Pannon Egyetem

Műszaki Informatikai Kar

[Tanszék neve]

[Szak neve]

**SZAK-/DIPLOMADOLGOZAT**

**[Dolgozat címe]**

**[Név]**

Témavezető: [Témavezető neve]

Külső/belső konzulens: [Konzulens neve]

[évszám]

Témakiírás

A szkennelt formában megkapott témakiírás beillesztése a dolgozatba.



Hallgatói nyilatkozat

Alulírott <<hallgató neve>> hallgató kijelentem, hogy a dolgozatot a Pannon Egyetem <<tanszék neve>>én készítettem a <<végzettség>> 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]

*<<hallgató neve>>*

Témavezetői nyilatkozat

Alulírott <<témavezető neve>> témavezető kijelentem, hogy a dolgozatot *<<*hallgató neve*>>* a Pannon Egyetem <<tanszék neve>>én készítette <<végzettség>> végzettség megszerzése érdekében.

Kijelentem, hogy a dolgozat védésre bocsátását engedélyezem.

Dátum: Veszprém, [év hónap nap]

*<<témavezető neve>>*

Köszönetnyilvánítás

A hallgató köszönetet nyilvánít mindazoknak, akiktől (elméleti, gyakorlati, erkölcsi stb.) segítséget kapott.

Tartalmi összefoglaló

Tartalmi összefoglaló magyarul. Az összefoglalónak tartalmaznia kell (rövid, velős és összefüggő megfogalmazásban) a következőket:

* téma megnevezése,
* megoldott feladat megfogalmazása,
* megoldási mód,
* elért eredmények,
* kulcsszavak (4-6 darab)
* terjedelme nem lehet több 1 A4-es oldalnál.

Az összefoglalót magyar és angol nyelven kell készíteni. Sorrendben a dolgozat nyelvével megegyező kerül előrébb. A cím Title stílusú, formázása: Times New Roman, nagybetű, 14 pt, félkövér, középre igazított; az összefoglaló Normál stílusú, formázása: Times New Roman, 12 pt, sorkizárt, 1.5-ös sortávolság.

**Kulcsszavak:** [4-6 kulcsszó felsorolása, vesszővel elválasztva]

Abstract

Abstract in English

**Keywords:** [list 4-6 keywords]

Tartalomjegyzék

[Jelölésjegyzék 9](#_Toc101441443)

[1. Bevezetés 10](#_Toc101441444)

[2. Versenytárs elemzés 11](#_Toc101441445)

[3. Saját megoldás ismertetése 12](#_Toc101441446)

[3.1. Backend 12](#_Toc101441447)

[3.2. Adatbázis 12](#_Toc101441448)

[3.3. Frontend 12](#_Toc101441449)

[3.4. Proxy 13](#_Toc101441450)

[4. Felhasznált technológiák 14](#_Toc101441451)

[4.1. MangaDex API(Külső API) 14](#_Toc101441452)

[4.2. Backend 14](#_Toc101441453)

[4.2.1. Node.JS 14](#_Toc101441454)

[4.2.2. Express 15](#_Toc101441455)

[4.2.3. MySQL(Node package) 15](#_Toc101441456)

[4.2.4. JSON Web Token (JWT) 16](#_Toc101441457)

[4.2.5. Socket Io 17](#_Toc101441458)

[4.2.6. Nodemailer 17](#_Toc101441459)

[4.2.7. Bcrypt(Node Package) 18](#_Toc101441460)

[4.2.8. Node Cron 19](#_Toc101441461)

[4.2.9. Uuid 19](#_Toc101441462)

[4.2.10. Joi 19](#_Toc101441463)

[4.3. Adatbázis 21](#_Toc101441464)

[4.3.1. MariaDB 21](#_Toc101441465)

[4.4. Frontend 22](#_Toc101441466)

[4.4.1. React 22](#_Toc101441467)

[4.4.2. Redux 24](#_Toc101441468)

[4.4.3. Axios 26](#_Toc101441469)

[4.4.4. Socket Io Client 26](#_Toc101441470)

[4.4.5. Typescript 27](#_Toc101441471)

[4.4.6. Classnames 27](#_Toc101441472)

[4.5. Proxy 28](#_Toc101441473)

[4.5.1. http-proxy-middleware 28](#_Toc101441474)

[5. Specifikáció 28](#_Toc101441475)

[5.1. Funkcionális követelmények 28](#_Toc101441476)

[5.2. Nem funkcionális követelmények 29](#_Toc101441477)

[5.3. Rendszer architektúra 29](#_Toc101441478)

[5.3.1. Kommunikációk rétegek között 30](#_Toc101441479)

[5.3.2. Frontend 32](#_Toc101441480)

[5.3.3. Backend 43](#_Toc101441481)

[5.3.4. Adatbázis 56](#_Toc101441482)

[5.3.5. Proxy 59](#_Toc101441483)

[6. Tesztek 63](#_Toc101441484)

[6.1. Backend 63](#_Toc101441485)

[6.2. Frontend 63](#_Toc101441486)

[7. Továbbfejlesztési lehetőségek 63](#_Toc101441487)

[8. Telepítési útmutató 63](#_Toc101441488)

[Irodalomjegyzék 65](#_Toc101441489)

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 meg a webfejlesztés témaköre is érdekelt, úgy gondoltam hogy tökéletes szakdolgozat téma lenne egy saját közösségi weboldalt készíteni ami a mangák körül forog.

A weboldal főleg olyan embereknek készült akik nem csak mangákat szeretnének olvasni, hanem más, manga iránt érdeklődő emberekkel szeretnének találkozni, beszélgetni.

# 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, 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 hotlinkingtől.

A manga olvasó weboldalak többsége a fejezetek alá való kommentelésen(gyakoribb), 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 ugye mangákat olvasni, saját listákat létrehozni ahová gyűjthetünk például olyan mangákat amik érdekelnek minket, így könnyebben lehet emlékezni hogy miket olvasunk.

# Saját megoldás ismertetése

## Backend

Backend szerverhez Node.js-t használom ami egy javascript futási környezetet biztosít, szóval lehet javascript kódokat lefuttatni a böngészőn kívül is [1].

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

* REST API
* 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 package(mysql) segítségével kommunikál.

## 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.

## 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 paneljeit, letöltse 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 hotlinking elkerülésére használom, a proxy használata tökéletes arra hogy biztonságos helyen(saját szerveren) tároljuk külső API-hoz szükséges hitelesítő kulcsokat, így nem áll fenn annak a veszélye hogy kikerül a nagyvilágba.

# Felhasznált technológiák

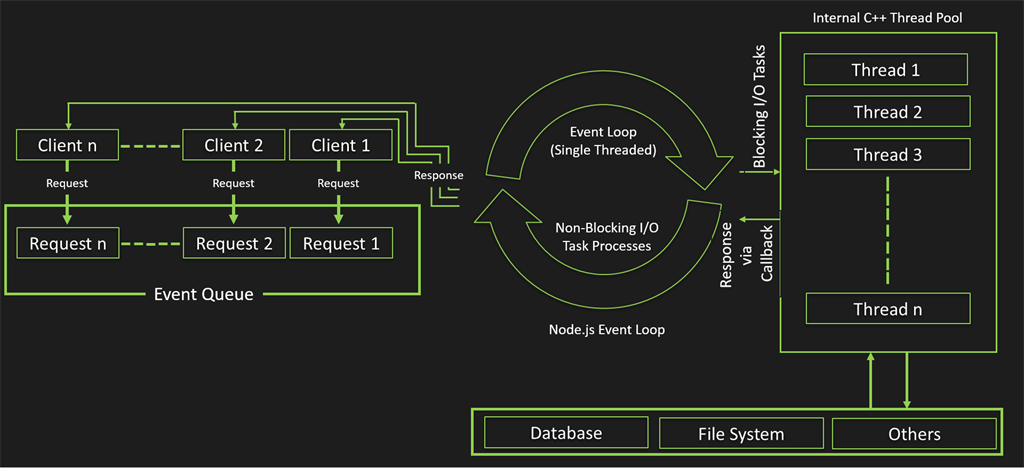
## MangaDex API(Külső API)

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 HTTP 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 a kapcsolatokat „elméletben” 1 szálon dolgozza fel aszinkron módon, gyakorlatban a fő szál(Event Loop) más szálaknak(Workers) osztja ki a feladatokat egy event sorból amiket ők elvégeznek a megfelelő C++ megvalósítással párhuzamosan vagy aszinkron módon és vissza küldik az eredményt a fő szálnak, amit vissza adhat 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 hálozatról érkező adatok olvasása, a nehézkes szálak kezelésével. De 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. Előnyhőz 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 megkönnyítik munkákat mert nem kell saját kódot írnunk hanem más, az esetek 99%-ban jobban megírt megvalósítását használhatjuk.

Viszont egyik hátránya a Node.js nek hogy a CPU intenzív feladatokkal nem tud olyan jól megbírkózni, mint a fennt 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(Nodejs package) amivel könnyen és gyorsan lehet NodeJS web applikációt, API-t létrehozni. A NodeJS alap http/https moduljait használja fel. Maga az Express sok „opcionális” middlewarel rendelkezik amiket külön lehet telepíteni, vagy saját middlewaret is írhatunk amik kibővítik az API vagy web applikációnak a funkcióit, és mivel ezeket a middlewareket külön lehet megszerezni és nem az Express alap csomagjában jönnek elég lightweight és könnyen testreszabható a web applikációnk vagy API-unk.

### MySQL(Node package)

Ez a Node Package egy driver-t biztosít amivel lehet csatlakozni egy MySQL adatbázishoz(Mi esetünkben egy MariaDB-hez de a MariaDB és a MySQL szinte azonosak), és egy olyan interfacet 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 biztosítására, hiba kezelésre, adat escapelésre hogy megvédjük szerverünket SQL injectionöktől, és az adatbázisban felmerülő típusokat SELECT query esetén Javascript típusokba kasztolja és egyéni kasztolásra is van lehetőség

### JSON Web Token (JWT)

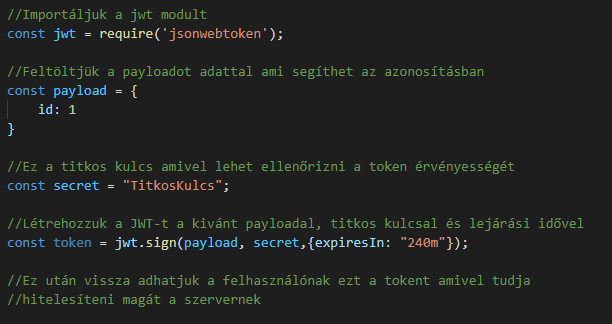
#### JSON Web Token

A JWT módot biztosít arra hogy kettő vagy több fél között biztonságosan tudjon adatokat küldeni. Ez a JWT egy hosszú string ami egy JSON-t tartalmaz amibe bármi féle adat lehet, és lelehet ellenőrizni ennek a tokennek a sértetlenségét, valóságát. Ezt a tokent tovább lehet titkosítani hogy a tartalmát is teljesen biztonságba tudjuk szállítani, mert alapból a JWT csak azt tudja biztosítani hogy ha a tokent megváltoztatták, akkor érvénytelen legyen, de attól még ha valaki megszerzi a tokent, 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 tokent amit a kliens eltárol, és az elkövetkezendő requestekhez hozzá teszi ezt a tokent és a szerveren megvizsgáljuk a tokent hogy érvényes-e, és ha az, akkor a szerver végrehajtja a requestet.

#### Jsonwebtoken(Node Package)

Ez a node package implementálja ezt a JWT technológiát, szóval lehet Javascriptben JWT-ket létrehozni, ellenőrizni, dekódolni. Könnyen belehet állítani a létrehozásnál hogy milyen algoritmust használjon a titkosításra, mikor járjon le az érvényessége, ki állította ki és még mást is.

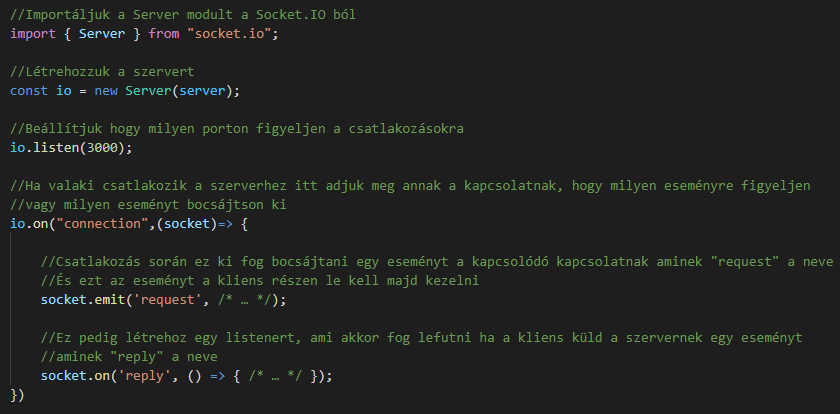


2. ábra: JWT létrehozása jsonwebtoken csomaggal

### Socket Io

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

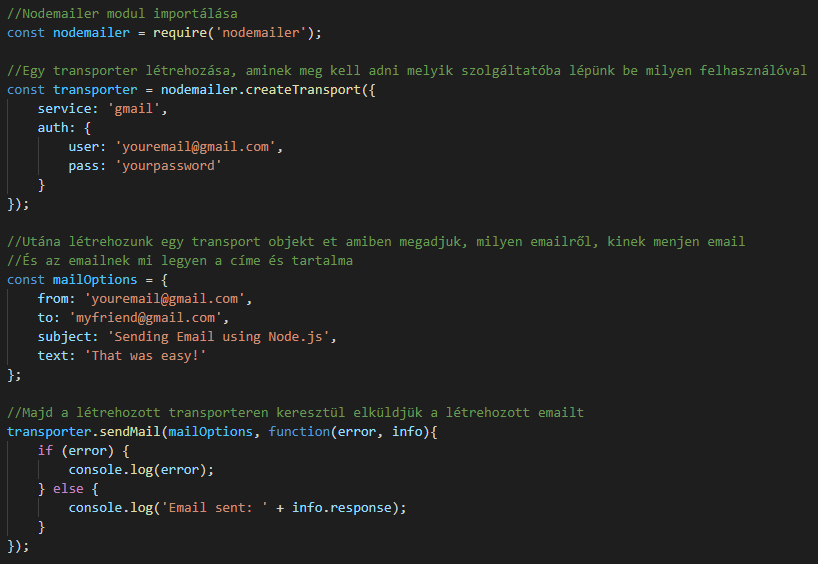
A Socket.IO esemény alapú amit annyit jelent, hogy a szerver és a kliens létrehozott eseményekre figyel, vagy eseményeket bocsájt ki



3. ábra: egyszerű szerver felállítása Socket.IO csomaggal

### Nodemailer

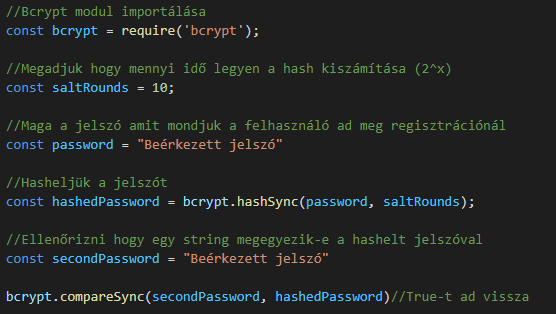
A Nodemailer egy olyan modul ami lehetővé teszi az e-mail-ek küldését a szerverről. Lehet saját email szerverről(Simple Mail Transfer Procotol – SMTP-vel) és más email szolgáltatásokkal(például: Gmail) emailt küldeni.



4. ábra: E-mail küldése nodemailer segítségébel

### Bcrypt(Node Package)

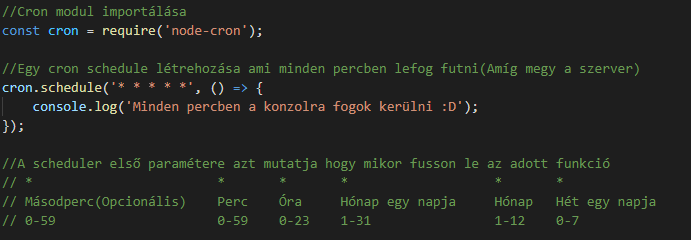
Maga a Bcrypt az egy jelszó hashing funkció ami egy elég megbízható titkosítást biztosít a jelszavaknak. Én a bcrypt Javascript implementálását használom.



5. ábra: jelszó titkosítása bcrypt-el

### Node Cron

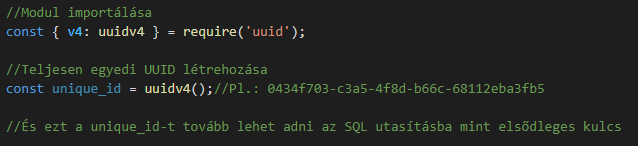
Node Cron lehetővé teszi hogy bizonyos időpontokban lefuttassunk automatikusan funkciókat. A modul szintaktikája a GNU operációs rendszer crontabján alapszik, ami mint a Node Cron, lehetővé teszi hogy bizonyos időpontban parancsok, fájlok fussanak le.



6. ábra: ütemezett funkció megadása node-cron al

### Uuid

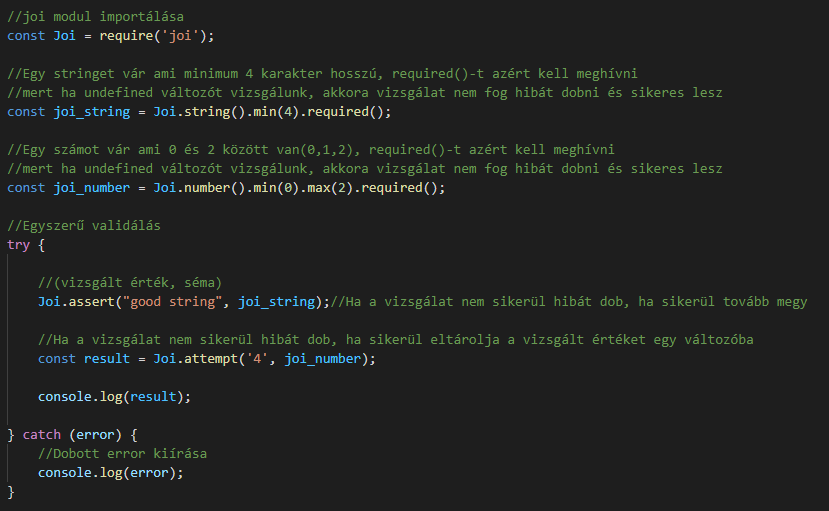
A Universal Unique Identifier(UUID) egy nagy valószínűséggel teljesen egyedi azonosító kódot jelent, amit sok különböző tényezőből például a létrehozásnál lévő időpont, hálózati cím segítségével hoz létre egy algoritmus. Én ennek az algoritmusnak a Node.Js beli implementációját használom(uuid Ami a node package nek a neve) az adatbázisban tárolt pár adatnak az azonosításához.



7. ábra: UUID létrehozása

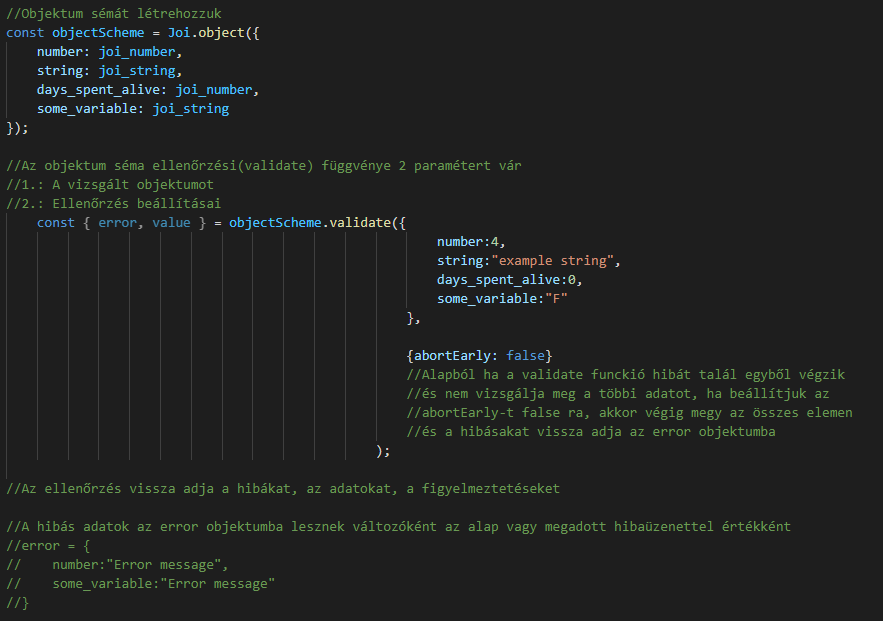
### Joi

A Joi az egy olyan modul amivel lehet adatsémákat létrehozni, és adatokat ellenőrizni hogy megfelelnek-e a várt formának, 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 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 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 általában jobban skálázhatóbb és gyorsabb mint mondjuk a MySQL. [4]

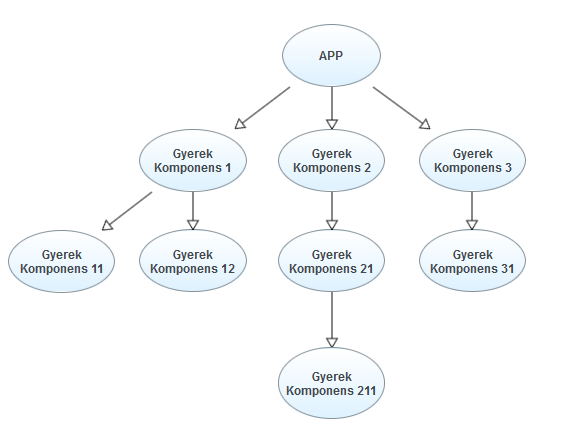
## Frontend

### React

React az egy javascript könyvtár felhasználói interfacek létrehozásához. Komponens alapú ami annyit jelent hogy valamilyen logikát, html-elemeket egy komponensbe összegyűjt, és lehet meghívni/felhasználni a kívánt helyen.

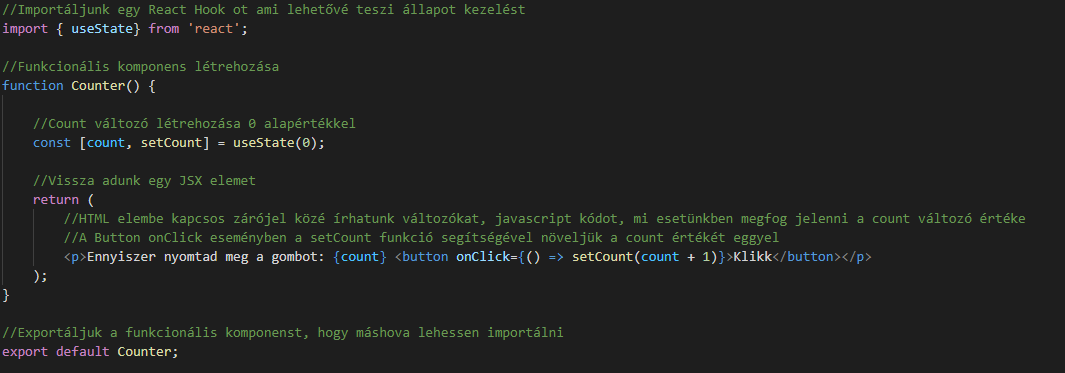
Ezeknek a komponenseknek van saját állapota is, vagyis a komponensen belül lehet azokat az adatokat tárolni, amiket felhasznál, de nem csak adatokat tárolhat hanem funkciókat is, amik megváltoztathajták a komponens állapotát, kinézetét.

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



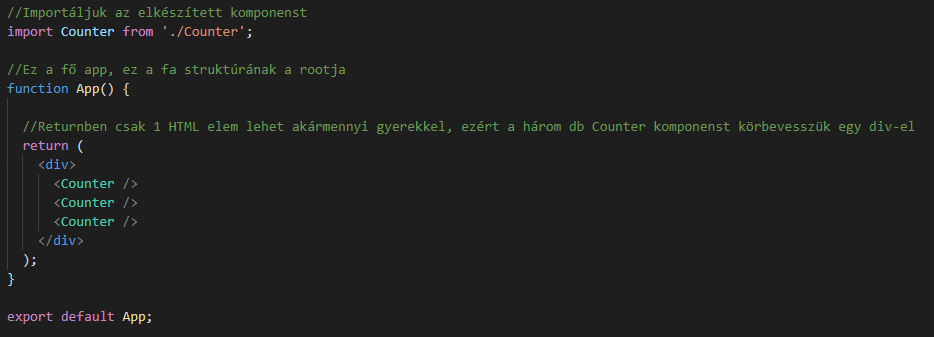
10. ábra: React alkalmazás komponens struktúrája

Egy gyors 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.

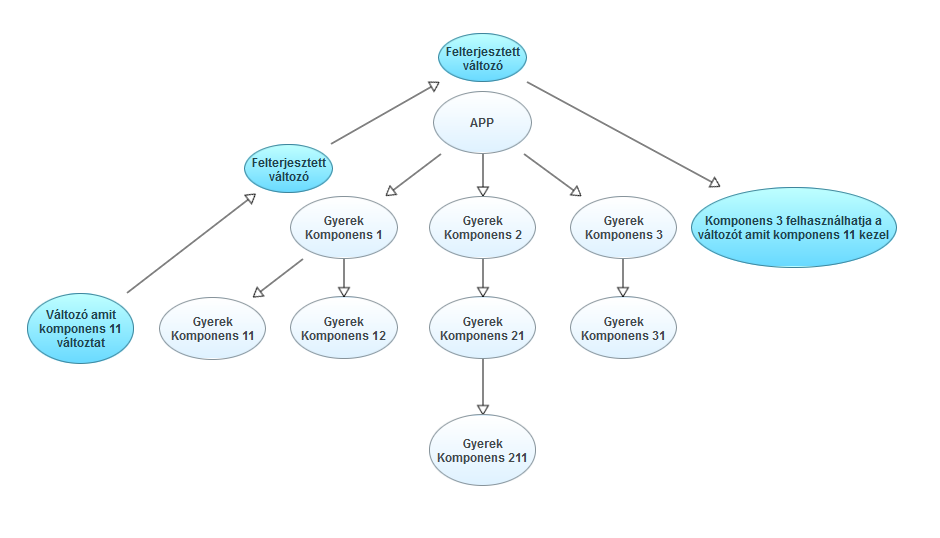
Ez csak egy bevezetés volt a React komponens rendszerébe, de sokkal több funkciót is ellehet vele végezni, mint például feltételes megjelenítést, ami annyit jelent hogy valamilyen feltételek hatására, a komponens mást fog megjeleníteni.

### Redux

Maga a Redux globális állapot tárolásra használható, amivel könnyen lehet úton követni a programnak a változásait és debuggolni. 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 manager? Azért, mert a React-ban a komponensek fa struktúrában helyezkednek el és ha egy komponensből átakarunk vinni egy másik komponensbe egy adatot, akkor azt felkell 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 megkell tennie az adatnak, akkor jobban járunk használjuk ezt a globális állapot megoldást.



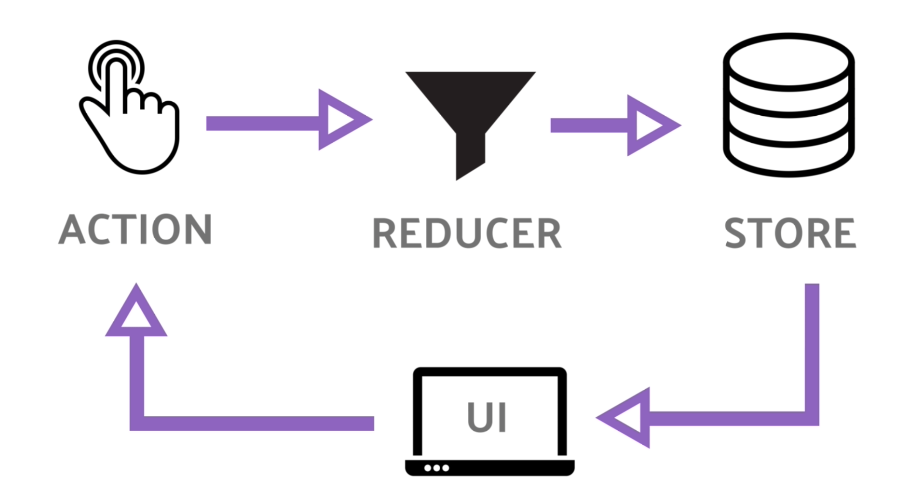
14. ábra: React fa struktúrájában adat megosztás más komponensekkel

Ahogy a példán látható a gyerek komponens(gyk) 11 ben változtatjuk a változó értékét, de hogy elérje egy másik komponens, felkell terjesztenünk a szülőbe ezt a változót, ami igazából annyi jelent hogy a szülő komponens ben 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 User Interface innen szerzi az adatokat amiket megjeleníthet vagy feldolgozhatja. 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 storeban 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 actiont, ami valamilyen adat feldolgozást csinálhat(pl.: kapott adat + 1) és utána az action akár több reducert is meghívhat amik megváltoztatják a storeban lévő értékeket, amiknek a változása megjelenik a felhasználói felületen, ha éppen megjeleníti azt.

### Axios

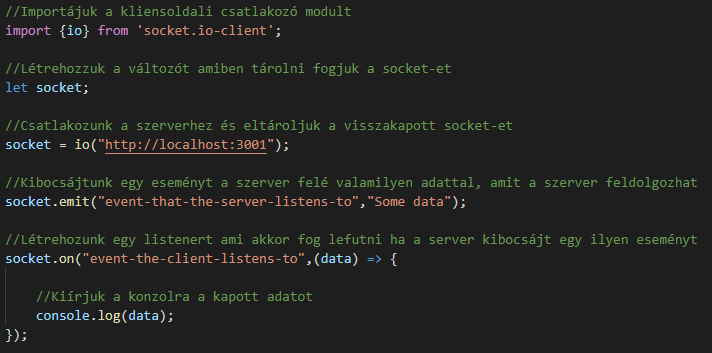
Axios az egy promise(ígéret) alapú http kliens amivel lehet http kéréseket küldeni böngészőből és nodejs alkalmazásból. Javascriptnek a Promise API-ját használja ezért az aszinkron kéréseket lehet kezelni .then és .catch megoldással, vagy az await operátorral hogy csak akkor történjen adat felhasználás, ha a kérésre érkezett válasz.

Van még több funkciója is a http kérések indításán kívül például a kimenő kéréseket interceptálja és új adatokat ad hozzá a kéréshez pl.: Minden kéréshez hozzáadja a Header-be az authentikációhoz szükséges tokent.

### Socket Io Client

Ez a szerveren használt socket es kommunikációnak a kliens oldali fele, itt is egy socketen lehet létrehozni eseményeket amire figyel és lefutnak azok a funkciók amiket akarunk ha a szerverről érkezik egy olyan esemény 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

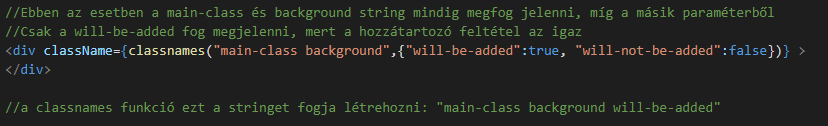
TypeScript az egy programozási nyelv, igazábol javascript csak lehetővé teszi hogy típusokat használjunk amik csökkenthetik a javascript gyengén típusos szerkezetéből adandó hibákat.

Lehetővé teszi még hogy a funkcióknak megadjuk milyen típusú paramétereket várjon és milyen típusú értéket adjon vissza. Lehet névtereket(namespace), enumerátorokat és saját objektum sablonokat is könnyen létrehozni. A saját készítésű típusokat például enumerátorok, objektum sablonokat is lehet hozzárendelni változókhoz, függvényekhez.

A TypeScript azért is jó mert compileolásnál ellenőrzi hogy az adott típusok jó fajta adatot kapnak, és ha érzékeli hogy valamilyen változó rossz fajta 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 hozták létre. Igazából ez a modul csak annyit csinál, hogy a megadott stringekből létrehoz egy új stringet amivel a html css osztályokat lehet hozzárendelni egy html elemhez. A feltételes összekapcsolást a stringekhez tartozó feltétel típus kiértékelésével végzi el, vagyis ha egy stringhez tartozik egy feltétel akkor az akkor fog megjelenni a végső stringbe 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 mielött továbbadja, lehet az logolás, érvényesség ellenőrzése, header vagy body módosítása. De fellehet 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, authentikációhoz adatokat nem a kliensen tároljuk hanem a proxy-n amihez a kliens nem fér hozzá.

# Specifikáció

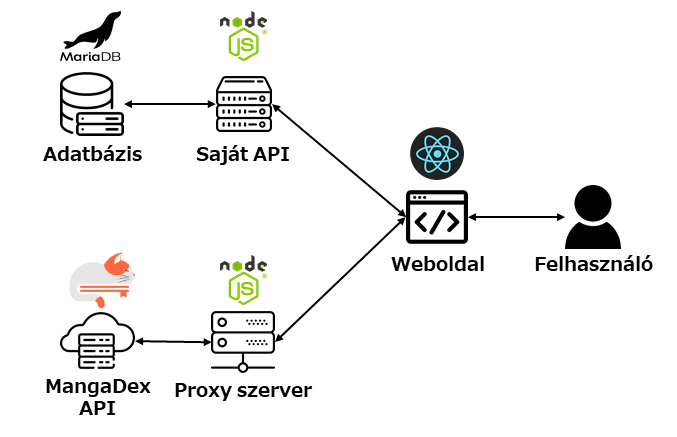
## Funkcionális követelmények

* Mangák keresése
* Mangák olvasása
* Mangák értékelése
* Manga írók, fordítók megjelenítése
* Mangák gyűjteményekbe 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 chatelés
* MangaDex-ről hotlinkelés elkerülése

## Nem funkcionális követelmények

* Backend szerver Node.Js API szerver
* Adatbázis MySQL(MariaDB)
* Frontend React-ban íródjon

## Rendszer architektúra



18. 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 mennek amik a mangákkal kapcsolatos például keresés, olvasás, míg a saját API felé főként a szociális funkciókért indít kéréseket az oldal például bejelentkezés, kommentelés, chat barátokkal.

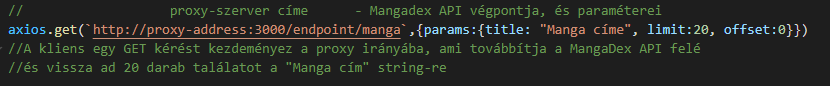
### Kommunikációk rétegek között

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, igaz, hogy ahhoz hogy beérkező üzenet jöjjön, egy másik kliensnek ezt kezdeményeznie kell, de a vizsgált architektúrában nincs jelen más felhasználó.\*/

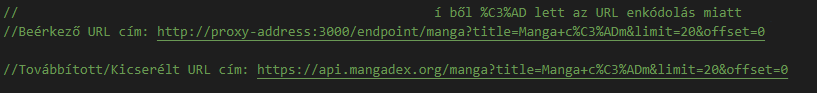
#### Frontend és MangaDex API között

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



19. 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.

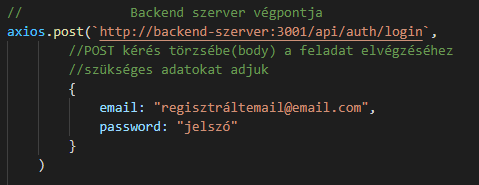


20. ábra: proxy által továbbított URL cím

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.



21. ábra: Backend szerver felé küldött kérés

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

#### Frontend és Backend Socket között

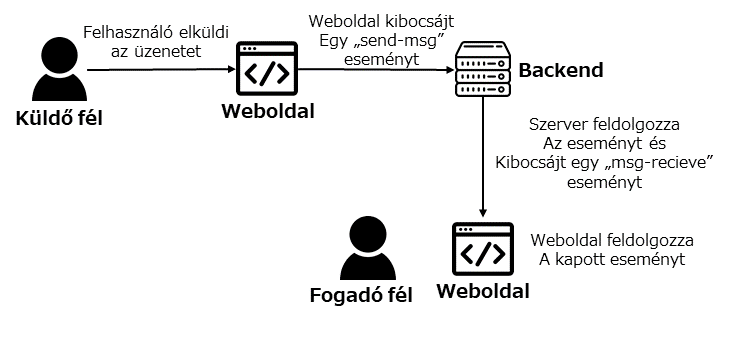
A kliens és backend nem csak http kérésekkel kommunikál hanem egy kétfolyamú valós idejű socket kommunikációs megoldás is történik. Ezt a fajta kommunikációt a Socket.IO node csomag biztosítja.

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 nyílván.

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



22. ábra: socketes 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

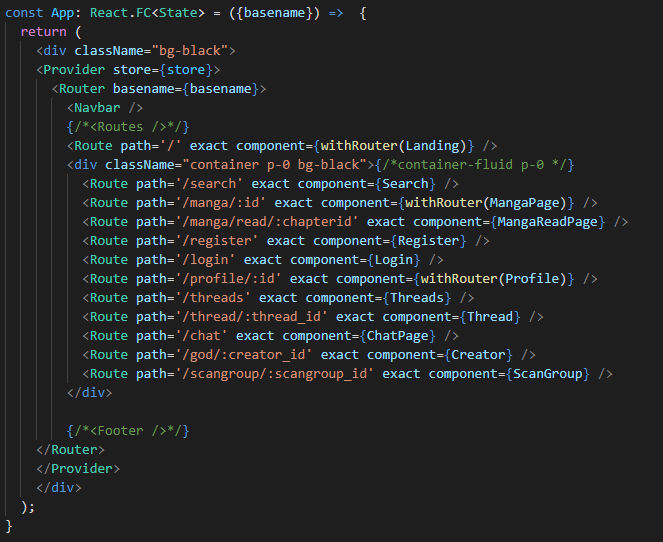
| +---modules

| +---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 fájlt, 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-re 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 átalakítás
* **layout mappa(src/app/layout):** Ebben a mappában az applikáció főoldala van és a navigációs bar
* **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 reduxhoz tartozó reducereket tárolja, a reducer fájlok a globális állapot egy-egy fő állapotát jelenti.
* **reducer mappa index.tsx:** Ez a fájl hozza össze a reducereket(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.



23. ábra: React alkalmazás hogyan kezeli az URL végpontokat

#### Redux store

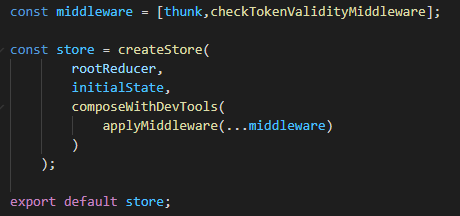
A store.tsx fájlban hozom létre a React store-t ami vár egy reducert(állapotok), kezdő értéket és utána opcionálisan meglehet adni middlewareket, amik minden action esetén lefut.



24. ábra: React Redux állapotainak importálása

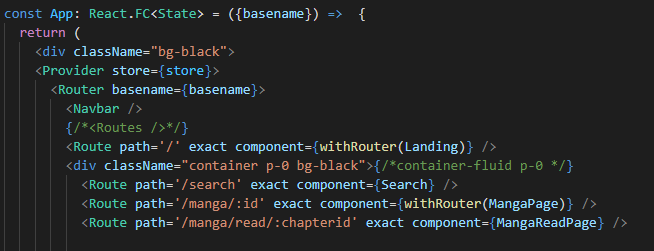


25. ábra: React Redux globális állapot alapértékeinek beállítása



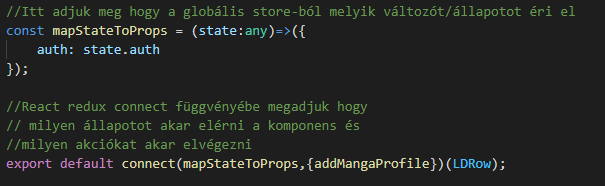
26. ábra: React Redux globális állapot létrehozása

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.



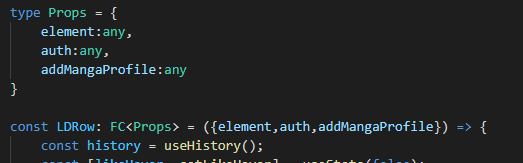
27. ábra: React Redux szolgáltatás hozzáadása a React applikációhoz

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 letudja kérdezni az értékét.

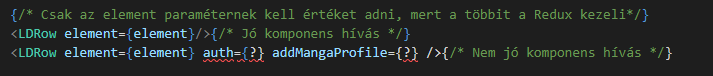


28. ábra: Komponenst csatlakoztatjuk a globális állapothoz

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



29. ábra: Komponensnek átadjuk hogy milyen adatokat kap



30. ábra: Komponens jó és rossz használata

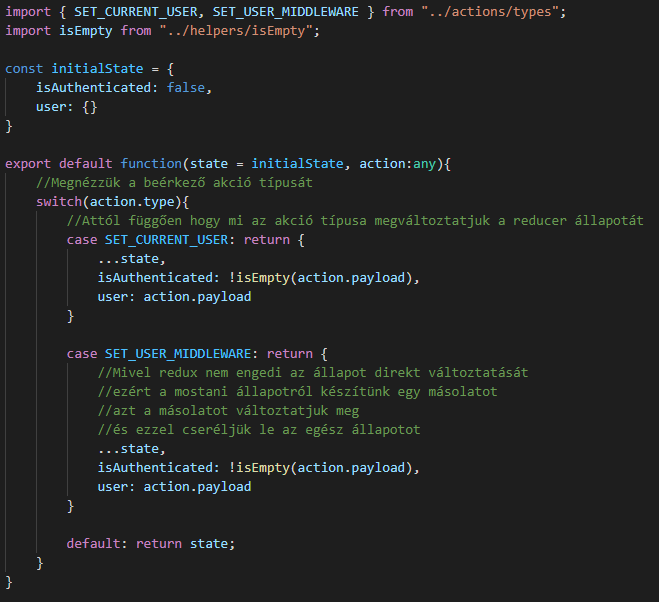
#### Redux reducerek

10 db reducert használok, ezek a reducerek a store-ban egy-egy fő érték objektumot reprezentálnak. Ezeket a reducereket a fő reducerben összesítem és adom át a store-nak mikor létrehozzuk a store-t



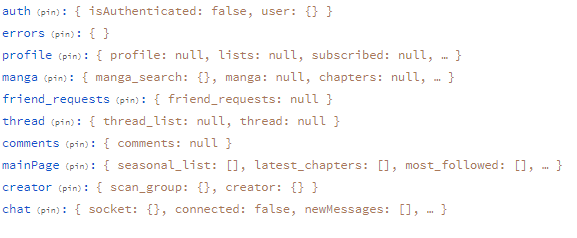
31. ábra: Fő reducerben összegyűjtjük a különböző reducereket

Minden reducer a saját kis állapotáért felel és tartja nyilván. Mindegyik reducerben 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.



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

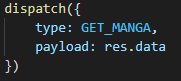
Hogy mik lehetnek egy akció típusa, az az actions mappában a types.tsx ben vannak definiálva, és onnan importálom a szükséges típusokat.



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

#### Redux akciók

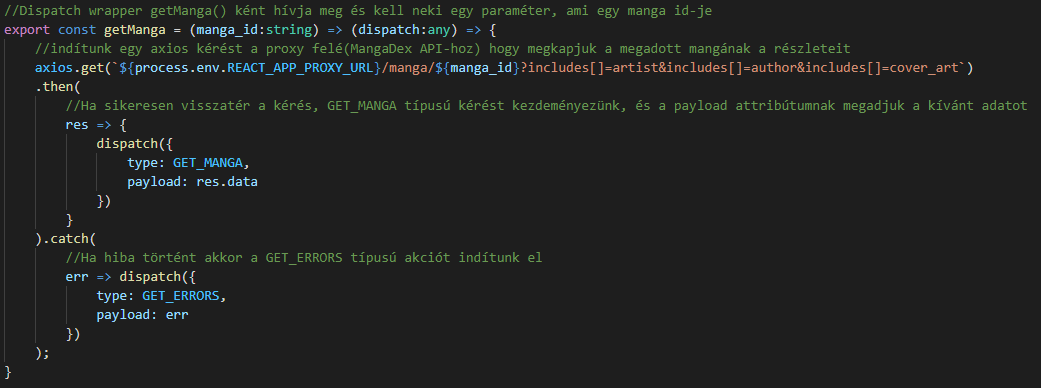
A Redux akciókkal kezdeményezünk változást a store felé ezt a redux dispatch() metódusával lehet csinálni, bárhol meglehet hívni ezt a dispatch() metódust, az a lényeg hogy paraméterként egy olyan objektumot kapjon, aminek van type attribútuma, mert ez mutatja meg a reduxnak hogy milyen fajta akciót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



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

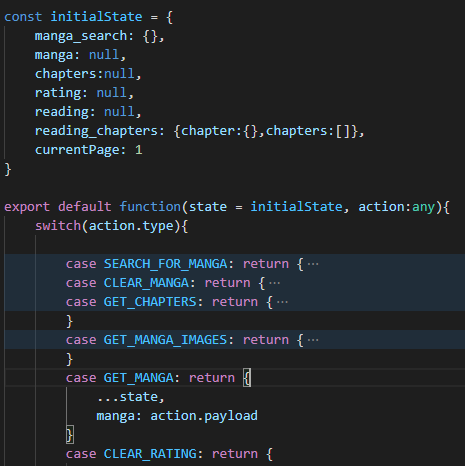
De így nem érdemes akciókat kezdeményezni mert elrontja a kód olvashatóságát ha mindenhol így hívjuk meg az akciókat és ha valami plusz adatfeldolgozást kéne elvégezni egy akcióhoz, azt mindig megkéne írni az adott helyen ahol meghívjuk az akciót.

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



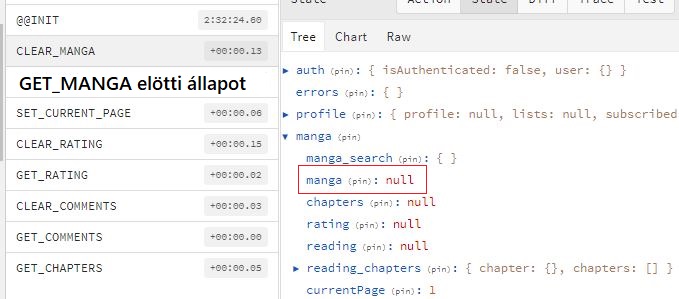
35. ábra: getManga metódus ami lekéri egy manga részleteit és eltárolja azt a React Redux storeban

És ilyenkor sikeres kérés esetén a megfelelő reducerben(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.

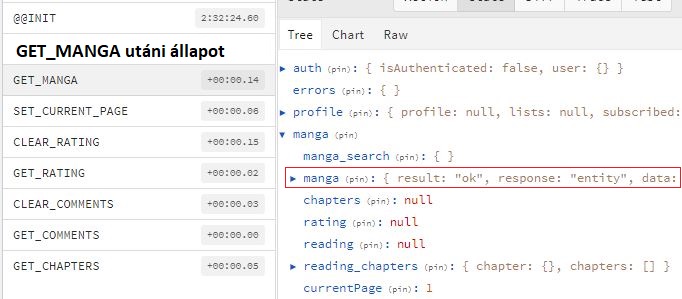


36. ábra: manga reducer felépítése

Következő képen lehet látni a weboldal állapotát, most azt az állapotot látjuk mielött a GET\_MANGA típusú akció meghívódott volna, a manga reducerben nincs értéke a manga változónak mert ez az alapállás.



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



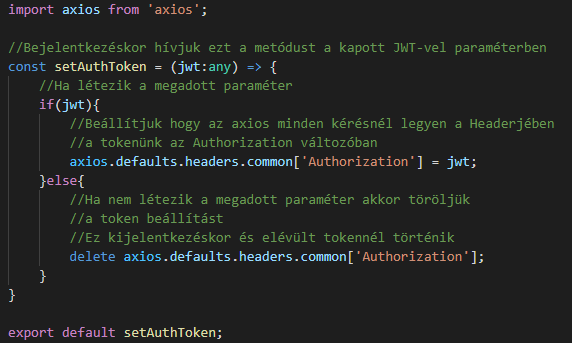
38. ábra: A weboldal azon állapota ami a GET\_MANGA hívás után van

Miután meghívtuk a GET\_MANGA akciót megváltoztatta a weboldal állapotát úgy hogy a manga reducerben lévő manga változónak a kérésből beérkezett adatot állította be.

#### Axios

Axios-t ugye http kérések küldésére használom, de a backend szerveren vannnak olyan végpontok amik eléréséhez szükséges a kérés headerjábe 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 tokent, persze csak ha el van tárolva ez a token. Ezt úgy érem el, hogy sikeres bejelentkezéskor meghívok egy olyan függvényt ami beállítja az axiosnak egy alap értékként.



39. 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 tokent minden akció esetén ellenőrizzük, hogy a token érvényességi ideje lejárt-e, ha lejárt token a felhasználót kijelentkeztetjük.



40. 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 egy adatbázison, a feladat működéséhez szükséges táblákat és tárolt eljárásokat létrehozza és pár szükséges adatot fel is visz 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 beszélnek.
* **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.

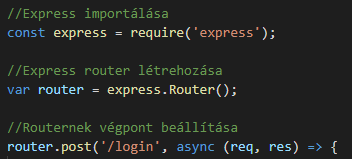
#### Routeok

A szerver végpontjait 8 db csoportba vannak osztva, és ezekhez a csoportokhoz 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 regisztációját, bejelentkezését kezeli.
* **Csevegés(chat.js):** Felhasználók közötti csevegést kezeli.
* **Kommentek(comments.js):** Egy elem kommentjeit kezeli, létrehozás, lekérdezés.
* **Manga listák(lists.js):** Felhasználó manga listáját kezeli.
* **Manga frissítések(mangaUpdates.js):** Manga frissítéseket kezel.
* **Fórum threadek(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.

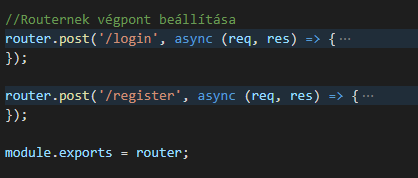
Ezekben a fájlokban egy express Router objektumnak állítok be végpontokat , azt hogy ezeken a végpontokon milyen fajta kéréseket dolgozhat fel(a kérések fajtái a http metódusok lehetnek, én leginkább a GET, POST, PUT és DELETE metódusokat használom) és azt hogy mit csináljon, ha erre a végpontra érkezik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



41. ábra: végpont beállítása express routeren

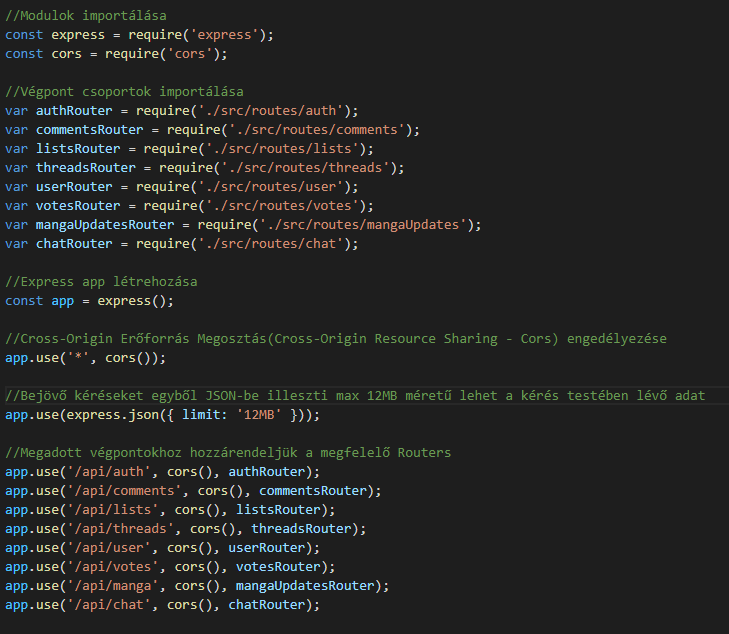
A fenti képen egy router létrehozását és annak egy végpont beállítását látjuk. A /login végpont egy POST kérés esetén lefuttatja a második paraméterben megadott funkciót, ami esetünkben egy anonim nyíl funkció, 2 paramétert kap, req(Request) és res(Response). A req változó a beérkező http kérést tárolja, a res változó pedig a választ, a funckión belül a req változóból letudjuk 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.



42. 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 majd authRouter néven fogom importálni.



43. ábra: express applikáció beállítása

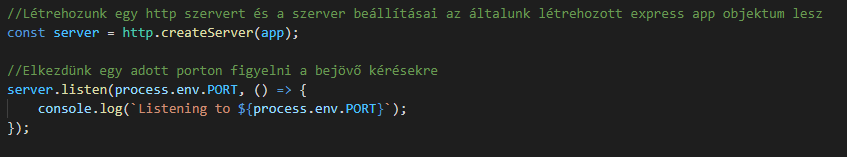
Az express applikáció use() függvénye vár:

* 1 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 string ami az útvonalat jelenti a megadott függvények listája minden kérésre lefut. Itt az importált routereket hozzárendeljük végpontokhoz ez annyit csinál hogy a stringként 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.

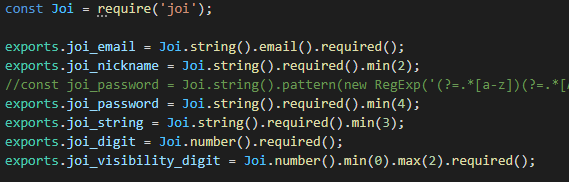
Miután megadtam az express applikációnak a beállításait, létrehozok egy http szervert egy beépített node.js modullal(http modul) és szerver beállításnak megadom az express applikációt, majd elindítom a szervert és megadom neki hogy milyen porton figyeljen bejövő kérésekre.



44. ábra: backend szerver létrehozása és elindítása

#### Ellenőrzés

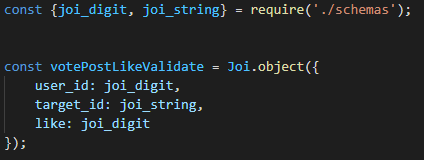
Minden végpontcsoporthoz külön ellenőrzés fájl tartozik és ebben a fájlban tárolom a sémákat és az ellenőrző függvényeket. Ezek a fájlok úgy néznek ki pontosan hogy egy fő séma fájlból importálok úgymond primitív sémákat amik egy-egy érték kivizsgálására jók.



45. ábra: skaláris sémák definiálása

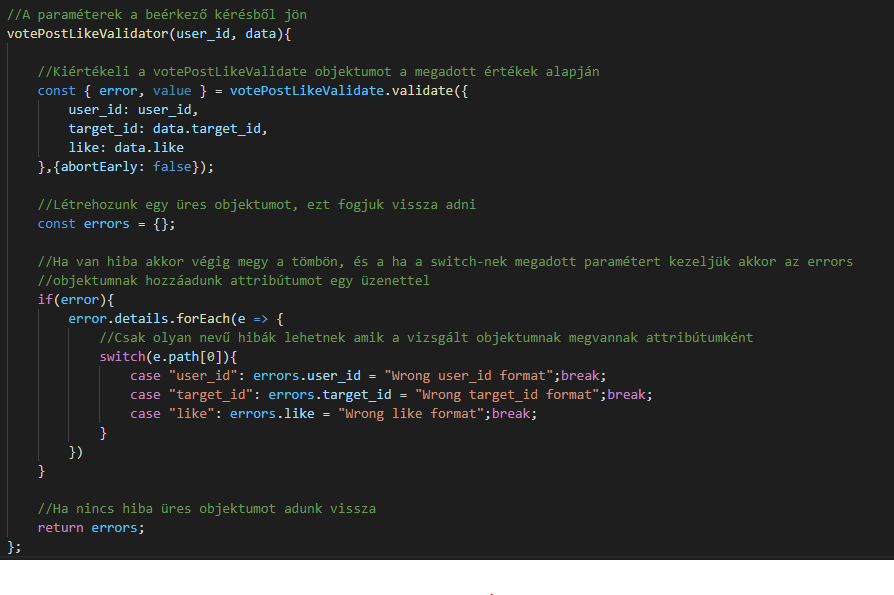
Ezek a sémák az src/helpers/validations mappában találhatók a schemes.js fájlban, ezeket a fájlon belül exportálom egyesével, hogy az ellenőrzés során csak azokat kelljen importálni, amikre szükségem van.

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



46. ábra: joi objektum létrehozása

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



47. ábra: adatellenőrző függvény megvalósítása

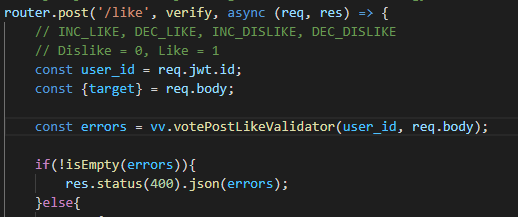
Egy ilyen ellenőrző osztálynak az összes végpontra különböző ellenőrző funkciói vannak a hibakezelés miatt.

Ezt az osztályt a megfelelő végpont fájlban importáljuk.



48. ábra: adatellenőrző osztály importálása

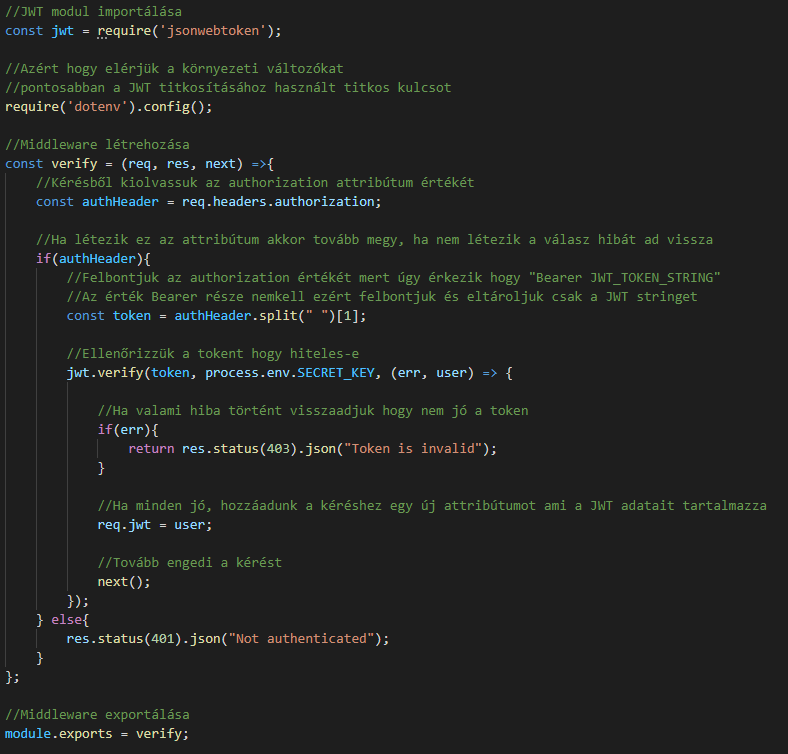
Megfelelő helyen meghívjuk a megfelelő függvényt és eltároljuk az eredményt, amit később megvizsgálunk hogy üres-e vagy nem, ha nem üres, akkor egyből vissza adok egy választ hogy hiba történt és adatként adom az ellenorzás során kapott hiba objektumot, ha nincs hiba akkor folytatódik tovább a program.



49. ábra: adatellenőrző függvény hívása és visszatérési érték ellenőrzése

#### Hitelesítés

A hitelesítést JSON Web Token-el történik amit a kliens bejelentkezésnél megkap és eltárol és ezt a JWT-t a kérések headerjében küldi át Authorization attribútum értékeként. Ennek a JWT ellenőrzésére és hitelesítésére egy saját hitelesítő middlewaret írtam.



50. ábra: hitelesítés middleware

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

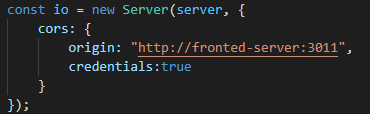
Először meghívom a hitelesítés middlewaret(paraméter listában ez az első metódus ami lefut), ha nem sikeres a hitelesítés akkor hibát ad vissza a kérőnek, ha sikeres akkor a middlewareben lévő next() metódussal tovább megyünk a paraméter listában lévő következő metódusra.



51. ábra: hitelesítés middleware használata végponton

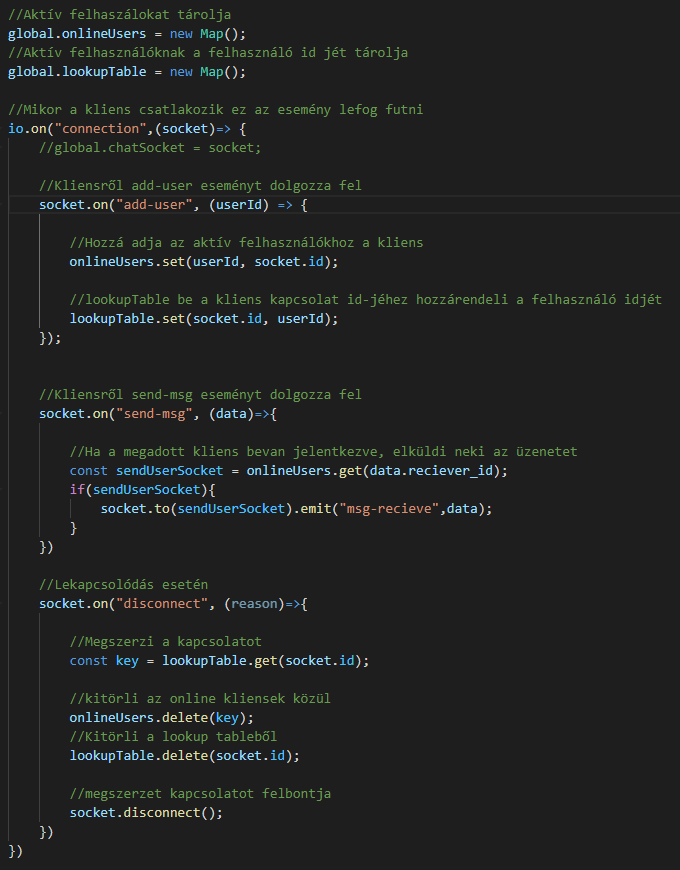
#### Csevegés

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.



52. ábra: socket.io szerver létrehozása

Globálisan tárolom az aktív kapcsolatokat, és egy lookup tablet, ami kapcsolat id-khez hozzárendeli a megfelelő felhasználó id-t, ezt a kapcsolat bontáskor használom. Majd megadom a socket.io szervernek hogy milyen eseményekre figyeljen és mit csináljon ha bekövetkeznek ezek az események.



53. 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.

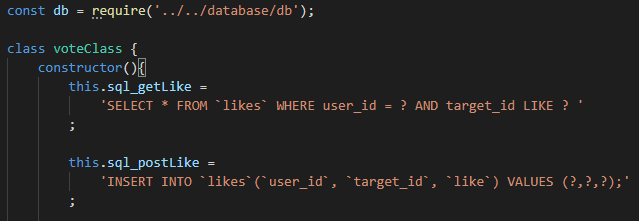
Az adatbázissal való kapcsolatot egy külön fájlban(db.js) létesítem amit exportálok és azokban a fájlokban importálom amiknek szüksége van a kapcsolatra.



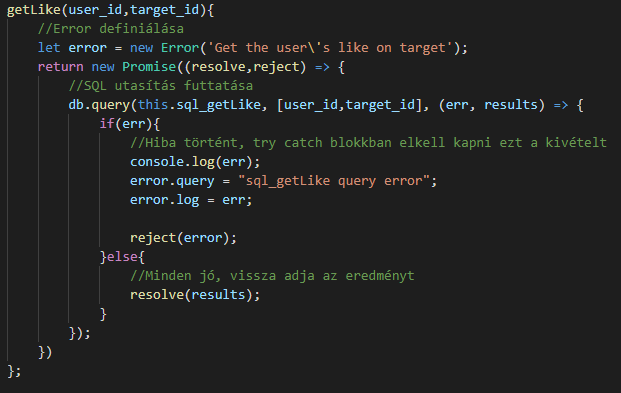
54. ábra: adatbázis kapcsolat létrehozása, exportálása

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

Ezek a fájlok osztályt definiálnak, amik SQL stringeket tartalmaznak és függvényeket amik elvégzik az SQL utasítást és visszaadják annak az eredményét.



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



56. ábra: adatbázissal kommunikáló osztály függvénye

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

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



57. 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 megtudom várni az await kulcsszóval.

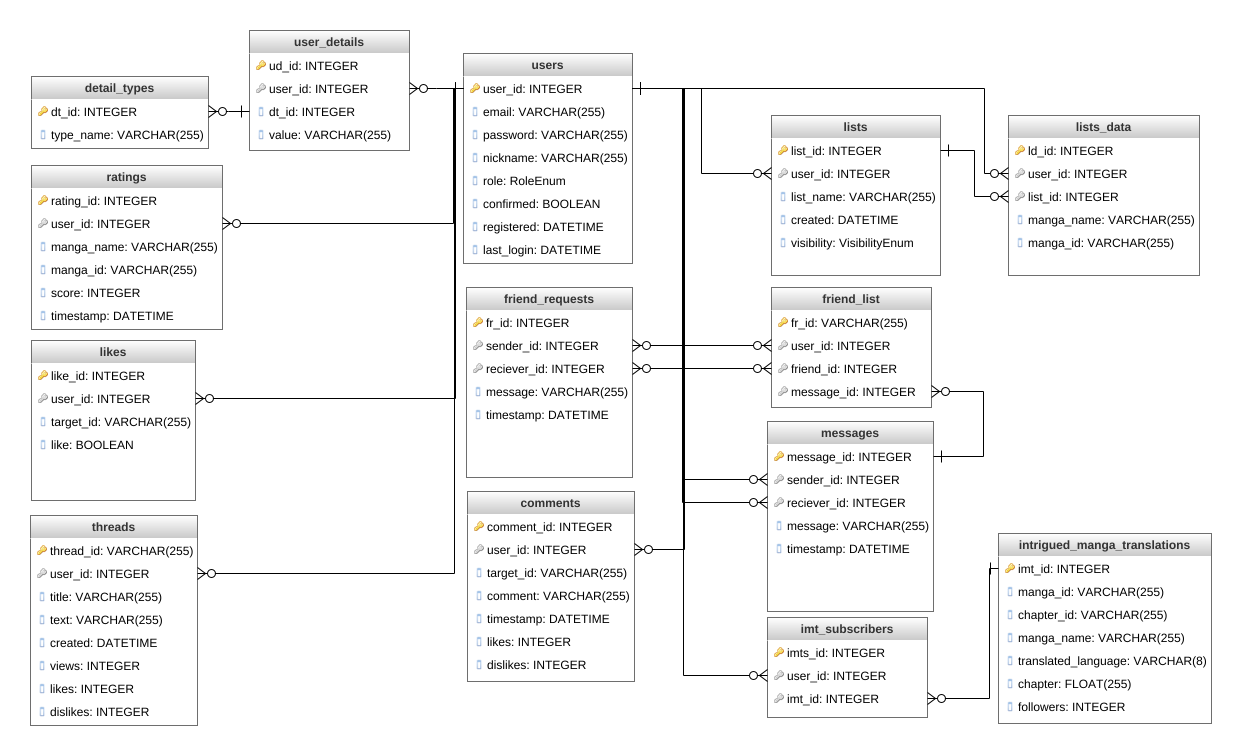


58. ábra: adatbázissal kommunikáló osztály használata

### Adatbázis

#### Struktúra

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



59. ábra: adatbázis felépítése

#### Adatbázis táblák

Egy gyors magyarázat az adatbázis táblákhoz:

* **Users**: Tartalmazza a felhasználó fő adatait, amik szükségesek az alap profilhoz, regisztációhoz. 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, leirás saját magárol, kedvelt és nem kedvelt mangák. Egy segéd táblát használva(detail\_types) több azonos kategóriába tartozó rekordot is feltudunk venni egy felhasználóhoz, ilyen a kedvelt manga „típus” mert egy emberhez több olyan mangát rendelhetünk amit kedvel és ennek az attributumnak(kedvelt manga) az idjét tárolja el mint idegen kulcs.
* **Detail\_types**: A létező felhasználói adatok típusát tárolja, mi a profil leírás, kedvelt, nem kedvelt mangák típust tároljuk, 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, dislikeokat tárol. A target\_id a kommentek, thread-ek id-ját tárolja amihez az adott felhasználó like,dislike ja tartozik.
* **Threads**: Tartalmazza hogy ki hozta létre a fórum threadet, mi a címe, a szövege, mikor hozták létre, mennyien nézték meg és a likeok, dislikeok 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 likeok és dislikeok 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-jét, utolsó fejezet id-jét, a fordítás nyelvét, a manga nevét a fordítás nyelvén(angol fordításhoz angol cím) ha létezik, 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 stringet ami lehet INC\_LIKE, DEC\_LIKE, INC\_DISLIKE, DEC\_DISLIKE és egy SQL case statement végrehatja a megfelelő műveletet a kommenten(like/dislike növelés/csökkentés). A végén vissza adja a kimeneti paraméteren keresztül az eredményt ami lehet 0 = SQL error, 1 = Sikeres változtatás, 2 = Nem lehet 0 alá csökkenteni a like,dislike számot, 3 = Nem megfelelő művelet stringet kapott.
* **Sp\_manage\_likes\_thread**: Ugyan az mint az sp\_manage\_likes, de a fórum threadek 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 stringet ami lehet INC vagy DEC és egy SQL case statement végrehatja a megfelelő műveletet a figyelt manga fordításon(feliratkozó növelés/csökkentés és manga fordítás törlése). A végén vissza adja a kimeneti paraméteren keresztül az eredményt ami lehet 0 = SQL error, 1 = Sikeres változtatás, 2 = Kitörölte a manga fordítás figyelés rekordot, 3 = Nem megfelelő művelet stringet 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

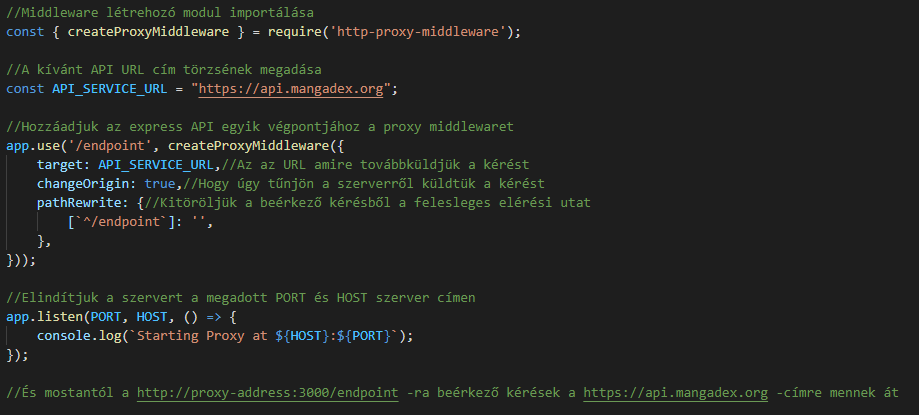
| package-lock.json

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.



60. ábra: proxy szerver beállítása és elindítása

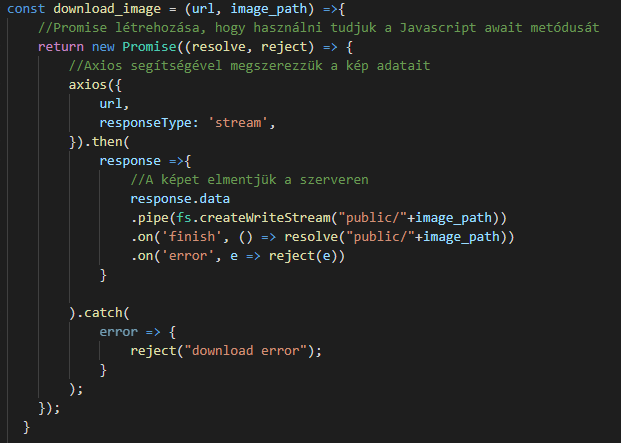
#### MangaDex Képek letöltése

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 borítókép <http://proxy-address:3000/img/:id/:filename> címre, a manga oldal megszerzéséhez a <http://proxy-address:3000/chapter/:hash/:filename> címre küldött GET kéréssel lehetséges. 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.



61. 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



62. 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 szerverről elérjék a képeket, megkell adni az express szervernek, hogy milyen mappából szolgáljon ki statikus tartalmat(képet).



63. 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.

# Továbbfejlesztési lehetőségek

* További szociális funkciók hozzáadása
* Felhasználói felület szépítése
* Fórumok, chat üzenetek, barátlista paginálása
* Frontend akadálymentesítése

# Telepítési útmutató

* Node.Js telepítése
* MariaDB(vagy MySQL) adatbázis szerver telepítése
* Github repó letöltése
* 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, email 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

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]. |