SEGUNDA PRACTICA ACCESO A DATOS

RUBEN GARCÍA-REDONDO

ALEJANDRO ABAD

Contenido

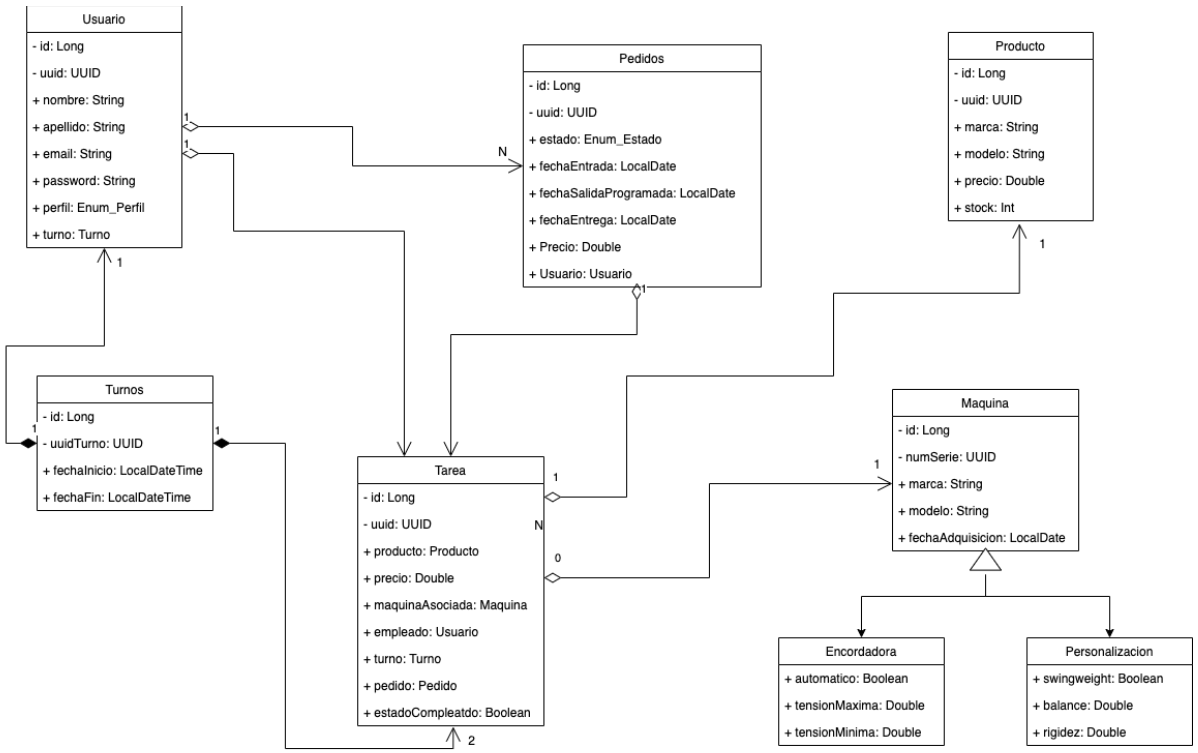
1. Introducción.

Esta es la segunda práctica de Acceso a Datos, De la práctica es Tennis Lab tendremos que hacer una colaboración para que Xavi pueda para gestionar los pedidos del Torneo Conde de Godó.

2. Requisitos de informacion

Para los requisitos de informacion estuvimos informandonos sobre el tema de como funcionan las Maquinas de Personalizacion y Encordar para poder crear los modelos de una forma correcta. Otra cosa que miramos fue el como encriptar la contraseña ya que para hacer pedidos o crear tareas necesitamos un usuario y su contraseña, que se guardan en la base de datos.

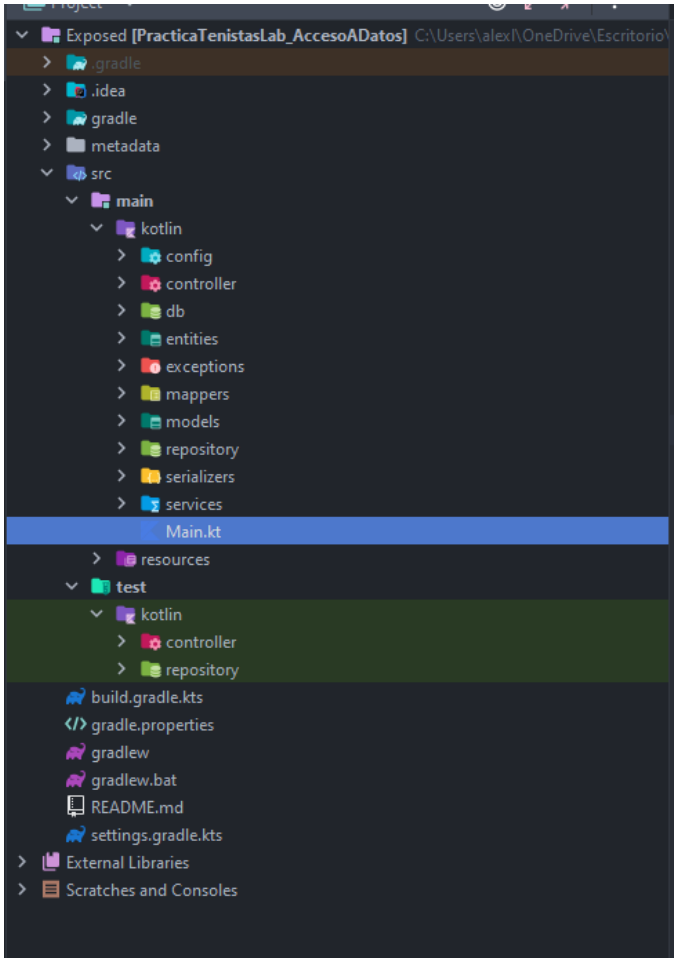
3. Diagrama de clases



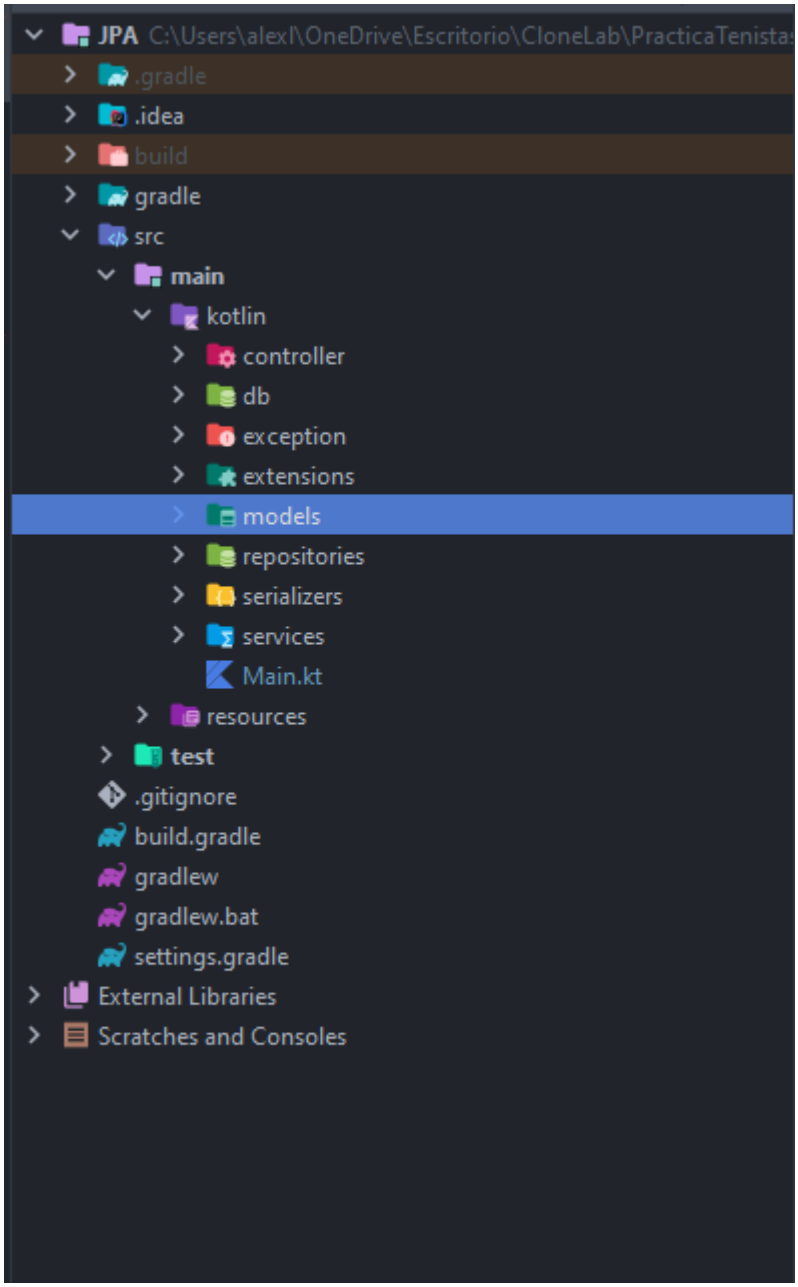
capar en el controlador que un empleado solo pueda tener pedidos de un mismo tipo

4. Arquitectura del sistema y patrones usados

Este proyecto está dividido en dos ya que está resuelto con Expose y con JPA por lo que en cada proyecto tendremos una estructura, para expose lo hemos ordenado de la siguiente manera :

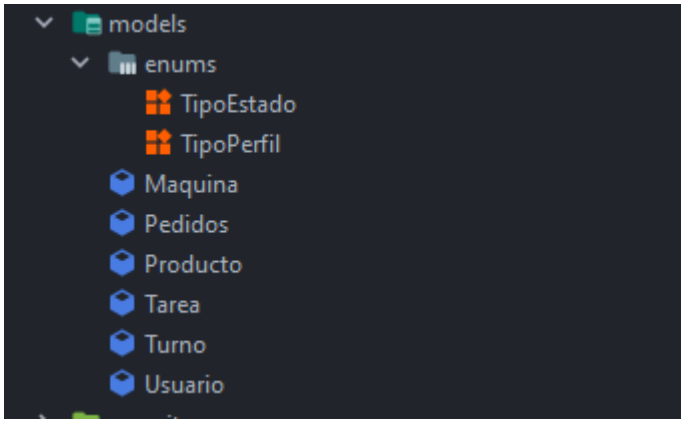


y para JPA lo hemos ordenado de esta manera :

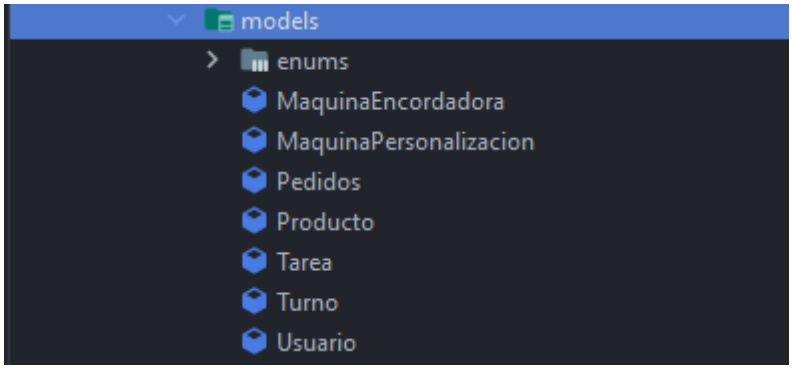


A primera vista se pueden ver diferencias, en expose necesitamos crearnos nosotros una carpeta entity en la cual crearemos las entidades de las tablas y sus DAO, también necesitaremos los mapper de cada uno de los modelos, en cambio, en JPA no ya que en los modelos de JPA gracias a las anotaciones esto lo hará automáticamente, otra diferencia es que en JPA tendremos en resources una carpeta META-INF con un archivo llamado persistence.xml en el cual tendremos que crear y nombrar los modelos que tenemos para que JPA pueda crear la base de datos y sus respectivas tablas.

(TITULO 3) Los modelos que hemos creado para esta practica son :
Exposed



JPA

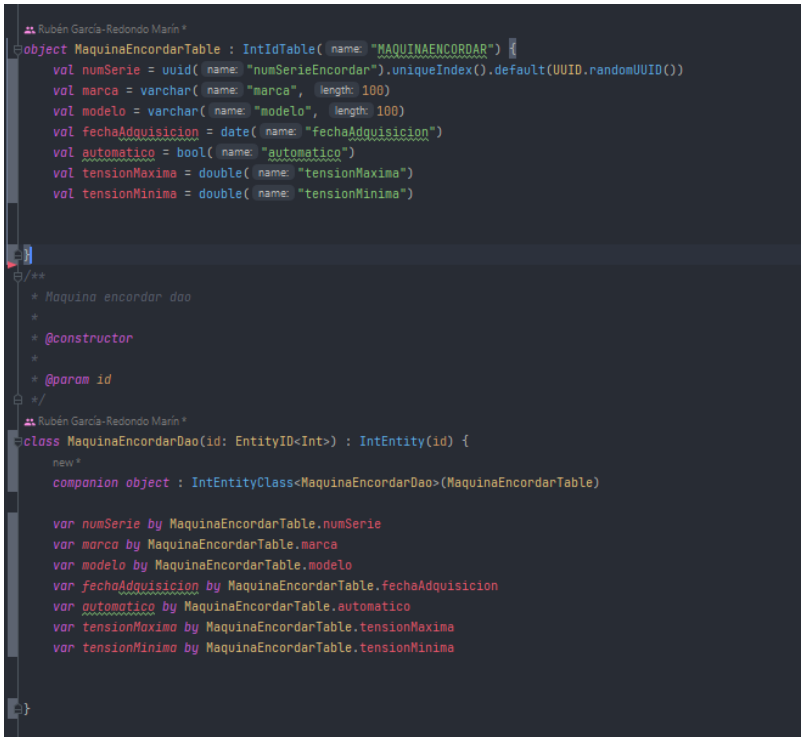


El modelo maquina :

```
Rubén Garcia Redondo Martin +3
@Serializable
sealed class Maquina() {

    /**
     * Maquina personalizacion
     *
     * @property id
     * @property numSerie
     * @property marca
     * @property modelo
     * @property fechaAdquisicion
     * @property swingweight
     * @property balance
     * @property rigidez
     * @constructor Create empty Maquina personalizacion
     */
    Rubén +2
    @Serializable
    data class MaquinaPersonalizacion(
        val id: Int,
        @Serializable(UUIDSerializer::class)
        val numSerie: UUID=UUID.randomUUID(),
        val marca: String,
        val modelo: String,
        @Serializable(LocalDateSerializer::class)
        val fechaAdquisicion: LocalDate,
        val swingweight: Boolean,
        val balance: Double,
        val rigidez: Double
    ){
    }

    /**
     * Maquina encordadora
     *
     * @property id
     * @property numSerie
     * @property marca
     * @property modelo
     * @property fechaAdquisicion
     * @property automatico
     * @property tensionMaxima
     * @property tensionMinima
     * @constructor Create empty Maquina encordadora
     */
    Rubén +2
    @Serializable
    data class MaquinaEncordadora(
        val id: Int,
        @Serializable(UUIDSerializer::class)
        val numSerie: UUID = UUID.randomUUID(),
        val marca: String,
        val modelo: String,
        @Serializable(LocalDateSerializer::class)
        val fechaAdquisicion: LocalDate,
        val automatico: Boolean,
        val tensionMaxima: Double,
        val tensionMinima: Double
```

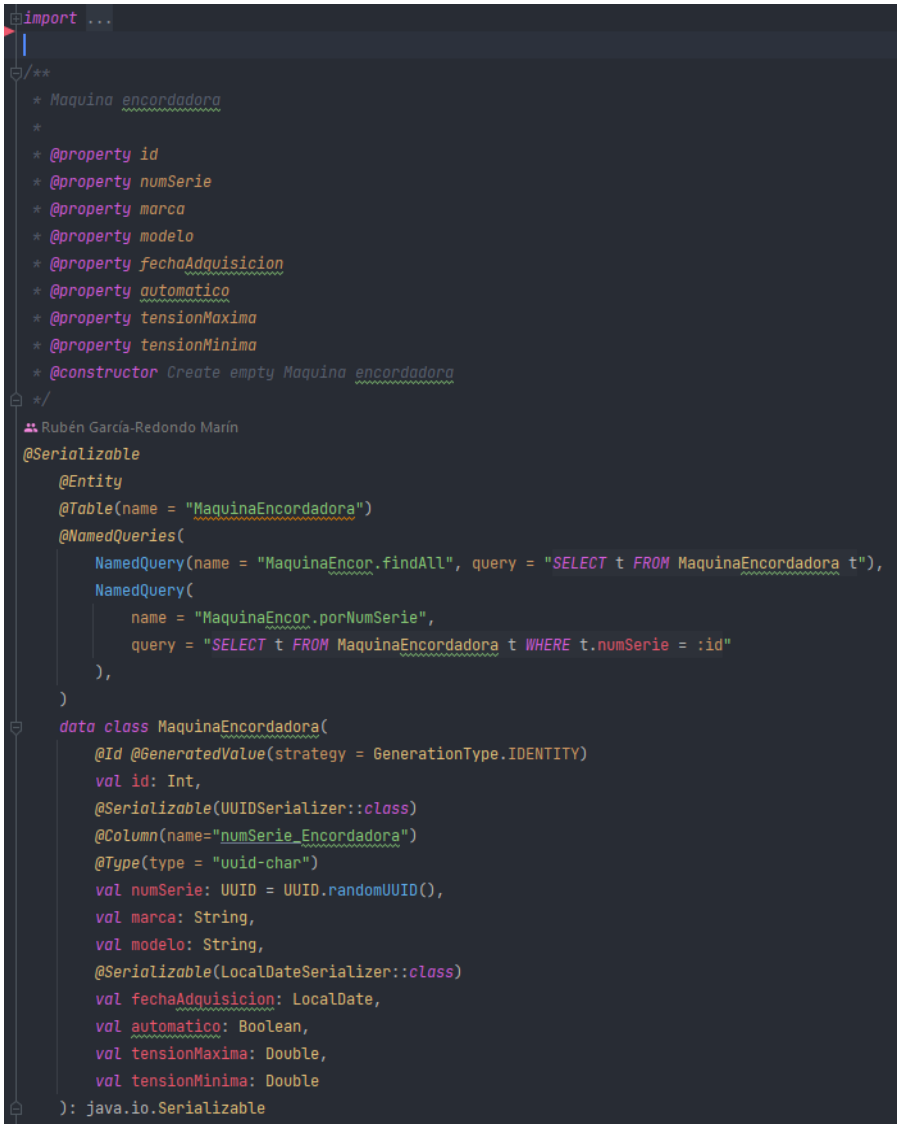
Como se puede observar hemos creado una Sealed class donde dentro se encuentran la Máquina de personalización y la Máquina de encordado, tienen atributos que se repiten y luego tienen algunos que son únicos para cada una de las máquinas, se puede observar también que hemos añadido la anotacion @Serializable con la que le decimos que ese modelo va a poder ser serializado, como por ejemplo a JSON, para trabajar con los UUID y los LocalDate tuvimos que crearnos nuestras propias clases con las que serializar esos tipos de datos. Como estamos trabajando con Exposed necesitamos la entidad de ese modelo para poder crear la tabla por lo que como nosotros tenemos dos tablas, una máquina Encordadora y otra máquina Personalización a continuación mostraré cómo son esas dos entidades.

```
Alejandro +1
object MaquinaPersonalizacionTable : IntIdTable( name: "MAQUINAPERSO") {

    val numSerie = uuid( name: "numSeriePersonalizar").uniqueIndex()
    val marca = varchar( name: "marca",  length: 100)
    val modelo = varchar( name: "modelo",  length: 100)
    val fechaAdquisicion = date( name: "fechaAdquisicion")
    val swingweight= bool( name: "swingweitght")//se puede medir o no la maniobrabilidad
    val balance= double( name: "balance")
    val rigidez=double( name: "rigidez")
}

/**
 * Maquina personalizacion dao
 *
 * @constructor
 *
 * @param id
 */
Alejandro
class MaquinaPersonalizacionDao(id: EntityID<Int>):  IntEntity(id) {

    Alejandro
    companion object : IntEntityClass<MaquinaPersonalizacionDao>(MaquinaPersonalizacionTable)

    var numSerie by MaquinaPersonalizacionTable.numSerie
    var marca by MaquinaPersonalizacionTable.marca
    var modelo by MaquinaPersonalizacionTable.modelo
    var fechaAdquisicion by MaquinaPersonalizacionTable.fechaAdquisicion
    var swingweight by MaquinaPersonalizacionTable.swingweight
    var balance by MaquinaPersonalizacionTable.balance
    var rigidez by MaquinaPersonalizacionTable.rigidez
}
```



```
Rubén García-Redondo Marín *
object MaquinaEncordarTable : IntIdTable( name: "MAQUINAENCORDAR") {
    val numSerie = uuid( name: "numSerieEncordar").uniqueIndex().default(UUID.randomUUID())
    val marca = varchar( name: "marca",  length: 100)
    val modelo = varchar( name: "modelo",  length: 100)
    val fechaAdquisicion = date( name: "fechaAdquisicion")
    val automatico = bool( name: "automatico")
    val tensionMaxima = double( name: "tensionMaxima")
    val tensionMinima = double( name: "tensionMinima")


}
/**
 * Maquina encordar dao
 *
 * @constructor
 *
 * @param id
 */
Rubén García-Redondo Marín *
class MaquinaEncordarDao(id: EntityID<Int>) : IntEntity(id) {
    new *
    companion object : IntEntityClass<MaquinaEncordarDao>(MaquinaEncordarTable)

    var numSerie by MaquinaEncordarTable.numSerie
    var marca by MaquinaEncordarTable.marca
    var modelo by MaquinaEncordarTable.modelo
    var fechaAdquisicion by MaquinaEncordarTable.fechaAdquisicion
    var automatico by MaquinaEncordarTable.automatico
    var tensionMaxima by MaquinaEncordarTable.tensionMaxima
    var tensionMinima by MaquinaEncordarTable.tensionMinima


}
```

En cambio con JPA creamos el modelo y la tabla a la vez gracias a las anotaciones de JPA



```kotlin
import ...

/**
 * Maquina encordadora
 *
 * @property id
 * @property numSerie
 * @property marca
 * @property modelo
 * @property fechaAdquisicion
 * @property automatico
 * @property tensionMaxima
 * @property tensionMinima
 * @constructor Create empty Maquina encordadora
 */
Rubén García-Redondo Marín
@Serializable
    @Entity
    @Table(name = "MaquinaEncordadora")
    @NamedQueries(
        NamedQuery(name = "MaquinaEncor.findAll", query = "SELECT t FROM MaquinaEncordadora t"),
        NamedQuery(
            name = "MaquinaEncor.porNumSerie",
            query = "SELECT t FROM MaquinaEncordadora t WHERE t.numSerie = :id"
        ),
    )
    data class MaquinaEncordadora(
        @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
        val id: Int,
        @Serializable(UUIDSerializer::class)
        @Column(name="numSerie_Encordadora")
        @Type(type = "uuid-char")
        val numSerie: UUID = UUID.randomUUID(),
        val marca: String,
        val modelo: String,
        @Serializable(LocalDateSerializer::class)
        val fechaAdquisicion: LocalDate,
        val automatico: Boolean,
        val tensionMaxima: Double,
        val tensionMinima: Double
    ): java.io.Serializable
```

```
/**
 * Maquina personalizacion
 *
 * @property id
 * @property numSerie
 * @property marca
 * @property modelo
 * @property fechaAdquisicion
 * @property swingweight
 * @property balance
 * @property rigidez
 * @constructor Create empty Máquina personalizacion
 */
Alejandro +1
@Serializable
@Entity
@Table(name = "MaquinaPersonalizacion")
@NamedQueries(
    NamedQuery(name = "MaquinaPer.findAll", query = "SELECT t FROM MaquinaPersonalizacion t"),
    NamedQuery(
        name = "MaquinaPerson.porNumSerie",
        query = "SELECT t FROM MaquinaPersonalizacion t WHERE t.numSerie = :id"
    ),
)
data class MaquinaPersonalizacion(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Int,
    @Serializable(UUIDSerializer::class)
    @Column(name="numSerie_Personalizacion")
    @Type(type = "uuid-char")
    val numSerie: UUID=UUID.randomUUID(),
    val marca: String,
    val modelo: String,
    @Serializable(LocalDateSerializer::class)
    val fechaAdquisicion: LocalDate,
    val swingweight: Boolean,
    val balance: Double,
    val rigidez: Double
): java.io.Serializable{
```

En cambio con JPA utilizaremos notaciones como @Entity la cual le va a decir a JPA que ese modelo es una entidad, utilizaremos Table por si queremos cambiarle el nombre a la tabla,
y luego las anotaciones @NamedQueries sirven para crear consultas que luego utilizaremos en los repositorios ya que JPA está automatizado pero no completamente, por lo que como se ve en MaquinaPer.findAll , eso seria simplemente el nombre con el lo llamaremos en el repositorio de MaquinaPersonalizacion, y en la query es donde haremos una Query generica con la que buscaremos todas las maquinas de personalizacion en la tabla MaquinaPersonalizacion. Tendremos que poner tambien @Id y @GeneratedValue para señalar cual va a ser la primary key de nuestra tabla, con el @Columm podremos ponerle un nombre a ese atributo en la tabla y podremos señalar el tipo de dato que es en la base de datos con @Type.

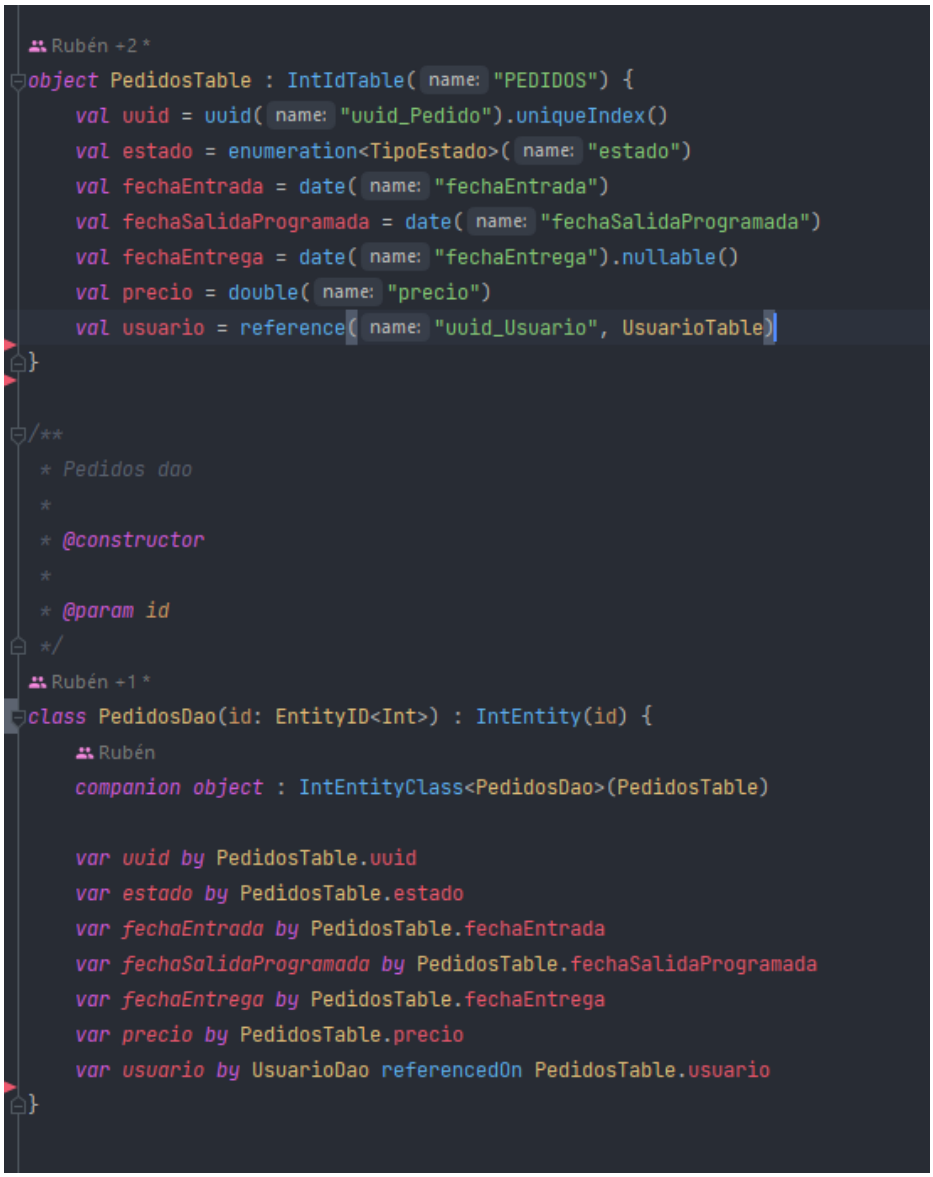
```kotlin
/**
 * Pedidos
 *
 * @property id
 * @property uuid
 * @property estado
 * @property fechaEntrada
 * @property fechaSalidaProgramada
 * @property fechaEntrega
 * @property precio
 * @constructor Create empty Pedidos
 */
Rubén García-Redondo Marín +2
@Serializable
data class Pedidos(
    val id:Int,
    @Serializable(UUIDSerializer::class)
    val uuid: UUID,
    val estado: TipoEstado,
    @Serializable(LocalDateSerializer::class)
    val fechaEntrada: LocalDate,
    @Serializable(LocalDateSerializer::class)
    val fechaSalidaProgramada: LocalDate,
    @Serializable(LocalDateSerializer::class)
    val fechaEntrega: LocalDate?,
    val precio: Double,
    val usuario: Usuario


)
```

El siguente modelo es Pedidos:
Este seria el modelo que utilizaremos en Exposed, cosas a destacar de el es que tiene un estado, que es un enum de estados.

```kotlin
Rubén García-Redondo Marín +1 *
enum class TipoEstado(val num: String) {
    RECIBIDO( num: "Recibido"),
    EN_PROCESO( num: "En proceso"),
    TERMINADO( num: "Terminado")

}
```

En pedidos podremos ver nuestra primera relación ya que un pedido tiene un Usuario

Aqui su entidad :



```
Rubén +2 *
object PedidosTable : IntIdTable( name: "PEDIDOS") {
    val uuid = uuid( name: "uuid_Pedido").uniqueIndex()
    val estado = enumeration<TipoEstado>( name: "estado")
    val fechaEntrada = date( name: "fechaEntrada")
    val fechaSalidaProgramada = date( name: "fechaSalidaProgramada")
    val fechaEntrega = date( name: "fechaEntrega").nullable()
    val precio = double( name: "precio")
    val usuario = reference( name: "uuid_Usuario", UsuarioTable)
}

/**
 * Pedidos dao
 *
 * @constructor
 *
 * @param id
 */
Rubén +1 *
class PedidosDao(id: EntityID<Int>) : IntEntity(id) {
    Rubén
    companion object : IntEntityClass<PedidosDao>(PedidosTable)

    var uuid by PedidosTable.uuid
    var estado by PedidosTable.estado
    var fechaEntrada by PedidosTable.fechaEntrada
    var fechaSalidaProgramada by PedidosTable.fechaSalidaProgramada
    var fechaEntrega by PedidosTable.fechaEntrega
    var precio by PedidosTable.precio
    var usuario by UsuarioDao referencedOn PedidosTable.usuario
}
```

En este caso como se puede ver la tabla pedido una de sus columnas es usuario y hacer referencia al campo uuid de la tabla usuario como se puede ver en la imagen, con JPA el resultado seria asi :

```
Alejandro +1
@Serializable
@Entity
@Table(name = "Pedidos")
@NamedQueries(
    NamedQuery(name = "Pedidos.findAll", query = "SELECT t FROM Pedidos t"),
    NamedQuery(
        name = "Pedidos.porUUID",
        query = "SELECT t FROM Pedidos t WHERE t.uuid = :id"
    ),
)
data class Pedidos(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id:Int,
    @Serializable(UUIDSerializer::class)
    @Column(name="UUID_Pedidos")
    @Type(type = "uuid-char")
    val uuid: UUID,
    val estado: TipoEstado,
    @Serializable(LocalDateSerializer::class)
    val fechaEntrada: LocalDate,
    @Serializable(LocalDateSerializer::class)
    val fechaSalidaProgramada: LocalDate,
    @Serializable(LocalDateSerializer::class)
    val fechaEntrega: LocalDate?,
    val precio: Double,
    @ManyToOne
    @JoinColumn(name = "usuario_uuid", referencedColumnName = "UUID_Usuario")
    val usuario:Usuario,

    ): java.io.Serializable
```

En este caso introducimos dos nuevas notaciones que son la @ManyToOne y @JoinColumn con estas dos notaciones lo que conseguiremos es decirle a JPA como debe mapear y que relacion tiene pedidos con usuarios que en este caso es N-1

Nuestro siguiente modelo es producto :

```
Rubén García-Redondo Marín +2
@Serializable
data class Producto(
    val id:Int,
    @Serializable(UUIDSerializer::class)
    val uuid: UUID= UUID.randomUUID(),
    val marca: String,
    val modelo: String,
    val precio: Double,
    val stock: Int,

    )
```

Este sería el modelo que utilizaremos en Exposed y su entidad

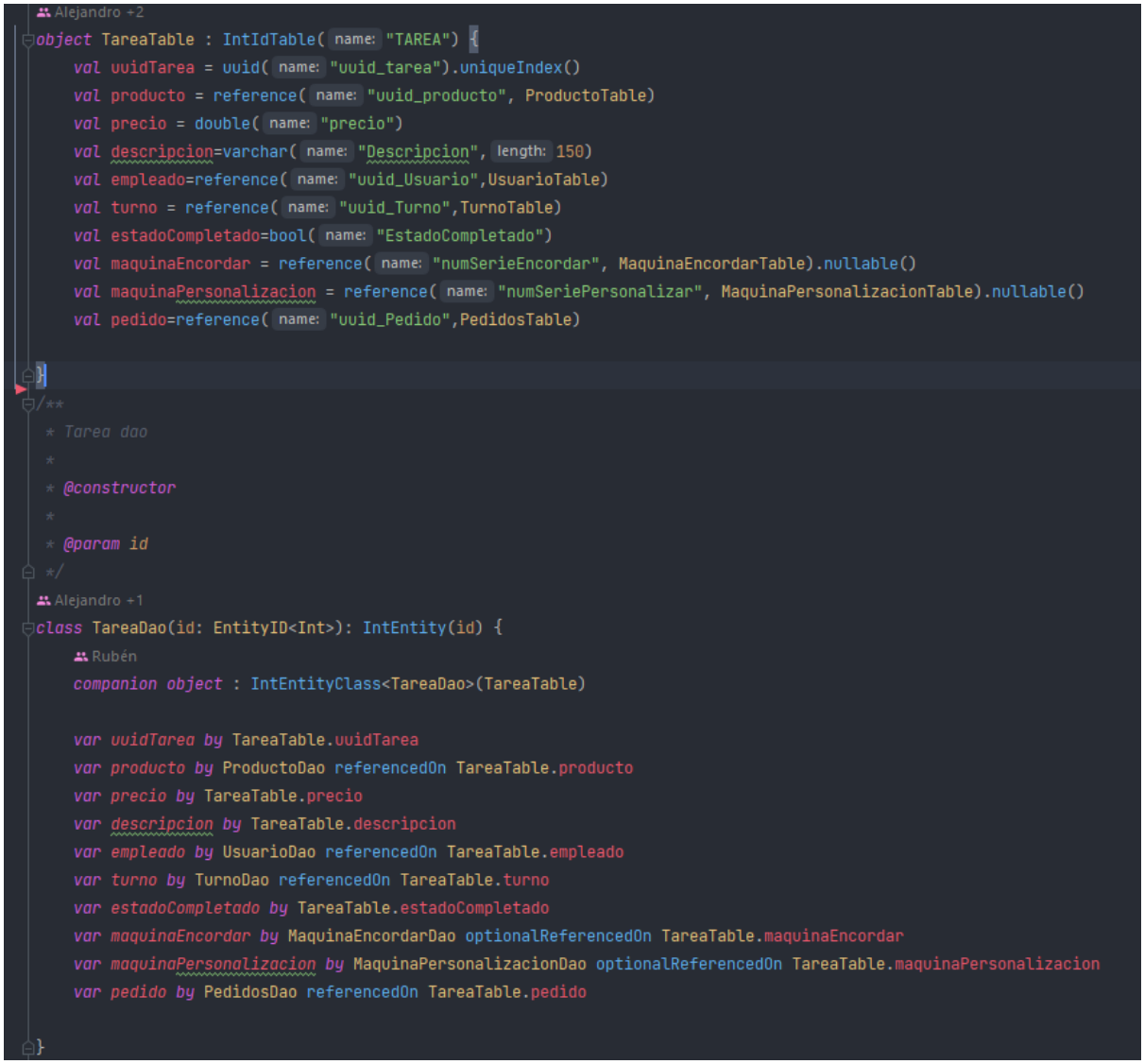
```
Rubén +1 *
object ProductoTable : IntIdTable( name: "PRODUCTO") {
    val uuid = uuid( name: "uuid_producto").uniqueIndex()
    val marca = varchar( name: "marca",  length: 100)
    val modelo = varchar( name: "modelo",  length: 100)
    val precio = double( name: "precio")
    val stock = integer( name: "stock")
}

/**
 * Producto dao
 *
 * @constructor
 *
 * @param id
 */
Rubén +2
class ProductoDao(id: EntityID<Int>): IntEntity(id) {
    Alejandro
    companion object : IntEntityClass<ProductoDao>(ProductoTable)

    var uuid by ProductoTable.uuid
    var marca by ProductoTable.marca
    var modelo by ProductoTable.modelo
    var precio by ProductoTable.precio
    var stock by ProductoTable.stock

}
```

Modelo JPA Producto

```
Alejandro +1
@Serializable
@Entity
@Table(name = "Productos")
@NamedQueries(
    NamedQuery(name = "Productos.findAll", query = "SELECT t FROM Producto t"),
    NamedQuery(
        name = "Producto.porUUID",
        query = "SELECT t FROM Producto t WHERE t.uuid = :id"
    ),
)
data class Producto(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id:Int,
    @Serializable(UUIDSerializer::class)
    @Column(name="UUID_Producto")
    @Type(type = "uuid-char")
    val uuid: UUID = UUID.randomUUID(),
    val marca: String,
    val modelo: String,
    val precio: Double,
    val stock: Int
): java.io.Serializable
```

Gracias a que estamos utilizando el intellij, si tenemos bien configurado nuestro archivo de persitence el propio intellij nos marcará que es una entidad y cual es la clave como se puede ver en la anterior imagen en el lateral izquierdo.

El siguiente modelo es Tarea (Poner que es Titulo3 o asi para que se note diferencia)
Con Expose seria asi :



```
Alejandro +2
object TareaTable : IntIdTable( name: "TAREA") {
    val uuidTarea = uuid( name: "uuid_tarea").uniqueIndex()
    val producto = reference( name: "uuid_producto", ProductoTable)
    val precio = double( name: "precio")
    val descripcion=varchar( name: "Descripcion", length: 150)
    val empleado=reference( name: "uuid_Usuario",UsuarioTable)
    val turno = reference( name: "uuid_Turno",TurnoTable)
    val estadoCompletado=bool( name: "EstadoCompletado")
    val maquinaEncordar = reference( name: "numSerieEncordar", MaquinaEncordarTable).nullable()
    val maquinaPersonalizacion = reference( name: "numSeriePersonalizar", MaquinaPersonalizacionTable).nullable()
    val pedido=reference( name: "uuid_Pedido",PedidosTable)

}
/**
 * Tarea dao
 *
 * @constructor
 *
 * @param id
 */
Alejandro +1
class TareaDao(id: EntityID<Int>): IntEntity(id) {
    Rubén
    companion object : IntEntityClass<TareaDao>(TareaTable)

    var uuidTarea by TareaTable.uuidTarea
    var producto by ProductoDao referencedOn TareaTable.producto
    var precio by TareaTable.precio
    var descripcion by TareaTable.descripcion
    var empleado by UsuarioDao referencedOn TareaTable.empleado
    var turno by TurnoDao referencedOn TareaTable.turno
    var estadoCompletado by TareaTable.estadoCompletado
    var maquinaEncordar by MaquinaEncordarDao optionalReferencedOn TareaTable.maquinaEncordar
    var maquinaPersonalizacion by MaquinaPersonalizacionDao optionalReferencedOn TareaTable.maquinaPersonalizacion
    var pedido by PedidosDao referencedOn TareaTable.pedido

}
```

Esta seria la entidad con más relaciones que tenemos en la práctica ya que la tarea tendrá productos , tendrá un empleado, que es un tipo de usuario, que será el que la realice, tendrá un turno porque ya que un empleado no puede hacer más de dos tareas por turno pues decidimos que cada tarea tenga el turno en la que se está haciendo, para completar esta tarea en algunos casos necesitaremos una de las dos maquinas , por lo que también las añadimos y claramente las tareas pertenecen a un pedido en concreto por lo que también tienen un pedido. Como se puede observar en la imagen de arriba tendremos que referenciar con el campo que queramos de la otra tabla en cuestión para que se cree una relación entre las dos entidades correspondientes.

En JPA lo haríamos de la siguiente manera :

```
data class Tarea(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Int,
    @Serializable(UUIDSerializer::class)
    @Column(name="UUID_Tarea")
    @Type(type = "uuid-char")
    val uuidTarea: UUID,
    @ManyToOne
    @JoinColumn(name = "producto_uuid", referencedColumnName = "UUID_Producto")
    val producto: Producto,
    val precio: Double,
    val descripcion: String,
    @ManyToOne
    @JoinColumn(name = "empleado_uuid", referencedColumnName = "UUID_Usuario")
    val empleado:Usuario,
    @ManyToOne
    @JoinColumn(name = "turno_uuid", referencedColumnName = "UUID_Turno")
    val turno:Turno,
    val estadoCompletado:Boolean,
    @ManyToOne
    @JoinColumn(name = "maquinaEncordar_uuid", referencedColumnName = "numSerie_Encordadora", nullable = true)
    val maquinaEncordar: MaquinaEncordadora?,
    @ManyToOne
    @JoinColumn(name = "maquinaPersonalizacion_uuid", referencedColumnName = "numSerie_Personalizacion", nullable = true)
    val maquinaPersonalizacion: MaquinaPersonalizacion?,

    @ManyToOne
    @JoinColumn(name = "pedidos_uuid", referencedColumnName = "UUID_Pedidos")
    val pedido:Pedidos
): java.io.Serializable {
```

Otra característica a mencionar es que en el joinColumn podemos añadir si es nullable o no, por lo que en este caso como en tareas puede tener una máquina asignada ya sea la de personalización o la de encordar o puede tener las dos a nulas ya que la tarea sería la de adquisición por lo que no se necesitan máquinas para realizar esa tarea.

El siguente modelo es Turno:

```
Alejandro +1
@Serializable
data class Turno(
    val id: Int,
    @Serializable(UUIDSerializer::class)
    val uuidTurno:UUID,
    @Serializable(LocalDateTimeSerializer::class)
    val fechaInicio: LocalDateTime,
    @Serializable(LocalDateTimeSerializer::class)
    val fechaFin:LocalDateTime
)
```

Este sería el modelo en Expose y su correspondiente entidad

```
Alejandro
object TurnoTable : IntIdTable( name: "TURNOS") {
    val uuidTurno = uuid( name: "uuid_Turno")
    val fechaInicio = datetime( name: "FechaInicio")
    val fechaFin = datetime( name: "FechaFin")
}

/**
 * Turno dao
 *
 * @constructor
 *
 * @param id
 */
Alejandro
class TurnoDao(id: EntityID<Int>) : IntEntity(id) {
    Alejandro
    companion object : IntEntityClass<TurnoDao>(TurnoTable)
    var uuidTurno by TurnoTable.uuidTurno
    var fechaInicio by TurnoTable.fechaInicio
    var fechaFin by TurnoTable.fechaFin
}
```

Y en JPA lo hariamos de la siguiente manera:

```
@Serializable
@Entity
@Table(name = "Turnos")
@NamedQueries(
    NamedQuery(name = "Turnos.findAll", query = "SELECT t FROM Turno t"),
    NamedQuery(
        name = "Turno.porUUID",
        query = "SELECT t FROM Turno t WHERE t.uuidTurno = :id"
    ),
)
data class Turno(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Int,
    @Serializable(UUIDSerializer::class)
    @Column(name="UUID_Turno")
    @Type(type = "uuid-char")
    val uuidTurno: UUID,
    @Serializable(LocalDateTimeSerializer::class)
    val fechaInicio: LocalDateTime,
    @Serializable(LocalDateTimeSerializer::class)
    val fechaFin: LocalDateTime
): java.io.Serializable {
```

Como se puede ver que antes lo he mencionado si miras las NamedQuery hay dos, la primera sera el findAll como he explicado antes, y luego hemos añadido otra que será filtrar la entidad por uuid , ya que todas nuestras entidades tienen uuid, como nuestro hibernatemanager se encarga de ciertas operaciones del crudRepository nosotros nos tenemos que encargar de las que faltan, una es el findAll como antes he mencionado y otra seria este findByUUID

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

```
data class Usuario(
    val id:Int,
    @Serializable(UUIDSerializer::class)
    val uuid: UUID,
    val nombre: String,
    val apellido: String,
    val email: String,
    val password: String,
    val perfil: TipoPerfil,
    val turno:Turno?,
) {
```

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

```
object UsuarioTable : IntIdTable( name: "USUARIO") {
    val uuid = uuid( name: "uuid_Usuario").uniqueIndex()
    val nombre = varchar( name: "nombre",  length: 100)
    val apellido = varchar( name: "apellido",  length: 100)
    val email = varchar( name: "email",  length: 100)
    val password = varchar( name: "password",  length: 100)
    val perfil = enumeration<TipoPerfil>( name: "perfil")
    val turno = reference( name: "uuidTurno", TurnoTable).nullable()
}

/**
 * Usuario dao
 *
 * @constructor
 *
 * @param id
 */
Rubén +1 *
class UsuarioDao(id: EntityID<Int>) : IntEntity(id) {
    Rubén
    companion object : IntEntityClass<UsuarioDao>(UsuarioTable)

    var uuid by UsuarioTable.uuid
    var nombre by UsuarioTable.nombre
    var apellido by UsuarioTable.apellido
    var email by UsuarioTable.email
    var password by UsuarioTable.password
    var perfil by UsuarioTable.perfil
    var turno by TurnoDao optionalReferencedOn UsuarioTable.turno
```

Y por ultimo tambien nuestro modelo usuario pero hecho en JPA

```
@Serializable
@Entity
@Table(name = "Usuarios")
@NamedQueries(
    NamedQuery(name = "Usuarios.findAll", query = "SELECT t FROM Usuario t"),
    NamedQuery(
        name = "Usuario.porUUID",
        query = "SELECT t FROM Usuario t WHERE t.uuid = :uuid"
    )
)
data class Usuario(
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id:Int,
    @Serializable(UUIDSerializer::class)
    @Column(name="UUID_Usuario")
    @Type(type = "uuid-char")
    val uuid: UUID,
    val nombre: String,
    val apellido: String,
    val email: String,
    val password: String,
    val perfil: TipoPerfil,
    @OneToOne
    val turno:Turno?,
) : java.io.Serializable {
```

Como se puede ver turno tiene una relacion OneToOne con usuario por lo que la clave la tenemos en la tabla usuario y en la tabla turno no aparece nada en relacion con usuario, si queremos saber que turno tiene un trabajador tendremos que ir a buscarlo a la tabla de Usuario.

```
/**
 * Controlador
 *
 * @property MaquinaEncordarRepositoryImpl
 * @property MaquinaPersonalizacionRepositoryImpl
 * @property PedidosRepositoryImpl
 * @property ProductoRepositoryImpl
 * @property TareaRepositoryImpl
 * @property UsuarioRepositoryImpl
 * @property TurnosRepositoryImpl
 * @property usuarioActual
 * @constructor Create empty Controlador
 */
Rubén García-Redondo Marín +2
class Controlador(
    val MaquinaEncordarRepositoryImpl: MaquinaEncordadoraRepositoryImpl,
    val MaquinaPersonalizacionRepositoryImpl: MaquinaPersonalizacionRepositoryImpl,
    val PedidosRepositoryImpl: PedidosRepositoryImpl,
    val ProductoRepositoryImpl: ProductosRepositoryImpl,
    val TareaRepositoryImpl: TareasRepositoryImpl,
    val UsuarioRepositoryImpl: UsuarioRepositoryImpl,
    val TurnosRepositoryImpl: TurnosRepositoryImpl,
    val usuarioActual: Usuario
) {
```

Utilizaremos el mismo controlador tanto en JPA como Exposed, instanciamos los repositorios y utilizaremos todos los metodos creados en los repositorios. A continuación mostraremos la funcion de guardar turno

```kotlin
/**
 * Guardar tarea
 *
 * @param tarea
 * @return guarda una Tarea
 *
 *En caso de que la tarea no este en un turno se podrá añadir ese turno
 * ,si no, no podrá añadirsea otro turno.
 */
Rubén García-Redondo Marín
fun guardarTarea(tarea: Tarea): Tarea? {
    val temp : List<Tarea> = listarTareas()
    val turnoActual : Turno? = encontrarTurnoUUID(tarea.turno.uuidTurno)
    val empleado : Usuario? = encontrarUsuarioUUID(tarea.empleado.uuid)
    return if (turnoActual != null && empleado != null) {
        val veces : Int = temp.filter { !it.estadoCompletado }.filter { it.turno.uuidTurno == turnoActual.uuidTurno }.count { it.empleado.uuid == empleado.uuid }
        if(veces < 2){
            TareaRepositoryImpl.save(tarea)
        }else{
            logger.debug { "No puede tener 2 tareas en el mismo turno a la vez el empleado con uuid: ${empleado.uuid}"}
            null
        }
    }else{
        logger.debug { "No existe el empleado: ${empleado!!.uuid}"}
        null
    }

}
```

Aquí podemos observar que para guardar una tarea debemos de tener en cuenta que un empleado no puede tener más de 2 tareas sin completar, por lo tanto filtraremos por eso. Después si tiene menos de 2 tareas procederemos a asignar la tarea. Por último si no existiese el empleado le avisaremos que ese empleado no existe.

```
Rubén García-Redondo Marín +1
interface CrudRepository<T, ID> {
    /**
     * Find all
     *
     * @return una lista de T
     */
    Alejandro
    fun findAll(): List<T> // List<T> es una lista de T

    /**
     * Find by id
     *
     * @param id
     * @return devuelve una entidad de tipo T
     */
    Rubén García-Redondo Marín
    fun findById(id: ID): T?

    /**
     * Findby u u i d
     *
     * @param uuid
     * @return devuelve una entidad de Tipo T
     */
    Alejandro
    fun findbyUUID(uuid: UUID): T?

    /**
     * Save
     *
     * @param entity
     * @return guarda una entidad de tipo T
     */
    Rubén García-Redondo Marín
    fun save(entity: T): T

    /**
     * Delete
     *
     * @param entity
     * @return borra una entidad de tipo T
     */
    Rubén García-Redondo Marín
    fun delete(entity: T): Boolean
```

Esta es la interfaz que hemos utilizado en todos los repositorios, el cual se basa en un CRUD. Encontrar todos los objetos de tipo T, encontrar por uuid e id, guardar y borrar.

```
Alejandro +1
private fun insert(entity: Tarea): Tarea {
    logger.debug { "save($entity) - creando" }
    return tareasDao.new() { this: TareaDao
        uuidTarea = entity.uuidTarea
        producto = ProductoDao.findById(entity.producto.id)?: throw ProductoException("El producto no existe con id: ${entity.producto.id}")
        precio = entity.precio
        descripcion = entity.descripcion
        empleado = UsuarioDao.findById(entity.empleado.id)?: throw UsuarioException("El empleado no existe con id: ${entity.empleado.id}")
        turno = TurnoDao.findById(entity.turno.id)?: throw TurnoException("El turno no existe con id: ${entity.turno.id}")
        estadoCompletado = entity.estadoCompletado
        maquinaEncordar = entity.maquinaEncordar?.let { MaquinaEncordarDao.findById(it.id) ?: throw MaquinaException("La Maquina no existe con id: ${entity.maquinaEncordar.id}") }
        maquinaPersonalizacion = entity.maquinaPersonalizacion?.let { MaquinaPersonalizacionDao.findById(it.id)?: throw MaquinaException("La Maquina no existe con id: ${entity.maquinaPersonalizacion.id}") }
        pedido = PedidosDao.findById(entity.pedido.id)?: throw PedidoException("El pedido no existe con id: ${entity.pedido.id}")
    }.fromTareaDaoToTarea()
}
```

Así se insertaría una tarea en Exposed

```
Alejandro +1
override fun findbyUUID(uuid: UUID): Tarea? {
    logger.debug { "findByuuid($uuid)" }
    var tarea: Tarea? = null
    HibernateManager.query {
        val query: TypedQuery<Tarea> = manager.createNamedQuery( name: "Tareas.porUUID", Tarea::class.java)
        query.setParameter( name: "id", uuid)
        tarea=query.singleResult
    }
    return tarea
}
```

Gracias a las “queries” introducidas en el modelo podemos encontrar una tarea por uuid, pasándole como parametro el uuid.