

**PROYECTO DE CICLO DE G.S. DE  
Desarrollo de Aplicaciones Web**

**FletWise**



**ALEJANDRO DEL SALTO  
MIMBRERA  
CURSO 2023/24**

## Índice

<b>1</b>	<b>Introducción</b>	<b>3</b>
1.1	Datos del proyecto . . . . .	3
1.2	Resumen . . . . .	3
<b>2</b>	<b>Tecnologías</b>	<b>7</b>
2.1	Entre las tecnologías usadas nos encontramos con . . . . .	7
<b>3</b>	<b>Análisis</b>	<b>8</b>
<b>4</b>	<b>Apariencia</b>	<b>9</b>
4.1	Login . . . . .	9
4.2	Home - Usuario . . . . .	10
4.3	Home - Admin . . . . .	10
4.4	Vehiculos . . . . .	11
4.5	Marcas . . . . .	11
4.6	Modelos . . . . .	12
4.7	Combustibles . . . . .	12
4.8	Perfil - Usuario . . . . .	13
4.9	Perfil - Administrador . . . . .	13
<b>5</b>	<b>Implementación</b>	<b>14</b>
5.1	Back-End . . . . .	14
5.1.1	Uso del contexto y los modelos . . . . .	14
5.1.2	Patrón a utilizar . . . . .	15
5.1.3	Repositorio General . . . . .	20
5.1.4	Servicios . . . . .	23
5.1.5	Controlador Vehículo . . . . .	25
5.1.6	Inyección . . . . .	26
5.1.7	Control de CORS . . . . .	27
5.1.8	Manejo de Swagger . . . . .	28
5.1.9	Personalización de Swagger . . . . .	29
5.1.10	DTO'S . . . . .	30
5.1.11	Login . . . . .	31
5.1.12	Cambio dinámico de nuestra cadena de conexión . . . . .	34
5.2	Front-End . . . . .	39
5.2.1	Peticiones axios . . . . .	39

---

5.3	Apartados destacados . . . . .	42
5.3.1	Cambio de empresa activa . . . . .	42
5.3.2	Paginación, filtrado, búsqueda y ordenación . . . . .	45
5.3.3	Control de la auditoría . . . . .	46
5.3.4	Mapeo de objetos . . . . .	47
5.3.5	Cómo lleva a cabo el mapeo? . . . . .	48
5.3.6	Inclusión de clases mediante Entity Framework . . . . .	50
5.3.7	Más funcionalidades de los mapeos . . . . .	52
5.3.8	Uso de secuencias para la generación de los id's . . . . .	53
5.4	Funciones extra . . . . .	55
5.4.1	Selector de tema . . . . .	55
5.4.2	Gestión de errores . . . . .	57
<b>6</b>	<b>Conclusiones</b>	<b>62</b>
<b>7</b>	<b>Referencias Web</b>	<b>63</b>

# 1 Introducción

## 1.1 Datos del proyecto

Nombre	Apellidos	Título	Año	Centro
Alejandro	del Salto Mimbrera	FletWise	2023	IES Virgen del Carmen

## 1.2 Resumen

Para poder entender la forma en la que vamos a consultar nuestros datos, es importante hacer una breve (quizás no tan breve) introducción.

El proyecto trata de una aplicación cuya función es realizar un mantenimiento del registro de vehículos de distintas empresas, para ello, contaremos con varios usuarios, empresas y por supuesto, vehículos los cuales contarán principalmente con una marca, un modelo y un tipo de combustible, los cuales a su vez, también pertenecen a una empresa en concreto (y el modelo pertenece a una marca determinada).

Contaremos con 2 posibilidades a la hora de iniciar sesión, que el usuario cuente o no con permisos de administración.

En caso de tenerlos, si vista principal (home) contará con acceso a las vistas de gestión de marcas, modelos y combustibles, y en la vista del perfil, donde un usuario puede realizar un cambio de empresa activa entre las diferentes empresas a las que pertenezca, un administrador puede hacer que cualquier empresa sea su empresa activa sin necesidad de pertenecer a ella, esto con la finalidad de poder hacer ajustes de vehiculos, marcas, modelos y combustibles en otras empresas.

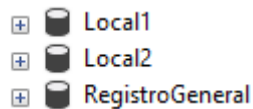
Ahora, adentrémonos en cosas más específicas.

Nuestra api será desarrollada en .NET con Entity Framework y haremos uso de un patrón repositorio-servicio además de comenzar mediante un database-first.

Esto significa que vamos a generar nuestro contexto y modelos en base a una base de datos ya creada en SQLServer, y ese contexto junto con los modelos nos permitirán hacer peticiones a nuestra base de datos mediante un repositorio (la capa de acceso directo a los datos), ese repositorio será llamado por un servicio el cual a su vez, será llamado por un controlador, este controlador contará con los endPoints (“funciones” a las cuales llamaremos desde nuestro FrontEnd) con cada una de las funcionalidades necesarias.

El contexto necesita una cadena de conexión para saber hacia que base de datos debe conectarse, hecho que nos será útil ya que con Entity Framework, podemos usar un mismo contexto para consultar varias bases de datos con la única condición de que estas sean copias exactas unas de otras, y lo único que necesitaremos hacer es cambiar la cadena de conexión.

Nosotros contaremos con 3 bases de datos las cuales como dije anteriormente, son copias exactas en cuanto a su estructura y lo único que varía es el contenido que se almacena en ellas.



**Figura 1:** Bases de datos

1 - Registro General, esta base de datos contará con la información relacionada con nuestros usuarios y empresas, es decir, podremos consultar cuantos usuarios y empresas hay, a que empresa o empresas pertenece cada usuario y que empresa tiene activa cada usuario en este preciso instante.

2 - Local1, esta base de datos contará con información de algunas empresas, es decir, contendrá vehículos, marcas, modelos y combustibles pertenecientes a x empresas.

3 - Local2, esta base de datos contará, al igual que Local 1, con vehículos, marcas, modelos y combustibles pertenecientes a x empresas (distintas de las empresas a las que se hace referencia en Local1).

Cabe recalcar que Entity Framework abre una nueva conexión con la base de datos cada vez que se hace una petición y esto nos será útil ya que para cambiar la cadena de conexión y saber hacia que base de datos realizar una petición, el secreto está en que en nuestra tabla de empresas de la base de datos de Registro General, contaremos con un campo llamado BaseActiva, cuyos valores serán Local1 o Local2, es decir, desde nuestra base de datos de registro general, podemos saber en que base de datos se encuentran almacenados los vehículos, marcas, modelos y combustibles de cada empresa.

	Nombre	BaseActiva	GUID_Registro	Id	FechaCreacion	FechaModificacion	FechaBorradoLogico	BorradoLogico	Id_Creador	Id_Editor	Id_Borrador
1	Del Salto S.L	Local1	3A37DDDF-39CD-43FC-AF3C-8383E6242ECC	157356	2024-05-28 09:26:22.597	NULL	NULL	NULL	1	NULL	NULL
2	ALMACÉN DE PRUEBAS	Local2	DD0CE55A-128B-4E2A-8C97-B25085174B34	157357	2024-05-28 09:30:36.363	NULL	NULL	NULL	1	NULL	NULL
3	Tech Labs	Local1	C8368781-C215-4083-8DC2-D98D6BE76D46	157366	2024-05-28 10:11:33.963	NULL	NULL	NULL	1	NULL	NULL
4	IESVIRGEN	Local2	2783CA29-A566-48D8-A25A-9950A67E18EE	157370	2024-05-28 10:24:28.253	NULL	NULL	NULL	1	NULL	NULL

**Figura 2:** Base Activa

Por lo tanto, solo necesitaremos cambiar el campo “database” de nuestra cadena de conexión y sustituirlo por la base activa de la empresa que queramos para así tener la cadena de conexión preparada para ser usada, el último paso sería crear una nueva instancia de nuestro repositorio para que este abra una conexión nueva con la base de datos a partir del contexto, pero esta vez no usará la cadena de conexión predeterminada, sino la cadena modificada que almacenamos anteriormente.

```
"ConnectionStrings": {
  "cadena": "server=ALEJANDRO;database=RegistroGeneral;Trusted_Connection=True;TrustServerCertificate=True"
```

**Figura 3:** Cadena de conexión

En nuestras bases de datos no se borrarán registros, tan solo se marcarán como borrados (borrado lógico) de tal forma que cuando hagamos peticiones, devolveremos los registros que no estén marcados como borrados.

	Nombre	BaseActiva	GUID_Registro	Id	FechaCreacion	FechaModificacion	FechaBorradoLogico	BorradoLogico	Id_Creador	Id_Editor	Id_Borrador
1	Del Salto S.L	Local1	3A37DDDF-39CD-43FC-AF3C-8383E624ECC	157356	2024-05-28 09:26:22.597	NULL	NULL	NULL	1	NULL	NULL
2	ALMACÉN DE PRUEBAS	Local2	DD0CE55A-128B-4E2A-8C97-B25085174B34	157357	2024-05-28 09:30:36.363	NULL	NULL	NULL	1	NULL	NULL
3	Tech Labs	Local1	C8368781-C215-4083-8DC2-D98D6BE76D46	157366	2024-05-28 10:11:33.963	NULL	NULL	NULL	1	NULL	NULL
4	IESVIRGEN	Local2	2783CA29-A566-48D8-A25A-9950A67E18EE	157370	2024-05-28 10:24:28.253	NULL	NULL	NULL	1	NULL	NULL

**Figura 4:** Borrado lógico

Contaremos con campos de auditoría, estos muestran quien y cuando se realizan los insert, updates y deletes en la base de datos.

	Nombre	BaseActiva	GUID_Registro	Id	FechaCreacion	FechaModificacion	FechaBorradoLogico	BorradoLogico	Id_Creador	Id_Editor	Id_Borrador
1	Del Salto S.L	Local1	3A37DDDF-39CD-43FC-AF3C-8383E624ECC	157356	2024-05-28 09:26:22.597	NULL	NULL	NULL	1	NULL	NULL
2	ALMACÉN DE PRUEBAS	Local2	DD0CE55A-128B-4E2A-8C97-B25085174B34	157357	2024-05-28 09:30:36.363	NULL	NULL	NULL	1	NULL	NULL
3	Tech Labs	Local1	C8368781-C215-4083-8DC2-D98D6BE76D46	157366	2024-05-28 10:11:33.963	NULL	NULL	NULL	1	NULL	NULL
4	IESVIRGEN	Local2	2783CA29-A566-48D8-A25A-9950A67E18EE	157370	2024-05-28 10:24:28.253	NULL	NULL	NULL	1	NULL	NULL

**Figura 5:** Auditoria

Haremos uso de DTO's - Data Transfer Objects, los cuales nos serán útiles para mostrar al usuario solo la información que necesita un ejemplo sería que al usuario no le interesa saber que un objeto posee un campo llamado borradoLógico.

Para convertir un modelo a DTO y viceversa será necesario realizar mapeos, los cuales serán llevados a cabo usando un paquete NuGet llamado Automapper, este paquete permite optimizar en gran cantidad el proceso de mapeo de objetos.

Nuestra aplicación realizará un paginado, filtrado, búsqueda y ordenación de los registros devueltos desde el BackEnd, eliminando carga de trabajo del FrontEnd ( tutorial de paginado, filtrado, búsqueda y ordenación en CodeMaze).

Nuestra autenticación se realizará mediante un token el cual será generado al hacer login y tendrá una vida útil limitada, en este token se almacenará información de interés como el nombre de usuario, la

empresa a la que pertenece y si es un usuario corriente o un administrador, ya que implementaremos permisos adicionales para los administradores.

El token puede ser renovado siempre y cuando hayan pasado menos de 5 minutos desde que caducó.

Para el FrontEnd usaremos Vuejs2 con Vuetify y Axios para las llamadas a la API.

Nuestro FrontEnd no tiene muchas cosas interesantes, lo único que podemos destacar es que consultaremos el token para ver si el usuario posee el rol de administrador y así mostrarles las características ocultas para los usuarios comunes (estas características son el poder administrar marcas, modelos y combustibles y el poder cambiar de empresa activa entre todas las empresas de nuestras bases de datos Local1 y Local2).

Por otro lado, Axios nos permite crear un interceptor que controlará nuestras peticiones tanto antes de realizarlas así como una vez ya han finalizado, de esta forma podemos hacer que antes de cada petición, enviemos nuestro token en las cabeceras de la misma (ya que si una petición se intenta hacer sin token, supondrá el cierre instantáneo de la sesión y la necesidad de realizar un logeo de nuevo ) por otro lado, si enviamos el token y está caducado, nuestra petición no será exitosa y aquí tenemos 2 posibilidades, que nuestro token haya caducado hace más de 5 minutos, lo que irremediablemente conducirá hacia el mismo resultado que si intentamos hacer una petición sin token, sin embargo, en caso de que nuestra petición falle pero nuestro token haya caducado hace menos de 5 minutos, se enviará una petición para tratar de renovarlo y en caso exitoso, se repetirá la petición que falló en un principio (si falla la renovación del token, simplemente nos mandará a logearnos de nuevo).

Anteriormente se nombró el concepto de empresa activa, esto hace referencia a la empresa seleccionada entre las empresas a las que un usuario pertenece, por lo que al hacer login, la información mostrada en la tabla de vehículos será la correspondiente a esa empresa, teniendo la posibilidad de cambiar a otra empresa activa para poder gestionar los registros pertenecientes a la otra empresa (como dije antes, un administrador puede hacer esto sin necesidad de pertenecer a ninguna empresa).

## 2 Tecnologías

### 2.1 Entre las tecnologías usadas nos encontramos con

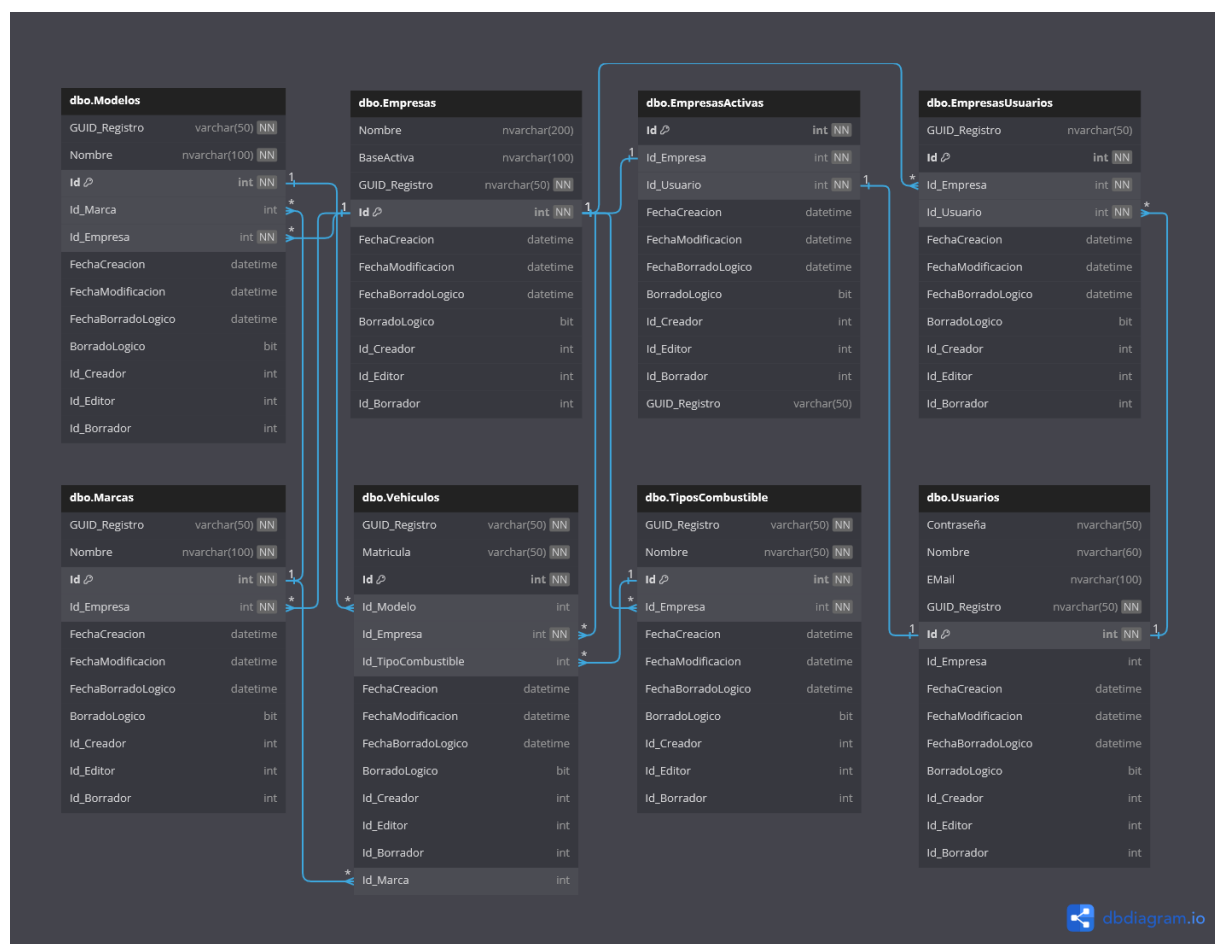
- Back-End
  - .NET Core
  - Uso de Entity Framework para la conexión con las bases de datos en SQLServer
  - El entorno de desarrollo usado será Visual Studio 2022
  - Se implementarán diferentes paquetes NuGet los cuales permitirán utilizar código de forma más eficiente
    - \* AutoMapper - Usado para el mapeo de modelos a DTO's y al contrario
    - \* Asp.NetCore.Authentication.JwtBearer - Usado para la generación y gestión de tokens
    - \* EntityFrameworkCore, EntityFrameworkCore.SqlServer y EntityFrameworkCore.Tools
      - Usados para poder realizar la conexión con nuestras bases de datos mediante un contexto y modelos
    - \* NLOG - Usado para nuestro middleware de control de errores
    - \* Swashbuckle.AspNetCore - Usado para la implementación del swagger
- Front-End
  - Vuejs2, el cual implementará:
    - \* Vue Router
    - \* Vuetify 2
  - Axios para las llamadas a la api
  - El Editor usado será Visual Studio Code



### 3 Análisis

A continuación, el diagrama Uml en el que mostraremos la estructura de nuestra base de datos, cabe recalcar que las relaciones mostradas son simbólicas ya que en la estructura real, no hay relaciones ya que se realizan a nivel de código

Si bien esta práctica no es necesaria llevarla a cabo en el proyecto realizado ( ya que no se acerca ni un poco a la complejidad de un proyecto empresarial real ), me han enseñado que en proyectos grandes, el hecho de tener relaciones en las tablas reduce el rendimiento, por lo que una forma de optimizarlo es mediante la eliminación de dichas relaciones



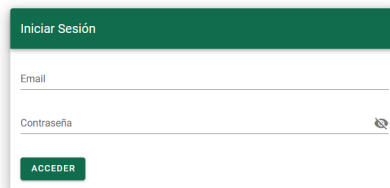
**Figura 6:** Diagrama UML

Al contar con un usuario administrador que puede convertir cualquier empresa en su empresa activa a pesar de no pertenecer a ella, justo en ese caso no cumpliremos con la relación que indica la imposibilidad de que una empresa sea empresa activa para más de un usuario simultáneamente

## 4 Apariencia

Aquí se mostrarán las diferentes vistas de nuestra aplicación antes de explicar el contenido y el funcionamiento del código ya que considero que teniendo la apariencia final del proyecto, a la hora de ver el código y los pasos seguidos es más fácil el comprender por que se están llevando a cabo

### 4.1 Login



The image shows a login form titled "Iniciar Sesión" (Log In). It features two input fields: "Email" and "Contraseña" (Password). The "Contraseña" field has a small eye icon to the right, indicating a toggle for password visibility. Below the fields is a green button labeled "ACCEDER" (Log In).

**Figura 7:** Login

## 4.2 Home - Usuario

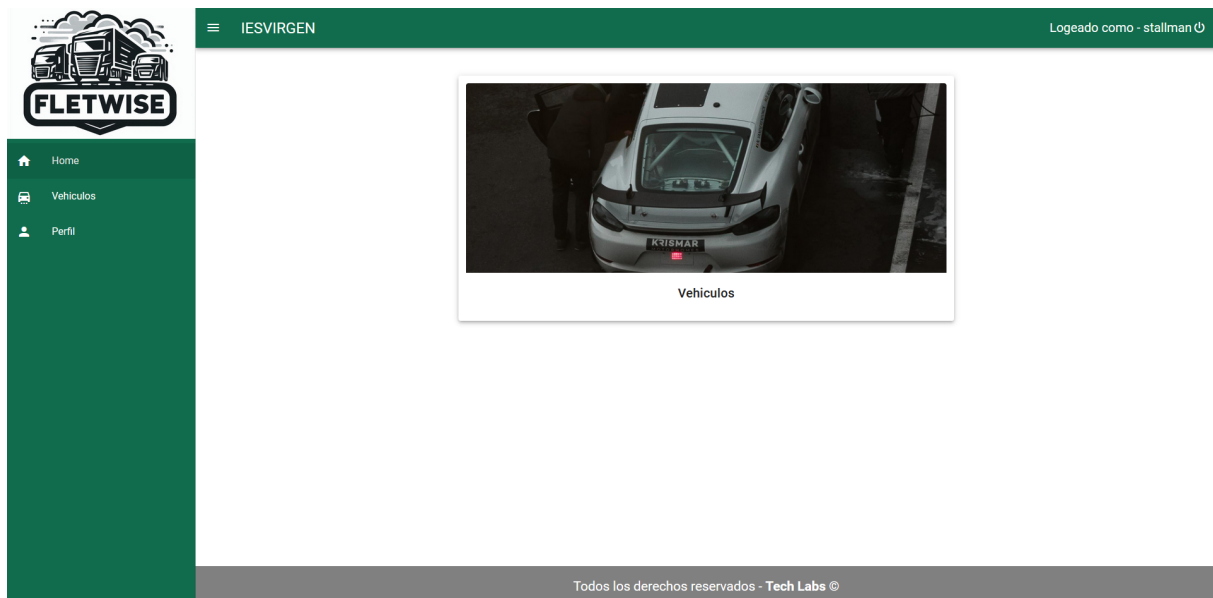


Figura 8: Home como usuario

## 4.3 Home - Admin

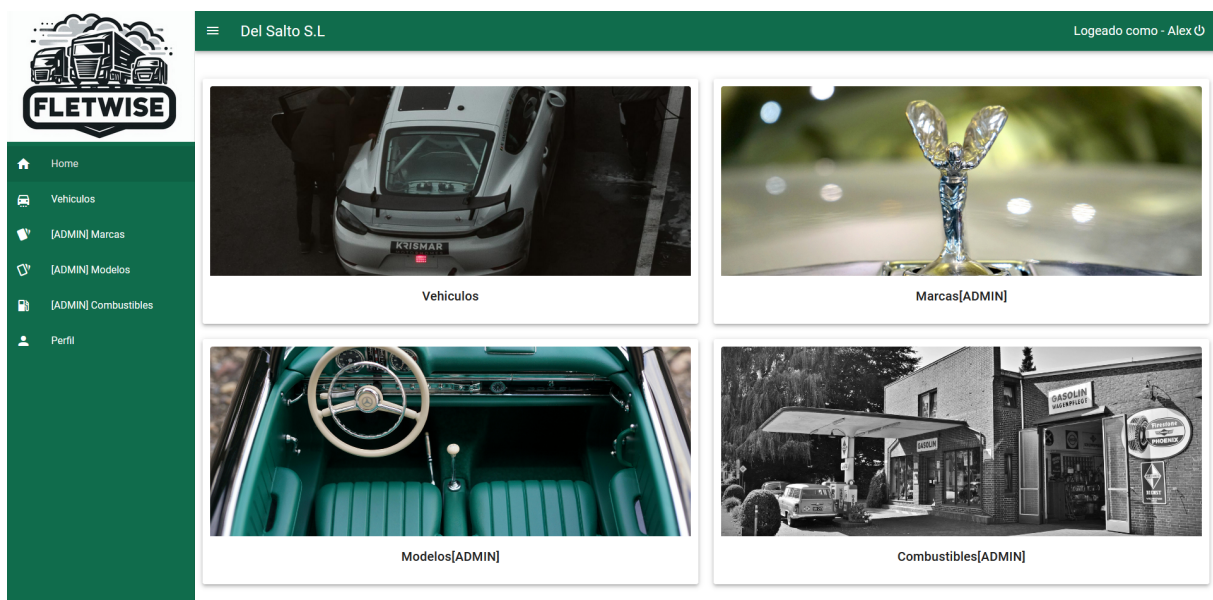


Figura 9: Home como administrador

## 4.4 Vehículos

Del Salto S.L.

Logeado como - Alex

Registro de Vehículos

Filtro marca

Filtro modelo

Filtro combustible

+

Matrícula	Marca	Modelo	Combustible
0004921	Audi	A1	GASOIL
001427D	Ford	Kuga	GASOIL
0063684	Citroen	C4	GASOIL
012F2ED	Ford	Kuga	GASOIL
0155FA8	Ford	Kuga	GASOIL

Buscar por matrícula

Items per page  
5 vehículos/página

<

1

2

3

...

198

199

200

>

Todos los derechos reservados - Tech Labs ©

Figura 10: Vista de vehículos

## 4.5 Marcas

Del Salto S.L.

Logeado como - Alex

Registro de Marcas

+

Nombre
Audi
Citroen
Ford
Mercedes

Buscar por nombre

Items per page  
5 marcas/página

<

1

>

Todos los derechos reservados - Tech Labs ©

Figura 11: Vista de marcas










## 4.6 Modelos

Del Salto S.L.

Logeado como - Alex

Registro de Modelos

Filtro marca

Nombre	Marca	
A1	Audi	 
A3	Audi	 
A4	Audi	 
C3	Citroen	 
C4	Citroen	 

Buscar por nombre

Items per page  
5 Modelos/página

<

1

2

>

Todos los derechos reservados - Tech Labs ©


**Figura 12:** Vista de modelos

## 4.7 Combustibles

Del Salto S.L.

Logeado como - Alex

Registro de Combustibles

Nombre	
GASOIL	 
Gasolina sin plomo 95	 

Buscar por nombre

Items per page  
5 Combustibles/página

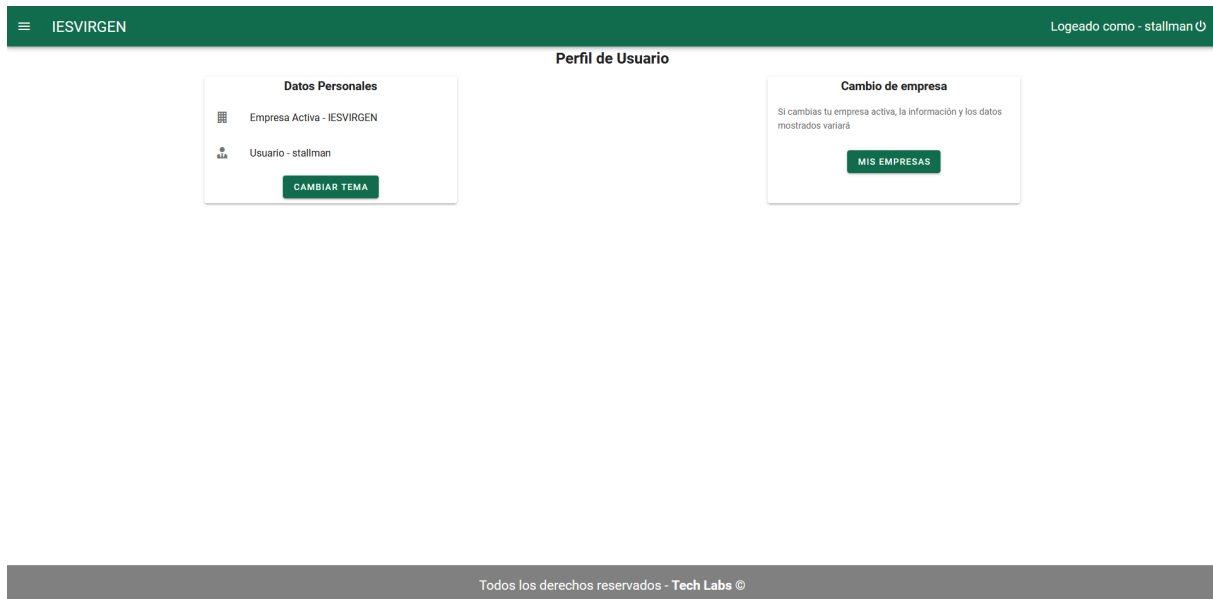
<

>

Todos los derechos reservados - Tech Labs ©

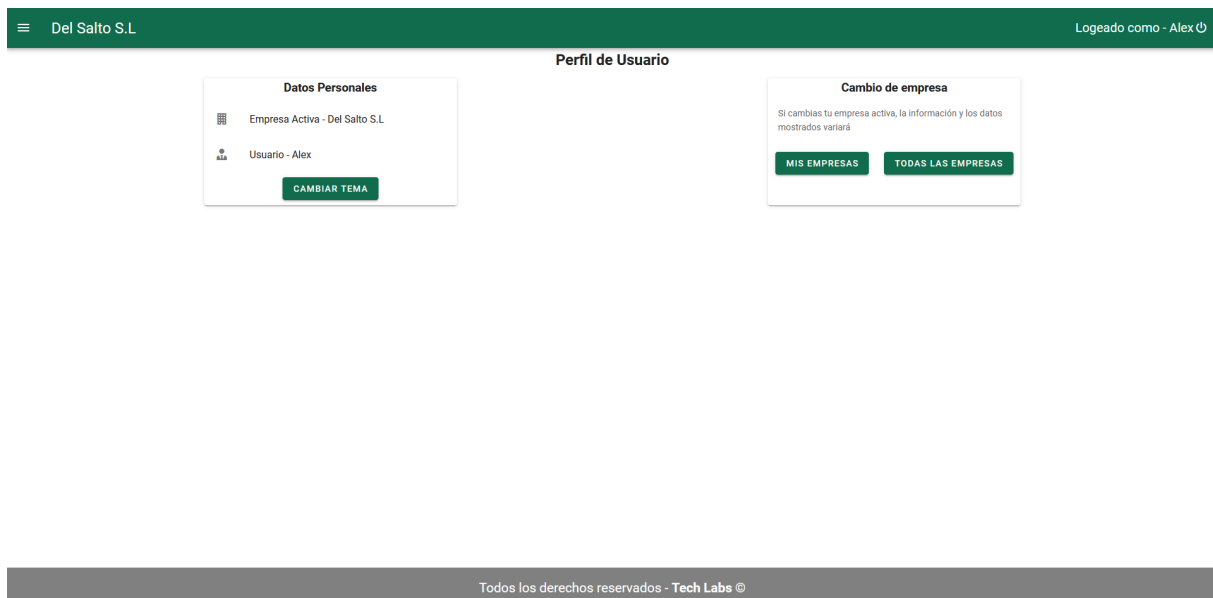
**Figura 13:** Vista de combustibles

## 4.8 Perfil - Usuario



**Figura 14:** Perfil como usuario

## 4.9 Perfil - Administrador



**Figura 15:** Perfil como administrador

## 5 Implementación

### 5.1 Back-End

Comenzando con el Back-End, partimos de unas bases de datos ya creadas, por lo que nuestro punto de partida es Database first, siendo así necesario generar el contexto y los modelos de la base de datos para hacer uso de ella mediante Entity Framework:

```
1 #Consola
2 dotnet ef dbcontext scaffold "server=(local);database=AppEF;
   Trusted_Connection=True" Microsoft.EntityFrameworkCore.SqlServer --
   context-dir Contexto --output-dir Modelos
```

Este es el comando usado para generar el contexto en nuestro proyecto, por lo que deberemos sustituir los valores de server y database por nuestra cadena de conexión y la base de datos que queremos usar.

#### 5.1.1 Uso del contexto y los modelos

Una vez generado el contexto y los modelos proseguiremos haciendo la estructura que va a tener nuestra api para poder conectarse con la base de datos.

La idea es añadir capas de abstracción entre nuestro controlador y nuestro contexto para mantener nuestro código ordenado y fácilmente accesible, además de facilitarnos la lectura del mismo, evitando tener un solo archivo con miles de líneas de código (aunque mientras más complejo es el proyecto, más difícil es evitar esto).

### 5.1.2 Patrón a utilizar

Haremos uso de un patrón repositorio-servicio, lo que significa que contaremos con nuestro repositorio, el cual se comunicará con el contexto y manejará los datos para devolver aquello que solicitamos, enviando esos datos al servicio y este los devolverá al Front-End.

La estructura será simple, contaremos una interfaz en nuestro repositorio, donde indicaremos las funciones que este incluirá.

```
1 //IVehiculoRepository.cs
2 public interface IVehiculoRepository {
3     Task<Vehiculo> GetById(int id);
4     Task Insert(Vehiculo entity, int idCreador, string conn);
5     void Update(Vehiculo entity, int idEditor);
6     Task DeleteById(int id, int idBorrador);
7     IQueryable<Vehiculo> GetAll(int idEmpresa, bool obtenerDesactivados
8         = false);
9     PagedList<Vehiculo> GetAllPaginated(VehiculoParams parameters, int
10         idEmpresa, bool obtenerDesactivados = false);
11     void SearchByName(ref IQueryable<Vehiculo> vehiculos, string
12         matricula);
13     void ApplySort(ref IQueryable<Vehiculo> query, string
14         orderByQueryString);
15     int NextIdSequence(string conn);
16 }
```

E interfaces en cada uno de nuestros servicios, donde también se mostrarán las funciones que estos incluirán.

```
1 //IVehiculoServicio.cs*
2 public interface IVehiculoService {
3     Task<VehiculoDTO> GetById(int Id, string cadenaConexion);
4     Task<Vehiculo> GetByIdNoMap(int Id, string cadenaConexion);
5     Task<List<VehiculoDTO>> GetVehiculos(int idEmpresa, string
6         cadenaConexion);
7     Task<(List<VehiculoDTO> listadoVehiculosDto, MetadataDto
8         metadataDto)> GetVehiculosPaginated(VehiculoParams parameters,
9         string cadenaConexion, int idEmpresa);
10     Task AddVehiculo(VehiculoDTO vehiculo, string cadenaConexion, int
11         idCreador);
12     Task UpdateVehiculo(VehiculoDTO vehiculo, string cadenaConexion,
13         int idEditor);
14     Task DeleteById(int id, string cadenaConexion, int idBorrador);
15 }
```

Con solo ver esto, os podréis imaginar que este proceso tiene que repetirse tantas veces como modelos tengamos, lo que a primera vista se siente como algo innecesario ya que .NET cuenta con algo llamado genéricos, y esto significa que podemos crear un repositorio con todas las funciones que sea de



tipo genérico (hace referencia a una clase indefinida, es decir, sirve para todo) y una vez hecho eso, simplemente deberíamos crear instancias del repositorio genérico pero asignándole un tipo, de esa forma solo tenemos un repositorio en vez de decenas de ellos.

Esto es totalmente válido ya que es una buena forma de trabajar, sin embargo, anteriormente nombramos que vamos a trabajar con campos de auditoría, estos concretamente indican qué persona hizo que cosa, ya sea quién insertó el registro, quien lo modificó y quien lo borró junto a sus respectivas fechas y horas.

No suena como algo negativo, pero como al trabajar con tipos genéricos, realmente no tiene un tipo definido, no es posible acceder a sus propiedades.

Esto claramente tiene solución, y es buscar la forma de indicar que ese repositorio genérico va a manejar objetos cuyas clases contengan los campos de auditoría que necesitamos, y realmente no es muy complejo de hacer, pero no siempre podemos usar este método así que inevitablemente en algunos casos vamos a tener que crear un repositorio por cada modelo que tengamos, sin embargo, el repositorio genérico y los específicos se suelen usar de forma conjunta ya que hay casos en los que vas a necesitar uno u otro, más adelante lo veremos.

```
1 //IGenericRepository
2 public interface IGenericRepository<TEntity> where TEntity : class {
3     Task<TEntity> GetById(int id);
4     Task Insert(TEntity entity);
5     Task Update(TEntity entity);
6     Task DeleteById(int id);
7     IQueryable<TEntity> GetAll();
8     PagedList<TEntity> GetAllPaginated(QueryStringParameters parameters
9     );
9 }
```

Ahora necesitamos un área de trabajo, ahí vamos a almacenar tanto nuestros repositorios específicos como las instancias de nuestro repositorio general y funciones que queramos darle.

```
1 //IUnitOfWork.cs
2 public interface IUnitOfWork : IDisposable {
3     IVehiculoRepository VehiculoRepository { get; }
4     IMarcaRepository MarcaRepository { get; }
5     IModeloRepository ModeloRepository { get; }
6     IGenericRepository<Usuario> UsuariosRepository { get; }
7     IGenericRepository<EmpresasActiva> EmpresasActivaRepository { get;
8     }
9     IEmpresaRepository EmpresaRepository { get; }
10    ICombustibleRepository CombustibleRepository { get; }
11    IGenericRepository<EmpresasUsuario> EmpresasUsuarioRepository {
12        get; }
13    ICambioEmpresaRepository RepositorioCambioEmpresa { get; }
14    public int SaveChanges();
15    public Task<int> SaveChangesAsync();
16 }
```

Con ver esta interfaz ya os podréis imaginar lo que está pasando, y la realidad es que como dije anteriormente, el repositorio genérico y los específicos pueden trabajar de forma conjunta, siendo así que en los casos en los que vaya a necesitar tocar campos de auditoría, voy a crear un repositorio nuevo con todo lo que conlleva, es decir, la duplicación de código, pero en los casos en los que no lo necesite como los usuarios, ya que en este caso no vamos a añadir usuarios, ni modificarlos o borrarlos, lo mismo pasa con nuestras empresas activas y empresas usuarios, no vamos a tocar auditoría, por lo que podemos hacer uso de nuestro repositorio genérico y simplemente le establecemos el tipo (la clase) a la que van a pertenecer.

De esta forma, desde nuestra unidad de trabajo vamos a tener acceso a todos los repositorios, sin importar que sean repositorios específicos o instancias del genérico.

```
1 //UnitOfWork.cs
2 public class UnitOfWork : IUnitOfWork, IDisposable {
3     private readonly RegistroGeneralContext _context;
4     private string _cadenaConexion;
5
6     public UnitOfWork(RegistroGeneralContext context) {
7         _context = context;
8     }
9
10    public UnitOfWork(string? cadenaConexion) {
11        _cadenaConexion = cadenaConexion;
12        _context = new RegistroGeneralContext (_cadenaConexion);
13    }
14
15    private IVehiculoRepository? _repositorioVehiculo;
16    private IModeloRepository? _repositorioModelo;
17    private IMarcaRepository? _repositorioMarca;
18    private IGenericRepository<Usuario>? _repositorioUsuario;
19    private IGenericRepository<EmpresasActiva>?
20        _repositorioEmpresasActiva;
21    private IEmpresaRepository? _repositorioEmpresa;
22    private ICombustibleRepository? _repositorioCombustible;
23    private IGenericRepository<EmpresasUsuario>?
24        _repositorioEmpresasUsuario;
25    private ICambioEmpresaRepository? _repositorioCambioEmpresa;
26
27    IGenericRepository<Usuario> IUnitOfWork.UsuariosRepositorio =>
28        _repositorioUsuario ??= new GenericRepository<Usuario>(_context)
29        ;
30    IGenericRepository<EmpresasActiva> IUnitOfWork.
31        EmpresasActivaRepositorio => _repositorioEmpresasActiva ??= new
32        GenericRepository<EmpresasActiva>(_context);
33    IGenericRepository<EmpresasUsuario> IUnitOfWork.
34        EmpresasUsuarioRepositorio => _repositorioEmpresasUsuario ??=
35        new GenericRepository<EmpresasUsuario>(_context);
36    public ICombustibleRepository CombustibleRepositorio =>
37        _repositorioCombustible ??= new CombustibleRepository(_context);
38    public IModeloRepository ModeloRepositorio => _repositorioModelo
39        ??= new ModeloRepository(_context);
40    public IMarcaRepository MarcaRepositorio => _repositorioMarca ??=
41        new MarcaRepository(_context);
42    public ICambioEmpresaRepository RepositorioCambioEmpresa =>
43        _repositorioCambioEmpresa ??= new CambioEmpresaRepository(
44        _context);
45    public IEmpresaRepository EmpresaRepositorio => _repositorioEmpresa
46        ??= new EmpresaRepository(_context);
47    public IVehiculoRepository VehiculoRepositorio =>
48        _repositorioVehiculo ??= new VehiculoRepository(_context);
```

```
34
35
36     public void Dispose() {
37         Dispose(true);
38         GC.SuppressFinalize(this);
39     }
40
41     protected virtual void Dispose(bool disposing) {
42         _context?.Dispose();
43     }
44
45     public int SaveChanges() {
46         return _context.SaveChanges();
47     }
48
49     public Task<int> SaveChangesAsync() {
50         return _context.SaveChangesAsync();
51     }
52 }
```

Y en nuestra unidad de trabajo, implementamos la interfaz.

### 5.1.3 Repositorio General

```
1 //GenericRepository.cs
2 public class GenericRepository : IGenericRepository
   where TEntity : class {
3
4     private readonly DbSet _entity;
5     public GenericRepository(RegistroGeneralContext context) {
6         _entity = context.Set();
7     }
8
9     public async Task GetById(int id) {
10         return await _entity.FindAsync(id);
11     }
12
13     public async Task Insert(TEntity entity) {
14         _entity.Add(entity);
15     }
16
17     public async Task Update(TEntity entity) {
18         _entity.Update(entity);
19     }
20
21     public async Task DeleteById(int id) {
22         var elem = await _entity.FindAsync(id);
23         if (elem != null) {
24             _entity.Remove(elem);
25         }
26     }
27 }
```

Ahora solo queda implementar interfaz en nuestro repositorio genérico y en nuestros repositorios específicos, la única diferencia será que nuestro genérico trabajará con TEntity haciendo referencia a una clase cualquiera, y en nuestros específicos podremos tratar con los modelos concretos y tendremos acceso a sus propiedades.

```
1 //VehiculoRepository
2 public class VehiculoRepository : IVehiculoRepository {
3     private readonly DbSet<Vehiculo> _vehiculos;
4     private readonly DbSet<Modelo> _modelos;
5     private readonly DbSet<Marca> _marcas;
6     private readonly RegistroGeneralContext _context;
7     public VehiculoRepository(RegistroGeneralContext context) {
8         _vehiculos = context.Set<Vehiculo>();
9         _modelos = context.Set<Modelo>();
10        _marcas = context.Set<Marca>();
11        _context = context;
12    }
13
14    public async Task<Vehiculo> GetById(int id) {
15        var query = _vehiculos.Include(e => e.Modelo).Include(e => e.
16            Marca).Include(e => e.TipoCombustible).Where(x => (x.
17                BorradoLogico == false || x.BorradoLogico == null) && x.
18                FechaBorradoLogico == null).ToList();
19        return query.Where(x => x.Id == id).FirstOrDefault();
20    }
21
22    public async Task Insert(Vehiculo entity, int idCreador, string
23        conn) {
24        entity.IdCreador = idCreador;
25        entity.FechaCreacion = DateTime.Now;
26        entity.GuidRegistro = Guid.NewGuid().ToString().ToUpper();
27        entity.Id = NextIdSequence(conn);
28        entity.Activo = true;
29        _vehiculos.Add(entity);
30    }
31
32    public void Update(Vehiculo entity, int idEditor) {
33        entity.IdEditor = idEditor;
34        entity.FechaModificacion = DateTime.Now;
35        _vehiculos.Update(entity);
36    }
37
38    public async Task DeleteById(int id, int idBorrador) {
39        var elem = await _vehiculos.FindAsync(id);
40        if (elem != null) {
41            elem.IdBorrador = idBorrador;
42            elem.FechaBorradoLogico = DateTime.Now;
43            elem.BorradoLogico = true;
44
45            _vehiculos.Update(elem);
46        }
47    }
48 }
```

Como se puede apreciar, hay bastante diferencia entre uno y otro y no vamos a mostrar el archivo entero ya que ahora mismo no es necesario.

Sin embargo, si hay una función importante que mostrar y esa es el GetAll.

```
1 //VehiculoRepository.cs
2 public PagedList<Vehiculo> GetAllPaginated(VehiculoParams parameters,
3     int idEmpresa, bool obtenerDesactivados = false) {
4     var query = _vehiculos.AsNoTracking().Include(e => e.Modelo).
5         Include(e => e.Marca).Include(e => e.TipoCombustible).Where(e =>
6             e.IdEmpresa == idEmpresa);
7
8     if (parameters.Marca != null) {
9         query = query.Where(e => e.MarcaId == parameters.Marca);
10    }
11
12    if (parameters.Modelo != null) {
13        query = query.Where(e => e.ModeloId == parameters.Modelo);
14    }
15
16    if (parameters.Combustible != null) {
17        query = query.Where(e => e.TipoCombustibleId == parameters.
18            Combustible);
19    }
20
21    if (!obtenerDesactivados) {
22        query = query.Where(e => e.BorradoLogico == null || e.
23            BorradoLogico == false);
24    }
25
26    SearchByName(ref query, parameters.Matricula);
27    ApplySort(ref query, parameters.OrderBy);
28
29    return PagedList<Vehiculo>.ToPagedList(query, parameters.PageNumber
30        , parameters.PageSize);
31 }
```

Primero que todo, estamos devolviendo un Pagedlist de tipo vehículo, pagedlist no es más que una clase de utilidad creada para poder hacer que nuestros getAll no devuelvan todos los registros de nuestras tablas, sino que se devuelvan paginados en tandas de 5, 10 o 20 registros para no saturar el front con datos, también tenemos un objeto de la clase VehiculoParams, que cuenta con la información del tamaño de página, numero de página y otros datos útiles.

Por otro lado, estaremos pasando un idEmpresa, el cual nos sirve para devolver solo los vehiculos que pertenezcan a la empresa que queremos, y un boolean preestablecido a falso, esto se debe a que no realizaremos borrados físicos sino lógicos, es decir, el registro no se borra de la base de datos, sino que rellenamos campos como borradoLogico, idBorrador y fechaBorradoLogico para filtrar por aquellos

elementos los cuales no estén marcados como borrados, y por ultimo tenemos una llamada hacia search by name y applySort.

Como indiqué anteriormente, toda la paginación, filtrado, búsqueda y ordenación se obtuvieron siguiendo el tutorial de CodeMaze → <https://code-maze.com/net-core-series/>.

#### 5.1.4 Servicios

Anteriormente mostramos la interfaz de nuestro servicio de vehículos, donde vemos aquellas funciones que queremos darle al servicio en concreto, esto lo que va a realizar es una llamada hacia la función correspondiente en nuestro repositorio.

```
1 //ServicioVehiculo.cs
2 public class VehiculoService : IVehiculoService {
3     private readonly IUnitOfWork _repositorioEspecifico;
4     private readonly RegistroGeneralContext _context;
5     private readonly IMapper _mapper;
6
7     public VehiculoService(IUnitOfWork repositorioEspecifico,
8         RegistroGeneralContext context, IMapper mapper) {
9         _repositorioEspecifico = repositorioEspecifico;
10        _context = context;
11        _mapper = mapper;
12    }
13
14    public async Task AddVehiculo(VehiculoDTO vehiculoDTO, string
15        cadenaConexion, int idCreador) {
16        using (IUnitOfWork repositorioEspecifico = new UnitOfWork(
17            cadenaConexion)) {
18            Vehiculo vehiculo = _mapper.Map<Vehiculo>(vehiculoDTO);
19            await repositorioEspecifico.VehiculoRepositorio.Insert(
20                vehiculo, idCreador, cadenaConexion);
21            repositorioEspecifico.SaveChanges();
22        }
23    }
24
25    public async Task DeleteById(int id, string cadenaConexion, int
26        idBorrador) {
27        using (IUnitOfWork repositorioEspecifico = new UnitOfWork(
28            cadenaConexion)) {
29            await repositorioEspecifico.VehiculoRepositorio.DeleteById(
30                id, idBorrador);
31            repositorioEspecifico.SaveChanges();
32        }
33    }
34 }
```



Aquí mostramos un par de funciones de nuestro servicio de vehículos, donde crearemos una nueva instancia de nuestro repositorio para cambiar hacia que base de datos hacemos la petición (esto ahora mismo no es importante, será explicado más adelante) y lo único que hacemos es una llamada a nuestro repositorio, o mapeamos el objeto recibido y lo enviamos al repositorio, esto se debe a que es bueno seguir una gestión de que parte trata con que tipos, pongamos un ejemplo.

En mi caso, quiero que mis controladores trabajen solo con dto's, que mis servicios realicen los mapeos y que mis repositorios solo trabajen con el modelo.

### 5.1.5 Controlador Vehículo

Una vez tenemos todos estos pasos, ya solo nos queda crear nuestro controlador, pongamos un ejemplo con nuestro controlador de vehículos.

```
1 //VehiculoController.cs
2 [ApiController]
3 [Route("[controller]")]
4 public class VehiculoController : ControllerBase {
5
6     private readonly IVehiculoService _vehiculoServicio;
7     private readonly IEmpresaService _empresaServicio;
8     private readonly ICombustibleService _combustibleServicio;
9     private readonly IMarcaService _marcaServicio;
10    private readonly IModeloService _modeloServicio;
11    private readonly IConfiguration _configuration;
12    private readonly JwtSettings _jwtSettings;
13
14    public VehiculoController(IVehiculoService vehiculoServicio,
15                             ICombustibleService combustibleServicio, IConfiguration
16                             configuration, IOptions<JwtSettings> options, IEmpresaService
17                             empresaServicio, IMarcaService marcaServicio, IModeloService
18                             modeloServicio) {
19        _vehiculoServicio = vehiculoServicio;
20        _combustibleServicio = combustibleServicio;
21        _configuration = configuration;
22        _jwtSettings = options.Value;
23        _empresaServicio = empresaServicio;
24        _marcaServicio = marcaServicio;
25        _modeloServicio = modeloServicio;
26    }
27
28    //GET
29    [HttpGet("getVehiculo")]
30    public async Task<ActionResult<VehiculoDTO>> GetVehicle(int id) {
31
32        try {
33            string conn = await _empresaServicio.
34                GenerarCadenaDeConexionAsync((int)JwtUtil.
35                ObtenerDatosEmpresaPetition(Request.Headers,
36                _configuration, _jwtSettings));
37            VehiculoDTO vehiculo = await _vehiculoServicio.GetById(id,
38                conn);
39
40            if (vehiculo == null) {
41                return StatusCode(404, "No se encuentra el vehículo
42                buscado");
43            } else {
44                return StatusCode(200, vehiculo);
45            }
46        }
47    }
```

```
37
38     } catch (Exception ex) {
39         return StatusCode(401, "Token no valido o inexistente");
40     }
41 }
42 }
```

Hagamos un repaso sobre este código:

Obtenemos un vehículo de nuestro servicio, el cual puede ser null, es decir, devolver un dato vacío, por lo que siempre debemos comprobar los errores y devolver el código de error para un correcto manejo desde el Front-End.

En caso de que el dato devuelto no sea vacío, haremos uso de una DTO para devolver los datos necesarios, ya que al hacer peticiones a bases de datos, estaríamos devolviendo el objeto entero, el cual contiene información que para el usuario es totalmente irrelevante.

### 5.1.6 Inyección

Cabe destacar que todo esto te dará un error si no inyectamos nuestro contexto ni nuestros repositorios y servicios, por lo que necesitaremos añadir unas cuantas líneas a nuestro program.cs.

```
1 //program.cs
2 builder.Services.AddDbContext<ContextoDB>(item => item.UseSqlServer(
    configuration.GetConnectionString("cadena")));
3 builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();
4 builder.Services.AddScoped<IVehiculoService, VehiculoService>();
```

Por cada servicio que añadamos, debemos inyectarlo también.

“cadena” es un string que tenemos guardado en nuestro appsettings.json y nos sirve precisamente para poder asignar nuestra cadena de conexión, ya que un contexto sin lugar al que apuntar no nos sirve de nada.

```
1 //appSettings.json
2 "ConnectionStrings": {
3     "cadena": "server=user;database=database;Trusted_Connection=True;
    TrustServerCertificate=True"
4 },
```

Trusted\_Connection hace referencia a que estamos usando el certificado de windows, pero será necesario cambiar esto y añadir nuestro usuario y contraseña para subir el proyecto y que funcione desde un hosting

### 5.1.7 Control de CORS

A la hora de intentar comunicar nuestro Front-End con la API, lo más probable que es que recibamos errores de CORS, por lo que también deberemos añadir en nuestro program.cs las siguientes líneas.

```
1 //program.cs
2 builder.Services.AddCors(options =>
3 {
4     options.AddDefaultPolicy(
5         policy => {
6             policy.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod();
7         });
8 });
```

Por defecto, no nos interesa añadir restricciones a la aplicación, queremos aceptar a todos los usuarios sin importar el origen, cabeceras ni peticiones que realicen.

### 5.1.8 Manejo de Swagger

El swagger es una herramienta realmente útil a la hora de probar nuestra API cuando no tenemos Front-End creado, nos permite acceder a cada uno de nuestros EndPoints(Funciones por cada uno de nuestros controladores) y así comprobar su funcionamiento, el uso es similar a PostMan.

Sin embargo, Swagger por defecto hay cosas que no implementa, y nosotros deberemos añadirlas, y la principal es el uso de token.

En nuestra aplicación, cada petición realizada consultará las cabeceras en busca de un token, el cual pasará por un proceso de validación, y si desde nuestras peticiones por Swagger no incluimos token, siempre recibiremos errores, por lo que para activar esta opción, deberemos escribir las siguientes líneas.

```
1 //program.cs
2 builder.Services.AddSwaggerGen(opt =>
3 {
4     opt.SwaggerDoc("v1", new OpenApiInfo { Title = "Titulo", Version =
5         "v1" });
6     opt.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
7     {
8         In = ParameterLocation.Header,
9         Description = "Please enter token",
10        Name = "Authorization",
11        Type = SecuritySchemeType.Http,
12        BearerFormat = "JWT",
13        Scheme = "bearer"
14    });
15    opt.AddSecurityRequirement(new OpenApiSecurityRequirement
16    {
17        {
18            new OpenApiSecurityScheme
19            {
20                Reference = new OpenApiReference
21                {
22                    Type=ReferenceType.SecurityScheme,
23                    Id="Bearer"
24                }
25            },
26            new string[]{}
27        }
28    });
29 });
```

Y también necesitaremos establecer el MiddleWare, que será el que compruebe la existencia del token en nuestras peticiones.

```
1 //Program.cs
2 //Validación del token
3 builder.Services.AddAuthentication(JwtBearerDefaults.
    AuthenticationScheme)
4     .AddJwtBearer(options =>
5         options.TokenValidationParameters = new TokenValidationParameters
6         {
7             ValidateIssuer = true,
8             ValidateAudience = false,
9             ValidateLifetime = true,
10            ValidateIssuerSigningKey = true,
11            ValidIssuers = new[] { configuration["JwtSettings:Issuer"],
12                                   configuration["JwtSettings:Audience"] },
13            ValidAudiences = new[] { configuration["JwtSettings:Issuer"],
14                                     configuration["JwtSettings:Audience"] },
15            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.
16                GetBytes(configuration["JwtSettings:KeySecret"])),
17            ClockSkew = TimeSpan.Zero
18        });
19
20 builder.Services.AddAuthorization(options => {
21     options.FallbackPolicy = new AuthorizationPolicyBuilder()
22         .RequireAuthenticatedUser()
23         .Build();
24 });
```

Esto hará que las proximas veces que se abra el Swagger, contemos con una opción para añadir token en las peticiones que realicemos.

Sin embargo, actualmente no contamos con nuestra creación de token, por lo que lo dejaremos para más adelante.

### 5.1.9 Personalización de Swagger

Otro dato a destacar, es que el Swagger acepta hojas de estilo, por lo que podemos ponerlo en modo oscuro si nos resulta más cómodo consultando esta página → Swagger en modo oscuro → <https://dev.to/amoenus/turn-swagger-theme-to-the-dark-mode-4l5f>.

### 5.1.10 DTO'S

A la hora de hacer consultas y devolver datos, anteriormente dijimos que no es rentable el devolver el objeto con todas sus propiedades, por lo que debemos crear DTO's para cada objeto que queramos devolver, de hecho, en algunas ocasiones, nos interesará devolver el mismo objeto pero con diferentes propiedades dependiendo de la petición que se realice, esto se hace estableciendo las propiedades que variarán como nullables (permiten registrar el valor null), ya que en ocasiones, simplemente no nos es útil un valor del objeto y no queremos que de un error por no ponerlo, aunque otra opción es crear otra dto para el mismo objeto, cada persona decide su forma de trabajar.

```
1 //VehiculoDTO.cs
2 public class VehiculoDTO
3 {
4     public int? Id { get; set; }
5     public string? Matricula { get; set; }
6     public int? IdMarca { get; set; }
7     public string? Marca { get; set; }
8     public int? IdModelo { get; set; }
9     public string? Modelo { get; set; }
10    public int? Id_TipoCombustible { get; set; }
11    public string? Combustible { get; set; }
12 }
13 //MarcaDTO.cs
14 public class MarcaDTO
15 {
16     public int Id { get; set; }
17     public string? Nombre { get; set; }
18     public string? Observaciones { get; set; }
19     public string? Codigo { get; set; }
20     public int? Id_Empresa { get; set; }
21 }
22 //ModeloDTO.cs
23 public class ModeloDTO
24 {
25     public int? Id { get; set; }
26     public string? Nombre { get; set; }
27     public string? Observaciones { get; set; }
28     public string? ReferenciaExterna { get; set; }
29     public int? Id_Empresa { get; set; }
30     public int? Id_Marca { get; set; }
31 }
32 //CombustibleDTO.cs
33 public class CombustibleDTO
34 {
35     public int? Id { get; set; }
36     public string? Nombre { get; set; }
37     public int IdEmpresa { get; set; }
38 }
```

### 5.1.11 Login

Pasaremos a nuestro EndPoint de login, el cual hace uso de un servicio y un repositorio, sin embargo, una vez enseñada la estructura básica, es más que evidente que para hacer el login, vamos a hacer uso de un GetAll en busca del listado de usuarios, el cual nos devolverá un IQueryable que consultaremos con un where, haciendo búsqueda por su Email.

Una vez tengamos el usuario, debemos comprobar la contraseña, pero en la base de datos no guardamos contraseñas en texto plano sino que están encriptadas, por lo tanto, será necesario encriptar la contraseña que el usuario pase en el inicio de sesión y compararla con la contraseña asociada al Email en la tabla de usuarios, en caso de ser correcto, daremos acceso, generaremos el token y lo devolveremos.

```
1
2 //LoginController.cs
3 [HttpGet("login")]
4 public async Task<ActionResult<UsuarioEmpresaDTO>> CheckUser(string
   email, string pass)
5 {
6     try
7     {
8         Usuario user = await _sesionServicio.CheckUser(email, pass);
9         EmpresasActiva empresaActiva = await _empresaActivaServicio.
            CheckEmpresa(user.Id);
10        Empresa empresa = await _empresaServicio.CheckEmpresa(
            empresaActiva.IdEmpresa);
11        if (user == null || empresaActiva == null)
12        {
13            return StatusCode(404, "Error al iniciar sesión");
14        }
15        else
16        {
17            JwtSettings settings = new JwtSettings(
18                keySecret: _configuration.GetSection("JwtSettings").
                    GetSection("KeySecret").Value,
19                audience: _configuration.GetSection("JwtSettings").
                    GetSection("Audience").Value,
20                issuer: _configuration.GetSection("JwtSettings").GetSection
                    ("Issuer").Value);
21
22            (string token, DateTime fechaExpiracion) = JwtUtil.
                GenerateJwtEmpresaToken(email, user.GuidRegistro, user.
                    Id, empresa.GuidRegistro, empresaActiva.IdEmpresa,
                    settings);
23
24            UsuarioEmpresaDTO userDTO = new UsuarioEmpresaDTO(user.
                Nombre, user.Email, empresaActiva.IdUsuario,
                empresaActiva.IdEmpresa, token, fechaExpiracion, user.
```



```
25         GuidRegistro, empresa.GuidRegistro);  
26         return userDT0;  
27     }  
28     catch (Exception ex)  
29     {  
30         return StatusCode(400, ex.Message);  
31     }  
32     {  
33     }  
34 }
```

En este controlador hay más cosas de las explicadas, y esto se debe a que necesitamos datos como la empresa activa del usuario (ya que se implementará un sistema para cambiar la empresa activa, y con ello, los datos mostrados), y vemos que para la generación del token, hacemos uso de JwtSettings, cuyas 3 propiedades son consultadas en nuestro appSettings.json, el issuer es el emisor, el audience es el destinatario y el keySecret es el código que se utilizará para firmar ese token, muy importante mantenerlo en secreto.

```
1 //appSettings.json  
2 "JwtSettings": {  
3     "KeySecret": "example",  
4     "Issuer": "example",  
5     "Audience": "example"  
6 },
```

En cuanto a la función que necesitamos para generar el token, es esta.

```
1 //JwtUtil.cs
2 public static (string, DateTime) GenerateJwtEmpresaToken(string
    email, string guidUsuario, int idUsuario, string guidEmpresa,
    int idEmpresa, /*string urlComunidad,*/
3     JwtSettings jwtSettings, bool? esAdministrador = false) {
4
5
6     // Generamos claims en funcion de los parámetros
7     List<Claim> claims = new() {
8         new Claim(JwtRegisteredClaimNames.Email, email),
9         new Claim(Constants.GUID_USUARIO, guidUsuario),
10        new Claim(Constants.ID_USUARIO, idUsuario.ToString()),
11        new Claim(Constants.GUID_EMPRESA, guidEmpresa),
12        new Claim(Constants.ID_EMPRESA, idEmpresa.ToString()),
13        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().
            ToString()),
14        //new Claim(., urlComunidad)
15    };
16
17    if (esAdministrador == true) {
18        claims.Add(new Claim(Constants.ESADMINISTRADOR, "
            ESADMINISTRADOR"));
19    } else {
20        claims.Add(new Claim(Constants.ESUSUARIO, "ESUSUARIO"));
21    }
22
23
24    SymmetricSecurityKey keySecret = new(Encoding.UTF8.GetBytes(
        jwtSettings.KeySecret));
25    SigningCredentials credentials = new(keySecret,
        SecurityAlgorithms.HmacSha256);
26
27    DateTime expiration = DateTime.Now.AddHours(10);
28
29    JwtSecurityToken token = new(
30        issuer: jwtSettings.Issuer,
31        audience: jwtSettings.Audience,
32        claims: claims,
33        expires: expiration,
34        signingCredentials: credentials);
35
36    return (new JwtSecurityTokenHandler().WriteToken(token),
        expiration);
37 }
```

Mediante claims estamos añadiendo aquello que buscamos que nuestro token contenga, de esa forma podemos hacer que nuestro token indique si el usuario logeado es o no administrador.

### 5.1.12 Cambio dinámico de nuestra cadena de conexión

Al tratar con varias bases de datos en entity framework, es muy importante saber como funcionan el contexto y los modelos, ya que, cuando un contexto apunta a una base de datos, sus estructuras deben ser idénticas, es decir, no puede haber ninguna diferencia ni en nombres de tablas, ni en tipos de datos...

Esto, a pesar de verse como algo complicado de mantener, permite otra cosa, y es la utilización del mismo contexto con múltiples bases de datos cuya estructura interna sea idéntica (lo cual vamos a aprovechar para este proyecto).

Lo complicado será manejar las instancias del contexto, ya que nuestra idea es crear instancias de contextos que apunten a diferentes bases de datos en tiempo de ejecución, y para ello, necesitaremos hacer lo siguiente.

```
1 //context.cs
2 public partial class ContextoDB : DbContext
3 {
4
5     public ContextoDB(string cadenaConexion) {
6         _cadenaConexion = cadenaConexion;
7     }
8
9     private DbContextOptionsBuilder _optionBuilder;
10
11     private string? _cadenaConexion = string.Empty;
12
13
14
15     protected override void OnConfiguring(DbContextOptionsBuilder
16         optionsBuilder)
17     {
18
19         if (!optionsBuilder.IsConfigured)
20         {
21             optionsBuilder.UseSqlServer(_cadenaConexion);
22         }
23
24         _optionBuilder = optionsBuilder;
25     }
26 }
```

Es decir, además de los constructores que se generan al hacer el scaffold (creación del contexto a través del comando → Database-First), debemos crear un constructor que acepte una cadena de conexión, necesitaremos el optionsBuilder para establecer configuracion, y definiremos un string vacío, el cual nos será útil para almacenar nuestra cadena de conexión.

Tras esto, sobrescribiremos el método `onConfiguring` (este se encarga de establecer la forma en la que la base de datos se relaciona con el contexto), por lo que en caso de no haber configuración previa, simplemente conectaremos el contexto con la base de datos a la que apunte la cadena de conexión, pero ya veremos como generaremos la cadena más adelante.

Ahora deberemos volver a nuestra unidad de trabajo, y recordemos que teníamos un constructor que usaba el contexto, el cual vamos a mantener debido a que no siempre vamos a querer cambiar la cadena de conexión, tenemos una cadena predefinida y esa seguirá siendo usada.

```
1 //UnitOfWork.cs
2 public class UnitOfWork : IUnitOfWork, IDisposable {
3     private readonly RegistroGeneralContext _context;
4     private string _cadenaConexion;
5
6     public UnitOfWork(RegistroGeneralContext context) {
7         _context = context;
8     }
9
10    public UnitOfWork(string? cadenaConexion) {
11        _cadenaConexion = cadenaConexion;
12        _context = new RegistroGeneralContext (_cadenaConexion);
13    }
14 }
```

Crearemos otro constructor el cual acepte la cadena de conexión, y genere un nuevo contexto el cual apunte a una base de datos distinta.

Y ahora simplemente, en nuestros servicios necesitaremos recibir la cadena de conexión en nuestras funciones para poder hacer que estas se realicen en x base de datos.

```
1 //VehiculoService.cs
2 public async Task AddVehiculo(VehiculoDTO vehiculoDTO, string
3     cadenaConexion, int idCreador) {
4     using (IUnitOfWork repositorioEspecifico = new UnitOfWork(
5         cadenaConexion)) {
6         Vehiculo vehiculo = _mapper.Map<Vehiculo>(vehiculoDTO);
7         await repositorioEspecifico.VehiculoRepositorio.Insert(vehiculo
8             , idCreador, cadenaConexion);
9         repositorioEspecifico.SaveChanges();
10    }
11 }
```

Se crea una nueva instancia de la unidad de trabajo la cual apunta, como ya vimos con el constructor, crea una nueva instancia del contexto apuntando a x base de datos, por lo que usaremos esa instancia para hacer las proximas operaciones (este proceso debe ser repetido en todas y cada una de nuestras funciones en los servicios excepto los servicios relacionados con el manejo de las empresas, recordemos que los datos de las empresas ya figuran en la base de datos a la que apuntamos por

defecto).

Sin embargo, estamos usando interfaces, por lo que las interfaces también deben ser cambiadas indicando que reciben un string llamado cadenaConexion.

Una vez hecho esto, solo queda generar ese string.

Para ello, necesitamos añadir una función a nuestro servicio de empresas con nombre GenerarCadenaDeConexionAsync, obteniendo este un id.

```
1 //EmpresasService.cs
2 public async Task<string> GenerarCadenaDeConexionAsync(int id)
3 {
4
5     string cadenaDeConexion = "";
6
7     var empresa = await GetById(id);
8
9     string baseActiva = empresa.BaseActiva;
10    string conn = _configuration.GetConnectionString("cadena");
11
12    int igual1 = conn.IndexOf('=');
13    int igual2 = conn.IndexOf('=', igual1 + 1) + 1;
14
15    int punto1 = conn.IndexOf(';');
16    int punto2 = conn.IndexOf(';', punto1 + 1);
17
18    int length = punto2 - igual2;
19
20    string oldConn = conn.Substring(igual2, length);
21    cadenaDeConexion = conn.Replace(oldConn, baseActiva);
22
23    return cadenaDeConexion;
24 }
```

Como veremos, estamos realizando una llamada a una función que se encuentra dentro del mismo servicio en que definimos esta, por lo que nos resulta bastante útil.

Buscaremos nuestra empresa por id y accederemos a su campo de base activa, esto es importante hacerlo por que el nombre de nuestras bases de datos corresponde con los valores de base activa, por lo que una vez hecho esto, solo debemos jugar con la cadena de conexión para conseguir reemplazar el nombre de la base de datos por el nombre de la base de datos a la que queremos apuntar.

Una vez tenemos esto hecho, tan solo queda llamar a esta funcion desde el constructor, y pasar este valor en las peticiones que hagamos.

```
1 //VehiculoController.cs
2 [HttpGet("getVehiculo")]
3 public async Task<ActionResult<VehiculoDTO>> GetVehicle(int id) {
4
5     try {
6         string conn = await _empresaServicio.
            GenerarCadenaDeConexionAsync((int)JwtUtil.
            ObtenerDatosEmpresaPeticion(Request.Headers, _configuration,
            _jwtSettings));
7         VehiculoDTO vehiculo = await _vehiculoServicio.GetById(id, conn
            );
8
9         if (vehiculo == null) {
10             return StatusCode(404, "No se encuentra el vehículo buscado
                ");
11         } else {
12             return StatusCode(200, vehiculo);
13         }
14     } catch (Exception ex) {
15         return StatusCode(401, "Token no valido o inexistente");
16     }
17 }
18 }
```

Probablemente os preguntéis que es esto

```
1 //VehiculoController.cs
2 (int)JwtUtil.ObtenerDatosEmpresaPeticion(Request.Headers,
    _configuration, _jwtSettings)
```

Esto es una función que tenemos en una clase de utilidad la cual nos permite obtener el id de empresa en base a un token que le pasemos, por lo que mirará entre sus claims hasta que encuentre el id de empresa y simplemente lo devolverá

```
1 //JwtUtil.cs
2 public static int? ObtenerDatosEmpresaPeticion(IHeaderDictionary
   headers, IConfiguration configuration, JwtSettings jwtSettings, bool
   validarToken = true) {
3     int? idEmpresa = null;
4     try {
5         if (headers.ContainsKey("Authorization")) {
6             JwtSecurityToken? validatedToken = CheckToken(headers["
               Authorization"].ToString()[7..], configuration,
               jwtSettings, validarToken);
7             if (validatedToken != null) {
8                 idEmpresa = int.Parse(validatedToken.Claims.
               FirstOrDefault(x => x.Type == Constantes.ID_EMPRESA)
               ?.Value ?? "-1");
9             }
10        }
11    } catch (Exception ex) {
12        return null;
13    }
14    return idEmpresa;
15 }
```

Esta función la he obtenido buscando información sobre los tokens jwt por internet, por lo que lo único que he tenido que hacer ha sido adaptarla a mis necesidades.

## 5.2 Front-End

Una vez tenemos el Back-End Terminado, podemos comenzar a realizar el apartado visual, recordemos que estamos haciendo uso de Vuejs con vuetify, por lo que el sistema que usaremos será el de componentes y vistas, sin embargo, aquí lo más destacable no será el código html ni css, sino el código JavaScript, es decir, como realizamos nuestras peticiones y como manejamos los datos obtenidos.

### 5.2.1 Peticiones axios

Haremos uso de las peticiones que componen el patron CRUD, es decir, usaremos GET, POST, PUT y DELETE

La estructura de una petición es realmente simple, contamos con la cadena de conexión, los parámetros que enviaremos, los cuales son opcionales, un then en caso de éxito, y un catch en caso de fracaso.

```
1 //Login.vue
2 axios.get('https://localhost:7030/Login/login', {
3     params: { email: email, pass: pass },
4 })
5     .then(function (response) {
6         localStorage.setItem("token", JSON.stringify(response.data));
7
8         $router.push("/");
9     })
10    .catch(function (error) {
11        console.log("Error al tratar de hacer login");
12    });
```

Sin embargo, a la hora de realizar peticiones, dijimos que era importante enviar siempre el token en las cabeceras, ya que en caso contrario, recibiríamos un error → (401), Unauthorized.

En ese caso, será necesario añadir las siguiente línea en nuestro código antes de la petición.

```
1 const token = JSON.Parse(localStorage.GetItem('token'));
2 axios.defaults.headers.common["Authorization"] = 'Bearer ${token.token}';
```

Como en nuestro token vamos a guardar muchos datos que nos van a ser útiles, token.token hace referencia al código generado que debe ser validado, pero también guardaremos datos como token.fechaExpiracion, aunque esta viene dentro del propio token.

Hemos enseñado como mandar un token en las cabeceras de nuestra petición, sin embargo, ¿es realmente buena idea hacer esto?.

Bueno, dado a que nuestro token tiene un tiempo de vida limitado, eso significa que en cuanto caduque, dejará de ser válido y nuestra sesión debe expirar, por lo tanto, no es recomendable controlar en cada



petición una por una si el token es o no válido, para esto, hacemos uso de los interceptores de axios.

Un interceptor establece el patrón de funcionamiento para todas y cada una de nuestras peticiones y respuestas, por lo que contaremos con un interceptor para peticiones, y otro para respuestas.

```
1 //Axios.ts
2 axios.interceptors.request.use(
3   function (config) {
4
5     const token = localStorage.getItem("token");
6
7     if (token) {
8       const tokenParse = JSON.parse(token);
9       config.headers.Authorization = 'Bearer ${tokenParse.token}';
10    }
11    return config;
12  },
13  function (error) {
14    return Promise.reject(error);
15  }
16 );
```

En todas nuestras peticiones, vamos a comprobar si tenemos token en el localStorage y vamos a enviarlo con las cabeceras.

```
1 //Axios.ts
2 import axios from "axios";
3 import router from "../router";
4
5 export default function axiosSetUp() {
6   axios.defaults.baseURL = "https://localhost:7030/";
7   axios.interceptors.response.use(
8     function (response) {
9
10      return response;
11    },
12    async function (error) {
13
14      const originalRequest = error.config;
15
16      const tokenNoParse = localStorage.getItem("token");
17      if (tokenNoParse) {
18        const token = JSON.parse(tokenNoParse);
19        const fecha = new Date(token.fechaExpiracion);
20
21        if (error.response.status === 401 && (Date.now() - fecha.getTime()) >= 300000) {
22          localStorage.clear();
23          router.push("/login");
24          return Promise.reject(error);
25        } else if (error.response.status === 401 && (Date.now() - fecha
```

```
        .getTime() < 300000)) {
26         renovarToken();
27         originalRequest._retry = true;
28         return axios(originalRequest);
29     }
30     return Promise.reject(error);
31 }
32 }
33 );
34 }
```

Y en todas nuestras respuestas vamos a comprobar si hay errores, por lo que en caso de obtener un error con http status 401, es decir, unauthorized, y el tiempo que nuestro token ha estado caducado sea mayor a 5 minutos, no vamos a permitir renovarlo, nos mandará al login.

Sin embargo, si el token caducó hace menos de 5 minutos, trataremos de renovarlo, siendo así que en caso de conseguirlo, repetiremos de nuevo la petición que nos dio el error unauthorized instantáneamente.

```
1 //Axios.ts
2 const renovarToken = () => {
3     const tokenSinParse = localStorage.getItem("token");
4     if (tokenSinParse) {
5         const token = JSON.parse(tokenSinParse);
6         const params = {
7             params: {
8                 nombre: token.nombre,
9                 email: token.email,
10                 userId: token.userId,
11                 idEmpresa: token.idEmpresa,
12                 guidEmpresa: token.guid_Empresa,
13                 guidUsuario: token.guid_Usuario,
14             },
15         };
16         axios
17             .get("https://localhost:7030/Login/RestoreToken", params)
18             .then((response) => {
19                 localStorage.setItem("token", JSON.stringify(response.data));
20                 localStorage.setItem("refresh", JSON.stringify(response.data));
21             })
22             .catch((error) => {
23                 localStorage.clear();
24                 router.push('/login')
25             });
26     }
27 }
```

Para nuestra función de renovar token, simplemente usaremos el token caducado con los datos válidos, y generaremos uno nuevo, es decir, tan solo cambiará el token en concreto (la cadena de caracteres serializada), y la fecha de expiración, por lo que al verificar el token de nuevo, volvemos a tener el token funcional durante el tiempo de vida del mismo.

Finalmente, para añadir el interceptor, entramos en nuestro main.ts, importamos el archivo y hacemos una llamada a la función.

```
1 //Main.ts
2 import axiosSetup from '../src/helpers/axios';
3
4 axiosSetup();
```

## 5.3 Apartados destacados

### 5.3.1 Cambio de empresa activa

Como se dijo en un principio, contamos con varias empresas en nuestra base de datos principal, las cuales contienen sus datos en otras bases de datos, referenciado mediante el campo base activa.

Ahora solo nos queda añadir la funcionalidad del cambio de empresa, y la idea es simple.

Necesitaremos un repositorio exclusivo para este proceso, por lo que necesitaremos interfaz, repositorio, otra interfaz, servicio y un controlador, aunque como anteriormente ya tenía un controlador de empresas creado, no será necesario añadir uno nuevo.

Las funciones necesarias serán las de GetById y GetAll, ya que la idea es recibir 2 id's, uno de la empresa activa actual, y otro de la empresa a la que queremos cambiar.

```
1 //IRepositorioCambioEmpresa.cs
2 public interface IRepositorioCambioEmpresa
3 {
4     Task<Empresa> GetById(int id);
5     Task CambioEmpresaActiva(int empresaAnterior, int empresaSiguiete)
6     ;
7     Task<IQueryable<EmpresasActiva>> GetAllEmpresasActivas();
8     public int saveChanges();
9 }
10
11 //ICambioEmpresaActivaServicio.cs
12 public interface ICambioEmpresaActivaServicio
13 {
14     Task<Empresa> GetById(int id);
15     Task CambioEmpresaActiva(int empAnterior, int empSiguiete);
16     Task<IQueryable> GetAllEmpresasActivas();
17 }
```

Haremos un GetById de ambas para verificar la existencia en la base de datos y tras eso, un GetAll de nuestra tabla de Empresas activas, ya que ahí aparece la empresa activa actual, cuando la tengamos, haremos un update de su campo idEmpresa y guardaremos los cambios.

Esto hará que la próxima vez que, como para conseguir la empresa activa se hace referencia a este id concretamente, al hacer un cambio del id estaremos referenciando a otro registro de la tabla de empresas.

Una vez terminado eso, queremos que se nos renueve el token para no tener que reiniciar la sesión, pues simplemente crearemos en nuestro controlador de empresas un EndPoint de cambio de empresa activa en el cual recibamos los datos para la renovación del token y los id's de las empresas, con esos id's llamaremos a los servicios, que se encargarán de conectar con el repositorio y realizar lo que nombramos anteriormente, y una vez hecho eso, hacemos una llamada a nuestra función de generar token pero esta vez con los datos actualizados.

```
1 //EmpresaController.cs
2 public async Task<ActionResult<UsuarioEmpresaDTO>> CambiarEmpresaActiva
   (int empAnterior, int empSiguiente, string nombre, string email,
   string guidUsuario, int userId)
3 {
4     try
5     {
6         string conn = await _empresaServicio.
           GenerarCadenaDeConexionAsync((int)JwtUtil.
           ObtenerDatosEmpresaPeticion(Request.Headers, _configuration,
           _jwtSettings));
7     }
8 }
9 catch (Exception ex)
10 {
11     return StatusCode(401, "Token no valido o inexistente");
12 }
13
14 try
15 {
16     Empresa empresaAnterior = await _cambioEmpresaActivaServicio.
       GetById(empAnterior);
17     Empresa empresaSiguiente = await _cambioEmpresaActivaServicio.
       GetById(empSiguiente);
18
19     if (empresaAnterior is null || empresaSiguiente is null)
20     {
21         return StatusCode(404, "Empresa o empresas no encontradas")
           ;
22     }
23     else
24     {
25         try
```

```
26         {
27             await _cambioEmpresaActivaServicio.CambioEmpresaActiva(
                empresaAnterior.Id, empresaSiguiente.Id);
28
29             JwtSettings settings = new JwtSettings(
30                 keySecret: _configuration.GetSection("JwtSettings")
                    .GetSection("KeySecret").Value,
31                 audience: _configuration.GetSection("JwtSettings").
                    GetSection("Audience").Value,
32                 issuer: _configuration.GetSection("JwtSettings").
                    GetSection("Issuer").Value);
33
34             (string token, DateTime fechaExpiracion) = JwtUtil.
                GenerateJwtEmpresaToken(email, guidUsuario, userId,
                empresaSiguiente.GuidRegistro, empSiguiente,
                settings);
35
36             UsuarioEmpresaDTO userDTO = new UsuarioEmpresaDTO(
                nombre, email, userId, empSiguiente, token,
                fechaExpiracion, guidUsuario, empresaSiguiente.
                GuidRegistro);
37             return userDTO;
38         }
39         catch (Exception ex)
40         {
41             return StatusCode(400, ex.Message);
42         }
43     }
44 }
45 catch (Exception ex)
46 {
47     return StatusCode(400, ex.Message);
48 }
49 }
```

Desde nuestro FrontEnd, solo debemos hacer una llamada al EndPoint pasando los datos, y en el .then guardaremos en el localStorage un token del response.data.

```
1 //ProfileView.vue
2 Confirm() {
3     const token = JSON.parse(localStorage.getItem('token'));
4     const url = 'https://localhost:7030/Empresa/CambiarEmpresaActiva'
5     ;
6     const params = {
7         params: {
8             empAnterior:token.idEmpresa,
9             empSiguiente: this.empresaSelect.id_Empresa,
10            nombre:token.nombre,
11            email:token.email,
12            guidUsuario:token.guid_Usuario,
13            userId:token.userId
14        }
15    }
16
17    axios.put(url)
18    .then((response) => {
19        localStorage.setItem('token', JSON.stringify(response.data));
20        $router.go(0);
21    })
22    .catch((error) => {
23        console.log(error.response);
24    })
25 }
```

De esta forma, tras realizar la petición, en caso de éxito refrescaremos la página.

### 5.3.2 Paginación, filtrado, búsqueda y ordenación

Si bien con un datatable (Vuetify) y con un datagrid (DevExtreme) podemos conseguir esto de forma automática y sin necesidad de preocupaciones, hay ciertos casos en los que necesitamos realizar estos procesos desde el Back-End, pongamos un ejemplo.

En un caso en el cual nuestras consultas vayan a devolver pocos registros de nuestras tablas, o mejor dicho, en el caso en que nuestras tablas contengan pocos registros, nos es totalmente indiferente que hayamos una petición que devuelva 100, 200 o incluso 500 registros, el datatable o el datagrid se encargará de hacer los procesos nombrados sin ningún problema.

Sin embargo, hay casos en los que una consulta no te va a devolver pocos registros, sino que puede llegar a devolver decenas de miles de registros, y trabajar con esos datos reduciría el rendimiento en gran cantidad.

Por eso, si dejamos esta tarea al Back-End podemos no devolver 10000 registros, sino devolver los 20 primeros, posteriormente otros 20 y así constantemente.

Toda la información sobre esto, ha sido sacada de CodeMaze → <https://code-maze.com/net-core-series/> en el apartado “Advanced ASP.NET Core Web API Concepts”.

### 5.3.3 Control de la auditoría

Para llevar un correcto seguimiento sobre lo que sucede con los datos almacenados en nuestras bases de datos, es recomendable hacer uso del control de auditoría, y esto no es más que indicar creación, actualización y borrado de registros, mostrando la fecha en la que se realizó y quién lo realizó (recordemos que no usamos borrado físico de datos, por lo que al borrar un registro, este simplemente no será devuelto en las consultas pero permanecerá en nuestra base de datos).

Para esto será necesario enviar en las peticiones post, put y delete, el id del usuario que realizó la petición, y la fecha simplemente podemos cambiarla desde nuestro Back-End con un `Datetime.Now`.

Sin embargo, es importante destacar que para realizar el control de la auditoría, necesitaremos hacer que nuestros repositorios específicos (que recordemos que eran instancias de un repositorio general al cual le indicábamos un tipo) pasen a convertirse en repositorios independientes, los cuales agruparemos en nuestra unidad de trabajo, esto se debe a que a la hora de trabajar con genéricos, no podemos acceder a las propiedades de una clase que no conocemos, y aunque existe la posibilidad de hacer uso de un repositorio genérico que nos permita realizar control de auditoría, es un proceso más complejo y decidí hacerlo de esta otra forma, cabe recalcar que ambas formas de trabajar son correctas, sin embargo, la forma usada en este caso implica el uso de más código.

```
1 //VehiculoRepository.cs
2 public async Task DeleteById(int id, int idBorrador)
3 {
4     var elem = await _entity.FindAsync(id);
5     if (elem != null)
6     {
7         elem.IdBorrador = idBorrador;
8         elem.FechaBorradoLogico = DateTime.Now;
9         elem.BorradoLogico = true;
10        _entity.Update(elem);
11    }
12 }
```

De esta forma, como `_entity` apunta a la clase vehículo, podemos acceder a las propiedades `idBorrador`, `FechaBorradoLógico` y `BorradoLógico`.

Algo que puede parecer contradictorio, es que realmente estamos actualizando el elemento, sin embargo, tanto los nombres de las funciones, como la petición en nuestro controlador y en la llamada

de axios, van a ser un put, es decir, un update, esto se debe a que lo que nos interesa es que la “intención” sea la de borrar el elemento, por ello nuestro proceso va a ser el de un borrado normal, excepto en el repositorio que como hemos visto, acabaremos haciendo un update.

#### 5.3.4 Mapeo de objetos

A la hora de devolver un objeto, hay muchas propiedades que no nos interesan devolver como por ejemplo la información de auditoría, y para ello, hicimos uso de las DTO's.

Si recibimos 10 datos de nuestra base de datos y los pasamos a DTO, no nos supondrá un problema, sin embargo, si pasamos 1000, la estrategia de un foreach que va accediendo una por una a las propiedades y creando un objeto al que se las asigna, deja de ser útil, por lo que necesitaremos de algo que optimice este proceso.

Aquí es donde AutoMapper entra en juego, el cual se encarga de transformar un objeto (source) a otro objeto (destination), suena a que nosotros estabamos haciendo lo mismo, sin embargo, este paquete NuGet está preparado para realizar todo este proceso de una forma mucho más eficiente.

```
1 //Program.cs
2 builder.Services.AddAutoMapper(typeof(AutoMapperProfile));
```



### 5.3.5 Cómo lleva a cabo el mapeo?

La explicación es bastante simple, para poder llevar a cabo el mapeo, es necesario que en nuestra dto, las propiedades se llamen igual que en nuestro objeto base.

Sin embargo, esto no siempre es posible y automapper viene preparado para esto, ya que tenemos la posibilidad de crear perfiles “Funciones de mapeo” en las cuales definiremos como se llevará el mapeo entre dos clases en específico, es decir, una función de mapeo es el conjunto de reglas que se aplican al realizar un mapeo entre 2 clases en concreto, veamos un ejemplo.

```
1 //FuncionesMapeos.cs
2 public class MapeoVehiculos_VehiculoDtoTypeConverter : ITypeConverter<
   Vehiculo, VehiculoDTO> {
3     public VehiculoDTO Convert(Vehiculo source, VehiculoDTO destination
   , ResolutionContext context) {
4         destination = new();
5
6         if(source is not null) {
7             destination.Id = source.Id;
8             destination.Matricula = source.Matricula;
9             if(source.Marca is not null) {
10                 destination.IdMarca = source.Marca.Id;
11                 destination.Marca = source.Marca.Nombre;
12             }
13
14             if(source.Modelo is not null) {
15                 destination.IdModelo = source.Modelo.Id;
16                 destination.Modelo = source.Modelo.Nombre;
17             }
18
19             if(source.TipoCombustible is not null) {
20                 destination.Id_TipoCombustible = source.TipoCombustible
   .Id;
21                 destination.Combustible = source.TipoCombustible.Nombre
   ;
22             }
23         }
24
25         return destination;
26     }
27 }
```

Estas reglas solo se aplicarán para los mapeos en los que enviemos un vehiculo y querramos obtener un vehiculoDTO, y en cuanto a su funcionamiento, simplemente comprobamos que las propiedades que pueden ser null no lo sean, y establecemos la relación entre ambas clases para que a la hora de hacer el mapeo, no haya ningún error y las propiedades no puedan quedar marcadas como null.

Una vez hecho esto, debemos establecer los perfiles, es decir, cada una de las relaciones que vamos a

llevar a cabo.

```
1 //AutoMapperProfile.cs
2 public class AutoMapperProfile : Profile {
3     public AutoMapperProfile() {
4         #region MapeoVehiculos
5         CreateMap<Vehiculo, VehiculoDTO>().ConvertUsing<
6             MapeoVehiculos_VehiculosDtoTypeConverter>();
7     }
8 }
```

“#Region” no es más que una forma de agrupar código en bloques, si bien en archivos pequeños no es necesario, cuando tienes mucho código es una buena forma de organizarlo para localizar rápidamente aquello a lo que quieres acceder, se pueden contraer y expandir las regiones.

Y como estamos trabajando con inyección de dependencias, en nuestro program.cs debemos poner el servicio de nuestro automapper.

```
1 //Program.cs
2 builder.Services.AddAutoMapper(typeof(AutoMapperProfile));
```

Ahora, en nuestro controlador de vehiculos no va a ser necesario realizar la creación y asignación de valores en la DTO.

```
1 //VehiculoController.cs
2 [HttpGet("getAllVehiculos")]
3 public async Task<ActionResult<List<VehiculoDTO>>> GetAllVehiculos([
4     FromQuery] VehiculoParams parameters, int idEmpresa) {
5     try {
6         string conn = await _empresaServicio.
7             GenerarCadenaDeConexionAsync((int)JwtUtil.
8                 ObtenerDatosEmpresaPetición(Request.Headers, _configuration,
9                     _jwtSettings));
10
11         (List<VehiculoDTO> listadoVehiculosDto, MetadataDto metadataDto
12             ) vehiculos = await _vehiculoServicio.GetVehiculosPaginated(
13             parameters, conn, idEmpresa);
14
15         Response.Headers.Append("X-Pagination", JsonSerializer.
16             Serialize(vehiculos.metadataDto));
17
18         return StatusCode(200, vehiculos.listadoVehiculosDto);
19     } catch (Exception ex) {
20         return StatusCode(401, "Token no valido o inexistente");
21     }
22 }
```

Ahora simplemente haremos la llamada a nuestro servicio, que es donde realizaremos el mapeo de los objetos (no es obligatorio realizarlo en el servicio, puede ser desde el controlador, o incluso desde el repositorio).

```
1 //VehiculoService.cs
2 public async Task<(List<VehiculoDTO> listadoVehiculosDto, MetadataDto
  metadataDto)> GetVehiculosPaginated(VehiculoParams parameters,
  string cadenaConexion, int idEmpresa) {
3     PagedList<Vehiculo> arrayVehiculos = new PagedList<Vehiculo>();
4
5     using (IUnitOfWork repositorioEspecifico = new UnitOfWork(
      cadenaConexion)) {
6         arrayVehiculos = repositorioEspecifico.VehiculoRepositorio.
            GetAllPaginated(parameters, idEmpresa);
7
8     }
9
10    List<VehiculoDTO> vehiculoDTOS = _mapper.Map<List<VehiculoDTO>>>(
        arrayVehiculos);
11    MetadataDto metadataDto = new MetadataDto(arrayVehiculos.TotalCount
        , arrayVehiculos.PageSize, arrayVehiculos.CurrentPage,
        arrayVehiculos.TotalPages, arrayVehiculos.HasNext,
        arrayVehiculos.HasPrevious);
12
13    return (vehiculoDTOS, metadataDto);
14 }
```

Esto es todo lo que necesitamos, y hemos conseguido optimizar nuestras peticiones gracias al uso de automapper.

También funciona en el caso contrario, es decir, nosotros insertamos un objeto vehiculoDTO, y ese objeto lo mapeamos al tipo vehiculo.

### 5.3.6 Inclusión de clases mediante Entity Framework

Si visteis el anterior apartado, os pudo haber generado curiosidad el mapeo de vehiculo a vehiculoDTO, ya que estábamos accediendo a propiedades dentro de las propiedades, como en este ejemplo.

```
1 //FuncionesMapeos.cs
2 destination.Marca = source.Marca.Nombre;
```

Sin embargo, en nuestro modelo de vehiculo no contamos con una propiedad que se llame Marca, es como si tuviésemos una clase dentro de la propia clase de vehiculo.

Esto que habéis visto, no es más que un método que tiene Entity Framework de enviar un objeto que contenga propiedades que no le pertenecen.

Lo explicaremos más adelante, pero para comprender un poco a lo que nos estamos refiriendo, esto nos permitiría enviar a nuestra tabla en el front, objetos vehiculoDTO que contengan tanto el id como el nombre de la marca, modelo y combustible, pero ahorrándonos las llamadas extra a la base de datos.

Para ello, necesitaremos modificar nuestro modelo de vehiculos y hacer lo siguiente.

```
1 //Vehiculo.cs
2 public int? ModeloId { get; set; }
3 public int? MarcaId { get; set; }
4 public int? TipoCombustibleId { get; set; }
5
6
7 public virtual Modelo? Modelo { get; set; }
8 public virtual Marca? Marca { get; set; }
9 public virtual TiposCombustible? TipoCombustible { get; set; }
```

Entre sus propiedades vamos a añadir modelo, marca y combustible, que harán referencia a los modelos que se encuentran en nuestra base de datos.

Y en cuanto a los id's, anteriormente sus nombres eran IdMarca, IdModelo e IdTipoCombustible, sin embargo, la regla es que para que Entity sepa que estamos incluyendo modelos dentro de otro modelo, su clave foránea debe ser Nombre del modelo + identificador, por lo que deberemos cambiarlos.

Cabe recalcar que estableceremos las propiedades como nullables para que solo nos devuelva los modelos cuando lo necesitemos, aunque lo explicaremos en el siguiente ejemplo.

**5.3.6.1 Comprobación** Ahora, para hacer que esto funcione, solo debemos hacer una cosa a la hora de hacer peticiones a nuestra base de datos.

```
1 //VehiculoRepository
2 var query = _vehiculos.Include(e => e.Modelo).Include(e => e.Marca).
    Include(e => e.TipoCombustible).Where(e => e.IdEmpresa == idEmpresa)
    ;
```

¿Simple, no?, solo necesitamos indicar en la petición que queremos incluir modelo, marca y tipocombustible, siendo así que cuando hagamos nuestra petición, nos devolverá el objeto o la lista de objetos con todas las propiedades tanto suyas, como las de los modelos incluidos, esto se resume en que es un join que nos permite almacenar en nuestro objeto propiedades pertenecientes a otros modelos, y si lo juntamos con el mapeo del punto anterior, no tendremos dificultad alguna para trabajar con un objeto que tenga tantas propiedades, simplemente nuestra función de mapeo nos lo hace automáticamente sin que tengamos que preocuparnos.

### 5.3.7 Más funcionalidades de los mapeos

Una vez tenemos implementados nuestros mapeos, podremos usarlos para cualquier acción, ya sea la inserción, actualización y obtención de elementos.

Como nuestro controlador siempre va a trabajar con DTO's, eso significa que los mapeos nos serían útiles para transformar esas DTO's a los modelos almacenados, por lo que necesitamos poder mapear de modelos a DTO's y de DTO's a modelos.

```
1 //AutoMapperProfile.cs
2 #region MapeoVehiculos
3 CreateMap<Vehiculo, VehiculoDTO>().ConvertUsing<
    MapeoVehiculos_VehiculosDtoTypeConverter>();
4 //Mapeo inverso
5 CreateMap<VehiculoDTO, Vehiculo>().ConvertUsing<
    MapeoVehiculosDTO_VehiculosTypeConverter>();
6 #endregion
```

Simplemente implementamos el mapeo inverso, de tal forma que la función creada para mapear de modelo a DTO, la volveremos a hacer pero al revés.

Esto no se aplica a todos los casos, ya que como dijimos anteriormente, si las propiedades de nuestro modelo y DTO coinciden en nombre, no es necesario implementar función de mapeo, por lo que nuestro código se reduciría a esto.

```
1 // AutoMapperProfile.cs
2 #region MapeoMarcas
3 CreateMap<Marca, MarcaDTO>().ReverseMap();
4 #endregion
5
6 #region MapeoModelos
7 CreateMap<Modelo, ModeloDTO>().ReverseMap();
8 #endregion
9
10 #region MapeoEmpresas
11 CreateMap<Empresa, EmpresaDTO>().ReverseMap();
12 #endregion
```

En este caso, como los nombres de las propiedades coinciden, para nuestras marcas, modelos y empresas, no necesitaremos crear función de mapeo.

### 5.3.8 Uso de secuencias para la generación de los id's

Como las tablas que usaremos no cuentan con id autoincremental y no es muy buena idea ir poniendo id's al azar al insertar valores, haremos uso de una secuencia.

```
1 --SQLServer Query
2 CREATE SEQUENCE [dbo].[GENERACIONID]
3 AS [int]
4 START WITH 100
5 INCREMENT BY 1
6 MINVALUE -2147483648
7 MAXVALUE 2147483647
8 CACHE 250
```

En nuestro caso, estamos haciendo uso de SQLServer, por lo que desde el Management Studio, accederemos a la base de datos en la que queramos generar la secuencia (En nuestro caso es en todas) y simplemente ejecutamos la consulta, por lo que una vez hecho esto, podremos acceder a ella desde la ventana de consultas.

Sin embargo, a nosotros no nos interesa el acceder a la secuencia desde el Management Studio, sino que buscamos generar nuestro id en tiempo de ejecución al realizar el insert.

En este punto, desconozco si desde Entity Framework se puede acceder a las secuencias almacenadas en nuestra base de datos (Doy por hecho que si ya que aparece reflejado en nuestro contexto), sin embargo, lo haremos de forma manual haciendo uso de SQLClient y SQLCommands.

```
1 //VehiculoRepository.cs
2 public int NextIdSequence(string conn) {
3     int nextId = 0;
4     using (SqlConnection connection = new SqlConnection(conn)) {
5         connection.Open();
6         using (SqlCommand cmd = new SqlCommand("SELECT NEXT VALUE FOR
7             GENERACIONID", connection)) {
8             nextId = (int)cmd.ExecuteScalar();
9         }
10        connection.Close();
11    }
12    return nextId;
13 }
```

Necesitaremos una cadena de conexión para acceder a la base de datos, por lo que si recordamos, anteriormente implementamos la forma de realizar consultas a una base de datos u otra dependiendo de nuestra empresa activa, es decir, ya contamos con la cadena de conexión necesaria.

El funcionamiento es simple, tan solo abrimos una conexión, creamos el comando de consulta y lo ejecutamos, para guardar el valor en una variable y devolverla.

```
1 //VehiculoRepository.cs
2 public async Task Insert(Vehiculo entity, int idCreador, string conn) {
3     entity.IdCreador = idCreador;
4     entity.FechaCreacion = DateTime.Now;
5     entity.GuidRegistro = Guid.NewGuid().ToString().ToUpper();
6     entity.Id = NextIdSequence(conn);
7     _vehiculos.Add(entity);
8 }
```

Y posteriormente, desde nuestra función de insert, llamaremos a esta otra función pasando la cadena.

## 5.4 Funciones extra

### 5.4.1 Selector de tema

Para implementar el apartado de preferencias de color, vamos a necesitar hacer uso de las variables en nuestro css, lo que nos permite asignar un valor a cada variable, estos valores serán en nuestro caso colores.

Por otro lado, necesitaremos un archivo donde almacenar los diferentes temas con los colores que contienen cada uno.

Y finalmente, implementar el cambio mediante nuestro código, comencemos enseñando el fichero en el que guardaremos los diferentes temas.

```
1 //themes.ts
2 const predeterminado = {
3   primary: "#106c4c",
4   secondary: "#ccc",
5   accent: "#000",
6   error: "#ff0000",
7 }
8 const morado = {
9   primary: "#B03BFF",
10  secondary: "#000",
11  accent: "#000",
12  error: "#FF8B81",
13 }
14 const naranja = {
15   primary: "#f59120",
16   secondary: "#000",
17   accent: "#000",
18   error: "#ff613d",
19 }
20
21
22 export default {
23   predeterminado: predeterminado,
24   morado: morado,
25   naranja: naranja
26 }
```

Tenemos un objeto por cada tema con cada uno de los colores que utilizaremos posteriormente.



```
1 //App.vue
2 <script lang="ts">
3 import Vue from 'vue';
4 import themes from "@/assets/themes"
5 type typesThemes = 'predeterminado' | 'morado' | 'naranja'
6
7 export default Vue.extend({
8   name: 'App',
9   computed:{
10     cssProps(){
11       const current = themes[localStorage.getItem("tema") as
12         typesThemes]
13       return{
14         '--primary-color' : current ? current.primary : themes['
15           predeterminado'].primary,
16         '--secondary-color' : current ? current.secondary : themes['
17           predeterminado'].secondary,
18         '--accent' : current ? current.accent : themes['predeterminado'
19           ].accent,
20         '--error' : current ? current.error : themes['predeterminado'].
21           error,
22       }
23     }
24   },
25   data: () => ({ }),
26 });
27 </script>
```

En nuestro App.vue importaremos el fichero .ts con nuestro temas, de esta forma podremos llamarlos y acceder a sus propiedades.

El truco está en usar cssProps, lo que nos permitirá declarar variables globales (justo lo que queremos).

Primero seleccionamos el tema guardado en nuestro localStorage, y una vez hecho eso definimos una variable y le asignamos el valor de la propiedad deseada en base a nuestro tema almacenado, si no tenemos ningun tema almacenado en el localStorage, simplemente usamos el tema por defecto.

Si hemos hecho esto, ahora solo quedaría escribir nuestro css de esta forma.

```
1 /*styles.css*/
2 .botonesModal button {
3   height: 3rem;
4   color: white;
5   font-weight: 500;
6   background-color: var(--primary-color);
7   border-right: 1px solid white;
8   border-left: 1px solid white;
9   width: 50%;
10 }
```

Vayamos un poco más lejos, si partimos de la idea de que tenemos una aplicación en la que el código css es normal que se repita, como a la hora de darle estilo a botones, tablas, dialogs...

Repetir el mismo código css por cada una de nuestras vistas no es una práctica muy eficiente por nuestra parte, aquí es donde entra en juego una simple hoja de estilos independiente.

Podemos crear en nuestra carpeta de assets un fichero css y ahí escribir los estilos que se van a repetirse en nuestra vistas, luego simplemente importamos el fichero en nuestro main.ts y somos libres de asignar esas clases a nuestras etiquetas sin importar en que fichero se encuentren, al tenerlo definido e importado, estaríamos reduciendo el código repetido en cada una de nuestras vistas o componentes.

```
1 //Main.ts
2 import './assets/estilos.css';
```

### 5.4.2 Gestión de errores

Si bien estamos haciendo uso de try catch en nuestros controladores, hecho que nos permite atrapar los errores que surjan y controlarlos a nuestra voluntad, no es una buena práctica llenar nuestros controladores con código de gestión de errores, para esto solemos hacer uso de un middleware, el cual hará (como su nombre indica) de capa intermedia entre las peticiones y podremos establecer la lógica del manejo de errores ahí, veamos que es lo que necesitamos para implementarlo.

**5.4.2.1 Logger** Logger nos permitirá escribir mensajes en consola o guardar dichos registros en ficheros de registro (logs) para poder llevar seguimiento de fallos y en general, del funcionamiento de nuestra api.

```
1 //ILoggerManager.cs
2 public interface ILoggerManager {
3     void LogInfo(string message);
4     void LogWarn(string message);
5     void LogDebug(string message);
6     void LogError(string message);
7 }
```

Comenzaremos con la interfaz de nuestro Logger, la cual simplemente contará con una función por cada mensaje de log que vamos a enviar, en este caso será info, debug, warning y error.

```
1 //LoggerManager.cs
2 public class LoggerManager : ILoggerManager {
3     private static ILogger logger = LogManager.GetCurrentClassLogger();
4
5     public LoggerManager() {
6         LogManager.Setup().LoadConfigurationFromFile(String.Concat(
7             Directory.GetCurrentDirectory(), "/nlog.config"));
8     }
9
10    public void LogInfo(string message) {
11        logger.Info(message);
12    }
13
14    public void LogWarn(string message) {
15        logger.Warn(message);
16    }
17
18    public void LogDebug(string message) {
19        logger.Debug(message);
20    }
21
22    public void LogError(string message) {
23        logger.Error(message);
24    }
25 }
```

Seguimos con nuestra clase que implementará dicha interfaz, y en su constructor debemos indicarle el archivo de donde obtendrá la configuración, el cual debemos crear.

```
1 <!-- nlog.config -->
2 <nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd" xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-instance">
3   <targets>
4     <target name="console" xsi:type="ColoredConsole" layout="{
      longdate} [{whenEmpty:whenEmpty=${threadid}:inner=${threadname}}] $
      {level} {logger} {message} {exception:format=tostring}">
5       <highlight-row condition="level == LogLevel.Error"
        foregroundColor="Red" />
6       <highlight-row condition="level == LogLevel.Warn"
        foregroundColor="Yellow" />
7       <highlight-row condition="level == LogLevel.Debug"
        foregroundColor="Blue" />
8       <highlight-row condition="level == LogLevel.Info"
        foregroundColor="White" />
9     </target>
10    <target xsi:type="File" name="file" layout="{longdate} {level}
      {logger} {message} {exception:format=tostring}"
      fileName="{basedir}/logfile.log" keepFileOpen="false"
      encoding="iso-8859-2" />
11  </targets>
12  <rules>
13    <logger name="*" minlevel="Info" writeTo="console" />
14  </rules>
15 </nlog>
```

Tras esto, solo queda implementar nuestro logger en el program.cs.

```
1 //Program.cs
2 var logger = app.Services.GetRequiredService<ILoggerManager>();
```

Cabe recalca que para usar logger, debemos instalar la el paquete nuGet NLog.Web.AspNetCore.

Una vez tenemos nuestro logger configurado e implementado, podemos comenzar con el middleware (tutorial obtenido de CodeMaze → <https://code-maze.com/global-error-handling-aspnetcore/>).

Dado que la explicación aparece en la propia página y en la guía proporcionada por el chico (yo simplemente he seguido los pasos) dejaré el código a continuación.

Dentro de nuestro directorio de modelos, crearemos una clase llamada ErrorDetails.

```
1 //ErrorDetails.cs
2 public class ErrorDetails
3 {
4     public int StatusCode { get; set; }
5     public string Message { get; set; }
6     public override string ToString() {
7         return JsonSerializer.Serialize(this);
8     }
9 }
```

Crearemos un directorio llamado CustomExceptionMiddleware y dentro, un fichero llamado ExceptionMiddleware.

```
1 //ExceptionMiddleware.cs
2 public class ExceptionMiddleware {
3     private readonly RequestDelegate _next;
4     private readonly ILoggerManager _logger;
5
6     public ExceptionMiddleware(RequestDelegate next, ILoggerManager
7         logger) {
8         _logger = logger;
9         _next = next;
10    }
11
12    public async Task InvokeAsync(HttpContext httpContext) {
13        try {
14            await _next(httpContext);
15        } catch (AccessViolationException avEx) {
16            _logger.LogError($"A new violation exception has been
17                thrown: {avEx}");
18            await HandleExceptionAsync(httpContext, avEx);
19        } catch (Exception ex) {
20            _logger.LogError($"Something went wrong: {ex}");
21            await HandleExceptionAsync(httpContext, ex);
22        }
23    }
24
25    private async Task HandleExceptionAsync(HttpContext context,
26        Exception exception) {
27        context.Response.ContentType = "application/json";
28        context.Response.StatusCode = (int)HttpStatusCode.
29            InternalServerError;
30
31        var message = exception switch {
32            AccessViolationException => "Access violation error from
33                the custom middleware",
34            _ => "Internal Server Error from the custom middleware."
35        };
36
37        await context.Response.WriteAsync(new ErrorDetails() {
38            StatusCode = context.Response.StatusCode,
39            Message = message
40        }.ToString());
41    }
42 }
```

Tras esto, nuevamente crearemos un directorio llamado Extensions y dentro, un fichero llamado ExceptionMiddlewareExtensions.

```
1 //ExceptionMiddlewareExtensions
2 public static class ExceptionMiddlewareExtensions {
3     public static void ConfigureCustomExceptionMiddleware(this
4         WebApplication app) {
5         app.UseMiddleware<ExceptionMiddleware>();
6     }
7 }
```

Para finalmente volver a nuestro Program.cs y añadir la siguiente línea.

```
1 //Program.cs
2 app.ConfigureCustomExceptionMiddleware();
```

Si hemos seguido todos los pasos del tutorial, podremos eliminar nuestro bloques de try catch de los controladores ya que el manejo de errores será llevado a cabo de forma automática por nuestro middleware.

Y por supuesto, podemos hacer uso de logger para mostrar por consola la información que se requiera, mostraré un ejemplo.

```
1 //VehiculoController.cs
2 [HttpGet("getVehiculo")]
3 public async Task<ActionResult<VehiculoDTO>> GetVehicle(int id) {
4     _logger.LogInfo("Fetching vehicle with ID: " + id);
5
6     string conn = await _empresaServicio.GenerarCadenaDeConexionAsync((
7         int)JwtUtil.ObtenerDatosEmpresaPeticion(Request.Headers,
8         _configuration, _jwtSettings));
9     VehiculoDTO vehiculo = await _vehiculoServicio.GetById(id, conn);
10
11     if (vehiculo == null) {
12         _logger.LogWarn("Vehicle with ID: " + id + " not found.");
13         return NotFound("No se encuentra el vehículo buscado");
14     } else {
15         _logger.LogInfo($"Returning vehicle with ID: {id}");
16         return Ok(vehiculo);
17     }
18 }
```

Como se ha mostrado, nos olvidamos de la gestión de errores, simplemente devolvemos nuestro statusCode con el contenido que se requiera.

## 6 Conclusiones

A pesar de haber sido un proyecto con dificultad reducida, el estar ante un lenguaje de programación y herramientas desconocidas ha causado que desde un principio haya tenido muchas complicaciones, sin embargo, a medida que iba avanzando y el grado de dificultad aumentaba, lo que en un principio me resultaba complicado lo comprendía y entendía por que x cosas se realizaban de x manera, todo esto hablando de la parte del BackEnd ya que en cuanto al Front, a pesar de no ser mi punto fuerte, con ver ejemplos de alguien haciendo lo que tu quieres llevar a cabo, puedes conseguirlo de forma más simple que pelearte con el código en el Back para comprender por que algo da error o simplemente por que no funciona.

En general no ha sido algo simple, en el front he podido descubrir la existencia y el funcionamiento de vuetify, el cual facilita mucho el diseño de las páginas, así como Axios y el la posibilidad de crear un interceptor para mantener todas las peticiones bien controladas.

En el apartado del BackEnd podría no terminar de hablar, a pesar de ser un lenguaje bastante parecido a java, el cual aprendimos hace un año con Rafael, puede complicarse tanto como tu te lo propongas.

La idea de un crud de vehículos es algo bastante simple ya hemos hecho cruds en clase bastantes veces, sin embargo, es la primera vez que realizo algo así en este lenguaje, aprendiendo un patrón de trabajo nuevo, con un medio de conexión con bases de datos que nunca había visto, hemos implementado la posibilidad de hacer consultas a distintas bases de datos, una autenticación por token, mapeo de objetos, gestión de errores mediante middleware... todo esto ha ido sumando dificultad y ha hecho que finalmente quede satisfecho con el proyecto.

## 7 Referencias Web

A continuación, la documentación y páginas consultadas para realizar este proyecto, fueron:

- Back-End
  - Curso Realizado
    - \* Curso sobre .NET → [https://www.campusmvp.es/catalogo/Product-Desarrollo-con-la-plataforma-.NET-y-C\\_242.aspx](https://www.campusmvp.es/catalogo/Product-Desarrollo-con-la-plataforma-.NET-y-C_242.aspx)
  - Guías básicas
    - \* Paginación → <https://code-maze.com/paging-aspnet-core-webapi/>
    - \* Filtrado → <https://code-maze.com/filtering-aspnet-core-webapi/>
    - \* Búsqueda → <https://code-maze.com/searching-aspnet-core-webapi/>
    - \* Ordenación → <https://code-maze.com/sorting-aspnet-core-webapi/>
    - \* Middleware para gestión de errores → <https://code-maze.com/global-error-handling-aspnetcore/>
    - \* Microsoft → <https://learn.microsoft.com/en-us/aspnet/tutorials>
    - \* AutoMapper → <https://docs.automapper.org/en/latest/Getting-started.html>
- Front-End
  - Guías básicas
    - \* Vuetify → <https://vuetifyjs.com/en/>
    - \* Axios → <https://axios-http.com/es/docs/intro>
- Conjunto
  - StackOverflow → <https://stackoverflow.com/>