**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**

[Introducere 5](#_Toc136348787)

[1.1 Prezentarea generală a aplicației 5](#_Toc136348788)

[1.1.1 Descrierea backend-ului 5](#_Toc136348789)

[1.1.2 Descrierea frontend-ului 5](#_Toc136348790)

[1.2 Starea curentă a pieței auto 6](#_Toc136348791)

[1.3 Scopul și motivația alegerii temei 6](#_Toc136348792)

[1.4 Structura lucrării 6](#_Toc136348793)

[1.5 Termeni 7](#_Toc136348794)

[Limbaje și tehnologii 7](#_Toc136348795)

[2.1 Visual Studio Community 7](#_Toc136348796)

[2.1.1 Instalare 7](#_Toc136348797)

[2.1.2 Pachete necesare 8](#_Toc136348798)

[2.2 ASP.NET și C# 9](#_Toc136348799)

[2.3 Microsoft SQL Server și SSMS 10](#_Toc136348800)

[2.4 Node.JS și React 10](#_Toc136348801)

[2.4.1 Pachete auxiliare 10](#_Toc136348802)

[2.5 Visual Studio Code 11](#_Toc136348803)

[2.6 TypeScript și SCSS 12](#_Toc136348804)

[Implementarea Web API-ului 13](#_Toc136348805)

[3.1 Structura soluției 13](#_Toc136348806)

[3.1.1 Repository pattern 13](#_Toc136348807)

[3.1.2 Specification pattern 15](#_Toc136348808)

[3.1.3 Structura directoarelor 16](#_Toc136348809)

[3.2 Baza de date 16](#_Toc136348810)

[3.3 Securitatea 19](#_Toc136348811)

[3.3.1 Stocarea informațiilor sensibile 19](#_Toc136348812)

[3.3.2 Autentificarea 19](#_Toc136348813)

[3.3.3 Vulnerabilități 21](#_Toc136348814)

[3.4 Controllere și funcționalitate 21](#_Toc136348815)

[Implementarea Frontend-ului 22](#_Toc136348816)

[Concluzii 23](#_Toc136348817)

**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.

## Structura lucrării

NEFINAL, va fi modificat ulterior

Lucrarea este structurată în patru capitole, conținutul fiecăruia fiind prezentat în continuare.

Primul capitol conține o descriere succintă a aplicației, starea curentă a pieței pe care se bazează funcționalitatea, scopul lucrării și motivația alegerii temei.

În al doilea capitol sunt descrise limbajele, tehnologiile și programele utilizate în implementarea aplicației, cât și motivația alegerii acestora. De asemenea, sunt expuse alternative viabile.

Al treilea capitol prezintă tehnicile de implementare și structura codului.

În al patrulea capitol sunt descrise concluziile.

## Termeni

Scopul acestui subcapitol este explicarea termenilor folosiți din limba engleză, care nu au neapărat un sinonim direct în limba română.

**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 debugger 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.

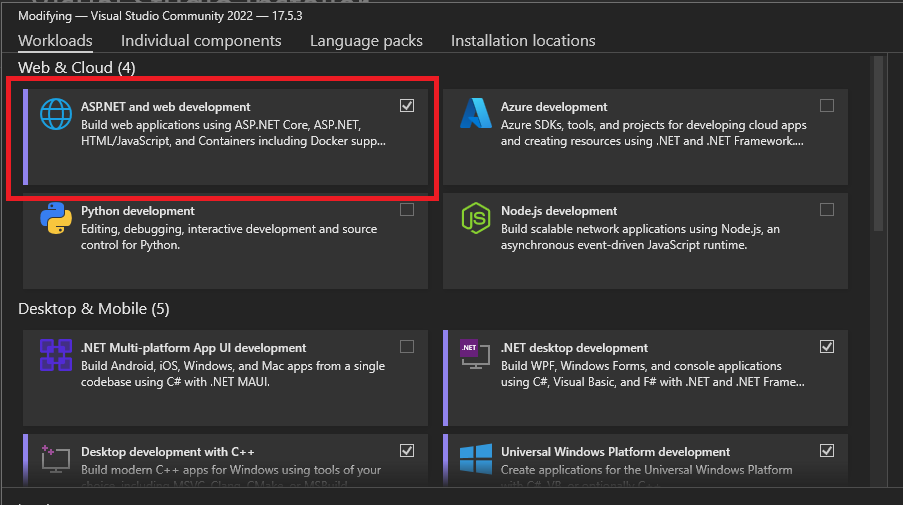


Figura 2.2 – Alegerea unui profil de distribuire

### 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.

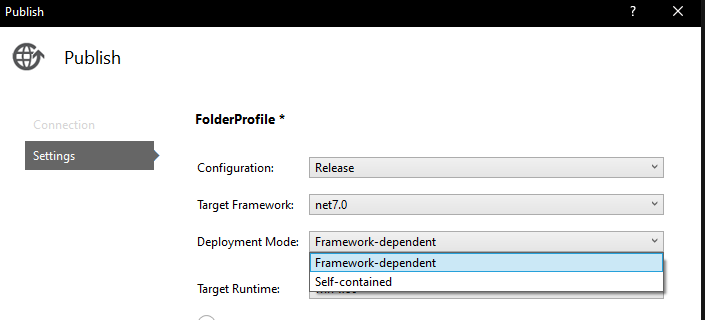
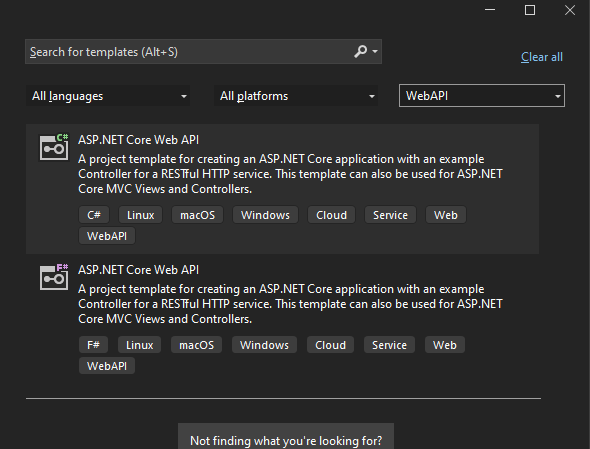
C# este un limbaj de programare orientat pe obiecte cu sintaxa similară limbajului Java, fiind unul dintre cele mai populare limbaje la momentul actual. Am ales acest limbaj în favoarea lui F# deoarece Microsoft pune la dispoziție mai multe resurse de învățare în documentația oficială.

Figura 2.2 – Alegerea unui profil de distribuire

Figura 2.3 – Șabloanele în limbajele C# și F#

## 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). 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.

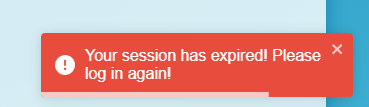
Folosind react-toastify experiența utilizatorilor poate fi îmbunătățită prin afișarea unor notificări de tip toast. O astfel de notificare este exemplificată mai jos, în figura 2.4.

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.

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; }

}

În cazul precedent nu este necesară nicio configurare suplimentară, Entity framework deducând automat anumite detalii. Cheia externă va fi dedusă 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 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 mai sus 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 ramuri 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

**Capitolul IV**

# Implementarea Frontend-ului

**Capitolul IV**

# Concluzii

**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>

Anexa 1

