



# Memoria Proyecto DAMLLER

Álvaro Del Val

Mario Bueno

Mohamad El Sayed

## Contenido

Memoria Proyecto DAMLLER .....	0
1- Introducción .....	2
2 - Objetivos de nuestro proyecto .....	2
3 - Metodología y Desarrollo del proyecto .....	3
4 – Planificación de tareas y de responsabilidades .....	3
5 - Tecnologías utilizadas .....	4
6 - Desarrollo e Implementación Web .....	5
Diseño inicial de la web .....	5
Descripción de la Arquitectura: .....	5
7 - Diseño de bases de datos .....	7
Modelo entidad relación: .....	7
Modelo físico .....	8
Tablas Implementadas en DataGrip .....	9
Scripts .....	9
Script para la creación de la base de datos: .....	9
Script para la creación de procedimientos, disparadores y eventos: .....	10
8 - Diagramas y desarrollo del programa .....	12
Diagrama de caso de usos .....	12
Diagrama de secuencia .....	13
Diagrama para crear una nueva cita .....	13
Diagrama para actualizar el trabajador de una cita .....	13
Diagrama para imprimir un informe .....	14
Diagrama de clases de modelo .....	14
9 - Aplicación de Escritorio Gestión de Citas: .....	15
Controladores : .....	16
ModelViews : .....	16
Repositorios .....	18
Servicios .....	19
Storages .....	19
DatabaseManger .....	20
Railway oriented programming .....	20
Inyección de Dependencias con Koin .....	21
Dependencias: .....	22
Interfaz Grafica .....	23
Vista Principal .....	23

## 1- Introducción

DAMLLER busca ocuparse de proporcionar un Sistema Informático completo para la gestión de distintas sedes de ITV.

Este proyecto estará dividido en 3 grandes bloques, una página web, un sistema de bases de datos para almacenar información y una aplicación de escritorio para que los trabajadores gestionen las citas de los clientes.

## 2 - Objetivos de nuestro proyecto

El Objetivo principal de este proyecto es poder recrear el proceso de planificación, de creación, de diseño y de implementación de un proyecto para una ITV para distintos tipos de vehículo ,con la idea de poder recrear un proceso de Sprint de una semana de duración.

Dados los requisitos proporcionados por un cliente deberemos :

- Saber poder identificar los requisitos funcionales, no funcionales y de información del proyecto.
- Realizar la correcta organización del proyecto a través de un sistema de gestión de tareas y responsabilidades con la metodología Kanban, organizando las tareas de la forma más eficiente posible y tener un seguimiento de estas.
- Saber utilizar un sistema de control de versiones como Git a través de GitHub incluyendo el correcto uso de aplicaciones como GitKraken y GitHub Desktop.
- Diseñar e implementar una página web que se encargue de dar a conocer nuestra empresa al público y que permita al usuario pedir una Cita con facilidad.
- Diseñar e implementar una Base de datos que se encargue de almacenar toda la información de las Citas, incluyendo los vehículos y los clientes junto a los informes de estas, incluyendo un historial de cambios a largo del tiempo de los informes almacenados y un sistema de borrado de las citas con más de dos de antigüedad.
- Una aplicación de escritorio que permita a nuestros trabajadores gestionar las citas de los usuarios, encargándose de generar informes con la información de las distintas inspecciones.
- Realizar un análisis tecnológico para el completo desarrollo del proyecto.



### 3 - Metodología y Desarrollo del proyecto

Para llevar a cabo el desarrollo de este proyecto se harán uso de distintos tipos de metodologías ágiles y buenas prácticas en el mundo del desarrollo del software.

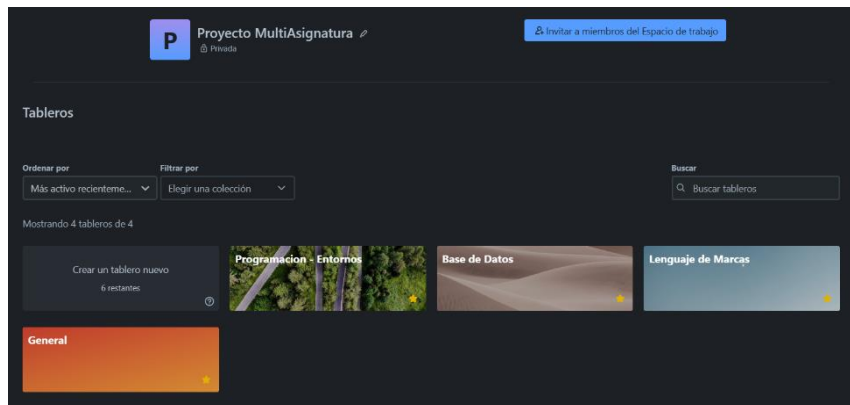
Se utilizará la metodología Kanban para la correcta gestión de tareas y responsabilidades de todos los miembros del equipo.

El empleo de los Principios SOLID para la correcta implementación del código y el futuro mantenimiento y expansión de este, lo que permitirá arreglar los posibles errores y ampliar de una forma cómoda el código a nuevas funcionalidades en el futuro .

El uso de Railway oriented programming para el control y gestión de errores.

### 4 – Planificación de tareas y de responsabilidades

Todo proyecto consta de varias partes y etapas de su desarrollo. Para poder trabajar de una forma cómoda y organizada usaremos Trello , una aplicación web que utiliza la metodología Kanban que nos permite organizar cada parte del proyecto, asignándoles prioridades, fechas de entrega, tags personalizadas , advertencias y mensajes entre otras cosas.



Tendremos distintos tableros, cada uno orientado a una de las grandes partes del proyecto y una individual para los apartados generales. Las tarjetas estarán organizadas en las distintas columnas :

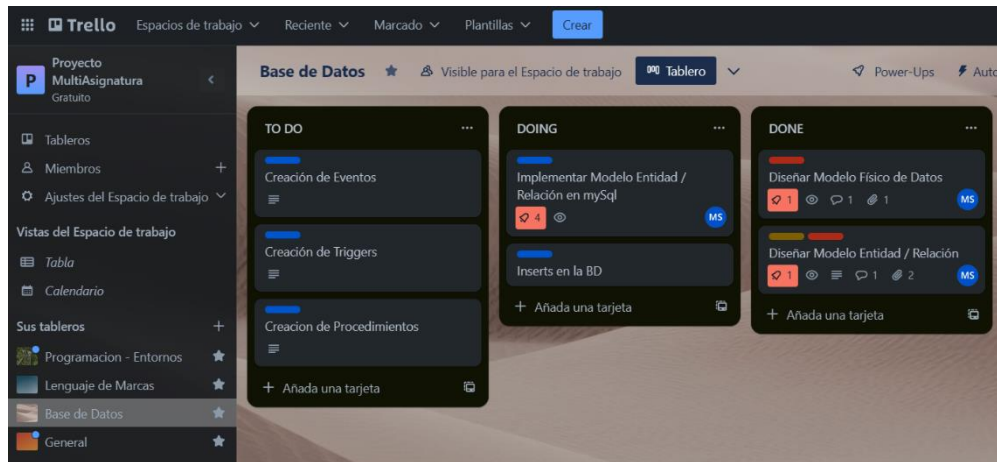
**TO DO** : Una tarea que todavía no ha comenzado ni esta asignada, a la espera que otras tareas sean completadas

**DOING** : Una tarea que se está realizando. Estará asignada a uno o varios integrantes del equipo y se podrán ir haciendo actualizaciones de su estado

**DONE** : Una tarea que se ha terminado, a la espera de revisión.

**Revisión** : Una tarea que se está comprobando, buscando errores en esta y comprobando su correcta Implementación.

**Merged** : Una tarea que ha sido revisada y ya está publicada y fusionada con el resto de las tareas



## 5 - Tecnologías utilizadas

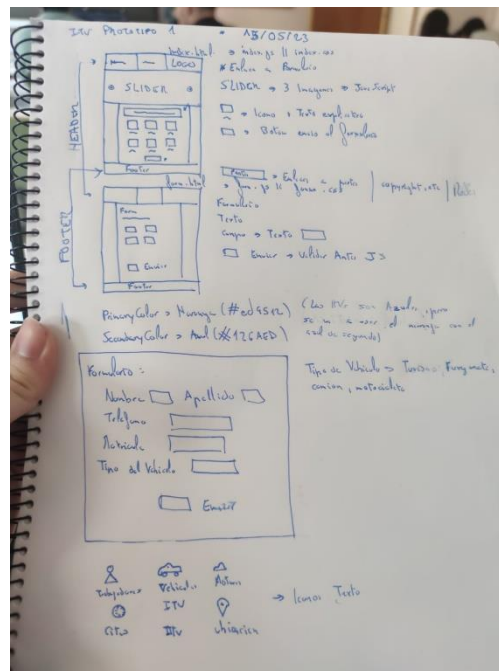
Durante el desarrollo de este proyecto hemos utilizado los siguientes programas y tecnologías:

- Draw.io como programa principal para la creación y diseño de todos los diagramas del proyecto, incluyendo el Modelo E/R, el diagrama de Clases de Modelo, los diagramas de Secuencia entre otros.
- Visual Studio Code como editor de Código para toda la parte de desarrollo web, incluyendo el desarrollo del HTML, del código JavaScript y de los estilos CSS.
- DataGrip y WorkBench para la creación de la base de datos y sus Scripts .
- Kotlin como lenguaje principal para el desarrollo de la Aplicación de escritorio.
- IntelliJ IDEA como el IDE principal para el desarrollo de la Aplicación de Escritorio.
- Programas de control de versiones como GitKraken y GitHub Desktop y GitHub para el repositorio remoto.
- Trello como sistema de Planificación de tareas y responsabilidades.

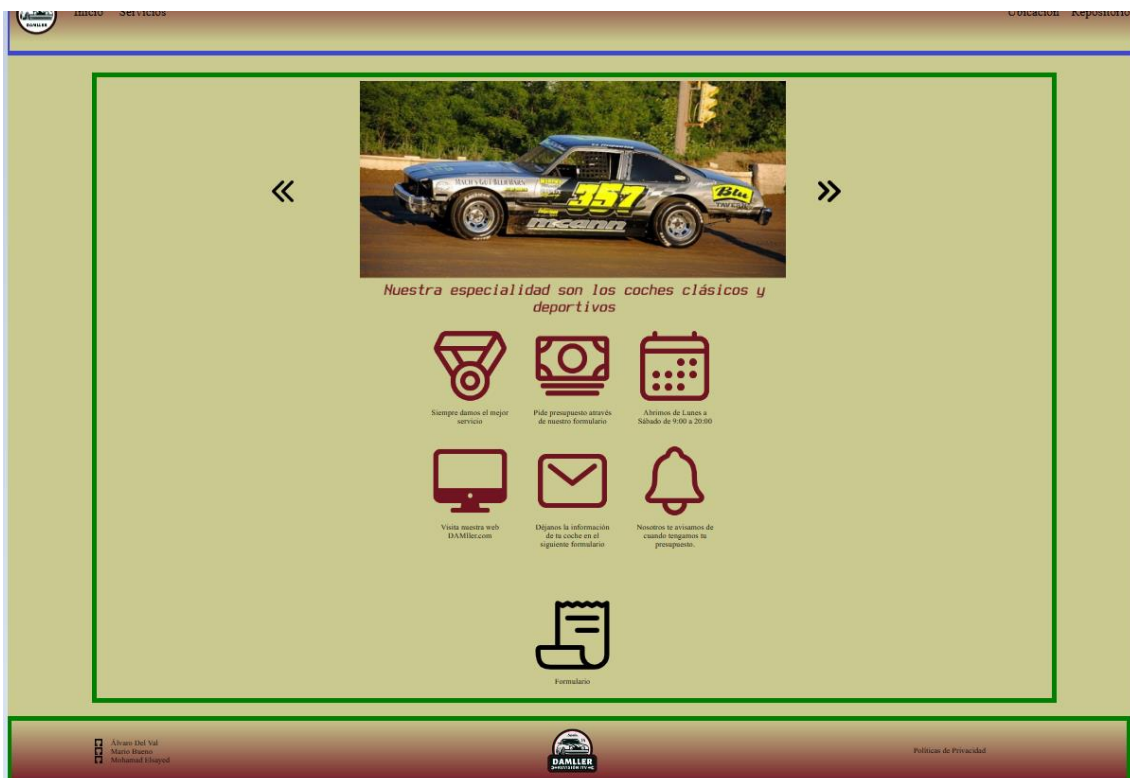


## 6 - Desarrollo e Implementación Web

### Diseño inicial de la web



### Descripción de la Arquitectura:



La página web se compone :

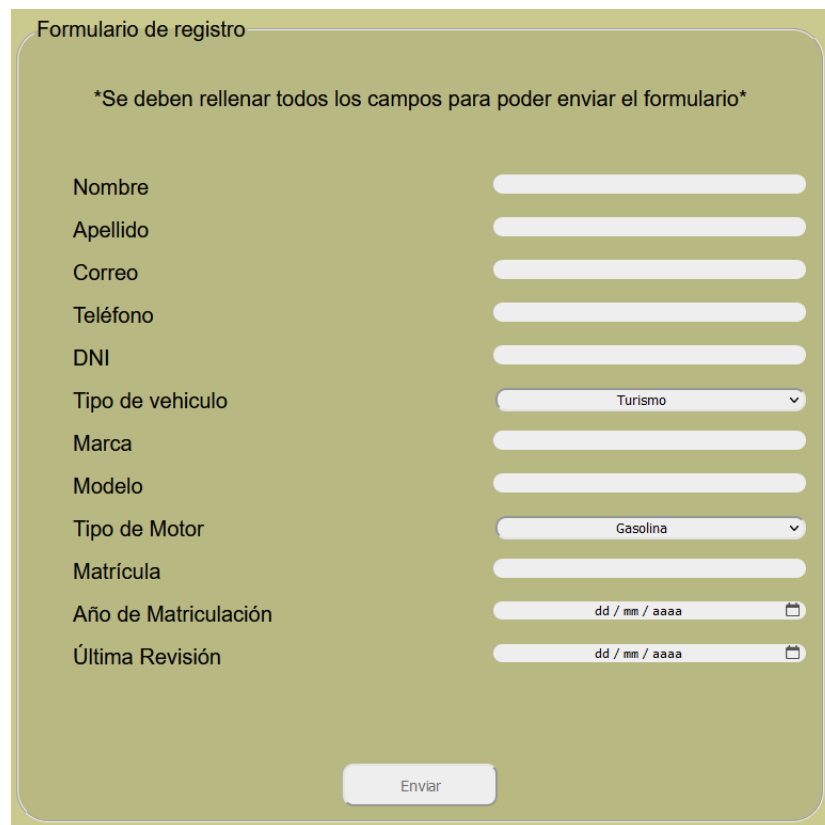
Un encabezado o Header en el que mostraremos un menú con enlaces a nuestro repositorio, un enlace a Google Maps con la ubicación de nuestro instituto y el logo como enlace a la página principal

Un Slider de imágenes programado con JavaScript cambiante con un temporizador y flechas funcionales.

Iconos con una leve descripción de nuestra empresa

Una imagen SVG que sirve de enlace a nuestro formulario.

Un Footer con enlaces a nuestros GitHub personales , nuestro logo Principal y un enlace a nuestra página de Políticas de privacidad



Formulario de registro

\*Se deben rellenar todos los campos para poder enviar el formulario\*

Nombre	<input type="text"/>
Apellido	<input type="text"/>
Correo	<input type="text"/>
Teléfono	<input type="text"/>
DNI	<input type="text"/>
Tipo de vehiculo	<input type="text" value="Turismo"/>
Marca	<input type="text"/>
Modelo	<input type="text"/>
Tipo de Motor	<input type="text" value="Gasolina"/>
Matrícula	<input type="text"/>
Año de Matriculación	<input type="text" value="dd / mm / aaaa"/>
Última Revisión	<input type="text" value="dd / mm / aaaa"/>

Enviar

Un formulario para que el usuario pueda pedir una cita con nosotros con facilidad.

Todo los campos del formulario están validados con JavaScript.

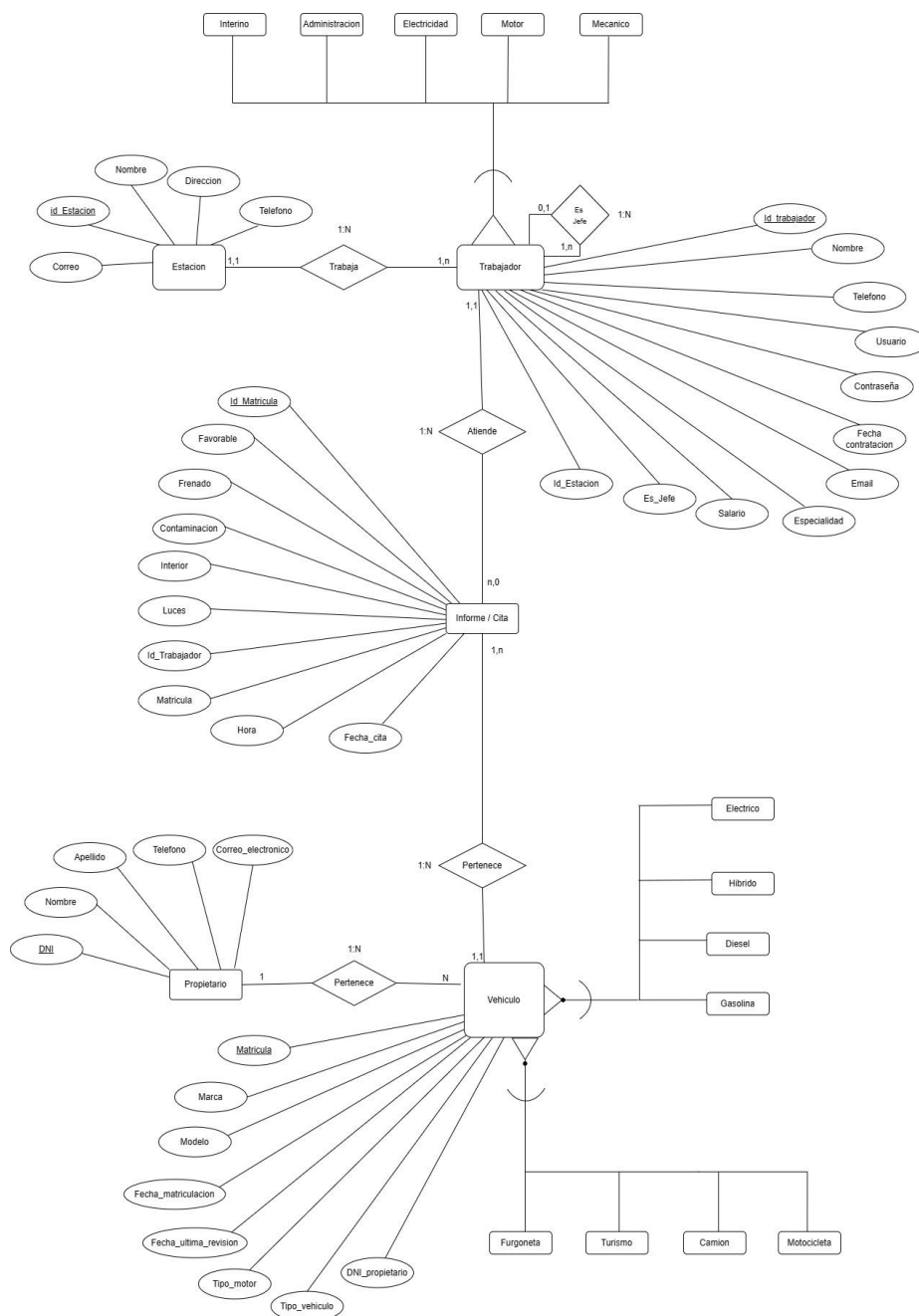
Los datos del formulario serán enviados por correo electrónico para los trabajadores puedan crear una cita

## 7 - Diseño de bases de datos

### Modelo entidad relación:

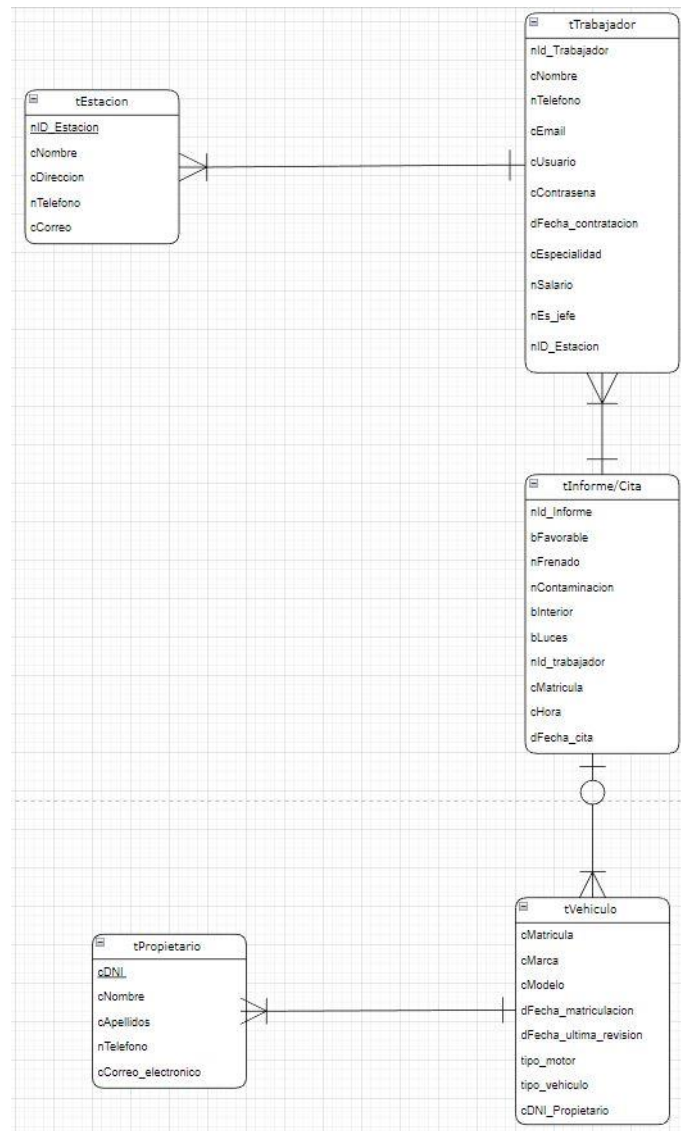
Se ha creado la base de datos y el modelo entidad relación según el enunciado propuesto y se han establecido las siguientes restricciones:

- Un trabajador pertenece a una única estación, y en una estación trabajan varios trabajadores.
- Un trabajador tiene un jefe y un jefe es jefe de varios trabajadores.
- Un trabajador atiende varias citas/informes y una cita/informe solo puede ser atendida por un único trabajador.
- Un informe/cita solo pertenece a un único coche, y un coche puede tener varios informes/citas.
- un vehículo pertenece a un único propietario y un propietario le pueden pertenecer varios vehículos.

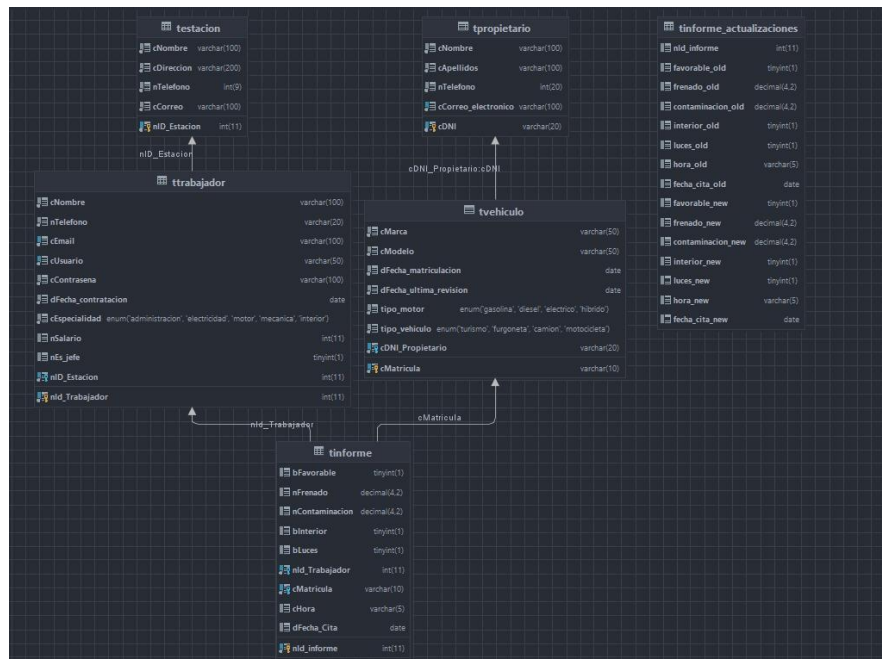




## Modelo físico



## Tablas Implementadas en DataGrip



## Scripts

### Script para la creación de la base de datos:

Se han creado las sentencias SQL para la creación de las tablas con sus respectivos campos y restricciones siguiendo el modelo entidad relación creado, añadiendo restricciones a los campos en los campos que sean necesarios,

```

1  create database IF NOT EXISTS bbITV ;
2  use bbITV;
3  #CREACION DE TABLAS
4  CREATE TABLE IF NOT EXISTS tEstacion ( nID_Estacion ... );
11
12  CREATE TABLE IF NOT EXISTS tTrabajador( nID_Trabajador ... );
28
29  CREATE TABLE IF NOT EXISTS tPropietario ( cDNI VARCHAR... );
36
37
38  CREATE TABLE IF NOT EXISTS tVehiculo ( cMatricula ... );
53
54
55  CREATE TABLE IF NOT EXISTS tInforme( nID_informe ... );
74
75  #Tabla donde se almacenaran los datos cuando salte el disparador Actualizacion_tInforme
76  CREATE TABLE IF NOT EXISTS tInforme_actualizaciones ( nID_informe ... );
93

```

Script para la creación de procedimientos, disparadores y eventos:

1-Procedimiento que liste los trabajadores por estación:

```

178 CREATE PROCEDURE trabajadoresPorEstacion (in Estacion INT)
179 BEGIN
180     DECLARE Nombre varchar (100);
181     DECLARE Telefono int(9);
182     DECLARE Email varchar (100);
183     DECLARE Especialidad VARCHAR (20);
184     DECLARE Salario INT ;
185     DECLARE error BOOLEAN ;
186     DECLARE cur1 CURSOR FOR
187         SELECT cNombre, nTelefono, cEmail, cEspecialidad, nSalario
188         FROM tTrabajador
189         WHERE nID_Estacion=Estacion;
190     DECLARE CONTINUE HANDLER FOR NOT FOUND SET error = TRUE;
191     SET error = FALSE;
192     OPEN cur1;
193     loop1: LOOP
194         FETCH cur1 INTO Nombre, Telefono, Email, Especialidad,Salario;
195         Select Nombre,Telefono,Email,Especialidad,Salario;
196         IF error THEN
197             LEAVE loop1;
198         END IF;
199     end LOOP;
200     CLOSE cur1;
201 END $$

```

2-Disparador para Controlar inspecciones, guardando la información previa y la información que se ha modificado.

```

DELIMITER $$
CREATE TRIGGER Actualizacion_tInforme
AFTER UPDATE ON tInforme
FOR EACH ROW
BEGIN
    INSERT INTO tInforme_actualizaciones (
        nId_informe, favorable_old, frenado_old, contaminacion_old,
        interior_old, luces_old, hora_old, fecha_cita_old,
        favorable_new, frenado_new, contaminacion_new,
        interior_new, luces_new, hora_new, fecha_cita_new
    )
    VALUES (
        OLD.nId_informe, OLD.bFavorable, OLD.nFrenado, OLD.nContaminacion,
        OLD.bInterior, OLD.bLuces, OLD.cHora, OLD.dFecha_Cita,
        NEW.bFavorable, NEW.nFrenado, NEW.nContaminacion,
        NEW.bInterior, NEW.bLuces, NEW.cHora, NEW.dFecha_Cita
    );
END $$
DELIMITER ;

```

## 3-Evento para el borrado de citas cada dos meses

```

# Procedimiento para borrar todas las citas con fecha de mas de dos meses con respecto a la fecha actual
DELIMITER $$
CREATE PROCEDURE BorrarCitasBimestrales()
BEGIN
    DELETE FROM tInforme
    WHERE dFecha_Cita ≤ CURDATE() - INTERVAL 2 MONTH;
END;
# Evento para que se ejecute un procedimiento cada dos meses
CREATE EVENT IF NOT EXISTS EventoBorrar
ON SCHEDULE
    EVERY 2 MONTH
    STARTS CURDATE()
DO
    CALL BorrarCitasBimestrales();

```

## 4-Trigger adicional para el cálculo del salario según la especialidad del trabajador

```

DELIMITER $$

CREATE TRIGGER salario_trabajador
BEFORE INSERT ON tTrabajador
FOR EACH ROW
BEGIN
    CASE NEW.cEspecialidad
        WHEN 'ADMINISTRACION' THEN SET NEW.nSalario = 1650;
        WHEN 'ELECTRICIDAD' THEN SET NEW.nSalario = 1800;
        WHEN 'MOTOR' THEN SET NEW.nSalario = 1700;
        WHEN 'MECANICA' THEN SET NEW.nSalario = 1600;
        WHEN 'INTERIOR' THEN SET NEW.nSalario = 1750;
    END CASE;
END $$

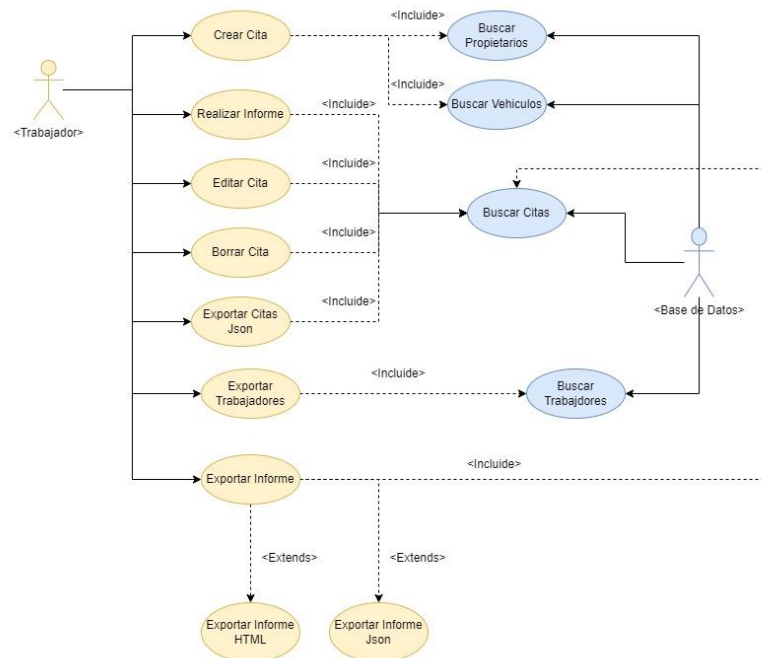
DELIMITER ;

```

## 8 - Diagramas y desarrollo del programa

### Diagrama de caso de usos

Diagrama de casos de uso de un trabajador :



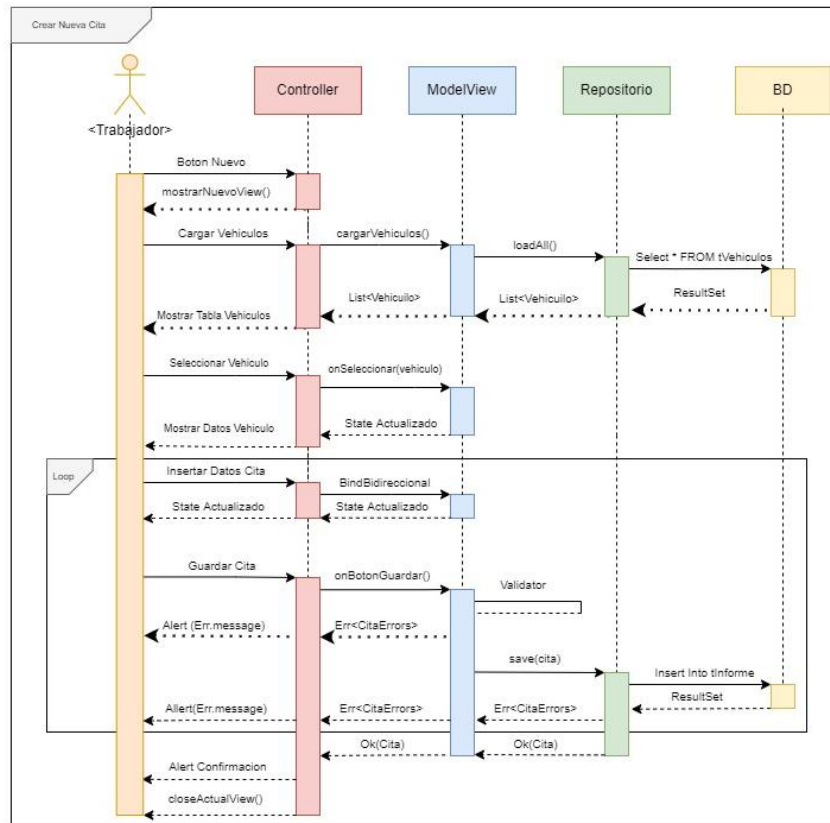
En este diagrama podemos comprobar como un trabajador que usa la aplicación podrá :

- Crear una nueva cita : Para crear una nueva cita el trabajador necesitara los datos de los propietarios y sus trabajadores. Estos serán proporcionados por nuestra base de datos
- Realizar un informe de una cita : El trabajador rellenará los datos del informe y los guardará en nuestra base de datos
- Editar una cita : El trabajador solo podrá cambiar la hora, el día y el trabajador asignado de la cita. En una cita nunca se modificará el vehículo asignado.
- Borrar una cita : El trabajador borrará una cita y su informe (aunque no su historial) de nuestra base de datos
- Exportar todas las citas a JSON : Todas las citas de nuestra base de datos , incluyendo la información de la cita, el propietario , el vehículo y el nombre y correo electrónico del trabajador asignado, serán exportados en formato JSON.
- Exportar un informe concreto a HTML o JSON : El trabajador exportará en formato HTML o JSON toda la información del informe, incluyendo la cita en la que se hizo, el vehículo y propietario y el nombre y correo electrónico del trabajador asignado.
- Exportar todos los trabajadores en formato CSV : Se exportará toda la información de todos los trabajadores de la Base de datos , incluyendo los campos privados excepto la contraseña.

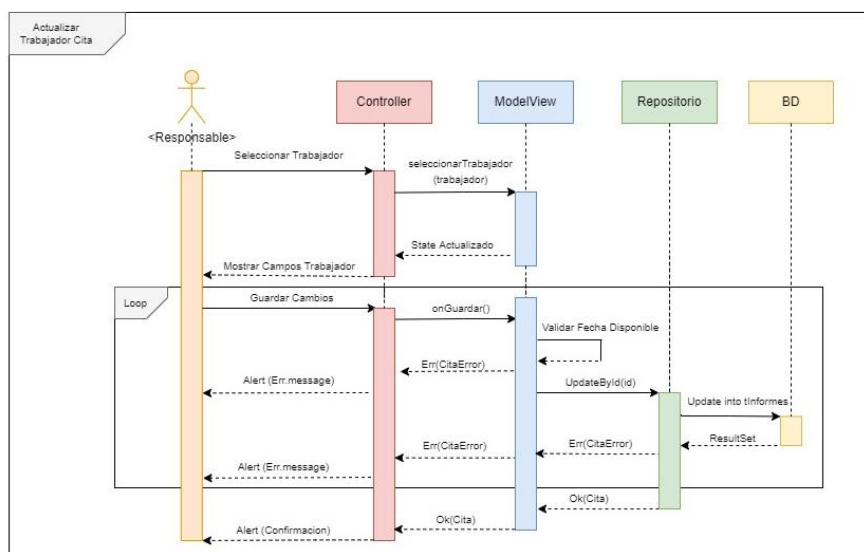
## Diagrama de secuencia

Los diagramas de secuencia nos sirven para mostrar las llamadas que ocurrirán dentro de nuestra aplicación y como se va recorriendo por las distintas clases y funciones de nuestro programa desde que el trabajador inicia una acción hasta que ha sido completada o se termina.

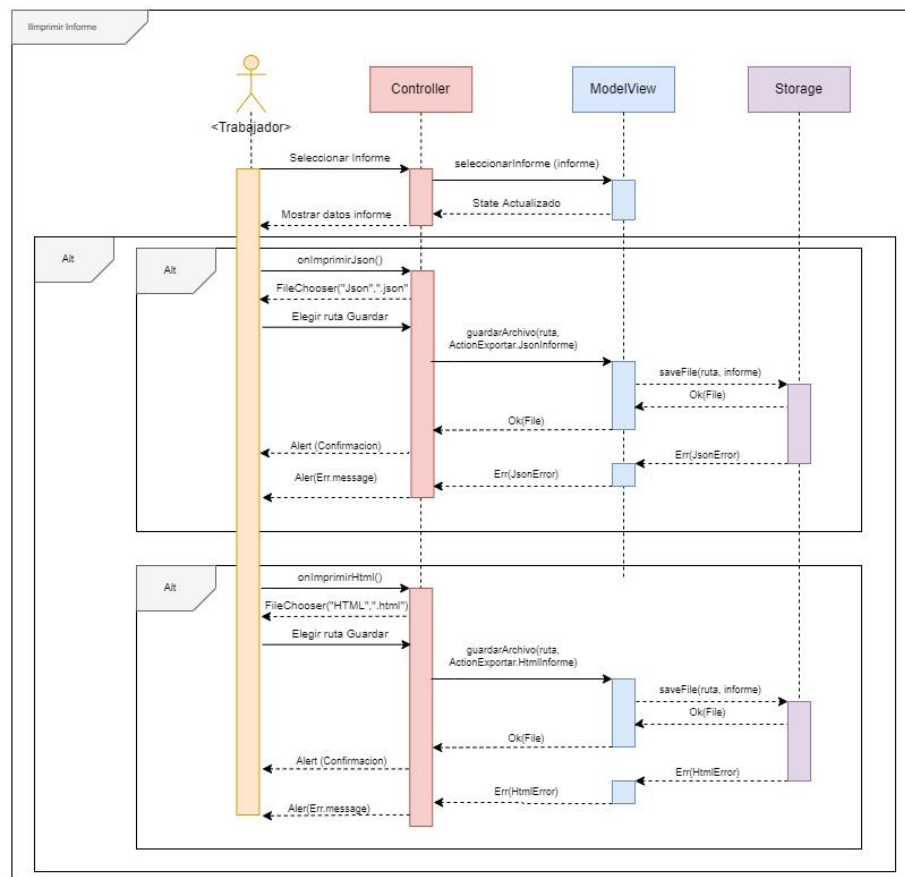
### Diagrama para crear una nueva cita



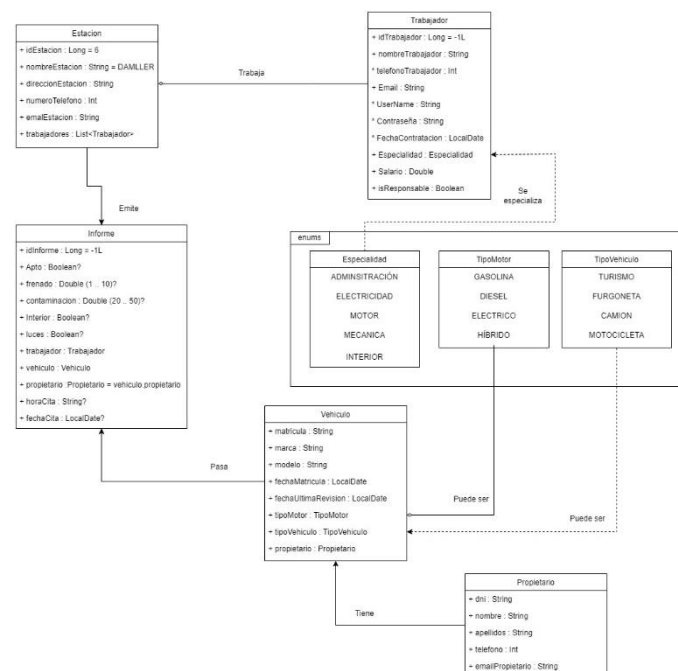
### Diagrama para actualizar el trabajador de una cita



## Diagrama para imprimir un informe

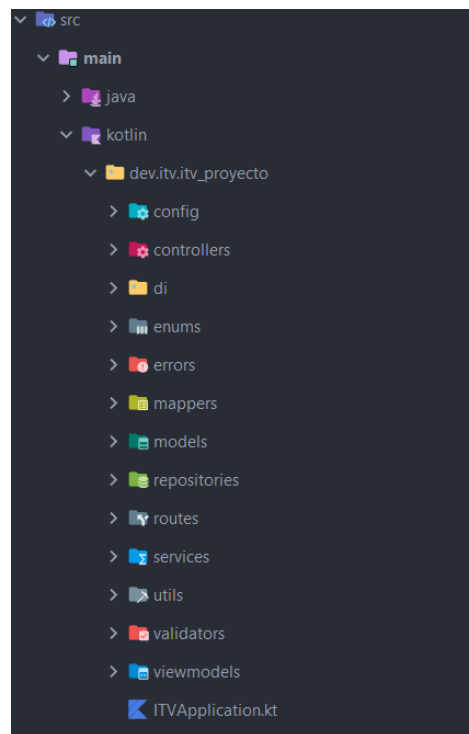
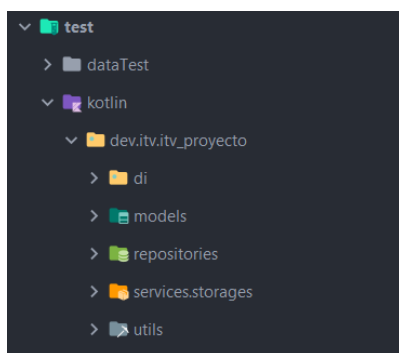


## Diagrama de clases de modelo



## 9 - Aplicación de Escritorio Gestión de Citas:

Para el desarrollo de la Aplicación hemos utilizado la Arquitectura MVVM (Model View ViewModel o Modelo Vista VistaModelo ) que nos permite organizar las clases y funciones del código de una forma controlada y lógica, haciendo que el código esté lo menos acoplado posible, permitiendo su fácil ampliación y mantenimiento en el tiempo.



Como podemos ver todas las clases están organizadas por funcionalidad en su respectivo paquete, lo que mejora la navegabilidad por el código y el fácil acceso a todas las clases y funciones de este igual que su fácil lectura para todas las personas que tengan que mantener este código en un futuro.



## Controladores :

Los controladores se encargarán de gestionar la interfaz gráfica y las partes con las que interactuara el usuario. Todo Controlador pertenece a una vista (un archivo .fxml) que se encarga de definir la parte visual mientras que el controlador gestiona las acciones de la vista junto a su funcionalidad interna.

```
IndexController.kt x
1  package dev.itv.itv_proyecto.controllers
2
3  > import ...
18
19  private val logger = KotlinLogging.logger { }
20  class IndexController : KoinComponent{
21
22  ⚡ private val mainViewModel : MainViewModel by inject()
23  @FXML
24  private lateinit var menuExportarJSON : MenuItem
25  @FXML
26  private lateinit var menuExportarHTML : MenuItem
27  @FXML
28  private lateinit var menuExportarCsv : MenuItem
29  @FXML
30  private lateinit var menuSalir : MenuItem
31  @FXML
32  private lateinit var menuAcercaDe : MenuItem
33  @FXML
34  private lateinit var buscadorNombre : TextField
35  @FXML
36  private lateinit var buscadorTipoVehiculo : ComboBox<String>
37  @FXML
38  private lateinit var buscadorMotor : ComboBox<String>
```

```
views
  </> AcercaDe.fxml
  </> Index.fxml
  </> NuevoEditor.fxml
```

```
controllers
  AcercaController
  IndexController
  NuevoEditorController
```

## ModelViews :

Los modelViews se encargan de interconectar la interfaz gráfica y la lógica interna del programa, a la vez que se encarga de controlar los cambios que se mostrarán en la interfaz gráfica.

```
viewmodels
  EditorViewModel
  MainViewModel
```

```

package dev.itv.itv_proyecto.viewmodels

import ...

private val logger = KotlinLogging.logger { }
class MainViewModel (
    val repositorioTrabajador : TrabajadorRepositoryImpl,
    val repositorioInforme : InformeRepositoryImpl,
    val csvStorage : CsvTrabajadoresStorage,
    val htmlStorage : HtmlInformesStorage,
    val jsonStorage : JsonInformesStorage
) {

    val listaInformes = FXCollections.observableArrayList<Informe>()
    val listaInformesDto = FXCollections.observableArrayList<InformeDto>()
    val tiposMotor = FXCollections.observableArrayList<String>()
    val tiposVehiculos = FXCollections.observableArrayList<String>()
    val listaTrabajadores = FXCollections.observableArrayList<Trabajador>()
    val listaHoras = FXCollections.observableArrayList<String>()

    val state = MainState()

    init {
        logger.info { "Iniciando MainModelView" }

        iniciarInterfaz()
    }
}

```

Por ejemplo, en este modelView tenemos unas cuantas Listas Observables que estarán unidas con un Binding a las distintas tablas, columnas y selectores de la Interfaz gráfica. Al ser observables permiten que cada vez que haya cambios en las listas la interfaz reacciona a estos sin necesidad de ninguna llamada. También tenemos un objeto MainState() que será una representación de la interfaz gráfica hecha clase.

```

data class MainState (
    val dniPropietario : SimpleStringProperty = SimpleStringProperty(),
    val nombrePropietario : SimpleStringProperty = SimpleStringProperty(),
    val apellidosPropietario : SimpleStringProperty = SimpleStringProperty(),
    val telefonoPropietario : SimpleStringProperty = SimpleStringProperty(),
    val emailPropietario : SimpleStringProperty = SimpleStringProperty(),

    val idTrabajador : SimpleStringProperty = SimpleStringProperty(),
    val nombreTrabajador : SimpleStringProperty = SimpleStringProperty(),
    val emailTrabajador : SimpleStringProperty = SimpleStringProperty(),

```

Esta clase se encarga de mostrar los cambios del viewmodel al controlador a través de bindings.

También , el viewModel al ser la interconexión entre los distintos puntos de la aplicación es el sitio ideal para poder validar los datos a través de distintos validators.

```

/**
 * Validá los campos que hacen referencia al state de la parte del informe
 *
 * @param editarState Estado de la Vista
 */
private fun validarInforme(editarState: EditarState): Result<EditarState, ModelViewError.AccionError> {
    if (editarState.frenadoInforme.value.isNullOrBlank()) return Err(ModelViewError.AccionError("No has introducido un valor al Frenado del informe"))
    if (editarState.contaminacionInforme.value.isNullOrBlank()) return Err(ModelViewError.AccionError("No has introducido un valor a la Contaminación del info"))
    if (editarState.trabajadorInforme.value.isNullOrBlank()) return Err(ModelViewError.AccionError("No has introducido un valor al Trabajador del informe"))
    if (editarState.matriculaInforme.value.isNullOrBlank()) return Err(ModelViewError.AccionError("No has introducido un valor a la Matricula del informe"))
    if (editarState.dniPropietario.value.isNullOrBlank()) return Err(ModelViewError.AccionError("No has introducido un valor al DNI del informe"))
    if (editarState.horaCita.value.isNullOrBlank()) return Err(ModelViewError.AccionError("No has introducido una Hora en el informe"))

    return Ok(editarState)
}

```

Este validador comprueba que el trabajador no ha dejado ningún campo vacío. Gracias al uso de los Result (Railway oriented programming) podemos mostrar al usuario los errores deseados.

## Repositorios

Los repositorios se encargan de gestionar las funcionalidades más importantes de la aplicación como la comunicación directa con la Base de datos. Cada repositorio estará asociado un tipo de Modelo concreto . Todos los repositorios implementarán una interfaz (un contrato) que obligará a los repositorios a implementar las funcionalidades deseadas. Si un repositorio concreto tiene funcionalidades extras hará otra interfaz mas especializada.

```
interface ModelsRepository <T, ID, ERROR> {

    fun findById(id : ID) : Result<T, ERROR>

    fun loadAll() : Result<List<T>, ERROR>

}
```

Esta interfaz obligara a implementar sus funciones. Debido a que nuestra aplicación solo gestiona citas y sus informes y no otros modelos esta interfaz solo obligara a implementar las funciones comunes. Esta interfaz hace uso de genéricos para T – Tipo , ID – id y ERROR – Error especializado.

```
interface InformeRepository : ModelsRepository<Informe, Long, InformeErrors> {

    fun saveInforme(informe: Informe) : Result<Informe, InformeErrors>

    fun deleteInformeById(id : Long) : Result<Informe, InformeErrors>

    fun updateInformeById(id : Long, informe: Informe) : Result<Informe, InformeErrors>

}
```

Esta interfaz obliga a implementar 3 funciones de gestión que solo usarán las citas.

```
class InformeRepositoryImpl : InformeRepository, KoinComponent {
    private val logger = KotlinLogging.logger { }
    val manager : DatabaseManager by inject()
    var database = manager.bd

    /**
     * @return Informe guardado o los Posibles errores al guardarlo en la base
     * @param informe El informe que guardaremos en la base de datos
     */
    override fun saveInforme(informe: Informe): Result<Informe, InformeErrors> {

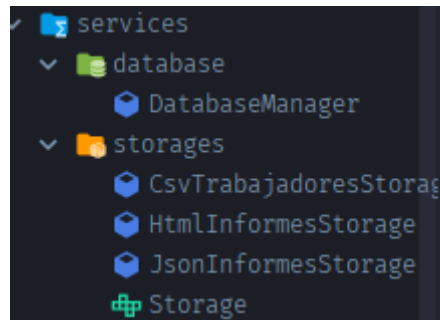
        logger.debug { " InformeRepositoryImpl -- SaveInforme($informe) " }

        val sql : String = """
            INSERT INTO tInforme VALUES (null,?,?,?,?,?,?,?,?,?)
        """
    }
```

Todos los repositorios tendrán una conexión directa con la Base de datos

## Servicios

Los servicios se encargarán de dar funcionalidades extras.



Tenemos dos tipos de servicios :

### Storages

Se encargarán de guardar objeto y clases de nuestro código en distintos archivos

```
fun interface Storage <T> {

    fun saveFile(list: List<T>, url: String): Result<File, StorageErrors>

}
```

```
class JsonInformesStorage() : Storage<Informe> , KoinComponent {

    val manager : DatabaseManager by inject()

    /**
     * Función que guarda los trabajadores en un fichero json
     *
     * @param list Lista de Informes que se guardarán
     * @param url Path donde se guardará el fichero Json
     * @return Devolverá el archivo con los datos o los posibles errores con Result
     */
    override fun saveFile(list: List<Informe>, url: String): Result<File, StorageErrors> {
        val logger : KLogger = KotlinLogging.logger { }
        logger.warn { "StorageInformeJson — SaveFile()" }
        val listaMap : List<InformeDto> = list.map { Mappers().toDto(it) }
        return try {
            val gson : Gson! = GsonBuilder().setPrettyPrinting().create()
            logger.warn { " GsonBuilder Creado " }
            val json : String! = gson.toJson(listaMap)
            logger.warn { " Lista pasada a String " }
            val file : File = File(url).apply { this: File
                writeText(json)
            }
            logger.warn { " File guardado " }
            Ok(file)
        } catch (e : Exception) {
            Err(StorageErrors.JsonStorageError("No se ha podido guardar el fichero: ${e.message}"))
        }
    }
}
```

## DatabaseManger

El databaseManager se encargará de conectar nuestra aplicación con nuestra base de datos

```
class DatabaseManager : KoinComponent{

    private val logger = KotlinLogging.logger {}

    val appConfig: AppConfig by inject()
    // Se necesita una base de datos ya creada para poder conectarse. Las tablas se crean automáticamente si
    // no existen
    var urlBd = "jdbc:mariadb://${appConfig.bdPath}:3306/${appConfig.bdName}"
    val bd: Connection by lazy {
        try {
            Class.forName(className: "com.mysql.cj.jdbc.Driver")
            DriverManager.getConnection(urlBd, user: "root", password: "") ^lazy
        } catch (e: Exception) {
            logger.error(msg: "Error al establecer la conexión con la base de datos", e)
            throw e
        }
    }

    init {
        logger.debug { "Iniciando DataBaseManager()" }
        createTables()
    }
}
```

En el caso de nuestro databaseManager aparte tendremos funciones que se encargarán de crear las tablas si estás no existiesen en la base de datos.

## Railway oriented programming

El railway oriented programming nos permite programar nuestro código como si fuese un camino con distintas intersecciones. Siempre que el programa funcione a la perfección decimos que vamos por el camino correcto o Happy Path ,pero si durante el camino ocurre algún error, cambiaremos de camino y devolveremos al usuario el error que ha ocurrido en el momento de desviarse del camino.

```
override fun deleteInformeById(id: Long): Result<Informe, InformeErrors> {
    logger.debug { " InformeRepositoryImpl -- DeleteInformeById ($id) " }
    val old : Informe = findById(id).onFailure { it: InformeErrors
        return Err(it)
    }.component1()!!

    val sql : String = """
        DELETE FROM tInforme WHERE nId_informe = ?;
    """.trimIndent()

    database.prepareStatement(sql).use { preparedStatement →
        preparedStatement.setLong( parameterIndex: 1,id)
        preparedStatement.executeUpdate()
        preparedStatement.close()
    }

    return Ok(old)
}
```

En esta función, si no se encuentra el Informe con la id buscada se devolverá el error de que no existe (Camino Incorrecto).

Si lo encuentra se devolverá el valor borrado (Camino Correcto o Happy Path).

## Inyección de Dependencias con Koin

En todo el proyecto hemos necesitado múltiples dependencias para nuestras Clases. Para poder hacerlo de una forma más cómoda hemos utilizado Koin, debido a que es fácil de usar y nos permite inyectar las dependencias de una forma rápida y sin mucha complicación, ya que solo necesitaremos declarar nuestras dependencias en un modulo

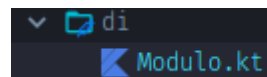
```
val Modulo = module { this: Module
    single { AppConfig() }
    single { DatabaseManager() }
    single { RoutesManager }

    // Repositorios
    factory { InformeRepositoryImpl() }
    factory { PropietarioRepositoryImpl() }
    factory { TrabajadorRepositoryImpl() }
    factory { VehiculoRepositoryImpl() }

    // Storages
    factory { JsonInformesStorage() }
    factory { CsvTrabajadoresStorage() }
    factory { HtmlInformesStorage() }

    // ViewModel
    singleOf(::MainViewModel)
    factoryOf(::EditarViewModel)
}
```

```
startKoin { this: KoinApplication
    logger.debug { " Cargando Modulo Koin " }
    modules(Modulo)
}
```



di  
Modulo.kt

## Dependencias:

```
// Result
implementation("com.michael-bull.kotlin-result:kotlin-result:1.1.17")

// Logger
implementation("io.github.microutils:kotlin-logging-jvm:2.0.11")
implementation("ch.qos.logback:logback-classic:1.4.7")

// Koin
implementation("io.insert-koin:koin-core:3.4.0")
// https://mvnrepository.com/artifact/io.insert-koin/koin-test
implementation("io.insert-koin:koin-test:3.2.0")

// Open Browser
implementation("com.vaadin:open:8.5.0")

// Mockito
// https://mvnrepository.com/artifact/org.mockito/mockito-junit-jupiter
testImplementation("org.mockito:mockito-junit-jupiter:5.3.1")
// https://mvnrepository.com/artifact/org.mockito.kotlin/mockito-kotlin
testImplementation("org.mockito.kotlin:mockito-kotlin:4.1.0")
// https://mvnrepository.com/artifact/org.mockito/mockito-core
testImplementation("org.mockito:mockito-core:5.3.1")

// Conexion BD
// https://mvnrepository.com/artifact/org.mariadb.jdbc/mariadb-java-client
implementation("org.mariadb.jdbc:mariadb-java-client:3.1.4")
implementation("mysql:mysql-connector-java:8.0.28")

// GSON
implementation("com.google.code.gson:gson:2.8.9")
```

- Result : Aunque Kotlin nos proporciona un Result de base, esta dependencia nos proporciona el uso de errores propios y personalizados.
- Logger : Nos permite poder depurar mejor nuestro código, ya que en el momento de ejecución podremos ver todos nuestros loggers según se van activando.
- Open : Nos permite abrir enlaces en nuestra interfaz gráfica.
- Mockito : Dependencia para poder testear con dobles
- Conector MariaDB y mysql : Dependencias necesarias para poder conectar nuestra aplicación con nuestra Base de datos.
- Gson : Dependencia para exportar e importar con comodidad los JSON. Hemos utilizado Gson en vez de Moshi por un problema con el modulo de javaFx, ya que Moshi-Kotlin y Moshi normal hacen uso de dependencias comunes y no pueden funcionar a la vez de una forma comoda.
- Koin : Inyector de dependencias comodo y muy fácil de usar.

## Interfaz Grafica

Para la creación de nuestra interfaz gráfica hemos usado la aplicación de SceneBuilder y JavaFX, ya que nos encontramos cómodos con estos y existe una gran cantidad de documentación en internet para arreglar posibles errores y añadir diversas funcionalidades.

### Vista Principal

ID	DNI	Nombre	Apellidos	Telefono	Matricula	Tipo Vehiculo	Tipo Motor	ID Trabajador	Resultado
1	21232345O	Ricardo	Lopez	876765456	2343C43	TURISMO	ELECTRICO	1	false
2	56751234P	Alonso	Perez	848475121	AAAAAOP	CAMION	DIESEL	1	false
7	66538954P	Jesus	Arce Velazquez	968345768	OPELCORSA	TURISMO	ELECTRICO	1	true
11	56745674Y	Lumis	Rick	564649586	POER345	FURGONETA	HIBRIDO	2	true
12	45659834T	Pedro	Reverte	456234675	RICA3245	TURISMO	GASOLINA	1	true
13	94565486P	Luis Angel	Los Santos	345765345	MORDEMVP	MOTOCICLETA	ELECTRICO	2	true
14	94565486P	Luis Angel	Los Santos	345765345	ARAALOP	CAMION	DIESEL	2	false
15	74685958P	Luis	Barreras	457126456	POKE456	TURISMO	ELECTRICO	2	true
17	58476512P	Mario	Luque	567475645	8745RTA	FURGONETA	GASOLINA	3	true

Tenemos una tabla con todas las citas, los datos del propietario y vehículo al que está asignado al igual que el resultado del informe. Al pulsa una de las citas se rellenan los datos de la parte inferior con toda la información.

