UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA

DE

MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

Aplicație Web pentru căutarea destinațiilor de vacanță destinată grupurilor de turiști

Absolvent

RADU DENIS FLORIAN

Coordonator ştiinţific

IONESCU RADU

București, februarie 2021

Cuprins

Rezumat

Aplicația web pentru căutarea destinațiilor de vacanță destinată grupurilor de turiști este o aplicație ce are ca obiectiv formarea și sugerarea celor mai potrivite destinații turistice pe care o persoană sau un grup de persoane le poate avea.

Ideea în sine a plecat de la situațiile reale pe care le au oamenii în momentele în care își doresc să planifice o excursie sau o călătorie, dar intervin factori precum bugetul alocat, clima dorită, țara aleasă, s.a.m.d. Realizarea unor sugestii pentru dorințele unei singure persoane este relativ ușor, aceasta este stăpână pe modul în care își structurează interesele și nevoile în ceea ce privește locația pentru care optează, dar problema adevărată intervine în momentul în care un grup de persoane cu interese diferite apar și acestea nu reușesc să cadă de comun acord pe o destinație anume, cauzele putând fi multiple și diversificate. În situația aceasta, aplicația realizează pe baza unui algoritm, filtrarea intereselor fiecărei persoane și caută cele mai apropiate rezultate de alegerile fiecăruia per total.

Situațiile speciale în care niciun rezultat nu a fost găsit pentru un grup, algoritmul va reanaliza filtrele și va extinde aria de căutare, având posibilitatea să intervină cu sugestii ce pot ieși din aria de interes pe care persoanele din grup le-au sugerat, făcând posibilă propunerea unor destinații pe care aceștia nu le-au luat în vedere, dar care prezintă anumite criterii de selecție per individ și care pot fi incluse în rezultatul final ca opțiuni secundare.

În paralel, persoanele unui grup au posibilitatea de a comunica prin intermediul unei mesagerii, iar fiecare conversație va fi disponibilă și diferită pentru fiecare grup în parte, dând posibilitatea unei planificări mai ușoare și avantajul de a putea actualiza lista de interese pentru una sau mai multe persoane. De asemenea, fiecare persoană dispune de o pagină principală care la un anumit interval de timp va fi actualizată și vor fi prezentate multiple recomandări de destinații turistice, obținute pe baza setărilor de interese generale pe care aceasta le-a specificat.

Un sistem de notificări permite utilizatorilor să fie înștiințați cu privire la noi invitații în grupuri, excluderea lor de către un administrator dintr-un grup și noi cereri de prietenie de la alți utilizatori.

Aplicația este adaptată pentru toate dispozitivele și este fiabilă pentru orice browser, iar tehnologiile utilizate sunt actualizate și compatibile cu versiunea limbajului pe care rulează, atât pe partea de front-end, cât și pe cea de back-end.

Summary

The web application built for searching holiday destinations for groups of tourists is an application that aims to form and suggest the most suitable tourist destinations a person or a group of people can have.

The idea itself has been based on the real situations people have when they want to plan a trip or a trip, but factors such as the budget allocation, the desired climate, the chosen country, and so on. Making suggestions for the wishes of a single person is relatively easy, it has ownership of how it structures its interests and needs in terms of the location of its choice, but the real problem arises when a group of persons with different interests emerge, and they fail to agree on a particular destination, the causes can be multiple and diverse. In this situation, the application performs on the basis of an algorithm, filtering the interests of each individual and seeking the closest results to each individual's choices in total.

Special situations where no results have been found for a group, the algorithm will re-analyze the filters and extend the search area, with the possibility of intervening with suggestions that may leave the area of interest suggested by the people in the group, making it possible to propose destinations which they have not considered but which have certain selection criteria per individual and which can be included in the final result as secondary options.

In parallel, people in a group have the opportunity to communicate via a messaging, and each conversation will be available and different for each group, allowing easier planning and the advantage of being able to update the list of interests for one or more people. Also, each person has a home page that will be updated at a certain time frame and multiple tourist destination recommendations will be presented based on the general interest settings it has specified.

A notification system allows users to be notified of new invitations in groups, their exclusion by an administrator from a group, and new friend requests from other users.

The application is tailored for all devices and is reliable for any browser, and the technologies used are up-to-date and compatible with the version of the language on which it is running, both front-end and back-end.

I. Introducere

I.1. Tipul lucrării și subdomeniul specific în care se încadrează tema

Lucrarea este o aplicație web, structurată pe cele două părți importante ale programării, partea de front-end și partea de back-end [1], astfel încât construcția acesteia să fie încadrată tiparului general de aplicație, cel putin când ne referim la arhitectura oricărei aplicații sau oricărui site web. Decizia de a realiza o aplicație web a pornit în urma perioadei de stagiu într-o companie românească, la care am descoperit demersurile și necesitățiile principale pentru realizarea unei astfel de aplicații. În plus, pe perioada stagiaturii, am înțeles care pot fi diferențele între cele două părți ale unei aplicații și cum pot fi dezvoltate astfel încât să fie păstrate convențiile stabilite la nivel global pentru acestea.

Dorința de a pune bazele unei aplicații de acest gen a fost accentuată și mai tare la următorul loc de muncă, la care am descoperit cu adevărat ce înseamnă o aplicație web și de la care am „furat" idei și cunoștiințe, oferindu-mi toate resursele necesare pentru a construi din toate punctele de vedere aplicația ce urmează a fi prezentată. Aici am învățat cum creez un proiect, cum îl configurez, cum îl structurez și cu ce ustensile și framework-uri pot să lucrez pentru a obține o aplicație web cât de cât profesionist realizată.
Idea pe baza căreia am început dezvoltarea a venit dintr-o necesitate generală pe care majoritatea oamenilor o manifestă și care vine sub forma unui mic asistent în momentul în care suntem indecișși, avem dubii sau poate chiar nu știm care este varianta favorabilă pentru ce ne dorim în momentul acela.

Tema aleasă este încadrată turismului, combinat cu social-media, deoarece putem spune că ideea aplicației se bazează pe rezolvări și răspunsuri ce se încadrează turismului, dar oferind un mediu prietenos de socializare între utilizatori.

I.2. Prezentarea generală, scopul și motivația alegerii temei

Aplicația web pentru căutarea destinațiilor de vacanță adresată grupurilor de turiști, este o aplicație bazată pe nevoia utilizatorilor și dorințelor lor, astfel încât să li se poată pună la dispoziție un număr mare de opțiuni pentru a alege cât mai ușor o destinație turistică.
Idea de aplicație pentru generarea unor destinații turistice a pornit de la nevoia comună pe care o au oamenii atunci când doresc să călătorească sau să petreacă o vacanță în anumite locații pe care

fie nu le-au vizitat vreodată, fie despre care nu au auzit până în acel moment, dar care au o șansă destul de mare să fie pe placul acestora.

Cu ce vine totuși diferit aplicația față de altele care sunt deja cunoscute și existente pe piață de mult timp? Ei bine, spre deosebire de aplicațiile existente pentru generarea unor sugestii bazate pe interesele și opțiunile alese de utilizatori, vine în plus cu un mecanism prin care le este oferită posibilitatea utilizatorilor să creeze grupuri prin care să poată invita alți oameni să participe la generarea unor destinații turistice prin selectarea unor interese și opțiuni dorite. Mecanismul vine în ajutorul familiilor, grupurilor de prieteni, excursiilor școlare, taberelor și multor altor adunări de cel puțin doi oameni în care există o posibilitate foarte mare ca cel puțin o persoană să aibă interese diferite și să își dorească vizitarea altor atracții turistice, dar totuși să se poată ajunge la un consens între membrii grupurilor pentru alegerea celor mai favorabile destinații turistice încât să îi poată mulțumi pe aproape toți participanții.

Pe lângă mecanismul principal în jurul căruia fost construită ideea aplicației, mai există și opțiunea de a creea un set de destinații turistice pentru o singură persoană, dar asupra căruia nu vor putea fi adăugate și alte persoane pentru modelare. De asemenea, pe pagina principală vor exista mereu destinații turistice sugerate automat de către server pentru utilizatori, în funcție de setările pe care aceștia le vor face în pagina de administrare a contului.

Fiecare grup dispune de o pagină modală de administrare a grupului, prin care administratorii desemnați vor putea invita alți utilizatori să se alăture grupului și vor putea să dea afară utilizatori nedoriți, iar oricare membru al grupului va avea posibilitatea de a reseta lista de interese aleasă și de a părăsi grupul.
Un lucru foarte important fără de care am considerat că nu poate exista un grup, a fost un chat pe care l-am alocat fiecărui grup în parte, făcând posibilă o comunicare deschisă între membri, să poată discuta cât mai ușor despre destinațiile și interesele dorite și alese.

Pentru a putea gestiona invitațiile primite, a fi înștiințați de grupurile din care au fost dați afară și cererile de prietenie primite, utilizatorilor le este pus la dispoziție un sistem de notificări, care este actualizat de fiecare dată când aceștia reîncarcă orice pagină de pe aplicație.

În cazul în care utilizatorii doresc să revadă atât grupurile, cât și sugestiile generate, sau drumețiile pe cont propriu, fiecare dispune de o pagină în care sunt memorate toate acestea și sunt accesibile pentru a fi vizitate. Totuși, în cazul în care un utilizator nu mai face parte dintr-un grup sau a abandonat crearea drumeției individuale, acesta nu va mai putea accesa paginile acestora, ci doar vor fi înștiințați că au făcut parte din ele.

Înainte de a putea utiliza aplicația, utilizatorii vor fi nevoiți să își creeze un cont pe baza căruia vor fi stocate toate interesele.

I.3. Contribuția proprie in realizarea lucrării

Realizarea lucrării a fost făcută pe cont propriu, astfel toate apoartele asupra arhitecturii, modul de organizare a structurii, separarea codului în multiple fișiere, decizia legată de limbajul folosit pentru partea de server, cât și framework-urile aferente, limbajul pentru front-end cât și librăriile utilizate, au fost alese bazându-mă pe cunoștiintele pe care le dețineam până la momentul începerii proiectului.

Ulterior proiectul a suferit modificări majore, de la structura gândită inițial, până la regândirea interfeței pe partea de client. Cel mai semnificativ lucru a fost implementarea unor design pattern-uri [2] pentru modelarea proiectului într-o manieră eficientă și ușor de modificat sau alterat ulterior.

I.4. Structura lucrării

În momentul începerii aplicației, aceasta a fost construită pe baza a două proiecte C# și un proiect backup în care era ținută structura bazei de date și alterată cu fiecare modificare adusă.
Proiectul principal reprezenta serverul aplicației și totodată partea de client, fiind un serviciu web bazat pe conexiuni REST API [3] și construit folosindu-mă de modelul arhitectural „Model-View-Controller" [4] pentru a permite apelarea request-urilor HTTP de client către partea de server. Framework-ul principal pentru dezvoltarea serviciului web a fost .NET Core 2.2 [5], asupra căruia au fost setate diferite configurări pentru cuplarea corectă cu partea de client. De asemenea, pentru partea de front-end, am utilizat framework-ul AngularJs [6] împreună cu Javascript pur pentru funcționalitate, dar împreună cu framework-ul C# Razor [7] pentru a putea folosi diferite metode ale serverului în interiorul interfeței.

Acest proiect a fost secționat pe mai multe foldere precum Controllers, Models, Views, AppViews, AppStyle, App, Images, Scripts, Workers acestea fiind asociate următoarelor lucruri:

- Controllers -> folderul principal unde se află toate fișierele în care se regăsesc clasele intermediare dintre client și server, acestea oferind prin diferite metode accesul către server prin request-uri HTTP.

- Models -> aici se regăsesc toate entitățile folosite că intermediari pentru preluarea unor date ce vin în request-uri către server sau utilizate ca răspuns pentru metode pentru a trimite obiectul JSON copie al entității către partea de client.

- Views -> reprezintă zona principală unde se regăsesc fișierele Razor cu extensia CSHTML, ele fiind la bază de fapt fișiere HTML, dar care permit utilizarea sintaxelor C# în acestea. Acest folder este obligatoriu să existe, deoarece framework-ul .NET Core împreună cu arhitectura MVC, asociază toate fișierele controller din folderul Controllers cu view-urile aferente din acest folder, folosind convenția de nume în care folderul are numele controller-ului, iar fișierul CSHTML are denumirea index.

- AppViews -> în acest folder există toate fișierele HTML create pentru interfața aplicației, secționate în funcție de zonele de care aparțin. Aceste fișiere sunt utilizate în interiorul unui fișier global CSHTML care reprezintă părintele aplicației la nivel de interfață, astfel aplicația se încadrează categoriei de site cu o singură pagină, deoarece orice trecere către altă pagină, de fapt reprezintă alt html ce înlocuiește cel curent fără a apela un nou DNS request.

- AppStyle -> folderul în care stilizările CSS ale paginilor HTML se regăsesc, secționate pentru a putea fi diferențiate în funcție de nevoie.

- App -> reprezintă folderul vital pentru funcționarea corectă a interfeței, deoarece aici se află toate fișierele Javascript cu AngularJs, fișiere ce au rolul de controllere, request-uri către server, fișiere ajutătoare ce conțin metode reutilizabile, directive și fișiere de constante. În cazul în care un fișier HTML nu ar avea asociat un fișier controller JS, nu ar avea nicio funcționalitate și ar fi un fișier static.

- Images -> așa cum spune și numele folderului, acesta conține imagini utilizate pe interfață. Acestea au fost salvate local, deoarece erau criticale, iar pierderea lor ar fi stricat interfața, dar există și imagini ce sunt încărcate din mediul online, acestea fiind suplimentare și fără să fie nevoie de un backup.

- Scripts -> folder în care există librării și framework-uri externe precum cel de AngularJs, Bootstrap pentru CSS, librării adiționale pentru AngularJs sau Javascript, evitând astfel legături către librării din mediul online în cazul în care ar exista o problemă de conexiune. De asemenea, sunt utilizate și librării externe, dar care sunt suplimentare și nu pot cauza o discrepanță.

- Workers -> toată funcționalitatea server-ului, mai precis metodele și variabilele claselor ce oferă funcționalitatea server-ului, erau păstrate în acest folder, dar nu orice funcționalitate, ci doar cea în care erau folosite metodele principale ce se vor regăsi în proiectul ce urmează să fie prezentat mai jos.

Proiectul secundar poartă denumirea de DataLayer, zona unde sunt executate toate comenzile principale către baza de date, cât și query-uri mai complexe, dar toate având ca scop interacțiunea cu baza de date, precum metode de inserare, ștergere, modificare și retragere a datelor din baza de date. Pe lângă aceste funcționalități, tot în acest proiect se regăseau toate interfețele, modelele și clasele utilizate în proiectul principal.
Specificat și anterior, aplicația folosește o bază de date gratuită de la Microsoft, mai precis SQL Server [8], instalată pe calculatorul local. Pentru a putea funcționa aplicația, aceasta verifică la runtime dacă conexiunea către baza de date poate fi realizată conform unui text ce conține conexiunea către SQL Server.

Pe parcursul modelării aplicației, aceasta a suferit schimbări majore, printre care și reorganizarea

proiectelor, astfel că proiectul de DataLayer a fost împărțit în trei proiecte sub denumirile Domain, DataLayer, Helpers. Asemenea proiectului principal, i-au fost aduse schimbări ce au rezultat în separarea de proiectul principal a funcționalităților din folderul Workers ce au fost duse într-un nou proiect denumit BusinessLogic. Formarea acestei separări a venit de-a lungul procesului meu de învățare la compania la care am lucrat pe parcursul dezvoltării acestei aplicații, unde am înțeles nevoia unei mai bune și eficiente organizări, astfel am aplicat design pattern-ul ce poartă denumirea de "Repository Pattern" [9]. Acest pattern presupune consolidarea următoarei arhitecturi:

- Crearea proiectului Domain în care vor fi ținute toate modelele/entitățile utilizate pentru maparea datelor din sau spre baza de date, temporare pentru diferite prelucrări sau diferite clase în care sunt ținute constante sau enum-uri.

- Crearea proiectului Helpers în care vor fi ținute toate clasele ce conțin metode ajutătoare, mai precis metode și clase statice sau diferite constante ce vor fi refolosite fie în metodele ajutătoare, fie în restul aplicației.

- Păstrarea proiectului DataLayer în care rămân doar interfețele și clasele cu metodele primare ce accesează baza de date pentru modelarea tabelelor sau preluarea datelor din ele.

- Crearea proiectului BusinessLogic ce conține doar logică de prelucrare a datelor venite prin controller, utilizând toate ustensilele puse la dispoziție de proiectele Domain, Helpers și DataLayer.

Folosind pattern-ul menționat, permite aplicației să fie scalabilă și să poată fi create și alte servicii web precum proiectul principal, care la rândul lor să poată utiliza clasele, metodele și variabilele din proiectele ajutătoare.

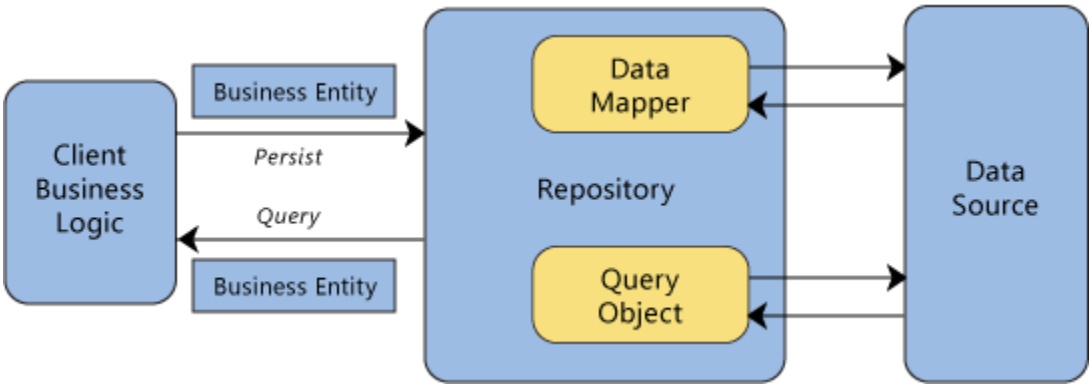

Fig. 1 Exemplu schemă de arhitectură folosind pattern-ul "Repository"

Pentru conexiunea către baza de date, am ales să utilizez framework-ul “Entity Framework” [10] care mi-a fost recomandat de ceilalți colegi de la locul de muncă, dar pe care îl cunoșteam la rândul meu și știam exact cum trebuie implementat. Avantajele acestui framework este simplitatea sintaxei și posibilitățile oferite pentru prelucrarea datelor, folosindu-mă de metode predefinite în acest framework ce utilizau în spate query-uri optimizate către SQL Server. Pe lângă Entity Framework, am fost nevoit să utilizez și librăria ADO.NET [11], o librărie ce permite crearea unei conexiuni directe cu baza de date și interogarea acesteia prin query-uri de SQL create manual. Motivul principal pentru care am ales să adaug și această modalitate, a fost imposibilitatea de a lucra simultan cu două conexiuni de Entity Framework pornite în paralel, motiv pentru care aceasta rezulta mereu într-o excepție aruncată.

Ulterior, proiectul a migrat de la versiunea .NET Core 2.2 către .NET Core 3.1, deoarece aceasta venea cu îmbunătățiri și soluționari la diferite probleme ale framework-ului. Odată cu această modificare, au apărut și modificări la nivelul configurării făcute în proiectul principal, care au forțat și schimbarea anumitor parametrii ale unor metode din fișierele controller existente.

I.5. Repere istorice și rezultate cunoscute

În trecut oamenii planificau excursiile în funcție de locațiile de care auziseră sau văzuseră în poze, sau chiar în funcție de zonele pe care deja le vizitaseră. Rareori se întâmpla ca cineva să meargă într-o locație fără să știe un minim de detalii despre aceasta, astfel multe atracții turistice ale pământului rămâneau nevizitate. Evoluția tehnologiei a început să acopere aceste lipsuri și să

creeze noi aplicații web disponibile pentru orice tip de dispozitiv, ca oamenii să se poată ajuta de ele pentru a fi informați despre zone turistice existente, în care puteau să vizualizeze poze cu peisaje, să descopere tradițiile, climatul, modul de transport și multe alte repere turistice legate de locația aleasă. Aplicații Web precum Booking, Trivago și multe altele, au pornit acest concept prin care expun utilizatorilor diferite locații și oferă informații referitoare la condițiile și beneficiile pe care le vor avea turiștii.

Deși la nivel de turist sunt prezentate diferite recomandări, personal nu am descoperit o aplicație web care să prezinte recomandări pentru grupuri de oameni, care să își pună problema dacă aceștia au individual recomandări diferite și vor totuși să obțină sugestii care să fie plăcute pentru toate taberele. Lipsa unui astfel de mecanism limitează foarte mult grupurile de oameni, mai ales în perioada sărbătorilor în care site-urile de traveling au un trafic imens, pentru că familii și grupuri de prieteni doresc să plece în diferite locații, desigur aceștia având cel mai probabil interese diferite.

Aplicația aceasta acoperă lipsul menționat, dar vine și ea cu lipsuri ce sunt acoperite de celelalte aplicații menționate, precum lipsa unor cazări și rezervarea acestora, achiziționarea unor modalități de transport către acele locații și numărul limitat de atracții turistice menționate, deoarece capacitatea de date stocată este limitată la posibilitatea personală de a găsi și adăuga referințe turistice, în comparație cu aplicațiile celelalte care dispun de un grup de oameni ce completează aceste informații.

II. Tehnologii

II.1. Tehnologii Back-End

Alegerea tehnologiilor pentru partea de back-end a aplicației a fost relativ intuitivă, deoarece majoritatea dintre acestea îmi erau familiare la momentul începerii proiectului, având experiență precedentă cu ele la locul de munca. Desi le cunoșteam și știam la ce sunt utile, provocarea a venit în momentul în care a trebuit să le integrez, să le configurez și să le utilizez cât mai eficient. Pentru realizarea aplicației, am decis să folosesc IDE-ul Visual Studio Community 2017, ulterior migrând la Visual Studio Community 2019, fiind cel mai recomandat pentru dezvoltarea aplicațiilor web în C# în condițiile în care acesta era și este în continuare gratuit, destinat pentru studenți și proiecte personale.

Serverul este creierul unei aplicații, iar acesta trebuie să fie cât mai bine organizat și să se folosească de cât mai multe lucruri benefice pentru a obține o performanță cât mai bună, evitarea a câtor mai multe excepții și formarea unei scalabilități eficiente pentru mărirea numărului de funcționalități și diminuarea greșelilor de bună practică pentru limbajul C#.

Așa cum am menționat și anterior, serverul este bazat pe limbajul C#, iar framework-ul principal pentru a fi posibilă crearea unui server web, este .NET Core. Prima versiune pe care am abordat-o și utilă la momentul acela, a fost versiunea 2.2, deoarece era ultima disponibilă și apărută, dar și compatibilă cu toate framework-urile utilizate, și desigur cea mai potrivită din punct de vedere al optimizărilor aduse framework-ului și rezolvărilor de erori pe care le avusese. Deși existau alte alternative precum framework-ul .NET Standard 4.5, acesta nu era gratuit și open-source pentru a fi accesibil codul oricărui dezvoltator, făcând limitat accesul spre îmbunătățiri și idei ce ar fi putut crea din acesta un framework căutat și ușor de utilizat. În plus, comunitatea persoanelor ce utilizau .NET Standard era destul de restrânsă în comparație cu cea pentru .NET Core, astfel soluțiile și părerile referitoare la acesta erau puține și de multe ori, fară un răspuns la anumite probleme.

Pentru a putea compila soluția pentru partea de back-end, au fost instalate fișierele SDK și Runtime ce aparțineau de .NET Core și aferente versiunii utilizate. Fișierul SDK este utilizat pentru a avea acces la librării și ustensile oferite de framework-ul .NET Core pentru a putea realiza aplicația sau cel puțin partea de back-end și conexiunea către partea de front-end, dar în special permite compilarea soluțiilor. Fișierul Runtime este utilizat pentru a permite rularea aplicațiilor în urma compilărilor făcute, fară de care nu ar putea fi accesibile spre utilizare.

Datorită framework-ului .NET Core, am utilizat librării precum Microsoft.AspNetCore.Http și Microsoft.AspNetCore.Http.Abstractions pentru a putea crea sau primi request-uri HTTP, fie prin intermediul conexiunii cu partea de client, fie prin alte modalități precum programul Postman, pentru depanarea problemelor și/sau a codului implementat. Programul Postman este extrem de utilizat în rândul programatorilor, utiliatatea sa primară fiind creearea și interograrea oricărui tip de request HTTP către un server anume pe baza unor multiple aspecte, precum IP-ul sau DNS-ul calculatorului pe care se află serverul, port-ul asociat acestuia, protocolul utilizat pentru request (GET,POST,PUT,DELETE,UPDATE) [12] și multe alte configurări de acces atribuite în mod normal unui request HTTP. Astfel, cu ajutorul acestui program am putut vizualiza daca serverul recepționa apelurile făcute și returna mesajul dorit, sau dacă exista vreo problemă ce ar fi putut periclita buna funcționare, dar în plus oferă o alternativă în cazul în care nu există partea de front-end sau nu este posibil accesul către aceasta, deoarece simulează un request ce ar putea fi trimis de către client către server.
Pe lângă Postman, o altă unealtă folosită pentru a putea testa și verifica dacă request-urile apelate din front-end funcționeză, este framework-ul cunoscut sub numele de Swagger, ce utilizează în principal două librării "Swashbuckle.AspNetCore.dll" și "Swashbuckle.AspNetCore. Swagger.dll". Swagger este un limbaj de descriere a interfeței pentru descrierea API-urilor RESTful exprimate folosind JSON. Swagger este utilizat împreună cu un set de instrumente software open-source pentru a proiecta, construi, documenta și utiliza serviciile web RESTful [13]. Framework-ul este foarte ușor de integrat, oferă o pagină principală din care pot fi apelate toate metodele expuse și existente din controllerele MVC, practic simulând astfel un request real ce ajunge la partea de back-end.

Pe lângă cele menționate, cele doua librării oferă posibilitatea realizării unor clase middleware [14], ce mai poartă numele de "omul în mijloc" sau "clasa intermediară", aceasta având scopul interceptării request-urilor înainte ca acestea să ajungă la controller-ele existente și la funcționalitatea server-ului, în mod normal pentru a putea fi validate și/sau prelucrate.

Un alt framework foarte important pe care l-am utilizat este Entity Framework [15] și este un cadru de mapare obiect-relațional open source pentru ADO.NET, fară de care nu ar fi fost posibilă aproape țoată conexiunea și interogarea bazei de date. Acesta permite crearea conexiunii către baza de date în funcție de un string de conexiune. Pentru interpretarea tabelelor, framework-ul

recunoaște automat maparea către acestea în funcție de o convenție de nume utilizată în cod pe care o atribuim unor seturi de obiecte ce se leagă fiecare de cate un tabel. În general pentru interograrea unei baze de date sunt folosite scripturi sau query-uri SQL, dar Entity Framework folosește query-uri predefinite ce nu sunt accesibile pentru modificare, dar optimizate și reutilizabile pentru multiple tipuri de interogări. Query-urile vin sub forma unor metode C#, făcând mai simplă implementarea acestora fără a ști în mod direct limbajul SQL pentru a putea înțelege cum sunt prelucrate datele într-o bază de date relațională. Astfel framework-ul este recunoscut pentru simplitatea sintaxei și ușurinței pentru configurarea acestuia. În ultimul rand, framework-ul oferă o ustensilă ce are ca scop crearea unor fișiere de migrare pentru crearea automată a bazei de date și a tabelelor, în funcție de clasele create și mapate, reducând timpul și necesitatea mentenanței bazei de date, dar și portabilitatea acesteia pe baza unui backup.

Cu toate acestea, personal nu am utilizat această ustensilă, ci am folosit o ustensilă a IDE-ului prin care baza de date a fost replicată în interiorul folderului proiectului [16], dar pe care am actualizat-o treptat folosindu-mă de aceeași unealtă. Folosind această variantă de replicare și backup, pot vizualiza codul sursă pentru generarea întregii baze de date și alterarea acesteia fară a impacta codul serverului, deoarece uneori mapările nu se schimbă, dar apar mici modificări insesizabile asupra bazei de date, iar această modalitate permite modificarea bidirecțional.

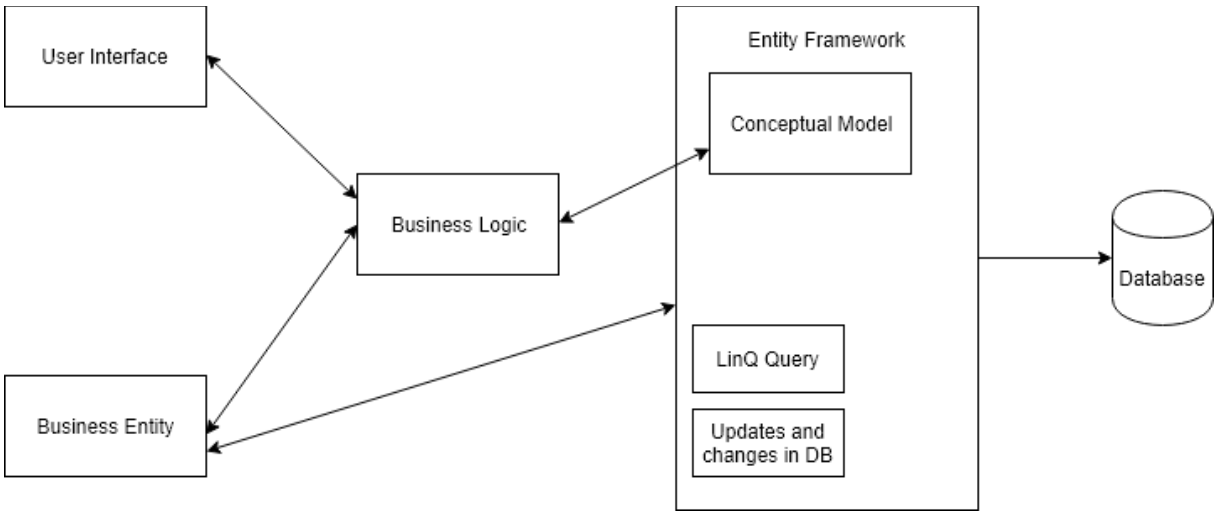

Fig. 2 Exemplu schemă de arhitectură folosind Entity Framework

O tehnologie care completează un pattern pe care l-am implementat ulterior și datorită căreia am redus semnificativ utilizarea memoriei, este framework-ul Ninject [17], pe care l-am folosit împreună cu pattern-ul Composition Root [18] pentru a implementa o clasă reutilizabilă și în același timp singleton, utilizată pe baza depedency injection, astfel încât aceasta leagă orice implementare a claselor "worker" sau "repository", fără a aloca mereu o nouă adresă de memorie datorită instanțierii continue atunci când aveam nevoie de utilizarea unor metode din cadrul acestora. Astfel folosind acest framework, am legat orice interfață de clasa în care era implementată și am făcut posibilă injectarea automată a implementării în diferite workere, prin intermediul constructorului worker-ului specific. Utilizând acest pattern, se va aloca de la lansarea proiectului în memorie spațiu pentru clasa Composition Root, împreună cu implementările instanțiate, pe care o vom injecta mereu prin request-uri către controllerele aferente ce o vor folosi, urmând ca în interiorul acestora să utilizez implementările menționate doar în cazul în care e nevoie de acestea și fără a aloca spațiu suplimentar în memorie.

Acest pattern este recunoscut în cadrul multor companii și utilizat foarte mult în proiecte, deoarece oferă o bună organizare și o ușoară utilizare a logicii din spatele oricărui server.

Anterior am prezentat tehnologia folosită pentru a conecta serverul la baza de date și a putea prelua, insera și altera seturi de date, având numele Entity Framework. În continuare vorbesc despre SQL Server, baza de date aleasă pentru stocarea informațiilor aplicației. Alegerea a venit ca urmare a utilizării acesteia și pe parcursul anilor universitari pentru diferite proiecte școlare, cât și pentru proiecte personale, ca ulterior să fie aprofundată la stagiul dintre anii II și III, și mai apoi în locurile de muncă ce au mai urmat. MS SQL Server este o bază de date relațională oferită de Microsoft, ce vine atât în varianta gratuită, cât și contra cost, și care poate fi accesată atât prin intermediul unor comenzi în terminalul sistemului de operare, dar și prin aplicația pusă la dispoziție de către aceștia ce poartă denumirea de Sql Server Management Studio. Aplicația oferă o interfață vizuală pentru prelucrarea bazei de date și configurarea sa oferind o ușoară interacțiune cu aceasta. Personal am lucrat și cu alte baze de date relaționale, precum MySQL, PostgreSQL, Oracle, cât și nerelaționale cum ar fi MongoDB sau ElasticSearch, dar aceasta oferă avantajul unei conexiuni mult mai ușoară cu orice aplicație ce este realizată pe baza framework-ului .NET Core. De asemenea, IDE-ul Visual Studio oferă o ustensilă menționată anterior pentru crearea locală a unui backup pentru structura bazei de date, sau pentru migrarea sa. Din acest motiv și adăugând experiența anterioară, am decis

că mă voi folosi de ea pentru realizarea întregii aplicații, optând pentru varianta gratuită, deoarece oferea majoritatea opțiunilor pe care le deținea și varianta contra cost.

Pe lângă toate tehnologiile menționate, am mai încercat integrarea și a altor framework-uri, care ulterior s-au dovedit a fi ineficiente pentru tipul acesta de aplicație și care nu ar fi adus un plus de performanță sau de scalabilitate.

II.2. Tehnologii Front-End

Tehnologiile pentru partea de client au reprezentat o provocare, deoarece mă consider o persoană ce îndrăgește și stăpânește mai mult partea de server și logica unui proiect, astfel realizarea interfeței a fost un impediment inițial și a reprezentat un timp de gândire și research, ca într-un final să descopăr framework-ul AngularJs pentru Javascript. Inițial acesta a fost destul de greu de înțeles, sintaxa și conceptele sale nu erau prietenoase cu o persoană începătoare, dar avantajele au fost ușurința integrării, documentațiile și soluțiile propuse de comunitatea ce dezvoltă în limbajul C# partea de server și la rândul lor au putut realiza aplicații web folosindu-se de AngularJs. Astfel o perioadă de timp a constat în acomodarea cu acest framework, încercarea de a-l înțelege și a gândi o arhitectură pentru a fi cât mai scalabilă și eficientă.

Folosind acest framework, nevoia de a folosi cod Javascript pur a dispărut, dar fundamentele acestuia au rămas și la nivelul acestui framework, deoarece sintaxa și algoritmica ce stăteau la baza framework-ului erau aceleași ca la Javascript. Pot spune că Javascript nu îmi era necunoscut, deoarece l-am utilizat încă din perioada liceului și chiar și pe perioada facultății la anumite materii pentru realizarea proiectelor sau la laboratoare. Adăugând suplimentar, o persoană ce nu ar fi cunoscut fundamentele Javascript-ului nu ar fi avut o înțelegere ușoară pentru acest framework și cel mai probabil ar fi fost aproape imposibil de integrat, deoarece așa cum spuneam și anterior, conceptele propuse de AngularJs sunt destul de complexe și necesită o înțelegere mai bună.

AngularJS spre deosebire de utilizarea propriu-zisă a limbajului Javascript pur, oferă dinamicitate interfeței și permite o conexiune cu paginile HTML extrem de rapidă. În plus, posibilitatea creării unor validări și alterări ale interfeței în timp real fac ca acesta să fie o alegere bună pentru acest tip de aplicație web.

Interfața paginilor web a fost realizată folosind HTML și CSS, având un impuls mai mare datorită framework-ului Bootstrap [19], pe care l-am utilizat pentru prima dată în primul an de facultate pentru realizarea unui proiect personal și pe care ulterior am ajuns să îl refolosesc la orice altă aplicație web pe care am dezvoltat-o sau o dezvolt și în prezent, deoarece acesta oferă o multitudine de stilizări predefinite în CSS pentru a crea o interfață receptivă și cu un aspect cât mai plăcut. Cu toate acestea, stilizările oferite de Bootstrap nu erau suficiente și am fost nevoit să recurg la implementări adiționale în CSS pentru crearea unor stilizări sau alterarea celor oferite de framework.

Pe parcursul proiectului au apărut foarte multe fișiere ce conțineau cod de AngularJs sau cod CSS, făcând imposibilă depanarea problemelor direct din consolă pentru dezvoltatori din toate browse-rele, astfel am folosit o ustensilă în Visual Studio ce compila pe baza unui fișier de configurare toate fișierele de AngularJS sau JS selectate, într-un singur fișier principal, iar pentru CSS aceeași procedură, doar că fișierele de CSS trebuiau separate în mai multe fișiere pentru a nu suprapune denumiri identice de stilizări.

Folosind această metodă minimalistă [20], calitatea depanării pentru diferite probleme întâmpinate în timpul dezvoltării a fost sporită și a redus din timpul alocat.

III. Perspectiva utilizatorului

Interacțiunea pusă la dispoziție utilizatorilor este una cât mai simplă, astfel opțiunile pe care aceștia trebuie să le selecteze sau să le parcurgă sunt relativ la îndemâna lor, sau pot fi găsite printr-o căutare într-un interval de timp foarte scurt.

Pentru a putea utiliza aplicația, utilizatorii sunt obligați să aibă un cont de utilizator. Acesta va fi automat redirecționat către pagina de logare, în care trecând doar emailul și parola vor fi validați și în cazul în care credențialele sunt regăsite în baza de date, acesta va fi trimis către pagina principală, altfel va primi un mesaj de eroare. Pentru a crea un cont, utilizatorul are un mesaj ancoră imediat sub formularul de logare, prin care este încărcată pagina de înregistrare în locul celei de logare, la care va urma procedeul de completare a tuturor câmpurilor cerute și după înregistrarea cu succes, acesta va fi trimis către pagina de logare din nou.

Trecând de autentificare, utilizatorii vor observa pe pagina principală inițial că nu există nimic la rubrica de sugestii bazate pe interesele lor, ci doar în rubrica de sugestii ce ar putea să fie de interes sau nu, mai precis aleatorii. Motivul pentru care prima rubrică este goală vine din lipsa unor interese setate în pagina de interese generale, ce urmează să fie prezentată mai jos. Imediat ce un utilizator va avea cel puțin un interes setat, instant pagina principală va avea cel puțin o sugestie pe care să o poată prezenta în acea rubrica. Aspectul acestor sugestii este făcut sub forma unui panou în care este prezentată o poză aleator aleasă de pe internet pentru a înfățișa cât mai mult cu putință orașul promovat. Sub poza acestuia se vor regăsi țara și numele orașului la care se face referire, iar în dreapta acestora vor fi transpuse prin ilustrații, text sau iconițe, toate atracțiile turistice stocate în baza de date referitor la orașul sugerat, cât și informații despre climatul ce poate fi întâlnit și transportul de care dispun turiștii doar pentru acea zonă. Interfața pentru cele două rubrici este bazată pe același aspect, diferența este făcută de dimensiunile alocate în interiorul paginii principale, astfel rubrica secundară pentru sugestii aleatorii este mai restrânsă în partea dreaptă. Motivul pentru care rubrica principală ocupă o mare parte din interfața acestei pagini se datorează unor cercetări făcute în care erau explicate de ce site-uri mari ale lumii au informația principală centrată pe ecranul utilizatorilor și extremele paginilor sunt inexistente. Pe scurt, psihologic s-a demonstrat că oamenii acordă o atenție mai mare informațiilor primite vizual daca aceștia nu sunt nevoiți să "forțeze" privirea pentru a capta informația din locuri ce nu sunt direct intuitive pentru a fi accesate de utilizatori, precum colțul stâng de jos, margina stângă a oricărei pagini și posibil multe alte zone știute sau neștiute [21]. Concluzionând și motivul pentru care rubrica secundară este restrânsă și poziționată în extrema dreaptă, e pentru a oferi o alternativă utilizatorilor, nefiind de interes principal pentru căutarea acestora, aceasta vine ca un sprijin și rămâne ca reper secundar. Înainte de a continua cu următorul aspect al aplicației, menționez că fiecare rubrică este realizată sub forma unui depanator pentru multiple sugestii, acesta fiind setat automat ca la cinci secunde să treacă la următoarea sugestie, iar la final să fie reluată lista de sugestii, cu posibilitatea de a reveni sau a trece mai departe la oricare sugestie pe baza unor săgeți puse în colțul drept al rubricii.

Trecând mai departe, fiecare pagină disponibilă, dispune de un antet ce conține denumirea aplicației web și un meniu de navigare și informare. Denumirea aplicației pe lângă informarea utilizatorilor în legătură cu ce site au accesat, mai oferă posibilitatea redirecționarii către pagina principală printr-un click pe aceasta.

Meniul de navigație vine în ajutor cu patru opțiuni puse la vedere pentru utilizatori. Prima opțiune poartă numele de “Parties” și apăsând pe ea va fi deschis un meniu dropdown în care se vor regăsi toate grupurile pentru drumeții create fie de utilizatorul ce a accesat meniul, fie în care acesta a fost invitat și a acceptat. Fiecare grup are alăturat un contor pentru numărul de persoane existente și active în grup. Apăsând pe unu din grupuri, utilizatorul va fi redirecționat către pagina grupului. Alăturat numelui opțiunii, există un contor pentru a ști în cate grupuri utilizatorul se află, grupuri ce încă există, deoarece grupurile șterse nu vor mai fi disponibile.

Trecând la a doua opțiune, aceasta se numește “Create your Trip” prin care utilizatorii pot crea grupuri. Denumirea de grupuri, drumeții, excursii sau orice alt cuvânt ce poate fi asociat acestui reper, este valid și va fi reutilizat în continuare pentru a nu propune ideea de grup ca fiind principală, deoarece și numele opțiunii face referirea la o drumeție, dar toate acestea împreună dau înțelesul ideii principale pe baza căruia a fost creată aplicația web. Revenind, accesând opțiunea, utilizatorul va fi redirecționat pe pagina în care va alege ce fel de drumeție urmează să creeze, una pe cont propriu sau una de grup. Pentru prima variantă, crearea drumeției este simplă și necesită doar un nume pentru cum se va numi aceasta, iar pentru cealaltă variantă, va fi afișat pe lângă câmpul desinat numelui drumeției, și un câmp pentru adăugarea utilizatorilor invitați pe baza email-ului acestora, cât și un tabel în care vor apărea utilizatorii invitați, din care vor putea fi șterși în cazul în care utilizatorul se răzgândește. La final, este accesat butonul de creare și daca validările aferente trec, iar drumeția este creată, atunci utilizatorul va fi redirecționat pe pagina nou creată grupului.

Cea de-a treia opțiune nu are o denumire vizibilă, dar este realizată sub o formă generală pe care o întâlnim pe majoritatea platformelor, și anume notificările ce sunt reprezentate printr-o iconiță având un clopoțel și un contor în dreapta acesteia pentru numărul de notificări la care nu a fost dat un răspuns sau nu au fost vizualizate. Apăsând clopoțelul, un meniu dropdown va fi extins în cazul în care există cel puțin o notificare, și aici vom regăsi trei tipuri de notificări. Una dintre notificări este cea pentru invitații în grupuri, la care utilizatorii au informație despre denumirea grupului și persoana care i-a invitat, și desigur două opțiuni de acceptare sau respingere a invitației. O altă notificare este cea în care utilizatorul va fi înștiințat dacă a fost dat afară dintr-un grup de care aparținea, de către un administrator. Iar cea de-a treia notificare este legată de cererile de prietenie primite de la alți utlizatori, deoarece fiecare persoană are dreptul de a alege daca dorește să facă

parte din lista de contacte a altei persoane și să aibă acces rapid către aceasta. Notificările vor fi actualizate atunci când utilizatorul schimbă sau reîncarcă aceeași pagină.

Ultima opțiune reprezintă și opțiunea de gestiune pentru contul utilizatorului, regăsită sub denumirea de "Account" care în momentul accesării, va afișă un dropdown cu multiple opțiuni. Prima opțiune reprezintă pagina de interese generale pe care utilizatorul le va configura astfel încât prima pagină să aducă sugestii în rubrica principală discutată anterior, bazate pe interesele selectate aici. Pagina de interese permite utilizatorului să selecteze dintr-o varietate de alegeri, țările și orașele de interes, climatele favorite, modalitățile de transport preferate și atracțiile turistice ce doresc a fi vizitate sau ca punct de reper. Modalitatea de a alege aceste interese, se face pe baza unei pagini modale lansate în momentul accesării fiecăreia dintre opțiunile prezentate, acestea afișând multiple dropdown-uri cu date disponibile spre alegere.

Următoarea opțiune poartă denumirea de "Friends" și precum spune numele, utilizatorul va fi redirecționat către o pagină de gestiune a persoanelor adăugate în lista de contacte. Acestea ulterior vor putea fi utilizate în momentul în care utilizatorul invită persoane la o anumită drumeție, pe baza unui dropdown special generat pentru a-i putea găsi mai ușor.

Opțiunea a treia afișează o pagină ce afișează toate drumețiile create vreodată, atât cele vechi și șterse, cât și cele existente inactive sau active, folosind un tabel în care sunt afișate și având posibilitatea ca utilizatorul să părăsească grupuri sau să șteargă o anumită drumeție în cazul în care acesta va fi singurul rămas în acea drumeție. De asemenea, fiecare drumeție afișată poate fi accesată din acest tabel și utilizatorul va fi redirecționat către aceasta printr-un click pe numele acesteia. Nu în ultimul rand, utilizatorul poate modifica denumirea drumețiilor din pagină dacă deține gradul de administrator în acea drumeție.

Penultima opțiune este destul de simplă și permite modificarea datelor de logare ale contului, precum email-ul și parola.

A cincea opțiune și ultima este cea care deconectează utilizatorul și îl redirecționează către pagina de logare.

Trecând la următorul aspect, mai precis paginile drumețiilor, acestea pun la îndemână în prima parte a paginii, rubrica de sugestii ce urmează a fi afișată pe baza intereselor selectate de fiecare membru al grupului. Prezentarea sugestiilor de locații este asemănătoare celor din prima pagină a

site-ului, astfel aspectul va fi similar, iar parcurgerea acestora va fi posibilă sub forma unui depanator de imagini. Sugestiile vor fi actualizate atunci când utilizatorul reîncarcă pagina, fie când modifică sau accesează rubrica de modificat interese.

Pentru a putea selecta interesele dorite, o rubrică asemănătoare cu cea din pagina de setări generale pentru interese, se va regăsi imediat sub rubrica de prezentat sugestiile, iar pentru a putea discuta și gestiona utilizatorii grupului, în partea dreaptă a rubricii de interese se va regăsi un chat în care pot comunica utilizatorii, iar atașat de acesta în partea dreaptă superioară vor putea accesa o iconiță ce deschide un meniu dropdown pentru a gestiona utilizatorii grupului, cât și posibilitatea de a reseta interesele selectate sau a părăsi grupul și drumeția.

Legând toate aceste aspecte, aplicația oferă un aspect cât mai simplist și ușor de navigat, făcând experiența utilizatorului plăcută și simplă.

Culorile alese pentru crearea interfeței au fost selectate în urma unor cercetări pentru a înțelege gândirea unei persoane ce navighează pe site și ce l-ar putea convinge să rămână în continuare pentru a utiliza aplicația. În urma acestor căutări, s-a constatat faptul că majoritatea persoanelor navighează pe site-uri ce au culori deschise și foarte puțin proeminente, fără a deranja ochiul uman și fără a forța utilizatorul să acorde o atenție mai mare pentru a înțelege lucrurile din interfață. Inițial dorința personală a fost de a folosi o temă întunecată, deoarece temele care folosesc culoarea neagră predominant sunt considerate elegante, dar numărul utilizatorilor care optează pentru această perspectivă este mai mic, motiv pentru care nevoia de a selecta o varietate de culori pentru a dezvolta o interfață deschisă la culoare, a fost mai mult o necesitate pentru a satisface nevoia majorității utilizatorilor, scopul principal fiind acela de a atrage utilizatori cât mai mulți.

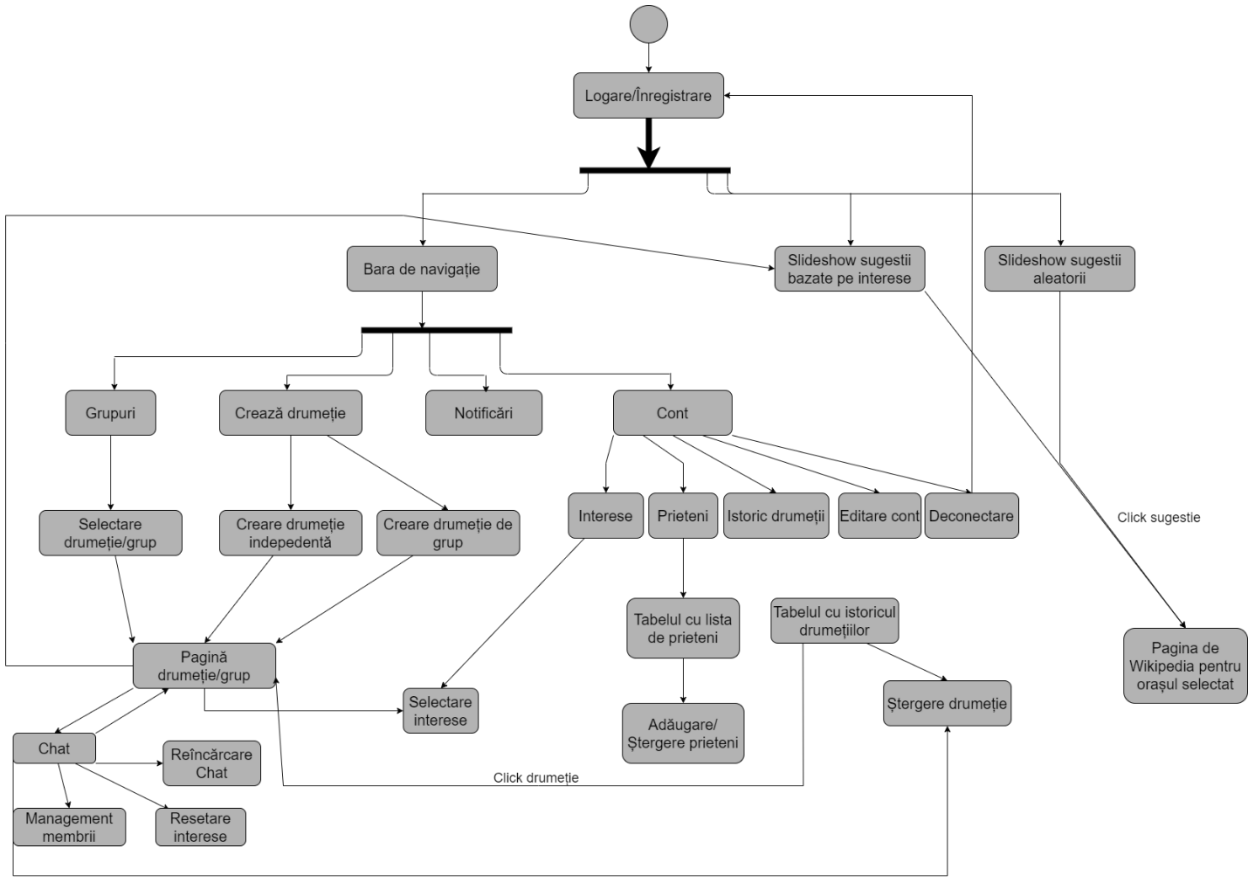

Fig. 3 Diagrama de activitate

IV. Perspectiva dezvoltatorului

Când am ales să încep dezvoltarea acestei aplicații, mi-am propus să mă folosesc atât de ce știu mai bine, cât și de ce știu mai puțin. Prin urmare, decizia de a realiza partea de server a aplicației a venit în urma cunoștințelor și experienței acumulate folosind limbajul C# împreună cu framework-ul principal .NET Core, dându-mi un avantaj pentru realizarea și gândirea arhitecturii astfel încât să pot obține o soluție scalabilă, alături de care să vin în completare cu diferite design pattern-uri învățate sau descoperite în timpul dezvoltării pentru a obține o aplicație cât mai bine structurată și optimă. Pe de altă parte, dorința personală de a învăța tehnologii noi și aplicarea acestora în proiectul ales, m-a împins către framework-ul AngularJS pentru partea de interfață sau de client. Nu pot nega faptul că baza de date utilizată reprezenta un punct forte când am decis să folosesc SQL Server din suita Microsoft, deoarece până la momentul începerii proiectului am

utilizat acest tip de baze de date destul de puțin în comparație cu capacitatea totală pe care aceasta o punea la dispoziție.

Punând cap la cap aceste tehnologii principale, dezvoltarea aplicației a început inițial prin proiectarea arhitecturii soluției realizate cu ajutorul Visual Studio Community, fiind pe tot parcursul dezvoltării principalul IDE utilizat. Arhitectura inițial a fost structurată din trei proiecte, și anume un proiect principal în care există toată logica de business, server-ul propriu-zis și fișierele interfeței, un proiect ce conținea metodele de bază pentru operațiuni cu baze de date, cât și modelele mapate acestor operațiuni, cât și celor din logica de business, iar ultimul proiect reprezenta zona de backup pentru structura bazei de date. În prezent, așa cum am putut menționa și mai sus, proiectul a suferit modificări la nivel de organizare și logică de funcționare, motiv pentru care acesta s-a împărțit în multiple proiecte în urma aplicării pattern-ului numit "Repository Pattern". Acest patern crează un mediu favorabil scalării proiectului pentru permiterea realizării unor aplicații pe alte tipuri de dispozitive, folosindu-mă de aceleași proiecte ce vin sub forma unor librării pentru serverul aplicației web.

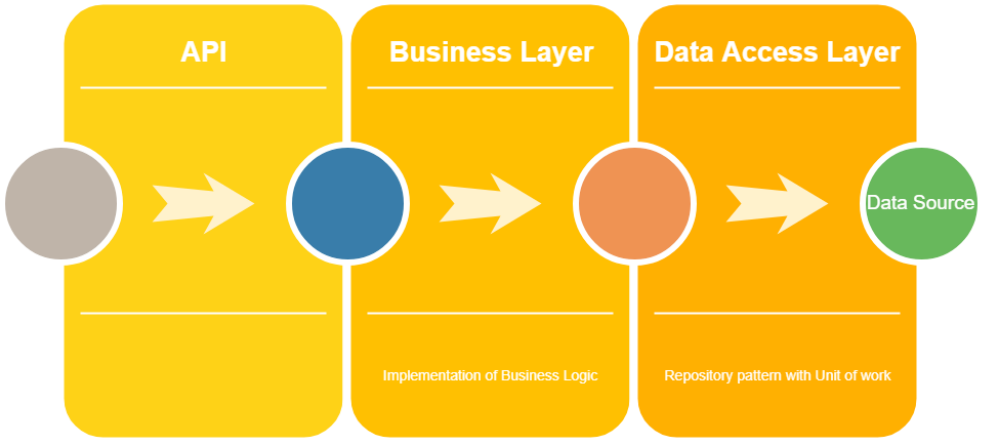

Fig. 4 Schema structurii proiectului folosind pattern-ul "Repository"

Pe lângă acest pattern, am venit în ajutor cu un nou pattern numit „Composition Root Pattern" descoperit în momentul în care am început să învăț despre ce înseamnă și cum poate fi utilizată injectarea dependințelor. Injectarea dependințelor este o tehnică în care un obiect primește alte obiecte de care depinde. Aceste alte obiecte sunt numite dependințe. Practic acest pattern reduce numărul instanțelor create pentru clasele utilizate, făcând transferul unei singure instanțe necesare

prin requestul HTTP ce vine în momentul accesării API-ului asociat unui controller. Injectarea poate fi de trei tipuri [22] și anume, injectarea tranzitoriu în care fiecare instanță va fi diferită la orice request și orice controller accesat, injectarea cu scop în care instanța este diferită la orice request, dar aceeași în orice controller ar fi folosită pe parcursul aceluiași request, și injectarea singleton sau globală cum îi mai spun personal, în care instanța va fi una singură pe tot parcursul rulării serviciului, astfel orice request va refolosi aceeași instanță a unui obiect.

Folosind pattern-ul „Composition Root”, am creat o dependință injectată singleton pentru clasa composition root, deoarece aceasta trebuia să fie mereu aceeași pe tot parcursul utilizării serviciului. În schimb, metodele și clasele injectate folosind această clasă intermediar, erau puse sub forma injectării tranzitoriu. Inițial nu exista injectare tranzitorie la nivelul acestora, ci era una cu scop, dar motivul principal pentru trecerea aceasta a fost depistarea unei probleme ce bloca reutilizarea unui obiect pe care îl foloseau simultan două clase diferite, fapt pentru care rezulta într-o excepție aruncată aproximativ la câteva secunde după pornirea serverului.

```
private static volatile CompositionRootBackend _instance; //obiectul ce
păstrează o singură instanță. Sintaxa Volatile indică tuturor thread-urilor
dacă instanța s-a modificat, iar orice modificare se manifestă în timp real,
indiferent daca alt task paralel accesează variabila _instance
private static object _syncRoot = new object(); //obiectul care asigură
creearea unei cozi de acces asupra creării unei noi instanțe în cazul în
care nu există
public static CompositionRootBackend Instance
{
    get
    {
        if (_instance == null) //prima verificare pentru existența unei
instanțe
        {
            lock (_syncRoot) // blocarea accesului folosind variabila
syncRoot, care se modifică când un task a intrat în această bucată de cod și
revine la starea inițiala cand task-ul a ieșit din blocul de cod lock
            {
                if (_instance == null) // a doua verificare pentru existența
unei instanțe si cea care asigură limitarea la o singură instanță
                {
                    _instance = new CompositionRootBackend(); //crearea
instanței
                }
            }
        }
        return _instance;
    }
}
```

Bucata de cod atașată reprezintă forma prin care clasa Composition Root creează o singură instanță folosind pattern-urile „Singleton” și „Double-checked locking” [23]. Pentru a înțelege mai bine, conceptul de Singleton presupune limitarea numărului de instanțe create pentru o clasă la o singură instanță, deoarece este foarte util atunci când un singur obiect trebuie să coordoneze acțiunile din întreg sistemul, iar acesta nu are voie să fie diferit pentru orice fel de request ar ajunge la server. De asemenea, „Double-check locking” reprezintă modalitatea de a securiza instanțierea unui obiect singleton, fără a oferi posibilitatea ca două surse să acceseze instanțierea simultan, astfel punând „lacăt” și oprind continuarea creării unei noi instanțe pentru o parcugere paralelă a unui număr mai mare sau egal de două surse. Acest aspect a fost testat folosind o listă cu două elemente care sunt accesate folosind obiectul Parallel din librăria System.Threading.Tasks.Parallel.dll, care trecea simultan prin cele două elemente ale listei și accesau instanța singleton. În primul caz în care nu există a doua verificare “_instance == null”, ambele task-uri treceau simultan de prima verificare, dar intrau pe rând în blocul de cod “lock”, astfel fiecare ajungeau să creeze o nouă instanță. Însă, adăugând ulterior și a doua validare pentru instanță, primul task apucă sa creeze o instanță, iar cel de-al doilea era deja înștiințat de noua modificare oprind posibilitatea ca acesta să mai recreeze o nouă instanță.

Codul descris a reprezentat un factor important pe care l-am învățat în cadrul dezvoltării oricărei aplicații, deoarece există o șansă mare de a introduce cel puțin pattern-ul Singleton pentru o posibilă funcționalitate. Fără cel de-al doilea pattern specificat, posibilitatea ca două sau mai multe task-uri paralele ce accesează obiectul instanță în care fiecare creează o nouă instanță, este destul de mare, deoarece sintaxa „lock” nu ar fi suficientă pentru a opri continuarea codului imediat după prima instanță, astfel modalitatea aceasta de a verifica de două ori dacă instanța a fost creeată, va asigura că o referință a fost deja creeată pentru acea instanță.

Un ultim pattern care a îmbunătățit calitatea codului și a obligat crearea unor funcționalități de bază repetitive pentru multiple clase, se numește „Strategy Pattern” [24] și este un design pattern care permite să definim funcționalități diferite, să punem fiecare funcționalitate într-o clasă separată și să facem obiectele interschimbabile. Practic, fiecare clasă repository ce execută operațiuni către baza de date, fără îndoială va executa cel puțin un query de inserare a unor date către o tabelă, o modificare a unuia sau a mai multor câmpuri unei tabele, cât și preluarea sau ștergerea unor rânduri din aceste tabele. Aceste query-uri de multe ori sunt aceleași, cât și

funcționalitatea implementată în codul C#, dar diferă tabela pe care se execută comanda, motiv pentru care fiecare dintre aceste clase repository, sunt obligate să implementeze metodele de bază pe care le vom regăsi în fiecare dintre acestea. Astfel, viitoarele clase ce ar putea fi create, vor avea asigurată o funcționalitate obligatorie prin care va acoperi cazurile cele mai frecvent utilizate.

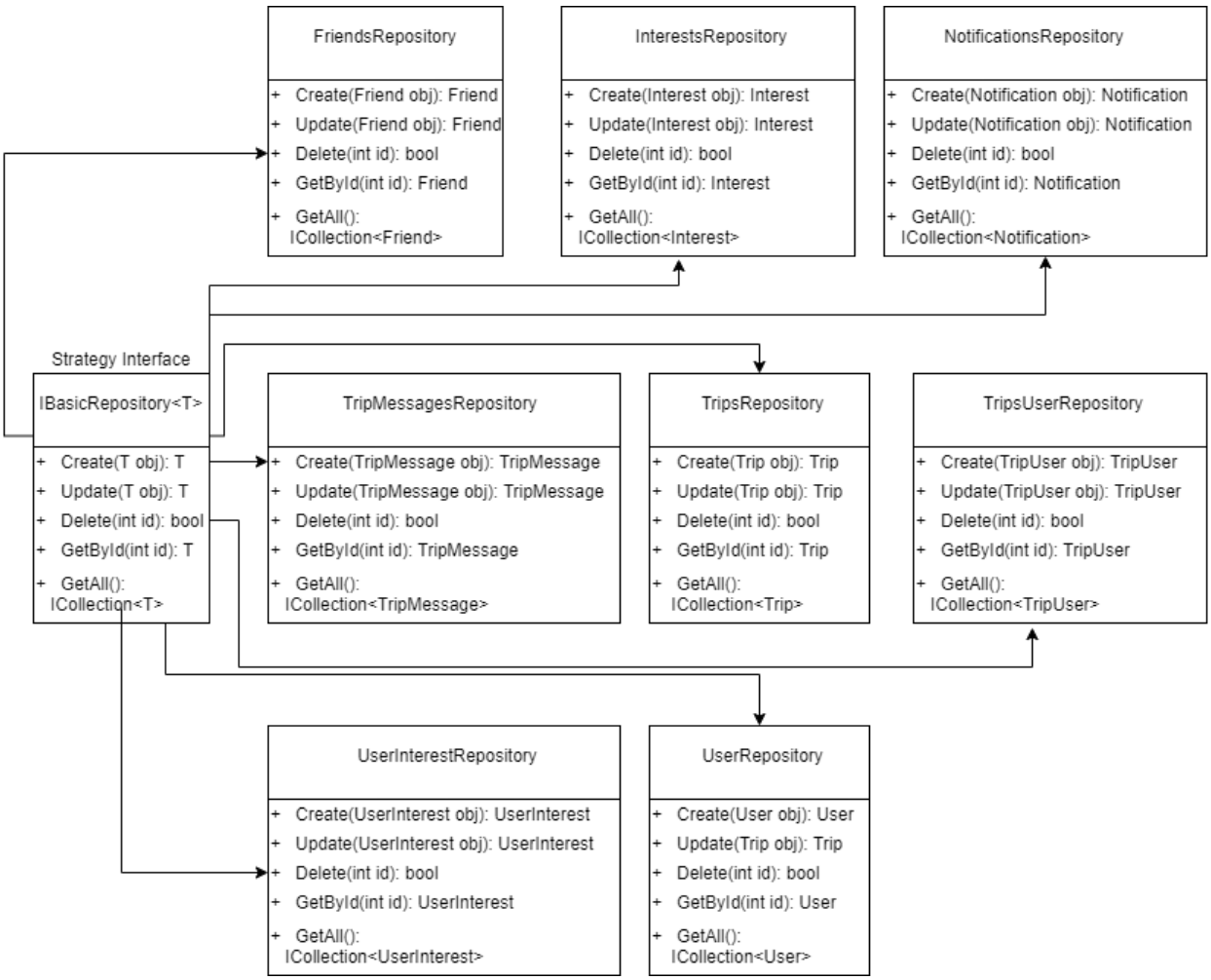

Fig. 5 Strategy Pattern utilizat in aplicație

Trecând mai departe, realizarea acestor metode pentru apelarea unor query-uri la nivelul bazei de date, a fost făcută în proporție de 90% cel puțin, cu ajutorul framework-ului Entity Framework, fiind unul dintre cele mai răspândite și utilizate la nivel global în proiectele companiilor de renume, cât și în proiecte personale. Entity Framework abreviat și EF, este un cadru open source de mapare relațională pe obiect (ORM) [25] pentru ADO.NET. În spate, acest framework a fost conceput

utilizând un set de tehnologii ADO.NET optimizate pentru a obține cea mai bună performanță și a fi reutilizate pentru dezvoltarea de aplicații software orientate pe date. Pentru a pune în funcțiune acest framework, au fost necesari următorii pasi:

- Crearea unei clase ce moștenește clasa “DbContext” fiind clasa principală pe care se creează maparea datelor din baza de date atașată conexiunii.

- Setarea și configurarea acesteia să mapeze pe baza unor DbSet-uri, toate tabelele pe care dorim să le extragem ulterior.

- Crearea modelelor utilizate în cod pentru a putea fi mapate către tabelele asociate, astfel fiecare properitate să reprezinte un câmp anume din tabelă.

- Definirea instrucțiunilor prin care fiecare model să fie interpretat corect de EF pentru a înțelege tipul coloanei mapate sau felul în care e dorită convertirea acesteia, cât și alte limitări sau impuneri la nivel de funcționalitate.

- Setarea conexiunii către baza de date folosind textul de conexiune, care poate fi generat automat sau manual creat folosind formatul propus.

Ulterior după ce toate aceste configurări au fost făcute, mai rămâne doar crearea unei instanțe pentru clasa context nou generată și accesarea modelelor mapate.

Un lucru interesant de menționat pe care Entity Framework îl pune la dispoziție, este posibilitatea de a obține informații din altă tabelă despre o intrare ce aparține tabelei prelucrate în acel moment, prin două linii de cod. Luând ca exemplu tabela Friends și configurarea mapării sale la clasa din codul C#.

```
static internal class FriendDefinitions
{
    public static void Set(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Friend>().HasKey(x => x.Id);
        modelBuilder.Entity<Friend>().Property(x => x.UserId).IsRequired();
        modelBuilder.Entity<Friend>().Property(x =>
x.FriendId).IsRequired();
        modelBuilder.Entity<Friend>().Property(x =>
x.IsDeleted).IsRequired();
        modelBuilder.Entity<Friend>().HasOne(c =>
c.FriendAccount).WithMany().HasForeignKey(c => c.FriendId);
    }
}
```

ModelBuilder reprezintă clasa prin care EF creează un set de reguli, fie de mapare, fie de restricții sau reguli de interpretare a datelor ce urmează sa fie transferate sau preluate din tabelă, prin care specificăm oricărei proprietăți ce aparține clasei asociate, în cazul acesta Friend, cum să se manifeste în cadrul unei mapări prin Entity Framework. Ce e totuși interesant aici și despre care voiam să specific, este ultima mapare din metodă.

```
modelBuilder.Entity<Friend>()
            .HasOne(c => c.FriendAccount)
            .WithMany()
            .HasForeignKey(c => c.FriendId);
```

Câmpul FriendAccount reprezintă un obiect de tipul User, în care sunt preluate informațiile despre utilizatorul care este asociat coloanei FriendId, respectiv proprietății FriendId din clasa Friend. De regulă, proprietatea aceasta este ignorată în momentul interacțiunii prin EF cu tabela respectivă, doar dacă nu este utilizată metoda „Include", ce are rolul unui „JOIN" din cadrul SQL și care preia din tabela Users toate informațiile ce pot fi mapate la proprietatea FriendAccount.
Această funcționalitate a fost implementată în cadrul clasei FriendsRepository, în care a fost implementată metoda GetByUserId prin care preluam toți prietenii unui utilizator, dar despre care doream să obțin și informații suplimentare precum emailul acestora.

```
public ICollection<Friend> GetByUserId(int userId, bool includeDeleted =
false)
{
    using (TripPlanner context = new TripPlanner())
    {
        return context.Friends.Where(c => c.UserId == userId && c.IsDeleted
== includeDeleted).Include(x => x.FriendAccount).ToList();
    }
}
```

Astfel, în cadrul context-ului, este interogată tabela Friends pe baza asocierii id-ului utilizatorului și ca acesta să aibă incă o relație de prietenie cu persoanele selectate (c.IsDeleted trebuie să aibă valoarea negativă pentru a sugera că cei doi încă au o conexiune de prietenie), ca mai apoi pentru rezultatele obținute să fie aduse suplimentar și informații despre prieten prin legarea proprietății FriendAccount de conexiunea deschisă.

Prin urmare, această unealtă oferită de EF simplifică interacțiunea cu baza de date și înlocuiește nevoia implementării unui query mai complex pentru realizarea acestui tip de instrucțiune.

Având maparea către tabelele utilizate, mai departe aplicația trebuie construită în jurul acestora, astfel atât proiectul cu logica de business, cât și cel cu metodele de bază, vor fi construite pentru fiecare în parte, mai precis în foldere specifice.

Pentru a înțelege mai bine ce reprezintă fiecare tabel, mai jos voi prezenta pe scurt fiecare în parte:

- Principala tabelă este cea a utilizatorilor denumită "Users", în care vor exista conturile stocate în momentul înregistrării și mai ales cele cu care se pot loga în aplicație. Fiecare utilizator va fi recunoscut în funcție de emailul ales, deoarece invitațiile ce vor fi trimise sunt bazate pe emailurile acestora.

- Pentru a putea crea un set de interese, aveam nevoie de o multitudine de interese dintre care utilizatorii să poată alege, motiv pentru care a fost necesară crearea tabelei "Interests". Alterarea acesteia se poate face doar manual accesând baza de date, deoarece valorile stocate sunt predefinite, mai precis utilizatorii nu au acces la alterarea acestora.

- Cu toate că există tabela cu interese, utilizatorii trebuie să aibă cum să poată alege aceste interese și să rămână stocate undeva, motiv pentru care apare în discuție tabela "UserInterests" în care sunt stocate fie interesele generale ale utilizatorului, fie interesele selectate pentru fiecare grup sau drumeție în parte.

- Trecând mai departe, utilizatorii am spus mai devreme că pot face parte din grupuri turistice pentru planificarea unor drumeții, sau chiar pe cont propriu, prin urmare tabela "TripUsers" va reprezenta locația unde sunt mapati utilizatorii alături de drumeția de care aparțin, drumeție care la rândul ei va fi creată și stocată în tabela "Trips".

- În grupuri, utilizatorii pot discuta pentru a planifica execursia dorită, de aceea mesajele acestora vor fi stocate pentru a putea revedea ulterior discuțiile avute, astfel tabela "TripMessages" va stoca fiecare mesaj trimis de utilizatori, împreună cu drumeția la care a fost trimis mesajul respectiv.

- Cu toate că putem invita utilizatorii să facă parte din anumite grupuri, aceștia trebuie să poată vedea aceste invitații, sau daca s-a întâmplat ceva anume cu grupul și a fost desființat sau o persoană a fost dată afară, de asemenea trebuie utilizatorul înștiințat, de aceea tabela "Notifications" va stoca toate notificările pe care utilizatorii le pot primii, având posibilitatea să interacționeze cu anumite tipuri de notificări pentru a le accepta sau respinge.

- Ultima tabelă și care vine în ajutorul utilizatorilor, este accea de "Friends" prin care sunt memorate conexiunile dintre utilizatori, deoarece fiecare utilizator are posibilitatea de a adauga în

lista de prieteni unul sau mai mulți utilizatori pe baza email-ului acestora. Tabela oferă ajutor suplimentar în momentul în care administatorii de grupuri vor să invite diferite persoane și doresc să economisească timp pentru a căuta email-ul persoanei dorite.

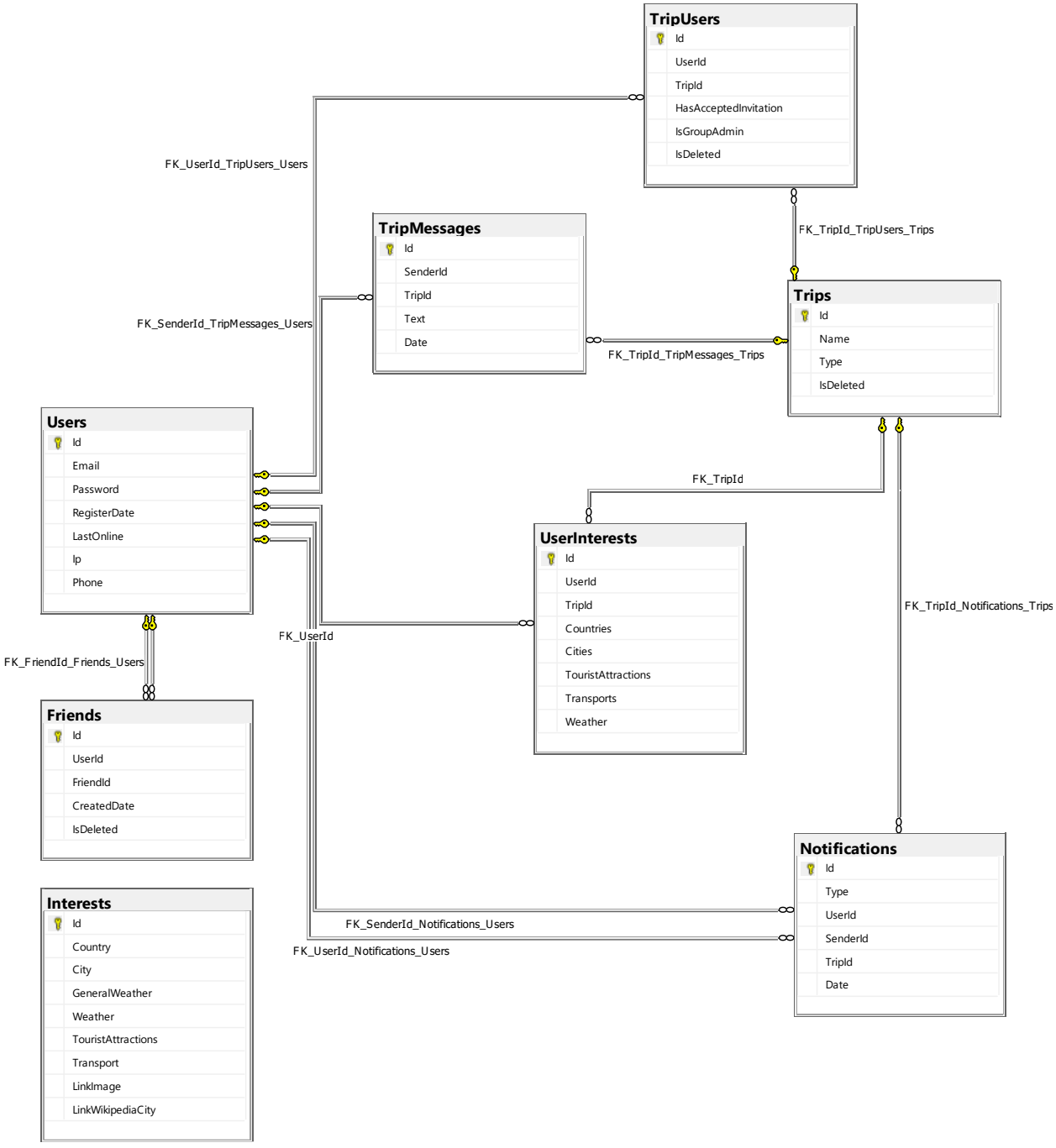

Fig. 6 Diagrama bazei de date

Folosind diagrama bazei de date, am expus legăturile între tabele prin figura de mai sus. Fiecare tabelă conține un ID principal care este un Primary Key, dar în același timp și o identitate ce e incrementată automat la fiecare intrare înregistrată nouă în tabela respectivă. De asemenea, se pot observa tabelele principale Users și Trips, deoarece nu au nicio legatură dependentă cu o altă tabelă, în schimb toate celelalte tabele prin coloane Foreign Key precum UserId, TripId, SenderId, FriendId formează conexiuni directe cu cele două, forțând orice date ce urmează a fi inserate în acele coloane să existe automat și în coloana asociată din Users sau Trips. Această modalitate ajută foarte mult la validarea intrărilor noi, deoarece sunt excluse și aruncate ca excepție cele care conțin valori inexistente în tabela părinte. Singura tabelă fără nicio conexiune este Interests, deoarece stochează doar informațiile ce pot fi selectate de utilizatori, însă nu există certitudinea că acestea vor rămâne aceleași mereu, deoarece informații legate de climat, transport, atracții, ș.a.m.d., pot să se schimbe oricând, devenind irelevantă alegerea utilizatorului.

Având structura bazei de date înconjurul căreia se desfășoară toată activitatea aplicației, trebuie menționată și configurarea serverului principal, cât și legătură pe care o are interfața cu acesta.

Atunci când am gândit arhitectura soluției, aveam cunoscute doua mari concepte ale dezvoltării de aplicații, anume conceptul de arhitectura monoltică și conceptul de arhitectură bazată pe microservicii [26]. Arhitectura monolitică se bazează pe crearea unui server ce adună toată funcționalitatea și logica aplicației și o gestionează din aceeași locație, pe când arhitectura bazată pe microservicii separă fiecare logică de business în multiple „mini servere” le-aș numi personal, fiecare fiind independentă una de cealaltă. Avantajul pe care îl are o arhitectură monolitică față de cea bazată pe microservicii e ușurința de accesare a funcționalității și modificarea acesteia, cât și conexiunea directă și rapidă între resurse. Pe de altă parte, arhitectura pe microservicii oferă o gestiune mai bună a traficului pe care îl produc utilizatorii când execută anumite request-uri, dar și o ușoară alterare a anumitor servicii fără a fi nevoie să impacteze întreaga soluție, deoarece acestea pot fi făcute separat.

Cu toate acestea, decizia mea a fost aceea de a folosi arhitectura monolitică din următoarele motive:

- traficul utilizatorilor nu ar fi prea mare, deoarece aplicația în sine nu necesită prezența utilizatorilor mult timp.

- funcționalitatea este una oarecum simplă și nu necesită accesarea multor request-uri.

- resursele sunt într-o continuă conexiune, motiv pentru care este necesară o accesare rapidă a acestora.

Pentru o bună funcționalitate si garanția că serverul poate să primească și să interpreteze orice request, am menționat în subcapitolul Tehnologii Back-End despre frameworkul Swagger utilizat pentru a testa și simula request-urile apelate de către client spre server. Configurarea acestuia a fost relativ simplă și ușor de înțeles, și necesită modificarea fișierului Startup.

Prima dată a fost nevoie ca metoda Configure [27] să fie modificată prin adăugarea următoarelor linii de cod.

```
app.UseSwagger();

app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Multishare Backend API");
});
```

Metoda UseSwagger activează middleware-ul pentru a servi Swagger generat ca punct final JSON, iar cea de-a doua UseSwaggerUI, împreună cu metoda SwaggerEndpoint, activează middleware-ul pentru a servi interfața Swagger (creată prin HTML, CSS, JS, etc.) și specifică punctul final pentru JSON-ul Swagger. Sub o formă mai simplă, prima metodă creează conexiunea către server și configurează mesajul JSON trimis către acesta, iar cealaltă metodă pune la dispoziție interfața ajutătoare prin care pot fi executate aceste request-uri ce trimit JSON-ul configurat către server.
Dar încă nu este suficient, mai trebuie adăugată o mică setare pentru fi funcțional mediul de testare prin Swagger, în cadrul metodei opționale numită ConfigureServices.

```
services.AddSwaggerGen();
```

Metoda AddSwaggerGen înregistrează generatorul Swagger, definind unul sau mai multe documente Swagger, adică serviciul web pe care rulează Swagger este configurat să înțeleagă legătura către acesta și să îl poată interpreta pentru a fi afișat când utilizatorul accesează pagina sa.

Aceste mici configurări permit o utilizare suficientă pentru a testa toate request-urile posibile către server, dar suplimentar există opțiuni ce facilitează diverse funcționalități pentru framework-ul Swagger.

Mai jos puteți observa un cadru din interiorul paginii de testare generată prin această ustensilă.

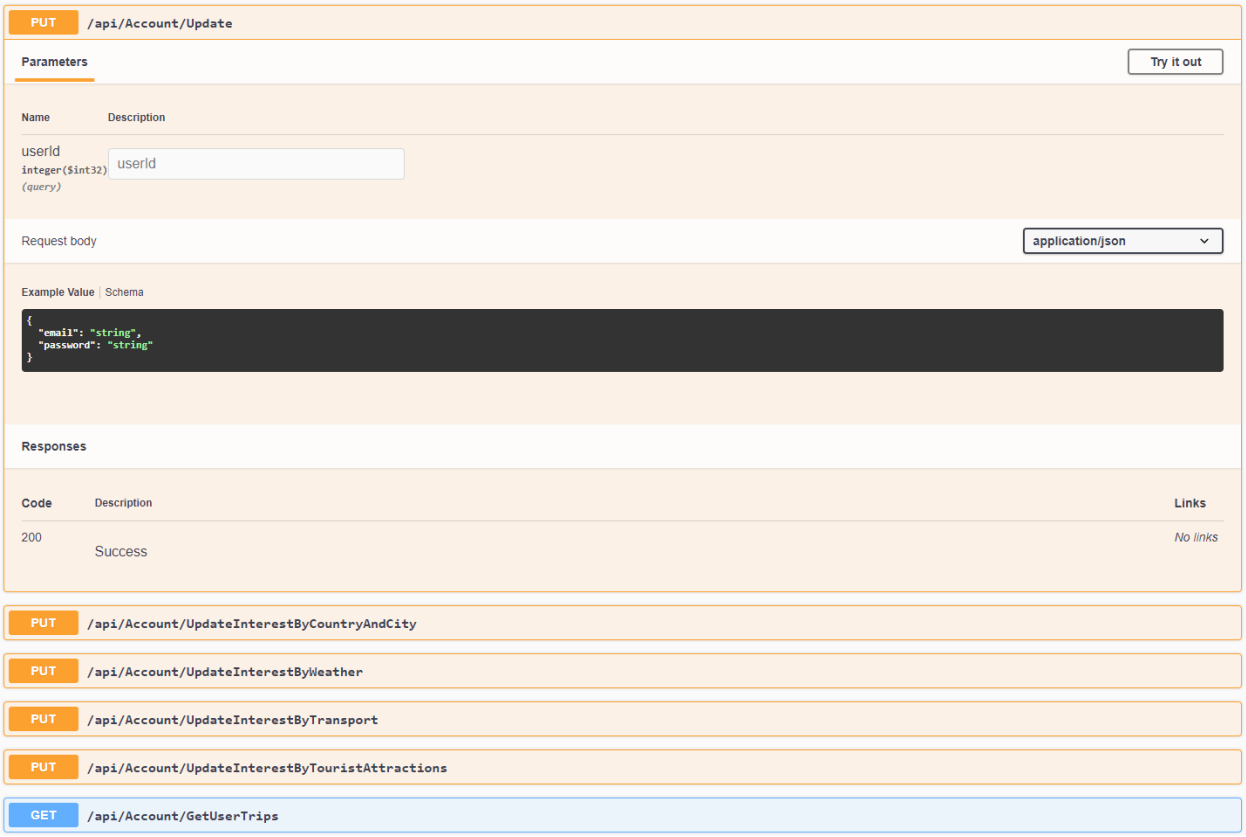

Fig. 7 Exemplu de interfață Swagger

Fiecare request are asociat protocolul corespunzător, destinația, exemplul de corp și parametrii pe care trebuie să îl conțină request-ul, dar și rezultatul final în caz de succes (de asemenea, pot fi prelucrate să fie returnate și alte coduri pe langă cel vizibil de 200 pentru succes), iar butonul „Try it out” permite editarea formularului pentru construirea request-ului.

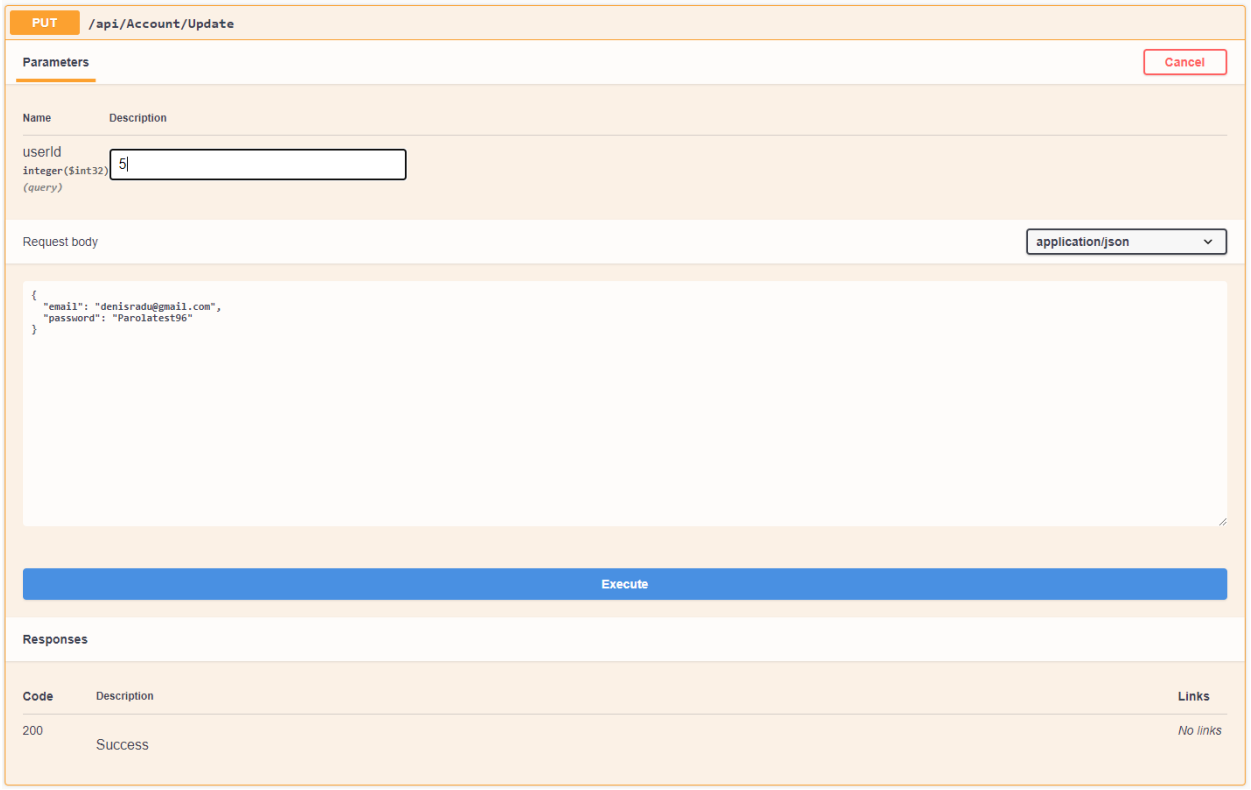

Fig. 8 Interacțiune cu interfața Swagger

Odată apăsat butonul, formularul poate fi editat cu informațiile necesare, iar la final executat pentru a fi trimis către metoda controller-ului asociat acestui request.

De precizat un aspect foarte important, toate request-urile necesită autorizare pe partea de server pentru a putea accesa controllerele și metodele, mai puțin pagina în care utilizatorul se poate înregistra sau loga pe aplicație, deoarece reprezintă punctul de start de la care utilizatorul poate obține acces în aplicație. Această modalitate de securitate este dată de un mecanism format din librăriile:

-Microsoft.AspNetCore.Authorization

-Microsoft.AspNetCore.Authentication.JwtBearer

-Microsoft.IdentityModel.Tokens

În primul rând, pentru a putea ca server-ul să înțeleagă modalitatea de autorizare și autentificare, sunt necesare anumite setări în fișierul Startup.

```
services.AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
```

```
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(x =>
{
    x.RequireHttpsMetadata = false;
    x.SaveToken = true;
    x.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = false,
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.ASCII.GetBytes(GlobalConstants.JwtSecretKey)),
        ValidateAudience = false
    };
});
```

Prin configurația de mai sus, sunt setate criteriile de interpretare ale token-ului ce vine prin header-ul request-ului. Pentru forma predefinită de validare pe baza unui token, este utilizată autentificarea Bearer, care este și valoarea returnată de variabila JwtBearerDefaults.AuthenticationScheme. Autentificarea Bearer, mai poartă numele de Autentificare Token și este o schemă de autentificare HTTP care implică token-uri de securitate numite "bearer tokens". Apoi, instrucțiunea AddJwtBearer îi spune serviciului web, ce criterii suplimentare de validare trebuie să întrunească token-ul pentru a fi valid. Personal am avut nevoie de validarea unui cod secret stocat într-o constantă ce este atribuit mereu fiecărui token generat, fiind suficient ca o simplă condiție suplimentară pentru veridicitatea acestuia, dar am fost nevoit să opresc validarea condițiilor pentru audiență și emitent, deoarece nu erau configurate și aplicația nu putea funcționa.

În continuare, ca server-ul să știe pe ce fel de autorizare să se bazeze, a fost setată interpretarea token-ului configurat anterior, pentru a reprezenta modalitatea de acces.

```
services.AddAuthorization(options =>
{
    options.DefaultPolicy = new AuthorizationPolicyBuilder()

.AddAuthenticationSchemes(JwtBearerDefaults.AuthenticationScheme)
                                    .RequireAuthenticatedUser()
                                    .Build();
});
```

Metoda AddAuthorization setează server-ul să permită restricționarea accesului asupra sa, iar opțiunea DefaultPolicy reprezintă modalitatea generală pentru orice autorizare, căruia îi este atribuită autentificarea Bearer setată anterior.

Mai departe, serviciul web trebuie să stie unde să se uite în timpul request-ului pentru a interpreta și valida token-ul primit și în același timp, metoda de autorizare primită de la client.

```
app.UseAuthentication();
app.UseAuthorization();
```

Cele două metode configurează server-ul să înțeleagă modalitatea de autentificare prin token și să o folosească pentru a permite, sau nu, accesul în aplicație.

Mai departe pentru a pune în aplicare restricția de acces, tot ce trebuie este adăugarea fie la nivel de clasă controller, fie la nivel de metodă a unui controller, atributul "Authorize" care reprezintă un middleware ce validează accesul și permite trecerea la următorul pas, sau în caz negativ, este întors codul status 401, ce reprezintă în protocolul HTTP răspunsul care precizează că accesul nu a fost permis.

Nu în ultimul rând, token-ul generat care va permite accesul, este generat folosind următorul bloc de cod:

```
var claims = new[]
{
    new Claim(JwtRegisteredClaimNames.Sub, user.Id.ToString()),
    new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
    new Claim(JwtRegisteredClaimNames.Email, user.Email)
};

var tokenHandler = new JwtSecurityTokenHandler();

var symmetricSecurityKey = new
SymmetricSecurityKey(Encoding.ASCII.GetBytes(GlobalConstants.JwtSecretKey));
var signingCredentials = new SigningCredentials(symmetricSecurityKey,
SecurityAlgorithms.HmacSha512Signature);

var jwtSecurityToken = new JwtSecurityToken(
    claims: claims,
    signingCredentials: signingCredentials,
    expires: DateTime.UtcNow.AddMonths(3)
    );

var token = tokenHandler.WriteToken(jwtSecurityToken);
return token;
```

Prima dată sunt setate claim-urile, ce reprezintă însușiri ale utilizatorului. Pe baza unor claim-uri, pot fi setate date importante, senzitive, ce sunt folosite pentru recunoașterea utilizatorului în momentul autorizării. În această aplicație nu se ține cont de acestea, însă sunt folosite pentru generarea unui token unic, care să nu mai poată fi asignat altui utilizator. Apoi, este setată cheia secretă de acces către acest serviciu pe care am configurat-o si anterior, dar care este criptată (și decriptată ulterior de server), în setul criptografic Sha512 pentru o mai bună securitate. La final

token-ul este generat folosind opțiunile alese și returnat utilizatorului pe client. Acesta va fi stocat în browser-ul clientului cât timp este valabil.

Trecând mai departe, serverul principal și proiectul principal în același timp, este configurat să interpreteze fișierele interfeței sub formă statică, mai precis interfața este încă de la pornirea serverului setată să fie asociată aceluiași IP și aceluiași port, făcând ușoară trecerea către și dinspre client-server. Cu alte cuvinte, partea de front-end și back-end sunt hostate pe același dispozitiv, fiind într-o dependință strânsă.

Fiind o aplicație web, framework-ul .NET Core permite compilarea fișierelor statice pentru interfață sub o formă prestabilită într-un fișier de configurare, astfel toate fișierele de Javascript sau AngularJs vor fi compilate sub un singur fișier general, dar și fișierele de stilizare vor fi puse sub aceeași configurație. La compilarea acestora, ele sunt mutate automat într-un folder specific numit “wwwroot” [28], în care apare tot codul compilat sub noua formă și care va fi citit pentru interpretare în momentul lansării aplicației.

Realizarea interfeței nu ar fi fost posibilă, fără puțin ajutor din partea framework-ului Bootstrap, care oferă o gama largă de stilizări predefinite și gata de utilizat. Acesta a făcut posibilă crearea unor tabele, formulare și butoane utilizate în mod frecvent pe întreaga aplicație. De asemenea, stilizări manuale au fost necesare pentru a putea obține aspectul dorit, motiv pentru care au fost create fișiere suplimentare de CSS.

Fiecare pagina pentru a putea funcționa corespunzător, are asociat un controller creat în AngularJs, care creează și request-urile ce vor prelua sau vor trimite informații către server.

Un lucru important de menționat, utilizatorii pentru a rămâne conectați la aplicație chiar și după părăsirea site-ului, sunt stocate date esențiale precum id-ul acestora și token-ul generat de către server la logarea pe platformă despre care am prezentat anterior, în zona de stocare numită “localstorage” [29] oferită de fiecare browser în parte. Acest storage reține informații specifice site-urilor web pentru o ușoară viitoare interacțiune.

Înainte de a încheia partea dezvoltării interfeței, vreau să prezint o directivă creată pentru a reproduce slideshow-ul reutilizat atât pe pagina principală, cât si pe cea a fiecărei drumeții, deoarece aceasta a redus considerabil numărul de linii de cod ce ar fi trebui rescrise și a pus în aplicare acest concept foarte utilizat în AngularJs. Pentru a fi mai clar, directivele sunt funcții

JavaScript care manipulează și adaugă comportamente elementelor HTML DOM. Directivele pot fi foarte simpliste sau extrem de complicate, prin urmare, este esențial să fie luate în calcul numeroasele opțiuni și funcții care le manipulează [30].

```
globalModule.directive('interestsSlider', function ($timeout, $window) {
    return {
        restrict: 'E',
        scope: {
            interests: '='
        },
        link: function (scope, elem, attrs) {

            var transportsIcons = {
                Bus: 'Bus',
                Metro: 'Metro',
                Train: 'Train',
                Boat: 'Boat',
                Tram: 'Tram',
                Taxi: 'Taxi',
                Underground: 'Underground',
                Subway: 'Subway',
                Helicopter: 'Helicopter',
                Plane: 'Plane',
                Rent: 'Rent'
            }

            scope.getTransportIcon = function (transport) {
                transport = transport.trim();
                switch (transport) {
                    case transportsIcons.Bus:
                        return "Images/Common/BusIcon.png";
                    case transportsIcons.Metro:
                        return "Images/Common/MetroIcon.png";
                    case transportsIcons.Train:
                        return "Images/Common/TrainIcon.png";
                    case transportsIcons.Boat:
                        return "Images/Common/BoatIcon.png";
                    case transportsIcons.Tram:
                        return "Images/Common/TramIcon.png";
                    case transportsIcons.Taxi:
                        return "Images/Common/TaxiIcon.png";
                    case transportsIcons.Underground:
                        return "Images/Common/UndergroundIcon.png";
                    case transportsIcons.Subway:
                        return "Images/Common/SubwayIcon.png";
                    case transportsIcons.Helicopter:
                        return "Images/Common/HelicopterIcon.png";
                    case transportsIcons.Plane:
                        return "Images/Common/PlaneIcon.png";
                    case transportsIcons.Rent:
                        return "Images/Common/RentIcon.png";
```

```
                }
            }

            scope.currentIndex = 0; // initial indexul este pe prima pozitie

            scope.next = function () {
                scope.currentIndex < scope.interests.length - 1 ?
scope.currentIndex++ : scope.currentIndex = 0;
            };

            scope.prev = function () {
                scope.currentIndex > 0 ? scope.currentIndex-- :
scope.currentIndex = scope.interests.length - 1;
            };

            scope.openWikipediaTab = function (link) {
                $window.open(link, '_blank');
            };

            scope.$watch('currentIndex', function () {
                scope.interests.forEach(function (interest) {
                    interest.visible = false; // toate interesele nu mai
sunt vizibile
                });

                if (scope.interests.length > 0) {
                    scope.interests[scope.currentIndex].visible = true; //
interesul curent devine vizibil
                }
            });

            var timer;
            var sliderFunc = function () {
                timer = $timeout(function () {
                    scope.next();
                    timer = $timeout(sliderFunc, 4000);
                }, 4000);
            };

            sliderFunc();

            scope.$on('$destroy', function () {
                $timeout.cancel(timer); // cand variabila scope este
distrusă, opresc timerul
            });
        },
        templateUrl: 'AppViews/Home/home-suggested-interests-
slidetemplate.html'
    };
});
```

Directiva a fost numită “interestsSlider” și este construită să accepte un atribut numit „interests” prin care poate fi trimisă o listă de interese și afișată în interfață. Apoi a fost declarat obiectul în

care stochez toate denumirile iconițelor pentru transporturile existente, pe care îl voi folosi în cadrul metodei getTransportIcon ce este accesibilă din HTML-ul asociat și care va prelua automat iconița aferentă denumirii trimise ca parametru acestei metode (aici fiind vizibile „case-urile” pentru fiecare iconiță si selectarea imaginii aferente). După care a fost creată variabila prin care țin evidența numărului sugestiei afișată utilizatorului. Metodele „next” și „prev” sunt utilizate pentru a schimba sugestia afișată cu următoarea sau cea precedentă, prin intermediul a două butoane sub forma unor săgeți prezente în slideshow. De menționat că aceste două metode sunt configurate să recunoască dacă utilizatorul a ajuns la ultima sugestie și înaintează, sau la prima sugestie și apasă să afișeze precedenta, astfel să fie resetat contorul fie la începutul listei, fie la finalul acesteia. Următorul aspect este metoda prin care este lansată pagina de Wikipedia corespunzătoare orașului prezent în sugestia propusă. Mai departe, este prezentă o funcționalitate interesantă „$watch” care permite urmărirea în timp real a unui obiect, astfel încât dacă acesta se modifică, va putea lansa o acțiune în urma acestuia automat. În cazul curent, a fost necesară implementarea acestei metode, deoarece sugestiile puteau să fie schimbate manual de utilizator sau automat după patru secunde, așa cum poate fi observat mai jos blocul de cod al funcției „sliderFunc” ce pornește un temporizator care trece la următoarea sugestie din patru în patru secunde. Prin urmare, ar fi fost greu de gestionat cazul în care utilizatorul apasă manual să schimbe sugestia, iar în același timp ar fi fost automat schimbată sugestia de temporizator, și chiar în plus ar fi cauzat probleme ale interfeței, deoarece nu ajungeau sa fie închise la timp sugestiile vizibile, înainte să fie înlocuite. Iar la finalul directivei, mai există încă o metodă ce asigură oprirea temporizatorului când pagina se schimbă sau este înlăturat slideshow-ul, dar și setarea de recunoaștere a locației pentru fișierul HTML utilizat să creeze depanatorul de imagini.
Ultimul pas în care mai trebuia introdusă directiva în HTML, este realizată în felul următor:

```
<interests-slider ng-if="suggestedInterestsLoaded" interests="suggestedInterests" />
```

Acesta este codul ce trebuia inserat în HTML, tag-ul fiind numele directivei (litera mare din numele acesteia trebuie scris cu literă mică și cratimă în fața sa, deoarece aceasta este convenția de interpretare a codului HTML în combinație cu AngularJs). Se poate observa atributul „interests” menționat și anterior căruia i-a fost atribuită lista de sugestii ce urmează sa fie afișată.

Trecând la ultimul aspect al dezvoltării, algoritmul principal pe baza căruia sunt generate sugestiile în funcție de interesele alese de utilizatori este făcut sub forma unei căutări extinse, mai precis

algoritmul unește toate interesele primite și caută prima dată sugestiile ce există doar pe baza intereselor alese. În cazul în care nu există rezultate, algoritmul extinde căutarea pe baza conexiunilor predefinite pentru interese, în sensul în care spre exemplu o locație este asemănătoare cu o altă locație, iar gradul de asemănare este ridicat cu interesele selectate de utilizator, atunci rezultatul este preluat și arătat utilizatorilor. Acesta extinde căutarea cât timp există conexiuni către care poate înainta, asemănător cu parcurgerea printr-un graf.

V. Probleme intâmpinate și tratarea acestora

Ca în orice aplicație, problemele nu ezită să apară. Pe parcursul întregii dezvoltări au apărut multiple probleme, de natură tehnică mai ales ce impuneau fie o refactorizare amănunțită a codului, fie regândirea logicii sau dacă norocul era de partea mea, rezolvări relativ simple ce nu necesitau un efort mai mare.

O problemă comună a fost tratarea excepțiilor. În momentul în care aveam obiecte ce nu erau instanțiate sau nu aveau referință către o adresă de memorie, server-ul arunca excepții. Pentru a rezolva această problemă, un ajutor era oferit de Visual Studio, deoarece fiecare metodă importată de la un anumit framework, specifica și ce fel de excepție ar putea să arunce, astfel nevoia de tratare a excepțiilor era vizibilă, dar pentru alte tipuri de excepții întâmpinate precum cele în care coloanele nu erau mapate corespunzător la baza de date, conversii ce nu puteau fi efectuate, s.a.m.d., ajungeau să fie rezolvate în timpul testărilor. Cu această ocazie, înțelegerea blocului de cod “try-catch” a devenit mai clară și ușor de implementat.

Pe lângă problema excepțiilor, o altă problemă majoră întâlnită a fost cea a imposibilității de a avea pe același request către server, două contexte deschise simultan către baza de date prin Entity Framework. Problema apărea în momentul în care utilizatorul intra pe site și avea nevoie să primească notificările noi, dar în același timp să primească și lista de sugestii. Deși în alte cazuri aceasta nu a fost o problemă, deoarece contextul era automat închis după prelucrarea comenzii date, în acest caz acesta rămânea încă deschis. Soluția care a venit la îndemână a fost utilizarea directă a tehnologiei ADO.NET pe baza căreia este și EF creat, astfel evitam să folosesc obiecte ale EF-ului, ci deschideam manual o conexiune către baza de date și executam un query. Asemănător, a fost nevoie să refactorizez și alte metode care se suprapuneau, desi metodele erau

făcute să aștepte una după cealaltă. Cu acest prilej, am putut să îmi reamintesc și să refolosesc conceptele învățate despre baze de date și comenzile de selectare, inserare și modificare pentru a realiza comenzile ADO.NET.

Tot la nivelul bazei de date, o problemă a apărut în cadrul unui query apelat prin ADO.NET, care nu returna valorile dorite, deși codul era la prima vedere bun, însă un mic aspect reușea să strice întreg rezultatul.

```
private SqlCommand GetNotificationByUserIdQuery(SqlConnection connection,
int userId)
{
    SqlCommand command = connection.CreateCommand();
    string query = @"SELECT N.*,
                            U.Email as 'SenderEmail',
                            T.Name as 'TripName'
                    FROM Notifications N
                    JOIN Users U
                    ON U.Id = N.SenderId
                    JOIN Trips T
                    ON T.Id = N.TripId
                    WHERE N.UserId = @userId
                    ORDER by N.Date DESC";

    command.CommandText = query;
    command.Parameters.AddWithValue("@userId", userId);

    return command;
}
```

Metoda de mai sus conține query-ul problematic, care este asociat unui obiect de tipul SqlCommand, împreună cu parametrul aferent “userId” ce va fi interpretat în cadrul acestuia, iar la final va returna obiectul pe care conexiunea sql îl va utiliza pentru a executa intrucțiunea mai departe. Query-ul selectează toate notificările unui utilizator, interogând tabelele ce dețin Foreign Key-urile asociate tabelei, pentru a prelua informații adiționale precum email-ul utilizatorului și numele drumeției asociate notificării, ca la final, rezultatele să fie ordonate descrescător. Însă problema apăruse în cazul în care notificarea nu era asociată unei drumeții, ci unei cereri de prietenie de exemplu, astfel câmpul drumeției ar fi avut valoarea null, iar selecția nu era construită să accepte și valorile null.

```
LEFT JOIN Trips T
```

Pentru a rezolva cazul menționat, a fost necesară introducerea cuvântului “LEFT” care obligă interogarea să preia atât informații inexistente, cât și existente, indiferent dacă condiția ON T.Id =

N.TripId nu găsește nicio conexiune. Prin urmare, query-ul a ajuns să selecteze toate notificările care erau sau nu destinate drumețiilor, acesta fiind factorul ce cauza problema.

Pe lângă problemele menționate anterior, a existat o problemă a organizării proiectelor existente. Până a pune în practica pattern-urile prezentate anterior, proiectul nu a fost organizat pentru a putea fi scalabil și nici nu era eficient. De fiecare dată când mă foloseam de un repository sau de o clasă din logica de business, cream chiar și de două ori în aceeași metodă două instanțe pentru același obiect, lucru ce impacta atât performanța, cât și memoria alocată. Astfel implementarea pattern-ului Composition Root a redus semnificativ numărul de instanțe create, dar a crescut și performanța codului, iar pattern-urile "Strategy" și "Repository" au ajutat soluția prin crearea unor proiecte adiționale în care a fost mutată logica pusă haotic anterior și care ulterior a devenit ușor de gestionat și modificat, iar aplicația a devenit ușor de scalat.

O problemă ce ținea mai mult de duplicarea codului de prea multe ori, a apărut în momentul în care metodele din clasele controller primeau parametru principal id-ul utilizatorului care a pornit request-ul către server, ce trebuia mereu validat pentru a fi sigur că acesta există în baza de date, deoarece pot apărea cazuri în care un utilizator nu mai există din diferite motive, iar procesul de inserare a unei noi intrări asociate acestui ID va cauza o excepție aruncată de baza de date, deoarece coloana care este Foreign Key către tabela Users pe coloana Id, va returna un mesaj în care precizează că acel ID nu există. Astfel, am decis să rezolv această problemă a reutilizării unor linii de cod pentru validarea utilizatorului, prin crearea unui atribut ce este asociat fiecărui parametru "UserId" care vine prin metodele controller-elor. Un atribut este o etichetă declarativă utilizată pentru a transmite informații în timp de execuție despre comportamentele diferitelor elemente, precum clasele, metodele, structurile, enumeratoarele, ansamblurile, ș.a.m.d. [31].

```
public class ValidateUserAttribute : ValidationAttribute
{
    private readonly UserWorker _userWorker;

    public ValidateUserAttribute() : base()
    {
        var compositionRoot = CompositionRootBackend.Instance;
        _userWorker = compositionRoot.GetImplementation<UserWorker>();
    }

    public override bool IsValid(object value)
    {
        var userId = (value as int?).GetValueOrDefault();
```

```
        if (userId == 0)
        {
            return false;
        }
        var user = _userWorker.GetById(userId);
        if (user == null)
        {
            ErrorMessage = "The account could not be retrieved!";
            return false;
        }

        return true;
    }
}
```

Pentru a crea o clasă atribut este obligatoriu să aibă sufixul denumirii “Attribute” și să moștenească clasa Attribute, dar în cazul acesta clasa ValidationAttribute (care la randul ei moștenește Attribute), deoarece aceasta este destinată validarii elementelor asociate, la fel cum și alte atribute derivate sunt destinate altor operațiuni, iar ele fac parte din librăria “System.ComponentModel. Annotations”. Asadar, constructorul clasei e folosit pentru a pregăti proprietățile sau valorile necesare ce vor fi folosite în interiorul metodei de validare suprascrise numită “IsValid”. În metoda de validare, obiectul primit trebuie convertit la tipul trimis ca parametru, aici fiind int, deoarece este vorba despre id-ul utilizatorului, iar în cazul în care parametrul nu a putut fi convertit și primește valoarea default (adică 0), atunci validarea a eșuat, altfel este continuat blocul de cod prin interogarea bazei de date și obținerea utilizatorului, care dacă nu există este returnat un mesaj de eroare ce va fi afișat în interfață.

Însă, pentru a putea interpreta mesajul de eroare primit, a fost nevoie de o configurație suplimentară în fișierul “Startup” în metoda ConfigureServices (metodă de bază pentru setarea configuratiilor specifice oricărei aplicații web, dar care este opțională).

```
services.AddMvc(options =>
{
    options.EnableEndpointRouting = false;
}).ConfigureApiBehaviorOptions(opt =>
{
    opt.InvalidModelStateResponseFactory = context =>
    {
        var modelState = context.ModelState.Values;
        var firstError =
modelState.FirstOrDefault()?.Errors?.FirstOrDefault()?.ErrorMessage;

        return new BadRequestObjectResult(firstError);
    };
});
```

Inainte de toate, obiectul “services” reprezintă parametrul metodei și în același timp, punctul de acces către serviciul web pentru a-l putea configura. Metoda apelată AddMvc configurează server-ul să poată utiliza diferite setări specifice serviciilor MVC, astfel aplicația să permită injectarea dependințelor, dar mai ales să poată înțelege și interpreta request-urile primite. Totuși, ce ne interesează in codul sursă, este a doua metodă numită “ConfigureApiBehaviorOptions” asupra căreia este setată proprietatea InvalidModelStateResponseFactory ce reprezintă răspunsul pe care îl returnează server-ul după prelucrarea unui request, în cazul în care valorile primite ca parametru nu sunt valide. Deoarece eroarea transmisă prin atribut este asociată răspunsului, aceasta poate fi extrasă exact din prima eroare a obiectului “modelState”, ca mai apoi să fie returnat mai departe către client codul de eroare 400 împreună cu mesajul erorii.

Pe partea de interfață problemele au fost în mare parte la crearea și stilizarea paginilor html, deoarece alegerea variabilelor corespunzătoare în CSS este relativ dificilă știind dependințele părinte-copil existente în HTML și felul în care acestea pot fi alterate de multe ori fară să ne dăm seama chiar de un părinte la care nu ne așteptam al unui tag. Astfel au existat probleme de aliniere în pagină, de dimensiuni, de font, de poziționare și multe alte probleme comune în rândul dezvoltatorilor de aplicații web pe această latură. Aceste probleme au fost rezolvate luând la rand de la cel mai “vârstnic” părinte, până la copilul tag în cauză de la care pornea problema, deoarece trebuia verificat pas cu pas dacă poate cineva să impacteze problema de stilizare. Pentru a putea face asta, browserele în mod normal pun la dispoziție o consolă de inspectare pentru a parcurge sau a opri codul creat și a-l verifica. Personal dezvoltarea a fost făcută folosind Google Chrome, dar pentru teste adiționale a fost folosit și Microsoft Edge.

Însă, la nivel de AngularJs, au apărut probleme în momentul implementării funcționalității de autorizare și autentificare folosind token. Deși am vorbit mai devreme despre felul în care acesta este implementat să fie folosit și să ajungă la client, problema a apărut atunci când token-ul trebuia atașat fiecărui request trimis de client, iar în cazul în care nu mai era valid, trebuia deconectat utilizatorul și retrimis la pagina de logare pentru a se loga și a obține un token nou. Inițial a fost asociat fiecărui request manual token-ul, dar nu era o practică tocmai bună, iar dacă acesta nu mai era valid, nu exista o metodă să fie înlocuit, deoarece nu exista un mecanism să înștiințeze utilizatorul că acel token a expirat sau nu mai e de actualitate.

Pentru a rezolva acest aspect, am implementat un interceptor ce preia orice request înainte să fie trimis la server sau înainte să ajungă la client, și prelucrat pentru a vedea dacă token-ul există si este valid.

```
globalModule.factory('authInterceptor', [
    '$rootScope', '$q', '$location', '$window', '$timeout', '$localStorage',
    function ($rootScope, $q, $location, $window, $timeout, $localStorage) {
        return {
            request: function (config) {
                if ($localStorage.AccessToken !== undefined &&
$localStorage.TPUserId !== undefined) {
                    config.headers.Authorization = "Bearer " +
$localStorage.AccessToken;
                }
                return config || $q.when(config);
            },
            response: function (response) {
                if (response) {
                    if (response.config.url.toLowerCase() ===
"api/landingpage/login") {
                        $localStorage.AccessToken = response.data.token;
                        $localStorage.TPUserId = response.data.userId;
                    }
                }
                return response;
            },
            responseError: function (response) {
                if (response) {
                    //unauthorized || forbidden - sesiunea a expirat
                    if (response.status === httpStatusCodeEnum.Unauthorized
|| response.status === httpStatusCodeEnum.Forbidden) {
                        $window.location.href = '/welcome';
                    }
                }

                return $q.reject(response);
            }
        }
    }
]);
```

Interceptorul a fost creat ca un Factory în cadrul modulului global declarat în AngularJs pentru controlarea tuturor obiectelor din acest framework și create sub acest modul. Factory reprezintă o funcție simplă care ne permite să adăugăm o logică unui obiect creat și să returnăm obiectul creat. În cadrul funcției este returnat formatul unui request ce vine sau pleacă de la client. Astfel în prima parte unde pleacă request-ul, este verificat localstorage-ul dacă conține atât token-ul de acces, cât și id-ul utilizatorului pentru a asocia header-ului token-ul ce va fi recepționat de server pentru

autorizare. În acest caz, dacă nu este trimis niciun token către serviciul web, acesta va transmite înapoi către client acest „responseError” ce este în partea de jos a codului, fiind zona prin care sunt transmise excepțiile și codurile status ale protocolului HTTP negative precum 400, 401, 500, ș.a.m.d.. În acest caz, utilizatorul trebuie să fie redirecționat dacă a fost semnalat accesul interzis, către pagina de logare. Iar obiectul rămas numit „response”, primește orice protocol pozitiv sau cu succes, astfel aici a fost interpretat răspunsul de la server pe metoda de logare ce oferă token-ul și id-ul utilizatorului, acestea fiind stocate în localstorage.
Prin urmare, mecanismul implementat a optimizat traficul și a automatizat rezolvarea problemelor ce pot apărea la nivel de autorizare între partea de front-end și cea de back-end.

Mici alte probleme au mai apărut și în momentul configurărilor pentru comunicarea dintre client și server sau de natură tehnică ce nu puteau fi înțelese în mod direct, dar acestea au fost rezolvate pe baza ajutoarelor de pe diferite site-uri, în care alte persoane au întâmpinat aceleași probleme sau apropiate.

VI. Surse de inspirație și concluzii

Aplicația web a putut fi realizată în urma cunoștințelor acumulate pe parcursul anilor de studenție, cât și în cadrul firmelor în care am fost angajat în timpul acesta. Importanța unor baze ale programării și cunoașterea tehnicilor și tehnologiilor necesare, au fost factori esențiali în stadiul incipient, fără de care nu ar fi putut exista o structură a proiectului, o funcționalitate, sau chiar o idee care să poată fi implementată.

Datorită anilor de studenție, am reușit să aprofundez și să aplic noțiuni importante ale programării orientate pe obiecte precum clasele, obiectele, moștenirea, încapsularea, polimorfismul, s.a.m.d, care bineînțeles stau la baza oricărei aplicații bune din acest domeniu și fară de care acestea nu ar putea evolua. De asemenea, aprofundarea structurilor de date mi-a venit ca un ajutor încă din start, mai ales în condițiile în care aplicația utilizează într-o cantiate mare liste, dar și vectori pentru prelucrarea unor informații multiple și stocarea acestora.

Cu toate acestea, cunoașterea pe larg a limbajului Javascript și a modalității de dezvoltare a paginilor web, a fost bine pusă la punct în cadrul cursului numit Tehnici Web, datorită căruia am

putut înțelege sintaxa și funcțiile importante ale Javascript-ului, dar mai ales cum poate fi legat acesta de paginile HTML și CSS, fără de care nu ar fi fost posibilă realizarea interfeței.

Și nu în ultimul rand, meritele pentru cunoștințele de bază acumulate în ceea ce privesc limbajul C# și bazele de date relaționale, sunt destinate cursurilor pentru Dezvoltarea Aplicațiilor Web și Baze de Date, alaturi de care am putut învăța cum poate fi folosit limbajul C# pentru crearea unor aplicații web, cum poate fi conectată o aplicație la o bază de date și mai ales, cum poate fi utilizată o bază de date și sintaxa pe baza căreia aceasta reacționează.

Toate acestea, plus alte informații suplimentare învățate și studiate pe parcursul facultății, au format baza prin care am reușit să realizez aplicația.

Totuși, pentru a putea dezvolta aplicația la un nivel scalabil și profesionist, am apelat la cunoștințele acumulate ca angajat, dar folosind și tehnologii studiate pe cont propriu. Pe perioada în care am fost stagiar la o firmă românească, am înțeles cum funcționează într-un proiect complex o bază de date, ce înseamnă să creezi un index, o cheie primară compusă, un trigger, și multe alte informații importante pentru aceasta. Totodată, am descoperit din ce este compusă o aplicație, cum sunt diferențiate părțile de front-end și back-end, dar mai ales cum pot să îmi organizez proiectul în funcție de un tool planificator. Poate una dintre cele mai importante lucruri învățate, a fost cum să utilizez platforma de versionare GitHub [32] în care pot stoca proiectul și pot avea istoricul acestuia, fără de care personal am concluzionat că nu poate exista o aplicație la un nivel profesionist, fără o platformă de acest gen pe care să fie dezvoltat.

De asemenea, o altă firmă în care am mai lucrat m-a învățat ce înseamnă să lucrezi cu framework-ul pe care l-am folosit și eu, anume .NET Core și mai ales cu tehnologiile asociate acestuia. În prezent pot spune că aceasta a devenit tehnologia principală pe baza căruia îmi propun să dezvolt aplicații și ulterior. Acest framework permite crearea aplicațiilor sau serviciilor web într-o manieră foarte ușoară, mai ales în momentul cunoașterii sale, și care oferă acces la o multitudine de tehnologii ajutătoare pentru a obține cea mai bună variantă din aplicația dezvoltată.

Deși am descoperit singur AngularJs pentru front-end, am avut norocul ca ulterior să îl aprofundez și la locul de muncă, unde am înțeles și mai bine conceptele de controller, de modul, de directivă, fără de care nu aș fi putut obține conexiunea dorită între cele două părți ale aplicației.

Aceste cunoștinte au permis realizarea aplicației, punând în schemă noțiuni și practici utilizate și la nivel de angajat într-o firmă, motiv pentru care a reprezentat un prilej foarte bun de a experimenta și a utiliza și mai mult toate cele învățate.

VII. Referințe

[1] Wikipedia, the free encyclopedia [2018], https://en.wikipedia.org/wiki/Front_end_and_back_end

[2] Kanchan Naik, Design Patterns in C# .NET [2020], https://www.c-sharpcorner.com/UploadFile/bd5be5/design-patterns-in-net/

[3] Rick Anderson, Kirk Larkin, Mike Wasson, Create a web API with ASP.NET Core [2018], https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-2.2&tabs=visual-studio

[4] Rick Anderson, Get started with ASP.NET Core MVC [2018], https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-2.2&tabs=visual-studio

[5] Wikipedia, the free encyclopedia [2018], https://en.wikipedia.org/wiki/.NET_Core

[6] W3Schools, AngularJS Tutorial [2019], https://www.w3schools.com/angular/default.asp

[7] Rick Anderson, Ryan Nowak, Introduction to Razor Pages in ASP.NET Core [2018], https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-2.2&tabs=visual-studio

[8] Tutorialspoint, MS SQL Server Tutorial [2018], https://www.tutorialspoint.com/ms_sql_server/index.htm

[9] Shadman Kudchikar, Repository Pattern C# [2020], https://codewithshadman.com/repository-pattern-csharp/

[10] Tutorialspoint, Entity Framework Tutorial [2018], https://www.tutorialspoint.com/entity_framework/index.htm

[11] Docs Microsoft, ADO.NET Code examples [2019], https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-code-examples

[12] Wikipedia, the free encyclopedia [2019], https://ro.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[13] Wikipedia, the free encyclopedia [2020], https://en.wikipedia.org/wiki/Swagger_(software)

[14] Tutorialsteacher, Middleware [2019], https://www.tutorialsteacher.com/core/aspnet-core-middleware

[15] Wikipedia, the free encyclopedia [2018], https://en.wikipedia.org/wiki/Entity_Framework

[16] Brian Pohl, Schema Comparisons using Visual Studio SQL Data Tools [2019], https://www.decisivedata.com/blog/schema-comparisons-using-visual-studio-sql-data-tools

[17] Github Ninject, Ninject Framework [2020], https://github.com/ninject/Ninject

[18] Ondrej Balas, How to refactor for Dependency Injection, Part 2: Composition Root [2020], https://visualstudiomagazine.com/articles/2014/06/01/how-to-refactor-for-dependency-injection.aspx

[19] Bootstrap, World's most popular framework for building responsive pages [2018], https://getbootstrap.com/docs/4.0/getting-started/introduction/

[20] Mads Kristensen, Bundler & Minifier [2019], https://marketplace.visualstudio.com/items?itemName=MadsKristensen.BundlerMinifier

[21] Mikhail Mitra, 10 Most important Interaction Design Principles [2020], https://www.mantralabsglobal.com/blog/10-basic-principles-of-interaction-design/

[22] Stackoverflow, Question about AddTransient, AddScoped and AddSingleton Services Differences [2020], https://stackoverflow.com/questions/38138100/addtransient-addscoped-and-addsingleton-services-differences

[23] Wikipedia, the free encyclopedia [2020], https://en.wikipedia.org/wiki/Double-checked_locking

[24] Marinko Spasojevic, C# Design Patterns – Strategy Design Pattern [2020], https://code-maze.com/strategy/

[25] Wikipedia, the free encyclopedia [2019], https://en.wikipedia.org/wiki/Object-relational_mapping

[26] Anton Kharenko, Monolithic vs Microservices Arhitecture [2019], https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59

[27] Tutorialspoint, What is the use of the Configure() method of startup class in C# Asp.net Core? [2020], https://www.tutorialspoint.com/what-is-the-use-of-the-configure-method-of-startup-class-in-chash-asp-net-core

[28] Rick Anderson, Scott Addie, Static files in ASP.NET Core [2018], https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files?view=aspnetcore-2.2

[29] Mozilla Docs, Localstorage [2019], https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

[30] Eric W.Greene, AngularJS Tutorial: Demystifying Custom Directives [2020], https://www.toptal.com/angular-js/angular-js-demystifying-directives

[31] Tutorialspoint, Attributes [2021], https://www.tutorialspoint.com/csharp/csharp_attributes.htm

[32] Github Guides, Hello World [2018], https://guides.github.com/activities/hello-world/