento y ejecuta todas las funciones disponibles del programa.

Models:

Clases POKO con los constructores necesarios para la creación de entidades para la base de datos.

Repositories:

Clases con funciones suspendidas para realizar las funciones CRUD, hay clases que hacen funciones específicas ya tratadas en la sección de controladores, el repositorio requiere que se le inyecte la clase o clases DAO correspondientes para su funcionamiento mediante transacciones y haciendo uso de los mappeadores correspondientes para el tipo de objeto.

- Services: Las clases service sirven para realizar las funciones llamando a los repositorios necesarios, mappeandolos y devolviendo los datos después de realizar la acción con corrutinas.
- Útil: Este archivo contiene las funciones para poder codificar la contraseña del usuario a sha512 utilizando MessageDigest para ello. La función betweenXandY sirve para los menus donde tú le pasas una cadena de texto y lo convierte a int, y si te devuelve un null devuelve false si no puede castearlo a número, si puede castearlo devolverá true. La función waitingText requiere que se le pase un deferred por parámetro, con ello comprueba si ha sido completado o no para imprimir por pantalla puntos hasta que se haya resuelto el deferred, utiliza una espera de 100 milisegundospara ello.

- Main: La clase main tiene varias corrutinas encargadas de insertar datos de la base y cargar los datos de la base de datos, mientras los job no estén completados imprimirá un punto por pantalla hasta que terminen todos y realice el joinAll. Lanza una corrutina por cada tipo de objeto en la base de datos para obtener todos los resultados y después comienza el proceso de registro o inicio de sesión.

Decisiones:

Hemos decidido utilizar Hikari para poder realizar el programa con funciones concurrentes.

Solo hemos creado una única excepción para no tirar abajo el programa en caso de fallo, hemos establecido mensajes por defecto de error para notificar los errores.

Los modelos tienen un constructor en vez de usar data class porque de esa manera nos dejaban insertar y comprobar datos a través de los mappers sin necesidad de tener que hacer obligatoriamente inserciones de UUID's desde fuera del propio modelo.

Hemos creado los servicios para realizar las acciones del repositorio con corrutinas y sus correspondientes esperas para la obtención de los datos usando en esto una clase abstracta de la que extendemos para hacer uso de sus funciones.

Hemos creado menús para una interfaz visual en consola para el funcionamiento de la aplicación.

Los mappers están hechos de manera que casaban los valores necesarios como requería para pasar de DAO a POKO.

Entidades con las relaciones requeridas para el funcionamiento del programa. Hemos establecido que una tarea puede contener varios productos y un usuario realizar varias tareas, un pedido puede tener varias tareas, pero una tarea solo puede pertenecer a un único pedido y ese pedido puede realizarse en más de un turno, pero no en más de 2 y ese mismo turno puede tener entre 1 y varios pedidos. Un usuario puede tener un único turno y un turno tiene asignado una única máquina, mientras que una máquina puede no estar siendo utilizada en un turno.