

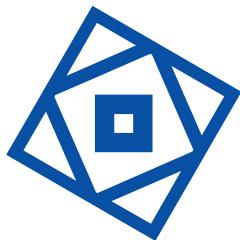
Universitatea
Transilvania
din Brașov
FACULTATEA DE MATEMATICĂ
ȘI INFORMATICĂ

LUCRARE DE LICENȚĂ

Conducător științific:
Profesor dr. Nănău Corina-Stefania

Absolvent:
Trofin George Ionuț

BRAŞOV, 2024



Universitatea
Transilvania
din Brașov
FACULTATEA DE MATEMATICĂ
ȘI INFORMATICĂ

LUCRARE DE LICENȚĂ

Relocation and booking services - Aplicație web
pentru servicii de închiriere și mutare în alt oraș

Absolvent: Trofin George Ionuț
Coordonator: Lector dr. Nănău Corina-Ştefania

Brasov Iunie
2024

Cuprins

1 Introducere	5
1.1 Prezentarea acestei lucrări	5
1.2 Motivul și scopul acestei temei	6
1.3 Platforme web similare	6
1.4 Structura acestei lucrări.....	7
2 Dezvoltarea proiectului	8
2.1 Medii de dezvoltare și sisteme de gestionare a bazelor de date	8
2.1.1 Visual Studio IDE	8
2.1.2 Microsoft SQL Server.....	10
2.2 Implementare server-side.....	11
2.2.1 NuGet	11
2.2.2 Entity Framework Core.....	13
2.2.3 Rest API.....	17
2.2.4 Rutare Aplicație.....	20
2.2.5 Controller	21
2.3 Implementare client-side.....	22
2.3.1 Views.....	22
2.3.2 Legarea datelor platformei	24
2.3.3 Bootstrap 5.3.....	25
2.3.4 Javascript.....	27

3 Considerații pentru implementare	28
3.1 Structura aplicației web.....	28
3.1.1 Arhitectura server-side.....	29
3.1.2 Arhitectura client-side.....	33
3.1.3 Baza de date	34
3.2 Specificații funcționale	36
3.2.1 Actori	36
3.2.2 Implementarea în .NET Core	37
3.2.3 Implementarea în Razor Views	46
3.2.4 Securitate	48
4 Ghid de utilizare al aplicației	51
4.1 Prezentare	51
4.2 Partea de autentificare.....	51
4.3 Pagina principală post-autentificare.....	55
4.4 Pagina de profil a utilizatorului.....	56
4.5 Paginile cu oferte pentru utilizator.....	58
4.6 Pagina cu lista de email-uri.....	59
4.7 Pagina cu vizualizarea email-ului.....	60
4.8 Pagina cu compunerea unui email.....	61
4.9 Pagina cu ofertele poste.....	62
5 Concluzii și posibilități de dezvoltare	64
5.1 Concluzii	64
5.2 Posibilități de extindere	65

Capitolul 1

Introducere

1.1 Prezentarea acestei lucrări

De-alungul vieții o să trebuiască să te muți dintr-un oraș în altul din tot felul de motive. Fie că prețul chiriei casei tale a devenit prea scump, sau îți-ai pierdut locul de muncă dintr-o dată, sau , pur și simplu, vrei să te muți în alt oraș din România care îți place.

Această aplicație web destinată relocării poate ajuta enorm de mult pe oricine își dorește să-și asigure toate condițiile necesare mutării în altă reședință din alt oraș. Companii din tot felul de industrii (de cărat mobilă, de transport persoane, de închiriat apartamente sau vehicule și aşa mai departe) folosesc platforma ca să poată să-și posteze ofertele companiei lor tuturor celor care au nevoie de aşa ceva, putând căuta ofertele după orașul destinație în care respectivul dorește să se mute. Dacă o familie întreagă dorește să se mute, există un formular special de completat pentru a menționa acest lucru și , prin email, utilizatorul din acea industrie va primi un mail special că tu dorești să te muți cu familia ta întreagă. De asemenea, e posibil să cauți și job pentru tine să lucrezi în noul oraș, cât și o școală pentru copiii familiei, fie că sunt la preuniversitară sau trebuie să dea la facultate. Poți contacta orice utilizator de la companii oricând, fiindcă în ofertele lor mereu va fi vizibil numărul de telefon și, dacă nu vrei să treci prin tot procesul de aplicare și de trimis mail-uri, poți să-i suni oricând.

Mereu vei putea să comunici cu corporațiile care doresc să-și lase ofertele pe această platformă web la secțiunea lor respectivă din bara de căutare, și mereu vei putea să sortezi toate ofertele lor după oraș și să discuți cu oamenii lor prin mail-ul din contul lor, iar ei vor putea să te contacteze la rândul lor prin email și telefon, deoarece când le trimiti tu un email sau aplici la o ofertă ce aparține lor, îți vor putea vedea numele, mail-ul și telefonul tău. Așa că nu-ți fă griji, mereu va fi posibil să vorbești și tu cu ei, și ei cu tine.

Datele tale vor fi într-o bază de date securizată, iar parola ta criptată după un protocol de criptare foarte greu de spart, în contextul în care dorești să-ți faci un cont pe această platformă web. Dacă nu, poți să te loghezi cu contul de Google, nefiind o cale a se putea găsi parola ta vreodată din cauza noastră. Și dacă ai o imagine de profil super, sau chiar un gif, va apărea ca poza ta de profil.

1.2 Motivul și scopul acestei temei

Nu este atât de ușor să găsești companiile care se ocupă de serviciile necesare unei reallocări. Nici clienților, nici companiilor. Așa că le-ar fi mult mai simplu să aibă un intermedier între ele, o platformă web, în care companiile își trimit oameni de-al lor să posteze oferte și clienții pot veni oricând să le vadă și să afle de firmele respective. Scopul este să faci mult mai simplă comunicarea între client și companie pentru a nu pierde timp și nervi căutând pe net diverse firme sigure care oferă servicii de calitate, și nici companiile să caute clienți în tot felul de locuri care mai de care. Este mult mai simplu să ai un intermedier de încredere care verifică dacă ambele tabere sunt de încredere.

Folosind aplicația aceasta, toată lumea se poate bucura de calitatea software-urui oferit, care ușurează destul de mult procesul de a te găsi cu oamenii de care ai nevoie pentru a putea să-ți faci treaba rapid, sigur și bine.

1.3 Platforme web similare

Un software web cât de cât similar ar fi platforma "[Konnecting](#)". El face posibile găsirea de meserii în țările unde ar dori cineva să se mute. Pe site-ul lor poți să-ți aleagă pe ce industrie ai vrea să îți cauți de muncă și să aleagă o țară anume. După poți să optezi și la ce școli sau universități ar fi valabile în zonele respective, ca mai apoi să îți cauți o casă. Dar, aplicația web de reallocare și închirieri făcută de mine, îți permite să te muti în orice oraș din România dorești, contactând firme, în principal, din România și poți să optezi la universități și școli care, sunt pur specific românești (în mare parte). Companiile care oferă chirii sunt de încredere și chiar sunt și firme care îți închiriază vehicule la prețuri acceptabile, ceea ce "[Konnecting](#)"-ul nu oferă, din păcate.

În Relocation and booking services, toate companiile își trimit un angajat de-al lor să-și facă un cont la noi, fie cu parolă (normal) sau fără (logându-se prin Google). Nu se iau date de pe alte site-uri direct, ci fiecare își introduce ofertele lor manual cu toate detaliile. Acuma, atât "[Konnecting](#)" cât și Relocation and booking services permit legătura dintre clienți și corporații prin email și telefon, clienții găsind detaliiile acestea prin ofertele posteate, iar angajații firmelor și școlilor (sau universităților) printr-un email trimis automat la finalul procesului de aplicare la serviciul cutare.

Față de acest concurrent formidabil, totuși, Relocation and booking services este singura platformă web din România care îți permite să obții servicii de mutare, angajare, de găsit școli și universități, de închiriat mașini (și multe altele) și de transport. Nu avem concurență în țară, de aceea, firmele ne folosesc software-ul ca să își găsească clientela, iar oamenii în nevoie, soluția.

1.4 Structura acestei lucrări

Scopul primului capitol este, în mare parte, de a furniza o prezentare concisă a contextului și obiectivelor aplicației, urmând ca funcționalitățile ei și detaliile de implementare complexe să fie aprofundate în capitolele ce urmează. Capitolul acesta a fost compus numai din părțile esențiale pentru o scurtă înțelegere logică a lucrării. Se prezintă avantajele și motivul începerii acestui proiect, alături de acest rezumat al capitolelor următoare.

Obiectivul esențial al celui de-al doilea capitol este în prezentarea aspectelor tehnice ale aplicației (baza de date, back-end, front-end), inclusiv descrierea tehnologiilor utilizate de-a lungul timpului în cadrul proiectului acesta. S-au utilizat tot felul de facilități atât de la Microsoft, cât și din alte surse pentru a putea crea acest proiect reușit. BCrypto și logarea la Google au fost esențiale în realizarea acestei aplicații, pentru a putea da siguranță necesară și ușurință folosirii acestei platforme web.

Capitolul trei cuprinde detalii referitoare la implementare și explică specificațiile funcționale ale aplicației. Cum a fost gândită partea din spate a proiectului, cât de utilă și eficientă este ea și cum ar ajuta design pattern-urile folosite pentru viitoare echipe care ar putea prelua acest proiect. Cum a fost concepută partea de front-end, vizibilă utilizatorilor, cât de prietenoasă a fost făcută pentru ei, cum a fost structurat codul în Razor Views alături de javascript, și cum arată baza de date în sine, cum se leagă tabelele între ele, cât de sigură este și în ce mod sunt stocate datele acolo, atât cele criptate și necriptate. De asemenea, și ce metode și atribută ce vin de la Microsoft s-au folosit pentru a securiza conexiunile între paginile web, de a automatiza crearea, updatarea, adăugarea și ștergerea din baza de date și tot așa.

În capitolul patru, denumit ghid de utilizare, se oferă detalii despre funcționalitățile aplicației și se prezintă instrucțiuni de utilizare. Se va parcurge întreaga aplicație în toate paginile ei pentru a putea vedea cum arată fiecare pagină respectivă și ce facilități sunt disponibile. Ele vor fi explicate pentru utilizatorul obișnuit, acest capitol fiind destinat acestuia. Vor fi poze valabile în acest document pentru a vedea clar și limpede tot ceea ce este necesar și util.

Capitolul final conține concluziile finale, oferind un rezumat și punând accent pe posibilitățile de extindere ce s-ar putea realiza în viitor. Se menționează scurt detaliile esențiale din fiecare capitol care, îmbinate aici, să pună într-o lumină bună lucrarea aceasta și, după, se scrie ce ar putea să se adauge de alți programatori. Totul e scris pe scurt și la obiect ca să nu se piardă timpul și, dacă o să mai fie cineva sau, poate, altă echipă care va citi această documentație, să știe bine și ce s-a folosit și cum, dar și ce idei de implementare ar putea încerca să codeze, dacă le permite timpul, bugetul și ce alte detalii ar mai fi în acele contexte.

Capitolul 2

Dezvoltarea proiectului

2.1 Medii de dezvoltare si sisteme de gestionare a bazelor de date

2.1.1 Visual Studio IDE

Visual Studio este un mediu de dezvoltare integrat, capabil să citească diverse limbi de programare , deobicei, de tipul „strong typed”, adică care au un tip de dată deja stabilit de programator când își scrie codul,dar permite și limbi de tipul “weak type” (javascript, PHP, etc) pentru o mai bună rezolvare a proiectelor, după contextul dat. Așadar, este un IDE¹ creat și îmbunătățit încontinuu de Microsoft.

Datorită posibilității de a avea limbi de tipul "strong type", se pot evita tot felul de erori la compilare, care pot să ducă în erori logice la rulare. Iar erorile logice sunt o bătaie mare de cap de rezolvat, mai ales într-un proiect de dimensiuni mari. Această facilitate a acestui IDE ușurează și citirea codului pentru developerii care sunt alocați pe proiecte și, mulțumită acestei oportunități, pot să scrie cod calitativ, performant,scurt și ușor de înțeles pentru orice alt programator care ar mai veni după el. Pe când la limbajele de tipul "weak type", compilatorul decide tipul de date al variabilelor cutare. "[Strong type and weak type](#)" sunt esențiale în orice mediu de dezvoltare integrat.

Acesta oferă un set de instrumente puternice și funcționalități pentru dezvoltarea de aplicații software, printre care și web, în diferite limbi de programare, cum ar fi C#, C++, C, Python, Angular, JavaScript și altele. Pot să fie combinate în cadrul unor framework-uri ce pot fi folosite când vrei să creezi un proiect nou.

Cu ajutorul Visual Studio, dezvoltatorii pot construi, testa, depana și implementa aplicații pentru diverse platforme, precum Windows (toate framework-urile de .NET, pot lucra pe windows, un exemplu fiind Windows Presentation Foundation (WPF)), web (un exemplu ar fi ASP. NET Core pentru web development), cloud și telefoane (de exemplu, .NET MAUI targetează Android și iOS drept sisteme de operare mobile pentru a integra aplicațiile cand se face mobile development).

Acet mediu de dezvoltare integrat (IDE) oferă un editor de cod foarte avansat, ce are funcționalități precum evidențierea sintaxei, completarea automată a propriului tău cod (dacă e scris corect), refactorizarea, care de multe ori te ajută, și navigarea relativ, ușoară prin proiect.

Prin Intellisense se fac toate sugestiile și posibilele corectări de erori sau mici nepotriviri. El e un ajutor de completare de cod ce te ajută cu listarea tuturor metodelor și parametrilor dintr-un obiect sau clasă statică, să obții informații despre despre elementul respectiv (indiferent ce este el în cod), la fel și cu metodele și atributele lui, cât și despre parametrii metodelor. Este capabil, ca orice ajutor de completări, să-ți ofere completarea codului unde ai tu nevoie, și, apăsând pe tasta Tab, să o facă.

[Intellisense](#) face comparații de tipul camel case cu textul incomplet, fiind suficient să scrii chiar și o singură literă, ca mai apoi, apăsând pe tasta Space, să vezi toate opțiunile disponibile (metode, proprietăți, atrbute) și selectezi una cu tasta Enter.

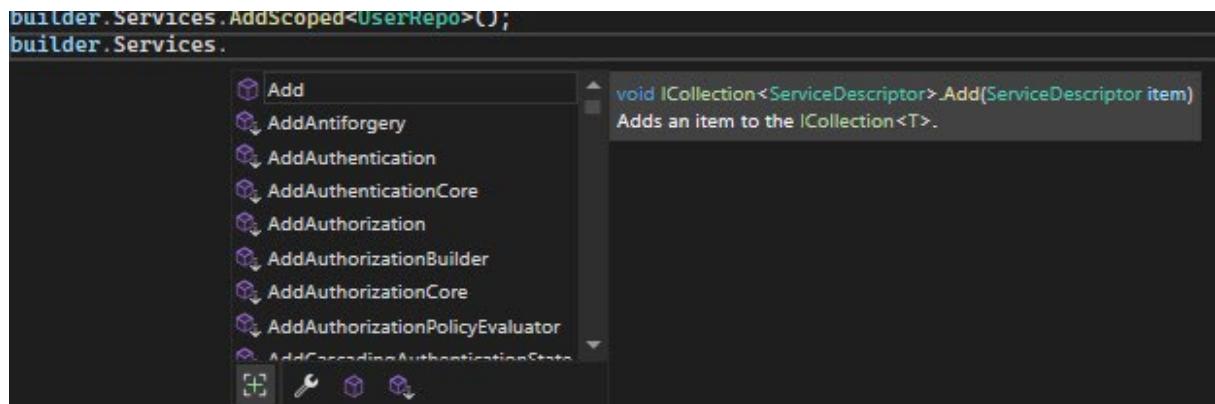


Figura 1: Exemplu Intellisense listă opțiuni cu informații scurte



Figura 2: Exemplu de sugestii de completare a codului cu Intellisense

Un aspect important al environment-ului Visual Studio este faptul că are un suport extensibil. Dezvoltatorii pot adăuga extensii și plugin-uri pentru a îmbunătăți funcționalitățile IDE-ului sau pentru a integra alte instrumente și servicii, adaptând astfel mediul de dezvoltare la nevoile lor specifice. Avantajul aici este că poți să îți creezi propria extensie și să fie posibil să o introduci în orice proiect din visual studio pentru ce ai avea tu nevoie

2.1.2 Microsoft SQL Server

[Microsoft SQL Server](#) este unul dintre cele mai populare sisteme de gestionare a unor baze date, folosindu-se de Microsoft SQL Server Management Studio(SSMS) pentru a se putea configura și administra componentele din sistemul pentru gestionarea bazelor de date rapid și eficient. Prin sarcini poți să creezi și să modifici baze de date, să le creezi tabele, membri ai tabeliei și să le legi cum dorești. Tot prin unealta de Studio (SSMS) poți să scrii "query-uri" ca să aplici task-urile și modificările în baza de date, deobicei tabelelor lor, să configurezi permisiunile utilizatorilor. Iar prin SQL Server Integration Services (SSIS), poți să ai datele tale importate din alte aplicații, transformate în limbaj SQL și , după ce le așezi și modifici după caz, poți să le importi înapoi la aplicația care ti le-a trimis, transformându-le înapoi în tipul de date acceptat de aplicație.

SQL Server este compatibil cu o gama enormă de limbaje de programare (C#, Java, C++, C Python,etc) și funcționează atât pe sistemele de operare Windows, cât și cele de Linux, și e capabil să fie pus atât pe web cât și pe cloud. Deci, el funcționează cu destul de multe framework-uri care conțin limbajele de programare acceptate și le poate oferi tot felul de facilități, depinzând de varianta pe care fiecare poate să o aibă.

Varianta Enterprise oferă cea mai înaltă securitate posibilă, opțiuni de analitica avansată și chiar deblochează serviciul numit SQL Server Machine Learning Service și poți executa scripturi în python și R, sau chiar să îți lansezi modelele de machine learning înăuntrul bazei tale de date. Scriptul va rula unde se află datele date modelului, eliminând astfel nevoia de a transfera datele altui server.

Restul variantelor nu sunt așa puternice, dar sunt, în mare parte, gratis. Variantele Standard, Web și Developer sunt folosite pentru proiecte. Standard-ul e pentru aplicații de dimensiuni maxim medii, pe când cea web e pentru gazde web. Ultima e pentru demonstrații și teste (poți chiar să faci o licență cu ea)

Față de alte baze de date precum MySql, PostgreSQL și Oracle cu depozitarea eficientă, schimbarea și administrarea datelor relationale. Developerii ca să acceseze bazele de date de tipul SQL Server, deobicei se folosesc de Transact-SQL (T-SQL), care este o extensie a standardului SQL.

Un avantaj mare este că variantele de Developer, Standard și Web au multe opțiuni de business, care, în alte baze de date ar fi fost necesar să ai versiunea Enterprise ca să beneficiezi de ele într-un proiect comercial.

Dar cel mai mare dintre avantaje ar fi că ,fiindcă aparține de Microsoft, poate să fie integrată în Microsoft Cloud, Azure Sql Database și chiar mașina virtuală Azure. Toate acestea permit mutarea bazei de date în cloud dacă, din diverse motive, se dorește acest lucru, fie că devine foarte greu de manageriat sau e pur și simplu, prea complexă și greu de administrat.

2.2 Implementare server-side

2.2.1 NuGet

NuGet este un sistem dezvoltat de Microsoft pentru gestionarea și distribuția pachetelor de cod reutilizabil în ecosistemul .NET. Este folosit pentru a facilita instalarea, actualizarea și gestionarea bibliotecilor și resurselor în cadrul proiectelor .NET. Aceste pachete

Prin intermediul NuGet, dezvoltatorii pot adăuga cu lejeritate tot felul de pachete externe în lucrările lor .NET. Aceștia pot căuta și instala ce au nevoie din Visual Studio sau pot utiliza linia de comandă dacă ei doresc.

Unul dintre avantajele majore ale utilizării pachetelor NuGet este posibilitatea de a realiza actualizări simple ale bibliotecilor și resurselor. Atunci când este disponibilă o nouă versiune a unui pachet, dezvoltatorii pot actualiza pachetul în proiectul lor cu câteva click-uri sau cu câteva comenzi în linia de comandă.[6]

Aceste pachete în sine sunt bibliotecile scrise de alți programatori și, deobicei, sunt scrise în proiecte de tipul dll, care conțin cod compilat. Acuma, un pachet specific NuGet (adică special pentru proiectele de tipul .NET sau .NET Core) este un fișier arhivă (ZIP) cu extensia .nupkg, care conține codul, alte fișiere necesare și un manifest ce are versiunea pachetului (ce versiune ai descărcat de la cei care tot actualizează biblioteca anume). În mod normal, creatorii acestor pachete și le publică pe gazde fie private, fie publice, iar cei ce au nevoie de ele le obțin din gazde potrivite pentru ei și cheamă anumite facilități din tot pachetul cutare, de restul detaliilor ocupându-se NuGet în sine.

Drept gazdă publică, această unealtă virtuală este un repozitoriu central pentru peste 100000 de pachete unice pe nuget.org și sunt implementate neconveniente de dezvoltatori de .NET zilnic, deci, e destul de popular în zilele noastre, fiind o unealtă sigură de folosit și pe termen lung.

Poate fi și o gazdă privată, dacă îți încarci proiectele pe cloud (de exemplu pe Azure DevOps), o rețea privată (cum mai au corporațiile și fabricile) sau chiar în fișierele tale locale din dispozitivul tău, astfel, e posibil să se facă o selecție asupra persoanelor ce pot avea acces la pachetul sau pachetele de pe acea gazdă privată și poți chiar selecta și din care gazdă anume se poate extrage pachetul anume sau o funcționalitate din el.

Indiferent de tipul respectiv al gazdei alese, ea face conexiunea dintre dezvoltatorul pachetului și cel ce se va folosi de acel pachet dat, și, mulțumită acestei chestiuni, și API-ul pachetului va putea fi accesat când e folosit de consumatorul lui.

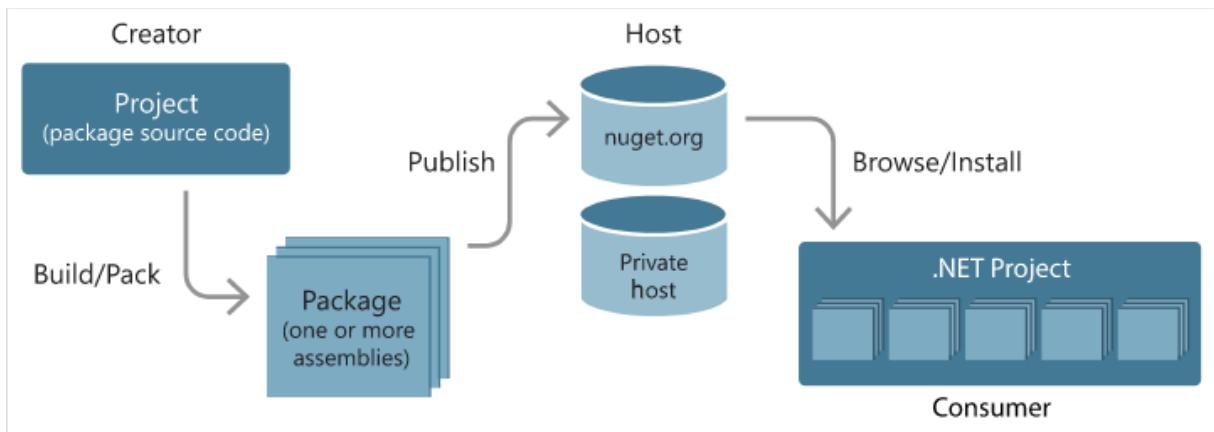


Figura 3: fluxul pachetelor între creatori,gazde, și utilizatorii lor

Ca notă de referință, dezvoltatorii pachetelor pot targeta fie un singur framework sau pot ținti către o gamă mai largă de opțiuni. În general , pentru a se atinge apogeul compatibilității pachetului, se targetează framework-ul .NET Standard, care poate fi "consumat" atât de proiectele de tipul .NET, cât și de cele de tipul .NET Core. Este cea mai bună variantă posibilă pentru ambele părți (creatori și consumatori) ca un singur pachet să meargă pentru toate proiectele, pachetul având un singur assembly.

Cei care au nevoie de API-uri înfăra framework-ului .Net Standard, sunt obligați să conceapă assembly-uri separate pentru fiecare framework targetat pe care vor să-și facă pachetul posibil de folosit și să creeze un pachet, ce conține toate assembly-urile acelor framework-uri necesare. La instalarea acestui tip de pachet, sunt instalate numai assembly-urile necesare proiectului în cauză ca să consume mai puțină memorie și să fie mai puțin vizibil. Dezavantajul este că un astfel de proiect, numit "multi-target project", este mult mai greu de întreținut de un dezvoltator.

Una dintre cele mai folosite variante pentru a căuta și descărca pachete folosind NuGet este NuGet Package Manager, care are o interfață intuitivă și tot ce trebuie să faci este să dai click dreapta pe proiectul în care vrei să instalezi pachetul anume și îi se deschide o fereastră în care poți să scrii numele pachetului, cu tot cu varianta lui, poți chiar să-ți selectezi părțile care le vrei din acel pachet, cu tot cu ce versiuni să aibă în parte fiecare, iar la instalarea pachetului, se vor instala și celelalte proiecte la care este legat ca să funcționeze cum trebuie (dacă este cazul să fie).Poți mereu să-i schimbi versiunea chiar dacă l-ai instalat, poți să-l ștergi destul de rapid prin interfața aceasta de Nuget și poți să vezi toate pachetele instalate, și să fi anunțat dacă unul din ele poate fi actualizat la o nouă versiune, dacă este lansată de dezvoltatorul ei.

2.2.2 Entity Framework Core

[Entity Framework Core](#) (EF Core) este o versiune mai mică, extensibilă , open source și cross-platform a bine cunoscutului Entity Framework, care este o tehnologie de accesare a datelor unică.

EF Core este un object relational mapper (ORM) care permite developerilor de .NET să lucreze cu baze de date (în special SQL Server) folosind obiecte de tipul .NET și elimină nevoie de a scrie codul pentru a face conexiunea dintre baza de date și chiar și-o creează dacă n-o ai, îți poate face legăturile între tabele dacă pui fie un atribut de forma [ForeignKey("nume_clasa")] sau dacă pui drept proprietate a clasei obiectul clasei către care vrei să se facă cheia străină (e o proprietate ORM, numită navigație).

Îți face automat operațiile de creare, citire, modificare și stergere (CRUD-urile) indiferent de modul de abordare, code-first approach sau database-first approach.

Ai niște metode deja valabile pentru accesarea bazei de date (crud-urile) și poți să folosești LINQ (language integrated query) ca să dictezi ce mai exact vrei din operația respectivă și, în funcție de ce adaugi sau modifici în modelele folosite pentru crearea tabelelor din baza de date, la următoarea migrație, modificările vor fi vizibile și în tabelele cutare.

Pentru început, ai nevoie de un model ca să poți crea tabela dintr-o baza de date sau care să se lege la ea, dacă există deja o baza de date, iar un model este compus dintr-o clasă entitate și un obiect de context (cum ar fi DbContext) ce ar reprezenta o sesiune-n baza de date, iar obiectul contextului îți permite să salvezi toate modificările.

Poți să dezvolti un model în 3 feluri: fie îl scrii tu de mână, fie îl generezi din baza de date și , ca să creezi sau să schimbi ceva din baza de date, te folosești de Entity Framework Migrations (EF Migrations).

Așa poți să permiti evoluarea bazei de date.

Cât despre folosirea [LINQ](#) în EF Core, ea e esențială pentru a interoga datele din baza de date, îți permite să folosești c# (sau orice limbaj din .NET) ca interogările să fie sigure.

Se folosește de contextul tău și clasele entități (clasele folosite pentru creearea tabelelor) că să facă legătură cu obiectele din baza de date.

EF Core dă interogării LINQ către furnizorul bazei de date, ce o transformă în limbaj specific bazelor de date, iar interogările vor fi executate întotdeauna.

Pentru aceasta, există obiectul DbContext și DbSet-urile, DbSet-urile fiind tabelele tale în varianta de cod, iar DbContext fiind contextul curent al bazei de date în codul tău.

```
List<T> theItems= _dbSet.Select(item => item).ToListAsync().Result;
```

Figura 4: Cum să obții datele dintr-o tabelă prin LINQ

Migrațiile au ca rol să se sincronizeze cu modelul de date și să își mențină baza de date deja existentă. Ele funcționează în 2 feluri.

În primul, când un model este introdus, se folosește EF Core Tools ca să se facă și să se bage o migrație cu detalii necesare ca să se mențină schema bazei de date sincronizate bine. Modelele sunt comparate cu un "snapshot" cu modelele anterioare ca să se observe discrepanțele și să se adauge fișierele acelea pentru nouă migrație posibilă.

În al doilea mod, când migrația a fost deja generată, este cu puțință să se aplice asupra bazei de date în tot felul de varietăți.

Toate migrațiile sunt stocate într-un istoric de migrații, care e o tabelă undeva prin baza de date.

Acest istoric ajută enorm pentru a putea lucra în echipă cu ceilalți programatori și pentru a putea gestiona splendid dezvoltarea întreagă a lucrării anume, deoarece absolut fiecare schimbare este prezentă acolo, necontînd cât de mică este.

Mereu poți să te întorci înapoi la o migrație anterioară datorită acestui istoric minunat.

Pentru o migrare reușită în Entity Framework Core, trebuie, totuși, urmat un plan anume.

Primul pas ar fi, evident, crearea modelelor, care le-am menționat mai sus.

Ele reprezintă structura datelor proiectului, iar proprietățile sunt echivalentul coloanelor din tabelă, inclusiv cele care au atribut ce ar determina chei străine sau obiecte ale altor clase. Acestea se întâmplă numai când transformarea are loc cu succes.

Partea a 2 a ar fi crearea contextului (DbContext).

Trebuie creată o clasă cu orice nume se dorește (de pildă RelocationDbContext), dar care trebuie să extindă clasa DbContext și poate configura orice relație dintre orice entitate, cu tot cu restricțiile și ce mai cere baza de date. Configurația determină maparea entităților la tabele, dar și cum se gestionează relația dintre acestea. Se leagă una de alta.

După, urmează generarea migrărilor în sine. Cum am mai spus, ele sunt o cale de a și vizualiza posibilele modificări ce sunt dorite asupra bazei date, cât și un mod de a le pune în acțiune, iar ca să poți începe un astfel de procedeu, trebuie mai întâi ca în Visual Studio să scrii vreo 2 comenzi specifice migrărilor, prima fiind simplă.

Ea urmează un tipar de forma add-migration [nume_migratie], după se creează, și vei putea decide dacă să o aplici sau să mai modifici ceva înainte de asta.

Și, într-un final, după ce te-ai hotărât, poți să aplici migrația respectivă și să vezi modificările că au fost aplicate pe baza de date.

Ca un dezvoltator să poate să facă una ca aceasta, el trebuie să scrie în linia de comandă (consola) o comandă destinată migrărilor, care are forma update-database, și asta este tot. Schimbările au avut loc și migrarea anterioară a fost salvată și se poate reveni la ea oricând dacă este cazul.

DbContext este una dintre cele mai importante entități din tot framework-ul .NET. Datorită lui poți avea legătura dintre proiectele ce folosesc .NET și bazele de date respective, devenind o parte esențială pentru Entity Framework în sine, nu doar în Entity Framework Core, el făcând mare parte din ce ar face un object relational mapper (ORM).

Fără el nu ar exista Entity Framework aşa cum îl ştim.

Prin ea se începe și se manageriază interacțiunea dintre modelele de date și infrastructura materială a bazei de date. Sau, cu alte cuvinte, prin ea putem defini tabelele cu DbSet<nume_clasa>, ce le va mapa în baza de date care, fie o va crea, dacă nu există, fie să o modifice, dacă este deja prezentă în aplicație.

Și relațiile (cheile străine, restricțiile, indexi și tot aşa) vor fi mapate, indiferent de modul cum au fost scrise în cod (fie că s-a pus un tag deasupra unei proprietăți a clasei, fie că s-a folosit Fluent API).

Cum s-a menționat și mai înainte în descrierea EF Core-ului, se vor face automat metodele de creare, citire, modificare și stergere (CRUD-urile) asupra obiectelor ce au servit drept scop crearea tabelelor, ele fiind prezente la nivel de cod (aici în C#), și ele vor fi transformate în interogări SQL (în contextul acesta), când vor fi folosite în program aceste funcții preimplementate.

Ca cineva să poată să folosească aceste facilități este necesar să se deriveze clasa DbContext, ce vine cu pachetul, care poartă același nume că namespace-ul, Microsoft.EntityFrameworkCore.

Ai nevoie de o clasa să moștenească DbContext ca să poți beneficia de crearea, gestionarea și modificarea bazei de date, toate fiind automatizate.

Sunt și atâtea metode esențiale care-ți ușurează viață aşa de tare că nu ai mai dori să scrii vreodată o bază de date de la 0 și să-i faci conexiunea manual în cod vreodată.

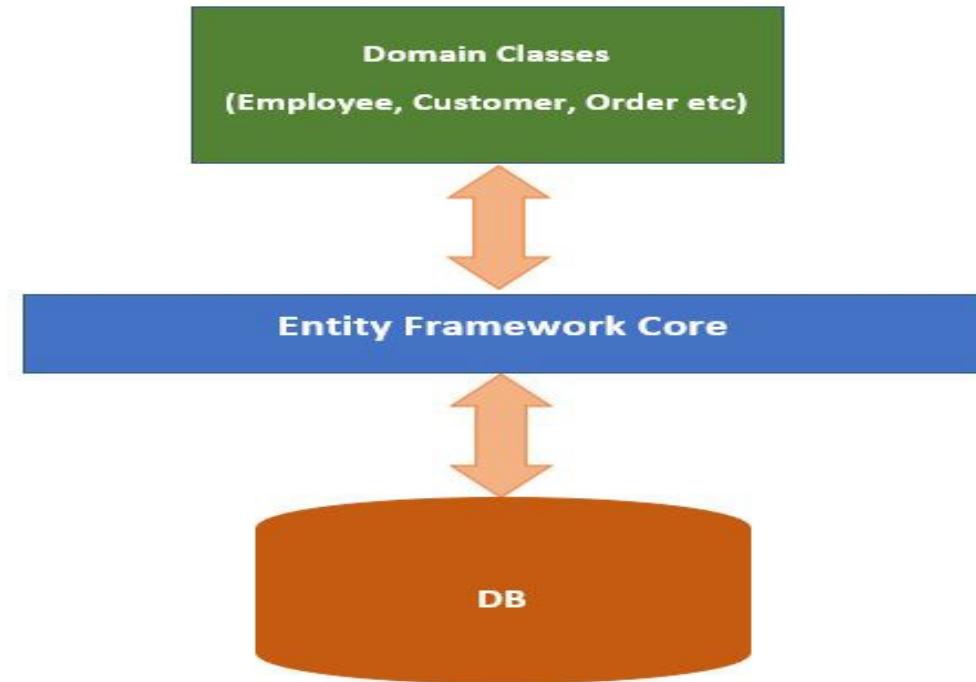


Figura 5: Schemă pentru modul de operare al Entity Framework Core

Este știut faptul că EF Core face o treabă extraordinară în abstractizarea a multor detaliu de programare, dar există niște practici recomandate pentru orice ORM pe piață.

Să ai informații măcar de nivel intermediar despre server-ul bazei de date este chintesențial ca să-l concepi în mintea ta, să-i faci debug și să poți să faci migrațiile bine în aplicații ce țin mult la performanță.

De pildă, să știi restricțiile, cheile străine, indecși, normalizări, tipuri de date, și tot aşa.

Altă practică bună ar fi testarea funcțională și de integrare.

Dacă poți să faci o copie cât mai apropiată de versiunea de producție ca să poți găsi erori , dacă apar, când folosești o versiune sau altfel de ediție a bazei de date , iar, alt context, să-ți dai seama de unde apar diverse erori greu de rezolvat, cauzate, probabil, de versiunile de EF Core, AutoMapper și alte unelte virtuale când le actualizezi.

Alt detaliu ar fi să poți să planifici cum vor fi aplicate migrațiile în istoric asupra proiectului când e stadiul de producție.

Dacă ai vrea să o faci când se lansează, e chiar destul de posibil să se înfrunte cu erori de concurență sau să nu aibă permisiunea să facă anumite schimbări ce le doresc migrațiile.

De asta e bine să se încerce un "staging environment", ca să poată aplicația să-și revină din erori fatale.

Un ultim principiu util este examinareameticuloasă și testarea multă a acestor migrații.

Ele trebuieesc riguros testate înainte să fie aplicate pe date de producție. Când proiectul e deja în stadiul de producție, nu mai este la fel de ușor să schimbi formă schemei ,iar coloanele tabelelor cu atât mai puțin.

2.23 Rest API

Pentru început, un [API](#) în sine (Application Programming Interface) este un intermedier pentru a face posibilă legătura dintre diverse aplicații și pentru a putea transfera date între ele.

Poți să faci cereri către cealaltă aplicație cu ce date are nevoie aplicația ta să afișeze, sau să altereză ca apoi să o afișeze, iar server-ul celeilalte platforme , în mod normal îți trimit datele ei dacă sunt disponibile, altfel îți dă un Bad Request.

Niște avantaje enorme ar fi că sunt foarte ușor de integrat în orice fel de proiect.

Tot ce ai nevoie e să faci legătura între ce API-uri ai nevoie, iar datele le poți avea furnizate instant ca să te apuci să lucrezi cu ele numai decât.

Altul ar fi că reduce enorm volumul de muncă necesar proiectului, deoarece nu mai este nevoie să stai tu să creezi o facilitate de dimensiuni medii sau mari ca să-ți poată furnizație datele pe care ai nevoie să le afișezi.

Poți direct să faci o cerere către o aplicație care face deja treaba cu procuratul datelor finale, și tot ce ai tu de făcut este să-i faci o cerere de obținere a lor (un GET request) și, gata, deja le ai și poți lucra cu ele.

La fel și cu securitatea aplicației. Poți folosi software extern prin aceste API-uri și tu să primești, de exemplu, validarea necesară, sau chiar parola criptată de software-ul din afară, și câte și mai câte.

Revenind la subiectul principal, un REST API (Representational State Transfer Application Programming Interface) este un protocol de comunicare, ce a fost făcut să urmeze niște reguli și principii stricte pentru comunicarea pe web.

Ei se ocupă de cererile clientilor, fie că doresc să vadă niște resurse într-un format inteligibil lor, sau să modifice date din server.

Sunt 6 principii esențiale pentru a urma arhitectura REST API eficient și bine.

Primul este Separarea clientului de server, adică software-ul destinat clientului, partea vizuală, să fie separată total de partea de back-end, nevizuală a proiectului, ca să se poate lucra separat. (nu e neapărat nevoie să fie în instanțe diferite de IDE-uri).

Ajută mult la dezvoltarea proiectului, deoarece pot lucra echipe separate pe ele și se pot forma independent una de cealaltă.

A doua ar fi interfața uniformă. Ea este necesară pentru ca partea clientului să poată comunica cu tot felul de API-uri de tipul REST, adică să fie o conexiune compatibilă cu orice back-end posibil și să poți face cereri de date indiferent de situație.

Datele primite deobicei vin și cu niște detalii ce pot fi arătate utilizatorului final în caz că nu înțelege ceva.

Al treilea este să nu fie nevoie ca anumite cereri să fi fost folosite pentru a putea face un ciclu de cerere și răspuns între platformele web (Stateless principle).

E bine să existe independență între toate cererile.

Al patrulea ar fi ca sistemul să fie împărțit pe straturi (Layered system).

Adică se recomandă să mai existe niște proiecte (măcar) care să se ocupe de securitate, de trafic și să verifice cererile sau răspunsurile înainte să se ajungă la partea de servicii din server.

Acuma, poți să le faci și direct în partea de server, fără alte proiecte la nevoie.

Al cincilea este să te poți folosi de memoria cache pentru a folosi date, nu aşa de importante, dar utile, temporar pentru utilizatorul final care se folosește de aplicația ta (Cacheable).

Și ultimul principiu ar fi să poți să trimiti cod clientului (code on demand).

Este opțional și poți să-i trimiti printr-o metodă în controller un sir de caractere ce conține codul sau ce date tehnice sunt, sau, dacă poți, un JSON, asta dacă chiar e absolut necesar.

Fără aceste principii, nu ar fi posibilă o adaptare între partea clientului și cea a serverului fără intervenții de la programatori frecvent, și fără acest design pattern, ar rămâne în urmă aplicația respectivă.

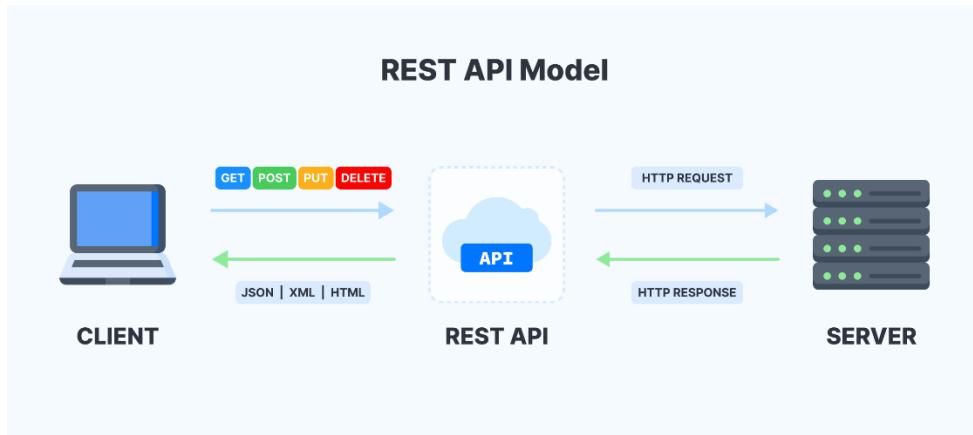


Figura 6: Schemă pentru model de REST API

Un aspect fundamental când se folosește un REST API sunt metodele HTTP și e obligatoriu să fie într-un proiect ca să poată să se facă comunicarea între client și server sau servere. În mare parte, se folosesc 4 metode diferite.

Prin metoda GET poți să obții informații de la server-ul care se ocupă cu datele pe care clientul le-a cerut și i le trimite ca răspuns clientului. Acest tip de metodă nu e intenționat să modifice nimic din server sau baza de date.

Prin metoda POST se preia informația dată de client și trimisă către server ca, din ea, să se creeze o nouă resursă și, cel mai probabil, să se adauge în baza de date, dacă toate condițiile pentru a face resursă sunt îndeplinite de utilizator. Deobicei se va arăta o notificare către client cu ceva informații legate de resursa creată și că acțiunea a fost finalizată cu succes.

Prin metodă PUT, se va compara resursă deja existență, dacă există, cu nouă versiune a resursei trimisă de client și se va schimbă cu noile date, dacă corespund cu cerințele platformei web și ce alte restricții sunt pe acel context de aplicație.

Și nu contează de câte ori se întâmplă aceeași cerere (dacă e identică) către server, că resursă va avea aceleași date. (idempotență).

Prin metodă DELETE, se șterge resursa menționată de utilizator din baza de date, în cazul în care există și are permisiunea să o eliminate.

Acestea sunt metodele HTTP principale și cele mai folosite în orice REST API, nefiind musai nevoie de altele.

2.24 Rutare Aplicație

Rutarea (routing) este, fără dar și poate, încă un element esențial pentru orice web API existent. Ea determină cu ce metode se potrivesc cererea clientului, deoarece fiecare metodă dintr-un controller are o rută anume și, poate chiar făcută să accepte anumite tipuri de cereri, sau poate doar una singură.

Astfel, se decide care metodă trebuie folosită pentru cererea respectivă. Sistemul de rute folosește niște şabloane de rute ca să definească pattern-urile pentru url-uri.

Acuma, în controlere poți să te folosești de tag-uri precum `Route("url")` ca să poți seta o ruta (endpoint) pentru metoda anume, dar nu se va uita și ce fel de cerere vine către ea.

Dacă folosești, de exemplu, `HttpPost("url")`, vei putea seta și ruta și ce fel de cerere e permisă.

În versiunile mai noi, configurarea rutelor se face în clasa `Program.cs`, fiind nevoie să se utilizeze obligatoriu niște cod precum `app.UseRouting()` și `app.MapControllers()` sau `app.MapControllersWithViews()` ca să poți permite sistemului de rutare să funcționeze în ASP.NET Core.

Astfel, rutele metodelor din controlere să poată fi comparate cu cele ce provin de la o cerere din front-end, partea vizuală a aplicației, de unde clientul a apăsat, cel mai probabil un buton, ce conținea și o rută.

Acestea se întâmplă datorită unui mecanism intern de rutare, creat de Microsoft și care e mult mai simplu de folosit, decât să stai să scrii tu middleware-uri ca să determine în ce ordine trebuie traversate cererile.

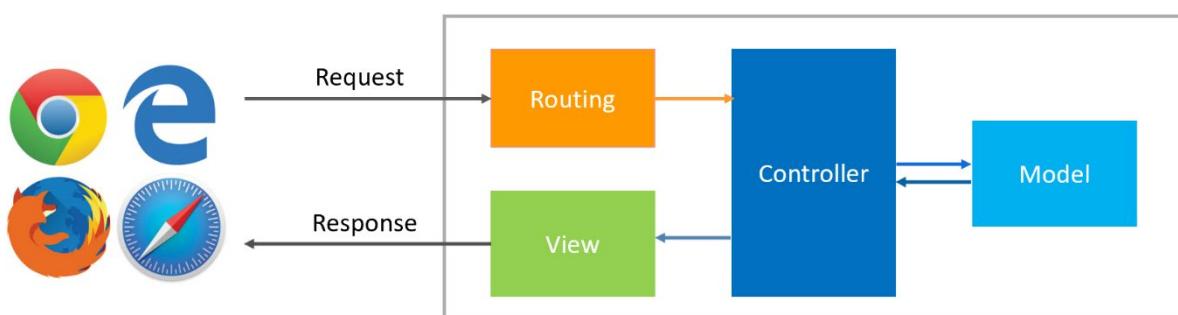


Figura 7: Schemă pentru rutare în ASP.NET Core

2.25 Controller

Controller-erele sunt făcute să creeze și să grupeze un set de diverse acțiuni în cod. Aceste acțiuni, sunt, defapt, metode ale clasei controller, care se ocupă de rezolvarea cererilor din front-end. Fiindcă acțiunile sunt grupate într-un controller anume, se pot aplica alte rute și chiar măsuri de securitate pe toată clasa controller respectivă, ca toată gruparea de acțiuni să aibă parte de reglementările acelea.

O clasă controller se creează, deobicei într-un fișier numit Controllers, sau se preia din namespaceul Microsoft.AspNetCore.Mvc.Controller. Convenția este că acea clasă, care moștenește clasa Controller, să conțină și numele "Controller" în denumirea ei, sau minimul necesar, să facă doar moștenirea. Funcționează și doar dacă îi pui tag-ul [Controller] deasupra clasei mai nou.

Până la urmă, aceste controller-ere, sunt niște abstractizări ale interfeței grafice. Obligativitățile ei sunt ca informațiile cerute să fie valide și să aleagă care view (pagină web), în cazul de aici, ar trebui trimis.

Revenind la discuția cu modele, toate acestea împreună formează pattern-ul MVC (model- view-controller), fiindcă datele, care sunt ținute în obiectele instantiate, au o clasă drept unul din modele, controller-ul are metodele cu rutele care manipulează datele cutare și tot prin el se trimit view-ul necesar către utilizator.

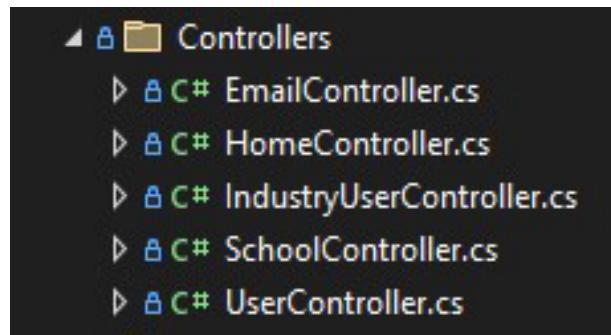


Figura 8: Fișier Controllers cu clase controller

2.3 Implementare client-side

2.31 Views

În design pattern-ul Model View Controller (MVC) , un view (un fel de pagină web) face posibilă vizualizarea datelor finale destinate utilizatorului final într-o formă acceptabilă pentru el și îi permite să și interacționeze cu ea (să o modifice, să o șteargă, mai adaugă ceva, etc). Un view în sine este un şablon HTML ce are încorporat înăuntru sintaxa de Razor, adică cu un fel de cod ce se poate îmbina cu codul HTML pentru a putea face o pagină web cu un aspect bun pentru înțelegerea șifolosirea sa de către orice om obișnuit.(cum ar fi un client)

În contextul folosirii framework-ului ASP.NET Core , creat de Microsoft, care folosește limbajul de programare C#, și aceste view-uri vor putea folosi cod C# îmbinat cu limbajele de front-end (html, css, javascript, bootstrap, etc), chiar dacă codul de C# este terminat primul și abia și cele de front-end, deoarece ele au terminația .cshtml, ce fac posibile aceste chestiuni.

Când se urmează modelul MVC în ASP.NET Core, view-urile sunt puse în fișiere care, în mare parte din cazuri, au numele controller-ului care le tot targetează cu acțiunile (metodele) lui. Aceasta se întâmplă în cazul în care un singur controller folosește view-urile respective, altfel vor fi puse într-un fișier la comun (Shared). Toate aceste fișiere, ce conțin view-urile, sunt puse într-un fișier numit Views. Aceasta este o practică necesară datorită modului de funcționare a ASP-NET Core-ului.

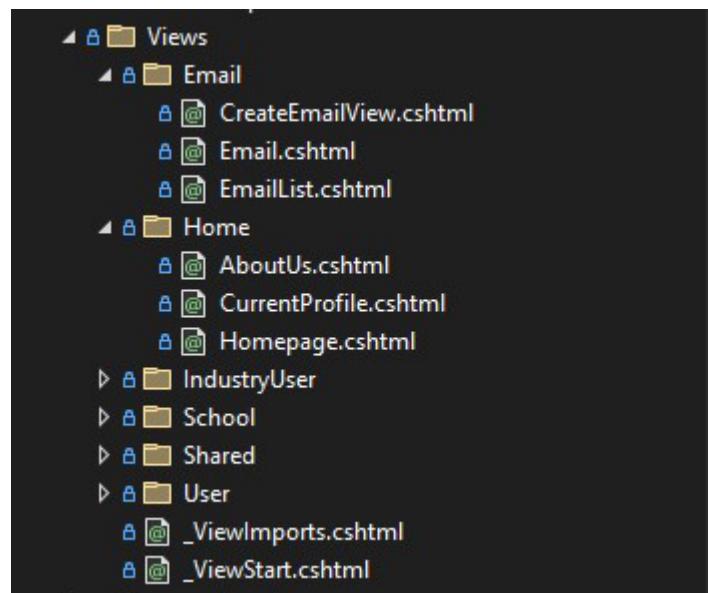


Figura 9: Ierarhie view-uri

Când se creează fișierul Shared, este recomandat ca în el să se afle tot timpul view-ul cu denumirea "_Layout.cshtml", deoarece el va fi comun tuturor view-urilor care vor fi parcuse prin platforma web de către utilizator.

În el poți să pui, de exemplu, o imagine de fundal a aplicației, o bară de navigare și ce fel efecte dorești (dacă e cazul).

Astfel mai reduci și repetarea codului. Ele mai contin și detaliile de <header>, cum ar fi denumirea platformei, date meta și restul datelor necesare de a putea fi găsită platforma web când o cauți pe net de exemplu.

Tot acolo poți aduce fișiere externe, cum ar fi o versiune de bootstrap, de javascript, css, jquery, etc.

Există și _ViewStart.cshtml, în care se poate seta Layout-ul principal prin proprietatea Layout, ce așteaptă calea către view-ul ce servește drept layout principal al aplicației web.

Uneori merge și direct să scrii doar numele întreg al view-ului, dacă se află în fișierul Shared.

Când numele unui view parțial e dat, motorul de căutare Razor views se activează și va căuta o parte din ierarhia din fișierul Views menționată mai sus până găsește view-ul cu același nume.

Se folosește un proces standard de descoperire (standard discovery process), adică, mai întâi se caută fișierul ce are același nume cu al controller-ului, apoi fișierul Shared.

View-ul _ViewImports.cshtml e destinat importării namespace-urilor pentru a putea folosi clasele necesare când definești structura modelului ce ar veni din back-end.

Poți să mai adaugi de exemplu importul de tag-uri pentru views și alte namespace-uri create special de Microsoft pentru view-uri.

Restul view-urilor din fișierul Shared sunt [view-uri parțiale](#) (partial views), deoarece ele pot fi apelate în alte view-uri (deobicei din fișiere ce poartă numele controller-elor aferente). Astfel, se poate evita repetiția redundantă.

Un view parțial poate fi apelat cu sintaxa <partial name = "nume_view_partial" /> sau @await Html.RenderPartialAsync("nume_view_partial"), astfel conținutul view-ului parțial va apărea în view-ul principal indiferent de ce s-ar afla în el și, desigur, fără erori. Se recomandă a două varianta mai mult.

Un beneficiu enorm al existenței view-urilor într-un proiect MVC este că ajută la formarea principiului de design numit separarea preocupărilor (separation of concerns), care împarte aplicația în 3 părți.

Un proiect pentru partea cu bazele de date și modelele ei, un proiect pentru partea de logică și manipulare a datelor, iar al treilea (în cazul de aici, un fișier principal într-un proiect de rulare) partea de vizualizare a rezultatelor finale și a conexiunii dintre view-uri). E ideal să existe acest principiu, căci, fără el, n-ar exista MVC în sine.

Astfel, e mult mai simplu să dezvolți platforma web, fiindcă totul e foarte bine organizat, mai ales fișierele cu view-urile.

Le găsești imediat când ai nevoie să schimbi sau să adaugi ceva la ele și câte și mai câte. Alt beneficiu esențial este că dezvoltarea părților din platformă (în special a view-urilor) se poate face independent.

Deci, la o modificare în view-uri, nu trebuie să stai să modifici și alte părți în tot proiectul mare.

2.32 Legarea datelor platformei

S-a mai menționat că view-urile sunt legate de un controller, deoarece ele sunt plasate într-un fișier cu numele controller-ului care le folosește, iar dacă sunt la comun între controller-ere, sunt puse în fișierul Shared.

Ele sunt returnate de acțiunile controller-elor ca un ViewResult, ce face parte din clasa ActionResult, motiv pentru care metodele ce returnează un view sunt, deobicei, cu tipul return IActionResult.

Datorită faptului că clasele ce servesc drept controller-ere moștenesc clasa Controller, pot folosi funcția View() pentru a le fi mai ușre.

Dacă scrii doar View(), se va căuta view-ul cu numele metodei prin view-uri, iar dacă pui View("nume_view"), va căuta exact acel string identic prin numele view-urilor.

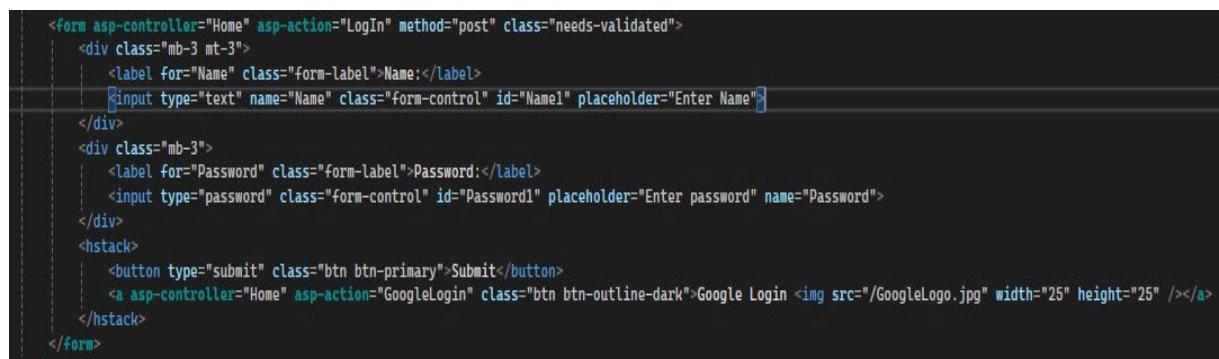
Poți pasa și un model pentru view, dacă poate susține unul, scriind View("nume_view", "nume_model"), iar modelul e structurat în view.

Când se returnează un view, procesul cu numele de descoperirea view-ului (view discovery) pentru determinarea view-ului ideal are loc.

Detalierea celor 3 metode de View() descrie perfect procesul de căutare al view-ului dorit. Se menționează că e un proces diferit față de căutarea view-urilor parțiale, acolo lucrând Razor Views engine. Cel mai simplu mod de a trimite date din front-end spre back-end este printr-un formular (form), care conține câmpuri (deobicei de tipul input), care dețin un id și un nume.

În back-end le poți accesa prin numele lor, în front end, prin javascript, le poți accesa prin id și să le trimiti de acolo într-un controller pentru a putea valida și primi cu succes datele clientului. Când utilizatorul apasă pe butonul cu funcționalitatea de trimisere, formularul trimite datele către controller-ul cu acțiunea ce are aceeași rută identică cu cea menționată în formular când a fost compus de programator.

Formularul conține un atribut numit acțiune (action) care poate primi drept valori un url sau un sir de caractere ca să se specifice unde mai exact trebuie trimise datele formularului respectiv. Server-ul (controller-urile) se ocupă de restul detaliilor, trimițând o notificare dacă acțiunea a fost finalizată cu succes sau nu.



```
<form asp-controller="Home" asp-action="LogIn" method="post" class="needs-validation">
    <div class="mb-3 mt-3">
        <label for="Name" class="form-label">Name:</label>
        <input type="text" name="Name" class="form-control" id="Name1" placeholder="Enter Name">
    </div>
    <div class="mb-3">
        <label for="Password" class="form-label">Password:</label>
        <input type="password" class="form-control" id="Password1" placeholder="Enter password" name="Password">
    </div>
    <hstack>
        <button type="submit" class="btn btn-primary">Submit</button>
        <a asp-controller="Home" asp-action="GoogleLogin" class="btn btn-outline-dark">Google Login </a>
    </hstack>
</form>
```

Figura 10: Exemplu de formular într-un view.

2.33 Bootstrap 5.3

Vorbind despre Bootstrap, el este un framework de front-end încă destul de folosit pe piață, datorită faptului că e mult mai ușor de pus în practică, fiind un înlocuitor de CSS foarte util, având chiar și funcționalități de javascript în el.

Cu el poți dezvolta aplicații web în timp record, fiind destinat interfeței pentru client. A fost creat de fostul Twitter (acum numit X) și a avut succes imediat în industria web.

Caracteristicile esențiale ale acestei unele nu sunt puține la număr, dar câteva pot fi detaliate.[15]

Responsivitatea: în el se află un sistem de grile ușor adaptabil, redimensionarea ecranului pentru aplicație fiind foarte leșne de îndeplinit chiar automat, la diverse dimensiuni pe ecran.

Motiv pentru care Bootstrap e foarte bun și pentru dezvoltarea aplicațiilor mobile, nu doar pe web.

Componente predefinite: poți să folosești forme cu diverse atribute precum bara de navigare, butoane cu numeroase opțiuni de stilizări, alerte de ce dimensiuni dorești, iconițe de afișare pentru notificări la un sistem de mail, formulare, slideshow-uri, tot ce ai vrea vreodată când vrei să-ți faci de cap cu o aplicație fie de web, fie de telefon.

Stilizare și teme consistente: Dezvoltatorii Bootstrap au asigurat seturi predefinite de CSS pentru a putea stiliza o foarte mare parte din componente predefinite (mai mult), dar se poate, într-o oarecare măsură, și cele normale de html să le stilizeze.(de exemplu la poziție și dimensiuni).

Aspectul vizual devine consistent, iar aplicația e chiar atrăgătoare pentru a o folosi de oricine.

Suport pentru browsere: Suportă o gamă destul de respectabilă de browsere de pe piață precum Google, Microsoft Edge, Brave și multe altele.

Se poate asigura o experiență coerentă tuturor, necontînd tipul de platformă folosit, poate fi folosit pentru orice fel de aplicație web și oferă redimensionarea platformei respective după dimensiunea dispozitivului curent.

Documentație fenomenală: Din documentația lor este imposibil să nu înțelegi sau să găsești ce ai nevoie pentru ati îndeplini sarcina, deoarece ea estemeticulos detaliată, dar și explicată în cuvinte cât mai simple posibil, fiind chiar și exemple de utilizare atât de simple, că poti să înveți programare de front-end neștiind deloc informatică sau să fi din domeniul acesta.

2.34 Javascript

Acest [limbaj de programare](#) este interpretat ușor (compilat la fix) cu funcții care sunt tratate ca o variabilă oarecare (pentru a putea fi pasate ca parametri în alte metode și pentru a fi asignate variabilelor, și chiar să fie returnată de altă metodă).

În general, [javascript](#) e știut pentru a fi un limbaj de scripting pentru web, dar e folosit de alte aplicații care nu sunt pe web (cum ar fi Adobe Acrobat).

E și un limbaj de tip prototip (poți crea obiecte fără să le definești o clasă, le poate deriva din instanța altiei clase sau chiar să île adaugi unui obiect gol).

Mai este și de tip multi paradigmă (adică suportă mai mult decât object oriented programming, de exemplu, programare imperativă, declarativă/funcțională, etc). Este și de tipul single-threaded, și dinamică.

Datorită faptului că javascript este dinamic, el poate avea liste de parametri variabile, variabile funcții și alte facilități avansate.

Un detaliu esențial pentru folosirea lui, mai ales pe web este programarea asincronă în javascript.

Așa poți da start-ul unui task de lungă durată, păstrând însă responsivitatea limbajului, nefiind nevoie să aștepți până se termină sarcina în sine, iar la finalul ei vezi și rezultatul.

Acea programare a fost creată ca funcțiile normale, precum cele oferite de browser, adică `fetch()`, pot dura foarte mult uneori, făcând utilizatorul să aștepte mult timp până se termină procesul început.

Motivul este că, în programarea sincronă în javascript, trebuie să aștepți fiecare linie de cod să se execute pe rând ca să poată să treacă la următoarea, nefiind permisă trecerea mai rapidă sau concurrentă aici, iar asta, uneori, poate costa mult timp. și fiindcă e single-threaded, dacă mai ai alte facilități pe lângă cea utilizată la acel moment, se poate bloca temporar până la final sarcinei anterioare.

De aceea, programarea asincronă salvează ziua, eliminând toate inconveniențele acestea. Acest limbaj e și capabil să trimită date către back-end din front-end și chiar să cheme metode din controler-ere în front-end și să facă automat convertirea lor prin JSON pentru a le expune în structurile html din paginile web respective.

Capitolul 3

Considerații relativ la implementare

3.1 Structura aplicației

Această platformă web este compusă dintr-o parte de back-end, ce se ocupă atât de conexiunea cu bazele de date aferente, cât și de manipularea datelor ce ies și se duc din ea, de ținerea modelelor folosite pentru clase și de modificările aduse lor în serviciile folosite și chiar controller-urile din portiunea de back-end ce face legătura cu front-end-ul în sine.

Această secțiune principală a fost făcută în ASP.NET Core.

Acuma, există și partea de front-end, care a fost făcută în Razor Views, dar care urmează design pattern-ul MVC, motiv pentru care nu s-au folosit toate facilitățile specifice Razor Views, deoarece nu ar fi funcționat cum trebuie. Aceste view-uri servesc drept pagini web ce permit cod html și toate structurile lui, putându-se astfel să se folosească și css și javascript pentru un aspect frumos al fiecărei pagini web în parte, dar și pentru a avea o responsivitate rapidă.

Fiindcă se permite folosirea css-ului, se poate folosi și bootstrap, ce are n componentă o groază de componente și variante de costumizare simplificate.

Și cum view-urile nu necesită un IDE separat, ci doar un fișier numit Views, lucrurile devin mult mai simple.

Această parte s-a ocupat de afișarea datelor din back-end într-o formă cât de plăcută s-a putut realiza.

Baza de date a fost făcută în SQL Server și , în ea, s-au stocat toate datele necesare în tabele create prin Entity Framework Core, gestiunea acestei baze devenind extrem de ușoară și rapid de schimbă ceva la ea prin CRUD-uri.

3.1.1 Arhitectura server-side

S-a folosit o arhitectură curată (clean architecture) pentru a împărți tot proiectul într-o ordine chiar clară și a devenit, astfel, ușor de navigat prin cod, iar părțile între ele comunică eficient, toate lucrând împreună.

Ea este compusă din mai multe straturi, unul din ele fiind stratul de prezentare. El este partea superioară a arhitecturii oricărui fel de aplicație existentă și, în ea, clientul trebuie să poată să interacționeze cu ușurință și să și primească informațiile cerute de acesta cât se poate de intelibile unui om normal.

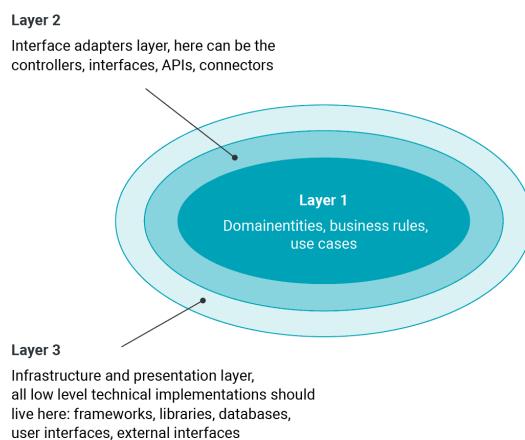


Figura 11: Schemă pentru arhitectura curată

Acest strat este un intermediar între clientul ce se folosește de platformă și restul întregului proiect.

Rolul esențial al ei este, literalmente, să traducă datele tehnice (intelibile, probabil, programatorului), adică responsabilitățile și rezultatele, pe limba clientului. Astfel, el poate și acționa asupra datelor, deoarece le poate înțelege.

Și, fiindcă comunicarea dintre interfața grafică (UI-ul) și back-end-ul aplicației se realizează prin controller-e, apar noi oportunități în acest strat de prezentare.

Precum ar fi gestionarea interacțiunii cu utilizatorul și trimiterea cererilor și răspunsurilor dintre aceste controller-e și interfața grafică a acestei aplicații.

Datorită acestui intermediar, se poate implica navigarea între paginile web, validarea datelor scrise de client și afișarea și actualizarea datelor în diferite ecrane sau facilități ale aplicației web.

```
public class UserController : Controller
{
    private readonly ServiceWrapper _serviceWrapper;
    private static List<int> ChosenServices = new();
    3 references
    private static string Destination { get; set; }
    0 references
    public UserController(IBookingService bookingService, IFurnitureService furnitureService, IJobService jobService, IRentingService rentingService, ITransportService transportService,
        IUserService userService, IIndustryUserService industryUserService, ISchoolService schoolService)
    {
        _serviceWrapper = new(bookingService, furnitureService, jobService, rentingService, transportService, userService, industryUserService, schoolService);
    }
    [Route("user home")]
    0 references
    public IActionResult UserHome()
        => View("UserView", $"Welcome {CurrentUser.Name} , you have logged as an user.");
    #region IService
    5 references
    private List<AbstractModel> FilterItems(IService service)
        => service.GetItems().Where(item => item.Location.Equals(Destination)).ToList();
}
```

Figura 12: Exemplu de controller în cod

Se observă că aici, acest controller destinat acțiunilor pentru utilizatori este format dintr-o clasă ce moștenește clasa controller, primește niște servicii prin injectarea de dependințe pentru a putea folosi cu succes serviciile necesare și a le trimit view-urilor prin metodele sale.

Ele au deasupra fie atributul Route cu url-ul aferent, fie unul din atrbutele HttpPost, HttpGet, HttpPut, HttpDelete (care, din păcate, nu apar în această poză).

Orice metodă cu o rută alocată ei este o acțiune a controller-ului ce are ca scop să returneze un view anume, cu sau fără un model de trimis către ea, acestea fiind posibile deoarece s-a moștenit comportamentul clasei Controller.

Și datorită dependency injection-ului (DI-ului), se pot folosi facilități externe ale aplicației în cadrul controller-ului. Aceste servicii provin din stratul principal al platformei și ele sunt menite ca să se poată interacționa cu logica de afaceri din controller.

În partea de logică și business, ar fi proiectul Core, care ar fi cel mai important strat din proiect, pentru că el conține clasele de C# ce au drept scop manipularea datelor atât provenite din stratul de date, cât și din cel de prezentare, și, datorită acestei caracteristici, poate servi drept intermediar între aceste 2 straturi.

Aici se află serviciile, ce manipulează datele primite, de orice fel și sunt trimise în controller-ere prin injectarea de dependințe, controller-ul respectiv primind serviciul prin constructorul său.

Chintesația acestei părți este, totuși, că prin el se face implementarea bunei judecăți de afaceri, sau, cu alte cuvinte, logica aplicației și regulile domeniului în sine sunt bine puse la punct pentru a avea certitudinea că operațiile desfășurate sunt cele necesare cu cerințele obiectivelor caracteristice afacerii.

Tot el are rolul obligatoriu de a valida sau invalida datele care vin către el, pentru a decide dacă sunt apte să meargă la următorul strat, care ar fi stratul de date. Validarea aici constă-n corectitudinea și integritatea datelor care au ajuns până în acest punct în aplicație.

Alt rol ar fi să facă calcule pe datele primite și chiar transformări pentru a trece prin toate etapele de verificare necesare și chiar pentru a putea merge mai departe așa în stratul următor.

Faptul că acest strat există aduce, cu sine, destule avantaje, precum mențenanța codului, deoarece ea poate permite și arhitecturi mulți-nivel, schimbările din cod devenind rapid vizibile și ușor de depistat.

Alt avantaj este securitatea, deoarece stratul superior, cel de prezentare, nu comunică direct cu stratul de date, adică cu baza de date, ca să aplique normele de securitate pentru datele furnizate de client.

Astfel, nu se pot pierde date aiurea și datele, când vor merge mai departe, vor fi într-un format securizat și acceptabil pentru a putea ajunge în baza de date, trecând prin stratul următor.

Un bun avantaj este că, în mare parte, doar proiectul ce se ocupă de logică și business-ul platformei web este de obicei modificat la versiuni noi ale aplicației, astfel, aplicația va rula mult mai ușor și mai repede când acestea se finalizează.

Și este și mult mai simplu atât de pus în cod toate acestea, cât și de învățat aplicația, deoarece poți să știi sigur că toate părțile dintr-un proiect lucrează împreună la realizarea stratului cutare, și dacă programatorul știe teoria acestui design pattern, îi va veni mult mai ușor și rapid să înțeleagă ce face acea parte din proiect și, la final, cum se leagă proiectele între ele să formeze platforma web în sine.

Stratul de date este o componentă din arhitectura software , aici, a arhitecturei curate, care se ocupă cu asigurarea conexiunii dintre stratul de business și logică cu baza de date.

În contextul acesta, s-ar afla în proiectul numit Datalayer și e capabil să aplice operații de modificare, ștergere, inserare, etc (CRUD-urile).

Una din sarcinile ei este de a conecta la baza de date și a menține conexiunea stabilă, ca să se poată executa operația sau operațiile ce vor avea loc asupra bazei de date, deoarece acesta este motivul pentru care s-a deschis o conexiune inițială.

Alt rol este să se execute cod sql (dacă biblioteca folosită pentru folosirea bazei de date permite acest lucru) sau să folosească metode predefinite (cum ar fi cele din DbContext și DbSet-uri) pentru a executa CRUD-urile.

Alt rol este de a permite maparea datelor din baza de date în obiectele ce au menirea dată de dezvoltator, să țină în ele datele respective, conexiunea fiind ținută de acest strat special. Se menționează că maparea se poate face și vice versa (mai ales cu EF Core).

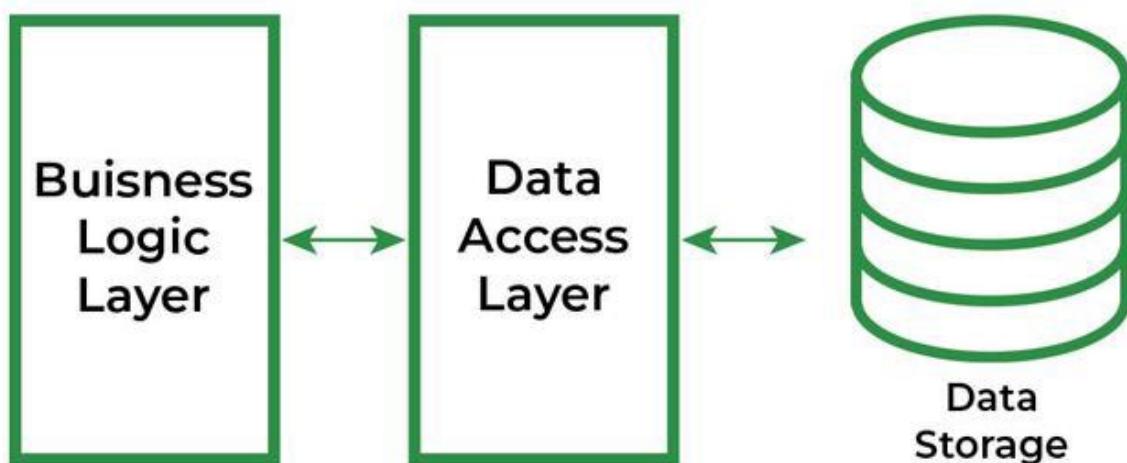


Figura 13: Schemă de reprezentare a modului de funcționare pentru stratul de date

3.1.2 Arhitectura client-side

Toată partea vizuală este creată în Razor views, care face parte din design pattern-ul MVC, iar un view este, în sine, o pagină web ce are cerința de a prezenta datele utilizatorului. Un Razor View face parte din ASP.NET MVC și este un template HTML cu sintaxa de Razor (poate folosi C#). O acțiune a unui controller poate invoca să apară pe ecranul clientului.

În arhitectura curată, ea funcționează după urmatoarele criterii.

Prin separarea preocupărilor (separation of concerns) view-urilor Razor ajută la menținerea separării dintre straturi, contribuind la această ținându-se separat de stratul de logică și business. Acest principiu face proiectul mai ușor de testat și scalat.

Și, deoarece face parte din design pattern-ul Model-View-Controller(MVC), Razor view-urile se vor plasa într-un fișier cu numele controller-ului care le afișează clientului, în contextul în care acel controller e singurul care folosește acel Razor View. Dacă mai multe controller-ere îl folosesc sau este folosit înăuntrul altui view (adică e partial view), va fi plasat într-un fișier numit Shared. Toate aceste fișiere aparțin de un fișier numit Views.

Această împărțire ierarhică în fișiere urmează principul separation of concerns și contribuie enorm la design pattern-ul Model-View-Controller.

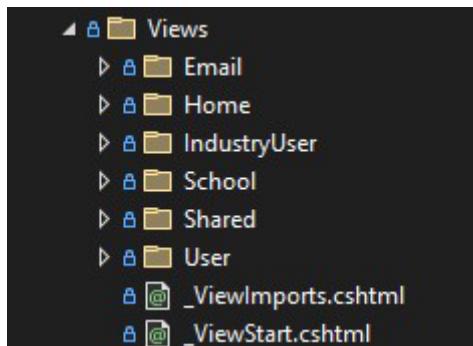


Figura 14: Ierarhia secțiunii vizuale

3.1.3 Baza de date

În această schemă se vor vedea tabelele și relațiile dintre ele.



Figura 15: Schema bazei de date

În mare parte, tabelele au chei străine către tabela User, iar tabela AbstractModel conține datele tuturor claselor ce au moștenit acest model abstract, atributul Discriminator fiind elementul ce le desparte când se face citirea din această tabelă.

Tabela Users conține toate datele comune tuturor utilizatorilor din platformă, indiferent de rolul lor în aceasta.

informațiile care se vor stoca despre o persoană sunt urmatoarele: un nume, un telefon, o adresă mail, un sex, iar, la alegere, pot să aibă o imagine de profil, o descriere, să se logheze cu un cont de Google dacă doresc, păstrându-se id-ul aceluia cont și, de când își creează contul, vor avea asignat rolul tipului de cont creat.

Se menționează faptul că dacă se șterge un utilizator, se vor șterge automat și toate datele lui din celelalte tabele, dacă există, deoarece, tabelele au fost setate ca, la ștergerea unui rând, să se aplice ON DELETE CASCADE.

Tabela IndustryUsers conține informațiile aferente utilizatorilor ce provin de la diverse companii și în ea se vor afla doar datele specifice acestui tip de utilizatori, restul putându-se extrage din tabela Users.

De aceea, în ea se vor găsi doar numele companiei de la care provin și tipul de serviciu prestat de companie (închiriat Apartamente, închiriat vehicule, transport mobilă, transport persoane, să te ajute să-ți găsești un job, etc). Iar id-ul de user este o cheie străină către tabela User pentru o eliminare rapidă din ambele tabele când e cazul.

Tabela SchoolUsers e destinată utilizatorilor ce încearcă să tot găsească oferte de a merge la școală pe diverse categorii. Fie pentru clasa primară, fie pentru clasa gimnazială, fie pentru liceu sau facultate.

Fiecare utilizator își alege în ce categorie poate găsi școli și să le pună ofertele pe platformă. De aceea, în această tabelă se află tipul de serviciu prestat, deoarece fiecare școală va avea un nume diferit pe platformă. Și, la fel, id-ul user-ului se află pentru a se aplica ON DELETE CASCADE fără probleme

Tabela Emails conține fiecare email trimis și primit de la fiecare client. Orice mail trebuie să aibă un titlu, un conținut, o dată când a fost emis sau primit, dacă a fost deschis sau nu și cine a fost creat (CreatorId) și către cine a fost trimis (UserId).

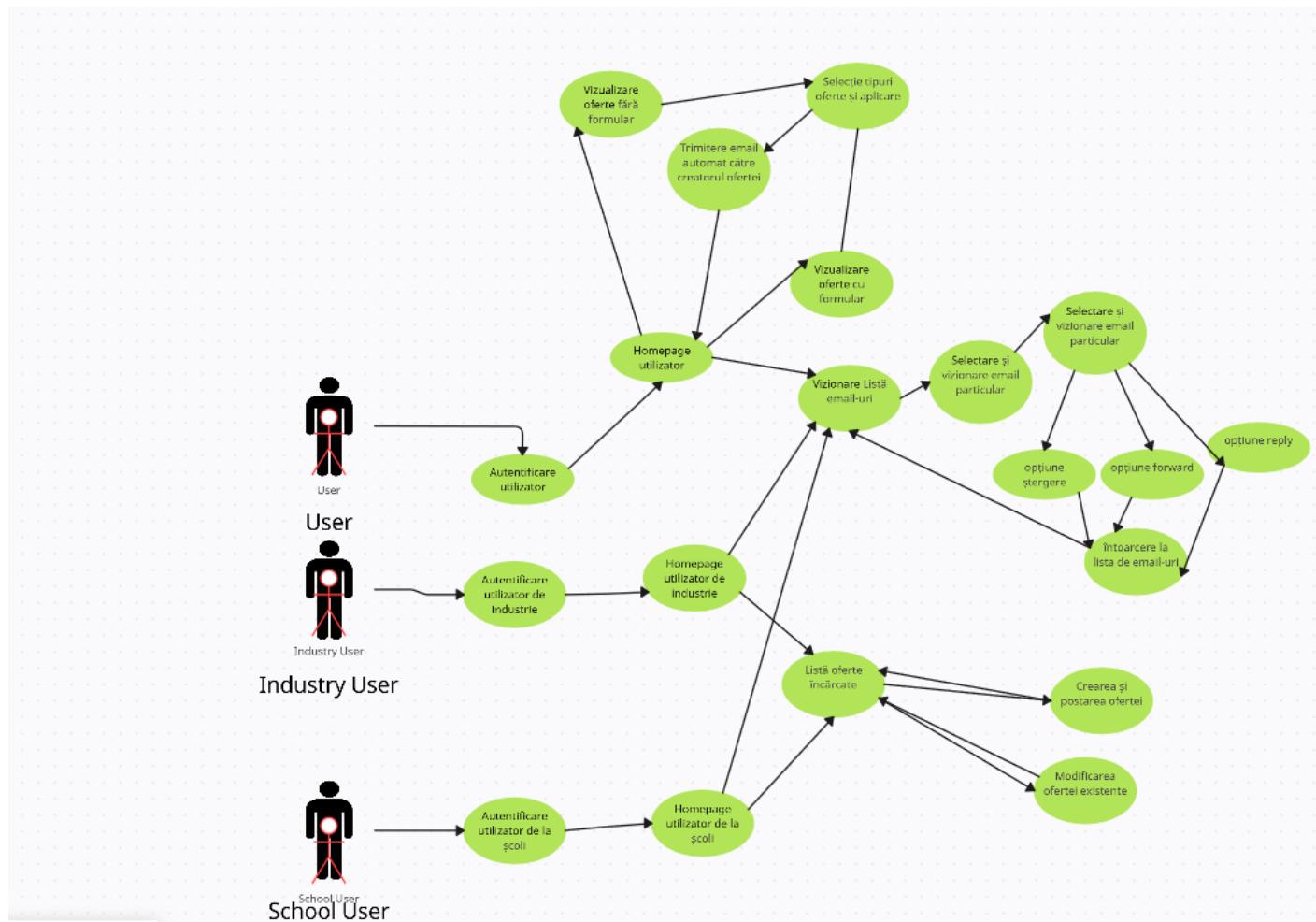
La nivel de cod, când se trimite un mail, ambii utilizatori vor avea o copie a aceluiasi mail, inversându-se doar id-urile între ele la a doua adăugare în baza de date. Aici, CreatorId este folosit pentru a se aplica eliminarea din baza de date când se face o ștergere în cascadă.

Și ultima tabelă, AbstractModel, reprezintă toate ofertele încărcate atât de utilizatorii din industrii ,cât și cei de la școli, ambele conțin aceleași caracteristici.

Toate ofertele au un nume , o descriere , ofertele de la industrie au preț, iar cele de la școli doar o secțiune anume,un link către site-ul companiei sau școlii cutare, în ce locație se află oferta, o imagine a ofertei sau a școlii , data când a fost încărcată pe platformă oferta, iar discriminatorul e doar la nivel de baze de date, creat de Entity Framework Core automat pentru a face distingerea între rândurile tabelei când se face citirea din baza de date și că obiectele folosite la crearea bazei de date, să poată primi automat datele lor.

3.2 Specificații funcționale

3.2.1 Actori



În această diagramă a aplicației, se evidențiază faptul că te poți autentifica fie ca un utilizator obișnuit, fie ca unul de la companii (din industrie), fie ca unul trimis de școli.

Utilizatorul obișnuit își face un cont pentru așa căuta serviciile necesare unei mutări în alt oraș, putând completa un formular pentru ce are el nevoie mai exact și se poate menționa faptul că este cu familia sau poate cu un copil mic de exemplu.

Un mail automat este trimis atunci când clientul alege să aplice la una din ofertele de care are nevoie, iar furnizorul ofertei poate lua legătura cu el.

Cât despre utilizatorii din industrie, cât și cei trimiși de școli, pot să își facă un cont destinat lor și să-și încarce ofertele lor.

Cei din corporații, ofertele valabile de la firma la care lucrează, iar cei de la școli, ce oferte prezintă școlile respective (adică dacă sunt dispuse să primească elevi sau studenți) din categoria aleasă de el.

3.2.2 Implementarea în .NET Core

Despre Unit of Work, ca să poate fi înțeles, trebuie mai întâi menționată ideea de repository pattern , deoarece se leagă mult de acest concept.

Un repositoriu este o clasă creată de dezvoltator destinată unei entități din proiect, ce detine toate operațiile posibile necesare pentru a fi entitate, în special operațiile de ștergere, inserare și modificare. Sunt 2 moduri de a implementa un repositoriu. Fie se face câte unul pentru fiecare entitate din proiect, fie se face unul care va fi folosit de toate.

Acum, înapoi la conceptul de unit of work, el este înțeles ca o singură tranzacție ce implică multiple operații de gestionare a bazelor de date și ,chiar , obiectele repositoriilor în sine.

Dacă se folosește acest principiu, orice acțiune de inserare,ștergere sau modificare a unui obiect mereu va folosi această clasă ca să acceseze obiectul reprezentatului necesar pentru a finaliza cu succes acțiunea. Cu alte cuvinte, se realizează totul într-o singură tranzacție, nu mai multe.

Astfel, se poate evita problema existenței obiectelor multiple de DbContext în aplicație, repositoriile nefiind nevoie să mai țină în ele câte un obiect al acelei clase.

Pot apărea probleme de sincronizare majore, deoarece fiecare obiect are istoricul lui propriu și baza de date poate deveni, până la urmă, inconsistentă din cauza acestei dileme.

```

public class UnitOfWork
{
    8 references
    public BookingRepo? Apartments { get; set; }
    8 references
    public FurnitureRepo? Furnitures { get; set; }
    8 references
    public JobRepo? Jobs { get; set; }
    8 references
    public RentingRepo? Vehicles { get; set; }
    8 references
    public TransportRepo? Transports { get; set; }
    7 references
    public SchoolRepo? Schools { get; set; }
    8 references
    public UserRepo? Users { get; set; }
    10 references
    public EmailRepo? Emails { get; set; }
    8 references
    public SchoolUserRepo? SchoolUsers { get; set; }
    13 references
    public IndustryUserRepo? IndustryUsers { get; set; }
    private readonly RelocationDbContext? _relocationDbContext;
    0 references
    public UnitOfWork(BookingRepo? br,FurnitureRepo? fr ,JobRepo? jr,RentingRepo? rp ,TransportRepo?tr,SchoolRepo? sr ,UserRepo? ur ,EmailRepo? er ,SchoolUserRepo? sur ,IndustryUserRepo? iur ,RelocationDbContext? rdc)
    {
        Apartments = br;
        Furnitures = fr;
        Jobs = jr;
        Vehicles = rp;
        Transports = tr;
        Schools = sr;
        Users = ur;
        Emails = er;
        SchoolUsers = sur;
        IndustryUsers = iur;
        _relocationDbContext = rdc;
        Apartments.InitialiseItems(_relocationDbContext);
        Furnitures.InitialiseItems(_relocationDbContext);
        Jobs.InitialiseItems(_relocationDbContext);
        Vehicles.InitialiseItems(_relocationDbContext);
        Transports.InitialiseItems(_relocationDbContext);
        Schools.InitialiseItems(_relocationDbContext);
        Users.InitialiseItems(_relocationDbContext);
        Emails.InitialiseItems(_relocationDbContext);
        SchoolUsers.InitialiseItems(_relocationDbContext);
        IndustryUsers.InitialiseItems(_relocationDbContext);
    }
}

```

Figura 17: Exemplu de clasă unit of work în platforma de realocare și servicii

Toate repositoriile publice ce fac parte din clasa Unit of Work au datoria de a fi o punte între datele oferite de clasele servicii și baza de date (în acest context repositoriile fiind strict operații de gestionare a tabelei cu același nume din bazele de date).

Constructorul acestei clase primește , prin injectarea dependintelor, repository-urile instantiatе ,cât și contextul actual al bazei de date, ce va rămâne singurul în toată aplicația, deoarece aşa funcționează acest design pattern cu unit of work, motiv pentru care fiecare repositoriu va primi contextul printr-o metodă comună ca fiecare să-și dețină DbSet-ul aferent lui, adică, ceea ce corespunde cu tabela lui din baza de date.

Implementarea repositoriilor este una, în mare parte comună, moștenind o clasa generică ce conține metode și atrbute similare.

```

public class BaseRepo<T> where T : BaseEntity
{
    private DbSet<T> _dbSet;
    private List<T> _items;
    0 references
    public BaseRepo()
    {
    }
    10 references
    public void InitialiseItems(RelocationDbContext context)
    {
        _dbSet = context.Set<T>();
        _items = _dbSet.Select(item => item).ToListAsync().Result;
    }
    11 references
    public async Task<T> Add(T item)
    {
        try
        {
            _dbSet.Add(item);
            _items.Add(item);
            await _dbSet.GetDbContext().SaveChangesAsync();
        }
        catch (Exception ex) { Console.WriteLine(ex.InnerException?.Message); }
        return item;
    }
    22 references
    public List<T> GetItems()
        => _items;
}

```

Figura 18: O parte din codul clasei BaseRepo

Aceasta e clasa generică care conține tabela, sub formă de cod, a bazei de date, de tipul T, care va fi decis când un repositoriu va moșteni această clasa, și o listă cu obiectele ce vor fi citite din tabela bazei de date.

Încă din clasa unit of work se putea observa că, după ce contextul bazei de date era primit, se pasa în fiecare repositoriu.

Aceasta a fost special făcută ca să se poate primi tabela aferentă fiecărei repository- ul în parte și să se poate aplica operațiile de stergere, modificare , inserare și citire în ea.

După finalizarea operațiilor, obiectul entității cheamă contextul din care provine pentru a salva complet modificările în baza de date.

Metoda aleasă pentru această facilitate este , de așa natură, încât nu se fac copii asupra contextului, tabela chemând exact acel context specific, care a trecut drept parametru, prin metoda specială de inițializare.

Prin urmare, se menține principiul clasei unit of work, adică, să existe un singur context în toată aplicația, toate depinzând de aceasta.

```
public class BaseEntity
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    65 references
    public int Id { get; set; } = default;
}
```

Figura 19: Clasa BaseEntity

În poza cu clasa generică, se poate observa că tipul T este o clasă de tipul BaseEntity, care conține drept proprietate un element numit ID, care are din start valoarea default, și s-a dat un atribut de cheie primară ca să-l recunoască Entity Framework Core-ul că este o cheie primară și toate tabelele ce conțin acest atribut trebuie să-l aibă configurat astfel.

Celălalt atribut, ce-i dă proprietatea de proprietate identity, face ca incrementarea id-ului să se facă automat în baza de date, nemaifiind nevoie să stea dezvoltatorul să țină cont de incrementarea id-ului când se fac inserări în baza de date.

Varianta code-first a fost introdusă în Entity Framework 4.1 și, în general, urmează design-ul după modele (domain driven design).

Sau, cu alte cuvinte, development-ul este concentrat pe partea de domeniu a aplicației (partea de back-end) și se fac și clase pentru entitățile domeniului înaintea creării bazei de date cu tabelele ei, iar clasele sunt obligate să corespundă cu tabelele pentru care le gestionează datele.

Entity Framework Core este capabil să creeze baza de date după clasele domeniului (entitățile) și după cum sunt configurate (cu restricții, chei strâine, etc). Deci, dacă se codează clasele în limbajul de programare C#, adică, un limbaj creat de Microsoft și folosit frecvent de aceștia, varianta code-first va funcționa aproape întotdeauna.

După crearea și configurarea claselor domeniului, se recurge la migrațiile din consolă pentru aplicarea lor pe baza de date, dacă există. Dacă nu există, o va crea cu configurațiile prescrise.

Se menționează că Entity Framework Core este calea recomandată de a crea o bază de date în .NET, în special faptului că permite lucrul cu bazele de date fie dacă s-a început dezvoltarea la nivel de back-end, fie la nivel de baze de date mai întâi și fiindcă permite interogări SQL scrise din cod sub formă de Language integrated query (LINQ).

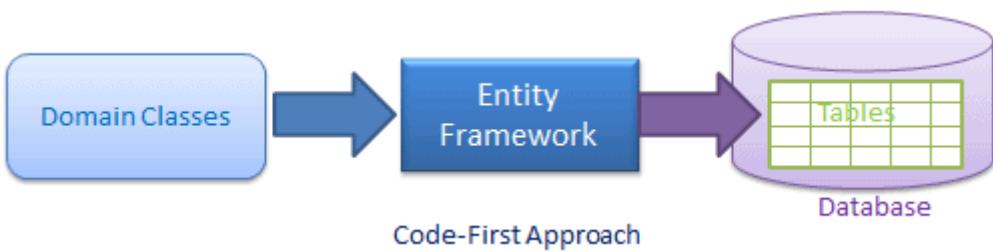


Figura 20: Schema code-first approach

Din figura de mai sus se observă mai clar legăturile dintre clasele entități, Entity Framework Core și tabelele generate de acest obiect relational mapper. Ele constituie esența tehnicii code-first approach în orice proiect în care se aplică aceasta.

```

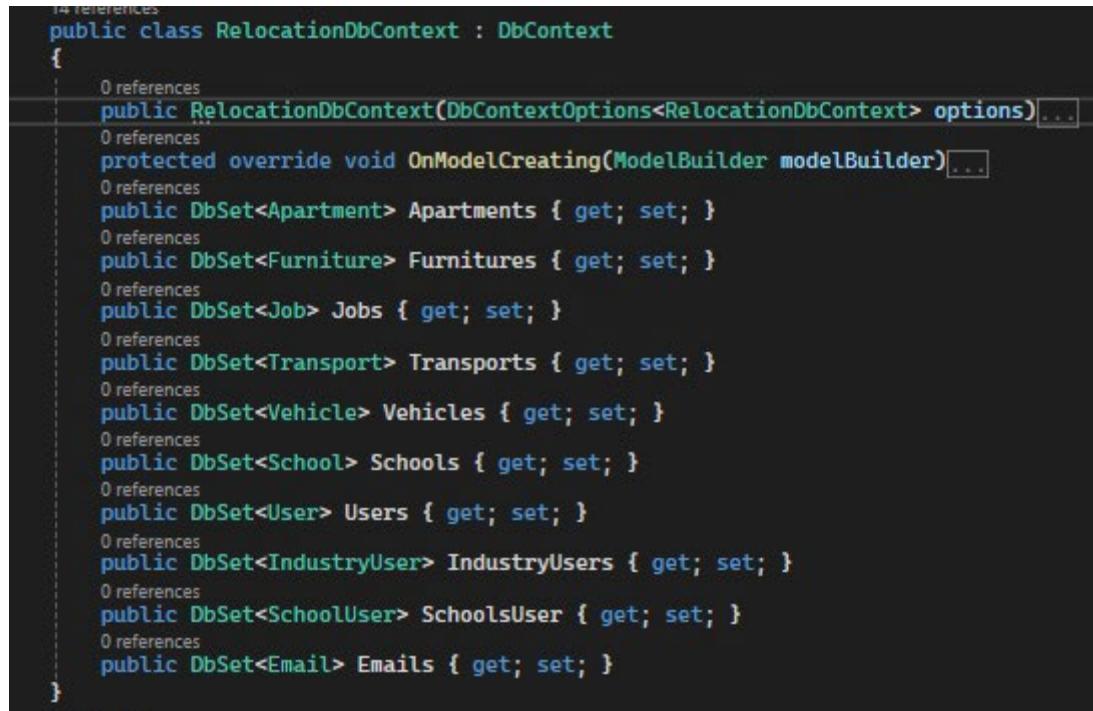
public class IndustryUser : BaseEntity
{
    [ForeignKey("User")]
    public int? UserId { get; set; }
    public string? CompanyName { get; set; }
    public int? ServiceType { get; set; }
}

```

Figura 21: Exemplu de clasă entitate

În această clasă se află proprietățile specifice ei și un id destinat obiectului de utilizator ce corespunde cu id-ul din obiectul care ar fi creat după această clasă, iar tag-ul [ForeignKey] are drept scop să arate că această proprietate are menirea să fie o cheie străină către tabela User, iar EF Core-ul să aibă la cunoștiință acest detaliu când creează sau modifică tabela User, cât și tabela IndustryUser.

Ea moștenește clasa BaseEntity, ca să se poate urma o structură ierarhică de moștenire în cadrul acestei aplicații, atât pentru o bună funcționare în folosirea EF Core-ului, dar și pentru o dezvoltare mai simplă, orice fiind mai ușor de modificat, dacă e nevoie, deoarece o moștenire ierarhică determină anumite proprietăți și facilități ce vor avea clasele derivate.



```
14 references
public class RelocationDbContext : DbContext
{
    0 references
    public RelocationDbContext(DbContextOptions<RelocationDbContext> options) ...
    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder) ...
    0 references
    public DbSet<Apartment> Apartments { get; set; }
    0 references
    public DbSet<Furniture> Furnitures { get; set; }
    0 references
    public DbSet<Job> Jobs { get; set; }
    0 references
    public DbSet<Transport> Transports { get; set; }
    0 references
    public DbSet<Vehicle> Vehicles { get; set; }
    0 references
    public DbSet<School> Schools { get; set; }
    0 references
    public DbSet<User> Users { get; set; }
    0 references
    public DbSet<IndustryUser> IndustryUsers { get; set; }
    0 references
    public DbSet<SchoolUser> SchoolsUser { get; set; }
    0 references
    public DbSet<Email> Emails { get; set; }
}
```

Figura 22: Clasa RelocationDbContext

În RelocationDbContext se regăsesc tabelele, la nivel de cod, cu entitățile aferente pentru crearea sau modificarea tabelelor printr-o migrație la nivelul bazei de date.

Incluzând toate aceste entități într-un DbSet înăuntrul acestei clase ce moștenește DbContext, care are drept scop instanțierea contextului de baze de date la nivel de cod și permiterea operațiunilor de CRUD tabelelor sale, se va putea dezvolta o bază de date ideală contextului utilizatorului.

Prin constructorul actual, va moșteni toate cele de trebuință din clasa DbContext pentru a putea fi recunoscută de EF Core că acesta este contextul actual al aplicației și doar în acesta o să se uite ce tabele conține și prin obiectele de acest tip se vor putea salva datele, indiferent de tipul operației, în baza de date către care este conectat.

```

0 references
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<AbstractModel>()
        .HasOne<User>()
        .WithMany()
        .HasForeignKey(e => e.CreatorId)
        .onDelete(DeleteBehavior.Cascade);
    modelBuilder.Entity<IndustryUser>()
        .HasOne<User>()
        .WithMany()
        .HasForeignKey(e => e.UserId)
        .onDelete(DeleteBehavior.Cascade);
    modelBuilder.Entity<SchoolUser>()
        .HasOne<User>()
        .WithMany()
        .HasForeignKey(e => e.UserId)
        .onDelete(DeleteBehavior.Cascade);
    modelBuilder.Entity<Email>()
        .HasOne<User>()
        .WithMany()
        .HasForeignKey(e => e.UserId)
        .onDelete(DeleteBehavior.Cascade);
}

```

Figura 23: Activarea posibilității de ON CASCADE DELETE

Metoda `OnModelCreating` provine din clasa `DbContext` și este suprascrisă, utilizând diferit obiectul `modelBuilder`, care permite alterarea relațiilor dintre entități, forma lor și cum vor fi mapate către baza de date.

În cazurile de mai sus, aceste clase au o relație de one to one către clasa `User`, iar clasa `User` are o relație de one to many către acestea, apoi se precizează în aceste clase că se află o cheie străină legată de tabela `User`. Astfel, la următoarea migrație, toate aceste entități vor avea o cheie străină către entitatea `User`.

După ce această restricție a fost impusă, se mai setează și un comportament specific acestui caz, în care, dacă un rând către tabela părinte (entitatea `User`) este șters din baza de date, atunci și toate rândurile din tabelele ce conțin o cheie străină către acel rând șters din tabela `User` să fie și ele șterse din tabelele lor.

ON DELETE CASCADE permite acest lucru la nivelul bazei de date și se va traduce astfel când se prelucrează operațiile acestea la acel nivel.

Pentru crearea bazei de date, a tuturor tabelelor și includerii datelor lor, se va folosi în linia de comandă în visual studio comanda `add-migration [nume_migratie]` pentru a putea începe o migrație și, astfel, se va putea vedea în fișierul aferent ei detaliile care au fost create doar la nivel de cod și se poate chiar modifica acel fișier, la nevoie.

Cât despre punerea în funcțiune a modificărilor aduse de migrația respectivă, se va folosi, tot în linia de comandă, comanda `update-database` pentru ca baza de date actuală să reflecte modificările migrației sau, dacă nu există, să fie creată de migrația curentă alături de detaliile tehnice aduse de aceasta.

Pentru trimitera de mail-uri pe conturile de google (în general) s-a folosit un third-party numit Sendgrid, ce este un email provider eficient pentru aproape orice tip de mail ce conține un domeniu cât de cât cunoscut (precum google, yahoo, etc).

Pachetul Sendgrid, care a fost adus în proiect prin NuGet, permite folosirea clasei `SendGridClient`, care necesită ca parametru doar cheia API-ului lor, care poate fi generat de orice utilizator cu un cont de Twilio.

Odată ce cheia este creată, este posibilă orice acțiune de reply, forward și de a compune un mail către celălalt utilizator din aplicație care deține o adresă de mail cu un domeniu valid și cât de cât cunoscut (de preferat gmail).

Sendgrid deține un mail transfer protocol proprietar, care este foarte avansat și este un third-party foarte popular printre furnizorii de mail-uri, motiv pentru care există o bună garanție că mail-ul respectiv va ajunge direct în căsuța poștală a celuilalt utilizator.

Înainte de a se aplica funcția `SendEmailAsync()`, trebuie să primească drept parametru, mail-ul în sine, care trebuie să conțină adresa expeditorului și adresa destinatarului, dar și numele lor, titlul mail-ului, conținutul mail-ului și dacă se dorește HTML pentru aspectul lui.

Prin `MailHelper.CreateSingleMail()` cu toate aceste obiecte, se va crea obiectul mail-ului ce trebuie trimis și se va păsa că parametru metodei menționate mai sus.

În aplicație s-a folosit și Twilio SMS, pentru trimitera unei parole noi prin SMS la numărul de telefon al utilizatorului. A fost setat ca doar numerele din România să fie cunoscute.

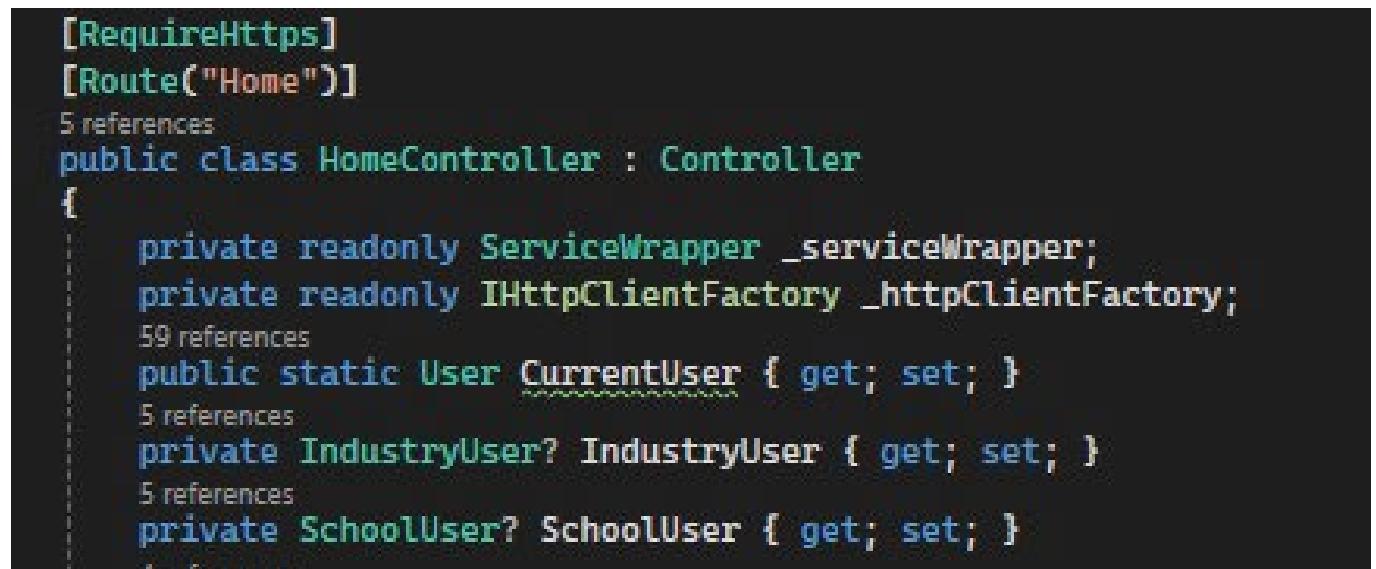
Pentru folosirea pachetului Twilio în ASP.NET Core, trebuie să furnizezi id-ul contului de pe site-ul Twilio și un `authToken` generat la crearea contului pentru a te putea loga în `TwilioClient` și să începi procesul de folosire a serviciilor Twilio prin `TwilioClient.Init()`.

Prin clasa CreateMessageOption se poate furniza numărul de telefon către care se trimise SMS-ul, iar celălalt trebuie să fie un telefon virtual aprobat de cei de la Twilio (deobicei furnizează unul gratuit pentru teste).

Parola aleatorie , în ambele cazuri, este create după un tipar anume. Primele 5 litere sunt de mâna, după 5 cifre, următoarele 5 litere sunt cu majuscule, după sunt iar 5 cifre.

Pentru siguranță ca mail-ul trimis prin Sendgrid să ajungă direct în căsuța poștală a clientului, nu în spam sau la gunoi, trebuie ca aplicația să fie securizată.

De aceea, toate conexiunile sunt forțate să fie HTTPS, folosind tag-ul [RequireHttps], în care, dacă cererea nu a venit prin HTTPS, este retrimisă din nou prin HTTPS automat.



```
[RequireHttps]
[Route("Home")]
5 references
public class HomeController : Controller
{
    private readonly ServiceWrapper _serviceWrapper;
    private readonly IHttpClientFactory _httpClientFactory;
    59 references
    public static User CurrentUser { get; set; }
    5 references
    private IndustryUser? IndustryUser { get; set; }
    5 references
    private SchoolUser? SchoolUser { get; set; }
    1 reference
}
```

Figura 24: Exemplu folosire [RequireHttps]

Controller-urile sunt cele care conțin metode cu rute, motiv pentru care ,dacă se pune acest tag pentru un controller, toate metodele lui vor avea această obligație. Obligația de a primi doar conexiuni venite prin HTTPS pentru o siguranță mai bună a transferului de date.

3.2.3 Implementarea în Razor Views

Prin aceste view-uri s-au putut forma toate structurile html, care au fost stilizare alături de bootstrap (în mare parte) și făcute responsive prin javascript pentru ca utilizatorul final să aibă parte de o experiență cât mai plăcută când folosește această aplicație web.

La fiecare navigare de pagină va fi un navbar care va avea câte un nav-item care îl va direcționa către ruta dorită din aplicație.

```
<ul class="navbar-nav d-none" id="industryDetails">
    <li class="nav-item">
        <a class="nav-link text-white" asp-controller="Home" asp-action="Homepage">Homepage</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-white" asp-controller="IndustryUser" asp-action="ServiceList">Your services</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-white" asp-controller="Email" asp-action="Emails">Emails</a>
        <span class="position-absolute top-1 start-99 translate-middle badge rounded-pill bg-danger" id="industryUserNewEmailNumber">
            </span>
    </li>
    <li class="nav-item">
        <a class="nav-link text-white" asp-controller="Home" asp-action="AboutUs">About Us</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-white" asp-controller="Home" asp-action="Logout">Sign out</a>
    </li>
</ul>
```

Figura 25: Exemplu de navbar in Razor Views

Înțial, acest navbar este invizibil utilizatorului, deoarece este dedicat clienților care s-au logat ca utilizatori din industrie, dar după logarea lor în contul cu acest specific, prin javascript, se va elimina elementul d-none din atributele structurii acesteia.

Fiecare nav-item duce către o metodă anume dintr-un controller anume, deoarece Razor Views este susținut și de Tag Helpers, făcuți de Microsoft pentru o mai simplă direcționare către back-end, metodele trebuind doar să aibă puse doar rutele deasupra lor.

Există 4 tipuri de navbar-uri. Unul este dedicat utilizatorilor neautentificati, având disponibile doar opțiunile de logare sau de creare cont, altul este pentru clienții obișnuiți, putând să se uite prin ofertele disponibile și pe mail-ul lor. Alt navbar este pentru utilizatorii de la școli, similar cu cel dedicat pentru cei proveniți din industrie și ultimul este că penultimul.

```

    T REFERENCE
    ↳ async function loggedUser() {
        let option = getTheRole()
        option = await getTheRole(option);
        let crucialDetails = document.getElementById("crucialDetails");
        let userDetails = document.getElementById('userDetails');
        let industryDetails = document.getElementById('industryDetails');
        let schoolUserDetails = document.getElementById('schoolUserDetails');
        switch (option)...
        KeepPicture();
        getNewEmails();
    }

```

Figura 26: Exemplu funcție din javascript

Această metodă este plasată într-un fișier javascript, toate metodele din acest fișier putând fi targetate de orice structură din orice view din proiect, indiferent de nivelul din ierarhie.

Ea se ocupă de vizibilitatea navbar-ului corect, punând elementul d-none în clasa structurii navbar-ului care nu e dedicat contului respectiv și eliminând d-none (dacă există) din clasa navbar-ului ideal pentru contul curent.

```

<div class="modal fade" id="forgotPasswordModal">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <h6 class="modal-title">Enter your name and your gmail and you will receive a new password</h6>
            </div>
            <form asp-controller="Home" asp-action="ForgotPassword" method="post">
                @Html.AntiForgeryToken()
                <br>
                <div class="form-floating d-flex justify-content-center">
                    <input class="form-control w-75" id="nameInput" name="name" />
                    <label class="form-label" for="nameInput">Name</label>
                </div>
                <br>
                <div class="form-floating d-flex justify-content-center">
                    <input class="form-control w-75" id="emailInput" name="email" />
                    <label class="form-label" for="emailInput">Email</label>
                </div>
                <div class="modal-footer">
                    <button class="btn btn-primary" type="submit">Submit</button>
                    <a data-bs-toggle="modal" data-bs-target="#recoverWithPhone" class="button">Recover with phone</a>
                </div>
            </form>
        </div>
    </div>

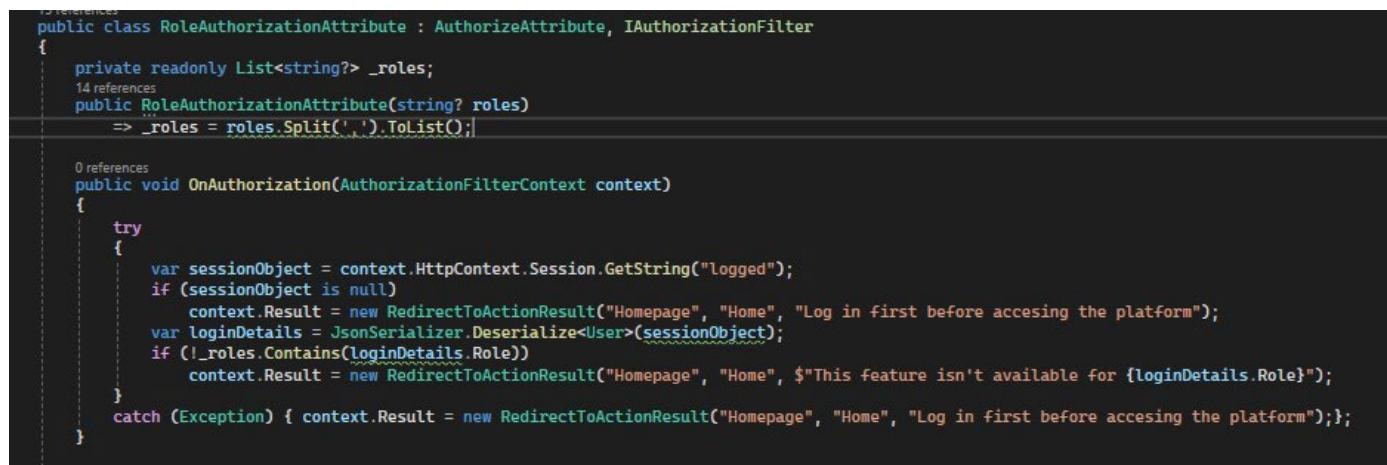
```

Figura 27: Exemplu de modal cu formular

Un formular din HTML are drept scop transmiterea unor date de la client către server, motiv pentru care tipul conexiunii va fi de tipul POST, iar metoda, în back-end va accepta doar cereri de tipul [HttpPost] și, ca o măsură în plus de siguranță, se va transmite un AntiforgeryToken către acea metodă în back-end , care o va valida doar dacă are deasupra sa tag-ul [ValidateAntiForgeryToken]. După ce clientul își introduce datele, poate să își primească nouă parolă de 20 de caractere pe adresa sa de mail legată de contul lui existent.

3.2.4 Securitate

Pe lângă faptul că toate conexiunile sunt forțate să se facă prin HTTPS și că la fiecare formular se va trimite mereu un Antiforgery Token pentru a opri request-urile false de tipul cross-site (astfel, nimeni nu va putea să încerce să păcălească utilizatorii normali ca să trimită cereri către server care nu le-au intenționat să le facă), mai este și un filtru făcut manual, bazat pe rolurile disponibile care le poate avea un utilizator pe platformă.



```
public class RoleAuthorizationAttribute : AuthorizeAttribute, IAuthorizationFilter
{
    private readonly List<string> _roles;
    public RoleAuthorizationAttribute(string? roles)
        => _roles = roles.Split(',').ToList();
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        try
        {
            var sessionObject = context.HttpContext.Session.GetString("logged");
            if (sessionObject is null)
                context.Result = new RedirectToActionResult("Homepage", "Home", "Log in first before accesing the platform");
            var loginDetails = JsonSerializer.Deserialize<User>(sessionObject);
            if (!_roles.Contains(loginDetails.Role))
                context.Result = new RedirectToActionResult("Homepage", "Home", $"This feature isn't available for {loginDetails.Role}");
        }
        catch (Exception) { context.Result = new RedirectToActionResult("Homepage", "Home", "Log in first before accesing the platform"); }
    }
}
```

Figura 28: Filtrul de securitate

Atunci când un utilizator intră pe contul său sau își creează unul nou, își atribuți și rolul aferent, iar obiectul instanțiat va fi adăugat în sesiunea curentă, care va fi accesată la fiecare intrare o metodă cu o rută (precum cele arătate în imaginile anterioare), care are și acest atribut menționat, obligatoriu având drept parametru rolul sau rolurile acceptate pe această facilitate restricționată.

Fiindcă această clasă moștenește AuthorizeAttribute, ea poate fi plasată exact precum tag-urile deasupra metodelor și controller-elor, și fiindcă moștenește și IAuthorizationFilter, este recunoscută, la nivel intern, ca o clasă ce se ocupă de autorizare, ASP.NET Core-ul ocupându-se de restul.

```

builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromMinutes(15);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
    options.Cookie.MaxAge = TimeSpan.FromMinutes(30);
});

```

Figura 29: Detaliile sesiunii

Și în configurarea sesiunii aplicației s-au luat câteva măsuri de securitate, nepermittându-se inactivitatea mai mult de 15 minute și nici o sesiune să dureze mai mult de 30 minute. La finalul oricărui timp dintre acestea două, utilizatorul va fi delegat și trimis înapoi la pagină inițială de start, putând să se logheze înapoi. Se menționează faptul că atacurile de tipul XSS, care ar fi scripturi venite din partea clientului către server sunt blocate, deoarece opțiunea `options.Cookie.HttpOnly` blochează această posibilitate de a ataca server-ul.

```

builder.Services.AddRateLimiter(options =>
{
    options.GlobalLimiter = PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
        RateLimitPartition.GetSlidingWindowLimiter(
            partitionKey: httpContext.User.Identity?.Name ?? httpContext.Request.Headers.Host.ToString(),
            factory: partition => new SlidingWindowRateLimiterOptions
            {
                AutoReplenishment = true,
                PermitLimit = 80,
                QueueLimit = 20,
                Window = TimeSpan.FromMinutes(1),
                SegmentsPerWindow = 4
            }));
    options.OnRejected = (context, cancellationToken) =>
    {
        if (context.Lease.TryGetMetadata(MetadataName.RetryAfter, out var retryAfter))
        {
            context.HttpContext.Response.Headers.RetryAfter = retryAfter.TotalSeconds.ToString();
        }

        context.HttpContext.Response.StatusCode = StatusCodes.Status429TooManyRequests;
        context.HttpContext.Response.WriteAsync("Too many requests. Please try again later.");
        return new ValueTask();
    };
});

```

Figura 30: Rate limiter

Rolul unui rate limiter este de a bloca spam-ul de rute. De exemplu, dacă un utilizator ar spama diverse rute 80 de ori, acumulându-se până la 100, totul într-un singur minut, atunci utilizatorul ar fi delegat din cont și trimis înapoi la pagina principală.

Acest limitator este setat global, ca această măsură de siguranță anti-spam să se aplique la absolut toate rutele posibile din aplicație. și nu contează dacă utilizatorul este logat sau nu, se pot aplica limitele și pe gazda aplicației pentru a putea opri orice fel de atac spam.

Acest rate limiter este împărțit în 4 ferestre pentru a optimiza puțin procesul, fiecare fereastră urmărind, timp de 15 secunde și putând lua maxim 20 de cereri, sirul rutelor folosite în timpul aferent. Mereu se vor refa singure, deoarece opțiunea de autoreplenish este mereu activată.

Dacă se ajunge în cazul în care utilizatorului să i se respingă request-ul către server, se poate încerca să vadă dacă există o opțiune să încearcă din nou să mai folosească aplicația, dacă nu, va trebui să reentre pe pagina principală.

Atunci când se creează tradițional un cont în aplicație (fără google), parola trebuie criptată înainte de a se transfera datele în baza de date pentru a preveni eventuale furturi de parole în cazul unui data breach de exemplu.

Pentru aceea, s-a folosit algoritmul bcrypt pentru criptarea parolelor, adus de pachetul Bcrypt.Net în C# prin Nuget, care aplică o funcție de hashing asupra parolei, nemaiînd posibilă reîntoarcerea la versiunea inițială.

Mereu când un utilizator se loghează, parola introdusă de el va fi și ea trecută într-o metodă de hash, și se va compara salt-ul ei cu cel al parolei inițiale criptate, căci de la ambele versiuni se pot extrage un salt, care se poate compara.

Acest salt este o grupare mică aleatoare de date, care se introduce în hash-ul rezultat (prin el se blochează atacurile de tipul rainbow table și brute-force).

Un avantaj, din punct de vedere al siguranței utilizatorului, este că Bcrypt are și un coeficient de cost (cost factor), care determină numărul de iteratii și câte hash-uri trebuie aplicate până la forma criptată finală.

Acest factor indică și cât timp durează și câte resurse se folosesc în proces, el putând fi setat la funcția de criptare de programator. Astfel, se poate decide cum se preferă criptarea în sine, fie se poate axa pe siguranța utilizatorilor, dar la prețul unui cost de resurse mai ridicat și un cost de timp mai mare, fie invers.

În aplicație, acest factor a fost setat la 14 pentru o siguranță mai sporită decât cea recomandată, 13, care este mijloc.

Capitolul 4

Ghid de utilizare al aplicației

4.1 Prezentare

Aplicația aceasta de servicii de realocare și mutare în alt oraș servește ca un intermediar între utilizatorii ce doresc să își caute serviciile necesare unei realocații cu toate condițiile de care au nevoie, totul fiind la un click distanță pe această platformă web. Utilizatorii normali își pot crea cont și să aplique la orice fel de oferte au nevoie, iar utilizatorii care oferă servicii își pot posta orice serviciu pe platformă ca să poată fi selectat de cine are nevoie din clientelă. Se includ funcționalități precum afișarea ofertelor de oricare tip, trimiterea unui mail automatizat către creatorul ofertei, un sistem de mail pentru gestionarea mail-urilor individuale și trimiterea unui mail altor persoane, trimiterea unui mail pe contul de gmail personal, recuperarea parolei fie printr-o adresa de mail, fie prin SMS, filtrarea ofertelor după oraș, filtrarea email-urilor după data postării, încărcarea de servicii pe secțiunea pe care și-au înregistrat contul respectiv, etc.

4.2 Partea de autentificare



Figura 31: Pagina de bun venit

Utilizatorul nelogat o să fie trimis către o pagină de bun venit în aplicație și va avea 2 opțiuni. Să se logheze în cont sau să creeze un cont, în ambele cazuri o poate face tradițional sau prin logarea în contul de google, dacă utilizatorul dorește.

The screenshot shows a 'Create Account' form. At the top, there are four input fields labeled 'Name:', 'Password:', 'Email:', and 'Phone:'. Below these is a section for selecting user type with radio buttons: 'User', 'Industry User', 'School User', and 'Male' (which is selected). There is also a 'Female' and 'Other' option. A 'Browse...' button with the message 'No file selected.' is present. To the right of the form is a teal-colored user icon. At the bottom are two buttons: a blue 'Create Account' button and a white 'Try with Google' button with a blue outline.

Figura 32: Crearea contului

Pentru a crea un cont, trebuie să introduci numele utilizatorului, parola dorită, o adresă de e-mail și neapărat un număr de telefon, tipul contului de utilizator dorit.

Dacă se va alege tipul industry user, atunci vor trebui să introduci și numele companiei de la care provine utilizatorul, alături de tipul de industrie din care face parte.

Dacă se selectează tipul school user, atunci trebuie să precizezi doar tipul de școli din care se dorește să se posteze oferte (fie doar de școală primară, sau gimnazială, sau de universitate, etc).

Este bine să se știe sexul utilizatorului pentru a știi cui te adresezi, motiv pentru care este o opțiune obligatorie și, în final, utilizatorul poate să își aleagă o imagine de profil după bunul plac, putând să o schimbe în pagina de profil, dacă este nevoie.

Mai există și o opțiunea de ați creea contul cu google direct, fiind necesar doar numărul de telefon, tipul contului de utilizator, sexul utilizatorului și, eventual, detaliile aferente tipurilor de conturi de utilizatori de industrie sau de școli.

The image shows a login interface. On the left, the text "Log in" is displayed above two input fields: "Name" and "Password". Below these fields are three buttons: a blue "Submit" button, a "Google Login" button featuring the Google logo, and a blue link "Forgot Password".

Figura 33: Logarea

Trebuie să introducă doar numele utilizatorului și parola sa, care sunt atașate unui cont deja existent în aplicație. Dacă utilizatorul are atașat un cont de google la contul de pe platformă, se poate loga direct cu contul de google, fără a mai fi nevoie de a se loga tradițional.

The image shows a form for recovering a password. It includes a descriptive text: "Enter your name and your email address and you will receive a new password". Below this are two input fields: "Name" and "Email". At the bottom are two buttons: a blue "Submit" button and a blue link "Recover with phone".

Figura 34: Recuperare parolă prin email

În cazul în care utilizatorul nu-și mai amintește parola, din oricare motiv, poate să folosească optinea de Forgot Password și să-și scrie numele contului existent, alături de adresa de email atașată contului de utilizator și va primi un mail cu nouă să parolă în căsuța poștală. Utilizatorul își poate schimba noua parolă în alta din secțiunea de profil, dacă nu-i place.

Enter your name and your phone number

Name

Phone

Submit

Figura 35: Recuperare parolă cu telefonul prin SMS

Dacă se întâmplă ca utilizatorului să nu îi fie fiabilă varianta de recuperare a parolei prin adresa de mail, poate să încerce și varianta cu trimiterea unui SMS cu noua parolă formată pe numărul de telefon folosit și atașat de contul utilizatorului, în speranța că această metodă îi va putea furniza noua parolă cu succes, astfel încât să nu-și piardă contul.

Au fost gândite destule soluții pentru ca orice utilizator al acestei aplicații web să poate să își creeze un cont cu ușurință, fie tradițional, fie prin Google.

Să se logheze în aplicație cu contul aferent lui, fie tradițional, fie prin Google, iar recuperarea parolei este mult mai posibilă, deoarece există 2 moduri de a o putea recupera, fie printr-o adresă de email, fie prin telefonul mobil, cu condiția ca ambele să fie atașate contului existent.

Posibilitatea de a exista orice impersonare este, de asemenea, foarte redusă, deoarece aceste date sunt valabile doar utilizatorului și nimeni altcineva nu poate să-i acceseze căsuța poștală de pe adresa lui de mail, sau telefonul mobil.

4.3

Pagina principală post-autentificare

După autentificarea utilizatorului în cont, există 3 pagini de post-autentificare, fiecare pentru tipul de cont al utilizatorului curent. Astfel, toți utilizatorii pot beneficia de facilitățile contului la fiecare pas în aplicație.



Figura 36: Opțiunile utilizatorului normal

Un utilizator normal beneficiază de posibilitatea de a vizualiza și chiar aplica la toate ofertele încărcate pe platformă, motiv pentru care, în bara lui de căutare o să existe 6 opțiuni de vizualizare de oferte, 5 provenite din industrii, iar una pentru tipurile de școli.

Există și opțiunea de mail-uri, pentru a le putea viziona și gestiona după bunul plac, opțiunea de a te întoarce la pagina principală de post autentificare, opțiunea de a completa un formular, care e specifică doar acestui tip de cont, în care clientul precizează de ce servicii are nevoie, dacă e student sau adult și alte detalii.

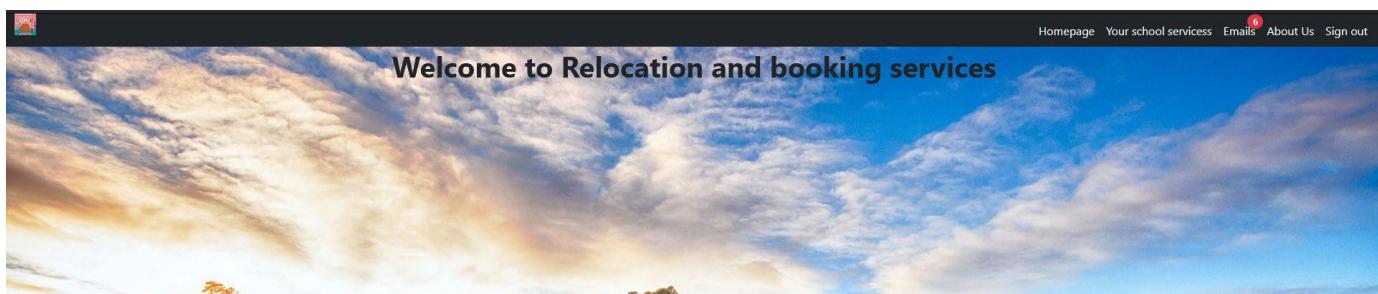


Figura 37: Opțiunile utilizatorului din industrie

Când un utilizator din industrie va intra în contul său, el va avea o pagină de post autentificare specifică lui, motiv pentru care în bara de navigare se vor afla doar opțiunile de returnare la pagina lui de start, la vizualizarea ofertelor sale încărcate, unde le poate gestiona cum dorește, la vizualizarea și gestionarea mail-urilor, la pagina de About Us, care povestește despre serviciile platformei (e valabilă la toți utilizatorii logați) și opțiunea de delogare din cont.

Cât despre utilizatorul din școli, el va avea la fel o pagină personalizată de post autentificare, care se asemănă destul de mult cu cea a utilizatorului din industrie, doar că ofertele ce le va putea vizualiza și gestiona au subiect toate tipurile de școli în loc de ofertele valabile în industrie.

4.4 Pagina de profil a utilizatorului

The screenshot shows a user profile page with the following fields:

- Name: Mario
- Email: mario@luigi.com
- Phone: 483949534
- Gender: Male
- Self Description: (empty text area)
- File Upload: Browse... No file selected.

At the bottom, there are four buttons: Submit update (green), Delete Account (red), Change password (blue), and Sync in your google account (blue).

Figura 38: Pagina de profil

Înțial, la sosirea paginii de profil, toate câmpurile vor fi dezactivate, ele fiind doar pentru citire, iar, la apăsarea butonului Update, câmpurile vor fi deschise pentru orice fel de modificare. Numele poate fi schimbat (cât timp nu este folosit noul nume), adresa de mail, telefonul, până și sexul, poza de profil și e posibilă și adăugarea unei descrieri mici.

Mai există și posibilitatea de a se asocia un cont de gmail la contul curent și chiar și desocierea contului curent de gmail de contul curent. Stergerea contului se poate realiza dacă se dorește.

Utilizatorii din industrii au tot ce conține pagina de profil al unui utilizator normal și încă 2 câmpuri, unul pentru compania din care fac parte și unul pentru tipul industriei pe care își postează ofertele. Iar utilizatorii de la școli au și ei la tot ce conține pagina utilizatorului, cu un câmp în plus care indică pe ce tip de școli își postează ofertele. Aceste câmpuri extra pot fi modificate.

The screenshot shows a user interface for changing a password. At the top, the title "Change password" is displayed. Below it are three input fields: "Old Password", "New Password", and "Retype Password". To the right of these fields is a blue "Submit" button. The entire form is set against a white background with thin gray horizontal lines separating the sections.

Figura 39: Schimbarea parolei

Această opțiune este valabilă tuturor utilizatorilor care și-au creat un cont și tot ce trebuie să facă este de a introduce parola veche, parola nouă și a rescrie din nou nouă parola ca să fie siguri că au scris această parolă bine și să se evite neplăcarea de a fi nevoiți să primească o nouă parolă automată prin email sau SMS.

Toată secțiunea cu pagina de profil cuprinde multe facilități importante, mai ales schimbarea parolei și asocierea sau desocierea contului de Google de contul curent, motiv pentru care această secțiune este considerată importantă pentru toți utilizatorii platformei. Este recomandat să se modifice cu grijă datele contului, fiindcă e posibil să eșueze modificările dacă nu se respectă condițiile pentru fiecare în parte.

Ștergerea contului trebuie să fie mereu o opțiune lăsată tuturor utilizatorilor cu un cont creat pe platformă, în caz că nu mai doresc să folosească serviciile oferite sau să încarce ei servicii de categoria lor pe aplicația web ca să poată fi vizualitate și dorite de clienții aplicației. Conturile fără activitate nu au rost să existe în baza de date, deoarece consumă memorie, iar această chestiune poate fi costisitoare.

4.5

Paginile cu oferte pentru utilizator

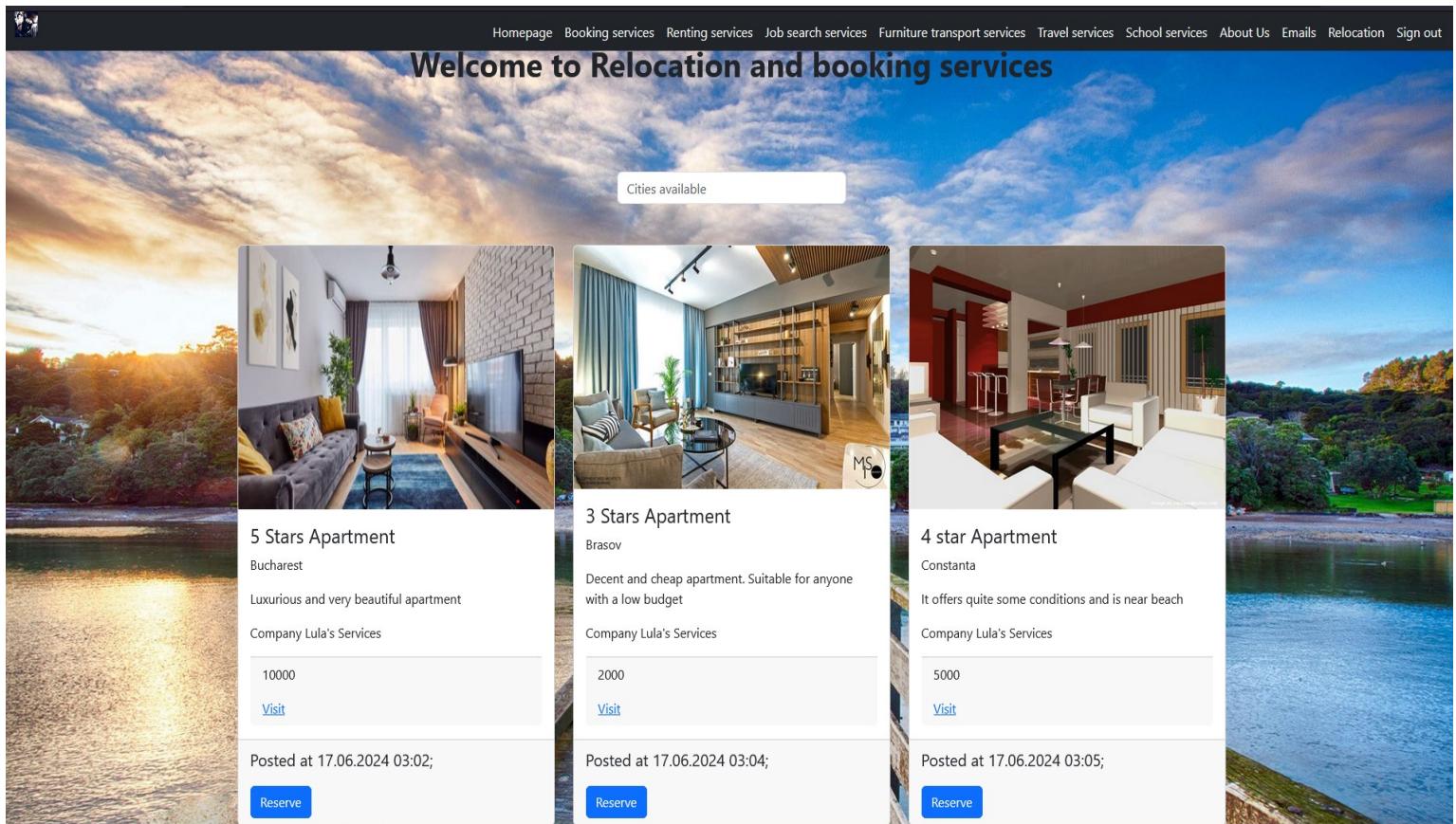


Figura 40: Pagina cu oferte de apartamente

Orice utilizator poate să vizualizeze oferte din industrie precum cele de închirieri de apartamente, de închirieri vehicule, oferte de job-uri, servicii de transport mobilă și servicii de transport persoane, iar din ofertele pentru școli, se pot alege școli de la grădiniță până la universități la care se pot aplica.

Toate acestea au același format de ofertă, care precizează numele ofertei, descrierea, dată la care a fost postată oferta, numele companiei sau al școlii (după caz), prețul (dacă este cazul) și site-ul către compania sau școala către care a postată oferta.

Ofertele pot fi filtrare după orașe, dacă utilizatorul dorește să caute oferte în orașul în care mută, de exemplu.

4.6

Pagina cu lista de email-uri

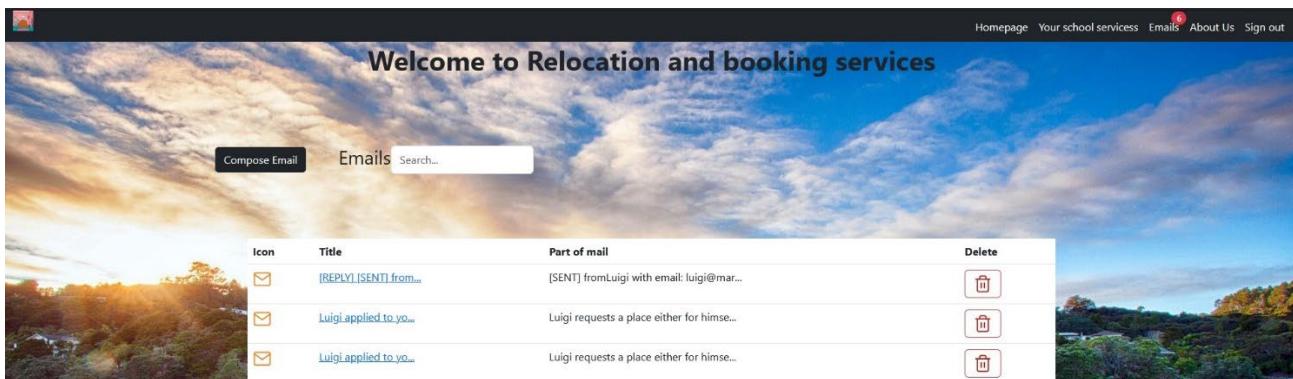


Figura 41: Pagina cu lista de mail-uri

Orice utilizator logat pe contul său poate să își acceseze mail-urile și să le vadă și să le gestioneze cum dorește în această secțiune dedicată mail-urilor personale.

Ele pot fi șterse direct din lista de mail-uri apăsând pe iconița de ștergere și este posibilă vizualizarea mail-ului individual dacă se apasă pe numele mail-ului respectiv.

Tot în această secțiune este posibilă și direcționarea către secțiunea care permite compunerea unui mail și căutarea de mail-uri în lista de mail-uri după un șir de caractere dat de client în bara de căutare mică deasupra listei de mail-uri.

Lista se modifică la fiecare schimbare de caracter în șirul de caractere din bara de căutare.

Această secțiune cuprinde doar esența mail-urilor, deoarece se menționează o parte din titlul ei și o parte din conținutul ei, restul detaliilor putând fi vizionate în întregime când se deschide mail-ul.

Astfel, se evită lărgirea rândurilor tableei cu lista de mail-uri până la niște dimensiuni neplăcute la vedere și experiențafolosirii platfomei rămâne și plăcută și cât de cât optimă pentru client, că nu trebuie să se complice să caute mail-uri printre un șir neechilibrat și să dea scroll în jos până îl găsește ca să facă ce nevoie are cu acel mail respectiv, ci poate să vadă niște mail-uri cu un aspect ordonat și câteva detalii menționate pentru a-l informa despre ce este acel mail și poate să-l caute după un string anume (numele mail-ului) ca să-l găsească instant.

4.7

Pagina cu vizualizarea email-ului

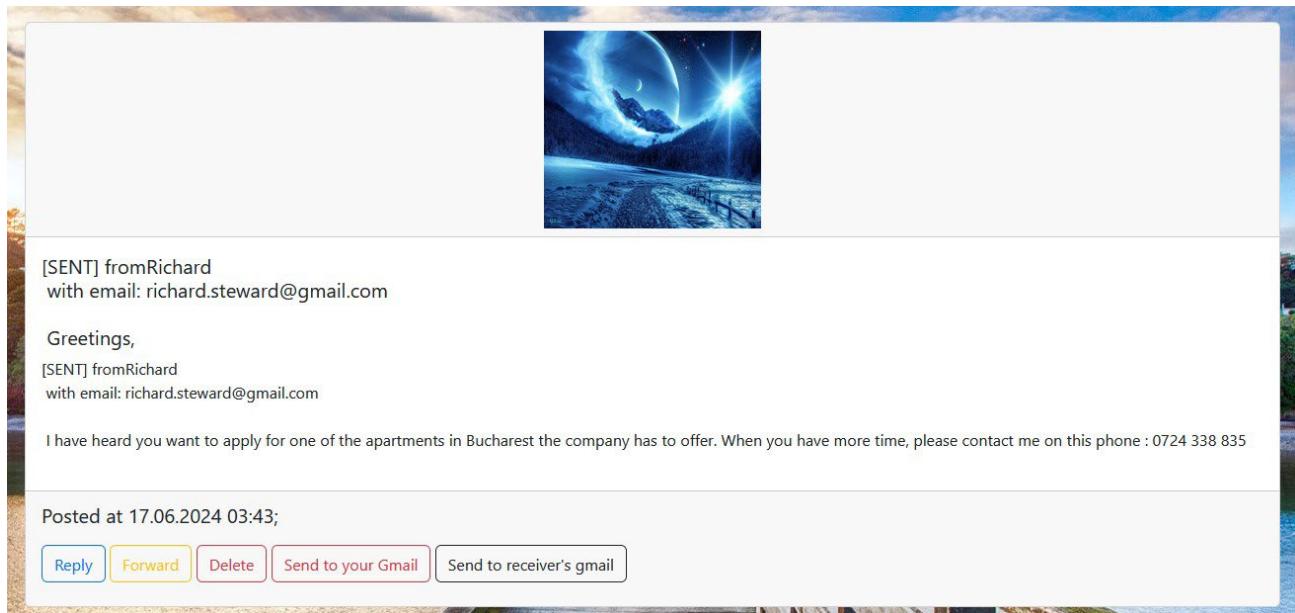


Figura 42: Vizualizarea mail-ului

În această secțiune se poate vizualiza individual fiecare mail în parte și se pot vedea poza de profil a celui care a trimis mail-ul respectiv, titlul și conținutul mail-ului.

Destinatarul poate să răspundă la mail cu butonul de reply, unde i se deschide mai jos o opțiune de a scrie un mesaj rapid către cel care a trimis mail-ul.

Sau poate să dea forward la alții utilizatori la mail-ul primit (se pune în practică similar că la opțiunea de reply) sau poate să fie șters direct.

Dacă utilizatorul curent are asociat un cont de Google, își poate redirecționa mail-ul pe adresa sa de Gmail ca să-l păstreze acolo, iar dacă cel care a trimis mail-ul deține un cont de Google asociat pe platforma, destinatarul poate să-i răspundă direct pe adresa principală de mail a emițătorului.

4.8

Pagina cu compunerea unui email

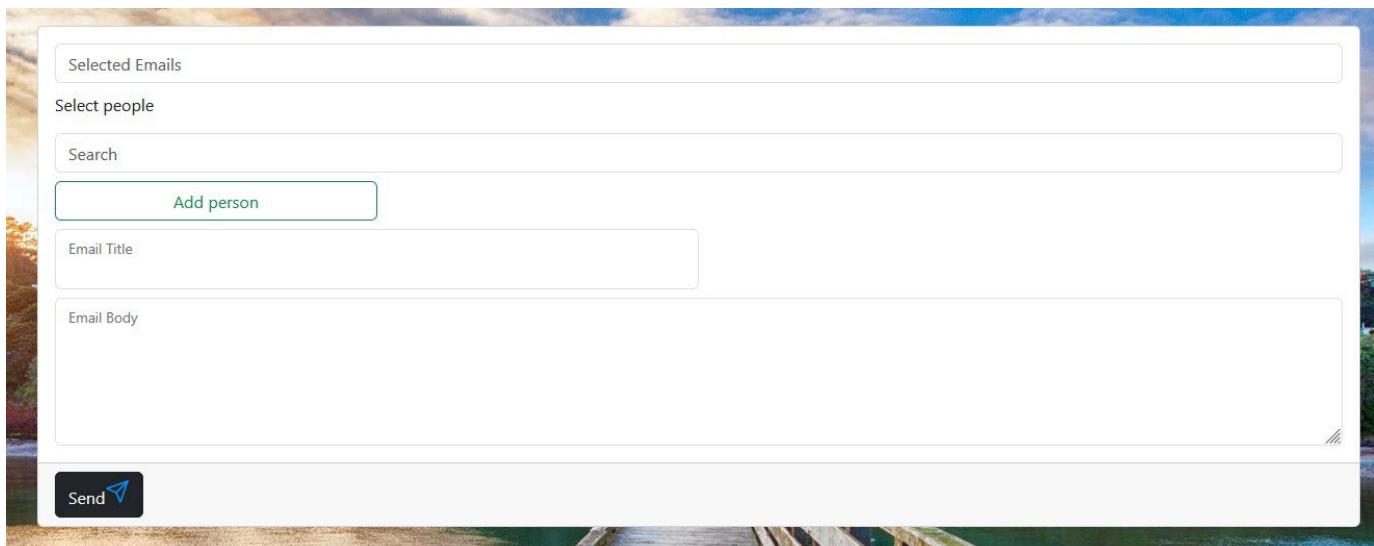


Figura 43: Compunerea unui mail

În această secțiune se poate compune un mail , compunându-se titlul lui și conținutul lui, ca mai apoi să se selecteze persoanele către care se va trimite acest mail.

Pentru selectarea persoanelor, se va scrie adresa de mail a respectivului (la fiecare caracter introdus apare o listă cu adresele de mail care conțin string-ul introdus , deci se poate găsi relativ ușor) și se adaugă la secțiunea cu persoanele selectate.

După ce toate persoanele au fost selectate se poate apăsa butonul Send ca să se și trimită la toți cei care au fost selectați drept destinatari pentru acest mail.

Mail-urile sunt ,pentru această aplicație, importante, deoarece prin mail-uri se notifică utilizatorii ce pot posta oferte că cineva a aplicat la serviciul oferit de compania sau școala pentru care postează oferta, iar acele mail-uri conțin și numerele de telefon ale clienților în nevoie, simplificându-se comunicarea dintre cei 2, deoarece pot oricând muta conversația la telefon pentru toate detaliile care trebuie discutate.

Tot prin mail-uri se poate face o comunicare între 2 sau mai mulți utilizatori pentru a le transmite niște informații anume, poate precum un anunț care-l face un utilizator către un alt utilizator sau mai mulți utilizatori.

4.9

Pagina cu ofertele poste

The screenshot shows a user interface for posting offers. At the top right are navigation links: Homepage, Your services, Emails, About Us, and Sign out. A red notification bubble with the number '1' is visible next to the 'Emails' link. The main content area has a blue header 'Welcome to Relocation and booking services'. A large blue button on the left says 'Add Offer'. Below it are three apartment listings:

- 5 Stars Apartment**
Luxurious and very beautiful apartment
Company: Lula's Services
Location: Bucharest
Price: 10000
[Visit](#)
[Update](#) [Delete](#)
Posted on: 17.06.2024;
At 03.02.47;
- 3 Stars Apartment**
Decent and cheap apartment. Suitable for anyone with a low budget
Company: Lula's Services
Location: Brasov
Price: 2000
[Visit](#)
[Update](#) [Delete](#)
Posted on: 17.06.2024;
At 03.04.10;
- 4 star Apartment**
It offers quite some conditions and is near beach
Company: Lula's Services
Location: Constanta
Price: 5000
[Visit](#)
[Update](#) [Delete](#)
Posted on: 17.06.2024;
At 03.05.02;

Figura 44: Pagina cu ofertele poste

În această secțiune, utilizatorul din industrie poate să își vadă toate ofertele care le-a postat pentru compania de la care provine, și este capabil să adauge noi oferte, să le modifice sau să le șteargă de tot.

O oferta cuprinde poza cu serviciul sau produsul în sine, numele serviciului sau produsului, o scurtă descriere despre el, numele companiei, locația din care se poate beneficia de acesta, prețul lui (dacă e cazul) și site-ul companiei sau al școlii în numele căreia s-a postat oferta și, desigur, data și ora la care a fost postat.

Se menționează faptul că secțiunea de vizualizare a propriilor oferte încărcate de utilizatorii de școli este aproape identică cu secțiunea de oferte poste de utilizatorii din industrie, cu mențiunea că la utilizatorii de la școli nu va fi același nume al școlii peste toate ofertele lor, aşa cum este cu numele companiei din care fac parte utilizatorii din industrie, iar la modificarea unei oferte, la utilizatorii de la școli nu va apărea o notiță că, pentru afișarea diferită a numelor de companii, schimbarea să se facă în secțiunea de profil a utilizatorului.

Add Offer

Offer Title

Description

Price

Site's url'

Location

Browse... No file selected.

 Submit

Figura 45: Adăugare ofertă

Atât pentru adăugarea ,cât și modificarea unei oferte, vor fi luate în vedere mereu numele ofertei, descrierea ei, prețul aferent (dacă este cazul) , link-ul către site-ul companiei (sau al școlii pentru utilizatorii de la școli) , locația unde se află serviciul sau bunul material și poza serviciului în sine.

După respectarea acestor chestiuni, oferta se poate încărca pe platformă și clienții pot să o vadă în secțiunea din care face parte utilizatorul respectiv. La modificarea ofertei se vor afișa diferit doar datele noi introduse în locul celor vechi la o ofertă deja existentă.

Capitolul 5

Concluzii și posibilități de dezvoltare

5.1 Concluzii

Aplicația aceasta de relocare și mutare în alt oraș este o soluție ideală pentru oricine are nevoie să se mute dintr-un oraș în altul, toate serviciile de care are nevoie fiind asigurate de companiile și școlile care doresc să își posteze oferta pe această platformă web pentru o comunicare rapidă, simplă și ușoară atât pentru client, dar și pentru corporatist sau om al școlii.

Aplicația oferă o interfață prietenoasă și ușor de utilizat pentru oricine, astfel încât să nu existe deloc dificultăți în a o folosi, interacțiunea fiind simplă.

Această interfață ușor de înțeles și de lucrat cu ea conține numeroase elemente intuitive de navigare, un aspect vizual destul de atractiv pentru ochiul oricărui utilizator și o simplitate în folosirea ei, de nu este posibil să nu înțeleagă cineva cum să o folosească.

Utilizatorii vor avea o experiență fluidă, putând căuta fără nicio problemă oferte sau emailuri pe platformă, iar ceilalți să le încarce fără nicio grija și repede.

Razor Views aduce multe beneficii în fluiditatea acestei aplicații, deoarece toate structurile de date vizuale se pot împărti în mai multe view-uri, sau mai bine spus, în mai multe pagini web pentru a avea o distribuție perfectă de astfel de view-uri, putând să treci de la o pagină la alta în funcție de nevoie, iar structurile afișate anterior, nu pun greutăți pe fluiditatea aplicației.

Iar .NET Core permite securizarea optimă a aplicației, astfel încât nimeni să nu poată să comită nicio infracțiune și să compromită un utilizator al acestei aplicații, deoarece sistemul de autentificare și autorizare avansat, nu va permite acest lucru, datele utilizatorilor fiind în siguranță în baza de date SQL.

5.2 Posibilități de extindere

Să se permită extragerea ofertelor de pe site-urile companiilor și școlilor printr-un API specific (cel mai probabil proprietar) pentru a automatiza procesul de postare , modificare și poate chiar și de ștergere a postărilor

Să se permită crearea de cont sau logarea în cont folosind și contul Facebook și poate și de LinkedIn și chiar Twitter (numit acum X)

Să se folosească .NET identity pentru a lega mult mai bine datele de securitate din back-end și pentru a se permite folosirea JWT-urilor

Să se convertească baza de date actuală SQL la AWS sau Azure și să fie pusă pe cloud pentru o siguranță sporită a datelor și o fluiditate și mai sporită a aplicației

Să se poată implementa un sistem de rating al serviciilor poste, dar și al clienților întâlniți de utilizatorii fie din industrii, fie în școli

Să se treacă la cea mai recentă versiune de C# (momentan cea mai disponibilă în 2024 este C# 10)

Listă figuri

1. Exemplu Intellisense listă opțiuni cu informații scurte.....	9
2. Exemplu de sugestii de completare a codului cu Intellisense.....	9
3. fluxul pachetelor între creatori,gazde, și utilizatorii lor.....	12
4. Cum să obții datele dintr-o tabelă prin LINQ.....	14
5. Schemă pentru modul de operare al Entity Framework Core.....	16
6. Schemă pentru model de REST API.....	19
7. Schemă pentru rutare în ASP.NET Core.....	20
8. Fișier Controllers cu clase controller.....	21
9. Ierarhie view-uri.....	22
10. Exemplu de formular într-un view.....	25
11. Schemă pentru arhitectura curată.....	29
12. Exemplu de controller în cod.....	30
13. Schemă de reprezentare a modului de funcționare pentru stratul de date.....	32
14. Ierarhia secțiunii vizuale.....	33
15. Schema bazei de date.....	34
16. Diagrama cazurilor de utilizare pentru platforma web.....	36
17. Exemplu de clasă unit of work în platforma de realocare și servicii.....	38
18. O parte din codul clasei BaseRepo.....	39
19. Clasa BaseEntity.....	40
20. Schema code-first approach.....	41
21. Exemplu de clasă entitate.....	41
22. Clasa RelocationDbContext.....	42
23. Activarea posibilității de ON CASCADE DELETE.....	43
24. Exemplu folosire [RequireHttps].....	45
25. Exemplu de navbar in Razor Views.....	46
26. Exemplu funcție din javascript.....	47
27. Exemplu de modal cu formular.....	47
28. Filtrul de securitate.....	48
29. Detaliile sesiunii.....	49
30. Rate limiter.....	49
31. Pagina de bun venit.....	51
32. Crearea contului.....	52
33. Logarea.....	53
34. Recuperare parolă prin email.....	53
35. Recuperare parolă cu telefonul prin SMS.....	54
36. Opțiunile utilizatorului normal.....	55
37. Opțiunile utilizatorului din industrie.....	55
38. Pagina de profil.....	56
39. Schimbarea parolei.....	57
40. Pagina cu oferte de Apartamente.....	58

41. Pagina cu lista de mail-uri.....	59
42. Vizualizarea mail-ului.....	60
43. Compunerea unui mail.....	61
44. Pagina cu ofertele postate.....	62
45. Adăugare oferta.....	63

Bibliografie

- [1] What is the difference between strongly and weakly typed programming languages?
<https://www.linkedin.com/advice/0/what-difference-between-strongly-weakly-typed-eqwl>
- [2] IntelliSense in Visual Studio
<https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2022>
- [3] Kemal Birer, ASP.NET Core for Jobseekers: Build Career in Designing Cross-Platform Web Applications Using Razor and Entity Framework Core, BPB Publications, 2022.
- [4] Microsoft SQL Server: Everything you need to know
<https://datascientest.com/en/microsoft-sql-server-everything-you-need-to-know>
- [5] Comparing Database Management Systems: MySQL, PostgreSQL, MSSQL Server, MongoDB, Elasticsearch, and others
<https://www.altexsoft.com/blog/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/>
- [6] Xavier Decoster (auth.) Maarten Balliauw, Pro NuGet, a 2-a ed., Apress, 2013.
- [7] An introduction to NuGet
<https://learn.microsoft.com/en-us/nuget/what-is-nuget>
- [8] Entity Framework Core
<https://learn.microsoft.com/en-us/ef/core/>
- [9] Migrations Overview
<https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli>
- [10] Querying Data
<https://learn.microsoft.com/en-us/ef/core/querying/>
- [11] Everything You Need To Know About REST APIs
<https://medium.com/@amirarahma.aa/everything-you-need-to-know-about-rest-apis-90d659dc9fc9>
- [12] Handle requests with controllers in ASP.NET Core MVC
<https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/actions?view=aspnetcore-8.0>
- [13] Views in ASP.NET Core MVC
<https://learn.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-8.0>
- [14] Partial views in ASP.NET Core
<https://learn.microsoft.com/en-us/aspnet/core/mvc/views/partial?view=aspnetcore-8.0>
- [15] Sufyan bin Uzayr, Mastering Bootstrap A Beginner's Guide, CRC Press, 2022.
- [16] Introducing asynchronous JavaScript
<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing>
- [17] JavaScript
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
- [18] Business-Logic Layer
<https://www.geeksforgeeks.org/business-logic-layer/>
- [19] Data-Access Layer
<https://www.geeksforgeeks.org/data-access-layer/>
- [20] What is Code-First?

<https://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx>

testare documentatie

ORIGINALITY REPORT

1
%

SIMILARITY INDEX

0
%

INTERNET SOURCES

0
%

PUBLICATIONS

0
%

STUDENT PAPERS

MATCH ALL SOURCES (ONLY SELECTED SOURCE PRINTED)

< 1%

★ Submitted to Politehnica University of Timisoara

Student Paper

Exclude quotes On

Exclude bibliography On

Exclude matches Off