UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

**Facultatea de Informatică**

****

Lucrare de licenţă

**BookishNet**

Propusă de

***Ursan Teofil-Cosmin***

**Sesiunea:** Iulie, 2017

Coordonator Ştiinţific:

***Asistent, dr. Vasile Alaiba***

UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

**Facultatea de Informatică**

**BookishNet**

***Ursan Teofil-Cosmin***

**Sesiunea:** Iulie, 2017

Coordonator Ştiinţific:

***Asistent, dr. Vasile Alaiba***

**DECLARAŢIE PRIVIND ORIGINALITATE ŞI RESPECTAREA**

**DREPTURILOR DE AUTOR**

Prin prezenta declar că Lucrarea de licență cu titlul „*BookishNet*” este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învățământ superior din ţară 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 şi în traducere proprie din altă limbă,

sunt scrise între ghilimele şi deţin referinţa precisă a sursei;

* reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa

precisă;

* codul sursă, imagini etc. preluate din proiecte open source sau alte surse sunt utilizate

cu respectarea drepturilor de autor şi deţin referinţe precise;

* rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Iaşi,

Absolvent Ursan Teofil-Cosmin

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(semnătura în original)

**DECLARAŢIE DE CONSIMŢĂMÂNT**

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

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

Iaşi,

Absolvent Ursan Teofil-Cosmin

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (semnătura în original)

Cuprins

[Introducere 8](#_Toc485835121)

[Motivaţie 8](#_Toc485835122)

[Context 8](#_Toc485835123)

[Cerinţe funcţionale 8](#_Toc485835124)

[Abordare tehnică 9](#_Toc485835125)

[Baza de date 9](#_Toc485835126)

[Backend 9](#_Toc485835127)

[Frontend 9](#_Toc485835128)

[Framework-uri 9](#_Toc485835129)

[Entity Framework Core 9](#_Toc485835130)

[Bootstrap 10](#_Toc485835131)

[Selenium 10](#_Toc485835132)

[Swagger 10](#_Toc485835133)

[Moq 10](#_Toc485835134)

[Contribuţii 10](#_Toc485835135)

[Proiectare 11](#_Toc485835136)

[Arhitectura soluției 11](#_Toc485835137)

[Capitolul 1: Dezvoltarea aplicaţiei client 12](#_Toc485835138)

[1.1. Arhitectura generală a aplicației 12](#_Toc485835139)

[1.2. Angular Views 13](#_Toc485835140)

[1.3. Angular Controllers 13](#_Toc485835141)

[1.4. Angular Services 14](#_Toc485835142)

[Capitolul 2: Dezvoltarea aplicaţiei server 14](#_Toc485835143)

[2.1. Arhitectura generală a aplicaţiei 14](#_Toc485835144)

[2.2. API Controllers 15](#_Toc485835145)

[2.3. Service Layer 16](#_Toc485835146)

[2.4. Data Layer 16](#_Toc485835147)

[Capitolul 3: Testarea funcţionalităţilor şi asigurarea calităţii 16](#_Toc485835148)

[1. Unit tests 16](#_Toc485835149)

[2. Integration tests 18](#_Toc485835150)

[3. Acceptance tests 19](#_Toc485835151)

[Modelarea datelor 20](#_Toc485835152)

[Arhitectura bazei de date 20](#_Toc485835153)

[Entităţi 21](#_Toc485835154)

[1. Books 21](#_Toc485835155)

[2. Reviews 21](#_Toc485835156)

[3. Genres 21](#_Toc485835157)

[4. BookAuthor 22](#_Toc485835158)

[5. Users 22](#_Toc485835159)

[6. Roles 22](#_Toc485835160)

[7. Messages 22](#_Toc485835161)

[Relaţii între entităţi 22](#_Toc485835162)

[1. Users-Books 22](#_Toc485835163)

[2. Users-Messages 23](#_Toc485835164)

[3. Users-Roles 23](#_Toc485835165)

[4. Books-Reviews 23](#_Toc485835166)

[5. Books-Genres 23](#_Toc485835167)

[Protocoale de comunicare client – server 23](#_Toc485835168)

[BookishNet API 24](#_Toc485835169)

[Interfața cu utilizatorul 29](#_Toc485835170)

[Home Page 29](#_Toc485835171)

[Register 30](#_Toc485835172)

[Welcome Back Page 31](#_Toc485835173)

[Books 32](#_Toc485835174)

[Book Page 33](#_Toc485835175)

[Users 34](#_Toc485835176)

[User Page 35](#_Toc485835177)

[Implementare 37](#_Toc485835178)

[Data Layer 37](#_Toc485835179)

[Data Layer Tests 38](#_Toc485835180)

[MVC 39](#_Toc485835181)

[Autentificarea 39](#_Toc485835182)

[Trimitere email 40](#_Toc485835183)

[Directiva pentru confirmarea parolei 42](#_Toc485835184)

[BookService 42](#_Toc485835185)

[GetBooks 42](#_Toc485835186)

[UpdateBook 43](#_Toc485835187)

[LoginController 44](#_Toc485835188)

[MVC Tests 47](#_Toc485835189)

[Manual de utilizare 47](#_Toc485835190)

[Referințe 47](#_Toc485835191)

# Introducere

## Motivaţie

În această lucrare voi propune o soluţie pentru dezvoltarea unei aplicaţii web, atât pentru client cât şi pentru server, care să ofere informaţii utilizatorilor despre cărţi pe care doresc să le împrumute şi nu le deţin sau nu le pot procura de la o bibliotecă sau librărie.

De asemenea, consider că ar putea fi considerată şi o aplicaţie socială deoarece utilizatorii vor avea posibilitatea de a cunoaşte persoane noi şi de a comunica cu acestea prin intermediul chat-ului din interiorul aplicaţiei.

## Context

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

Conform ultimelor statistici oficiale[[1]](#footnote-1) din anul 2011 interesul pentru lectură este destul de scăzut. De asemenea, datorită numărului continuu în creştere de utilizatori în mediul online[[2]](#footnote-2) consider că o asemenea aplicaţie online ar putea fi benefică.

## Cerinţe funcţionale

Pentru a avea un impact pozitiv dar şi pentru eventualitatea intrării pe piaţă a aplicaţiei, ea va trebui să respecte următoarele cerinţe funcţionale:

* Utilizatorul va avea posibilitatea de a-şi crea un cont nou
* Utilizatorul va putea folosi ca şi metode de autentificare Google, Facebook sau contul creat la inregistrare
* Utilizatorul va putea să caute cartea dorită în funcţie de anumite opţiuni: titlu, numele autorului, anul publicării cărţii
* Utilizatorul va putea să-şi editeze informaţiile personale
* Utilizatorul va putea să-şi managerieze cărţile pe care le deţine
* 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 prezenţi pe site şi cărţile pe care aceştia le deţin
* Utilizatorul va avea posibilitatea de a se loga/deloga

## Abordare tehnică

Baza de date

Pentru baza de date am ales soluţia oferită de cei de la Microsoft, şi anume, Microsoft SQL Server. Este o bază de date relaţională, care are capacitatea de stocare şi expunere a datelor pe baza unor cereri primite de la solicitant şi poate fi rulată pe acelaşi computer sau pe un computer din reţea.

### Backend

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

### Frontend

La nivel de frontend se vor folosi AngularJS, framework de Javascript (limbaj de nivel înalt şi interpretat), care foloseşte 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 şi 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 canditatul ideal pentru partea de web design.

Pentru crearea efectivă a paginilor se va folosi HTML. Este un [limbaj de marcare](https://ro.wikipedia.org/wiki/Limbaj_de_marcare) utilizat pentru crearea paginilor web ce pot fi afișate într-un [browser](https://ro.wikipedia.org/wiki/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](https://ro.wikipedia.org/wiki/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 şi mai uşoară comparativ cu Entity Framework. EF Core este un mapper de obiecte relaţionale (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 şi suport pentru mai multe motoare de baze de date.

#### Bootstrap

Este un framework pentru web pe partea de frontend gratuit şi open-source, care ajută la realizarea design-ului pentru site-uri sau aplicaţii web. Conţine anumite şabloane de design - HTML şi CSS - pentru elementele vizibile în interfaţa utilizatorului dar şi opţional, 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 OAS[[3]](#footnote-3). 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.

# Contribuţii

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

* Capitolul 1: Dezvoltarea aplicației client
* Capitolul 2: Dezvoltarea aplicației server
* Capitolul 3: Testarea funcţionalităţilor şi asigurarea calităţii

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

* Să fie user-friendly şi intuitivă
* Să fie rapidă
* Să fie scalabilă în funcţie de dispozitivul folosit
* Utilizarea de design-pattern-uri pentru separarea responsabilităţilor

Pe partea de server s-a avut în vedere ca aplicaţia să facă faţă 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 responsabilităţilor

În ceea ce priveşte 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

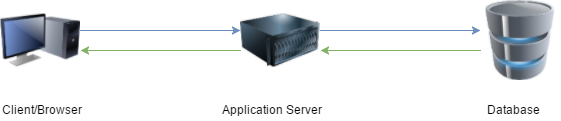
Testarea funţionalităţii şi asigurarea calităţii are ca scopuri esenţiale asigurarea faptului că aplicaţia respectă cerinţele funcţionale şi posibilitatea de dezvoltare, îmbunătăţire, aspecte ce pot fi aduse pe viitor:

* Crearea de unit teste pentru fiecare modul şi layer în parte
* Crearea de teste de integrare pentru verificarea funcţionalităţii modulelor împreună
* Crearea de teste automate pentru verificarea funcţionalităţii din punctul de vedere al utilizatorului final

# Proiectare

## Arhitectura soluției

Arhitectura generală a soluţiei(aplicaţiei) nu este diferită faţă de cea a unei aplicaţii web obişnuite. Ea cuprinde trei părţi mari şi importante: clientul, serverul şi 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 şi se aşteaptă un răspuns, altfel serverul va evalua cererea primită de la client şi 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. Următoarea diagramă ilustrează arhitectura generală a aplicaţiei:

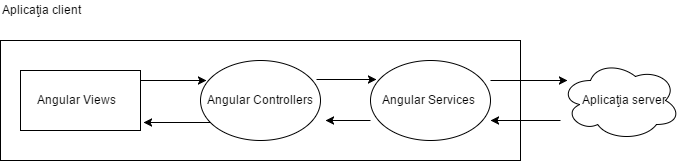


### Capitolul 1: Dezvoltarea aplicaţiei client

Aplicaţia client are ca scop facilitarea procesului de comunicare, de obţinere şi oferire de informaţii între utilizatori prin intermediul unei conexiuni la Internet. Aplicaţia a fost dezvoltată cu ajutorul editorului pus la dispoziţie de cei de la Microsoft – Visual Studio 2017 Enterprise. Pe parcursul dezvoltării s-a avut în vedere utilizarea tehnicilor necesare pentru obţinerea unei aplicaţii de tip SPA[[4]](#footnote-4) astfel încât utilizatorul să obţină rezultate mai rapide afişate în pagină iar aplicaţia să fie utilizabilă în condiţii reale, pe multiple dispozitive. Tehnicile folosite vor fi descrise, pe larg, în subcapitolele ce vor urma.

#### Arhitectura generală a aplicației

Arhitectura aplicaţiei client este puţin diferită de cea a unei aplicaţii client de tip web obişnuite. Astfel, cererile utilizatorului vor fi preluate din partea de view (fie că este vorba de butoane, câmpuri de input, hyperlink-uri) şi vor fi transmise către controllerul Angular specific (fiecare view, în parte, are asignat câte un controller care se ocupă de administrarea primului menţionat). De aici, controllerul Angular preia cererea şi dacă este necesar o trimite mai departe către un serviciu Angular, urmând apoi să aştepte un răspuns. Acesta, în cele din urmă, va fi delegatul care va trimite cererea către aplicaţia server. Serviciul va aştepta un răspuns de la server după care va întoarce rezultatele necesare către controller, iar acesta, la rândul său, va transmite modificările necesare către view. Această aplicaţie trebuie să fie capabilă să ofere utilizatorului o experienţă plăcută de utilizare, să fie uşor de folosit şi să notifice utilizatorul în legătură cu mesaje noi. Diagrama aplicaţiei client poate fi văzută mai jos, după cum urmează:



#### Angular Views

Utilizatorul va avea contact direct cu partea de Angular Views a aplicaţiei client. Deşi ele sunt menţionate la plural, component principal este reprezentată de un view general, similar unei pagini de home. Diferenţa esenţială faţă de aplicaţiile web clasice vine acum în discuţie. Deoarece Angular permite dezvoltarea de aplicaţii de tip SPA3, asta înseamnă că nu vom încărca de fiecare data o întreagă pagină web ci doar o anumită porţiune, porţiune care este definite sub formă de view. De aceea avem mai multe view-uri în aplicaţie, însă scheletul este acelaşi, având astfel posibilitatea de a modifica doar anumite porţiuni ale view-ului principal şi renunţând astfel la clasica reîncărcare a tuturor elementelor de pe o pagină HTML standard.

Sintaxa pentru un asemenea view este similară cu cea a unei pagini HTML, cu menţiunea 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>, <span></span> ş.a. Singura excepţie de la această abordare o reprezintă view-ul general, aşa numitul index.html. Această pagină arată în totalitate ca o pagină HTML autentică relativ la părţile de <head>, <body>, <script>. Aşadar, view-urile sunt cele pe care utilizatorul le va întâlni în aplicaţie şi va interacţiona cu aceasta prin intermediul lor.

#### Angular Controllers

A doua componentă a aplicaţiei client, extrem de importantă şi 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 şi pentru respectarea principiului de separare a responsabilităţilor. Aşadar, 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 şi în partea de servicii. Astfel, el este cel care notifică view-ul în privinţa unor schimbări şi 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 şi datele care trebuiesc transmise către aplicaţia server. Totodată, deoarece abordarea a fost aleasă în acest mod, câte un controller pe fiecare view, va duce la creşterea vitezei de execuţie şi rezolvare a cererilor, un aspect esenţial urmărit de aplicaţiile contemporane şi implicit, de aplicaţia de faţă.

#### Angular Services

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

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

* Iniţializarea în momentul în care avem nevoie de anumite resurse
* Crearea unei singure instanţe de-a lungul rulării aplicaţiei
* Evitarea consumului inutil de resurse
* Creşterea performanţelor datorită consumului mai redus de resurse

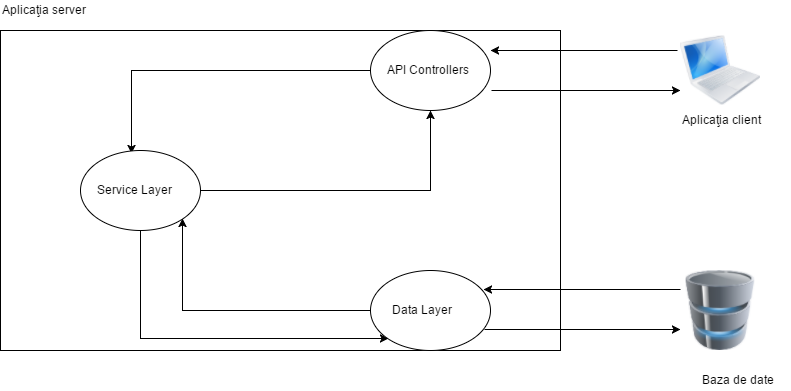
De asemenea, serviciile Angular sunt cele care fac cererile către API-ul aplicaţiei, un API care oferă resursele necesare aplicaţiei client. Această componentă va fi detaliată mai pe larg în capitolul dedicate aplicaţiei server. Aşadar, serviciile sunt cele care fac legătura între aplicaţia server şi cererile primite de la utilizator – pot fi considerate, pe bună dreptate, legătura între cele două aplicaţii.

### Capitolul 2: Dezvoltarea aplicaţiei server

#### 2.1. Arhitectura generală a aplicaţiei

Pe partea de server s-a avut în vedere ca aplicaţia să facă faţă unui volum consistent de utilizatori, să fie sigură și să păstreze confidențialitatea informațiilor personale despre utilizatori sau cărți. Astfel, utilizatorul poate accesa aplicaţia fie autentificat, fie nu, cu menţiunea că dacă nu este autentificat accessul la anumite resurse din aplicaţie îi va fi restricţionat acelui utilizator.

De asemenea, accesul în aplicaţie, ca şi 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 funcţie de rolul pe care îl deţine, utilizatorul va avea access sau nu la mai multe resurse, totul depinzând de gradul de încredere oferit de rolul asignat. O diagramă sumară a aplicaţiei server poate fi vizualizată mai jos:



#### 2.2. API Controllers

Odată cu trimiterea cererii de la client şi sosirea ei pe partea de server, prima componentă cu care are loc interacţiunea este unul dintre controllerele serviciului REST[[5]](#footnote-5) realizat folosind Web API[[6]](#footnote-6). De la acest nivel, serverul propagă cererea la nivelul următor, urmând apoi să aştepte primirea unui răspuns, fie de succes, fie de eşec.

După ce se primeşte 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 eşec se poate returna status code-ul 404 – cu semnificaţia că resursa solicitată nu a fost găsită).

Un lucru important care se realizează aici şi merită menţionat este injectarea serviciului corespunzător pentru rezolvarea cererii în constructorul controller-ului. Acest procedeu se foloseşte 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 şi operaţiile 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 obţine informaţii din baza de date sau chiar pentru a modifica anumite date. În acest caz, accesul ar putea fi restricţionat doar pe partea de client, iar un atacator ar putea profita destul de uşor de această vulnerabilitate a aplicaţiei pentru a citi/modifica date preţioase.

#### 2.3. Service Layer

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

Lucrurile sunt similare în ceea ce priveşte drumul pe care îl are de parcurs cererea. Se primeşte cererea, se trimite către nivelul de mai jos şi se aşteaptă un răspuns. De asemenea, se poate spune că acest nivel este mai mult unul de transport pentru a facilita transmisia datelor între diferitele nivele ale aplicaţiei. Acest lucru a fost realizat pentru a ajuta şi mai mult la separarea responsabilităţilor.

#### 2.4. Data Layer

La acest nivel, aplicaţia pregăteşte interacţiunea cu baza de date. Pentru a facilita accesul la baza de date dar şi pentru a preveni modificarea comenzilor Sql trimise către baza de date s-a folosit Entity Framework Core. Astfel, acest nivel conţine atât modelele pentru crearea bazei de date dar şi Repository-uri pentru a apela efectiv operaţiile ce vor fi operate pe baza de date.

Fiecare model în parte are o interfaţă şi un repository care implementează interfaţa specific. S-au creat atât operaţiile CRUD[[7]](#footnote-7) dar şi alte operaţii suplimentare pentru a aduce doar anumite rânduri din baza de date. Crearea de interfeţe a fost aleasă pentru a respecta al patrulea principiu din SOLID, şi anume, principiul segregării interfeţei. În acest mod, fiecare clasă ce va implementa o anumită interfaţă se va ocupa de un model în parte, realizându-se, în acelaşi timp şi separarea în funcţie de entitatea dorită.

### Capitolul 3: Testarea funcţionalităţilor şi asigurarea calităţii

#### Unit tests

Pentru a verifica funcționalitatea fiecărui modul în parte am folosit așa numita strategie de Unit Testing[[8]](#footnote-8). 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.

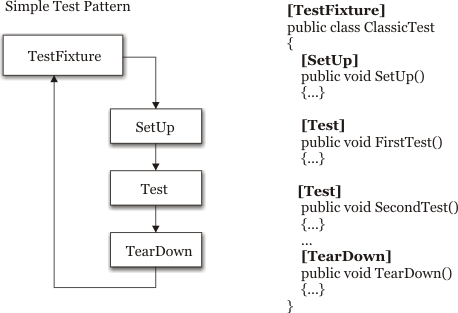
Această tehnică are și avantaje și dezavantaje pe care le voi prezenta pe scurt în cele ce urmează. Ca și dezavantaje ale scrierii de Unit Teste putem enumera: timpul alocat pentru a realiza teste potrivite poate fi semnificativ, nu toate erorile pot fi detectate aici – spre exemplu, cele de integrare.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

Având în minte toate aceste lucruri am hotărît 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 hotărît 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 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 imaginea de mai jos, urmând ca apoi să ofer detalierea fiecărei componente în parte:



Când se dorește crearea unei clase de test se folosește o adnotare deasupra acesteia. Această adnotare diferă în funcție de framework-ul[[9]](#footnote-9) 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 sa trateze acea clasă în mod diferit, ca pe una de test, și se va aștepta să găsească în interiorul ei metodă/metode de test.

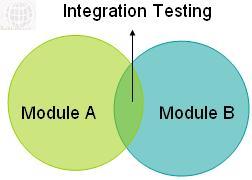
Al doilea atribut prezent în figură este *SetUp*. De menționat faptul că toate atributele prezente în această diagramă sunt specifice pentru NUnit. Atributul *SetUp* se folosește, de regulă când există teste care doresc să împartă resurse comune. Metoda care are atributul acesta specificat deasupra ei 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 data 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.

#### Integration tests

În subcapitolul precedent am discutat despre importanța testării fiecărui modul în parte, prin intermediul Unit Testelor. Având teste solide și funcționale pentru fiecare modul în parte este momentul potrivit să mergem la următorul nivel și să mai facem un pas pentru ca aplicația de față să fie testate așa cum se cuvine. Următorul nivel este cel al testelor de integrare. Pentru a vă face o idee generală despre arhitectura lor am atașat în continuare o fotografie:



Spre deosebire de Unit Teste, ale căror scop principal era asigurarea faptului că fiecare modul funcționeză corespunzător și independent față de celelalte, testele de integrare sunt cu un nivel mai sus în lanțul de testare. Astfel, ele se ocupă cu funcționarea modulelor împreună. Este excelent să avem teste funcționale pe fiecare modul în parte, dar până la urmă, aplicația este alcătuită din aceste module puse împreună. De aceea este un pas important să verificăm dacă, punând împreună diferite module acestea funcționează corespunzător.

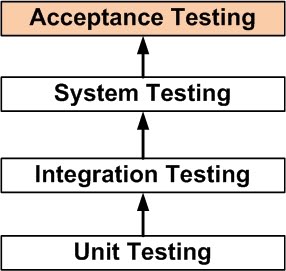
Pentru a realiza acest tip de teste există mai multe abordări: big-bang, top-down, bottom-up, mixed (sandwich), risky-hardest. Strategia aleasă în prezenta lucrare este cea de tip bottom-up[[10]](#footnote-10). Ideea generală a acestei strategii este să începi de la componentele mai simple (de bază) și să ajungi la componentele mai complexe, componente care se folosesc de cele de bază. Am ales această strategie deoarece procesul de dezvoltare a fost tot de la componentele din nivelele inferioare și apoi am continuat către nivelele superioare.

Ca și structură, aceste teste au aceleași componente de bază ca și unit testele. Cu alte cuvinte, se vor folosi adnotări pentru recunoașterea metodelor în cauză ca fiind metode de test. Conținutul acestor metode va fi puțin mai complex comparativ cu cel al precedentelor teste. Acceptul va cădea pe anumite fluxuri posibile în aplicație. Spre exemplu, pentru a modifica informații în baza de date, trebuie să fii autentificat, nu poți modifica informațiile altui utilizator și lista poate continua.

#### 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 testing[[11]](#footnote-11). 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 data 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:



După cum se poate vedea și în diagramă, 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.

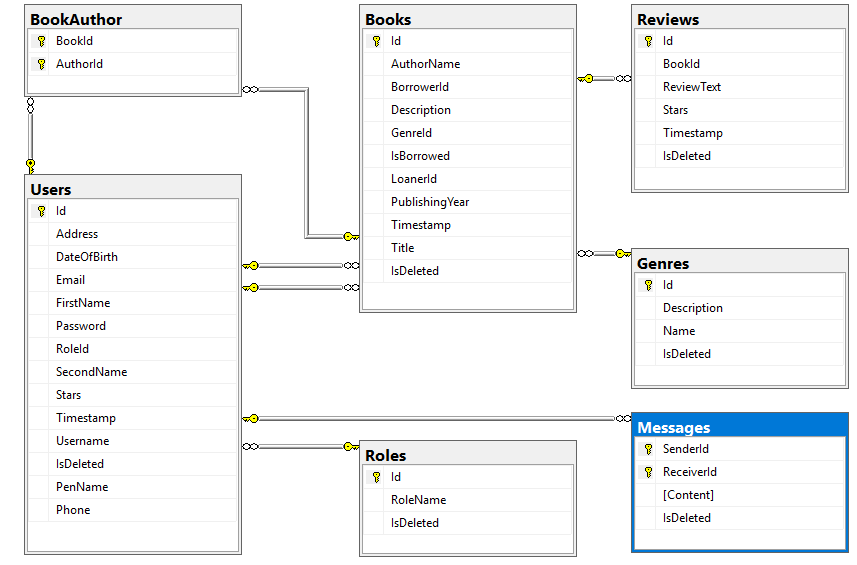
Bineînțeles că aceste tipuri de teste pot fi create și fără existența celorlalte teste din straturile inferioare, dar, implicit, acest lucru ar putea duce frecvent la diverse probleme și erori la rularea acestor teste. De aceea, este recomandat ca toate aceste nivele de testare să existe indiferent cât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

Așadar, având această suită de teste (Unit Tests, Integration Tests, Acceptance Tests) aplicația capătă un grad mai ridicat de încredere din punct de vedere al 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 încredere.

## Modelarea datelor

### Arhitectura bazei de date

Pentru realizarea bazei de date s-a avut în vedere consistenţa datelor şi dimensiunea redusă a acesteia. S-a dorit ca atât numărul de tabele dar şi relaţiile aferente între ele să fie cât mai redus. S-a recurs astfel la soluţia de a se introduce şapte entităţi, aşa cum se poate vedea şi în diagrama de mai jos, entităţi care vor fi detaliate în cele ce urmează.



### Entităţi

#### Books

Entitatea centrală a acestei scheme este reprezentată prin tabela Books. Ea conţine diverse informaţii necesare pe care o carte le poate conţine. Se poate vedea cu uşurinţă care este cheia primară, şi anume, coloana Id care are proprietetea de autoincrementare şi este de tip număr întreg. Pe baza acestei coloane se poate identitifica, în mod unic, o înregistrare de tip carte a tabelei. Celelalte câmpuri ale tabelei se împart în două secţiuni: 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 menţionat nu sunt obligatorii.

#### Reviews

Este o entitate secundară, mai redusă ca şi dimensiune decât precedent şi are ca rol principal reţinerea unor informaţii 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 şi aici câmpuri obligatorii: BookId, Timestamp, IsDeleted şi câmpuri neobligatorii: ReviewText, Stars.

#### Genres

Această entitate are ca scop determinarea unui gen anume pentru o entitate de tip carte, aşadar, 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 ş.a. Ca şi câmpuri prezente în tabelă avem aceeaşi împărţire clasică – obligatorii: Name, IsDeleted – şi neobligatorii: Description.

#### BookAuthor

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

#### Users

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

#### Roles

Această entitate, deşi mai redusă ca dimensiuni este un element destul de important al schemei deoarece aici sunt stocate rolurile disponibile în aplicaţie dar şi informaţii despre cine deţine rolul respective. Tabela are doar trei câmpuri, toate dintre ele fiind obligatorii. Primul câmp este cel de Id, care este şi cheia primară pentru tabelă. Celelalte două câmpuri sunt RoleName, unde se reţine efectiv valoarea rolului şi IsDeleted, câmp de tip Boolean pentru a determina dacă înregistrarea a primit sau nu o cerere de a fi ştearsă. Acest câmp a fost folosit în toate tabelele pentru a evita operaţia clasică de ştergere şi a o înlocui cu una virtuală[[12]](#footnote-12), astfel datele nu vor fi pierdute total la o cerere de ştergere.

#### Messages

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

### Relaţii între entităţi

#### Users-Books

Relaţia între utilizatori şi cărţi este una de tip many-to-many. Asta înseamnă că o carte poate avea mai mulţi utilizatori (acest aspect a fost gândit pentru situaţia în care o carte este scrisă de mai mulţi autori) dar şi un utilizator poate avea mai multe cărţi (ele pot fi împrumutate sau gata spre a fi împrumutate). Soluţia cea mai simplă, dar cu redundanţe multiple ar fi fost să se introducă câte o cheie străină reprezentând identificatorul celelilate entităţi în fiecare tabel. Pentru a evita o asemenea situaţie s-a creat un tabel separate, BookAuthor, unde se vor reţine doar identificatorii unici ai fiecărei părţi implicate.

#### Users-Messages

Relaţia între aceste entităţi este de tip one-to-many. Aceasta implică faptul că un utilizator poate avea mai multe mesaje, cu alte cuvinte o listă de mesaje. Referinţa acestui fapt s-a făcut prin cheia primară compusă din tabela Messages, alcătuită din două chei străine: SenderId şi ReceiverId. Relaţia poate fi văzută şi în schema generală a bazei de date.

#### Users-Roles

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

#### Books-Reviews

O carte are nevoie şi de comentarii, păreri, recomandări, de aceea am ajuns la concluzia că este necesar pentru a stabili această legătură între entităţi de o relaţie 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 cheiă străină pentru Reviews venită de la tabela Books.

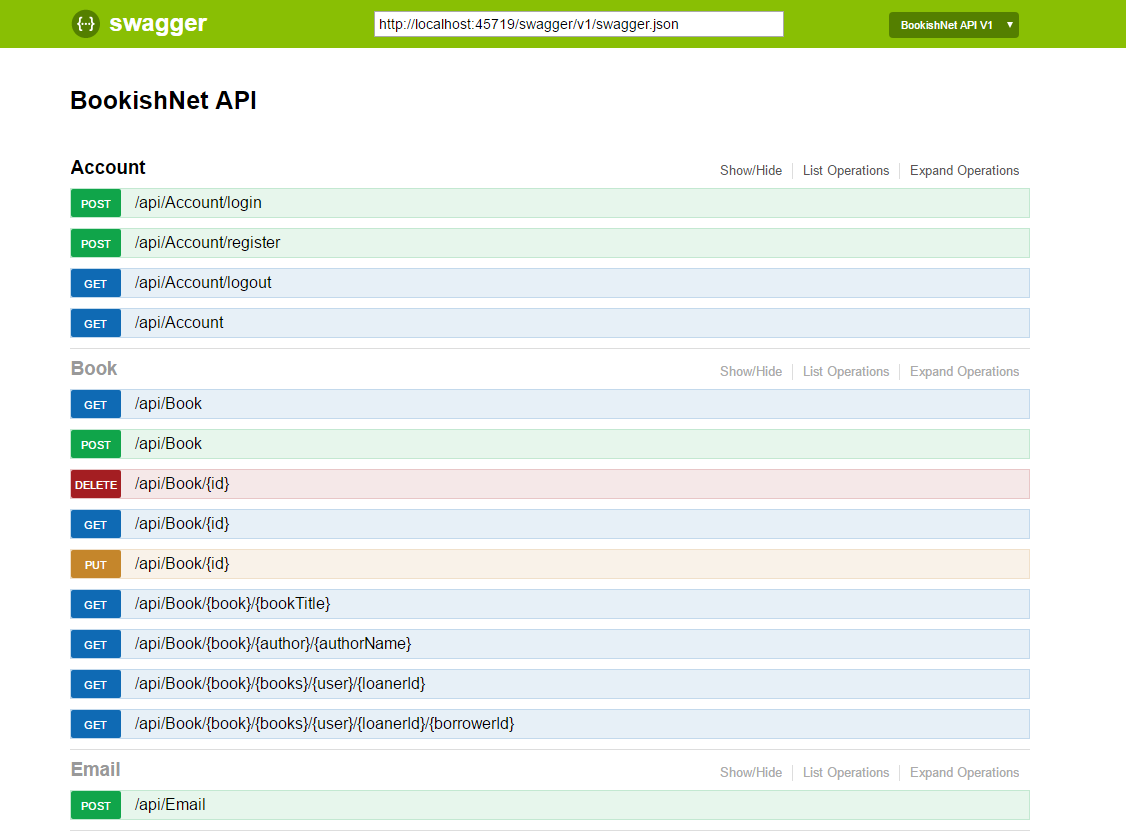
#### Books-Genres

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 şi Genres. În acelaşi timp, acelaşi gen, poate să fie acontat la mai multe cărţi aşadar, relaţia 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 şi cheie străină cu referinţă 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.

### BookishNet API

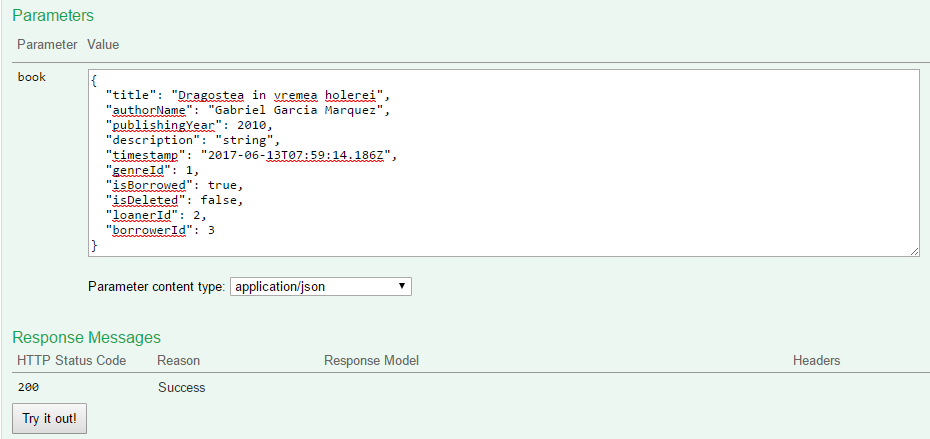


În figura de mai sus se poate observa interfața pe care o creează, în mod automat, Swagger. Pentru fiecare controller API se creează o zonă specifică. Spre exemplu, în figura precedentă se pot observa operațiile posibile pentru controller-ele ***Acount, Book*** și ***Email.*** Pentru detalierea serviciilor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

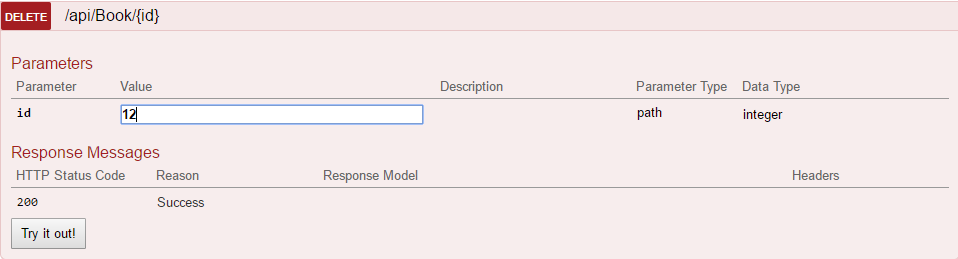
Primul apel API este cel de Show All Books. Acest apel se găsește la endpoint-ul /api/Book. Verbul HTTP care se folosește este GET. Nu este nevoie de parametri URL și nici parametri de tip date. În caz de succes se va returna o listă de cărți sau o listă vidă, având status code-ul 200. În caz de de eroare se va afișa o listă vidă și un status code cu valoarea 404 – NOT FOUND sau 500 – INTERNAL SERVER ERROR. Apelul este ilustrat și mai jos:



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 de eroare se va returna un status code cu valoarea 500 – INTERNAL SERVER ERROR. Apelul este ilustrat mai jos:



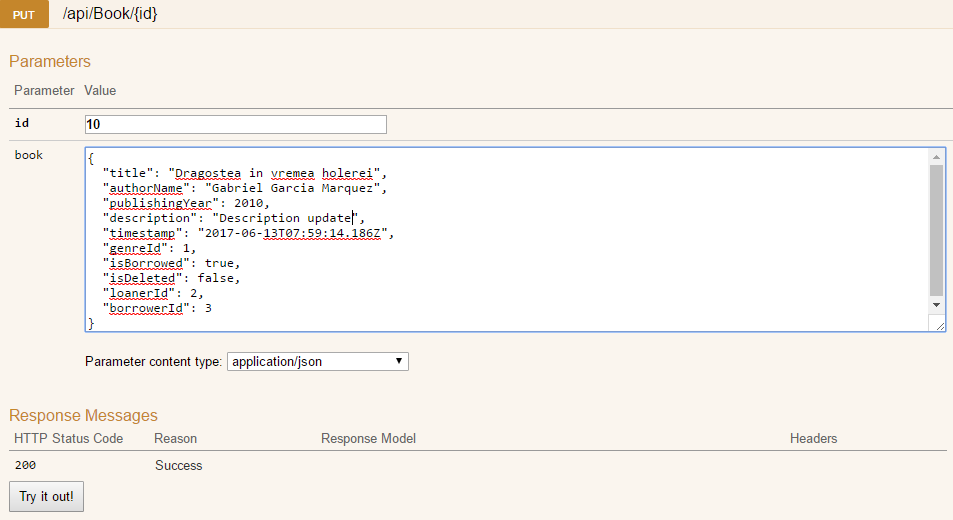
Al treilea apel este cel de Delete Book. Acest apel se găsește la endpoint-ul /api/Book/id. Verbul HTTP care se folosește este DELETE. Este necesar ca parametrul URL *id* să fie de tip întreg dar nu sunt necesari parametri de tip date. În caz de succes se va returna status code-ul 200. În caz de de eroare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



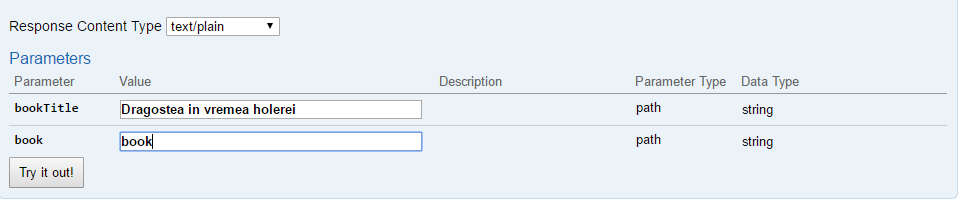
Al patrulea apel este cel de Get Book By Id. Acest apel se găsește la endpoint-ul /api/Book/id. Verbul HTTP care se folosește este GET. Este necesar ca parametrul URL *id* să fie de tip întreg dar nu sunt necesari parametri de tip date. În caz de succes se va returna status code-ul 200 și un obiect de 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:



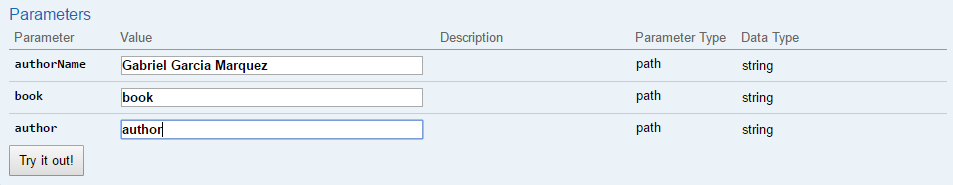
Al cincilea apel este cel de Update Book. Acest apel se găsește la endpoint-ul /api/Book/id. Verbul HTTP care se folosește este PUT. Este necesar ca parametrul URL *id* să fie de tip întreg și un parametru de tip carte. În caz de succes se va returna status code-ul 200. În caz de eroare se va returna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



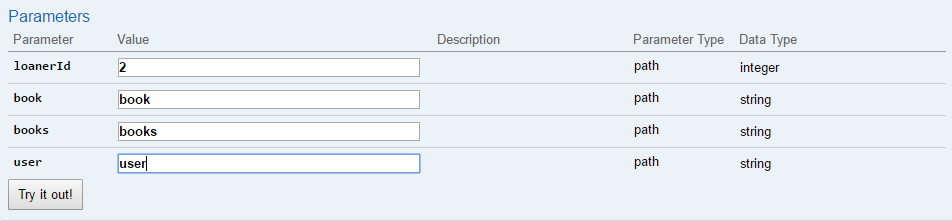
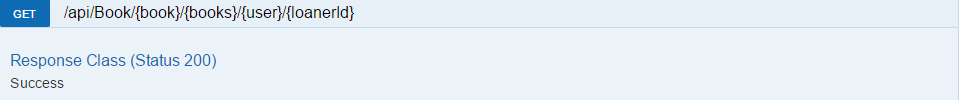
Al șaselea apel este cel de Get Book By Title. Acest apel se găsește la endpoint-ul /api/Book/book/bookTitle. Verbul HTTP care se folosește este GET. Este necesar ca parametrul URL *bookTitle* să fie de tip string dar nu sunt necesari parametri de tip date. De asemenea, parametrul URL *book* poate avea 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 care ilustrează apelul se poate vedea mai jos:



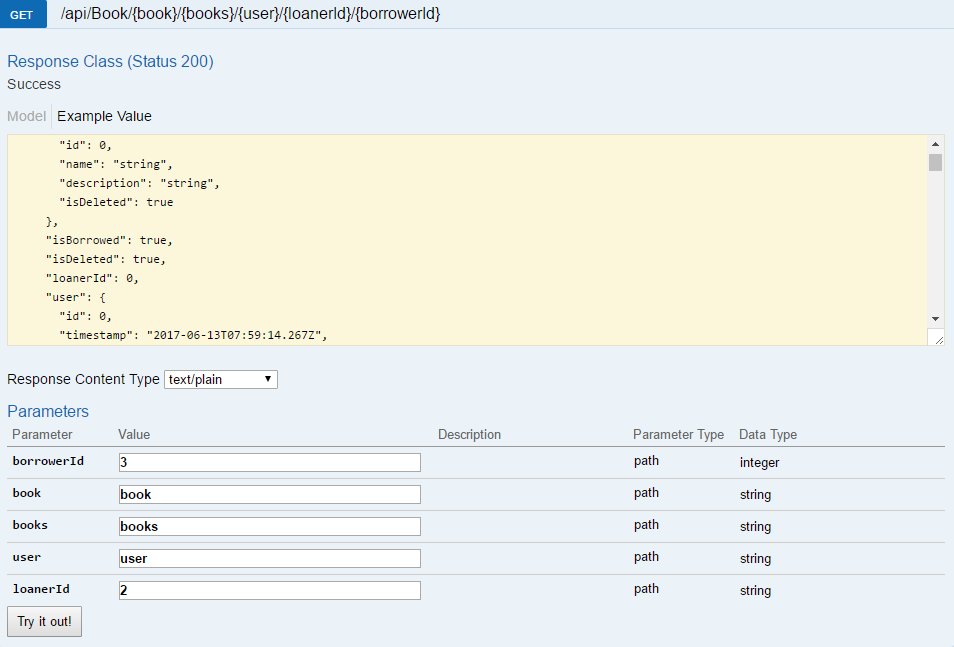
Al șaptelea apel este cel de Get Book By Author Name. Acest apel se găsește la endpoint-ul /api/Book/book/author/authorName. Verbul HTTP care se folosește este GET. Este necesar ca parametrul URL *authorName* să fie de tip string dar nu sunt necesari parametri de tip date. Ceilalți parametri URLpot avea 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:



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 URLpot 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:



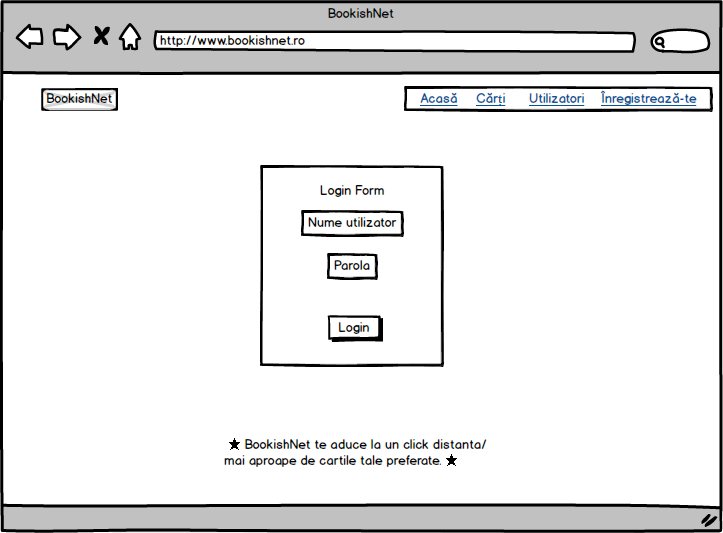
Al nouălea apel este cel de Get Book By Borrower Id. Acest apel se găsește la endpoint-ul /api/Book/book/books/user/loanerId/borrowerId. Verbul HTTP care se folosește este GET. Este necesar ca parametrul URL *borrowerId* să fie de tip întreg dar nu sunt necesari parametri de tip date. Ceilalți parametri URLpot 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 care ilustrează apelul se poate vedea mai jos:



## 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 de mai jos:

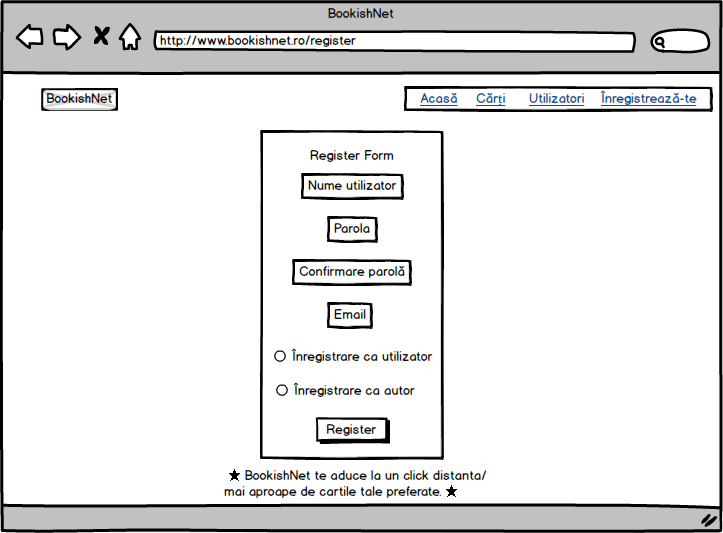


Ca și interacțiuni pe care le poate face utilizatorul amintim următoarele: link-urile de pe logo și *Acasă* din meniu te aduc pe pagina inițială. Link-ul *Cărți* face redirecționarea către pagina ***Books,*** pagină prezentată mai jos. Link-ul *Utilizatori* face redirecționarea către pagina ***Users***,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



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 păatrează, 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 următoare:

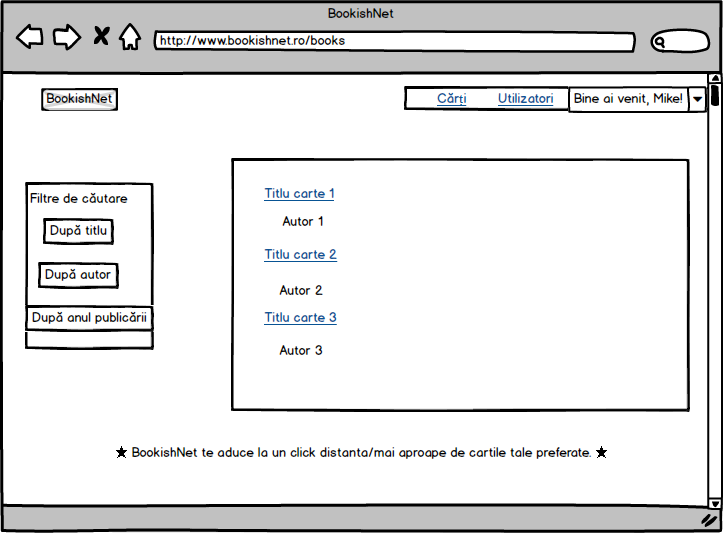


Pe această pagină utilizatorul poate interacționa cu elementele rămase și prezentate în paginile precedente. În plus, în meniul de drop-down există trei opțiuni: link-ul pentru *Acasă,* link-ul pentru *Profil* și link-ul *Deloghează-te.* Primul link a fost deja prezentat în ***Home Page.*** Interacțiunea 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 afișata 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 de mai jos:

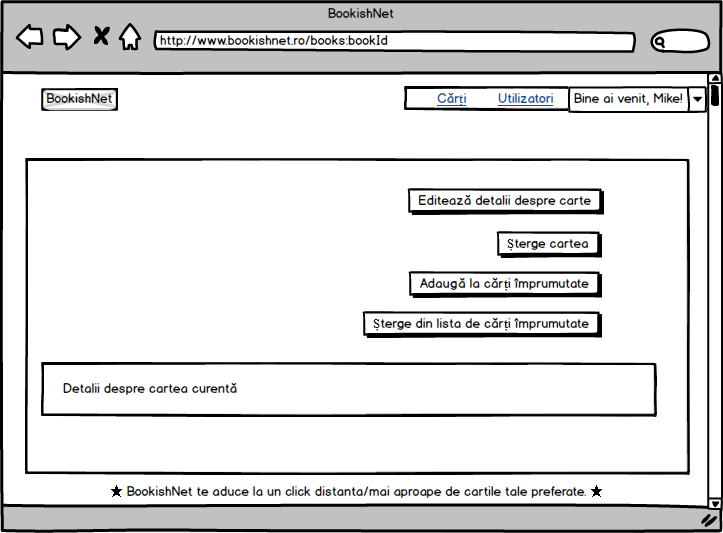


Pe lângă interacțiunile care s-au prezentat deja, utilizatorul (logat sau nu) va putea să folosească filtrele din partea stângă. Astfel, el va putea să caute o carte după nume, sau după numele autorului sau după anul în care a fost publicată cartea folosindu-se de câmpurile puse la dispoziție pentru 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 redirecționat 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 de mai jos ilustrează aspectul general al acestei pagini:

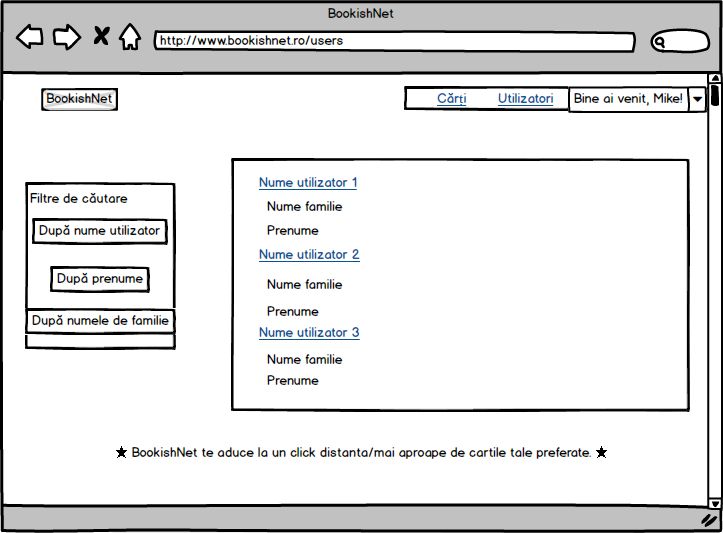


Primul lucru care se merită menționat este faptul că un utilizator care nu este logat nu poate avea acces la această pagină chiar și în condiția în care modifică manual adresa URL deoarece nu îi va fi afișat conținutul. De asemenea, pentru a avea vizibile butoanele este necesar ca respectiva carte să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

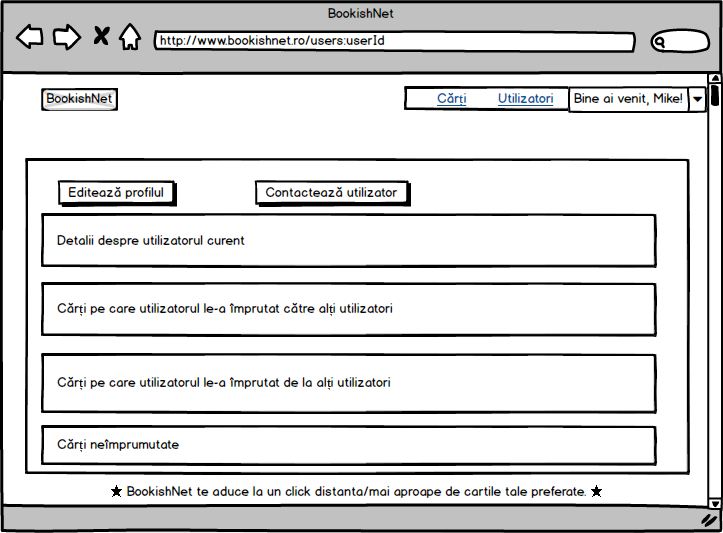


Ca și interacțiuni, utilizatorul (logat sau nu) va putea să folosească filtrele din partea stângă. Astfel, el va putea să caute după numele de utilizator, sau după numele de familie al acestuia sau după prenumele acestuia folosindu-se de câmpurile puse la dispoziție pentru a realiza aceste operații (ilustrate 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 de mai jos ilustrează aspectul general al acestei pagini:



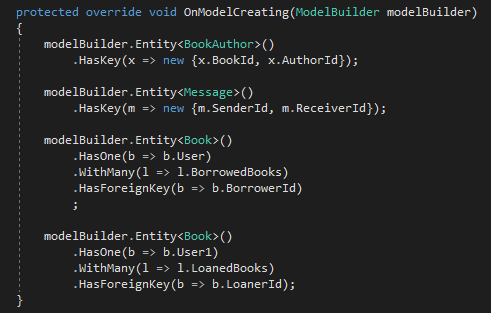
Cele două butoane nu vor fi afișate simultan în pagină. Astfel, dacă utilizatorul curent a intrat pe pagina sa de profil va putea interacționa cu butonul de editare care îi va deschide un modal pentru a-și edita detaliile despre profil. Dacă s-a intrat pe pagina unui alt utilizator atunci se va putea 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 Aproach[[13]](#footnote-13) și Code-First Approach[[14]](#footnote-14). 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.



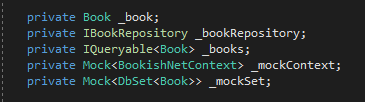
Partea mai dificilă de realizat la acest nivel al aplicației a fost legat de cheile primare alcătuite din două chei străine și numirea unor chei străine astfel încât să corespundă valoarile din model cu cele din tabela ***Book***. În acest sens, am suprascris metoda *OnModelCreating(ModelBuilder modelBuilder)* 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 precedentă. Pentru entitatea/modelul *Message* s-a folosit aceeași tehnică doar că diferă componența cheii primare: *SenderId* și *ReceiverId.*

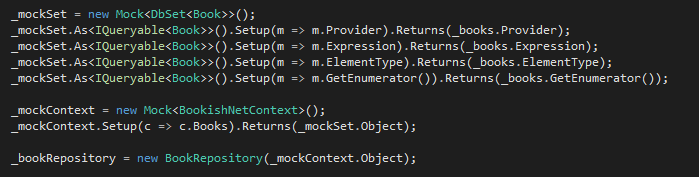
De asemenea, un alt lucru care s-a rezolvat suprascriind această metodă a fost specificarea clară a numelor a două câmpuri în tabela *Books*. Entity Framework Core creea de fiecare dată încă două coloane cu chei străine denumite *UserId* și *UserId1*, proprietăți care nu erau specificate în modelul *Book.* Evitarea și repararea acestei situații a fost făcută tot cu ajutorul EF Core Fluent API[[15]](#footnote-15). 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

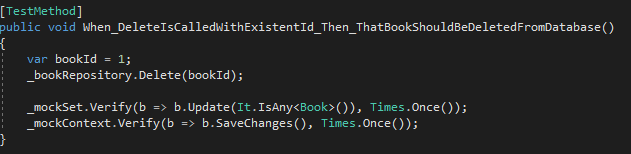
Pe partea de testare a acestui nivel al aplicației s-au folosit Mock-uri așa cum am menționat și în capitolul trei. În prima figură se poate vedea cum se declară un obiect de tip Mock. Se aseamănă cu Generics din Java. Se crează astfel variabilele cu următoarele nume: *\_book, \_bookRepository, \_books,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



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 înterogările 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.



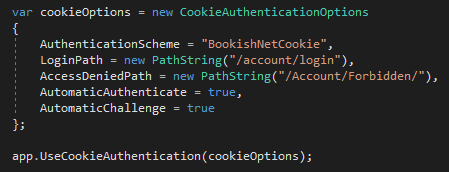
O metodă de test poate fi văzută în imaginea ce urmează. Metoda de față testează dacă o carte există în *\_mockSet* și se apelează metoda *Delete(int Id)* acea carte va fi înlăturată. Astfel, atribuim valoarea 1 (valoare ce există în lista de cărți) variabilei bookId. Apoi se apelează metoda de ștergere 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.



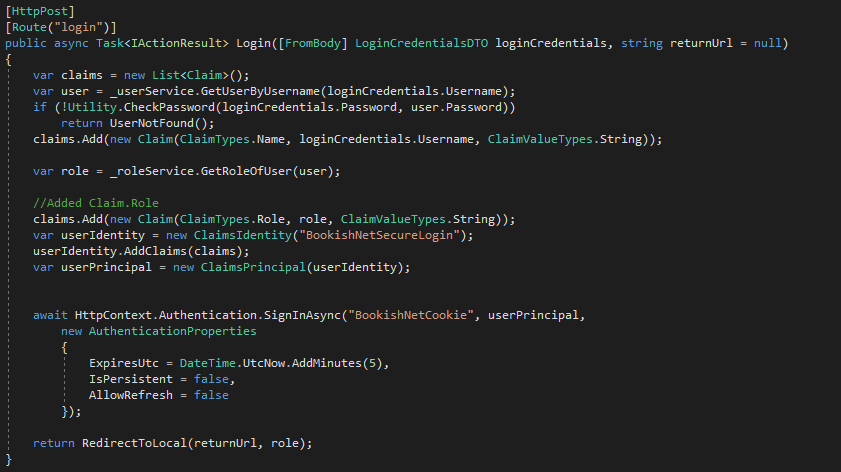
## MVC

### 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, *AuthomaticAuthenticate*  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, *AuthomaticChallenge* 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.*

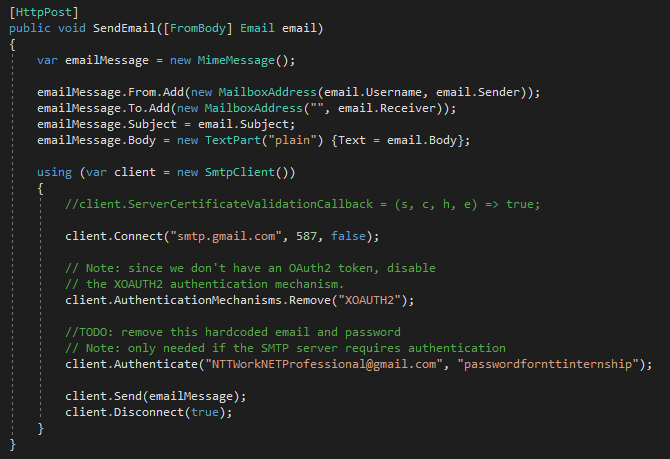


În imaginea precendentă s-au prezentat pașii pentru inițializarea cookie-ului. În cele ce urmează se vor prezenta următorii pași. Pentru a putea realiza autentificarea s-a creat o listă de *claim-uri[[16]](#footnote-16).*Se încearcă apoi aducerea din baza de date a utilizatorului cu respectivul nume 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.



### Trimitere email

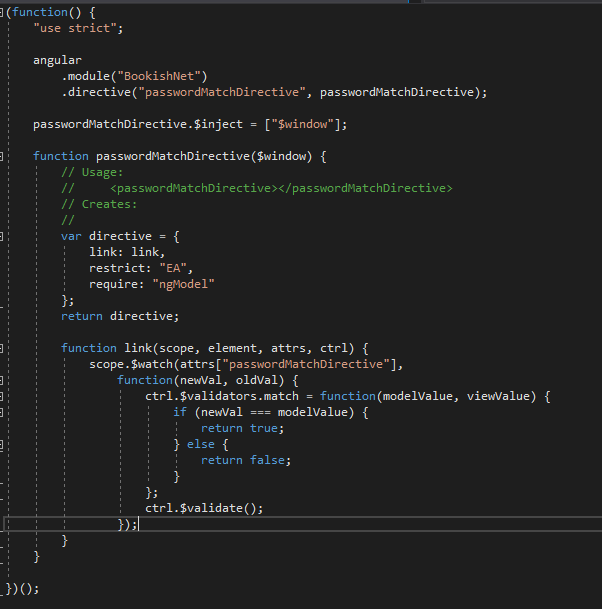
O altă funcționalitate interesantă a aplicației este că permite trimiterea de email-uri. Astfel, utilizatorii vor putea comunica între ei folosind editorul simplificat din interiorul aplicației. În cele ce urmează se poate vizualiza metoda care realizează această operație, *SendEmail*:



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 *SmtpClient* 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-ul prin apelarea metodei *Send,* după care am deconectat clientul conectat mai sus.

### Directiva pentru confirmarea parolei

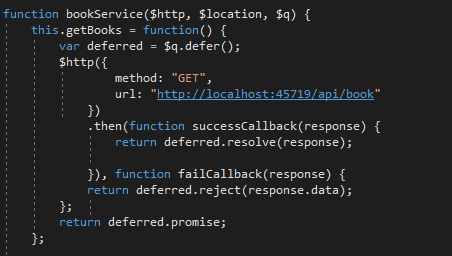


### BookService

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 figură nu se poate observa, la fel ca și la directiva de mai sus, pentru a folosi un anumit serviciu Angular (creat de noi sau existent implicit în framework) este nevoie de injectarea respectivului serviciu folosind proprietatea *$inject*. Cele trei servicii Angular injectate sunt *$hhtp, $location* și *$q,* pe care le-am trimis ca și parametri funcției *bookService()* (simulează într-un fel comportamentul unei contructor doar că nu creează obiecte noi, ci se aplică unui obiect de tip scope) pentru a le putea folosi în interiorul acesteia.



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 funcţiilor în mod asincron şi foloseşte valorile (sau excepţiile) returnate de aceste funcţii după ce acestea îşi încheie execuţia. Serviciul *$location* parsează URL-ul afişat în browser (se bazează pe acelaşi principiu ca *window.location* din Javascript) şi cu ajutorul lui se poate prelua URL-ul aplicaţiei. O schimbare în URL se reflectă în serviciul $location şi o schimbare în serviciul $location se reflectă în URL-ul afişat de browser.

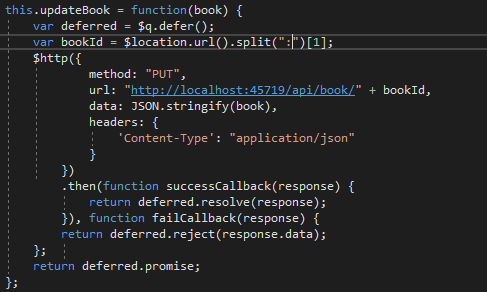
Pentru preluarea tuturor cărţilor disponibile în baza de date, am creat funcţia *getBooks().* Am creat variabila denumită *deffered* iar apelul către *$q.defered()* semnifică faptul că această variabilă va primi ca şi valoare un task ce se va executa în mod asincron.

Apoi s-a pregătit cererea HTTP, apelând serviciul *$http*. Am setat metoda cererii ca fiind de tip GET şi url-ul către [*http://localhost:45719/api/book*](http://localhost:45719/api/book)*,* adresă la care se găseşte resursa dorită. Metoda *.then()* se foloseşte de *deffered/promise Api* expus de către serviciul *$q.*

Astfel, în caz de succes vom returna un promise primit de la funcţia *deffered.resolve().* În caz de eroare la trimiterea răspunsului pentru cererea făcută *deffered.reject()* va returna un promise de respingere a cererii. În final, vom returna promise-ul obţinut; 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 proporţie de aproximativ 90% similare, în cele ce urmează voi încerca să pun în evidenţă doar deosebirile.



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

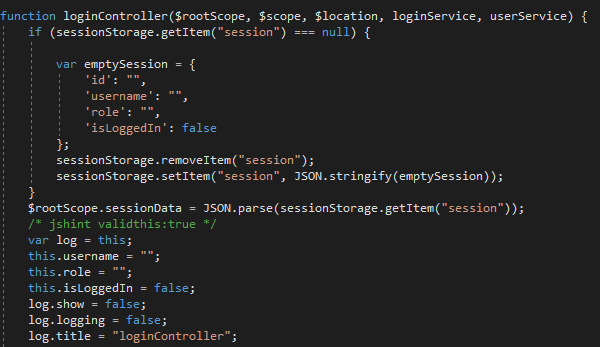
În serviciul $http apar şi aici modificări. Astfel, verbul HTTP folosit pentru parametrul method este PUT. Apoi, se foloseşte parametrul data pentru a trimite obiectul dorit. Am folosit metoda *JSON.stringify(book)* pentru a transforma obiectul de tip javascript book într-un obiect de tip string JSON. 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 şi decodării datelor transmise, JSON fiind şi un format uşor de citit. Celelalte funcţii sau variabile care nu s-au prezentat exclusiv au acelaşi comportament precum cele din funcţia *getBooks()*, prezentată anterior.

### LoginController

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

După cum am spus şi în subcapitolul Angular Controllers, am încercat, pe cât posibil, ca fiecare view să conţină un singur controller. Astfel, pagina de start a aplicaţiei este manageriată de *loginController.* Prima parte a codului se poate vedea în figura următoare:



În acest controller am injectat următoarele obiecte, servicii Angular dar şi servicii create de mine: *$rootScope, $scope, $location, loginService* şi *userService.* Fiecare aplicaţie are un singur scope rădăcină ($rootScope). Toate celelalte scope-uri sunt descendenţi ai rădăcinei. Obiectele de tip scope oferă sau asigură separarea între model şi 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 stabileşte legătura între client şi server, am realizat că este nevoie şi de o metodă de a ţine memorat acest fapt şi pe partea de client Angular. În acest fel, am putut să restricţionez accesul către anumite resurse, pagini sau acţiuni.

Astfel, cu aceste detalii în minte, am căutat o soluţie şi pentru partea de Angular şi am ajuns la concluzia că una din metodele disponibile este cea bazata 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 existenţei unui obiect în sesiune prin intermediul *sessionStorage.getItem(“session”).* Dacă nu exista acest obiect, se creea obiectul *emptySession* cu parametrii:

* *id* – pentru reţinerea identificatorului utilizatorului
* *username* – pentru stocarea numelui de utilizator
* *role* – pentru stocarea rolului utilizatorului
* *isLoggedIn –* pentru stocarea faptului că ultilizatorul 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 funcţia JSON.stringify) cu ajutorul următoarei linii: *sessionStorage.setItem(“session”, JSON.stringify(emptySession));.* Apoi am preluat acest obiect şi l-am adăugat pe $rootScope: *$rootScope.sessionData = JSON.parse(sessionStorage.getItem(“session”));.* Am apelat funcţia 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 şi 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 reţinerea rolului
* log.isLoggedIn – setat implicit pe false
* log.show – pentru afişarea erorii ,,Combinaţie greşită nume utilizator/parolă”. Setat implicit pe false
* log.logging – folosită pentru afişarea unui element ce arată faptul că cererii este în curs de procesare după solicitarea logării în aplicaţie. Setat implicit pe false
* log.title – stochează numele controllerului

În figura următoare se poate vedea efectiv metoda de logare în aplicaţie. 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 afişat utilizatorului în timpul procesării cererii prin intermediul *log.loadingLogin.* Se declară apoi variabila *username* care primeşte ca valoare obiectul trimis pe scope prin intermediul *log.username,* acelaşi lucru fiind similar şi în cazul variabilei *password.* Se creează apoi obiectul *dto*, folosind variabilele *username* şi *password*.

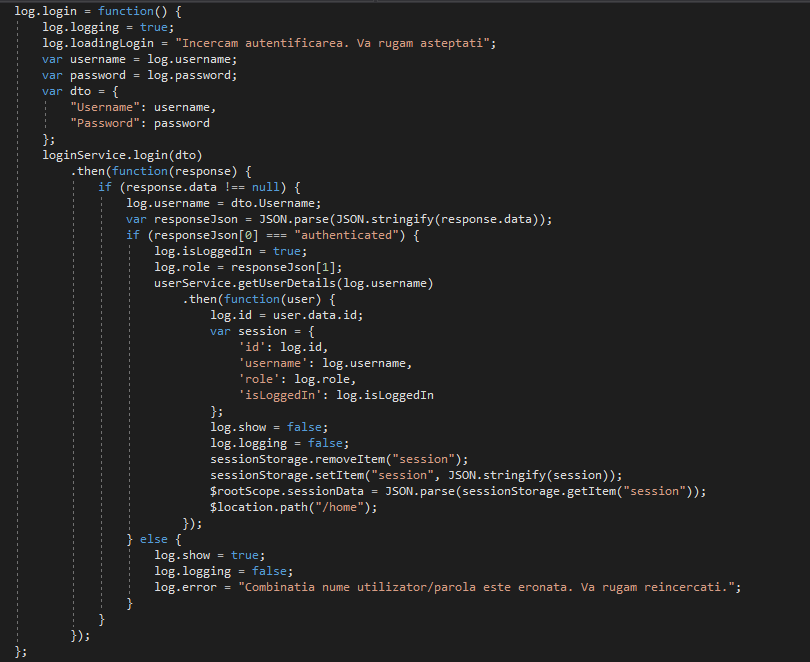
Având toate aceste pregătiri urmează apelul către *loginService* prin intermediul funcţiei *login(dto)* care primeşte ca parametru obiectul *dto* creat anterior. Fiind o cerere asincron nu vom primi avem nevoie de apelul către funcţia *.then()* care ne asigură faptul că îşi va continua execuţia după primirea promise-ului menţionat şi la *bookService*.

După primirea răspunsului, a promise-ului verificăm că există date primite. În caz contrar, lucrurile se termină repede. Execuţia funcţiei 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 condiţiei este adevărată se execută mai mulţi paşi. Setăm valoarea pentru *log.username* cu valoarea din *dto.Username.* Facem o parsare a răspunsului primit şi îl reţinem în variabila responseJson. Dacă prima parte a răspunsului are acelaşi tip şi are valoarea “authenticated”, urmează o nouă succesiune de paşi ce urmează a fi prezentaţi în următorul paragraf.

Se setează pe true valoarea pentru *log.isLoggedIn.* Se preia rolul utilizatorului în *log.role* din *responseJson[1].* Se apelează apoi funcţia *getUserDetails(username)* cu parametrul *log.username* din *userService.* Se apelează din nou funcţia *.then()* pentru aşteptarea 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 proprietăţi necesare pentru stocarea în sesiune. Se setează *log.show* şi *log.loading* pe false. Se înlătură obiectul creat inţial în sesiune şi se adaugă noul obiect folosind *session* apoi se preia valoarea în $rootScope şi se redirecţionează browserul către */home.*

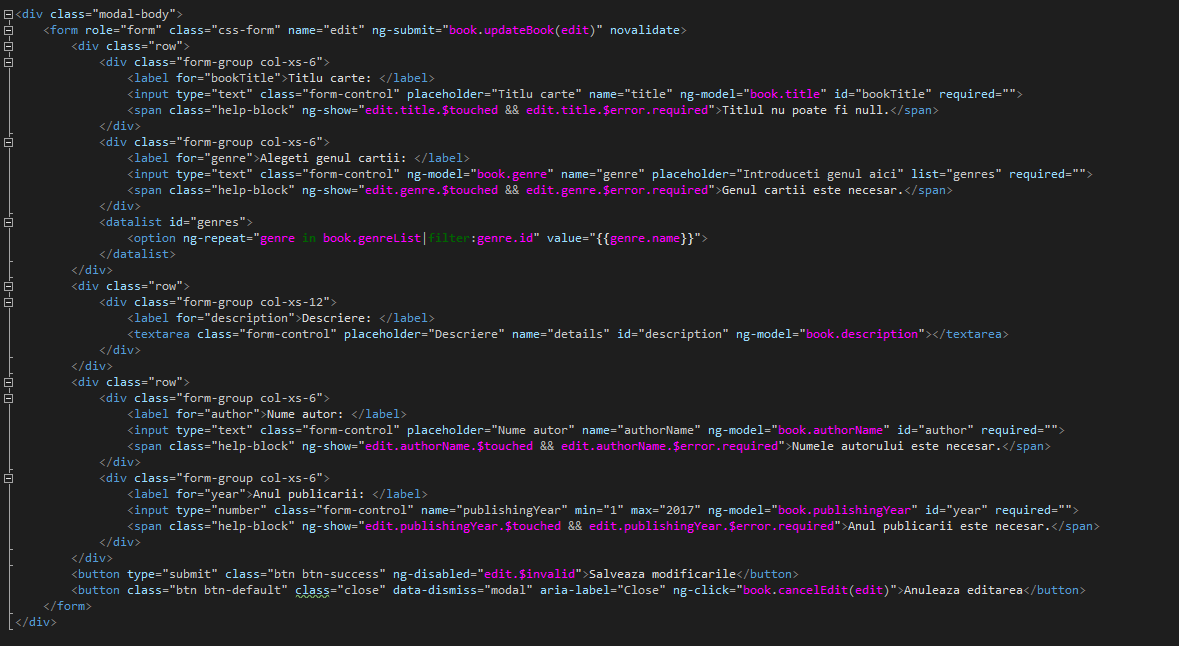
În caz contrar, lucrurile se termină repede. Se setează *log.show* pe true pentru a notifica view-ul că trebuie să afişeze mesajul de eroare, setăm *log.logging* pe false ca să oprim notificarea referitoare la încărcarea cererii şi apoi setăm un mesaj pentru *log.error,* mesaj ce va fi afişat în view,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



### View binding

Legătura între controller şi view se realizează prin aşa 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 şi 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 şi binding-ul de la controller către view, deşi nu am menţionat în mod explicit. Pentru a realiza aceasta voi folosi *BookEditor.html* view ce poate fi văzut în figura următoare:



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

Avem aici un formular de editare a unei cărţi care se deschide într-un modal. Acest formular conţine patru input-uri în care se pot introduce valori de la tastatură, un element asemănător unui meniu dropdown de unde poţi alege o opţiune, o zonă de text, mai mare decât un input, unde de asemenea se pot introduce valori de la tastatură, un buton de submit şi unul de anulare a operaţiei.

Elementele specific pentru Angular se pot identifica cel mai uşor dacă se urmăreşte 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 acţiunea ce se va realiza la trimiterea formularului. Astfel, la realizarea acestei acţiuni se va apela funcţia *book.updateBook(edit).* Deoarece această funcţie nu a fost creată pe scope este necesară folosirea şi a variabilei *book* care reprezintă, de fapt, bookController.

Directiva ng-show este folosită pentru a transmite către browser ce elemente se vor afişa în pagină şi care nu. De menţionat faptul că ele se creează în DOM, dar nu se afişează efectiv. În cazul de faţă această directivă a fost folosită pentru afişarea anumitor texte ajutătoare cu privire la completarea corectă a formularului. Unele câmpuri sunt obligatorii, altele nu.

Directiva *ng-click* este folosită pentru a specifica acţiunea ce se va realiza la apăsarea butonului de anulare a operaţiei. Astfel, la realizarea acestei acţiuni se va apela funcţia *book.cancelBook(edit).* Deoarece nici această funcţie nu a fost creată pe scope este necesară folosirea şi a variabilei *book* care reprezintă, de fapt, bookController.

Directiva *ng-repeat* este folosită pentru a introduce de obicei o listă de elemente şi este şi preferata mea. Nu este restricţionată de tipul elementului ce se doreşte a fi introdus. În cazul de faţă *ng-repeat* creează mai multe opţiuni pentru elementul *datalist.* Ea funcţionează asemănător cu o instrucţiune for. În cazul de faţă, se ia câte un *genre* din *book.genreList* şi se face o filtrare după id-ul genului. Elementele ce vor fi afişate vor lua valoarea dată de *{{genre.name}}.* În funcţie de dimensiunea *book.genreList* vor fi şi atâtea opţiuni disponibile. Binding-ul se face cu ajutorul parantezelor {{}}.

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

## MVC Tests

# Manual de utilizare

# Referințe

1. Obiceiuri lectură - http://www.ires.com.ro/articol/172/obiceiurile-de-lectura-ale-romanilor [↑](#footnote-ref-1)
2. Navigare pe Internet - http://ec.europa.eu/eurostat/statistics-explained/index.php/Information\_society\_statistics\_-\_households\_and\_individuals/ro [↑](#footnote-ref-2)
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). [↑](#footnote-ref-3)
4. SPA(Single-page application) – aplicaţie de tip web sau site web al cărui conţinut să poate fi mapat pe o singură pagină, urmărindu-se astfel simularea unei aplicaţii desktop ca şi experienţă a utilizatorului. Într-o asemenea aplicaţie, fie tot codul necesar – HTML, Javascript, CSS – este primit la o singură încărcare de pagină, fie anumite resurse sunt încărcate în mod dynamic şi adăugate în pagină, ca răspuns la acţiunile utilizatorului. [↑](#footnote-ref-4)
5. **Representational state transfer -** Stil architectural de dezvoltare a aplicaţiilor web cu focalizare asupra reprezentării datelor [↑](#footnote-ref-5)
6. **API -** interfaţă de programare a unei aplicaţii pentru un server sau un browser web. Este un concept de dezvoltare web, este folosit de obicei de o aplicaţie web pe partea de client [↑](#footnote-ref-6)
7. CRUD – Create, Read, Update, Delete respectiv operaţiile de adăugare, citire, modificare şi ştergere [↑](#footnote-ref-7)
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.* [↑](#footnote-ref-8)
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 variatate de limbaje de programare. Cele mai folosite framework-uri în Visual sunt: MSTest/Visual Studio, NUnit și xUnit.NET. [↑](#footnote-ref-9)
10. Strategie folosită pentru testele de integrare unde prima data sunt testate componentele din nivelele de mai jos și apoi acestea sunt folosite pentru a facilita testarea componentelor de nivel mai înalt. [↑](#footnote-ref-10)
11. 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. [↑](#footnote-ref-11)
12. Oferirea posibilităţii de a reactiva anumite înregistrări care s-au droit a fi eliminate în trecut. Renunţarea la metoda clasică de ştergere a înregistrărilor din baza de date fără posibilitate de recuperare. [↑](#footnote-ref-12)
13. 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. [↑](#footnote-ref-13)
14. 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. [↑](#footnote-ref-14)
15. Oferă metode pentru configurarea diferitelor aspecte, proprietăți referitoare la un model anume. [↑](#footnote-ref-15)
16. O pereche nume-valoare ce reprezintă ce anume este subiectul și nu ce poate face acesta. [↑](#footnote-ref-16)