MAKINO, AN INTUITIVE CAR MARKETPLACE FOR THE ALBANIAN MARKET AND A DIARY COMPANION

By

Mateo Sheme

Muhamedali Kura

A THESIS

Submitted to

KOLEGJI UNIVERSITAR "Instituti Kanadez i Teknologjisë"

University College "CANADIAN INSTITUTE OF TECHNOLOGY" Faculty of Engineering

Department of Software Engineering

In partial fulfillment of the requirements for the degree of :

Bachelor of Science in Software Engineering

Submitted on 06.13.2025

Bachelor of Science in Engineering, Profile Software

Copyright

Submitted on [06 13, 2025]

Approved:

_____ [Muhamedali Kura]

(signature)

_____ [Irena Fata]

(signature)

I understand that my thesis will become part of the collection of Canadian Institute of Technology. My signature below authorizes release of my thesis to any reader upon request. I also affirm that the work represented in this thesis is my own work.

_____ Mateo Sheme, Author

(signature)

Abstract

Title of thesis: Makino, an intuitive car marketplace for the Albanian market and a diary companion

Here in Albania, thousands of cars are being imported each day: electrical, diesel ones and many of these sellers are looking for buyers, but nowhere to advertise. Besides that remembering your maintenance of the car is a hassle so why not make a solution for both of them.? The rationale behind this project is to offer a modern and accessible solution that simplifies the vehicle selling process while also helping users track vital information about their cars. The application not only serves as a marketplace but also includes a diary for maintenance records.

To achieve this, a full-stack web application was developed using .NET for the backend, React for the frontend, and SQL Server for data storage. The system is hosted on a cloud server and secured behind Cloudflare, ensuring availability and protection. The result is a mobile friendly web application where users can create listings, browse vehicles and maintenance records with a database, backend and front and deployed so that it is scalable and available at all times.

In conclusion, this project demonstrates how modern web technologies can be used to address real-world problems in the automotive sector, providing a robust platform for both buyers and sellers while improving long-term vehicle ownership experiences.

Keywords: Car marketplace, maintenance records, full-stack web app, ASP.NET, React, SQL Server, Cloudflare, vehicle maintenance

Acknowledgment

Professor MSC.Muhamedali Kura, for being my mentor, advisor as well as helping me during all the stages of this Thesis!

Professor Ph.D.(c)Irena Fata, for always helping me and being always cheerful!

MSC.Mariglen Pupa, my one and only brov for providing me with advice, insights and course materials regarding the project, as well as criticizing me when needed!

Çini-2009 shpk, for allowing me to conduct my internship there as well as letting me access their server for deployment as well as their CloudFlare domain name!

Instructor Bhrugen Patel, who through his course ASP.NET Core MVC [.NET 8] - The Complete Guide at learning.oreilly.com guided me a lot as his explanation was top notch!

Table of contents

Copyright 2
Abstract 3
Acknowledgment 4
Table of contents 5
List of Figures and Tables 6
List of Appendices 8
Chapter I: INTRODUCTION 9
Chapter II: LITERATURE REVIEW 10
Chapter III: MVC ARCHITECTURE 13
Chapter IV: FRONTEND 16
4.1: CREATING AND SETTING UP THE PROJECT 17
4.2: REACT FOLDER STRUCTURE 19
4.3: THE WEBPAGE FUNCTIONALITIES 21
4.4: THE CONTEXT FILES 28
4.5: LOGIN, REGISTER AND MOBILE RESPONSIVES 31
Chapter V: BACKEND 33
5.1: APPLICATION DB CONTEXT 34
5.2: CAR CONTROLLER 36
5.3: AUTH CONTROLLER 41
5.4: DIARY CONTROLLER 43
Chapter VI: DATABASE ENGINE 49
6.1: DATABASE SCHEMATIC 50
6.2: ENTITY FRAMEWORK 53
Chapter VII: DEPLOYMENT 57
Conclusions 60
Bibliography 61

List of Figures and Tables

Figure 2.1……..…10
Figure 2.2……..…10
Figure 2.3……..…11
Figure 2.4……..…11
Figure 2.5……..…12
Figure 3.1……...14
Figure 3.2……...15
Figure 3.3……..…15
Figure 4.1.1……....17
Figure 4.1.2……....17
Figure 4.1.3……....18
Figure 4.1.4……....18
Figure 4.2.1……....19
Figure 4.2.2……....19
Figure 4.2.3……....20
Figure 4.3.1……....21
Figure 4.3.2……....21
Figure 4.3.3……....22
Figure 4.3.4……....22
Figure 4.3.5……22
Figure 4.3.6……....23
Figure 4.3.7……....23
Figure 4.3.8……....24
Figure 4.3.9……....24
Figure 4.3.10……..25
Figure 4.3.11……..…25
Figure 4.3.12……..26
Figure 4.3.13……..26
Figure 4.3.14……..27
Figure 4.4.1……28
Figure 4.4.2……28
Figure 4.4.3……29
Figure 4.4.4……29
Figure 4.4.5……30
Figure 4.4.6……30
Figure 4.5.1……31
Figure 4.5.2……31
Figure 4.5.3……32
Figure 4.5.4……32
Figure 5.0.1……33
Figure 5.0.2……33
Figure 5.1.1……34
Figure 5.2.1……36

Figure 5.2.2……37
Figure 5.2.3……38
Figure 5.2.4……38
Figure 5.2.5……39
Figure 5.3.1……41
Figure 5.3.2……42
Figure 5.3.3……42
Figure 5.4.1……43
Figure 5.4.2……43
Figure 5.4.3……44
Figure 5.4.4……44
Figure 5.4.5……45
Figure 5.4.6……46
Figure 5.4.7……47
Figure 5.4.8……47
Figure 5.4.9……47
Figure 5.4.10……...…48
Figure 5.4.11……...…48
Figure 6.0.1……49
Figure 6.1.1……50
Figure 6.2.1……53
Figure 6.2.2……53
Figure 6.2.3……54
Figure 6.2.4……54
Figure 6.2.5……55
Figure 6.2.6……55
Figure 6.2.7……56
Figure 6.2.8……56
Figure 6.2.9……56
Figure 7.0.1……58
Figure 7.0.2……59
Figure 7.0.3……59

List of Appendices

Appendix A =

```
function Welcome() {
        return
                <h1>Hello, World!
        </h1>;
}
```

Appendix B =

```
return (
  <>

    <h1>Title</h1>

    <p>Description</p>

  </>

);
```

Appendix C =

```
<Card>

  <Title text="Welcome" />

  <Content />

</Card>

var builder = WebApplication.CreateBuilder(args);var app =builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Chapter I: INTRODUCTION

When asked, “When and where did you buy your current car?” Most drivers in Albania can provide an immediate answer. Typically, young and middle-aged men in Albania group together and travel abroad, primarily to Germany, to purchase used vehicles. They often browse platforms such as auto.de and autoscout24.com, negotiate prices in person, and are usually satisfied with their final choice after inspecting and test-driving the vehicle.

On the other hand, exclusive sellers such as car dealerships or importers typically advertise only on social platforms like Instagram or through local listing sites like merrjep.al. The latter, while intended for both individual and business use, is bloated with mixed listings and lacks the structure of a true car marketplace.

As of May 14, 2025, INSTAT reports that 976,946 cars circulate in Albania (INSTAT, 2025a). With a population of 2,773,350 (Worldometer, 2025), this translates to approximately 352 cars per 1,000 people, or roughly 0.3 cars per person. This data reflects a growing demand for vehicles in the country. For comparison, in 2024 there were 887,321 vehicles on the road, indicating an increase of nearly 90,000 cars within a year (INSTAT, 2025b).

To sum up a number of 0.3 cars per person, an increasing demand and more and more imported cars each year. A car marketplace specifically designed for this case is a much needed help for selling as well as making it easier for everyone to pinpoint his or her car according to her needs. Of course the Albanian mentality goes according to known friends or relatives who deal with selling or trading cars as those are the ones that are trustworthy. By having this marketplace you meet face-to-face with your seller and can test the car as much as needed and negotiate as the marketplace only acts as a bridge connecting seller to buyer.

Chapter II: LITERATURE REVIEW

Doing a simple google search "car marketplace albania" these results are shown at Figure 2.1. All marketplaces doing what they are supposed to do. Acting as sole bridges of connection between sellers and buyers. However they look really barebone and something you would use once and leave or even forget about your listing all together.

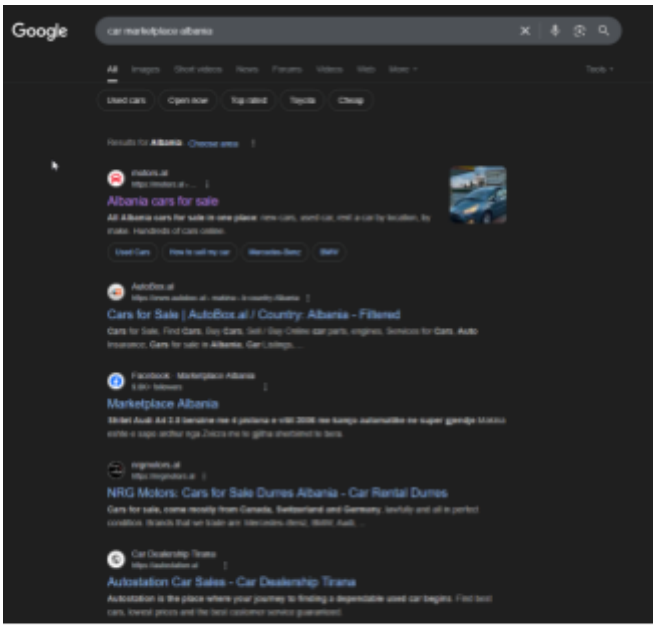


Figure 2.1
Google search for car marketplaces in albania

Not even something that can stand out from the crowd, something that you are able to remember even after a while, very blended.

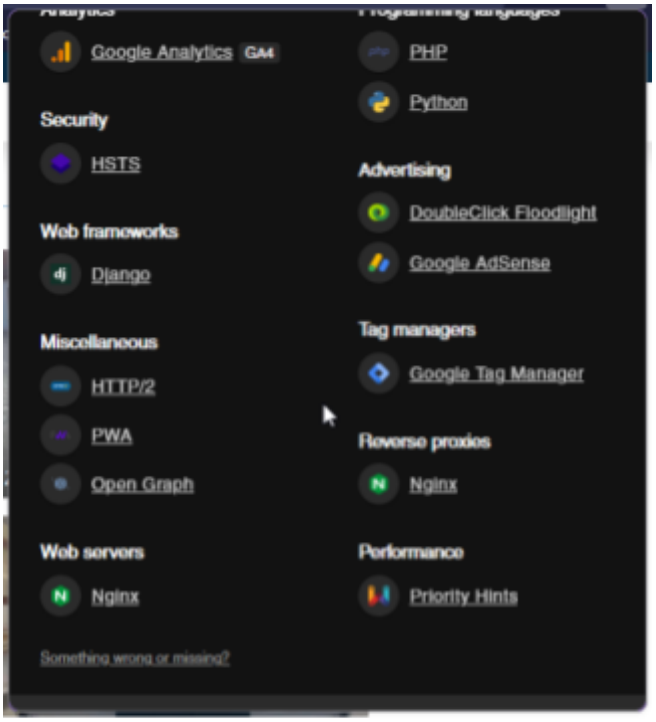


Figure 2.2
The software stack of one of the marketplaces
Source : google.com

Looking also at the technology stack at Figure 2.2 used in one of the motors.al. I am guessing php for the backend, django for the frontend. It is good however for scalability don't think so much as usually tech stacks like to work with their own derivatives better. Besides that not a lot of space to work for future-proof, if you need to add extra features.

So after this search I have come to conclusion that the marketplace needs to be :

1. Design attractive
2. Solid backend and frontend framework
3. A feature to stand out

As for design this is the logo (Figure 2.3) that I designed myself. Coming in two versions for dark theme and for white theme content (Figure 2.4), so this makes it flexible to be used everywhere if needed for posters or fairs in the future.

Figure 2.3
Dark theme logo

Figure 2.4
Light theme logo

It is composed of Makinë, the Albanian word for car and Go, remove the ë and G and Makino is born. Really catch and attractive name.

I also made sure that the logo wasn't the same car dealership logo which has a black background and an outline of the car. It can also be noticed that the o is above the right tire which intensifies the Go more.

For the backend I chose ASP.NET framework, as I saw its capabilities and difficulties and is one of the solutions that can fulfill the project's needs for future development. For the front between React and Angular, I chose React as more flexible and easier to learn.

Finally the feature that I think makes it stand out from the other marketplaces and also is a solution to a nonexistent problem that all drivers face but no one has managed to pin point yet. A car diary, where you are able to see all your owned cars and each maintenance record of oil change, engine check, car check up, insurance expiration. All of the things you have to remember or scribble somewhere on a sticky note and attach it to the car then it gets lost. Writing it on your phone and forgetting where you put it or else not having a solid structure.

I believe that these decisions will help make Makino stand out from the rest of the

marketplaces, of course a lot of marketing is needed as well as Search Engine Optimizations but it's possible. As for income the marketplace will work with advertising banners (Figure 2.5) for now, later the users will be able to pay to have their listings at the top. This way of income is the only way as the system isn't able and it is more costly to host all the chats and calls between the seller and buyer, besides that inconvenient. So you can't monitor that but you can monitor where people land clicking at your page which is the home page and where it will be a big banner with advertising.

Figure 2.5
Banner of advertising used on the home page

Chapter III: MVC ARCHITECTURE

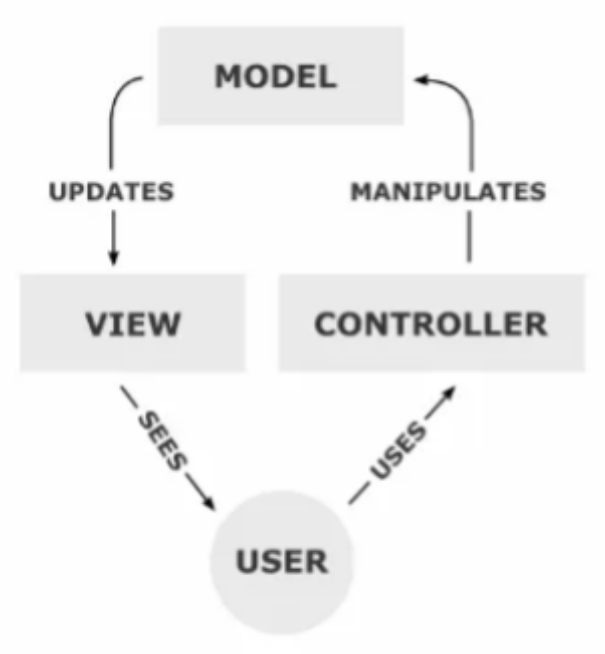
So in order to fulfill the need of a marketplace and diary of cars accessible by everyone, a full-stack application is needed. As it needs to be scalable and future proof of expanding. The MVC methodology achieves that. The framework that I picked was ASP.NET Framework.

The Model-View-Controller pattern divides an application into three distinct layers:

- Model : Represents the data and business logic. It is responsible for interacting with the database, performing calculations, and managing the state of the application.
- View : Represents the user interface. It is responsible for rendering data to the user and presenting input forms.
- Controller : Acts as the intermediary between the View and the Model. It receives user input, processes it, and calls the appropriate functions in the Model. It then selects the appropriate view to display the result.

At Figure 3.1 there is a schematic on how it works. So to translate them into programmer language. The model is the database connection in ASP.NET. There is a Entity Relationship Framework which is inside the Microsoft.EntityFrameworkCore Nuget package as seen on Figure 3.2. I will name them entities from now on. So first are the entities which are equal to the tables of the database and the relations between each other. After the model is created it updates the view in our case the front-end. The data is loaded from the entity to the front-end and the user is able to see the front-end filled with its according data. Then the user is able to do requests from the front to the back by using the controller. The controller is the back-end and where all the logic happens. The controller is the one that triggers different actions to be done by the entity or database. Finally this all loops together back to the model. The MVC architecture is the backbone for many modern apps as it offers quite a few features for making development, maintenance, and scaling easier.

- Separation of Concerns: Each layer of the application handles a distinct responsibility, making the codebase cleaner and more manageable.
- Reusable : many of its components are cohesive as each component is tied with each-other but with one responsibility each. It can also be said that they are loose coupled, meaning that changing one component doesn't break the other components.
- Scalable : the applications can start as simple applications but overtime more features are required, more integrations and ASP.NET is able to handle that with ease.
- Testability : the models(entities) can be tested without UI or on mock Views.



Figaure 3.1
MVC architecture

As mentioned for the backend I chose ASP.NET Framework (Figure 3.3), with React JS (Figure 3.3) for the framework and Microsoft SQL Server (Figure 3.3) for the database. The reason why I chose this stack, is first I relied on the backend and followed a course last summer while I was preparing for the September make-up exams and it was a breeze about it. I wanted to get a rough idea on the amount of work and knowledge needed leading to this Diploma and I can say it served me really well.

It was a breeze and really easy to get my head around. Pair that with Visual Studio 2022 IDE editor and it was a must to use Microsoft SQL Server because of its convenience. Furthermore, I have been using and tinkering with Microsoft SQL Server since high-school. As that was the database engine we were taught and during the Fundamentals of Programming and Computer Applications I used it for the two final

projects there. Finally the React frontend was chosen mostly to learn something new and it is considered as an easy frontend framework to get started.

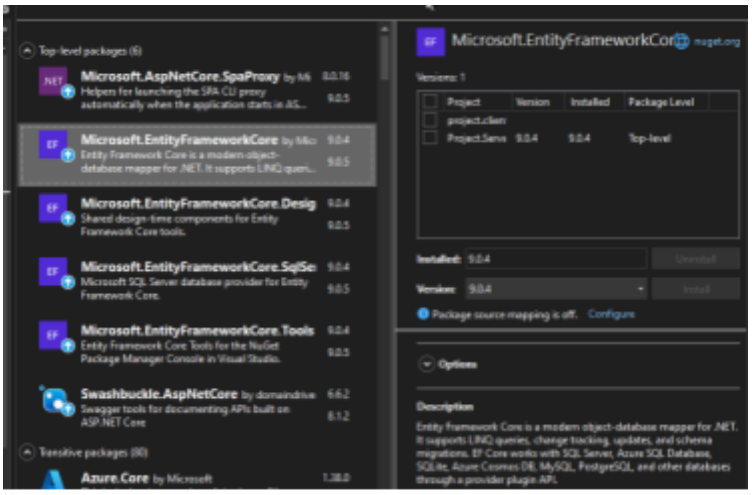


Figure 3.2
Package of Entity Framework Core from NuGet

Figure 3.3
The logos of each respective frontend, backend and database

Chapter IV: FRONTEND

React is a JavaScript library designed to simplify the creation of user interfaces, both for web and native applications. Unlike frameworks, a library like React offers a set of tools and functions you can selectively use in your code to boost development, especially for building dynamic and responsive interfaces.

JavaScript itself adds interactivity to web pages by handling user events and modifying the page content without reloading it. React builds on this by providing a structured way to manage these interactions efficiently. It reduces complexity and helps developers write more maintainable and less error-prone code (Schwarzmüller, 2023).

JSX (JavaScript XML) is a syntax extension for JavaScript" (Smith, 2018, p. 23). used in React, a popular library for building user interfaces. It allows developers to write HTML-like code within JavaScript, which React transforms into React.createElement() calls behind the scenes. JSX makes code more readable, expressive, and easier to visualize, especially when building component-based UIs.JSX is not required to use React, but it's widely adopted because it simplifies writing UI components. Rather than separating markup (HTML) and logic (JavaScript), JSX allows both to coexist in the same file, promoting component cohesion and readability. (see Appendix A). This is transformed by a compiler into: React.createElement('h1', null, 'Hello, World!'); JSX looks like HTML, but it's not. It's syntactic sugar for React.createElement, and each JSX element is compiled into a JavaScript object that React uses to build the Virtual DOM. JSX elements must be wrapped in a single parent. This can be done using a div, or a special empty tag called Fragment <> JSX supports all standard HTML attributes like src, alt, disabled, etc., but written in camelCase for custom or DOM properties. (see Appendix B)

You can also nest JSX elements as children (see Appendix C)

4.1: CREATING AND SETTING UP THE PROJECT

Visual Studio IDE already has templates ready for creating different types of projects (Figure 4.2.1), ready baked and to go. Saving the hassle of running commands back and forth to have everything work as needed.

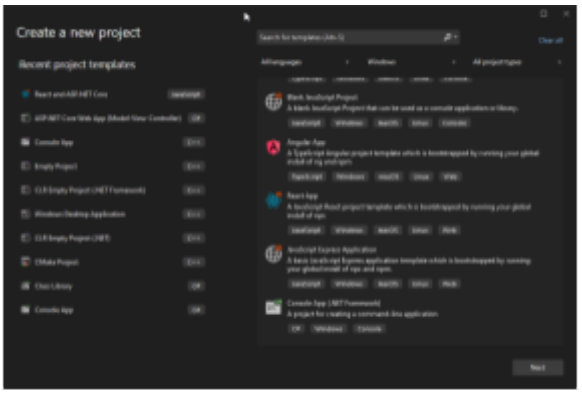


Figure 4.1.1
Creating a new project wizard from the Visual Studio 2022

An example of creating and running the project is as follows on the terminal :

- winget install OpenJS.NodeJS.LTS
- node -v
- npm -v
- npm install -g create-react-app
- npx create-react-app my-react-app
- cd home/myproject
- npm start

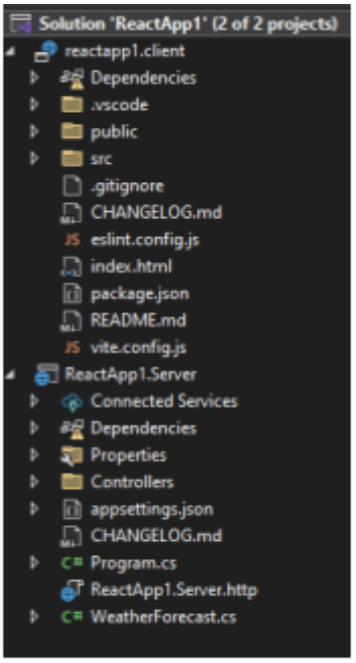


Figure 4.1.2
Template of React and ASP.NET

I still need to install NodeJS beforehand as that is the engine which has npm bundled in. That is used for installing the create react app utility which has the npx and that is the final necessary component to create the project. After that, running npm start would publish the project to http://localhost:port. After creating the project from the Visual Studio template it ends up with the setup on the right (Figure 4.1.2). Where two distinct components can be seen projectname.client and projectname.Server. Judging by the name the client is the front and the react app structure and the server is the backend logic which is .NET. Since the explanation started with the front it will continue with that.

In order to save the progress of the project I used Git which acted as the version control software which keeps track of all the changes by the commits I made. Git works with branches where to test multiple things you can create multiple branches without harming the main one. After that you can merge with the main one or rather to production. Beside that it makes the project open-source, anyone can contribute to it and the codebase is available at Figure 4.1.3: https://github.com/mateo-sheme/Diploma-Makino.

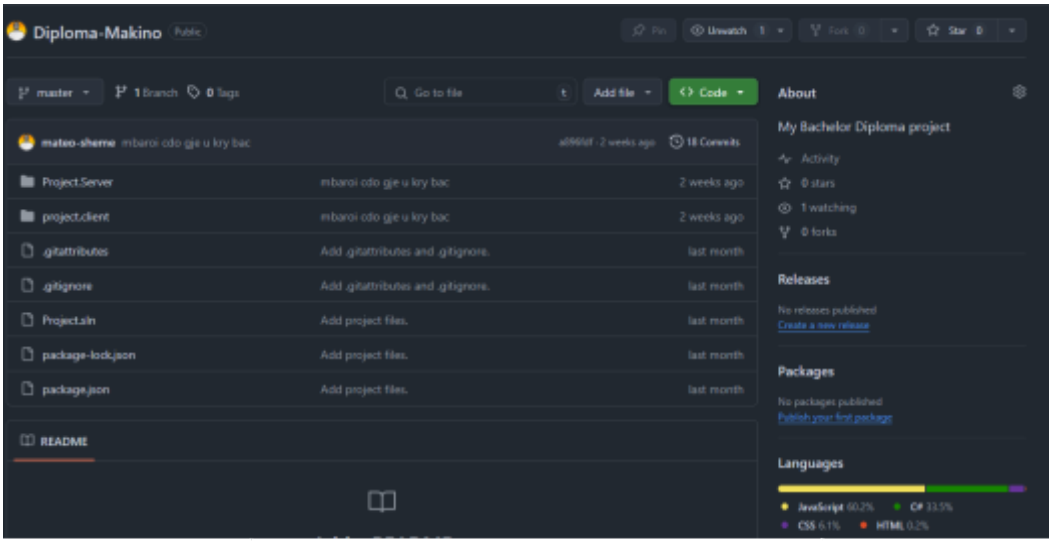


Figure 4.1.3
Github repository page of the project

At Figure 4.1.4 are all the commits of the project, where version-control helped was at the middle; it can be seen as a detour of orange. That means that at that time a second branch existed. I had some problems there and needed to revert to a commit which creates a temporary branch. After I managed to fix my problem I merged back with my main branch.

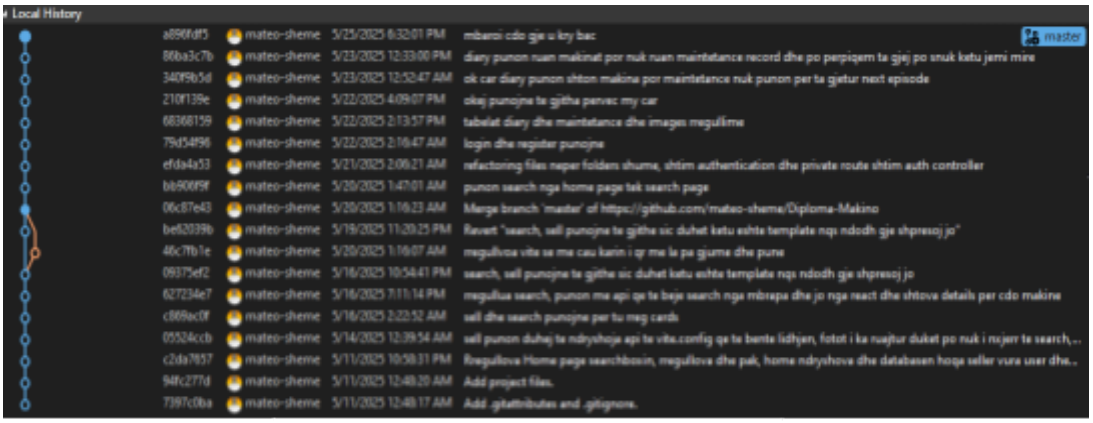


Figure 4.1.4
Github repository page of the project

4.2: REACT FOLDER STRUCTURE

At Figure 4.2.1 the react folder structure is composed of dependencies which are filled by the npm packages. Launch.json which tells the project which browser to run to. Public folder has the vite logo which is the logo of the tab of the project when opened in the browser left to the name.

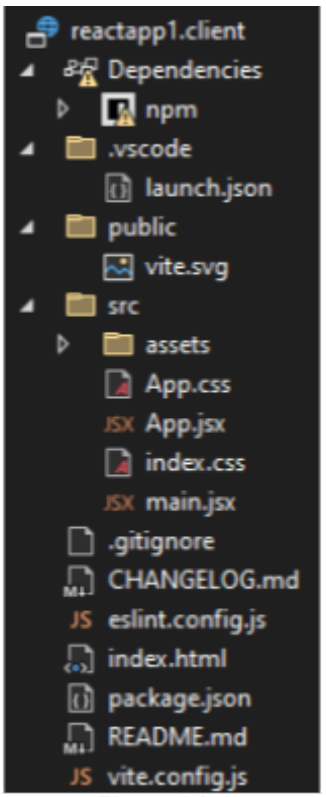


Figure 4.2.1
Template of React

Speaking of Vite, it is a frontend build tool that dramatically improves development speed, build performance, and developer experience. It is packed alongside React.

The src folder which keeps the css files as well as the react jsx files. This is the main core of the whole front-end pages as they all reside here.

The main entry point of the app is main.jsx (Figure 4.2.2) which renders <App />. That directs the main.jsx to the app.jsx which is the landing page of the project and where everything will be modified accordingly. Besides the default importings I have also imported the BrowserRouter library which will be necessary for using links later on.

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'
import { BrowserRouter } from "react-router-dom"


createRoot(document.getElementById('root')).render(
  <StrictMode>
    <BrowserRouter>
        <App />
    </BrowserRouter>
  </StrictMode>,
)
```

Figure 4.2.2
Main.jsx file

At Figure 4.2.3 is the App.jsx of the project which is the pinnacle of React's component work methodology. React handles all imports as separate standalone components which can be re-used and imported as many times as needed and in different ways without having to copy and paste the same code over and over everywhere.

Let's say we need a navbar and that needs to be on all the pages, it would need to be first defined and adapted to css accordingly then copied and pasted to the other page along with css and pray for god that nothing out of the ordinary comes out and you need to re-adjust it again from the beginning and then implementing the adjustments to everywhere where the navbar is. It's a nightmare! In React the components that will be reused will be declared at the components folder and from there they can be imported to any file in the project as well as other projects or solutions. Below is the App.jsx of the project and notice the Navbar that is imported from the components folder as component <Navbar /> and then the main App class is wrapped in the Navbar component meaning that all the pages inside there will have the navbar available to all of them. Besides that also the pages are imported as components but they are warped each by a route which allows for the pages to have a link when opened. Using that the navbar uses these links later on to have separate pages allowing the user to navigate back and forth as needed.

```
import { } from 'react';
import './App.css';
import {Route, Routes } from "react-router-dom"
import { Navbar } from "./components/Navbar.jsx"
import "bootswatch/dist/Litera/bootstrap.css";
import "bootswatch/dist/Litera/bootstrap.min.css";
import { AuthProvider } from './contexts/AuthContext';
import { PrivateRoute } from './contexts/PrivateRoute';
import { LanguageProvider } from './contexts/LanguageContext';

import Home from './pages/Home/Home.jsx'
import Search from './pages/Search/Search.jsx'
import Sell from './pages/Sell/Sell.jsx'
import CarDetails from './pages/Search/CarDetails.jsx';
import Login from './pages/Authentication/Login.jsx';
import Register from './pages/Authentication/Register.jsx';
import MaintenancePage from './pages/MyCar/MaintenanceRecord';
import DiaryCar from './pages/MyCar/DiaryCar.jsx';

function App() {
    return (
        <div className="App">
            <AuthProvider>
                <LanguageProvider>
                <Navbar />
                <Routes>
                    <Route path="/" element={<Home />} />
                    <Route path="/search" element={<Search />} />
                    <Route path="/sell" element={<PrivateRoute><Sell /></PrivateRoute>} />
                    <Route path="/diarycar" element={<PrivateRoute><DiaryCar />></PrivateRoute>} />
                    <Route path="/diarycar/:carId/maintenance" element={<MaintenancePage />} />
                    <Route path="/car/:id" element={<CarDetails />} />
                    <Route path="/login" element={<Login />} />
                    <Route path="/register" element={<Register />} />
                    </Routes>
                </LanguageProvider>
            </AuthProvider>
        </div>
    );
}
```

Figure 4.2.3
App.jsx file

4.3: THE WEBPAGE FUNCTIONALITIES

On the side at Figure 4.3.1 is the final frontend structure that I ended up with, well structured and everything separated in folders. Starting from the components, I have created only two which are the Navbar and the Searchbox which both have their own css file and are used at the home page at Figure 4.3.2. Both are modular and editable as needed. The logo serves as the home page button.

The homepage serves as the main navigation panel as well as the first landing page of the user in which the user can see an advertisement right away and also search for different types of cars that he is interested in. After filling some of the searchboxes it directly directs the user to the search page, with the filled data right away at Figure 4.3.6.

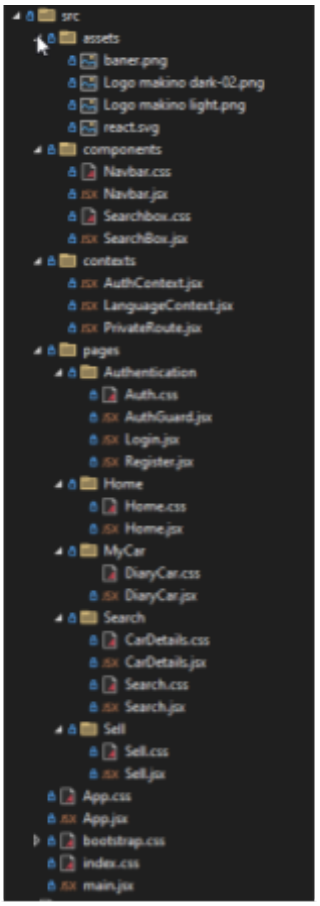


Figure 4.3.1
Finalized folder structure of the frontend

Figure 4.3.2
Home page

This is achieved by this constant at Figure 4.3.3 which loops through all the inserted values and saves them as params and directs the user to the user page filling the filters as strings. At the search page the user can input different filters and discover available cars based on their preferences such as brand, model, fuel type, transmission, and price range. It provides an intuitive way to narrow down options without needing to reload the entire page or manually input parameters. 0

```
const handleSearch = (e) => {
    e.preventDefault();
    const params = new URLSearchParams();
    Object.entries(searchParams).forEach(([key, value]) => {
        if (value) params.append(key, value);
    });
    navigate(`/search?${params.toString()}`);
};
```

Figure 4.3.3
Constant which parses the dates and sends to search

When a user interacts with any filter option (e.g., selects a brand or sets a price range), the frontend captures these values and updates the URL with corresponding query parameters, enabling both live results and shareable urls.

```
const Search = () => {
    const navigate = useNavigate();
    const { t, language } = useLanguage();
    const [searchParams] = useSearchParams();
    const [filters, setFilters] = useState({
        brand: searchParams.get('brand') || '',
        model: searchParams.get('model') || '',
        fuelType: searchParams.get('fuelType') || '',
        transmission: searchParams.get('transmission') || '',
        minPrice: searchParams.get('minPrice') || '',
        maxPrice: searchParams.get('maxPrice') || '',
        minKm: searchParams.get('minKm') || '',
        maxKm: searchParams.get('maxKm') || '',
        minYear: searchParams.get('minYear') || '',
        maxYear: searchParams.get('maxYear') || ''
    });

    const [cars, setCars] = useState([]);
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState(null);

    const fetchCars = async () => {
        try {
            setLoading(true);
            setError(null);

            const queryString = Object.entries(filters)
                .filter(([, value]) => value !== '')
                .map(([key, value]) => `${key}=${encodeURIComponent(value)}`)
                .join('&');

            navigate(`?${queryString}`, { replace: true });

            const response = await fetch(`/api/car/search?${queryString}`);
            if (!response.ok) throw new Error(t('failedToFetchCars'));

            const data = await response.json();
            setCars(data);
        } catch (err) {
            setError(err.message);
        } finally {
            setLoading(false);
        }
    };

    const handleFilterChange = (e) => {
        const { name, value } = e.target;
        setFilters(prev => ({
```

Figure 4.3.4
Logic of search const

All the cars will be loaded through the fetch call at the response which forwards the url to /api/car/search?{queryString} at Figure 4.3.4 which is the string filled from the controller based on the filters applied which are declared as constants filters at the beginning alongside as searchParams which then at the constant queryString are mapped as values and joined together, which then fed to the navigate(`?${queryString}`, { replace: true });, which replaces the url to the query string provided. The /api/car/search endpoint parses query parameters into a

https://makino.alegria.com.al/search?brand=BMW&model=X5

Figure 4.3.5
Url of the application

dynamic filter set which queries the database for matching cars. It supports partial matches and range queries, ensuring accurate and relevant results. Finally the response is finalized by the /api alongside the queryString to the url at Figure 4.3.5.

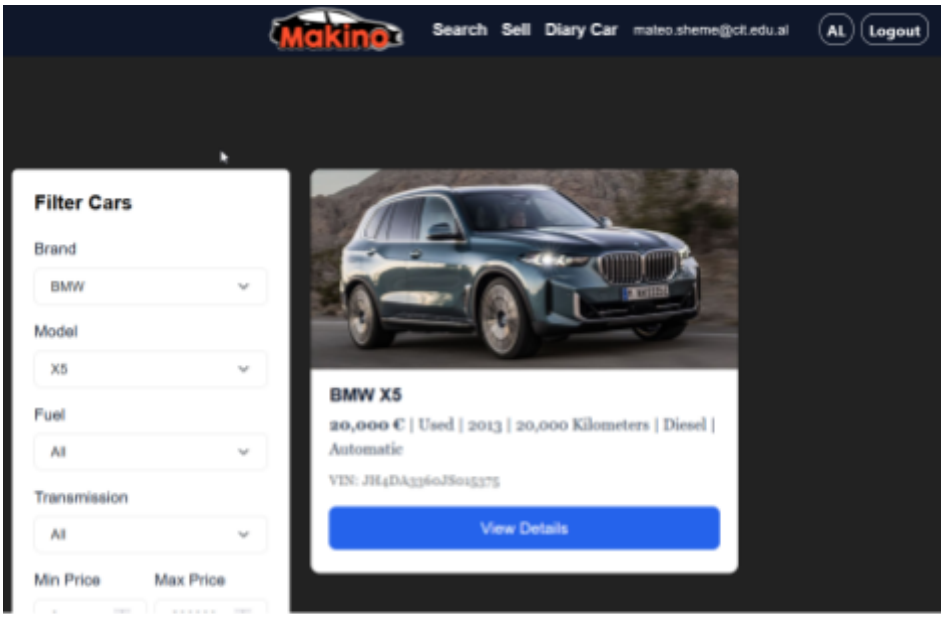


Figure 4.3.6
Search page

After finding the desired car the user clicks the more detailed car page at Figure 4.3.7 and can see beside the first image on the listings a carousel of many images, which then after inspecting from up close can click the contact seller button which on mobile directs the number associated to the listing default calling app.

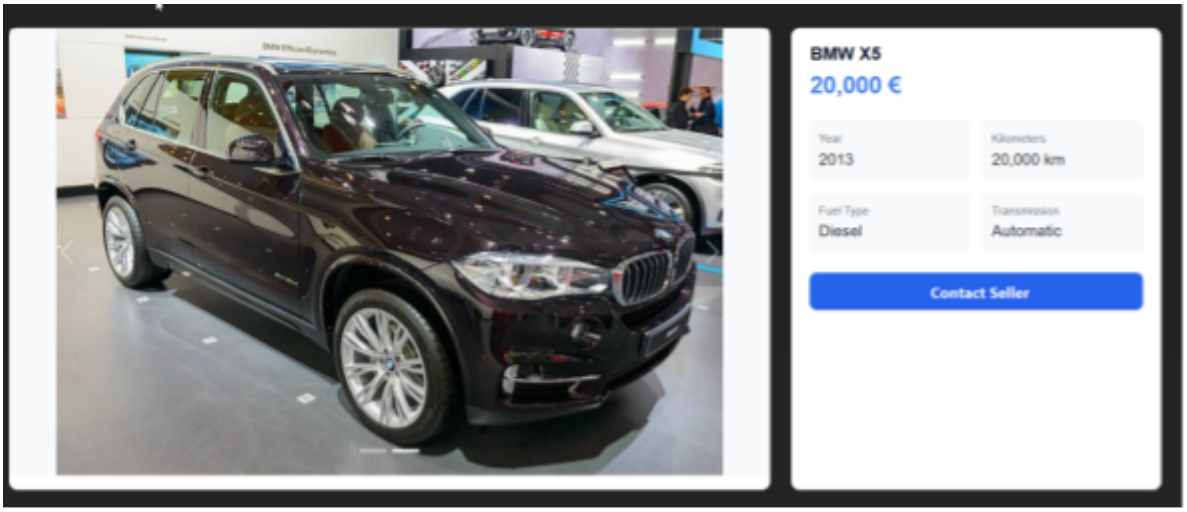


Figure 4.3.7
Car details page

Figure 4.3.8 is mostly the logic of the page, as it gets the data from the /api/car and the id of the car which is filled from the previous search page when clicking the button.

```
import { useParams } from 'react-router-dom';
import { useEffect, useState } from 'react';
import { Carousel } from 'react-bootstrap';
import "bootswatch/dist/Litera/bootstrap.min.css";
import './CarDetails.css';
import { useLanguage } from "../../contexts/LanguageContext";

3 references
function CarDetailsPage() {
    const { id } = useParams();
    const { t    } = useLanguage();
    const [car, setCar] = useState(null);
    const [error, setError] = useState(null);
    const [activeIndex, setActiveIndex] = useState(0);

    useEffect(() => {
        const fetchCarDetails = async () => {
            try {
                const response = await fetch(`/api/car/${id}`);
                if (!response.ok) throw new Error('Car not found');
                const data = await response.json();
                setCar(data);
            } catch (err) {
                setError(err.message);
            }
        };
        fetchCarDetails();
    }, [id]);

    const handleSelect = (selectedIndex) => {
        setActiveIndex(selectedIndex);
    };

    if (error) return <div className="alert alert-danger mt-4">Error: {error}</div>;
    if (!car) return <div class="text-center mt-4"><div class="spinner-border" role="status"></div></

    return (
        <div class="container mt-4">
            <div class="row">
                {/* Image Gallery Column */}
                <div class="col-lg-8">
                    {/* Main Carousel */}
                    <div class="card mb-3">
                        <Carousel activeIndex={activeIndex} onSelect={handleSelect} interval={null}>
                            {car.images?.map({image, index} => {
                                <Carousel.Item key={index}>
                                    <img
                                        class="d-block w-100 main-car-image"
                                        src={`/api/car/${car.car_ID}/image/${image.image_ID}`}
                                        alt={`${car.brand} ${car.model} - ${index + 1}`}
```

Figure 4.3.8
Logic of car details

This is the sell page, at Figure 4.3.9 where each user can upload any car listings of cars that he desires. The uploading is handled by the constant handleSubmit which first gets the entries from the form data, after getting the user id so that on the database each uploaded car is assigned to a user.

After all of that the images that are uploaded are bundled together alongside the formdata and is sent as a POST request to the backend to the /api/car url.

Figure 4.3.9
Sell page

Final thing to notice, is the credentials: “include”, I also added a security feature where based on the logged in user credentials are translated to the id. This id is then used at the previous user id. Which then after fetching is used at the POST request to tell the database that this car is assigned to this user, as he is the one that uploaded it.

```
const handleSubmit = async (e) => {
    e.preventDefault();
    setError(null);
    setSuccess(false);
    setIsSubmitting(true);

    if (!validateForm()) {
        setIsSubmitting(false);
        return;
    }

    try {
        const formDataToSend = new FormData();

        // adds all fields of the formdata
        Object.entries(formData).forEach(([key, value]) => {
            formDataToSend.append(key, value);
        });

        // gets the userid
        formDataToSend.append("userId", userId);

        // adds all the images of the formdata
        images.forEach((image) => {
            formDataToSend.append("images", image);
        });

        const response = amait fetch("/api/car", {
            method: "POST",
            body: formDataToSend,
            credentials: 'include'
        });

        if (!response.ok) {
            const errorData = amait response.json().catch(() => null);
            throw new Error(errorData?.message || "Failed to submit form");
        }

        setSuccess(true);
        setFormData(initialForm);
        setImages([]);
```

Figure 4.3.10
Logic of submitting cars in sell

This doesn't affect the other pages home or search, as the purpose of those two is mostly advertising and seeing. If this security is implemented as well then all the visitors would have to login in order to see the cars or even be able to search.

In summary how this security route works, as seen on Figure 4.3.10 there is a folder context composed of the 3 files AuthContext.jsx, LanguageContext.jsx and PrivateRoute.jsx.

The final page of the application is the DiaryCar page at Figure 4.3.11, the most important and yet most difficult to have working appropriately. The main purpose of this page is for the user to see the maintenance records of his cars. The user is able to insert multiple cars as well as multiple types of maintenance records.

Figure 4.3.11
Diary car page

The cars table of this page is separate from the cars listed for sale and that is connected to the user table. The question arises why? Well if the same table would be used for the car listings and diary, both cars would be published to the search page ready for selling and would cause chaos between the users, as they would receive calls for the car which they don't intend on selling.

At Figure 4.3.12 is the code that besides fetches the car from the /api/diarycar it also assigns data variables of the front to the database table fields and besides that for each fetched car it loads the local maintenance records based on the user car id.

```
useEffect(() => {
    if (currentUser?.userId && initialLoad) {
        const loadData = async () => {
            setLoading(true);
            try {
                await fetchUserCars();
            } finally {
                setInitialLoad(false);
                setLoading(false);
            }
        };
        loadData();
    }
}, [currentUser?.userId, initialLoad]);

useEffect(() => {
    if (selectedCar?.User_Car_ID) {
        const records = localMaintenanceRecords[selectedCar.User_Car_ID] || [];
        setMaintenanceRecords(records);
    }
}, [selectedCar?.User_Car_ID]);

const fetchUserCars = async () => {
    setLoading(true);
    try {
        const response = await fetch('/api/diarycar', {
            credentials: 'include'
        });
        if (!response.ok) throw new Error("Failed to fetch cars");
        const data = await response.json();

        const normalizedCars = data.map(car => ({
            User_Car_ID: car.user_Car_ID || car.User_Car_ID,
            User_ID: car.user_ID || car.User_ID,
            Nickname_Car: car.nickname_Car || car.Nickname_Car || "",
            VIN: car.vin || car.VIN || "",
            Brand: car.brand || car.Brand || "",
            Model: car.model || car.Model || "",
            License_Plate: car.license_Plate || car.License_Plate || "",
            Current_Kilometers: car.current_Kilometers || car.Current_Kilometers || 0,
            Fuel: car.fuel || car.Fuel || "",
            Car_Image: car.car_Image || car.Car_Image || null,
            Content_Type: car.content_Type || car.Content_Type || "",
            Insurance_Expiry: car.insurance_Expiry || car.Insurance_Expiry || null,
            Inspection_Expiry: car.inspection_Expiry || car.Inspection_Expiry || null,
            Maintenance_Record: car.maintenance_Record || car.Maintenance_Record || []
        }));

        setCars(normalizedCars);
        setSelectedCar(normalizedCars.length > 0 ? normalizedCars[0] : null);

        normalizedCars.forEach(car => {
            if (!localMaintenanceRecords[car.User_Car_ID]) {
                localMaintenanceRecords[car.User_Car_ID] = [];
            }
        });
```

Figure 4.3.12
Logic of the fetching cars

On figure 4.3.13 is the adding car logic of the car where it gets the id if from the editingCarId, if the car already existsIf it does exist, this means the car already exists in the database and should be updated. Therefore, the function performs a PUT request to the /api/diarycar/:id endpoint. If it does not exist, the car is considered new, and a POST request is sent to /api/diarycar to create it.

```
try {
    const formData = new FormData();
    Object.entries(carForm).forEach(([key, value]) => {
        if (value) formData.append(key, value);
    });
    formData.append("User_ID", currentUser.userId);
    if (image) formData.append("Car_Image", image);

    const url = editingCarId
        ? `/api/diarycar/${editingCarId}`
        : "/api/diarycar";
    const method = editingCarId ? "PUT" : "POST";

    const response = await fetch(url, {
        method,
        body: formData,
        credentials: 'include'
    });

    if (!response.ok) {
        const errorData = await response.json();
        throw new Error(errorData.message || "Failed to save car");
    }

    const savedCar = await response.json();

    if (editingCarId) {
        setCars(cars.map(c =>
            c.User_Car_ID === savedCar.User_Car_ID ? savedCar : c
        ));
        if (selectedCar?.User_Car_ID === savedCar.User_Car_ID) {
            setSelectedCar(savedCar);
        }
    } else {
        setCars([...cars, savedCar]);
        setSelectedCar(savedCar);
        localMaintenanceRecords[savedCar.User_Car_ID] = [];
    }

    setShowAddCarForm(false);
    setEditingCarId(null);
    setCarForm({
        Nickname_Car: "",
        VIN: "",
        Brand: "",
        Model: "",
        License_Plate: "",
        Current_Kilometers: "",
        Fuel: "",
        Insurance_Expiry: "",
        Inspection_Expiry: "
```

Figure 4.3.13
Logic of diary car fetching

Before sending the request, a FormData object is created and filled with the data from the carForm and the current user's ID. If an image has been uploaded, it is included as well. The response is processed to update the local state at cars, selectedCar, and to clear the form, and show success or error messages accordingly.

```
const handleEditCar = (carId) => {
    const carToEdit = cars.find(c => c.User_Car_ID === carId);
    if (!carToEdit) return;

    setEditingCarId(carId);
    setCarForm({
        Nickname_Car: carToEdit.Nickname_Car || "",
        VIN: carToEdit.VIN || "",
        Brand: carToEdit.Brand || "",
        Model: carToEdit.Model || "",
        License_Plate: carToEdit.License_Plate || "",
        Current_Kilometers: carToEdit.Current_Kilometers || "",
        Fuel: carToEdit.Fuel || "",
        Insurance_Expiry: carToEdit.Insurance_Expiry ? carToEdit.Insurance_Expiry.split
        Inspection_Expiry: carToEdit.Inspection_Expiry ? carToEdit.Inspection_Expiry.sp
    });

    setImage(null);
    setShowAddCarForm(true);
};

const handleEditCarSubmit = async (e) => {
    e.preventDefault();
    setError(null);
    setSuccess(null);

    try {
        const formData = new FormData();
        Object.entries(carForm).forEach(([key, value]) => {
            if (value) formData.append(key, value);
        });
        formData.append("User_ID", currentUser.userId);

        if (image) {
            formData.append("Car_Image", image);
        } else {
            formData.append("KeepExistingImage", "true");
        }

        const response = await fetch(`/api/diarycar/${editingCarId}`, {
            method: "PUT",
            body: formData,
            credentials: 'include'
        });

        if (!response.ok) {
            const errorData = await response.json();
            throw new Error(errorData.message || "Failed to update car");
        }

        const updatedCar = await response.json();

        setCars(cars.map(c =>
            c.User_Car_ID === updatedCar.User_Car_ID ? updatedCar : c
```

Figure 4.3.14
Logic of editing diary car

On figure 4.3.14 is the editing logic of the car which is triggered when a car with a specific User_Car_ID is selected for editing. Finds the selected car in the list of user cars and loads its data into the carForm so the user can modify it. Sets the form into edit by updating editingCarId, and shows the car form to the user. Once the form is submitted, the handleEditCarSubmit function handles the PUT request to update the car. If no new image is selected, a flag (KeepExistingImage) is added to preserve the current image on the server. The same logic is applied for the maintenance records when added or edited.

4.4: THE CONTEXT FILES

The AuthContext is the one on Figure 4.4.1. What it does is it gets the logged in user and stores that user at the storedUser as json.

It also handles login which stores the user in the localstorage which then is parsed by the previous useEffect. After logging it it takes the user to the home page and also the opposite happens when logging out by removing it from the localStorage and taking the user to the login.

Finally the return at the bottom children, it means the pages at App.jsx which will be wrapped with the context in order to not leave the visitors visiting pages that we don't intend to without first logging in.

At Figure 4.2.3 is the App.jsx which the context wrapping can be seen as AuthProvider which is the file imported directly from the context folder. So to sum up Authcontext acts as the main responsible file to log in the user to the application itself.

```
import { useNavigate } from 'react-router-dom';
import PropTypes from 'prop-types';

const AuthContext = createContext();

1 reference
export function AuthProvider({ children }) {
    const [currentUser, setCurrentUser] = useState(null);
    const navigate = useNavigate();

    // shikon user te loguar per here te pare
    useEffect(() => {
        const storedUser = localStorage.getItem('user');
        if (storedUser) {
            setCurrentUser(JSON.parse(storedUser));
        }
    }, []);

    const login = (userData) => {
        localStorage.setItem('user', JSON.stringify(userData));
        setCurrentUser(userData);
        navigate('/'); // mbasi ben login te con te home
    };

    const logout = () => {
        localStorage.removeItem('user');
        setCurrentUser(null);
        navigate('/login'); // mbas login te con te logout
    };


    return (
        <AuthContext.Provider value={{ currentUser, login, logout }}>
            {children}
        </AuthContext.Provider>
    );
}

AuthProvider.propTypes = {
    children: PropTypes.node.isRequired,
};
```

Figure 4.4.1
AuthContext context file

```
import { useContext } from 'react';
import { Navigate } from 'react-router-dom';
import { AuthContext } from '../contexts/AuthContext';
import PropTypes from 'prop-types';

const PrivateRoute = ({ children }) => {
    const { currentUser } = useContext(AuthContext);
    return currentUser ? children : <Navigate to="/login" />;
};

PrivateRoute.propTypes = {
    children: PropTypes.node.isRequired,
};

export { PrivateRoute };
```

Figure 4.4.2
PrivateRoute context file

At Figure 4.4.2 is the PrivateRoute which is the security feature that I mentioned before, which doesn't allow the visitor to see the sell page and later the diary car page without logging in. If the visitor would be allowed to there would no id associated with the listing or entries and that would cause problems with the database.

It gets the AuthContext logged in user mentioned earlier, and if it doesn't find it, sends the visitor to the login page. Below that are also the childrens declared similar to AuthContext and at the App.jsx at Figure 4.2.3 sell page and diary car have a private route declared which forwards them here.

```
            cancel: "Anulo",
            maintenanceHistory: "Historia e Mir\u00EBmbajtjes",
            addMaintenance: "Shto Mir\u00EBmbajtje",
            editMaintenance: "Ndrysho Regjistrin e Mir\u00EBmbajtjes",
            addMaintenanceRecord: "Shto Regjist\u00EBr Mir\u00EBmbajtjeje",
            noMaintenanceRecord: "Asnj\u00EB historik mir\u00EBmbajtje",
            saveRecord: "Ruaj Regjistrin",
            updateRecord: "P\u00EBrdit\u00EBso Regjistrin",
            date: "Data",
            type: "Lloji",
            nextMaintenanceKm: "KM Tjet\u00EBr",
            notes: "Sh\u00EBnime",
            action: "Veprim",
            licensePlate: "Targa",
            currentKm: "Km t\u00EB tanishme",

            filterCars: "Filtro Makinat",
            all: "T\u00EB gjitha",
            loadingCars: "Duke ngarkuar makinat...",
            noCarsFound: "Nuk u gjet\u00EBn makina q\u00EB p\u00EBrputhen me kriteret tuaja.",
            noImage: "Asnj\u00EB Foto",
            viewDetails: "Shiko Detajet",
            failedToFetchCars: "D\u00EBshtoi n\u00EB marrjen e makinave",
        }
    };

    const t = (key) => translations[language][key] || key;

    return (
        <LanguageContext.Provider value={{ language, setLanguage, t }}>
            {children}
        </LanguageContext.Provider>
    );
};

LanguageProvider.propTypes = {
    children: PropTypes.node.isRequired
};


export const useLanguage = () => useContext(LanguageContext);
```

Figure 4.4.3
Albanian translations at the language context file

The final context file present is the LanguageContext.jsx file which is responsible for doing the translation from english to albanian and vice versa. On Figure 4.4.3 are the Albanians counter names.

So how it works the same variable name is declared at both translations and each is then called whenever the language is changed by the user.

```
<div className="d-flex align-items-center">
    {currentUser && (
        <span className="nav-email text-white me-3">{currentUser.email}</span>
    )}
    <button
        className="btn btn-outline-light btn-sm mx-2"
        onClick={toggleLanguage}
    >
        {language === 'en' ? 'AL' : 'EN'}
    </button>

    {currentUser ? (
        <button
            className="btn btn-outline-light btn-sm"
            onClick={handleLogout}
        >
            Logout
        </button>
    ) : (
        <Link
            to="/login"
            className="btn btn-outline-light btn-sm"
        >
            Login
        </Link>
```

Figure 4.4.4
Language button at the navbar

Finally at Figure 4.4.3 again, there is a const t which is the constant marked on the other pages telling it to change and finally returned as a context page to be used in the App.jsx (Figure 4.2.3) all the pages are wrapped in the context alongside the previous contextes. Figure 4.4.4 is how the user tells the application to change the language as there is the button at the navbar which calls the language context and by default english is default.

The final context file is the PrivateRoute Figure 4.4.5 which is the big brother of logged in users, where if there isn't a current user logged in it can't access to the children pages which are then the pages at the App.jsx again at Figure 4.2.3. At the App.jsx is then called and the pages are wrapped with the private route Figure 4.4.6 if the specified pages are marked like this as not all pages are required for the private route.

```
import { useContext } from 'react';
import { Navigate } from 'react-router-dom';
import { AuthContext } from '../contexts/AuthContext';
import PropTypes from 'prop-types';

const PrivateRoute = ({ children }) => {
    const { currentUser } = useContext(AuthContext);
    return currentUser ? children : <Navigate to="/login" />;
};

PrivateRoute.propTypes = {
    children: PropTypes.node.isRequired,
};

export { PrivateRoute };
```

Figure 4.4.5
PrivateRoute file

```
<Route path="/" element={<Home />} />
<Route path="/search" element={<Search />} />
<Route path="/sell" element={<PrivateRoute><Sell /></PrivateRoute>} />
<Route path="/diarycar" element={<PrivateRoute><DiaryCar /></PrivateRoute>} />
```

Figure 4.4.6
Private route declaration at the App.jsx

4.5: LOGIN, REGISTER AND MOBILE RESPONSIVES

At figure 4.5.1 is the registration logic where it accepts 3 fields, email, password and password. All three fields have regex in order to handle correct data. After all three fields are filled and the requirements met then it tries to do a POST request to /api/auth/register as a json header and then the body the fields and if everything goes well the registration succeeds and sends the user to the login page. If not it fails and an error message is thrown.

```
const handleSubmit = async (e) => {
    e.preventDefault();
    setIsLoading(true);
    setSuccess(false);
    setError("");

    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    const passwordRegex = /^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d]{6,15}$/;

    if (!userData.Email || !userData.PasswordHash || !userData.Phone) {
        setError("All fields are required");
        setIsLoading(false);
        return;
    }

    if (!emailRegex.test(userData.Email)) {
        setError(t("invalidEmail"));
        setIsLoading(false);
        return;
    }

    if (!passwordRegex.test(userData.PasswordHash)) {
        setError(t("passwordRequirements"));
        setIsLoading(false);
        return;
    }

    if (!validatePhone(userData.Phone)) {
        setError(t("phoneRequirements"));
        setIsLoading(false);
        return;
    }

    try {
        const response = await fetch("/api/auth/register", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({
                Email: userData.Email,
                PasswordHash: userData.PasswordHash,
                Phone: userData.Phone
            }),
        });

        const data = await response.json();

        if (!response.ok) {
            if (data.field === "email") {
                setError(data.message);
            } else {
                setError(data.message || t("registrationFail"));
            }
            return;
        }

        setSuccess(true);
        setTimeout(() => navigate("/login"), 2000);
    } catch (err) {
        setError(err.message || "Registration failed. Please try again.");
```

Figure 4.5.1
Register logic

```
const handleSubmit = async (e) => {
    e.preventDefault();
    setIsLoading(true);
    setError("");

    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

    if (!emailRegex.test(email)) {
        setError("Invalid email format.");
        setIsLoading(false);
        return;
    }

    try {
        const response = await fetch("/api/auth/login", {
            method: "POST",
            headers: { "Content-Type": "application/json", },
            body: JSON.stringify({
                Email: email,
                PasswordHash: password
            }),
            credentials: 'include'
        });

        const data = await response.json();

        if (!response.ok) {
            throw new Error(data.message || "Login failed");
        }

        login({
            userId: data.userId,
            email: data.email
        });
        navigate("/");

    } catch (err) {
        setError(err.message);
    } finally {
        setIsLoading(false);
    }
};
```

Figure 4.5.2
Login logic

At figure 4.5.2 is the login logic where the registered user logs in. It first awaited the email and password from the front page which after filled. Email field has a regex just like registration and after the email and password are sent to the /api/auth/login. Important here is that the credentials are stored along with user id and this is then used later for the authcontext and private

route so that only logged in users can access specific pages.

At Figure 4.5.3 is the register page that was mentioned on the previous page. The similar is for the login page. Starting the development I struggled a lot with the navbar alone and since this app should stand out and be as easy to interact as possible for the user. When opened on a mobile device in the Figure is Samsung Note 20 the navbar shows the hamburger icon which when clicked it opens the sidebar on Figure 4.5.4.

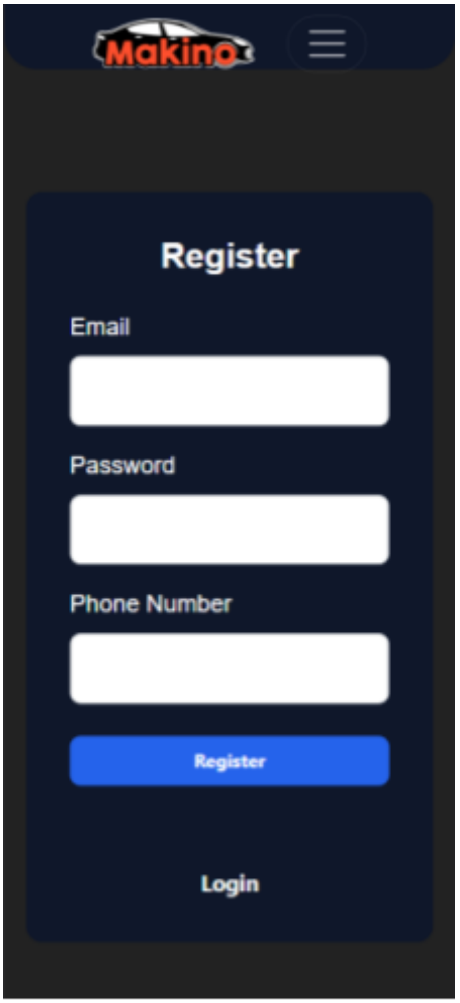


Figure 4.5.3
Register page in mobile

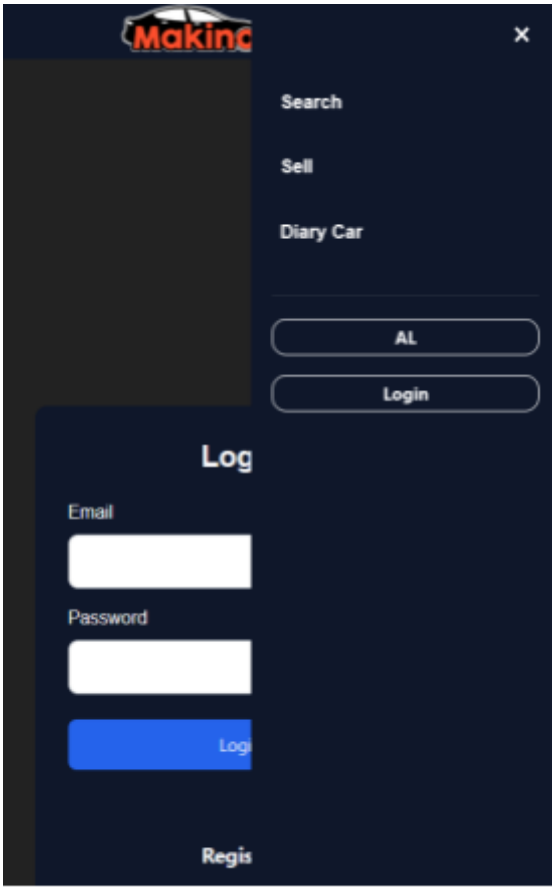


Figure 4.5.4
Login page in mobile with expanded sidebar

At Figure 4.5.4 is the login page similar to the register page. Bootstrap is used to make the whole project more flexible and responsive. A template from bootswatch.com is imported called Litera and based on that template and the bootstrap grid system I was able to build the frontend.

Chapter V: BACKEND

The backend for the project is ASP.NET Core Framework version 8. ASP.NET applications are structured around a core entry point, typically the Program.cs file. This file initializes the application using WebApplicationBuilder, which configures essential services and defines the application behavior. The file is divided into two main sections, first is service configuration, responsible for various services, middleware, and dependencies are registered, responsible for service activation, where middleware components like routing and authorization are added to the request pipeline.

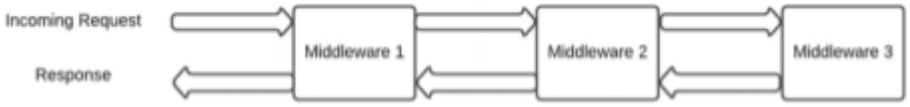


Figure 5.0.1
The ASP.NET Core middleware pipeline
Reprinted from *Coding Clean, Reliable, and Safe REST APIs with ASP.NET Core 8: Develop Robust Minimal APIs with .NET 8 * (p. 2, Figure 2.1), by A. Giretti, 2023, APress, Copyright © Anthony Giretti 2023

ASP.NET leverages features like top-level statements and global using directives to streamline code in .NET 8 templates. One key architectural principle is dependency injection, which promotes loose coupling and testability by allowing services to be injected where needed rather than instantiated manually. Services in ASP.NET can be registered with different lifetimes: transient (new instance every time), scoped (one instance per request), and singleton (one shared instance for the application's lifetime). These services are typically injected into classes via constructors, following interface-based abstraction principles. (A. Giretti, 2023, Chapter 2)

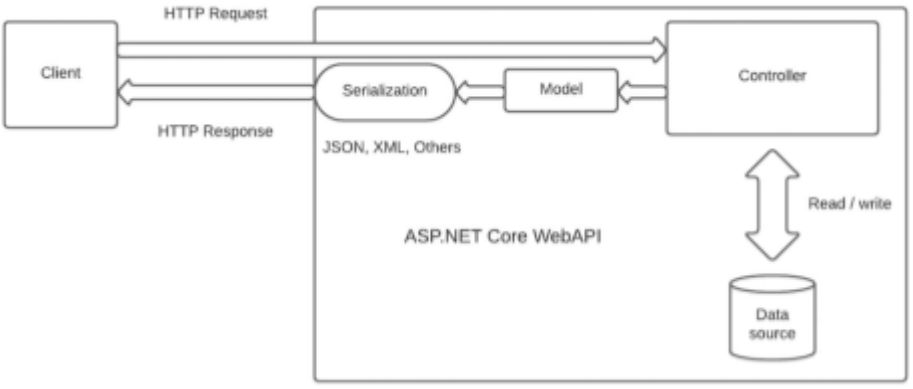


Figure 5.0.2
ASP.NET Core Web API architecture
Reprinted from *Coding Clean, Reliable, and Safe REST APIs with ASP.NET Core 8: Develop Robust Minimal APIs with .NET 8 * (p. 2, Figure 2.4), by A. Giretti, 2023, APress, Copyright © Anthony Giretti 2023

5.1: APPLICATION DB CONTEXT

Before doing anything with the backend you have to first declare the connection to the database. This is the file at Figure 5.1.1 that connects the entity framework to the database as well as used for all the controllers when interacting with the database.

I have declared it as DbContext and then below are the table names that are called to be created by using DbSet and also used as calling names later for the controllers. As it can be seen I have 5 of these statements meaning 5 tables. Out of these 5 tables 3 are the most important, the user table, the cars_sale and the diary_car tables. They also need relations to work properly. This is the purpose of the modelBuilder at the bottom.

```
using Project.Server.Entities;
using Microsoft.EntityFrameworkCore;
namespace Project.Server.Data;


23 references
public class ApplicationDbContext : DbContext
{
    0 references
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
    {

    }

    4 references
    public DbSet<Cars_Sale> Cars_Sale { get; set; } //Dbset <emri i klases qe deklarova> me emrin
    3 references
    public DbSet<User> Users { get; set; }
    1 reference
    public DbSet<Car_Image> Car_Images { get; set; }
    9 references
    public DbSet<Diary_Car> Diary_Car { get; set; }
    5 references
    public DbSet<Maintenance_Record> Maintenance_Record { get; set; }

    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        //mardheniet e tabelave
        modelBuilder.Entity<User>()
        .HasMany(u => u.Cars)
        .WithOne(c => c.User)
        .HasForeignKey(c => c.User_ID)
        .OnDelete(DeleteBehavior.Cascade);

        modelBuilder.Entity<Cars_Sale>()
            .HasMany(c => c.Images)
            .WithOne(i => i.Car)
            .HasForeignKey(i => i.Car_ID)
            .OnDelete(DeleteBehavior.Cascade);

        modelBuilder.Entity<Diary_Car>()
           .HasMany(d => d.Maintenance_Record)
           .WithOne(m => m.Car)
           .HasForeignKey(m => m.User_Car_ID)
            .OnDelete(DeleteBehavior.Cascade);
    }
}
```

Figure 5.1.1
Application Db Context file

The modelBuilder is a Fluent API, it gives more control compared to data annotations as it handles the relations of the tables in the database. So the main table is the User table which is connected to Cars_Sale and the Diary_Car and a connection 1 to many. One User can have many Cars_Sale which are noted as Cars. Each Car_Sale has one User and is referenced by the foreign key User_ID. If a User is deleted all their cars will be deleted too using the cascade effect.

The Cars_Sale table is connected to the Car_Image table in order to handle multiple pictures when listing a car for sale. So one car can have multiple car images. Each image belongs to a car that is dictated by the .WithOne. Each image belongs to one car and is linked via Car_ID. If the car is deleted any linked images are also deleted.

The Diary_Car table is connected to the Maintenance_Record in order to handle multiple maintenance records for a specific car. It has a one to many relationship. Only one diary car can have many maintenance records and each maintenance record is linked via the User_Car_ID, which is the foreign key of the car table which has the User_ID as foreign key. So its foreign key referencing a foreign key referencing a primary key. This is the reason it has User_Car before the ID as it needs to denote both of the keys beforehand. Finally it has the cascade effect when a Car is deleted from the diary all its maintenance records will be deleted.

5.2: CAR CONTROLLER

The Controller on the MVC architecture holds the logic of the entire application. Multiple controllers can be used in order to have a cohesive application. I decided to use a controller for each major activity in the application. The first one is the Car Controller and its endpoints are responsible for managing the car listing to either list them or fetch them accordingly. At Figure 5.2.1 is the header of the controller and this is replicated to all the other ones. The attribute[ApiController] tells ASP.NET Core that this controller is for building web API and not a normal MVC view controller.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Project.Server.Data;
using Project.Server.Entities;

namespace Project.Server.Controllers
{
    [ApiController]
    [Route("api/[controller]")] //rutoni i api qe ben te dali te url id e makines
    1 reference
    public class CarController : Controller
    {
        private readonly ApplicationDbContext _db;

        0 references
        public CarController(ApplicationDbContext db)
        {
            _db = db;
        }

        [HttpGet("search")]
        0 references
        public async Task<ActionResult<IEnumerable<Cars_Sale>>> SearchCars(
            [FromQuery] string brand = null,
            [FromQuery] string model = null,
            [FromQuery] string fuelType = null,
            [FromQuery] string transmission = null,
            [FromQuery] int? minPrice = null,
            [FromQuery] int? maxPrice = null,
            [FromQuery] int? minKm = null,
            [FromQuery] int? maxKm = null,
            [FromQuery] int? minYear = null,
            [FromQuery] int? maxYear = null)
        {
            try
            {
                var query = _db.Cars_Sale
                    .Include(c => c.Images)
                    .AsQueryable();

                if (!string.IsNullOrEmpty(brand))
                    query = query.Where(c => c.Brand.Contains(brand));
                if (!string.IsNullOrEmpty(model))
                    query = query.Where(c => c.Model.Contains(model));
                if (!string.IsNullOrEmpty(fuelType))
                    query = query.Where(c => c.Fuel == fuelType);
                if (!string.IsNullOrEmpty(transmission))
                    query = query.Where(c => c.Transmission == transmission);
                if (minPrice.HasValue)
                    query = query.Where(c => c.Price >= minPrice.Value);
                if (maxPrice.HasValue)
                    query = query.Where(c => c.Price <= maxPrice.Value);
```

Figure 5.2.1
Car controller file

- It enables model validation auto-check which when an error is spotted it automatically return 400 Bad Request
- [FromBody]/[FromQuery] inference, which tells the frameworks where to bind parameters from body or query
- Consistent error responses

After [ApiController] is the attribute [Route("api/[controller]")] which sets the base route URL path for all actions in the controller, meaning everytime this controller is called the url will supply on its own any request done by the user and letting the user know exactly where he is or what is requesting by the url. So in an example the attribute would turn into /api/car.

Inside the method Controller there are two variables declared, one of which is the ApplicationDbContext which I mentioned on the previous page which is responsible for calling the entity framework and the entities or models. The first endpoint at Figure 5.2.2 is a HttpGet("search") which by its name is the one responsible for getting the cars from the db and it is called SearchCars. Besides those it also has [FromQuery] which beside calling the car it also gets the fields from the table that will be associated with each car. This is used for the filters on the search page.

```
public async Task<ActionResult<IEnumerable<Cars_Sale>>> SearchCars(
    [FromQuery] string brand = null,
    [FromQuery] string model = null,
    [FromQuery] string fuelType = null,
    [FromQuery] string transmission = null,
    [FromQuery] int? minPrice = null,
    [FromQuery] int? maxPrice = null,
    [FromQuery] int? minKm = null,
    [FromQuery] int? maxKm = null,
    [FromQuery] int? minYear = null,
    [FromQuery] int? maxYear = null)
{
    try
    {
        var query = _db.Cars_Sale
            .Include(c => c.Images)
            .AsQueryable();

        if (!string.IsNullOrEmpty(brand))
            query = query.Where(c => c.Brand.Contains(brand));
        if (!string.IsNullOrEmpty(model))
            query = query.Where(c => c.Model.Contains(model));
        if (!string.IsNullOrEmpty(fuelType))
            query = query.Where(c => c.Fuel == fuelType);
        if (!string.IsNullOrEmpty(transmission))
            query = query.Where(c => c.Transmission == transmission);
        if (minPrice.HasValue)
            query = query.Where(c => c.Price >= minPrice.Value);
        if (maxPrice.HasValue)
            query = query.Where(c => c.Price <= maxPrice.Value);
        if (minKm.HasValue)
            query = query.Where(c => c.Kilometers >= minKm.Value);
        if (maxKm.HasValue)
            query = query.Where(c => c.Kilometers <= maxKm.Value);
        if (minYear.HasValue)
            query = query.Where(c => c.Year >= minYear.Value);
        if (maxYear.HasValue)
            query = query.Where(c => c.Year <= maxYear.Value);

        var results = await query.ToListAsync();
        return Ok(results);
    }
    catch (Exception ex)
    {
        return StatusCode(500, $"Error searching cars: {ex.Message}");
    }
}
```

Figure 5.2.2
First endpoint inside the Car Controller

After that is a try catch which declares the query from the database and the table Cars_Sale, beside that it also includes the images that are assigned to the car itself. Below are many if statements that act as defense mechanisms in order not to fetch empty data, as if something has happened in the insert and empty data has been inserted, this makes sure that if empty data is inserted it cant be shown.

After that the results are bundled in a results variable which tells the query from the db to list with the async attribute which tells this method is asynchronous. At the end if anything goes wrong it catches the error and mostly this is used for telling where it can't fetch the search car according to the filters.. In the end the purpose of this endpoint is to return a list of cars matching the filters added and their images.

This endpoint at Figure 5.2.3 is responsible for getting the details of the car according to the car id. Now the missing piece of the previous endpoint is fetching the id of the car itself. So it does fetch all its attributes but not the id of the car itself. So it calls the GET /api/car/5 if the id of the car is 5 alongside the images. If the id is null then it returns 404 not found.

```
[HttpGet("{id}")]
1 reference
public async Task<ActionResult<Cars_Sale>> GetCar(int id)
{
    var car = await _db.Cars_Sale
        .Include(c => c.Images)
        .FirstOrDefaultAsync(c => c.Car_ID == id);
    if (car == null)
    {
        return NotFound();
    }
    return Ok(car);
}
```

Figure 5.2.3
Endpoint for getting the id of car

The coming endpoint at Figure 5.2.4 is used for being able to open images in a separate tab. I included this mostly for debugging purposes as I had difficulties when inserting images. What it does is calls images from the Car_Images table and the image id and the car id. If the image is null it returns not found and finally. It returns the image as a file by the data and the extension of the file could be .jpg or .png.

```
[HttpGet("{carid}/image/{imageId}")]
0 references
public IActionResult GetImage(int carid, int imageId)
{
    var image = _db.Car_Images
        .FirstOrDefault(i => i.Image_ID == imageId && i.Car_ID == carid);

    if (image == null) return NotFound();

    return File(image.Image_Data, image.Content_Type);
}
```

Figure 5.2.4
Endpoint for getting image id based on the car id

This endpoint Figure 5.2.5 is responsible for creating the car listing. It does that by sending a POST request to the database meaning creating a new row of data. It first gets the logged in user id from the http session cookie as userId. If it equalizes to null it means that the user isn't authenticated. After that it equalizes the car User ID field of the database to the userId value of the logged in user. This is used so that each listed car has a user id in order to denote the user who posted it. After that an if check is there to check if any of the form data is empty. After that it checks for the uploaded images and sees if the image doesn't have any bytes in it or if no image is added at all and pops the message for one image is required

```
public async Task<ActionResult<Cars_Sale>> CreateCar([FromForm] Cars_Sale car, [FromForm] List<IFormFile> images)
{
    var userId = HttpContext.Session.GetInt32("UserId");

    if (userId == null)
    {
        return Unauthorized("User not authenticated.");
    }

    car.User_ID = userId.Value;

    if (!ModelState.IsValid)
    {
        return BadRequest(new
        {
            message = "Invalid form data",
            errors = ModelState.Values
                .SelectMany(v => v.Errors)
                .Select(e => e.ErrorMessage)
        });
    }

    if (images == null || !images.Any())
    {
        return BadRequest(new { message = "At least one image is required" });
    }

    var carImages = new List<Car_Image>();  //liste per imazhet
    foreach (var imageFile in images) //loop per cdo imazh upload
    {
        if (imageFile.Length > 0)
        {
            using var memoryStream = new MemoryStream();
            await imageFile.CopyToAsync(memoryStream);

            carImages.Add(new Car_Image
            {
                Image_Data = memoryStream.ToArray(),
                Content_Type = imageFile.ContentType,
                Is_Primary = carImages.Count == 0
            });
        }
    }

    car.Images = carImages;

    _db.Cars_Sale.Add(car);  //lidh application db context me metoden async qe te shtoje rresht tek tabela
    await _db.SaveChangesAsync(); //pret ndryshimin ne sql
    return CreatedAtAction(nameof(GetCar), new { ID = car.Car_ID }, car); //rikthen id te ruajtur
}
```

Figure 5.2.5
Endpoint for creating the car listing

Then the image logic starts, how it works is that it turns the images to raw binary data for storing in the database. The reason is because it is more efficient than storing the usual base64 as it takes less storage and is performance better.

The images are first stored in an empty List and then we have a for loop which loops through each uploaded image from images and if the byte length is higher then 0, so that no corrupted image is allowed. Then the memory stream is utilized by creating an in memory stream to hold file content.

Using ensures that the stream is automatically cleaned up after use. After that await imageFile. CopyToAsync(memoryStream), asynchronously copies the uploaded file's content into the memory stream. What that affects is that it reads the file's binary content. After that the images are added it adds a new Car_Image object to the list and then it holds the image binary data. It then turns the data into a byte array.

Finally it stores the MIME type, the extension name as well as making the first image uploaded the primary one by Is_Primary = carImages.Count == 0, while the other images have Is_Primary = false.

Summing everything up the database is called by adding a car and then awaits the database to save the changes and returns the saved ID of the car.

5.3: AUTH CONTROLLER

The auth controller at Figure 5.3.1 is responsible for registering and logging in as well as logging out. The same structure is evident as it starts [ApiController] and the new url for auth is /auth evident at [Route("api/auth")]. It has also the ApplicationDbContext attribute as well as a new attribute called PasswordHasher. This will be used later for hashing the password when registering. The first method is a POST request with the endpoint register. This will be used for creating new users. First it looks if all the required fields are filled. After that check it looks if the entered email already exists from the Users table, Email field. If a conflict occurs, meaning that the returned success is false then this email doesn't exist and is good to be installed. Then from the user form at the font it takes the Email and Phone field by the user and automatically fills the CreatedAt field. Finally it takes the password from the user as plain text and hashes it using the PasswordHasher attribute that was declared earlier. As for the hashing it uses PBKDF2 with HMAC-SHA256 internally, which is

```
[ApiController]
[Route("api/auth")]
1 reference
public class AuthController : Controller
{
    private readonly ApplicationDbContext _db;
    private readonly PasswordHasher<User> _passwordHasher;

    0 references
    public AuthController(ApplicationDbContext db)
    {
        _db = db;
        _passwordHasher = new PasswordHasher<User>();
    }

    [HttpPost("register")]
    0 references
    public async Task<IActionResult> Register([FromBody] User userInput)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        if (await _db.Users.AnyAsync(u => u.Email == userInput.Email))
        {
            return Conflict(new
            {
                success = false,
                message = "Email already exists",
                field = "email"
            });
        }

        var user = new User
        {
            Email = userInput.Email,
            Phone = userInput.Phone,
            CreatedAt = DateTime.UtcNow
        };

        // ben hash passwordin
        user.PasswordHash = _passwordHasher.HashPassword(user, userInput.PasswordHash);

        try
        {
            _db.Users.Add(user);
            await _db.SaveChangesAsync();

            return Ok(new
            {
                message = "Registration successful",
                userId = user.User_ID
            });
        }
        catch (Exception ex)
        {
            return StatusCode(500, new { message = ex.Message });
        }
    }
```

Figure 5.3.1
Endpoint for registering

stronger and designed specifically for password hashing. It is specifically created for this purpose by the ASP.NET framework. Finally there is a try catch with the filled data that tries to create a new user from the database. If everything goes well the Registration is successful and it returns the database-assigned ID of the newly registered user.

```
[HttpPost("login")]
0 references
public async Task<IActionResult> Login([FromBody] User userInput)
{
    var user = await _db.Users.FirstOrDefaultAsync(u => u.Email.ToLower() == userInput.Email.ToLower());

    if (user == null)
    {
        return Unauthorized(new
        {
            success = false,
            message = "Invalid email"
        });
    }

    var result = _passwordHasher.VerifyHashedPassword(user, user.PasswordHash, userInput.PasswordHash);
    if (result != PasswordVerificationResult.Success)
    {
        return Unauthorized(new { message = "Invalid password" });
    }

    HttpContext.Session.SetInt32("UserId", user.User_ID);

    return Ok(new
    {
        userId = user.User_ID,
        email = user.Email,
        phone = user.Phone
    });
}
```

Figure 5.3.2
Endpoint for logging in

At figure 5.3.2 is the login endpoint which is responsible for logging the registered user in. It first gets the list of users from the db it also turns the inputted email to lowercase as well the email from the db so that no the email stays consistent. If the email entered isn't found from the table it returns the same logic as registration as Invalid email. After that it checks the password. It hashes the entered password and verifies the hash with the other registered passwords. If the result doesn't match invalid password is thrown. A variable to save the user id as session http context. In the end it returns an ok message and returns user info as email, phone etc.

The final method is the logout which clears the user id and throws the logged out successfully message.

5.4: DIARY CONTROLLER

The final and most complex controller Figure 5.4.1 is the one responsible for all the actions at the diary car page. The structure is the same as before but now at [Route ("api/[controller]")] is a GET request from the Diary_Car table. It first verifies that the user id is stored in the HttpContext.Session as this is important. If no user id is found it throws the User not authenticated. When it is found it fetches the cars from the Diary_Car table based on the user id saved it fetches the car associated with it and the Maintenance_Records. Finally it returns the ok message.

```
[ApiController]
[Route("api/[controller]")]
1 reference
public class DiaryCarController : Controller
{
    private readonly ApplicationDbContext _db;

    0 references
    public DiaryCarController(ApplicationDbContext db)
    {
        _db = db;
    }

    [HttpGet]
    0 references
    public async Task<ActionResult<IEnumerable<Diary_Car>>> GetUserCars()
    {
        var userId = HttpContext.Session.GetInt32("UserId");
        if (userId == null)
        {
            return Unauthorized("User not authenticated.");
        }

        var cars = await _db.Diary_Car
            .Where(c => c.User_ID == userId)
            .Include(c => c.Maintenance_Record)
            .ToListAsync();

        return Ok(cars);
    }
```

Figure 5.4.1
Endpoint for fetching the car id based on the user id

At Figure 5.4.2 is a similar endpoint but the difference is that it has Get id. What this is used for is for later to edit the car data if entered incorrectly. It has the same logic all together, the main difference is that at the variable to fetch cars from the Diary_Car table it equalizes the User_Car_ID with the User_ID so that this info can be used later for other purposes and finally if the car found is null it return not found otherwise returns the car.

```
[HttpGet("{id}")]
1 reference
public async Task<ActionResult<Diary_Car>> GetCar(int id)
{
    var userId = HttpContext.Session.GetInt32("UserId");
    if (userId == null)
    {
        return Unauthorized("User not authenticated.");
    }

    var car = await _db.Diary_Car
        .Include(c => c.Maintenance_Record)
        .FirstOrDefaultAsync(c => c.User_Car_ID == id && c.User_ID == userId);

    if (car == null)
    {
        return NotFound();
    }
    return Ok(car);
}
```

Figure 5.4.2
Endpoint for getting id for editing

At Figure 5.4.3 is the endpoint for calling the car image. Different from the Car_Images table which its purpose is to handle multiple images. Here only one image per car is allowed. It fetches the table from Diary_car where it is fetched by the user id of the logged in user. After fetching it checks if the car image is null or no data is inside the file and sends the not found error. Instead it returns the file itself as well as the extension of the file.

```
[HttpGet("{id}/image")]
0 references
public IActionResult GetCarImage(int id)
{
    var car = _db.Diary_Car.FirstOrDefault(c => c.User_Car_ID == id);
    if (car == null || car.Car_Image == null || car.Car_Image.Length == 0)
    {
        return NotFound();
    }

    return File(car.Car_Image, car.Content_Type);
}
```

Figure 5.4.3
Endpoint for fetching the image based on the id of car

At figure 5.4.4 is the inserting the car logic. It first gets the logged in user id, if not User not authenticated is shown. Then the user id is assigned to the car User_ID itself. Then it checks whether the fields are valid. These are first made required by the front end. If something doesn't seem right Invalid form data is thrown. After that it checks for the image uploaded with the car. It first checks the Car_Image if it is null or if no data is present. Then using the same logic as the sell page where it saves it to a memory stream

and then to an array , saving as well the content type, the extension. It then adds the car and saves the changes to the database and returns the user id of the car.

```
[HttpPost]
public async Task<ActionResult<Diary_Car>> AddCar([FromForm] Diary_Car car, [FromForm] IFormFile Car_Image)
{
    var userId = HttpContext.Session.GetInt32("UserId");
    if (userId == null)
    {
        return Unauthorized("User not authenticated.");
    }

    car.User_ID = userId.Value;

    if (!ModelState.IsValid)
    {
        return BadRequest(new
        {
            message = "Invalid form data",
            errors = ModelState.Values
                .SelectMany(v => v.Errors)
                .Select(e => e.ErrorMessage)
        });
    }

    if (Car_Image != null && Car_Image.Length > 0)
    {
        using var memoryStream = new MemoryStream();
        await Car_Image.CopyToAsync(memoryStream);
        car.Car_Image = memoryStream.ToArray();
        car.Content_Type = Car_Image.ContentType;
    }
    else
    {
        car.Car_Image = Array.Empty<byte>();
        car.Content_Type = string.Empty;
    }

    _db.Diary_Car.Add(car);
    await _db.SaveChangesAsync();

    return CreatedAtAction(nameof(GetCar), new { id = car.User_Car_ID }, car);
}
```

Figure 5.4.4
Endpoint for adding the car image

At Figure 5.4.5 is the edit car details logic. How it works is similar to the add car logic. However an existingCar variable is declared, which first awaits the cars listed from the Diary_Car tables and takes the User_Car_ID of the car and then checks whether all the entities of the car are changed based on the form data entered at the front end. If any of the fields are changed it updates them to the database and saves changes. If something doesn't go right then an Error updating car is thrown.

```
[HttpPut("{id}")]
public async Task<ActionResult<Diary_Car>> UpdateCar(int id, [FromForm] Diary_Car carUpdate, [FromForm] IFormFile Car_Image)
{
    var userId = HttpContext.Session.GetInt32("UserId");
    if (userId == null)
    {
        return Unauthorized("User not authenticated.");
    }

    var existingCar = await _db.Diary_Car
        .FirstOrDefaultAsync(c => c.User_Car_ID == id && c.User_ID == userId);

    if (existingCar == null)
    {
        return NotFound();
    }

    existingCar.Nickname_Car = carUpdate.Nickname_Car ?? existingCar.Nickname_Car;
    existingCar.VIN = carUpdate.VIN ?? existingCar.VIN;
    existingCar.Brand = carUpdate.Brand ?? existingCar.Brand;
    existingCar.Model = carUpdate.Model ?? existingCar.Model;
    existingCar.License_Plate = carUpdate.License_Plate ?? existingCar.License_Plate;
    existingCar.Current_Kilometers = carUpdate.Current_Kilometers;
    existingCar.Fuel = carUpdate.Fuel ?? existingCar.Fuel;
    existingCar.Insurance_Expiry = carUpdate.Insurance_Expiry;
    existingCar.Inspection_Expiry = carUpdate.Inspection_Expiry;

    if (Car_Image != null && Car_Image.Length > 0)
    {
        using var memoryStream = new MemoryStream();
        await Car_Image.CopyToAsync(memoryStream);
        existingCar.Car_Image = memoryStream.ToArray();
        existingCar.Content_Type = Car_Image.ContentType;
    }


    try
    {
        await _db.SaveChangesAsync();
        return Ok(existingCar);
    }
    catch (DbUpdateException ex)
    {
        return StatusCode(500, "Error updating car: " + ex.Message);
```

Figure 5.4.5
Endpoint for editing car details

At Figure 5.4.6 is the delete logic of the car. Similar to the previous ones it first gets the user id from the database based on the car and after that the remove command is given and saves the changes to the database. The car part is finished now moving to the maintenance records. They are saved on a separate table that is connected to the car id.

```
[HttpDelete("{id}")]
0 references
public async Task<IActionResult> DeleteCar(int id)
{
    var userId = HttpContext.Session.GetInt32("UserId");
    if (userId == null)
    {
        return Unauthorized("User not authenticated.");
    }

    var car = await _db.Diary_Car
        .FirstOrDefaultAsync(c => c.User_Car_ID == id && c.User_ID == userId);

    if (car == null)
    {
        return NotFound();
    }

    _db.Diary_Car.Remove(car);
    await _db.SaveChangesAsync();

    return NoContent();
}
```

Figure 5.4.6
Endpoint for deleting diary car

At Figure 5.4.7 is the GET request which fetches the records itself based on the car id. It again takes the user id and does the checking if logged in or not. After that it fetches the car id and checks again if it exists. Then it gets the record from the table based on the previous fetched id and sorts them in descending order. If everything goes well it returns the records.

```
[HttpGet("{carId}/maintenance")]
1 reference
public async Task<ActionResult<IEnumerable<Maintenance_Record>>> GetMaintenanceRecords(int carId)
{
    var userId = HttpContext.Session.GetInt32("UserId");
    if (userId == null)
    {
        return Unauthorized("User not authenticated.");
    }

    var carExists = await _db.Diary_Car
        .AnyAsync(c => c.User_Car_ID == carId && c.User_ID == userId);

    if (!carExists)
    {
        return NotFound("Car not found");
    }

    var records = await _db.Maintenance_Record
        .Where(r => r.User_Car_ID == carId)
        .OrderByDescending(r => r.Record_Date)
        .ToListAsync();

    return Ok(records);
}
```

Figure 5.4.7
Endpoint for getting maintenance based on car id

At Figure 5.4.8 is the adding maintenance logic through a POST request. As previously mentioned, first gets the user id, then it gets the car id. After that it only adds two fields from here. The user id and the Record_Date the rest are taken from the frontend as they are sent a json request.

```
[HttpPost("{carId}/maintenance")]
0 references
public async Task<ActionResult<Maintenance_Record>> AddMaintenanceRecord(int carId, [FromBody] Maintenan
{
    var userId = HttpContext.Session.GetInt32("UserId");
    if (userId == null)
    {
        return Unauthorized("User not authenticated.");
    }

    var car = await _db.Diary_Car
        .FirstOrDefaultAsync(c => c.User_Car_ID == carId && c.User_ID == userId);

    if (car == null)
    {
        return NotFound("Car not found");
    }

    record.User_Car_ID = carId;
    record.Record_Date = DateTime.UtcNow;

    _db.Maintenance_Record.Add(record);
    await _db.SaveChangesAsync();

    return CreatedAtAction(nameof(GetMaintenanceRecords), new { carId = carId }, record);
}
```

Figure 5.4.8
Endpoint for inserting maintenance based on car id

At Figure 5.4.9 is the front sending the request to the api as well as the json request.

```
const url = editingMaintenance
    ? `/api/diarycar/${selectedCar.User_Car_ID}/maintenance/${editingMaintenance.Record_ID}`
    : `/api/diarycar/${selectedCar.User_Car_ID}/maintenance`;
const method = editingMaintenance ? "PUT" : "POST";


const response = await fetch(url, {
    method,
    headers: { 'Content-Type': 'application/json' },
    credentials: 'include',
    body: JSON.stringify(maintenanceData)
});
```

Figure 5.4.9
Endpoint for editing maintenance based on car id

At Figure 5.4.10 is the editing maintenance record logic. The same as the editing car logic but here the user can only edit the Kilometers, Maintenance Type, Next Maintenance Kilometers and Notes. Not the maintenance id and the date created the maintenance. After that it saves the changes to the database and returns the newly updated record.

```
[HttpPut("{carId}/maintenance/{recordId}")]
0 references
public async Task<ActionResult<Maintenance_Record>> UpdateMaintenanceRecord(int carId, int recordId, [From
{
    var userId = HttpContext.Session.GetInt32("UserId");
    if (userId == null)
    {
        return Unauthorized("User not authenticated.");
    }

    var existingRecord = await _db.Maintenance_Record
        .Include(r => r.Car)
        .FirstOrDefaultAsync(r => r.Record_ID == recordId &&
                                  r.User_Car_ID == carId &&
                                  r.Car.User_ID == userId);

    if (existingRecord == null)
    {
        return NotFound("Record not found");
    }

    existingRecord.Kilometers = recordUpdate.Kilometers;
    existingRecord.Maintenance_Type = recordUpdate.Maintenance_Type;
    existingRecord.Next_Maintenance_Km = recordUpdate.Next_Maintenance_Km;
    existingRecord.Notes = recordUpdate.Notes;

    await _db.SaveChangesAsync();
    return Ok(existingRecord);
}
```

Figure 5.4.10
Endpoint for editing maintenance based on car id

At Figure 5.4.11 is the final endpoint of the controller which is responsible for deleting the maintenance records. It does this by getting the id of the logged in user, which then it gets the id of the car and then the id of the record. After it finds everything it gives the Remove command to the database.

```
[HttpDelete("{carId}/maintenance/{recordId}")]
0 references
public async Task<IActionResult> DeleteMaintenanceRecord(int carId, int recordId)
{
    var userId = HttpContext.Session.GetInt32("UserId");
    if (userId == null)
    {
        return Unauthorized("User not authenticated.");
    }

    var record = await _db.Maintenance_Record
        .Include(r => r.Car)
        .FirstOrDefaultAsync(r => r.Record_ID == recordId &&
                                  r.User_Car_ID == carId &&
                                  r.Car.User_ID == userId);

    if (record == null)
    {
        return NotFound("Record not found");
    }

    _db.Maintenance_Record.Remove(record);
    await _db.SaveChangesAsync();

    return NoContent();
}
```

Figure 5.4.11
Endpoint for deleting maintenance based on car id

Chapter VI: DATABASE ENGINE

Working with ASP.NET and Visual Studio 2022 IDE using Microsoft SQL Server seemed the right choice because of its deep integration. Microsoft SQL Server is a comprehensive database platform that encompasses a wide array of components beyond just the core database engine. These include services such as Analysis Services, Reporting Services, Graph Databases, and Machine Learning Services. It is a popular choice for developers and organizations building business-oriented applications for several reasons.

First, SQL Server is deeply integrated with the Windows operating system, making it a natural and cost-effective option for Windows-based environments that also includes Visual Studio 2022. Since the release of SQL Server 2017, Microsoft has expanded support to include other operating systems such as several Linux distributions such as : Debian, Red Hat and macOS.

The Database Engine itself stands out for its ease of use, offering a user-friendly Windows-style interface and a suite of tools that assist with object creation, application optimization, and system administration. In summary, SQL Server is more than just a traditional relational database. It functions as a complete platform capable of handling a variety of data types—from structured to unstructured and graph data—while also delivering robust tools for operational tasks and data analysis in enterprise environments.

(Petkovic, 2019)

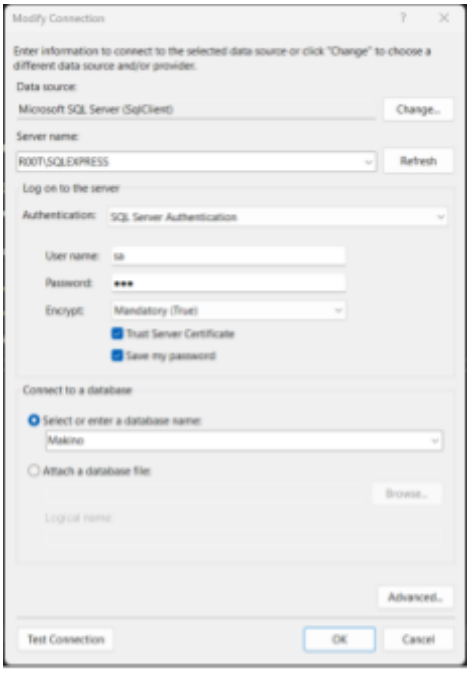


Figure 6.0.1
Database connection string

At Figure 6.0.1 is the connection string of the database engine.

6.1: DATABASE SCHEMATIC

At Figure 6.1.1 is the database schematic of the application, as seen everything circles around the User table. So it is first the User table. The User table has User_ID with int attribute and autoincrement enabled, which acts as primary key. Then there are the other

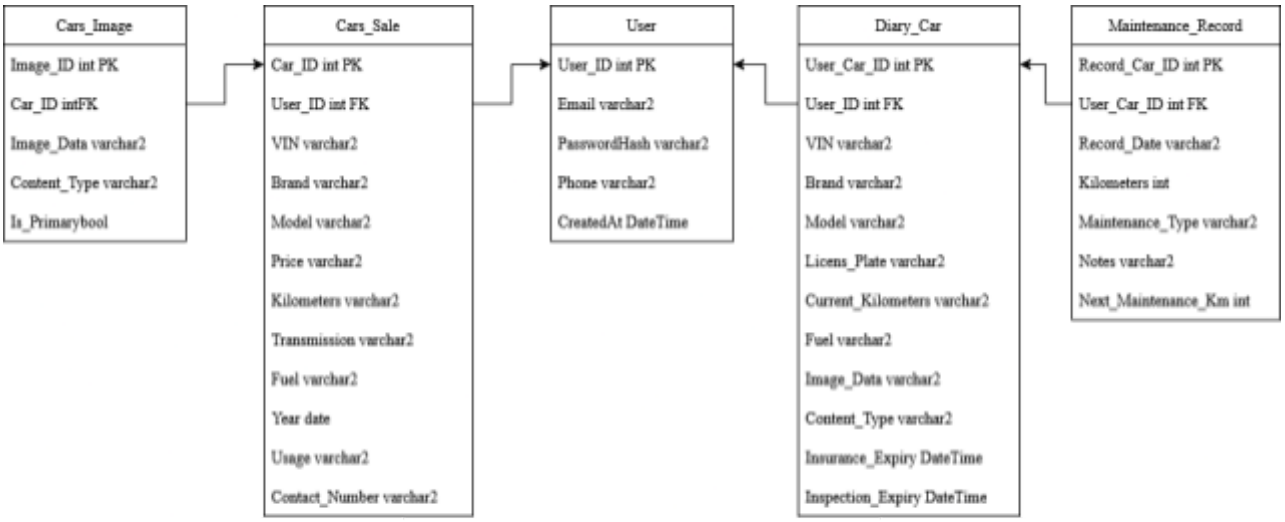


Figure 6.1.1
Database schemtaic

fields for the user which are Email, PasswordHash, Phone and CreatedAt. Each stores their respective data, specifically PasswordHash stores the entered password hashed through the backend password-hashing function mentioned before.

Around the User table on one side is the Cars_Sale table, responsible for storing the cars that users want to sell. Cars_Sale has Car_ID as int primary key with autoincrement, and User_ID as foreign key linking each record back to the seller in the User table. The VIN field holds the vehicle identification number, Brand and Model identify the make and model, Price contains the sale price, Kilometers shows how many kilometers the car has run, Transmission and Fuel describe its gearbox and fuel type, Year records its production year, Usage gives a short description of its condition, and Contact_Number stores the seller's phone number. All of these except for Car_ID, User_ID and Year are stored as varchar2.

On the opposite side of the User table is the Diary_Car table, which keeps track of cars that a user owns for personal reference rather than sale. Diary_Car uses User_Car_ID as its int primary key with auto increment and links back to the User table with User_ID. It also repeats VIN, Brand and Model to identify the vehicle, adds License_Plate for its registration number, Current_Kilometers to show how far it's been driven, Fuel to note its fuel type, and stores a thumbnail or full image in Image_Data along with its Content_Type. Insurance_Expiry_Date and Inspection_Expiry_Date are DateTime fields that record when the car's insurance and inspection are due.

Linked to Diary_Car by the User_Car_ID foreign key is the Maintenance_Record table, which logs every service or repair event. Maintenance_Record has Record_Car_ID as primary key, Record_Date as DateTime to mark when it happened, Kilometers as int to note the odometer reading at service time, Maintenance_Type and Notes as varchar2 to describe what was done and any comments, and Next_Maintenance_Km as int to suggest the odometer reading when the next maintenance should occur.

Finally, associated with Cars_Sale is the Cars_Image table, which holds all the photos for sale listings. Cars_Image has Image_ID as primary key, Car_ID as foreign key to tie it to a sale record, Image_Data to store the binary or string data array data as varchar2, Content_Type to record its MIME type, and Is_Primary as a boolean flag to indicate which image should appear first in the listing's gallery.

In conclusion this table is 3NF normalized. First Normal Form is satisfied. All the fields in the tables are atomic, meaning that there are no repeating groups or multivalued fields. Each column holds a single value per row. For example, fields like Kilometers, Fuel, Brand, and others contain one specific value and are not grouped or repeated in any way.

The Second Normal Form is also satisfied. All non-key attributes in each table depend on the entire primary key. Since each table uses a single-column primary key such as User_ID, Car_ID, or User_Car_ID, there are no partial dependencies. For instance, in

the Cars_Sale table, all the columns such as VIN, Brand, Price, and so on are directly dependent on the Car_ID, which acts as the primary key of the table.

Third Normal Form is likely satisfied as well. There are no visible transitive dependencies. Each non-key attribute depends only on the primary key and not on another non-key attribute. For example, in the Diary_Car table, fields like VIN, Brand, Model, Fuel, and even the image-related fields such as Image_Data and Content_Type all describe the car and are linked through the primary key User_Car_ID. However, a possible overlap exists between Diary_Car and Cars_Image, since both contain fields for image data and content type. If each car in Diary_Car is meant to store a personal image while Cars_Image stores listing images for Cars_Sale, then this design is still valid and not redundant. Otherwise, if the image data is shared and duplicated across tables, this would slightly break 3NF. But as long as the purposes are distinct, the current structure satisfies Third Normal Form.

6.2: ENTITY FRAMEWORK

ASP.NET offers the Entity Framework which allows the user from creating the entities otherwise called models to be created from inside the IDE and creating and changing the tables as needed. At Figure 6.1.2 is the NuGet package which enables the sql connection.

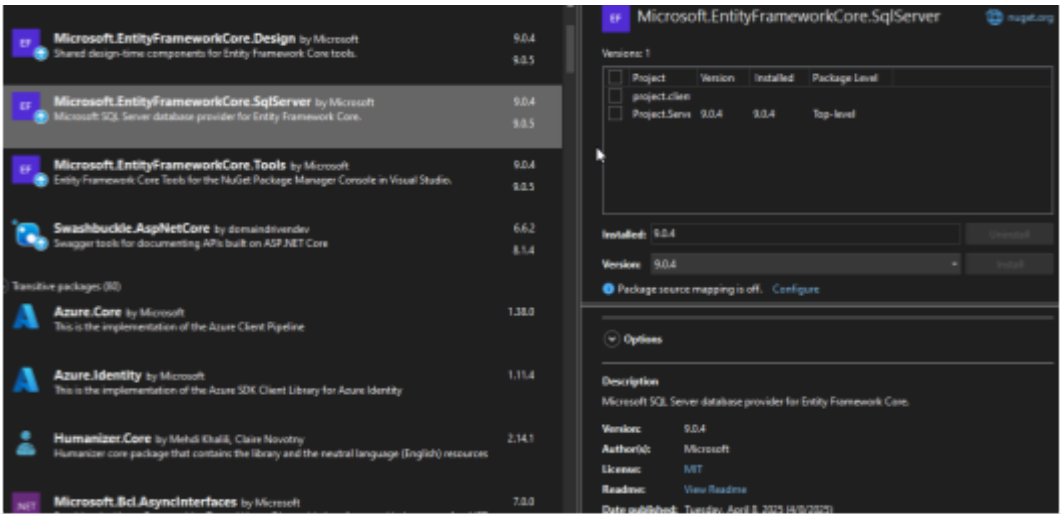


Figure 6.2.1
Package of Sql server from NuGet

After enabling the package alongside the entity framework package at Figure 3.2 I am able to create the entities as well as manage the relationships between them. At Figure 6.1.3 is the User entity. All fields mentioned before are here as well as some annotations, such as [Key] which tells the framework this is the primary key. Others are self explanatory, important are also the

```
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace Project.Server.Entities
{
    9 references
    public class User
    {
        [Key]
        3 references
        public int User_ID { get; set; }

        [Required]
        [EmailAddress]
        7 references
        public string Email { get; set; } = string.Empty;

        [Required]
        [StringLength(256)]
        4 references
        public string PasswordHash { get; set; } = string.Empty;

        3 references
        public string Phone { get; set; } = string.Empty;

        1 reference
        public ICollection<Cars_Sale> Cars { get; set; } = new List<Cars_Sale>();
        0 references
        public ICollection<Diary_Car> Diary_Car { get; set; } = [];
        1 reference
        public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
    }
}
```

Figure 6.2.2
User entity

ICollection at the bottom which are the other tables that this table has relation to.

At Figure 6.2.3 is the Cars_Sale table. An annotation which I didn't mention was the [Required] which makes the field required on the database. At the end is the [ForeignKey("User_ID")] which tells the framework that the field User_ID will act as the foreign key to the User table, which is declared a line below it. It has also a ICollection which is for the Car_Image table which holds the car images.

```
namespace Project.Server.Entities
{
    9 references
    public class Cars_Sale
    {
        [Key]
        2 references
        public int Car_ID { get; set; }

        [Required]
        0 references
        public string VIN { get; set; } = string.Empty;

        [Required]
        1 reference
        public string Brand { get; set; } = string.Empty;

        [Required]
        1 reference
        public string Model { get; set; } = string.Empty;

        [Required]
        2 references
        public int Price { get; set; }

        [Required]
        2 references
        public int Kilometers { get; set; }

        [Required]
        1 reference
        public string Transmission { get; set; } = string.Empty;

        [Required]
        1 reference
        public string Fuel { get; set; } = string.Empty;

        [Required]
        2 references
        public int Year { get; set; }

        0 references
        public string? Usage { get; set; }

        [Required]
        0 references
        public string Contact_Number { get; set; } = string.Empty;

        [ForeignKey("User_ID")]
        2 references
        public int User_ID { get; set; }
        1 reference
        public User? User { get; set; }

        4 references
        public ICollection<Car_Image> Images { get; set; } = new List<Car_Image>();
    }
```

Figure 6.2.3
Cars sale entity

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;

namespace Project.Server.Entities
{
    5 references
    public class Car_Image
    {
        [Key]
        1 reference
        public int Image_ID { get; set; }

        2 references
        public byte[] Image_Data { get; set; }
        2 references
        public string Content_Type { get; set; }

        [Required]
        1 reference
        public bool Is_Primary { get; set; }

        [ForeignKey("Car_ID")]
        2 references
        public int Car_ID { get; set; }
        [JsonIgnore]
        1 reference
        public Cars_Sale? Car { get; set; }
    }
}
```

Figure 6.2.4
Car Image entity

At Figure 6.2.4 is the Car_Image table which holds the images, and how it does that is by storing the data of the images as byte and then the extension at the content type field. It is connected to the Car_ID field of the Cars_Sale table.

At Figure 6.2.5 is the Diary_Car table it has the same logic and annotation as the Cars_Sale page however here is that the fields of image present in the Car_Image are here. The reason is because only one image is allowed per diary car. It has also the Maintenance_Records table declared which will be used by the next entity. User_ID field as previously is the foreign key to the User table.

```
namespace Project.Server.Entities
{
    10 references
    public class Diary_Car
    {
        [Key]
        7 references
        public int User_Car_ID { get; set; }
        10 references
        public int User_ID { get; set; }
        [ForeignKey(nameof(User_ID))]
        0 references
        public User? User { get; set; }
        3 references
        public string Nickname_Car { get; set; } = string.Empty;
        [Required]
        3 references
        public string VIN { get; set; } = string.Empty;
        [Required]
        3 references
        public string Brand { get; set; } = string.Empty;
        [Required]
        3 references
        public string Model { get; set; } = string.Empty;
        [Required]
        3 references
        public string License_Plate { get; set; } = string.Empty;
        [Required]
        2 references
        public int Current_Kilometers { get; set; }
        [Required]
        3 references
        public string Fuel { get; set; } = string.Empty;
        6 references
        public byte[]? Car_Image { get; set; }
        4 references
        public string? Content_Type { get; set; } = string.Empty;

        3 references
        public ICollection<Maintenance_Record> Maintenance_Record { get; set; }

        2 references
        public DateTime? Insurance_Expiry { get; set; }
        2 references
        public DateTime? Inspection_Expiry { get; set; }
    }
}
```

Figure 6.2.5
Diary car entity

```
namespace Project.Server.Entities
{
    8 references
    public class Maintenance_Record
    {
        [Key]
        2 references
        public int Record_ID { get; set; }
        5 references
        public int User_Car_ID { get; set; }
        [ForeignKey("User_Car_ID")]
        5 references
        public Diary_Car? Car { get; set; }

        [Required]
        2 references
        public DateTime Record_Date { get; set; } = DateTime.UtcNow;

        [Required]
        2 references
        public int Kilometers { get; set; }

        [Required]
        2 references
        public string Maintenance_Type { get; set; } = string.Empty;

        2 references
        public string Notes { get; set; } = string.Empty;

        [Required]
        2 references
        public int Next_Maintenance_Km { get; set; }
    }
}
```

Figure 6.2.6
Maintenance record entity

At Figure 6.2.6 is the final table and it is Maintenance_Records, and the foreign key field is the User_Car_ID which is connected to the Diary_Car table

Let's say there are changes that are done to the entities, in order for the framework to know it, the add-migration command needs to be executed along with the name of the migration in " ". Then the framework automatically generates the migration. At Figure 6.2.8 is an existing migration which has been generated. After it has been generated you are able to see all the lines and can pinpoint if there is an error and revoke the migration. If everything is fine the command update-database is executed and the migration is stored just like version control.

```
Package Manager Console
Package source: All        Default project: Project.Server
Package Manager Console Host Version 6.12.2.1

Type 'get-help NuGet' to see all available NuGet commands.

PM> add-migration "showcasethesis"
```

Figure 6.2.7
Add-migration

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "User",
        columns: table => new
        {
            User_ID = table.Column<int>(type: "int", nullable: false)
                .Annotation("SqlServer:Identity", "1, 1"),
            Name = table.Column<string>(type: "nvarchar(max)", nullable: false),
            Contact_number = table.Column<string>(type: "nvarchar(max)", nullable: false),
            Email = table.Column<string>(type: "nvarchar(max)", nullable: false),
            Password = table.Column<string>(type: "nvarchar(max)", nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_User", x => x.User_ID);
        });

    migrationBuilder.CreateTable(
        name: "Cars_Sale",
        columns: table => new
        {
            Car_ID = table.Column<int>(type: "int", nullable: false)
                .Annotation("SqlServer:Identity", "1, 1"),
            VIN = table.Column<string>(type: "nvarchar(max)", nullable: false),
            Brand = table.Column<string>(type: "nvarchar(max)", nullable: false),
            Model = table.Column<string>(type: "nvarchar(max)", nullable: false),
            Price = table.Column<int>(type: "int", nullable: false),
            Kilometers = table.Column<int>(type: "int", nullable: false),
            Transmission = table.Column<string>(type: "nvarchar(max)", nullable: false),
            Fuel = table.Column<string>(type: "nvarchar(max)", nullable: false),
            Year = table.Column<int>(type: "int", nullable: false),
            Usage = table.Column<string>(type: "nvarchar(max)", nullable: true),
            User_ID = table.Column<int>(type: "int", nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Cars_Sale", x => x.Car_ID);
            table.ForeignKey(
                name: "FK_Cars_Sale_User_User_ID",
                column: x => x.User_ID,
                principalTable: "User",
                principalColumn: "User_ID",
                onDelete: ReferentialAction.Cascade);
        });
```

Figure 6.2.8
Migration generation

At Figure 6.2.9 are all the migrations that are done to the project along with the comment on what each does.

```
Migrations
    20250511094457_Car_SalesCar_ImagesAndUser.cs
    20250513212510_numberpercar.cs
    20250513213306_jsonignore.cs
    20250515233009_storeimages.cs
    20250520213548_permiresim table users.cs
    20250520215413_permiresim table usrs dhe shtim i user
    20250520215542_permiresim i table users dhe rikrijim i c
    20250520215907_ri krijim i gjithe tables te db me user id
    20250521222138_rritjepasswordhashlength.cs
    20250521224130_removereqphone.cs
    20250522121312_diarydhemaintetancerecordsdhendrysl
    20250522202846_nicknamediarycar.cs
    20250522214027_optionalcardiaryimage.cs
    ApplicationDbContextModelSnapshot.cs
```

Figure 6.2.9
List of migrations generated

Chapter VII: DEPLOYMENT

During my internship I created a self-hosted server where I am able to self-host multiple applications. This project is hosted there, inside a virtual machine. How it works is the application needs to be containerized so that each of the three components front, back and database. A technology which is king in containerization is Docker, the open-source container engine, running the world. So first, the dockerfiles are needed for each component, first of react. Then the second dockerfile for the .NET Backend. Then third for the database for the SQL Server. The dockerfile basically works as the executing commands to the terminal, but automated and executed by Docker itself so that it can do the execution of them on setup. All the three parts will have to be published as images in order to be fetched later when going to the server. So the font and back need to be built specifically and published as images in the Docker Repository. The database doesn't need as the image is ready, just the data needs to be filled and tables created accordingly.

After publishing each image needs to be grouped together to be called somewhere. Docker Compose does this, and a docker-compose.yml file is the file responsible which tells Docker, the services, the apps, what Dockerfile to execute as well as which port to be opened.

Thankfully, Visual Studio IDE also supports docker templates and orchestration and generated for me the Dockerfile and the orchestration file. I choose the Linux environment as that is the final destination of the app. At Figure 7.0.1 is the Dockerfile, starting from the top is stage 1 base, it declares aspnet:8.0 as base, meaning that it is a stage named base that is using ASP.NET Core 8.0 runtime, then declares that the container is going to run as a non-root user. Then declares the working directory as /app meaning inside the container. Finally it exposes 2 ports 8080 for the backend API and 8081 for the frontend dev server which is Vite.

Stage 2 is build, now a new stage name is created named build which is using the .NET SDK image. Defines a build-time variable build_configuration with value Release, used later for dotnet build and publish. Then changes the working directory inside this build

stage to /src. Then copies only the .csproj and .esproj files into the container so that dotnet restore can resolve dependencies faster before copying everything. Then Run dotnet restore NuGet packages for the server project. This is done before copying all the source code to use Docker's cache effectively. Copy . . copies everything from your solution folder (where the Dockerfile is) into /src. Then the working directory is changed to where the main .csproj file is. Finally the server project is built in the given

```
# See https://aka.ms/customizecontainer to learn how to customize your debug container and how Visual Studio

# This stage is used when running from VS in fast mode (Default for Debug configuration)
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
USER $APP_UID
WORKDIR /app
EXPOSE 8080
EXPOSE 8081


# This stage is used to build the service project
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY ["Project.Server/Project.Server.csproj", "Project.Server/"]
COPY ["project.client/project.client.esproj", "project.client/"]
RUN dotnet restore "./Project.Server/Project.Server.csproj"
COPY . .
WORKDIR "/src/Project.Server"
RUN dotnet build "./Project.Server.csproj" -c $BUILD_CONFIGURATION -o /app/build

# This stage is used to publish the service project to be copied to the final stage
FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "./Project.Server.csproj" -c $BUILD_CONFIGURATION -o /app/publish /p:UseAppHost=false

# This stage is used in production or when running from VS in regular mode (Default when not using the Debug
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Project.Server.dll"]
```

Figure 7.0.1
Dockerfile generated by the IDE

configuration, and outputs binaries to /app/build.

Stage 3 publish, a new stage is started from the previous build called publish. Then a redeclaration of the build_configuration is done in this stage too. Finally it publishes the app to /app/publish, which creates an optimized self-contained version. UseAppHost= false avoids copying platform-specific hosting binaries which are useful for Docker.

Stage 4 final, starts the final stage from the base image, then ensures that it is working from the /app directory again. Copies the published output from the publish stage into this final container. Sets the command that starts the app when the container runs.
In this case, it runs the compiled ASP.NET server with dotnet Project.Server.dll.

Now the image needs to be built and pushed to the repository, this can be achieved by using the command docker build -t makino:latest . which builds the project in the root of the folder with the tag name makino and then pushed to the repo with docker push Doodle1Noodle/makino:latest. Now the image can be pulled from anywhere.

```
version: "3.9"

services:
  makino-api:
    image: Doodle1Noodle/makino:latest
    depends_on:
      - sqlserver
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ConnectionStrings__DefaultConnection=Server=sqlserver;Database=Makino;User=sa;Password=123;TrustServerCertificate=True
    ports:
      - "8000:8080"
    networks:
      - makino-net

  sqlserver:
    image: mcr.microsoft.com/mssql/server:2022-lts
    environment:
      - ACCEPT_EULA=Y
      - SA_PASSWORD=123
    ports:
      - "1433:1433"
    volumes:
      - sqlvolume:/var/opt/mssql
    networks:
      - makino-net

volumes:
  sqlvolume:

networks:
  makino-net:
    driver: bridge
```

Figure 7.0.2
Docker compose file which pulls the image of makino and mssql

Going to the server the docker compose file at Figure 7.0.2 needs to be imported like a file or just copy-paste and run docker-compose up. Docker then will automatically pull the image of the project, pull sql server and run dockerfile as well as the environment commands to sql server to be connected. Then 2 containers are created each for respective roles and the application locally is accessible at http:127.0.0.1:8080. The server is running CloudFlare reverse proxy, cloudflared tunnels zero trust service, which allows to give the application a subdomain makino Figure 7.0.3 and where to get the service from. In this case I gave https since .net 8 is already preconfigured for https and the ip of the vm that is running on. Through the Cloud Flare network the application is also secured as the TLS certificate is provided by them making the connection secure.

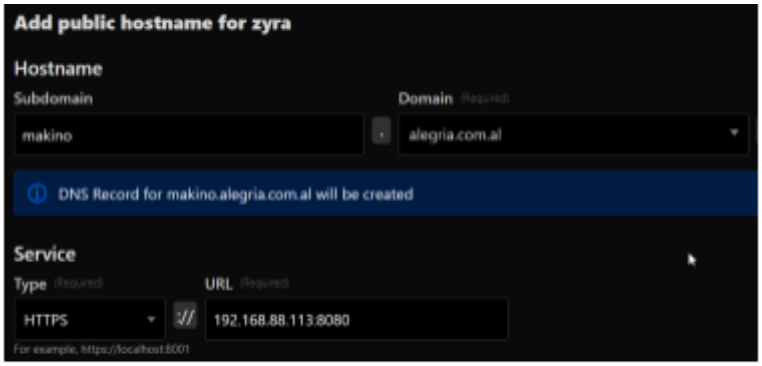


Figure 7.0.3
Cloudflare tunnel

Conclusions

This project concludes with a working car marketplace web app tailored to the Albanian market. The app allows users to list their vehicles, search for available cars, and also offers a personal diary feature for managing owned cars. The frontend was built in React, the backend in .NET, and the database in Microsoft SQL Server. It covers the essential needs of both buyers and sellers in a simple and efficient way. Looking at the current Albanian market, most transactions happen via word of mouth or through limited platforms like Instagram or merrjep.al, which are bloated, confusing, and not specialized in cars. Other existing marketplaces are forgettable, look generic, and have no real standout features. Most of them just dump listings without offering a structured experience or any kind of user retention. Also, their tech stack doesn't help scalability or make them future-proof.

My app tries to solve that with three clear goals: it must look good, be solid in tech, and offer something different. The branding and name Makino were built from scratch with this in mind, something that's both unique and easily recognizable. The backend is done in ASP.NET for reliability and future development, and the frontend in React for speed and flexibility. The diary feature is what really makes this app different. A structured way to track car maintenance, from oil changes to insurance renewals. No more sticky notes in the glovebox or lost phone reminders. This feature is meant to solve something all car owners face but no one has built properly until now.

For now, income will come through banner ads and later from promoted listings. There's no in-app chat or call hosting because it adds unnecessary complexity. The system simply connects people, and from there, the deal is done face-to-face, which fits perfectly with how things already work in Albania. All in all, the app is functional, targeted, and scalable. It offers more than just a car listing platform – it offers a full experience that aligns with how Albanians actually buy and manage their cars.

Bibliography

INSTAT. (2025a). Statistikat e transportit rrugor, tremujori I, 2025 [Road transport statistics, Q1 2025]. https://www.instat.gov.al/media/15261/statistikat-e-transportit_tr-i-t%C3%AB_2025-rrugor-rishikuar.pdf

INSTAT. (2025b). Statistikat e transportit rrugor, tremujori I, 2024 [Road transport statistics, Q1 2024]. https://www.instat.gov.al/media/13232/statistikat-e-transportit_tr-i-r%C3%AB_2024-rrugor%C3%AB.pdf

Worldometer. (2025). Albania population. https://www.worldometers.info/world-population/albania-population/

Giretti, A. (2023). *Coding clean, reliable, and safe REST APIs with ASP.NET Core 8: Develop robust minimal APIs with .NET 8*. APress Publishing.

Petkovic, D. (2019). *Microsoft SQL Server 2019: A beginner's guide* (7th ed.). McGraw-Hill Education.

Brown, M. (2020). *Modern Web Development with React*. TechPress.

Schwarzmüller, M. (2023). *React key concepts* (2nd ed.). Packt Publishing.

Smith, T. (2018). JSX and React Explained. *Journal of Frontend Engineering, 12*(4), 21–30. https://doi.org/10.1000/jfe.2018.122