



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Matók Kinga

KÖNYV KÖZPONTÚ KÖZÖSSÉGI OLDAL TERVEZÉSE ÉS MEGVALÓSÍTÁSA

DR. EKLER PÉTER

BUDAPEST, 2021

Tartalomjegyzék

Összefoglaló	5
Abstract.....	6
1 Bevezetés	7
1.1 Témaválasztás	7
1.2 Áttekintés	8
2 Felhasznált technológiák	9
2.1 Backend	9
2.1.1 Spring Boot.....	9
2.1.2 JPA.....	9
2.1.3 Hibernate.....	9
2.1.4 Microsoft SQL Server Express	10
2.2 Frontend	10
2.3 Frontend és Backend közötti kommunikáció.....	11
2.3.1 HTTP	11
2.3.2 JWT.....	12
2.3.3 WebSocket	12
3 Feladatspecifikáció.....	14
3.1 Feladat részletes leírása.....	14
3.2 Használati eset diagram	15
4 A rendszer architektúrája.....	16
4.1 Az alkalmazás felépítése.....	16
4.2 Adatbázis séma	17
4.3 Backend architektúra	20
4.4 Frontend architektúra	21
5 Megvalósítás	23
5.1 Backend	23
5.1.1 Adatelérési réteg	23
5.1.2 Üzleti logikai réteg.....	25
5.1.3 REST API	27
5.1.4 WebSocket	28
5.1.5 Spring Security	31

5.2 Frontend	31
5.2.1 Regisztráció	32
5.2.2 Bejelentkezés	33
5.2.3 Profil	34
5.2.4 Keresés.....	35
5.2.5 Könyv kezelése gyűjteményben	37
5.2.6 Barátság	38
5.2.7 Chat.....	40
6 Tesztelés	41
7 Összefoglalás és továbbfejlesztési lehetőségek	43
8 Telepítési útmutató	44
9 Irodalomjegyzék.....	45

HALLGATÓI NYILATKOZAT

Alulírott **Matók Kinga**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2021. 12. 10.

.....
Matók Kinga

Összefoglaló

A közösségi portálok manapság mindennapjaink szerves részévé váltak, az élet mindannyi területén megjelentek. Használatuk elterjedéséhez nagyban hozzájárul, hogy a legtöbb interneteléréssel rendelkező eszközről elérhetőek és használhatóak. Ezek az eszközök a technológia fejlődésével és terjedésével már a háztartások nagyrésztében megtalálhatóak.

Ezen a gondolatmeneten haladva pedig kézenfekvő, hogy egyes érdeklődési körökhöz, hobbihoz saját közösséget építeni legegyszerűbb egy kifejezetten ezen célú közösségi portál segítségével lehet. Mára több könyvkatalógus jellegű weboldal jött létre, amelyeknek funkciói közé tartoznak a könyvek adatbázisában való keresés, saját listák létrehozása és könyvek értékelése. Azonban olyan közösségi portál még nem jött létre, amelynek kifejezetten a könyvolvasók közösségének építése lenne a célja, az alkalmazásom ezt az űrt hivatott betölteni.

A szakdolgozat célja egy olyan webalkalmazás létrehozása, amely egy közösségi felületet biztosít könyvolvasók számára. Az alkalmazás frontend részéhez felhasznált technológia az Angular keretrendszer, a backend részéhez pedig a Spring Boot keretrendszer. Az alkalmazással szemben támasztott főbb követelmények, hogy a felhasználó tudjon regisztrálni, vagyis profilt létrehozni, be tudjon jelentkezni, és beszélgetést tudjon kezdeményezni más felhasználóval.

Abstract

Nowadays social media websites have become an integral part of our everyday lives. The fact that they can be reached from almost any device with an internet connection largely contributes to their wide spread over the world. Due to the development of technology these devices can be found in most households.

It seems that the easiest way to build your own community for certain interests and hobbies is on a social media website created for this exact purpose. Today, several book-catalog type websites have been created, the functions of which include searching their database, creating your own lists of books, and writing reviews. Still, no such website has been created yet, that has been designed to build a community of book readers, using chat functions. This is the gap, that my application is meant to fill.

The aim of the thesis is to create a web application that provides a community interface for book readers. The technology used to create the application is Angular framework for the frontend part and Spring Boot framework for the backend part. The main requirements for the application are that the user can create their own profile, log in, and initiate a conversation with another user.

1 Bevezetés

Ebben a fejezetben bemutatom a témaválasztásom hátterét, majd áttekintést adok a dolgozat tartalmi felépítéséről.

1.1 Témaválasztás

Az internet elterjedésével létrejött egy új informatikai ágazat, a webfejlesztés, melynek célja webalkalmazások létrehozása. A webalkalmazás röviden egy olyan applikáció, amely a böngészőben fut, azaz web szerveren, nem pedig lokálisan, a felhasználó eszközén. A különböző fejlesztési technológiák napról-napra fejlődnek, a felhasználói, valamint a fejlesztői elvárásokhoz igazodva. A webalkalmazások népszerűségének növekedésével megjelentek különböző alkalmazásfejlesztési keretrendszerek, melyek célja, hogy gyorsítsa és egyszerűsítse a fejlesztők munkáját azzal, hogy a leggyakrabban használt elemeket egységesítik és készen szolgáltatják. Ezeknek a keretrendszereknek a felhasználásával készülnek a modern Single Page Application (SPA) webalkalmazások, amelyek működésének az alapja az, hogy az alkalmazás dinamikusan frissíti az oldal tartalmát, nem navigál át másikra.

A webfejlesztőknek manapság már nem csak az asztali számítógépeken futtatott böngészőkkel való kompatibilitást kell figyelembe venniük a fejlesztés folyamata közben, hanem további okos eszközöket, okos telefonokat, táblagépeket, amelyekről az internet használatával a weboldalak szintén elérhetőek. Egy webalkalmazástól elvárás, hogy reszponzív legyen, felülete alkalmazkodjon a felhasználó eszközének képernyőfelbontásához. Ennek az alkalmazkodóképességnek az elérése a fejlesztő feladata.

Számomra a webfejlesztés a felhasználóközelségével, a grafikus felületnek köszönhetően a fejlesztés közbeni látványos haladással, valamint kézzelfogható végeredményével keltette fel az érdeklődésemet. Specifikusan a közösségi oldalt azért választottam, hogy a szakdolgozat elkészítése alatt megismerhessem és átláthassam egy sok felhasználót összekötő weboldal működését és felépítését. Ennek keretein belül is, egy azonnali üzenetküldő rendszer (chat) megvalósítása volt a célom.

Motivációm volt továbbá még a GoodReads, valamint Moly.hu nevű könyves közösségi oldalak, amelyek lehetőséget adnak a felhasználóknak az olvasmányaikat

listákba rendezni, értékelni, azokról véleményt írni. A dolgozatomban egy ezekhez hasonló portál elkészítését mutatom be.

1.2 Áttekintés

A következőkben ismertetem a szakdolgozatom részletes felépítését. A második fejezetben kerülnek bemutatásra a felhasznált technológiák, kliens, valamint szerver oldalon és két, a feladatomhoz hasonló weboldal bemutatása, amelyekből ötleteket merítettem. A harmadik fejezetben a feladat részletes leírása, valamint használati eset (use case) diagram szerepel, amelyen a program funkciói szerepelnek. A negyedik fejezetben magas szinten bemutatásra kerül a rendszer architektúrája, a felépítése és a komponensei, mind backend, mind frontend oldalon. Az ötödik fejezetben szerepel a program részletes megvalósításának leírása, ehhez társulnak diagramok, valamint kódrészletek. A hatodik fejezetben szerepel az elvégzett tesztek bemutatása. A hetedik fejezetben összefoglalom a dolgozat készítése alatt szerzett tapasztalatokat, valamint kifejtem a továbbfejlesztési lehetőségeket. A nyolcadik fejezetben az alkalmazás telepítéséhez szükséges lépések szerepelnek.

2 Felhasznált technológiák

Ez a fejezet áttekintést ad az általam felhasznált technológiákról.

2.1 Backend

Ebben az alfejezetben a szerver oldalon (backend) felhasznált technológiát mutatom be. A backend fejlesztéseket *Eclipse* fejlesztői környezetben végeztem, az adatbázist *Microsoft SQL Management Studio*-ban kezeltem.

2.1.1 Spring Boot

A *Spring Framework* [1] egy nyíltforráskódú keretrendszer, amely átfogó programozási és konfigurációs modellt biztosít *Java* alapú alkalmazásoknak, attól függetlenül, hogy azok milyen platformra készülnek.

A *Spring Boot* [2] a *Spring Framework*nek egy kiegészítése, amelynek segítségével egyszerűen lehet önálló, nagyvállalati minőségű *Spring* alapú alkalmazásokat létrehozni, amit „csak futtatni kell”. Minimális konfigurációt igényelnek felhasználói oldalról, ugyanis egy *Spring Boot* projekt létrehozása egy generált kezdőprojekttel történik, amiben a *Spring* definiál alapvető függőségeket és konfigurációkat az alapján, hogy a felhasználó milyen céllal/célokkal készíti a programot.

2.1.2 JPA

A *Java Persistence API (JPA)* [3] szabványosítja a perzisztencia azaz adatmegmaradás használatát a *Java* platformon, ezzel biztosítva a szabványos mechanizmusokat az *objektum-relációs leképezéshez (ORM – Object Relational Mapping)*. A *Java* nyelvben az ORM-et megvalósító réteg felelős azért, hogy átalakítsa a *Java* osztályokat és objektumokat, hogy azok relációs adatbázisban tárolhatóak és kezelhetőek legyenek.

2.1.3 Hibernate

ORM keretrendszerként a *Hibernate* [4] az szintén az adatok perzisztenciájával foglalkozik. A saját "natív" *Application Programming Interface (API)*-ja mellett a *Hibernate* a *JPA* specifikáció implementációja is. Mint ilyen, könnyen használható bármilyen *JPA*-t támogató környezetben, beleértve *Spring Boot*ot is.

A *Hibernate* támogatja a lusta inicializálást, számos lekérési stratégiát és az optimista zárolást automatikus verziókezeléssel és időbélyegzéssel. Teljesítmény szempontjából az előnye az, hogy a *Structured Query Language (SQL)* nagy részét a rendszer inicializálásakor állítja elő, nem pedig futás közben.

2.1.4 Microsoft SQL Server Express

A *Microsoft SQL Server Express* [5] a *Microsoft SQL Server* relációs adatbázis-kezelő rendszerének egy verziója, amely ingyenesen letölthető, terjeszthető és használható. Kifejezetten beágyazott és kisebb méretű alkalmazásokhoz készült adatbázist tartalmaz.

A *Microsoft SQL Server Express LocalDB* [5] a *Microsoft SQL Server Express* egyik verziója, az *SQL Server* motor igény szerinti felügyelt példánya. Ez egy kifejezetten fejlesztőknek szóló szolgáltatás, ezért korlátozásokkal rendelkezik. Legfeljebb 10 Gb lehet az adatbázis mérete, valamint csak lokális hálózatról lehet hozzá kapcsolódni.

2.2 Frontend

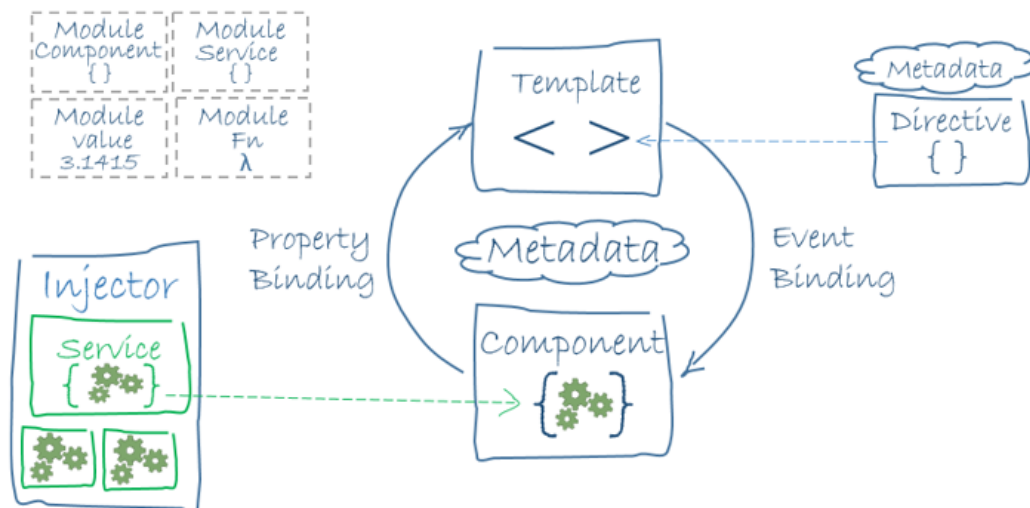
Ebben az alfejezetben bemutatom a kliens oldalon (frontend) felhasznált technológiákat. A fejlesztést *Visual Studio Code* fejlesztői környezetben végeztem.

Az *Angular* [6] egy *TypeScript* nyelven írt, nyíltforráskódú webalkalmazás keretrendszer. Használatával dinamikus *SPA (Single Page Application)* alkalmazások készíthetők *Hyper Text Markup Language (HTML)* és *TypeScript* alapokon. Olyan alapvető és opcionális funkcionalitásokat implementál *TypeScript* könyvárakként, amelyek bármely *Angular* alkalmazásba importálhatóak.

Az *Angular* alapvető építőkövei a komponensek, amelyek funkcionalitás szerint modulokba rendeződnek. Egy alkalmazás modulok összességéből áll, minden alkalmazásban legalább a gyökér modulnak léteznie kell.

A komponensek nézeteket definiálnak, amelyek a képernyőn található elemekből állnak. Ezek közül az *Angular* választhat és módosíthat a program logikájának és adatainak megfelelően.

A komponensek szolgáltatásokat használnak, amelyek nézetekhez nem köthető funkciókat valósítanak meg. A szolgáltatások a komponensekbe függőségként beilleszthetők, így a kód moduláris, újrafelhasználhatóvá és hatékonyabbá válik.



1. ábra - Angular architektúra (Forrás: <https://angular.io/guide/architecture>)

A *Bootstrap* [7] egy nyílt forráskódú keretrendszer mely *HTML*, *Cascading Style Sheets* (CSS) és *JavaScript* technológiákat használ. Használatával könnyedén lehet bármilyen képernyőmérethez alkalmazkodó weboldalat készíteni.

Az *Angular Material* [8] egy olyan felhasználói felület komponenskönyvtár, amely a *Material Design*-t Angularban valósítja meg.

A *Material Design* [9] egy Google által fejlesztett formanyelv (design language). A formái letisztultak, gyakran használva a rácsos elrendezési formát. Rendelkezik rezponzív animációkkal, áttünésekkel, kitöltésekkel és a mélység hatását keltő effektekkel, mint a megvilágítás és az árnyékok vetése.

2.3 Frontend és Backend közötti kommunikáció

Ebben az alfejezetben bemutatásra kerülnek az általam választott technológiákat a kliens és szerver közti kommunikáció megvalósítására.

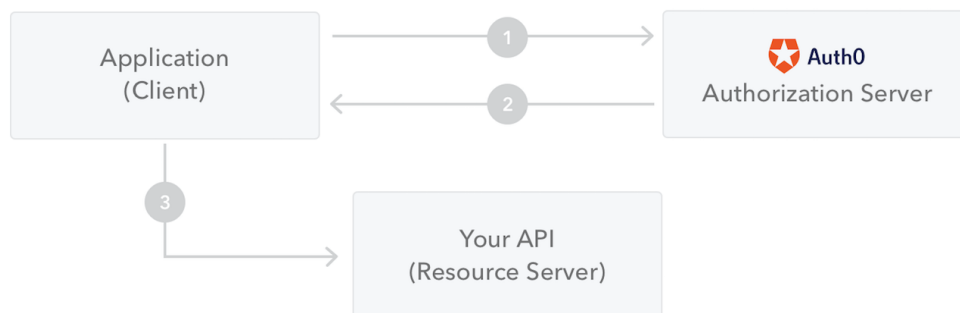
2.3.1 HTTP

A *Hypertext Transfer Protocol* (HTTP) [10] egy alkalmazásrétegű protokoll hipermédiás dokumentumok, például *HTML* továbbítására. Webböngészők és webszerverek közötti kommunikációra tervezték, de más célokra is használható. A HTTP a klasszikus kliens-szerver modellt követi, az kliens megnyitja a kapcsolatot, hogy kérést küldjön, majd megvárja, amíg választ nem kap. A HTTP állapot nélküli protokoll, ami azt jelenti, hogy a szerver nem tárol semmilyen adatot (állapotot) két kérés között.

2.3.2 JWT

A *JSON Web Token (JWT)* [11] egy nyílt szabvány (RFC 7519), amely kompakt és önálló módszert határoz meg az információk biztonságos továbbítására a felek között *JSON-objektumként*. Ez az információ ellenőrizhető és megbízható, mert digitálisan alá van írva. A *JWT*-k aláírhatók titkos vagy nyilvános/privát kulcspárral. Az aláírt tokenek ellenőrizhetik a benne foglalt követelések sértetlenségét, míg a titkosított tokenek elrejtik ezeket a követeléseket más felek elől. Amikor a tokeneket nyilvános/privát kulcspárokkal írják alá, az aláírás azt is igazolja, hogy csak a magánkulcs birtokában lévő fél írta alá.

A projektemben a *JWT* technológiát backend részen a felhasználó autorizációjára alkalmaztam, regisztrációnál és bejelentkezésnél.



2. ábra - JWT működésének diagramja (Forrás: <https://jwt.io/introduction>)

2.3.3 WebSocket

A *WebSocket* [12] kétirányú, teljesen duplex protokoll, azaz a kliens és a szerver közötti kétirányú csatornán képes egyidejű adatátvitelre és vételre. A *HTTP*-vel ellentétben a *ws://* vagy *wss://*-ről indul. Ez egy állapotalapú protokoll, ami azt jelenti, hogy a kliens és a szerver közötti kapcsolat mindaddig életben marad, amíg bármelyik meg nem szakítja. Miután a kapcsolatot az egyik résztvevő lezárja, a kapcsolat mindkét végről megszakad.

A *Simple (vagy Streaming) Text Orientated Messaging Protocol (STOMP)* [13] egy interoperábilis vezetékes formátumot biztosít, így a *STOMP* kliensek bármely *STOMP* üzenetközvetítővel (message broker) kommunikálhatnak, hogy egyszerű és széles körű üzenetküldési együttműködést biztosítsanak számos nyelv, platform és bróker

között. A protokoll nagy részében hasonlít a *HTTP*-hez, ugyanúgy *Transmission Control Protocol (TCP)* felett helyezkedik el.

A *SockJS* [14] egy JavaScript-könyvtár, amely *WebSocket*-szerű objektumot biztosít. A *SockJS* koherens, böngészőkön átívelő JavaScript API-t szolgáltat a fejlesztőknek, amely alacsony késleltetésű, teljesen duplex, tartományok közötti kommunikációs csatornát hoz létre a böngésző és a webszerver között.

Az alkalmazásomban a *WebSocket* technológiát, *STOMP* és *SockJS* protokollal kiegészítve arra használtam, hogy létre tudjam hozni a chat funkciót, aminél az azonnali üzenetküldés elérése érdekében szükséges volt teljesen duplex protokoll használata.

3 Feladatspecifikáció

Ebben a fejezetben részletesen bemutatom a feladatom specifikációját.

Az alkalmazásom fő célja egy könyves közösségi oldalt létrehozása, amelyen felhasználók létre tudják hozni a saját profiljukat, tudnak böngészni a könyv adatbázisban, valamint azonnali üzenetküldéssel beszélgetést tudnak kezdeményezni egymással. Fontos szerepet kapott a munkám során a bővíthetőség is, ezért már a tervezés során fel akartam erre készíteni az alkalmazást.

3.1 Feladat részletes leírása

Az egyik alapvető funkció, hogy a weboldalt böngésző látogató tudjon regisztrálni a portálra. Ehhez meg kell adnia egy egyedi felhasználónevet, az e-mail címét és egy jelszót. Ezt követően létrejön a felhasználó profilja, amibe be tud jelentkezni.

A következő fontos funkció, a böngészés a weboldal könyv adatbázisában, ezt a lehetőséget regisztráció és bejelentkezés nélkül is lehet használni. A kereső funkcióval lehetséges megadott címre vagy íróra szűrni a könyveket. A keresővel nem csak a könyvek között lehet keresni, hanem az adatbázisban szereplő felhasználók felhasználó nevére is lehet szűrni, az eredmények közül választva pedig el lehet navigálni adott felhasználó profiljára. Hasonlóan a felhasználókhoz, írókra is rá lehet keresni, majd elnavigálni a részletes adatlapjukra.

Egy további lényeges funkció, hogy a regisztrált felhasználóknak automatikusan létrejön két könyvlistája, az elolvasandó könyvek és a már olvasott könyvek. Ezekhez a listákhoz a felhasználó az egyes könyvek részletes adatlapjára ellátogatva hozzáadhat könyveket, vagy megváltoztathatja, hogy a már gyűjteményében szereplő könyv melyik listához tartozzon, vagy eltávolíthatja teljesen a gyűjteményéből.

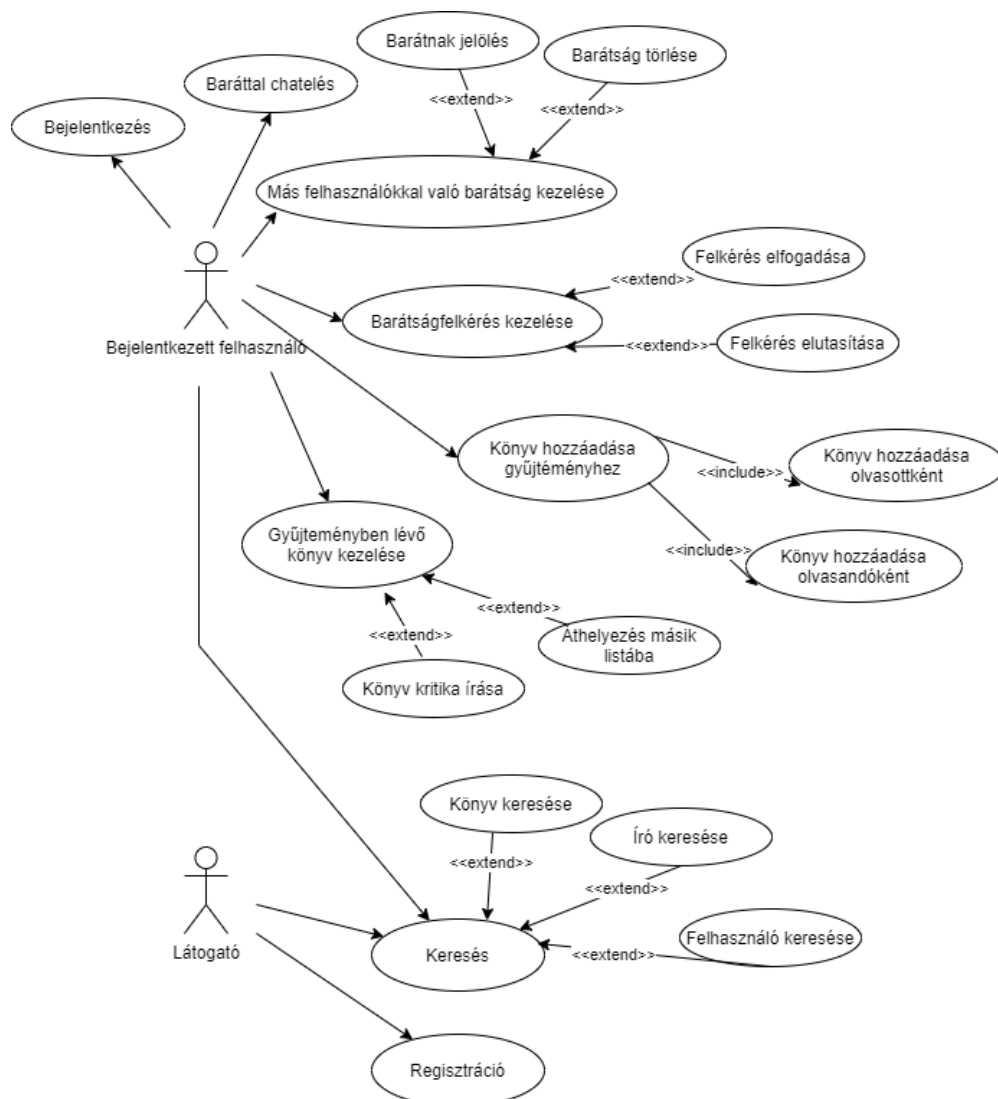
Szintén egy jelentős funkció, amiről a programomat végül elneveztem (reviewero), ami a könyv kritika/könyvismertető írása. Amennyiben egy felhasználónak szerepel a gyűjteményében egy adott könyv, tud róla a könyv adatlapján kritikát írni, valamint 1-től 10-ig terjedő skálán értékelni. Ezek a bejegyzések a weboldal bejelentkezetlen látogatóinak számára is láthatóak.

A közösségi oldalak alapvető szolgáltatása, a barátok hozzáadása. A felhasználók tudnak más felhasználóknak barátság felkérést küldeni, amivel felkerülnek egymás barátlistájára. A felhasználónak lehetősége van visszautasítani is egy ilyen kérést. Egyes felhasználók barátlistája csak annak a felhasználónak látható.

A felhasználók közötti barátság kötéssel elérhető az a funkció, hogy a barátok egymás közt privát chatben tudjanak üzenetet váltani. A chatben a résztvevők láthatják a régebbi, egy másik alkalommal küldött üzeneteiket is, valamint minden üzeneteknek annak a pontos időpontját, amikor a feladó elküldte.

3.2 Használati eset diagram

Ebben az alfejezetben grafikusán szemléltetem az alkalmazásom használati eset diagramját két forgatókönyv, a weboldal látogatója és a bejelentkezett felhasználó szerint.



3. ábra - Használati eset diagram

4 A rendszer architektúrája

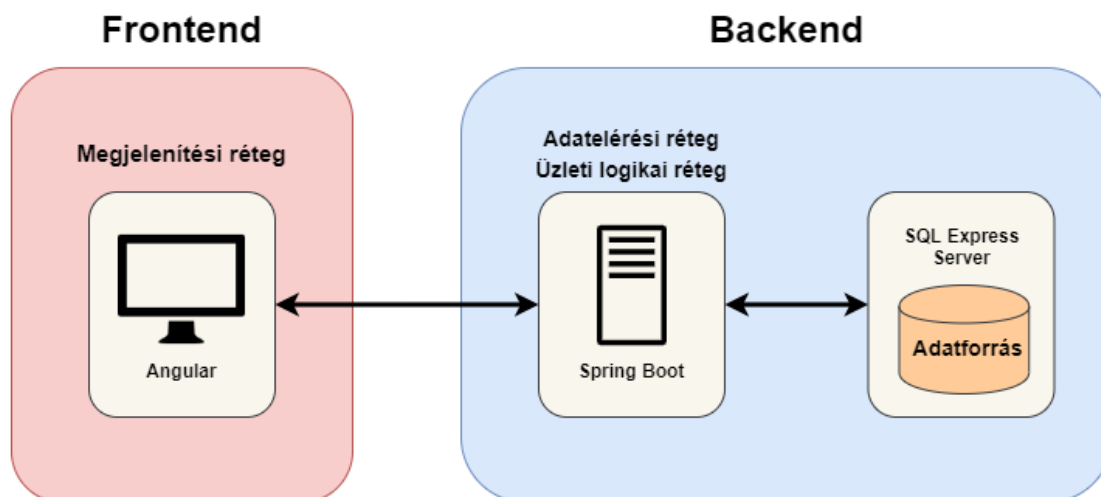
A következő fejezetben ismertetem az alkalmazásom architektúráját, valamint az adatbázis felépítését.

4.1 Az alkalmazás felépítése

A webalkalmazásom a háromrétegű kliens-szerver architektúrát követi, amelyben a megjelenítés, az üzleti logika, valamint az adatelérés különálló folyamatokként kerülnek megvalósításra. A kliens felelős a megjelenítési rétegért, azaz a felhasználói felület megjelenítéséért, ez a frontend.

A *Spring Boot* alapon készült szerver valósítja meg az üzleti logikai és az adatelérési réteget. Az üzleti logikai rétegben kerülnek megvalósításra azok a funkciók, amelyek az alkalmazás lényegét képezik, azaz a keresés, listázás, autentikáció, regisztráció stb. Az adatelérési biztosítja, hogy az adatforrás kényelmesen elérhető legyen, olyan adatkezelési műveletekkel, mint egy rekord hozzáadása, törlése, módosítása. Az adatok leképezése *Hibernate Object Relational Mapping (ORM)* eszközzel történik *entitásokká*, amik egy-egy rekordot tartalmaznak. Az entitásokat ezután *JpaRepository*-kba rendezi. Az adatok tárolásáért egy Microsoft SQL Express Server felelős, ami a lokális hálózaton fut. Az üzleti logikai réteg, adatelérési réteg és az adatbázis együtt alkotja az alkalmazás backend részét.

A frontend és a backend nagyrészt *HTTP* kérések használatával kommunikálnak. Ez alól kivétel a *WebSocket* használata. A kliens beérkező *HTTP* kéréseit a szerver *REST API* végpontjai dolgozzák fel és az üzleti logikai és az adatelérési réteg szolgálja ki. A *WebSocket* kéréseket *STOMP* végpontok dolgozzák fel, a kiszolgálása pedig a *HTTP*-hez hasonlóan történik.



4. ábra - Az alkalmazásom architektúrája

4.2 Adatbázis séma

Az adatbázisom tervezése közben törekedtem annak elérésére, hogy később könnyen bővíthető legyen és a sémája megfeleljen a harmadik normálforma normalizációs elveinek.

A *users* tábla tárolja a felhasználókat és azok regisztrációkor megadott adatait. A jelszó hasheléssel van titkosítva, ez biztosítja a felhasználók adatainak biztonságát.

Az másik fontos tábla a *books*, amely tárolja a könyveket, a leírásukat, az oldalszámukat, kiadásuk idejét és a kiadót, amelyet idegenkulcs azonosít a *publishers* táblából.

A felhasználókat a könyvekkel a *user_books* kapcsolótábla köti össze, amiben akkor születik kapcsolat köztük, ha a felhasználó hozzáadja egy listájához a könyvet, ekkor az adatbázisba bekerül a hozzáadás időpontja, valamint, hogy melyik listához adta hozzá. A listát a *state_id* idegenkulcs azonosítja a *book_states* referencia táblából.

Egy könyv több műfajhoz is tartozhat, ezt a kapcsolatot valósítja meg a *book_genres* tábla, amely a könyveket a *genres* referencia táblából *genre_id*-val köti össze.

Az írókat az *authors* tábla tárolja, a teljes nevük és életútjukkal együtt. Bővítési szempontból került még a *user_id* idegenkulcs a táblába, hogy amennyiben egy író rendelkezne saját felhasználóval összeköthető legyen a két profil. Az írókat a könyvekkel

az *author_books* tábla köti össze, mivel egy könyvnek több írója is lehet és természetesen egy író több könyvet is írhat.

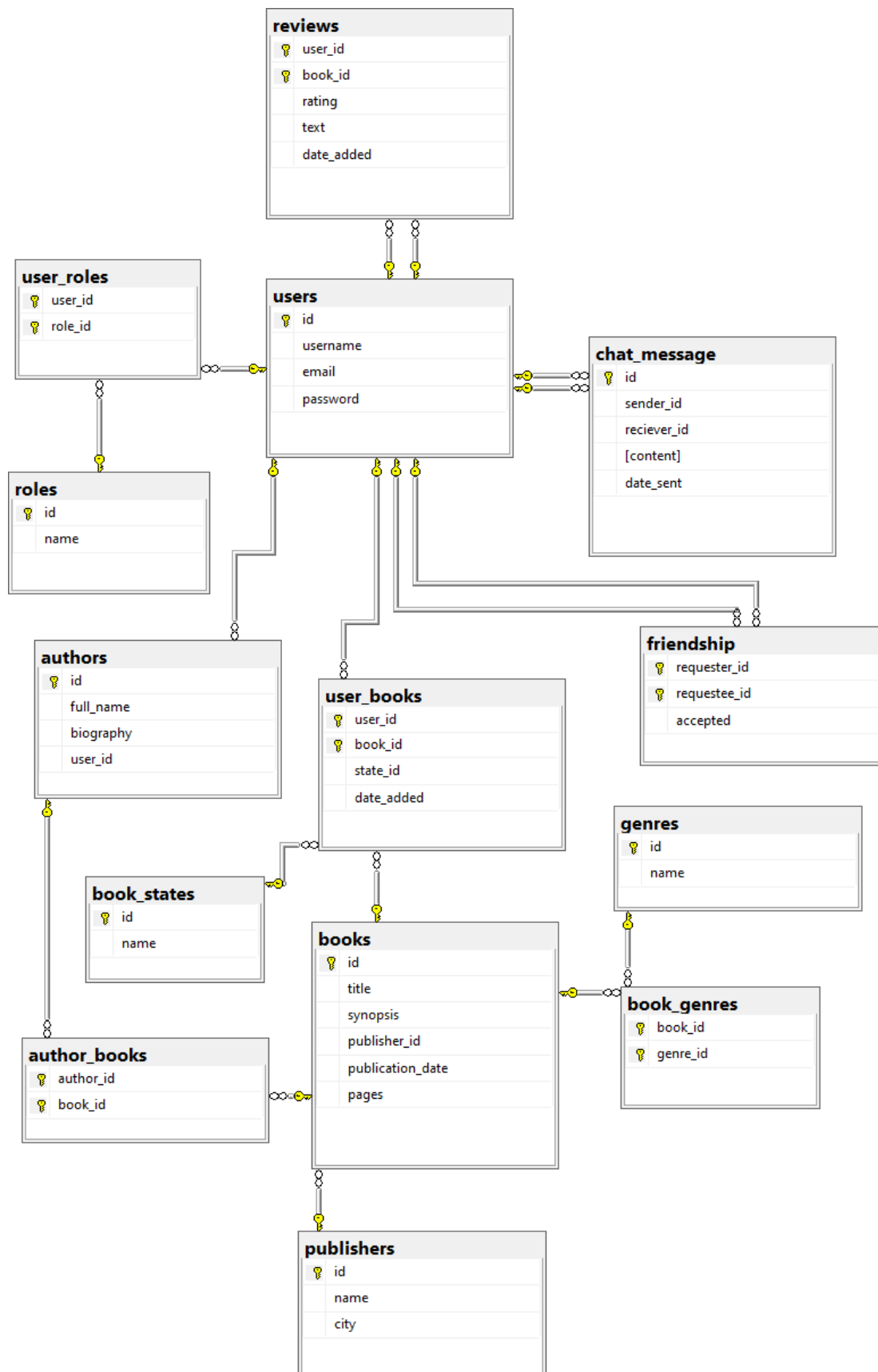
A *friendship* tábla tartalmazza a még függőben lévő és a már létrejött barátságokat. A felhasználókat két idegenkulcs mezővel, a *requester_id*-vel (kezdemenyező) és a *requestee_id*-vel (kezdemenyezett) köti össze. A tábla tárolja, hogy a felkérést a kezdemenyezett felhasználó elfogadta-e egy *accepted* nevű boolean típusú mezőben. Amennyiben még csak a felkérés történt meg, az *accepted* mező *false* értéket vesz fel, ha ezután a felhasználó visszautasítja a barátságfelkérést akkor az adott rekord törlődik az adatbázisból. Amennyiben a felhasználó, akihez a felkérés érkezett elfogadja a felkérést az *accepted* mező értéke frissül *true* értékre.

A *chat_message* tábla tárolja a chat üzeneteket. a *friendship* táblához hasonlóan két felhasználót köt össze a *sender_id* és *reciever_id* mezővel, vagyis az üzenet feladóját és vevőjét. Egy rekord tárolja továbbá az üzenet tartalmát, valamint a feladásának pontos időpontját.

A könyv kritikákat a *reviews* tábla tartalmazza. Tárolja a felhasználót, aki írta az adott kritikát, a könyvet, amiről készült a bejegyzés, egy 1-től 10-ig terjedő skálán történő értékelést, a kritika szövegét, valamint annak dátumát amikor publikálta az írója. A *user_id* és a *book_id* mezők idegenkulcsként kötik össze a táblát az adott felhasználóval és könyvvel.

A *roles* egy referencia tábla, amely a felhasználóknak adható szerepköröket tárolja, így könnyen bővíthető ezek listája. Mivel egy felhasználónak több szerepköre is lehet, ezért a felhasználókat a szerepükkel a *user_roles* kapcsolótábla köti össze.

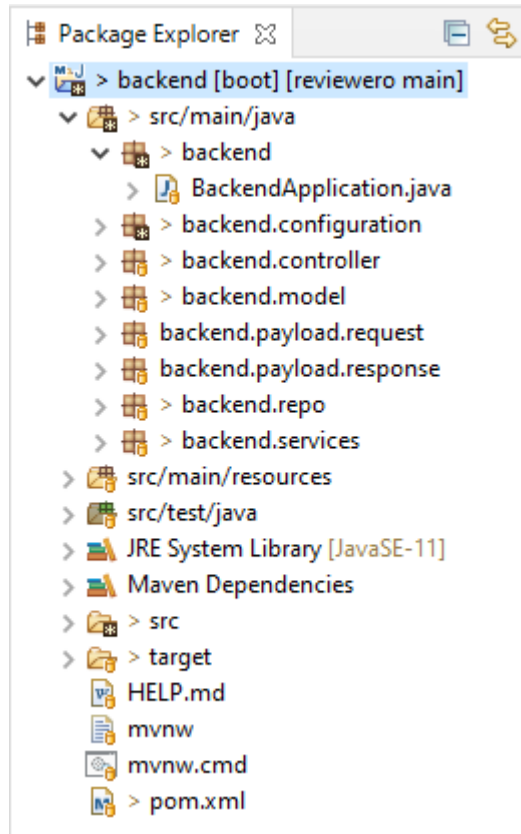
Az elkészült adatbázis sémáját a 5. ábra szemlélteti. 2.



5. ábra - Az adatbázisom sémája

4.3 Backend architektúra

A szerveroldali *Spring Boot* projekt felépítése során ügyeltem arra, hogy az átláthatóság és jól strukturáltság érdekében az egyes osztályok és interfészek a funkciójuknak megfelelően csoportosítva helyezkedjenek el.

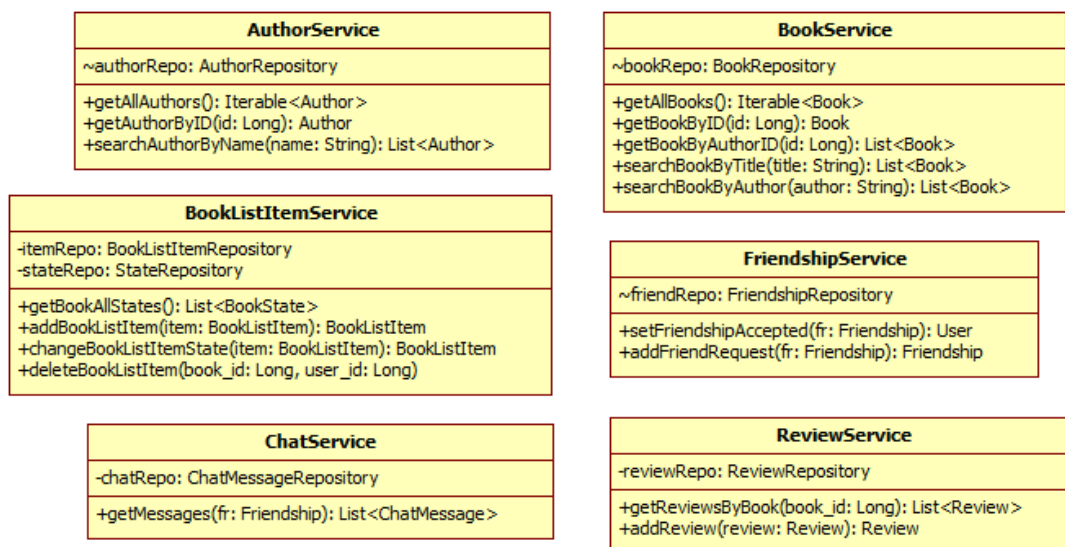


6. ábra - A projekt Eclipse Workspace felépítése

A *backend* nevű mappa a projekt *root package*-e, ezen belül közvetlenül a gyökérben található az alkalmazás *Main()* függvényét tartalmazó *BackendApplication* osztály.

Az adatelérési réteg megvalósításához tartozó elemeket a *Model package*-ben található *entitás osztályok*, beágyazható összetett kulcsként funkcionáló osztályok, valamint a *Repo package*-ben lévő *JpaRepository*-ből leszármaztatott *repository interfészek* alkotják.

Az üzleti logikai réteget megvalósító szolgáltatás osztályok a *Services package*-ben helyezkednek el. A szolgáltatás osztályok az egyetlen kapcsolódási pontja az adatelérési és az üzleti logikai rétegnek, a *Repository* interfészekon keresztül éri el.



7. ábra - A service osztályok

A *REST API* végpontokat a *Controller* osztályok dolgozzák fel, majd hívják meg a megfelelő *Service* osztályt, hogy feldolgozza a kérést.

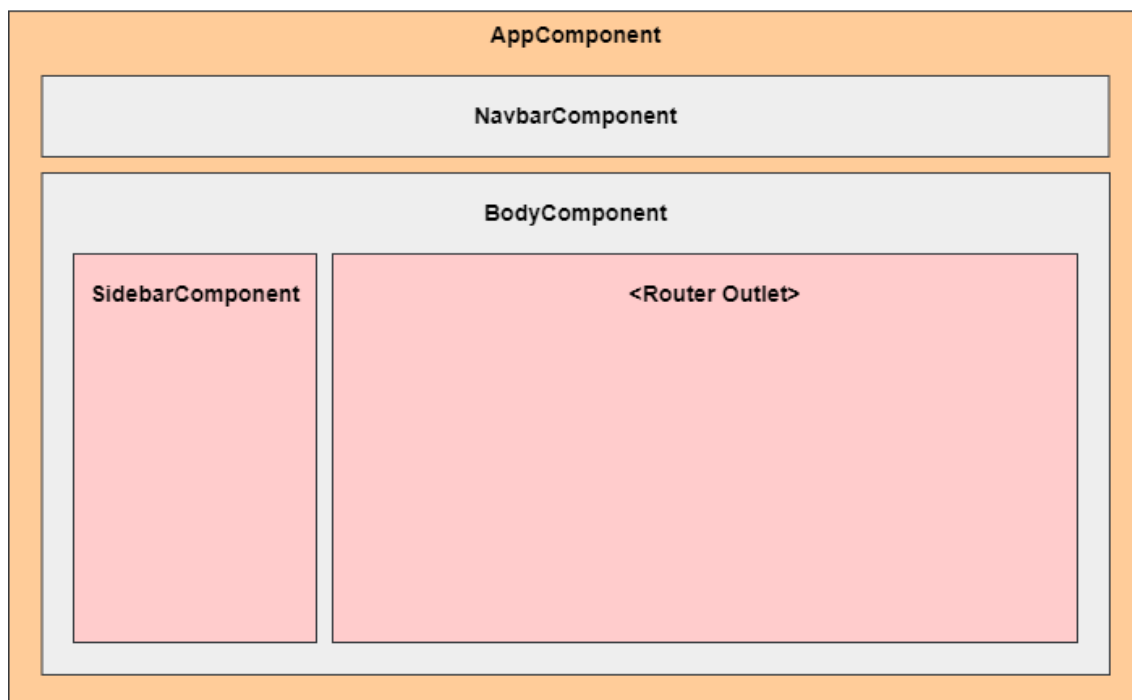
A felhasználói azonosításhoz szükséges osztályok a *Configuration* és *Payload* *package*-ekben helyezkednek el, céljuk, hogy a *Spring Security* és *JWT* technológiák segítségével konfigurálják az alkalmazás biztonsági folyamatait.

4.4 Frontend architektúra

A kliensoldali projekt tervezésénél követtem az Angular saját felépítési elveit, amelyeket a 2.2 fejezetben a felhasznált technológiák között már részletesen bemutatattam.

A *components* mappában a komponensek, a *model* mappában az entitás osztályoknak megfelelő osztályok, a *services* mappában pedig a szolgáltatások találhatóak. Az alkalmazásom egyetlen gyökeri modullal rendelkezik, ezért minden komponens ehhez tartozik.

A felhasználói felület azon felépítését, hogy a komponensek hogyan ágyazódnak egymásba a 8. ábra mutatja be. A Router Outlet dinamikusan változó egység, amely a weboldalon való navigálás hatására, az útvonaltól függően jelenít meg egy adott komponenst.



8. ábra - Felhasználói felület felépítése

5 Megvalósítás

A következő fejezetben részletesen bemutatom az alkalmazás megvalósítását, kiegészítve kódrészletekkel, valamint képernyőképekkel.

5.1 Backend

Ebben az alfejezetben kerülnek bemutatásra a szerver oldalon végzett fejlesztések.

5.1.1 Adatelérési réteg

A backend oldalon a *Spring Boot* projektben kerül megvalósításra az adatelérési réteg.

Az adatbázis és szerver összekapcsolásának elérése érdekében meg kell adni a projekt *resources* mappában található *application.properties* fájlban az adatbázis eléréséhez, valamint az autentikációhoz szükséges paramétereket,

Az *entitás osztályok* általánosan fogalmazva leképeznek az adatbázisból egy táblából egy darab rekordot. Ezek az *@Entity* annotációt kapják és az attribútumaik megegyeznek az adott adatbázis tábla attribútumaival. A megfelelő annotációk lényegesek a *Hibernate*-nek, ugyanis ez jelzi számára, hogy mit szükséges generálnia. Az *@Entity* annotációval együtt akkor érdemes használni a *@Table* annotációt, amennyiben az osztály és a tábla neve eltér egymástól. Amennyiben a táblában szerepel idegen kulcs, azt a megfelelő egyed-kapcsolat annotációval kell felvenni az entitás osztályban, ezek a *@OneToMany*, *@OneToOne*, *@ManyToOne* és *@ManyToMany* lehetnek.

Az alábbi egy példa az több az egyhez kapcsolatra a *BookListItem* osztályban:

```
@ManyToOne (cascade = {CascadeType.ALL})
@JoinColumn(name="state_id")
private BookState state;
```

A több a többhöz kapcsolat esetén a *Hibernate* megfelelő annotáció megadásával legenerálja a szükséges kapcsolótáblát, anélkül, hogy annak külön entitás osztályt kelljen létrehozni. Ebben az esetben az annotációban meg kell adni az adatbázisban szereplő kapcsolótábla nevét és kulcsait, ezt szemlélteti ez példa a *Users* osztályból:

```

@ManyToMany(fetch = FetchType.LAZY)
@JoinTable( name = "user_roles",
            joinColumns = @JoinColumn(name = "user_id"),
            inverseJoinColumns = @JoinColumn(name = "role_id"))
private Set<Role> roles = new HashSet<>();

```

Abban az esetben, ha egy táblában összetett kulcs szerepel szükséges felvenni az entitás osztály mellé egy beágyazható *@Embeddable* annotációval ellátott osztályt, ami az entitás osztályba *@EmbeddedId* annotációval beágyazva fogja jelképezni a tábla *Identity* azonosítóját.

A *Review* és *ReviewId* osztállyal bemutatva az összetett kulccsal rendelkező tábla leképezése:

```

@Entity
@Table(name = "reviews")
public class Review {

    @EmbeddedId
    ReviewId reviewId= new ReviewId();

    @ManyToOne
    @MapsId("userId")
    private User user;

    @ManyToOne
    @MapsId("bookId")
    private Book book;

    ...
}

```

ReviewId osztály:

```

@Embeddable
public class ReviewId implements Serializable{
    private static final long serialVersionUID = 100;
    private Long userId;
    private Long bookId;

    ...
}

```

A beágyazható osztály esetében szükséges, hogy implementálja a *Serializable* osztályt és felülírja a *hashCode()* és az *equals(obj: Object)* függvényét.

Minden entitás osztályhoz tartozik egy neki megfelelő *@Repository* annotációval rendelkező interfész is, aminek feladata az adatbázis műveletek végrehajtása. Ezek a *JpaRepository* interfészből származnak le, amelynek a *Spring Data JPA* megvalósításának előnye, hogy az adatelérési réteget nagyban leegyszerűsíti. Már a *JpaRepository* implementálása miatt is rendelkezik alapvető *CRUD* (*Create, Read,*

Update, *Delete*) műveletekkel. Lehetőség van ezen felül létrehozni olyan funkciókat, amelyeket nevük alapján a Spring automatikusan létrehoz, így nincs szükség implementálásra, csak a függvényt kell definiálni az interfészben. Ennek a használatára ad példát következő kódrészlet a *BookRepository* interfészből:

```
@Repository
public interface BookRepository extends JpaRepository<Book, Long> {
    Optional<Book> findByTitle(String title);
    List<Book> findByTitleContains(String title);
    List<Book> findByAuthors_Id(Long id);
    List<Book> findByAuthors_FullNameContains(String fullName);
}
```

A *findByTitle(String title)* függvény megkeresi a paraméterként átadott könyvcím szerint az adatbázisban a könyvet, amennyiben szerepel visszatér vele, ha nem akkor NULL értékkel tér vissza. A *findByAuthors_FullNameContains(String fullName)* függvény megkeresi azokat a könyveket amelyeknek az írójának a nevében szerepel a paraméterként átadott karaktersor, majd visszatér ezeknek a listájával. Ha nem talál egyet se, akkor üres listával tér vissza.

5.1.2 Üzleti logikai réteg

Az üzleti logikai réteget a Service osztályok valósítják meg, ezek *@Service* annotációt kapnak. Ennek az annotációnak a megjelölése a Spring Boot szempontjából fontos, ugyanis *@Service* a *@Component* annotáció egy speciális esete, és az alkalmazás így osztály-útvonal ellenőrzésnél automatikusan felismeri a komponenseket, amelyeket neki kell kezelni.

A *Service* osztályok metódusait a *Controller* osztályok hívják meg, majd azok az *@Autowired* annotációval ellátott *Repository*-k használatával tudják elérni az adatbázist. Minden *Controller* osztályhoz tartozik *Service* osztály, de ez nem egyenlő a *Repository*-k számával, ugyanis van, hogy egy *Service* osztály több *Repository*-t is kezel.

A következő kódrészlet mutatja be egy metódusát a *FriendshipService* osztálynak:

```

@Autowired
FriendshipRepository friendRepo;

public User setFriendshipAccepted(Friendship fr) {
    Long erId = fr.getRequester().getId();
    Long eeId = fr.getRequestee().getId();
    Friendship f=friendRepo.findByIdAndRequesteeId(erId, eeId);
    f.setAccepted(true);
    friendRepo.saveAndFlush(f);
    return f.getRequestee();
}

```

Ebben a függvényben az látható, hogy hogyan válik egy paraméterként átadott barátság elfogadottá az adatbázisban. Először a paraméterként kapott *Friendship* objektumból lekéri a kezdeményező, valamint a kezdeményezett felhasználó *id*-jét, majd az alapján a *Repository* egyedi lekérdezését meghívva lekéri az annak az összetett kulcsnak megfelelő rekordot az adatbázisból. Ennek az *accepted* attribútumát *true*-ra állítja, majd frissíti a rekordot, és visszatér a kezdeményezett felhasználóval. A visszatérési értéknek ebben az esetben az a magyarázata, hogy a kezdeményezett felhasználó volt, aki elindított ezt a folyamatot azzal, hogy elfogadta barátságfelkérést, ezért ebből a szempontból az a lényeges, hogy az ő barátság listája hogyan változott.

A következő kódrészlet azt mutatja be, hogy a Service osztály milyen módon gyűjti össze két felhasználó között váltott üzeneteket:

```

@Autowired
private ChatMessageRepository chatRepo;

public List<ChatMessage> getMessages(Friendship fr) {
    String userOneId=fr.getRequestee().getId().toString();
    String userTwoId = fr.getRequester().getId().toString();
    List<ChatMessage> messages =
chatRepo.findByReceiverAndSender(userOneId, userTwoId);
    messages.addAll(chatRepo.findByReceiverAndSender(userTwoId,
userOneId));
    return messages;
}

```

Az alkalmazásomban a *Service* osztályok sokszor ennél egyszerűbb függvényeket implementálnak, melyeknek célja a klasszikus architektúrális rétegek különválasztottságának fenntartása, az átláthatóság növelése.

Erre példa az *AuthService* osztályból:

```

@Autowired
AuthorRepository authorRepo;

public List<Author> searchAuthorByName(String name) {
    return authorRepo.findByFullNameContains(name);
}

```

Ez a függvény átadja a *Controller* osztálytól paraméterként kapott *stringet* az írókat kezelő *Repository*-nak az egyedi lekérdezéssel definiált függvényének, majd visszatér ugyanazzal az eredménnyel, mint a *Repository* *findByFullNameContains(name)* függvénye.

5.1.3 REST API

Az alkalmazásban a kliens és szerver közti kommunikáció, a chat funkción kívül *REST API*-val van megvalósítva. A szerverhez beérkező *HTTP* kérések lehetnek *GET*, *POST*, *DELETE* és *PUT* típusúak.

A *REST API*-t *controller package*-ben lévő *Controller* osztályok valósítják meg, melyek *@RestController* annotációval rendelkeznek. A *Spring Boot* ebből a megjelölésből tudja, hogy a *HTTP* kéréseket ezek a végpontok dolgozzák fel. A kapott adatokat ezután tovább küldik a *Service* osztályoknak, majd azok elvégzik a megfelelő műveleteket. Rendelkeznek még a *@CrossOrigin* annotációval, ezzel engedélyezve a tartományok közti erőforrás megosztást (*CORS* - *Cross Origin Resource Sharing*).

A *@RequestMapping()* annotációban megadható az adott végpontot azonosító *Uniform Resource Locator (URL)*, és hogy *GET*, *POST*, *DELETE*, és *PUT* közül melyik típusú kérés. Erre egy másik lehetőség, a típus szerinti annotációval megjelölés, mint a *@GetMapping*, *@PostMapping* stb.

A következő kód részlet egy *GET* típusú végpontot kezelő függvényre ad példát:

```

@GetMapping("api/review")
public List<Review> getReviewsByBook(@PathParam("book_id") Long book_id) {
    return reviewService.getReviewsByBook(book_id);
}

```

Ez a végpont visszaadja egy adott könyvhöz tartozó könyvkritikákat az adatbázisból, felhasználva a *@PathParam* annotációval ellátott paramétert. A *@PathParam* annotáció jelentése, hogy az adott paraméter az elérési útvonalon érkezik, azaz az előző példában bemutatott kódrészletnek így néz ki egy példa kérése:

```
GET <host>/api/review?book_id=1
```

A másik lehetőség *HTTP* kéréssel való adatátvitelre, az a kérés törzsének használata, így például *JavaScript Object Notation (JSON)* formátumú objektum is átadható. Ezt az átadott JSON objektumot az alkalmazás automatikusan deszerializálja és Java objektumot készít belőle. Erre mutat példát a következő függvény a *ReviewService* osztályból:

```
@PostMapping("api/review/add")
public Review addReview(@RequestBody Review review) {
    return reviewService.addReview(review);
}
```

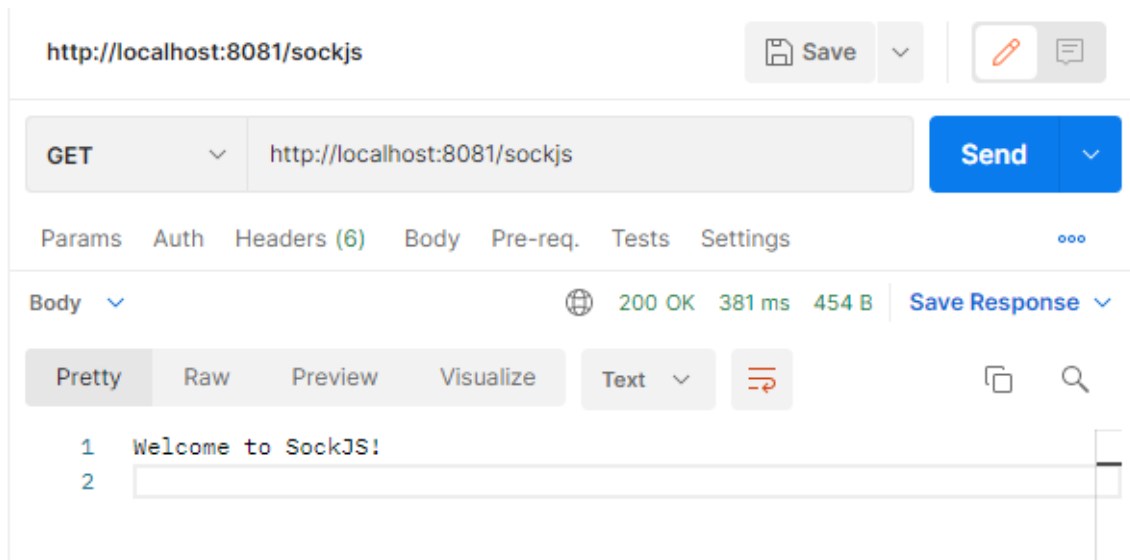
A függvény paraméterében szerepel az entitás osztály, amelynek használatával próbálja meg deszerializálni kapott objektumot. Amennyiben az entitás osztálynak nem létezik megfelelő konstruktora és/vagy az objektumban kapott attribútumokhoz Getter és Setter függvénye, akkor a kérés nem hajtodik végre, a kliens *HTTP 500 Internal Server Error* hibakódú üzenetet kap válaszként.

5.1.4 WebSocket

A *WebSocket* célja az alkalmazásban, hogy biztosítsa az chat funkcióhoz szükséges azonnali kommunikációt felhasználók között.

A *WebSocket* konfigurációja a *configuration package*-ben található, a *WebSocketConfiguration* osztály valósítja meg, amely megvalósítja a *WebSocketMessageBrokerConfigurer* interfészt és rendelkezik a *@Configuration* és az *@EnableWebSocketMessageBroker* annotációval.

Mivel a *WebSocket* maga a kommunikációs csatorna felépítéséért felelős, ezen kívül még szükséges egy protokoll, ami meghatározza csatornán áthaladó üzenetek szemantikáját. Ehhez én a *STOMP* protokollt választottam, ehhez a végpont beállítása a *registerStompEndpoints()* függvényben történik, ami az én alkalmazásom esetében a */socketjs*.



9. ábra - WebSocket kapcsolódás Postmanben

A függvény implementációja a következő:

```
@Override
public void registerStompEndpoints(StompEndpointRegistry registry) {
    registry.addEndpoint("/sockjs")
        .setAllowedOriginPatterns("*")
        .withSockJS();
}
```

A metódus először regisztrálja a végpontot, majd megadja, hogy a kérések a szerver felé milyen származási helyről érkehetnek. A *Spring Framework* biztosít *SockJS* implementációt is, ez abban az esetben lényeges, ha a felhasználó böngészője nem támogatja a *WebSockets*et, akkor létezen tartalékmegoldás (*fallback option*).

A másik függvény, amelyet implementál a konfigurációs osztály a *configureMessageBroker()*, ez a következő kódrészletben látható:

```
@Override
public void configureMessageBroker(MessageBrokerRegistry config) {
    config.setApplicationDestinationPrefixes("/app");
    config.setUserDestinationPrefix("/user");
}
```

A függvényben látható, hogy kliens oldalon a felhasználóknak a */user* prefixszel kell majd feliratkozniuk a szerverre, és a kliens az */app* prefixszel tud üzenetet küldeni a szervernek.

A *WebSocket* kérések a *@Controller* annotációval rendelkező *ChatWebSocketHandler* osztály dolgozza fel. Az üzenetek kezelésére a *Spring SimpMessagingTemplate* osztályát használja fel. A következő kódrészletben látható a chat üzeneteket feldolgozó függvény:

```

private final SimpMessagingTemplate template;

@Autowired
ChatWebSocketHandler(SimpMessagingTemplate template){
    this.template = template;
}

@MessageMapping("/chat")
public void processMessage(@Payload ChatMessage chatMessage) {
    ChatMessage saved = chatRepo.save(chatMessage);
    this.template.convertAndSendToUser(
        chatMessage.getReceiver(), "/queue/messages",
        new ChatMessage(
            saved.getId(),
            saved.getSender(),
            saved.getReceiver(),
            saved.getContent(),
            saved.getDateSent()));
}

```

Látható, hogy a *SimpMessagingTemplate* osztálynak létezik *convertAndSendToUser()* függvénye, amelynek használatával egyszerűen lehet egy specifikus felhasználónak üzenetet küldeni. A *Controller* a chat üzeneteket az *app/chat* végponton várja, az üzeneteket a megfelelő felhasználónak a */user/<felhasználó id>/queue/messages* címre továbbítja.

Opening Web Socket...	compat-client.js:32
Web Socket Opened...	compat-client.js:32
>>> CONNECT accept-version:1.0,1.1,1.2 heart-beat:10000,10000	compat-client.js:32
Received data	compat-client.js:32
<<< CONNECTED heart-beat:0,0 version:1.2 content-length:0	compat-client.js:32
connected to server undefined	compat-client.js:32
>>> SUBSCRIBE id:sub-0 destination:/user/3/queue/messages	compat-client.js:32
>>> SEND destination:/app/chat content-length:86	compat-client.js:32

10. ábra - Böngészőben a WebSocket működése

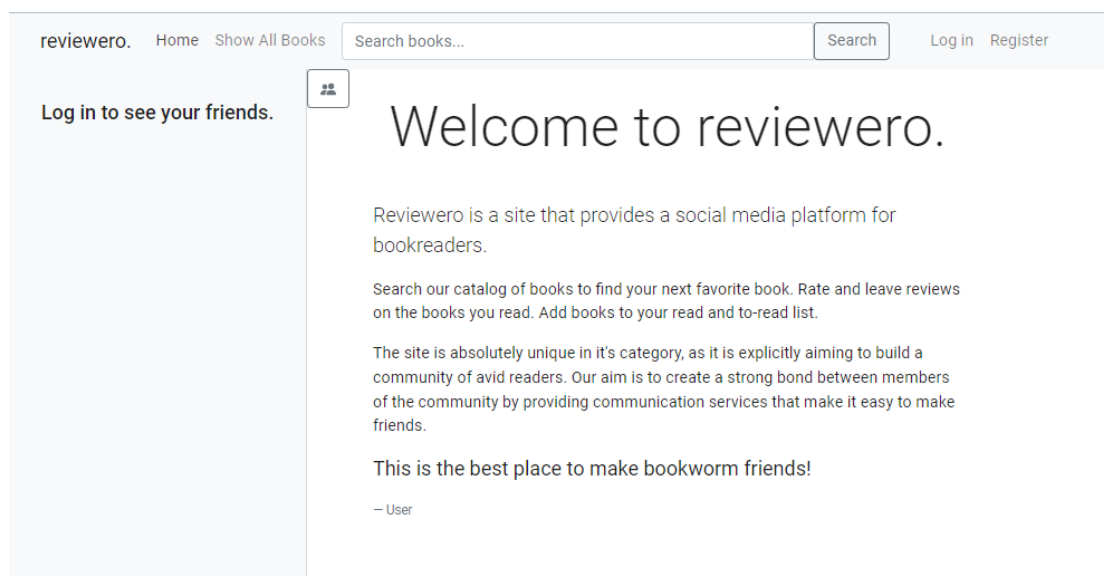
5.1.5 Spring Security

Spring Security konfigurálását igényelte, hogy a felhasználó adatai biztonságosan jussanak el, majd ugyanígy tárolva legyenek az adatbázisban. Létrehoztam a *SpringSecurityWebAppConfig* osztályt, amely a *WebSecurityConfigurerAdapter* osztályból öröklődik, és a *@Configuration* *@EnableWebSecurity* és *@EnableGlobalMethodSecurity* annotációkat kapta. Itt kerülnek konfigurálásra a *CORS* szabályok.

Az autentikáció az alkalmazásban *JWT* használatával lett megvalósítva. Az felhasználó azonosítását belépéskor az *AuthController* osztály *authenticateUser()* függvénye végzi el. A metódus elvégzi az adatok ellenőrzését, majd *JwtResponse payload*-ban visszaküldi a kliensnek, a felhasználó *id*-jét, valamint *JwtTokenUtils* osztály által generált *token*nel együtt. A *JwtTokenUtils* osztály az aláírást a HS512 algoritmussal és a titkos kulccsal végzi, amely az *application.properties* fájlban van tárolva. A *HTTP* kérések fejlécében érkező *JWT*-k validálásáért a *JwtRequestFilter* osztály felelős.

5.2 Frontend

Ebben az alfejezetben a kliens oldalon készült fő funkciók kerülnek bemutatásra. A szerverhez küldött *HTTP* és *WebSocket* kérésekhez az *URL*-ek az *environments* mappában az *environments.ts* fájlban lettek elhelyezve.



11. ábra - Az elkészült alkalmazás főoldala

5.2.1 Regisztráció

Az alkalmazással szemben állított egyik fő elvárás az volt, hogy a weboldalt böngésző látogató képes legyen regisztrálni, profilt létrehozni, ezért a frontend fejlesztést ennek a funkciónak a megvalósításával kezdtem.

A regisztráció a *Register* komponensben kerül megvalósításra, egy űrlap kitöltésével, amelyben a felhasználónak meg kell adnia a választott felhasználónevét, e-mail címét, és jelszavát. Az űrlap validációja a komponensen belül az *ngInit()* függvényben *FormGroup* használatával valósul meg a következő módon:

```
form: FormGroup;  
  
ngOnInit() {  
  this.form = this.builder.group({  
    username: ['', [Validators.required]],  
    email: ['', [Validators.required,  
      Validators.pattern("[^ @]*@[^ @]*")]],  
    password: ['', [Validators.required]],  
  });  
}
```

A validációs beállítások szerint egyik mező sem maradhat üresen, és az e-mail címnek követnie kell a reguláris kifejezéssel megadott e-mail cím mintát. Amennyiben a beírt értékek megfelelnek ennek, a *Submit* gomb kattinthatóvá válik. A *Submit* gomb megnyomása meghívja a *register()* függvényt, ami az *AuthService register()* metódusára feliratkozik. Ez a metódus visszatér a szervertől azzal, hogy regisztrálható volt-e a felhasználónév és jelszó, amennyiben igen, a regisztráció sikeres, és a *Router* átnavigál a bejelentkezési oldalra.

reviewero. Home Show All Books Search books... Search Log in Register

Log in to see your friends.

Register here.
Start reviewing today.

Username
Enter your username

Email
Enter your email address

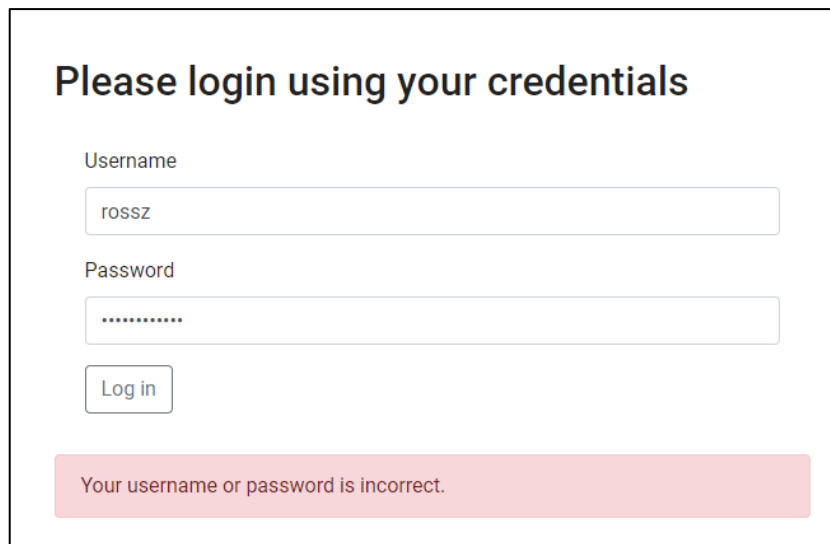
Password
Enter your password

Submit

12. ábra - Regisztrációs űrlap

5.2.2 Bejelentkezés

A bejelentkezési funkciót a *Login* komponens valósítja meg, a regisztrációhoz hasonlóan űrlap validációt használ, de itt megadandó értékek csak a felhasználónév és a jelszó. Amennyiben a mezők nem üresek a *Log in* gomb kattinthatóvá válik. A *Log in* gomb kattintásra meghívja a *login()* függvényt, amely az *AuthService login()* metódusának átadja a beírt felhasználónevet és jelszót. Ez a függvény a szerverről visszakapja, hogy sikeres volt-e a belépés, létezik-e ilyen felhasználó. A válaszüzenetben szerepel a bejelentkezett felhasználó és a hozzáféréshez szükséges access token. A bejelentkezett felhasználót az *AuthService* beállítja *currentUser Observable<User>* változó értékeként, és elmenti helyi tárhelyre, a *TokenStorageService* pedig az *access token* menti el helyi tárhelyre. Az *Observable* egy olyan objektum, ami „megfigyelhető”, ennek a változásaira feliratkoznak „megfigyelők”. Ebben az esetben erre azért van szükség, mert a felület megváltozik annak megfelelően, hogy van-e belépve felhasználó, és ha igen, ki a felhasználó.



13. ábra - Példa hibás belépésre

5.2.3 Profil

A profil megvalósítása a *PublicProfile* komponensben készült. A regisztrált felhasználók mind rendelkeznek egy publikus profillal, amely a nem bejelentkezett látogatók számára is megtekinthető. Egy felhasználó profilja a `/users/<id>` útvonalon érhető el. A profilon szerepel a felhasználó e-mail címe és az olvasandó és olvasott könyveinek az listája. Amennyiben a bejelentkezett felhasználónak nincs folyamatban lévő barátság felkérése az általa látogatott felhasználó felé, nem szerepel már a barátlistáján és nem a saját profilján áll, akkor megjelenik egy *Add Friend* gomb is. Ezzel a bejelentkezett felhasználó barátságfelkérést küldhet a másik felhasználónak.

Az alkalmazásban a könyvlisták megjelenítése a *BookList* komponenssel történik, ami egy újrafelhasználható gyerekkomponens. Bemenetként átvesz egy könyvekből álló tömböt, amelyet megjelenít és amennyiben a felhasználó rákattint a listából egy könyvre vagy íróra, az új útvonalat, amelyre a *Router*nek navigálnia kell *EventEmitter* segítségével átadja a szülőkomponensének. Ennek megvalósítása a következő kódrészletben látható:

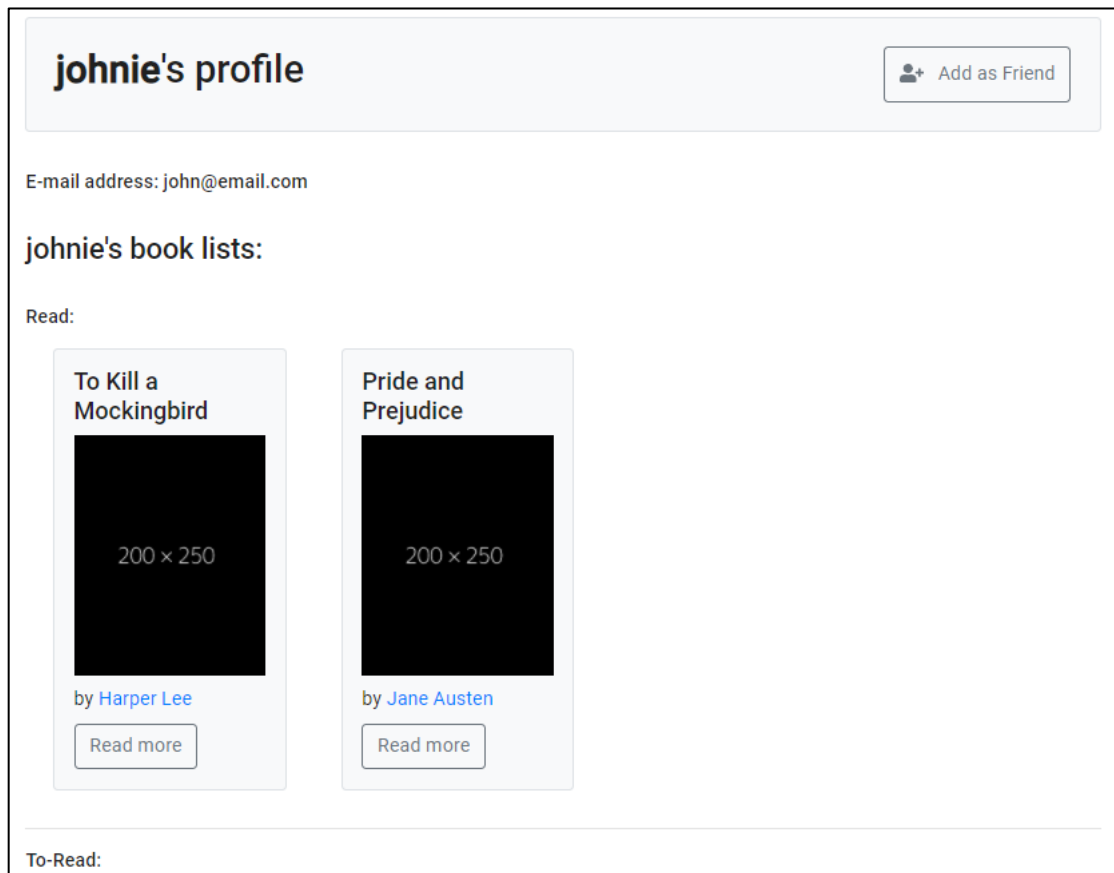
```
@Input() books: Book[];
@Output() newRoute = new EventEmitter<string>();

onBookClick(id: number) {
  this.newRoute.emit(`/books/${id}`)
}

onAuthorClick(id: number) {
  this.newRoute.emit(`/authors/${id}`)
}
```

Ebben az esetben a szülőkomponens a *PublicProfile*, a *BookList* komponenst az alábbi formában ágyazza be:

```
<app-book-list (newRoute)="changeRoute($event)"
               [books]="toReadList">
</app-book-list>
```




14. ábra - Példa publikus profilra

5.2.4 Keresés

A keresés első lépését a *Navbar* komponens valósítja meg, amennyiben ide a felhasználó beír egy kulcsszót, és rákattint a *Search* gombra, az elnavigálja a *Search* komponens oldalára, ami egy részletes keresőt jelenít meg. A navigáció közben a *Navbar* a *Search* komponensnek a beírt kulcsot *query paraméterként* adja át, amelyet az *ActivatedRoute params* attribútumából lehet lekérni. A részletes kereső oldalon lehetőség van könyvre cím vagy író szerint, íróra név szerint, felhasználóra felhasználónév szerint keresni.

Search our database by books, authors and users.


Pride and Prejudice



200 x 250

by [Jane Austen](#)


The Great Gatsby



200 x 250

by [F. Scott Fitzgerald](#)

To Kill a Mockingbird




200 x 250

by [Harper Lee](#)

15. ábra - Példa keresés könyvcímre

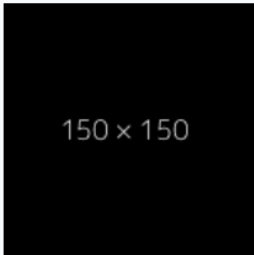
Search our database by books, authors and users.

user1



150 x 150

user2



150 x 150

16. ábra - Példa keresés felhasználónévre

A *BookList* komponenshez hasonlóan létezik *AuthorList* és *UserList* komponens, működésüknek az alapja ugyanaz. A keresett elemek lekéréséhez a megfelelő *Service* használja fel a komponens. Ezek az *Author*, a *Book* és az *Auth* szolgáltatások. A *Service* lekéri a szervertől a megfelelő paraméterre szűrt listát, amivel *Observable*-ként tér vissza. Erre a következő kódrészlet példa az *Author* szolgáltatásból:

```
public getAuthorByName(name: string): Observable<Author[]> {
    const params = new HttpParams().append('name', name);
    return this.http.get<Author[]>(this.URL + `search/`, {params})
}
```

5.2.5 Könyv kezelése gyűjteményben

A könyvek részletes adatlapjainak megjelenítéséért a *BookDetails* komponens felelős. Egy adott könyv adatlapja a */book/<id>* útvonalon érhető el. Az oldalon szerepel a könyv minden tulajdonsága, ami az adatbázisban szerepel, valamint minden könyvkritika, amit írtak hozzá felhasználók. Az oldalt nem regisztrált látogató is elérheti és elolvashatja a könyvkritikákat, de ő maga nem írhat.

Ha bejelentkezett felhasználó nyitja meg egy könyv adatlapját az, hogy milyen lehetőségeket lát még, attól függ, hogy már szerepel-e valamelyik gyűjteményében olvasottként vagy olvasandóként a könyv. Ha olvasottként szerepel, akkor lehetősége van új könyvkritikát írni, amihez 1-től 10-ig terjedő skálán pontozhatja a könyvet. Amennyiben az olvasandó listáján szerepel a felhasználónak, akkor lehetősége van megváltoztatni a könyv státuszát olvasandóról olvasottra, de kritikát nem tud írni.

Ha a könyv még egyáltalán nincs a felhasználó gyűjteményében, akkor egy *Add Book* gombot lát, és egy legördülő menüt, amiből kiválaszthatja, hogy az olvasott vagy olvasandó listájához adja hozzá.

A könyv hozzáadásához a következő metódus tartozik:

```
addBook() {  
  this.isBookAdded = true;  
  this.userHasReview = false;  
  let state = this.options.find((o) => o.id == this.stateId);  
  let item = new BookListItem(this.currentUser, this.book, state, new  
Date());  
  this.bookService.addBookListItem(item).subscribe((data) => {  
    this.usersBookItem = data;  
  });  
}
```

To Kill a Mockingbird

by [Harper Lee](#)

The unforgettable novel of a childhood in a sleepy Southern town and the crisis of conscience that rocked it. "To Kill A Mockingbird" became both an instant bestseller and a critical success when it was first published in 1960. It went on to win the Pulitzer Prize in 1961 and was later made into an Academy Award-winning film, also a classic.

324 pages

300 × 400

Book added

Change the book's state in your collection

Currently: read

to read

Change state

Reviews

+ Add new review

Rating:

I loved this!

Submit

johnie

6/10

Honestly it wasn't too bad. Not good enough to get a higher rating though.

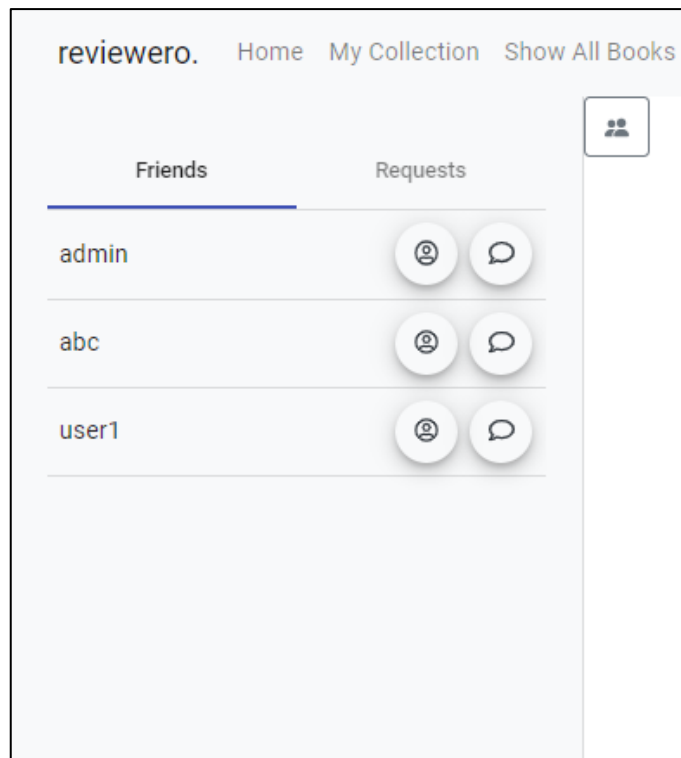
2021. 02. 22. 0:00:00

17. ábra - Egy könyv adatlapja

5.2.6 Barátság

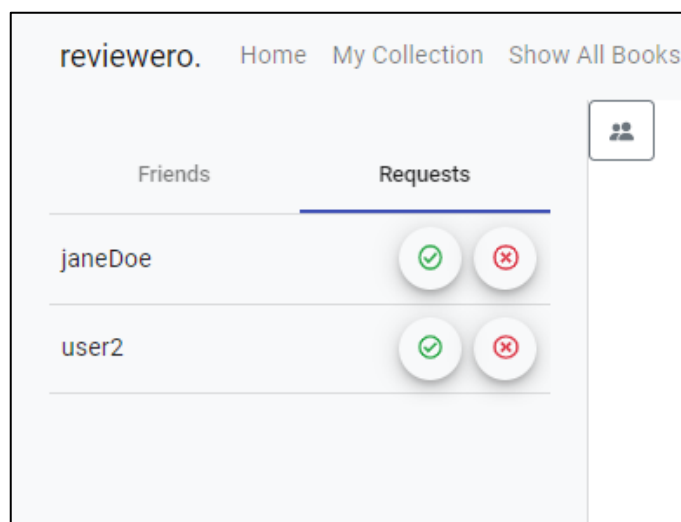
A barátságfelkérés elindítása funkció a 5.2.3 alfejezetben került bemutatásra.

A barátok és felkérések listájának kezelése és megjelenítése a *Sidebar* komponens felelőssége. A barátok és a felkérések két külön *tab*-on helyezkednek el, egyszerre csak az egyik látható. A barátok felhasználóneve mellett két gomb kap helyet, az egyik a profiljára navigál, a másik pedig egy privát chat ablakba azzal a felhasználóval.



18. ábra - Barátok listája

A barátság felkéréseknél a kezdeményező felhasználó neve mellett egy pipa és egy X gombbal lehet elfogadni vagy elutasítani a felkérést.



19. ábra – Felkérések listája

Mind a felkérés kezdeményezők, mind a barátok listáját a *User* szolgáltatás megfelelő függvényének meghívásával kapja meg a komponens. Minden felhasználónak két tömb attribútuma van, azok a barátság kapcsolatok, ahol ő a kezdeményező, és azok, ahol ő a kezdeményezett. Ezeket tömböket hivatottak összefésülni és szűrni a *User*

szolgáltatás metódusai. A következő függvénynek a visszatérési értéke az adott felhasználó felé barátság felkérést kezdeményezők listája:

```
public getFriendRequests(user: User) {  
    let friends: User[] = [];  
    for (var f of user.requestee)  
        if (!f.accepted) friends.push(f.requester)  
    return friends;  
}
```

5.2.7 Chat

A chat funkció megvalósítását a *Chat* komponens végzi a *WebSocket* szolgáltatás felhasználásával. A komponens a konstruktorában példányosítja a *WebSocket* szolgáltatást, az inicializálja a *WebSocket* kapcsolatot a szerverrel. Ezt követően a komponensben kliens feliratkozik a */user/<id>/queue/messages* útvonalon érkező üzenetekre. A kliens a szervernek üzenetet az */app/chat* útvonalon tud küldeni.

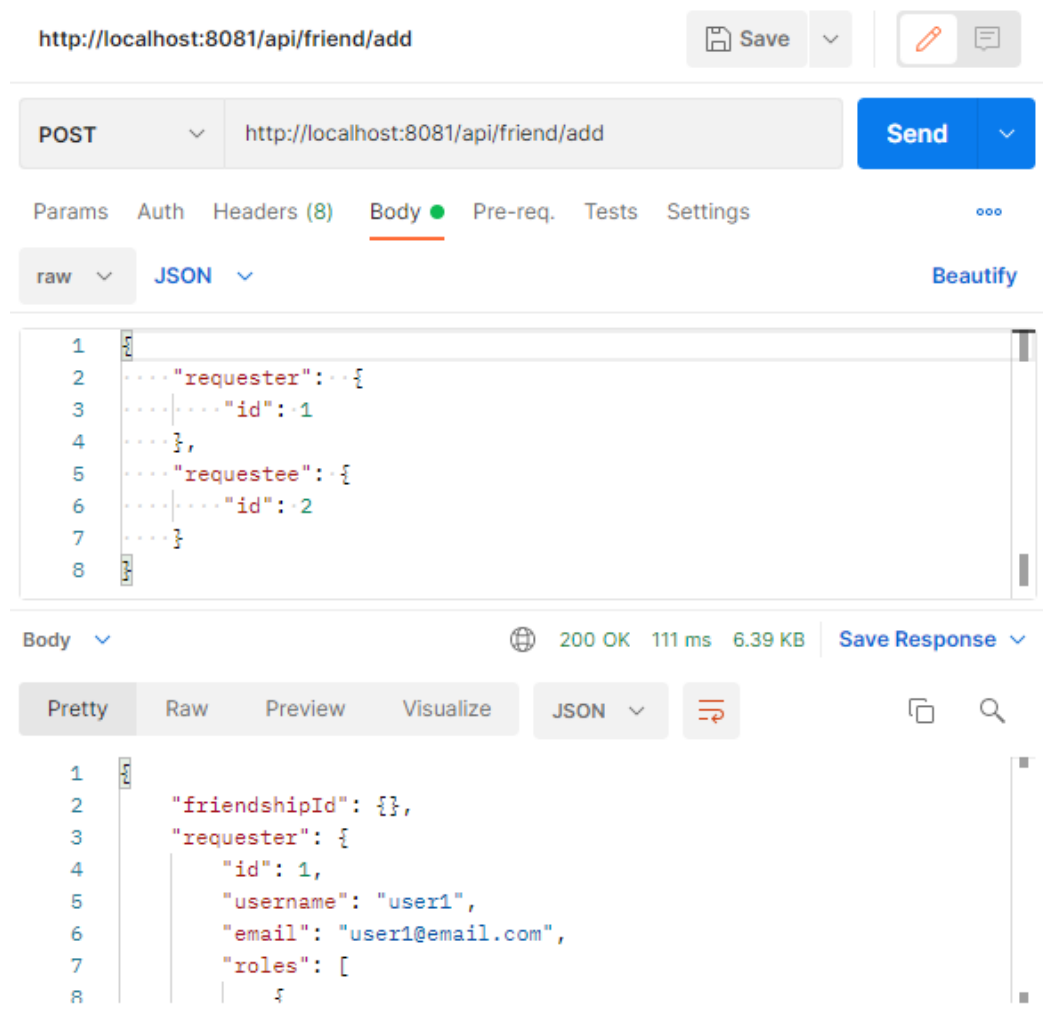


20. ábra – Chatablak

```
subscribeToIncomingMessages() {  
    const that = this;  
    this.websocketService.stompClient.connect({}, function(frame) {  
        that.websocketService.stompClient.subscribe("/user/" +  
that.loggedInUser.id + "/queue/messages", (message) => {  
            that.msgs=[JSON.parse(message.body)], ...that.msgs];  
        });  
    });  
}
```


6 Tesztelés

A fejlesztés folyamata alatt végig végeztem manuális tesztelést. Frontenden a felhasználói felületet teszteltem, valamint a böngészőben elérhető DevTools használatával. Backenden Postman alkalmazással küldtem a szervernek teszt kéréseket, amelyeknek válaszából tudtam következtetni a helyes működésre.



21. ábra - Manuális tesztelés Postmannel

A *Spring Boot* saját *JPA* integrációs tesztelésre készült `@DataJpaTest`-et alkalmaztam az adatelérési réteg tesztelésére. A továbbiakban egy tesztet mutatok be, mivel a *JPA* tesztek nagyban hasonlítanak egymáshoz.

Ennek a tesztnek a célja az, hogy tesztelje, hogy a *BookRepository*-nak a `findByTitleContains(title: string)` metódusa megfelelően működik-e. A függvény feladata az, hogy megkeresse a könyvek Repository-jában azokat könyveket, amelyeknek a címe

tartalmazza a paraméterként átvett karaktersort, majd visszatérjen a talált könyvek listájával. Előkészületként létrehoztam egy új *Book* példányt, *Title* címmel, majd az *entityManager*-nek ezt átadtam, hogy *perzisztens* legyen. Ezt követően meghívtam a *Repository findByTitleContains()* függvényét és ennek eltároltam az értékét. Az *asszertálást* arra hajtottam végre, hogy a kapott *Book* lista nulladik (azaz egyetlen) elemének a címe megegyezik-e az eredeti *Book* példány címével. A teszt sikeresen lefutott.

```
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
public class BookRepositoryTest {

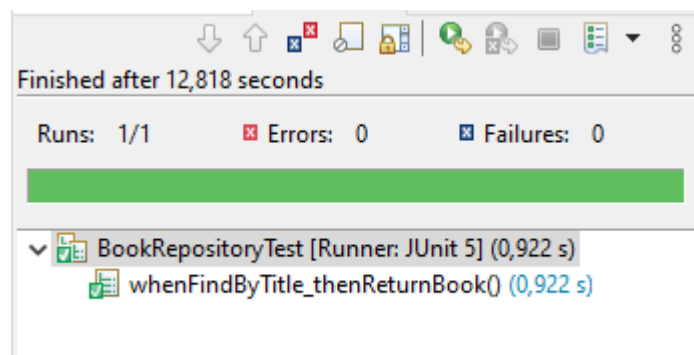
    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private BookRepository bookRepository;

    @Test
    public void whenFindByTitle_thenReturnBook() {
        // given
        Book book = new Book();
        book.setTitle("Title");
        entityManager.persist(book);
        entityManager.flush();

        // when
        ArrayList<Book> found =
        bookRepository.findByTitleContains(book.getTitle());

        // then
        assertThat(found.get(0).getTitle())
            .isEqualTo(book.getTitle());
    }
}
```



22. ábra - BookRepository teszt eredménye

7 Összefoglalás és továbbfejlesztési lehetőségek

Az elkészült alkalmazás sikeresen megvalósítja a specifikációban rögzített funkciókat. A barátságok létrehozása, könyvek listába rendezése, könyvkritikák írása, valamint a chat is mind megfelelően működik. A felhasználói felület reszponzív, helyesen igazodik a képernyőarány változásához.

Munkám alatt sok fejlesztési tapasztalattal lettem gazdagabb, bővíthettem az Angular és Spring Boot keretrendszerhez kapcsolódó ismereteimet. Emellett a WebSocket és a JWT mint technológia számomra teljesen új volt, így bár ütköztem nehézségekbe az implementációjuk közben, mégis sokat tanulhattam megismerésükből.

A továbbfejlesztési lehetőségek szempontjából, elmondhatom, hogy az alkalmazást azon elvet követve fejlesztettem, hogy könnyen bővíthető legyen, sok potenciális új funkció kaphasson még benne helyet.

A szerepkezelés bár implementálva van, nincs különbség téve egyes szerepek (felhasználó, admin, író stb.) között. Ezt a fonalat követve lehetne létrehozni admin jogokat igénylő funkciókat, mint a könyv hozzáadása és eltávolítása az adatbázisból.

A chat után közösségépítés szempontjából megfontolandó lehetne létrehozni fórumoknak felületet, amiket akár felhasználók is létrehozhatnának az általuk választott témában. Szintén a felhasználók közötti interakciót növelné, ha a könyvkritikákhoz hozzá lehetne szólni más felhasználóknak.

8 Telepítési útmutató

Az alkalmazás forráskódja elérhető az alábbi publikus GitHub repository-ban:
<https://github.com/kinmat/reviewero>

Mivel a projektet egyelőre csak lokálisan lehet futtatni, ezért a funkcionalitás tesztelésének érdekében első lépésként szükséges létrehozni egy új lokális adatbázist *reviewero* néven SQL Serveren, majd futtatni a repository gyökerében található *database_creation.sql* scriptet, amivel létrejönnek az megfelelő táblák és betöltődnek a minta adatok.

Ezután már futtatható a Spring Boot projektből létrehozott JAR fájl a backend mappa gyökerében a `java -jar target/backend-0.0.1-SNAPSHOT.jar` parancsot kiadva terminálban.

A JAR fájl helyett a Spring Boot projekt szintén futtatható Spring Boot támogatással rendelkező fejlesztői környezetben (például Eclipse).

A kliens futtatásához először szükséges kiadni egy új terminálban a frontend mappából az `npm i` parancsot, hogy a `node_modules` mappába betöltődjenek a szükséges npm modulok. Ezt követően az `ng serve` parancs kiadásával elindítható az alkalmazás frontend része.

9 Irodalomjegyzék

- [1] „Spring Framework,” [Online]. Available: <https://spring.io/projects/spring-framework>. [Hozzáférés dátuma: 28 november 2021.].
- [2] baeldung, „A Comparison Between Spring and Spring Boot,” 24. március 2021.. [Online]. Available: <https://www.baeldung.com/spring-vs-spring-boot>. [Hozzáférés dátuma: 29. november 2021.].
- [3] M. Tyson, „What is JPA? Introduction to the Java Persistence API,” 2. április 2019.. [Online]. Available: <https://www.infoworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html>. [Hozzáférés dátuma: 4. december 2021.].
- [4] „Hibernate ORM,” [Online]. Available: <https://hibernate.org/orm/>. [Hozzáférés dátuma: 3. december 2021.].
- [5] „SQL Server Express,” [Online]. Available: https://en.wikipedia.org/wiki/SQL_Server_Express. [Hozzáférés dátuma: 3. december 2021.].
- [6] „Introduction to Angular concepts,” [Online]. Available: <https://angular.io/guide/architecture>. [Hozzáférés dátuma: 29. november 2021.].
- [7] „Mi az a Bootstrap 4? Hogyan érdemes használnunk? Mit érdemes tudni róla?,” 9. augusztus 2021.. [Online]. Available: <https://gremmedia.hu/mi-az-a-bootstrap-4-hogyan-hasznaljuk>. [Hozzáférés dátuma: 29. november 2021.].
- [8] „Angular (web framework),” [Online]. Available: [https://en.wikipedia.org/wiki/Angular_\(web_framework\)](https://en.wikipedia.org/wiki/Angular_(web_framework)). [Hozzáférés dátuma: 29. november 2021.].
- [9] „Material design,” [Online]. Available: https://hu.wikipedia.org/wiki/Material_design. [Hozzáférés dátuma: 29. november 2021.].

- [10] „HTTP,” [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP>. [Hozzáférés dátuma: 3. december 2021.].
- [11] „Introduction to JSON Web Tokens,” [Online]. Available: <https://jwt.io/introduction>. [Hozzáférés dátuma: 3. december 2021.].
- [12] I. a. A. M. Fette, „RFC 6455, The WebSocket Protocol,” december 2011. [Online]. Available: <https://www.rfc-editor.org/info/rfc6455>.
- [13] „STOMP,” [Online]. Available: <https://stomp.github.io>. [Hozzáférés dátuma: 4. december 2021].
- [14] „sockjs/sockjs-client: WebSocket emulation - Javascript client,” [Online]. Available: <https://github.com/sockjs/sockjs-client>. [Hozzáférés dátuma: 3. december 2021.].
- [15] „What is web socket and how it is different from the HTTP?,” [Online]. Available: <https://www.geeksforgeeks.org/what-is-web-socket-and-how-it-is-different-from-the-http/>. [Hozzáférés dátuma: 3. december 2021.].