UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

APLICAȚIE DE TIP WEBSITE PENTRU OPTIMIZAREA RUTELOR

Absolvent

Neacșu Vlad

Coordonator științific

Titlul și Radu Mincu

București, iunie 2023

Rezumat

Abstract

Cuprins

Introducere

1.1 Scopul Aplicației

Această aplicație a fost făcută cu scopul de a crea și a optimiza un traseu, care poate fi parcurs de angajații unei firme. Aceasta analizează contractele firmei respective și ia în vedere valoarea acestora, data lor de expirare și distanța pentru a ajunge la locația unui contract. În același timp, ține cont de numărul de angajați/echipe și specializarea acestora pentru a determina cel mai bun mod de a aloca angajați/echipe pentru onorarea unui contract. Analizând aceste elemente, aplicația ar trebui să creeze o rută care oferă cel mai bun raport **distanță - valoare contract** pentru fiecare angajat/echipă, care ia în vedere specializarea acestora. Specializarea se referă la cea mai complexă sarcină, care poate fi asignată unui angajat/echipă. Angajatul/Echipa respectivă poate să fie asignată oricărui contract cu o dificultate mai mică sau egală specializării acesteia.

1.1.1 Exemplu pentru utilizarea specializărilor

Specializare Necesară Contract	Specializare Echipă
Contract 1 - Generală	Echipa 1 - Specială
Contract 2 - Specială	Echipa 2 - Avansată
Contract 3 - Medie	Echipa 3 - Medie

În exemplul de mai sus ierarhia specializărilor este următoarea:

Generală < Medie < Specială

Având în vedere această ierarhie, contractele pot fi repartizate în modul următor:

Ecipa 1 - Contract 1

Echipa 2 - Contract 1 sau Contract 2 sau Contract 3

Ehipa 3 - Contract 1 sau Contract 3

1.2 Context

Dezvoltarea acestui proiect a avut loc în contextul mediului de afaceri al unei firme. Mai specific aplicația prezentată încearcă să optimizeze organizarea pentru o firma care oferă servicii de curățenie.

În primul rând, în cadrul unei asemenea firme este important să fie repartizate într-un mod cât mai eficient echipele de curățare, deoarece acestea au diferite specializări. De exemplu, există echipe care se ocupă de curățări generale (clădiri de birouri, blocuri, mall-uri) și echipe care se ocupă de curățări speciale (deșeuri toxice, curățenie pe un șantier de construcții). Diferența dintre aceste echipe o face complexitatea echipamentelor și nivelul de instruire al angajaților. În același timp, numărul de echipe specializate este în general mai mic față de echipele de curățenie generală. Având în vedere aceste lucruri, managerul unei firme de curățări și-ar dori să utilizeze cât se poate de des echipele specializate strict pentru sarcinile complexe, fără să le piardă timpul cu sarcini mai simple.

În al doilea rând, trebuie să fie minimizat timpul pierdut pe drum spre o locație care necesită curățenie. Acest detaliu este relevant, deoarece timpul pierdut pe drum reprezintă în esență, timp pierdut pentru angajați și indirect pentru firmă. Dacă o echipa se află mai mult timp pe drum aceasta se ocupă de mai puține sarcini și este inaccesibilă mai mult timp în caz că apare o sarcina mai profitabilă.

În ultimul rând, ca în cazul oricărei firme, profitul trebuie maximizat. Ținând cont de precizările anterioare profitul acestei firme v-a crește indirect. Dacă echipele pierd mai puțin timp pe drum, acestea au timp să se ocupe de mai multe sarcini, mărind profitul. În același timp, dacă echipele sunt repartizate cât mai eficient conform specializărilor lor, numărul de sarcini speciale care pot fi onorate v-a crește. Acest lucru este relevant pentru creșterea profitului, deoarece în medie aceste sarcini sunt mai bine plătite.

Scopul aplicației, așa cum a fost prezentat la punctul anterior, este să rezolve aceste probleme, în plus, modelul și ideile principale ale acestei aplicații pot fi generalizate pentru a fi utilizată și în cadrul altor tipuri de firme (de exemplu: firme de livrări, firme de construcții).

În contextul dat, aplicația ar trebui să ofere un mediu pentru managerii unei firme în care pot introduce posibile contracte și echipele pe care le au la dispoziție în firmă. Aceste date odată introduse reprezintă baza pe care aplicația își desfășoară activitatea de eficientizare. În urma acestei optimizări traseul cel mai bun este prezentat managerului.

1.3 Aplicații similare și contribuții propuse

Proiectul prezentat se afla în competiție directa cu alte aplicații de planificare a rutelor cum ar fi: Verizon Connect, Route4Me sau Onfleet [1]. Acestea oferă ca și in cazul aplicației mele o modalitate pentru coordonatorii unei firme să-și planifice rutele în avans pentru a obține trasee cât mai eficiente. În plus acestea oferă și metode de localizare a flotelor lor de mașini, lucru care nu v-a fi prezent în cadrul acestui proiect.

Verizon Connect este o aplicație inovatoare de urmărire a flotelor de mașini a unei firme. Aceasta pune la dispoziție utilizatorilor diferite metode de a își monitoriza angajații pe drum și de a urmări performanța logisticii firmei lor [2].

Route4Me permite utilizatorilor să-și creeze o listă cu clienții lor și cu numărul curent de șoferi pe care îi au la dispoziție. Aplicația, utilizând aceste date, creează rută cea mai optimă pentru a economisi timp și combustibil [3].

Onfleet pune la dispoziție utilizatorilor o gamă largă de API-uri, care permit integrarea componentelor firmei lor. Acest mod de integrare eficient, împreună cu elementele specifice unei aplicații de monitorizare și optimizare a rutelor oferă utilizatorilor un mod eficient de a-și organiza firma [4].

Această aplicație își propune să aducă noi contribuții în domeniul optimizării rutelor. Elementul principal pe care dorește să-l introduc este optimizarea în funcție de specializarea unei echipe. Știind nivelul de pregătire și complexitatea echipamentelor pe care echipele unei firme le au la dispoziție pot ajuta la o determinare mai eficientă a rutelor.

În acest sens traseul unei echipe cu nivel de specializare ridicat ar prioritiza sarcinile complexe pe care le are la dispoziție firma. După ce aceste sarcini ar fi încheiate cu succes, echipa ar continua să se ocupe de ce a rămas.

Acest mod de optimizare al rutelor este mai eficient din mai multe puncte de vedere.

Asignarea eficientă a sarcinilor conform nivelului de pregătire necesar, ar reduce pierderile de timp și de bani. Dacă echipele specializate se ocupă în mod constant de sarcinile simple, este posibil ca atunci când firma are ocazia să accepte un contract care necesită un nivel de expertiză mai ridicat, să nu aibă echipe cu nivelul de pregătire necesar libere să se ocupe de sarcină. În același timp, apare posibilitatea ca echipele cu un nivel scăzut de pregătire să nu aibă contracte de care să se ocupe, acestea fiind deja asignate echipelor specializate. Scăpând de aceste probleme aplicația eficientizează logistica firmei, scăpând de posibile suprapuneri în organizarea echipelor și asigurând de cele mai multe ori faptul că existe

echipe libere să se ocupe de orice posibilă sarcină care poate apărea. În acest fel aplicația reduce timpul pierdut în rezervă și crește profitul firmei.

1.4 Motivație

Dezvoltarea acestui proiect a avut mai multe motive. Prima este dorința de mă îmbunătății ca programator pe cât mai multe planuri. În acest sens aplicație mea conține elemente de frontend, backend și algoritmică, regăsite in tehnologiile pe care le-am utilizat. A doua motivație a fost reprezentată de discuția cu un prieten care lucrează în domeniul firmelor care oferă servicii de curățenie. Acesta mi-a prezentat neajunsurile logistice din firma lui, care m-au determinat sa mă gândesc la o aplicație care ar putea să le rezolve.

Descrierea Aplicației

2.1 Backend

Backend-ul aplicației este dezvoltat pe baza framework-ului ASP.NET Core, folosind C#. ASP.NET Core este un framework, cross-platform, performant și open-source, utilizat pentru construirea aplicațiilor moderne, conectate la cloud și la internet. ASP.NET Core permite construirea aplicațiilor web, aplicații de tip IoT și backend-uri pentru aplicații de tip mobile. Acesta pune la dispoziție programatorilor tool-uri pentru a dezvolta proiecte în Windows, macOS și Linux [5].

Structura backend-ului este separată în mai multe folder-e pe baza funcționalității fișierelor pe care le conțin.

2.1.1 Entities

Acest folder conține toate fișierele de tip clasă, DTO (Data Transfer Object), și rolurile pentru utilizatorii aplicației. Fișierele de tip clasă reprezintă baza pentru dezvoltarea aplicației (Contract. Employee, Location, Role, Team, SessionToken, User, UserRole). Acestea sunt folosite pentru a reține informații sub forma dorita ca mai apoi să fie stocate într-o bază de date. Datele stocate pot fi după utilizate pentru orice altă nevoie în cadrul aplicației. Fișierele de tip DTO care vin în completarea claselor (ContractDTO, TeamDTO, EmployeeDTO, LocationDTO, LoginUserDTO, RegisterUserDTO) și CreateDTO care sun folosite în fișierele de tip controller, în metodele de POST (CreateContractDTO, CreateTeamDTO, CreateEmployeeDTO, CreateLocationDTO, CreateUserDTO). Acestea au rolul de a încapsula datele sub forma dorită pentru a fi transmise mai departe prin rețea.

Contract

Clasa Contract conține informații importante legate de un job. Aceasta stochează suma de bani câștigată în urma terminării unui job, perioada în care este valabil contractul, tipul contractului și id-ul unui user (managerul care se ocupă de contract). În plus aceasta creează referințe către clasele User și Location, care vor reprezenta relațiile în baza de date. Referința către Location este în special importantă, deoarece aplicația trebuie să știe locațiile aferente unui contract pentru a putea crea ruta optimizată.

STRUCTURĂ CLASEI CONTRACT		
Id	Public int	Reține id-ul contractului. Relevant pentru o stocare corectă în baza de date
StartDate	Public DateTime	Reține data de start a job-ului
FinishDate	Public DateTime	Reține data de incheiere a job-ului
Value	Public float	Suma de încasat la încheierea cu succes a contractului
IdUser	Public int	Reține id-ului managerului care se ocupă de contract
JobType	Public String	Reține gradul de pregătire pe care trebuie să-l aibă o echipă pentru a putea fi asignată acestui contract
User	Public virtual User	Relație de tip Many to One către clasa User
Locations	Public ICollection<Location>	Relație de tip One to Many către clasa Location

Location

Clasa Location conține informații legate de locația în care se desfășoară job-ul aferent unui contract. Aceasta stochează orașul și adresa locului în care trebuie onorată sarcina și id-ul contractului. În plus aceasta creează o referință spre clasa contract.

STRUCTURĂ CLASEI LOCATION		
Id	Public int	Reține id-ul locației. Relevant pentru o stocare corectă în baza de date
City	Public string	Reține orașul în care se desfășoară sarcina
Address	Public string	Reține adresa la care se desfășoara sarcina
IdContract	Public int	Suma de încasat la încheierea cu succes a contractului
Contract	Public virtual Contract	Relație de tip Many to One către clasa Contract

Team

Clasa Team conține informații legate de o echipă. Aceasta stochează informații legate de nivelul de pregătire al unei echipe și dacă aceasta este disponibilă pentru a fi trimisă să se ocupe de un job. În plus aceasta creează referințe către clasele User și Employee. Referința către Employee este importantă, deoarece aplicația trebuie să știe câți angajați conține o echipă pentru a determina dacă aceasta operează la capacitate optimă.

STRUCTURĂ CLASEI TEAM		
Id	Public int	Reține id-ul echipei. Relevant pentru o stocare corectă în baza de date
JobType	Public string	Reține specializarea / nivelul de pregătire al echipei
Availability	Public int	Reține 1 dacă echipa este disponibilă sau 0 în caz contrar
IdUser	Public int	Reține id-ului managerului care se ocupă de echipă
User	Public virtual User	Relație de tip Many to One către clasa User
Locations	Public ICollection<Employee>	Relație de tip One to Many către clasa Employee

Employee

Clasa Employee conține informații legate de un angajat. Aceasta stochează numele de familie și prenumele unui angajat. În plus creează o referință spre clasa Team.

STRUCTURĂ CLASEI EMPLOYEE		
Id	Public int	Reține id-ul angajatului. Relevant pentru o stocare corectă în baza de date
LastName	Public string	Reține numele de familiei al angajatului
Name	Public string	Reține prenumele angajatului
IdTeam	Public int	Reține id-ului echipei de care aparține angajatul
Team	Public virtual Contract	Relație de tip Many to One către clasa Team

User

Clasa User conține informații legate de contul pe care îl utilizează un manager și extinde clasa IdentityUser pusă la dispoziție de ASP.NET Core prin componenta Identity [6]. Aceasta stochează data la care a fost creat contul, numele și parola utilizatorului. În plus aceasta creează referințe către clasele Contracts, Teams și UserRoles. Referințele către Contracts și Teams sunt importante, deoarece prin acestea determinăm de ce echipe și contracte se ocupă un manager.

UserRoles este un tabel asociativ care face legătura cu clasa Roles, care extinde clasa IdentityRoles. Roles este utilizată pentru a determina ce rol are un manager în cadrul unei rețele. De exemplu putem avea rolurile manager local și manager regional. Conectând clasele Roles și User putem determina nivelul privilegiilor pe care îl are un cont.

STRUCTURĂ CLASEI USER		
Public User(): base()	Inițializează o nouă instanță a clasei IdentityUser<TKey>	
CreationDate	Public DateTime	Reține data la care a fost creat contul
Name	Public string	Reține numele utilizat de manager pentru a intra în cont
Password	Public string	Reține parola utilizată de manager pentru a intra în cont
Contracts	Public ICollection<Contract>	Relație de tip One to Many către clasa Contracts
Teams	Public ICollection<Teams>	Relație de tip One to Many către clasa Teams
UserRoles	Public ICollection<UserRole>	Relație de tip One to Many către clasa UserRole

2.1.2 Repositories

Acest folder conține toate fișierele de tip repository și Irepository. În fișierele de tip repository (ContractRepository, EmployeeRepository, LocationRepository, TeamRepository, UserRepository, SessionTokenRepository, GenericRepository, RepositoryWrapper) sunt implementate funcțiile care interacționează cu baza de date. Acestea vor fi utilizate mai departe în restul backend-ului. Fișierele Irepository (IContractRepository, IEmployeeRepository, ILocationRepository, ITeamRepository, IUserRepository, ISessionTokenRepository, IGenericRepository, IRepositoryWrapper) sunt interfețe care conțin definițiile funcțiilor menționate anterior. Acest mod de organizare produce un cod mult mai ușor de refolosit, care poate folosi Dependency Injection, care poate utiliza Unit Testing și care este mult mai lizibil din cauza faptului că urmează un design comun între toate fișierele.

Acest folder conține două elemente care se remarcă prin scopul lor diferit. GenericRepository, nu este utilizat direct nicăieri în cadrul aplicației, în schimb acesta conține funcții de esențiale pentru comunicarea cu baza de date. RepositoryWrapper, are rolul de a încapsula funcțiile din fișierele repository pentru a fi folosite mai departe în restul backend-ului.

STRUCTURA FUNCȚIILOR CONTRACT REPOSITORY		
GetAllContracts	Public async Task<List<Contract>>	Trimite un query bazei de date, care întoarce toate contractele deja stocate
GetContractsById	Public async Task<Contract>	Trimite un query bazei de date, care întoarce un singur contract specific id-ului său dat ca parametru (int Id)
GetContractsById	Public async Task<List<Contract>>	Trimite un query bazei de date, care întoarce toate contractele supervizate de managerul al cărui id este dat ca parametru (int IdUser)

STRUCTURA FUNCȚIILOR LOCATION REPOSITORY		
GetAllLocations	Public async Task<List<Location>>	Trimite un query bazei de date, care întoarce toate locațiile deja stocate
GetLocationsById	Public async Task<List<Location>>	Trimite un query bazei de date, care întoarce toate locațiile specifice contractului al cărui id este dat ca parametru (int Id)
GetAllLocationsByCity	Public async Task<List<Location>>	Trimite un query bazei de date, care întoarce toate locațiile care se află în același oraș al cărui nume este dat ca parametru (string City)

STRUCTURA FUNCȚIILOR TEAM REPOSITORY		
GetAllEmployees	Public async Task<List<Employee>>	Trimite un query bazei de date, care întoarce toți angajații deja stocați
GetAllEmployeesById	Public async Task<List<Employee>>	Trimite un query bazei de date, care întoarce toți angajații specifici echipei al cărei id este dat ca parametru (int IdTeam)

STRUCTURA FUNCȚIILOR TEAM REPOSITORY		
GetAllTeams	Public async Task<List<Team>>	Trimite un query bazei de date, care întoarce toate echipele deja stocate
GetTeamById	Public async Task<Team>	Trimite un query bazei de date, care întoarce o singură echipă specifică id-ului ei dat ca parametru (int Id)
GetAllTeamsById	Public async Task<List<Team>>	Trimite un query bazei de date, care întoarce toate echipele supervizate de managerul al cărui id este dat ca parametru (int IdUser)
GetAllTeamsByFunction	Public async Task<List<Team>>	Trimite un query bazei de date, care întoarce toate echipele care au același nivel de pregătire, care este specificat ca parametru (string JobType)

STRUCTURA FUNCȚIILOR TEAM REPOSITORY		
GetAllTeams	Public async Task<List<Team>>	Trimite un query bazei de date, care întoarce toate echipele deja stocate
GetTeamById	Public async Task<Team>	Trimite un query bazei de date, care întoarce o singură echipă specifică id-ului ei dat ca parametru (int Id)
GetAllTeamsById	Public async Task<List<Team>>	Trimite un query bazei de date, care întoarce toate echipele supervizate de managerul al cărui id este dat ca parametru (int IdUser)
GetAllTeamsByFunction	Public async Task<List<Team>>	Trimite un query bazei de date, care întoarce toate echipele care au același nivel de pregătire, care este specificat ca parametru (string JobType)

STRUCTURA FUNCȚIILOR SESSIONTOKEN REPOSITORY		
GetByJTI	Task<SessionToken>	Trimite un query bazei de date, care întoarce token-ul care corespunde identificatorului JTI dat ca parametru (string JTI)

STRUCTURA FUNCȚIILOR USER REPOSITORY		
GetAllUsers	Public async Task<List<User>>	Trimite un query bazei de date, care întoarce toți utilizatorii deja stocați
GetByIdWithRoles	Public async Task<User>	Trimite un query bazei de date, care întoarce un singur utilizator specificat prin id-ul lui dat ca parametru (int Id), luând în considerare și rolul acestuia
GetUserByEmail	Public async Task<User>	Trimite un query bazei de date, care întoarce primul user al cărui mail corespunde cu cel da ca parametru (string Email)
GetUserById	Public async Taskt<User>	Trimite un query bazei de date, care întoarce un singur utilizator specificat prin id-ul lui dat ca parametru (int Id)

STRUCTURA FUNCȚIILOR GENERIC REPOSITORY		
Create	public void	Creează o entitate temporară (TEntity entity) în baza de date
CreateRange	public void	Creează mai multe entități temporare (IEnumerable<TEntity> entities) în baza de date
Delete	public void	Șterge o entitate (TEntity entity) din baza de date (schimbarea nu se salvează automat)
DeleteRange	public void	Șterge mai multe entități (IEnumerable<TEntity> entities) din baza de date (schimbarea nu se salvează automat)
GetAll	public IQueryable<TEntity>	Trimite un query bazei de date, care întoarce toate entitățile stocate de tipul <TEnitity>
GetByIdAsync	public async Task<TEntity>	Trimite un query bazei de date, care întoarce o entitate specifică id-ului dat ca parametru (int id), de tipul <TEnitity>

GetByNameAsync	public async Task<TEntity>	Trimite un query bazei de date, care întoarce o entitate care stochează în ea numele dat ca parametru (int name), de tipul <TEnitity>
Update	public void	Modifică o entitate (TEntity entity) din baza de date (schimbările nu sunt permanente)
SaveAsync	public async Task<bool>	Salvează modificările făcute în funcțiile anterioare (permanent)

2.1.3 Controllers

Acest folder conține toate fișierele de tip controller (ContractController, LocationController, TeamController, EmployeeController ,UserController, AccountController) . Un controler este folosit pentru a defini și a grupa un set de acțiuni. O acțiune este o metodă dintr-un controller care gestionează cereri. Controller-ele separă acțiunile în grupuri cu scop similar. Această grupare a acțiunilor permite ca seturi comune de reguli, cum ar fi rutarea, stocarea în cache și autentificarea, să fie aplicate colectiv. Solicitările sunt direcționate către acțiuni prin rutare. În cadrul modelului Model-View-Controller, un controller este responsabil pentru procesarea inițială a cererii și instanțierea modelului. Controlorul preia rezultatul procesării modelului (dacă există) și returnează fie vizualizarea adecvată și datele de vizualizare asociate acesteia, fie rezultatul apelului API. Controlerul este o abstractizare la nivel de UI. Responsabilitățile sale sunt să se asigure că datele cererii sunt valide și să aleagă ce vizualizare (sau rezultat pentru un API) ar trebui returnat [7].

Metodele publice de pe un controler, cu excepția celor cu atributul [NonAction], sunt acțiuni. Parametrii acțiunilor sunt obligați să solicite date și sunt validați folosind model binding. Acțiunile ar trebui să conțină o logică pentru maparea unei cereri. Acțiunile pot returna orice, dar returnează frecvent o instanță de IActionResult (sau Task<IActionResult> pentru metodele asincrone) care produce un răspuns [7].

ContractController

Controlerul conține mai multe funcții care corespund diferitelor endpoint-uri HTTP, endpoint-ul de bază fiind [Route("api/contract")]:

1. Constructor: Inițializează controlerul prin injectarea unei instanțe a interfeței IContractRepository, care este utilizată pentru accesarea și manipularea datelor contractului.

2. GetAllContracts [HttpGet]: Preia toate contractele apelând metoda GetAllContracts a instanței IContractRepository. Acesta convertește contractele preluate într-o listă de obiecte ContractDTO și le returnează ca răspuns HTTP.
3. GetContractById [HttpGet("{id:int}")]: Preia un anumit contract prin ID-ul său, folosind metoda GetContractById din IContractRepository. Acesta convertește contractul preluat într-un obiect ContractDTO și îl returnează ca răspuns HTTP.
4. GetAllContractsById [HttpGet("contractsById/{id}")]: Preia toate contractele asociate cu un anumit ID de utilizator. Apelează metoda GetAllContracts din IContractRepository și filtrează contractele pe baza proprietății IdUser. Contractele filtrate sunt apoi convertite într-o listă de obiecte ContractDTO și returnate ca răspuns HTTP.
5. DeleteContract [HttpDelete("{id}")]: Șterge un contract pe baza ID-ului său. Mai întâi preia contractul folosind metoda GetByIdAsync din IContractRepository. Dacă contractul nu există, acesta returnează un răspuns "Contract does not exist!". În caz contrar, șterge contractul utilizând metoda Delete din IContractRepository, salvează modificările asincron folosind metoda SaveAsync.
6. CreateContract [HttpPost]: creează un nou contract prin primirea unui obiect CreateContractDTO în corpul cererii. Acesta creează o nouă instanță a clasei Contract, îi populează proprietățile cu valori din obiectul DTO și o adaugă în baza de date folosind metoda Create. Apoi salvează modificările asincron folosind metoda SaveAsync și returnează contractul nou creat ca răspuns HTTP.
7. UpdateAsync [HttpPut("UpdateForForm")]: Actualizează un contract existent. Primește un obiect Contract în corpul cererii și preia toate contractele folosind metoda GetAllContracts a IContractRepository. Găsește indexul contractului de actualizat pe baza ID-ului său și îl înlocuiește cu contractul actualizat în vector. În cele din urmă, returnează matricea actualizată de contracte ca răspuns HTTP.

LocationController

Controlerul conține mai multe funcții care corespund diferitelor endpoint-uri HTTP, endpoint-ul de bază fiind [Route("api/location")]:

1. Constructor: Inițializează controlerul prin injectarea unei instanțe a interfeței IContractRepository, care este utilizată pentru accesarea și manipularea datelor locației.

2. GetAllLocations [HttpGet]: Preia toate locațiile apelând metoda GetAllLocations a instanței ILocationRepository. Acesta convertește contractele preluate într-o listă de obiecte LocationDTO și le returnează ca răspuns HTTP.
3. GetAllLocationsById [HttpGet("{id}")]: Preia toate locațiile asociate cu un anumit ID de contract. Apelează metoda GetAllLocations din ILocationRepository și filtrează locațiile pe baza proprietății IdContract. Locațiile filtrate sunt apoi convertite într-o listă de obiecte LocationDTO și returnate ca răspuns HTTP.
4. DeleteLocation [HttpDelete("{id}")]: Șterge o locație pe baza ID-ului său. Mai întâi preia locația folosind metoda GetByIdAsync din ILocationRepository. Dacă locația nu există, acesta returnează un răspuns "Location does not exist!". În caz contrar, șterge locația utilizând metoda Delete din ILocationRepository, salvează modificările asincron folosind metoda SaveAsync.
5. CreateLocation [HttpPost]: creează o nouă locație prin primirea unui obiect CreateLocationDTO în corpul cererii. Acesta creează o nouă instanță a clasei Location, îi populează proprietățile cu valori din obiectul DTO și o adaugă în baza de date folosind metoda Create. Apoi salvează modificările asincron folosind metoda SaveAsync și returnează contractul nou creat ca răspuns HTTP.
6. UpdateAsync [HttpPut("UpdateForForm")]: Actualizează o locație existentă. Primește un obiect Location în corpul cererii și preia toate contractele folosind metoda GetAllLocations ILocationRepository. Găsește indexul locației de actualizat pe baza ID-ului ei și o înlocuiește cu locația actualizată în vector. În cele din urmă, returnează matricea actualizată de contracte ca răspuns HTTP.

TeamController

Controlerul conține mai multe funcții care corespund diferitelor endpoint-uri HTTP, endpoint-ul de bază fiind [Route("api/team")]:

1. Constructor: Inițializează controlerul prin injectarea unei instanțe a interfeței ITeamRepository, care este utilizată pentru accesarea și manipularea datelor echipei.
2. GetAllTeams [HttpGet]: Preia toate echipele apelând metoda GetAllTeams a instanței ITeamRepository. Acesta convertește echipele preluate într-o listă de obiecte TeamDTO și le returnează ca răspuns HTTP.
3. GetTeamById [HttpGet("{id:int}")]: Preia o anumită echipă prin ID-ul ei, folosind metoda GetTeamById din ITeamRepository. Acesta convertește echipa preluată într-un obiect TeamDTO și îl returnează ca răspuns HTTP.

4. GetAllTeamsById [HttpGet("teamsById/{id}")]: Preia toate echipele asociate cu un anumit ID de utilizator. Apelează metoda GetAllTeams din ITeamRepository și filtrează echipele pe baza proprietății IdUser. Echipele filtrate sunt apoi convertite într-o listă de obiecte TeamDTO și returnate ca răspuns HTTP.
5. DeleteTeam [HttpDelete("{id}")]: Șterge o echipă pe baza ID-ului ei. Mai întâi preia echipa folosind metoda GetByIdAsync din ITeamRepository. Dacă echipa nu există, acesta returnează un răspuns "Team does not exist!". În caz contrar, șterge echipa utilizând metoda Delete din ITeamRepository, salvează modificările asincron folosind metoda SaveAsync.
6. CreateTeam [HttpPost]: creează o nouă echipă prin primirea unui obiect CreateTeamDTO în corpul cererii. Acesta creează o nouă instanță a clasei Team, îi populează proprietățile cu valori din obiectul DTO și o adaugă în baza de date folosind metoda Create. Apoi salvează modificările asincron folosind metoda SaveAsync și returnează echipa nou creată ca răspuns HTTP.
7. UpdateAsync [HttpPut("UpdateForForm")]: Actualizează o echipă existentă. Primește un obiect Team în corpul cererii și preia toate echipele folosind metoda GetAllTeams a ITeamRepository. Găsește indexul echipei de actualizat pe baza ID-ului ei și o înlocuiește cu echipa actualizată în vector. În cele din urmă, returnează matricea actualizată de contracte ca răspuns HTTP.

EmployeeController

Controlerul conține mai multe funcții care corespund diferitelor endpoint-uri HTTP, endpoint-ul de bază fiind [Route("api/employee")]:

1. Constructor: Inițializează controlerul prin injectarea unei instanțe a interfeței IEmployeeRepository, care este utilizată pentru accesarea și manipularea datelor contractului.
2. GetAllEmployees [HttpGet]: Preia toți angajații apelând metoda GetAllContracts a instanței IEmployeeRepository. Acesta convertește angajații preluați într-o listă de obiecte EmployeeDTO și le returnează ca răspuns HTTP.
3. GetAllEmployeesById [HttpGet("{id}")]: Preia toați angajații asociați cu un anumit ID de echipă. Apelează metoda GetAllEmployees din IEmployeeRepository și filtrează angajații pe baza proprietății IdTeam. Angajații filtrați sunt apoi convertiți într-o listă de obiecte EmployeeDTO și returnate ca răspuns HTTP.

4. DeleteEmployee [HttpDelete("{id}")]: Șterge un angajat pe baza ID-ului său. Mai întâi preia angajatul folosind metoda GetByIdAsync din IEmployeeRepository. Dacă angajatul nu există, acesta returnează un răspuns "Employees does not exist!". În caz contrar, șterge angajatul utilizând metoda Delete din IEmployeeRepository, salvează modificările asincron folosind metoda SaveAsync.
5. CreateEmployees [HttpPost]: creează un nou angajat prin primirea unui obiect CreateEmployeeDTO în corpul cererii. Acesta creează o nouă instanță a clasei Employee, îi populează proprietățile cu valori din obiectul DTO și o adaugă în baza de date folosind metoda Create. Apoi salvează modificările asincron folosind metoda SaveAsync și returnează contractul nou creat ca răspuns HTTP.

EmployeeController

Controlerul conține mai multe funcții care corespund diferitelor endpoint-uri HTTP, endpoint-ul de bază fiind [Route("api/user")]:

1. Interfețele IRepositoryWrapper și IUserService sunt introduse în controler prin dependency-injection de constructor. Acest lucru permite controller-ului să interacționeze cu baza de date și serviciul utilizatorului pentru a efectua diverse operații asupra entităților utilizator.
2. GetAllUsers [HttpGet]: Preia toți utilizatorii apelând metoda GetAllUsers a instanței IRepositoryWrapper. Acesta convertește utilizatorii preluați într-o listă pe care o returnează ca răspuns HTTP.
3. GetTeamById [HttpGet("{id:int}")]: Preia un anumit utilizator prin ID-ul lui, folosind metoda GetByIdAsync din IRepositoryWrapper. Acesta returnează utilizatorul preluat ca răspuns HTTP.

AccountController

Controlerul conține mai multe funcții care corespund diferitelor endpoint-uri HTTP, endpoint-ul de bază fiind [Route("api/account")]. Acesta este responsabil pentru gestionarea solicitărilor HTTP legate de conturile de utilizator, cum ar fi înregistrarea și autentificarea.

1. Instanțele UserManager<User> și IUserService sunt introduse în controler prin *dependency-injection* de constructor. UserManager<User> este furnizat de ASP.NET Core Identity [6] și permite operațiuni legate de gestionarea utilizatorilor, cum ar fi găsirea utilizatorilor prin e-mail. IUserService este o interfață de servicii personalizată care oferă funcționalități legate de înregistrarea și autentificarea utilizatorilor.

2. GetAcountByEmail [HttpGet("{email}")]: Se ocupă de solicitarea HTTP GET pentru a prelua un cont prin e-mail. Apelează metoda FindByEmailAsync din UserManager<User> pentru a găsi un utilizator cu e-mailul specificat și îl returnează într-un OkObjectResult.
3. Register [HttpPost("register")]: Se ocupă de cererea HTTP POST pentru înregistrarea utilizatorului. Mai întâi verifică dacă un utilizator cu e-mailul furnizat există deja apelând FindByEmailAsync din UserManager<User>. Dacă un utilizator există, acesta returnează un răspuns BadRequest care indică faptul că utilizatorul este deja înregistrat. În caz contrar, apelează metoda RegisterUserAsync din IUserService pentru a înregistra utilizatorul în mod asincron. Dacă înregistrarea are succes, returnează un OkObjectResult cu rezultatul. În caz contrar, returnează un răspuns BadRequest.
4. Login [HttpPost("login")]: Se ocupă de cererea HTTP POST pentru autentificarea utilizatorului. Apelează metoda LoginUser din IUserService pentru a valida acreditările utilizatorului și pentru a genera un token. Dacă autentificarea are succes și este generat un token, acesta returnează un OkObjectResult cu token-ul generat. În caz contrar, returnează un răspuns neautorizat care indică autentificarea eșuată.

2.1.4 Special Components

Pe lângă folderele specifice unei aplicații de tip MVC, backend-ul conține câteva elemente cu un caracter special care se ocupă de partea de autentificare din cadrul proiectului (Services, SeedDB, Helpers). Acestea conțin funcții care se ocupă de procesarea datelor din cadrul autentificărilor și al înregistrărilor. Metodele acestea sunt după utilizate mai departe în controller-e.

UserService din Services este responsabil pentru înregistrarea și autentificarea utilizatorilor. Se bazează pe componenta ASP.NET Core Identity [6] pentru gestionarea conturilor de utilizator. Aceasta are structura următoare:

1. Clasa are un constructor care ia doi parametri: UserManager<User> userManager și depozitul IRepositoryWrapper. Acești parametri sunt injectați în clasă utiliând depndency-injection. UserManager<User> este furnizat de cadrul ASP.NET Core Identity [6] și este utilizat pentru operațiuni legate de utilizator, în timp ce IRepositoryWrapper oferă acces la toate funcțiile din Repositories.
2. Metoda RegisterUserAsync: Această metodă ia ca parametru un obiect RegisterUserDTO, care conține informațiile necesare pentru înregistrarea unui utilizator. În cadrul metodei, o nouă instanță de utilizator este creată și populată cu datele din DTO. Instanța UserManager este apoi utilizată pentru a crea utilizatorul în mod asincron. Dacă crearea utilizatorului are

succes, utilizatorului i se atribuie rolul „Utilizator” folosind metoda AddToRoleAsync și este returnat true. În caz contrar, se returnează false.

3. Metoda LoginUser: Această metodă se ocupă de funcționalitatea de autentificare. Este nevoie de un obiect LoginUserDTO ca parametru, care conține e-mailul și parola utilizatorului care încearcă să se autentifice. În cadrul metodei, UserManager este folosit pentru a găsi utilizatorul prin e-mail. Dacă utilizatorul este găsit și parola furnizată se potrivește cu parola stocată, se efectuează generarea unui JSON Web Token și stocarea lui.
4. Metoda GenerateJwtToken: Această metodă este responsabilă pentru crearea unui JWT

SessionTokenValidator din Helper este responsabil pentru validarea token-urilor în timpul autentificării folosind JWT (JSON Web Token). În această clasă verifică dacă utilizatorul autentificat are o revendicare de tip jti (ID JWT). Apoi preia jetonul de sesiune asociat cu valoarea jti obținută din baza de date. Dacă un token de sesiune valid este găsit în baza de date și data de expirare a acestuia este mai mare decât data și ora curente (DateTime.Now), jetonul este considerat valid și metoda revine fără nicio acțiune ulterioară. Dacă jetonul nu este găsit în baza de date sau data de expirare a trecut, metoda apelează context.Fail("") pentru a indica că validarea jetonului a eșuat.

SeedDB din SeedDB este responsabil pentru introducerea rolurilor în baza de date folosind cadrul ASP.NET Core Identity. Verifică dacă există deja roluri în baza de date. Dacă rolurile sunt deja prezente, metoda revine fără a efectua nicio acțiune suplimentară. Dacă nu există roluri, este definită o serie de nume de roluri „Admin” și „Utilizator”.

2.1.5 Database

Pentru a stoca datele într-o formă permanentă, aplicația utilizează o baza de date. Programul folosit pentru gestionarea bazei de date este SSMS. Pentru a stoca informațiile în baza de date aplicația definește un set de reguli pentru a relațiile dintre entitățile prezentate anterior. Aceste relații sunt după utilizate în transformarea din procesul migrării pentru generarea unui set de comenzi. Comenzile rezultate reprezintă șablonul pe care este creată baza de date din SSMS.

Relațiile sunt definite în AppDbContext din folderul Data. Această clasă moștenește de IdentityDbContext<User, Role, int, IdentityUserClaim<int>, UserRole, IdentityUserLogin<int>, IdentityRoleClaim<int>, IdentityUserToken<int>>. Această moștenire asigură că modelul include tabelele și funcționalitățile necesare pentru ASP.NET Core Identity.

Procedeul de migrare din EF Core oferă o modalitate de a actualiza progresiv schema bazei de date pentru a o menține sincronizată cu modelul de date al aplicației, păstrând în același

timp datele existente în baza de date. Când se introduce o modificare a modelului de date, dezvoltatorul folosește instrumentele EF Core pentru a adăuga o migrare corespunzătoare care descrie actualizările necesare pentru a menține schema bazei de date sincronizată. EF Core compară modelul actual cu o instanță a modelului vechi pentru a determina diferențele și generează fișiere sursă de migrare. Odată ce o nouă migrare a fost generată, aceasta poate fi aplicată unei baze de date în diferite moduri. EF Core înregistrează toate migrările aplicate într-un tabel special de istoric, permițându-i să știe ce migrații au fost aplicate și care nu [8].

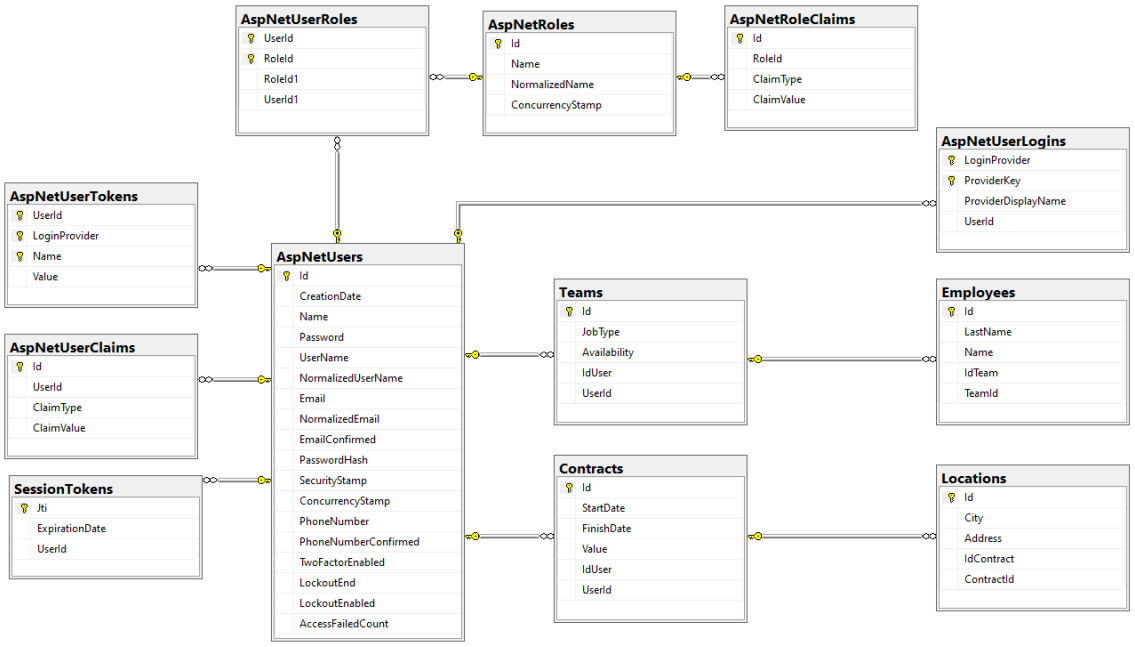

Fig. 1 diagrama UML

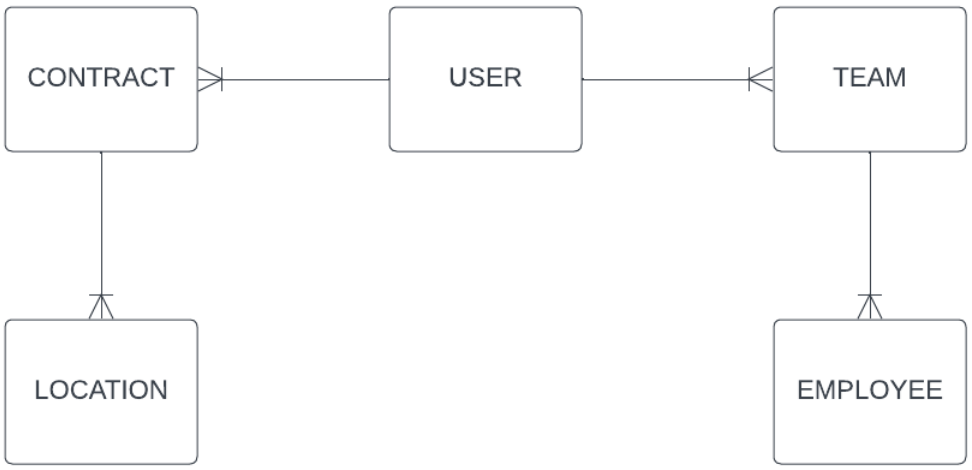

Fig. 2 diagrama relațiilor între entități

2.1.6 Startup

Clasa Startup este responsabilă pentru configurarea serviciilor de aplicație, a conductei de middleware și a altor setări necesare pentru rularea aplicației. Aceasta are următoarea structură:

1. Constructor: Clasa Startup are un constructor care preia un parametru IConfiguration. Este folosit pentru a prelua setările de configurare pentru aplicație.
2. ConfigureServices: Această metodă este apelată de runtime și este folosită pentru a configura serviciile pe care aplicația le va folosi. În cadrul acestei metode se realizează următoarele configurații și servicii:
   - Configurare CORS: CORS este activată folosind metoda services.AddCors. Permite originilor specifice (localhost:4200 și http://localhost:4200) să acceseze API-ul permițând orice metodă.
   - Dependency-Injection: Mai multe interfețe de depozit (ITeamRepository, IRepositoryWrapper, IUserService, IEmployeeRepository, ILocationRepository, IContractRepository) sunt înregistrate cu implementările corespunzătoare.
   - Configurare MVC: MVC (Model-View-Controller) este activat folosind services.AddControllers().
   - Configurare Swagger: Swagger este configurat folosind services.AddSwaggerGen pentru a genera documentație API pe baza specificației OpenAPI definite.
   - Configurare DbContext: AppDbContext este configurat să utilizeze o bază de date SQL Server cu șirul de conexiune furnizat în cod.
   - Configurarea identității: ASP.NET Core Identity este configurată folosind services.AddIdentity<User, Role>(). Specifică entitățile User și Role care vor fi utilizate cu AppDbContext și adaugă serviciile de identitate implicite.
   - Configurare serializare JSON: setările de serializare JSON sunt configurate pentru a ignora gestionarea buclei de referință folosind .AddNewtonsoftJson().
3. Configurare: Această metodă este apelată de runtime și este utilizată pentru a configura conducta de solicitări HTTP. În cadrul acestei metode, se aplică următoarele configurații și middleware:
   - Configurarea mediului de dezvoltare: Dacă mediul este setat la dezvoltare, aplicația folosește pagini de excepție pentru dezvoltatori și activează Swagger UI.
   - Redirecționare HTTPS: aplicația este configurată să redirecționeze cererile HTTP către HTTPS folosind app.UseHttpsRedirection().
   - Configurare rutare: rutarea este activată utilizând app.UseRouting().
   - Configurare autorizare: autorizarea este activată utilizând app.UseAuthorization().
   - Configurație CORS: CORS se aplică utilizând app.UseCors(SpecificOrigins).

- Configurare endpoint: punctele finale pentru aplicație sunt configurate folosind app.UseEndpoints() pentru a mapa controlerele.

Clasa Startup este o componentă crucială într-o aplicație ASP.NET Core, deoarece setează și configurează diverse servicii și middleware necesare pentru ca aplicația să funcționeze corect.

2.2 Frontend

Frontend-ul aplicației a fost dezvoltat utilizând Angular. Angular este o platformă de dezvoltare, construită pe TypeScript. Aceasta pune la dispoziție programatorilor mai multe beneficii:

1. Un framework modular pentru construirea de aplicații web scalabile
2. O colecție de biblioteci bine integrate care acoperă o mare varietate de caracteristici, inclusiv rutare, gestionare formularelor, comunicare client-server și multe altele
3. O multitudine de instrumente pentru dezvoltatori care ajută la dezvoltarea, construirea, testarea și actualizarea codului [9]

Structura frontend-ului este separată în mai multe element care sunt grupate în funcție de scopul pe care îl au.

2.2.1 Modules and Components

Modulele (contract-search, employee-edit, home, location-edit, login, material, register-user, team-search) sunt componenta principală a frontend-ului. Aplicațiile Angular sunt modulare și Angular are propriul său sistem de modularitate numit NgModules, care pune la dispoziție aceste elemente. Modulele sunt containere pentru un bloc coeziv de cod dedicat unui domeniu din aplicație, unui flux de lucru sau unui set de capabilități strâns legate. Ele conțin și componente, furnizori de servicii și alte fișiere de cod al căror domeniu este definit de modulul care le conține. Ele pot importa funcționalități care sunt exportate din alte module și pot exporta funcționalități selectate pentru a fi utilizate în alte module. În cadrul aplicației modulele sunt formate din mai multe elemente: app module, routing module și app component. App module oferă un context de compilare pentru componentele sale [10]. Routing module utilizează modelul de design lazy loading pentru încărcarea modelului. Acest design ajută la menținerea unei dimensiuni cât mai mici a pachetelor inițiale, ceea ce la rândul său ajută la reducerea timpilor de încărcare [11]. Structura modulelor este foarte asemănătoare la nivelul de app module și routing module.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { ContractSearchRoutingModule } from './contract-search-routing.module';
import { ContractSearchComponent } from './contract-search/contract-search.component';
import { MatCheckboxModule} from '@angular/material/checkbox';
import { MaterialModule } from '../material/material.module';
@NgModule({
  declarations: [
    ContractSearchComponent
  ],
  imports: [
    CommonModule,
    ContractSearchRoutingModule,
    MaterialModule,
    MatCheckboxModule
  ]
})
export class ContractSearchModule { }
```

Fig. 3 contract-search.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ContractSearchComponent } from './contract-search/contract-search.component';

const routes: Routes = [
  {
    path : '',
    redirectTo: 'contract-search',
  },
  {
    path : 'contract-search',
    component : ContractSearchComponent
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ContractSearchRoutingModule { }
```

Fig. 4 contract-search-routing.module.ts

App component este partea din modul care conferă funcționalitățile unei pagini. Componenta controlează o zonă de ecran numită view. Aceasta constă dintr-o clasă TypeScript, un șablon HTML și o foaie de stil SCSS. Clasa TypeScript definește interacțiunea șablonului HTML și structura DOM redată, în timp ce foaia de stil descrie aspectul acesteia [12]. Pe lângă acestea, în cadrul acestei aplicații, componenta conține un foldere cu imagini, SCSS și

JavaScript pentru a putea fișiere sursă mai ușor. Toate aceste elemente sunt utilizate pentru a defini și a controla diferitele aspecte ale aplicației.

Home

Această componentă gestionează funcționalități legate de geocodarea adreselor, generarea unei matrice de distanțe, calcularea rutei celei mai rapide și afișarea rezultatelor pe o hartă. Aceasta are următoarea structură:

1. Proprietăți principale:
   - addressVector: o matrice de obiecte care reprezintă adresele și alte date asociate.
   - distanceMatrix: o matrice de obiecte care reprezintă distanța și durata dintre locații.
   - visitedLocations: Un set pentru a ține evidența locațiilor vizitate.
2. Metode principale:
   - ngOnInit apelează metoda geocodeAddress().
   - geocodeAddress geocodifică asincron adresele utilizând API-ul Google Geocoding. Face solicitări HTTP pentru a prelua latitudinea, longitudinea și alte informații pentru fiecare adresă. Acesta populează matricea addressVector cu datele preluate.
   - generateDistanceMatrix generează în mod asincron o matrice de distanță utilizând API-ul Google Distance Matrix. Acesta calculează distanțele și duratele dintre locații. De asemenea, determină cea mai rapidă rută pentru fiecare echipă în funcție de tipul de job și afișează traseul pe o hartă folosind API-ul JavaScript Google Maps.
   - findClosest găsește cea mai apropiată locație de punctul curent pe baza anumitor criterii (tipul jobului, datele de începere și de încheiere, raportul distanta - valoare contract). Returnează indexul celei mai apropiate locații.
   - calculateFastestRoute calculează cea mai rapidă rută pentru o echipă pe baza tipului de job dat. Pornește de la locația inițială și găsește în mod iterativ cea mai apropiată locație până când sunt vizitate toate locațiile. Returnează o serie de indici reprezentând cea mai rapidă rută.
3. Interfețe:
   - addressLatLng: reprezintă structura de date pentru o adresă cu latitudine, longitudine și alte proprietăți asociate.
   - matrixElement: Reprezintă structura de date pentru un singur element din matricea de distanțe, inclusiv distanța și durata.
   - distanceMatrix: reprezintă structura de date pentru o locație din matricea distanțelor, inclusiv adresa, elementele asociate și alte proprietăți.

Această componentă conține logica principala a programului. Folosind toate informațiile stocate în baza de date, aceasta utilizează elementele descrise mai sus pentru a furniza managerului o imagine clară a traseelor optime pentru fiecare echipă. Aceasta folosește extensiv metode puse la dispoziție de Maps Javascript API:

1. Distance Matrix Service - calculează distanța și durata călătoriei între mai multe origini și destinații folosind un anumit mod de călătorie . [21]
2. Direction Service - calculează trasee (folosind o varietate de metode de transport) utilizând obiectul DirectionsService. Acest obiect comunică cu Google Maps API Directions Service care primește solicitărea și returnează o rută eficientă. Timpul de călătorie este factorul principal care este optimizat, dar pot fi luați în considerare și alți factori precum distanța, numărul de viraje și mulți alții. [22]
3. Geocoder Service - asigură conversia adreselor (cum ar fi „1600 Amphitheatre Parkway, Mountain View, CA") în coordonate geografice (cum ar fi latitudinea 37.423021 și longitudinea -122.083739). [23]

În esență, această componentă codifică adresele, generează o matrice de distanțe, calculează cea mai rapidă rută pentru echipe pe baza tipurilor de locuri de muncă și afișează rutele pe o hartă folosind Maps Javascript API.

Contract Search

Această componentă este responsabil pentru afișarea și gestionarea unui tabel de contracte. Componenta are următoarea structură:

1. Metode principale:
   - ngOnInit este apelată când componenta se inițializează. Aceasta apelează metoda displayContracts pentru a prelua și afișa datele contractelor.
   - displayContracts este responsabilă pentru preluarea datelor contractului din RecipesSearchService și pregătirea acestora pentru afișare. Utilizează contractService pentru a prelua datele contractului și datele locațiilor pentru fiecare contract.
   - addContract este apelată atunci când este adăugat un nou contract.
   - deleteContract este apelată atunci când un contract este șters.
2. Formulare:
   - Componenta definește un FormGroup numit addForm cu controale de formular pentru startDate, finishDate și value.
   - Metodele getter startDate, finishDate și value oferă acces convenabil la controalele formularului din șablon.

3. Alte metode:
   - Componenta definește metode suplimentare, cum ar fi editLocation și logout pentru gestionarea funcționalităților specifice, cum ar fi editarea unei locații și deconectarea.
4. Interfețe și constante:
   - Componenta definește un Element de interfață pentru a reprezenta structura unui element de contract.
   - De asemenea, definește o constantă ELEMENT_DATA care este o matrice goală de tip Element.

În componenta HTML, aceste metode sunt folosite mai departe pentru a conferi funcționalități paginii web. Elementele notabile sunt <form> și <mat-table> care utilizează addForm și dataSource pentru a comunica între componenta HTML și cea de TS. Formularul permite introducerea unor noi contracte, iar tabelul afișează contractele salvate și pune la dispoziție utilizatorului opțiunea de a adăuga noi locații.

În esență, ContractSearchComponent este responsabilă pentru preluarea datelor contractelor, afișarea lor într-un tabel și furnizarea de funcționalități pentru adăugarea, editarea și ștergerea contractelor.

Location Edit

Această componentă este responsabil pentru afișarea și gestionarea unui tabel de locații. Componenta are următoarea structură:

1. Metode:
   - ngOnInit este apelat după inițializarea componentei. Apelează metoda displayLocation() pentru a prelua și afișa datele despre locații.
   - displayLocation preia datele de locație din locationService pe baza ID-ului contractului stocat în memoria locală. După atribuie datele preluate sursei de date pentru a fi afișate în tabel.
   - addLocation este folosită pentru a adăuga o locație nouă. Aceasta creează un nou obiect Locație cu valorile introduse în formular și apelează metoda createLocation din serviciu pentru a-l adăuga la baza de date. După adăugarea cu succes, pagina este reîncărcată.
   - deleteLocation este apelată atunci când o locație trebuie să fie ștearsă. Apelează metoda DeleteLocation din serviciu pentru a șterge locația cu ID-ul specificat. După ștergere, pagina este reîncărcată.
   - logout este apelată atunci când utilizatorul dorește să se deconecteze. Setează rolul utilizatorului la „Anonim” (anonim) în memoria locală și navighează la pagina de

autentificare.

2. Formulare:
   - addForm: O instanță a FormGroup este creată pentru a gestiona controalele formularului pentru adăugarea unei noi locații. Conține controale de formular pentru city și address.
   - city și address: Aceste metode de obținere sunt utilizate pentru a accesa controalele de formular corespunzătoare din șablon.
3. Interfețe și constante:
   - Element: O interfață este definită pentru a reprezenta un element de tip locație. Include proprietăți pentru oraș, adresă, acțiuni și id.
   - ELEMENT_DATA: O matrice goală de obiecte Element este definită ca sursă inițială de date pentru tabelul de locații.

În componenta HTML, aceste metode sunt folosite mai departe pentru a conferi funcționalități paginii web. Elementele notabile sunt <form> și <mat-table> care utilizează addForm și dataSource pentru a comunica între componenta HTML și cea de TS. Formularul permite introducerea unor noi locații, iar tabelul afișează locațiile salvate.

În esență, LocationEditComponent este responsabilă pentru afișarea, adăugarea și ștergerea locațiilor din cadrul unui contract. Preia datele prin locationService, gestionează interacțiunile utilizatorului și actualizează vizualizarea în consecință.

Team Search

Această componentă este responsabil pentru afișarea și gestionarea unui tabel de echipe. Componenta are următoarea structură:

1. Metode:
   - ngOnInit este apelată după inițializarea componentei. Apelează metoda displayTeams() pentru a prelua și afișa datele echipei.
   - displayTeams preia datele echipei din teamService pe baza ID-ului utilizatorului stocat în memoria locală. Aceasta atribuie datele preluate sursei de date pentru a fi afișate în tabel. În plus, convertește valoarea disponibilității unei echipe în „Available” sau „Not available” în funcție de valoarea primită.
   - addTeam este folosită atunci când este adăugată o echipă. Aceasta creează un nou obiect Team cu valorile introduse în formular și apelează metoda CreateTeam din serviciu pentru a-l adăuga la baza de date. După adăugarea cu succes, pagina este reîncărcată.

- deleteTeam este apelată atunci când o echipă trebuie să fie ștearsă. Acesta preia toți angajații care aparțin echipei folosind metoda GetAllEmployeesById din serviciu. Apoi, iterează prin angajați și îi șterge folosind metoda DeleteEmployee. În cele din urmă, apelează metoda DeleteTeamById pentru a șterge echipa. După ștergere, pagina este reîncărcată.
- logout este apelată atunci când utilizatorul dorește să se deconecteze. Setează rolul utilizatorului la „Anonim” (anonim) în stocarea locală și navighează la pagina de conectare.

2. Formulare:
   - addForm: o instanță a FormGroup este creată pentru a gestiona controlul formularului pentru adăugarea unei noi echipe. Conține un control de formular pentru jobType.
   - jobType: O metodă getter este utilizată pentru a accesa controlul formularului jobType din șablon.

3. Interfețe și constante:
   - Element: O interfață este definită pentru a reprezenta un element de tip echipă. Include proprietăți pentru jobType, disponibilitate, acțiuni și id.
   - ELEMENT_DATA: O matrice goală de obiecte Element este definită ca sursă inițială de date pentru tabelul de echipă.

În componenta HTML, aceste metode sunt folosite mai departe pentru a conferi funcționalități paginii web. Elementele notabile sunt <form> și <mat-table> care utilizează addForm și dataSource pentru a comunica între componenta HTML și cea de TS. Formularul permite introducerea unor noi echipe, iar tabelul afișează echipele salvate și pune la dispoziție utilizatorului opțiunea de a adăuga noi angajați.

În esență, TeamSearchComponent este responsabilă pentru afișarea, adăugarea și ștergerea echipelor. Preia datele prin teamService, gestionează interacțiunile utilizatorului și actualizează vizualizarea în consecință.

Employee Edit

Această componentă este responsabil pentru afișarea și gestionarea unui tabel de angajați. Componenta are următoarea structură:

1. Metode:
   - displayRecipe preia datele angajaților din employeeService pe baza ID-ului echipei stocat în memoria locală. Aceasta atribuie datele preluate sursei de date pentru a fi

afișate în tabel.

- addEmployee este apelată atunci când este adăugat un angajat. Aceasta creează un nou obiect Employee cu valorile introduse în formular și apelează metoda CreateEmployee din serviciu pentru a-l adăuga la baza de date. După adăugarea cu succes, pagina este reîncărcată.
- deleteEmployee este apelată atunci când un angajat trebuie șters. Apelează metoda DeleteEmployee din serviciu pentru a șterge angajatul. După ștergere, pagina este reîncărcată.
- Logout este apelată atunci când utilizatorul dorește să se deconecteze. Setează rolul utilizatorului la „Anonim” (anonim) în memoria locală și navighează la pagina de conectare.

2. Formulare:
   - addForm: O instanță a FormGroup este creată pentru a gestiona controalele formularului pentru adăugarea unui nou angajat. Conține controale de formular pentru numele de familie și prenume.
   - lastName și name: metodele getter sunt utilizate pentru a accesa controalele de formulare respective din șablon.
3. Interfata și constante:
   - Element: O interfață este definită pentru a reprezenta un element de tip angajat. Include proprietăți pentru nume, nume, acțiuni și id.
   - ELEMENT_DATA: O matrice goală de obiecte Element este definită ca sursă inițială de date pentru tabelul angajat.

În componenta HTML, aceste metode sunt folosite mai departe pentru a conferi funcționalități paginii web. Elementele notabile sunt <form> și <mat-table> care utilizează addForm și dataSource pentru a comunica între componenta HTML și cea de TS. Formularul permite introducerea unor noi angajați, iar tabelul afișează angajatii salvați.

În esentță, EmployeeEditComponent este responsabil pentru afișarea, adăugarea și ștergerea angajaților dintr-o echipă. Preia datele prin employeeService, gestionează interacțiunile utilizatorului și actualizează vizualizarea în consecință.

Login

Această componenta gestionează pagina de autentificare a aplicației. Aceasta are funcționalitatea următoare:

1. Metode:

- Login este apelată atunci când utilizatorul face click pe butonul de conectare. Setează rolul utilizatorului la „Utilizator” în memoria locală și apelează metoda changeUserData din serviciul DataService pentru a actualiza datele utilizatorului. Apoi apelează metoda GetAcountByEmail din LoginRegisterService pentru a prelua datele contului de utilizator pe baza e-mailului introdus. Dacă e-mailul este găsit, ID-ul utilizatorului este stocat în memoria locală. Apoi, creează un nou obiect Login cu e-mailul și parola introduse în formular și apelează metoda LoginUser din LoginRegisterService pentru a autentifica utilizatorul. Dacă se primește un token valid în răspuns, rolul utilizatorului este setat la „Utilizator”, iar routerul navighează la pagina „/contract-search”.

2. Formulare:
   - loginForm: o instanță a FormGroup este creată pentru a gestiona controalele formularului pentru formularul de conectare. Conține controale de formular pentru e-mail și parolă.
   - email și password: metodele getter sunt utilizate pentru a accesa controalele de formulare respective din șablon.

În general, LoginComponent se ocupă de funcționalitatea de conectare a utilizatorului. Preia datele utilizatorului, autentifică utilizatorul și actualizează rolul utilizatorului în memoria locală. Interacționează cu formularul paginii pentru a obține e-mailul și parola introdusă și utilizează servicii pentru a face solicitări HTTP și a naviga la diferite pagini în funcție de rezultatul autentificării.

Register User

Această componentă gestionează pagina de înregistrare a aplicației. Aceasta are următoarea structură:

1. Metode:
   - Register este apelată atunci când utilizatorul face click pe butonul de înregistrare. Se creează un nou obiect Register cu valorile introduse în formularul de înregistrare. Apoi apelează metoda RegisterUser din LoginRegisterService și transmite obiectul Register ca argument. Metoda returnează o valoare booleană care indică succesul procesului de înregistrare. Dacă înregistrarea are succes, routerul navighează la pagina „/login”. În caz contrar, un mesaj de eroare este înregistrat în consolă.
2. Formulare:
   - createForm: o instanță a FormGroup este creată pentru a gestiona controalele

formularului pentru formularul de înregistrare. Conține controale de formular pentru nume, nume de utilizator, e-mail și parolă.

În general, RegisterUserComponent se ocupă de funcționalitatea de înregistrare a utilizatorilor. Obține datele de înregistrare de la utilizator, creează un obiect Register cu acestea și trimite datele către server pentru înregistrare folosind LoginRegisterService. Utilizează routerul pentru a naviga la pagina de autentificare după înregistrarea cu succes sau generează un mesaj de eroare dacă eșuează.

Material

Modulul material nu are componente și nu este utilizată în mod direct în cadrul aplicației. În schimb aceasta reprezintă un spațiu comun pentru librăriile celorlalte module. În material sunt importate toate librăriile dorite ca mai apoi să fie exportate mai departe. În acest fel nu trebuie importate în mod repetat toate aceste elemente. Ca o altă componentă să aibă acces la ele, doar trebuie să includă importuri din modulul material.

Material exportă următoarele librării:

1. MatButtonModule - Butoanele din Angular material sunt elemente native <button> sau <a> îmbunătățite cu stilul Material Design și ondulații de cerneală. [13]
2. MatFormFieldModule - <mat-form-field> este o componentă folosită pentru a împacheta mai multe componente Angular Material și pentru a aplica stiluri comune de câmpuri de text, cum ar fi sublinierea, eticheta flotantă și mesajele de tip indiciu. [14]
3. MatInputModule - matInput este o directivă care permite elementelor native <input> și <textarea> să lucreze cu <mat-form-field>. [15]
4. MatTableModule - mat-table oferă un tabel de date în stil Material Design care poate fi folosit pentru a afișa rânduri de date. Acest tabel se bazează pe baza tabelului de date CDK și folosește o interfață similară pentru intrarea și șablonul de date. [16]
5. MatPaginatorModule - <mat-paginator> oferă navigare pentru informații, utilizate de obicei cu un tabel. [17]
6. ReactiveFormsModule - Exportă infrastructura și directivele necesare pentru formularele reactive, făcându-le disponibile pentru import. [18]
7. MatIconModule - mat-icon facilitează utilizarea pictogramelor bazate pe vectori. Această directivă acceptă atât fonturile pentru pictograme, cât și pictogramele SVG, dar nu și formatele bazate pe bitmap (png, jpg etc.). [19]

2.2.2 Services

Serviciul este o categorie largă care cuprinde orice valoare, funcție sau caracteristică de care are nevoie o aplicație. Un serviciu este de obicei o clasă cu un scop restrâns, bine definit. Angular distinge componentele de servicii pentru a crește modularitatea și reutilizarea. O componentă ar trebui să prezinte proprietăți și metode pentru data binding pentru a media între vizualizare și logica aplicației. O componentă ar trebui să utilizeze servicii pentru sarcini care nu implică vizualizarea sau logica aplicației. Serviciile sunt bune pentru sarcini precum preluarea datelor de pe server, validarea intrărilor utilizatorului sau conectarea direct la consolă. Prin definirea unor astfel de sarcini de procesare într-o clasă de servicii injectabilă, acele metode devin disponibile oricărei componente [20].

Data Service

Serviciul conține un BehaviorSubject numit userSource. Un BehaviorSubject este un tip de Subiect din biblioteca RxJS care permite crearea unui flux de date observabile. Acesta păstrează valoarea curentă și o emite oricăror abonați atunci când este introdusă o nouă valoare. În acest caz, valoarea inițială a userSource este un obiect cu proprietăți de e-mail și parolă setate la șiruri goale. Serviciul expune o proprietate publică numită currentUser, care este un observabil obținut de la userSource folosind metoda asObservable(). Acest lucru permite componentelor să se aboneze la modificări în utilizatorul curent și să primească actualizări ori de câte ori un nou obiect utilizator este împins la userSource.

Serviciul oferă, de asemenea, o metodă publică numită changeUserData care acceptă un obiect utilizator care conține proprietăți de e-mail și parolă. Când este apelată, această metodă actualizează valoarea userSource prin împingerea noului obiect utilizator și orice abonat la currentUser va primi datele actualizate de utilizator.

În general, acest serviciu este utilizat pentru gestionarea și partajarea datelor utilizatorilor între diferite componente din aplicația Angular. Componentele se pot abona la modificări ale datelor utilizatorului prin observatorul currentUser și pot actualiza datele utilizatorului folosind metoda changeUserData.

Login-Register Service

Login-Register Service oferă metode pentru înregistrarea și conectarea utilizatorilor. Aceasta are următoarea structură:

1. Serviciul depinde de modulul HttpClient de la @angular/common/http, care îi permite să trimită cereri HTTP către API-uri externe.

2. Serviciul oferă trei proprietăți publice url_login, url_register și url_find_email, care stochează adresele URL pentru autentificare, înregistrare și, respectiv, găsirea conturilor în funcție de email.
3. Constructorul serviciului injectează o instanță de HttpClient pentru a fi folosită pentru a face cereri HTTP.
4. Serviciul conține două funcții POST: RegisterUser și LoginUser. Ambele funcții așteaptă un obiect de sarcină utilă ca argument (fie obiecte Register sau Login), care va fi serializat în JSON și trimis ca obiect de solicitare. Funcția RegisterUser trimite o solicitare POST către punctul final url_register, în timp ce funcția LoginUser trimite o solicitare POST către punctul final url_login. Ambele funcții includ corpul cererii și setează antetul „content-type" la „application/json".
5. Serviciul include, de asemenea, o funcție GET numită GetAcountByEmail. Această funcție ia un e-mail ca argument și trimite o solicitare GET către punctul final url_find_email cu e-mailul atașat la adresa URL. Returnează un observabil care va emite răspunsul de la cererea HTTP.
6. Serviciul definește două clase, Login și Register, care sunt folosite ca modele pentru datele de conectare și respectiv de înregistrare. Clasa Login are proprietăți de e-mail și parolă, în timp ce clasa Register are proprietăți creationDate, nume, userName, e-mail și parolă. De asemenea, clasa Register inițializează proprietatea creationDate cu data curentă folosind new Date().

În general, acest serviciu oferă metode pentru înregistrarea și conectarea utilizatorilor prin trimiterea de solicitări HTTP către punctele finale specificate. Include, de asemenea, o metodă de preluare a unui cont prin e-mail printr-o solicitare GET. Clasele Login și Register definesc structura datelor așteptate în cereri.

Route-Manager Service

Route Manager Service acest serviciu oferă metode pentru preluarea datelor și crearea de intrări în tabelele primare ale unei baze. Aceasta are următoarea structură:

1. Serviciul depinde de modulul HttpClient de la @angular/common/http, care îi permite să trimită cereri HTTP către API-uri externe.
2. Serviciul oferă mai multe proprietăți publice care stochează adresele URL pentru diferite puncte finale API legate de tabelele primare într-o bază de date. Tabelele principale includ Contract, Locație, Echipa și Angajat.
3. Constructorul serviciului injectează o instanță de HttpClient pentru a fi folosită pentru a face cereri HTTP.

4. Serviciul include diverse funcții GET pentru preluarea datelor de la punctele finale API. De exemplu, GetAllContracts trimite o solicitare GET pentru a prelua toate contractele, GetContractById trimite o solicitare GET pentru a prelua un anumit contract prin ID-ul său, GetAllLocations preia toate locațiile și așa mai departe. Adresele URL pentru aceste solicitări sunt construite folosind adresele URL ale punctelor finale respective și parametrii suplimentari.
5. Serviciul include și funcții POST pentru crearea de noi intrări în tabelele primare. De exemplu, CreateContract trimite o solicitare POST pentru a crea un nou contract, CreateLocation creează o nouă locație și așa mai departe. Funcțiile acceptă ca parametri obiecte din clasele corespunzătoare (Contract, Locație, Echipa, Angajat), le serializează în JSON și le includ în corpul cererii.
6. Serviciul definește clase pentru fiecare tabel principal (Contract, Locație, Echipa, Angajat). Aceste clase au proprietăți care reprezintă câmpurile tabelelor respective. Fiecare clasă are un constructor care inițializează proprietățile cu valorile furnizate.

În general, acest serviciu oferă metode pentru preluarea datelor și crearea de intrări în tabelele primare ale unei baze de date folosind cereri HTTP. Utilizează modulul HttpClient și definește clase pentru a structura datele trimise în cereri și primite în răspunsuri.

2.2.3 App routing și App guard

Modulul de rutare este responsabil pentru definirea rutelor și căilor de navigare în cadrul aplicației. Aceasta are următoarea structură generală:

1. Primele două obiecte rută au o cale goală (cale: '') și folosesc lazy loading pentru a încărca LoginModule și RegisterUserModule atunci când este accesată calea corespunzătoare.
2. Următorul obiect rută are o cale goală și este păzit de AuthGuard. Proprietatea canActivate permite accesul la rutele copiilor numai dacă AuthGuard returnează true. Rutele copii sunt definite într-o proprietatea children.
3. Fiecare rută copil are, de asemenea, o cale goală și utilizează lazy loading pentru a încărca diferite module de caracteristici (HomeModule, ContractSearchModule, LocationEditModule, TeamSearchModule și EmployeeEditModule) atunci când este accesată calea corespunzătoare.
4. Obiectul rută finală are și o cale goală și este păzit de Auth2Guarde. În prezent, această rută nu are definite trasee.

În esență, acest modul de rutare stabilește căile de navigare pentru aplicație. Acesta definește rute pentru autentificare, înregistrare și diverse module de caracteristici. De asemenea, include gărzi (AuthGuard și Auth2Guard) pentru a controla accesul la anumite rute pe baza

cerințelor de autentificare sau autorizare.

AuthGuard și AuthGuard2 definesc gradul pe care trebuie să îl aibă un utilizator pentru a accesa diferitele pagini din cardul aplicației. Acestea utilizează mai multe interfețe: CanActivate, CanActivateChild și CanDeactivate și au următoarea structură

5. Metoda constructorului este definită, iar serviciul Router este injectat în gardă. Routerul permite navigarea către alte rute.
6. canActivate este implementată din interfața CanActivate. Este nevoie de doi parametri: ruta și starea. Metoda verifică dacă utilizatorul este autorizat prin verificarea valorii articolului Rol din localStorage. Se așteaptă ca rolul să fie fie „Admin”, fie „Utilizator”. Dacă utilizatorul nu este autorizat, este înregistrat un mesaj de eroare, iar utilizatorul este redirecționat către pagina de conectare. Dacă utilizatorul este autorizat, metoda returnează true pentru a permite accesul la rută. Singura diferență dintre AuthGuard și AuthGuard2 apare în această funcție. În AuthGuard2 funcția oferă acces doar utilizatorilor cu gradul de “Admin”
7. canActivateChild este implementată din interfața CanActivateChild. Este nevoie de doi parametri: childRoute și stare. Metoda returnează în prezent true, permițând accesul la rutele copil.
8. canDeactivate este implementată din interfața CanDeactivate. Este nevoie de mai mulți parametri: component, currentRoute, currentState și, opțional, nextState. Metoda returnează în prezent true, permițând dezactivarea componentei.

Protecția AuthGuard este folosită pentru a proteja rutele împotriva accesului neautorizat. Verifică rolul utilizatorului stocat în localStorage și navighează la pagina de autentificare dacă utilizatorul nu este autorizat. Poate fi adăugat la configurațiile rutei pentru a controla accesul la anumite rute sau rute secundare pe baza cerințelor de autentificare sau autorizare.

Concluzii

Proiectul prezentat optimizează organizarea și traseul pentru o companie de curățenie, cu scopul de a maximiza eficiența, de a minimiza timpul de călătorie și, în cele din urmă, de a crește profitul. Aplicația se concentrează pe alocarea eficientă a echipelor de curățenie pe baza specializării lor, asigurându-se că echipele specializate se ocupă de sarcini complexe, evitând în același timp să-și piardă timpul cu altele mai simple. În plus, reducerea timpilor de deplasare este crucială, deoarece afectează în mod direct numărul de sarcini finalizate și disponibilitatea echipelor pentru alte misiuni generatoare de profit.

Aplicația își atinge scopul principal de a oferi o platformă în care managerii pot introduce detaliile contractului și echipele disponibile, servind drept bază pentru optimizare. Prin analiza datelor, aplicația generează rutele optime, prezentându-le managerului pentru luarea deciziilor.

Aplicațiile similare de pe piață, cum ar fi Verizon Connect, Route4Me și Onfleet, oferă caracteristici de planificare a rutei pentru companii și, în plus, includ și capabilități de urmărire a flotei, care nu sunt incluse în acest proiect. Verizon Connect se concentrează pe urmărirea flotei și monitorizarea performanței, în timp ce Route4Me optimizează rutele pentru a economisi timp și consum de combustibil. Onfleet oferă o gamă de API-uri pentru o integrare perfectă cu alte componente ale unei afaceri, oferind organizare eficientă și optimizare a rutei.

Aplicația implementează cu succes contribuția principală propusă, optimizarea bazata pe gradul de specializare, în domeniul eficientizării rutelor. Luând în considerare nivelul de pregătire și complexitatea echipamentelor disponibile fiecărei echipe, aplicația permite o determinare mai eficientă a rutei. Rutele pentru echipele foarte specializate acordă prioritate sarcinilor complexe, permițându-le să le abordeze mai întâi înainte de a trece la sarcini mai simple. Această abordare minimizează pierderile de timp și financiare, asigurând că echipele specializate sunt disponibile pentru contracte care necesită expertiza lor, utilizând în același timp eficient resursele.

Prin atribuirea optimizata a sarcinilor pe baza nivelului de expertiză necesar, aplicația reduce pierderile de timp și bani. Evită situațiile în care echipele specializate sunt ocupate cu sarcini mai simple, împiedicându-le să fie disponibile pentru contracte de nivel înalt. În același timp asigură că echipele cu calificare scăzută au sarcini, pentru a nu fi lăsate într-o stare de subtilizare de către echipele specializate. Prin abordarea acestor probleme, aplicația optimizează logistica companiei, minimizând suprapunerea în sarcinile echipei și sporind profitul general.

În concluzie, aplicația prezentată în acest proiect se concentrează pe optimizarea operațiunilor companiei de curățenie prin alocarea eficientă a echipelor specializate, minimizarea timpului de călătorie și maximizarea profitului. Încorporând optimizarea rutelor bazată pe specializare, aplicația eficientizează logistica, reduce timpii neproductivi și asigură disponibilitatea echipelor pentru orice sarcină potențială care ar putea apărea.

Bibliografie

[1] URL: https://www.g2.com/categories/route-planning/small-business, accesat 28.05.2023

[2] URL: https://www.verizonconnect.com/solutions/gps-fleet-tracking-software/, accesat 28.05.2023

[3] URL: https://route4me.com/, accesat 28.05.2023

[4] URL: https://onfleet.com/, accesat 29.05.2023

[5] D. Roth, R. Anderson, S. Luttin. *Overview of ASP.NET Core*, https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-7.0, accesat 02.06.2023

[6] R. Anderson. *Introduction to Identity on ASP.NET Core*, https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-7.0&tabs=visual-studio, accesat 02.06.2023

[7] S. Smith, S. Addie. *Handle requests with controllers in ASP.NET Core MVC*, https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/actions?view=aspnetcore-7.0, accesat 02.06.2023

[8] *Migrations Overview*, https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli, accesat 03.06.2023

[9] *What is Angular?*, https://angular.io/guide/what-is-angular, accesat 04.06.2023

[10] *Introduction to modules*, https://angular.io/guide/architecture-modules, accesat 04.06.2023

[11] *Lazy-loading feature modules*, https://angular.io/guide/lazy-loading-ngmodules, accesat 04.06.2023

[12] *Introduction to components and templates*, https://angular.io/guide/architecture-components, accesat 05.06.2023

[13] *Button*, https://v7.material.angular.io/components/button/overview, accesat 05.06.2023

[14] *Form field*, https://v5.material.angular.io/components/form-field/overview, accesat 05.06.2023

[15] *Input*, https://v5.material.angular.io/components/input/overview, accesat 05.06.2023

[16] *Table*, https://v5.material.angular.io/components/table/overview, accesat 05.06.2023

Bibliografie

[17] *Paginator*, https://v5.material.angular.io/components/paginator/overview, accesat 05.06.2023

[18] *ReactiveFormsModule*, https://angular.io/api/forms/ReactiveFormsModule?ref=console-log, accesat 06.06.2023

[19] *Icon*, https://v7.material.angular.io/components/icon/overview, accesat 06.06.2023

[20] *Introduction to services and dependency injection*, https://angular.io/guide/architecture-services, accesat 06.06.2023

[21] *Distance Matrix Service*, https://developers.google.com/maps/documentation/javascript/distancematrix, accesat 08.06.2023

[22] *Directions Service*, https://developers.google.com/maps/documentation/javascript/directions, accesat 08.06.2023

[23] *Geocoding Service*, https://developers.google.com/maps/documentation/javascript/geocoding, accesat 08.06.2023