**UNIVERSITATEA DIN**

**BUCUREȘTI**

**FACULTATEA DE**

**MATEMATICĂ ȘI**

**INFORMATICĂ**

**SPECIALIZAREA INFORMATICĂ**

**Lucrare de licență**

**Platformă web a unui comerciant de autovehicule**

**Absolvent**

**Vasile George-Cristian**

**Coordonator științific**

**Lect. dr. Ana Cristina Iova**

**București, iulie 2023**

**Rezumat**

Tema lucrării constă în dezvoltarea unei aplicații web a cărui scop fiind gestionarea unui lanț de magazine al unui comerciant de autovehicule rulate. Dezvoltarea aplicației va necesita atât proiectarea unei interfețe grafice intuitive, cât și a unui Web API care să gestioneze logica necesară funcționalităților și a stocării datelor. Aplicația va fi folosită atât de angajații magazinului cât și de potențialii clienți.

Funcționalitățile dedicate angajaților vor include adăugarea și modificarea locațiilor din lanț, gestionarea autovehiculelor prezente în locații și vizualizarea programărilor făcute de clienți prin intermediul platformei.

Clienții vor putea viziona autovehiculele scoase la vânzare. După crearea unui cont, aceștia vor putea face programări prin intermediul platformei. La crearea contului, clienții vor avea opțiunea de a include date personale pe baza cărora aplicația le va recomanda vehicule.

**Abstract**

The thesis’ subject consists of developing a web application with the purpose of managing a used car dealership chain. The application’s development will require designing both an intuitive graphical interface and a Web API that will manage the algorithms necessary for providing the application’s functionality and data storage. The application will be used by store employees and potential clients.

Functionalities dedicated to employees will include adding and modifying chain locations, management of the vehicles present in the locations and viewing appointments made by clients through the platform.

The clients will be able to view vehicles up for sale. After creating an account, they will be able to book appointments through the platform. On account creation, the clients will have the option to include some personal info based on which the application will recommend vehicles.

**Cuprins**

[I Introducere 5](#_Toc136791102)

[1.1 Prezentarea generală a aplicației 5](#_Toc136791103)

[1.1.1 Descrierea backend-ului 5](#_Toc136791104)

[1.1.2 Descrierea frontend-ului 5](#_Toc136791105)

[1.2 Starea curentă a pieței auto 6](#_Toc136791106)

[1.3 Scopul și motivația alegerii temei 6](#_Toc136791107)

[1.4 Termeni 6](#_Toc136791108)

[1.5 Structura lucrării 7](#_Toc136791109)

II [Limbaje și tehnologii 8](#_Toc136791110)

[2.1 Visual Studio Community 8](#_Toc136791111)

[2.1.1 Instalare 8](#_Toc136791112)

[2.1.2 Pachete necesare 9](#_Toc136791113)

[2.2 ASP.NET și C# 9](#_Toc136791114)

[2.3 Microsoft SQL Server și SSMS 10](#_Toc136791115)

[2.4 Node.JS și React 11](#_Toc136791116)

[2.4.1 Pachete auxiliare 11](#_Toc136791117)

[2.5 Visual Studio Code 12](#_Toc136791118)

[2.6 TypeScript și SCSS 13](#_Toc136791119)

III [Implementarea Web API-ului 14](#_Toc136791120)

[3.1 Structura soluției 14](#_Toc136791121)

[3.1.1 Repository pattern 14](#_Toc136791122)

[3.1.2 Specification pattern 16](#_Toc136791123)

[3.1.3 Structura directoarelor 17](#_Toc136791124)

[3.2 Baza de date 17](#_Toc136791125)

[3.3 Securitatea 19](#_Toc136791126)

[3.3.1 Stocarea informațiilor sensibile 19](#_Toc136791127)

[3.3.2 Autentificarea 19](#_Toc136791128)

[3.3.3 Vulnerabilități 21](#_Toc136791129)

[3.4 Controllere și funcționalitate 21](#_Toc136791130)

[3.4.1 Trimiterea de email-uri 22](#_Toc136791131)

[3.4.2 Paginația 22](#_Toc136791132)

[3.4.3 Filtre și sortări 23](#_Toc136791133)

[3.4.4 Expunerea intervalelor disponibile 24](#_Toc136791134)

[3.4.5 Algoritmul de recomandare 25](#_Toc136791135)

IV [Implementarea paginii web 27](#_Toc136791136)

[4.1 Structura proiectului 27](#_Toc136791137)

[4.2 Tehnici de îmbunătățire a experienței utilizatorilor 28](#_Toc136791138)

[4.2.1 Persistența datelor 28](#_Toc136791139)

[4.2.2 Feedback 29](#_Toc136791140)

[4.2.3 Timpi de încărcare 30](#_Toc136791141)

[4.3 Prezentarea interfeței 31](#_Toc136791142)

[4.3.1 Secțiunea de autentificare 31](#_Toc136791143)

[4.3.2 Lista de autovehicule 33](#_Toc136791144)

[4.3.3 Detaliile unui autovehicul 35](#_Toc136791145)

V [Concluzii 38](#_Toc136791146)

**Capitolul I**

# Introducere

## Prezentarea generală a aplicației

Asemenea multor alte aplicații web, aplicația este alcătuită din două componente interconectate. Prima componentă este un Web API – denumit în continuare backend, iar a doua componentă este o pagină web – denumită în continuare frontend. Frontend-ul comunică cu backend-ul prin intermediul unor apeluri de tip REST. Atât textul din interfața vizuală cât și totalitatea codului sunt scrise în limba engleză.

### Descrierea backend-ului

Backend-ul a fost realizat în framework-ul ASP.NET, fiind scris în limbajul C#. Acesta interacționează cu o bază de date SQL Server Express. Baza de date nu este creată folosind comenzi SQL, ci este generată printr-o abordare „code first” prin intermediul Entity Framework. Codul este structurat în conformitate cu „Repository Pattern” și „Specification Pattern”. Fiind un Web API, acesta își expune metodele de tip REST prin intermediul unor endpoint-uri din controllere.

### Descrierea frontend-ului

Frontend-ul este realizat cu ajutorul librăriei React și este scris în limbajul Typescript. Acesta își expune funcționalitățile în funcție de starea utilizatorului (autentificat/ neautentificat) și rolul acestuia. Comunicarea cu backend-ul se face prin apelarea metodelor REST menționate în secțiunea 1.1.1.

Secțiunea de autentificare – „auth” – este împărțită în 4 pagini: o pagină de autentificare, una de înregistrare, una pentru confirmarea prin email a unui cont nou și una pentru resetarea parolei unui cont existent.

Secțiunea principală – „main” – pune la dispoziție o pagină în care poate fi vizionată și filtrată lista de autovehicule și o pagină secundară care pune la dispoziție detalii amănunțite despre un vehicul selectat din listă. Funcționalitățile vizibile doar administratorilor sunt expuse pe pagina cu lista de autovehicule prin intermediul unor formulare de tip „Dialog”.

## 1.2 Starea curentă a pieței auto

De la apariția autovehiculelor până în trecutul apropiat piața a rămas in mare parte neschimbată, fiind dominată de motoarele ce ard combustibili fosili. În prezent, anumiți factori sociopolitici au demarat începuturile „electrificării” autovehiculelor, împingând dezvoltarea de tehnologii noi în domeniul auto. Această schimbare rapidă a pieței de la stabilitate la volatilitate a cauzat multe platforme de acest tip să nu poată ține pasul cu noile tehnologii.

## Scopul și motivația alegerii temei

Domeniul dezvoltării de aplicații web este unul care mă atrage deoarece necesită atât o aprofundare a metodelor de stocare și manipulare a datelor în backend, cât și proiectarea unei interfețe vizuale atractive în frontend, buna implementare a ambelor componente fiind crucială în realizarea unei platforme de succes.

Am ales această temă deoarece sunt nemulțumit de experiențele mele anterioare cu platforme de acest tip. Câteva probleme pe care îmi doresc să le rezolv sunt lipsa algoritmilor de recomandare satisfăcători și inabilitatea de a filtra conținutul pe baza unor atribute mai specifice. În plus, lucrarea își propune abordarea problemelor de scalabilitate cauzate de schimbările menționate anterior, în subcapitolul 1.2.

## Termeni

Voi folosi acest subcapitol pentru explicarea termenilor folosiți în limba engleză deoarece nu au neapărat un sinonim direct în limba română.

* Callback: o funcție pasată ca argument unei alte funcții
* Debugging: procesul de a depista și rezolva erorile dintr-un program
* Endpoint: un nod ce semnalează un punct de acces unde se pot face request-uri
* Enum: o enumerație a componentelor unui set cărora li se atribuie valori
* Framework: o implementare mai abstractă ce ca are scop de a sta la baza unei implementări mai specifice
* Open-source: descrie o aplicație sau un program cu codul disponibil oricui
* Request: o solicitare făcută de către un client către un endpoint
* Repository: o clasă ce expune doar metode de manipulare ale datelor (CRUD)
* Token: șir de caractere utilizat pentru autentificare, cu scopul secundar de a păstra sesiunea utilizatorului

## Structura lucrării

Lucrarea este structurată în cinci capitole.

Primul capitol conține o descriere succintă a aplicației, starea curentă a pieței pe care se bazează funcționalitatea, scopul lucrării și motivația alegerii temei. De asemenea, se explică pe scurt sensul folosit al cuvintelor din limba engleză.

În al doilea capitol sunt descrise atât limbajele, tehnologiile și programele utilizate în implementarea aplicației cât și motivația alegerii acestora.

Al treilea capitol se concentrează pe implementarea backend-ului, fiind prezentate noțiunile de securitate, structura logică a soluției, baza de date și funcționalitățile controllerelor.

Al patrulea capitol descrie pe scurt structura logică a implementării frontend-ului, scopul principal al acestuia fiind prezentarea interfeței vizuale și a metodelor de îmbunătățire a experienței utilizatorilor.

În al cincilea capitol sunt descrise concluziile, vorbindu-se despre experiența utilizării tehnologiilor, modul rezolvării problemelor descrise în acest capitol și conceptele învățate pe parcursul dezvoltării aplicației.

**Capitolul II**

# Limbaje și tehnologii

## Visual Studio Community

Visual Studio este un IDE – „Integrated Development Environment”, fiind și singurul care integrează dezvoltarea aplicațiilor pe bază de .NET. Acesta conține multe unelte care facilitează dezvoltarea aplicațiilor, printre care se numără un debugg er foarte detaliat ce permite chiar vizualizarea progresului la nivel de fir de execuție. Aplicațiile web pot primi un număr ridicat de cereri simultan așa că este vitală utilizarea firelor de execuție. Începând cu versiunea 2022, Visual Studio integrează și „Github Copilot”, o unealtă bazată pe inteligența artificială ce poate genera cod automat sau pe baza unei descrieri. Versiunea „Community” incorporează majoritatea funcționalităților și este gratuită pentru studenți.

### Instalare

Versiunea „Community” 2022 poate fi obținută de pe pagina oficială Microsoft, la adresa <https://visualstudio.microsoft.com/vs/community/>.[1] Pentru a putea dezvolta o aplicație pe baza ASP.NET trebuie instalată și componenta „ASP.NET and web development” din secțiunea „Web & Cloud”, ilustrată în figura 2.1.

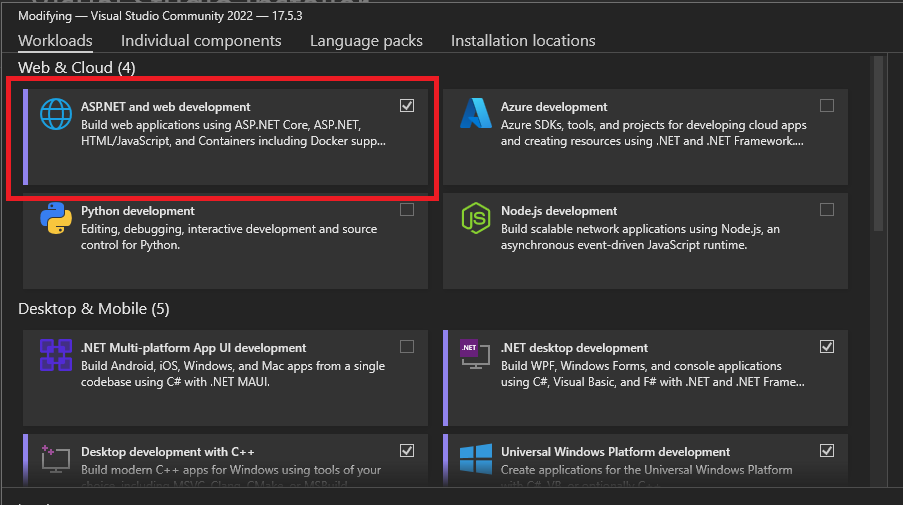


Figura 2.1 – Componenta „ASP.NET and web development”

### Pachete necesare

Un alt punct forte al Visual Studio este abilitatea de a adăuga pachete care să extindă funcționalitatea de bază din .NET. În interfața Visual Studio, navigând în panglică pe calea „Tools -> NuGet Package Manager -> Manage NuGet Packages for Solution…” pot fi gestionate pachetele NuGet instalate. Nu este necesar să instalăm aceste pachete manual, fiind descărcate automat la prima rulare a proiectului. În cadrul acestui proiect sunt folosite șase pachete NuGet.

Microsoft.AspNetCore.Authentication.JwtBearer permite folosirea de tokeni JWT pentru autentificare.

Microsoft.AspNetCore.Identity.EntityFrameworkCore este folosit pentru a adăuga funcționalitatea ce se gestionează conturile de utilizator la EntityFramework, generând automat tabelele necesare în baza de date în funcție de configurația făcută.

Microsoft.AspNetCore.Mvc.NewtonsoftJson adaugă abilitatea de a trimite și primi informații în formatul JSON prin intermediul endpoint-urilor din controllere.

Microsoft.EntityFrameworkCore.SqlServer conține logica necesară pentru a face legătura între EntityFramework și o bază de date de tipul Microsoft SQL Server.

Microsoft.EntityFrameworkCore.Tools adaugă comenzi utilitare, cum ar fi „Scaffold-DbContext” ce generează automat clase în C# pe baza tabelelor din baza de date sau comenzile inverse „Add-Migration” și „Update-Database”, prima generând automat o „Migrație” ce presupune operații de adăugare și modificare a tabelelor din baza de date în funcție de entitățile descrise prin clase de C# iar a doua rulând aceste comenzi pentru a actualiza baza de date.

Swashbuckle.AspNetCore permite documentarea endpoint-urilor și testarea acestora dintr-o interfață ce se deschide în interiorul unui navigator web.

## ASP.NET și C#

ASP.NET este un web framework dezvoltat de către Microsoft ce are ca scop facilitarea creări aplicațiilor web. Fiind parte din ecosistemul .NET, o aplicație dezvoltată cu ajutorul acestuia poate rula atât pe Windows, cât și pe MAC OS și Linux. Visual Studio conține două șabloane pentru crearea unui proiect de tip Web API, ilustrate în figura 2.3 aflată la sfârșitul acestui subcapitol.

Pentru a putea rula un Web API bazat pe ASP.NET în versiunea de producție există 2 opțiuni, distribuirea în variantă „Framework-dependent” și distribuirea în variantă “Self-contained”, precum se poate vedea în figura 2.2. Varianta „Framework-dependent” necesită ca versiunea țintă de .NET să fie instalată pe mașina care încearcă să îl ruleze. Opțiunea „Self-Contained” va cauza ca toate librăriile necesare rulării aplicației să fie incluse în directorul final.

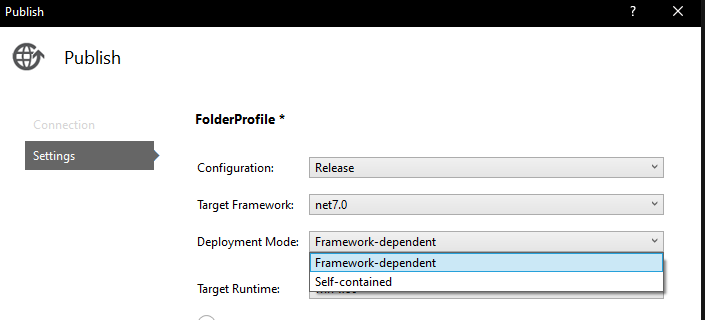
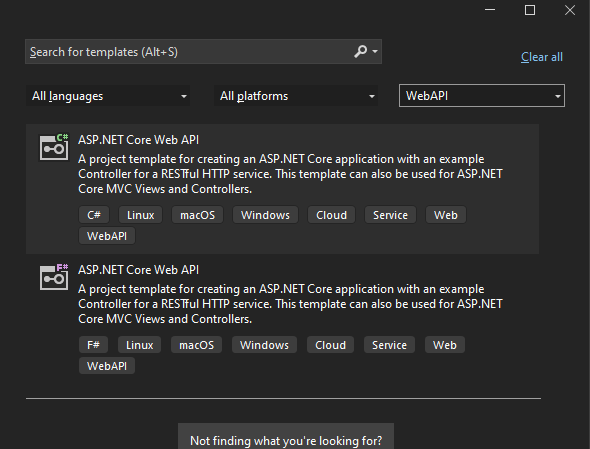
C# este un limbaj de programare orientat pe obiecte cu sintaxa similară limbajului Java, fiind unul dintre cele mai populare limbaje la momentul actual. Am ales acest limbaj în favoarea lui F# deoarece Microsoft pune la dispoziție mai multe resurse de învățare în documentația oficială.

Figura 2.3 – Șabloanele în limbajele C# și F#

Figura 2.2 – Alegerea unui profil de distribuire

## Microsoft SQL Server și SSMS

Entity Framework este compatibil cu o multitudine de baze de date printre care se numără SQL Server, Azure Cosmos DB și Maria DB.[2] Am ales Microsoft SQL Server deoarece, fiind dezvoltat tot de Microsoft, are o compatibilitate ridicată cu ecosistemul .NET. Versiunea „Express” este gratuită și consumă mai puține resurse deoarece nu include toate funcționalitățile care nu sunt necesare dezvoltării unei baze de date „code-first”. Ultima versiune stabilă poate fi descărcată de pe pagina oficială Microsoft la adresa <https://www.microsoft.com/en-us/sql-server/sql-server-downloads>. [3]

SSMS (SQL Server Management Studio) este un program dezvoltat de către Microsoft care incorporează multe unelte care facilitează proiectarea și menținerea unei baze de date relaționale de tip Microsoft SQL Server. În cadrul acestui proiect am folosit doar o mică parte din funcționalitățile existente, fiind totuși foarte util în verificarea manuală a relațiilor generate de către Entity Framework prin intermediul generării automate a unei diagrame entitate-relație. Ultima versiune poate fi descărcată de pe pagina oficială Microsoft la adresa <https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms> din secțiunea „Download SSMS”. [4]

## Node.JS și React

Node.JS este un mediu de execuție open-source pentru limbajul JavaScript.[5] Acesta este versatil și poate fi folosit pentru a construi atât frontend cât și backend. NPM (Node Package Manager) vine la pachet cu instalarea Node.JS și poate fi folosit pentru a instala extensii – numite pachete – în proiect. Adresa la care poate fi găsit installer-ul este <https://nodejs.org/en>. [5]

React este o librărie open-source pentru dezvoltarea de frontend, bazat pe componente plasate într-un arbore vizual. Poate fi folosită atât pentru aplicații web, cât și pentru aplicații mobile sau chiar aplicații desktop (cu Electron). În cazul aplicațiilor web, cu ajutorul lui se pot crea aplicații single-page, adică aplicații care nu necesită reîncărcare atunci când ruta se schimbă. O aplicație React nu are nevoie de Node.JS pentru a rula, ci pentru a fi creată și build-ată. Instalarea se face prin intermediul NPM, cu rularea comenzii „npm install create-react-app” într-un terminal powershell. Pentru a crea o nouă aplicație putem rula comanda “npx create-react-app <numele aplicației>”.

### Pachete auxiliare

Pe lângă conținutul librăriei React, proiectul mai folosește și unele pachete auxiliare. Înainte de a rula aplicația trebuie să instalăm pachetele necesare rulând comanda “npm update” folosind un terminal powershell în directorul aplicației.

Browser-image-compression ajută la comprimarea imaginilor și este folosit în cadrul proiectului pentru a reduce timpul de așteptare cauzat de încărcarea imaginilor prin comprimarea lor înainte de încărcarea către backend.

Pachetele @react-redux și @reduxjs/toolkit sunt o îmbunătățire a mecanismului de stare din React native. React suportă definirea unor variabile de stare care forțează o redesenare a arborelui vizual atunci când valoarea acesteia se schimbă, începând cu componenta în care se află variabila. Implicit, aceste variabile de stare pot fi declarate doar la nivel de componentă iar dacă este necesară comunicarea lor către altă componentă aceasta se face prin intermediul funcțiilor callback sau proprietăților componentelor. Redux toolkit extinde această funcționalitate, permițând definirea unor variabile de stare la nivel global.

Jwt-Decode este folosit pentru extragerea unor informații din tokenul de autentificare de tip JWT. Mai precis, în interiorul aplicației, este folosit pentru a verifica dacă sesiunea utilizatorului curent a expirat fără a mai fi nevoie de trimiterea unui request către backend.

@mui/material conține o multitudine de componente vizuale printre care se numără selectoare, câmpuri de text și casete de selecție. Material UI este destul de comun în aplicațiile moderne, fiind suportat atât de React cât și de Angular și chiar Android.

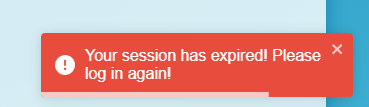
Folosind react-toastify experiența utilizatorilor poate fi îmbunătățită prin afișarea unor notificări de tip toast. O astfel de notificare este exemplificată mai jos, în figura 2.4.

Figura 2.4 – O alertă de tip toast

## Visual Studio Code

Visual Studio Code este un editor de text gândit pentru cod, fiind o alegere excelentă pentru a crea o aplicație de tip React. Editorul are capabilitatea de a crea terminale în propria fereastră, în care poate fi rulată aplicația. Datorită capabilităților de „hot-reload” ale React aplicația va reflecta modificările de cod imediat după salvarea lor, fără a mai fi nevoie de repornirea acesteia. Visual Studio Code nu vine implicit cu foarte multe funcționalități dar este puternic modificabil prin magazinul de extensii. Acesta poate fi descărcat de la adresa oficială <https://code.visualstudio.com>. [6]

## TypeScript și SCSS

TypeScript poate fi văzut ca o extensie a JavaScript deoarece respectă aceleași principii și are sintaxa foarte similară. Principala diferență intervine la gestionarea variabilelor deoarece cele din TypeScript au tipuri statice, adică sunt cunoscute la momentul compilării. Am ales TypeScript în favoarea JavaScript deoarece rezolvarea erorilor este mult mai ușoară atunci când tipul variabilelor este cunoscut, reducând mult timpul alocat debugging-ului. Merită menționat și că navigatoarele web nu sunt capabile să ruleze TypeScript, acesta fiind „tradus” în limbaj JavaScript la compilare. Pentru a crea o aplicație React cu limbajul TypeScript trebuie să adăugam “ –template typescript” la comanda create-react-app descrisă în subcapitolul 2.4.

SCSS (Sassy Cascading Style Sheets) este o variantă mai avansată de CSS. Printre îmbunătățirile aduse se numără încapsularea declarațiilor de stil și declararea de variabile. Pentru a folosi fișiere cu extensia .scss în interiorul proiectului este nevoie de instalarea pachetului sass prin rularea comenzii „npm install sass”.

**Capitolul III**

# Implementarea Web API-ului

## Structura soluției

### Repository pattern

Aplicația primește request-uri de tip REST în controllere care fac apel doar la metode din managere (BLL – Business Logic Layer). Managerele se ocupă de logica care manipulează datele și impune sau respectă anumite condiții. Managerele fac la rândul lor apel la metode din repository-uri (DAL – Data Access Layer) sau chiar din alte managere. Repository-urile se ocupă strict de adăugarea, modificarea și ștergerea datelor din baza de date. În cadrul aplicației acestea sunt conectate la un DbContext (Database Context) care „traduce” cererile repository-urilor în comenzi de SQL și le execută. Design pattern-ul descris mai sus se numește „repository pattern” și este folosit în contextul aplicațiilor mari deoarece permite o testare mai ușoară. De asemenea, abstractizarea straturilor aplicației permite o adaptare mai rapidă la cerințe noi sau schimbări ale celor existente. De exemplu, o schimbare a metodei de stocare implică doar modificarea repository-urilor, restul codului putând rămâne neschimbat.

O altă condiție a repository pattern este folosirea interfețelor. Din moment ce se dorește o ușoară înlocuire a managerelor și repository-urilor acestea nu sunt pasate direct ca argument în constructorii claselor care le necesită ci fiecare implementează o interfață care este dată de fapt ca parametru. Pentru a face legătura între interfață și implementare se folosește injectarea dependințelor (dependency injection) în clasa Startup a aplicației, ca în secvența următoare.

public void ConfigureServices(IServiceCollection services){

services.AddScoped<IVehicleRepository, VehicleRepository>();

}

La fiecare apariție a interfeței *IVehicleRepository* ca parametru al unui constructor se va face legătura la o instanță a clasei *VehicleRepository*. Metoda *AddScoped* specifică durata de viață a instanței, aceasta fiind la fel de lungă ca durata rezolvării request-ului din controller care a dus la crearea acesteia. Toate injectările din cadrul proiectului sunt făcute cu metoda *AddScoped*.

Fiecare entitate are un repository care să-i gestioneze interacțiunea cu baza de date. Deși funcționalitățile necesare diferă de la entitate la entitate, toate pot include metodele de bază de creare, modificare, ștergere și preluare a întregii tabele. Din acest motiv am ales să implementez un repository generic abstract ce poate fi extins de orice implementare a unui repository.

public abstract class RepositoryBase<EntityType> : IRepositoryBase<EntityType> where EntityType: Entity

{

protected readonly AppDbContext context;

protected DbSet<EntityType> entitySet = null;

public RepositoryBase(AppDbContext context)

=> this.context = context;

public virtual async Task Create(EntityType toCreate)

{

if (entitySet == null)

throw new System.Exception("The entity set was not initialised!");

await entitySet.AddAsync(toCreate);

await SaveAsync();

}

public virtual async Task Delete(EntityType toDelete)

{

if (entitySet == null)

throw new System.Exception("The entity set was not initialised!");

await Task.FromResult(entitySet.Remove(toDelete));

await SaveAsync();

}

public virtual async Task<List<EntityType>> GetAll()

{

if (entitySet == null)

throw new System.Exception("The entity set was not initialised!");

var objects = await entitySet.ToListAsync();

return objects;

}

public virtual async Task Update(EntityType updatedObject)

{

if (entitySet == null)

throw new System.Exception("The entity set was not initialised!");

await Task.FromResult(entitySet.Update(updatedObject));

await SaveAsync();

}

protected async Task SaveAsync() => await context.SaveChangesAsync();

}

Această implementare necesită ca variabila *entitySet* să fie initializată în constructor de către toate repository-urile ce o extind. Funcția *SaveAsync*() este echivalentă cu comanda SQL COMMIT.

### Specification pattern

O specificație este o descriere a unor reguli sau condiții aplicate pe un set de date, în acest caz pe o tabelă din baza de date. Specification pattern presupune definirea unor specificații ce pot fi înlănțuite și combinate în orice mod pentru a elimina scrierea de logică duplicată. Pentru a permite această funcționalitate trebuie să adăugam următoarele două funcții la implementarea repository-ului generic din secțiunea 3.1.1.

/// <summary>

/// Used to apply a specification on the current entity set

/// </summary>

protected IQueryable<EntityType> ApplySpecification(Specification<EntityType> specification)

{

if (entitySet == null)

throw new System.Exception("The entity set was not initialised!");

return SpecificationEvaluator.GetQuery(entitySet, specification);

}

/// <summary>

/// Used to apply a specification on a queryable

/// </summary>

protected IQueryable<EntityType> ApplySpecification(IQueryable<EntityType> queryable, Specification<EntityType> specification)

{

return SpecificationEvaluator.GetQuery(queryable, specification);

}

Putem lua ca exemplu următoarele 2 specificații. Prima filtrează programele de lucru după cheia primară a locației iar a doua după o zi a săptămânii

public class ScheduleByLocationIdSpecification : Specification<Schedule>

{

public ScheduleByLocationIdSpecification(string locationId)

: base(x => x.LocationId == locationId) { }

}

public class ScheduleByWeekdaySpecification : Specification<Schedule>

{

public ScheduleByWeekdaySpecification(WeekdayEnum weekday)

: base(x => x.Weekday == weekday) { }

}

Așadar, dacă avem nevoie de o metodă care să returneze programul unei locații dintr-o zi specifică putem folosi o înlănțuire a ambelor specificații.

### Structura directoarelor

public async Task<Schedule> GetByLocationIdAndWeekday(string locationId, WeekdayEnum weekday)

=> await ApplySpecification(

ApplySpecification(new ScheduleByLocationIdSpecification(locationId)),

new ScheduleByWeekdaySpecification(weekday)).FirstOrDefaultAsync();

Din moment ce aplicația are peste 100 de clase și interfețe este importantă o grupare logică în directoare și subdirectoare, care să asiste programatorul în găsirea rapidă a fișierului de care are nevoie. În continuare voi descrie succint structura acestor directoare.

Directorul „Configurations” conține configurațiile tabelelor ce se generează din entități. Controllerele sunt grupate în directorul „Controllers”. În directorul „Entities” se află definițiile tuturor entităților. Directorul „Interfaces” conține toate interfețele din proiect, iar interfețele pentru managere și repository-uri sunt grupate la rândul lor în subdirectoare. Directoarele „Managers” și „Repositories” conțin implementările interfețelor menționate anterior. Directorul „Migrations” este generat de Entity Framework și conține migrațiile generate de comanda „Add-Migration”. Directorul „Models” conține clase structurale, precum cele de intrare și ieșire prin intermediul request-urilor din controllere care sunt grupate în subdirectoarele „Input” și „Return”. În directorul „Specifications” se află toate specificațiile din proiect, fiind grupate în subdirectoare în funcție de entitatea pe care se aplică.

## Baza de date

Metoda de proiectare „code-first” presupune ca baza de date să nu fie creată și modificată pe baza unor comenzi SQL, ci ca aceasta să fie generată și menținută pe baza unor clase. Interacțiunea cu această bază de date este gestionată de un context (DbContext), care dictează structura bazei de date și e folosit pentru a interoga baza de date prin interiorul unor DbSet-uri care sunt practic referințe la tabele. Codul SQL pentru interogări este generat automat de Entity framework.

Pentru a marca o clasă ca fiind entitate este necesar să declarăm un DbSet ce conține acea clasă în interiorul contextului. Să presupunem că avem 2 entități, Vehicle și Wheel. Secvența următoare de cod exemplifică cum s-ar declara o relație de tip 1:M între cele două entități.

În cazul precedent nu este necesară nicio configurare suplimentară, Entity framework deducând automat anumite detalii. Cheia externă va fi presupusă ca fiind VehicleId, deoarece e compusă din numele entității la care face referire și „Id”. Referințele virtuale servesc dublu scop, întâi la creare pentru a stabili relație iar apoi la execuție pentru a face operația JOIN. Dacă în interiorul repository-ului clasei Wheel apelăm metoda „.Include(x => x.Vehicle)” Entity framework va face automat o operație de JOIN la interogare iar proprietatea Vehicle va conține vehiculul la care se face referire. În caz contrar, referința va fi null. Pentru relațiile M:M se generează automat o tabelă de asociere.

class Wheel

{

public string Id { get; set; }

public string VehicleId { get; set; }

public virtual Vehicle Vehicle { get; set; }

}

class Vehicle

{

public string Id { get; set; }

public virtual ICollection<Wheel> Wheels { get; set; }

}

class StatusConfiguration : IEntityTypeConfiguration<Status>

{

public void Configure(EntityTypeBuilder<Status> builder)

{

//primary foreign

builder

.HasKey(x => x.VehicleId);

builder

.Property(x => x.IsSold)

.HasDefaultValue(false);

//nullable

builder

.Property(x => x.DateSold)

.IsRequired(false);

//1 - M: User <-> Status

builder

.HasOne<User>(status => status.PurchasedBy)

.WithMany(user => user.PurchasedVehicleStatuses)

.HasForeignKey(status => status.PurchaserUserId)

.IsRequired(false);

}

}

În secvența anterioară de cod se află configurația entității Status, scopul fiecărei funcții fiind ușor deductibil din comentarii și numele metodelor aplicate.În anumite cazuri nu este destul să lăsăm Entity framework să deducă toate detaliile. Pentru aceste cazuri există opțiunea de a face configurații. Pentru a aplica o configurație trebuie să apelăm funcția *modelBuilder.ApplyConfiguration*() în metoda *OnModelCreating*() din *DbContext*.

Baza de date conține 22 de tabele, 7 dintre ele fiind generate de către Identity pentru gestionarea utilizatorilor. Anexa 1 reprezintă diagrama Entitate-Relație, tabelele generate de către Identity fiind marcate cu un cerc verde în colțul din dreapta sus.

## Securitatea

În contextul unui Web API securitatea este unul dintre cele mai importante aspecte. Este nevoie să protejăm atât integritatea și stabilitatea aplicației cât și confidențialitatea datelor utilizatorilor.

### Stocarea informațiilor sensibile

Informațiile secrete din componența unui proiect cum ar fi parolele și cheile private nu trebuie stocate niciodată în clar (numit și hard-codare). În cadrul acestui proiect informațiile de acest tip sunt preluate din fișierul appsettings.json. Totuși, din moment ce proiectul este stocat într-un repository pe GitHub acest fișier nu poate conține toate detaliile. Chiar dacă repository-ul este privat această acțiune ar crea riscuri de securitate. În interiorul mediului de dezvoltare fișierul appsettings.json conține doar un schelet al datelor, acestea fiind stocate de fapt în fișierul appsettings.Development.json care nu este încărcat în repository. Gestionarea acestor date în versiunea de producție se poate face automat prin servicii precum Microsoft Azure.

### Autentificarea

O aplicație de tip Web API expune o multitudine de endpoint-uri prin intermediul controllerelor, dar nu fiecare dintre ele trebuie să fie accesibil oricui cunoaște ruta. Pentru a rezolva această problemă se folosește framework-ul de autentificare OAuth 2.0 cu tokeni de tip „Bearer” descris în RFC 6750. Schema de autentificare „Bearer” presupune atașarea unui token în interiorul antetului request-ului, în secțiunea „Authorization”. Un token trebuie obținut de la server iar orice entitate care se află în posesia acestuia va avea acces la resursa protejată fără a se verifica cum acesta a fost obținut.[7] Pentru a configura acest tip de autentificare se apelează metoda *services.AddAuthentication*() în interiorul funcției *ConfigureServices*() din *Startup*, dându-se ca parametru schema de autentificare Jwt Bearer.

Pentru a decide cui să-i ofere acces la conținut prin tokenii menționați anterior avem nevoie de o metodă de validare și identificare a utilizatorilor. Am ales să folosesc framework-ul Identity în favoarea implementării manuale a entităților și metodelor ce gestionează conturile de utilizator. La nivel de configurare, a fost nevoie doar să extind clasa *IdentityUser* pentru a implementa interacțiunile (sau relațiile) necesare cu celelalte entități din cadrul proiectului. Adăugarea Identity este trivială, fiind necesar ca contextul să extindă clasa *IdentityDbContext* și ca metoda *services.AddIdentity*<clasă user, clasă rol>.*AddEntityFrameworkStores*<clasă DbContext>() să fie apelată în interiorul metodei *ConfigureServices*. Odată adăugat acesta pune la dispozitie clase injectabile de tip manager, precum *RoleManager*<clasă rol> și *UserManager*<clasă user> de care ne folosim pentru a gestiona conturile de utilizator.

În cadrul proiectului există trei roluri, User, Admin și Sysadmin. La pornirea proiectului se verifică dacă aceste roluri există, în caz contrar fiind create automat. Un cont de tip User poate fi creat de orice și are acces la endpointurile necesare vizualizării unui vehicul și creării unei programări. Un Admin are acces la toate metodele de gestionare a utilizatorilor de tip User, vehiculelor, locațiilor și programărilor iar un SysAdmin are toate permisiunile unui Admin plus abilitatea de a crea un nou cont de Admin. Nu este expus niciun endpoint pentru adăugarea unui cont de Sysadmin, crearea unuia fiind făcută automat la rularea proiectului dacă acesta nu există.

//create the default sysadmin

if (await userManager.FindByNameAsync("sysadmin") == null)

{

List<string> adminRoles = new() { "User", "Admin", "SysAdmin" };

var adminUser = new User()

{

UserName = "sysadmin",

Email = "",

EmailConfirmed = true

};

var result = await userManager.CreateAsync(adminUser, configuration.GetValue<string>("RootPasswords:sysAdminPassword"));

if (!result.Succeeded)

throw new Exception("Could not create the default sysadmin!");

foreach (string role in adminRoles)

await userManager.AddToRoleAsync(adminUser, role);

}

Secvența de cod de deasupra se ocupă de crearea automată a contului implicit de Sysadmin. Deși parola este preluată din appconfig.json, modificarea acesteia nu se va face dacă contul este deja creat fiind necesară întâi ștergerea manuală a utilizatorului din baza de date. Proprietatea *EmailConfirmed* este setată deoarece un utilizator nu se poate autentifica decât dacă își confirmă adresa de email, iar acest utilizator implicit nu are o adresă. Se poate observa că toate rolurile sunt adăugate acestui Sysadmin, inclusiv cele de User și Admin. Adăugarea lor este necesară deoarece în modul în care este configurat proiectul nu se creează o ierarhie a rolurilor, deci un Sysadmin nu este considerat mai presus unui Admin pentru a avea acces la endpoint-urile lui. Pentru a bloca accesul către un endpoint se folosește tag-ul Authorize și polița asignată rolului respectiv, ca în exemplul următor.

[HttpPost("signUpAdmin")]

[Authorize(Policy = "Sysadmin")]

public async Task<IActionResult> CreateAdmin([FromBody] UserCreateModel newUser)

### Vulnerabilități

Token-ul de tip bearer JWT nu este semnat digital, ceea ce înseamnă că Web API-ul nu are nicio metodă de a verifica cine îl trimite. Așadar, această metodă de autentificare este vulnerabilă la un atac de tip man-in-the-middle, un adversar putând să intercepteze token-ul și să impersoneze un utilizator.

## Controllere și funcționalitate

Web API-ul expune 7 controllere, care pot fi gândite ca ramificații ce separă endpoint-urile pe criteriul funcționalității. Voi descrie pe scurt capabilitățile acestora într-o ordine logică.

*AuthenticationController* conține metode ce se ocupă cu gestionarea utilizatorilor, cum ar fi crearea utilizatorilor, confirmarea adresei de email, resetarea parolei și obținerea tuturor numelor de utilizator.

*LocationController* expune endpoint-uri ce permit adăugarea, ștergerea, actualizarea și citirea locațiilor. Același lucru se poate spune și despre *BodyTypeController* și *FeatureController* relativ la entitățile respective.

*VehicleController* expune metodele CRUD de bază pentru modificarea vehiculelor. În plus, conține endpoint-uri pentru vizualizarea atât tuturor vehiculelor puse la vânzare cât și celor vândute anterior, incluzând diferite filtre, paginație și un algoritm de recomandare. *PictureController* poate fi considerat ca un controller ajutător deoarece conține doar metode pentru a prelua și modifica imaginile unui vehicul. Imaginile vehiculelor nu sunt returnate în același request ca și vehiculul deoarece sunt foarte mari și ar încetini încărcarea paginii. Pentru a îmbunătăți experiența utilizatorilor, informațiile de bază sunt afișate înainte ca imaginile să se încarce.

*AppointmentController* permite utilizatorilor să facă o programare în funcție de locația unde se află vehiculul de care sunt interesați. Pentru a face asta e nevoie de o metodă prin care să se pună la dispoziție intervalele libere care va fi prezentată în decursul acestui subcapitol. De asemenea, acest controller le permite utilizatorilor cu rol de Admin să vadă toate programările, implicit cele viitoare cu opțiunea afișării celor anterioare.

### Trimiterea de email-uri

În domeniul aplicației se trimit email-uri pentru confirmarea adresei de email, resetarea parolei și notificarea creării unei programări. Am putea teoretic să instanțiem un client de SMTP (Simple Mail Transfer Protocol) în managere de fiecare dată când dorim să trimitem un email dar această acțiune ar viola mai multe principii ce stau la baza unei aplicații reușite precum modularitatea, extensibilitatea și evitarea codului duplicat.

Ca soluție a acestei probleme de design am descris această funcționalitate prin interfața *IEmailManager* ce permite trimiterea unui mesaj HTML cu un titlu la o adresă specificată. De fiecare dată când este necesară trimiterea unui email se folosește serviciul injectabil *EmailManager*. Această acțiune se realizează prin conectarea la serverul de SMTP Gmail (smtp.gmail.com) prin protocolul TLS (port 587). Autentificarea se face cu un cont de gmail creat doar pentru acest scop. Din considerente de securitate Google nu mai permite autentificarea cu adresa de email și parola așa că a fost necesară generarea unui „app password” folosit pentru a permite accesul Web API-ului la serverul de SMTP. [8] O posibilă îmbunătățire ar fi implementarea unor metode de formatare a textului deoarece, în prezent, orice clasă manager apelează metoda de trimitere a unui email trebuie să dea ca parametru un text HTML.

### Paginația

Interfața vizuală afișează o listă de pagini ce conțin autovehicule. La o scară mică am putea afișa lista cu toate autovehiculele din baza de date dar într-un container scrolabil dar, pentru a respecta principiul scalabilității, este nevoie să ne asigurăm că utilizatorul va primi informațiile pe care le cere într-un timp relativ scurt indiferent de câte vehicule se află în baza de date.

Nu putem face această paginație la nivelul interogării bazei de date deoarece utilizatorii au opțiunea de a aplica diverse filtre și sortări. Așadar, paginația se realizează în *VehicleManager* după ce toate procesările au fost efectuate folosindu-se metoda *ApplyPagination* din secvența următoare de cod.

private static IEnumerable<VehicleModel> ApplyPagination(IEnumerable<VehicleModel> vehicles, VehiclePaginationModel pagination)

{

if (pagination.StartAt != null && pagination.NumberToGet != null)

return vehicles.Skip(pagination.StartAt.Value).Take(pagination.NumberToGet.Value);

else

return vehicles;

}

Prin metoda descrisă anterior se face o interogare a întregii tabele cu autovehicule de fiecare dată când se face un request la endpoint-urile ce returnează liste de autovehicule. La prima vedere poate părea ineficient dar, mulțumită tehnicii de „caching” implementate de SQL Server, acest proces este eficientizat prin stocarea anumitor date și optimizări în memoria RAM. În contextul acestui proiect tabela cu autovehicule nu se va modifica de multe ori pe zi sau poate chiar pe săptămână deci timpul de așteptare va rămâne scăzut.

Controller-ul *VehicleController* pune la dispoziție și un endpoint ce returnează numărul de autovehicule ce va fi transformat într-un număr de pagini în interfața vizuală, deoarece este important ca utilizatorul să cunoască această informație. Nu se poate returna direct numărul de pagini deoarece acesta variază în funcție de numărul de vehicule per pagină sunt selectat din interfața vizuală.

### Filtre și sortări

Din moment ce este vital ca utilizatorul să poată găsi rapid categoria de vehicul pe care și-o dorește trebuie ca aplicația să-i pună la dispoziție în interfață o gamă de metode de căutare. Odată selectate, aceste filtre și sortări se pot aplica fie în frontend, fie în backend. Dacă am alege filtrarea în frontend experiența utilizatorilor ar fi puternic afectată deoarece efortul computațional ar fi suportat de dispozitivul utilizatorului, ignorând faptul că limbajul JavaScript este semnificativ mai încet ca și C#. De asemenea, ar trebui să se renunțe la anumite funcționalități precum paginația menționată anterior.

Filtrarea se poate face după marcă, model, kilometraj, un preț minim și/sau unul maxim, o putere minimă și/sau una maximă, un an minim de fabricație, tipul de caroserie și tipul transmisiei. Toate filtrele sunt opționale, pot fi selectate simultan, și se fac prin metode LINQ din .NET ca în exemplul următor.

if (filters.MinPower != null)

vehicleQueryable = vehicleQueryable.Where(x => x.Power >= filters.MinPower.Value);

Pentru filtrarea după marcă și model se pot trimite în request orice string-uri. Totuși, *VehicleController* expune un endpoint ce returnează un dicționar cu toate mărcile ca chei și o listă cu toate modelele asignate respectivei mărci ca valori. Acest dicționar poate fi preluat de frontend pentru a expune utilizatorului o metodă de selecție ce conține doar mărcile și modelele (în funcție de marcă) disponibile în baza de date.

Tipurile de sortare sunt exclusive și se pot face după nume (lexicografic, marcă + model), preț, kilometraj și putere atât ascendent cât și descendent. În plus, se poate face o sortare după un scor calculat de algoritmul de recomandare care va fi descris în următoarea secțiune. Secvența următoare de cod reprezintă o variantă restrânsă a logicii ce se ocupă de sortare.

if (filters.Sort != null)

{

var sortMultiplier = 1;

if (filters.SortAsc != null && filters.SortAsc.Value == false)

sortMultiplier = -1;

switch (filters.Sort.Value)

{

case FiltersSortTypeEnum.Price:

result = result.OrderBy(x => sortMultiplier \* x.Price).ToList();

break;

case FiltersSortTypeEnum.Mileage:

result = result.OrderBy(x => sortMultiplier \* x.Odometer).ToList();

break;

default:

throw new Exception("Unsupported sort type!");

}

}

### Expunerea intervalelor disponibile

În momentul în care un potențial client dorește să facă o programare trebuie ca aplicația să îi expună o metodă de a alege un interval de timp pentru respectiva programare. O implementare naivă ar putea fi ca utilizatorul să aleagă orice perioadă de timp iar Web API-ul să returneze dacă acel interval este disponibil sau nu. Totuși, experiența utilizatorului ar fi puternic impactată de o asemenea implementare. Implementarea din cadrul proiectului este definită în *IAppointmentManager*, metoda *GetAvailableAppointmentTimes* returnează un dicționar ce are zile ca chei și timpurile posibile de începere ale programării din ziua respectivă, ca valoare.

Implementarea din *AppointmentManager* nu este una perfectă, fiind posibil ca citirile din baza de date să o încetinească atunci când vorbim de o tabelă foarte populată cu alte programări. Numărul de zile din viitor pentru care se caută intervale libere este primit ca parametru în request dar, pentru a evita un posibil atac în care un adversar ar cere un număr foarte mare de zile este setată o limită de 60 (*daysLimit* = 60). O posibilă limitare este că nu se pot face programări în ziua curentă, doar în ziua următoare, deoarece o locație ar trebui să pregătească autovehiculul care urmează sa fie prezentat, acțiune care necesită timp. Pentru a găsi zilele cu intervale se iterează prin fiecare zi (până la numărul necesar de zile din request) și se verifică dacă locația respectivă are program de lucru în ziua curentă. În caz pozitiv, se iau toate programările din acea locație în ziua respectivă și se iterează prin toate posibilele timpuri de start ale programării între începutul și finalul programului de lucru, în pași de 15 minute (*intervalSize* = 15). Pentru fiecare interval de tipul [timp curent de start, timp curent de start + durata programării] se verifică dacă există o programare în ziua curentă care s-ar suprapune cu acest interval. În caz negativ, timpul respectiv de start este adăugat la lista *validTimesForToday*. Dacă după iterarea prin toate timpurile posibile de start lista respectivă nu este goală atunci ziua și valorile pot fi adăugate în dicționarul rezultat.

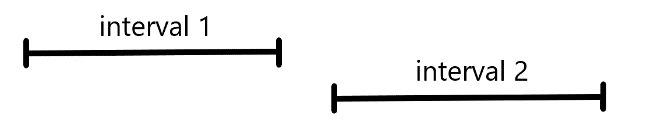
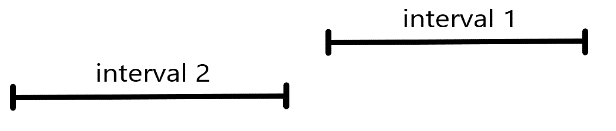
Pentru a verifica dacă 2 intervale se suprapun se folosesc două condiții - începutul intervalului 1 să fie înainte de sfârșitul intervalului 2 și sfârșitul intervalului 1 să fie după începutul intervalului 2. Figura 3.1 invalidează prima condiție, începutul intervalului 1 fiind după sfârșitul intervalului 2 iar Figura 3.2 invalidează a doua condiție, sfârșitul intervalului 1 fiind înainte de începerea intervalului 2. În orice alt caz, intervalele se suprapun.

Figura 3.2 – intervalul 2 complet înaintea intervalului 1

Figura 3.1 – intervalul 1 complet înaintea intervalului 2

### Algoritmul de recomandare

Filtrarea colaborativă (Collaborative filtering) este procesul filtrării sau evaluării unui obiect sau concept pe baza opiniei altor utilizatori similari. [9] Prin experiența personală am observat că preferințele în domeniul auto se bazează pe factori cum ar fi vârsta și mediul locuinței. Spre exemplu, o persoană mai în vârstă probabil va prefera un autovehicul mai practic, în contrast o persoană mai tânără dorind un autovehicul mai rapid sau mai confortabil.

La crearea unui cont utilizatorului i se oferă opțiunea de oferi câteva informații personale – sexul (Masculin/Feminin), grupa de vârstă (18-39, 40-59, 60+) și regiunea în care locuiește (Urban/Rural) care se stochează în tabela *UserDetails*, pe baza unor enum-uri. Chiar și în cazul unui atac informatic aceste detalii sunt destul de vagi cât să nu prezinte un pericol pentru anonimitatea și confidențialitatea utilizatorului. De fiecare dată când un utilizator apasă în interfața vizuală pe un vehicul, adică face un request către endpoint-ul ce returnează detaliile unui vehicul pe baza Id-ului, acea vizualizare se stochează în tabela *VehicleView*.

Atunci când se selectează criteriul de sortare „Recommended” se apelează metoda de sortare din managerul injectabil *RecommendationManager*. Inițial se iau din baza de date ultimele 10000 vizualizări (variabila *recommendationViewLimit* = 10000), interogarea fiind eficientă deoarece vizualizările se inserează în ordine cronologică. Următorul pas este colectarea tuturor utilizatorilor similari, adică care au toate cele 3 categorii de informații personale identice. Apoi, iterând prin toți utilizatorii similari, se alcătuiește o listă de vizualizări. Trebuie menționat că și utilizatorul curent se află în lista de utilizatori similari, iar vizualizările acestuia se vor adăuga de 5 ori pentru a le oferi de 5 ori mai multă importanță (variabila *personalRecommendationMultiplier* = 5). Având lista cu toate vizualizările relevante se calculează media prețului, media puterii motorului și un dicționar cu rol de vector de frecvență pentru tipurile de caroserie.

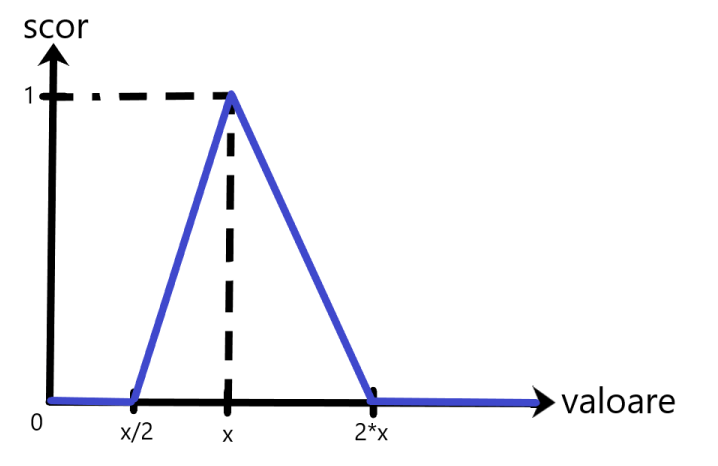
Pe baza informațiilor colectate se calculează, pentru fiecare vehicul, un scor de dezirabilitate. Pentru tipul de caroserie, scorul adăugat va fi numărul de apariții al tipului de caroserie respective împărțit la numărul de vizualizări relevante. Pentru putere și preț se folosește formula următoare: o valoare fix egală cu media va da un scor egal cu 1, acesta scăzând liniar spre 0 în jumătatea și dublul mediei, conform figurii 3.3 (unde *x* este media). Aceste trei sub-scoruri se însumează pentru a obține scorul final, cu ajutorul căruia se sortează descendent vehiculele pentru a obține lista recomandată.

Figura 3.3 – Regula de calcul al scorului valorilor numerice

**Capitolul IV**

# Implementarea paginii web

## Structura proiectului

Spre deosebire de backend, frontend-ul nu urmează o structură standard. Totuși, am încercat să mențin un grad de abstractizare prin separarea clară a funcționalității în componente, module și servicii.

Directorul „assets” a fost gândit să conțină toate resursele constante, precum imagini și fonturi dar în prezent conține doar imaginea ce se afișează când un vehicul nu are o imagine asignată.

Directorul „components” conține toate componentele reutilizabile create în React. Subdirectorul „Auth” conține toate cele 5 componente ce sunt sub ramura autentificării, fiecare reprezentând o pagină diferită. Subdirectorul „Error404” conține componenta care este afișată atunci când utilizatorul încearcă să acceseze o rută necunoscută. Subdirectorul „Loading” conține o componentă ce se afișează de fiecare dată când se așteaptă răspunsul unui request către backend. Subdirectorul „Main” conține toate componentele din structura paginii principale, inclusiv dialogurile.

Directorul „models” conține interfețele de tip model, ce au ca scop să definească structura informațiilor primite sau trimise prin intermediului apelurilor către backend. Fiecare interfață a tipurilor primite are și o metodă respectivă numită *mapJsonTo<<numele clasei>>* ce se ocupă de maparea corpului de tip JSON al request-ului la interfața definită. În subdirectorul „enums” se află descrierile tuturor enumerărilor ce sunt trimise ca date de intrare în request-urile către endpoint-uri.

//DriveTrainType enum in TypeScript

export enum DriveTrainTypeEnum {

    FWD = "FWD",

    RWD = "RWD",

    AWD = "AWD",

    FourWD = "FourWD"    //4WD, 4x4

}

//DriveTrainType enum in C#

public enum DriveTrainTypeEnum : byte

{

    FWD = 0,

    RWD = 1,

    AWD = 2,

    FourWD = 3      //4WD, 4x4

}

Directorul „redux” conține atât store-ul principal prin care se pot accesa variabilele de stare globale cât și fiecare store secundar care intră în componența lui. În directorul „services” se află toate colecțiile de funcții care incorporează logică independentă, precum metodele ce fac apeluri la backend. Un alt exemplu poate fi metoda *authenticatedFetch* care, apelat de alte metode ce comunică cu Web API-ul, deconectează automat utilizatorul în cazul primirii răspunsurilor 401 (Unauthorized) sau 403 (Forbidden).

export const authenticatedFetch = async (input: RequestInfo | URL, init?: RequestInit | undefined): Promise<Response> => {

    const result = await fetch(input, init);

    if (result.status === 401 || result.status === 403) {

        store.dispatch(forcedLogout()); //log the user out in redux

        //notify logout reason using toast

        generateToastError(errorReasons.get(result.status) || "");

        return Promise.reject("Token has expired!");

    }

    else {

        return result;

    }

}

## Tehnici de îmbunătățire a experienței utilizatorilor

În proiectarea unei pagini web unul dintre cele mai importante aspecte este experiența utilizatorilor. Chiar și o mică inconveniență precum un timp de încărcare ridicat sau un buton care nu funcționează corespunzător poate cauza o porțiune din utilizatori să părăsească pagina. De asemenea, fiecare pagină și componentă din cadrul acestui proiect este „responsive”, adică își păstrează un aspect utilizabil indiferent de dispozitivul pe care este vizionată pagina. În acest subcapitol voi descrie unele dintre metodele folosite pentru a îmbunătăți experiența acestora.

### 4.2.1 Persistența datelor

Datorită tokenilor JWT folosi

ți pentru autentificare putem menține sesiunea utilizatorului până când aceasta expiră. Din moment ce vrem ca aceste date să se păstreze la redeschiderea paginii, a navigatorului web sau chiar la repornirea dispozitivului nu putem folosi mecanismul de stări din React. Metoda prin care datele sunt stocate este componenta „Local storage” disponibilă în capabilitățile oricărui navigator web modern. De fiecare data când un utilizator se conectează se stochează rolul, numele de utilizator și tokenul JWT, atât într-un store din redux - *userStore* - cât și în *local storage*. La pornirea aplicației store-ul din redux încearcă să preia aceste date din *local storage*, dacă există.

Menținerea filtrelor și paginii curente din lista de autovehicule este de asemenea importantă deoarece aplicația este single-page și nu ne dorim ca utilizatorii să re-aplice filtrele sau să-și amintească și să reintroducă pagina de fiecare dată când revin din componenta ce permite vizionarea detaliilor unui vehicul. Pentru a realiza asta se folosește store-ul *vehiclesMainFiltersStore* unde se mențin toate filtrele, pagina curentă și numărul selectat de vehicule per pagină. Totuși, la revenirea în secțiunea cu lista, se retransmite request-ul deoarece există posibilitatea ca colecția de autovehicule din baza de date să se fi modificat și este indicat să oferim utilizatorilor informații curente.

### 4.2.2 Feedback

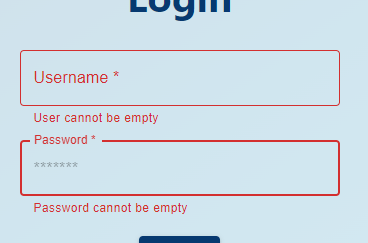
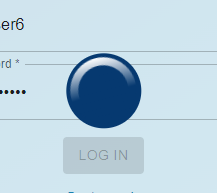
Feedback-ul poate fi împărțit în 2 categorii, cel care informează utilizatorul că o acțiune este în desfășurare și cel care îi notifică despre rezultatul unei acțiuni. De fiecare dată când se așteaptă răspuns de la un request se afișează componenta „Loading”, care este reprezentată de un cerc rotativ (loading circle) ce se amplasează automat în centrul containerului în care este adăugat. Animația se face cu ajutorul SCSS, mai exact aplicarea unei rotații folosind o transformare. Figura 4.1 ilustrează un exemplu al acestei componente pe pagina de conectare.

Figura 4.1 – Componenta reutilizabilă „Loading”

Figura 4.2 – Erori de validare în formular

Toate butoanele folosite în proiect provin din MaterialUI care oferă implementări implicite pentru feedbackul apăsării butoanelor. După confirmarea trimiterii unui formular se validează datele de intrare iar dacă unul sau mai multe câmpuri sunt incorecte utilizatorul este notificat cu ajutorul proprietăților *error* și *helperText* ale câmpurilor din MaterialUI (Figura 4.2). Setarea proprietății *error* ca *true* va da câmpului o margine roșie iar cu ajutorul *helperText* se poate afișa motivul erorii. Pentru o actualizare dinamică a acestor proprietăți se folosesc variabile de stare. Majoritatea câmpurilor din cadrul proiectului sunt definite cu structura din exemplul de mai jos. La apăsarea butonului de trimitere se apelează o funcție de validare care se ocupă de setarea variabilelor de stare menționate anterior.

//state variable for storing value

const [usernameValue, setUsernameValue] = useState("");

//state variables for error notifications

const [usernameError, setUsernameError] = useState(false);

const [usernameErrorText, setUsernameErrorText] = useState("");

//MUI TextField

<TextField value={usernameValue} label="Username \*" margin="dense" fullWidth

  onChange={(event) => setUsernameValue(event.target.value)}

  type="text" placeholder="Username" name="username"

  error={usernameError} helperText={usernameErrorText}/>

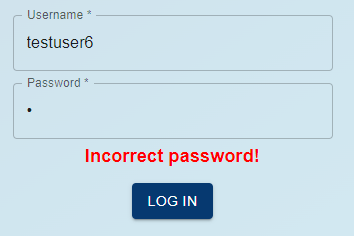
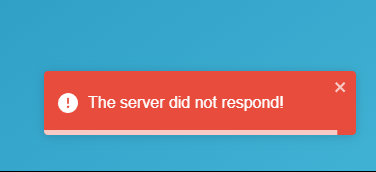
****După execuția unui request există 2 moduri de a notifica utilizatorul. În cazul în care backend-ul nu răspunde sau apare o eroare neașteptată se va afișa o notificare de tip *toast* (Figura 4.3) în colțul din dreapta jos. În cazul întoarcerii unui răspuns pozitiv (cod HTTP 200) se va afișa un mesaj cu font verde, iar în cazul întoarcerii unui răspuns negativ (cum ar fi HTTP 400 – Bad Request) se va afișa un mesaj cu text roșu (Figura 4.4).

Figura 4.3 – Notificare de tip *toast*

Figura 4.4 – Afișarea răspunsului negativ primit de la Web API

### Timpi de încărcare

În ultimele decenii performanța dispozitivelor a crescut exponențial, fiind unul din factorii care a cauzat dorința utilizatorilor pentru timpi din ce în ce mai scăzuți de încărcare. Cea mai mare problemă cu timpii de încărcare din cadrul proiectului a fost cauzată de imaginile autovehiculelor, acestea putând ajunge la dimensiuni foarte mari. Pentru a atenua această problemă am aplicat două metode.

Prima metodă presupune comprimarea imaginilor înainte de trimiterea acestora către backend cu ajutorul pachetului *browser-image-compression*, backend-ul având setată o limită de 2.5 MB pentru imagini și 0.5 MB pentru imaginile de prezentare (thumbnails).

import imageCompression from "browser-image-compression";

/\*\*

 \* Compresses an image in order to save bandwidth and storage space

 \* @param file - the image to be compressed

 \* @param maxSize - max size of the compressed image, in Mb

 \* @param maxDimension - max width or height of the compressed image

 \*/

export const compressImage = async (file: File, maxSize: number, maxDimension: number): Promise<File> => {

  const compressionOptions = {

    maxSizeMB: maxSize,

    maxWidthOrHeight: maxDimension

  }

  const compressedFile = await imageCompression(file, compressionOptions);

  return compressedFile;

}

A doua metodă implică o încărcare întârziată a imaginilor pe pagina de vizualizare a unui vehicul. Detaliile unui vehicul și imaginea de prezentare se încarcă în primul request, fiind mai rapid. După finalizarea acestui request și inițializarea paginii se execută încă un request care poate dura destul de mult, în funcție de numărul imaginilor și calitatea lor.

## Prezentarea interfeței

### Secțiunea de autentificare

Paginile de conectare (Figura 4.5) și înregistrare (Figura 4.6) sunt similare, fiecare conținând un formular și îndeplinind câte un singur scop. Sub butoanele ce execută acțiuni există și hyperlink-uri ce ajută la navigarea între secțiuni. Bifa de jos din pagina de înregistrare permite adăugarea detaliilor pentru algoritmul de recomandare, descris în secțiunea 3.4.5. Toate paginile din secțiunea *auth* au un aspect similar, având un formular centrat pe același fundal gradient cu același titlu în partea superioară.

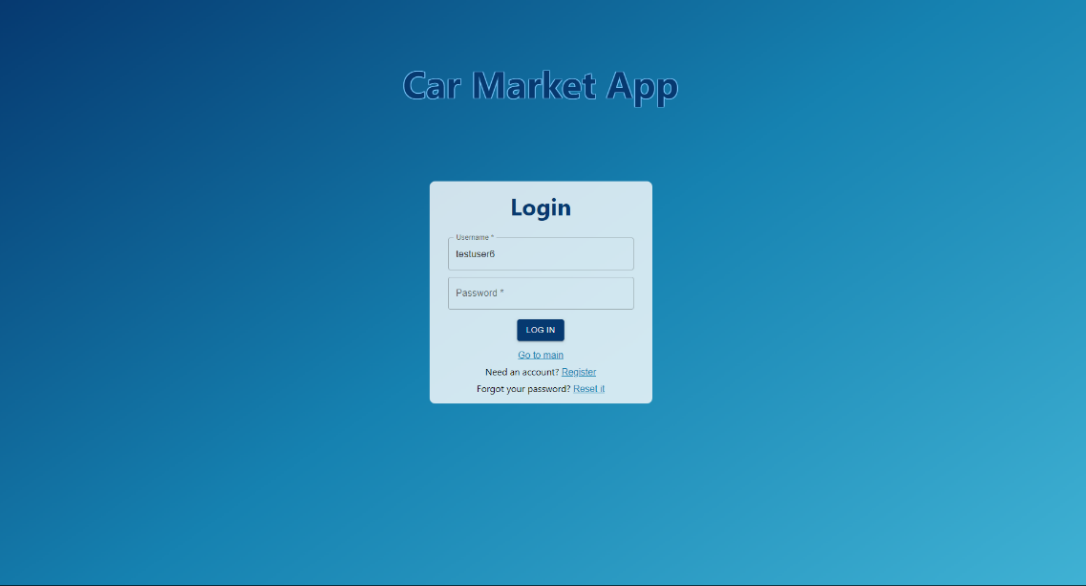
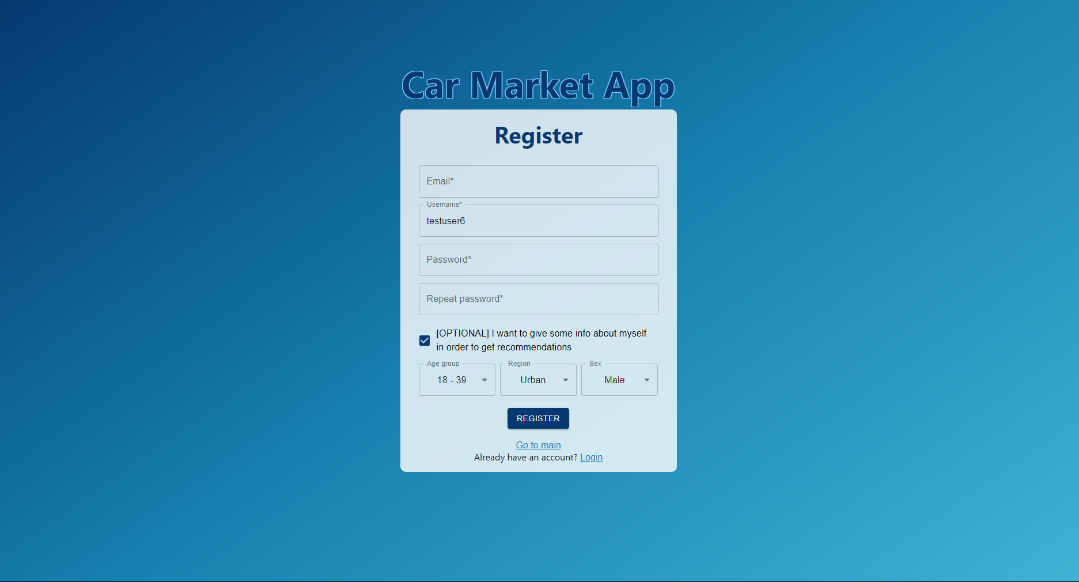
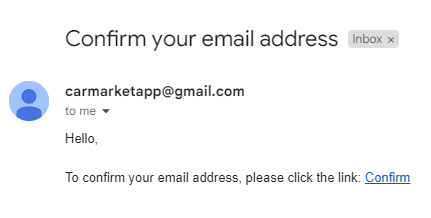


Figura 4.6 – Pagina de înregistrare (/auth/register)

Figura 4.5 – Pagina de conectare (/auth/login)

După înregistrarea unui cont nou de utilizator Web API-ul trimite un email de confirmare (Figura 4.7) ce conține un link cu un token în rută. Acel link redirecționează utilizatorul către formularul de confirmare a adresei (Figura 4.8).

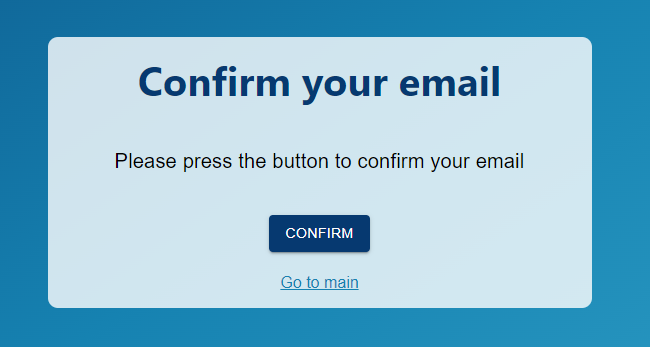


Figura 4.7 – Emailul de confirmare al creării contului

Figura 4.8 – Formularul de confirmare al adresei de email (/auth/confirmation/:token)

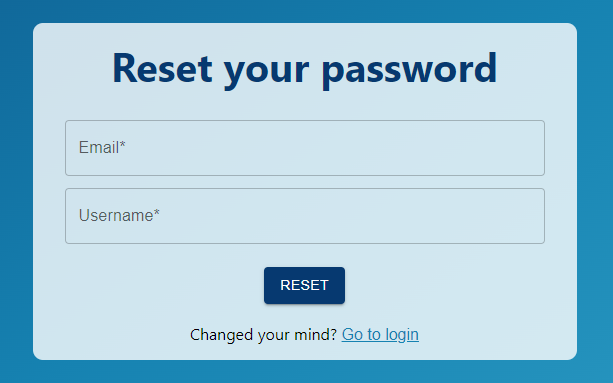
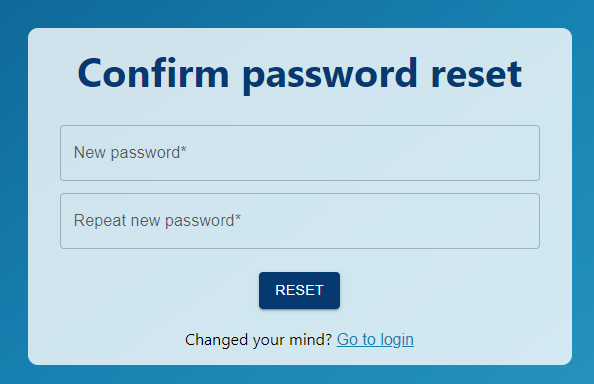
****Utilizatorii au opțiunea de a face o cerere de resetare a parolei, accesând formularul din Figura 4.9 prin intermediul link-ului de pe pagina de conectare. După introducerea adresei de email și a numelui de utilizator aceștia primesc un email de confirmare. Emailul de confirmare conține un link care îi redirecționează spre formularul de resetare al parolei (Figura 4.10). Ruta conține un token ce timp de expirare după 15 minute, necesar pentru a proteja împotriva unui potențial atac.

Figura 4.10 – Formularul de resetare al parolei (/auth/:username/:token)

Figura 4.9 – Formularul de cerere al resetării parolei (/auth/requestPasswordReset)

### Lista de autovehicule

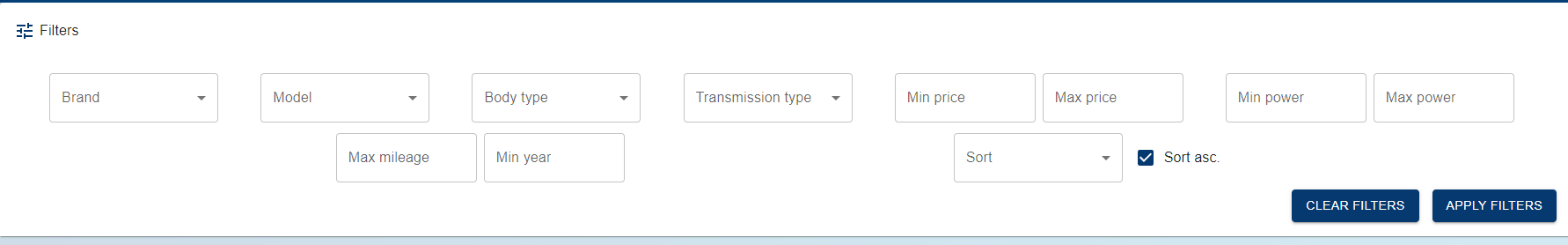
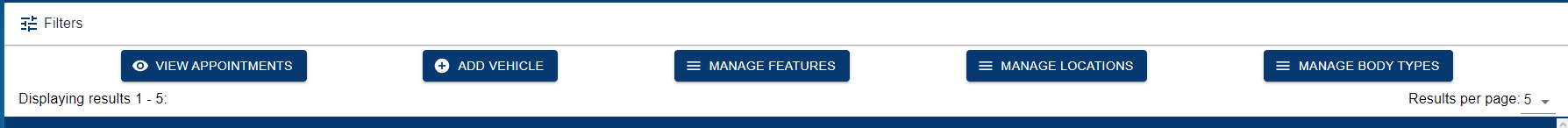
Ruta /main/vehicles afișează componenta *Vehicles* care conține o listă derulabilă (Figura 4.13) cu detalii restrânse despre vehiculele din baza de date puse la vânzare. Această rută este vizibilă și utilizatorilor fără cont. Utilizatorii conectați cu rol de *Admin* pot vizualiza și vehiculele deja vândute. Lista este împărțită în pagini, iar numărul de vehicule per pagină poate fi modificat (între valorile 5/10/15/20), fiind implicit 5. Apăsând pe linia albă cu titlul *Filters* utilizatorul poate aplica orice filtru sau sortare descrise în secțiunea 3.4.3 prin intermediul unor câmpuri și selectoare din Material UI (Figura 4.11).

Figura 4.12 – Meniul de administrare

Figura 4.11 – Secțiunea de filtre și sortări

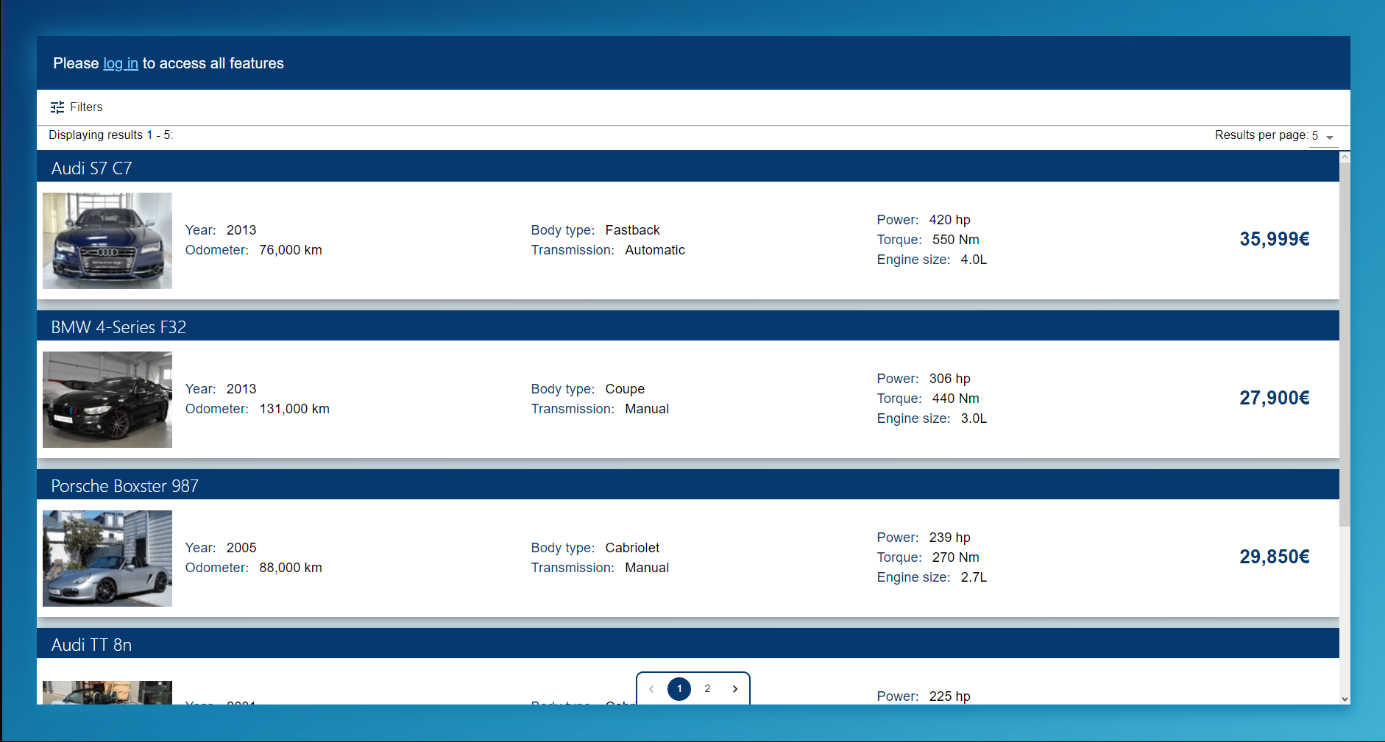


Figura 4.13 – Lista de autovehicule

Utilizatorilor conectați cu rolul de Admin li se va afișa un meniu adițional cu butoane (Figura 4.12) care deschid dialogurile de administrare. Teoretic, un utilizator își poate modifica rolul în local storage pentru a vedea acest meniu. Practic, un utilizator care își schimbă rolul în *Admin* nu va avea un token corespunzător iar Web API-ul va răspunde request-urilor cu 403 Unauthorized.

Butonul cu titlul *View Appointments* deschide un dialog de unde se poate vizualiza o listă cu programările curente, cu opțiunea de a afișa și programările anterioare. Apăsarea pe o programare din lista va duce utilizatorul la vehiculul pentru care este făcuta programarea.

Butonul etichetat *Add vehicle* deschide dialogul de adăugare al unui vehicul (Figura 4.14). Deschiderea acestui dialog necesită efectuarea unor request-uri către backend pentru a prelua informații precum locațiile și dotările disponibile.

Butonul etichetat *Manage features* arată subfereastra de unde se pot adăuga, șterge și modifica dotările. Ștergerea unei dotări este posibilă chiar dacă există autovehicule care o dețin.

Butonul cu titlul *Manage Locations* deschide dialogul de unde se pot adăuga, șterge și modifica locațiile. O locație atașata oricărui vehicul nu poate fi ștearsă. În plus, din interiorul acestui dialog se pot adăuga, șterge și modifica tipurile de programări per locație.

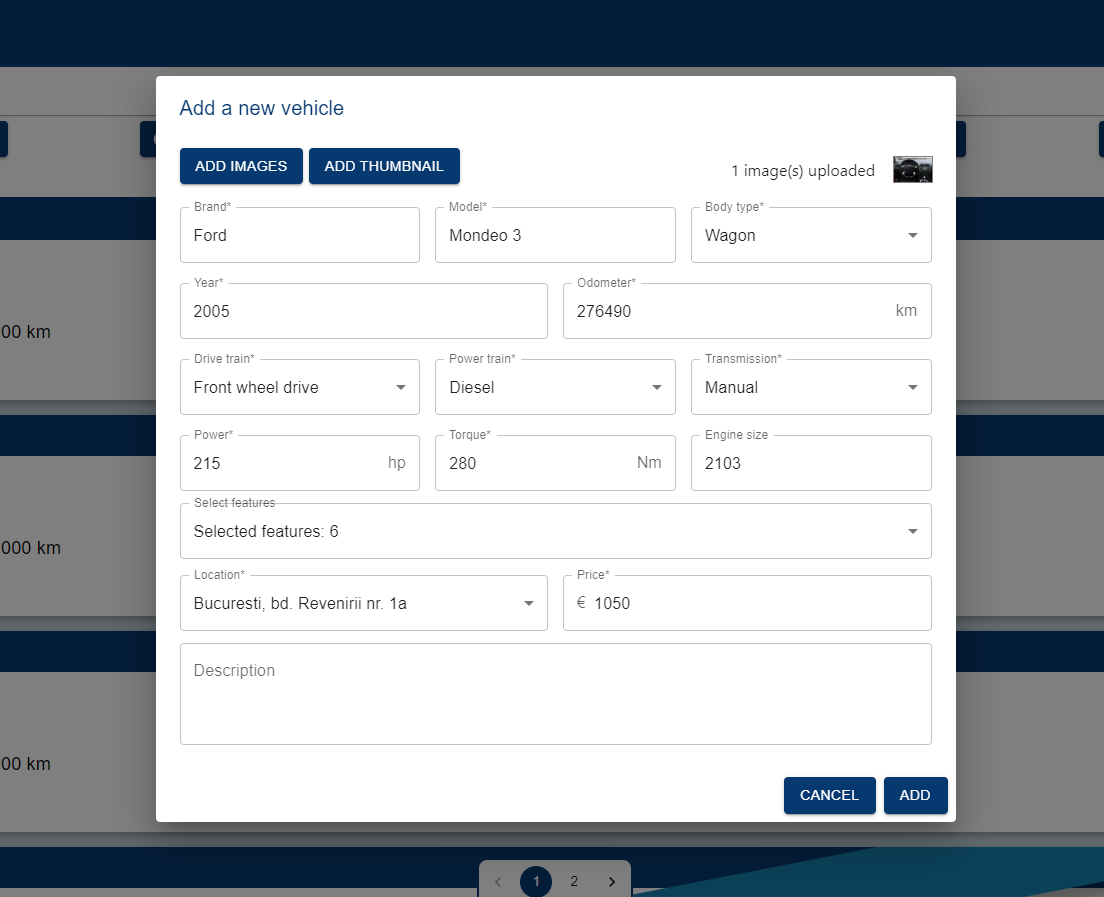
Butonul cu titlul *Manage body types* deschide un dialog de unde se pot adăuga și șterge tipurile de caroserie. Fiind o relație strictă, nu se pot șterge tipurile de caroserie care sunt atașate la vehicule. Un tip de caroserie nu este modificabil deoarece numele acestuia este și cheia primară din baza de date.

Figura 4.14 – Dialogul de adăugare al unui vehicul

### Detaliile unui autovehicul

La apăsarea pe un autovehicul din lista din secțiunea precedentă, utilizatorii conectați sunt redirecționați către o pagină unde sunt afișate informații detaliate despre vehiculul respectiv (Figura 4.15). Utilizatorilor care nu sunt conectați nu le este permis accesul, fiindu-le afișată o alertă de tip toast care îi anunță că această acțiune necesită un cont de utilizator.

Butonul etichetat *Main Page* din colțul stânga sus redirecționează utilizatorul înapoi către pagina cu lista de autovehicule. La deschiderea paginii se afișează imaginea *thumbnail*, adică o imagine cu rezoluție scăzută, iar utilizatorul poate viziona detaliile în timp ce restul imaginilor se încarcă într-un request separat. După încărcarea imaginilor acestea pot fi vizualizate într-o galerie extinsă (Figura 4.16), prin apăsarea pe containerul în care se află imaginea.

Butoanele din colțul din stânga jos sunt diferite în funcție de rolul utilizatorului conectat. Un utilizator cu rol de User are opțiunea să facă o programare pentru vehiculul selectat, apăsând pe butonul *Make an appointment* ce deschide dialogul din figura 4.17. Un utilizator cu rol de Admin poate marca un vehicul ca vândut – butonul *Set sold* -, modifica toate detaliile acestuia printr-un dialog similar cu cel din Figura 4.14 – butonul *Modify Vehicle* - sau poate modifica imaginile acestuia – butonul *Modify Images* - prin intermediul unui dialog care permite ștergerea individuală și adăugarea de imagini noi (Figura 4.18). O limitare a acestui dialog este inabilitatea de a reordona imaginile.

Figura 4.16 – Galeria de vizualizare a imaginilor

Figura 4.15 – Pagina cu detaliile unui autovehicul (/main/vehicles/view/:id)

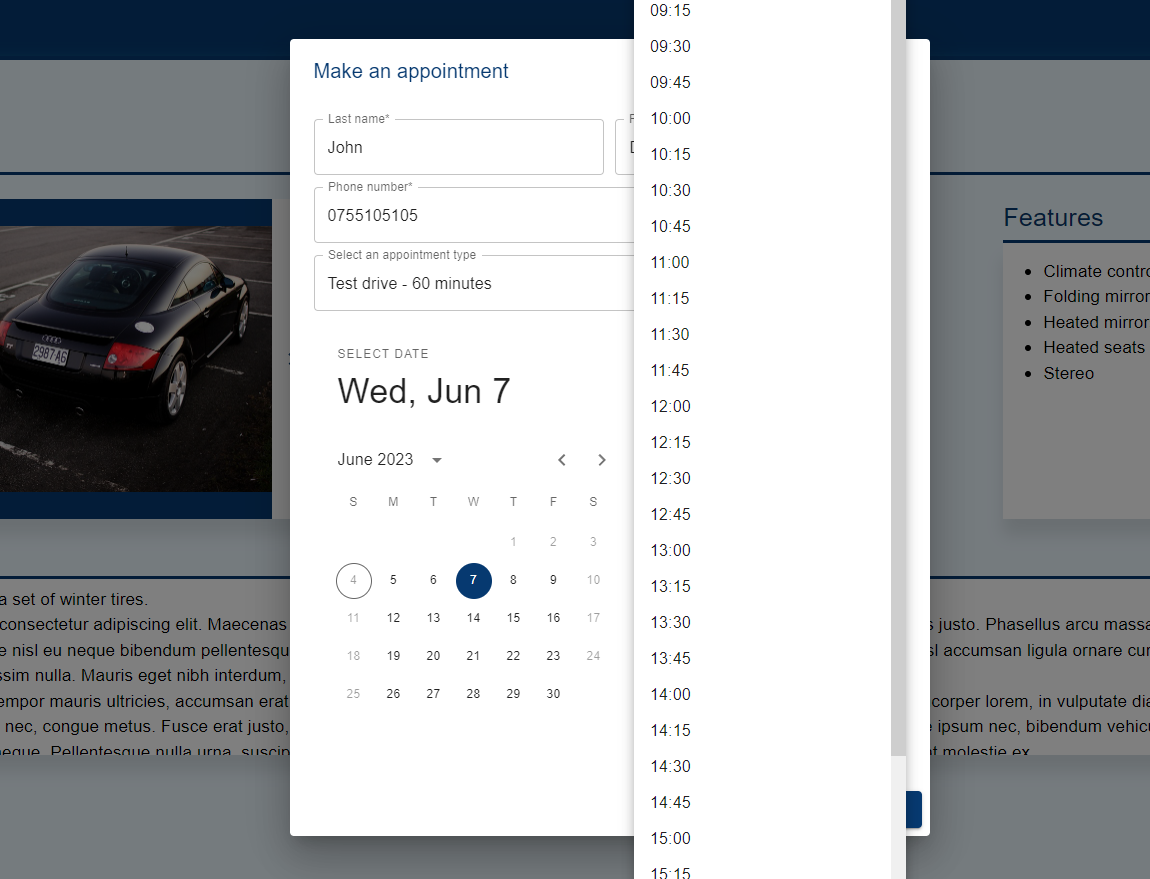
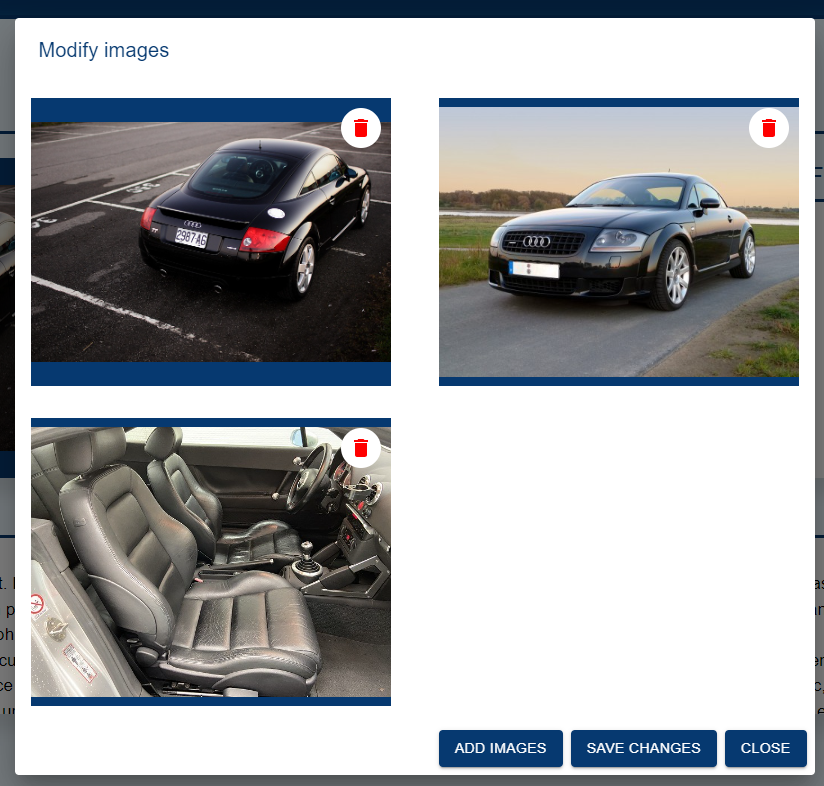
****

Figura 4.17 – Dialogul de efectuare al unei programări

Figura 4.18 – Dialogul de modificare a imaginilor unui vehicul

**Capitolul V**

# Concluzii

Consider că soluțiile găsite pentru rezolvarea problemelor descrise în capitolul I sunt satisfăcătoare, dar nu perfecte. Algoritmul de recomandare face recomandări satisfăcătoare, luând în calcul criteriile propuse. Aplicația este ușor extensibilă și se poate mula pe posibilele schimbări din cadrul pieței, fiind nevoie doar de adăugarea unor informații în interfață sau, în cel mai rău caz, adăugarea unor noi valori în cadrul enumerațiilor. Metodele de filtrare funcționează corespunzător dar ar putea fi îmbunătățite prin adăugarea mai multor criterii, cum ar fi filtrarea după dotări.

Atât frontend-ul cât și backend-ul rulează rapid, dar nu s-au efectuat teste la scară largă și cu seturi de date mai mari. Așadar, este posibil ca aplicația să nu fie scalabilă. Principalele posibile puncte de eșec sunt metodele de filtrare și algoritmul de recomandare.

Pe partea de backend, procesul dezvoltării în ASP.NET este ușurat de capabilitățile de debug incluse în Visual Studio Community. Problemele și erorile sunt ușor de identificat și rezolvat cu ajutorul breakpoint-urilor. Rularea pe thread-uri multiple este trivială, fiind nevoie doar de crearea unei funcții asincrone și rularea acesteia în interiorul unui *Task.*

Pe partea de frontend, consider că variabilele de stare reprezintă cel mai mare avantaj al React deoarece se execută o re-randare automată a componentelor atunci când valoarea acestora se schimbă. Alte framework-uri necesită efectuarea manuală a acestui proces. Acțiunea de debugging este posibilă doar prin instalarea unor pachete adiționale și nu există la fel de multe unelte ca cele incluse implicit în Visual Studio, pe partea de backend.

În realizarea acestei lucrări am învățat concepte importante despre multithreading și comunicarea dintre frontend și backend prin intermediul request-urilor HTTP. Pe viitor este necesar să-mi extind cunoștințele de scriere a testelor automate, toată testarea din cadrul proiectului fiind făcută manual.

**Bibliografie**

[[1]](#_Instalare) <https://visualstudio.microsoft.com/vs/community/> (accesat la 23.05.2023)

[[2]](#_Microsoft_SQL_Server) <https://learn.microsoft.com/en-us/ef/> (accesat la 24.05.2023)

[[3]](#_Microsoft_SQL_Server) <https://www.microsoft.com/en-us/sql-server/sql-server-downloads> (accesat la 24.05.2023)

[[4]](#_Microsoft_SQL_Server) <https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms> (accesat la 24.05.2023)

[5] <https://nodejs.org/en> (accesat la 24.05.2023)

[6] <https://code.visualstudio.com> (accesat la 25.05.2023)

[7] M. Jones, D. Hardt, *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. RFC 6750, Octombrie 2012, <https://www.rfc-editor.org/rfc/pdfrfc/rfc6750.txt.pdf>

[8] <https://support.google.com/accounts/answer/6010255> (accesat la 01.06.2023)

[9] J. B. Schafer, D. Frankowski, J. Herlocker, S. Sen, *Collaborative Filtering Recommender Systems*, The Adaptive Web, LNCS 4321, 291-324, Ianuarie 2007

Anexa 1

Diagrama Entitate-Relație

