



MATEMATIKAI ÉS INFORMATIKAI INTÉZET

# Közösségi hálózati szolgáltatás fejlesztése MERN architektúrával

**Készítette**

Vereb Barna

Programtervező Informatikus BSc

**Témavezető**

Balla Tamás

Tanársegéd

EGER, 2023

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>4</b>
<b>Bevezetés</b>	<b>4</b>
1.1. Motiváció . . . . .	4
1.2. A munka ismertetése . . . . .	5
<b>2. A fejlesztéssel kapcsolatos információk</b>	<b>6</b>
2.1. JavaScript . . . . .	6
2.2. MERN stack . . . . .	6
2.2.1. MongoDB adatbázis kezelő . . . . .	7
2.2.2. Node.js futtatókörnyezet . . . . .	7
2.2.3. Express.js webes keretrendszer . . . . .	7
2.2.4. React.js frontend keretrendszer . . . . .	8
2.3. Git . . . . .	8
2.3.1. Github . . . . .	8
2.3.2. GitKraken . . . . .	8
2.4. Javascript csomagkezelők . . . . .	9
2.4.1. npm csomagkezelő . . . . .	9
2.4.2. Yarn csomagkezelő . . . . .	9
2.5. Jest . . . . .	9
2.6. Socket.IO . . . . .	10
<b>3. Szoftverdokumentáció</b>	<b>11</b>
3.1. A webalkalmazás felépítése . . . . .	11
3.2. Adatbázis . . . . .	12
3.2.1. Az adatbázis létrehozása . . . . .	12
3.2.2. Collections . . . . .	14
3.3. Backend . . . . .	14
3.3.1. A szerver elindítása . . . . .	14
3.3.2. index.js . . . . .	14
3.3.3. Routes . . . . .	15
3.3.4. Kontrollerek . . . . .	16

3.3.5. Modellek . . . . .	18
3.4. Frontend . . . . .	19
3.4.1. A React szerver . . . . .	19
3.4.2. Redux . . . . .	20
3.4.3. Axios . . . . .	22
3.4.4. React komponensek . . . . .	23
3.5. Middleware . . . . .	27
<b>4. Felhasználói dokumentáció</b>	<b>29</b>
4.1. Reszponzív felhasználói felület . . . . .	29
4.2. Bejelentkezés és Regisztrációs oldal . . . . .	30
4.3. Főoldal . . . . .	31
4.4. Profil oldal . . . . .	31
4.5. Értesítések . . . . .	32
4.6. Felhasználók keresése . . . . .	33
4.7. Poszt oldal . . . . .	34
4.8. Beállítások . . . . .	34
4.9. Üzenetek . . . . .	35
4.10. Témakörök . . . . .	36
<b>Összegzés</b>	<b>37</b>
<b>Irodalomjegyzék</b>	<b>38</b>

# 1. fejezet

## Bevezetés

### 1.1. Motiváció

Lassan már mindenkinél van internet elérése és szinte mindenki aki rendelkezik internettel, az használ valamiféle szolgáltatást, amin keresztül tartja a kapcsolatot szülőkkel, barátokkal, ismerősökkel és igazából bárkivel, gondolok itt cégekre, egyesületekre. Ezért használunk valamilyen közösségi oldalt, ahol kapcsolatainkat fenntarthatjuk, felfrissíthetjük vagy új kapcsolatokat is teremthetünk. Kíváncsiak vagyunk mi történik a nagyvilágban, vagy ismerősökkel. Szeretünk különböző élményeket vagy gondolatokat megosztani másokkal. Ezt mind megtehetjük egy ilyen oldalon. A különböző információkra tudunk reagálni, akár egy kedvelés formájában, vagy kifejthetjük bővebb véleményünket hozzászólásokkal úgy, hogy azt a többi felhasználó is látni fogja, vagy akár privát üzenetben is meg lehet vitatni az adott témát.

Úgy gondolom, hogy az életünk szerves részét képezik már a közösségi média platformok. Vannak olyan személyek, sztárok, akik az életük pillanatait képekkel és videókkal dokumentálják. Alapvetően ebből még nem feltétlen lehetne pénzt keresni, viszont a cégek ezeket próbálják kihasználni, hogy szélesebb körben tudják a termékeit értékesíteni, ezért szoktak ajánlani különböző mértékű partneri szerződéseket, hogy reklámozzák a termékeiket. A nagyobb követőbázissal rendelkező felhasználók ezekből a szerződésekben nagyon jól meg tudnak élni.

Azt vettettem észre, hogy egyre többen használják ezeket az oldalakat a hírek követésére is. Elképzelhető, hogy a jövőben nem lesznek hírportálok, hanem itt fogunk értesülni a különböző hírekről. Példának hoznám a Twittert, ahol van külön szekció a sportesemények követésére. A Twitteren vannak tag-ek, amikkel jobban beazonosíthatók a posztok és ennek köszönhetően könnyebben kapunk számunkra érdekes tartalmakat.

Általános iskolás voltam még, amikor regisztráltam az iWiW-re és a Facebookra. Azóta szinte minden nap használok valamilyen közösségi média oldalt. A sok év használat után úgy gondoltam, hogy jó lenne létrehozni egy saját platformot, így az

eddig tapasztalatokból a jó és fontos funkciókat implementálnám. A projekt tervezésénél fontos volt, hogy milyen eszközöket választok. Próbáltam minél korszerűbb technológiákat felhasználni, amik meggyorsítják a fejlesztés menetét, ezért választottam a MERN fullstack-et. Adatbázisként a MongoDB-t, backend-nek Express.js webes keretrendszer, a frontendnek pedig React.js kliens oldali keretrendszer, a szerverek futtatásához, pedig a Node.js-t webszervert használtam.

## 1.2. A munka ismertetése

A MERN stack erősségeit kihasználva a cél egy közösségi hálózati szolgáltatás, mikroblog és közösségi hírek összesítésére képes weboldal fejlesztése. A felhasználók számítógépen és mobilon is egy webböngészőt használva tudják elérni a reszponzív felhasználói felülettel rendelkező oldalt. Próbáltam az egyszerűségre törekedni, ez az oldal felépítésére, különböző funkciók elérésére és a felhasználói felület kinézetére is igaz. Sötét témát választottam, tapasztalatim alapján a szem nem fárad el olyan könnyen, ha sötét a weboldal témaja és nincs tele vakító színekkel. A tervezett projektet a közismert weboldalak közül a Twitterhez vagy Reddithez tudnám hasonlítani.

Az ötletem, hogy egy regisztráció után a felhasználók különböző témaikban tudjanak létrehozni vagy megtekinteni posztokat. Ezeket lehet kedvelni, ha tetszik, vagy kommenteket is lehet a hozzáadni posztokhoz, akár képekkel. Ahhoz, hogy megkapjuk a többi felhasználó posztjait az idővonalunkra, be lehet követni őket és privát üzeneteiket is lehet írni. Regisztráció után van lehetőség a profilunk testreszabására, profilkép beállítására, leírás megadására a felhasználóról. A regisztrációs adatokat meg lehet változtatni a regisztráció után. A vendég felhasználók, pedig csak a bejelentkező oldalt tudják megtekinteni.

## 2. fejezet

# A fejlesztéssel kapcsolatos információk

A fejlesztéshez felhasznált technológiai eszközökről és ezeknek az előnyeit írom le ebben a fejezetben. Szó lesz a MERN stack komponenseiről, néhány fontosabb felhasznált dependency-ről és arról, hogy miért ezeket választottam. A projekt verziókövetéséhez felhasználtam a Git-et, azon belül a Github és a GitKraken szolgáltatásait vettettem igénybe.

### 2.1. JavaScript

Fontos megemlíteni a JavaScript[1] nyelvet, amin majdnem az összes, a projekthez felhasznált technológia alapszik. A projekt körülbelül 75%-a ezen a nyelven lett megírva. Ez egy gyengén típusos nyelv, dinamikusan rendeli az értékekhez a típusokat. A leggyakrabban használt futtatókörnyezetnek, a Node.js-nek is ez a nyelve, de főleg a kliens oldalak szkriptnyelveként ismert.

### 2.2. MERN stack

A ME(RVA)N stackek variációinak felépítése:

- MongoDB adatbázis kezelő
- Express.js backend webes keretrendszer
- React.js, Vue.js, Angular.js frontend keretrendszer
- Node.js futtatókörnyezet a szervereknek

A MERN[2] stacknél a frontendet nem az Angular.js-el valósítják meg, hanem ezt helyettesítik React.js-el. Említésre méltó még a MEVN stack is ahol a Vue.js a frontend,

ezt a fullstack megoldást is széles körben használják napjainkban. Ezek a keretrendszerek JavaScript nyelven vannak. Felhasználásuk ingyenes és nyílt forráskóddal rendelkeznek. A MERN stack-el igazából bármilyen webes projektet el tudunk készíteni, viszont olyan oldalak készítéséhez ideális, amik dinamikus webes felületet, vagy sok JSON fájlküldést igényelnek. Ezért választottam a MERN stack-et. A fórumok vagy social media platformok, pont ilyen oldalak.

### **2.2.1. MongoDB adatbázis kezelő**

A MongoDB[3] a MERN stack platformfüggetlen adatbázis-kezelő rendszere. A MongoDB egy NoSQL típusú adatbázis kezelő, JSON-szerű dokumentumokat használ opcionális sémákkal. Jól skálázható, gyorsak a lekérdezések az adatmodellnek köszönhetően, könnyű használni a fejlesztők számára. A MongoDB Inc. fejleszti, és SSPL licenc alatt áll. Több kiadása létezik, én a projekthez a MongoDB Atlas webes szolgáltatást használom, ami az AWS-en, Microsoft Azure-n és a Google Cloud platformján fut. Ez a legfejlettebb felhőalapú adatbázis-szolgáltatás jelenleg. Több régióban is lehet adatbázisokat létrehozni. Ha kíváncsiak vagyunk, szinte bármilyen információra az adatbázis kapcsolattal, vagy tárolt adatokkal kapcsolatban, akkor sok valós idejű információt tartalmaz a MongoDB Atlas webes szolgáltatás. Ennél a projektnél használtam először, viszont az atlas oldala és annak működése nagyon megtetszett, ezért más projekteknél is ezt a szolgáltatást választottam a későbbiekben.

### **2.2.2. Node.js futtatókörnyezet**

Webkiszolgálókat és hálózati eszközöket lehet létrehozni a segítségével, JavaScript nyelven. Nyílt forráskódú és több platformon is használható. Eseményvezérelt architektúrával rendelkezik, ami képes aszinkron input/output feldolgozásra, amely lehetővé teszi újabb feldolgozások indítását, mielőtt az előző befejeződne. A valós idejű webes alkalmazásokhoz kínál optimalizált átviteli teljesítményt és skálázhatóságot.[4]

### **2.2.3. Express.js webes keretrendszer**

Az Express.js[5] keretrendszer segítségével RESTful API-kat készíthetünk egy backend webes alkalmazáshoz. Használata ingyenes és nyílt forráskóddal rendelkezik. Segítségével könnyebbé tehetjük az alkalmazásunkban a routingot és a middleware-k használatát. A Node.js *de facto* szabványos kiszolgáló keretrendszerének nevezik. Nagyobb cégek is használják, többek között a Paypal, az Uber és az IBM is.

## **2.2.4. React.js frontend keretrendszer**

A React[6] egy ingyenes és nyílt forráskódú frontend keretrendszer a webes és natív felhasználói felületekhez. A Meta (előző nevén Facebook) és egy egyéni fejlesztőkből álló közösség tartja fent a kódmezőt. A React a deklaratív programozási paradigmát követi. Nézeteket hozhatunk létre az alkalmazás minden egyes állapotához, a React pedig frissíti és megjeleníti a különböző részeket, amikor az adatok változnak. Lehetővé teszi, hogy a felhasználói felületet kisebb modulokból, úgynevezett komponensekből építük fel. Ez nagyon meggysorsítja a fejlesztést, mert egy komponenst többször is fel lehet használni. Ezek valójában JavaScript függvények, így függvényhívásokkal tudunk megjeleníteni komponenseket. A React szintaxisa a JSX, ez a React fájlok kiterjesztése is.

## **2.3. Git**

Az egyetemi programozási feladataim során már többször előfordult, hogy egy projektben többen kellett egyszerre dolgoznunk. A Git[7] ezt teszi könnyebbé, a forrásfájlok változásának nyomon követésével. A Git megnöveli a nagy projektekk fejlesztési sebességet. Ennek köszönhetően a Git ma már a legszélesebb körben használt forráskód-kezelő eszköz. Számos szolgáltatásként kínált Git-tárhely létezik, a legnépszerűbbek a Github és a SourceForge.

### **2.3.1. Github**

A Github[8] egy internetes tárhelyszolgáltatás szoftverfejlesztéshez és verziókezeléshez a Git rendszer segítségével. A Git az elosztott verziókövetést biztosítja, ezen felül a Git-hubon lehet szabályozni a repository-khoz való hozzáférést, van hibakövetési funkció és akár leírást is lehet adni a különböző projekthez. Gyakran használják nyílt forráskódú szoftverfejlesztési projektek tárolására. Használata nagyon egyszerű, egy regisztráció után lehet létrehozni repository-kat, ahol a fájlokat, projektet tárolhatjuk. Ha ezek módosítását szeretnénk végrehajtani, akkor először le kell klónoznunk a Github-ról. Utána amikor valamilyen változtatást hajtunk végre a helyi mappában, létre kell hozni egy commitot, mielőtt fel szeretnénk tölteni a módosított vagy új fájlokat. Egy commitban általában az szerepel, hogy milyen tevékenységet csináltunk éppen az adott fájlokon. Ezt követően a commitokat fel lehet pusholni a Github-ra.

### **2.3.2. GitKraken**

A Git beépített GUI eszközökkel rendelkezik a commitoláshoz és a repository-k böngészéséhez. Én egy másik eszközt használtam, a GitKrakent. Ez is ugyanazokat a felada-

tokat látja el, viszont számomra a felhasználói felülete szímpatikusabb volt. Ingyenes verziója is létezik, diákként azonban lehet használni a Pro verziót is, ami egyébként fizetős lenne. Ez az alkalmazás nagyon megkönnyítette, hogy nem kellett minden terminálon keresztül commitolni, pullolni és pusholni.

## 2.4. Javascript csomagkezelők

Olyan eszközök a csomagkezelők, amelyek segítségével a fejlesztők automatizálni tudják a rendszerhez beszerezhető csomagok(dependency) megtalálását, telepítését, letöltését, konfigurálását, frissítését és eltávolítását. Sok ilyen létezik, én az npm-et és a Yarn-t használtam a fejlesztés során.

### 2.4.1. npm csomagkezelő

Az npm[9] a Node.js alapértelmezett csomagkezelője. Az npm egy parancssori kliens és egy online adatbázis, ahol fizetős vagy nyilvános csomagokat lehet böngészni. Különböző konfigurációkat is lehet beállítani, én a backend és a socket.io szerveremnek az indításához készítettem egy scriptet.

### 2.4.2. Yarn csomagkezelő

Az egyik fő JavaScript csomagkezelő, melyet a Meta fejleszt. Az npm csomagkezelő alternatívája, a Yarn[10] a Facebook, az Exponent és a Google együttműködésével jött létre, hogy megoldja a nagy kód-bázisok konziszenciá-, biztonsági és teljesítményproblémáit. A frontend szerverhez használtam a fejlesztés során.

## 2.5. Jest

A Javascripthez egy tesztelési keretrendszer a Jest[11], a Meta cég fejleszti. A Reactben ez egy alapértelmezett függőség, nem igényel sok konfigurációt, így könnyű a használata. Lehet vele készíteni kimutatást ami sok információt tartalmaz azzal kapcsolatban, hogy a tesztjeink a kód mekkora részét fedik le. Információban gazdag leírást kapunk, ha egy teszt nem megy át. A teszteket a teljesítmény maximalizálása érdekében a saját folyamatokban futtatva párhuzamosítja. Sok eszközzel szolgál, hogy a várt eredményt összehasonlítsuk a valós eredménnyel. A Jest-et a frontenes unit test-ekhez használtam.

## 2.6. Socket.IO

A projekt tervezése során szükségét láttam a valós idejű üzenetküldés megvalósításának, mivel ez egy elengedhetetlen része a közösségi média platformoknak. Ehhez használom a Socket.IO-t[18]. Lehetséges, hogy a kétirányú kommunikációt a webes kliens és a webszerver között. Két része van, az egyik a kliens oldalon, a másik pedig a szerver oldalon fut. A Socket.IO átláthatóan kezeli a kapcsolatokat és ha lehetséges akkor azokat automatikusan WebSocketekre cseréli. A WebSocket azért jó, mert full-duplex kommunikációs csatornákat biztosít egyetlen TCP-kapcsolaton keresztül.

## 3. fejezet

# Szoftverdokumentáció

A projekt elkészítésének kihívásairól és az implementált részek működéséről lesz szó ebben a fejezetben. A projekt felépítéséről és annak az elemeiről fogok részletesen beszélni. Megpróbálom olyan sorrendben bemutatni a részeket, mint ahogyan készültek.

### 3.1. A webalkalmazás felépítése

Az alkalmazás tervezésénél a MERN architektúra lényegében három főbb részből áll. Az adatbázis kezelő, a backend és a frontend. A valós idejű üzenetküldéshez szükség van még egy szerverre, ami a backend és a frontend között működik. Az alkalmazás készítését az adatbázis kezelő beüzemelésével kezdtem. A MongoDB Atlas szolgáltatását választottam, ezért ide be kellett regisztrálnom. Az adatbázis sikeres létrehozása után az Express backend szerver alapjait készítettem el. Miután sikeresen küldtem adatot a mongodb szervernek és az eltárolta az információkat, megcsináltam a frontdes React szerver alapját.

A szerverek struktúráját próbáltam úgy felépíteni, hogy azokat könnyen lehessen bővíteni a későbbiekben.

A mappák szerkezete:

- client (React szerver)
  - public (itt található az alap HTML dokumentum)
  - src (itt található a szervert elindító fájl és a többi hozzá tartozó komponens)
    - actions (az állapotváltozást irányító logikák)
    - api (a REST API hívások)
    - components (a React komponensek az oldalakhoz)
    - img (az oldalon felhasznált képek)
    - pages (az alkalmazás oldalai)
    - reducers (a reducerek logikái)

- store (a Redux tároló)
- server (Express szerver)
  - Controllers (a végpontok működéseinek implementációi)
  - Middleware (JWT[21] (JSON WEB Token) érvényességét ellenőrző függvény)
  - Models (az adatbázis modeljei)
  - public\images (a felhasználók által feltöltött képek)
  - Routes (REST API végpontok)
- socket (Socket.IO szerver)

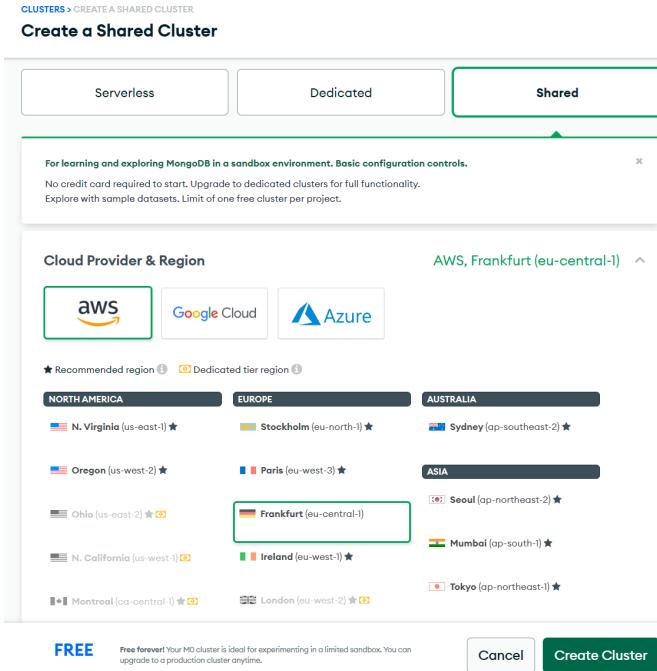
## 3.2. Adatbázis

Adatbázisnak, a már sokszor említett MongoDB Atlas ingyenes adatbázis szolgáltatását választottam. Azért gondoltam jó választásnak, mert ezt a felhőalapú adatbázist bárhonnan el tudom érni.

### 3.2.1. Az adatbázis létrehozása

Az MongoDB oldalán, sikeres regisztrációt követően lehet létrehozni projektet. Egy projektbe lehet felhasználókat vagy akár csapatokat is hozzáadni, ha többen szeretnének a projekt adatbázisait használni. Miután kész a projekt, Clustereket lehet hozzáadni, amik az adatbázisokat tartalmazzák.

Három típusú Clustert tudunk létrehozni. Az első a szerver nélküli instance, amit alkalmazásfejlesztéshez és teszteléshez ajánlanak. Ez egy fizetős szolgáltatás. Előnye, hogy az erőforrások jól skálázódnak és csak az elvégzett műveletekért kell fizetni. A második a dedikált Cluster. Ez is egy fizetős szolgáltatás. Sokrétű konfigurációt ígérnek a kiadásra szánt alkalmazásokhoz. A harmadik, amit választottam az adatbázisom elkészítéséhez, a megosztott Cluster. Ezt tanuláshoz és a MongoDB környezetének fel-fedezéséhez ajánlják. Tartalmazza az alapvető konfigurációs lehetőségeket. Egy projekten belül ebből csak egyet lehet létrehozni. A használata ingyenes, mivel megosztott erőforrásokat használ. Az ingyenes változat 512MB tárhelyet, megosztott RAM-ot és virtuális CPU-t kínál. A Cluster létrehozásához ki kell választanunk, hogy melyik felhőszolgáltatónál és milyen régióban legyenek majd az adatbázisaink. Érdemes olyat választani, ami a backend szerverünkhez legközelebb helyezkedik el, hogy az elérési időt minimalizáljuk. Én az Amazon AWS felhőszolgáltatását választottam, Frankfurt régióban.



3.1. ábra. Új Cluster létrehozása

Adatbázisokat ezt követően tudunk készíteni. A Cluster nevére kattintva megjelennek az adott Clusterhez való információk és beállítási lehetőségek. A Collections fül fog nekünk kelleni, itt tudunk új adatbázist hozzáadni. Ilyenkor meg kell adni az új adatbázis nevét és egy benne szereplő gyűjtemény nevét is. A gyűjteményeket az SQL-ból ismert adattáblákhoz tudnám hasonlítani. Ha rákattintunk egy adatbázis nevére, akkor a benne szereplő gyűjteményeket és a hozzájuk tartozó információkat fogjuk látni. Itt tudunk akár újat is hozzáadni. Én a backend szerveren elkészített sémákkal adtam hozzá a gyűjteményeket.

Collection Name	Documents	Logical Data Size
chats	5	677B
messages	28	4.85KB
notifications	19	2.83KB
posts	19	4.91KB
topics	3	455B
users	5	1.89KB

3.2. ábra. Adatbázisban szereplő gyűjtemények (collections)

### 3.2.2. Collections

A MongoDB a dokumentumokat gyűjtemények (collection)[12] formájában tárolja. Ahogy már említettem, egy gyűjtemény a relációs adatbázis kezelőknél az adattáblának a megfelelője. A MongoDB létrehozza nekünk azokat a gyűjteményeket, amik még nem szerepelnek az adatbázisban, viszont mi valamilyen adatot töltünk fel a nem-létező gyűjteménybe.

Nagyon jó tulajdonsága a gyűjteményeknek, hogy a felépítésükön lehet változtatni létrehozásuk után is. Több alkalom is volt, hogy fejlesztés közben jutott eszembe egy hiányzó mező, amit aztán a gyűjtemény felépítéséhez csatoltam. Egy gyűjteményben szereplő elemeknek nem kell ugyan azokkal a mezőkkal szerepelnie. Az SQL-hez hasonlóan itt is meg lehet adni, hogy mely mezőknek kell kötelezően szerepelnie.

## 3.3. Backend

### 3.3.1. A szerver elindítása

Az alkalmazás backend szerverének egy JavaScript alapú Express.js szervert használunk. Miután feltelepítettem a szerver megfelelő működéséhez szükséges függőségeket, utána elkezdhetem az indítás konfigurációjának a beállítását. A szerver elindításához a Nodemon csomagot használom, ehhez készítettem egy szkriptet. Ha el szeretném indítani a szervert, elég csak a terminálba beírnom, hogy *npm start*. Ezt a szkriptet a *package.json* nevű fájlban szereplő *scripts* objektumhoz adtam hozzá. A szkriptben pedig a következő utasítás szerepel: *nodemon index.js*, ez igazából a Nodemon segítségével futtatja az *index.js* fájlt. A *package.json* fájlban tárolódnak a különböző függőségeknek a verziói is. Itt lehet megadni a szerver nevét és leírását is. A nodemon paranccsal elindított fájlok a futtatást követően újra lesznek indítva, ha valamilyen változás történik a fájlt tartalmazó mappában. Ez a függőség egyszerűbbé tette a fejlesztést, mivel nem kellett nekem minden leállítom és újraindítanom a szervert.

### 3.3.2. index.js

A szerver *index.js* fájlján keresztül történik az Express.js szerver indítása. Beállítom a *body-parser* csomag *json* és *URL-encoded* hívások limitjét és azt, hogy a kérés tartalmazhat bármilyen értékeket, nem csak String típust.

Listing 3.1. index.js részelete

```
1 const app = express();  
2  
3 app.use(bodyParser.json({ limit: "30mb", extended:  
    ↳ true }));
```

```

4 app.use(bodyParser.urlencoded({ limit: "30mb",
5   ↪ extended: true }));
6 app.use(cors());
7
8 app.use(express.static("public"));
9 app.use("/images", express.static("images"));
10
11 dotenv.config();
12
13 mongoose
14   .connect(process.env.MONGO_DB, {
15     useNewUrlParser: true,
16     useUnifiedTopology: true,
17   })
18   .then(() =>
19     app.listen(process.env.PORT, () =>
20       console.log(`Listening at ${process.env.PORT}`)
21     )
22   )
23   .catch((error) => console.log(error));
24
25 app.use("/auth", AuthRoute);
26 app.use("/user", UserRoute);
27 app.use("/post", PostRoute);
28 app.use("/upload", UploadRoute);
29 app.use("/chat", ChatRoute);
30 app.use("/message", MessageRoute);

```

Ezt követően bekapcsolom a CORS-t, ami lehetővé teszi, hogy a szerver elérhető legyen más eredetű tartományokból is. Erre azért van szükség, mert külön van a frontend szerver. A következő sorokban, pedig a szerveren tárolt képek elérésének a helyét határozzom meg. A dotenv csomag segítségével tudok hivatkozni az `.env` fájlban tárolt környezeti változókra. Ezek a változók olyan adatokat tartalmaznak, amiket nem szívesen írnánk le a kódba. Én itt tárolom az adatbázis csatlakozásához szükséges információkat, a szerver portját és a JWT kulcsát.

A mongoose csomag segítségével tudok az adatbázishoz csatlakozni, vagy sémákat létrehozni. Az adatbázishoz való csatlakozás után a megadott porton fog a szerver figyelni a kérésekre. A kód végén adom hozzá az API-s különböző funkcióknak az elérési útjait és meghívom a hozzá tartozó *Route* fájlt.

### 3.3.3. Routes

Egy ilyen alkalmazás esetében kezelni kell a felhasználókat, a felhasználók által létrehozott bejegyzéseket, a feltöltött képeket és az üzeneteket is. Ennek megfelelően neveztem

el a REST API végpontjait. Az első, amit létrehoztam a felhasználó autentikációjáért felel.

Listing 3.2. Az *AuthRoute.js* kódja

```
1 import express from "express";
2 import { registerUser, loginUser } from "../
3   ↪ Controllers/AuthController.js";
4
5 const router = express.Router();
6
7 router.post("/register", registerUser);
8 router.post("/login", loginUser);
9
10 export default router;
```

Ebben a fájlban nincsen sok végpont definiálva. Két út[13] szerepel, a regisztráció és bejelentkezés. Ahhoz, hogy működjenek, a kontrollereknél definiált függvényeket meg kell hívni a megfelelő végponthoz. Definiálni kell az adott végpont HTTP metódusát is, amik ebben az esetben POST metódusok. A kód végén exportálnom kell a *router* változót, hogy felhasználhassuk az *index.js* fájlban.

Listing 3.3. *PostRoute.js* részelete

```
1 router.post("/", createPost);
2 router.get("/:id", getPost);
3 router.put("/:id", updatePost);
4 router.put("/:id/like", likePost);
5 router.get("/:id/timeline", getTimeLinePosts);
6 router.get("/:id/comments", getComments);
7 router.put("/:id/comments", updateComments);
8 router.get("/:id/user", getUserPosts);
9 router.get("/:id/topicposts", getTopicPosts);
10 router.get("/:id/trending", getTrendingTopics);
11 router.get("/:id/topicSearch", topicSearch);
```

A *PostRoute.js* fájlban szereplő utaknál lehet látni, hogy az URL-be meg lehet adni változókat is. Ezeket az adatokat fel tudom majd a kontrollereknél használni.

### 3.3.4. Kontrollerek

A *Controllers* nevű mappában vannak a kontrollerek[13]. minden *Routes* mappában lévő fájlhoz tartozik egy kontroller fájl is. A végpontok meghívásakor lefutó függvények vannak a kontrollerekben definiálva. Ezeket a függvényeket aszinkron kell definiálni. Paramétereiknek egy kérés és egy válasz mezőt kell definiálni. A kérés teste tartalmazhatja a feldogozandó JSON sztringet.

Amikor a felhasználó rákeres egy témara, akkor a *topicSearch* nevű függvény fog lefutni. A kérés paramétereiben szereplő keresési adatot eltárolom a *searchParam* nevű konstansba. Ezt követően egy trycatch blokkot hozok létre és lekértem a keresési paramétert kielégítő témaat. Az *await* kulcsszót kell használni, hogy megvárja az eredményét a lekérdezésnek. A lekérdezésben a téma nevére egy reguláris kifejezéssel keresek rá. Azokat a témaat fogom megkapni eredményként, amik nevében szerepel a keresett sztring.

Listing 3.4. Egy téma keresése

```

1 export const topicSearch = async (req, res) => {
2     const searchParam = req.params.id;
3     try {
4         const topics = await TopicModel.aggregate([
5             {
6                 $match: {
7                     name: new RegExp(".*" + searchParam +
8                         ".*"),
9                 },
10                { $project: { name: 1, numberOfTopics: { $size
11                    $: "$posts" } } },
12            ]);
13            res.status(200).json(topics);
14        } catch (error) {
15            res.status(500).json(error);
16        }
17    };

```

Eredménynek a témahoz tartozó bejegyzések számát és a téma nevét szeretném megkapni. Ezért a *numberOfTopics* mezőben a *posts* mező hosszát, azaz a benne szereplő adatok számát kérem le. Ezt követően a válasz státuszát 200-ra, azaz OK-ra állítom és visszaküldöm JSON formátumban a keresési eredményt. Ha valami hiba történik, akkor 500-as státuszt küldök vissza és a hibaüzenetet. Nagyjából ez bármilyen adat lekérdezésekor így zajlik.

Egy másik példa, amikor egy hozzászólást hozok létre és azt egy bejegyzéshez kell hozzácsatolnom. Szükségem van két azonosítóra, a hozzászóláséra és a bejegyzésére, ahoz hogy ezeket összekapcsoljam.

Listing 3.5. Új hozzászólás hozzáadása egy bejegyzéshez

```

1 export const updateComments = async (req, res) => {
2     const postId = req.params.id;
3     const { commentId } = req.body;
4     try {
5         const post = await PostModel.findById(postId);

```

```

6   if (!post.comments.includes(commentId)) {
7     await post.updateOne({ $push: { comments:
8       ↪ commentId } });
9     res.status(200).json("Comment added!");
10    } else {
11      res.status(400).json("Error!");
12    }
13  } catch (error) {
14    res.status(500).json(error);
15  };

```

A *post* nevű konstansba eltárolom a bejegyzés adatait, ezt követően ha a hozzászólás azonosítója még nem szerepel a *comments* tömbben, akkor belerakom, különben nem.

### 3.3.5. Modellek

A modellek tartalmazzák az adatbázisban tárolt gyűjteményeknek a struktúráját. Egy modell létrehozásához először egy sémát kell készíteni. Sémákat és modelleket a mongoose csomaggal tudok készíteni a MongoDB adatbázisomhoz. A mongoose sémát a *Schema()* függvényel lehet létrehozni. Ebben megadom a számomra szükséges mezőket. A *model()* függvény két mezőt kér, az elsőbe a modell nevét kell megadni, a másodikba az elkészített sémát.

A *PostSchema* nevű séma felel a hozzászólások és a bejegyzések struktúrájáért. Nagyon nagy a hasonlóság a két elem között. A hozzászólásnak van egy szülője, ami a bejegyzés, viszont nincsenek hozzászólásai. Ezeknek a mezőknek a létrehozását nem teszem kötelezővé, így amikor létrehozom az objektumot, elég csak a szükséges mezőket definiálnom.

Listing 3.6. A *Posts* modell

```

1 import mongoose from "mongoose";
2
3 const PostSchema = mongoose.Schema(
4   {
5     userId: { type: String, required: true },
6     desc: String,
7     likes: [],
8     image: String,
9     comments: [],
10    categories: [],
11    parent: { type: String, required: false },
12  },
13  {
14    timestamps: true,
15  }

```

```

16 );
17
18 var PostModel = mongoose.model("Posts", PostSchema);
19 export default PostModel;

```

Az *image* mező sem kötelező, mivel sem egy hozzászólásba sem egy bejegyzésbe nem kötelező képet rakni. Érdemes az objektum létrehozásának és átírásának az idejét elmenteni. Ezt egyszerűen megtehetjük, ha a *timestamps* nevű mezőt igazra állítjuk. Ennek köszönhetően ha a későbbiekben változik az objektum valamely része, akkor az *updatedAt* nevű mező is frissülni fog. A létrehozási időpont a *createdAt* nevű mezőben fog eltárolódni.

Listing 3.7. *Topics* modell

```

1 import mongoose from "mongoose";
2
3 const TopicSchema = mongoose.Schema(
4   {
5     name: { type: String, required: true },
6     posts: [] ,
7   },
8   {
9     timestamps: true ,
10   }
11 );
12
13 var TopicModel = mongoose.model("Topics", TopicSchema)
14 ↵ ;
14 export default TopicModel;

```

A *Topics* modellnél ha létre szeretnék hozni egy objektumot akkor abban szerepelnie kell név mezőnek is. Ha az objektumot perzisztálom, akkor kapni fog egy egyedi azonosítót is. Ennek a neve *\_id*, ami egy ObjectId típusú mező.

## 3.4. Frontend

### 3.4.1. A React szerver

Egy alap React szerver elkészítéséhez az *npx create react* parancsot kell a terminálba beírni. Ez valószínűleg túl sok dolgot fog felrakni egyszerre, ezért a nem szükséges dolgoktól érdemes megszabadulni. Ezt követően a *yarn* parancssal javasolt a többi csomagot feltelepíteni, mert ez gyorsabb, mint az *npm*. Egy szerveren belül nem ajánlott mind a kettőt használni, mert konfliktusok alakulhatnak ki a telepített csomagok között. A szervert a *yarn start* parancssal tudjuk elindítani.

A szerveren található *index.js* fájlban szereplő *react-dom* csomag segítségével tudok az alkalmazás felső szintjén DOM-specifikus metódusokat létrehozni.[14] Ezen belül fogom a létrehozott modulokat, komponenseket használni. A *react-redux* és a *react-router-dom* csomagjait is itt fogom használni.

### 3.4.2. Redux

A React Redux[15] csomagot a kliens oldali állapotváltozások kezelésére használom. A *yarn-add react-redux* parancsal tudom hozzáadni, ha még eddig nem volt hozzáadva a csomagokhoz. Szükségem lesz a *redux-thunk* csomagra is. A store-ba tárolom az összes állapotot, ezen belül a reducerek tárolják az állapotokat. Én négy reducert használom, a bejegyzéseknek, hozzászólásoknak, értesítéseknek és a felhasználó adatainak. Ezeknek a felépítését az adott típushoz igazítom. A hozzászólásoknak van egy *loading*, *error*, *uploading* és *comments[]* állapota. Ezeket szerintem nem fontos nagyon magyarázni, mert a nevükön rá lehet jönni, hogy mire használom őket. Ezen felül van még egy *action* mező is, ami leírja, hogy az állapotokkal különböző esetekben mik fognak történni.

Listing 3.8. *CommentReducer.js* tartalma

```

1  const commentReducer = (
2    state = {
3      comments: [],
4      loading: false,
5      error: false,
6      uploading: false,
7    },
8    action
9  ) => {
10   switch (action.type) {
11     case "UPLOAD_COMMENT_STARTED":
12       return { ...state, uploading: true, error:
13           false };
14
15     case "UPLOAD_COMMENT_SUCCESS":
16       return {
17         ...state,
18         comments: [action.commentData, ...state.
19             comments],
20         uploading: false,
21         error: false,
22       };
23
24     case "UPLOAD_COMMENT_FAIL":
25       return { ...state, uploading: false, error:
26           true };
27   }
28 }
```

```

25     case "LOADING_COMMENT_START":
26         return { ...state, loading: true, error: false
27             ↵ };
28
29     case "LOADING_COMMENT_SUCCESS":
30         return {
31             ...state,
32             comments: action.commentData,
33             loading: false,
34             error: false,
35         };
36
37     case "LOADING_COMMENT_FAILURE":
38         return { ...state, loading: false, error: true
39             ↵ };
40
41     default:
42         return state;
43     }
44 };

```

Állapotváltozást a React komponensek vagy API hívás idéz elő. A store-ból ki tudom olvasni a jelenlegi állapotot. Ha valami olyan történik az oldalon, aminek hatására változnia kell az állapotnak akkor ennek az egyik *action* végre fog hajtódni és megváltozik az állapot. A komponensekben az állapotváltozáshoz a *dispatch()* hook-ot fogom használni. Az *actions* nevű mappában vannak a *dispatch* függvények logikáinak az implementációi.

Listing 3.9. *CommentAction.js* tartalma

```

1 import * as PostApi from "../api/PostRequest"
2
3 export const getComments = (id) => async (dispatch) =>
4     ↵ {
5         dispatch({ type: "LOADING_COMMENT_START" });
6         try {
7             const { data } = await PostApi.getComments(id);
8             dispatch({ type: "LOADING_COMMENT_SUCCESS",
9                 ↵ commentData: data });
10        } catch (error) {
11            dispatch({ type: "LOADING_COMMENT_FAILURE" });
12            console.log(error);
13        }
14    };

```

Látható, hogy ez REST API hívást is végez, ezért aszinkron. Az állapot minden adott cselekménynek megfelelően fog változni. Érdemes jól elnevezni az *action* típusait,

mivel, ha több reducert használunk akkor egy idő után nehéz lehet megkülönböztetni őket. Éppen ezért a névbe beleraktam, hogy mit csinál, min csinálja és éppen melyik rész következik.

### 3.4.3. Axios

A frontend és a backend között biztosítani kell a folyamatos kommunikációt. Az Axios[16] egy ígéret-alapú HTTP kliens. A kliens oldalon XMLHttpRequest hívásokat használ, a szerver oldalon pedig a natív node.js HTTP modult. A szerverünkhez a *yarn add axios* parancssal tudjuk hozzáadni. Ezek a hívások az *api* mappában lévő fájlokban vannak implementálva.

A komponensekből vagy az állapotváltozáshoz szükséges esemény bekövetkeztekor hívódnak meg ezek az API hívások. Első lépésként beimportálom a szükséges csomagot. Ezt követően létrehozok egy új axios példányt, aminek a *baseURL*-jét átállítom a szerveremnek az IP címére és portjára. Érdemes statikus IP címet megadni a szerverünknek. Ha valamilyen ellenőrzést szeretnénk végrehajtani, mint például a JWT érvényességét, akkor egy interceptor metódust kell használnunk. Ezek a fő metódus előtt vagy után fognak végrehajtódni. Két fajta létezik, az egyik a hívás előtt fut le, a másik pedig mielőtt a válasz eléri a hívó félt. Ebben az esetben a hívás előtti interceptor használtam.

Listing 3.10. *UserRequest.js* tartalma

```

1 import axios from "axios";
2
3 const API = axios.create({ baseURL: "http
4   ↪ ://192.168.1.107:5000" });
5
6 API.interceptors.request.use((req) =>
7   if (localStorage.getItem("profile")) {
8     req.headers.authorization = `Bearer ${
9       JSON.parse(localStorage.getItem("profile"))
10      ↪ token
11    }`;
12  }
13  return req;
14});
15
16 export const getUser = (data) => API.get(` / user/${data
17   ↪ }`);
18 export const getUserByUsername = (data) => API.get(` /
19   ↪ user/${data}/username`);
20 export const updateUser = (id, data) => API.put(` / user
21   ↪ /${id}`, data);

```

```

17 export const getRecommendedUsers = (id) => API.get(`/
    ↵ user/${id}/recommended`);
18 export const followUser = (id, data) =>
    API.put(`user/${data}/follow`, { currentUserId: id
        ↵ });
20 export const unFollowUser = (id, data) =>
    API.put(`user/${data}/unfollow`, { currentUserId:
        ↵ id });
22 export const searchUser = (searchData) => API.get(`/
    ↵ user/${searchData}/search`);
23 export const getNotifications = (id) => API.get(`user
    ↵ /${id}/notification`);
24 export const updateNotification = (id) => API.put(`/
    ↵ user/${id}/notification`);
```

A felhasználó tokenjét ellenőrzöm mielőtt a hívás megtörténne és ha érvényes tokent használ akkor a hívás is meg fog történni, különben nem. Azt, hogy melyik végponton fog az ellenőrzés megtörténni, a backend szerveren kell megadni. Az én esetben azokon a végpontokon fogom ellenőrizni a token érvényességét, amely valamilyen módosítást hajtana végre az adatbázisban. A hívásokat exportálom, hogy fel tudjam használni. Kiválasztom, hogy milyen fajta HTTP hívásnak kell végrehajtódnia. Itt tudom megadni, hogy ha az URL-ben kell átadnom valamilyen adatot. Az URL ha két ‘jel közé írom, akkor a \${} belül megadott adatot a sztringbe fogja belefűzni. Ha a hívás testébe kell átadnom adatot, akkor azt az URL utáni paraméterben tudom megtenni.

### 3.4.4. React komponensek

Amikor ránézünk egy React-os weboldalra, akkor az több kis méretű komponensből[17] fog felépülni. Az oldalt, ha legalább egy 1024 pixel szélességű kijelzőn nézzük akkor három részre lesz felosztva, ha 1024 pixelnél kisebb, akkor csak egyre. Ezekben a részeken belül fognak a komponensek elhelyezkedni. A komponenseket be kell importálni. Ezek igazából JavaScript függvények. A *Topic* oldalon az adott témának a bejegyzéseit lékérem és azokat megjelenítem. A *BurgerComponent* csak akkor fog megjelenni, ha kis kijelzőn van megjelenítve az oldal. Ezekben a komponensek további kisebb komponensekből állnak. Ez lehetővé teszi, hogy a kis komponenseket többször is felhasználjuk a különböző helyzeteknek megfelelően.

Listing 3.11. *Topic.jsx* oldal kódjának tartalma

```

1 import React, { useEffect, useState } from "react";
2 import LeftSide from "../../components/leftSide/
    ↵ LeftSide";
3 import RightSide from "../../components/rightSide/
    ↵ RightSide";
4 import "./Topic.css";
```

```

5 import BurgerComponent from "../components/
  ↪ burgerComponent/BurgerComponent";
6 import * as PostApi from "../api/PostRequest";
7 import Post from "../components/postComponent/
  ↪ PostComponent.jsx";
8 import { useParams } from "react-router-dom";
9 const Topic = () => {
10   const [posts, setPosts] = useState([]);
11   const { id } = useParams();
12
13   useEffect(() => {
14     const fetchPosts = async () => {
15       const { data } = await PostApi.getTopicPosts(id)
         ↪ ;
16       setPosts(data);
17     };
18     fetchPosts();
19   }, []);
20
21   return (
22     <div className="Topic">
23       <BurgerComponent />
24       <LeftSide />
25       <div className="Posts">
26         {posts.map((post) => {
27           return <Post post={post} key={post._id} />;
28         })}
29       </div>
30       <RightSide />
31     </div>
32   );
33 };
34
35 export default Topic;

```

A *Posts* osztállyal ellátott div-ben fognak szerepelni az oldal közepén a bejegyzések. Ezeket a *map()* segítségével jelenítem meg.[20] minden egyes bejegyzésnél megjeleníték egy *PostComponent* nevű komponenst, aminek átadom a bejegyzés adatait. Ezek egy oszlopban egymás alatt fognak sorban megjelenni.

A *PostComponent*-ben nem elég számonra a bejegyzéshez tárolt információ. Egy bejegyzés megjelenítésénél szeretném, ha a szerzőjének a neve és felhasználóneve is látszódna. Emiatt ezeket az adatokat lekérdezem minden egyes bejegyzésnél. A létrehozás dátumát átalakítom egy könnyen olvasható formátumra.[19]

Listing 3.12. *PostComponent.jsx* kódjának részlete

```

1  const date = new Date(post.createdAt).
2      ↪ toLocaleDateString([], {
3      year: "numeric",
4      month: "long",
5      day: "numeric",
6      hour: "2-digit",
7      minute: "2-digit",
8  });
9
10 useEffect(() => {
11     const fetchUser = async () => {
12         const { data } = await UserApi.getUser(post.
13             ↪ userId);
14         setpostUser(data);
15     };
16     fetchUser();
17 }, []);

```

Ezt követően meg kell vizsgálnom, hogy ha az adott *post* nem bejegyzés hanem komment, akkor azt nem jelenítem meg. Attól függően, hogy éppen melyik oldalon van használva a komponens, úgy jelenítem meg a bejegyzéseket is. Ha valakinek a profil oldalán van, akkor tényleg csak azokat jelenítse meg.

A bejegyzések és hozzászólások ha tartalmaznak képet, akkor azokat jó lenne oly módon megjeleníteni, hogy a méretétől függetlenül szépen jelenjen meg. A képeket lekerekítve jelenítem meg maximum 30rem és minimum 20rem magassággal. A szélesség nem haladhatja meg a *Post* osztály szélességét. Középre állítom a képet és átméretezem úgy, hogy a kép megtartja az eredeti képarányait, de kitölți a dobozt.

A bejegyzésnek a szövegére kattintva, a felhasználót átirányítom a bejegyzésnek az oldalára. A reakciókat akkor jelenítem meg színesen, ha a felhasználó már reagált az adott módon, vagy a kurzort a reakció fölé húzza.

Listing 3.13. *PostComponent.jsx* kódjának részlete

```

1 <div className="Post">
2     <Link
3         style={{ textDecoration: "none", color: "
4             ↪ inherit" }}
5         to={`/profile/${postUser.username}`}
6     >
7         <div className="detail">
8             <span>
9                 <b>{postUser.name}</b>
10                </span>
11                <span className="username"> @{postUser.
12                    ↪ username}</span>
13         </div>

```

```

12      </Link>
13
14      <Link
15          style={{ textDecoration: "none", color: "
16              ↪ inherit" }}
17          to={`/status/${post._id}`}
18          className="Post-Content"
19      >
20          <div className="desc">
21              <span>{post.desc}</span>
22          </div>
23          <img
24              src={
25                  post.image ? process.env.
26                      ↪ REACT_APP_PUBLIC_FOLDER + post.image
27                      ↪ : ""
28              }
29              alt=""
30              hidden={!post.image}
31          />
32      </Link>
33      <div className="date">
34          <span>{date}</span>
35      </div>
36      <div className="PostReaction">
37          <div
38              className="option like"
39              style={liked ? { color: "var(--red)" } : {}}
40          >
41              <UiHeart className="like-icon" onClick={
42                  ↪ handleLike} />
43              <span>{likes}</span>
44          </div>
45          <div className="option comment">
46              <UiCommentAltLines />
47              <span>{comments}</span>
48          </div>
49      </div>
50  </div>

```

A reakciók képeit az IconScout ingyenesen felhasználható unicons ikontárból importálom be és használom. Ezek színezhető ikonok ezért jól lehet őket ebben az esetben használni. A hozzájuk tartozó kedvelések és hozzászólások számát a *post* objektum tartalmazza.

### 3.5. Middleware

A Socket.IO szerver azért kell, hogy a felhasználók az egymással való üzenetváltás közben megkapják az üzeneteket valós időben az oldal frissítése nélkül. Igazából ez is egy külön szerver, a frontend és a backend között, viszont ez csak a frontenddel kommunikál.

Elkészítem a socket példányt, a megfelelő címmel és egy porttal, amin majd elérhető lesz a szerver. Tárolnom kell az éppen elérhető felhasználókat, ezeket egy tömbben fogom tárolni. Ezt követően elkezdek figyelni a socket által bekövetkezett eseményekre. Három eseményt hoztam létre. Az elsővel egy új felhasználót adok az elérhető felhasználók közé és visszaadja a jelenlegi elérhetőket.

Listing 3.14. A socket szerver kódja

```
1 const io = require("socket.io")(8800, {
2   cors: {
3     origin: "http://192.068.1.107:3000",
4   },
5 });
6
7 let onlineUsers = [];
8
9 io.on("connection", (socket) => {
10   socket.on("add-new-user", (newUserId) => {
11     if (!onlineUsers.some((onlineUser) => onlineUser
12       .userId === newUserId)) {
13       onlineUsers.push({
14         userId: newUserId,
15         socketId: socket.id,
16       });
17     }
18     io.emit("get-online-users", onlineUsers);
19     console.log(onlineUsers);
20   });
21
22   socket.on("send-message", (message) => {
23     const { receiverId } = message;
24     const receiverUser = onlineUsers.find((user) =>
25       user.userId === receiverId);
26     if (receiverUser) {
27       io.to(receiverUser.socketId).emit("receive-
28         message", message);
29     }
30   });
31
32   socket.on("disconnect", () => {
33     const index = onlineUsers.findIndex(
34       (user) => user.socketId === socket.id);
35     if (index !== -1) {
36       onlineUsers.splice(index, 1);
37     }
38   });
39 });
40
41 io.listen();
42
43 module.exports = io;
```

```

30     onlineUsers = onlineUsers.filter((user) => user.
31         ↪ socketId !== socket.id);
32     });

```

A másodikkal lehet üzenetet küldeni ha, mind a két felhasználó elérhető. A végén pedig ki kell vennem az elérhető felhasználók közül a kijelentkező felhasználót.

A frontend oldalon fognak ezek a hívások elindulni és visszakap majd a hívásnak megfelelő választ. A kliens oldalon a szükséges függőségek beimportálása után a `useRef()` hook segítségével két renderelés között is megmaradnak majd az értékei a `socket`-nek.

Listing 3.15. A socket kliens oldali kódja

```

1 import { io } from "socket.io-client";
2 import React, { useRef } from "react";
3
4 const socket = useRef();
5
6 useEffect(() => {
7     socket.current = io("http://192.168.1.107:8800");
8     socket.current.emit("add-new-user", user._id);
9     socket.current.on("get-online-users", (users) => {
10         setOnlineUsers(users);
11     });
12 }, [user]);

```

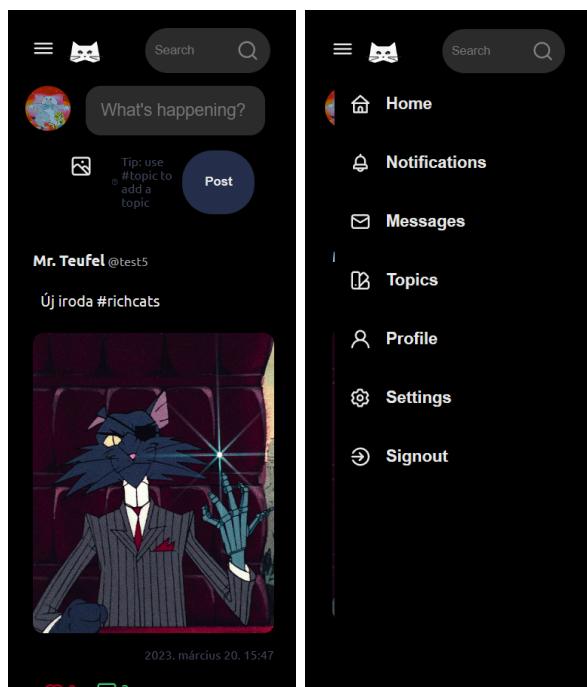
Beállítom, hogy melyik címre küldje a hívást. Ezt követően elküldi a szervernek a felhasználó azonosítóját, amit el fog tárolni a socket szerver, majd visszakapja az elérhető felhasználók listáját. A `disconnect` automatikusan meg fog hívódni, ha valami csatlakozási hiba van, szóval nekem ezt nem kell meghívnom külön.

## 4. fejezet

# Felhasználói dokumentáció

### 4.1. Reszponzív felhasználói felület

A felhasználói felület tervezésénél a szempont az egyszerűség volt. A weboldal sötét témát kapott, fekete háttér, fehér betűk, sötétkék gombok. Nem szeretem, ha egy oldalon a téma túl világos, mert zavarja a szememet, ezért választottam ezt a színvilágot. Az oldal neve Gato, ami magyarul macskát jelent, a logója is egy macska. A logó egy AI képgenerátorral készült.

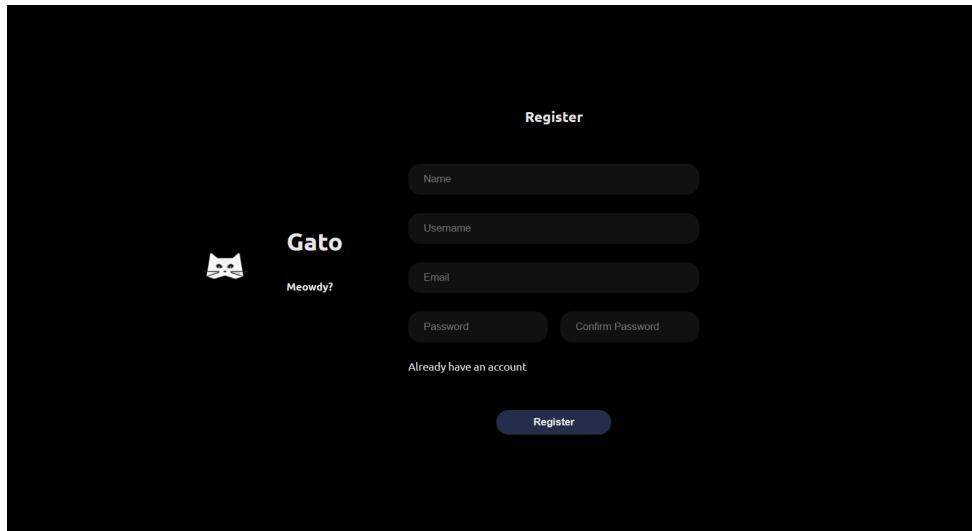


4.1. ábra. Hamburger menü a kis méretű kijelzőknél

Az oldalt lehet használni igazából tetszőleges méretű kijelzőn a reszponzív felhasználói felületnek köszönhetően. Az oldalsó navigációs menüt egy bizonyos képméret után már nem jelenítem meg, hanem úgynevezett hamburger menüt használok. Ha rákattintok az ikonjára, akkor jobb oldalról beúszik.

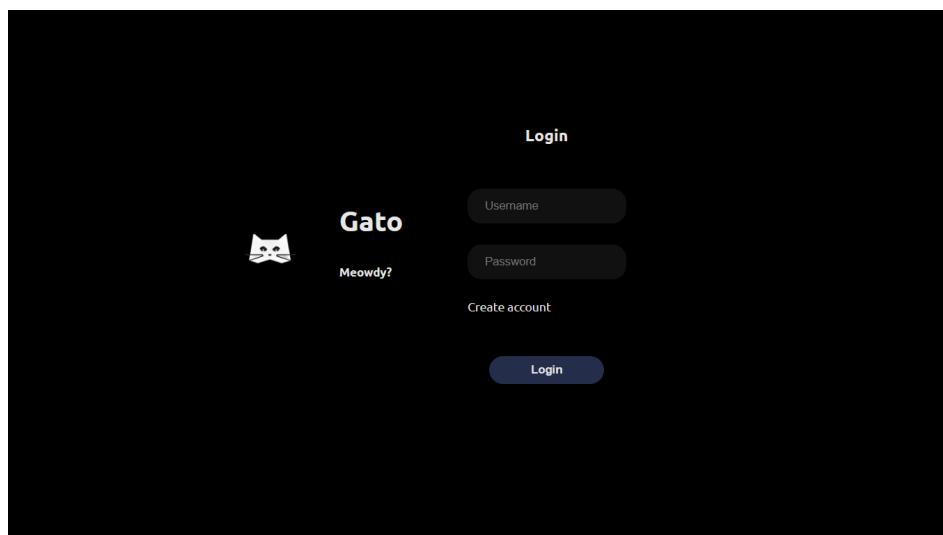
## 4.2. Bejelentkezés és Regisztrációs oldal

Az oldal megnyitásakor először az autorizációs oldalon fogjuk magunkat találni. Itt lehet kiválasztani, hogy ha már rendelkezünk felhasználói profillal, akkor be tudunk jelentkezni, ha pedig nem akkor létrehozhatunk egy újat.



4.2. ábra. Regisztrációs felület

Ha új felhasználói profilt szeretnénk készíteni, akkor egyedi felhasználónevet és emailcímét kell megadni. A profil neve lehet igazából bármi, a jelszót pedig kétszer kell megadni, hogy biztos ne legyen benne elírás. Ha nem egyezik akkor erről értesítést kapunk a felületen.

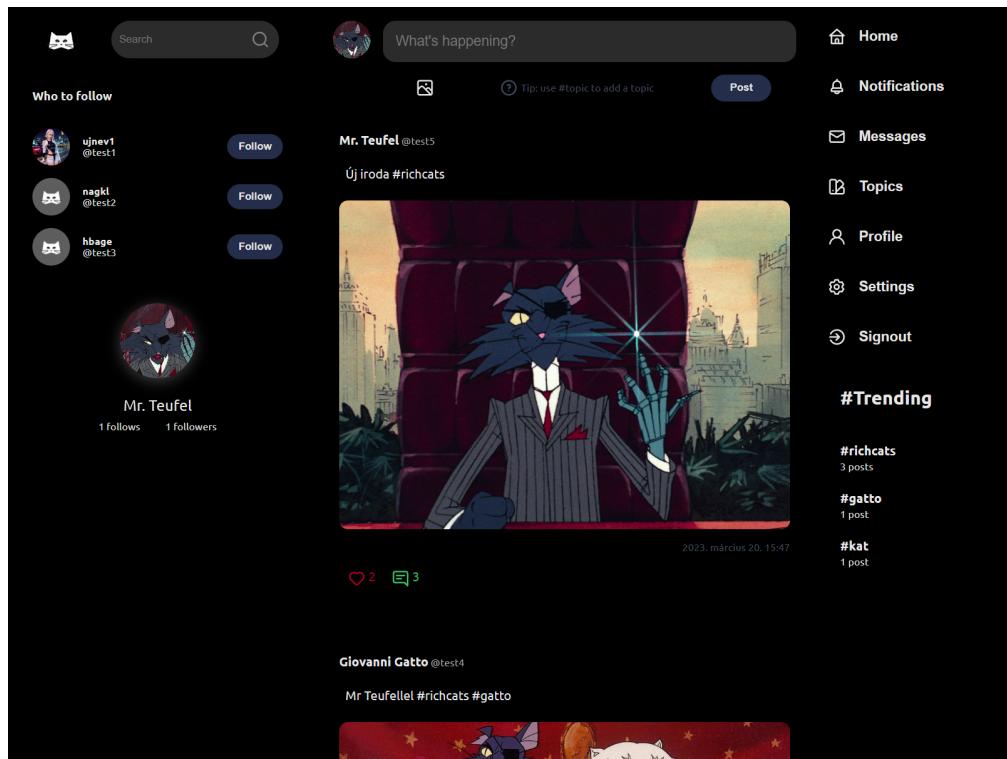


4.3. ábra. Bejelentkezés felület

Ha már regisztráltunk az oldalra, akkor a bejelentkezéshez csak két adatok kell megadni. Sikeres bejelentkezés, vagy regisztráció után a főoldalra navigál minket az oldal.

## 4.3. Főoldal

Sikeres autentikáció után a Főoldalon találjuk magunkat. Alapvetően három sávból áll az oldal. A bal oldalon találjuk a keresést, amivel felhasználókat kereshetünk, egy ajánlást, hogy kiket lehet érdemes követni és a saját profilunkról néhány adatot és a profilképünket. Középső sávban van lehetőségünk új posztot létrehozni. Poszt létrehozásakor, ha szeretnénk akkor képet is csatolhatunk a szöveg mellé. Témát is rendelhetünk egy poszthoz, ha a # karaktert használjuk. A középső sávban lesznek a posztok felsorolva. Ezek időrendileg csökkenő sorrendben fognak szerepelni. Csak azokat a posztokat fogjuk látni, amik más követett felhasználótól származnak.



4.4. ábra. Kezdőoldal

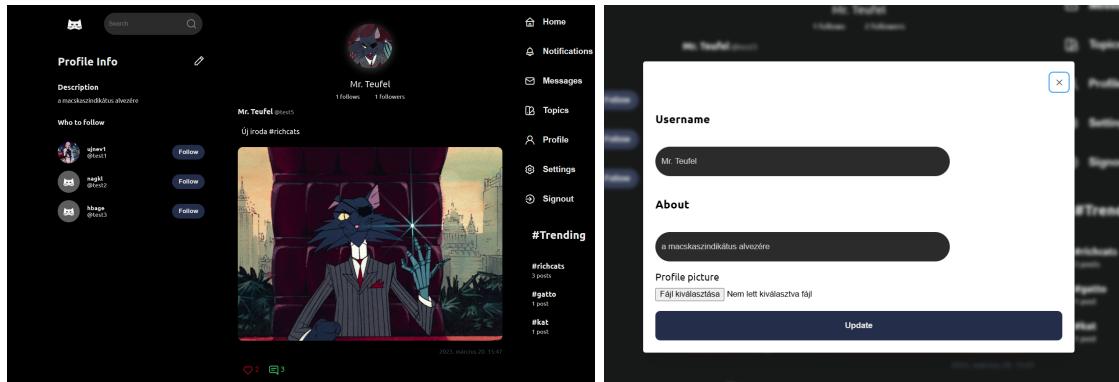
A bal oldali sávban helyezkedik el a navigációs menü és a trending komponens, ahol a legtöbbet használt téma láthatjuk. A navigációs menü látható szinte minden egyes oldalon, ennek segítségével tudunk a kívánt oldalra menni vagy kijelentkezni.

## 4.4. Profil oldal

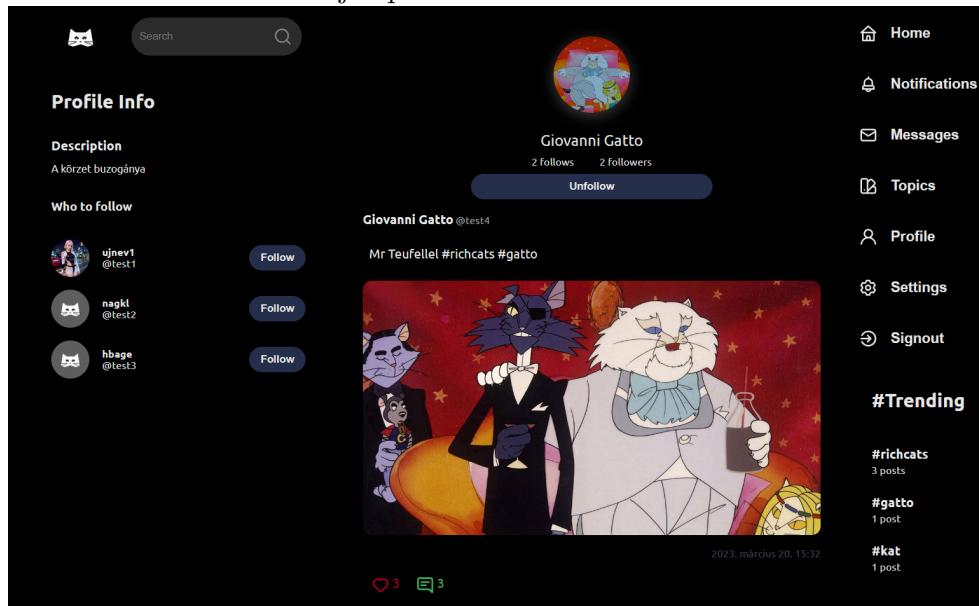
Két fajta profil oldal létezik, az egyik a saját profilunk oldala, a másik pedig az oldalra regisztrált másik felhasználóé.

Ha a saját profilunkat tekintjük meg, akkor van lehetőségünk megváltoztatni a hozzánk tartozó leírást, nevet és profilképet. Ezt a profil információ felirat melletti ceruza ikonra kattintva tehetjük meg. Ezt követően megjelenik egy ablak ahol láthatjuk

a jelenlegi nevünket és leírásunkat. Ezen az oldalon csak a saját posztjaink lesznek felsorolva.



4.5. ábra. Saját profil oldal és annak módosítása

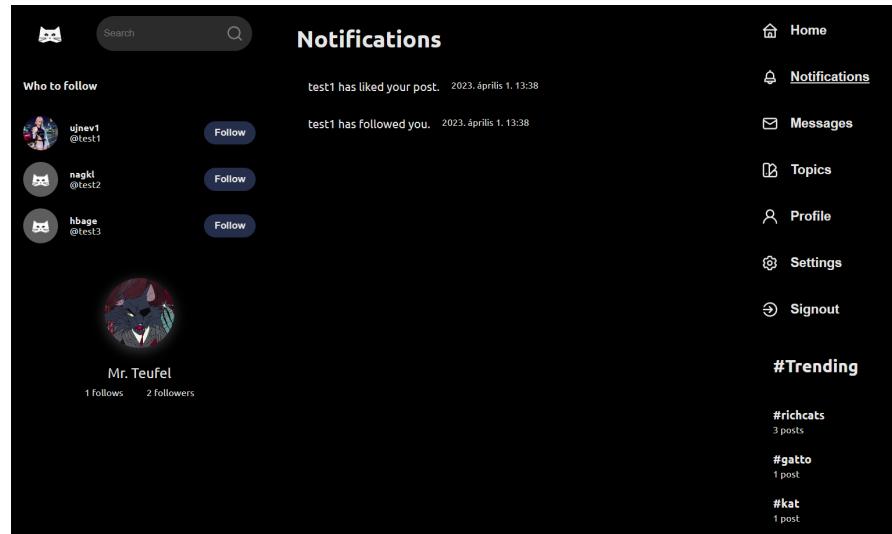


4.6. ábra. Egy másik felhasználó oldala

Ha egy másik felhasználó profil oldalát nyitjuk meg akkor nem fogjuk a módosítás ikont látni. Mindig az adott felhasználó posztjai lesznek felsorolva.

## 4.5. Értesítések

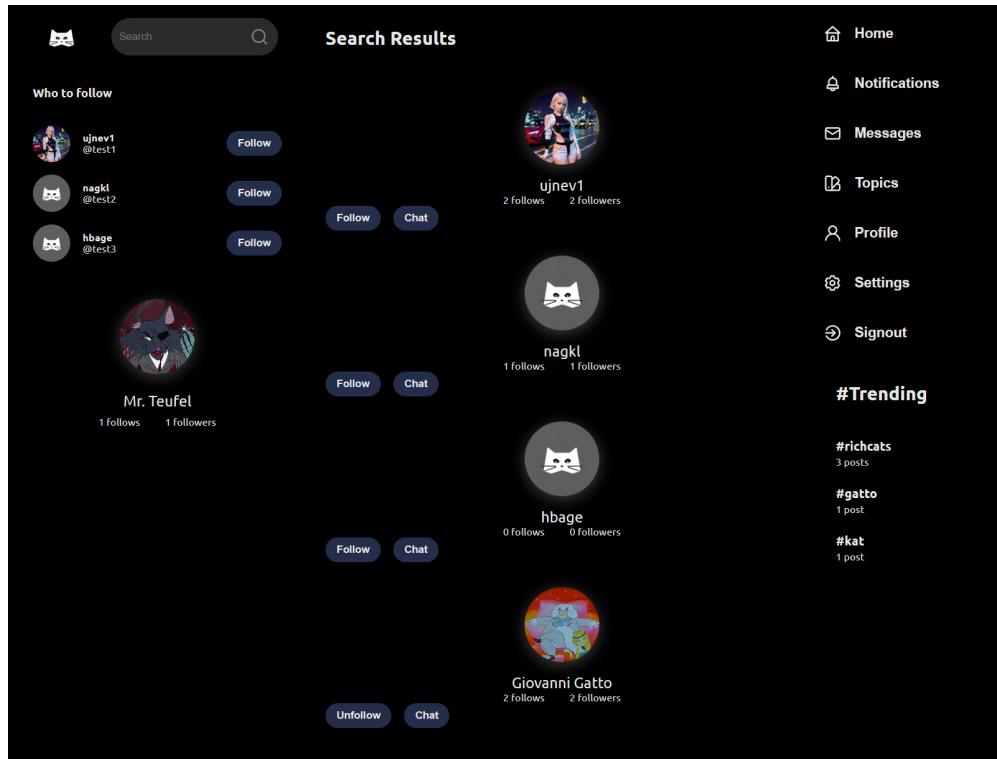
Ha kedvelik, vagy hozzászólnak a posztunkhoz, vagy egy új felhasználó követ be minket, akkor értesítést kapunk róluk. Ha van olvasatlan értesítésünk akkor a navigációs menüben alá lesz húzva az értesítések szöveg. A már megtekintett értesítések nem fognak megjelenni.



4.7. ábra. Értesítések oldal

## 4.6. Felhasználók keresése

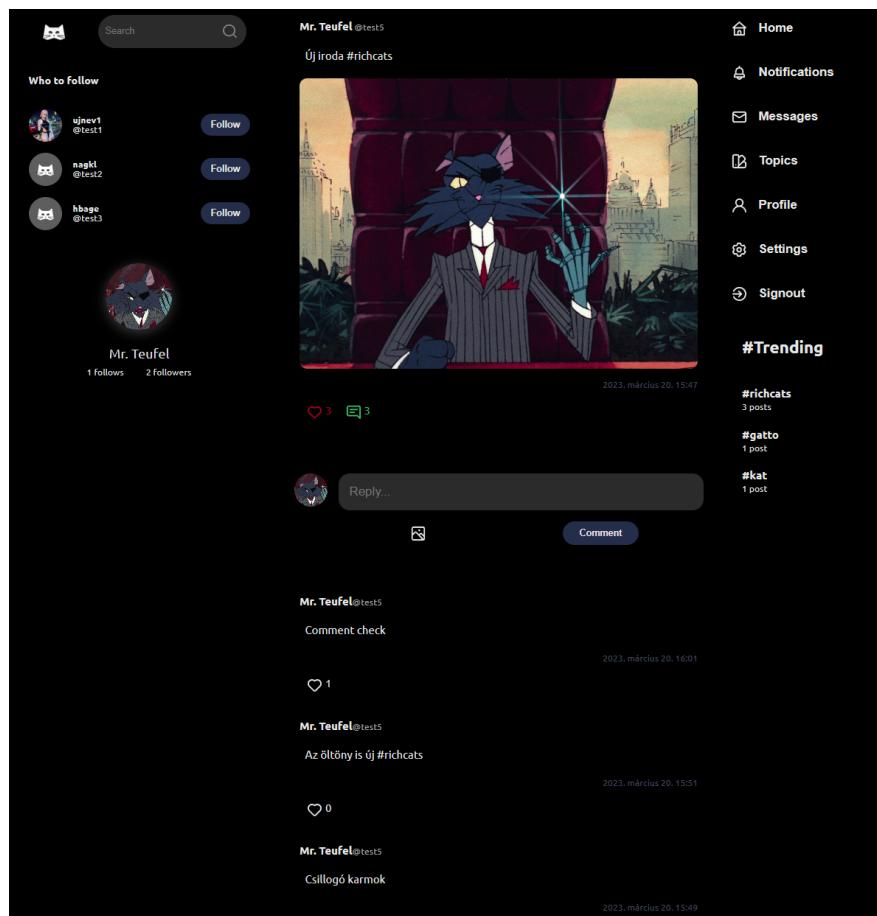
Ezzel a funkcióval tudjuk kilistázni azokat a felhasználókat amik eleget tesznek a keresési kritériumnak. Az eredményként kapott profilkot be tudjuk követ, vagy üzenetet tudunk nekik küldeni. A felhasználó nevére kattintva az adott felhasználónak a profiljára fogunk navigálni.



4.8. ábra. A keresés eredményei

## 4.7. Poszt oldal

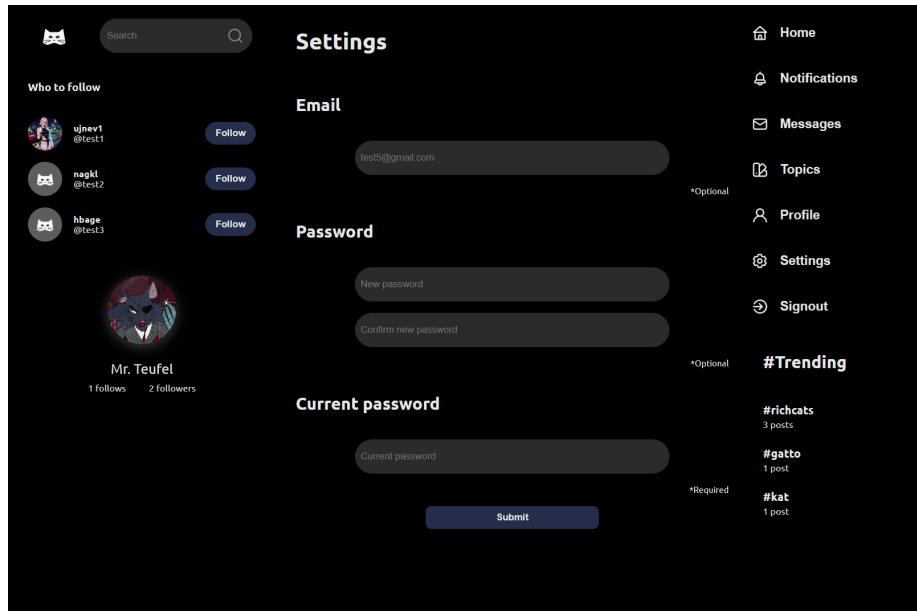
Az oldalt böngészve, ha egy poszt felkelti az érdeklődésünket és szeretnénk megtekinteni a hozzá tartozó hozzászólásokat, vagy sajátot akarunk hozzáfűzni, akkor a posztra kattintva a posztnak az oldala fog megjelenni. Itt van lehetőség akár képfeltöltésre hozzászólással együtt és reagálni a posztra, vagy a hozzá tartozó hozzászólásokra. Mindig a legújabb komment fog először megjelenni a posztok alatt. minden poszthoz tartozik kedvelés és hozzászólás ikon, a mellettük lévő számok mutatják, hogy éppen mennyien kedvelték vagy szóltak hozzá az adott poszthoz. Ezek az ikonok akkor fognak színesen megjelenni, ha kedveltük, vagy van már hozzászólásunk az adott poszthoz.



4.9. ábra. A poszt oldal

## 4.8. Beállítások

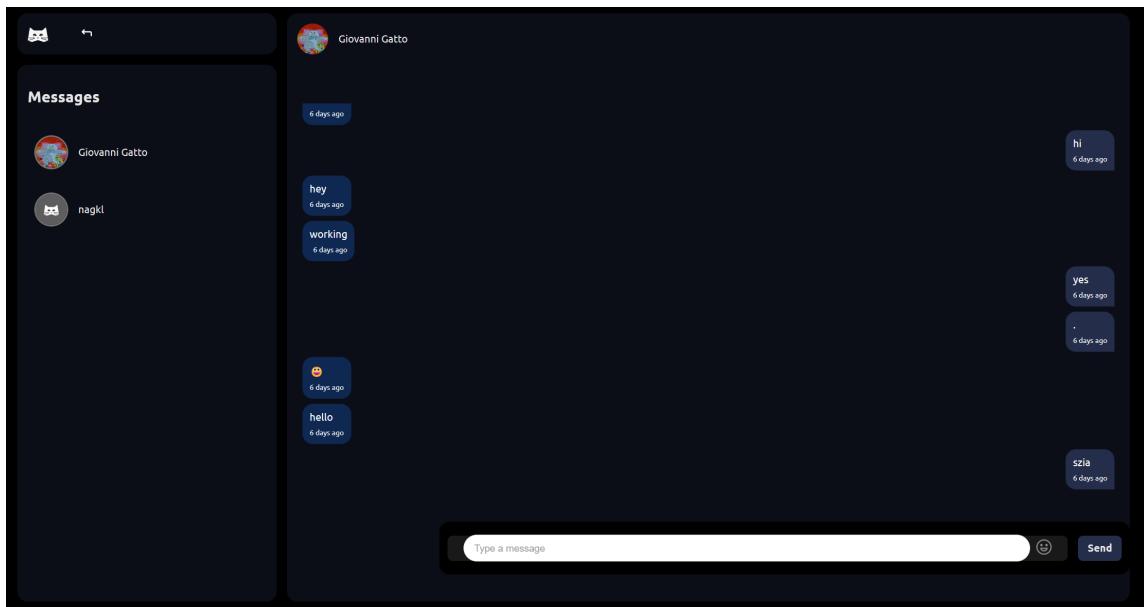
Regisztrációt követően van lehetőségünk megváltoztatni a felhasználói profilunk jelszavát és emailcímét. Szükséges megadni a jelenlegi jelszavunkat, különben sikertelen lesz a művelet.



4.10. ábra. Beállítások oldal

## 4.9. Üzenetek

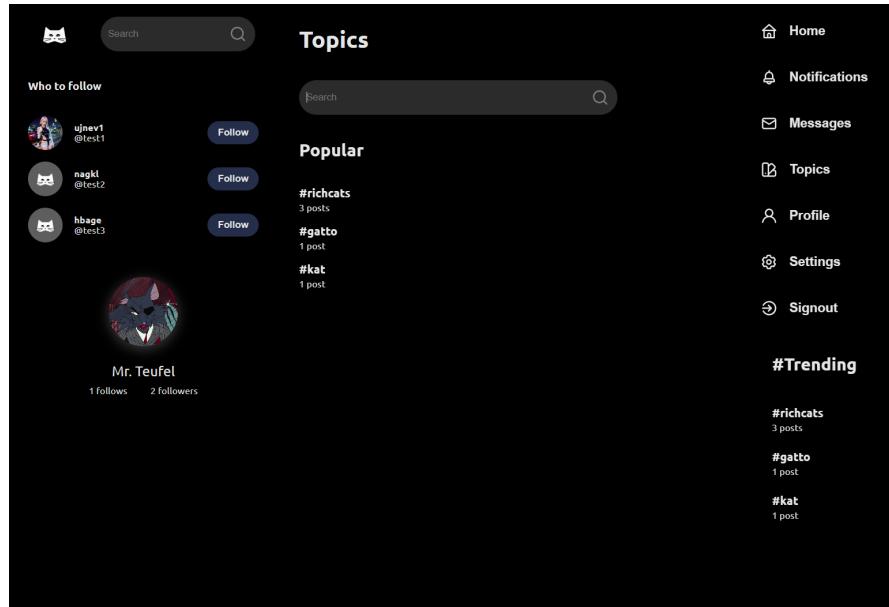
A felhasználók tudnak egymásnak privát üzeneteket küldeni. Az üzenetek tartalmazhatnak hangulatjeleket is. Zöld körrel van jelölve az a felhasználó, aki éppen online van az oldalon, szürkével, aki éppen nincs. Az üzenetek valós időben érkeznek meg, az oldalt nem kell frissíteni, hogy megkapjuk őket.



4.11. ábra. Üzenetek oldal

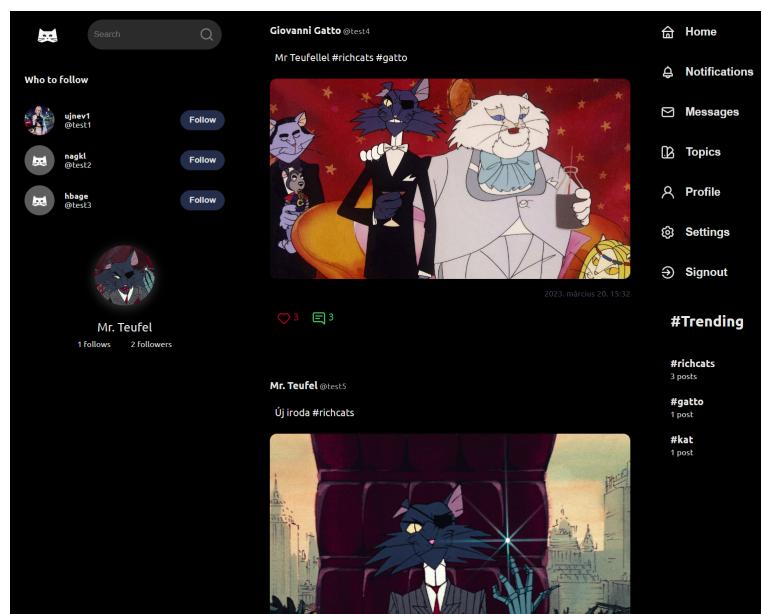
## 4.10. Témakörök

Témakörrel megjelölt posztokra is lehet szűrést végezni. Ezt a témakörök oldalon tudjuk megtenni. A keresés előtt felsorol nekünk néhány népszerű témakört, ezt követően a keresési eredményeket fogjuk látni.



4.12. ábra. Témakörök oldal

Ha rákattintunk egy témakörre, akkor azok a legfrissebb posztok fognak megjelenni, amik az adott témakörbe tartoznak. Egy bejegyzés több témakört is tartalmazhat.



4.13. ábra. A #richcats témakör oldala

# Összegzés

A céлом a projekt sikeres megvalósításán felül a saját tudásomnak, ismereteimnek a továbbfejlesztése volt. Ez volt az első projekt, ahol a MERN stack-et használtam. Rengeteg új dolgot tanultam, úgy gondolom ezeket sikeresen fel tudtam használni a fejlesztés során. Egy másik projektnél, már sikerült ezt az újonnan megszerzett tudást hasznosítanom. Fontos új technológiákat megismerni, mivel minden projekt más, így könnyebben ki tudja az ember választani, hogy milyen technológiát érdemes használni hozzá.

A weboldal előre eltervezett funkcióit sikeresen elkészítettem, sőt több minden került bele az elkészült projektbe, mint ami a tervekben szerepelt. Persze sok dologgal lehetne még bővíteni. Egy ötlet, hogy egyszerre több képet is hozzá lehetne adni egy bejegyzéshez vagy hozzászóláshoz és lehetőség képek küldésére privát üzeneteknél. Az oldal jelenleg csak fekete színben elérhető, lehetne készíteni hozzá világos témát is. Ami még eszembe jutott, hogy az értesítéseket jobban ki lehetne dolgozni.

Véleményem szerint sikerült egy jól működő közösségi média platformot létrehoznom a MERN stack segítségével. Az implementált funkciók használata minden felhasználó számára egyszerű, jól kezelhető. Az oldal felépítése logikus, könnyen áttekinthető és intuitív.

# Irodalomjegyzék

- [1] JavaScript (megtekintve: 2023.03.10.): <https://en.wikipedia.org/wiki/JavaScript>
- [2] MERN (megtekintve: 2023.02.20.): <https://www.mongodb.com/mern-stack>
- [3] MongoDB (megtekintve: 2023.02.20.): <https://en.wikipedia.org/wiki/MongoDB>
- [4] Node.js (megtekintve: 2023.02.20.): <https://en.wikipedia.org/wiki/Node.js>
- [5] Express.js (megtekintve: 2023.02.20.): <https://en.wikipedia.org/wiki/Express.js>
- [6] React (megtekintve: 2023.03.13.): <https://react.dev/reference/react>
- [7] Git (megtekintve: 2023.04.10.): <https://en.wikipedia.org/wiki/Git>
- [8] Github (megtekintve: 2023.04.10.): <https://en.wikipedia.org/wiki/GitHub>
- [9] npm (megtekintve: 2023.02.20.): [https://en.wikipedia.org/wiki/Npm\\_\(software\)](https://en.wikipedia.org/wiki/Npm_(software))
- [10] Yarn (megtekintve: 2023.02.20.): <https://yarnpkg.com/>
- [11] Jest (megtekintve: 2023.04.04.): <https://legacy.reactjs.org/docs/testing.html>
- [12] MongoDB gyűjtemények és adatbázisok (megtekintve: 2023.04.07.): <https://www.mongodb.com/docs/manual/core/databases-and-collections/>
- [13] Express Routes and Controllers (megtekintve: 2023.04.07.): [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\\_Nodejs/routes](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes)
- [14] React-DOM (megtekintve: 2023.04.07.): <https://legacy.reactjs.org/docs/react-dom.html>
- [15] React-Redux (megtekintve: 2023.04.08.): <https://react-redux.js.org/introduction/getting-started>

- [16] Axios (megtekintve: 2023.04.08.): [https://axios-http.com/docs/api\\_intro](https://axios-http.com/docs/api_intro)
- [17] React Component (megtekintve: 2023.04.10.): <https://react.dev/reference/react/Component>
- [18] Socket.IO (megtekintve: 2023.04.10.): <https://socket.io/>
- [19] LocaleDateString (megtekintve: 2023.04.10.): [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Date/toLocaleDateString](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/toLocaleDateString)
- [20] JavaScript Map Method (megtekintve: 2023.04.10.): [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)
- [21] JSON Web Token (megtekintve: 2023.04.11.): [https://en.wikipedia.org/wiki/JSON\\_Web\\_Token](https://en.wikipedia.org/wiki/JSON_Web_Token)

# Nyilatkozat

Alulírott Vereb Barna, büntetőjogi felelősségem tudatában kijelentem, hogy az általam benyújtott, Közösségi hálózati szolgáltatás fejlesztése MERN architektúrával című szakdolgozat önálló szellemi termékem. Amennyiben mások munkáját felhasználtam, azokra megfelelően hivatkozom, beleértve a nyomtatott és az internetes forrásokat is.

Aláírásommal igazolom, hogy az elektronikusan feltöltött és a papíralapú szakdolgozatom formai és tartalmi szempontból mindenben megegyezik.

Eger, 2023. április 11.

*Vereb Barna*  
aláírás