

UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAȘI

Facultatea de Informatică



Lucrare de licență

BookishNet

Propusă de

Ursan Teofil-Cosmin

Sesiunea: Iulie, 2017

Coordonator Științific:

Asistent, dr. Vasile Alaiba

UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAȘI

Facultatea de Informatică

BookishNet

Ursan Teofil-Cosmin

Sesiunea: Iulie, 2017

Coordonator Științific:

Asistent, dr. Vasile Alaiba

DECLARAȚIE PRIVIND ORIGINALITATEA ȘI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licență cu titlul „*BookishNet*” este scrisă de mine și nu a mai fost prezentată niciodată la o altă facultate sau instituție de învățământ superior din țară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și dețin referința precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alți autori deține referința precisă;
- codul sursă, imagini etc. preluate din proiecte open source sau alte surse sunt utilizate cu respectarea drepturilor de autor și dețin referințe precise;
- rezumarea ideilor altor autori precizează referința precisă la textul original.

Iași,

Absolvent Ursan Teofil-Cosmin

(semnătura în original)

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*BookishNet*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” Iași să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași,

Absolvent Ursan Teofil-Cosmin

(semnătura în original)

Cuprins

Introducere	7
Motivație	7
Context	7
Cerințe funcționale	7
Abordare tehnică	8
Baza de date	8
Backend	8
Frontend	8
Framework-uri	8
Entity Framework Core	8
Bootstrap	9
Selenium	9
Swagger	9
Moq	9
Utilitare folosite	9
Microsoft Visual Studio Enterprise 2017	9
Draw.io	9
Contribuții	10
Proiectare	11
Arhitectura soluției	11
Capitolul 1: Dezvoltarea aplicației client	11
1.1. Arhitectura generală a aplicației	11
1.2. Angular Views	12
1.3. Angular Controllers	12
1.4. Angular Services	13
Capitolul 2: Dezvoltarea aplicației server	13
2.1. Arhitectura generală a aplicației	13
2.2. API Controllers	14
2.3. Service Layer	15
2.4. Data Layer	15
Capitolul 3: Testarea funcționalităților și asigurarea calității	15

1. Unit tests	15
2. Acceptance tests	17
Modelarea datelor	19
Arhitectura bazei de date.....	19
Entități.....	19
1. Books	19
2. Reviews.....	20
3. Genres	20
4. BookAuthor.....	20
5. Users	20
6. Roles	20
7. Messages	21
Relații între entități	21
1. Users-Books	21
2. Users-Messages.....	21
3. Users-Roles	21
4. Books-Reviews	21
5. Books-Genres.....	21
Protocoale de comunicare client – server	22
BookishNet API	22
Interfața cu utilizatorul.....	28
Home Page	28
Register	29
Welcome Back Page	30
Books	31
Book Page	32
Users	33
User Page	34
Implementare	36
Data Layer.....	36
Data Layer Tests	37
MVC	39

Autentificarea.....	39
Trimitere email.....	40
Directiva pentru confirmarea parolei	41
BookService	42
GetBooks.....	42
UpdateBook	44
LoginController.....	45
View binding.....	49
MVC Tests	51
Page Object Design Pattern.....	51
Browser Factory	54
Page Generator.....	56
GoToBookPageTest.....	57
Manual de utilizare	59
Concluzii generale	68
Referințe.....	69

Introducere

Motivație

În această lucrare voi propune o soluție pentru dezvoltarea unei aplicații web, atât pentru client cât și pentru server, care să ofere informații utilizatorilor despre cărți pe care doresc să le împrumute și nu le dețin sau nu le pot procura de la o bibliotecă sau librărie. Am ales această idee și datorită pasiunii mele pentru lectură.

De asemenea, consider că ar putea fi considerată și o aplicație socială deoarece utilizatorii vor avea posibilitatea de a cunoaște persoane noi și de a comunica cu acestea prin intermediul chat-ului din interiorul aplicației.

Context

Dat fiind faptul că, în momentul în care cineva dorește să împrumute o carte are nevoie de permis la o bibliotecă, permis care nu se oferă în orice condiții, spre exemplu, anumite biblioteci sunt doar universitare; oferă cărți spre împrumut doar studenților/profesorilor sau anumite cărți le poți citi doar la sala de lectură. La o căutare pe google se poate observa că există destul de puține asemenea aplicații, cea mai cunoscută fiind Bookster (o bibliotecă pentru companii din România).

Conform ultimelor statistici oficiale¹ din anul 2011 interesul pentru lectură este destul de scăzut. De asemenea, datorită numărului continuu în creștere de utilizatori în mediul online² consider că o asemenea aplicație online ar putea fi benefică.

Cerințe funcționale

Pentru a avea un impact pozitiv dar și pentru eventualitatea intrării pe piață a aplicației, ea va trebui să respecte următoarele cerințe funcționale:

- Utilizatorul va avea posibilitatea de a-și crea un cont nou
- Utilizatorul va putea folosi ca și metodă de autentificare contul creat la înregistrare
- Utilizatorul va putea să caute cartea dorită în funcție de anumite opțiuni: titlu, numele autorului, anul publicării cărții
- Utilizatorul va putea să-și editeze informațiile personale
- Utilizatorul va putea să-și managerieze cărțile pe care le deține
- Utilizatorul va fi putea fi contactat prin email
- Utilizatorii vor putea comunica prin intermediul chat-ului
- Utilizatorul va putea să vadă ce utilizatori mai sunt prezenți pe site și cărțile pe care aceștia le dețin

¹ Obiceiuri lectură - <http://www.ires.com.ro/articol/172/obiceiurile-de-lectura-ale-romanilor>

² Navigare pe Internet - http://ec.europa.eu/eurostat/statistics-explained/index.php/Information_society_statistics_-_households_and_individuals/ro

- Utilizatorul va avea posibilitatea de a se loga/deloga

Abordare tehnică

Baza de date

Pentru baza de date am ales soluția oferită de cei de la Microsoft, și anume, Microsoft SQL Server. Este o bază de date relațională, care are capacitatea de stocare și expunere a datelor pe baza unor cereri primite de la solicitant și poate fi rulată pe același computer sau pe un computer din rețea.

Backend

Pentru partea de backend voi folosi C#, un limbaj de programare functional, generic, orientat obiect, imperativ, declarativ. De asemenea, această componentă va face legătura între baza de date și frontend. Tot aici se vor transmite notificările și mesajele de la un utilizator către un altul.

Frontend

La nivel de frontend se vor folosi AngularJS, framework de Javascript (limbaj de nivel înalt și interpretat), care folosește desing pattern-ul MVC. De asemenea, AngularJS oferă posibilitatea de a crea elemente HTML proprii, altele decât cele specifice limbajului. Structura de bază a acestui framework este bazată pe servicii, controllere și directive.

De asemenea, pentru a face partea vizuală cât mai plăcută pentru utilizatorul final se va folosi CSS, un standard pentru formatarea elementelor unui document HTML. Stilurile se pot atașa elementelor HTML prin intermediul unor fișiere externe sau în cadrul documentului prin elementul <style> și/sau atributul style. Cu alte cuvinte, CSS este candidatul ideal pentru partea de web design.

Pentru crearea efectivă a paginilor se va folosi HTML. Este un limbaj de marcare utilizat pentru crearea paginilor web ce pot fi afișate într-un browser (sau navigator). Scopul HTML este mai degrabă prezentarea informațiilor – paragrafe, fonturi, tabele ș.a.m.d. – decât descrierea semanticii documentului. Specificațiile HTML sunt dictate de World Wide Web Consortium (W3C).

Framework-uri

Entity Framework Core

Este o tehnologie folosită pentru a accesa o bază de date, o versiune independentă de platforma de dezvoltare, extensibilă și mai ușoară comparativ cu Entity Framework. EF Core este un mapper de obiecte relaționale (ORM, mai pe scurt), care le oferă dezvoltatorilor de .NET posibilitatea de a lucra cu o bază de date folosind obiecte .NET. Totodată, elimină nevoia de a scrie cod pentru a accesa baza de date, cod pe care în mod normal un programator oricum ar fi nevoit să-l scrie. EF Core are și suport pentru mai multe motoare de baze de date.

Bootstrap

Este un framework pentru web pe partea de frontend gratuit și open-source, care ajută la realizarea design-ului pentru site-uri sau aplicații web. Conține anumite șabloane de design - HTML și CSS - pentru elementele vizibile în interfața utilizatorului dar și opțional, extensii Javascript. Spre deosebire de multe alte framework-uri web, se concentrează doar pe dezvoltarea frontend-ului.

Selenium

Este un framework de testare software portabil folosit pentru aplicații de tip web. Oferă o unealtă pentru înregistrare/playback pentru teste fără a învăța un limbaj de testare specific. De asemenea, oferă posibilitatea de a scrie teste în diverse limbaje de programare cum ar fi: C#, Java, PHP, Python, Ruby ș.a. Testele pot rula pe browserele modern. Dezvoltarea lor se poate realiza atât pe Windows cât și pe Linux sau OS X. Un alt avantaj este că este open-source și poate fi descărcat gratuit.

Swagger

Este cel mai popular și cel mai mare framework realizat pentru OAS³. El permite procesul de dezvoltare asupra întregului ciclu de viață al unui API, de la design și documentație, până la testare și ajungerea la clientul final. Unul dintre beneficiile acestui framework este că îți permite crearea documentației pentru un serviciu web REST în mod automat.

Moq

În esență, un framework de mocking permite generarea unei false implementări a unei dependențe cu scopul de a o folosi în Unit Teste fără a fi nevoie de implementarea concretă. Moq a fost creat cu scopul de a fi ușor de folosit, strongly-typed, ușor de refactorizat și minimalist (dar fără a-și pierde din funcționalitate). De asemenea, este folosit pentru C#/.NET și este folosit în Unit Teste pentru izolarea clasei testate de dependențele ei.

Utilitare folosite

Microsoft Visual Studio Enterprise 2017

Include un set complet de instrumente de dezvoltare pentru generarea de aplicații ASP.NET, Servicii Web XML, aplicații desktop și aplicații mobile. Oferă suport pentru mai multe limbaje de programare și este cea mai nouă versiune apărută.

Draw.io

Este un software online pentru diagrame. Se pot crea fire de execuție ale unui process, diagrame de procese, diagrame organizaționale, UML, ER și diagrame de rețea. De asemenea, este oferit gratuit.

³ OpenAPISpecification – cunoscut initial ca și Swagger Specification, reprezintă o specificație pentru descrierea, producerea, consumarea și vizualizarea serviciilor web REST de către o mașină (computer).

Contribuții

Lucrarea este structurată în trei capitole mari, fiecare din ele tratând, pe rând, procesul de dezvoltare din cele două perspective principale (client și server). Se va oferi motivația pentru anumite decizii luate pe parcursul procesului de dezvoltare dar și detalii relativ la implementare sau eventuale diferențe care există comparând elemente similare. De asemenea, se vor oferi detalii despre asigurarea calității produsului și a funcționalității pe termen scurt și lung. Cele trei capitole sunt enumerate mai jos:

- Capitolul 1: Dezvoltarea aplicației client
- Capitolul 2: Dezvoltarea aplicației server
- Capitolul 3: Testarea funcționalităților și asigurarea calității

În cadrul dezvoltării aplicației client s-au avut în vedere aspecte precum:

- Să fie user-friendly și intuitivă
- Să fie rapidă
- Să fie scalabilă în funcție de dispozitivul folosit
- Utilizarea de design-pattern-uri pentru separarea responsabilităților

Pe partea de server s-a avut în vedere ca aplicația să facă față unui volum consistent de utilizatori, să fie sigură și să păstreze confidențialitatea informațiilor despre utilizatori sau cărți:

- Folosirea autentificării și autorizării
- Autentificare bazată pe roluri
- Cererile procesate asincron sau sincron
- Utilizarea principiilor SOLID
- Utilizarea de design-pattern-uri pentru separarea responsabilităților

În ceea ce privește baza de date s-a dorit a fi o bază de date de dimensiuni reduse dar robustă:

- Legăturile între tabele folosind chei străine
- Utilizarea de chei primare compuse din alte chei străine

Testarea funcționalității și asigurarea calității are ca scopuri esențiale asigurarea faptului că aplicația respectă cerințele funcționale și posibilitatea de dezvoltare, îmbunătățire, aspecte ce pot fi aduse pe viitor:

- Crearea de unit teste pentru fiecare modul și layer în parte
- Crearea de teste de integrare pentru verificarea funcționalității modulelor împreună
- Crearea de teste automate pentru verificarea funcționalității din punctul de vedere al utilizatorului final

Proiectare

Arhitectura soluției

Arhitectura generală a soluției(aplicației) nu este diferită față de cea a unei aplicații web obișnuite. Ea cuprinde trei părți mari și importante: clientul, serverul și baza de date. Comunicarea generală se face după cum urmează: clientul trimite o cerere către server, acesta din urmă o evaluează. Dacă este cazul, se trimite o cerere către baza de date și se așteaptă un răspuns, altfel serverul va evalua cererea primită de la client și va pregăti un răspuns. Indiferent de scenariu, după pregătirea răspunsului de către server, acesta va trimite înapoi către client acest răspuns, iar pe partea clientului vor apărea modificările necesare. *Figura 1* ilustrează arhitectura generală a aplicației:



Figura 1: Arhitectura generală a aplicației

Capitolul 1: Dezvoltarea aplicației client

Aplicația client are ca scop facilitarea procesului de comunicare, de obținere și oferire de informații între utilizatori prin intermediul unei conexiuni la Internet. Aplicația a fost dezvoltată cu ajutorul editorului pus la dispoziție de cei de la Microsoft – Visual Studio 2017 Enterprise. Pe parcursul dezvoltării s-a avut în vedere utilizarea tehnicilor necesare pentru obținerea unei aplicații de tip SPA⁴ astfel încât utilizatorul să obțină rezultate mai rapide afișate în pagină iar aplicația să fie utilizabilă în condiții reale, pe multiple dispozitive. Tehnicile folosite vor fi descrise, pe larg, în subcapitolele ce vor urma.

1.1. Arhitectura generală a aplicației

Arhitectura aplicației client este puțin diferită de cea a unei aplicații client de tip web obișnuite. Astfel, cererile utilizatorului vor fi preluate din partea de view (fie că este vorba de butoane, câmpuri de input, hyperlink-uri) și vor fi transmise către controllerul Angular specific (fiecare view, în parte, are asignat câte un controller care se ocupă de administrarea primului menționat). De aici, controllerul Angular preia cererea și dacă este necesar o trimite mai departe către un serviciu Angular, urmând apoi să aștepte un răspuns. Acesta, în cele din urmă, va fi delegatul care va trimite cererea către aplicația server. Serviciul va aștepta un răspuns de la server după care va întoarce rezultatele necesare către controller, iar acesta, la rândul său, va transmite

⁴ SPA(Single-page application) – aplicație de tip web sau site web al cărui conținut să poate fi mapat pe o singură pagină, urmărindu-se astfel simularea unei aplicații desktop ca și experiență a utilizatorului. Într-o asemenea aplicație, fie tot codul necesar – HTML, Javascript, CSS – este primit la o singură încărcare de pagină, fie anumite resurse sunt încărcate în mod dynamic și adăugate în pagină, ca răspuns la acțiunile utilizatorului.

modificările necesare către view. Această aplicație trebuie să fie capabilă să ofere utilizatorului o experiență plăcută de utilizare, să fie ușor de folosit și să notifice utilizatorul în legătură cu mesaje noi. Diagrama aplicației client poate fi văzută mai jos, în *Figura 2*:

Aplicația client

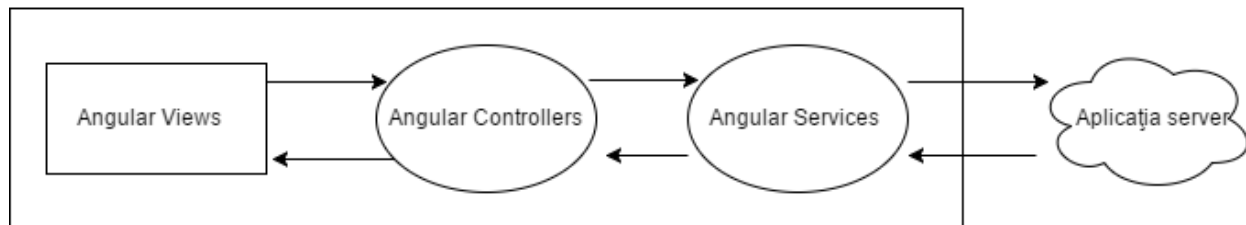


Figura 2: Arhitectura aplicației client

1.2. Angular Views

Utilizatorul va avea contact direct cu partea de Angular Views a aplicației client. Deși ele sunt menționate la plural, component principal este reprezentată de un view general, similar unei pagini de home. Diferența esențială față de aplicațiile web clasice vine acum în discuție. Deoarece Angular permite dezvoltarea de aplicații de tip SPA³, asta înseamnă că nu vom încărca de fiecare dată o întreagă pagină web ci doar o anumită porțiune, porțiune care este definită sub formă de view. De aceea avem mai multe view-uri în aplicație, însă scheletul este același, având astfel posibilitatea de a modifica doar anumite porțiuni ale view-ului principal și renunțând astfel la clasică reîncărcare a tuturor elementelor de pe o pagină HTML standard.

Sintaxa pentru un asemenea view este similară cu cea a unei pagini HTML, cu mențiunea că unele elemente pe care le găsim într-o pagină standard devin redundante. Unele dintre acestea ar fi: `<html></html>`, `<head></head>`, `<body></body>`. Astfel, un view poate să înceapă foarte bine, fără nicio problemă cu elemente de tipul: `<div></div>`, `<table></table>`, `` ș.a. Singura excepție de la această abordare o reprezintă view-ul general, așa numitul `index.html`. Această pagină arată în totalitate ca o pagină HTML autentică relativ la părțile de `<head>`, `<body>`, `<script>`. Așadar, view-urile sunt cele pe care utilizatorul le va întâlni în aplicație și va interacționa cu aceasta prin intermediul lor.

1.3. Angular Controllers

A doua componentă a aplicației client, extrem de importantă și partea centrală din punctul meu de vedere, este reprezentată de controllerele Angular. Aceste componente sunt definite, de regulă, câte unul pe fiecare view. Acest fapt este realizat pentru a aplica un best practice asupra codului și pentru respectarea principiului de separare a responsabilităților. Așadar, fiecare view are asignat un controller care se ocupă de managementul view-ului respectiv.

Ceea ce face controller-ul atât de puternic este faptul că el este cel care se ocupă de modificările survenite în partea de view dar și în partea de servicii. Astfel, el este cel care notifică view-ul în privința unor schimbări și tot el este cel care trimite cereri către servicii, pe care le vom

detalia în subcapitolul următor, pentru a primi datele cerute sau transmise de utilizator, prin intermediul view-ului.

De asemenea, controller-ul este puntea de legătură între view și datele care trebuiesc transmise către aplicația server. Totodată, deoarece abordarea a fost aleasă în acest mod, câte un controller pe fiecare view, va duce la creșterea vitezei de execuție și rezolvare a cererilor, un aspect esențial urmărit de aplicațiile contemporane și implicit, de aplicația de față.

1.4. Angular Services

Ultima componentă majoră și esențială a aplicației client este reprezentată de serviciile Angular. Aceste servicii sunt obiecte ce pot fi înlocuite și sunt legate împreună folosind dependency injection. Cu ajutorul lor se poate organiza și partaja codul în aplicație. Serviciile se bazează pe două design pattern-uri: Lazy initialization și Singleton.

Deoarece folosesc aceste design pattern-uri, serviciile Angular oferă diferite avantaje precum:

- Inițializarea în momentul în care avem nevoie de anumite resurse
- Crearea unei singure instanțe de-a lungul rulării aplicației
- Evitarea consumului inutil de resurse
- Creșterea performanțelor datorită consumului mai redus de resurse

De asemenea, serviciile Angular sunt cele care fac cererile către API-ul aplicației, un API care oferă resursele necesare aplicației client. Această componentă va fi detaliată mai pe larg în capitolul dedicat aplicației server. Așadar, serviciile sunt cele care fac legătura între aplicația server și cererile primite de la utilizator – pot fi considerate, pe bună dreptate, legătura între cele două aplicații.

Capitolul 2: Dezvoltarea aplicației server

2.1. Arhitectura generală a aplicației

Pe partea de server s-a avut în vedere ca aplicația să facă față unui volum consistent de utilizatori, să fie sigură și să păstreze confidențialitatea informațiilor personale despre utilizatori sau cărți. Astfel, utilizatorul poate accesa aplicația fie autentificat, fie nu, cu mențiunea că dacă nu este autentificat accesul la anumite resurse din aplicație îi va fi restricționat celui utilizator.

De asemenea, accesul în aplicație, ca și utilizator conectat este realizat pe bază de roluri. Astfel, un utilizator poate avea unul din cele trei roluri posibile: Admin, Author, User. Se poate deduce astfel că în funcție de rolul pe care îl deține, utilizatorul va avea access sau nu la mai multe resurse, totul depinzând de gradul de încredere oferit de rolul asignat. O diagramă sumară a aplicației server poate fi vizualizată în *Figura 3*:

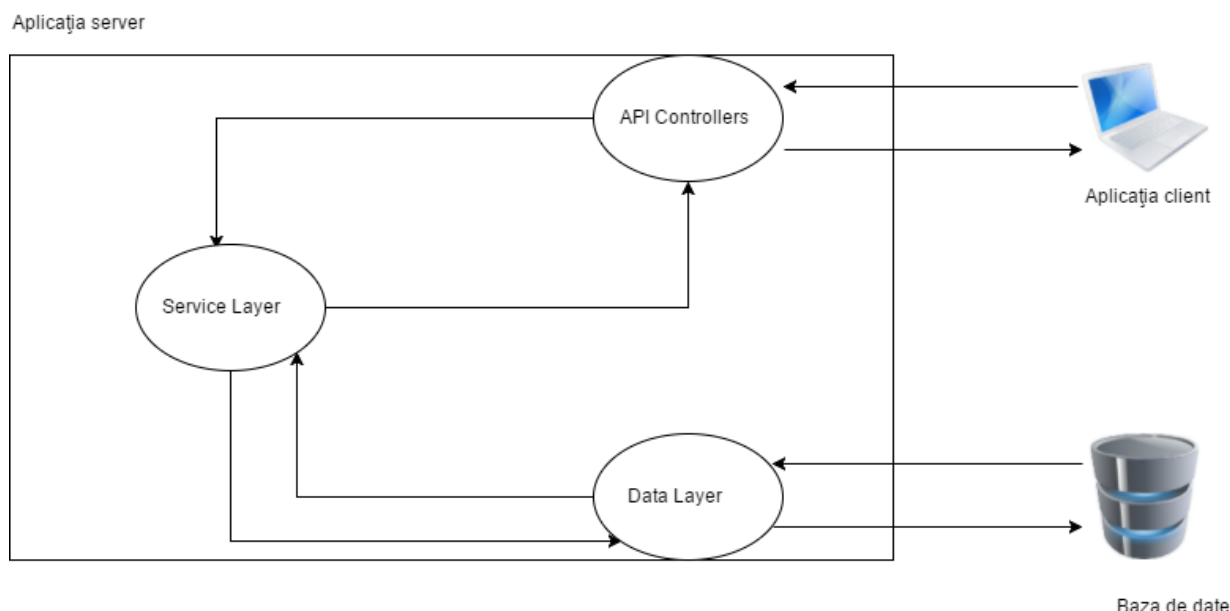


Figura 3: Arhitectura aplicației server

2.2. API Controllers

Odată cu trimiterea cererii de la client și sosirea ei pe partea de server, prima componentă cu care are loc interacțiunea este unul dintre controllerele serviciului REST⁵ realizat folosind Web API⁶. De la acest nivel, serverul propagă cererea la nivelul următor, urmând apoi să aștepte primirea unui răspuns, fie de succes, fie de eșec.

După ce se primește un răspuns, serverul întoarce înapoi clientului fie reprezentarea resursei dorite la o anumită adresă URL fie un răspuns cu eroarea întâlnită în timpul procesării cererii (de exemplu, în caz de eșec se poate returna status code-ul 404 – cu semnificația că resursa solicitată nu a fost găsită).

Un lucru important care se realizează aici și merită menționat este injectarea serviciului corespunzător pentru rezolvarea cererii în constructorul controller-ului. Acest procedeu se folosește pentru a duce la reducerea cuplajului între diferite clase. Totodată, un alt beneficiu al pattern-ului dependency injection este observabil la scrierea de Unit Teste, componente pe care le vom detalia în capitolul corespunzător.

De asemenea, tot la acest nivel se stabilesc și operațiile care pot fi făcute cu sau fără autentificare. Dacă atributul de autorizare ar lipsi din aceste controllere atunci, nu ar mai fi necesară autentificarea pentru a obține informații din baza de date sau chiar pentru a modifica anumite date. În acest caz, accesul ar putea fi restricționat doar pe partea de client, iar un atacator

⁵ **Representational state transfer** - Stil architectural de dezvoltare a aplicațiilor web cu focalizare asupra reprezentării datelor

⁶ **API** - interfață de programare a unei aplicații pentru un server sau un browser web. Este un concept de dezvoltare web, este folosit de obicei de o aplicație web pe partea de client

ar putea profita destul de ușor de această vulnerabilitate a aplicației pentru a citi/modifica date prețioase.

2.3. Service Layer

Acest nivel al aplicației a fost creat în mod special pentru a crea o legătură între controllerele din nivelul superior și nivelul de mai jos – Data layer. Am ales acest lucru pentru a separa și mai bine responsabilitățile fiecărui nivel. Astfel, am evitat contactul direct din acest nivel cu baza de date dar și contactul direct cu aplicația client.

Lucrurile sunt similare în ceea ce privește drumul pe care îl are de parcurs cererea. Se primește cererea, se trimite către nivelul de mai jos și se așteaptă un răspuns. De asemenea, se poate spune că acest nivel este mai mult unul de transport pentru a facilita transmitia datelor între diferitele nivele ale aplicației. Acest lucru a fost realizat pentru a ajuta și mai mult la separarea responsabilităților.

2.4. Data Layer

La acest nivel, aplicația pregătește interacțiunea cu baza de date. Pentru a facilita accesul la baza de date dar și pentru a preveni modificarea comenzilor Sql trimise către baza de date s-a folosit Entity Framework Core. Astfel, acest nivel conține atât modelele pentru crearea bazei de date dar și Repository-uri pentru a apela efectiv operațiile ce vor fi operate pe baza de date.

Fiecare model în parte are o interfață și un repository care implementează interfața specific. S-au creat atât operațiile CRUD⁷ dar și alte operații suplimentare pentru a aduce doar anumite rânduri din baza de date. Crearea de interfețe a fost aleasă pentru a respecta al patrulea principiu din SOLID, și anume, principiul segregării interfeței. În acest mod, fiecare clasă ce va implementa o anumită interfață se va ocupa de un model în parte, realizându-se, în același timp și separarea în funcție de entitatea dorită.

Capitolul 3: Testarea funcționalităților și asigurarea calității

1. Unit tests

Pentru a verifica funcționalitatea fiecărui modul în parte am folosit așa numita strategie de Unit Testing⁸. Astfel, pentru fiecare modul în parte, în cazul aplicației de față pentru fiecare Repository, s-a creat o clasă separată care să se ocupe cu verificarea funcționalităților pentru respectivul Repository. Așadar, fiecare metodă din clasa principală a fost testată și verificată că întoarce un rezultat corespunzător în clasa de test.

Această tehnică are și avantaje și dezavantaje pe care le voi prezenta pe scurt în cele ce urmează. Ca și dezavantaje ale scrierii de Unit Teste putem enumera: timpul alocat pentru a realiza teste potrivite poate fi semnificativ, nu toate erorile pot fi detectate aici – spre exemplu, cele de

⁷ CRUD – Create, Read, Update, Delete respectiv operațiile de adăugare, citire, modificare și ștergere

⁸ **Unit Testing** – nivel de testare software unde unități individuale/componente ale unui soft sunt testate. Scopul este de a valida că fiecare unitate component a software-ului funcționează corespunzător.

integrare. Unele dintre avantajele Unit Testelor sunt următoarele: erorile pot fi găsite într-o fază inițială a procesului de dezvoltare, codul este reutilizabil și mai ușor de făcut debugging, costurile de reparare a bug-urilor sunt mult mai reduse, eficiența crescută cu privire la mentenanța și schimbările ce pot surveni în cod.

Având în minte toate aceste lucruri am hotărât că este potrivit să folosesc această tehnică în cadrul aplicației pentru a repera mai ușor erorile deși inițial am alocat o bună măsură de timp pentru a le scrie. Acum, după ce le-am finisat, pot spune că s-a meritat timpul alocat pentru a le construi și sunt hotărât să folosesc această tehnică și pe viitor, ori de câte ori voi avea ocazia.

Un alt aspect important din punctul meu de vedere este că Unit Testele îți oferă posibilitatea de a scrie cod. Astfel, dacă ești un pasionat de IT cum este și cazul meu, acest aspect nu este deloc de neglijat.

De asemenea, pentru a-ți crea teste bune/potrivite trebuie să eviți să folosești datele reale care există în baza de date. În acest moment, ai la dispoziție două soluții pentru a nu folosi aceste date: fie folosești o bază de date de test, fie folosești mock-uri. Soluția aleasă de mine este cea varianta a doua pentru că simulează foarte bine, din punctul meu de vedere, o bază de date reală. Mai multe detalii despre Mock-uri se vor găsi în secțiunea de Implementare.

Ca și arhitectură, un Unit Test are câteva componente specifice, unele dintre ele obligatorii, altele nu. Diagrama generală a unei clase de test se află prezentată în *Figura 4*, urmând ca apoi să ofer detalierea fiecărei componente în parte:

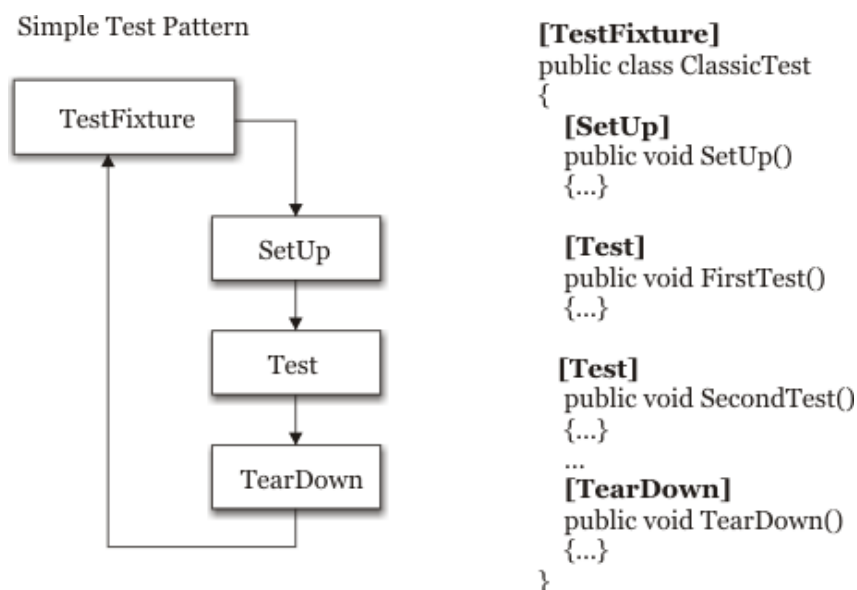


Figura 4: Arhitectura unei Clase de Test

Când se dorește crearea unei clase de test se folosește o adnotare deasupra acesteia. Această adnotare diferă în funcție de framework-ul⁹ folosit. În cazul de față, adnotarea *TestFixture* este specifică framework-ului **NUnit**. Ceea ce face de fapt acest atribut este faptul că marchează clasa ca fiind o clasă de test. Astfel, compilatorul va ști să trateze acea clasă în mod diferit, ca pe una de test, și se va aștepta să găsească în interiorul ei metodă/metode de test.

Al doilea atribut prezent în figură este *SetUp*. De menționat faptul că toate atributele prezente în această diagramă sunt specifice pentru NUnit. Atributul *SetUp* se folosește, de regulă, când există teste care doresc să împartă resurse comune. Metoda care are atributul acesta specificat deasupra ei va rula de fiecare dată înainte de a începe efectiv rularea metodei de test. Acest atribut nu este obligatoriu.

Atributul *Test* reprezintă efectiv metoda de test. În interiorul acesteia se introduce codul cu apelul către metoda din clasa de bază ce se dorește a fi testată. De regulă, este recomandat a se folosi un singur apel pe metodă pentru a testa un singur caz, dar există și posibilitatea de a face apeluri către mai multe metode – este considerat a fi un “bad practice”.

Ultimul atribut este cel de *TearDown* și este folosit pentru a elibera resursele folosite. Metoda cu acest atribut specificat deasupra ei, va rula de fiecare dată la finalul unui test. De regulă, este folosită când există și o metodă cu atributul *SetUp*. Aici se pot găsi instrucțiuni de închidere a conexiunii a baza de date sau golirea anumitor variabile.

2. Acceptance tests

Pentru a putea verifica dacă aplicația îndeplinește cerințele funcționale din punctul de vedere al utilizatorului final am ales să folosesc strategia numită Acceptance testing¹⁰. Prin aceasta se dorește să se simuleze cât mai bine comportamentul pe care aplicația îl va avea când va ajunge la clientul final. De asemenea, se va testa în același timp interacțiunea pe care o va putea avea utilizatorul cu aplicația.

Un alt lucru ce merită menționat este că aceste teste sunt automate. Astfel, am renunțat la varianta de a le folosi pe cele manuale. Un beneficiu major al acestui lucru este faptul că le scrii o singură dată și apoi le poți reutiliza, comparativ cu cele manuale pe care ești nevoit să le iei de fiecare dată de la capăt. O diagramă generală arată unde sunt poziționate aceste teste în procesul de dezvoltare în *Figura 5*:

⁹ Unit testing frameworks sunt cel mai des întâlnite sub formă de produse de tip third-party care nu sunt incluse în pachetul de bază al compilatorului. Ele ajută la simplificarea procesului de unit testing și sunt dezvoltate pentru o varietate de limbaje de programare. Cele mai folosite framework-uri în Visual sunt: MSTest/Visual Studio, NUnit și xUnit.NET.

¹⁰ Metodă de testare formală centrată pe respectarea nevoilor utilizatorului, a cerințelor și pe procesele business menite spre a determina dacă un sistem satisface criteriile de acceptare. De asemenea, permite utilizatorului, clientului sau altei entități autorizate să determine dacă sistemul construit va fi acceptat sau nu pentru livrare.

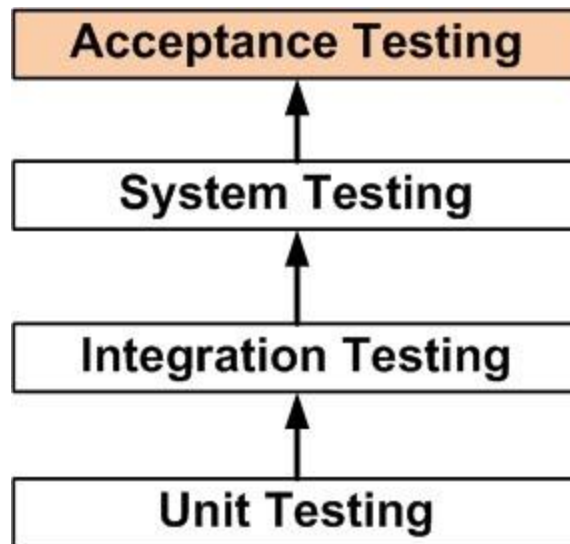


Figura 5: Arhitectura generală a nivelelor de testare

După cum se poate vedea și în *Figura 5*, Acceptance Testing reprezintă nivelul cel mai de sus în procesul de testare și asigurare a calității unui produs. Se poate spune despre aceste teste că reprezintă ultima metodă de verificare înainte ca produsul final să fie livrat către client. Așadar, este o metodă bună de a evita unele probleme care ar putea apărea după livrarea produsului către clientul final.

Bineînțeles că aceste tipuri de teste pot fi create și fără existența celorlalte teste din straturile inferioare, dar, implicit, acest lucru ar putea duce frecvent la diverse probleme și erori la rularea acestor teste. De aceea, este recomandat ca toate aceste nivele de testare să existe indiferent cât de solidă se cere a fi aplicația sau nu.

Așadar, având această suită de teste (Unit Tests, Acceptance Tests) aplicația capătă un grad mai ridicat de încredere din punct de vedere al depistării bug-urilor și un nivel mai scăzut de producere al erorilor în timpul utilizării acesteia. Datorită acestor beneficii principale aduse, consider că timpul alocat pentru dezvoltarea lor a meritat efortul, ducând aplicația finală la un alt nivel, un nivel superior de calitate.

Modelarea datelor

Arhitectura bazei de date

Pentru realizarea bazei de date s-a avut în vedere consistența datelor și dimensiunea redusă a acesteia. S-a dorit ca atât numărul de tabele dar și relațiile aferente între ele să fie cât mai redus. S-a recurs astfel la soluția de a se introduce șapte entități, așa cum se poate vedea și în *Figura 6* de mai jos, entități care vor fi detaliate în cele ce urmează.

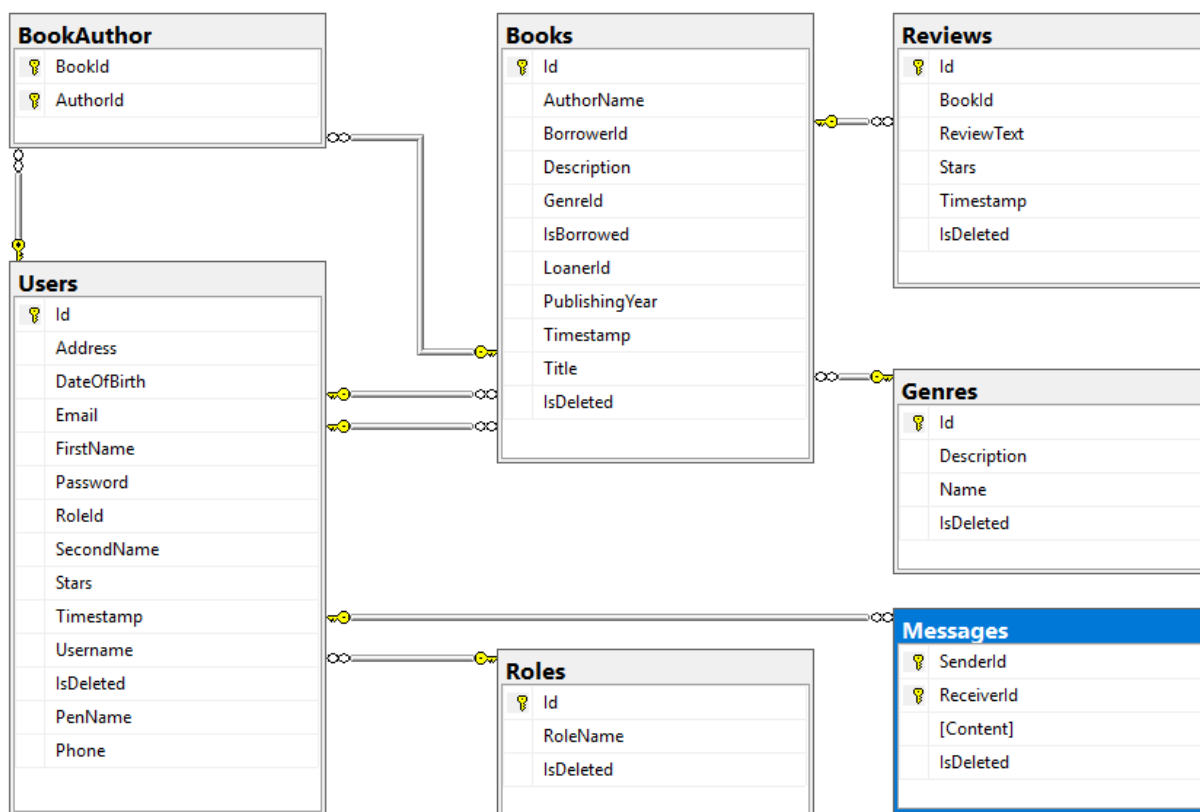


Figura 6: Diagrama bazei de date

Entități

1. Books

Entitatea centrală a acestei scheme este reprezentată prin tabela Books. Ea conține diverse informații necesare pe care o carte le poate conține. Se poate vedea cu ușurință care este cheia primară, și anume, coloana Id care are proprietatea de autoincrementare și este de tip număr întreg. Pe baza acestei coloane se poate identifica, în mod unic, o înregistrare de tip carte a tablei. Celelalte câmpuri ale tablei se împart în două secțiuni: obligatorii (trebuie introdusă o valoare în respectivele coloane) sau neobligatorii (câmpurile respective pot avea sau nu valori). Cele obligatorii sunt după cum urmează: Title, AuthorName, Timestamp, IsDeleted, GenreId, LoanerId iar celelalte care nu s-au menționat nu sunt obligatorii.

2. *Reviews*

Este o entitate secundară, mai redusă ca și dimensiune decât precedent și are ca rol principal reținerea unor informații suplimentare despre entitate de tip carte. Tabela are o cheie primară, denumită Id pe baza căreia se poate identifica în mod unic fiecare review primit de către o entitate carte. Există și aici câmpuri obligatorii: BookId, Timestamp, IsDeleted și câmpuri neobligatorii: ReviewText, Stars.

3. *Genres*

Această entitate are ca scop determinarea unui gen anume pentru o entitate de tip carte, așadar, orice carte va avea asignat un anume gen. Spre exemplu, o carte ar putea avea genul Romantic, o alta genul Aventura, o alta genul Science Fiction ș.a. Ca și câmpuri prezente în tabelă avem aceeași împărțire clasică – obligatorii: Name, IsDeleted – și neobligatorii: Description.

4. *BookAuthor*

Entitatea (tabela) BookAuthor poate fi văzută doar ca o unitate de legătură între entitatea de tip carte (prezentată mai sus) și entitatea de tip utilizator (ce urmează a fi prezentată mai jos). Spre deosebire de orice altă entitate din schemă, aceasta conține doar două câmpuri: BookId și AuthorId, fiecare dintre ele fiind o cheie străină către tabela Books, respectiv Users și împreună formează cheia primară a tablei, folosită pentru a identifica în mod unic fiecare înregistrare. Acest lucru duce la evitarea duplicatelor în tabela Books și/sau Users.

5. *Users*

Tabela de users este folosită pentru memorarea utilizatorilor, mai precis a detaliilor despre aceștia. Această entitate este cea de-a doua entitate principală a bazei de date, de ea legându-se alte entități (relațiile vor fi detaliate mai jos). Aici vom găsi informațiile necesare despre utilizatori, cum ar fi cele obligatorii: Email, Username, Password, RoleId, Timestamp, DateOfBirth, IsDeleted dar și informații neobligatorii: Address, Phone, FirstName, SecondName, PenName (în cazul în care utilizatorul este scriitor).

6. *Roles*

Această entitate, deși mai redusă ca dimensiuni este un element destul de important al schemei deoarece aici sunt stocate rolurile disponibile în aplicație dar și informații despre cine deține rolul respective. Tabela are doar trei câmpuri, toate dintre ele fiind obligatorii. Primul câmp este cel de Id, care este și cheia primară pentru tabelă. Celelalte două câmpuri sunt RoleName, unde se reține efectiv valoarea rolului și IsDeleted, câmp de tip Boolean pentru a determina dacă înregistrarea a primit sau nu o cerere de a fi ștearsă. Acest câmp a fost folosit în toate tabelele pentru a evita operația clasică de ștergere și a o înlocui cu una virtuală¹¹, astfel datele nu vor fi pierdute total la o cerere de ștergere.

¹¹ Oferirea posibilității de a reactiva anumite înregistrări care s-au droit a fi eliminate în trecut. Renunțarea la metoda clasică de ștergere a înregistrărilor din baza de date fără posibilitate de recuperare.

7. Messages

Ultima entitate a tabelii este reprezentată de un istoric al conversațiilor, așa cum ar mai putea fi numit. Aici se vor stoca mesajele pe care utilizatorii le-au purtat de-a lungul timpului în interiorul aplicației. Tabela conține o cheie primară, formată din două chei străine: SenderId și ReceiverId. De asemenea, mai există câmpul obligatoriu IsDeleted și câmpul optional Content.

Relații între entități

1. Users-Books

Relația între utilizatori și cărți este una de tip many-to-many. Asta înseamnă că o carte poate avea mai mulți utilizatori (acest aspect a fost gândit pentru situația în care o carte este scrisă de mai mulți autori) dar și un utilizator poate avea mai multe cărți (ele pot fi împrumutate sau gata spre a fi împrumutate). Soluția cea mai simplă, dar cu redundanțe multiple ar fi fost să se introducă câte o cheie străină reprezentând identificatorul celeilalte entități în fiecare tabel. Pentru a evita o asemenea situație s-a creat un tabel separate, BookAuthor, unde se vor reține doar identificatorii unici ai fiecărei părți implicate.

2. Users-Messages

Relația între aceste entități este de tip one-to-many. Aceasta implică faptul că un utilizator poate avea mai multe mesaje, cu alte cuvinte o listă de mesaje. Referința acestui fapt s-a făcut prin cheia primară compusă din tabela Messages, alcătuită din două chei străine: SenderId și ReceiverId. Relația poate fi văzută și în *Figura 6*.

3. Users-Roles

Așa cum am menționat și mai sus, în aplicație vor exista mai multe tipuri de roluri, fiecare utilizator având un rol specific. Aceasta implică o relație de tip one-to-many de la Roles către Users. Astfel, fiecare utilizator va avea un singur rol și numai unul la un moment dat iar un rol va putea avea mai mulți utilizatori. Această legătură s-a realizat prin adăugarea câmpului RoleId în tabela Users ca și cheie străină.

4. Books-Reviews

O carte are nevoie și de comentarii, păreri, recomandări, de aceea am ajuns la concluzia că este necesar pentru a stabili această legătură între entități de o relație one-to-many de la Books către Reviews. Pentru a realiza acest lucru, s-a adăugat în tabela Reviews un câmp denumit BookId, câmp ce reprezintă o cheie străină pentru Reviews venită de la tabela Books.

5. Books-Genres

Fiecare carte va avea asignat un anumit gen. Pornind de la această premisă am ajuns la concluzia că este necesară o legătură între tabelele Books și Genres. În același timp, același gen, poate să fie acontat la mai multe cărți așadar, relația care s-a definit a fost de tipul one-to-many în sensul Genres-Books. Astfel, tabela Books a primit un câmp suplimentar, GenreId, adăugat ca și cheie străină cu referință către o entitate de tip gen.

Protocoale de comunicare client – server

După cum s-a menționat în capitolul 2, aplicația pune la dispoziție un API de tip REST. În general, un web API pe partea de server este o interfață programatică care conține unul sau mai multe endpoint-uri expuse în mod public. Ele sunt reprezentate printr-un sistem de mesaje cerere-răspuns, de obicei în format JSON sau XML, mesaj care este expus via web, de cele mai multe ori având în spate un server HTTP. În cele ce urmează se vor detalia serviciile web/operațiile pe care le pune la dispoziție API-ul prezentei aplicații. [1]

BookishNet API

The screenshot displays the Swagger UI for the BookishNet API. At the top, there's a green header with the Swagger logo, the URL `http://localhost:45719/swagger/v1/swagger.json`, and a dropdown menu showing "BookishNet API V1". Below the header, the API is organized into three sections: Account, Book, and Email. Each section has a "Show/Hide", "List Operations", and "Expand Operations" link. The Account section lists four endpoints: POST /api/Account/login, POST /api/Account/register, GET /api/Account/logout, and GET /api/Account. The Book section lists eight endpoints: GET /api/Book, POST /api/Book, DELETE /api/Book/{id}, GET /api/Book/{id}, PUT /api/Book/{id}, GET /api/Book/{book}/{bookTitle}, GET /api/Book/{book}/{author}/{authorName}, GET /api/Book/{book}/{books}/{user}/{loanerId}, and GET /api/Book/{book}/{books}/{user}/{loanerId}/{borrowerId}. The Email section lists one endpoint: POST /api/Email.

Figura 7: O parte din API-ul aplicației vizualizat în Swagger

În Figura 7 se poate observa interfața pe care o creează, în mod automat, Swagger. Pentru fiecare controller API se creează o zonă specifică. Spre exemplu, în figura precedentă se pot observa operațiile posibile pentru controller-ele **Account**, **Book** și **Email**. Pentru detalierea serviciilor oferite voi detalia comportamentul unui singur controller, celelalte funcționând similar sau având comportament similar. Controller-ul ales este **Book**.

Primul apel API este cel de Show All Books. Acest apel se găsește la endpoint-ul /api/Book. Verbul HTTP care se folosește este GET. Nu este nevoie de parametri URL și nici

parametri de tip date. În caz de succes se va returna o listă de cărți sau o listă vidă, având status code-ul 200. În caz de eroare se va afișa o listă vidă și un status code cu valoarea 404 – NOT FOUND sau 500 – INTERNAL SERVER ERROR. Apelul este ilustrat în *Figura 8*:



Figura 8: Apelul către *getBooks*

Al doilea apel este cel de Post Book. Acest apel se găsește la endpoint-ul /api/Book. Verbul HTTP care se folosește este POST. Nu este nevoie de parametri URL dar este necesar să fie trimis un obiect de tip Book. În caz de succes se va returna status code-ul 200. În caz de eroare se va returna un status code cu valoarea 500 – INTERNAL SERVER ERROR. Apelul este ilustrat mai jos în *Figura 9*:

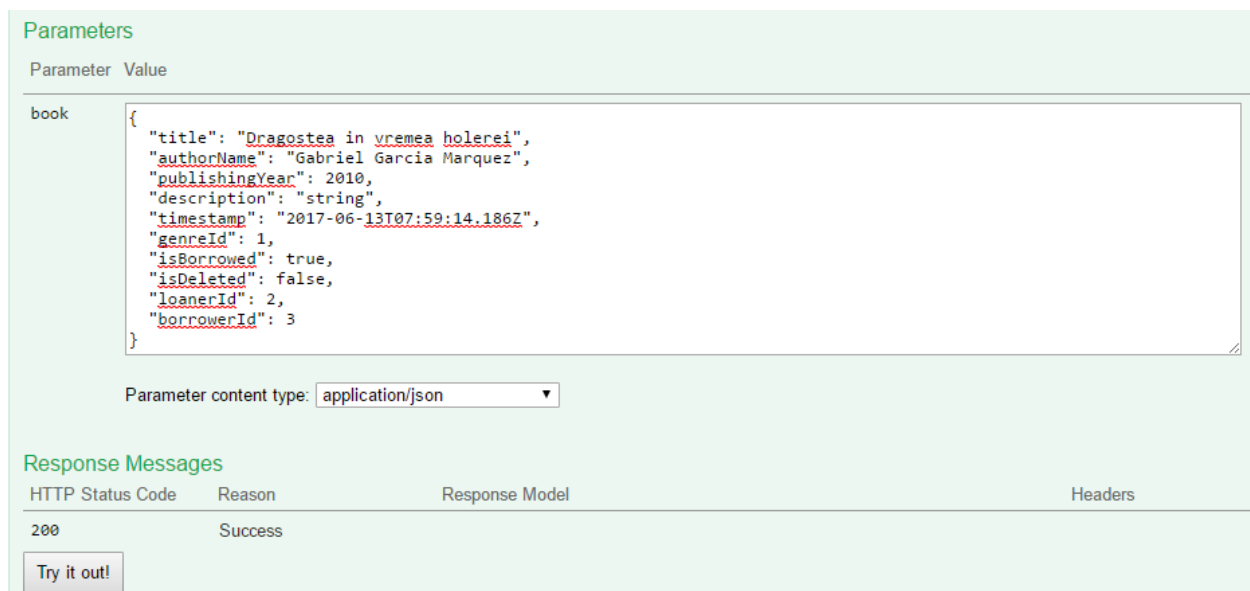


Figura 9: Apelul către *Post Book*

Al treilea apel este cel de Delete Book. Acest apel se găsește la endpoint-ul `/api/Book/{id}`. Verbul HTTP care se folosește este DELETE. Este necesar ca parametrul URL `id` să fie de tip întreg dar nu sunt necesari parametri de tip date. În caz de succes se va returna status code-ul 200. În caz de eroare se va returna un status code cu valoarea 404 – NOT FOUND sau cu valoarea 500 – INTERNAL SERVER ERROR. *Figura 10* următoare ilustrează acest apel:

DELETE `/api/Book/{id}`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	12		path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

[Try it out!](#)

Figura 10: Apelul către Delete Book

Al patrulea apel este cel de Get Book By Id. Acest apel se găsește la endpoint-ul `/api/Book/{id}`. Verbul HTTP care se folosește este GET. Este necesar ca parametrul URL `id` să fie de tip întreg dar nu sunt necesari parametri de tip date. În caz de succes se va returna status code-ul 200 și un obiect de tip carte. În caz de eroare se va returna un status code cu valoarea 404 – NOT FOUND sau cu valoarea 500 – INTERNAL SERVER ERROR. Apelul este ilustrat în cele ce urmează în *Figura 11*:

GET `/api/Book/{id}`

Response Class (Status 200)
Success

Model | **Example Value**

```
{
  "id": 0,
  "title": "string",
  "authorName": "string",
  "publishingYear": 0,
  "description": "string",
  "timestamp": "2017-06-13T07:59:14.197Z",
  "genreId": 0,
  "genre": {
    "id": 0,
    "name": "string"
  }
}
```

Response Content Type `text/plain`

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	11		path	integer

[Try it out!](#)

Figura 11: Apelul către Get Book By Id

Al cincilea apel este cel de Update Book. Acest apel se găsește la endpoint-ul `/api/Book/{id}`. Verbul HTTP care se folosește este PUT. Este necesar ca parametrul URL `id` să fie de tip întreg și un parametru de tip carte. În caz de succes se va returna status code-ul 200. În caz de eroare se va returna un status code cu valoarea 404 – NOT FOUND sau cu valoarea 500 – INTERNAL SERVER ERROR. Apelul este ilustrat și mai jos în *Figura 12*:

The screenshot shows a REST client interface for a PUT request to the endpoint `/api/Book/{id}`. The request method is PUT. The parameter `id` has a value of 10. The request body is a JSON object representing a book. The response is a 200 status code with the reason 'Success'.

Parameters

Parameter	Value
id	10

book

```
{
  "title": "Dragostea in vremea holerei",
  "authorName": "Gabriel Garcia Marquez",
  "publishingYear": 2010,
  "description": "Description update!",
  "timestamp": "2017-06-13T07:59:14.186Z",
  "genreId": 1,
  "isBorrowed": true,
  "isDeleted": false,
  "loanerId": 2,
  "borrowerId": 3
}
```

Parameter content type: `application/json`

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

Try it out!

Figura 12: Apelul către Update Book

Al șaselea apel este cel de Get Book By Title. Acest apel se găsește la endpoint-ul `/api/Book/book/bookTitle`. Verbul HTTP care se folosește este GET. Este necesar ca parametrul URL `bookTitle` să fie de tip string dar nu sunt necesari parametri de tip date. De asemenea, parametrul URL `book` poate avea orice valoare exceptând valoarea vidă. În caz de succes se va returna status code-ul 200 și obiectul de tip carte cu respectivul titlu. În caz de eroare se va returna un status code cu valoarea 404 – NOT FOUND sau cu valoarea 500 – INTERNAL SERVER ERROR. *Figura 13* care ilustrează apelul se poate vedea mai jos:

The screenshot shows a REST client interface for a GET request to the endpoint `/api/Book/book/bookTitle`. The response content type is `text/plain`. The parameters are `bookTitle` and `book`.

Parameters

Parameter	Value	Description	Parameter Type	Data Type
bookTitle	Dragostea in vremea holerei		path	string
book	book		path	string

Try it out!

Figura 13: Apelul către Get Book By Title

Al șaptelea apel este cel de Get Book By Author Name. Acest apel se găsește la endpoint-ul `/api/Book/book/author/authorName`. Verbul HTTP care se folosește este GET. Este necesar ca parametrul URL `authorName` să fie de tip string dar nu sunt necesari parametri de tip date. Ceilalți parametri URL pot avea orice valoare exceptând valoarea vidă. În caz de succes se va returna status code-ul 200 și o listă de obiecte de tip carte care au autorul menționat. În caz de eroare se va returna un status code cu valoarea 404 – NOT FOUND sau cu valoarea 500 – INTERNAL SERVER ERROR. Apelul este ilustrat mai jos în *Figura 14*:

Parameter	Value	Description	Parameter Type	Data Type
authorName	<input type="text" value="Gabriel Garcia Marquez"/>		path	string
book	<input type="text" value="book"/>		path	string
author	<input type="text" value="author"/>		path	string

Figura 14: Apelul către Get Book By Author Name

Al optulea apel este cel de Get Book By Loaner Id. Acest apel se găsește la endpoint-ul `/api/Book/book/books/user/loanerId`. Verbul HTTP care se folosește este GET. Este necesar ca parametrul URL `loanerId` să fie de tip întreg dar nu sunt necesari parametri de tip date. Ceilalți parametri URL pot avea orice valoare exceptând valoarea vidă. În caz de succes se va returna status code-ul 200 și o listă de obiecte de tip carte care au loanerId-ul specificat. În caz de eroare se va returna un status code cu valoarea 404 – NOT FOUND sau cu valoarea 500 – INTERNAL SERVER ERROR. Apelul este ilustrat în cele ce urmează în *Figura 15*:

GET `/api/Book/{book}/{books}/{user}/{loanerId}`

Response Class (Status 200)
Success

Parameter	Value	Description	Parameter Type	Data Type
loanerId	<input type="text" value="2"/>		path	integer
book	<input type="text" value="book"/>		path	string
books	<input type="text" value="books"/>		path	string
user	<input type="text" value="user"/>		path	string

Figura 15: Apelul către Get Book By Loaner Id

Al nouălea apel este cel de Get Book By Borrower Id. Acest apel se găsește la endpoint-ul `/api/Book/book/books/user/loanerId/borrowerId`. Verbul HTTP care se folosește este GET. Este necesar ca parametrul URL `borrowerId` să fie de tip întreg dar nu sunt necesari parametri de tip date. Ceilalți parametri URL pot avea orice valoare exceptând valoarea vidă. În caz de succes se va returna status code-ul 200 și o listă de obiecte de tip carte care au borrowerId-ul specificat. În

caz de eroare se va returna un status code cu valoarea 404 – NOT FOUND sau cu valoarea 500 – INTERNAL SERVER ERROR. *Figura 16* ilustrează apelul se poate vedea mai jos:

GET /api/Book/{book}/{books}/{user}/{loanerId}/{borrowerId}

Response Class (Status 200)
Success

Model

Example Value

```
{
  "id": 0,
  "name": "string",
  "description": "string",
  "isDeleted": true
},
{
  "isBorrowed": true,
  "isDeleted": true,
  "loanerId": 0,
  "user": {
    "id": 0,
    "timestamp": "2017-06-13T07:59:14.267Z",
```

Response Content Type text/plain ▼

Parameters

Parameter	Value	Description	Parameter Type	Data Type
borrowerId	<input type="text" value="3"/>		path	integer
book	<input type="text" value="book"/>		path	string
books	<input type="text" value="books"/>		path	string
user	<input type="text" value="user"/>		path	string
loanerId	<input type="text" value="2"/>		path	string

Try it out!

Figura 16: Apelul către Get Book By Borrower Id

Interfața cu utilizatorul

Home Page

Pagina inițială a aplicației, cea de home, are ca și elemente principale următoarele: în partea stângă sus un logo cu numele aplicației care are rol și de link, în partea dreaptă sus avem un meniu, în partea centrală avem un formular de logare în aplicație iar în partea de jos, centrat, avem un motto. Toate aceste elemente se pot observa în *Figura 17* de mai jos:

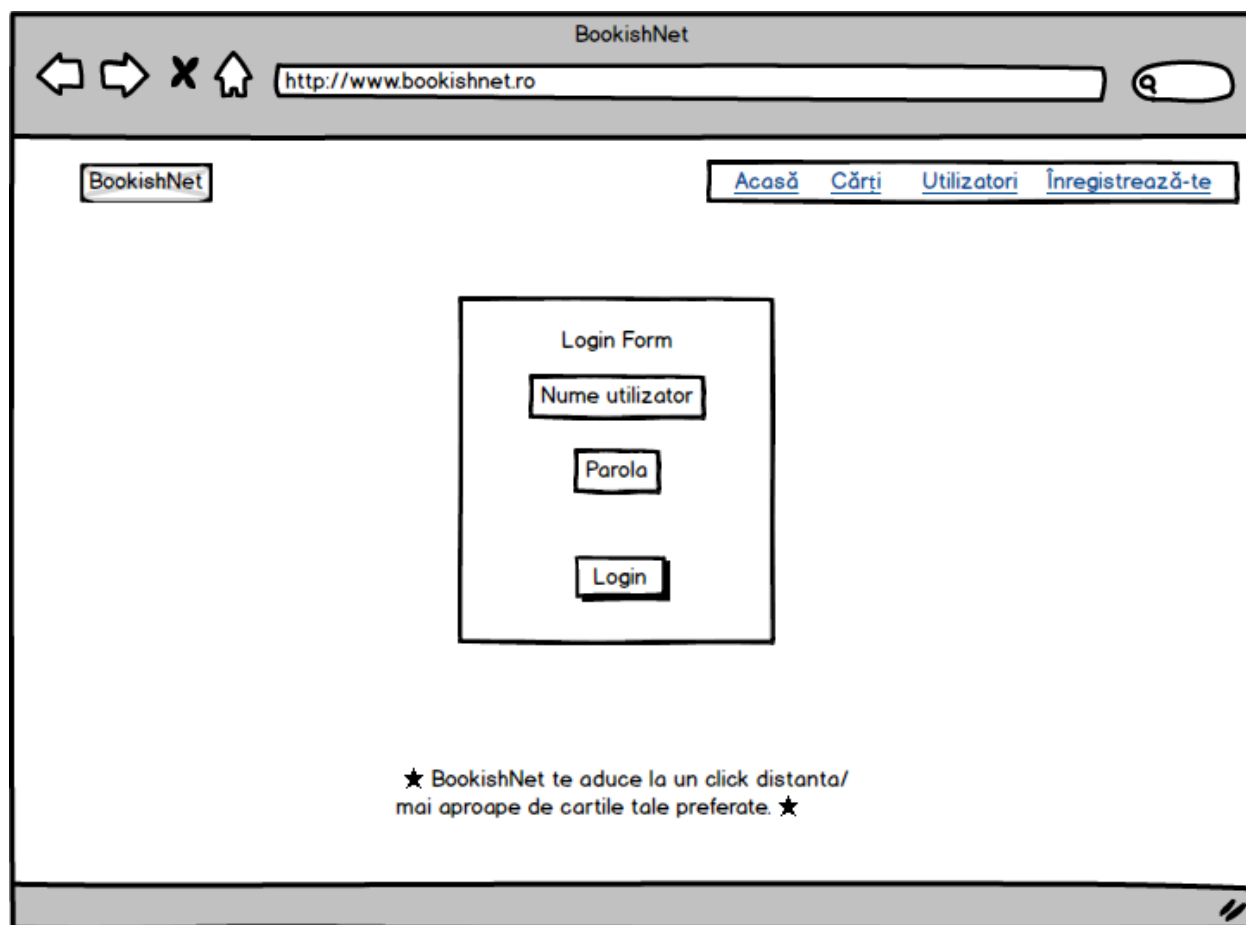


Figura 17: Home Page

Ca și interacțiuni pe care le poate face utilizatorul amintim următoarele: link-urile de pe logo și *Acasă* din meniu te aduc pe pagina inițială. Link-ul *Cărți* face redirectionarea către pagina **Books**, pagină prezentată mai jos. Link-ul *Utilizatori* face redirectionarea către pagina **Users**, care urmează a fi prezentată mai jos. Link-ul *Înregistrează-te* face redirectionarea către pagina de **Register** - următoarea ce va fi prezentată.

Formularul de logare permite și el interacțiuni. Astfel, utilizatorul va putea introduce un nume de utilizator și o parolă (preferabil ar fi să aibă și un cont existent în aplicație) în cele două input-uri ilustrate și în figura de mai sus. După completarea acestor date, el va avea posibilitatea

de a apăsa și butonul de *Login* (așa cum este ilustrat mai sus) și va fi redirecționat sau nu către o altă pagină în funcție de datele introduse.

Register

Pagina de register nu este cu mult diferită față de cea inițială. Diferența majoră se poate observa în partea centrală, unde formularul de logare de pe pagina inițială a fost înlocuit cu unul de înregistrare. Schița acestei pagini poate fi vizualizată în *Figura 18* ce urmează:

BookishNet

http://www.bookishnet.ro/register

BookishNet

Acasă Cărți Utilizatori Înregistrează-te

Register Form

Nume utilizator

Parola

Confirmare parolă

Email

☐ Înregistrare ca utilizator

☐ Înregistrare ca autor

Register

★ BookishNet te aduce la un click distanta/
mai aproape de cartile tale preferate. ★

Figura 18: Register Page

Ca și interacțiuni pe care utilizatorul le poate realiza, se poate observa clar dar am ales să menționez în mod specific acest aspect, că funcționalitatea link-urilor din meniu se păstrează, la fel și funcționalitatea link-ului de pe logo-ul aplicației. Textul din partea de jos, am omis să menționez mai sus, are doar un rol informativ. Nu are efectiv o funcționalitate ci doar impact vizual.

Comparativ cu formularul de pe pagina precedentă, formularul actual oferă mai multe interacțiuni utilizatorului. Astfel, utilizatorul are acum posibilitatea de a-și alege un nume de utilizator, o parolă, un email – pe care să le introducă în câmpurile corespunzătoare, va fi nevoit să confirme parola pe care a introdus-o, folosind câmpul de confirmare a parolei și își va putea

alege un rol, alegând unul din cele două butoane radio. După ce a introdus aceste date poate apăsa butonul de register.

Welcome Back Page

Această pagină este una dintre cele mai simple, iar după ce se realizează logarea poate fi considerată ca fiind noua pagină de home. Ca și modificări pfață de precedentele pagini, avem următoarele elemente: în partea centrală se va afișa un mesaj personalizat pentru utilizatorul care s-a logat în aplicație, iar în meniul din partea dreaptă sus dispar link-urile pentru *Acasă* și *Înregistrează-te*, ele fiind înlocuite cu un meniu de tip drop-down. Aceste elemente noi enumerate se pot vedea în *Figura 19*:

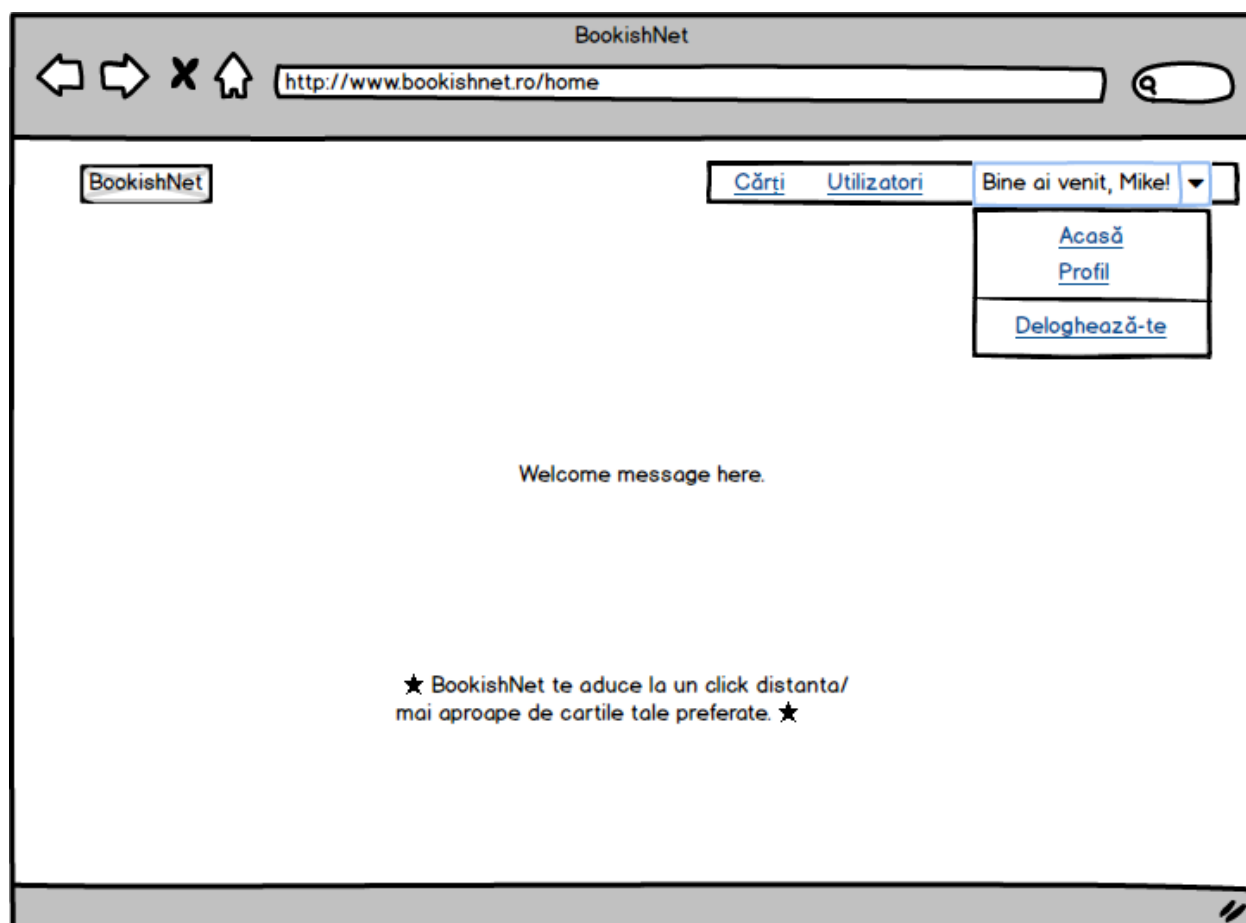


Figura 19: Welcome Back Page

Pe această pagină utilizatorul poate interacționa cu elementele rămase și prezentate în paginile precedente. În plus, în meniul de drop-down există trei opțiuni: link-ul pentru *Acasă*, link-ul pentru *Profil* și link-ul *Deloghează-te*. Primul link a fost deja prezentat în *Home Page*. Interacțiunea cu al doilea link va duce la redirecționarea către *User Page* care reprezintă, de fapt, pagina de profil a unui utilizator. Al treilea link va face redirecționarea către *Home Page* inițială, așa cum a fost prezentată mai sus.

În secțiunea centrală, unde este afișat mesajul de bun venit, nu se va oferi posibilitatea de interacțiune cu funcționalitate efectivă. De asemenea, o modificare se poate vedea și la adresa URL a paginii, aceasta fiind diferită de fiecare dată când se schimbă conținutul paginii.

Books

Această pagină poate fi vizualizată indiferent dacă utilizatorul este logat sau nu. În figură s-a ales varianta în care utilizatorul este logat. Modificările ce apar aici sunt în secțiunea centrală care este divizată în două zone: partea din stânga conține o zonă în care avem filtre după care se poate căuta o carte iar în partea dreaptă va fi afișată o listă de cărți, cu mențiunea că fiecare titlu de carte va fi în același timp și un link. Structura poate fi vizualizată în *Figura 20* de mai jos:

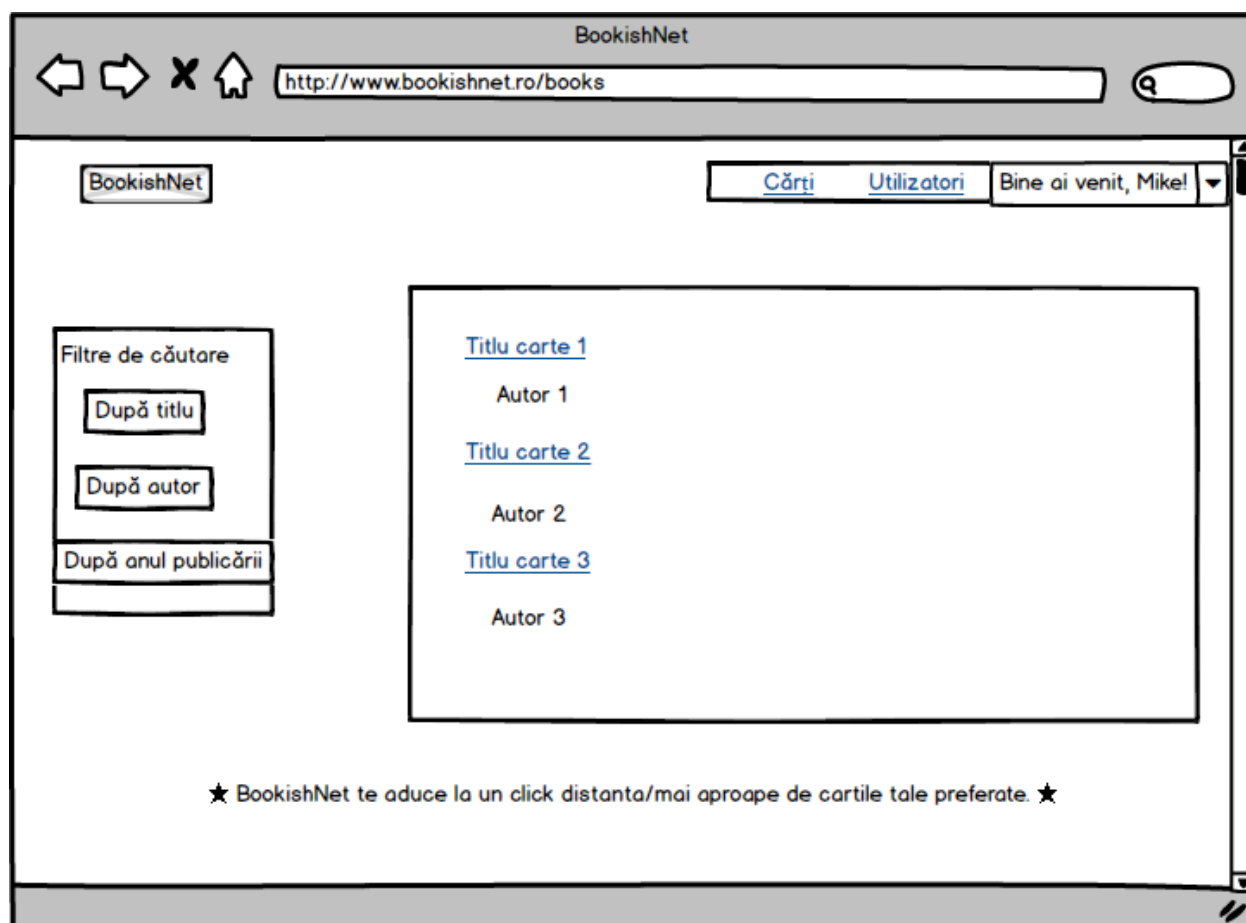


Figura 20: Books Page

Pe lângă interacțiunile care s-au prezentat deja, utilizatorul (logat sau nu) va putea să folosească filtrele din partea stângă. Astfel, el va putea să caute o carte după nume, sau după numele autorului sau după anul în care a fost publicată cartea folosindu-se de câmpurile puse la dispoziție pentru a realiza aceste operații (ilustrate în partea din stânga a figurii).

Un al doilea tip de interacțiune posibil va fi prin apăsarea link-ului disponibil pe titlul cărții după cum se poate vedea în secțiunea centrală, în partea dreaptă. În acest mod utilizatorul va putea

fi redirectionat în două direcții: către pagina cărții cu respectivul titlu (dacă este logat) sau către o pagină în care îi va fi afișat un mesaj pentru a se loga sau înregistra (acest fapt se va întâmpla doar dacă utilizatorul nu este logat).

Book Page

Pagina unei anumite cărți diferă de asemenea în secțiunea centrală. Avem astfel un container pentru această parte în care putem întâlni câteva elemente interesante: În partea superioară a container-ului se pot distinge patru butoane posibile iar în partea inferioară vor fi descrise și afișate efectiv detaliile despre acea carte. *Figura 21* ilustrează aspectul general al acestei pagini:

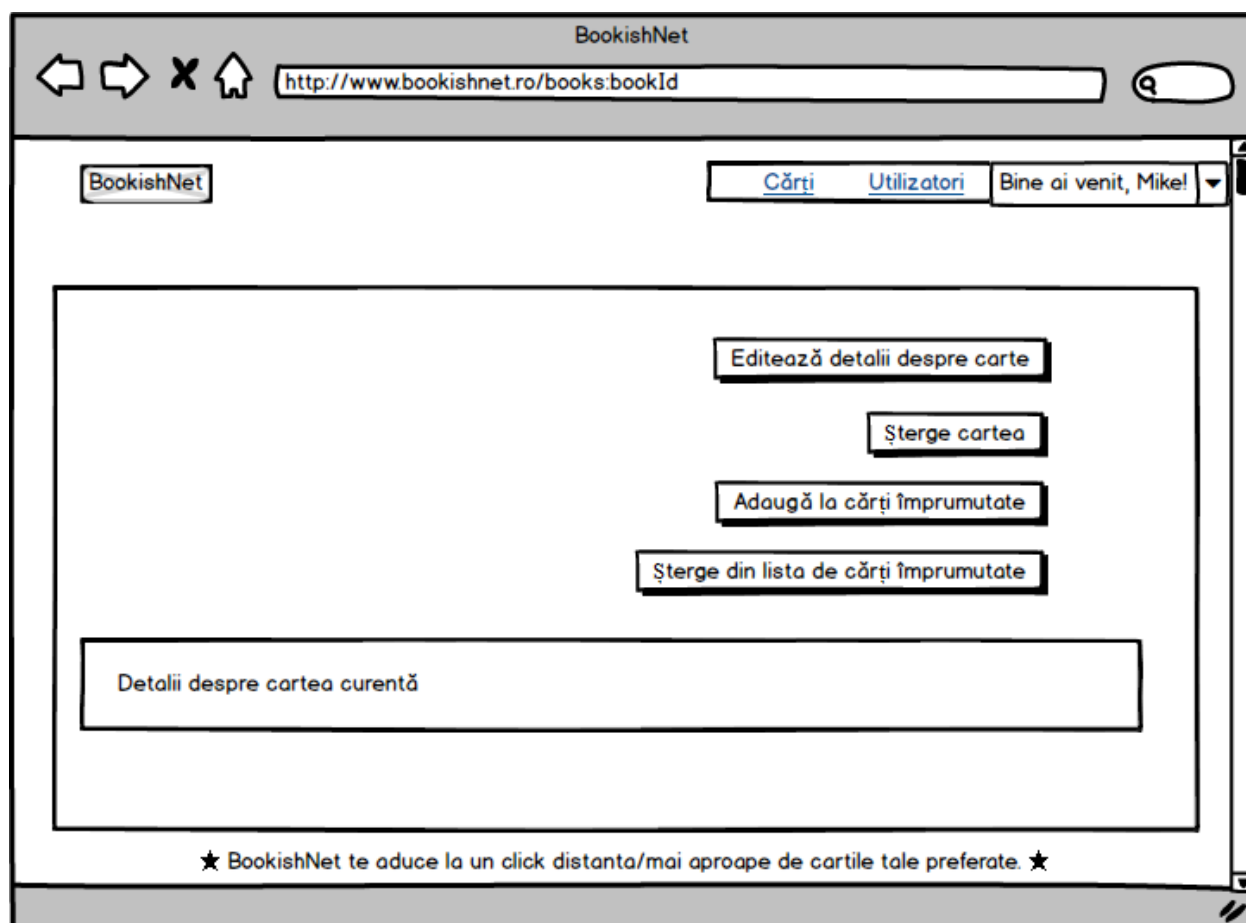


Figura 21: Book Page

Primul lucru care se merită menționat este faptul că un utilizator care nu este logat nu poate avea acces la această pagină chiar și în condiția în care modifică manual adresa URL deoarece nu îi va fi afișat conținutul. De asemenea, pentru a avea vizibile butoanele este necesar ca respectiva carte să fi fost adăugată și să-i aparțină utilizatorului actual logat.

Ca și interacțiuni posibile pentru utilizator sunt reprezentate de cele patru butoane. Astfel, primul buton va deschide un modal în care se vor putea edita detaliile despre cartea respectivă. Al

doilea buton va permite scoaterea respectivei cărți din lista cărților pe care utilizatorul le pune la dispoziție. Butonul de adăugare la cărți împrumutate permite ca respectiva carte să se adauge la o listă de cărți ce au fost împrumutate. Ultimul buton scoate cartea din lista de cărți împrumutate (vizibil doar dacă respectiva carte este în acea listă).

Users

Această pagină este într-un fel similar cu cea de **Books**. Astfel, secțiunea centrală va fi împărțită în două părți: cea din stânga și cea din dreapta. În secțiunea stângă vor exista filtre de căutare iar în partea dreaptă va fi afișată o listă cu utilizatori. Fiecare nume de utilizator va avea un link. În funcție de dimensiunea listei se va afișa și un scroll pentru pagină. Ilustrarea acestei pagini se poate vedea în cele ce urmează în *Figura 22*:

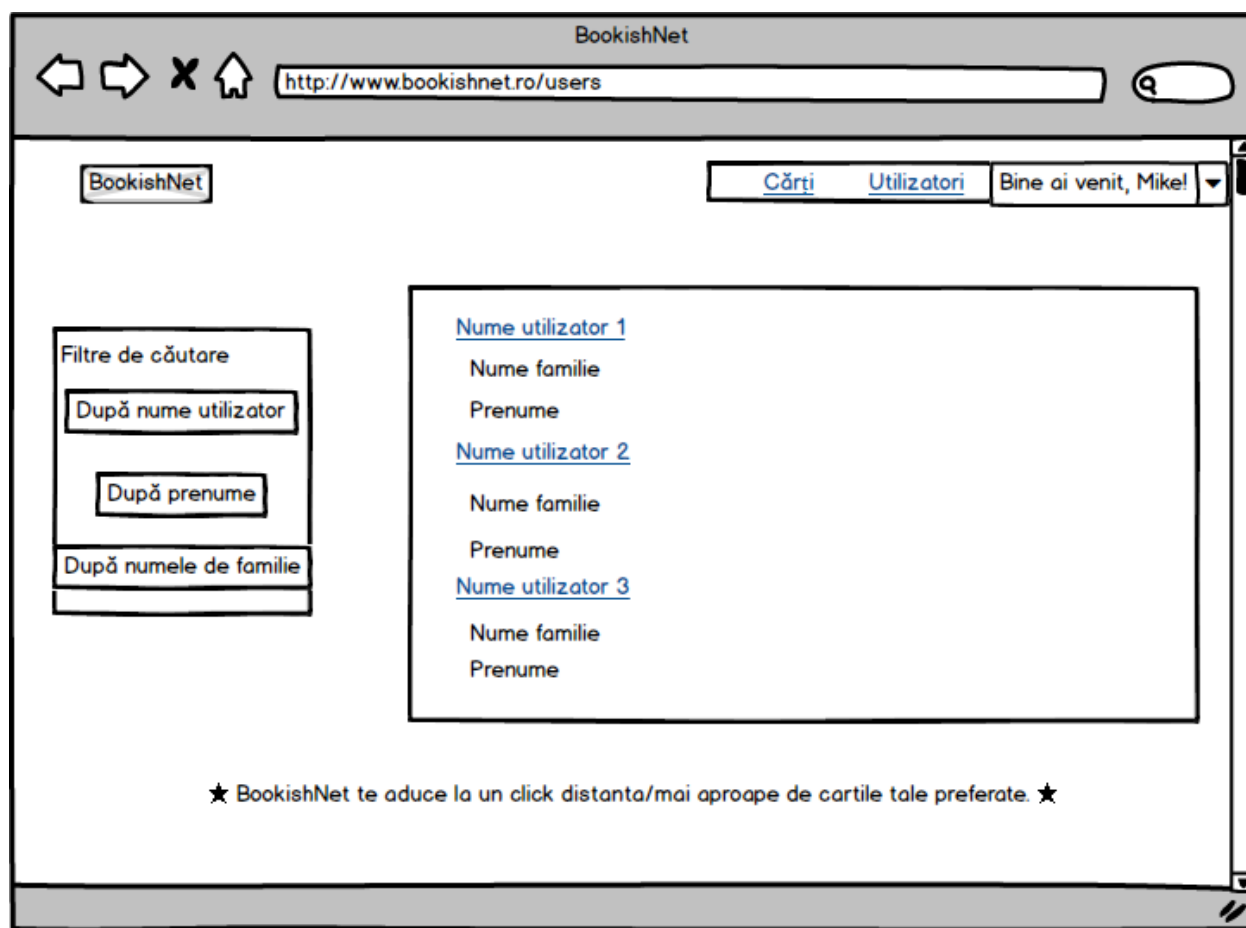


Figura 22: Users Page

Ca și interacțiuni, utilizatorul (logat sau nu) va putea să folosească filtrele din partea stângă. Astfel, el va putea să caute după numele de utilizator, sau după numele de familie al acestuia sau după prenumele acestuia folosindu-se de câmpurile puse la dispoziție pentru a realiza aceste operații (ilustrate în partea din stânga a figurii de mai sus). Rezultatele vor fi afișate în timp real, de aceea nu a fost necesar și un buton de căutare.

Un al doilea tip de interacțiune posibil va fi prin apăsarea link-ului disponibil pe numele utilizatorului după cum se poate vedea în secțiunea centrală, în partea dreaptă. În acest mod utilizatorul va putea fi redirecționat în două direcții: către pagina respectivului utilizator (dacă este logat) sau către o pagină în care îi va fi afișat un mesaj pentru a se loga sau înregistra (acest fapt se va întâmpla doar dacă utilizatorul nu este logat).

User Page

Pagina de profil al unui utilizator diferă de asemenea în secțiunea centrală. Avem astfel un container pentru această parte în care putem întâlni câteva elemente interesante: În partea superioară a container-ului se pot distinge două butoane posibile iar în continuare vor fi câteva secțiuni mai mici în care se vor putea regăsi informații despre utilizatorul respectiv sau despre cărțile puse la dispoziție de către aceștia. *Figura 23* de mai jos ilustrează aspectul general al acestei pagini:

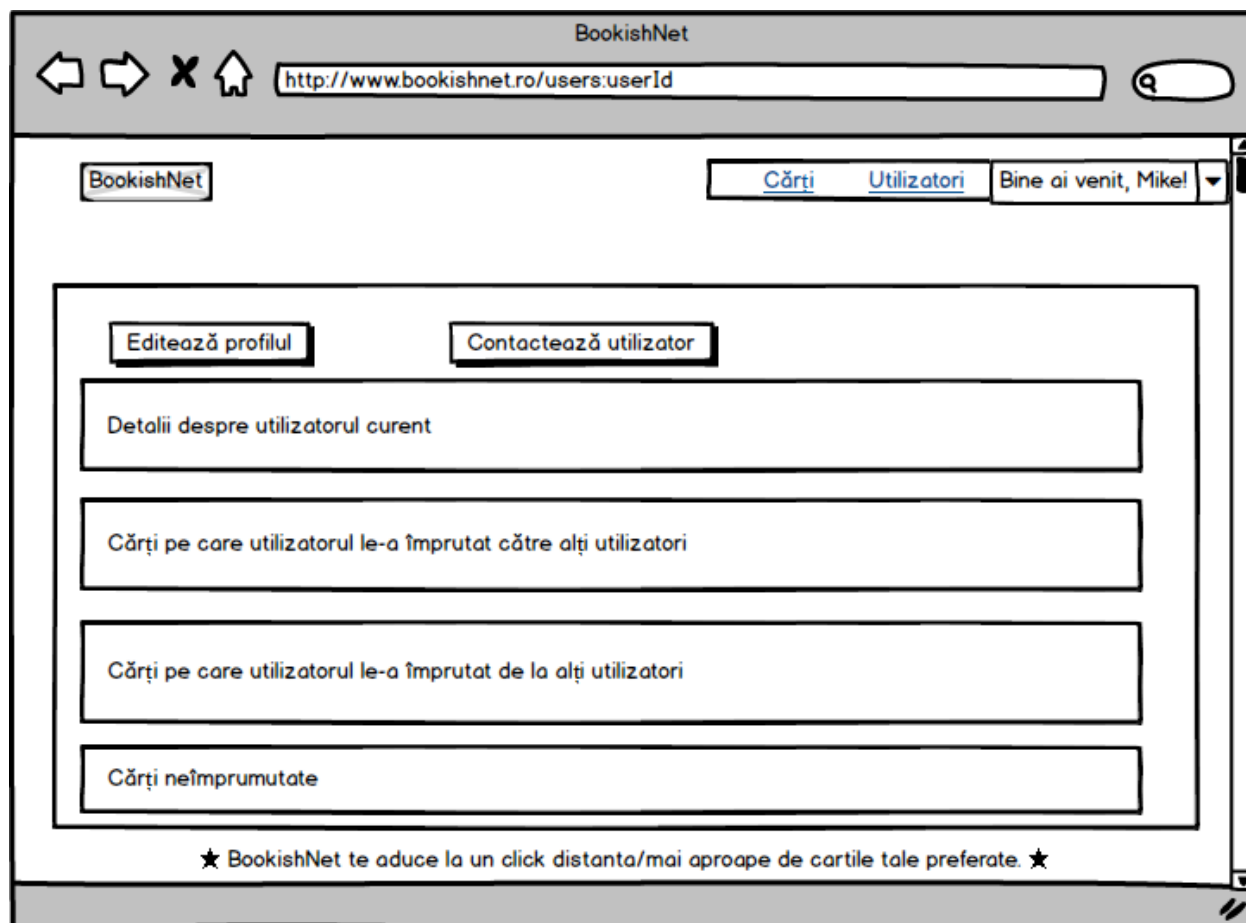


Figura 23: User Page

Cele două butoane nu vor fi afișate simultan în pagină. Astfel, dacă utilizatorul curent a intrat pe pagina sa de profil va putea interacționa cu butonul de editare care îi va deschide un modal pentru a-și edita detaliile despre profil. Dacă s-a intrat pe pagina unui alt utilizator atunci se va

putea interacționa cu butonul de contact care va deschide un modal în vederea trimiterii unui email către respectivul utilizator.

Fiecare dintre cele trei secțiuni: *Cărți împrumutate către alți utilizatori*, *Cărți împrumutate de la alți utilizatori* și *Cărți neîmprumutate* vor conține o listă de elemente. De asemenea, în aceste secțiuni vor exista link-uri pe titlul cărții și pe numele utilizatorului, acolo unde este cazul. Interacțiunea cu aceste link-uri va duce la redirecționarea către o pagină a unei cărți sau către o pagină a altui utilizator.

Implementare

Data Layer

Pentru realizarea bazei de date sau modificarea ei, Entity Framework Core oferă mai multe soluții: Database-First Approach¹² și Code-First Approach¹³. Metoda aleasă de mine a fost Code-First Approach deoarece permite realizarea bazei de date într-un mod diferit de cel tradițional unde primul pas era realizarea bazei de date în mod clasic. În același timp, aplicația se dorește a fi și una de tip business și această tehnică avantajează acest tip de aplicație.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<BookAuthor>()
        .HasKey(x => new {x.BookId, x.AuthorId});

    modelBuilder.Entity<Message>()
        .HasKey(m => new {m.SenderId, m.ReceiverId});

    modelBuilder.Entity<Book>()
        .HasOne(b => b.User)
        .WithMany(l => l.BorrowedBooks)
        .HasForeignKey(b => b.BorrowerId)
        ;

    modelBuilder.Entity<Book>()
        .HasOne(b => b.User1)
        .WithMany(l => l.LoanedBooks)
        .HasForeignKey(b => b.LoanerId);
}
```

Secțiunea de cod 1: DataLayer chei străine

Partea mai dificilă de realizat la acest nivel al aplicației a fost legat de cheile primare alcătuite din două chei străine și numirea unor chei străine astfel încât să corespundă valorile din model cu cele din tabela **Book**. În acest sens, am suprascris metoda *OnModelCreating(ModelBuilder modelBuilder)* din Entity Framework Core.

Pentru specificarea cheilor primare alcătuite din două chei străine s-a procedat după cum urmează. Pentru a putea face modificări ale configurărilor se folosește clasa *ModelBuilder*. Această clasă permite atașarea unei anumite entități pe care se dorește modificarea. În cazul de față, prima entitate a fost *BookAuthor*. Pe această entitate s-a setat proprietatea de cheie primară alcătuită din *BookId* și *AuthorId* folosind metoda *HasKey()* cum se poate vedea și în imaginea

¹² Tehnică folosită dacă baza de date este creată și avem schema acesteia. Modelele pot fi extrase folosindu-ne de schema bazei de date. Entity Framework știe să facă generarea de modele în funcție de tabelele pe care le conține baza de date.

¹³ Se creează modelele pentru crearea bazei de date și apoi se generează baza de date și schema acesteia. Cu alte cuvinte, opusul tehnicii menționate anterior.

precedentă. Pentru entitatea/modelul *Message* s-a folosit aceeași tehnică doar că diferă componența cheii primare: *SenderId* și *ReceiverId*.

De asemenea, un alt lucru care s-a rezolvat suprascriind această metodă a fost specificarea clară a numelor a două câmpuri în tabela *Books*. Entity Framework Core creea de fiecare dată încă două coloane cu chei străine denumite *UserId* și *UserId1*, proprietăți care nu erau specificate în modelul *Book*. Evitarea și repararea acestei situații a fost făcută tot cu ajutorul EF Core Fluent API¹⁴. Astfel, pe entitatea *Book* ce avea să fie trimisă pentru crearea bazei de date s-a specificat ca relația între tabela *Book* și *User* să fie de one-to-many, iar cheile străine s-au setat să aibă doar valorile *BorrowerId* și *LoanerId* ca și nume ale coloanelor în tabelă.

Data Layer Tests

Pe partea de testare a acestui nivel al aplicației s-au folosit Mock-uri așa cum am menționat și în capitolul trei. În *Secțiunea de cod 2* se poate vedea cum se declară un obiect de tip Mock. Se aseamănă cu Generics din Java. Se crează astfel variabilele cu următoarele nume: *_book*, *_bookRepository*, *_books*, *_mockContext* și *_mockSet*.

```
private Book _book;
private IBookRepository _bookRepository;
private IQueryable<Book> _books;
private Mock<BookishNetContext> _mockContext;
private Mock<DbSet<Book>> _mockSet;
```

Secțiunea de cod 2: Declararea mock-urilor

După ce s-au declarat aceste variabile se trece la următorii pași. Astfel, se instanțiază un nou obiect de tip *Mock<DbSet<Book>>*. Urmează apoi liniile în care se stabilește că ceea ce vor întoarce interogările LINQ să nu aducă date reale ci să le folosească pe cele false. Se crează astfel un fel de InMemory-Database. Acest *Setup* se realizează atât pentru Mock-ul de *DbSet* cât și pentru Mock-ul pentru *Context* al cărui metodă de *Setup* specifică să aducă datele din *DbSet*-u mock-ului și nu din baza de date. În ultimul rând, *Repository*-ul ale cărui metode se doresc a fi testate (*BookRepository* în cazul de față) vor folosi aceleași date mockuite.

```
_mockSet = new Mock<DbSet<Book>>();
_mockSet.As<IQueryable<Book>>().Setup(m => m.Provider).Returns(_books.Provider);
_mockSet.As<IQueryable<Book>>().Setup(m => m.Expression).Returns(_books.Expression);
_mockSet.As<IQueryable<Book>>().Setup(m => m.ElementType).Returns(_books.ElementType);
_mockSet.As<IQueryable<Book>>().Setup(m => m.GetEnumerator()).Returns(_books.GetEnumerator());

_mockContext = new Mock<BookishNetContext>();
_mockContext.Setup(c => c.Books).Returns(_mockSet.Object);

_bookRepository = new BookRepository(_mockContext.Object);
```

Secțiunea de cod 3: Setup pentru Mock-uri

¹⁴ Oferă metode pentru configurarea diferitelor aspecte, proprietăți referitoare la un model anume.

O metodă de test poate fi văzută în *Secțiunea de cod 4*. Metoda de față testează dacă o carte există în *_mockSet* și se apelează metoda *Delete(int Id)* acea carte va fi înlăturată. Astfel, atribuim valoarea 1 (valoare ce există în lista de cărți) variabilei *bookId*. Apoi se apelează metoda de ștergere pusă la dispoziție de *_bookRepository*. După acest pas, se apelează metoda de verificare pe *_mockSet* astfel: dacă s-a apelat metoda de *Update* pe o entitate de tip carte (nu s-a folosit metoda *Remove()* deoarece metoda de *Delete* nu înlătură obiectul din baza de date ci doar actualizează valoarea câmpului *isDeleted*). A doua verificare este pe context și anume dacă s-a apelat o dată metoda de *SaveChanges()*, metodă care se ocupă cu realizarea și finalizarea unei tranzacții.

```
[TestMethod]
public void When_DeleteIsCalledWithExistentId_Then_ThatBookShouldBeDeletedFromDatabase()
{
    var bookId = 1;
    _bookRepository.Delete(bookId);

    _mockSet.Verify(b => b.Update(It.IsAny<Book>()), Times.Once());
    _mockContext.Verify(b => b.SaveChanges(), Times.Once());
}
```

Secțiunea de cod 4: Metoda de test Delete cu id existent

MVC

Autentificarea

Pentru realizarea autentificării în aplicație am folosit metoda pusă la dispoziție de cei de la Microsoft bazată pe cookies. Astfel, în clasa *Startup* am adăugat opțiunile pentru crearea cookie-ului. *AuthenticationScheme* este folosit ca și nume/metodă de identificare pentru cookie-ul respectiv, *LoginPath* este calea relativă către care va fi redirecționat un utilizator neconectat care încearcă să obțină o resursă, *AccessDeniedPath* este calea relativă către care va fi redirecționat un utilizator care nu are autoritatea necesară pentru a accesa o anumită resursă, *AutomaticAuthenticate* este un bit ce indică faptul că middleware-ul ar trebui să ruleze pe fiecare cerere și să încerce verificarea și reconstruirea unei entități create, *AutomaticChallenge* este un bit ce indică faptul că middleware-ul ar trebui să facă redirectarea către *LoginPath* sau *AccessDeniedPath* când autentificarea eșuează. Apoi se pasează această variabilă ca și argument pentru funcția *UseCookieAuthentication*.

```
var cookieOptions = new CookieAuthenticationOptions
{
    AuthenticationScheme = "BookishNetCookie",
    LoginPath = new PathString("/account/login"),
    AccessDeniedPath = new PathString("/Account/Forbidden/"),
    AutomaticAuthenticate = true,
    AutomaticChallenge = true
};

app.UseCookieAuthentication(cookieOptions);
```

Secțiunea de cod 5: *CookieAuthenticationOptions*

În *Secțiunea de cod 5* s-au prezentat pașii pentru inițializarea cookie-ului. În cele ce urmează se vor prezenta următorii pași. Pentru a putea realiza autentificarea s-a creat o listă de *claim-uri*¹⁵. Se încearcă apoi aducerea din baza de date a utilizatorului cu respectivul nume solicitat apoi se verifică credențialele. Dacă acestea sunt corecte, se adaugă la lista de claim-uri numele utilizatorului, apoi se face un apel pentru preluarea rolului respectivului utilizator, după care se adaugă și acest rol la lista de claim-uri. Se stabilește apoi o valoare pentru identitatea utilizatorului și se adaugă și aceasta la lista de claim-uri după care se crează variabila *userPrincipal* care va folosi această identitate. Deoarece se dorește a fi o metodă asincronă, se adaugă la sfârșitul funcției *await* unde se face autentificarea asincronă folosind cookie-ul aplicației și *userPrincipal* de mai sus. De asemenea, tot aici se realizează anumite proprietăți de autentificare: după cât timp expiră acest cookie, dacă permite operația de refresh din browser, dacă este persistent.

¹⁵ O pereche nume-valoare ce reprezintă ce anume este subiectul și nu ce poate face acesta.


```

[HttpPost]
[Route("login")]
public async Task<IActionResult> Login([FromBody] LoginCredentialsDTO loginCredentials, string returnUrl = null)
{
    var claims = new List<Claim>();
    var user = _userService.GetUserByUsername(loginCredentials.Username);
    if (!Utility.CheckPassword(loginCredentials.Password, user.Password))
        return UserNotFound();
    claims.Add(new Claim(ClaimTypes.Name, loginCredentials.Username, ClaimValueTypes.String));

    var role = _roleService.GetRoleOfUser(user);

    //Added Claim.Role
    claims.Add(new Claim(ClaimTypes.Role, role, ClaimValueTypes.String));
    var userIdentity = new ClaimsIdentity("BookishNetSecureLogin");
    userIdentity.AddClaims(claims);
    var userPrincipal = new ClaimsPrincipal(userIdentity);

    await HttpContext.Authentication.SignInAsync("BookishNetCookie", userPrincipal,
        new AuthenticationProperties
        {
            ExpiresUtc = DateTime.UtcNow.AddMinutes(5),
            IsPersistent = false,
            AllowRefresh = false
        });

    return RedirectToLocal(returnUrl, role);
}

```

Secțiunea de cod 6: Metoda asincronă de logare în aplicație

Trimitere email

O altă funcționalitate interesantă a aplicației este că permite trimiterea de email-uri. Astfel, utilizatorii vor putea comunica între ei folosind editorul simplificat din interiorul aplicației. În cele ce urmează se poate vizualiza metoda care realizează această operație, *SendEmail*:

```

[HttpPost]
public void SendEmail([FromBody] Email email)
{
    var emailMessage = new MimeMessage();

    emailMessage.From.Add(new MailboxAddress(email.Username, email.Sender));
    emailMessage.To.Add(new MailboxAddress("", email.Receiver));
    emailMessage.Subject = email.Subject;
    emailMessage.Body = new TextPart("plain") {Text = email.Body};

    using (var client = new SmtpClient())
    {
        //client.ServerCertificateValidationCallback = (s, c, h, e) => true;

        client.Connect("smtp.gmail.com", 587, false);

        // Note: since we don't have an OAuth2 token, disable
        // the XOAUTH2 authentication mechanism.
        client.AuthenticationMechanisms.Remove("XOAUTH2");

        //TODO: remove this hardcoded email and password
        // Note: only needed if the SMTP server requires authentication
        client.Authenticate("NTTWorkNETProfessional@gmail.com", "passwordfornttinternship");

        client.Send(emailMessage);
        client.Disconnect(true);
    }
}

```

Secțiunea de cod 7: Funcția de trimitere email

Pentru realizarea acestei operații s-a folosit biblioteca *MimeKit*, ce permite prelucrarea și trimiterea informațiilor într-un mod mai simplu dar și consistent în același timp. S-a apelat constructorul pentru crearea unei noi instanțe a clasei *MimeMessage*. Pentru adăugarea destinatarului, expeditorului, subiectului și a conținutului mesajului s-au folosit metodele puse la dispoziție de clasa amintită mai sus. Aceasta a fost faza de pregătire a trimiterii informațiilor dorite.

S-a folosit apoi clasa *SmtplibClient* pentru a realiza transmiterea datelor. Astfel, ne-am conectat la un client de tip gmail, folosind portul 587. S-a înlăturat mecanismul de autentificare *OAuth2* deoarece nu folosim un asemenea token, apoi s-a apelat metoda de autentificare pentru un cont de gmail generic folosit în cadrul aplicației. S-a realizat apoi trimiterea email-ului prin apelarea metodei *Send*, după care am deconectat clientul conectat mai sus. [2]

Directiva pentru confirmarea parolei

Structura unei directive, dar și a unui serviciu sau controller Angular creat în Visual se poate vedea în *Secțiunea de cod 8*. Elementele commune tuturor sunt numele modulului setat cu apelul către `module()`. Apoi, se face apelul către `directive()` unde se setează numele directivei. Pentru controller și service se vor apela `controller()`, respectiv `service()`. [3]

Pentru accesarea anumitor servicii ce se doresc a fi folosite în interiorul corpului directivei se folosește *dependency injection* prin intermediul `$inject`. Totodată, această metodă de injectare folosește și la păstrarea suportului pentru *dependency injection* după ce codul este minimizat.

Am creat o variabilă pe care am denumit-o `directive`, asupra căreia am setat anumite atribute: opțiunea `link` (va fi detaliată mai jos), restrict cu atributul “EA”, acest fapt semnifică acțiunea directivei doar pe elemente sau pe atributul unui element, iar la `require` este setat controllerul care folosește `ngModel`. [4]

O directivă care folosește opțiunea `link` poate înregistra anumiți listener pe DOM sau chiar poate face modificări asupra acestuia. De asemenea, această opțiune folosește funcția `link` cu următoarea semnătură în cazul de față: *function link(scope, element, attrs, ctrl){}*. De pe `scope` se apelează funcția `$watch` căreia îi sunt transmiși ca parametri atributul pe care să îl urmărească („`passwordMatchDirective`” în cazul de față) și o funcție care joacă rol de listener între valoarea actuală (din câmpul de confirmare a parolei) și cea precedentă (din câmpul parolei). [5]

Validarea celor două câmpuri se face prin intermediul funcției `ctrl.$validators.match(modelValue, viewValue)`. În acest punct se compară dacă valoarea din câmpul de parolă coincide cu valoarea din câmpul de confirmare. Dacă evaluarea condiției este adevărată se returnează `true`, altfel se returnează `false`, după care se validează acest răspuns și pe controller astfel încât monitorizarea să înceteze dacă s-a ajuns la potrivire și să continue dacă nu există potrivire. [6]

```

(function() {
  "use strict";

  angular
    .module("BookishNet")
    .directive("passwordMatchDirective", passwordMatchDirective);

  passwordMatchDirective.$inject = ["$window"];

  function passwordMatchDirective($window) {
    // Usage:
    //   <passwordMatchDirective></passwordMatchDirective>
    // Creates:
    //
    var directive = {
      link: link,
      restrict: "EA",
      require: "ngModel"
    };
    return directive;

    function link(scope, element, attrs, ctrl) {
      scope.$watch(attrs["passwordMatchDirective"],
        function(newVal, oldVal) {
          ctrl.$validators.match = function(modelValue, viewValue) {
            if (newVal === modelValue) {
              return true;
            } else {
              return false;
            }
          };
          ctrl.$validate();
        });
    }
  }
})();

```

Secțiunea de cod 8: Directiva pentru confirmarea parolei

BookService

Pentru a exemplifica mai bine cum funcționează un serviciu în AngularJS am ales să folosesc două metode: *getBooks* și *addBook*. În acest fel, voi exemplifica modul în care se comportă un serviciu AngularJS atât pentru o funcție cu parametri dar și pentru una care nu are. Celelalte metode existente în *bookService* sau orice alt serviciu au comportament similar.

GetBooks

Deși în *Secțiunea de cod 9* nu se poate observa, la fel ca și în *Secțiunea de cod 8*, pentru a folosi un anumit serviciu Angular (creat de noi sau existent implicit în framework) este nevoie de injectarea respectivului serviciu folosind proprietatea *\$inject*. Cele trei servicii Angular injectate

sunt *\$http*, *\$location* și *\$q*, pe care le-am trimis ca și parametri funcției *bookService()* (simulează într-un fel comportamentul unei constructor doar că nu creează obiecte noi, ci se aplică unui obiect de tip *scope*) pentru a le putea folosi în interiorul acesteia.

```
function bookService($http, $location, $q) {
  this.getBooks = function() {
    var deferred = $q.defer();
    $http({
      method: "GET",
      url: "http://localhost:45719/api/book"
    })
    .then(function successCallback(response) {
      return deferred.resolve(response);
    }, function failCallback(response) {
      return deferred.reject(response.data);
    });
    return deferred.promise;
  };
};
```

Secțiunea de cod 9: Metoda getBooks din bookService

În cele ce urmează voi detalia cele trei servicii folosite. Serviciul *\$http* este un serviciu important din AngularJS care facilitează comunicarea între serverele HTTP prin intermediul browsere-lor folosind obiecte XMLHttpRequest sau prin intermediul JSONP. Serviciul *\$q* este un serviciu care ajută la rularea funcțiilor în mod asincron și folosește valorile (sau excepțiile) returnate de aceste funcții după ce acestea își încheie execuția. Serviciul *\$location* parsează URL-ul afișat în browser (se bazează pe același principiu ca *window.location* din Javascript) și cu ajutorul lui se poate prelua URL-ul aplicației. O schimbare în URL se reflectă în serviciul *\$location* și o schimbare în serviciul *\$location* se reflectă în URL-ul afișat de browser.

Pentru preluarea tuturor cărților disponibile în baza de date, am creat funcția *getBooks()*. Am creat variabila denumită *deferred* iar apelul către *\$q.deferred()* semnifică faptul că această variabilă va primi ca și valoare un task ce se va executa în mod asincron.

Apoi s-a pregătit cererea HTTP, apelând serviciul *\$http*. Am setat metoda cererii ca fiind de tip GET și url-ul către <http://localhost:45719/api/book>, adresă la care se găsește resursa dorită. Metoda *.then()* se folosește de *deferred/promise* Api expus de către serviciul *\$q*.

Astfel, în caz de succes vom returna un promise primit de la funcția *deferred.resolve()*. În caz de eroare la trimiterea răspunsului pentru cererea făcută *deferred.reject()* va returna un promise de respingere a cererii. În final, vom returna promise-ul obținut; fie de succes, fie de respingere.

UpdateBook

Cealaltă metodă aleasă pentru detaliere este reprezentată de *updateBook()*, al cărei cod se poate observa în figura de mai jos. Întrucât cele două metode sunt în proporție de aproximativ 90% similare, în cele ce urmează voi încerca să pun în evidență doar deosebirile.

```
this.updateBook = function(book) {
  var deferred = $q.defer();
  var bookId = $location.url().split(":")[1];
  $http({
    method: "PUT",
    url: "http://localhost:45719/api/book/" + bookId,
    data: JSON.stringify(book),
    headers: {
      'Content-Type': "application/json"
    }
  })
  .then(function successCallback(response) {
    return deferred.resolve(response);
  }, function failCallback(response) {
    return deferred.reject(response.data);
  });
  return deferred.promise;
};
```

Secțiunea de cod 10: Metoda *updateBook* din *bookService*

Deoarece la modificarea unui obiect este necesar id-ul respectivului obiect, am creat o variabilă denumită *bookId* care se folosește de serviciul *\$location* pentru a prelua id-ul. Astfel, am apelat metoda *url()* disponibilă prin intermediul serviciului AngularJS și apoi am apelat funcția *split()* folosind ca separator „:”, [1] reprezintă că iau valoarea de după acest separator. În cazul în care aș fi folosit [0] aș fi luat valoarea de dinainte de separator.

În serviciul *\$http* apar și aici modificări. Astfel, verbul HTTP folosit pentru parametrul *method* este PUT. Apoi, se folosește parametrul *data* pentru a trimite obiectul dorit. Am folosit metoda *JSON.stringify(book)* pentru a transforma obiectul de tip javascript *book* într-un obiect de tip string JSON. În cele din urmă, am setat pentru parametrul *headers* ca headerul HTTP pe care serviciul îl trimite să fie de tip *application/json*.

În acest fel, am ales o variantă mai rapidă pentru realizarea encodării și decodării datelor transmise, JSON fiind și un format ușor de citit. Celelalte funcții sau variabile care nu s-au prezentat exclusiv au același comportament precum cele din funcția *getBooks()*, prezentată anterior.

LoginController

În cele ce urmează voi prezenta unul dintre controllere-le Angular. Întrucât are dimensiunea cea mai mare ca și întindere și este chiar primul controller prezent în aplicație (acționează pe prima pagină a aplicației), am hotărât să prezint structura acestui controller.

După cum am spus și în subcapitolul Angular Controllers, am încercat, pe cât posibil, ca fiecare view să conțină un singur controller. Astfel, pagina de start a aplicației este manageriată de *loginController*. Prima parte a codului se poate vedea în *Secțiunea de cod 11*:

```
function loginController($rootScope, $scope, $location, loginService, userService) {
    if (sessionStorage.getItem("session") === null) {

        var emptySession = {
            'id': "",
            'username': "",
            'role': "",
            'isLoggedIn': false
        };
        sessionStorage.removeItem("session");
        sessionStorage.setItem("session", JSON.stringify(emptySession));
    }
    $rootScope.sessionData = JSON.parse(sessionStorage.getItem("session"));
    /* jshint validthis:true */
    var log = this;
    this.username = "";
    this.role = "";
    this.isLoggedIn = false;
    log.show = false;
    log.logging = false;
    log.title = "loginController";
}
```

Secțiunea de cod 11: loginController partea 1

În acest controller am injectat următoarele obiecte, servicii Angular dar și servicii create de mine: *\$rootScope*, *\$scope*, *\$location*, *loginService* și *userService*. Fiecare aplicație are un singur scope rădăcină (*\$rootScope*). Toate celelalte scope-uri sunt descendenți ai rădăcinei. Obiectele de tip scope oferă sau asigură separarea între model și view, folosind un mecanism de monitorizare a modelului pentru detectarea schimbărilor survenite între timp.

Întrucât pe partea de backend am folosit o autentificare bazată pe cookie-uri, care stabilește legătura între client și server, am realizat că este nevoie și de o metodă de a ține memorat acest fapt și pe partea de client Angular. În acest fel, am putut să restricționez accesul către anumite resurse, pagini sau acțiuni.

Astfel, cu aceste detalii în minte, am căutat o soluție și pentru partea de Angular și am ajuns la concluzia că una din metodele disponibile este cea bazată pe *sessionStorage*, proprietate a

browser-ului de a accesa un anumit obiect din sesiunea curentă, obiect ce este stocat până la închiderea sesiunii, a browser-ului.

Primul pas a fost verificarea existenței unui obiect în sesiune prin intermediul `sessionStorage.getItem("session")`. Dacă nu exista acest obiect, se creaa obiectul `emptySession` cu parametrii:

- `id` – pentru reținerea identicatorului utilizatorului
- `username` – pentru stocarea numelui de utilizator
- `role` – pentru stocarea rolului utilizatorului
- `isLoggedIn` – pentru stocarea faptului că utilizatorul este logat sau nu

În continuare, am setat acest obiect în sesiune (după ce l-am transformat într-un obiect de tip `string` `JSON` folosind funcția `JSON.stringify`) cu ajutorul următoarei linii: `sessionStorage.setItem("session", JSON.stringify(emptySession));`. Apoi am preluat acest obiect și l-am adăugat pe `$rootScope`: `$rootScope.sessionData = JSON.parse(sessionStorage.getItem("session"))`. Am apelat funcția `JSON.parse()` pentru parsarea string-ului `JSON` aflat în sesiune. Am ales să pun acest obiect pe `$rootScope` pentru a nu apela din fiecare controller `sessionStorage`, această metodă fiind mai convenabilă dat fiind faptul că `$rootScope`-ul poate fi expus în fiecare controller și astfel avem acces mai rapid la resursa necesară.

Deoarece acest controller are un constructor, am ales să folosesc variabila `log` pentru stabilirea mai precisă a numelui controller-ului, variabilă în care am preluat `this`. De asemenea, s-a mai creat alte variabile ajutătoare:

- `log.role` – pentru reținerea rolului
- `log.isLoggedIn` – setat implicit pe false
- `log.show` – pentru afișarea erorii „Combinație greșită nume utilizator/parolă”. Setat implicit pe false
- `log.logging` – folosită pentru afișarea unui element ce arată faptul că cererii este în curs de procesare după solicitarea logării în aplicație. Setat implicit pe false
- `log.title` – stochează numele controllerului

În Secțiunea de cod 12 se poate vedea efectiv metoda de logare în aplicație. Primul pas este setarea pe `true` a variabilei `log.logging` pentru notificarea utilizatorului cu privire la încărcarea cererii. Apoi, se setează mesajul ce îi va fi afișat utilizatorului în timpul procesării cererii prin intermediul `log.loadingLogin`. Se declară apoi variabila `username` care primește ca valoare obiectul trimis pe scope prin intermediul `log.username`, același lucru fiind similar și în cazul variabilei `password`. Se creează apoi obiectul `dto`, folosind variabilele `username` și `password`.

```

log.login = function() {
  log.logging = true;
  log.loadingLogin = "Incercam autentificarea. Va rugam asteptati";
  var username = log.username;
  var password = log.password;
  var dto = {
    "Username": username,
    "Password": password
  };
  loginService.login(dto)
    .then(function(response) {
      if (response.data !== null) {
        log.username = dto.Username;
        var responseJson = JSON.parse(JSON.stringify(response.data));
        if (responseJson[0] === "authenticated") {
          log.isLoggedIn = true;
          log.role = responseJson[1];
          userService.getUserDetails(log.username)
            .then(function(user) {
              log.id = user.data.id;
              var session = {
                'id': log.id,
                'username': log.username,
                'role': log.role,
                'isLoggedIn': log.isLoggedIn
              };
              log.show = false;
              log.logging = false;
              sessionStorage.removeItem("session");
              sessionStorage.setItem("session", JSON.stringify(session));
              $rootScope.sessionData = JSON.parse(sessionStorage.getItem("session"));
              $location.path("/home");
            });
        } else {
          log.show = true;
          log.logging = false;
          log.error = "Combinatia nume utilizator/parola este eronata. Va rugam reincercati.";
        }
      }
    });
};

```

Secțiunea de cod 12: loginController partea a 2-a

Având toate aceste pregătiri urmează apelul către *loginService* prin intermediul funcției *login(dto)* care primește ca parametru obiectul *dto* creat anterior. Fiind o cerere asincron nu vom primi avem nevoie de apelul către funcția *.then()* care ne asigură faptul că își va continua execuția după primirea promise-ului menționat și la *bookService*.

După primirea răspunsului, a promise-ului verificăm că există date primite. În caz contrar, lucrurile se termină repede. Execuția funcției se termină aici, fapt ce semnifică fie producerea unei erori la apelul serviciului fie returnarea unei promisiuni de respingere.

Pe de altă parte, dacă evaluarea condiției este adevărată se execută mai mulți pași. Setăm valoarea pentru *log.username* cu valoarea din *dto.Username*. Facem o parsare a răspunsului primit și îl reținem în variabila *responseJson*. Dacă prima parte a răspunsului are același tip și are valoarea “authenticated”, urmează o nouă succesiune de pași ce urmează a fi prezentați în următorul paragraf.

Se setează pe *true* valoarea pentru *log.isLoggedIn*. Se preia rolul utilizatorului în *log.role* din *responseJson[1]*. Se apelează apoi funcția *getUserDetails(username)* cu parametrul *log.username* din *userService*. Se apelează din nou funcția *.then()* pentru așteptarea promise-ului. După primirea acestuia, se setează *log.id* cu valoarea corespunzătoare primită ca răspuns. Se creează un obiect javascript *session*. Se setează cele patru proprietăți necesare pentru stocarea în sesiune. Se setează *log.show* și *log.loading* pe false. Se înlătură obiectul creat inițial în sesiune și

se adaugă noul obiect folosind *session* apoi se preia valoarea în *\$rootScope* și se redirecționează browserul către */home*.

În caz contrar, lucrurile se termină repede. Se setează *log.show* pe *true* pentru a notifica view-ul că trebuie să afișeze mesajul de eroare, setăm *log.logging* pe *false* ca să oprim notificarea referitoare la încărcarea cererii și apoi setăm un mesaj pentru *log.error*, mesaj ce va fi afișat în view, pe pagina de logare. [7]

View binding

Legătura între controller și view se realizează prin așa numitul binding sau double-binding numit în Angular. Ce vrea să spună aceasta este că o modificare/modificări în view determină o modificare/modificări în controller dar și că o modificare/modificări în controller determină o modificare/modificări sau modificări în view.

Mai jos, voi exemplifica binding-ul dinspre view către controller. Anterior a fost prezentat și binding-ul de la controller către view, deși nu am menționat în mod explicit. Pentru a realiza aceasta voi folosi *BookEditor.html* view ce poate fi văzut în *Secțiunea de cod 13*:

```
<div class="modal-body">
  <form role="form" class="css-form" name="edit" ng-submit="book.updateBook(edit)" novalidate>
    <div class="row">
      <div class="form-group col-xs-6">
        <label for="bookTitle">Titlu carte: </label>
        <input type="text" class="form-control" placeholder="Titlu carte" name="title" ng-model="book.title" id="bookTitle" required="">
        <span class="help-block" ng-show="edit.title.$touched && edit.title.$error.required">Titlul nu poate fi null.</span>
      </div>
      <div class="form-group col-xs-6">
        <label for="genre">Alegeti genul cartii: </label>
        <input type="text" class="form-control" ng-model="book.genre" name="genre" placeholder="Introduceti genul aici" list="genres" required="">
        <span class="help-block" ng-show="edit.genre.$touched && edit.genre.$error.required">Genul cartii este necesar.</span>
      </div>
      <datalist id="genres">
        <option ng-repeat="genre in book.genreList | filter:genre.id" value="{{genre.name}}">
      </datalist>
    </div>
    <div class="row">
      <div class="form-group col-xs-12">
        <label for="description">Descriere: </label>
        <textarea class="form-control" placeholder="Descriere" name="details" id="description" ng-model="book.description"></textarea>
      </div>
    </div>
    <div class="row">
      <div class="form-group col-xs-6">
        <label for="author">Nume autor: </label>
        <input type="text" class="form-control" placeholder="Nume autor" name="authorName" ng-model="book.authorName" id="author" required="">
        <span class="help-block" ng-show="edit.authorName.$touched && edit.authorName.$error.required">Numele autorului este necesar.</span>
      </div>
      <div class="form-group col-xs-6">
        <label for="year">Anul publicării: </label>
        <input type="number" class="form-control" name="publishingYear" min="1" max="2017" ng-model="book.publishingYear" id="year" required="">
        <span class="help-block" ng-show="edit.publishingYear.$touched && edit.publishingYear.$error.required">Anul publicării este necesar.</span>
      </div>
    </div>
    <button type="submit" class="btn btn-success" ng-disabled="edit.$invalid">Salveaza modificarile</button>
    <button class="btn btn-default" class="close" data-dismiss="modal" aria-label="Close" ng-click="book.cancelEdit(edit)">Anuleaza editarea</button>
  </form>
</div>
```

Secțiunea de cod 13: Exemplificare view binding

Pentru o putea urmări mai ușor cele ce voi urma a spune, vă sugerez să urmăriți culoarea roz din imaginea precedentă. Voi face mai întâi o scurtă descriere general și apoi voi trece la elementele specific folosirii Angular-ului.

Avem aici un formular de editare a unei cărți care se deschide într-un modal. Acest formular conține patru input-uri în care se pot introduce valori de la tastatură, un element asemănător unui meniu dropdown de unde poți alege o opțiune, o zonă de text, mai mare decât un input, unde de asemenea se pot introduce valori de la tastatură, un buton de submit și unul de anulare a operației. [5]

Elementele specific pentru Angular se pot identifica cel mai ușor dacă se urmărește prefixul *ng-* sau parantezele duble *{{ }}* (folosite în mod special pentru binding). Toate atributele care încep cu *ng-* sunt, de fapt, directive Angular. Se pot observa: *ng-submit*, *ng-show*, *ng-click*, *ng-repeat*, *ng-model*.

Directiva *ng-submit* este folosită pentru a specifica acțiunea ce se va realiza la trimiterea formularului. Astfel, la realizarea acestei acțiuni se va apela funcția *book.updateBook(edit)*. Deoarece această funcție nu a fost creată pe scope este necesară folosirea și a variabilei *book* care reprezintă, de fapt, *bookController*.

Directiva *ng-show* este folosită pentru a transmite către browser ce elemente se vor afișa în pagină și care nu. De menționat faptul că ele se creează în DOM, dar nu se afișează efectiv. În cazul de față această directivă a fost folosită pentru afișarea anumitor texte ajutătoare cu privire la completarea corectă a formularului. Unele câmpuri sunt obligatorii, altele nu.

Directiva *ng-click* este folosită pentru a specifica acțiunea ce se va realiza la apăsarea butonului de anulare a operației. Astfel, la realizarea acestei acțiuni se va apela funcția *book.cancelBook(edit)*. Deoarece nici această funcție nu a fost creată pe scope este necesară folosirea și a variabilei *book* care reprezintă, de fapt, *bookController*.

Directiva *ng-repeat* este folosită pentru a introduce de obicei o listă de elemente și este și preferata mea. Nu este restricționată de tipul elementului ce se dorește a fi introdus. În cazul de față *ng-repeat* creează mai multe opțiuni pentru elementul *datalist*. Ea funcționează asemănător cu o instrucțiune *for*. În cazul de față, se ia câte un *genre* din *book.genreList* și se face o filtrare după id-ul genului. Elementele ce vor fi afișate vor lua valoarea dată de *{{genre.name}}*. În funcție de dimensiunea *book.genreList* vor fi și atâtea opțiuni disponibile. Binding-ul se face cu ajutorul parantezelor *{{ }}*. [8]

Directiva *ng-model* este folosită pentru binding-ul bidirecțional. Astfel, avem următoarele variabile: *book.title*, *book.genre*, *book.description*, *book.authorName*, *book.publishingYear*. Aceste variabile fac legătura între *bookController* și view-ul prezentat aici. Astfel, când utilizatorul introduce o valoare într-unul din câmpurile care au atributul *ng-model*, respective variabilă va fi preluată și de către controller, în timp real. În acest fel, orice modificare din view este transmisă către controller și ținând cont și de cealaltă posibilitate, și anume de a transmite date/valori din controller către view, se poate așadar spune că binding-ul realizat cu ajutorul Angular-ului este unul bidirecțional, fapt care mie mi se pare foarte util.

MVC Tests

Pentru testarea aplicației web am folosit în principal metoda de testare automată. Astfel, s-au creat diferite test case-uri pentru verificarea anumitor funcționalități. Fiecare test case a reprezentat scrierea unei metode care să execute anumiți pași. Un exemplu de test case este următorul: se deschide browser-ul. Se introduce URL-ul aplicației. Se verifică dacă este prezent butonul de logare pentru validarea testului. Metoda de abordare și câteva exemple concrete se vor găsi în subcapitolele ce urmează: *Page Object Design Pattern*, *Browser Factory* și *Page Generator*. [9]

Page Object Design Pattern

Acest pattern modelează zone ale unei interfețe utilizator ca obiecte din cadrul codului de testare, care poate fi considerat un Object Repository pentru elementele web din interfață. Clasele funcționale, denumite *PageObjects* în acest pattern, reprezintă o relație logică între paginile aplicației. Fiecare clasă este referențiată ca un *PageObject* și returnează alte *PageObjects* pentru a facilita legătura între pagini. Clasa de tip *PageObject* este responsabilă cu găsirea elementelor de tip *WebElements* de pe acea pagină și oferă metode pentru a realiza operațiuni pe aceste elemente.

Pentru a exemplifica punerea în aplicare a acestui pattern am ales să folosesc clasa *LoginPage*, ce se poate vedea în *Secțiunea de cod 14* de mai jos. Am creat mai mulți membri privați de tip *IWebElement*: *RegisterLink*, *UsernameInput*, *PasswordInput*, *LoginInput*. Fiecare dintre acești membri are setate două atribute: *FindsBy* și *CacheLookup*.

```
public class LoginPage : NavBarContent
{
    [FindsBy(How = How.CssSelector, Using = "body > nav > div > ul > li:nth-child(4) > a")]
    [CacheLookup]
    private IWebElement RegisterLink { get; set; }

    [FindsBy(How = How.Id, Using = "user")]
    [CacheLookup]
    private IWebElement UsernameInput { get; set; }

    [FindsBy(How = How.Id, Using = "password")]
    [CacheLookup]
    private IWebElement PasswordInput { get; set; }

    [FindsBy(How = How.CssSelector, Using = "#container > div > div > div > div.form-box > form > button")]
    [CacheLookup]
    private IWebElement LoginButton { get; set; }

    public IWebElement GetLoginButton()
    {
        return LoginButton;
    }

    public void FollowRegisterLink()
    {
        RegisterLink.ClickOnIt("RegisterLink");
    }

    public void Login(string username, string password)
    {
        UsernameInput.EnterText(username, "UsernameInput");
        PasswordInput.EnterText(password, "PasswordInput");
        Thread.Sleep(2000);
        LoginButton.ClickOnIt("LoginButton");
    }
}
```

Secțiunea de cod 14: Structura paginii de login și exemplificarea pattern-ului PageObject

FindsBy este folosit pentru identificarea respectivului element în pagină. De asemenea, trebuie specificat cum trebuie să găsească acel element. Există mai multe posibilități puse la dispoziție de Selenium cum ar fi: *Id*, *TagName*, *CssSelector*, *XPath*, *Name*, *Class*. În cazul de față s-au folosit *CssSelector* și *Id*. Proprietatea *Using* reprezintă efectiv valoarea *Id*-ului și a *CssSelector*-ului căutat. Atributul *CacheLookup* este folosit pentru a prelua elementele din cache și în acest mod se evită inițializarea și căutarea de fiecare dată a acestora.

Deoarece membrii acestei clase au fost declarați privați am creat un Getter pentru *LoginButton*. Am avut nevoie de acest element în testul *GoToBookPageTest*. Am creat funcția *FollowRegisterLink* în interiorul căreia am apelat funcția *ClickOnIt* pe care o voi prezenta mai jos. Prin intermediul funcției de Login se va realiza logarea în aplicație. Elementele de input apelează metoda *EnterText* iar butonul apelează *ClickOnIt*.

În continuare, urmează să prezint metodele *EnterText*, *IsDisplayed* și *ClickOnIt*. Pentru a avea un control și asupra evenimentelor ce le realizează Selenium. Am creat o clasă static denumită *ElementExtensions* în interiorul căreia am creat metode proprii pe care să le pot apela de pe elementele din clasele *PageObjects* dar care să folosească în interiorul lor metodele din Selenium. Pentru a realiza acest lucru este esențial ca atât clasa cât și metodele să fie statice.

Metoda *EnterText()* primește ca și parametri instanța elementului de pe care se face apelul, un text ce se dorește a fi introdus și numele elementului. Deoarece această metodă se folosește pe elementele editabile este bine să apelăm mai întâi metoda *Clear()* pentru a șterge eventual anumite informații precedente și apoi se apelează metoda *SendKeys()* care trimite textul către acel element. Pentru debugging am am afișat un mesaj la consolă.

Metoda *IsDisplayed()* primește ca și parametri instanța elementului de pe care se face apelul și numele elementului. Am creat o variabilă booleană pe care o voi folosi pentru returnarea rezultatului execuției funcției. Am creat un block try-catch deoarece atributul *Displayed* din Selenium aruncă o excepție dacă elementul nu este afișat. Astfel, dacă nu apare nici o excepție *result* va fi true și afișăm un rezultat corespunzător în consolă. Dacă orice excepție apare, setăm *result* pe false. La final returnăm valoarea variabilei *result*. [9]

```

public static class ElementExtensions
{
    public static void EnterText(this IWebElement element, string text, string elementName)
    {
        element.Clear();
        element.SendKeys(text);
        Console.WriteLine(text + " entered in the " + elementName + " field.");
    }

    public static bool IsDisplayed(this IWebElement element, string elementName)
    {
        bool result;
        try
        {
            result = element.Displayed;
            Console.WriteLine(elementName + " is Displayed.");
        }
        catch (Exception)
        {
            result = false;
            Console.WriteLine(elementName + " is not Displayed.");
        }

        return result;
    }

    public static void ClickOnIt(this IWebElement element, string elementName)
    {
        element.Click();
        Console.WriteLine("Clicked on " + elementName);
    }
}

```

Secțiunea de cod 15: Clasa ElementExtensions, extinderea metodelor oferite de Selenium

Browser Factory

Menținerea și transferul unui obiect de tip `WebDriver` de-a lungul mai multor teste este un proces mai delicat. De asemenea, complexitatea crește când trebuie să menținem o singură instanță de `WebDriver` pe parcursul rulării unui test. Pentru a combate această problemă a instanțierii și a menținerii instanței am creat o clasă denumită *BrowserFactory*.

Am creat doi membri, unul de tip dicționar, *Drivers*, pentru a reține numele browser-ului și instanța acestuia, celălalt fiind de tip web driver, *Driver*, pentru instanța unică. În funcția *InitBrowser()* am creat un switch după numele browser-ului. Astfel, în funcție de numele acestuia și dacă membrul *Driver* este null se creează o instanță specifică de web driver: `FirefoxDriver()`, `InternetExplorerDriver()` sau `ChromeDriver()`.

```
internal class BrowserFactory
{
    private static readonly IDictionary<string, IWebDriver> Drivers = new Dictionary<string, IWebDriver>();

    public static IWebDriver Driver { get; private set; }

    public static void InitBrowser(string browserName)
    {
        switch (browserName)
        {
            case "Firefox":
                if (Driver == null)
                {
                    Driver = new FirefoxDriver();
                    Drivers.Add("Firefox", Driver);
                }
                break;

            case "IE":
                if (Driver == null)
                {
                    Driver = new InternetExplorerDriver();
                    Drivers.Add("IE", Driver);
                }
                break;

            case "Chrome":
                if (Driver == null)
                {
                    Driver = new ChromeDriver();
                    Drivers.Add("Chrome", Driver);
                }
                break;
        }
    }
}
```

Secțiunea de cod 16: Clasa BrowserFactory folosită pentru menținerea unei singure instanțe în timpul rulării testului partea 1

Funcția *LoadApplication* maximizează fereastra deschisă și introduce la adresă URL-ul transmis ca parametru. Funcția *CloseAllDrivers()* parcurge toate cheile din dicționar, închide respectivul browser deschis de driver și apoi închide instanța acestuia. Poate fi considerată o funcție de curățare.

```

        case "":
            if (Driver == null)
            {
                Driver = new ChromeDriver();
                Drivers.Add("Chrome", Driver);
            }
            break;
    }
}

public static void LoadApplication(string url)
{
    Driver.Manage().Window.Maximize();
    Driver.Url = url;
}

public static void CloseAllDrivers()
{
    foreach (var key in Drivers.Keys)
    {
        Drivers[key].Close();
        Drivers[key].Quit();
    }
}

```

Secțiunea de cod 17:Clasa BrowserFactory folosită pentru menținerea unei singure instanțe în timpul rulării testului partea a 2-a

Page Generator

Pentru a reduce și mai mult codul dintr-un test case dar și pentru a instanția mai ușor un obiect de tipul *PageObject* și după ce am urmărit mai multe tutoriale, am ajuns la concluzia că este un lucru potrivit crearea unei pagine generice. Ideea este ca această clasă să instanțieze un anumit *PageObject* în funcție de nevoie.

S-a creat astfel, pentru fiecare *PageObject* câte un membru în clasa *Page*. Pentru fiecare pagină am returnat tipul specific după cum se poate observa în *Secțiunea de cod 18* apelând constructorul *GetPage<>*.

Constructorul *GetPage<>* primește și returnează un tip generic T. În interiorul acestuia se creează o nouă pagină de tip T, apoi se face apelul către metoda *InitElements* pusă la dispoziție de către clasa *PageFactory*. Această clasă este o extensie a Design Pattern-ului *PageObject*. Este un concept POM¹⁶ esențial pentru Selenium WebDriver și este foarte bine optimizat. Este folosită pentru inițializarea elementelor unui *PageObject* sau chiar *PageObject*-ul în sine. [10]

```
public static class Page
{
    public static HomePage Home => GetPage<HomePage>();

    public static LoginPage Login => GetPage<LoginPage>();

    public static RegisterPage Register => GetPage<RegisterPage>();

    public static BookPage Book => GetPage<BookPage>();

    public static BooksPage Books => GetPage<BooksPage>();

    public static NotLoggedInBookPage NotLoggedInBook => GetPage<NotLoggedInBookPage>();

    public static UserPage User => GetPage<UserPage>();

    public static UsersPage Users => GetPage<UsersPage>();

    public static NotLoggedInUserPage NotLoggedInUser => GetPage<NotLoggedInUserPage>();

    public static UserProfilePage UserProfile => GetPage<UserProfilePage>();

    private static T GetPage<T>() where T : new()
    {
        var page = new T();
        PageFactory.InitElements(BrowserFactory.Driver, page);
        return page;
    }
}
```

Secțiunea de cod 18: Clasa *Page* folosită pentru generarea tuturor paginilor folosite în teste

¹⁶ Page Object Model

GoToBookPageTest

Deoarece fiecare test avea partea inițială și comună am decis să folosesc atributele *SetUp* și *TearDown* oferite de NUnit. În acest fel am asigurat faptul că înainte de rularea unui test se va executa funcția marcată cu atributul *[SetUp]* și după încheierea execuției fiecărei metode de test se va executa funcția marcată cu atributul *[TearDown]*.

Metoda marcată cu *[SetUp]* este denumită *TestInitialize()*. Ce face această metodă este apelarea metodelor *InitBrowser(applicationName)* și *LoadApplication(url)* oferite de către clasa *BrowserFactory*, metode ce au fost prezentate în paginile anterioare. Pe scurt, pentru a reaminti funcționalitatea lor este crearea unei instanțe de *WebDriver*, deschiderea unui browser și introducerea URL-ului aplicației în browser-ul deschis. Cu alte cuvinte, această metodă folosește la crearea condițiilor necesare rulării testelor.

```
[SetUp]
public void TestInitialize()
{
    // Start Selenium drivers
    BrowserFactory.InitBrowser("Chrome");
    BrowserFactory.LoadApplication(ConfigurationManager.AppSettings["URL"]);
}
```

Secțiunea de cod 19: TestInitialize, metoda executată înainte de fiecare test

Aplicând toate procedeele menționate mai sus, în subcapitolele următoare se poate vedea că un test este destul de intuitiv și ușor de citit. Astfel, primul pas este apelarea metodei de *Login()* de pe pagina cu același nume. De aici suntem redirecționați către pagina de *Home* de unde dorim să navigăm către pagina *Books*, prin intermediul metodei *FollowBooksLink()*. De pe această pagină apelăm metoda *FollowFirstBookLink()* pentru a ajunge pe pagina specifică primei cărți afișată pe pagina *Books*. Pasul final este verificarea faptului că pe pagina *Book* se află butonul de contactează utilizator și este vizibil. Verificarea este realizată prin apelul către funcția *IsTrue()* pusă la dispoziție de clasa *Assert* din C#.

```
[Test]
public void GoToBookPage()
{
    Page.Login.Login("John", "john12A");

    Page.Home.FollowBooksLink();

    Page.Books.FollowFirstBookLink();

    Assert.IsTrue(Page.Book.GetContactUserButton().IsDisplayed("ContactUserButton"));
}
```

Secțiunea de cod 20: metoda de test GoToBookPage

Metoda marcată cu *[TearDown]* este denumită *TestCleanup()*. Aceasta apelează metoda *CloseAllDrivers()* pusă la dispoziție de clasa *BrowserFactory*, atât metoda cât și clasa au fost detaliate în subcapitolul ***Browser Factory***.

```
[TearDown]
public void TestCleanup()
{
    BrowserFactory.CloseAllDrivers();
}
```

Secțiunea de cod 21: metoda TestCleanup(), executată după rularea fiecărui test

Manual de utilizare

Pagina principală a aplicației se poate vedea în *Figura 24*. Utilizatorul are mai multe opțiuni disponibile: să acceseze link-urile acasă, cărți, utilizatori, înregistrează-te sau să se autentifice. Vom presupune că primul pas va fi autentificarea, care va redirecționa utilizatorul către pagina din *Figura 25*. [11]

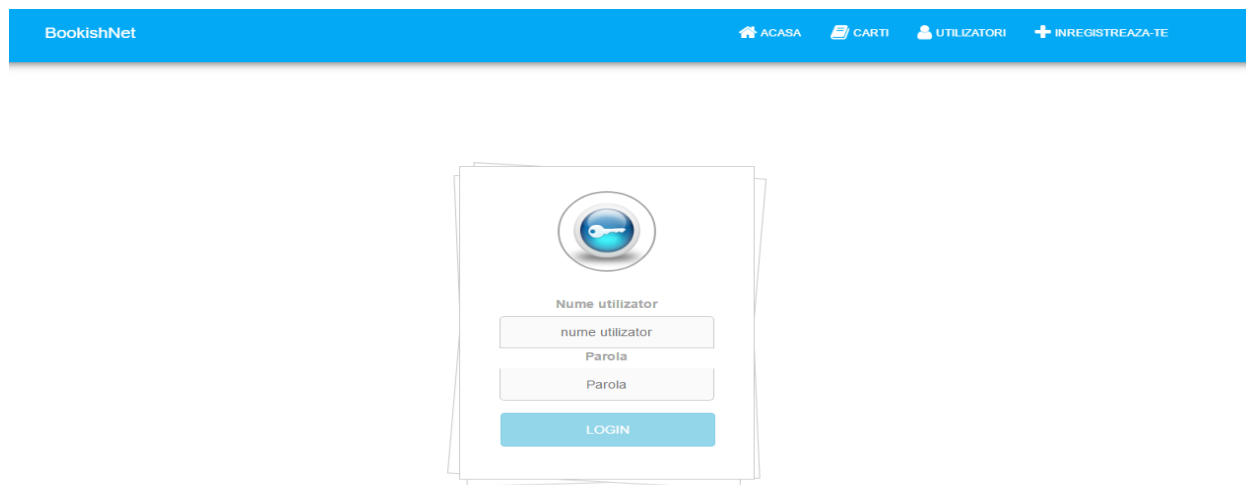


Figura 24: Pagina principală a aplicației

Pe această pagină se păstrează posibilitatea de a accesa link-urile cărți și utilizatori. Apar însă ca și elemente noi, un chat general, unde utilizatorul poate trimite mesaje ce vor putea fi văzute de orice utilizator online și un meniu dropdown de unde se pot selecta link-urile: acasă, profil și deloghează-te. La apăsarea link-ului deloghează-te, utilizatorul conectat va fi redirecționat către pagina din *Figura 24*. Pagina afișată după apăsarea link-ului de profil se poate vedea în *Figura 26* și *Figura 27*. [12] [13]

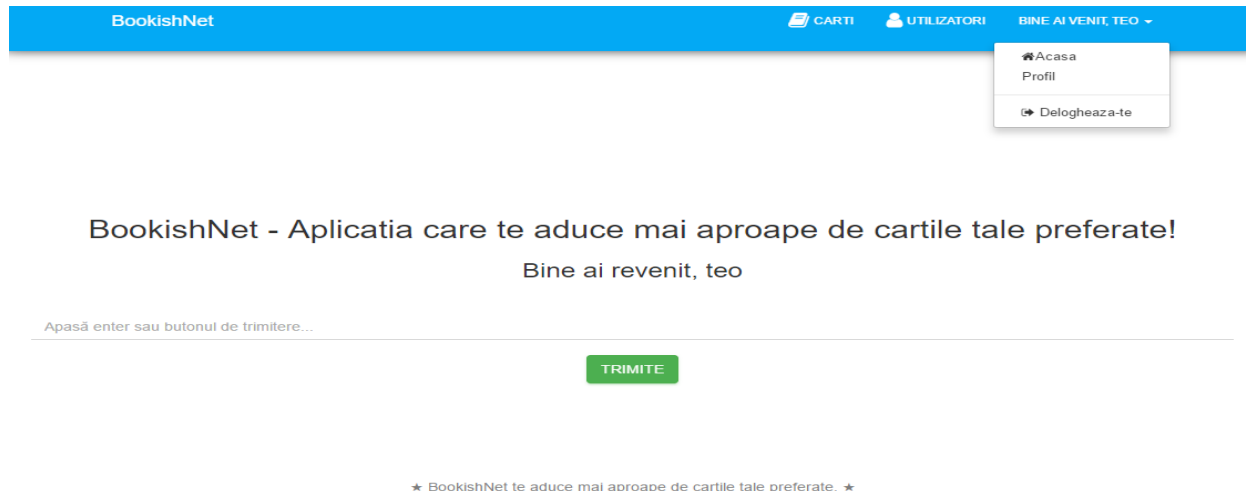


Figura 25: Pagina de acasă

Pe pagina de profil, utilizatorul poate vizualiza toate detaliile despre contul său pe care le-a introdus până la momentul respectiv. De asemenea, va putea să vadă o listă a cărților împrumutate de el, o listă a cărților pe care el le-a împrumutat de la alți utilizatori și o listă a cărților neîmprumutate (*Figura 27*) dar și un buton prin care poate modifica anumite detalii despre profilul său. Modalul care se deschide se poate vizualiza în *Figura 28*.

De asemenea, în respectivele liste ale cărților utilizatorul va avea posibilitatea de a vizualiza pagina cărții dorite sau a utilizatorului care a făcut împrumutul sau de la care s-a făcut împrumutul. Aceste pagini sunt exemplificate în *Figura 31*, *Figura 40* și *Figura 34*.

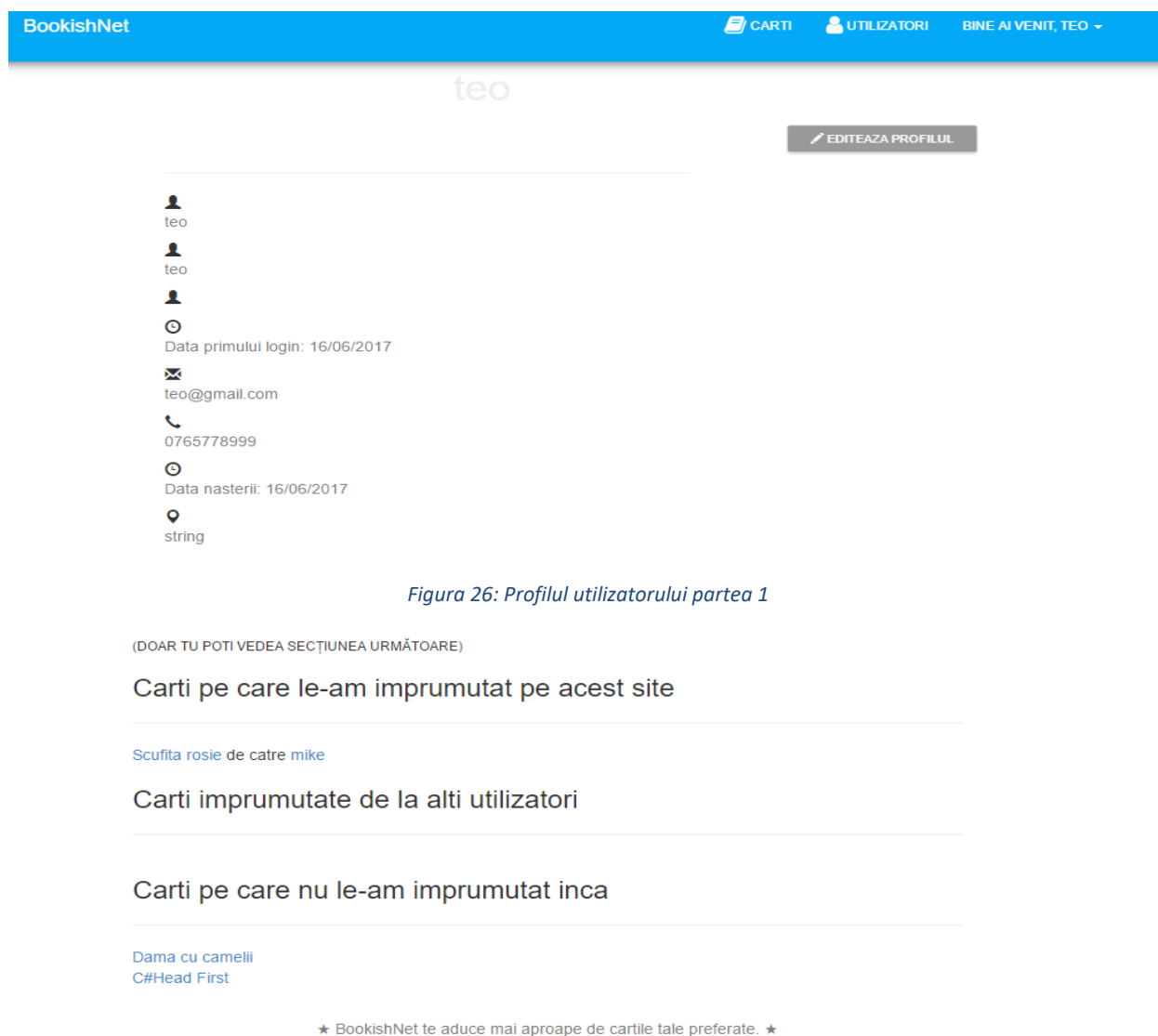


Figura 26: Profilul utilizatorului partea 1

Figura 27: Profilul utilizatorului partea a 2-a

Modalul de editare a profilului conține mai multe câmpuri editabile prin intermediul cărora utilizatorul își poate modifica informațiile - *Figura 28*. Pentru realizarea modificărilor, utilizatorul va apăsa butonul *Salvează modificările* iar pentru anulare butonul *Anulează editarea*.

Editeaza informatiile despre profilul tau

Nume:

teo

Prenume:

teo

Pseudonim:

Pseudonim

Numar telefon:

0765778999

Adresa:

string

Email:

teo@gmail.com

Nume utilizator:

teo

Zi de nastere

06/16/2017

SALVEAZA MODIFICARILE

ANULEAZA EDITAREA

Figura 28: Modalul care editează informațiile utilizatorului

La apăsarea link-ului cărți, disponibil în meniul, utilizatorul va fi redirecționat către pagina din Figura 29. De pe această pagină, utilizatorul va putea căuta o carte folosind filtrele din partea stângă a paginii sau va putea adăuga o nouă carte, exemplificat în Figura 30, sau va putea selecta unul din link-urile de pe titlul cărților din listă și va fi redirecționat către pagina din Figura 31 sau Figura 40.

BookishNet

CARTI
UTILIZATORI
BINE AI VENIT, TEO

Cauta cartea preferata

Titlu

Nume autor

Anul publicarii

ADAUGA O CARTE

Dama cu camelii

Alexandre Dumas

2000

Harry Potter si Pocalul de Foc, vol. 4

J.K. Rowling

2017

C#Head First

Jennifer Greene

2013

Figura 29: Pagina cu lista de cărți

Modalul prin intermediul căreia se poate adăuga o nouă carte se poate vizualiza în *Figura 30*. Pentru a putea apăsa butonul *Adaugă*, utilizatorul va trebui să completeze câteva câmpuri obligatorii cum sunt: titlul, genul, numele autorului, anul publicării. Bineînțeles, operațiunea poate fi anulată oricând apăsând butonul *Anulează adăugarea*. [4] [5]

Figura 30: Modalul care adaugă o nouă carte

Figura 31 exemplifică pagina unei cărți adăugate de utilizatorul autentificat. Acesta are trei opțiuni la dispoziție: să adauge cartea la o listă de cărți împrumutate, să o șteargă sau să editeze detalii despre aceasta (*Figura 32*).

Figura 31: Pagina cărții unui utilizator

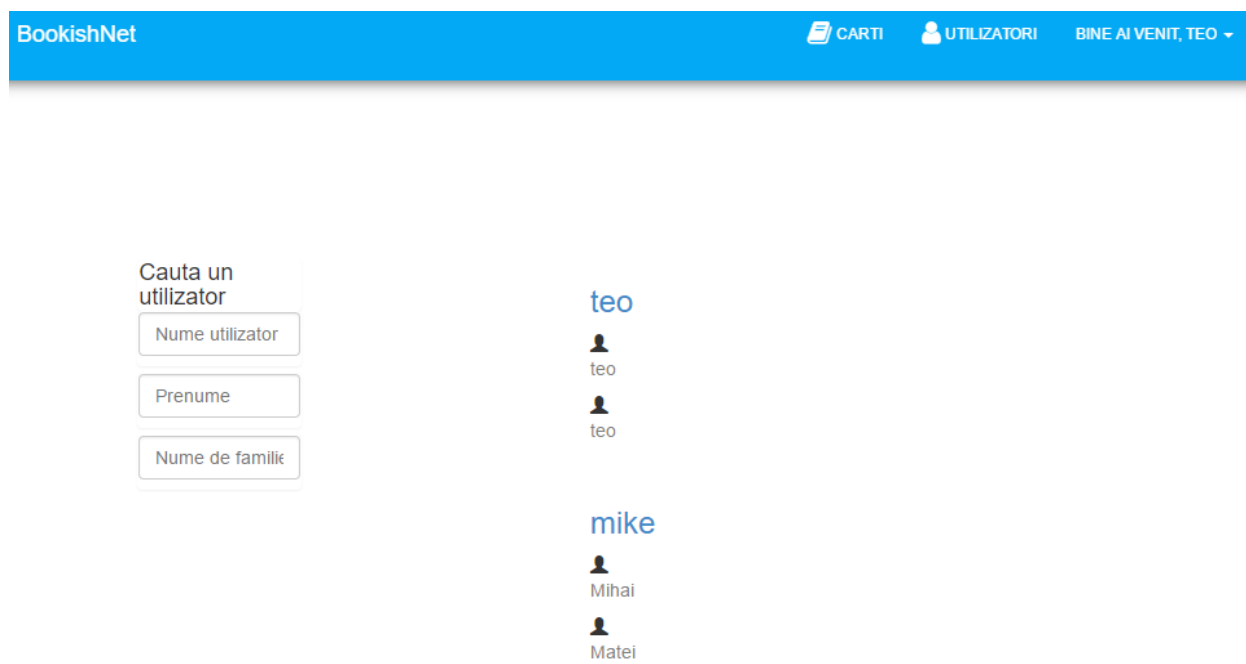
Editarea unei cărți se face cu ajutorul modalului prezentat în *Figura 32*. Astfel, utilizatorul poate efectua schimbări asupra unei cărți pe care o deține prin intermediul butonului *Salvează modificările* sau le poate anula apăsând butonul *Anulează editarea*.



The image shows a modal window titled "Editeaza detalii despre carte" (Edit book details). It contains several input fields for book information: "Titlu carte:" (Book title) with the value "Dama cu camellii", "Alegeti genul cartii:" (Choose the book genre) with the value "Romantic", "Descriere:" (Description) with the value "Awesome book", "Nume autor:" (Author name) with the value "Alexandre Dumas", and "Anul publicarii:" (Publication year) with the value "2000". At the bottom, there are two buttons: a green "SALVEAZA MODIFICARILE" (Save changes) button and a grey "ANULEAZA EDITAREA" (Cancel edit) button. Below the modal, the text "Awesome book" is visible.

Figura 32: Modalul care editează informațiile unei cărți

În figura *Figura 33* se poate vedea pagina cu utilizatori. Aici se pot căuta anumiți utilizatori cu ajutorul filtrelor din partea stângă a paginii sau se poate alege un link pus la dispoziție pe numele unui utilizator.



The image shows a web page for searching users. The header is blue and contains the text "BookishNet" on the left and "CARTI UTILIZATORI BINE AI VENIT, TEO" on the right. The main content area has a search form on the left with the title "Cauta un utilizator" (Search a user) and three input fields: "Nume utilizator" (User name), "Prenume" (First name), and "Nume de familie" (Last name). On the right, there are two sections of search results. The first section is titled "teo" and shows two user profiles, each with a person icon and the name "teo". The second section is titled "mike" and shows two user profiles, each with a person icon and the name "Mihai" and "Matei".

Figura 33: Pagina cu utilizatori

Figura 34 exemplifică pagina unui utilizator, diferit față de cel autentificat. Aici, utilizatorul poate apăsa butonul *Contactează utilizatorul* care va deschide un modal arătat în Figura 35.

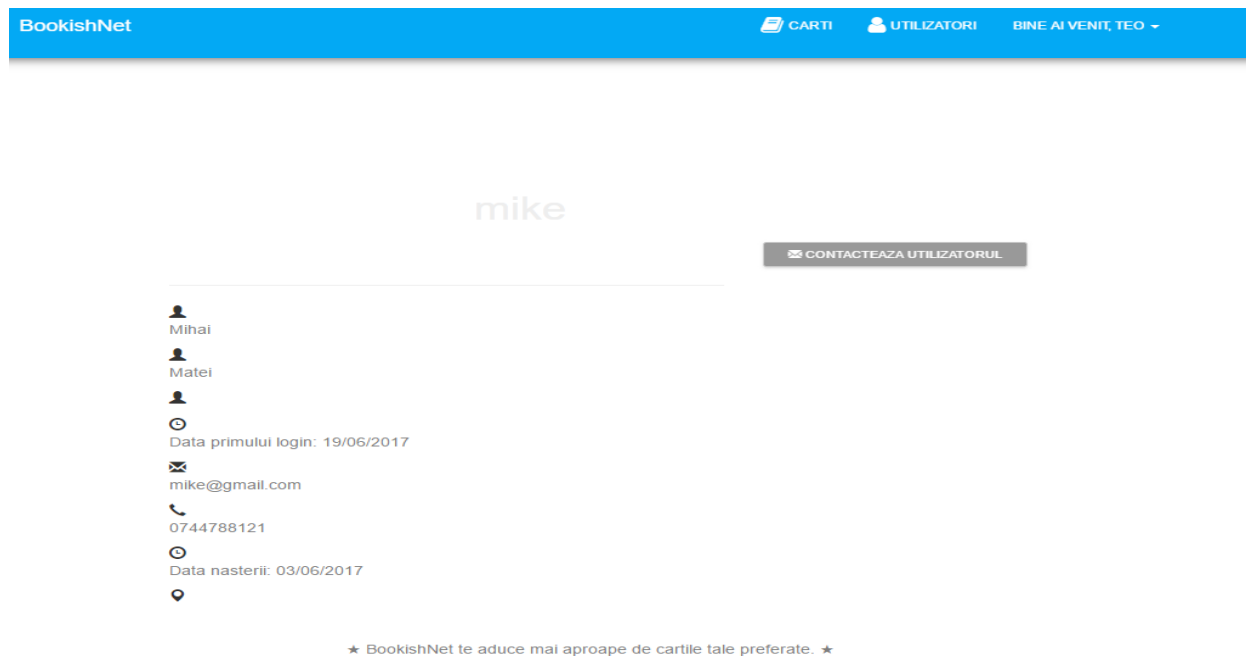


Figura 34: Pagina altui utilizator decât cel logat

În acest modal utilizatorul va fi nevoit să completeze toate câmpurile pentru a putea finaliza operațiunea și să acționeze butonul *Trimite email* sau îl va putea închide folosind butonul *Anulează*.



Figura 35: Modalul care asigură contactarea prin email a unui utilizator

Dacă un utilizator nu este înregistrat acesta își va putea crea un nou cont pe pagina din *Figura 36*. După completarea datelor necesare, butonul de *Register* va fi disponibil pentru apăsare. În caz contrar, butonul nu poate fi acționat.

The screenshot shows the registration page of the BookishNet application. At the top is a blue navigation bar with the text 'BookishNet' on the left and four links on the right: 'ACASA' (with a house icon), 'CARTI' (with a book icon), 'UTILIZATORI' (with a user icon), and 'INREGISTREAZA-TE' (with a plus icon). The main content area is white and contains a vertical stack of input fields and buttons. The fields are labeled: 'Nume utilizator' (with a placeholder 'nume utilizator'), 'Email' (with a placeholder 'Email'), 'Parola' (with a placeholder 'Parola'), 'Confirma parola' (with a placeholder 'Confirmare parola'), 'Zi de nastere' (with a placeholder 'mm/dd/yyyy'), 'Inregistrare ca autor' (with a radio button), 'Inregistrare ca utilizator' (with a radio button), and 'Pseudonim' (with a placeholder 'pseudonim'). At the bottom of the form is a blue button labeled 'REGISTER'.

Figura 36: Pagina de înregistrare în aplicație

Dacă un utilizator neautentificat încearcă să acceseze pagina unei cărți sau pagina unui utilizator, accesul îi va fi restricționat către acestea. Paginile ce se vor încărca în acest caz sunt exemplificate în *Figura 37* și *Figura 38*.

The screenshot shows the top part of a page in the BookishNet application. It features the same blue navigation bar as Figure 36, with 'BookishNet' on the left and links for 'ACASA', 'CARTI', 'UTILIZATORI', and 'INREGISTREAZA-TE' on the right. Below the navigation bar, the page content is mostly blank, indicating that the main content area is restricted for non-authenticated users.

Pentru a vedea mai multe detalii despre cartea dorita este necesara logarea.

Daca aveti un cont, logati-va [aici](#)

Daca nu aveti un cont, creati-va unul nou [aici](#)

★ BookishNet te aduce la un click distanta/mai aproape de cartile tale preferate. ★

Figura 37: Pagina unei cărți dacă utilizatorul nu este autentificat

Ne pare rau, dar pentru aceasta actiune este necesara logarea.

Daca aveti un cont, logati-va [aici](#)

Daca nu aveti un cont, creati-va unul nou [aici](#)

★ BookishNet te aduce la un click distanta/mai aproape de cartile tale preferate. ★

Figura 38: Pagina unui utilizator fără autentificare

Pe pagina unui autor, prezentată în *Figura 39*, apare un element nou - o secțiune în care se pot vedea cărțile scrise de acesta. Pe titlul fiecărei cărți există un link către respectiva carte. Utilizatorul poate apăsa acel link și va fi redirecționat către pagina din *Figura 40*.

teo12

✉ CONTACTEAZA UTILIZATORUL



Teofil



Ursan



teo



Data primului login: 27/06/2017



ursan.teofil@yahoo.com



Data nasterii: 17/11/1992



Carti scrise de mine



[Short Story](#)

★ BookishNet te aduce mai aproape de cartile tale preferate. ★


Figura 39: Pagina unui autor văzută de un alt utilizator

În cazul în care utilizatorul curent nu deține cartea prezentă pe link-ul accesat, va putea vizualiza pagina din *Figura 40*. Astfel, el va putea vizualiza informații despre respectiva carte, dar nu va mai putea modifica detalii despre aceasta. În schimb va putea apăsa butonul *Contactează utilizatorul* care va deschide modalul exemplificat în *Figura 35*.


BookishNet

 CARTI  UTILIZATORI BINE AI VENIT, TEO ▾

Harry Potter si Pocalul de Foc, vol. 4



J.K. Rowling




2017

Descriere

Scoala pentru Vrajitoare si Vrajitori Hogwarts se pregateste pentru Trimagiciada, o competitie la care au voie sa participe doar vrajitorii cu varsta de peste saptesprezece ani. Harry Potter are doar paisprezece ani, dar tot viseaza ca va castiga cumva Trimagiciada. Apoi, de Halloween, cand Pocalul de Foc hotaraste candidatii, Harry e uluit sa vada ca numele lui se gaseste printre cele alese de cupa magica. Va infrunta probe care sfideaza moartea, dragoni si vrajitori intunecati, dar cu ajutorul lui Ron si Hermione, s-ar putea s-o scoata la capat...

Contactează utilizatorul



0744788121

CONTACTEAZA UTILIZATORUL >>

★ BookishNet te aduce mai aproape de cartile tale preferate. ★

Figura 40: Pagina unei cărți ce aparține altui utilizator decât cel conectat

Concluzii generale

Scopul acestei lucrări a fost acela de a facilita comunicarea dar și procesul de împrumutare a cărților între utilizatori. Prin intermediul aplicației utilizatorii pot afla cine deține cartea pe care ei o doresc și pot intra în legătură cu utilizatorul care o deține pentru a stabili de comun acord în ce condiții se poate realiza împrumutul.

Implementarea aplicației client a ridicat în principal mai multe dificultăți în realizarea design-ului. Datorită Angular-ului partea de Javascript a devenit mult mai accesibilă și partea de logică a aplicației a putut fi realizată mult mai ușor. De asemenea, memorarea anumitor date și menținerea acestora pe tot timpul sesiunii a fost facilitată prin intermediul controller-elor care împărțeau același rootScope comun.

Pe partea de server au fost dificultăți mai mari cu privire la .NET Core. Unele librării nu au avut suport la momentul începerii aplicației și anumite sarcini au fost mai dificil de realizat sau chiar au fost abandonate. A fost o provocare datorită compatibilității cu anumite librării, în principal, dar și datorită faptului că la momentul creării aplicației, tehnologia era cea mai nouă apărută. Unele setări pentru introducerea unor funcționalități au fost făcute manual.

Pe partea de testare, provocările au venit atât în cadrul unit testelor cât și în cazul celor automate. Referitor la unit teste am evitat folosirea datelor reale din baza de date prin intermediul Mock-urilor. Testele automate au fost o provocare datorită pattern-ului nou folosit dar și prin faptul că unele lucruri nu au avut rezultatele așteptate inițial și au avut nevoie de ajustări, mai mici sau mai mari, în funcție de caz.

Soluția de față poate fi îmbunătățită printr-o serie de funcționalități noi, nedisponibile momentan. Astfel, se pot crea chat-uri individuale, unde utilizatorii să poată conversa direct, similar cu aplicațiile de mesagerie. Totodată, se pot introduce funcționalități pentru adăugarea fotografiilor de profil sau fotografii ale cărților.

Crearea unui sistem de notificări prin care utilizatorii să poată vedea cine este online. Adăugarea cărților în format pdf sau orice alt format în care se poate citi. În același timp, se poate integra o autentificare cu Facebook, Google sau alți distribuitori dar și crearea unor pagini pentru administrator. Posibilitatea de a oferi stele pentru autori și cărți.

Referințe

- [1] S. Buraga, "Servicii Web prin REST," 30 July 2009. [Online]. Available: <https://www.slideshare.net/busaco/servicii-web-prin-rest>. [Accessed May, June 2017].
- [2] A. P, "How to Send Emails from ASP.NET Core," 9 May 2016. [Online]. Available: <http://dotnetthoughts.net/how-to-send-emails-from-aspnet-core/>. [Accessed May, June 2017].
- [3] W. Anwar, "Building Custom AngularJS Directives using real world examples," 20 March 2016. [Online]. Available: <http://www.ezzylearning.com/tutorial/building-custom-angularjs-directives-using-real-world-examples>. [Accessed May, June 2017].
- [4] AngularJS, "AngularJS Forms," Angular, [Online]. Available: <https://docs.angularjs.org/guide/forms>. [Accessed May, June 2017].
- [5] AngularJS, "AngularJS Inputs," Angular, [Online]. Available: <https://docs.angularjs.org/api/ng/input>. [Accessed May, June 2017].
- [6] mkyong, "How to validate password with regular expression," 11 October 2012. [Online]. Available: <https://www.mkyong.com/regular-expressions/how-to-validate-password-with-regular-expression/>. [Accessed May, June 2017].
- [7] C. Smith, Angular Basics, ScriptyBooks LLC, 2015.
- [8] "AngularJS FormController," Angular, [Online]. Available: <https://docs.angularjs.org/api/ng/type/form.FormController>. [Accessed May, June 2017].
- [9] L. Sharma, "Automation Framework," 27 March 2016. [Online]. Available: <http://toolsqa.com/selenium-webdriver/c-sharp/pagefactory-in-c/>.
- [10] D. Kovalenko, Selenium Design Patterns and Best Practices, Birmingham, United Kingdom: Packt Publishing Limited, 2014.
- [11] S. Saini, "AngularJS User Registration in MVC with password encryption," 7 May 2016. [Online]. Available: <http://www.c-sharpcorner.com/article/user-registration-in-mvc-with-angularjs/>. [Accessed May, June 2017].
- [12] Gunnar, "ASP.NET Core: Building chat room using WebSocket," [Online]. Available: <http://gunnarpeipman.com/2017/03/aspnet-core-websocket-chat/>. [Accessed May, June 2017].
- [13] M. Otto, "Bootstrap dropdowns," Bootstrap, [Online]. Available: <https://v4-alpha.getbootstrap.com/components/dropdowns/>. [Accessed May, June 2017].

- [14] P. Tichy, "How To Create A Custom Preloading Screen," [Online]. Available: <https://ihatetomatoes.net/create-custom-preloading-screen/>. [Accessed May, June 2017].
- [15] D. Mestdagb, "More fun with AngularJS \$http: a loading indicator," 24 November 2015. [Online]. Available: <https://g00glen00b.be/fun-angularjs-http-loading-indicator/>. [Accessed May, June 2017].
- [16] R. Larsen, "Easy Autocomplete with the datalist Element, the list Attribute and AngularJS's ng-repeat Directive," 5 August 2013. [Online]. Available: <http://htmlcssjavascript.com/javascript/easy-autocomplete-with-the-datalist-element-the-list-attribute-and-angularjss-ng-repeat-directive/>. [Accessed May, June 2017].
- [17] F. Karim, "Simplest Chat App using SignalR with AngularJS," 24 June 2014. [Online]. Available: <https://www.youtube.com/watch?v=LDdjheM2gDM>. [Accessed June 2017].