Pannon Egyetem

Műszaki Informatikai Kar

Matematikai Tanszék

Programtervező Informatikus BSc

**SZAK-/DIPLOMADOLGOZAT**

**Közösségi szolgáltatások biztosítása webes felületen**

**Staub József Dávid**

Témavezető: Lipovits Ágnes

2022



Hallgatói nyilatkozat

Alulírott Staub József Dávid hallgató kijelentem, hogy a dolgozatot a Pannon Egyetem Matematikai tanszékén készítettem a Programtervező Informatikus BSc végzettség megszerzése érdekében.

Kijelentem, hogy a dolgozatban lévő érdemi rész saját munkám eredménye, az érdemi részen kívül csak a hivatkozott forrásokat (szakirodalom, eszközök stb.) használtam fel.

Tudomásul veszem, hogy a dolgozatban foglalt eredményeket a Pannon Egyetem, valamint a feladatot kiíró szervezeti egység saját céljaira szabadon felhasználhatja.

Dátum: Veszprém, [év hónap nap]

*Staub József Dávid*

Témavezetői nyilatkozat

Alulírott Lipovits Ágnes témavezető kijelentem, hogy a dolgozatot Staub József Dávida Pannon Egyetem Matematikai tanszékén készítette Programtervező Informatikus BSc végzettség megszerzése érdekében.

Kijelentem, hogy a dolgozat védésre bocsátását engedélyezem.

Dátum: Veszprém, [év hónap nap]

*Lipovits Ágnes*

Köszönetnyilvánítás

Köszönöm témavezetőmnek, Lipovits Ágnesnek a türelmét, a rám áldozott idejét és a szakmai tanácsait, amelyek segítették a fejlesztést.

Tartalmi összefoglaló

A feladat egy közösségi weboldal létrehozása volt webes felületen. A weboldal célközönsége a japán képregények (Mangák) iránt érdeklődők, akik szeretnének más emberekkel beszélgetni és/vagy mangákat olvasni.

A weboldalon lehet mangákat keresni, olvasni, értékelni, saját listába gyűjteni őket, fórumokat létrehozni, fórumokat olvasni, fórumokhoz hozzászólni, barátokat felvenni, barátokkal valós idejű csevegést folytatni. Ezek a weboldal fő funkciói.

A projekt egy backend szerverből, API proxyból amiket Node.Js segítségével írtam, a felhasználói felületet pedig React-al csináltam. Adattárolásra MariaDB-t használok. És egy külső API-t (MangaDex API) is használok, amit a mangák és azokhoz létező adatok megszerzésére használok.

**Kulcsszavak:** Webfejlesztés, React, Node.Js, Redux, MariaDB, Manga

Abstract

The task was to create a social website on the web, but not just a general social site, but with a certain group in mind, and that group is those who are interested in Japanese comics (Manga).

On the website the user can search for mangas, read them, rate them, collect them in a list, create forum threads, read them, comment on them, add friends and speak with them in a real-time chat. These are the main functions of the site.

The project is made out of a backend server, API proxy that's written with the help of Node.Js, the user interface is made with React. For the data that needs to be stored I store it in a MariaDB database. And I use a third-party API (MangaDex API) to get the informations of the mangas, be it the manga details or the chapters that has been translated.

**Keywords:** Web development, React, Node.Js, Redux, MariaDB, Manga

Tartalomjegyzék

[Jelölésjegyzék 10](#_Toc102298072)

[1. Bevezetés 11](#_Toc102298073)

[2. Versenytárs elemzés 12](#_Toc102298074)

[3. Felhasznált technológiák 13](#_Toc102298075)

[3.1. MangaDex API (Külső API) 13](#_Toc102298076)

[3.2. Backend 13](#_Toc102298077)

[3.2.1. Node.JS 13](#_Toc102298078)

[3.2.2. Express 14](#_Toc102298079)

[3.2.3. MySQL (Node.js csomag) 14](#_Toc102298080)

[3.2.4. JSON Web Token (JWT) 15](#_Toc102298081)

[3.2.5. Socket Io 16](#_Toc102298082)

[3.2.6. Nodemailer 16](#_Toc102298083)

[3.2.7. Bcrypt (Node.js csomag) 17](#_Toc102298084)

[3.2.8. Node Cron 18](#_Toc102298085)

[3.2.9. Uuid 18](#_Toc102298086)

[3.2.10. Joi 18](#_Toc102298087)

[3.3. Adatbázis 20](#_Toc102298088)

[3.3.1. MariaDB 20](#_Toc102298089)

[3.4. Frontend 21](#_Toc102298090)

[3.4.1. React 21](#_Toc102298091)

[3.4.2. Redux 23](#_Toc102298092)

[3.4.3. Axios 25](#_Toc102298093)

[3.4.4. Socket Io Client 25](#_Toc102298094)

[3.4.5. Typescript 26](#_Toc102298095)

[3.4.6. Classnames 26](#_Toc102298096)

[3.5. Proxy 27](#_Toc102298097)

[3.5.1. http-proxy-middleware 27](#_Toc102298098)

[4. Specifikáció 28](#_Toc102298099)

[4.1. Funkcionális követelmények 28](#_Toc102298100)

[4.2. Nem funkcionális követelmények 28](#_Toc102298101)

[4.3. Use case diagram 29](#_Toc102298102)

[5. Rendszer architektúra 30](#_Toc102298103)

[5.1. Rétegek bemutatása 30](#_Toc102298104)

[5.1.1. Frontend 30](#_Toc102298105)

[5.1.2. Backend 31](#_Toc102298106)

[5.1.3. Adatbázis 31](#_Toc102298107)

[5.1.4. Proxy 31](#_Toc102298108)

[5.2. Kommunikációk rétegek között 32](#_Toc102298109)

[5.2.1. Frontend és MangaDex API között 32](#_Toc102298110)

[5.2.2. Frontend és Backend API között 32](#_Toc102298111)

[5.2.3. Frontend és Backend Socket között 33](#_Toc102298112)

[5.3. Frontend 34](#_Toc102298113)

[5.3.1. Struktúra 34](#_Toc102298114)

[5.3.2. React 36](#_Toc102298115)

[5.3.3. Redux store 41](#_Toc102298116)

[5.3.4. Redux reducerek 42](#_Toc102298117)

[5.3.5. Redux akciók 45](#_Toc102298118)

[5.3.6. Axios 47](#_Toc102298119)

[5.3.7. JWT tárolás 48](#_Toc102298120)

[5.4. Backend 49](#_Toc102298121)

[5.4.1. Struktúra 49](#_Toc102298122)

[5.4.2. Routeok 51](#_Toc102298123)

[5.4.3. Ellenőrzés 54](#_Toc102298124)

[5.4.4. Hitelesítés 56](#_Toc102298125)

[5.4.5. Csevegés 58](#_Toc102298126)

[5.4.6. Adatbázis kapcsolat 59](#_Toc102298127)

[5.5. Adatbázis 62](#_Toc102298128)

[5.5.1. Struktúra 62](#_Toc102298129)

[5.5.2. Adatbázis táblák 63](#_Toc102298130)

[5.5.3. Tárolt eljárások 65](#_Toc102298131)

[5.6. Proxy 66](#_Toc102298132)

[5.6.1. Struktúra 66](#_Toc102298133)

[5.6.2. MangaDex Átirányítás 66](#_Toc102298134)

[5.6.3. MangaDex Képek letöltése 67](#_Toc102298135)

[5.6.4. Letöltött képek szolgáltatása 69](#_Toc102298136)

[6. Tesztek 70](#_Toc102298137)

[6.1. Backend 70](#_Toc102298138)

[6.2. Frontend 70](#_Toc102298139)

[7. Továbbfejlesztési lehetőségek 70](#_Toc102298140)

[8. Összefoglalás 71](#_Toc102298141)

[Irodalomjegyzék 72](#_Toc102298142)

[Mellékletek 73](#_Toc102298143)

[9. Felhasználói útmutató 73](#_Toc102298144)

[9.1. Regisztráció, bejelentkezés 73](#_Toc102298145)

[9.2. Manga keresése és olvasása 74](#_Toc102298146)

[9.3. Készítők és fordítók megtekintése 76](#_Toc102298147)

[9.4. Profil változtatása 78](#_Toc102298148)

[9.4.1. Leírás változtatása 78](#_Toc102298149)

[9.4.2. Barát kérések kezelése 79](#_Toc102298150)

[9.4.3. Új fejezet feliratkozások kezelése 80](#_Toc102298151)

[9.4.4. Kedvelt, nem kedvelt mangák kezelése 81](#_Toc102298152)

[9.4.5. Manga lista kezelése 81](#_Toc102298153)

[9.5. Barátokkal csevegés 83](#_Toc102298154)

[9.6. Fórum létrehozása 84](#_Toc102298155)

[9.7. Kommentelés 84](#_Toc102298156)

[10. Telepítési útmutató 85](#_Toc102298157)

[10.1. GitHub repositoryból való telepítés 85](#_Toc102298158)

[10.2. Docker 86](#_Toc102298159)

Jelölésjegyzék

|  |  |
| --- | --- |
| SQL: | Structured Query Language (Struktúrált lekérdezőnyelv) |
| API: | Application Programming Interface (Alkalmazásprogramozási Felület) |
| JWT: | JSON Web Token |
| IRC: | Internet Relay Chat |
| SMTP: | Simple Mail Transfer Protocol (Egyszerű Levél Szállítás Protokoll) |
| UUID: | Universal Unique Identifier (Univezrálisan Egyedi Azonosító) |
|  |  |

# Bevezetés

Azért választottam ezt a témát, mert szakdolgozat téma választás idejében találtam rá a MangaDex API-ra, ami lehetővé teszi, hogy saját manga olvasó programot/weboldalt hozhassunk létre, és mivel szeretem a mangákat és a webfejlesztés témaköre is érdekelt, úgy gondoltam, hogy tökéletes szakdolgozat téma lenne egy saját közösségi weboldalt készíteni, ami a mangák körül forog.

A weboldal olyan embereknek készült, akik szeretnének más emberekkel találkozni és beszélgetni, akikkel közös hobbijuk van, ami ebben az esetben a japán képregények (Manga), de azok az emberek, akik érdeklődnek Mangák iránt. Nagy esetben a japán animációs sorozatok (Anime) és novellák (novellák annyira nem elterjedtek, mint az animék vagy mangák) iránt is érdeklődnek, ez a weboldal platformot biztosít mindegyik fajta szórakoztató médiáról való beszélgetéshez, de csak képregény tartalmat tudnak a felhasználók fogyasztani. A weboldal olyan felhasználóknak ajánlott, akik csak tartalmat akarnak fogyasztani és másokkal interaktálni, mangák (fejezet feltöltés stb.) vagy más fajta tartalom feltöltésére ez a weboldal nem alkalmas, ezt a MangaDex oldalán el lehet végezni.

Az elkészített weboldal más manga olvasóktól annyiban különbözik, hogy közösségi funkciókat is lehet végezni például barátokkal csevegni, publikus profilt beállítani, fórum szálakat (Forum thread) létrehozni és azokon belül beszélgetni. Érdekes funkciónak lehet tekinteni azt is, hogy fel lehet iratkozni manga fordításokra, amikre ha új fejezet érkezik, a regisztrált e-mail címre a felhasználó kap egy e-mailt, ami tartalmazza, hogy milyen mangából, milyen nyelven, új fejezet került feltöltésre, ezért nem kell folytonosan manuálisan ellenőrizni, hogy van-e új fejezet.

# Versenytárs elemzés

Nagyjából 3 fajta manga olvasó weboldal létezik az interneten:

* Kiadó által üzemeltetett oldalak
* Fordító oldalak
* Gyűjtő oldalak

A kiadó által működtetett, és a fordító oldalakkal nem fogunk most foglalkozni, mert bár lehet rajtuk mangákat olvasni, vannak hátrányaik, mint például a hiányzó közösségi funkciók (Ez egy univerzális hiányosság a manga olvasó oldalak többségénél), a fordítói oldalaknál a kevés kínálat, a kiadói oldalaknál az előfizetés és néhány esetben a kevés kínálat.

A gyűjtő oldalak többsége nagy kínálatot biztosít és legtöbbször reklámokkal/adományokkal biztosítják fennmaradásukat. Ezek az oldalak az esetek túlnyomó részében web scraper megoldással különböző oldalakról (Fordító oldalakról vagy más manga kiszolgáló oldalakról) szedik össze a mangákat és adják át az olvasóknak.

A MangaDex (az az oldal, ami a szakdolgozatban elkészített weboldalhoz biztosítja a manga API-t) is gyűjtő oldalnak mondható, de ők a manga szolgáltatáshoz nem web scraper megoldással szerzik a mangákat, hanem saját szerverükre töltik fel a mangákat, amiket a fordítók nyújtanak így is kímélik a fordító oldalakat a fölösleges hotlinkeléstől (Más szerveréről töltjük be a képeket).

A manga olvasó weboldalak többsége a fejezetek alá való kommentelésen (gyakoribb) vagy fórumokon (ritkább) kívül nem rendelkezik közösségi funkciókkal és inkább más platformokon (Discord vagy IRC-ken) folytatnak aktív beszélgetéseket. De a közösségi funkciókon kívül szinte az összes manga olvasó weboldalon lehet mangákat olvasni, saját listákat létrehozni ahová gyűjthetünk például olyan mangákat, amik érdekelnek minket, így könnyebben lehet emlékezni, hogy miket, olvasunk.

# Felhasznált technológiák

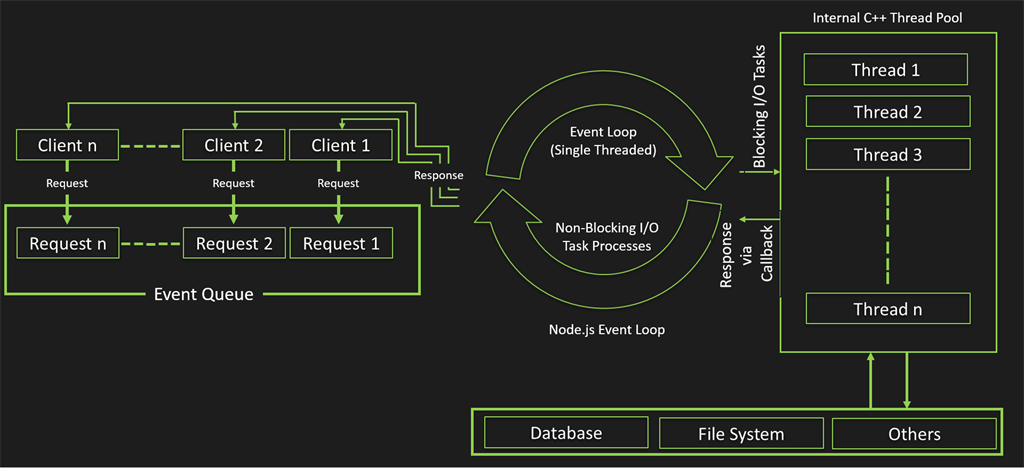
## MangaDex API (Külső API)

A MangaDex API lehetőséget biztosít arra, hogy a MangaDex hatalmas manga tárházával interaktálni tudjunk, legyen az manga információk lekérdezése, új manga feltöltése, fordító csapatok létrehozása és még sok más. Az adatokat REST API módon szolgáltatják, tehát URL címekre történő HTTP kérésekkel lehet lekérni, módosítani, feltölteni és törölni a kívánt adatokat.

## Backend

### Node.JS

A Node.js egy cross-platform futási környezetet biztosít, ami lehetővé teszi a jól skálázható szerverek építését Javascript nyelvben. Node.js a kapcsolatokat „elméletben” 1 szálon dolgozza fel aszinkron módon, gyakorlatban a fő szál (Event Loop) más szálaknak (Workers) osztja ki a feladatokat egy esemény sorból (Event queue) amiket ők elvégeznek a megfelelő C++ megvalósítással párhuzamosan vagy aszinkron módon és visszaküldik az eredményt a fő szálnak, amit visszaadhat a kérést kezdeményezett kapcsolatnak.



1. ábra: Node.js által használt kérés feldolgozás architektúrája [2]

A Node.js legnagyobb előnye a hatékony Input/Output kezelés, mint például a fájlrendszerre való írás/olvasás vagy a hálózatról érkező adatok olvasása. Mivel már elég népszerű lett ez a futási környezet ezért az interneten megtalálható nagyon sok segítség, és mivel Javascriptet használ, ezért a frontend fejlesztők is könnyen beléphetnek a backend világába. Előnyhöz lehetne sorolni a Node.js csomagkezelőjét (Package manager) a Node Package Managert (npm), mert ennek segítségével nagyon hamar adhatunk hozzá a programunkhoz olyan kódokat, amik megkönnyíthetik munkákat, mert nem nekünk kell egyes alap funkciókat megírni, hanem mások által megírt megoldást felhasználhatjuk.

Viszont egyik hátránya a Node.js-nek hogy a CPU intenzív feladatokkal nem tud olyan jól megbirkózni, mint a feljebb említett I/O feladatokkal, mert azokat a műveleteket, amik a CPU-t veszik, igénybe szinkron módon futnak, míg az I/O műveletek aszinkron módon. [1]

### Express

Express.js egy framework (Node.js csomag) amivel könnyen és gyorsan lehet Node.js web applikációt, API-t létrehozni. A Node.js alap http/https moduljait használja fel. Maga az Express sok „opcionális” middleware-el rendelkezik, amiket külön lehet telepíteni, vagy saját middleware-t is írhatunk, amik kibővítik az API vagy a web alkalmazásnak a funkcióit, és mivel ezeket a middleware-eket külön-külön lehet megszerezni és nem az Express alap csomagjában jönnek, elég könnyűsúlyú (lightweight) és könnyen testreszabható a web applikációnk vagy API-nk.

### MySQL (Node.js csomag)

Ez a Node csomag egy driver-t biztosít, amivel csatlakozni lehet egy MySQL adatbázishoz (Mi esetünkben egy MariaDB-hez de a MariaDB és a MySQL szinte azonosak), és egy olyan interfészt biztosít, amivel könnyen lehet az adatbázis felé lekérdezéseket indítani. Ez nem egy ORM, vagyis az SQL utasításokat nekünk kell teljes egészében megírni és lefuttatni.

Ez a Driver lehetőséget ad több egy idejű csatlakozás létesítésére, hibakezelésre, adat eszképelésre (Escaping) hogy megvédjük szerverünket SQL befecskendezés (SQL Injection) és az adatbázisban felmerülő típusokat SELECT query esetén Javascript típusokba kasztolja és egyéni kasztolásra is van lehetőség.

### JSON Web Token (JWT)

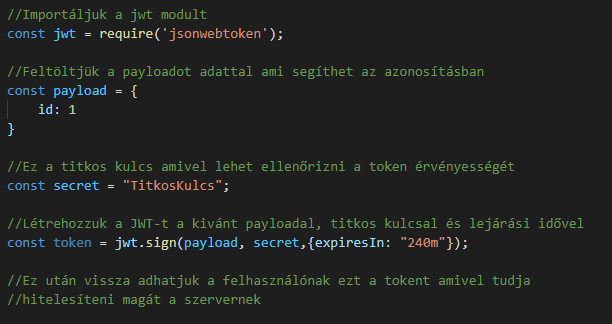
#### JSON Web Token

A JWT módot biztosít arra, hogy kettő vagy több fél között biztonságosan tudjunk adatokat küldeni. Ez a JWT egy hosszú szöveg (String) ami egy JSON-t tartalmaz, amibe bármi féle adat lehet, és le lehet ellenőrizni ennek a token-nek a sértetlenségét, valóságát. Ezt a token-t tovább lehet titkosítani, hogy a tartalmát is teljesen biztonságba tudjuk szállítani, mert alapból a JWT csak azt tudja biztosítani, hogy ha a token-t megváltoztatták, akkor érvénytelen legyen, de ~~attól, még~~ ha valaki megszerzi a token-t, dekódolhatja és a benne lévő adatokat megszerezheti, ezért ha nem használunk további titkosítást csak olyan adatokat küldjünk, amikkel a támadó nem tud mit kezdeni.

Nagyon jól lehet használni hitelesítéshez a JWT-t, például ha a felhasználó bejelentkezik, átadunk neki egy ilyen token-t amit a kliens eltárol, és az elkövetkezendő HTTP kérésekhez hozzá teszi ezt a token-t és a szerveren megvizsgáljuk, hogy érvényes-e. Pozitív vizsgálat esetén a szerver végrehajtja a kérést.

#### Jsonwebtoken (Node.js csomag)

Ez a Node.js csomag implementálja ezt a JWT technológiát. Ez lehetővé teszi Javascript-ben JWT-ket létrehozni, ellenőrizni, dekódolni. Könnyen be lehet állítani a létrehozásnál, hogy milyen algoritmust használjon a titkosításra, mikor járjon le az érvényessége és ki állította ki.

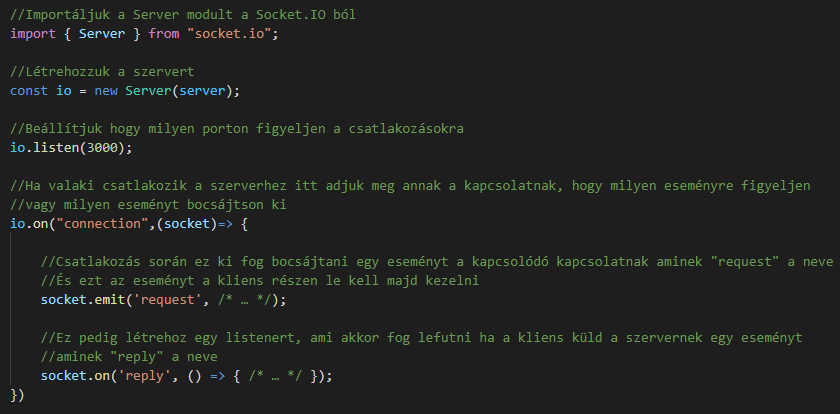


2. ábra: JWT létrehozása jsonwebtoken csomaggal

### Socket Io

Socket.IO egy alacsony késleltetésű, kétirányú esemény alapú kommunikációs könyvtár, ami lehetővé teszi a kliens és szerver közötti valós idejű kommunikációt. [3]

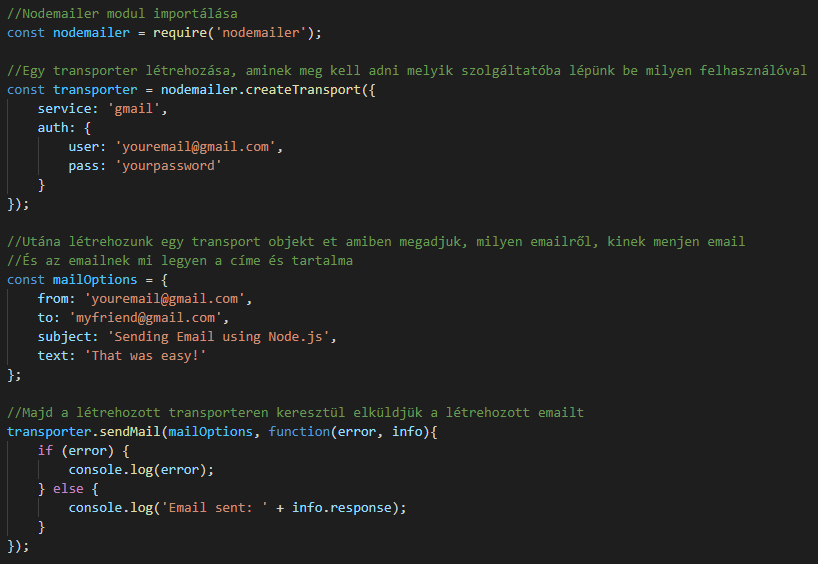
A Socket.IO esemény alapú, ami annyit jelent, hogy a szerver és a kliens, létrehozott eseményekre figyel vagy eseményeket bocsájt ki.



3. ábra: egyszerű szerver felállítása Socket.IO csomaggal

### Nodemailer

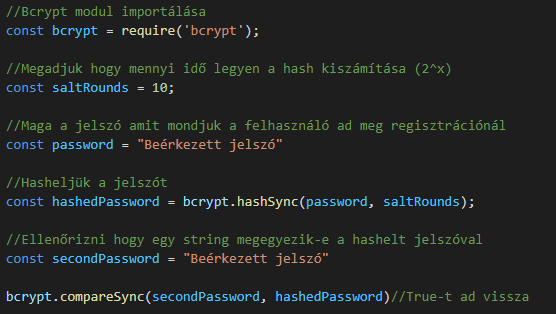
A Nodemailer egy olyan modul, ami lehetővé teszi az e-mail-ek küldését a szerverről. Lehet saját email szerverről (Simple Mail Transfer Procotol – SMTP-vel) és más e-mailszolgáltatásokon keresztül (például: Gmail) emailt küldeni.



4. ábra: E-mail küldése nodemailer segítségével

### Bcrypt (Node.js csomag)

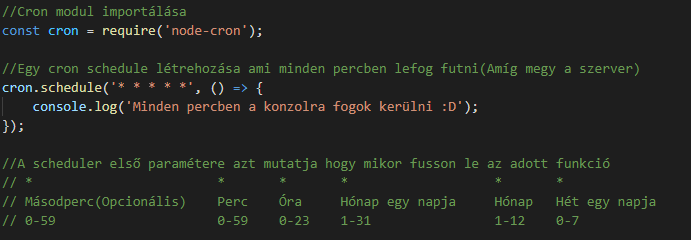
Maga a Bcrypt ~~az~~ egy jelszó hashing funkció, ami egy elég megbízható titkosítást biztosít a jelszavaknak. Én a bcrypt Javascript implementálását használom.



5. ábra: jelszó titkosítása bcrypt-el

### Node Cron

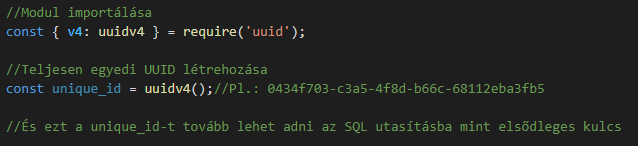
Node Cron lehetővé teszi, hogy bizonyos időpontokban lefuttassunk automatikusan funkciókat. A modul szintaktikája a GNU operációs rendszer crontab-ján alapszik, ami mint a Node Cron, lehetővé teszi, hogy bizonyos időpontban parancsok, fájlok fussanak le.



6. ábra: ütemezett funkció megadása node-cron-al

### Uuid

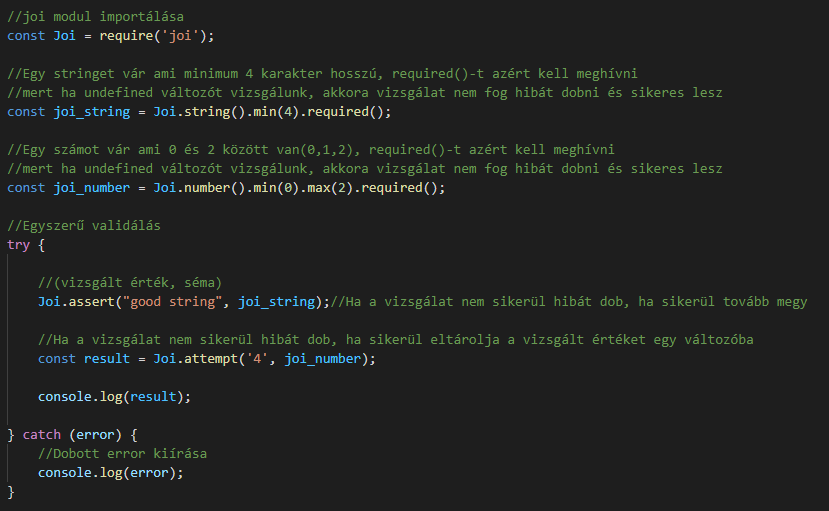
A Universal Unique Identifier (UUID) egy nagy valószínűséggel teljesen egyedi azonosító kódot jelent, amit sok különböző tényezőből például a létrehozásnál lévő időpont, hálózati cím segítségével hoz létre egy algoritmus. Én ennek az algoritmusnak a Node.js-beli implementációját használom (uuid ami a Node.js csomagnak a neve) az adatbázisban tárolt adatok azonosításához.



7. ábra: UUID létrehozása

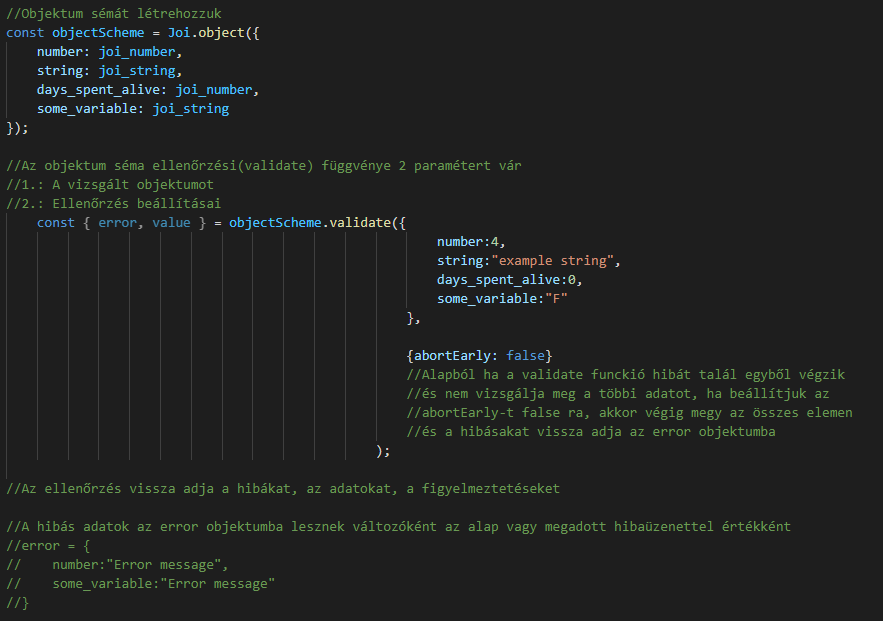
### Joi

A Joi az egy olyan modul, amivel adatsémákat lehet létrehozni, és adatokat ellenőrizni, hogy megfelelnek-e a várt típusnak, tartalomnak, legyen az egy szám 1-től 10-ig vagy egy minimum 8 karakter hosszú, minimum egy nagybetűvel, minimum 2 számmal és minimum 1 speciális karakterrel rendelkező szöveg (String). Ezeknek a sémáknak a létrehozására vannak beépített metódusok, amik megkötik, hogy milyen adat legyen, szám, szöveg, mi a minimum, maximum érték, hossz stb. De lehet reguláris kifejezéseket is használni, hogy komplexebb eseteket értékeljünk ki.



8. ábra: Skaláris adatellenőrzés JOI csomaggal

De egyesével vizsgálni, például egy objektumnak a sok értékét, időigényes, ezért az ajánlott módszer igazából egy objektum sémát ellenőrizni.



9. ábra: Komplexebb adatszerkezet ellenőrzése

## Adatbázis

### MariaDB

MariaDB egy nyílt forráskódú (Open source) relációs adatbázis, eredetileg a MySQL egy továbbfejlesztett változata. Nagyon magas a MySQL-el a kompatibilitása, ez látható abban is, hogy a backend szerveren én igazából egy MySQL drivert használok a MariaDB eléréséhez és minden jól működik és a MariaDB általában jobban skálázhatóbb és gyorsabb, mint mondjuk a MySQL. [4]

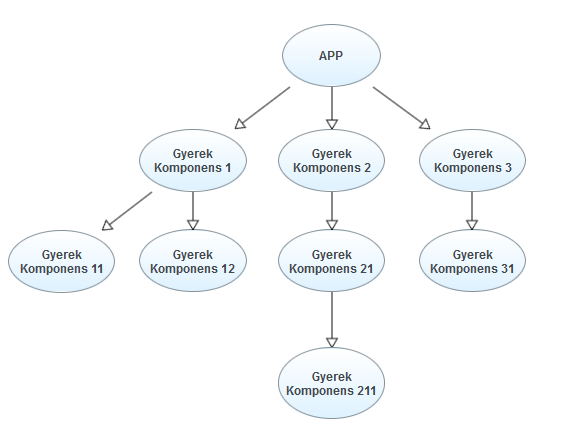
## Frontend

### React

React ~~az~~ egy Javascript könyvtár felhasználói felületek létrehozásához. Komponens alapú, ami annyit jelent, hogy valamilyen logikát és HTML elemeket egy komponensbe összegyűjt, és meghívni/felhasználni lehet a kívánt helyen.

Ezeknek a komponenseknek van saját állapota is, vagyis a komponensen belül lehet azokat az adatokat tárolni, amiket felhasznál, de nem csak adatokat tárolhat, hanem funkciókat is, amik megváltoztathatják a komponens állapotát, kinézetét.

Ezek a komponensek mindig valamilyen más komponensnek a gyermekei és minden egy fő komponensből indul ki (gyökér komponens) és fa struktúrában helyezkednek el.



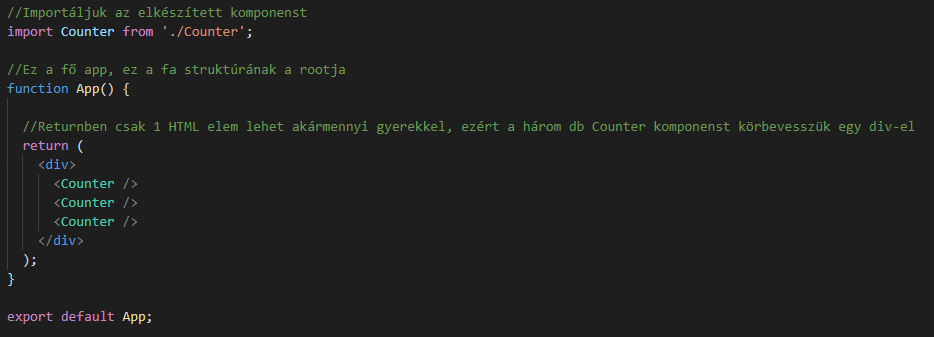
10. ábra: React alkalmazás komponens struktúrája

Egy példa a React komponens rendszerére.



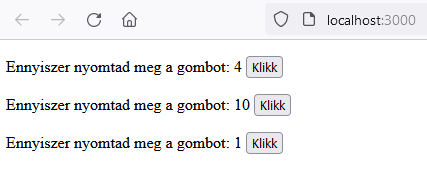
11. ábra: React funkcionális komponens létrehozása

A fenti kód létrehoz egy komponenst, aminek saját állapota van, és azt kezeli (count változó).



12. ábra: React alkalmazásban komponens használata

Itt pedig az elkészített komponenst importáljuk az App komponensbe, ami az applikációnak a gyökér eleme.



13. ábra: 3 komponens bemutatása a böngészőben

És ha elindítjuk a kliens szervert, lehet látni a 3 komponenst, amit a gyökér komponensbe beletettünk, és ha rákattintunk az egyik klikk gombra, akkor a mellette lévő számláló növekedni fog.

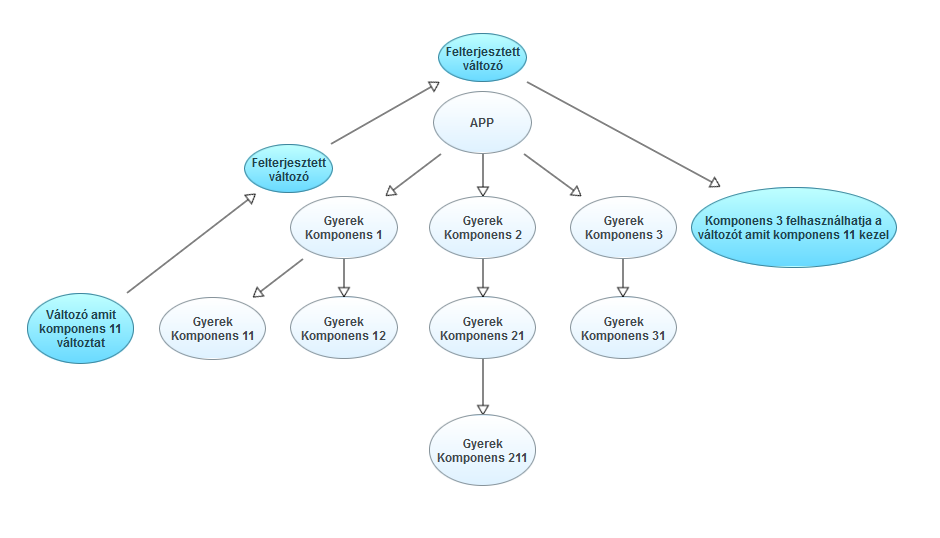
Ez csak egy bevezetés volt a React komponens rendszerébe, de sokkal több funkciót is ellehet vele végezni, mint például feltételes megjelenítést, ami annyit jelent, hogy valamilyen feltételek hatására, a komponens mást fog megjeleníteni.

### Redux

Maga a Redux globális állapot tárolásra használható, amivel könnyen nyomon lehet követni a programnak a változásait, továbbá hibakeresésre is alkalmazható. Ez bármilyen Javascript applikációhoz használható, de a React-hoz van egy hivatalos React Redux változat is, ami lehetővé teszi a komponenseknek a store-al (Az a hely ahol tároljuk az applikációnak az adatait, állapotát) való kommunikációt.

Redux alapból immutable (változhatatlan) vagyis nem direkt változtatjuk az adatok állapotát, hanem egy másolatot készítünk az adott állapotról, amit változtathatunk és utána a másolattal felülírjuk az eredeti állapotot.

Hogy miért kell egy globális állapot kezelő? Azért, mert a React-ban a komponensek fa struktúrában helyezkednek el, és ha egy komponensből át akarunk vinni egy másik komponensbe egy adatot, akkor azt fel kell terjeszteni a szülőbe, ami majd a másik komponensnek átadja az adatot. Ez a megoldás kicsi utakon, kicsi projektben még használható, de ha nagy a projekt vagy az út, amit meg kell tennie az adatnak, akkor jobban járunk, ha használjuk ezt a globális állapot megoldást.



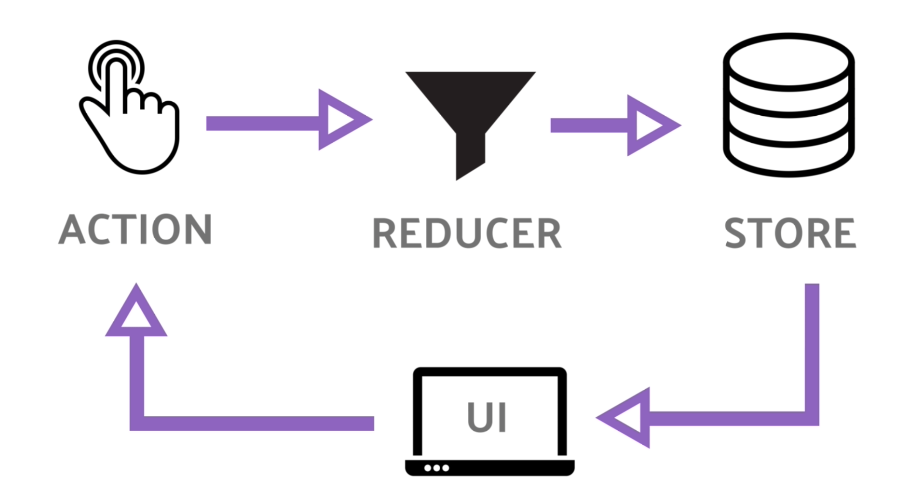
14. ábra: React fa struktúrájában adat megosztás más komponensekkel

Ahogy a példán látható a gyerek komponens (gyk) 11 ben változtatjuk a változó értékét, de hogy elérje egy másik komponens, fel kell terjesztenünk a szülőbe ezt a változót, ami igazából annyi jelent, hogy a szülő komponensben is létezik ez a változó, de a gyk-nak továbbadjuk ezt a változót, és a változtatáshoz szükséges metódusokat. Ez a megoldás az érintett komponenseken végig viszi a változót, míg a Redux segítségével egy közös, globális helyen kezelhetjük azokat az adatokat, amiket 1 vagy több komponens is használni akar.

Redux 3 fő elemből áll

* Store
* Action
* Reducer

A store tartalmazza az adatokat, és a felhasználói interfész (UI) innen kapja az adatokat, amiket megjeleníthet vagy feldolgozhat. Az action az igazából egy funkció, amit meghívhatunk, adhatunk neki adatot, amit feldolgoz és a legfontosabb része az, hogy a reducer-nek átadja ezt az adatot. A reducer meg a store-ban lévő adatokat változtatja meg legtöbbször az action-től kapott adatra.



15. ábra: React Redux folyamat ábrája [5]

A Redux adat folyama a fönti ábrán látható módon működik. A felhasználói felületen automatikusan vagy valamilyen esemény hatására, meghívunk egy action-t, ami valamilyen adatfeldolgozást csinálhat (pl.: kapott adat + 1) és utána az action akár több reducer-t is meghívhat, amik megváltoztatják a store-ban lévő értékeket, amiknek a változása megjelenik a felhasználói felületen, ha éppen megjeleníti azt.

### Axios

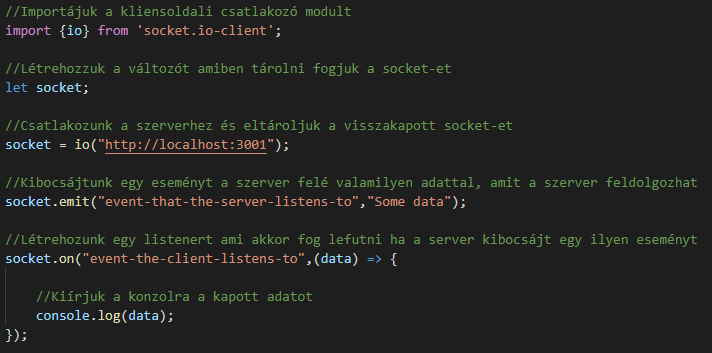
Axios az egy promise (ígéret) alapú HTTP kliens, amivel HTTP kéréseket lehet küldeni böngészőből és Node.js alkalmazásból. Javascript-nek a Promise API-ját használja ezért az aszinkron kéréseket lehet kezelni .then() és .catch() megoldással, vagy az await kulcs szóval hogy csak akkor történjen adat felhasználás, ha a kérésre érkezett válasz.

Van még több funkciója is a HTTP kérések indításán kívül például a kimenő kéréseket intercept-álja és új adatokat ad hozzá a kéréshez pl.: Minden kéréshez hozzáadja a Header-be az hitelesítéshez szükséges token-t.

### Socket Io Client

Ez a szerveren használt socket es kommunikációnak a kliens oldali fele, itt is egy socket-en lehet létrehozni eseményeket, amire figyel a kliens és lefutnak a megadott funkciók, ha a kliens kap egy olyan eseményt. Vagy lehet eseményeket küldeni a szerver felé.

Ahhoz hogy ezt a socket-et megszerezzük, ami tartalmazza az információkat ahhoz, hogy a szerverrel kommunikáljunk csatlakozni, kell a szerverhez és a visszakapott socket-et tároljuk.



16. ábra: Socket.IO esemény kibocsájtás és figyelés példa

### Typescript

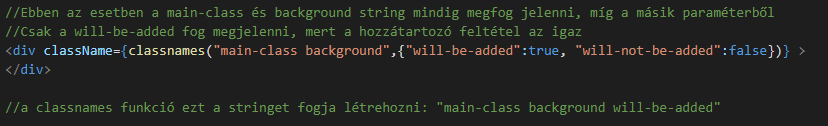
TypeScript az egy programozási nyelv, igazából Javascript, csak lehetővé teszi, hogy típusokat használjunk, amik csökkenthetik a Javascript gyengén típusos szerkezetéből adandó hibákat.

Lehetővé teszi még, hogy a funkcióknak megadjuk, milyen típusú paramétereket várjon és milyen típusú értéket adjon vissza. Névtereket (namespace), enumerátorokat és saját objektum sablonokat is könnyen létre lehet hozni. A saját készítésű típusokat például enumerátorok, objektum sablonokat is hozzá lehet rendelni változókhoz, függvényekhez.

A TypeScript azért is jó mert fordításnál (compile) ellenőrzi, hogy az adott típusok jó fajta adatot kapnak, és ha érzékeli, hogy valamilyen változó rosszfajta értéket kap, vagy olyan funkciót akarunk elvégezni, rajta amit a típusán nem, lehetne, azt jelzi a TypeScript.

### Classnames

Ez egy Javascript modul, amit HTML elemek osztályneveinek feltételes összekapcsolás könnyítésére hoztak létre. Igazából ez a modul csak annyit csinál, hogy a megadott szövegekből (string) létrehoz egy új szöveget (string) amivel a CSS osztályokat lehet hozzárendelni egy HTML elemhez. A feltételes összekapcsolást a szövegekhez (string) tartozó feltétel kiértékelésével végzi el, vagyis ha egy szöveghez (string) tartozik egy feltétel, az akkor fog megjelenni a végső szövegben (string) ha a feltétele igaz.



17. ábra: classnames csomag használatára példa

## Proxy

### http-proxy-middleware

Ezzel a Node.js modullal létre lehet hozni egy proxy middleware-t, amit egy HTTP proxy működtetésére lehet felhasználni. Egy HTTP proxy kettő vagy több szolgáltatás között létezik, és rajta keresztül kommunikálnak ezek a résztvevők.

Főként a kliens és a szerver között helyezkedik el, ami annyit jelent, hogy a kliensről érkező kéréseket továbbítja a megfelelő szolgáltatásnak, ami lehet saját szerver vagy külső szolgáltatás is (Valamilyen API).

De a proxy végezhet valamilyen műveletet is ezekkel a kérésekkel mielőtt továbbadja, lehet az naplózás, érvényesség ellenőrzése, header vagy body módosítása. De fel lehet használni akár arra is, hogy megfelelő módon elossza a terhet a különböző szerverek között, hogy ne terhelődjenek aránytalanul.

Az applikációnkat biztonságosabbá is lehet tenni azzal, hogy, elrejtjük azokat az erőforrásokat, amiket felhasználunk, így nem fog kikerülni a nagyvilágba, hogy milyen szolgáltatásokat használunk, azokat hol lehet elérni és a legfontosabb, hogy a szolgáltatáshoz használt kulcsokat és hitelesítéshez szükséges adatokat nem a kliensen tároljuk, hanem a proxy-n, amihez a kliens nem fér hozzá.

# Specifikáció

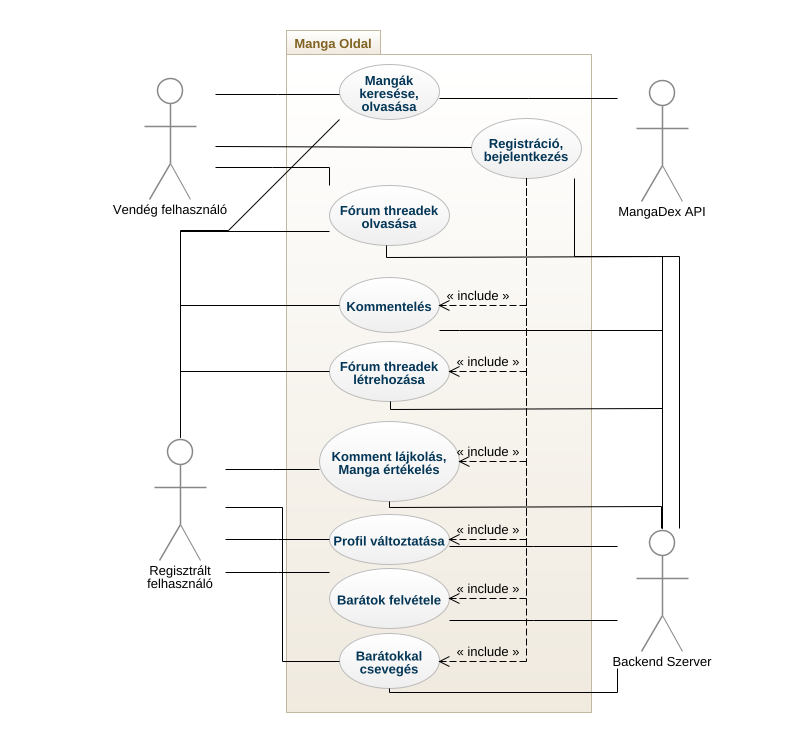
## Funkcionális követelmények

* Mangák keresése.
* Mangák olvasása.
* Mangák értékelése.
* Manga írók, fordítók megjelenítése.
* Mangák listákba gyűjtése.
* Manga alá kommentelés.
* Manga fejezet alá kommentelés.
* Fórum thread létrehozása.
* Fórum thread alá kommentelni.
* Kommentek értékelése.
* Bejelentkezés.
* Regisztráció.
* Profil adatok változtatása.
* Barátok felvétele.
* Barátokkal csevegés.
* MangaDex-ről hotlinkelés elkerülése.

## Nem funkcionális követelmények

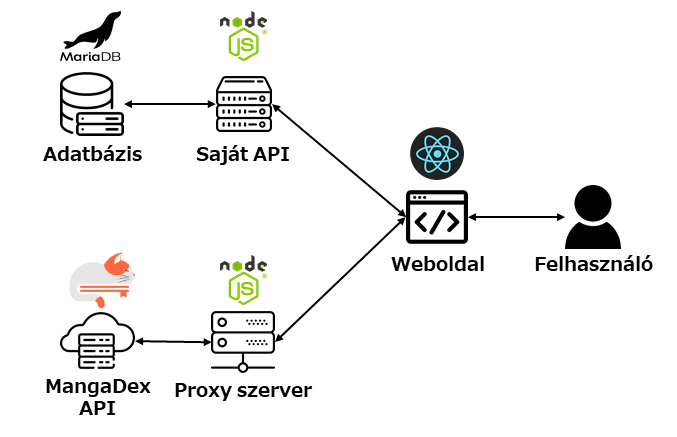
* Backend szerver Node.js API szerver.
* Adatbázis MariaDB.
* Frontend React-ban íródjon.

## Use case diagram



18. ábra: Program Use Case diagrammja

# Rendszer architektúra



19. ábra: Szakdolgozat programjának az architektúrája

A weboldalról két féle kérést lehet indítani, a MangaDex felé és a saját API felé. A MangaDex felé olyan kérések küldhetők, amik a mangákkal kapcsolatos például keresés, olvasás, míg a saját API felé főként a szociális funkciókért indít kéréseket az oldal például bejelentkezés, kommentelés vagy barátokkal csevegés.

## Rétegek bemutatása

### Frontend

A weboldalhoz React-ot használok, mert az a fajta modularitás, amit a React nyújt, átlátható és könnyen kezelhető kódot biztosít komponensek formájában.

A frontenden adatok tárolására React Redux-ot használok, ami egy globális állapotban (Store) tud tárolni adatokat, amiket a program bármelyik komponenséből elérhetünk, és könnyen lehet követni a Store-ban történő változásokat, ami elősegíti az hibakeresést.

### Backend

Backend szerverhez Node.js-t használok, ami egy Javascript futási környezetet biztosít, szóval lehet Javascript kódokat futtatni a böngészőn kívül is [1].

A projekthez használt backendnek fő feladata a közösségi funkciók szolgáltatása a frontendnek, de 2 fajta módon szolgálja ki a klienst:

* REST API (HTTP kérések)
* Valós idejű szerver-kliens kommunikációval

A valós idejű szerver-kliens kommunikáció biztosítja a barátok közötti beszélgetés lehetőségét, míg a REST API HTTP végpontokkal biztosítja az adatok szolgáltatását és változtatásának a lehetőségét.

Az API végpontjainak kezelésére Express.js-t használok, a szerver-kliens valós kommunikációhoz pedig Socket.IO-t.

### Adatbázis

Az adatok tárolására MariaDB-t használok, ami egy relációs adatbázis. A backend és az adatbázis egy Node.Js csomag (mysql) segítségével kommunikál.

### Proxy

Magához a weboldalhoz még használok egy proxy szervert is, ami ugyan úgy Node.js-ben fut Express.js és egy Node.js csomag (http-proxy-middleware) segítségével. Ez a proxy a frontendről érkező azon kéréseket kapja, amik a külső API felé mennek.

Hogy miért használok proxyt? Elsősorban azért, hogy a MangaDex-en lévő képeket legyen az Manga borítókép vagy a manga fejezetek paneljei, letöltsem saját tárhelyre, hogy elkerüljük a hotlinkelést, szóval, ha valaki mangát olvas a mi oldalunkon, akkor a képeket nem a MangaDex szerveréről fogja megkapni, hanem saját szerverről így csökkentjük a MangaDex szerverén a terhelést. A másik indok, amiért proxyt használok, hogy elrejtsem az API útvonalát. Míg én most csak útvonal elrejtésre, és hotlinkelés elkerülésére használom, a proxy használata tökéletes arra, hogy biztonságos helyen (saját szerveren) tároljuk külső API-khoz szükséges hitelesítő kulcsokat, így nem áll fenn annak a veszélye, hogy kikerül a nagyvilágba.

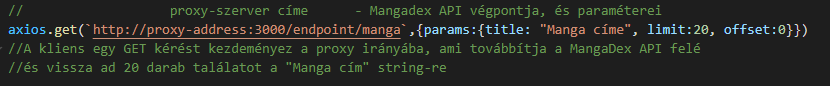
## Kommunikációk rétegek között

Legtöbbször a rétegek közötti kommunikációt a kliens kezdeményezi, ez lehet felhasználó által indított esemény miatt vagy automatikus, például Manga keresésnél a felhasználó beír egy címet, amire keresni akar, vagy maga az oldal a betöltésnél HTTP kéréseket küld a MangaDex API felé, hogy kapjon Manga részleteket, borítóképet, amit megjelenít a felhasználói felületen.

Azért legtöbbször a kliens kezdeményezi a kommunikációt, mert a chat esetében a beérkező üzenet a szervertől jön.

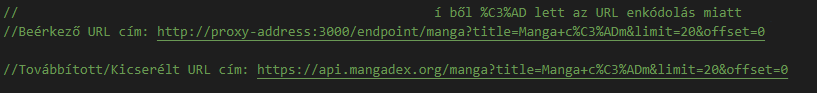
### Frontend és MangaDex API között

A frontend és a külső API között a kommunikáció általában úgy halad, hogy a proxy szerverre küldök egy HTTP kérést az Axios Node.js csomag segítségével, úgy hogy a proxy URL után a megfelelő MangaDex API végpontot, és megfelelő paramétereket megadom.



20. ábra: Frontend hogyan, milyen kérést küld a Manga szolgáltató felé

Miután elküldöm a kérést a proxynak, a proxy a beérkező URL-nek a proxy címét lecseréli a megfelelő MangaDex API szerver címére.



21. ábra: proxy által továbbított URL cím

Majd a továbbított URL címről visszakapott választ visszaadja a kliensnek, ami megfelelően feldolgozza azt.

### Frontend és Backend API között

A frontend és a saját backend API közötti kommunikáció annyiban egyszerűsödik, hogy nem megy keresztül egy proxyn az egész folyamat, hanem egyből a szerverre küldi a kéréseket.



22. ábra: Backend szerver felé küldött kérés

És miután megkapja a kliens a kérés eredményét, megfelelően feldolgozza azt.

### Frontend és Backend Socket között

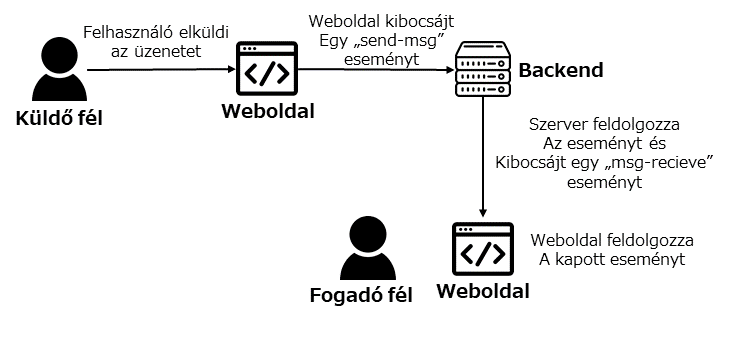
A kliens és backend nem csak HTTP kérésekkel kommunikál, hanem egy kétfolyamú valós idejű socket kommunikáció is történik. Ezt a fajta kommunikációt a Socket.IO Node.js csomag biztosítja.

Az alkalmazásban akkor történik kommunikáció a szerverrel, amikor a felhasználó bejelentkezik és csatlakozik a szerverhez, amikor a kliens üzenetet kap, és amikor a kliens üzenetet küld.

Csatlakozás során a kliens küld egy „add-user” eseményt a szerver irányába, amit az lekezel és hozzáadja a kapcsolatot egy globális listához, ami az aktív felhasználókat tartja nyilván.

Amikor üzenetet küld a kliens, akkor a szervernek küld egy „send-msg” eseményt, amit a szerver feldolgoz és továbbítja a megfelelő kliensnek.

Üzenetet akkor kap egy kliens, ha egy másik kliens küld neki egy üzenetet. A másik kliens küldött üzenetét a szerver feldolgozza, és továbbítja a megfelelő kliensnek az üzenetet egy „msg-recieve” esemény kibocsátásával, amit a fogadó kliens megkap, és feldolgozza az üzenetet.



23. ábra: socket-es szerver-kliens kommunikáció ábrája

## Frontend

### Struktúra

Nagyon sok fájl van ahhoz, hogy a struktúrát felsoroljam láthatóan, ezért csak a fő fájlokat és a mappákat sorolom fel és írom, le miket tartalmaznak.

+client

| .env

| package.json

+---node\_modules

+---public

+---src

| | index.tsx

| +---app

| | App.tsx

| | store.tsx

| +---actions

| | | types.tsx

| +---common

| +---helpers

| +---layout

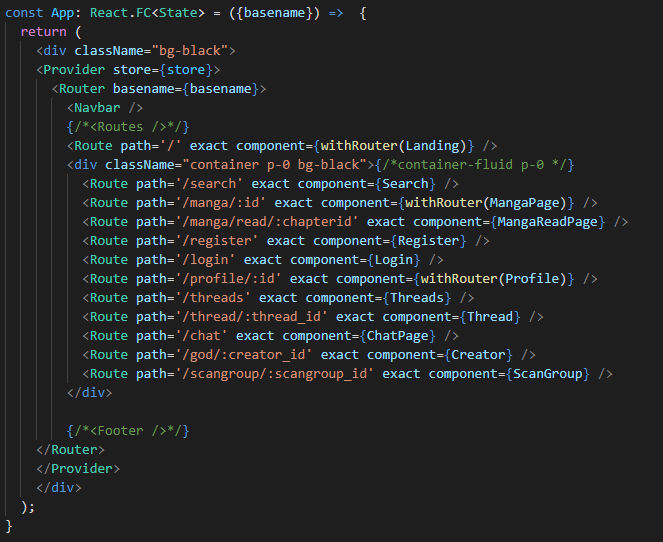
| +---modules

| +---reducers

| | index.tsx

* **.env:** Egy olyan fájl, amivel szimulálni tudjuk a környezeti változók jelenlétét, anélkül hogy a szervert futtató rendszeren ténylegesen beállítanánk, ez segít a futtatáshoz szükséges globális változókat egy helyre összegyűjteni.
* **Package.json:** A program meta adatait tartalmazza, mi a projekt neve, verzió szám, használt csomagok listája, futtatási scriptek és még sok más beállítás.
* **Node\_modules mappa:** A felhasznált csomagok találhatók, ez a mappa automatikusan kerül generálásra.
* **Public mappa:** Itt tárolja a fő HTML fájlokat, amibe a React beilleszti a megfelelő komponenseket.
* **Index.tsx (src/index.tsx):** Ez a fájl felelős azért, hogy a fő HTML fájlt össze kösse a React Applikáció fő komponensével.
* **App.tsx (src/app/App.tsx):** Ez az applikáció fő komponense, itt tárolom, hogy melyik URL-en milyen komponens jelenjen meg
* **store.tsx (src/app/store.tsx):** Ez a fájl felelős a React Redux felállításáért és beállításáért.
* **actions mappa (src/app/actions):** Ebben a mappában tárolom a Redux által használt akciókat.
* **actions mappa types.tsx:** Ebben a fájlban vannak a lehetséges akciók gyűjteménye.
* **common mappa (src/app/common):** Itt olyan komponensek vannak, amiket sok helyen használunk.
* **helpers mappa (src/app/helpers):** Olyan funkciókat tárol, amiket meghívok valamilyen feladat elvégzésére például idő formátum átalakításához.
* **layout mappa (src/app/layout):** Ebben a mappában az applikáció főoldala van és a navigációs sáv.
* **modules mappa (src/app/modules):** Itt azok a komponensek vannak, amik a különböző oldalakhoz tartoznak például profil, fórum, csevegés.
* **reducers mappa (src/app/reducers):** A Redux-hoz tartozó reducer-eket tárolja, a reducer fájlok a globális állapot egy-egy fő állapotát jelenti.
* **reducer mappa index.tsx:** Ez a fájl fűzi össze a reducer-eket (állapotokat) egy fő állapottá.

Az applikáció különböző oldalai a fő komponensben (App.tsx) kerül beállításra. A react-router-dom csomagban van egy komponens, ami egy megadott URL címre egy megadott komponenst jelenít meg.

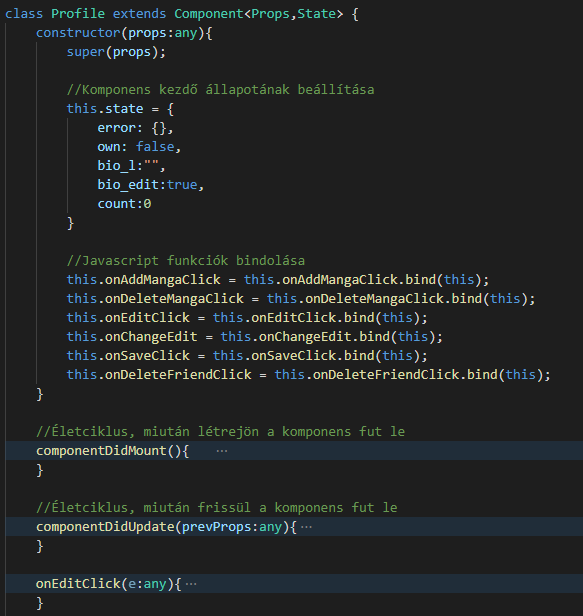


24. ábra: React alkalmazás hogyan kezeli az URL végpontokat

### React

#### Komponens fajták

Kétféle komponens van a React-ban, osztály (class) komponens és funkcionális komponens, a class komponens az a React komponens osztály implementálása, míg a funkcionális komponens egy Javascript funkció, ami egy JSX element-et ad vissza. A class komponensben lehet életciklusokat definiálni és az osztály konstruktorában a komponens állapotát beállítani.



25. ábra: React osztály komponens konstruktorja, életciklusai

Míg a funkcionális komponensben úgynevezett hook-okkal lehet a komponensnek életciklusokat és állapotot megadni.



26. ábra: React funkcionális komponens

A kettő megközelítés annyiban különbözik, hogy a funkcionális komponens szintaktikája jobban olvasható, mert a class komponensben használni kell a this kulcsszót.

#### Design

A designhoz bootstrap 5 (CSS), bootstrap icons (ikonok), fontawesome (ikonok), react-multi-carousel (képek körhintás megjelenítése) és saját CSS fájlt használok. A saját CSS fájlban inkább csak szín osztályok vannak, mert az elemek alakját, elhelyezkedését a bootstrap-el állítom be.



27. ábra: HTML elem CSS osztályai

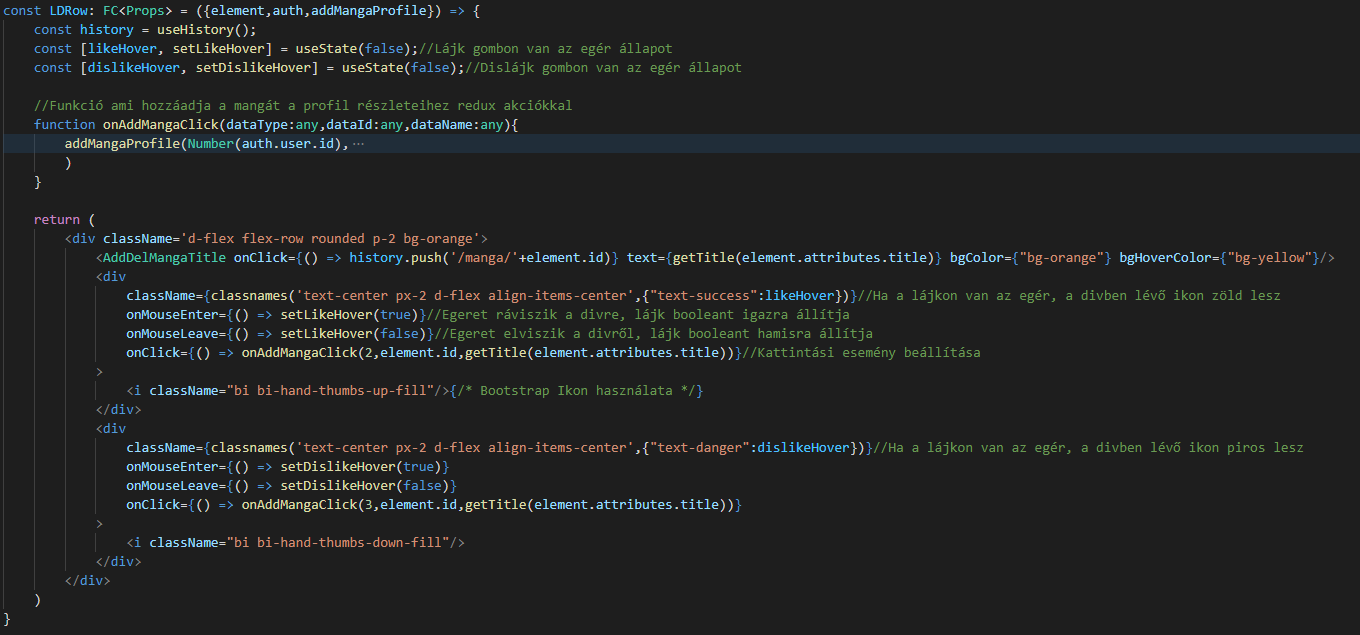
Szezonális mangák div CSS beállítása, narancssárga háttér, 3px vastag fekete szegély a div alján, lekerekített sarkú, 2px es margóval div alján, és pár pozíció CSS osztály.



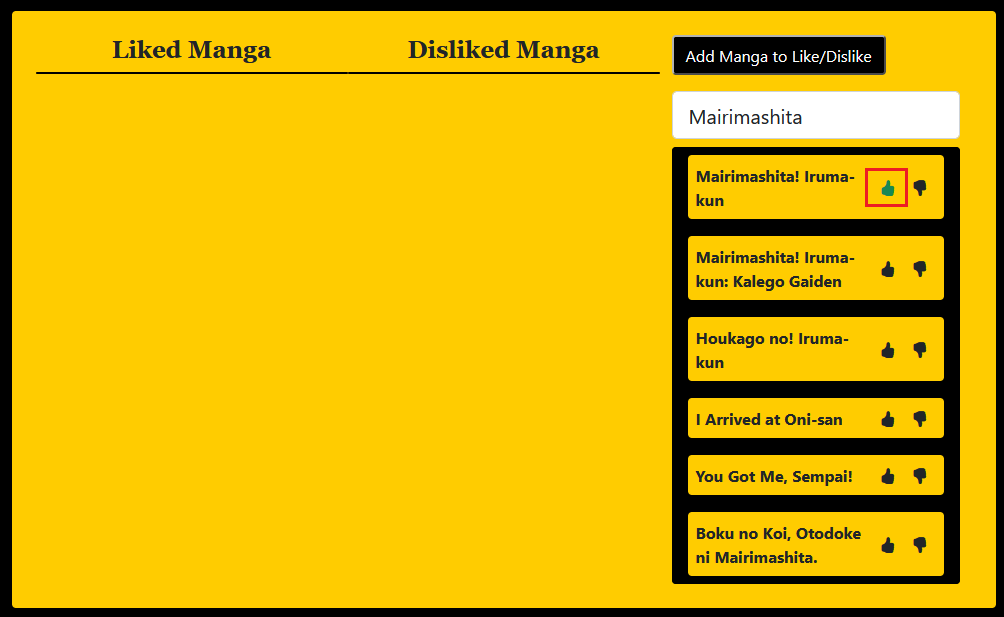
28. ábra: Program főoldala

#### Classnames

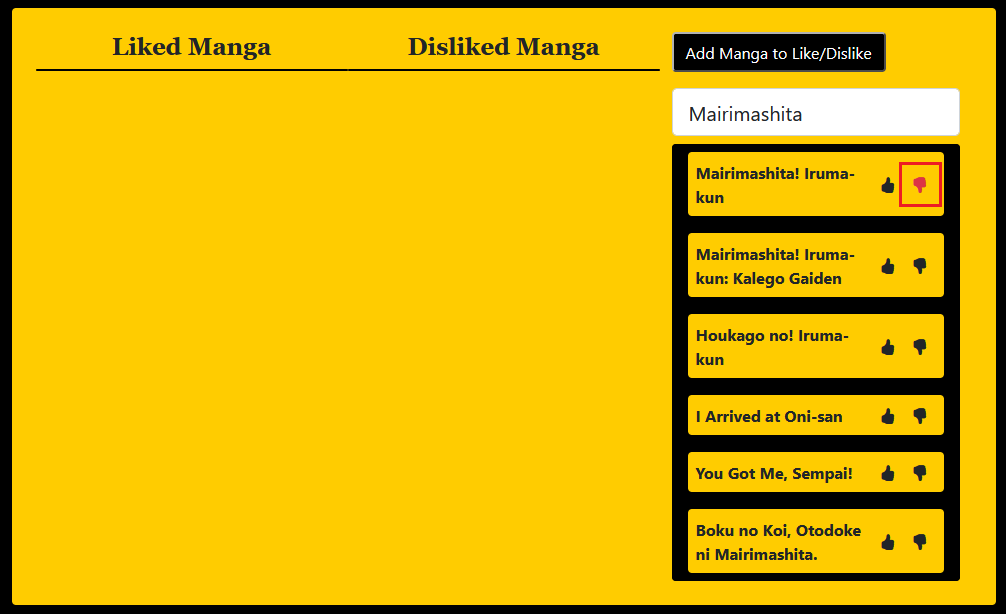
A classnames csomag ugye egy modul, amit HTML elemek osztályneveinek feltételes összekapcsolásának könnyítésére hoztak létre. Ezt a feltételes összekapcsolást a HTML elemek színeinek megváltoztatására használtam főként. Mikor ráviszik az egeret, hogy vizuális jelet adjon, hogy rá lehet kattintani és mutassa, hogy min van az egere a felhasználónak.



29. ábra: Manga kedvelés, nem kedvelés komponens kódja



30. ábra: Manga kedvelés, nem kedvelés komponens



31. ábra: Manga kedvelés, nem kedvelés komponens

### Redux store

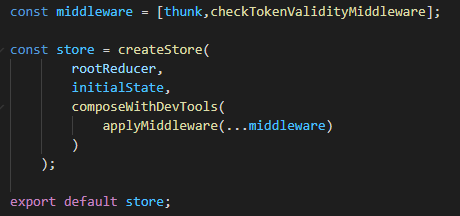
A store.tsx fájlban hozom létre a React store-t, ami vár egy reducer-t (állapot), kezdő értéket és utána opcionálisan meglehet adni middleware-eket, amik minden action esetén lefut.



32. ábra: React Redux állapotainak importálása

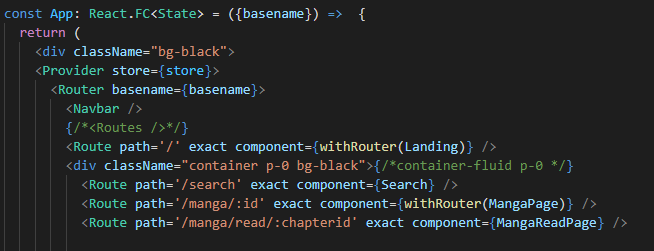


33. ábra: React Redux globális állapot alapértékeinek beállítása



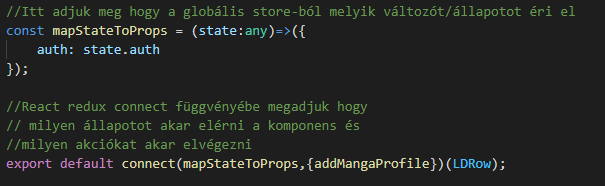
34. ábra: React Redux globális állapot létrehozása

Miután létrehozom a store-t a megadott beállításokkal exportálom és a fő komponensben (App.tsx) egy Provider komponensen keresztül az összes gyerek komponens számára elérhetővé teszem a store-t.



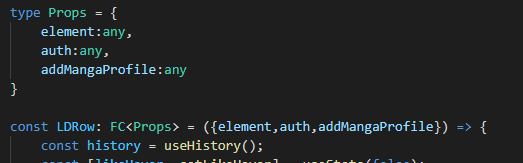
35. ábra: React Redux szolgáltatás hozzáadása a React applikációhoz

A komponensek úgy tudják elérni a globális állapotot, hogy a komponenst a React Redux connect függvényével becsomagolva exportálom, ebbe a függvénybe megadom, hogy a store-nak melyik részét tudja elérni, és utána a komponensnek megadhatjuk azt az állapotot bemeneti paraméterként, amiből utána le tudja kérdezni az értékét.

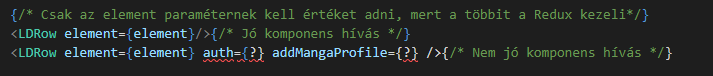


36. ábra: Komponenst csatlakoztatjuk a globális állapothoz

Amit itt megadtam állapot (36.ábra, auth) azt a komponens megírásakor meg kell adni (37.ábra), de mikor más komponensben akarjuk használni, akkor nem kell annak az adattagnak értéket megadni (38.ábra), mert azokat a Redux store szolgáltatja.



37. ábra: Komponensnek átadjuk, hogy milyen adatokat kap



38. ábra: Komponens jó és rossz használata

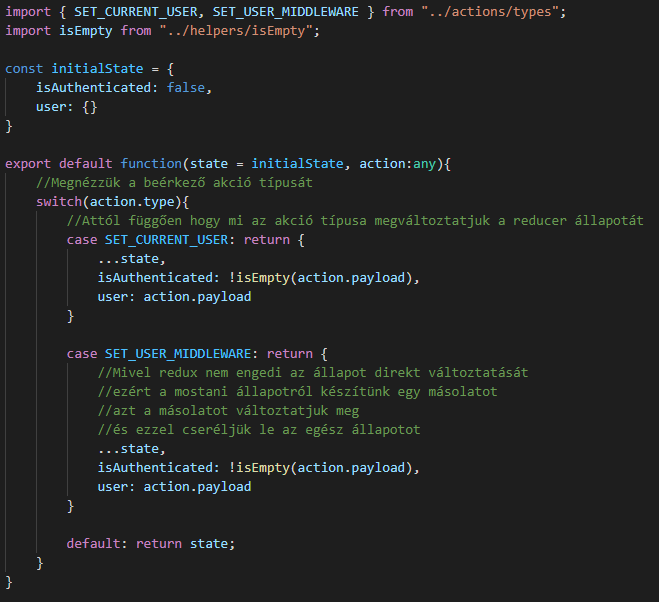
### Redux reducerek

10 db reducer-t használok, ezek a reducer-ek a store-ban egy-egy fő érték objektumot reprezentálnak. Ezeket a reducer-eket a fő reducer-ben összesítem és adom át a store-nak, amikor létrehozzuk a store-t.



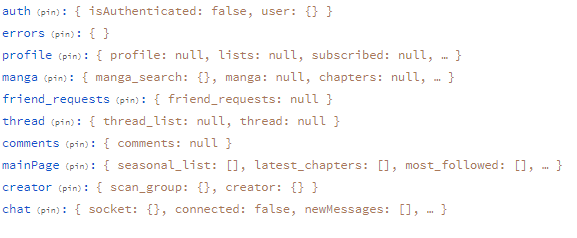
39. ábra: Fő reducerben összegyűjtjük a különböző reducereket

Minden reducer a saját kis állapotáért felel és tartja nyilván. Mindegyik reducer-ben létrehozunk egy alap állapotot, ami megfogalmazza a reducer fő struktúráját. Azon belül igazából egy funkciót exportálok, amiben egy switch case statement kezeli az adatokkal történő lehetséges eseményeket (40.ábra).



40. ábra: auth reducer felépítése (kezdő adatok, milyen akciókat dolgoz fel)

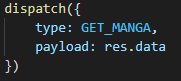
Hogy mik lehetnek egy akció típusa, az az actions mappában a types.tsx-ben vannak definiálva, és onnan importálom a szükséges típusokat.



41. ábra: React Redux store struktúrája mikor az oldal elindul

### Redux akciók

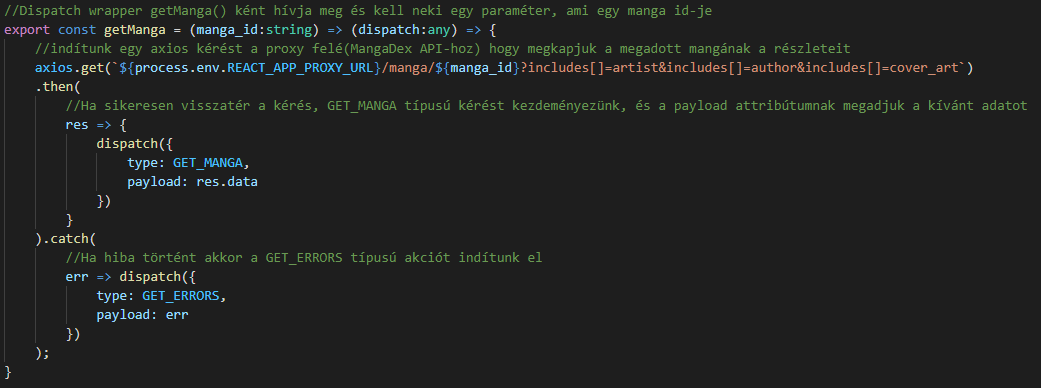
A Redux akciókkal kezdeményezünk változást a store felé ezt a redux dispatch() metódusával lehet csinálni, bárhol meglehet hívni ezt a dispatch() metódust, az a lényeg hogy paraméterként egy olyan objektumot kapjon, aminek van type attribútuma, mert ez mutatja meg a redux-nak hogy milyen fajta akciót akar elvégezni.



42. ábra: GET\_MANGA akció meghívása dispatch() metódussal

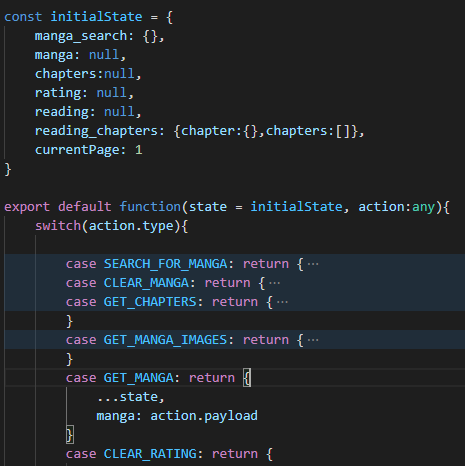
De így nem érdemes akciókat kezdeményezni, mert elrontja a kód olvashatóságát, ha mindenhol így hívjuk meg az akciókat, és ha valami plusz adatfeldolgozást kéne elvégezni egy akcióhoz, azt mindig megkéne írni az adott helyen ahol meghívjuk az akciót.

Ezért a legtöbb akció hívást egy másik metódussal körül veszem ahol az akcióhoz szükséges adatfeldolgozást, adat hívásokat elvégzi és utána megkezdi az akciót (43.ábra).



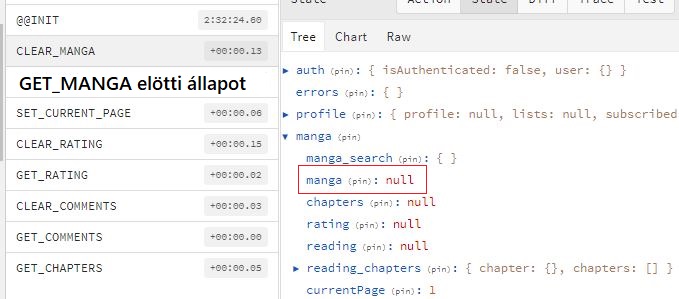
43. ábra: getManga metódus, ami lekéri egy manga részleteit és eltárolja azt a React Redux storeban

És ilyenkor sikeres kérés esetén a megfelelő reducer-ben (mangaReducer.js) lefut a GET\_MANGA switch eset, ami annyit csinál, hogy a manga állapotban a manga változó értékét a kapott adat (payload)-ra változtatja (44.ábra).

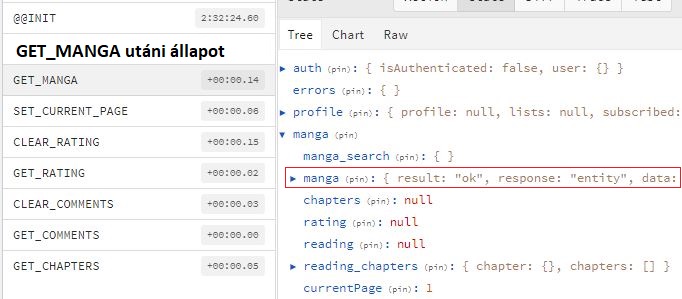


44. ábra: manga reducer felépítése

Következő képen (45.ábra) lehet látni a weboldal állapotát, most azt az állapotot látjuk mielőtt a GET\_MANGA típusú akció meghívódott volna. A manga reducer-ben nincs értéke a manga változónak, mert ez az alapállás.



45. ábra: A weboldal azon állapota, ami a GET\_MANGA hívás elött van



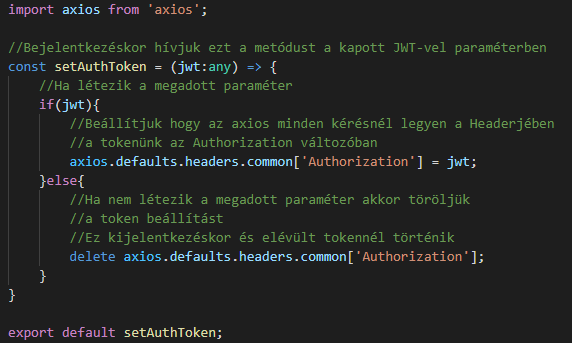
46. ábra: A weboldal azon állapota, ami a GET\_MANGA hívás után van

Miután meghívtuk a GET\_MANGA akciót (46.ábra), megváltoztatta a weboldal állapotát úgy, hogy a manga reducer-ben lévő manga változónak a kérésből beérkezett adatot állította be.

### Axios

Axios-t ~~ugye~~ HTTP kérések küldésére használom, de a backend szerveren vannak olyan végpontok, amik eléréséhez szükséges a kérés headerjében az Authorization változóban a JWT-t tárolni, amit a felhasználó kapott bejelentkezéskor, mert a védett végpontokon a backend megvizsgálja a kérésben lévő JWT-t és ha nincs vagy nem jó a token akkor a kérés nem sikerül.

Ezért az összes kéréshez hozzáteszem ezt a token-t, persze csak ha el van tárolva ez a token. Ezt úgy érem el, hogy sikeres bejelentkezéskor meghívok egy olyan függvényt, ami beállítja az axios-nak egy alap értékként (47.ábra).



47. ábra: Összes axios kérés header alap értékének megadása

### JWT tárolás

A sikeres bejelentkezés során kapott JWT-t a böngésző lokális tárhelyében tároljuk, kijelentkezésnél innen töröljük. A token-t minden akció esetén ellenőrizzük, hogy a token érvényességi ideje lejárt-e, ha lejárt a token, a felhasználót kijelentkeztetjük.



48. ábra: loginUser metódus ahol JWT tárolása, axios kérés beállítás és bejelentkezett profil beállítása történik

## Backend

### Struktúra

+api

| .env

| db\_tables.sql

| index.js

| package.json

| package-lock.json

+---node\_modules

+---src

| +---database

| | db.js

| +---helpers

| | checkIfFriends.js

| | checkIfHasMessageProperty.js

| | checkIfMangaExists.js

| | getMangaTitle.js

| | getMessageKey.js

| | getVisibility.js

| | isEmpty.js

| | sendEmail.js

| | verify.js

| | verify\_check.js

| +---queries

| +---validations

| +---routes

| | auth.js

| | chat.js

| | comments.js

| | lists.js

| | mangaUpdate.js

| | threads.js

| | user.js

| | votes.js

Struktúra magyarázat:

* **.env** fájl: Egy olyan fájl, amivel szimulálni tudjuk a környezeti változók jelenlétét, anélkül hogy a szervert futtató rendszeren ténylegesen beállítanánk, ez segít a futtatáshoz szükséges globális változókat egy helyre összegyűjteni.
* **Db\_tables.sql:** Olyan SQL utasításokat tartalmaz, amiket ha lefuttatunk az adatbázison, a feladat működéséhez szükséges táblákat, tárolt eljárásokat létrehozza és pár szükséges adatot is felvisz a rendszerbe.
* **Index.js:** A program fő fájlja, ebben a fájlban indítjuk el a szervert és adjuk meg a szervernek a beállításait.
* **Package.json:** A program meta adatait tartalmazza, mi a projekt neve, verzió szám, használt csomagok listája, futtatási scriptek és még sok más beállítás.
* **Package-lock.json:** Az összes használt csomagnak és azoknak a csomagok által felhasznált függőségeknek a verziószámát tárolja, hogy ha újra generálják a node\_modules mappát ne lépjen fel olyan hiba, hogy az egyik csomag által használt függőség máshogy működik egy frissítés miatt.
* **Node\_modules mappa:** A felhasznált csomagok találhatók, ez a mappa automatikusan kerül generálásra.
* **Src mappa:** Ebben a mappában találhatóak a programban használt kódok.
* **Database mappa (src/database):** Azt a fájlt tartalmazza (db.js) ami kapcsolatot létesít az adatbázissal és rajta keresztül tudunk SQL utasításokat lefuttatni az adatbázison.
* **Helpers mappa (src/helpers):** Olyan fájlok, találhatóak itt, amiket többször meghívunk a program futása során, vannak olyan fájlok, amik azonos céllal rendelkeznek, ezeket további mappákba csoportosítjuk (queries, validations).
* **Queries mappa (src/helpers/queries):** A fájlok osztályokat tartalmaznak, amik az adatbázissal kommunikálnak.
* **Validations mappa (src/helpers/validations):** Különböző adatok ellenőrzésére szolgáló funkciókat tartalmaznak a fájlok.
* **Routes mappa (src/routes):** Az API szerver végpontjai vannak itt, az azonos szerepkörű végpontok egy-egy fájlban vannak összeszedve.

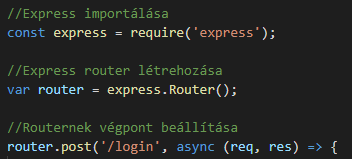
### Routeok

A szerver végpontjai 8 db csoportba vannak osztva, és ezekhez a csoportokhoz külön-külön tartozik: egy fájl, ami a HTTP kéréseket kezeli, egy fájl, ami az adat ellenőrzést kezeli és egy fájl, ami az adatbázissal kapcsolatos funkciókat tárolja.

Csoportok listája (ezek a fájlok az src/routes mappában találhatóak):

* **Hitelesítés (auth.js):** Felhasználó regisztrációját, bejelentkezését kezeli.
* **Csevegés (chat.js):** Felhasználók közötti csevegést kezeli.
* **Kommentek (comments.js):** Egy elemnek (Fórum, Manga, Manga fejezet) a kommentjeit kezeli, létrehozás, lekérdezés.
* **Manga listák (lists.js):** Felhasználó manga listáját kezeli.
* **Manga frissítések (mangaUpdates.js):** Manga frissítéseket kezel.
* **Fórum thread-ek (threads.js):** Fórummal kapcsolatos akciókat kezeli.
* **Felhasználó (user.js):** Felhasználó profiljával kapcsolatos kéréseket kezeli
* **Értékelés (votes.js):** Manga értékelést, kommentek és fórum thread-ek lájkolását kezeli.

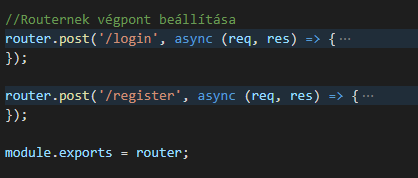
Ezekben a fájlokban egy Express.js Router objektumnak állítok be végpontokat. Azt, hogy ezeken a végpontokon milyen fajta kéréseket dolgozhat fel (a kérések fajtái a HTTP metódusok lehetnek, én leginkább a GET, POST, PUT és DELETE metódusokat használom) és azt hogy mit csináljon, ha erre a végpontra érkezik egy HTTP kérés.



49. ábra: végpont beállítása Express.js routeren

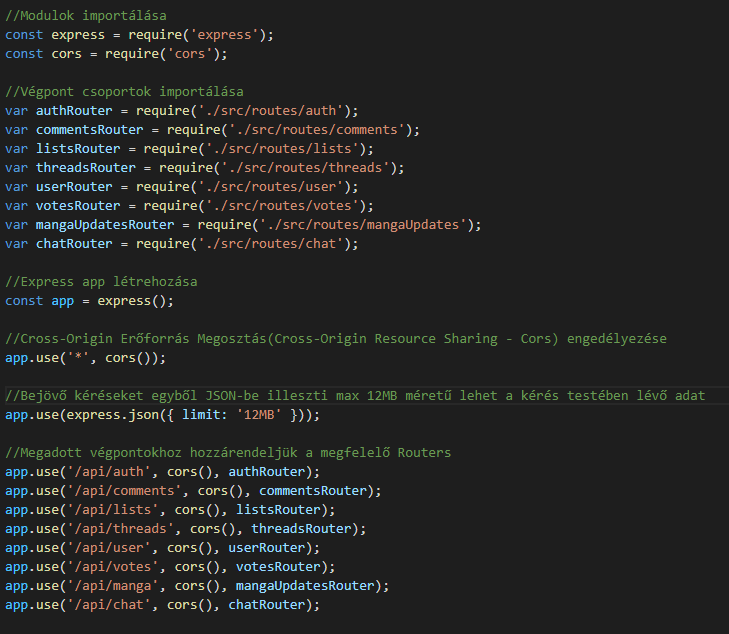
A fenti képen egy router létrehozását és annak egy végpont beállítását látjuk. A /login végpont egy POST kérés esetén lefuttatja a második paraméterben megadott funkciót, ami esetünkben egy anonim nyíl funkció, 2 paramétert kap, req (Request - Kérés) és res (Response - Válasz). A req változó a beérkező HTTP kérést tárolja, a res változó pedig a választ, a funkción belül a req változóból le tudjuk kérni a beérkező adatokat, a res változóba pedig megadom a várt adatokat és vissza küldöm a kérőnek, ha hiba történik, akkor azt jelezzük a válaszban.

Az összes végpont létrehozása után exportálom ezt a routert hogy a fő fájlban be tudjam importálni és megadjam a szervernek ezeket a végpontokat.



50. ábra: hitelesítéshez végpontok

Ez a fajta megoldás az összes végpont csoport fájljában így történik, és ezeket a modulokat importáljuk a fő fájlba (index.js) mert itt adom meg a szerver beállításait, ezt a modult (49. és 50. ábra) majd authRouter néven fogom importálni (51.ábra).



51. ábra: Express.js applikáció beállítása

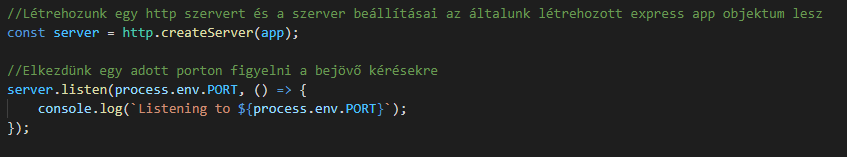
Az Express.js applikáció use() metódusa vár:

* 1 szöveg (string) paramétert, ami a végpont elérési útja és 1 vagy több callback függvényt vagy
* 1 vagy több callback függvényt

app.use([path,] callback [, callback...])

Abban az esetben, ha nincs a paraméterek között egy szöveg (string) ami az útvonalat jelenti, a megadott függvények listája minden kérésre lefut. Itt az importált routereket hozzárendeljük, végpontokhoz ez annyit csinál, hogy a szövegként (string) megadott útvonalhoz a routerben megadott végpontok hozzáfűződnek, vagyis az authRouter-ben lévő /login és /register végpontot majd a <http://server-address:3001/api/auth/login> és a <http://server-address:3001/api/auth/register> URL címen tudom elérni.

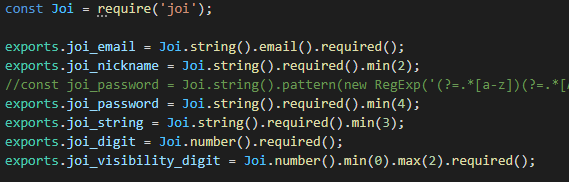
Miután megadtam az Express.js applikációnak a beállításait, létrehozok egy HTTP szervert egy beépített Node.js modullal (HTTP modul) és szerver beállításnak megadom az Express.js applikációt, majd elindítom a szervert és megadom neki, hogy milyen port-on figyeljen bejövő kérésekre.



52. ábra: backend szerver létrehozása és elindítása

### Ellenőrzés

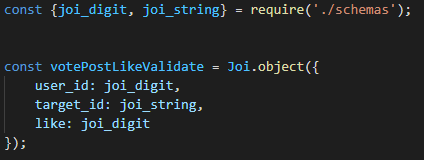
Minden végpontcsoporthoz külön ellenőrzés fájl tartozik és ebben a fájlban tárolom a sémákat és az ellenőrző függvényeket. Ezek a fájlok úgy néznek ki pontosan, hogy egy fő séma fájlból importálok úgymond primitív sémákat, amik egy-egy érték kivizsgálására jók.



53. ábra: skaláris sémák definiálása

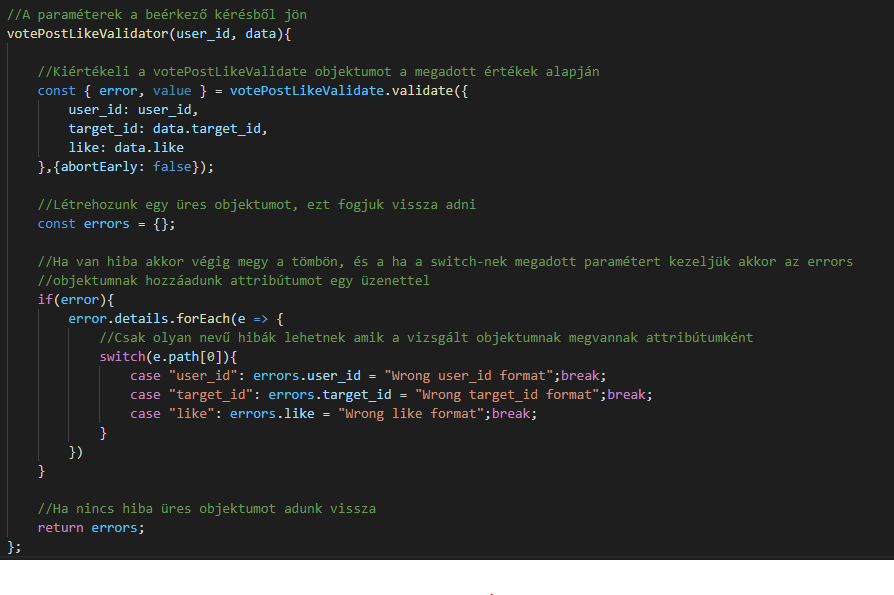
Ezek a sémák az src/helpers/validations mappában találhatók a schemes.js fájlban, ezeket a fájlon belül exportálom egyesével, hogy az ellenőrzés során csak azokat kelljen importálni, amikre szükségem van.

Az ellenőrző fájlban importáljuk a szükséges sémákat és létrehozunk olyan objektumokat, amik az egyes végpontok beérkezett adatait kapja meg és vizsgálja. (46.ábra)



54. ábra: joi objektum létrehozása

Majd ugyanabban a fájlban egy osztályt hozok létre, amikben ellenőrző függvényeket definiálok, amik kiértékelik az adatokat, és hibák esetén egy errors objektumba hozzáadja a hibákat egyedi hiba üzenettel. (55. ábra)



55. ábra: adatellenőrző függvény megvalósítása

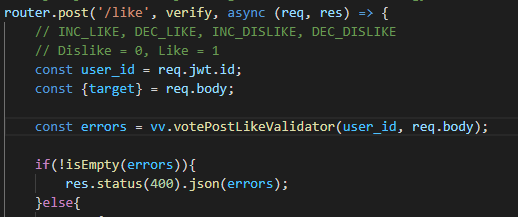
Egy ilyen ellenőrző osztálynak az összes végpontra különböző ellenőrző funkciói vannak a hibakezelés miatt.

Ezt az osztályt a megfelelő végpont fájlban importáljuk.



56. ábra: adatellenőrző osztály importálása

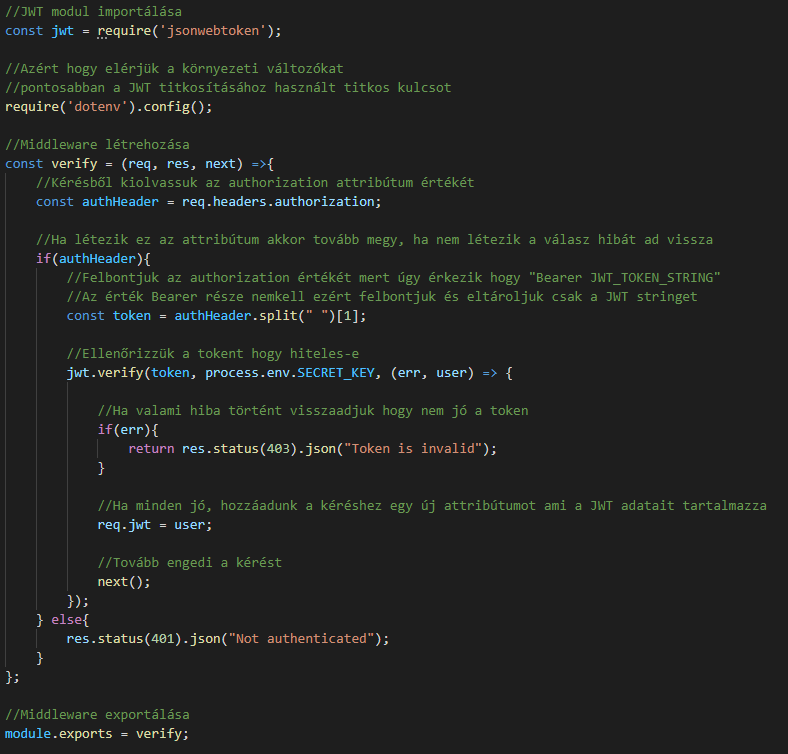
Megfelelő helyen meghívjuk a megfelelő függvényt és eltároljuk az eredményt, amit később megvizsgálunk, hogy üres-e vagy nem, ha nem üres, akkor egyből visszaadok egy választ, hogy hiba történt és adatként adom az ellenorzás során kapott hiba objektumot, ha nincs hiba, akkor folytatódik tovább a program.



57. ábra: adatellenőrző függvény hívása és visszatérési érték ellenőrzése

### Hitelesítés

A hitelesítést JSON Web Token-el történik, amit a kliens bejelentkezésnél megkap és eltárol. Ezt a JWT-t a kérések header-jében küldi át Authorization attribútum értékeként. Ennek a JWT ellenőrzésére és hitelesítésére egy hitelesítő middleware-t használok.



58. ábra: hitelesítés middleware

Az exportált modult úgy lehet használni, hogy az ellenőrizni kívánt végpontokhoz hozzáadom ezt a middleware-t és utána azt a végpontot csak létező és hiteles JWT jelenlétében tudja használni.

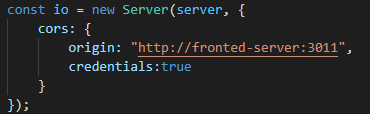
Először meghívom a hitelesítés middleware-t (paraméter listában ez az első metódus, ami lefut), ha nem sikeres a hitelesítés, akkor hibát ad vissza a kérőnek, ha sikeres, akkor a middleware-ben lévő next() metódussal tovább megyünk a paraméter listában lévő következő metódusra.



59. ábra: hitelesítés middleware használata végponton

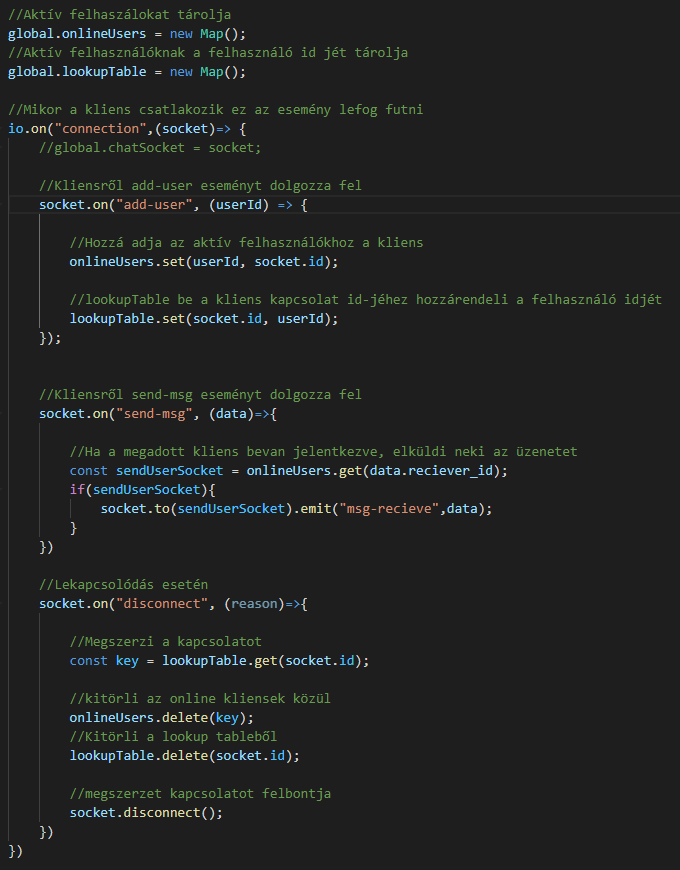
### Csevegés

Valós idejű kommunikációhoz a Socket.IO Node.js modult használom, létrehozok egy Socket.IO szerver objektumot megadom neki a backend által használt szerver objektumot és a Socket.IO szerver beállításait például hogy honnan várjuk a csatlakozásokat és eseményeket.



60. ábra: Socket.IO szerver létrehozása

Globálisan tárolom az aktív kapcsolatokat, és egy lookup table-t, ami kapcsolat id-khez hozzárendeli a megfelelő felhasználó id-t, ezt a kapcsolat bontáskor használom fel. Majd megadom a Socket.IO szervernek, hogy milyen eseményekre figyeljen, és mit csináljon, ha bekövetkeznek ezek az események. (61.ábra)



61. ábra: backend Socket.IO által használt események

### Adatbázis kapcsolat

A szerver és az adatbázis közötti kommunikációt egy mysql nevű Node.js csomag biztosítja, ami egy MySQL driver, de lehet MariaDB re is csatlakozni vele.

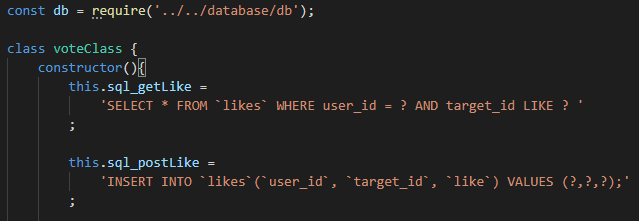
Az adatbázissal való kapcsolatot egy külön fájlban (db.js) létesítem, amit exportálok és azokban a fájlokban importálom, amikben szükség van a kapcsolatra.



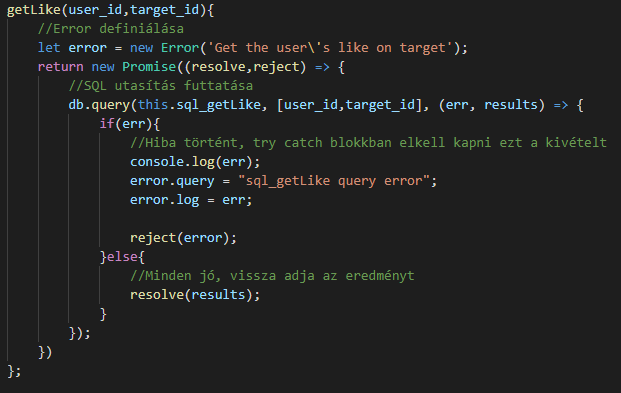
62. ábra: adatbázis kapcsolat létrehozása, exportálása

Környezeti változókból beolvasom a csatlakozáshoz szükséges adatokat és csatlakozok az adatbázishoz, ha nem sikerül hibát dob (62.ábra). Ezt a kapcsolatot fogják használni azok a fájlok, amik az adatbázissal, beszélnek.

A queries fájlok osztályt definiálnak, amik SQL utasításokat tartalmaznak és függvényeket, amik elvégzik az SQL utasítást és visszaadják annak az eredményét.



63. ábra: adatbázis kapcsolat importálása, adatbázissal kommunikáló osztály SQL utasítása



64. ábra: adatbázissal kommunikáló osztály függvénye

Ilyen fajta függvényeket tartalmaznak az osztályok, ezek a függvények a Javascript Promise API-t használják, hogy meg tudjam várni a fő programban az aszinkron függvény eredményét. Az adatbázis utasítás (db.query()) a Promise függvényben fog lefutni. A db.query() metódusnak azt paraméterezését használom, hogy első paraméter egy SQL szöveg (string), második paraméter egy tömb, ami az SQL szövegben (string) lévő kérdőjelek helyére teszi az elemeit eszképelve, harmadik paraméter egy callback függvény, ami visszaadja a futtatás hibáját vagy eredményét. Gyors hiba ellenőrzés után, ha van hiba, akkor a függvény hibát dob a reject() függvénnyel, ha nincs hiba, akkor visszaadja az eredményt a resolve() függvénnyel.

A fájl végén ezt az osztályt exportálom és a megfelelő fájlban importálom.



65. ábra: adatbázissal kommunikáló osztály importálása

És a megfelelő helyen meghívóm a megfelelő függvényt egy try catch blokkon belül és mivel a függvény a Promise API-t használja, az aszinkron művelet eredményét meg tudom várni az await kulcsszóval.

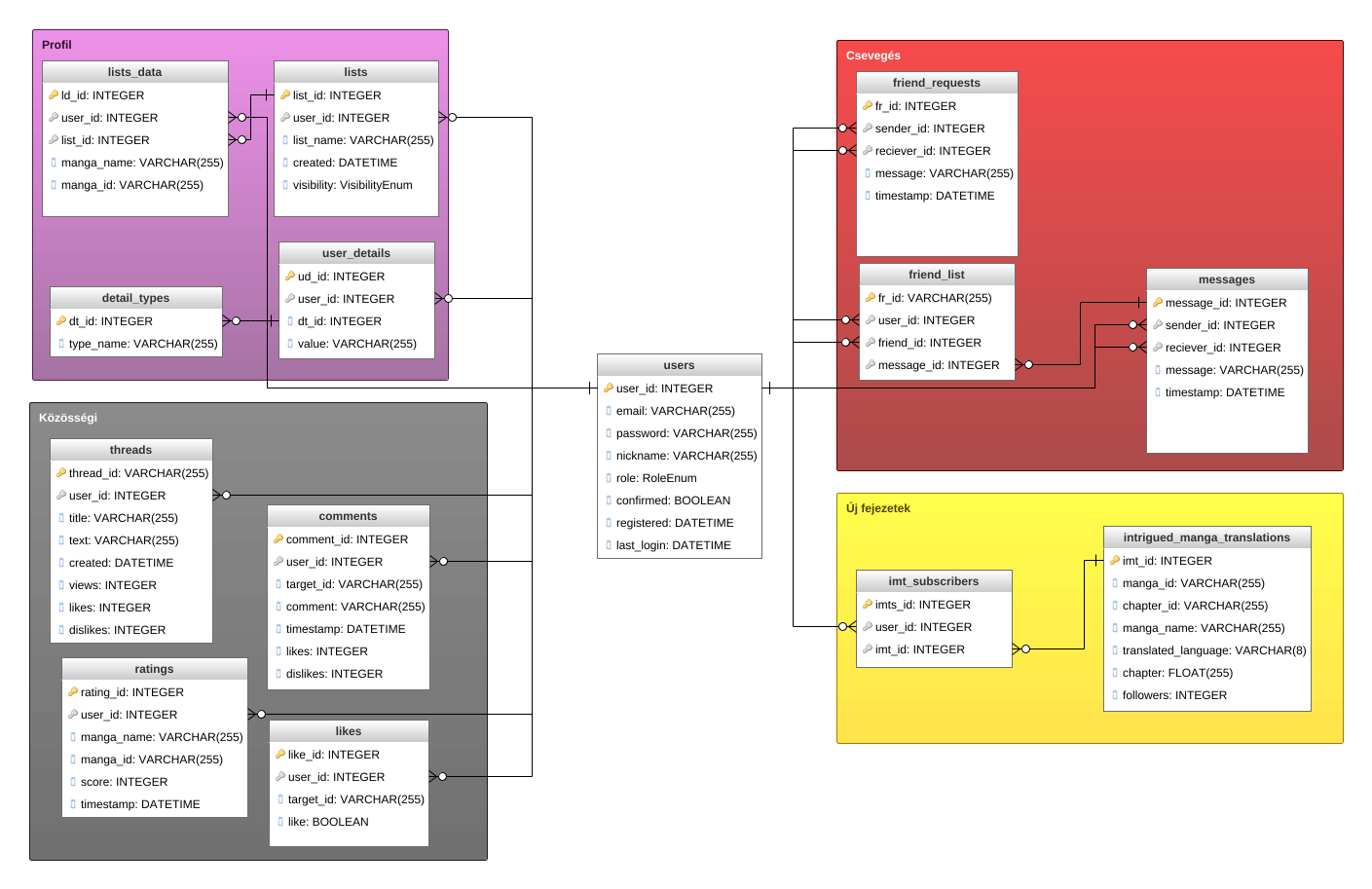


66. ábra: adatbázissal kommunikáló osztály használata

## Adatbázis

### Struktúra

A táblák legtöbbször a users táblának a user\_id-jára referálnak, mint külső kulcs, mert szükség van a legtöbb adat azonosítására tulaj szerint.



67. ábra: adatbázis felépítése

* **Profil csoport (Lila/Rózsaszín)**: Azok a táblák, amik a felhasználó profiljához tartoznak ilyen például a manga listák, és profil részletek (kedvelt, nem kedvelt mangák és profil leírás).
* **Csevegés csoport (Vörös):** Valós idejű csevegéshez szükséges táblák (barátlista, üzenetek és barát felkérések).
* **Közösségi csoport (Szürke):** Olyan táblák, ami más emberekkel való interaktálást biztosítja (fórumok, kommentek, lájkok és értékelések).
* **Új fejezetek csoport (Sárga):** Feliratkozott manga fordításokat, és ezekre a fordításokra feliratkozott felhasználókat tárolom.

### Adatbázis táblák

Egy gyors magyarázat az adatbázis táblákhoz:

* **Users**: Tartalmazza a felhasználó fő adatait, amik szükségesek az alap profilhoz, regisztrációhoz. A confirmed boolean változó azt jelenti, hogy megerősítette-e a regisztrációját a felhasználó, ha false akkor nem tud bejelentkezni.
* **User\_details**: Olyan információkat tartalmaz a felhasználóról, amik a profilján megjelennek, leírás saját magáról, kedvelt és nem kedvelt mangák. Egy segéd táblát használva (detail\_types) több azonos kategóriába tartozó rekordot is fel tudunk venni egy felhasználóhoz, ilyen a kedvelt manga „típus” mert egy emberhez több olyan mangát rendelhetünk, amit kedvel és ennek az attribútumnak (kedvelt manga) az id-ját tárolja el, mint idegen kulcs.
* **Detail\_types**: A létező felhasználói adatok típusát tárolja, én a profil leírás, kedvelt, nem kedvelt mangák típust tárolom, de használhatunk plusz típusokat is pl.: allergia és akkor a user\_details táblában egy felhasználóhoz több allergiát is tudunk rendelni, ami mindegyik 1-1 rekord.
* **Ratings**: Azt tárolja, hogy ki, mikor milyen mangát mennyire értékelt egy 1-től 10-es skálán.
* **Likes**: Létező lájkokat, diszlájkokat tárol. A target\_id a kommentek és thread-ek id-ját tárolja, amihez az adott felhasználó lájk vagy diszlájkja tartozik.
* **Threads**: Tartalmazza, hogy ki hozta létre a fórum thread-et, mi a címe, a szövege, mikor hozták létre, mennyien nézték meg és a lájkok, diszlájkok számát.
* **Lists**: A felhasználónak a listáit tartalmazza, tárolja a tulajt, mi a lista neve, mikor hozta létre, és a láthatóságát, ami lehet publikus, privát vagy barátoknak publikus.
* **Lists\_data**: Ez a listákhoz tartozó mangákat tartalmazza.
* **Friend\_requests**: Az elküldött barát felkéréseket tartalmazza, ki küldte, kinek küldte, mikor és egy üzenetet tárol.
* **Comments**: Tárolja a komment tulaját, mihez tartozik ez a komment (fórum thread, manga, manga fejezet), a komment tartalmát, mikor hozták létre, a lájkok és diszlájkok számát.
* **Friend\_list**: Felhasználóknak a barátait és az utolsó elküldött üzenetet is tárolja.
* **Messages**: 2 felhasználó között történő üzeneteket tárolja.
* **Intrigued\_manga\_translations**: Azokat a mangákat tartalmazza, amikre feliratkozott valaki, hogy kapjon értesítést az új fejezetekről. Különböző nyelvű fejezet fordításokat is követhetnek az emberek. Tárolja a manga id-ját, utolsó fejezet id-ját, a fordítás nyelvét, a manga nevét a fordítás nyelvén (angol fordításhoz angol cím), a fejezet számát és azt, hogy hányan vannak feliratkozva erre a fordításra.
* **Imt\_subscribers**: Felhasználóknak a feliratkozásait tárolja.

### Tárolt eljárások

Tárolt eljárások listája és magyarázata:

* **Sp\_manage\_likes**: 2 bemeneti paramétere van és 1 kimeneti paraméter. Az eljárás megkap egy komment id-t és egy művelet szöveget (string) ami lehet INC\_LIKE, DEC\_LIKE, INC\_DISLIKE, DEC\_DISLIKE és egy SQL case statement végrehajtja a megfelelő műveletet a kommenten (lájk/diszlájk növelés/csökkentés). A végén visszaadja a kimeneti paraméteren keresztül az eredményt, ami lehet 0 = SQL hiba, 1 = Sikeres változtatás, 2 = Nem lehet 0 alá csökkenteni a lájk, diszlájk számot, 3 = Nem megfelelő művelet szöveget (string) kapott.
* **Sp\_manage\_likes\_thread**: Ugyan az, mint az sp\_manage\_likes, de a fórum thread-ek lájkját kezeli.
* **Sp\_manage\_manga\_subscribers**: 2 bemeneti paramétere van és 1 kimeneti paraméter. Az eljárás megkap egy intrigued\_manga\_translations id-t és egy művelet szöveget (string) ami lehet INC vagy DEC és egy SQL case statement végrehajtja a megfelelő műveletet a figyelt manga fordításon (feliratkozó növelés/csökkentés és manga fordítás törlése). A végén visszaadja a kimeneti paraméteren keresztül az eredményt, ami lehet 0 = SQL hiba, 1 = Sikeres változtatás, 2 = Kitörölte a manga fordításfigyelés rekordot, 3 = Nem megfelelő művelet szöveget (string) kapott.

## Proxy

A proxy szerver egy Node.Js Express.Js API szerver, ami a http-proxy-middleware segítségével az API-t kiegészíti proxy szerver készítését elősegítő funkciókkal.

### Struktúra

+proxy

+---node\_modules

+---public

| index.js

| package.json

| package-lock.json

Magyarázat:

* **Node\_modules mappa:** A felhasznált csomagok találhatók, ez a mappa automatikusan kerül generálásra.
* **Public mappa:** Itt találhatóak, azok a képek, amit a MangaDex-ről letölt a proxy és a frontendnek szolgáltatja.
* **Index.js:** A program fő fájlja, itt indul a program és végzi el a megfelelő műveleteket a beérkező kéréseknek.
* **Package.json:** A program meta adatait tartalmazza, mi a projekt neve, verzió szám, használt csomagok listája, futtatási scriptek és még sok más beállítás.
* **Package-lock.json:** Az összes használt csomagnak és azoknak a csomagok által felhasznált függőségeknek a verziószámát tárolja, hogy ha újra generálják a node\_modules mappát ne lépjen fel olyan hiba, hogy az egyik csomag által használt függőség máshogy működik egy frissítés miatt.

### MangaDex Átirányítás

A lényeg az, hogy létrehozunk egy proxy-t a http-proxy-middleware segítségével. A megfelelő beállításokat a proxy létrehozásánál tudjuk beállítani.



68. ábra: proxy szerver beállítása és elindítása

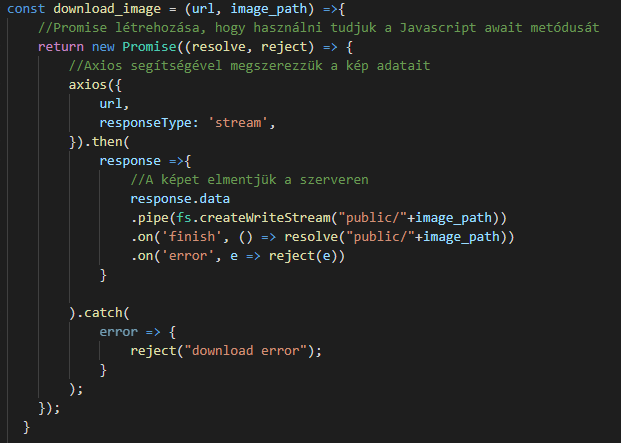
### MangaDex Képek letöltése

Az oldalon 2 fajta kép jelenik meg, borítókép és Manga fejezetnek az oldalai. A frontendről ennek a két képnek az elérési helyét két külön végponttól kapják meg. A borítóképért a <http://proxy-address:3000/img/:id/:filename> címre, a manga oldalért a <http://proxy-address:3000/chapter/:hash/:filename> címre küldött GET kéréssel lehet lekérni a kép helyét. Az URL címekben lévő :id, :filename, :hash elérési utakba bármilyen adat lehet, de a sikeres kép megszerzéséhez a MangaDex által szolgáltatott képek id-ja vagy hash-kódja és a hozzájuk tartozó fájlneveket adjuk meg. Akkor a proxy szerver ellenőrzi, hogy már letöltötte-e azt a fájlt, ha igen, akkor visszaadja a szerveren lévő elérési útját, ha nincs ilyen fájl, akkor letölti a MangaDex szerveréről, és letöltés után adja vissza az elérési utat, amit a frontend megad az <img> HTML -el tagek forrásának.



69. ábra: proxy szerveren kép elérési útjának lekérése, kép letöltése

A manga oldal kép végpont is ugyan ilyen, csak a MangaDex szerveren lévő elérési címe kicsit más.



70. ábra: kép letöltése adott címről

Ez a Javascript funkció pedig egy aszinkron funkció, ami a megadott URL címen lévő képet letölti a megadott mappába a megadott fájlnévvel.

### Letöltött képek szolgáltatása

Ahhoz hogy az Express.js szerverről elérjék a képeket, meg kell adni az Express.js szervernek, hogy milyen mappából szolgáljon ki statikus tartalmat (képet).



71. ábra: proxy szerver statikus tartalom szolgáltatására beállítás

Ez a beállítás annyit csinál, hogy a public mappában lévő fájlokat ellehet érni egy URL címmel például a <http://proxy-address:3000/image.jpg> URL megjeleníti a public mappában lévő image.jpg fájlt a böngészőben, ha létezik.

# Tesztek

## Backend

A végpontok tesztelését a postman.com-on keresztül végeztem, erről az oldalról könnyen lehet HTTP kéréseket kezdeményezni, minden végpontra miután elkészült küldtem jó, hibás és hiányos kéréseket.

## Frontend

Frontend tesztelését manuálisan végeztem, hibás, hiányos és jó adatok megadásával. Black box test jelleggel.

# Továbbfejlesztési lehetőségek

* Közös valós idejű manga fejezet olvasás.
* Fórumok, chat üzenetek, barátlista paginálása.
* További szociális funkciók hozzáadása (profil státuszok).
* Felhasználói felület szépítése.
* Frontend akadálymentesítése.
* Admin, moderátor szerep.

# Összefoglalás

A végeredmény egy közösségi weboldal lett, ami a mangák (japán képregények) után érdeklődőket célozza. A program külső API-t (MangaDex API) használ a mangák szerzésére, a felhasználói felület React (Javascript library) segítségével készült, a backend pedig Javascript-ben íródott Node.js környezetben. Az program készítése elején megvolt, hogy mit akartam létrehozni, de az hogy ezt hogyan vigyem végbe nem, ezért a program struktúrája, rétegei folyton változtak, ahogy kísérleteztem, hogy milyen workflow-ot kedvelek, tartok jónak, de a végére már kialakult egy olyan struktúra és workflow, amivel jobban átláttam a programot és az új funkciókat is könnyebben tudtam megvalósítani. Úgyhogy legközelebbi projektemnél biztosan több időt fektetek az alapok definiálására, hogy megkíméljem magamat a későbbi zűrzavartól.

Sok lehetőség van tovább fejleszteni ezt a programot, főleg a közösségi funkciók terén, például más kommentekhez hozzászólni, amik egy komment láncot alkothatnak, multimédia fájlok használata kommentekben, chat rulett szerű ismerkedés vagy akár valós idejű közös manga olvasást is lehetne implementálni. De a közösségi funkciók implementálásán kívül lehetne moderációt bevezetni, akár manuálist vagy automatikust, meg az oldal dizájnjának a szépítése is egy opció.

Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | „NodeJS About Page,” [Online]. Available: https://nodejs.org/en/about/. [Hozzáférés dátuma: 21 04 2022]. |
| [2] | S. Somani, „NodeJs Event Loop Architecture,” [Online]. Available: https://www.c-sharpcorner.com/article/node-js-event-loop/. [Hozzáférés dátuma: 21 04 2022]. |
| [3] | „Socket.IO Introduction,” [Online]. Available: https://socket.io/docs/v4/. [Hozzáférés dátuma: 21 04 2022]. |
| [4] | „MariaDB and MySQL difference,” [Online]. Available: https://mariadb.com/database-topics/mariadb-vs-mysql/. [Hozzáférés dátuma: 21 04 2022]. |
| [5] | M. Yumak, „How To Use React Redux,” [Online]. Available: https://itnext.io/how-to-use-redux-with-react-143de57d0bab. [Hozzáférés dátuma: 21 04 2022]. |

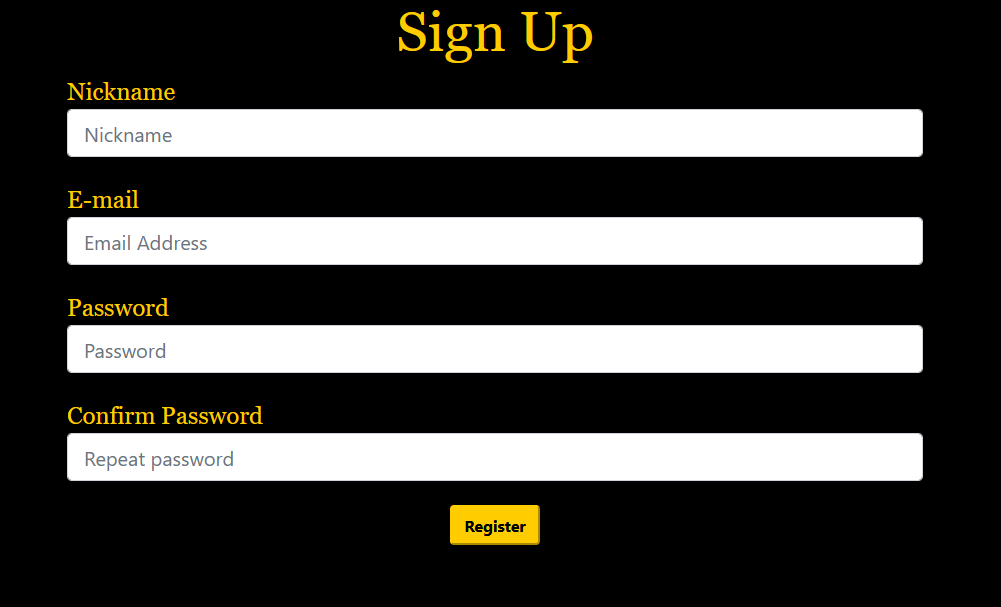
Mellékletek

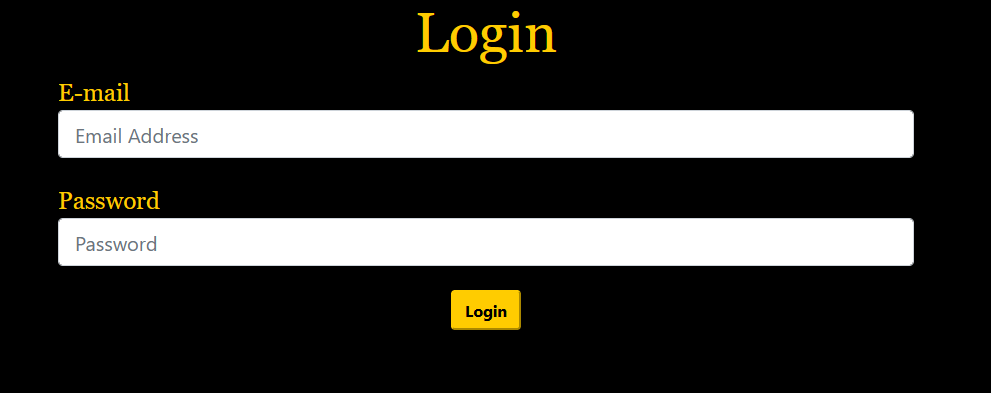
# Felhasználói útmutató

## Regisztráció, bejelentkezés

A mindenhol megjelenő navigációs sávon, ha a felhasználó nincs bejelentkezve, a jobb oldalon megjelenik egy „Sign Up” és „Login” felirat, amire ha rákattint, a megfelelő oldalra kerül.

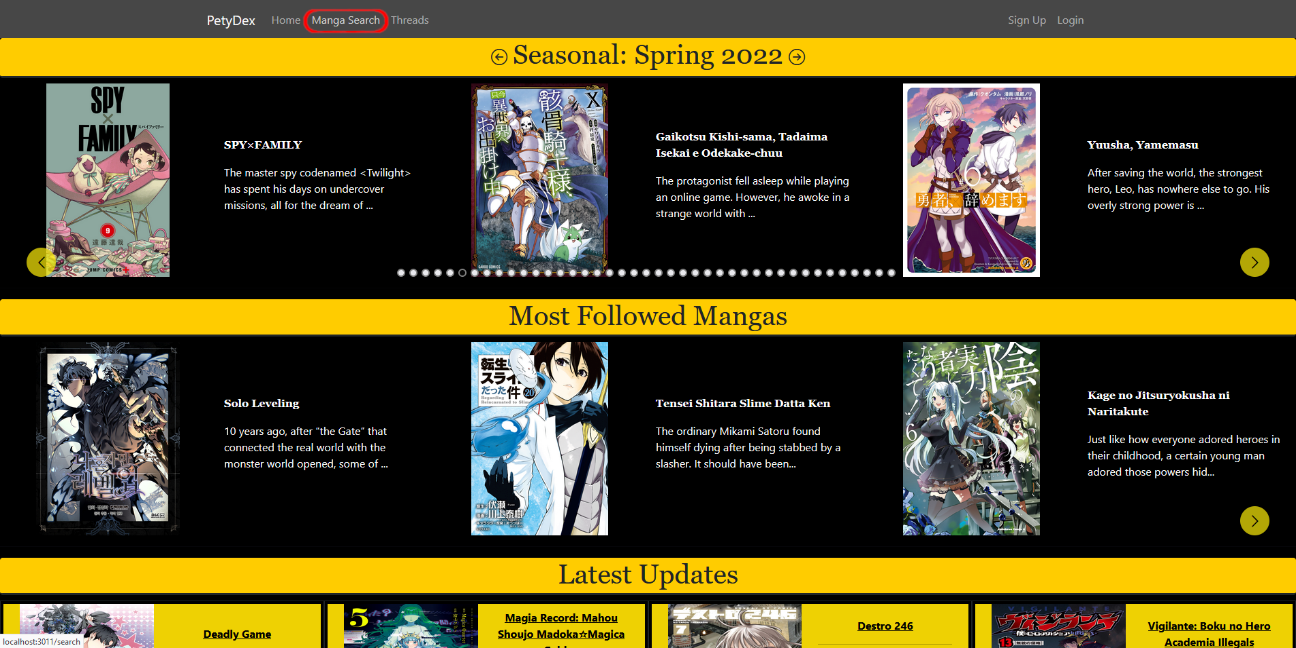






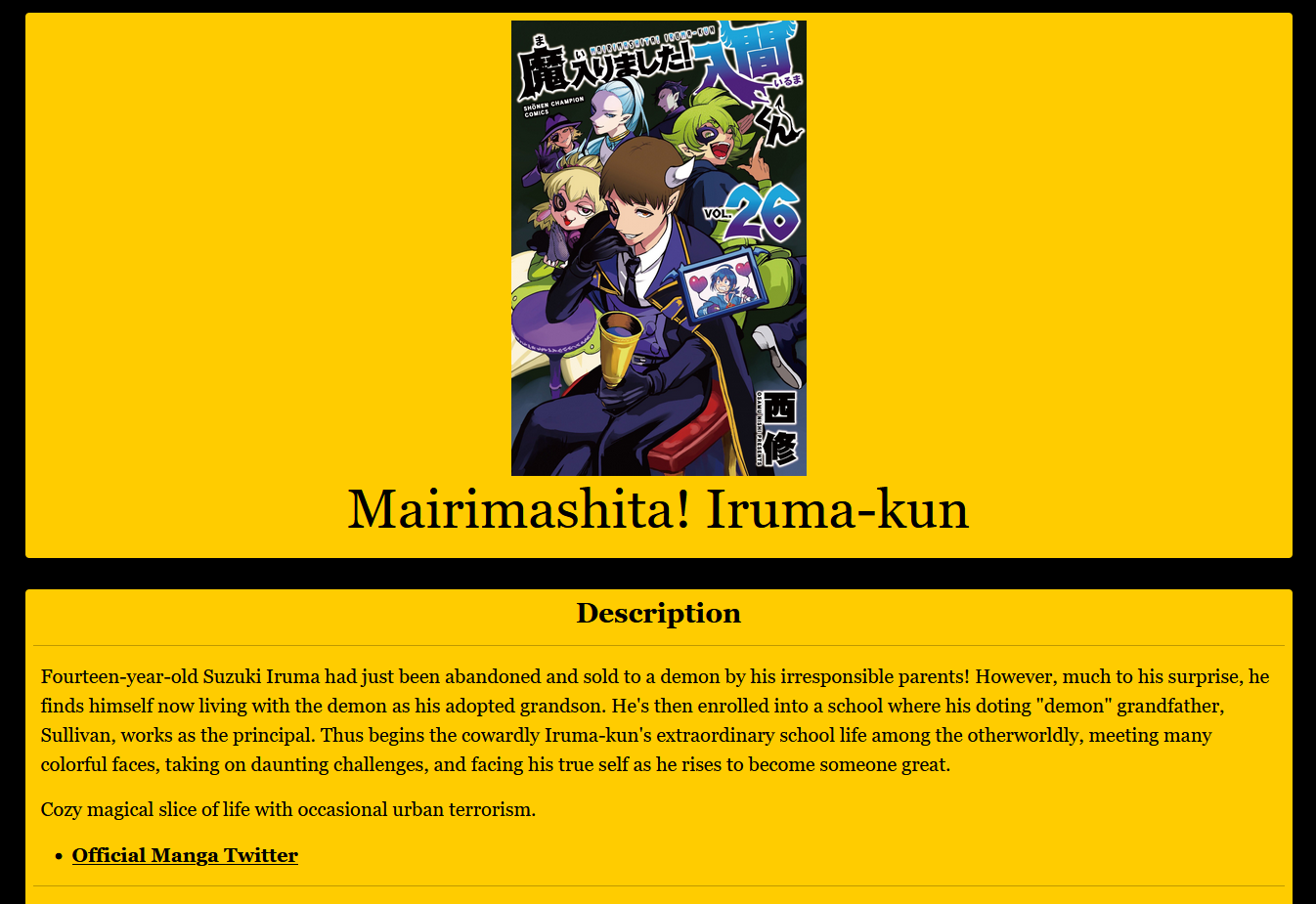
## Manga keresése és olvasása

A mindenhol megjelenő navigációs sávon a „Manga Search” feliratra kattintva a felhasználó a kereső felületre kerül. Itt a beviteli mezőbe Manga cím beírásával lehet keresni és létező találatok esetén megjelennek, amikre kattintva átkerül a felhasználó a Manga oldalára.









A manga oldal aljára görgetve, megjelennek a megnézhető fejezetek.



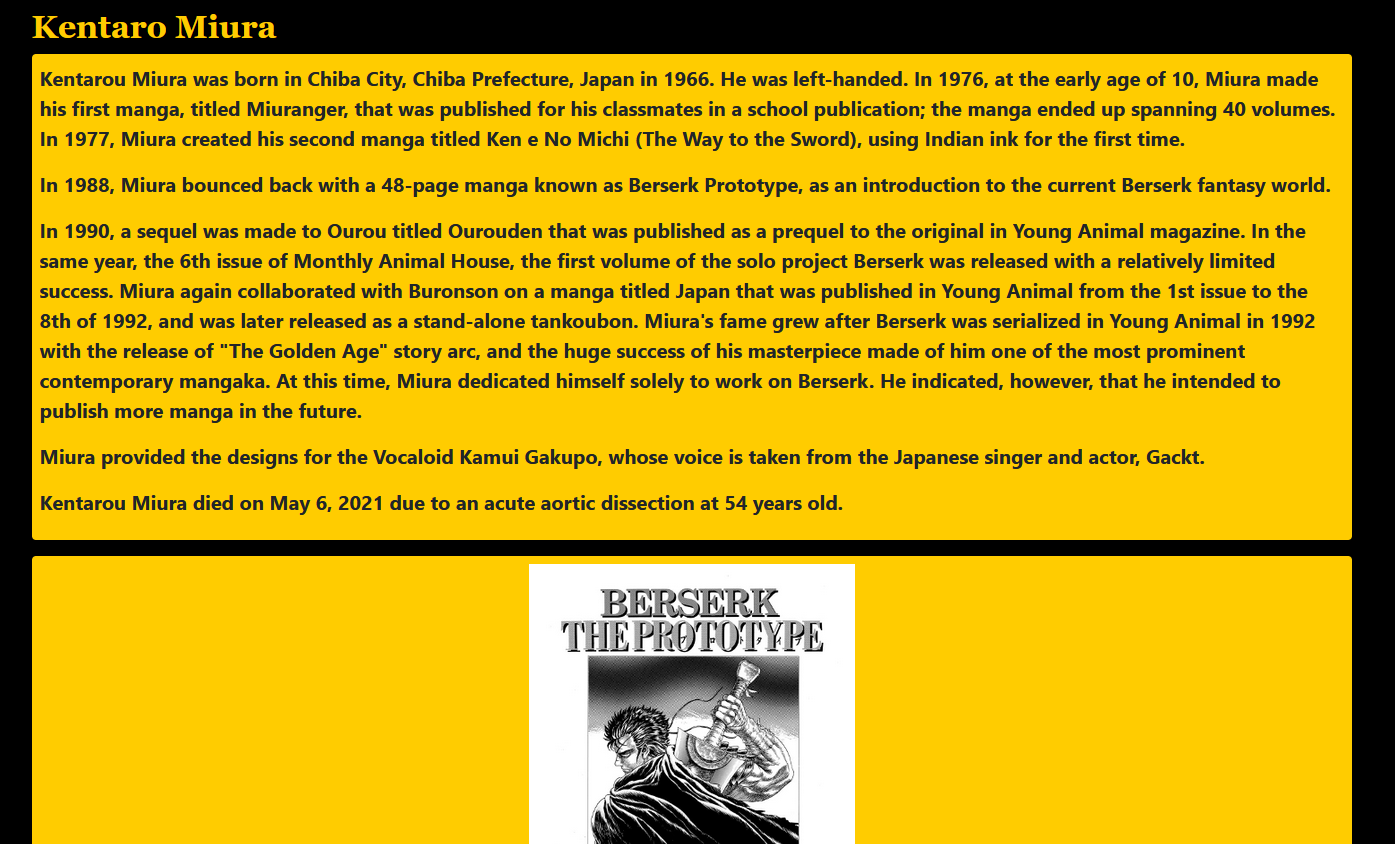
Rákattintva egy fejezet címére a felhasználó átkerül az olvasó felületre, ahol görgetéssel, vagy a nyilak használatával lehet tovább menni.



## Készítők és fordítók megtekintése

A manga készítői a manga oldalán találhatóak a „Creators” kártyában, itt a nevekre kattintva átmegy a felhasználó a megfelelő készítő oldalára.





A fordítókat meg a fejezeteknél lehet megnézni, és a fordító csapatra rákattintva átviszi a felhasználót a fordító csoport oldalára.





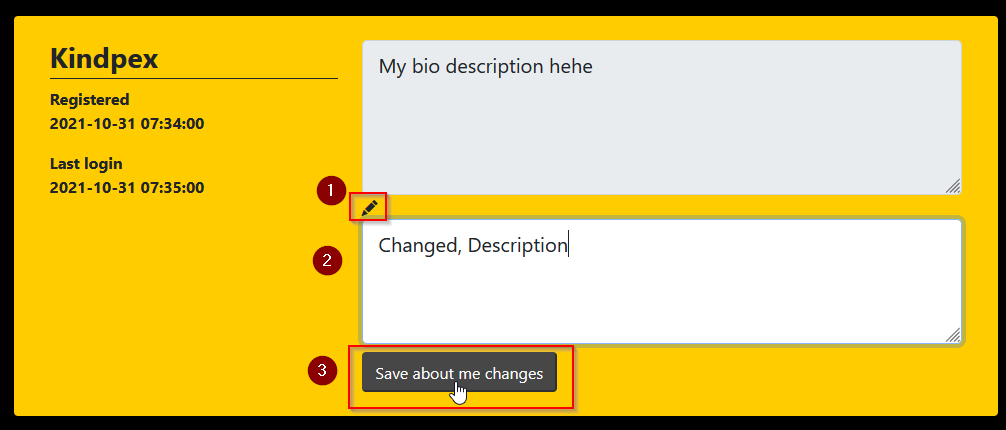
## Profil változtatása

Profil változtatásra bejelentkezés után van lehetőség. A navigációs sávon, a jobb oldalon lévő felhasználónévre kattintva átkerül a felhasználó a saját profiljára.



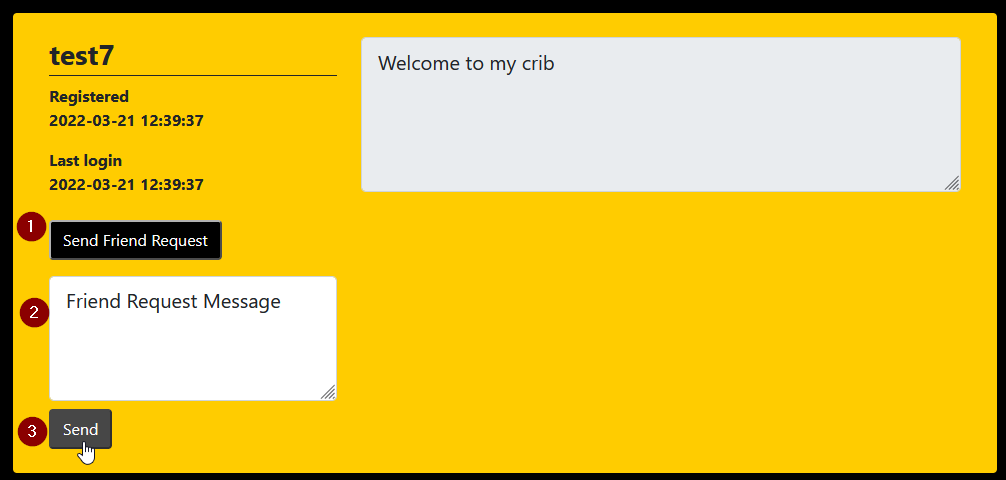
### Leírás változtatása

A profilnál a kis ceruzára kattintva megjelenik egy beviteli mező, ahol változtathatunk a saját profil leírásunkon, majd mikor végeztünk a „Save about me changes” gombra kattintva, ez elmentődik.



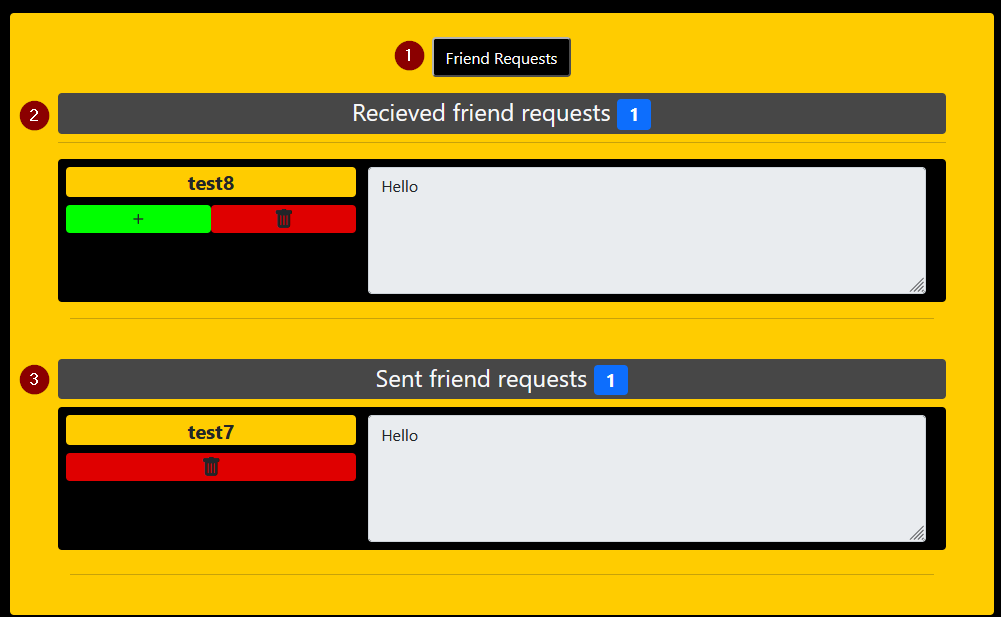
### Barát kérések kezelése

Másoknak barátfelkérést az ő profiljukon lehet küldeni.



A profiljukon megjelenő „Send Friend Request” gombra kattintva egy beviteli mező megjelenik, amiben tudunk egy kis üzenetet küldeni, majd a „Send” gombra kattintva a barátkérelem elküldésre kerül.

Maguk a barátkérések a saját profilunkon jelennek meg. Ezeket a „Friend Requests” gomb nyomásával, illetve utána a megfelelő címre kattintva tekinthetjük meg. A plusz jelre kattintva elfogadjuk a barát felkérést, a kuka ikonra kattintva visszautasítjuk vagy töröljük a felkérést.

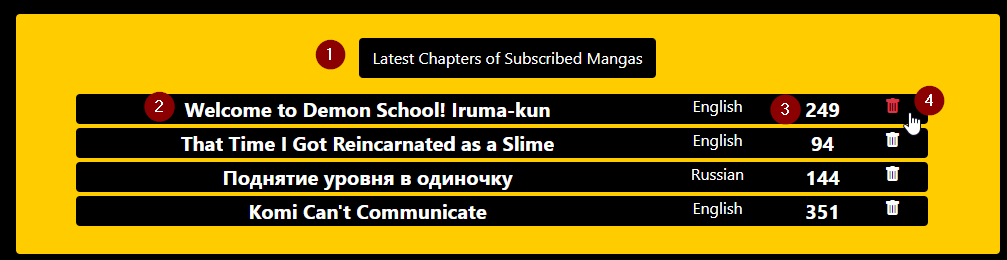


### Új fejezet feliratkozások kezelése

Feliratkozni fordításokra úgy lehet, hogy a manga oldalán megjelenő „Subscribe to get email of new chapters” gombra kattintunk, ott megjelennek az elérhető fordítások, és a plusz jelre kattintva feliratkozik az adott fordításra.

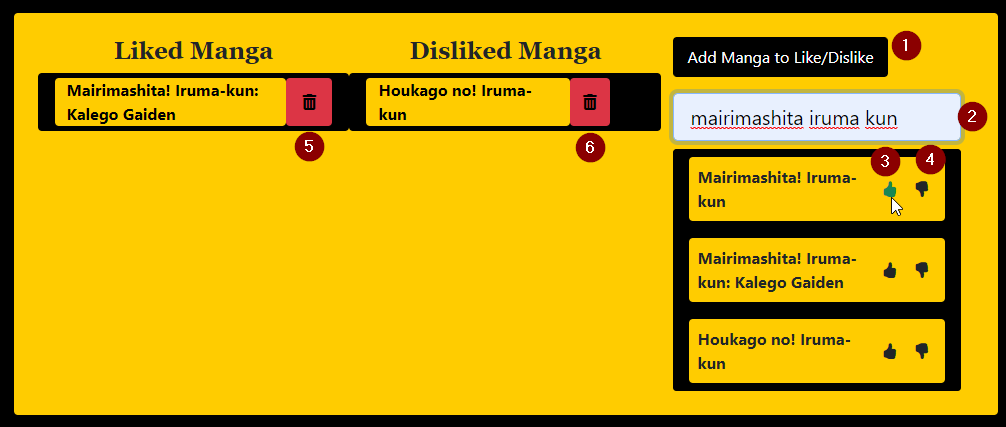


Leiratkozni a felhasználó profiljában lehet, a „Latest Chapters of Subscribed Mangas” gombra kattintva megjelenik az összes feliratkozás, itt a manga címére kattintva a felhasználó a manga oldalára kerül, a fejezet számra kattintva egyből a fejezetet kezdi el olvasni, a kuka ikonra kattintva pedig törli a feliratkozást.



### Kedvelt, nem kedvelt mangák kezelése

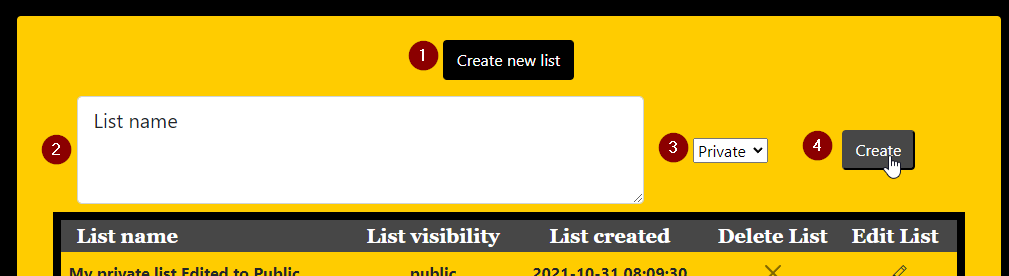
A profilnál a „Liked Manga” és „Disliked Manga” kártyánál, az „Add Manga to Like/Dislike” gombra kattintva megjelenik egy beviteli mező, itt kereshetünk mangákra, majd a lájk vagy diszlájk ikonra kattintva beteszi a megfelelő listába. Ezeket a lista tételeket lehet törölni a kuka ikonra kattintva.



### Manga lista kezelése

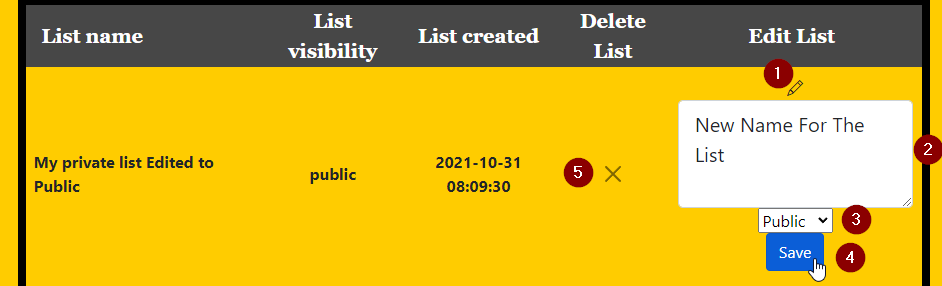
#### Lista létrehozása

A „Create new list” gombra kattintva megjelenik egy beviteli mező (amibe a lista neve megy), egy legördülő lista (ahol kiválaszthatjuk, hogy a lista milyen láthatóságú legyen), és egy „Create” gomb (amivel létrehozzuk a listát).



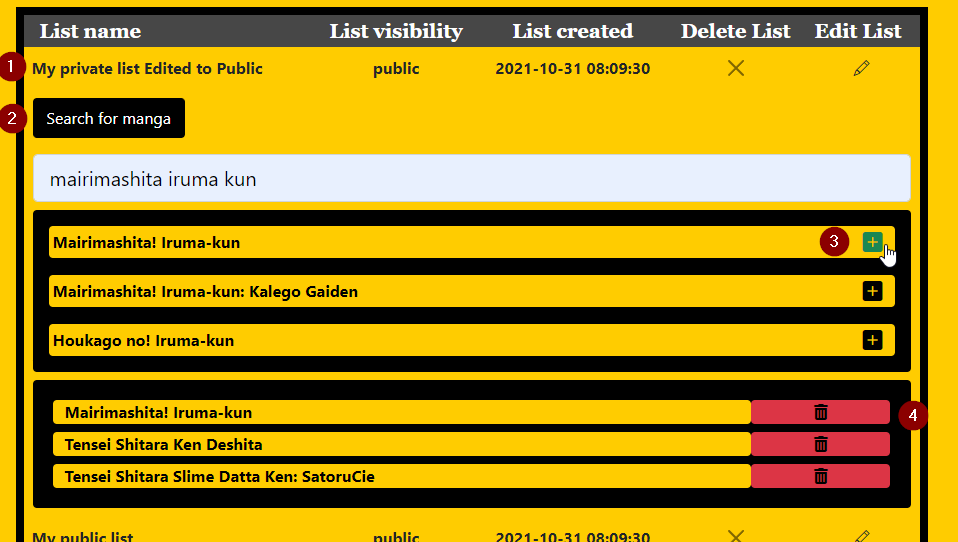
#### Lista változtatása, törlése

A lista sorában lévő ceruzára kattintva megjelenik egy beviteli mező (amibe a lista nevét írjuk), egy legördülő lista (ahol kiválaszthatjuk, hogy a lista milyen láthatóságú legyen), és egy „Save” gomb (amivel elmentjük a lista változtatásait). A lista sorában lévő X ikonnal pedig töröljük a listát.



#### Listához manga hozzáadás, törlése

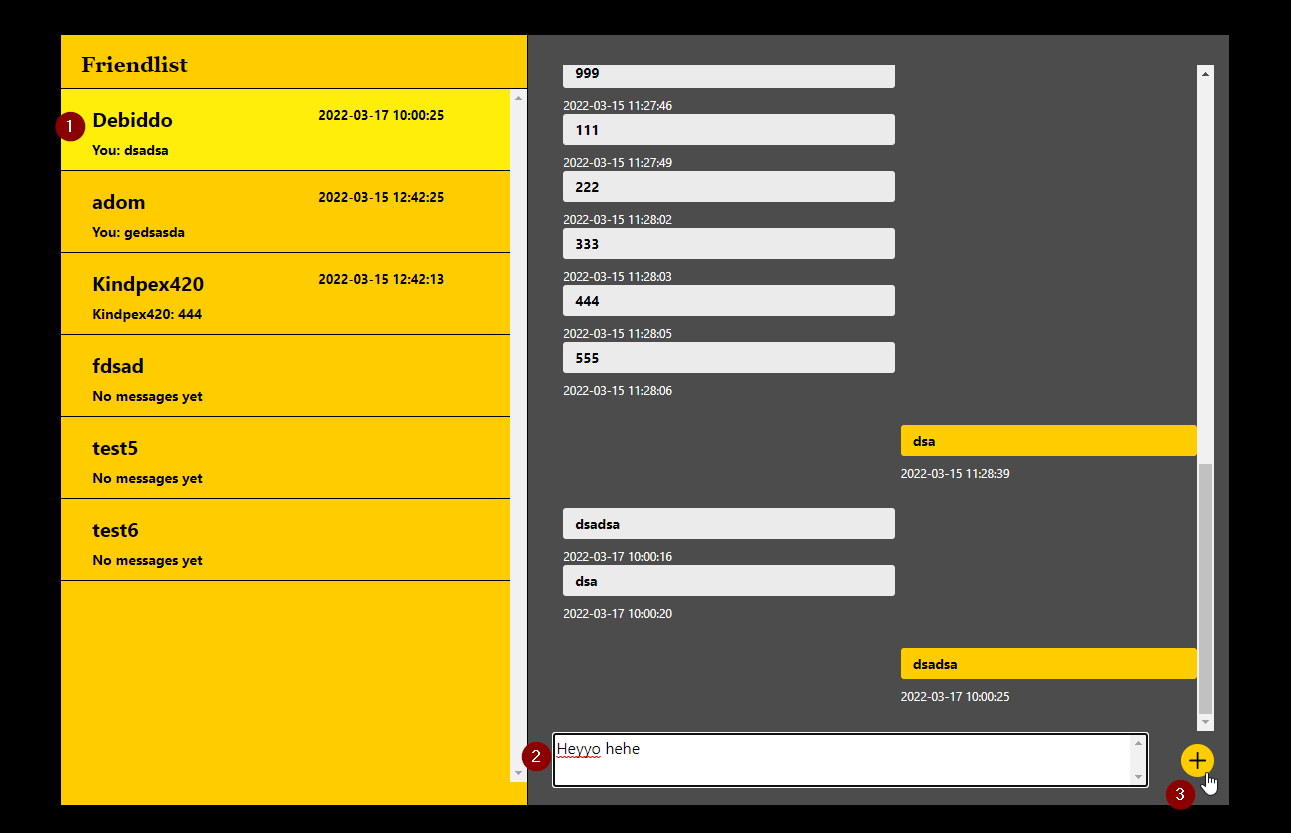
Lista tartalma a lista nevére kattintva megjelenik. A „Search for Manga” gombra kattintva megjelenik egy beviteli mező, amiben tudunk keresni mangákra és azokat a plusz ikonra kattintva hozzáadjuk a listához. Törölni mangát a lejjebb lévő listában található kuka ikonokra kattintva tehetjük meg.



## Barátokkal csevegés

A csevegő oldalra a navigációs sávon lévő „Chat” feliratra kattintva jutunk el.



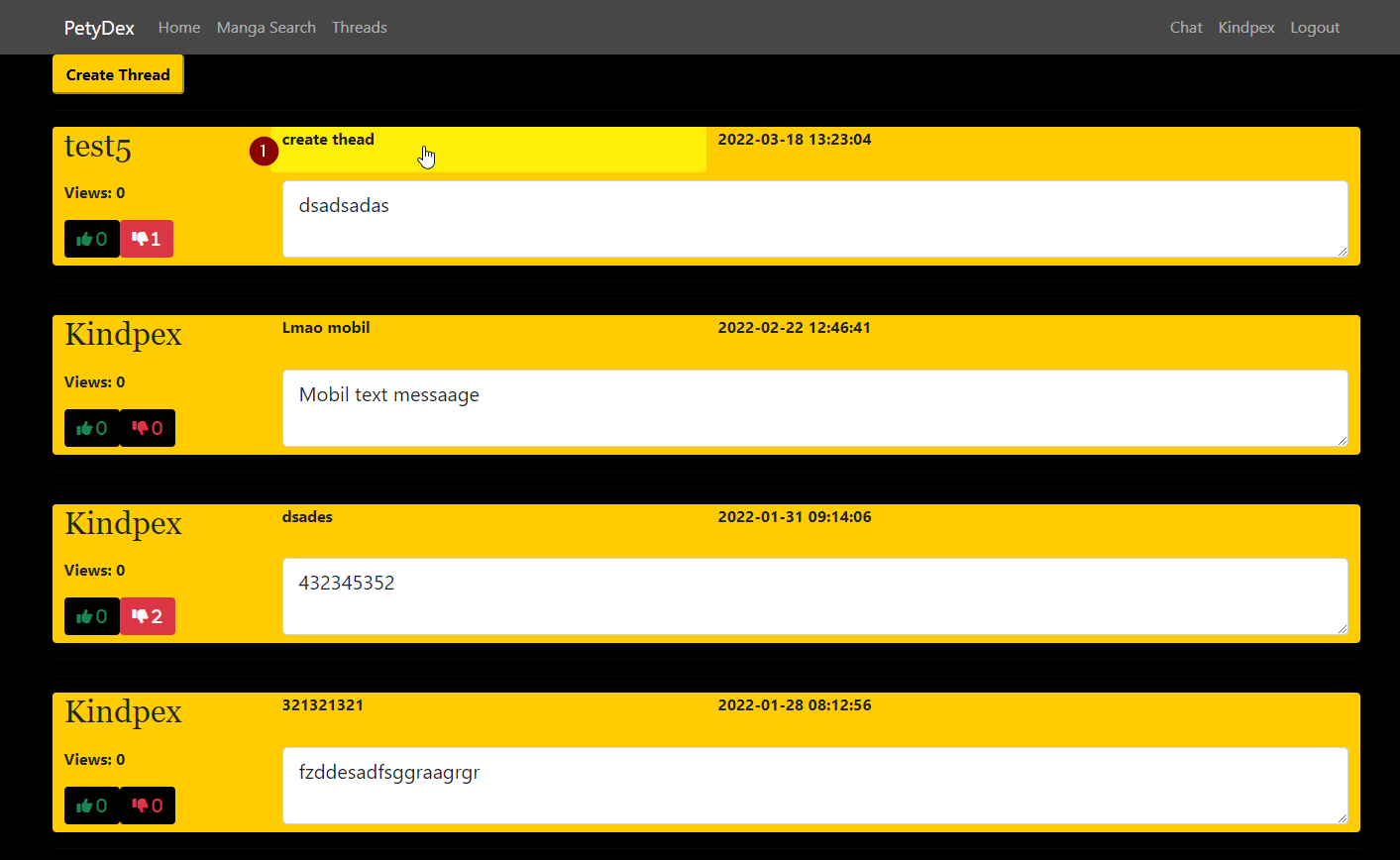


Utána a barát nevére kattintva megjelennek a két fél között küldött üzenetek, és az üzenetek alján lévő beviteli mezőbe lehet írni az üzenetet, amit a plusz gombra kattintva lehet elküldeni.

## Fórum létrehozása

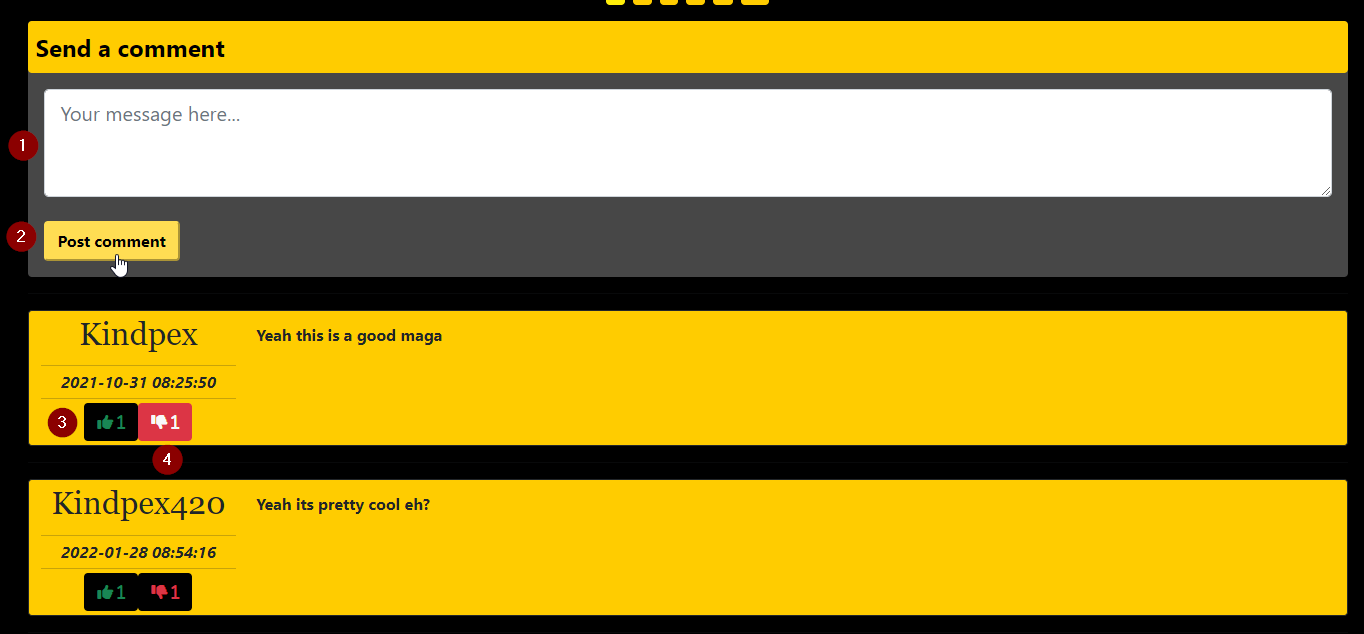
Fórumokat keresni, és létrehozni a Threads oldalon lehet, ehhez, a navigációs sávban lévő „Threads” feliratra kattintva lehet eljutni. Majd a „Create Thread” gombra kattintva megjelenik 2 beviteli mező, az egyik a thread-nek a címét várja, a másik a thread témáját, majd a „Create” gombra kattintva létre lehet hozni ezt. Egy thread-et megnyitni meg a thread címére való kattintásával lehet.





## Kommentelés

Kommentelni lehet a fórum thread-ek, mangák, és manga fejezetek alá. A beviteli komment mező és a kommentek a fő tartalom alatt helyezkednek el. Kommentelés a beviteli mezőbe való írás, majd a „Post comment” gombra kattintva történik. Más kommenteket lehet lájkolni, vagy diszlájkolni.



# Telepítési útmutató

## GitHub repositoryból való telepítés

* Node.Js telepítése
* MariaDB (vagy MySQL) adatbázis szerver telepítése
* Github repó letöltése https://github.com/StaubJDavid/SzakDolgozat
* MariaDB-n egy adatbázis létrehozása
* A backend szerver mappájában (api) a .env fájlban állítsuk be a frontend szerver címét (http://frontend-címe), a szerver port-ot, az adatbázis csatlakozásához szükséges adatokat (adatbázis szerver címe, felhasználó, jelszó és az adatbázis neve), titkos kulcsot, az email szolgáltató szerverét, portját és az ott használt fiók jelszava, e-mail címe
* Futtassuk le az api mappában lévő db\_tables.sql-t az adatbázis szerveren, hogy létrehozzuk a szükséges táblákat, tárolj eljárásokat, adatokat
* A kliens mappájában (client) a .env fájlban adjuk meg a kliens portját, a weboldal címét, a backend szerver címét, a proxy szerver címét, a proxy címét /endpoint végződéssel és a proxy szerver címét /image végződéssel
* A proxy mappában az index.js fájlban a PUBLIC\_ADDRESS változót állítsuk be a megfelelő címre
* Ha ezek megvannak, egy parancssorral navigáljunk egyesével az api, client és proxy mappába, és mindegyiknél írjuk be ezt a parancsot „npm install” ez letölti az összes használt csomagot, amik szükségesek a futtatáshoz.
* Ha mindegyik mappánál megvan az npm install, indítsuk el az adatbázis szervert és 3 Parancssor segítségével bemegyünk ugyan úgy a proxy, api és client mappákba (1 parancssor 1 mappa) és beírjuk, hogy „npm start” mindegyik parancssorba és ilyenkor elkéne indulniuk a szervereknek, az api szerver, ha elsőre nem indul, próbáljuk meg még egyszer elindítani, mert néha az adatbázishoz a csatlakozás túl sok időbe telik

## Docker

* Docker telepítése és futtatása
* Programhoz szükséges imagek letöltése:
  + Adatbázis (MariaDB): docker pull davidkah20/petydex-mariadb
  + Backend: docker pull davidkah20/petydex-api
  + Proxy: docker pull davidkah20/petydex-proxy
  + Kliens: docker pull davidkah20/petydex-client
* Következő portok felszabadítása, ha fut szerver ezeken:
  + 3000 (Proxy)
  + 3001 (Backend)
  + 3011 (Kliens)
  + 3306 (Adatbázis)
* GitHub repositoryból (<https://github.com/StaubJDavid/SzakDolgozat>) a docker-compose.yml fájlt töltsük le, vagy az egész repositoryt
* Nyissunk meg egy parancssort (cmd), navigáljunk abba a mappába ahol a letöltött docker-compose.yml fájl van és futtassuk le a „docker-compose up” parancsot.
* Elindításnál a backend szerver addig fog crashelni és ujraindulni, amíg nem tud csatlakozni az adatbázishoz. 3-5-ször crashel a backend indulása, ennyi idő alatt az adatbázis általában el szokott indulni és csatlakozható állapotban van.
* A weboldalt a <http://localhost:3011> -en tudjuk elérni a böngészőből.