

MÁSTER EN FULL STACK DEVELOPER

FULL STACK APPLICATION

AUTOR: Guillermo Martín Guevara

DIRECTOR:

FECHA: 06/07/2023

Índice

[4.Objetivos 4](#_Toc139540720)

[5.Descripción 5](#_Toc139540721)

[6.Cronología 7](#_Toc139540722)

[7.Detalle y justificación 8](#_Toc139540723)

[8.Trabajo futuro 9](#_Toc139540724)

[9.Conclusiones 10](#_Toc139540725)

# 4.Objetivos

El objetivo de este TFM es crear una aplicación full stack para comparar coches.

Se podrá utilizar nuestra aplicación en dispositivos de cualquier tamaño.

El usuario podrá ver que coches hay disponibles en la aplicación, que serán cargados previamente por el usuario con sus datos más importantes cómo potencia, cilindrada ect y una imagen.

Se pueden comparar dos coches entre sí y además también se pueden ordenar según sus características.

Desde el punto de vista funcional, el objetivo es que tanto el backend como el frontend funcionen perfectamente y de manera segura para el cliente.

Para asegurar la seguridad de nuestra aplicación se utilizarán librerías destinadas a ello.

Otro objetivo, es que la aplicación sea responsive y que el diseño se adapte a cualquier dispositivo, sin importar el tamaño.

# 5.Descripción

Para desarrollar nuestra aplicación, hemos diferenciado el backend del frontend.

Para el backend hemos elegido Node.js y express, para crear una api rest, que será consumida por nuestro frontend.

Para aumentar la seguridad de nuestra aplicación hemos utilizado la librería JWT, que es un estándar para transmitir información de forma segura en internet, por medio de archivos en formato JSON.

Para el frontend hemos utilizado Angular para dispositivos más grandes que móviles, y Android Studio (Flutter) para dispositivos móviles.

Ambos, consumirán la misma api, ya que tendrán las mismas funcionalidades.

El backend será testeado con Postman.

Una página web considero que tiene que funcionar bien pero también se tiene que ver atractiva para el cliente.

Para ello es importante hacer uso de estilos que hagan que los usuarios tengan una buena experiencia al navegar en la aplicación.

Considero que cuanto más sencilla sea mejor.

Hay que tener en cuenta a cualquier tipo de usuario.

# 6.Cronología

Lo primero que hice fue pensar que tecnologías iba a usar para el backend y para el frontend.

Lógicamente, iba a usar tecnologías vistas en el máster.

Para el backend decidí usar node js y express.

Para testear este backend decidí usar Postman.

Para el frontend decidí usar angular para dispositivos más grandes que los móviles, y Android studio para móviles.

A la hora de realizar mi solución empecé por el backend.

Pensé que era lo más lógico, ya que para hacer el frontend iba a necesitar llamar a backend para obtener datos y pintarlos por la pantalla.

El backend es el mismo para angular como para Android studio, lo que facilita mucho las cosas.

Una vez que realizaba mis servicios en el backend los testeaba con postman para ver que estaban bien.

No lo he mencionado pero también tuve que crear las tablas en la base de datos mongodb.

Cuando creaba los schemas en el back, me metía en el mongodb y creaba las tablas, de forma que cuadrarán los tipos de datos ect.

En cuanto al frontend empecé por angular y finalmente desarrollé con Android Studio.

Una vez desarrolle el frontend lo teste desde mi localhost, usando todos los servicios que tiene la aplicación.

Finalmente realicé la memoria del trabajo.

# 7.Detalle y justificación

Voy a empezar hablando por el backend.

Como he mencionado antes está solución de back está desarrollada en node js y express, debido a que son las tencologías de backend que más me han gustado durante el máster.

### Resumen Funcional de nuestro backend:

* **Conecta a MongoDB** para almacenar y recuperar datos.
* **Middleware de CORS** para permitir solicitudes de orígenes cruzados.
* **Rutas protegidas** que requieren autenticación mediante token.
* **Rutas de API** para manejar autenticación de usuarios, gestión de coches, y más.

Vamos a ver que es node js.

Node.js es un entorno de tiempo de ejecución de JavaScript (de ahí su terminación en .js haciendo alusión al lenguaje JavaScript). Este entorno de tiempo de ejecución en tiempo real incluye todo lo que se necesita para ejecutar un programa escrito en JavaScript. También aporta muchos beneficios y soluciona muchísimos problemas, por lo que sería más que interesante realizar nuestro curso de Node.js para obtener las bases, conceptos y habilidades necesarias que nos motiven a profundizar en sus opciones e iniciar la programación.

Node.js fue creado por los desarrolladores originales de JavaScript. Lo transformaron de algo que solo podía ejecutarse en el navegador en algo que se podría ejecutar en los ordenadores como si de aplicaciones independientes se tratara. Gracias a Node.js se puede ir un paso más allá en la programación con JavaScript no solo creando sitios web interactivos, sino teniendo la capacidad de hacer cosas que otros lenguajes de secuencia de comandos como Python pueden crear.

Tanto JavaScript como Node.js se ejecutan en el motor de tiempo de ejecución JavaScript V8 (V8 es el nombre del motor de JavaScript que alimenta Google Chrome. Es lo que toma nuestro JavaScript y lo ejecuta mientras navega con Chrome). Este motor coge el código JavaScript y lo convierte en un código de máquina más rápido. El código de máquina es un código de nivel más bajo que la computadora puede ejecutar sin necesidad de interpretarlo primero, ignorando la compilación y por lo tanto aumentando su velocidad.

### 1. **Importación de Módulos y Configuración Básica**

* **Express**: Express es un framework de desarrollo para Node.js y actualmente casi estándar de facto para la mayoría de aplicaciones desarrolladas en Node.js en la web de hoy.
* **Mongoose**: Biblioteca para interactuar con MongoDB.
* **Body-Parser**: Middleware para analizar cuerpos de solicitudes.
* **Cors**: Middleware para permitir solicitudes entre orígenes cruzados.
* **dotenv**: Para cargar variables de entorno desde un archivo .env.

### 2. **Configuración del Middleware**

* **Body-Parser**: Configurado para analizar cuerpos de solicitudes URL-encoded y JSON.
* **Cors**: Habilitado para permitir solicitudes de orígenes cruzados.

### 3. **Conexión a la Base de Datos**

* Conexión a una base de datos MongoDB en mongodb://127.0.0.1/tfm.
* Mongoose se configura para usar el nuevo analizador de URL y la topología unificada.
* Mensajes de consola para confirmar la conexión o reportar errores.

### 4. **Importación de Rutas**

* authRoutes: Rutas relacionadas con la autenticación.
* dashboadRoutes: Rutas relacionadas con el dashboard.
* newcarRoutes: Rutas para la creación de coches nuevos.
* list\_carsRoutes: Rutas para listar coches.
* loginRoutes: Rutas para el login.
* verifyToken: Middleware para validar tokens de autenticación.

### 5. **Configuración de Middleware para las Rutas**

* Las rutas del dashboard y de usuario requieren verificación de token para el acceso (verifyToken).
* Rutas configuradas para manejar solicitudes en /api/dashboard, /api/user, /api/newcar, /api/login, y /api/list\_cars.

### 6. **Inicio del Servidor**

* El servidor escucha en un puerto definido en las variables de entorno (PORT) o en el puerto 4001 si no se define.

Para la seguridad en la aplicación he ha hecho uso de la librería jwt.

JSON Web Token (JWT) sirve para transmitir información de forma segura en internet, por medio de archivos en formato JSON, que es un tipo de archivo de texto plano con el cual se pueden crear parámetros y asignarles un valor.

Tanto en la parte del backend como en la parte del frontend hemos incluido en algunas llamadas la necesidad de verificar mediante token estas mismas.

En el backend, la ruta de registro lleva consigo la función verifytoken, esto quiere decir que para que un usuario se quiera registrar, va a tener que pasar un token correcto por el headers.

Este token se crea al hacer login en la aplicación.

Más tarde veremos como se gestiona esto desde el frontend.

Las posibilidades que presenta nuestro backend son las siguientes:

-Dar de alta a un usuario

-Hacer login

-Dar de alta un vehículo

-Ver todos los vehículos

-Ver los vehículos por propiedad asc y desc(Ejemplo: Precio desc).

-Comparar dos vehículos

-Eliminar un vehículo

Todas estas posibilidades serán utilizadas en el frontend.

Básicamente nuestro backend se compone de una carpeta rutas, donde están todas las rutas que proveen estas posibilidades.

Es importante tener en cuenta el método a utilizar.

Hemos usado los métodos post, get y delete.

Para el registro de usuario, donde se necesita una contraseña, hemos utilizado la librería bcrypt, para encriptar esta misma y mandarla a la base de datos.

Con la librería Joi hacemos que los campos email y contraseña para este caso, tengas unas condiciones determinadas (Que sean campos requeridos, con un mínimo de caracteres ect).

En este backend también se produce la conexión a la base de datos mongodb, para poder enviar nuestros datos a la misma.

Para la gestión de estos datos hice uso de mongodb compass.

La otra carpeta es la de los modelos, donde se encuentran nuestros schemas.

Hemos hecho uso de Mongoose, donde un **Schema (Esquema)** es la estructura que indica cual es la forma en la que están estructurados los documentos que se almacenan en una colección de la bdd MongoDB. Cada schema está **compuesto por campos y tipos de datos permitidos**, además de opciones para llevar a cabo validación de dichos documentos.

El nombre de la bdd contraseña y otros datos necesarios para la conexión están en el fihero.env

Crear un nuevo vehículo

A continuación se explica cómo vamos a dar de alta un nuevo vehículo en nuestro back-end y que librerías hemos utilizado:

### 1. **Importación de Módulos y Configuración Básica**

* **Express Router**: Para definir rutas de la aplicación.
* **Multer**: Middleware para manejar la carga de archivos.
* **Joi**: Biblioteca para la validación de datos.
* **Modelos de Mongoose**: Importa los modelos Car y User.

### 2. **Definición del Esquema de Validación con Joi**

* schemaCar: Define el esquema para validar los datos del coche, asegurando que todos los campos requeridos están presentes y cumplen con las restricciones de tipo y longitud.

### 3. **Configuración de Multer para la Carga de Archivos**

* **Memoria**: Configura multer para almacenar los archivos en memoria.
* upload.single('imagen'): Define que la ruta aceptará un solo archivo en el campo imagen.

### 4. **Definición de la Ruta POST para Crear un Nuevo Coche**

* **Validación de Datos**: Usa Joi para validar los datos del coche antes de proceder.
* **Creación del Objeto Car**: Crea un nuevo objeto Car con los datos del cuerpo de la solicitud y la imagen cargada.
* **Guardado en la Base de Datos**: Intenta guardar el nuevo coche en la base de datos.
* **Manejo de Errores**: Responde con un error si la validación falla o si ocurre un error al guardar el coche.

Listado de coches

En este código tenemos varias rutas para manejar las operaciones CRUD (Crear, Leer, Actualizar, Borrar) relacionadas con los coches en una base de datos MongoDB usando Express. Aquí hay una explicación detallada de cada parte del código:

### 1. **Importación de Módulos y Configuración Básica**

* **Router de Express**: Se usa para definir y agrupar rutas relacionadas con los coches.
* **Modelos de Mongoose**: Car se importa desde el archivo ../models/Cars. Este modelo se utiliza para interactuar con la colección de coches en la base de datos MongoDB.

const router = require("express").Router();

const Car = require('../models/Cars');

const db = require('../models/Cars');

### 2. **Ruta para Obtener Todos los Coches**

* **GET /**: Esta ruta se utiliza para obtener todos los coches de la base de datos.
* **Manejo de Errores**: Si ocurre un error durante la operación de búsqueda, se pasa al siguiente middleware de manejo de errores.

router.get('/', async (req, res, next) => {

try {

const allcars = await Car.find({});

res.json(allcars);

} catch (error) {

next(error);

}

});

### 3. **Ruta para Obtener la Imagen de un Coche por ID**

* **GET /imagen/:id**: Esta ruta obtiene la imagen de un coche específico usando su ID.
* **Configuración del Tipo de Contenido**: La respuesta se configura para tener el tipo de contenido adecuado (por ejemplo, image/jpeg).
* **Envío de la Imagen**: La imagen se envía como una respuesta binaria.
* **Manejo de Errores**: Si ocurre un error durante la operación de búsqueda, se responde con el error.

router.get('/imagen/:id', async (req, res) => {

let id = req.params.id;

Car.findById(id).then((rpta) => {

res.set('Content-Type', rpta.imagen.contentType);

return res.send(rpta.imagen.data);

}).catch((err) => {

res.json({ err : err });

});

});

### 4. **Ruta para Eliminar un Coche por ID**

* **DELETE /delete/:id**: Esta ruta elimina un coche específico usando su ID.
* **Manejo de Errores**: Si ocurre un error durante la operación de eliminación, se pasa al siguiente middleware de manejo de errores.

router.delete('/delete/:id', async (req, res, next) => {

try {

console.log(req.params.id);

const allcars = await Car.deleteOne({\_id: req.params.id});

res.json(allcars);

} catch (error) {

next(error);

}

});

### 5. **Exportación del Router**

* El router se exporta para que pueda ser utilizado en otras partes de la aplicación.

module.exports = router;

### Resumen Funcional:

1. **Obtener Todos los Coches**: La ruta GET / obtiene y devuelve todos los coches almacenados en la base de datos.
2. **Obtener Imagen de un Coche**: La ruta GET /imagen/:id devuelve la imagen de un coche específico identificado por su ID.
3. **Eliminar un Coche**: La ruta DELETE /delete/:id elimina un coche específico identificado por su ID.
4. **Manejo de Errores**: Todas las rutas incluyen manejo de errores para capturar y manejar cualquier problema que ocurra durante las operaciones de base de datos.

Autenticación:

Este código crea un módulo de Express que maneja la ruta para registrar nuevos usuarios en la aplicación. Aquí hay una explicación detallada de cada parte del código:

### 1. **Importación de Módulos y Configuración Inicial**

* **Express Router**: Se usa para definir rutas específicas relacionadas con el registro de usuarios.
* **Modelo de Usuario**: User se importa desde ../models/User para interactuar con la colección de usuarios en la base de datos MongoDB.
* **CORS**: Permite compartir recursos entre diferentes dominios (Cross-Origin Resource Sharing).
* **Joi**: Se usa para la validación de los datos del usuario.
* **bcrypt**: Se usa para hashear las contraseñas de los usuarios antes de almacenarlas en la base de datos.
* **jsonwebtoken**: Se usa para generar y verificar tokens JWT para autenticación.
* **app**: Se importa desde ../../app, aunque en este código no se utiliza explícitamente.

const router = require('express').Router();

const User = require('../models/User');

const cors = require('cors');

const Joi = require('@hapi/joi');

const bcrypt = require('bcrypt');

const jwt = require('jsonwebtoken');

const app = require('../../app');

### 2. **Definición del Esquema de Validación**

* **schemaRegister**: Define las reglas de validación para los datos del usuario utilizando Joi. Cada campo tiene restricciones específicas (por ejemplo, longitud mínima y máxima).

const schemaRegister = Joi.object({

nombre: Joi.string().min(2).max(255).required(),

apellido: Joi.string().min(2).max(255).required(),

email: Joi.string().min(6).max(255).required().email(),

contrasena: Joi.string().min(6).max(1024).required()

});

### 3. **Ruta de Registro de Usuarios**

* **CORS**: Permite solicitudes desde http://localhost:4200.
* **Validación de Datos**: Usa Joi para validar los datos del usuario. Si hay un error, responde con un mensaje de error.
* **Verificación de Email Único**: Verifica si el email ya está registrado en la base de datos.
* **Hash de Contraseña**: Usa bcrypt para generar un hash seguro de la contraseña del usuario.
* **Creación de Usuario**: Crea un nuevo usuario con los datos proporcionados y la contraseña hasheada.
* **Guardado del Usuario**: Guarda el usuario en la base de datos y responde con los datos del usuario si el registro es exitoso.

router.post('/register', async (req, res) => {

res.set('Access-Control-Allow-Origin', 'http://localhost:4200');

// Validar usuario

const { error } = schemaRegister.validate(req.body);

if (error) {

return res.status(400).json({ error: error.details[0].message });

}

// Validar email único

const isEmailExist = await User.findOne({ email: req.body.email });

if (isEmailExist) {

return res.status(400).json({ error: 'Email ya registrado' });

}

// Hash contraseña

const salt = await bcrypt.genSalt(10);

const contrasena = await bcrypt.hash(req.body.contrasena, salt);

const user = new User({

nombre: req.body.nombre,

apellido: req.body.apellido,

email: req.body.email,

contrasena: contrasena

});

try {

const saveUser = await user.save();

res.json({

error: null,

data: saveUser,

});

} catch (error) {

res.status(400).json(error);

}

});

### Resumen Funcional:

1. **Validación de Datos del Usuario**: Usa Joi para asegurar que los datos proporcionados cumplan con las reglas definidas.
2. **Verificación de Email Único**: Asegura que no haya usuarios registrados con el mismo email.
3. **Hash de Contraseña**: Usa bcrypt para asegurar que las contraseñas se almacenen de forma segura.
4. **Creación y Guardado de Usuario**: Crea un nuevo usuario y lo guarda en la base de datos.
5. **Manejo de Errores**: Proporciona retroalimentación adecuada si ocurre algún error durante el proceso de registro.

Para terminar de hablar sobre este backend, hay que tener en cuenta también que en nuestro package.json tenemos todas las dependencias de nuestro proyecto con su versión correspondiente.

Ahora voy a hablar del Frontend.

Primero voy hablar de mi solución en angular.

Angular es una plataforma de desarrollo, construida sobre TypeScript. Es un framework basado en componentes para crear aplicaciones web escalables. Una colección de bibliotecas bien integradas que cubren una amplia variedad de características, que incluyen enrutamiento, administración de formularios, comunicación cliente-servidor y más. Un conjunto de herramientas para desarrolladores que permiten desarrollar, compilar, probar y actualizar el código fuente de la aplicación.

Con Angular, estás aprovechando una plataforma que puede escalar desde proyectos de un solo desarrollador hasta aplicaciones de nivel empresarial. Angular está diseñado para que la actualización sea lo más sencilla posible, y así aprovechar los últimos desarrollos con un mínimo de esfuerzo. Lo mejor de todo es que el ecosistema Angular consta de un grupo de casi 2 millones de desarrolladores, autores de bibliotecas y creadores de contenido.

La idea es que tanto la solución con angular como la solución con Android studio tengan las mismas posibilidades, es decir llamen a los mismos métodos de nuestro backend.

Lo primero que hicé fue realizar la navbar de nuestra aplicación.

Esto se hace en angular en nuestro app.component.html.

En esta barra tendremos la posibilidad de ir a todas las páginas de la aplicación como puede ser la página del listado de coches.

Las posibilidades son:

-Hacer login

-Registro de un usuario

-Ver listado de coches y filtrado por categoría asc y comparar dos coches selecionados

-Logout

-Crear un coche con sus características, entre ellas una imagen.

La mayoría de las rutas están protegidas mediante la función userGuard, esto significa que solo se podrá acceder a las mismas si el token está presente y es correcto.

Sino se redirecciona a la página de login.

Para la gestión del token en la aplicación, se ha hecho uso de la librería CookieService.

Con esto lo que se consigue es que el token que es devuelto por el backend se guarde en las cookies del navegador.

Para hacer logout, se elimina esta cookie y el usuario tendrá que volver a hacer login si quiere introducirse en las otras páginas.

Este front está dividido principalmente en componentes modelos y servicios.

Para crear componentes y servicios lo he hecho desde la consola usando el comando ng generate component o ng generate service.

En los servicios las llamadas a nuestro backend se produce gracias a la librería HttpClient.

En otro archivo tenemos todas las url a las cuales se va a peticionar, las cuales han sido previamente testeadas y sabemos que van a funcionar bien.

Al igual que en el backend, aquí también tenemos nuestra carpeta modelos.

En este caso tenemos tres modelos.

Uno para el coche, otro para el login y otro para el usuario.

Para que la aplicación se vea bonita se han hecho uso de estilos css.

Por un lado tenemos los estilos del app.component.css y por otro cuando he creído oportuno también he cambiado los css directamente en el componente css.

Todas las rutas están en app.routing.ts y como hemos dicho antes, usando userGuard.

En la carpeta assets/images tenemos las fotos de nuestra aplicación, en este caso solo tenemos el background.

Para crear la lista de coches decidí que se cargasen 4 filas por página, es decir añadimos la capacidad de paginación en este componente.

Y creamos un botones por categoría para filtrar los datos de manera que filtremos de manera ascendente .

Para que el usuario sepa que se ha logeado correctamente en la aplicación he creado un pop-up que desaparece a los 3 segundos o dando click en la pantalla

También lo he creado cuando se da de alta un coche.

Cuando se crea un coche desde el servicio correspondiente, se crea una respuesta, dentro de esta tenemos el campo error.

Cuando se ha registrado un coche de forma satisfactoria, el error es null.

Es por eso, que usamos este popup usando esta condición.

Tener en cuenta que durante el desarrollo de nuestra aplicación he tenido manejar muchos errores.

Todos los errores tienen su cógido de error y por ello es más fácil solucionarlos.

Un ejemplo puede ser el error 500(Internal server error).

Un archivo muy importante en nuestra aplicación es el package.json, donde están todas las dependencias de nuestra aplicación.

Al inicio de crear está solución, instale todas las dependencias que necesitaba.

Me gustaría destacar bootstrap .

Bootstrap es un framework front-end utilizado para desarrollar aplicaciones web y sitios mobile first, o sea, con un layout que se adapta a la pantalla del dispositivo utilizado por el usuario.

También hemos usado froala editor.

Froala es un potente editor de texto enriquecido WYSIWYG que puede satisfacer las necesidades de los programadores que trabajan en todas las industrias. Este rapidísimo editor javascript puede iniciarse rápidamente y resistir cualquier prueba rigurosa.

Para la paginación de la que hemos hablado antes he usado ngx-pagination.

Rxjs es una librería para programación reactiva usando obvservables que hacen más fácil la creación de código asincrono o basado en callbacks.

jQuery es una biblioteca multiplataforma de JavaScript, que se usa oara simplificar la manera de interactuar con los documentos HTML, manipular el árbol DOM, manejar eventos, desarrollar animaciones y agregar interacción con la técnica AJAX a páginas web.

Una técnica que me ha resultado útil a la hora de desarrollar esta aplicación es usar el comando console.log para ver en el navegador cual es el valor de lo que quiero analizar.

Ahora se va a entrar más en detalle en el componente que realiza la lógica para nuestra pagina donde se ven todos los coches, que en mi opinión es el componente más complejo:

### Explicación por secciones:

#### Imports:

import { Component, OnInit } from '@angular/core';

import { ListcochesService } from 'src/app/services/list-coches.service';

import { Router } from '@angular/router';

* **Component, OnInit**: Importa Component y OnInit desde @angular/core. Component es un decorador que se utiliza para definir un componente Angular. OnInit es una interfaz que define el método ngOnInit, el cual se llama cuando Angular inicializa el componente.
* **ListcochesService**: Importa el servicio ListcochesService desde el archivo list-coches.service.ts, que se utiliza para interactuar con la lógica de negocio relacionada con los coches.
* **Router**: Importa Router desde @angular/router, que se utiliza para la navegación dentro de la aplicación Angular.

#### Component Decorator (@Component):

@Component({

selector: 'app-lista-coches',

templateUrl: './lista-coches.component.html',

styleUrls: ['./lista-coches.component.css'],

providers: [ListcochesService]

})

* **selector**: Define el selector CSS que se utiliza para instanciar este componente en el HTML.
* **templateUrl**: Especifica la ruta al archivo HTML que define la estructura de la vista del componente.
* **styleUrls**: Especifica la(s) ruta(s) al archivo CSS que define los estilos para este componente.
* **providers**: Registra el servicio ListcochesService en el nivel del componente. Esto significa que cada instancia de ListaCochesComponent tendrá su propia instancia de ListcochesService.

#### Clase ListaCochesComponent:

* **Propiedades**:
  + totalCars, page, limit, cars, paginatedCars: Variables que almacenan datos relacionados con la lista de coches, la paginación y el ordenamiento.
  + sortField, sortOrder: Variables que almacenan el campo y el orden actual para el ordenamiento de la lista de coches.
  + selectedCars: Arreglo que almacena los coches seleccionados para comparar.
* **Constructor**:
  + Inyecta (private) instancias de ListcochesService y Router en el componente. Esto permite que el componente acceda a las funcionalidades del servicio para obtener y eliminar coches, así como para la navegación dentro de la aplicación.
* **Métodos**:
  + ngOnInit(): Método del ciclo de vida de Angular que se llama cuando el componente se inicializa. Llama al método cargarData() para obtener los coches.
  + onPageChange(page: number): Actualiza el número de página y llama a sortAndPaginate() para reordenar y paginar los coches.
  + getTotalPages(): Calcula el número total de páginas basado en la cantidad total de coches y el límite por página.
  + cargarData(): Utiliza el servicio ListcochesService para obtener la lista de coches y luego ordenar y paginar los resultados.
  + getImageUrl(car: any): Construye y devuelve la URL de la imagen del coche a partir de los datos binarios almacenados.
  + arrayBufferToBase64(buffer: ArrayBuffer): Convierte un búfer de ArrayBuffer en una cadena Base64, utilizado para convertir los datos binarios de la imagen a formato legible.
  + toggleCompare(car: any): Maneja la lógica para seleccionar o deseleccionar coches para comparar.
  + deleteCar(id: string): Llama al método deleteCoche del servicio ListcochesService para eliminar un coche por su ID después de confirmar la acción.
  + Métodos de ordenamiento (precioasc(), maleteroasc(), etc.): Estos métodos establecen el campo y el orden de clasificación y luego llaman a sortAndPaginate() para reordenar y paginar la lista de coches.
  + sort(field: string, order: string): Establece el campo y el orden de clasificación, reinicia la página a 1 y llama a sortAndPaginate().

#### Resumen:

Este componente Angular proporciona una interfaz de usuario para mostrar una lista de coches paginada y ordenable. Utiliza inyección de dependencias para interactuar con un servicio que maneja la lógica de negocio relacionada con los coches, como obtenerlos y eliminarlos. Además, incluye funcionalidad para ordenar los coches por diferentes campos, paginar los resultados, seleccionar coches para comparar y eliminar coches específicos de la lista.

Vamos a entrar en detalle también a nuestro servicio para esta página:

El código que has proporcionado es un servicio en Angular que se encarga de realizar operaciones relacionadas con la obtención y manipulación de datos de coches a través de peticiones HTTP. Vamos a explicar cada parte detalladamente:

### Importaciones

import { Injectable } from "@angular/core";

import { HttpClient } from '@angular/common/http';

import { Observable, Subject, tap } from 'rxjs';

import { list\_coches } from "./global";

import { delete\_coche } from "./global";

* **Injectable**: Es un decorador que se usa para permitir que una clase sea inyectada como dependencia en otras clases. En este caso, se aplica al servicio ListcochesService.
* **HttpClient**: Es el servicio de Angular para realizar peticiones HTTP.
* **Observable, Subject, tap**: Parte del módulo rxjs, utilizado para trabajar con programación reactiva y observables en Angular.
* **Importaciones de constantes**: Importa diferentes constantes (list\_coches, delete\_coche, etc.) desde el archivo global.ts, que asumo define las URL de las API utilizadas por este servicio.

### Decorador Injectable y Constructor

@Injectable()

export class ListcochesService {

public url: string;

public url\_delete: string;

private \_refresh$ = new Subject<void>();

constructor(

public \_http: HttpClient

) {

// Asignación de las URL desde las constantes importadas

this.url = list\_coches.url;

this.url\_delete = delete\_coche.url;

}

* **Injectable()**: Este decorador permite que la clase ListcochesService sea inyectada con dependencias, como el HttpClient.
* **Constructor**: Aquí se inicializan las propiedades del servicio, incluyendo las URLs para realizar las peticiones HTTP. Estas URLs se obtienen de las constantes importadas desde global.ts.

### Métodos del Servicio

getcoches(): Observable<any[]> {

return this.\_http.get<any[]>(this.url);

}

deletecoche(id: string): Observable<any> {

return this.\_http.delete(this.url\_delete + id).pipe(

tap(() => {

this.\_refresh$.next();

})

);

}

// Getter para el Subject \_refresh$

get refresh$() {

return this.\_refresh$;

}

* **Métodos para obtener datos**: Estos métodos realizan peticiones HTTP GET para obtener diferentes datos relacionados con los coches (marcas, modelos, precios, etc.). Están diseñados para devolver observables (Observable<any>).
* **Método deletecoche**: Realiza una petición HTTP DELETE para eliminar un coche específico identificado por su ID. Utiliza la URL url\_delete concatenada con el ID proporcionado. Después de la eliminación, emite una señal a través de \_refresh$.next() para notificar a otros componentes que los datos han sido actualizados y que necesitan ser recargados.

### Conclusiones

Este servicio ListcochesService proporciona métodos para interactuar con datos de coches a través de peticiones HTTP, manejar actualizaciones de datos utilizando Subjects de RxJS, y asegurar que los datos estén disponibles para otros componentes Angular de manera eficiente y reactiva.

Mi otra solución para el frontend está hecha en Android studio.

También tiene la funcionalidad de comparar coches.

Android Studio es el entorno de desarrollo integrado (IDE) oficial que se usa en el desarrollo de apps para Android. Basado en el potente editor de código y las herramientas para desarrolladores de IntelliJ IDEA, Android Studio ofrece aún más funciones que mejoran tu productividad cuando compilas apps para Android, como las siguientes:

Un sistema de compilación flexible basado en Gradle

Un emulador rápido y cargado de funciones

Un entorno unificado donde puedes desarrollar para todos los dispositivos Android

Ediciones en vivo para actualizar elementos componibles en emuladores y dispositivos físicos, en tiempo real

Integración con GitHub y plantillas de código para ayudarte a compilar funciones de apps comunes y también importar código de muestra

Variedad de marcos de trabajo y herramientas de prueba

Herramientas de Lint para identificar problemas de rendimiento, usabilidad y compatibilidad de versiones, entre otros

Compatibilidad con C++ y NDK

La solución ha sido programada con flutter.

Flutter es un SDK desarrollado por Google para crear aplicaciones móviles tanto para Android como para iOS (Apple). Fue desarrollado como un software para uso interno dentro de la compañía pero vieron el potencial que tenia y decidieron lanzarlo como proyecto de código libre. Actualmente es uno de los proyectos de desarrollo de aplicaciones móviles que más está creciendo. Además, desde la última versión estable, también es posible realizar aplicaciones Web y de escritorio para Windows y Mac. Aunque en estas dos plataformas aún está en fase experimental (beta).

Estas son las tres principales ventajas que ofrece Flutter respecto a otras frameworks de desarrollo de aplicaciones multiplataforma:

Compila en nativo, tanto en Android como en iOS.

La creación interfaces gráficas es muy flexible, puedes combinar diferentes Widgets (elementos gráficos) para crear las vistas.

El desarrollo es muy rápido, permite ver el resultado de forma instantánea mientras se escribe el código.

Primero voy a hablar del archivo pubspec.yaml

El archivo pubspec.yaml es un archivo de configuración fundamental en proyectos Flutter y Dart. Aquí te explico las secciones y los elementos comunes que puedes encontrar en este archivo:

### Nombre y Descripción del Proyecto

name: nombre\_del\_proyecto

description: descripción\_del\_proyecto

* **name**: Define el nombre del proyecto. Debe ser único en el repositorio de paquetes de Dart.
* **description**: Proporciona una breve descripción del proyecto.

### Versión del Proyecto

version: 1.0.0+1

* **version**: Indica la versión actual del proyecto. Sigue un formato de tres números separados por puntos (por ejemplo, 1.0.0), siguiendo las convenciones de versionado semántico.

### Entorno de Ejecución y Dependencias

environment:

sdk: '>=3.0.5 <4.0.0'

dependencies:

flutter:  
 sdk: flutter  
cupertino\_icons: ^1.0.2  
google\_fonts: ^5.0.0  
flutter\_svg: ^2.0.7  
image\_picker: ^0.8.4+1

...

* **environment**: Especifica la versión mínima y máxima del SDK de Dart que se necesita para ejecutar el proyecto.
* **dependencies**: Lista los paquetes de Dart y Flutter que el proyecto necesita para funcionar correctamente. Cada dependencia tiene un nombre seguido de una versión específica o un rango de versiones (^1.0.2 significa que se aceptan versiones desde 1.0.2 hasta la próxima versión mayor compatible).

### Dependencias de Desarrollo

dev\_dependencies:

flutter\_test:

sdk: flutter

...

* **dev\_dependencies**: Contiene las dependencias que son necesarias únicamente durante el desarrollo, como pruebas unitarias (flutter\_test) u otras herramientas de desarrollo.

### Archivos de Configuración y Recursos

flutter:

assets:

- assets/images/

* **flutter**: Especifica configuraciones específicas para aplicaciones Flutter.
  + **assets**: Lista los directorios o archivos que se deben incluir como activos en la aplicación.
  + **fonts**: Define las fuentes personalizadas que se deben cargar en la aplicación.

### Scripts y Otros Ajustes

scripts:

build:

- flutter build apk

analyze:

- flutter analyze

* **scripts**: Permite definir comandos personalizados que se pueden ejecutar desde la línea de comandos. Esto es útil para automatizar tareas como la construcción (build), análisis (analyze), entre otros.

### Conclusiones

El archivo pubspec.yaml es esencial para la configuración y la gestión de dependencias en proyectos Flutter y Dart. Define desde el nombre del proyecto hasta las dependencias necesarias, configuraciones de activos y fuentes, scripts personalizados y más. Es fundamental para asegurar que el entorno de desarrollo y producción del proyecto esté correctamente configurado y mantenido.

Empezamos hablando por el login:

### Importaciones

import 'dart:convert';

import 'package:flutter/material.dart';

import 'package:frontendtfm/signup.dart';

import 'package:frontendtfm/dashboard.dart';

import 'package:frontendtfm/user.dart';

import 'package:google\_fonts/google\_fonts.dart';

import 'package:http/http.dart' as http;

* **dart:convert**: Proporciona funciones para codificar y decodificar JSON.
* **flutter/material.dart**: Contiene los widgets y utilidades básicas de Flutter para construir la interfaz de usuario.
* **frontendtfm/signup.dart**: Importa la pantalla de registro (Signup) para navegar a ella si el usuario desea registrarse.
* **frontendtfm/dashboard.dart**: Importa la pantalla del dashboard (Dashboard) a la que se navegará si el inicio de sesión es exitoso.
* **frontendtfm/user.dart**: Importa la clase User que probablemente contiene la estructura de datos del usuario.
* **google\_fonts/google\_fonts.dart**: Permite utilizar fuentes de Google Fonts en la aplicación Flutter.
* **http/http.dart**: Importa el paquete http de Dart que se utiliza para realizar peticiones HTTP a la API backend.

### Clase Signin (StatefulWidget)

class Signin extends StatefulWidget {

Signin({Key? key}) : super(key: key);

@override

\_SigninState createState() => \_SigninState();

}

* **Signin**: Es un StatefulWidget que define la pantalla de inicio de sesión. Contiene un estado mutable (\_SigninState) que permite actualizar la interfaz de usuario en respuesta a las interacciones del usuario.

### Estado \_SigninState

class \_SigninState extends State<Signin> {

final \_formKey = GlobalKey<FormState>();

String errorMessage = '';

User user = User('', '', '', '');

// Otras propiedades y métodos

}

* **\_formKey**: Es una clave global (GlobalKey<FormState>) que identifica el formulario en esta pantalla de inicio de sesión.
* **errorMessage**: Almacena mensajes de error que pueden ocurrir durante el proceso de inicio de sesión.
* **user**: Instancia de la clase User que contiene los datos del usuario (email y password).

### Método save()

Future save() async {

try {

var res = await http.post(

Uri.parse("http://192.168.1.134:4001/api/login/login"),

headers: <String, String>{

'Content-Type': 'application/json; charset=UTF-8',

},

body: jsonEncode(<String, String>{

'email': user.email,

'contrasena': user.password,

}),

);

// Lógica para manejar la respuesta de la petición HTTP

} catch (e) {

// Manejo de errores de conexión

}

}

* **save()**: Método asíncrono que realiza una solicitud POST a la API de inicio de sesión con el correo electrónico (email) y la contraseña (password) del usuario codificados en JSON.
* **http.post**: Utiliza el paquete http para hacer la solicitud POST a la URL especificada.
* **jsonEncode**: Convierte el mapa de datos (<String, String>) en formato JSON para el cuerpo de la solicitud.
* **Manejo de Respuestas y Errores**: Se implementa lógica para manejar la respuesta del servidor (res) y manejar errores de conexión (catch (e)).

### Método build(BuildContext context)

@override

Widget build(BuildContext context) {

return Scaffold(

body: SafeArea(

child: Stack(

children: [

// Contenido de la interfaz de usuario (UI)

],

),

),

);

}

* **build(BuildContext context)**: Construye la interfaz de usuario utilizando el widget Scaffold, que proporciona una estructura básica de página.
* **SafeArea**: Asegura que el contenido de la interfaz de usuario no se superponga con áreas de sistema (como la barra de notificaciones).
* **Stack**: Coloca los widgets unos encima de otros en capas.

### Interfaz de Usuario (UI)

* Muestra un formulario con campos para ingresar el correo electrónico y la contraseña.
* Utiliza TextFormField para los campos de entrada, que capturan cambios en tiempo real (onChanged).
* Utiliza FloatingActionButton para el botón de "Iniciar sesión".
* InkWell permite navegar a la pantalla de registro (Signup) cuando se hace clic en "Regístrate".
* Muestra mensajes de error (errorMessage) en caso de problemas durante el inicio de sesión.

### Navegación

* Después de un inicio de sesión exitoso, se navega automáticamente a la pantalla del dashboard (Dashboard).
* Se utiliza Navigator.push para manejar la navegación entre pantallas en Flutter.

### Conclusión

Este código en Flutter proporciona una base funcional para una pantalla de inicio de sesión que interactúa con una API backend para autenticar usuarios. Utiliza widgets de Flutter para construir una interfaz de usuario receptiva y maneja la comunicación HTTP asíncrona con el backend para realizar el inicio de sesión.

Ahora voy a explicar el archivo que se encarga de crear el coche:

#### Importaciones

import 'package:flutter/material.dart';

import 'package:image\_picker/image\_picker.dart';

import 'package:http/http.dart' as http;

import 'dart:io';

import 'dart:convert';

import 'dashboard.dart';

* **flutter/material.dart**: Biblioteca básica de Flutter para construir interfaces de usuario.
* **image\_picker.dart**: Biblioteca para seleccionar imágenes desde la galería o la cámara.
* **http.dart**: Biblioteca para realizar solicitudes HTTP.
* **dart**

: Biblioteca para manipular archivos y otras operaciones de entrada/salida.

* **dart**

: Biblioteca para convertir datos a formato JSON.

* **dashboard.dart**: Importación de una pantalla llamada Dashboard.

#### Definición de la Clase NuevoCoche

class NuevoCoche extends StatefulWidget {

@override

\_NuevoCocheState createState() => \_NuevoCocheState();

}

Define un StatefulWidget llamado NuevoCoche. Esto es necesario porque se maneja el estado interno (como la imagen seleccionada) que puede cambiar durante la vida útil del widget.

#### Estado de la Clase \_NuevoCocheState

class \_NuevoCocheState extends State<NuevoCoche> {

File? \_image;

final picker = ImagePicker();

final TextEditingController \_marcaController = TextEditingController();

final TextEditingController \_modeloController = TextEditingController();

final TextEditingController \_precioController = TextEditingController();

final TextEditingController \_maleteroController = TextEditingController();

final TextEditingController \_kmController = TextEditingController();

final TextEditingController \_potenciaController = TextEditingController();

final TextEditingController \_cilindradaController = TextEditingController();

final TextEditingController \_consumourbanoController = TextEditingController();

final TextEditingController \_consumoextraurbanoController = TextEditingController();

final TextEditingController \_traccionController = TextEditingController();

* **\_image**: Variable para almacenar el archivo de imagen seleccionado.
* **picker**: Instancia de ImagePicker para seleccionar imágenes.
* **Varios TextEditingController**: Controladores para manejar el texto ingresado en los campos de entrada.

#### Método para Seleccionar Imagen

Future<void> \_pickImage() async {

final pickedFile = await picker.getImage(source: ImageSource.gallery);

setState(() {

if (pickedFile != null) {

\_image = File(pickedFile.path);

} else {

print('No image selected.');

}

});

}

* **\_pickImage**: Método que abre la galería y permite al usuario seleccionar una imagen. Si se selecciona una imagen, se almacena en \_image. Si no, se imprime un mensaje indicando que no se seleccionó ninguna imagen.

#### Método para Subir Datos

Future<void> \_uploadData() async {

if (\_image == null) {

print('No image selected.');

return;

}

if (\_marcaController.text.isEmpty ||

\_modeloController.text.isEmpty ||

\_precioController.text.isEmpty ||

\_maleteroController.text.isEmpty ||

\_kmController.text.isEmpty ||

\_potenciaController.text.isEmpty ||

\_cilindradaController.text.isEmpty ||

\_consumourbanoController.text.isEmpty ||

\_consumoextraurbanoController.text.isEmpty ||

\_traccionController.text.isEmpty) {

print('Por favor, llene todos los campos.');

return;

}

try {

final request = http.MultipartRequest(

'POST',

Uri.parse('http://10.0.2.2:4001/api/newcar'), // Use 10.0.2.2 for Android emulator

);

request.fields['marca'] = \_marcaController.text;

request.fields['modelo'] = \_modeloController.text;

request.fields['precio'] = \_precioController.text;

request.fields['maletero'] = \_maleteroController.text;

request.fields['km'] = \_kmController.text;

request.fields['potencia'] = \_potenciaController.text;

request.fields['cilindrada'] = \_cilindradaController.text;

request.fields['consumourbano'] = \_consumourbanoController.text;

request.fields['consumoextraurbano'] = \_consumoextraurbanoController.text;

request.fields['traccion'] = \_traccionController.text;

request.files.add(await http.MultipartFile.fromPath('imagen', \_image!.path));

final response = await request.send();

if (response.statusCode == 200) {

print('Uploaded successfully');

Navigator.pushReplacement(

context,

MaterialPageRoute(builder: (context) => Dashboard()),

);

} else {

print('Failed to upload. Status code: ${response.statusCode}');

final responseBody = await response.stream.bytesToString();

print('Response body: $responseBody');

}

} catch (e) {

print('Error: $e');

}

}

* **\_uploadData**: Método que verifica si se ha seleccionado una imagen y si todos los campos están llenos.
  + Si falta la imagen o algún campo, imprime un mensaje y retorna.
  + Si todo está en orden, crea una solicitud multipart para enviar los datos y la imagen al servidor.
  + Maneja la respuesta del servidor, imprimiendo un mensaje de éxito o error según corresponda.
  + En caso de éxito, redirige al usuario al Dashboard.

#### Método build

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text('Nuevo Coche'),

),

body: Center(

child: SingleChildScrollView(

child: Padding(

padding: const EdgeInsets.all(16.0),

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

\_image == null

? Text('No image selected.')

: Image.file(\_image!),

TextField(

controller: \_marcaController,

decoration: InputDecoration(labelText: 'Marca'),

),

TextField(

controller: \_modeloController,

decoration: InputDecoration(labelText: 'Modelo'),

),

TextField(

controller: \_precioController,

decoration: InputDecoration(labelText: 'Precio'),

),

TextField(

controller: \_maleteroController,

decoration: InputDecoration(labelText: 'Maletero'),

),

TextField(

controller: \_kmController,

decoration: InputDecoration(labelText: 'Km'),

),

TextField(

controller: \_potenciaController,

decoration: InputDecoration(labelText: 'Potencia'),

),

TextField(

controller: \_cilindradaController,

decoration: InputDecoration(labelText: 'Cilindrada'),

),

TextField(

controller: \_consumourbanoController,

decoration: InputDecoration(labelText: 'Consumo urbano'),

),

TextField(

controller: \_consumoextraurbanoController,

decoration: InputDecoration(labelText: 'Consumo extraurbano'),

),

TextField(

controller: \_traccionController,

decoration: InputDecoration(labelText: 'Tracción'),

),

ElevatedButton(

onPressed: \_pickImage,

child: Text('Seleccionar Imagen'),

),

ElevatedButton(

onPressed: \_uploadData,

child: Text('Crear Coche'),

),

],

),

),

),

),

);

}

}

* **build**: Método que construye la interfaz de usuario.
  + Usa Scaffold para una estructura de página básica con una barra de aplicaciones.
  + Usa SingleChildScrollView y Column para permitir el desplazamiento y organizar los elementos verticalmente.
  + Usa TextField para cada campo de entrada, con un TextEditingController correspondiente para gestionar el texto ingresado.
  + Usa ElevatedButton para los botones de seleccionar imagen y crear coche.
  + Muestra la imagen seleccionada si hay una.

### Conclusión

Este código proporciona una interfaz de usuario funcional y robusta para agregar un nuevo coche, manejando tanto la selección de imágenes como el envío de datos al servidor.

# 

# Postman

Para ver todas las requests que hemos testeado con postman he exportado la colección en formato json.

Postman en sus inicios nace como una extensión que podía ser utilizada en el navegador Chrome de Google y básicamente nos permite realizar peticiones de una manera simple para testear APIs de tipo REST propias o de terceros.

Gracias a los avances tecnológicos, Postman ha evolucionado y ha pasado de ser de una extensión a una aplicación que dispone de herramientas nativas para diversos sistemas operativos como lo son Windows, Mac y Linux.

# Github

Todo el trabajo ha sido subido a Github.

Github es un portal creado para alojar el código de las aplicaciones de cualquier desarrollador, y que fue comprada por Microsoft en junio del 2018. La plataforma está creada para que los desarrolladores suban el código de sus aplicaciones y herramientas, y que como usuario no sólo puedas descargarte la aplicación, sino también entrar a su perfil para leer sobre ella o colaborar con su desarrollo.

Para hacer los commit ect de forma más sencilla para mí, me descargue Github desktop, que sirve para hacer este tipo de comandos desde el escritorio desde la aplicación que nos proporcionan.

# 8.Trabajo futuro

Hay muchas cosas que se podrían mejorar de mi solución como es normal.

Se podría incluir la posibilidad de subir fotos a la aplicación o incluso realizar fotos desde la cámara.

También se podría mejorar el diseño de la página web, introduciendo más estilos css por ejemplo.

La velocidad de carga también se podría mejorar cargando los datos desde la caché.

En este casó no lo he hecho, porque creo que no es necesario, pero si la cantidad de datos a manejar es muy grande, creo que sería una buena opción.

La parte de Android studio puede ser mejorada visualizando las imágenes, porque si que se cargan pero no se visualizan.

# 9.Conclusiones

He intentado que en está solución se viesen los mínimos mas importantes vistos en este master.

Por una lado tenemos un backend, donde mediante llamadas al mismo un usuario puede darse de alta, logearse de manera segura y disfrutar de la capacidad de comparar los diferentes coches que han sido previamente dados de alta.

El frontend realiza las llamadas al backend y además el diseño no carece de estilos css que hacen que el usuario disfrute más de la experiencia.

Además se han usado dos tecnologías diferentes para el frontend teniendo en cuenta la cantidad de dispositivos diferentes que existen hoy en día en el mercado.

ENTIDAD COLABORADORA