### 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

1. [Introducere](#_bookmark0) 5
   1. Scopul Aplicației 5
      1. Exemplu pentru utilizarea specializărilor...................................................................5
   2. Context.....................................................................................................................................6
   3. Aplicații similare și contribuții propuse...................................................................................7
   4. Motivație 8
2. Descrierea Aplicației 9
   1. Backend....................................................................................................................................9
      1. Entities........................................................................................................................9
      2. Repositories..............................................................................................................12
      3. Controllers................................................................................................................16
      4. Special Components.................................................................................................21
      5. Database...................................................................................................................22
      6. Startup......................................................................................................................24
3. [Concluzii](#_bookmark5) 7

[Bibliografie](#_bookmark6) 8

# Introducere

## 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. Această aplicație 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. **Exemplu pentru utilizarea specializărilor**

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

În exemplul de mai sus ierarhia specializărilor este următoarea:

**Generală < Medie < Avansată**

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

**Ecipa 1 - Contract 1**

**Echipa 2 - Contract 1, Contract 2, Contract 3**

**Ehipa 3 - Contract 1, Contract 3**

## 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 de curățări.

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). 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 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, dar chiar dacă ținta principală sunt firmele de curățenie, 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 optimizare. În urma acestei optimizări traseul cel mai optim este prezentat managerului.

* 1. **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 traseuri 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. Aceast 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.

## 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 lucreaza în domeniul firmelor de curățări. 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

## 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 clod 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.

* + 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* (Create*ContractDTO*, *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 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 |
| 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 c*ontract.*

|  |  |  |
| --- | --- | --- |
| **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 loca*l 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* |

* + 1. **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, comune între toate fișierele. *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) |

* + 1. **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.
   * 1. **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”.

* + 1. **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].

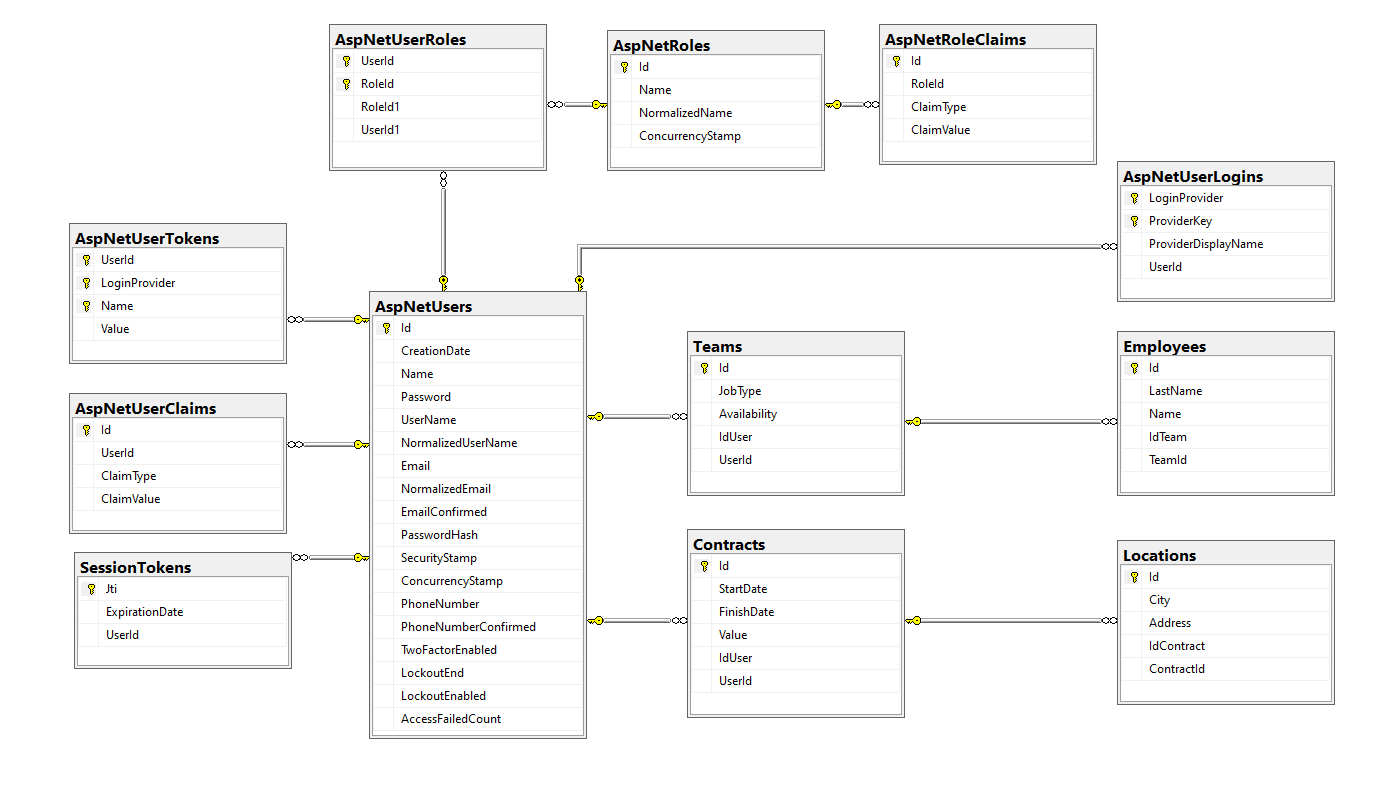


Fig. 1 diagrama UML

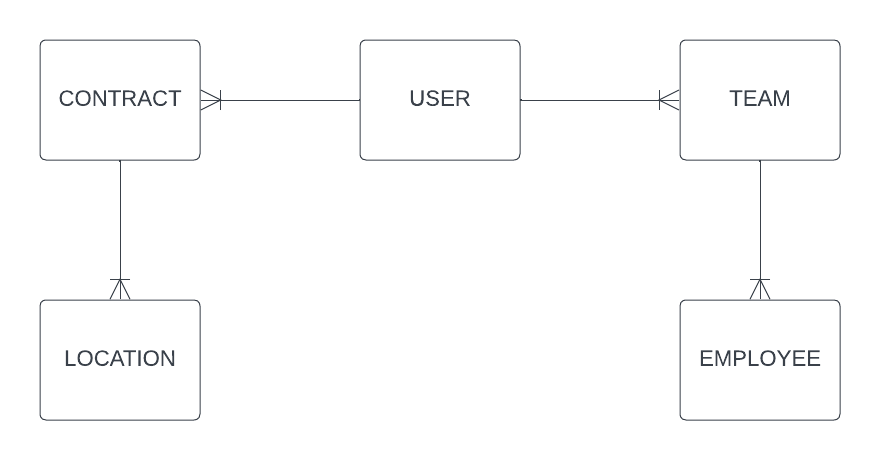


Fig. 2 diagrama relațiilor între entități

* + 1. **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.

## 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.

* + 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 pun la dispoziție aceste elemente de tip modul. 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.

# 

Fig. 3 contract-search.module.ts

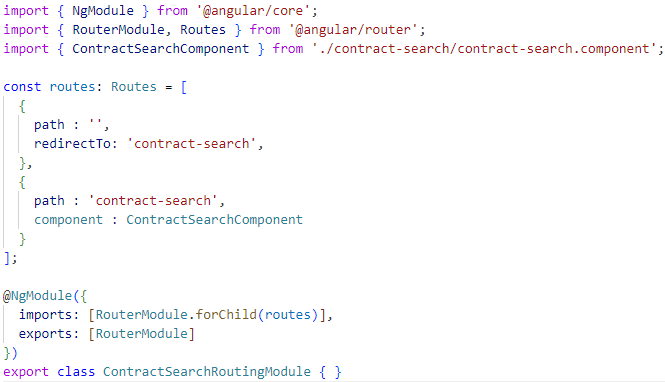


Fig. 4 contract-search-routing.module.ts

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 lăngă 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.

**Contract Search**

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

1. Declarații de import:
   * Componenta importă diverse module și servicii Angular, cum ar fi Component, OnInit, ViewChild, Router, MatTableDataSource, MatPaginator și MatSnackBar din @angular/core și @angular/material.
   * De asemenea, importă Contract și RecipesSearchService dintr-un fișier de serviciu personalizat (recipes-search.service) și AbstractControl, AbstractFormGroupDirective, FormControl și FormGroup din @angular/forms.
2. Decorator componente:
   * Componenta este decorată cu @Component decorator, care specifică selectorul, URL-ul șablonului și URL-urile de stil pentru componentă.
3. Clasa componente:
   * Clasa de componente ContractSearchComponent implementează interfața OnInit, indicând că are o metodă ngOnInit care va fi executată la inițializarea componentei.
   * Clasa are proprietăți precum displayedColumns\_contracts, id, dataSource și dataSourceC pentru gestionarea datelor și a configurației tabelului.
   * De asemenea, are o proprietate MatPaginator numită paginator, care este accesată folosind decoratorul @ViewChild pentru a interoga elementul paginator în vizualizarea componentei.
4. 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.
5. FormGroup și Form Controls:
   * 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.
6. Alte metode:
   * Componenta definește metode suplimentare, cum ar fi editLocation și logout pentru gestionarea funcționalităților specifice, cum ar fi editarea unei locații și deconectarea.
7. 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.

# Concluzii

1. URL: [https://www.g2.com/categories/route-planning/small-business](https://www.g2.com/categories/route-planning/small-business )
2. URL: <https://www.verizonconnect.com/solutions/gps-fleet-tracking-software/>
3. URL: <https://route4me.com/>
4. URL: <https://onfleet.com/>