Cuprins

[Introducere 3](#_Toc484511485)

[Motivaţie 3](#_Toc484511486)

[Context 3](#_Toc484511487)

[Cerinţe funcţionale 3](#_Toc484511488)

[Abordare tehnică 4](#_Toc484511489)

[Baza de date 4](#_Toc484511490)

[Backend 4](#_Toc484511491)

[Frontend 4](#_Toc484511492)

[Framework-uri 4](#_Toc484511493)

[Entity Framework Core 4](#_Toc484511494)

[Bootstrap 5](#_Toc484511495)

[Selenium 5](#_Toc484511496)

[Contribuţii 5](#_Toc484511497)

[Proiectare 6](#_Toc484511498)

[Arhitectura soluției 6](#_Toc484511499)

[Capitolul 1: Dezvoltarea aplicaţiei client 6](#_Toc484511500)

[1.1. Arhitectura generală a aplicației 7](#_Toc484511501)

[1.2. Angular Views 7](#_Toc484511502)

[1.3. Angular Controllers 8](#_Toc484511503)

[1.4. Angular Services 8](#_Toc484511504)

[Capitolul 2: Dezvoltarea aplicaţiei server 9](#_Toc484511505)

[2.1. Arhitectura generală a aplicaţiei 9](#_Toc484511506)

[2.2. API Controllers 9](#_Toc484511507)

[2.3. Service Layer 10](#_Toc484511508)

[2.4. Data Layer 10](#_Toc484511509)

[Capitolul 3: Testarea funcţionalităţilor şi asigurarea calităţii 11](#_Toc484511510)

[1. Unit tests 11](#_Toc484511511)

[2. Integration tests 13](#_Toc484511512)

[3. Acceptance tests 13](#_Toc484511513)

[Modelarea datelor 13](#_Toc484511514)

[Arhitectura bazei de date 13](#_Toc484511515)

[Entităţi 14](#_Toc484511516)

[1. Books 14](#_Toc484511517)

[2. Reviews 14](#_Toc484511518)

[3. Genres 14](#_Toc484511519)

[4. BookAuthor 14](#_Toc484511520)

[5. Users 14](#_Toc484511521)

[6. Roles 15](#_Toc484511522)

[7. Messages 15](#_Toc484511523)

[Relaţii între entităţi 15](#_Toc484511524)

[1. Users-Books 15](#_Toc484511525)

[2. Users-Messages 15](#_Toc484511526)

[3. Users-Roles 15](#_Toc484511527)

[4. Books-Reviews 16](#_Toc484511528)

[5. Books-Genres 16](#_Toc484511529)

[Protocoale de comunicare client – server 16](#_Toc484511530)

[Interfața cu utilizatorul 16](#_Toc484511531)

[Implementare 16](#_Toc484511532)

[Manual de utilizare 16](#_Toc484511533)

[Referințe 16](#_Toc484511534)

# 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 sau 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.

# 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

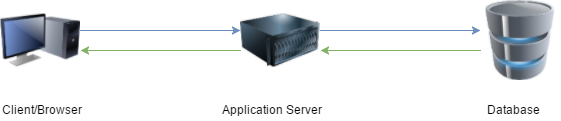
Testarea funţionalităţii şi asigurarea calităţii are ca scopuri esenţiale asigurarea faptului că aplicaţia respectă cerinţele funcţionale şi posibilitatea de dezvoltare, îmbunătăţire, aspecte ce pot fi aduse pe viitor:

* 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:

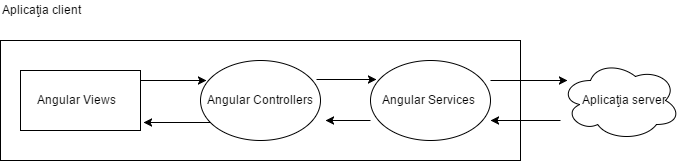


### Capitolul 1: Dezvoltarea aplicaţiei client

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[[3]](#footnote-3) 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

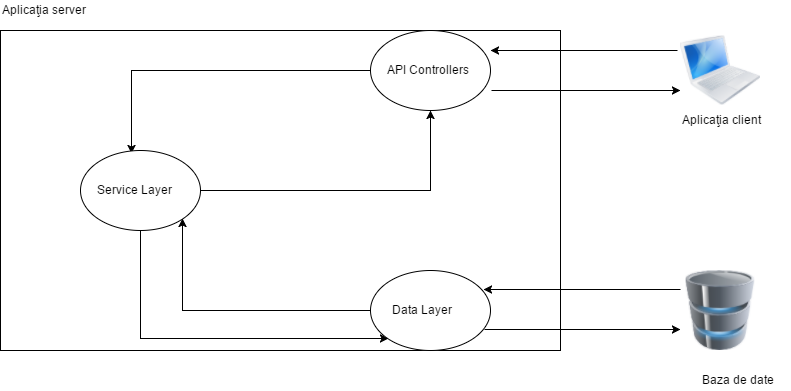
De asemenea, serviciile Angular sunt cele care fac cererile către API-ul aplicaţiei, un API care oferă resursele necesare aplicaţiei client. Această componentă va fi detaliată mai pe larg în capitolul dedicate aplicaţiei server. Aşadar, serviciile sunt cele care fac legătura între aplicaţia server şi 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[[4]](#footnote-4) realizat folosind Web API[[5]](#footnote-5). 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[[6]](#footnote-6) 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[[7]](#footnote-7). 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.

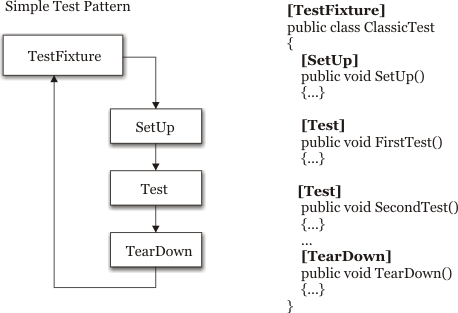
Această tehnică are și avantaje și dezavantaje pe care le voi prezenta pe scurt în cele ce urmează. Ca și dezavantaje ale scrierii de Unit Teste putem enumera: timpul alocat pentru a realiza teste potrivite poate fi semnificativ, nu toate erorile pot fi detectate aici – spre exemplu, cele de integrare.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



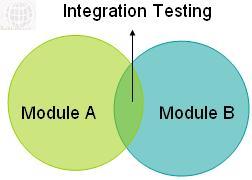
Când se dorește crearea unei clase de test se folosește o adnotare deasupra acesteia. Această adnotare diferă în funcție de framework-ul[[8]](#footnote-8) folosit. În cazul de față, adnotarea *TestFixture* este specifică framework-ului ***NUnit.*** Ceea ce face de fapt acest atribut este faptul că marchează 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.

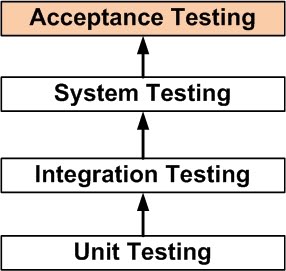
#### Integration tests



#### Acceptance tests

Pentru a putea verifica că aplicația îndeplinește cerințele funcționale din punctul de vedere al utilizatorului final am ales să folosesc strategia numită Acceptance testing[[9]](#footnote-9). 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.

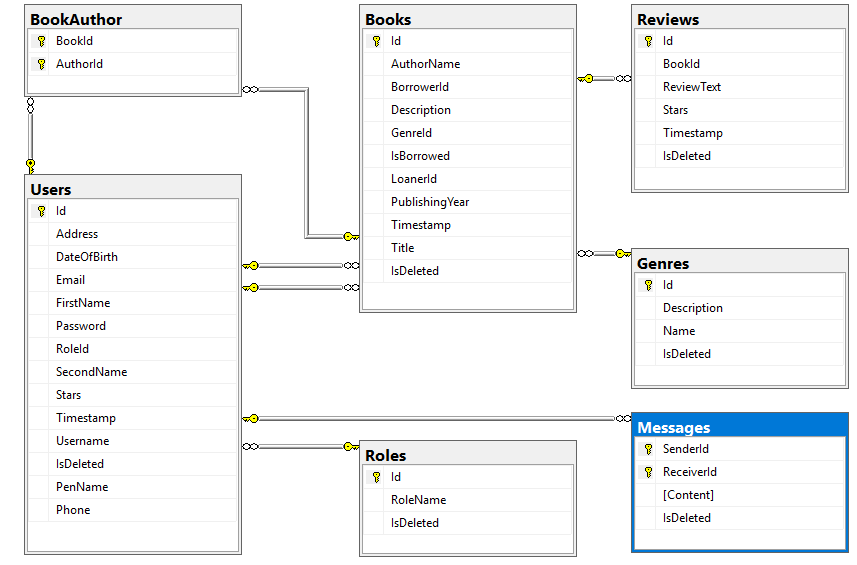
Un alt lucru ce merită menționat este că aceste teste sunt automate. Astfel, am renunțat la varianta de a le folosi pe cele manuale. Un beneficiu major al acestui lucru este faptul că le scrii o singură data și apoi le poți reutiliza, comparativ cu cele manuale pe care ești nevoit să le iei de fiecare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



## 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[[10]](#footnote-10), 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

## Interfața cu utilizatorul

# Implementare

# 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. 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-3)
4. Stil architectural de dezvoltare a aplicaţiilor web cu focalizare asupra reprezentării datelor [↑](#footnote-ref-4)
5. **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-5)
6. CRUD – Create, Read, Update, Delete respectiv operaţiile de adăugare, citire, modificare şi ştergere [↑](#footnote-ref-6)
7. ***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-7)
8. 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-8)
9. 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-9)
10. 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-10)