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

Sereg András

TESTRE SZABHATÓ KOKTÉLADATBÁZIS KÉSZÍTÉSE

React.js és Node.js segítségével

KONZULENS

Kövesdán Gábor

BUDAPEST, 2024

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2024. 05. 23.

...…….
Sereg András

Összefoglaló

A dolgozatban bemutatom az alkalmazáshoz használt különböző technológiákat és a tervezési folyamatokat. Itt részletesebben kifejtem az összes adatbázisban megtalálható típust és a felhasználói szerepköröket, valamint a főbb funkciókat.

Nem utolsó sorban az alkalmazás implementálásáról fogok írni, részletekbe menően szó esik a React segítségével készült frontendről, és a Node.js-ben írt backendről is.

Abstract

In this thesis, I will present the various technologies used for the application and the design processes. Here, I will elaborate in detail on all the types found in the database and the user roles, as well as the main functionalities.

1 .Bevezetés

Ahány ember, annyi féle preferencia. Nincs ez másképp, ha a szórakozási vagy italfogyasztási szokásokról beszélünk. Van, hogy egyedül is nehezen tudunk dönteni arról, hogy hol és hogyan töltsük el a péntek-szombat estéinket, egy több fős társasággal pedig még nehezebb dűlőre jutni.

A dolgozatom célja, hogy egy olyan alkalmazás készítsek, mellyel a felhasználók minél több szempont szerint alakíthassák a preferenciáikat, és azok alapján megtalálják Budapest számukra legmegfelelőbb vendéglátói illetve szórakoztatói egyégét.

A dolgozatom során a 2. fejezetben ismertetem az alkalmazás elkészítése során használt technológiákat, a 3. fejezetben a már létező, az alkalmazáshoz hasonló szolgáltatásokat hasonlítom össze. A 4. fejezetben a tervezési lépéseket ismertetem, az 5. fejezetben pedig az implementáció részleteit mutatom be, majd végül az alkalmazás használatba helyezéséről fogok írni a 6. fejezetben.

2 Felhasznált technológiák

2.1 Backendhez használt technológiák

2.1.1 Node.js

A Node.js [1] egy nyílt forráskódú, szerveroldali javascript futtatókörnyezet, ami az eseményvezérelt modelljével lehetővé teszi a hatékony és skálázható alkalmazások fejlesztését.

Egyik nagy előnye - mivel a webalkalmazások túlnyomó részének a kliens oldala is javascript nyelven íródott – hogy egységesíti a kódbázist az adott projektben, így segíti a fejlesztők közötti kommunikációt és a kód megértését mindenki számára.

Az npm (Node Package Manager) segítségével könnyen felhasználhatunk már létező könyvtárakat az alkalmazásunk fejlesztése során, ezzel megannyi funkcióhoz és modulhoz jutva, melyekkel átláthatóbb és tisztább kódot írhatunk.

2.1.2 MongoDB

A MongoDB[2] egy nyílt forráskódú, dinamikus adatmodellt alkalmazó NoSQL adatbázis. JSON-szerű dokumentumokban tárolja az adatokat, ezzel skálázhatóságot és magas teljesítményt nyújtva.

Sémamentes tervezése révén könnyű és dinamikus adatmodellezést tesz lehetővé, ami ideális félig strukturált adatok kezelésére.

2.2 Frontendhez használt technológiák

2.2.1 React

A React.js[3] egy nyílt forráskódú Javascript könyvtár, amellyel felhasználói felületeket készíthetünk, elsősorban Single Page Application (későbbiekben SPA) formájában.

Egyik főbb jellemzője a virtuális DOM, ami lehetővé teszi a hatékony és gyors tartalomfrissítéseket az alkalmazásunkban, anélkül, hogy az oldalt újra kelljen töltenünk. Ezt úgy éri el, hogy a komponensek változásakor a DOM-nak egy absztrahált változatát módosítja, ezzel csak az érintett állapotok változnak meg

A Reactban az alkalmazások úgynevezett komponensekből épülnek fel, melyek elősegítik az újrahasználhatóságot, és könnyebben karbantarthatóvá és tesztelhetővé teszi a kódot.

Lehetőségünk van különböző Hookok használatára, aminek segítségével függvényként készíthetjük el a komponenseinket, valamint lehetőséget biztosít a fejlesztőknek hogy különválasszák a megjelenítést a működési logikát.

2.2.2 PrimeReact

A PrimeReact[4] egy nyílt forráskódú, fejlett, és könnyen testreszabható UI komponenskönyvtár, amely a React alkalmazások számára kínál modern és stílusos felhasználói felületeket. A könyvtár széles skálájú kész komponenseket tartalmaz, beleértve a gombokat, űrlapokat, táblázatokat, paneleket és sok más elemet, amelyekkel gyorsan és hatékonyan építhetünk felhasználói felületeket. A PrimeReact átfogó dokumentációt kínál a komponensekről, példákkal és használati utasításokkal, amelyek segítenek a fejlesztőknek megérteni és használni ezeket a komponenseket az alkalmazásaikban.

2.2.3 React Redux

A Redux egy előre kiszámítható állapotkezelő könyvtár, amely egyetlen állapotfát használ az alkalmazás állapotának tárolására és módosítására. A React Redux[5] pedig egy olyan könyvtár, amely lehetővé teszi a Redux állapotkezelés könnyű integrálását a React alkalmazásokba. A React Redux segítségével a komponensek egyszerűen elérhetik és módosíthatják az alkalmazás állapotát, anélkül hogy közvetlenül hozzáférnének az állapothoz vagy a Redux műveletekhez. Ezáltal a kódbázis tisztább és jobban karbantartható lesz. A React Redux további előnyei közé tartozik a könnyű tesztelhetőség és az állapotmenedzsment különböző aspektusainak egyszerű kezelése, például az aszinkron műveletek vagy a globális állapot megosztása komponensek között. Az alkalmazásban betöltött pontos szerepét később részletesebben kifejtem.

2.2.4 React Router

A React Router egy népszerű routing könyvtár, amely lehetővé teszi a React alkalmazások számára a kliensoldali útvonalkezelést. A React Router segítségével könnyedén hozhatunk létre dinamikus és többoldalas alkalmazásokat, anélkül hogy újratöltenénk az oldalt. A könyvtár egyszerű és intuitív API-t kínál, amely jól integrálható

a React komponensekkel, lehetővé téve az alkalmazás struktúrájának és navigációjának logikus és könnyen követhető kialakítását. A könyvtár további előnyei közé tartoznak az átirányítások és a késleltetett navigáció támogatása, ami hozzájárul a felhasználói élmény javításához.

2.2.5 axios

Webalkalmazások létrehozásakor gyakori feladat a HTTP protokollon keresztül való kommunikáció, amit az egyik legegyszerűbben az *Axios*[6] könyvtárral tudunk implementálni. Használatával könnyedén küldhetünk mindenféle http kérést, mint a get, post vagy delete, valamint kezelhetjük az ezekre adott válaszokat. Az *Axios* számos előnyt kínál a Javascriptben alapból elérhető *fetch()* metódushoz képest, például automatikusan kezeli a JSON adatok átalakítását, és képes kezelni az időtúllépéseket és hibakezelést. A könyvtár támogatja az aszinkron működést, így *Promise*-okkal és *async/await* szintaxissal is használható, ami megkönnyíti az aszinkron műveletek kezelését és a kód olvashatóságát. Az Axios emellett lehetővé teszi az egyedi konfigurációkat, például alapértelmezett fejlécértékek, alap URL-ek és más beállítások megadását, amelyekkel testreszabhatjuk az alkalmazás HTTP kéréseinek viselkedését

2.2.6 Typescript

A javascript webfejlesztésben való elterjedésével rohamosan derültek ki a nyelv anomáliái és hiányosságai. Ezt hivatott orvosolni a Typescript[7], ami a Javascript egy szigorúbb kiterjesztése. Segítségével fordítási időben is kezelhetünk hibákat, az erős típusellenőrzésnek köszönhetően pedig a legtöbb hiba már a fejlesztőkörnyezetünkben jelentkezik. Létrehozhatunk benne egyedi típusokat és interfaceket is

2.3 Szolgáltatások

2.3.1 Verziókezelés

Fejlesztés során fontos nyomon követni a fájlokban végrehajtott változtatásokat. A verziókezelés lehetővé teszi, hogy visszatérjünk korábbi változatokhoz és új verziókat hozzunk létre. Erre a legelterjedtebb eszköz az, ingyenes és nyílt forráskódú git[8]. Kisebb és nagyobb projektekben is használható, egyedül inkább verziókövetésre alkalmas, de hasznos tud lenni ha különböző eszközökön fejlesztünk. Több felhasználó számára

lehetőséget biztosít arra, hogy ugyanazon a kódon dolgozzanak, és könnyedén kommunikáljanak egymás között.

A git parancsait kiadhatjuk manuálisan az operációs rendszerünk parancssorából, vagy a git saját Command Line Interface-ében (CLI), de léteznek különböző asztali alkalmazások, mint például a Github Desktop, ami gyorsabbá és felhasználóbarátabbá teszi a könyvtárak verziókezelését. Emellett a legtöbb modern fejlesztőkörnyezetben elérhetők különböző bővítmények, melyek segítségével mellőzhető a CLI-k használata

A Github egy olyan szolgáltatás, ami git könyvtárak(repository-k) tárolására szolgál. Könnyen átlátható felületet biztosít a változtatásokhoz és lehetővé teszi különböző műveletek végrehajtását, mint a kódértékelés, vagy úgynevezett issue-k létrehozása, melyek segítségével hatékonyan nyilvántarthatóvá válnak az elvégzendő részfeladatok.

Mivel egyedül fejlesztettem az alkalmazást, így a git használatakor gyakran előforduló merge conflictokat nem tapasztaltam, sokkal inkább verziókövetésre használtam, illetve nagyban elősegítette a több eszközről való munkát.

2.3.2 Google Maps API

A Google Cloud[9] egy teljes körű felhőalapú szolgáltatásokat nyújtó platform, melynek segítségével könnyen implementálható megoldásokat kapunk olyan problémákra, mint az adattárolás, adatelemzés, vagy gépi tanulás.

Ezek között található a Google Maps API, aminek segítségével interaktív térképeket jeleníthetünk meg az alkalmazásunkban. Megadhatjuk a típusát, hogy műholdas vagy utcai képet mutasson, különböző rétegeket jeleníthetünk meg rajta, mint a forgalmas területek, tömegközlekedési útvonalak, vagy időjárás. Mindemellett különböző, szintén egyedien átalakítható jelölőket, és ezekhez tartozó információs ablakokat jeleníthetünk meg, valamint kezelni tudjuk vele a felhasználói interkaciókat, mint a húzások, nagyítások vagy kattintások, ezzel javítva a térkép dinamikusságát.

3 Hasonló alkalmazások

3.1 Untappd

1. ábraAz Untappd logója

Az Untappd egy mobilon és weben egyaránt elérhető alkalmazás, ami kifejezetten a kézműves sörök piacára van kiélezve. Az alkalmazásban megtalálhatóak világszerte elérhető kocsmák és bárok, a profiljukban szerepel minden fontos információ a helyről.

A menüt a kocsmák maguk tudják változtatni, így naprakészen tartva azt hogy éppen mi elérhető náluk. A sörök valamennyi paramétere, mint a típus, származási hely, alkoholszázalék, vagy a kiszerelés mérete és ára megtalálható a menükben.

Emellett az alkalmazásban lehetőség nyílik különböző interakciókra a kocsmákkal és más felhasználókkal: be tudunk jelentkezni egy adott helyről, hogy hol, kivel, és mit fogyasztunk, és a fogyasztott terméket tudjuk értékelni, mind egy 1-5-ig terjedő skálán, mind pedig különböző tagek hozzáadásával.

3.2 Yelp

A Yelp egy széles körben használt, többfunkciós platform, amely segítségével a felhasználók különböző helyi vállalkozásokat, beleértve az éttermeket, bárokat, kávézókat és szórakozóhelyeket kereshetnek és értékelhetnek.

2. ábra A Yelp logója

A felhasználók értékelés mellett véleményeket is írhatnak, vagy feltölthetnek különböző fotókat, ezzel is segítve másokat a döntéshozatalban. Elérhető weben és mobilalkalmazásként is.

Személy szerint ezt az alkalmazást nem használtam, de sokoldalúsága miatt, hogy nem csak bárokra van kialakítva, hasznos és effektív lehet a felhasználók számára

3.3 Cocktail Flow

3. ábra A Cocktail Flow logója

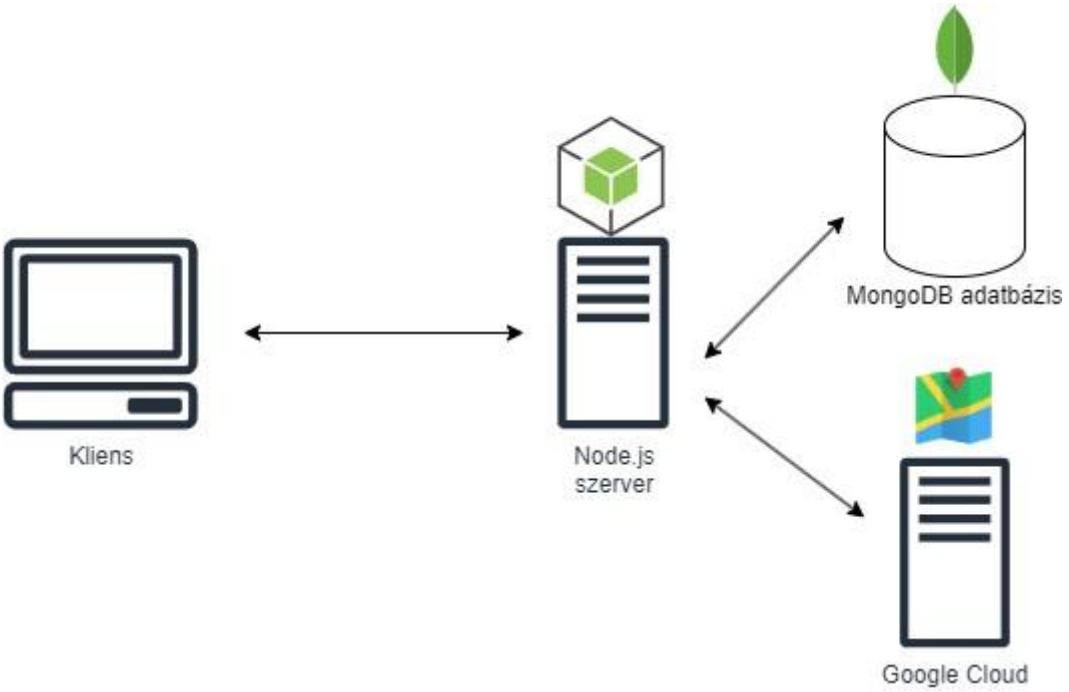
A Cocktail Flow egy népszerű koktélrecept alkalmazás, amely segít a felhasználóknak különféle koktélokat készíteni otthon, vagy akár új italokat felfedezni. Az alkalmazás intuitív felhasználói felülettel rendelkezik, és rengeteg receptet kínál, amelyek közül sok híres mixológusok által van összeállítva.

A felhasználók kereshetnek az összes koktél képekkel ellátott listájából, vagy különböző alapanyagok szerint szűrve. A legtöbb ital el van látva különböző ismertetőjegyekkel, ami megkönnyítheti a felhasználók döntését. Nagy előnye a szép, letisztult és felhasználó-barát design. Az előzőhöz hasonlóan ezt az oldalt sem ismertem, de utólag megvizsgálva néhány felépítést érintő kérdésben hasonlóan jártam el a dolgozatomban.

4 Tervezés

4.1 Architektúra

Az alkalmazás architektúrája két fő részből áll: frontend alkalmazás és backend szerver. A kettő közötti kommunikáció HTTP protokollon, REST API segítségével történik. Emellett a szerver kommunikál egy MongoDB adatbázissal, ahol az alkalmazáshoz szükséges adatok találhatók, valamint a Google Clouddal, ahonnan a térkép megjelenítéséhez használt API-t veszi igénybe.



4. ábra Az alkalmazás architektúrája

4.2 Felhasználói szerepkörök

Az alkalmazás jelenleg két felhasználótípust különböztet meg: admint és felhasználót.

- Felhasználó

  A felhasználó számára megjelenik a 3 különböző lista, ezek elemeit tudja a kedvencei közé helyezni. A kocsmák esetében lehetősége van pontozni a helyet, a sok helyen elterjedt 1-5 csillagos skálán.

Ezen kívül a felhasználók tudnak baráti kérelmet küldeni egymásnak, elfogadni vagy elutasítani ezeket a kérelmeket, és törölni a barátlistájukról embereket.

- Admin

  Az admin felhasználók számára ezen felül lehetőségük van hozzáadni új elemeket bármelyik listához, megváltoztatni bármely elemet, valamint törölni elemeket a listákból.

4.3 Felhasználói esetek

5. ábraAz alkalmazás felhasználói esetei

4.4 Adatmodellek

A feladat megoldása során négy különböző adatmodellt hoztam létre, hármat a 3 különböző eltárolt objektumról és egyet a felhasználók számára.

4.4.1 Ingredient

A hozzávalók információit tárolja, amelyek:

- a hozzávaló neve
- a hozzávaló alkoholszázaléka
- a hozzávaló típusa

Ezeken kívül lehetett volna részletesebben típusokba sorolni az összetevőket, illetve más tulajdonságokat tárolni hozzájuk, mint például ízprofil vagy származási hely, de az alkalmazásba a fent felsoroltak alapján implementáltam.

4.4.2 Drink

Az italok információit tárolja:

- az ital neve
- az ital típusa
- az italhoz ajánlott pohár
- az italhoz tartozó kép
- az italt alkotó hozzávalók listája

Ezekkel a tulajdonságokkal szinte minden szükséges részét le lehet fedni egy bizonyos italnak. Ezt a modellt szét lehetett volna bontani két különböző modellre, az egyszerű italokra és a koktélokra, de az összevont megoldás mellett döntöttem a tervezés során.

4.4.3 Pub

A kocsmák információit tárolja:

- a kocsma neve
- a kocsma címe
- a kocsma szélességi és hosszúsági koordinátája
- a kocsmára érkezett értékelések tömbje
- a kocsma itallapja
- a kocsma nyitvatartási ideje

A kocsmák tárolásához létre kellett hoznom specifikus típusokat, hogy a nyitvatartási idő, és a menüben található elemek eltárolhatóak legyenek

4.4.4 User

A felhasználókról szóló információkat tárolja. Az alkalmazásban két felhasználó típust különböztetünk meg: admin és (mezei) felhasználó. Nem láttam értelmét ezeknek külön adatmodellt létrehozni, redundancia-elkerülés miatt. A következő adatokat tárolom el a felhasználókról:

- a felhasználó adatai, mint az email cím és felhasználónév
- a felhasználó titkosított jelszava
- a felhasználó típusát
- a felhasználó által kedvelt hozzávalók
- a felhasználó által kedvelt italok
- a felhasználó által kedvelt kocsmák
- a barátok és baráti kérelmek listája

4.5 REST API végpontok

4.5.1 REST

A REST (Representational State Transfer)[10] architektúra szabályok és elvárások összessége. Ezeknek az elvárásoknak való megfelelés segíti az HTTP API-k készítését,

ezáltal a rendszereket kezelhetőbbé, gyorsabbá és könnyebben skálázhatóvá teszi. Az architektúra alapjait képezi a kliens-szerver modell, ahol a kommunikáció az HTTP protokollon keresztül zajlik. Emellett fontos, hogy a komponensek közötti kommunikáció egy egységes interfészen keresztül történjen, és a szerver válaszai világosak és érthetőek legyenek a kliens számára

4.6 Végpontok

Az alkalmazás az alábbi végpontokon keresztül kommunikál a szerverrel. Ezeken keresztül történik az erőforrások adatbázishoz adása, olvasása és módosítása.

Metódus	URL	Leírás
GET	/routes/ingredients/	A hozzávalók lekérdezése
POST	/routes/ingredients/add	Egy hozzávaló hozzáadása
POST	/routes/ingredients/update/:id	Egy hozzávaló módosítása
DELETE	/routes/ingredients/:id	Egy hozzávaló törlése
GET	/routes/drinks/	Az italok lekérdezése
POST	/routes/drinks/add	Egy ital hozzáadása
POST	/routes/drinks/update/:id	Egy ital módosítása
DELETE	/routes/ drinks /:id	Egy ital törlése
GET	/routes/pubs/	A kocsmák lekérdezése
POST	/routes/pubs/add	Egy kocsma hozzáadása
POST	/routes/pubs/update/:id	Egy kocsma adatainak változtatása
POST	/routes/pubs/update-rating/:id	Egy kocsma értékeléseinek változtatása
DELETE	/routes/pubs/:id	Egy kocsma törlése
GET	/routes/users/	A felhasználók lekérdezése
GET	/routes/users/:id	Egy felhasználó lekérdezése
POST	/routes/users/add	Egy felhasználó hozzáadása

POST	/routes/users/update/:id	Egy felhasználó adatainak frissítése
POST	/routes/users/login	Egy felhasználó beléptetése
POST	/routes/users/register	Egy felhasználó regisztrációja
GET	/routes/users/friends/:id	Az adott felhasználó barátainak az id-jeinek lekérdezése
GET	/routes/users/requests/:id	Az adott felhasználó kérelmeinek az id-jeinek lekérdezése
GET	/routes/users/friendsNames/:id	Az adott felhasználó barátainak a neveinek lekérdezése
GET	/routes/users/requestsNames/:id	Az adott felhasználó kérelmeinek a neveinek lekérdezése
POST	/routes/users/update-friends/: id	Az adott felhasználó barátainak frissítése
POST	/routes/users/update-requests/:id	Az adott felhasználó kérelmeinek a frissítése
POST	/routes/users/sendRequest	Baráti kérelem elküldése
POST	/routes/users/acceptRequest	Baráti kérelem elfogadása
POST	/routes/users/declineRequest	Baráti kérelem elutasítása
POST	/routes/users/deleteFriend	Barát törlése

4.7 Fontosabb funkciók

4.7.1 Kedvencekhez adás

A felhasználóknak lehetőségük van mind a három listában szereplő elemek közül a kedvenceikhez adni elemeket. Ezt a függvényt egy gomb valósítja meg ami az adott típusú elem paneljében található. A gombnyomás hatására az alkalmazás megvizsgálja, hogy a felhasználónak szerepel-e az elem a listájában. Ha szerepel, akkor kiveszi onnan, ha nem szerepel akkor pedig hozzáadja, így oldja meg a függvény, hogy egyszerre történjen a kedvencekhez adás és a kedvencekből való kivétel kezelése. Ezt követően az axios könyvtár segítségével egy POST requesttel a megfelelő URL-re elküldi a megváltoztatott, kedvenceket tároló tömböt.

4.7.2 Listák szűrése

A listák szűrését úgy oldottam meg, hogy létrehoztam egy *applyFilter()* metódust, ami egy szűrt listát ad vissza, melyet az alkalmazásban a *.map()* függvény segítségével listázok majd ki. A függvényben fontos szerepet játszik a state-ben tárolt *formResult* változó, ami a szűrés gomb megnyomására állítódik be. Ennek a típusa megegyezik az listában tárolt elem típusával, és kezdetben az alkalmazásban definiált alapértelmezett érétket veszi fel.

A szűrés során először ellenőrzöm, hogy a *formResult* megegyezik-e az alapértelmezett objektummal. Amennyiben igen, akkor az egész listát fogja visszaadni a függvény. Ha nem egyeznek, akkor a Javascriptben tömbökön alkalmazható *.filter()* metódus segítségével kiválasztom a szűrési feltételeknek megfelelő elemeket, és az így létrejött tömböt adom vissza. Amennyiben lehet sorrendet is állítani, azt ezután alkalmazom a visszatérési értékre, és csak azután adom vissza a keresett tömböt. A szűrés törlésekor egyszerűen visszaállítom a *formResult* változót az alapértelmezett értékre, és ekkor az első logikai elágazásnál visszatér a függvény az egész listával.

5 Implementáció

5.1 -Backend:

Az alkalmazás backendjét egy Node.js szerver szolgálja. Ide érkeznek be a REST API hívások a webalkalmazásból, amik feldolgozásra és végrehajtásra kerülnek. A szerverhez kapcsolódik egy MongoDB adatbázis, amelyben a definiált sémák szerint tárolom el a beérkező adatokat. A modellekhez kapcsolódik egy Javascript fájl, amelyben a hozzájuk tartozó kérések kerülnek feldolgozásra. Mindezt egy szerver fájl fogja össze, amiben konfiguráltam a futáshoz szükséges adatokat és megadtam a megfelelő útvonalakat ahova a hívások érkezhetnek.

5.1.1 MongoDB adatmodellek

Az előző fejezetben kifejtett különböző típusú elemek tárolásához először létrehoztam a hozzájuk tartozó sémákat. Ezek specifikus JSON objektumok, amik meghatározzák az eltárolandó entitás adattagjainak nevét és típusát, illetve egy tömbben meg lehet adni, hogy ezek közül melyikeket kötelező specifikálni, valamint default értéket is megadhatunk.

Minden tárolandó adatnak létrehoztam egy külön **_model.js* fájlt, amiből a létrehozott Schema exportálásra kerül. Itt specifikálom az adattagok nevét és típusát. A tartalmazásokat, mint például mikor az italok összetevőit tárolom el, egy string tömbbel oldottam meg, amiben az eltárolt dolgok nevei találhatóak. Itt általánosan egy jobb, de főleg időtállóbb megoldás lehetett volna az elemek azonosítóinak az eltárolása, de úgy gondoltam, hogy az adatok komplexitása nem feltétlen követeli ezt meg, és logikailag rendben lehet az, hogy egyesetleg már kitörölt alapanyagot tartalmazzon egy ital, így maradtam a nevek tárolásánál.

```
const mongoose = require("mongoose")
const Schema = mongoose.Schema

const drinkSchema = new Schema({
  name: { type: String },
  type: { type: String },
  ingredients: {type: Array},
  glass: {type: String},
  img: {type: String}
})
const Drinks = mongoose.model("Drinks", drinkSchema)
module.exports = Drinks
```

A felhasználók esetében viszont az entitások azonosítója került eltárolásra, a kedvencekhez adott dolgok, valamint a barát és kérelem listák esetében is. Erre az adatok konzisztenciája miatt volt szükség, hiszen számos hibába futhatunk, ha már nem létező felhasználókra vagy listaelemekre akarunk hivatkozni.

5.1.2 Routeok

Ahhoz hogy a HTTP kérések fogadását ki tudjam szervezni külön fájlokba, a strukturáltság és átláthatóság miatt, bevezettem különböző route-okat az alkalmazásba. Ezek segítségével, ha egy adott URL-en érkezik egy kérés az app felé, az átirányítja a megfelelő osztály számára.

```
const ingredientsRouter = require("./routes/ingredients")
const drinksRouter = require("./routes/drinks")
const pubsRouter = require("./routes/pubs")
const usersRouter = require("./routes/users")

app.use("/ingredients", ingredientsRouter)
app.use("/drinks", drinksRouter)
app.use("/pubs", pubsRouter)
app.use("/users", usersRouter)
```

Így a frontenden egyértelműen és szeparáltan tudtam megadni a különböző kéréseket, amire azért volt szükség, mert a három tárolt dolgot érintő függvények nagyban megegyeztek egymással, és így nem történhetett olyan kavarodás, hogy rossz típusú adatbázisba próbáltam menteni az adatokat.

5.1.3 Felhasználó kezelés

Az alkalmazásban első nekifutásra a Firebase autentikációs részével próbáltam megoldani a felhasználókezelést. Ez egy olyan felhőalapú szolgáltatás, ami egy konfigurációs fájl és némi könnyen integrálható kód segítségével maga oldja meg a felhasználó adatainak megfelelő titkosítását és hitelesítését. A megszokott email-jelszó pároson kívül több fajta bejelentkezési lehetőséget biztosít, mint a Google, Facebook vagy X. A dolgozatomban viszont egy saját megvalósítású rendszert használtam, így csak az emaillel való bejelentkezésre van lehetőség.

A regisztráció során a frontendről érkező felhasználó és jelszó párost az említett *bcrypt* könyvtár segítségével lehashelem. Ez a függvény két paramétert vár: a titkosítandó szöveget, és egy úgynevezett só (salt) paramétert, ami a titkosítás erősségét határozza meg. Ezt követően létrehozok egy *User* példányt az adatokkal, és visszaküldöm ezt a frontendnek. Bejelentkezéskor hasonló történik, viszont a felhasználó létrehozása helyett

kikeresem az adatbázisból az emailhez tartozó hashelt jelszót, és a *bcrypt compare* függvénye segítségével összehasonlítom a kapott stringet az adatbázisból szerzett titkosított stringgel. Ha a kettő egyezik, akkor sikeres választ kap vissza a frontend, és megtörténik a bejelentkezés. Ha nem egyező eredményt ad vissza a függvény, akkor a hibát küldi vissza a szerver, amit az alkalmazásban egy felugró ablak jelez.

Amit a felhasználókkal kapcsolatban meg kellett oldanom, az a barát- és kérelemlista menedzselése. Az következő esetekre kellett végpontokat vagy segédfüggvényeket létrehoznom:

- A felhasználó barát-kérelmet küld B felhasználónak: Ekkor az A felhasználó id-ja bekerül a B felhasználó kérelem-listájába
- B felhasználó elutasítja a kérést: Ekkor értelemszerűen kikerül az A felhasználó azonosítója a kérelem-listából.
- B felhasználó elfogadja a kérést: Ebben az esetben is kikerül az elfogadott felhasználó azonosítója a kérelem listából, és emellett B felhasználó barátlistájába bekerül az A azonosítója, valamint az A barátlistájába is bekerül a B id-ja.
- A és B felhasználó már barátok, de valamelyik törli a másikat, ekkor mindketten törlésre kerülnek egymás barátlistájából.

  Az ezekre megírt útvonalakra érkeznek be a frontendről a kérések az esetnek megfelelő email és id párossal, majd a szükséges felhasználók megkeresése után frissülnek a szükséges listák, majd mentésre kerülnek a felhasználók, és az alkalmazásban a naprakész listák fognak szerepelni.

```
router.route("/acceptRequest").post(async (req, res) => {
  const { id, email } = req.body
  try {
    const acceptingUser = await Users.findById(id)
    if (!acceptingUser) {
      return res.status(404).json({ message: "User not found" })
    }
    const sendingUser = await Users.findOne({ email })
    if (!sendingUser) {
      return res.status(404).json({ message: "Sending user not found" })
    }
    sendingUser.requests = sendingUser.requests.filter(
      (requestId) => requestId.toString() !== acceptingUser._id.toString()
    )
    sendingUser.friends.push(acceptingUser._id)
    acceptingUser.requests = acceptingUser.requests.filter(
      (requestId) => requestId.toString() !== sendingUser._id.toString()
    )
```

```
    acceptingUser.friends.push(sendingUser._id)

    await acceptingUser.save()
    await sendingUser.save()
    return res.status(200).json({ message: "Request accepted successfully"
})
  } catch (error) {
    return res.status(500).json({ message: "Internal server error" })
  }
})
```

5.2 -Frontend:

5.2.1 Komponensek

A komponensek felépítésénél próbáltam minél jobban az újrafelhasználhatóságra törekedni, és hogy az egy csoportba tartozó komponensek szeparálva legyenek tárolva. Külön mappákba csoportosítottam a különböző panelek – hozzávalók, italok, kocsmák, térkép, felhasználók – komponenseit, valamint a komponensek gyökérkönyvtárában hagytam a közösen használt és általános osztályokat.

Egy panel alapvetően 4 komponensből áll:

- Az individuális elem megjelenítésért felelős komponens
- A komponenseket kilistázó és interakciókat kezelő komponens
- Egy formot és az ehhez használt függvényeket tartalmazó komponens, amit a hozzáadáskor és szerkesztéskor használok
- Egy másik formot, ami a megjelenített lista filterezését végzi

Az elemek komponensében a props-ban paraméterként átadott elem szükséges információit jelenítem meg, valamint ide vettem fel az elem szerkesztéséért felelős formot, ami egy felugró dialógus-ablakban jelenik meg. Mindenhol megjelenik az eltárolt elem neve, ezen kívül összetevő esetében a típusa, italok esetében a hozzá társuló kép és összetevők, kocsmák esetében pedig az értékelés található meg az elemet reprezentáló kártyán.

Hozzávalók és italok esetében az információk mellett a kedvencekhez adáshoz található egy gomb, a kocsmáknál ez mellett helyet kaptak gombok az értékelésre és a menü megtekintésére, valamint az egyik gomb segítségével az alkalmazás átirányít a térkép felületre, a kiválasztott kocsma megjelenítésével együtt.

Az hozzáadást és szerkesztést ellátó űrlapokat megvalósító komponensek viszonylag egyszerűen épülnek fel. Mind egy HTML form elemet ad vissza, az adott típusú elem adataihoz tartozó és illő beviteli mezőkkel. Ezek értelemszerűen a névnél és hasonló szöveges adatoknál egy egyszerű input mező, alkoholfok vagy koordináta megadásánál számok, egyéb specifikus adatoknál pedig legördülő menüben lehet kiválasztani a kívánt értéket. Ilyen például az ital hozzáadásánál a típus vagy a hozzá tartozó pohár fajtája.

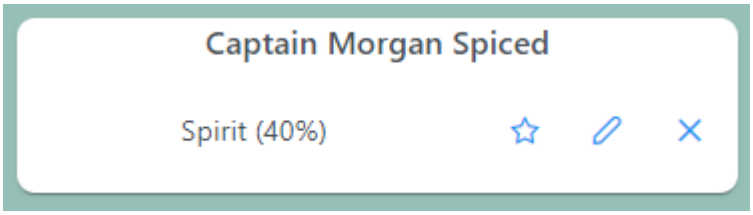
A szűrők beállításáért felelős űrlap hasonló az előzőhöz, csak itt felugró dialógus-ablak helyett a lista felett jelennek meg a beviteli mezők. Az elemek között rá tudunk szűrni a névre, típusra, italoknál arra, hogy milyen összetevőt tartalmazzon, valamint szórakozóhelyek esetében értékelés alapján is tudjuk szűkíteni a keresést. Itt nyilván kevesebb beviteli mezőt kellett megadni, illetve a form elrendezése a legtöbb weboldalon megszokotthoz hasonlóan vertikálisan lett megoldva.

A térkép oldal 2 fő komponensből áll, egy ami a térkép megjelenítésért felel, és egy ami az adott pinekhez tartozó információs panelt jeleníti meg.

A kocsmákért felelős komponenseknél jobban szétbontottam a működés megvalósítását, mert az itt tárolt adatok komplexitása miatt a hozzáadáshoz és szerkesztéshez több komponenst kellett létrehoznom a formokhoz. A menü szerkesztéséhez egy külön felugró ablakot hoztam létre, ahol látható a menüben már megtalálható elemek listája, valamint egy egyszerű form segítségével, ami egy név – ár párost vár, hozzáadhatunk elemeket a menühöz. A nyitva tartás szerkesztésénél táblázat-szerűen jelennek meg a hét napjai és mellettük a hozzájuk tartozó nyitás és zárás időpontja. Az alkalmazás jelenleg úgy kezeli azt az esetet, hogy az adott napon zárva tart a hely, hogy 0:00 – 0:00-ig van megadva a nyitvatartása.

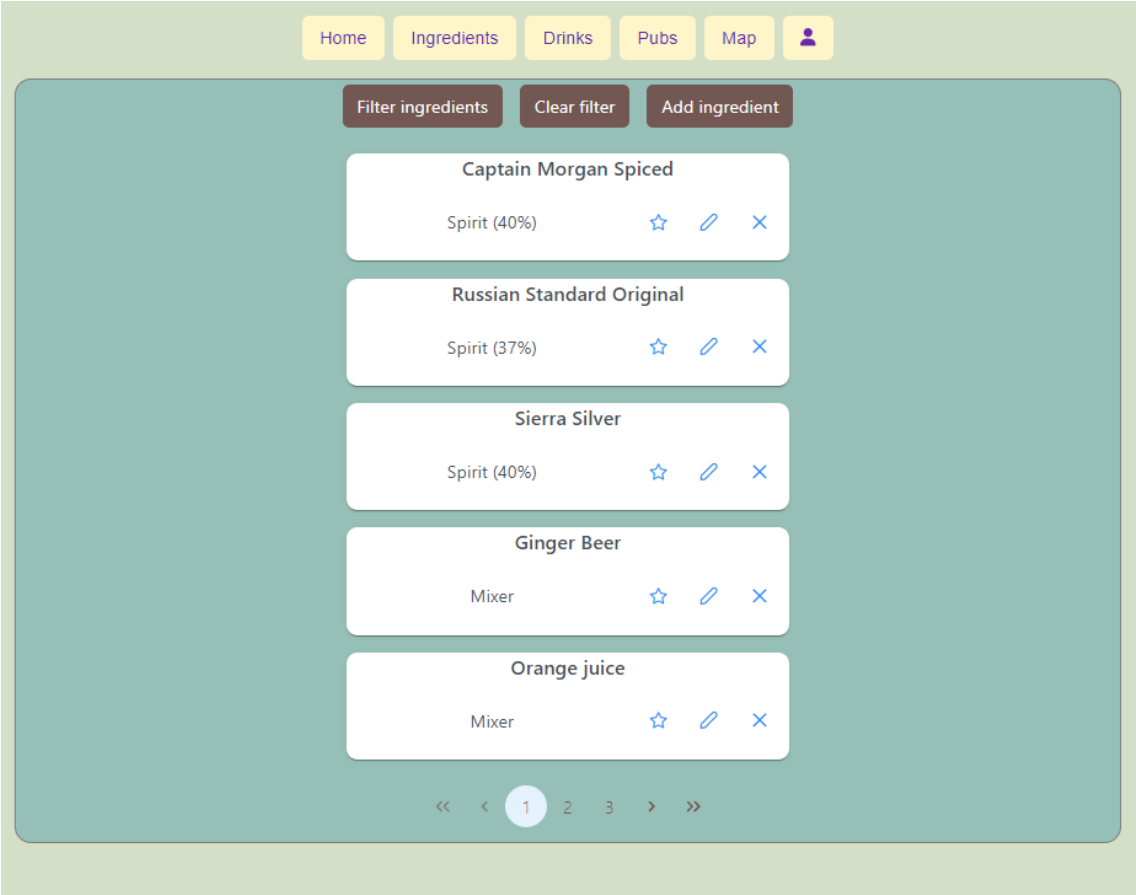
A paneleket alkotó komponensek felépítését a következőképpen valósítottam meg. Minden komponenshez definiáltam egy típust, ami a hozzá tartozó props elemeit határozza meg. Ezt követően a komponens arrow function-nal való létrehozása után először definiálom a felhasznált konstans változókat, szinte mindenhol a useState hook segítségével, hogy később változtatni lehessen az adatokat, és dinamikusan megjeleníteni ezeket. Aztán a komponensben felhasznált segédfüggvényeket hozom létre, majd következik a komponens által visszaadott HTML elem megalkotása.

Ez az individuális elemek komponenseiben egy PrimeReactból importált Card elem, ami tartalmaz egy Title elemet az objektum nevével, valamint egy Stack elemet a komponenssel való interakcióhoz szükséges gomboknak. Alap esetben három gomb található a komponensekben, egy a kedvencekhez adásért felel, egy a szerkesztést, az utolsó pedig a törlést valósítja meg. Ezek mellett megtalálható még valamilyen ismertető szöveg az adott elemről, és egyes komponensek esetén egy képet is megjelenítek.



6. ábra Egy megjelenített elem

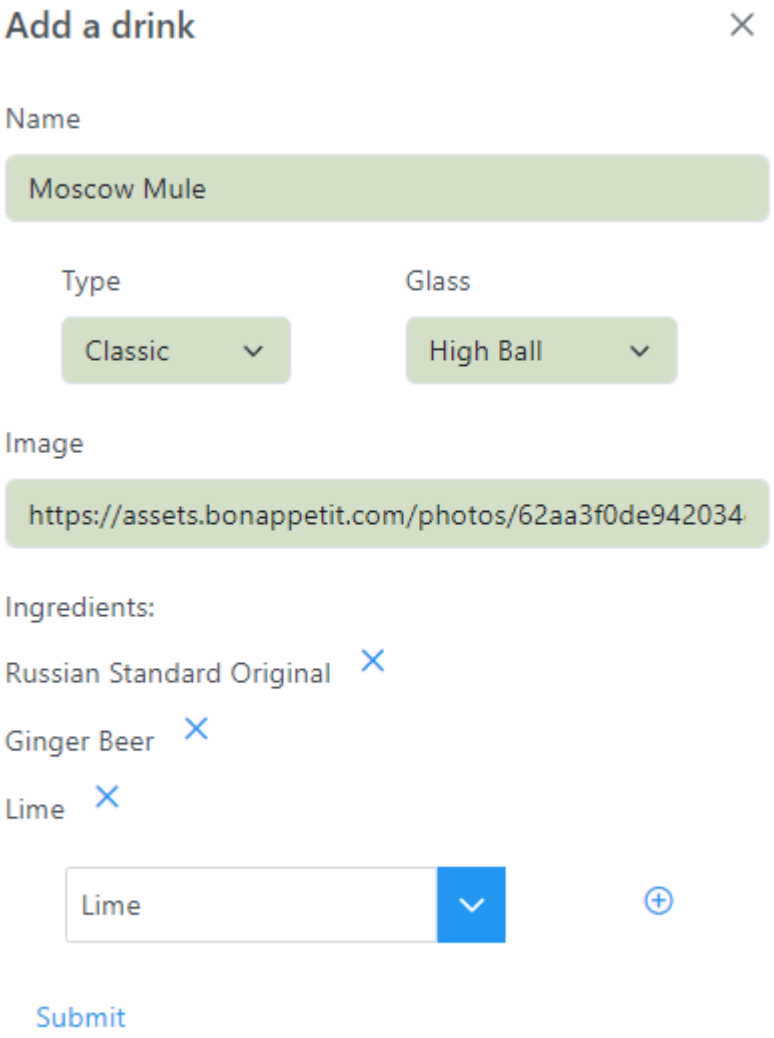
Magát a panelt kezelő komponens felépítése hasonlít az indiviudális elemekéhez, a visszaadott elem felépítésében azonban nagyban eltér. A panel legtetején találhatóak a filtert kezelő elemek. Egy gomb segítségével tudjuk megjeleníteni vagy elrejteni a kereső mezőket, és a szűrést végző gombot. Ezután az objektumok hozzáadását rejtő form dialógusablaka található, de ez is értelemszerűen csak a megfelelő gomb megnyomására válik láthatóvá. Ez alatt listázódnak ki a panelhez tartozó elemek, a javascript tömbökön használható *.map()* metódussal. Végül a PrimeReact Pagination komponens egy eleme található itt, ami paraméterül kapja az oldal méretet és a kilistázott lista hosszát, és ez alapján jeleníti meg az oldalak közötti váltást megvalósító gombokat.



7. ábra A hozzávalók panelje

Az elemek hozzáadásáért és szerkesztéséért felelős formok mezői az elem típusától nagyban függenek, de ahol lehetett label-öket helyeztem el a beviteli mezőkön, és ahol logikusnak érződött, például az italok hozzáadásánál a hozzávalók kiválasztása esetében, ott legördülő menü segítségével hoztam létre az adatbevitelhez szükséges mezőt.

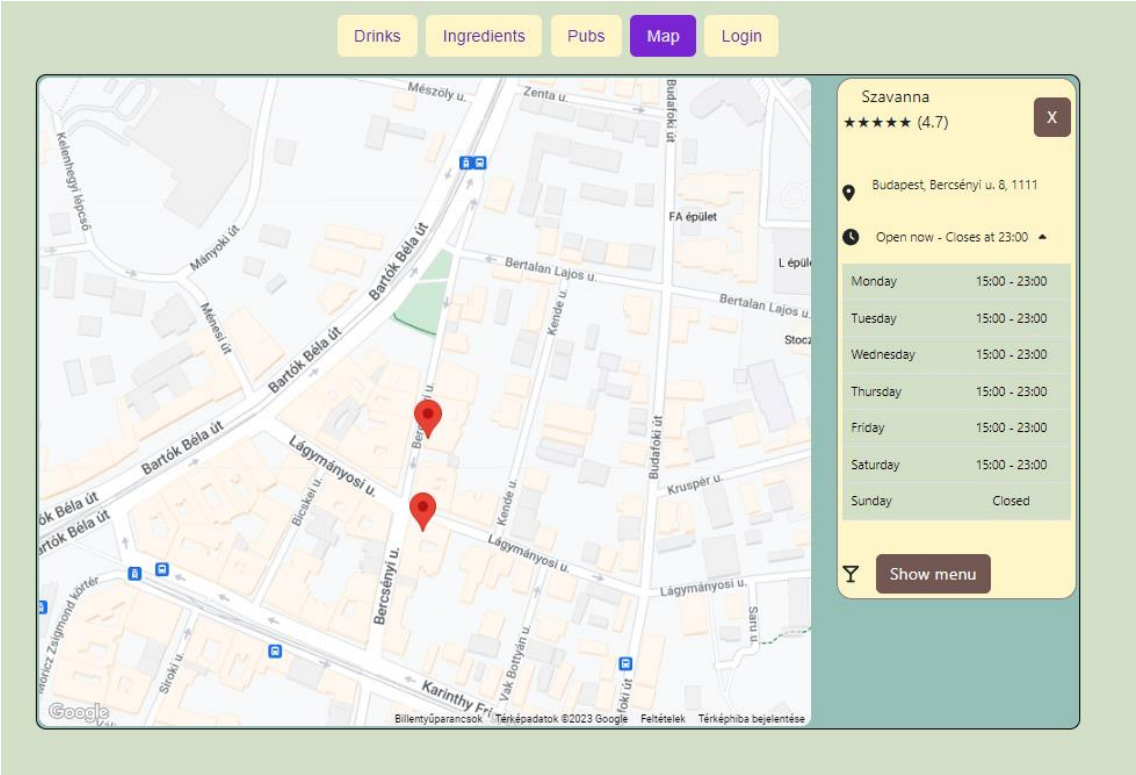
A kocsmák hozzáadása esetén, a menü és nyitvatartási idő elemeinek összetettsége miatt, ezek módosítását egy külön felugró ablakban található formban valósítottam meg, de ezekben is hasonló logika mentén jártam el.



8. ábra Az italok hozzáadását kezelő form

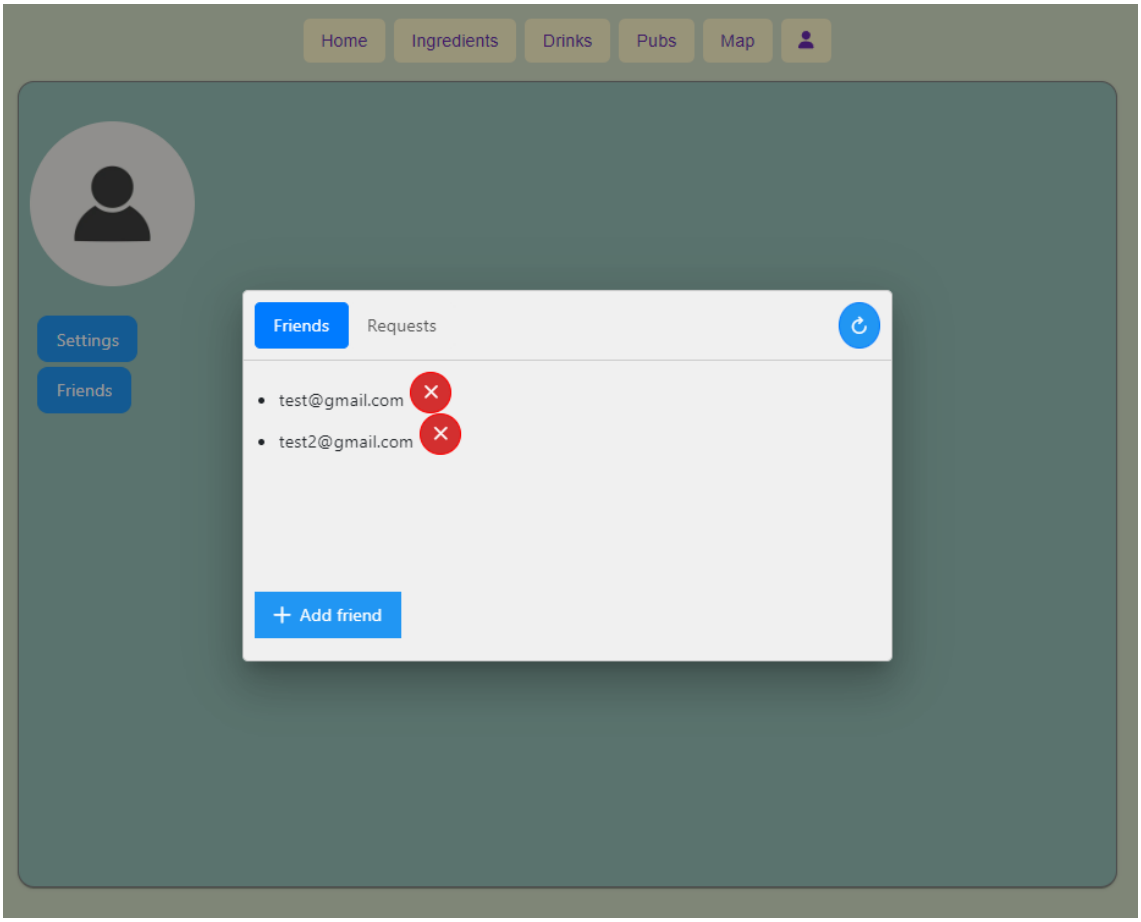
A térkép panel két fő részből áll, magát a térképet tartalmazó komponensből, és a kijelölt vendéglátói egység információit megjelenítő oldalsó panelből. Ez a térképen található gombostűkre kattintva jeleníthető meg illetve frissíthető a tartalma. A térkép egy react-google-maps könyvtárból importált komponens, ami paraméterként megkapja a komponensben korábban specifikált adatokat a térkép megjelenítéséhez. Ilyen például a térkép stílusa, ahol ki lehet kapcsolni a számunkra nem szükséges gombostűket a térképen, vagy a térkép színpalettáját konfigurálhatjuk. A térképen belül kilistázom az adatbázisban szereplő kocsmák helyzetét az eltárolt szélességi és hosszúsági koordináták segítségével.

Az oldalsó panel paraméterül megkapja a kiválasztott kocsmát, és megjeleníti az arról eltárolt információkat. A menü egy felugró ablakként jelenik meg, a nyitvatartási idő pedig kattintásra lenyitható, hogy ne csak az adott napi nyitva tartást mutassa, hanem a többi naphoz tartozót is.



9. ábra A térkép panel

A felhasználót megjelenítő panelen két gomb megnyomására felugró dialógus ablak található. Az első egy olyan formot hoz fel, amivel a felhasználó szerkesztheti az adatait, mint a felhasználóneve vagy a jelszava, valamint törölheti a profilját. A másik gomb rejti a barátlistát és a kérelmeket. Ez a kettő között a fejlécben elhelyezett gombokkal tudunk váltani. A barátokat egy mellettük megjelenő gomb segítségével szükség szerint törölhetjük, a kérelmek mellett a törlésen kívül egy elfogadásra szolgáló gomb is helyet kapott. A barát hozzáadása gomb megnyomására megjelenik egy beviteli mező, ahova beírhatjuk a hozzáadni kívánt felhasználó valamilyen elérhetőségét.



5.2.2 Jogosultságok

Az alkalmazásban a jogosultság kezelés még elég kezdetleges, egy boolean változó felel azért, hogy éppen megjelenít-e adott komponenseket az app, vagy sem. Így az alap felhasználók nem férnek hozzá olyan metódusokhoz mint az elemek szerkesztése vagy törlése.

10. ábra Különbség a felhasználók által látott elemek között

5.2.3 Routing

Kezdetben az alkalmazás teljes mértékben egy SPA-nak (Single Page Application) indult, de néhány fontos funkció, és az egyszerűbben olvasható és navigálható kód miatt is meg kellett valahogy oldanom az oldalak közötti navigálást. Ezt a React Router nevezetű könyvtár segítségével tettem meg.

A könyvtár használatához a következőképpen kell felépítenünk a komponensfánkat: Legfelül egy Router komponensnek kell lennie, ebből több típust biztosít számunkra a könyvtár, én a BrowserRouter-t használtam. Ezen belül a Routes komponens található, amin belül az eltérő útvonalakat kezelő Route van. Ezeknek a Route komponenseknek meg kell adnunk a relatív elérési utat, és hogy milyen komponenst jelenítsen meg.

```
<BrowserRouter>
     <Routes>
          <Route path="/" element={<Menu />}>
          <Route path="/user" element={<UserPanel />}></Route>
          <Route path="/ingredients"
               element={<IngredientsPanel />}></Route>
          <Route path="/drinks" element={<DrinkPanel />}></Route>
          <Route path="/pubs" element={<PubPanel />}></Route>
          <Route path="/map" element={<Map />}></Route>
          </Route>
     </Routes>
</BrowserRouter>
```

Az oldalak közti navigáláshoz a könyvtár által nyújtott Link komponenst használtam. Itt a gyerekéül adott komponens - ami jellegzetesen, és az én esetemben is

egy gomb - jelenik meg, és az erre való kattintással az alkalmazás a Link-nek paraméterül adott oldalra navigál.

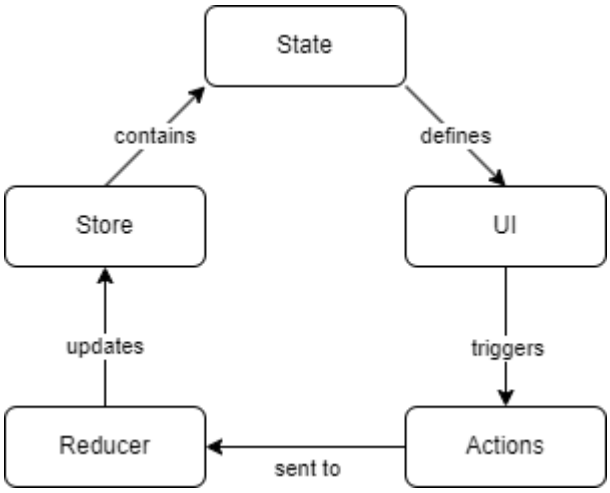
```
<Link to="/user">
     <button>
          <FontAwesomeIcon icon={faUser} />
     </button>
</Link>
```

Emellett a useNavigate hook által biztosított navigate függvény segítségével akár függvényhívásokon belül is végezhetünk két oldal közötti navigálást.

```
const onLocationPinClick = (pubId: string) => {
    dispatch(setSelectedPubId(pubId))
    navigate("/map")
  }
```

5.2.4 Redux

Ahhoz, hogy az alkalmazásban tárolt statek könnyen beállíthatók, lekérdezhetők, és ami a legfontosabb, konzisztensek legyenek, a React Redux könyvtárat használtam. Ez biztosít egy úgynevezett *store*-t amiben a tervezési logika alapján szétbontva tároljuk a különböző állapotváltozóinkat, és az ezeket manipuláló *reducereker*



11. ábraA Redux működési ciklusa

A redux működéséhez definiálnunk kell két fontos hookot, a *useAppDispatch*-et és a *useAppSelector*-t. Az előbbi látja el a az állapotok változtatását, míg az utóbbi segítségével olvashatjuk ki a szükséges state-eket. Ahogy az ábrán is látható, a

komponensekben nem közvetlenül változtatjuk az állapotokat, hanem egy, a *useAppDispatch* hook által szolgáltatott *dispatch* függvény hívásával elküldjük a megfelelő reducernek, hogy hajtsa végre az állapot frissítését a store-ban. Mivel a komponensekben ezeket az állapotokat a useAppSelector segítségével adjuk meg, ezért ha a store-ban frissül valamelyik, az őt tartalmazó komponens ennek hatására újra-renderelődik, így oldja meg a Redux, hogy az alkalmazásban mindig a legfrissebb állapotok jelenjenek meg.

A store használatához először inicializáltam a megfelelő importokkal és változók meghatározásával, majd az egész alkalmazást egy Provider wrapper komponensbe tettem,hogy mindenhonnan elérhetőek legyenek a Redux nyújtotta funkciók.

```
export const store = configureStore({
  reducer: {
    uiState: UIStateReducer,
    user: UserReducer,
    lists: ListReducer,
    ingredients: IngredientReducer,
    pub: PubReducer,
  },
})

export type AppDispatch = typeof store.dispatch
export type RootState = ReturnType<typeof store.getState>
```

Itt láthatók a különböző logikai egységeket összefogó reducer-ek, amiket a Redux sajátos szintaktikájával kell létrehozni. Ezekben a fájlokban először létrehozom az ott kezelt elemek típusát, és ha szükséges, akkor egy példányt belőle, ami az alapállapotot fogja nyújtani. Ezt követően megírtam magát a Slice-ot, ami az állapot beállításokat végzi. Ennek meg kell adni egy nevet, egy kezdeti állapotot, és a különböző reducer-eket. Ezek azok a függvények, amelyeket az alkalmazásban meghívhatunk a dispatch() metódus segítségével. Az alap reducereken kívül meg lehet még adni úgynevezett extra reducereket, amelyek bizonyos események után futnak le, és általában bonyolultabb működést valósítanak meg. Egyik legjellemzőbb használata ezeknek, ha különböző aszinkron hívások sikeres befejezésekor akarunk lefuttatni bizonyos kódrészleteket. Az alkalmazásban nem láttam szükségesnek ezeknek a használatát. Ezen kívül létrehoztam különböző lekérdező metódusokat a szükséges állapotokhoz.

```
export interface User {
  email: string
  username: string
  user_id: string
  role: string
  favourited: Favourited
  friends: string[]
  requests: string[]
}

const initialState: User = {
  email: "",
  username: "",
  user_id: "f",
  role: "user",
  favourited: { ingredients: [], drinks: [], pubs: [] },
  friends: [],
  requests: [],
}

export const UserSlice = createSlice({
  name: "user",
  initialState,
  reducers: {
    setEmail: (state, action: PayloadAction<string>) => {
      state.email = action.payload
    },
    setUsername: (state, action: PayloadAction<string>) => {
      state.username = action.payload
    },
     . . .
  },
})

export const getUserId = (state: RootState) => state.user.user_id
export const getEmail = (state: RootState) => state.user.email
. . .

export const {
  setEmail,
  setUsername,
  . . .
} = UserSlice.actions

export default UserSlice.reducer
```

6 Telepítési útmutató

Az alkalmazás jelenleg csak lokálisan futtatható. Ahhoz, hogy bárhonnan lehessen futtatni az alkalmazást, néhány változtatást kellett alkalmaznom a használt szolgáltatások konfigurációjában. Alap esetben egy MongoDB clusterhez csak előre megadott IP címekről lehet csatlakozni. Az adatbázis felhőalapú alkalmazásában, a MongoDB Atlasban ideiglenesen hozzáadtam a 0.0.0.0/0 IP címet, ezáltal bármilyen számítógépről elérhetővé vált. A Google Cloudban, nem találtam IP cím szerinti hozzáférést, ott egy API kulcs védi az adatokhoz való hozzáférést. Ez megtalálható a feltöltött dolgozatban, így elérhetőnek kell lennie az alkalmazás indításakor. Szűk keresztmetszet lehet az, hogy a Google Cloudban ingyenesen 3 hónapig lehet használni a szolgáltatásokat. Ez a dolgozat beadása után nagyjából egy hónappal fog lejárni.

Az alkalmazás futtatásához egy telepített Node.js-re van szükség, amit egy előre összerakott telepítővel, vagy akár parancssorból, valamilyen node verziókezelővel telepíthetünk. Ezután a szerver futtatásához el kell navigálnunk a /barfinder-server/ mappába, ahol a megfelelő függőségek telepítése és frissítése után el tudjuk indítani a szervert.

```
npm install
npm start
```

A kliens futtatása hasonlóképpen történik, ugyanezt a két parancsot kell kiadnunk egy másik terminálból, amiben a /barfinder/ mappába navigáltunk

7 Összegzés

A diplomamunkám során egy italokkal és szórakozóhelyekkel foglalkozó webalkalmazást és a hozzá tartozó szervert készítettem el. Azért is esett erre a választás, mert valamilyen full stack alkalmazást képzeltem el témaként, és az adta az ihletet, hogy hobbi szinten szoktam pultozni, így volt egy alapvető kezdeti motiváció.

Mindenképpen a lehető legtöbbet akartam tanulni a dolgozat elkészítése alatt, felmerült az is, hogy egy más, számomra azelőtt ismeretlen technológiával írjam meg a szervert, de az túl nagy kihívásnak bizonyult. Először implementáltam valamilyen felhőszolgáltatást az alkalmazásomban, és a Typescript-tel is csak enyhén ismerkedtem, mielőtt belevágtam volna az alkalmazás fejlesztésébe.

Az elkészült alkalmazás alapjai működnek, az elérhető italok és helyek közötti keresés, és a felhasználók autentikációja és egymás közötti interakciója. A feladatkiírásban szereplő különböző helyeken való bejelentkezéseket, és az ebből generálódó hírfolyamot végül nem sikerült implementálnom.

A jövőben ezen kívül valamivel jobban részleteire bontanám az eltárolt italokat, hogy még specifikusabban lehessen keresni köztük, valamint számon lehetne tartani a felhasználók profiljában egy otthon elérhető alapanyag készletet, és az alkalmazás ezek alapján is tudna szűrni italokat. Tervben volt még valamilyen módon a mesterséges intelligencia használata, valamilyen következtető modell alkalmazásával implementálni egy ajánló rendszert a korábbi preferenciák alapján.

8 Köszönetnyilvánítás

Szeretnék köszönetet mondani a konzulensemnek, Kövesdán Gábornak, aki türelmessége és rugalmassága mellett segítette a munkámat. Továbbá szeretném megköszönni a családomnak és barátaimnak a támogatását.

Irodalomjegyzék

[1] Node.js https://nodejs.org/docs/latest/api/

[2] MongoDB https://www.mongodb.com/docs/

[3] React https://react.dev/

[4] Primereact https://primereact.org/

[5] React Redux https://react-redux.js.org/

[6] Axios https://axios-http.com/docs/intro

[7] Typescript https://www.typescriptlang.org/docs/

[8] git https://git-scm.com/doc

[9] Google Cloud https://cloud.google.com/docs

[10] REST API https://www.ibm.com/topics/rest-apis