

Java backend javascript frontend alapú webalkalmazás

Hódi Szilárd

Programtervező informatikus BSc

Konzulens: Dr. Vályi Sándor

**2024**

**Tartalomjegyzék**

[1. BEVEZETÉS 4](#__RefHeading___Toc2890_2368991513)

[1.1. Alkalmazás célja 4](#__RefHeading___Toc1781_1730593459)

[2. IRODALMI ÁTTEKINTÉS 4](#__RefHeading___Toc1874_4059630159)

[2.1. JavaScript 4](#__RefHeading___Toc1783_1730593459)

[2.2. TypeScript 5](#__RefHeading___Toc1785_1730593459)

[2.3. React 5](#__RefHeading___Toc1787_1730593459)

[2.4. Mi az a SPA? 5](#__RefHeading___Toc1789_1730593459)

[2.5. Redux 5](#__RefHeading___Toc1791_1730593459)

[2.6. React rooter: 6](#__RefHeading___Toc1793_1730593459)

[2.7. Java 6](#__RefHeading___Toc1795_1730593459)

[2.8. Spring Boot 6](#__RefHeading___Toc1797_1730593459)

[2.9. Maven. 6](#__RefHeading___Toc1799_1730593459)

[2.10. Mi az API: 7](#__RefHeading___Toc1803_1730593459)

[3.11. Visual Studio Code 7](#__RefHeading___Toc1812_1730593459)

[2.12. Swager UI / Postman 8](#__RefHeading___Toc1814_1730593459)

[2.13. Chrome extensions 8](#__RefHeading___Toc1816_1730593459)

[2.14. Chacra UI 8](#__RefHeading___Toc1818_1730593459)

[2.15. Xampp 8](#__RefHeading___Toc1820_1730593459)

[3. A FEJLESZTÉS MENETÉNEK LEÍRÁSA 8](#__RefHeading___Toc4195_1388287306)

[3.1. Frontend 8](#__RefHeading___Toc1848_1730593459)

[3.1.1. Navbar: 9](#__RefHeading___Toc1852_1730593459)

[3.1.2 Icon: 10](#__RefHeading___Toc1854_1730593459)

[3.1.3 Témák: 10](#__RefHeading___Toc1858_1730593459)

[3.1.4 Műveletek: 10](#__RefHeading___Toc1860_1730593459)

[3.1.5 Keresési mező 11](#__RefHeading___Toc1862_1730593459)

[3.1.6 Regisztráció 14](#__RefHeading___Toc1866_1730593459)

[3.1.7 Bejelentkezés 17](#__RefHeading___Toc1917_3355630758)

[3.1.8 Felhasználók listája 19](#__RefHeading___Toc1934_3355630758)

[3.1.9 Új cikk hozzáadása 21](#__RefHeading___Toc1936_3355630758)

[3.1.10 Cikk szerkesztése 22](#__RefHeading___Toc1938_3355630758)

[3.1.11. Carousel komponenes 22](#__RefHeading___Toc1950_3355630758)

[3.1.12 Single user page 23](#__RefHeading___Toc2060_3355630758)

[3.1.13 News 25](#__RefHeading___Toc2062_3355630758)

[3.1.13.1 NewsListProvider 25](#__RefHeading___Toc2066_3355630758)

[3.1.13.2 NewsList 26](#__RefHeading___Toc2068_3355630758)

[3.1.13.3 NewsListItem 27](#__RefHeading___Toc2070_3355630758)

[3.1.13.4 NewsDescriptionProvider 27](#__RefHeading___Toc2074_3355630758)

[3.1.13.5 NewsDescription 29](#__RefHeading___Toc2076_3355630758)

[3.1.14 Comment 29](#__RefHeading___Toc4319_662140629)

[3.1.14.1 MyComment 29](#__RefHeading___Toc2078_3355630758)

[3.1.14.2 CommentForm 29](#__RefHeading___Toc2080_3355630758)

[3.1.15 Store directory 29](#__RefHeading___Toc2082_3355630758)

[3.1.15.1 Store.ts 30](#__RefHeading___Toc2088_3355630758)

[3.1.15.2 Slices 31](#__RefHeading___Toc2090_3355630758)

[3.1.15.3 News-api.ts 32](#__RefHeading___Toc2092_3355630758)

[3.1.15.4 Query 33](#__RefHeading___Toc2094_3355630758)

[3.1.15.5 Mutation 33](#__RefHeading___Toc2096_3355630758)

[3.1.16 Hookok 34](#__RefHeading___Toc2098_3355630758)

[3.1.17 Utils directory 36](#__RefHeading___Toc2100_3355630758)

[3.1.18 Componensek renderelésére, alias nevek használata 36](#__RefHeading___Toc2102_3355630758)

[3.2. Backend 42](#__RefHeading___Toc2084_3355630758)

[3.2.1 Configuration 42](#__RefHeading___Toc2322_4277652606)

[3.2.2 Controller 45](#__RefHeading___Toc2104_3355630758)

[3.2.2.1AutenticationContoller: 45](#__RefHeading___Toc2106_3355630758)

[3.2.2.2 UsersContoller ,NewsController: 46](#__RefHeading___Toc2108_3355630758)

[3.2.2.3.RolesContoller 46](#__RefHeading___Toc2110_3355630758)

[3.2.2.4 CommentContoller 46](#__RefHeading___Toc2112_3355630758)

[3.2.2.5 ImageController 46](#__RefHeading___Toc2114_3355630758)

[3.2.3 Dto 47](#__RefHeading___Toc2116_3355630758)

[3.2.4 Entity 47](#__RefHeading___Toc2118_3355630758)

[3.2.5 Service 48](#__RefHeading___Toc2120_3355630758)

[3.2.6 Email küldés. 48](#__RefHeading___Toc2324_4277652606)

[3.2.7 Hibakezelés 48](#__RefHeading___Toc2326_4277652606)

[3.2.8 Repository 48](#__RefHeading___Toc4321_662140629)

[3.3 Adatbázis 49](#__RefHeading___Toc2086_3355630758)

[3.3.1 Java Persistence API (JPA): 49](#__RefHeading___Toc2328_4277652606)

[3.4. További fejlesztési lehetőségek: 50](#__RefHeading___Toc2330_4277652606)

[3.4.1 Email 50](#__RefHeading___Toc2332_4277652606)

[3.4.2 Levelezés 51](#__RefHeading___Toc2334_4277652606)

[3.4.3 User Details kibővítése, értesítések 51](#__RefHeading___Toc2336_4277652606)

[3.4.3 Password 51](#__RefHeading___Toc2338_4277652606)

[4.Eredmények: 51](#__RefHeading___Toc4341_662140629)

[4.1 Frontend: 51](#__RefHeading___Toc4593_662140629)

[4.2 Backend: 52](#__RefHeading___Toc4595_662140629)

[5.Elemzés: 52](#__RefHeading___Toc4343_662140629)

[6.Összefoglalás: 52](#__RefHeading___Toc2340_4277652606)

[9.IRODALMI JEGYZÉK 53](#__RefHeading___Toc4660_1388287306)

[10. MELLÉKLET 53](#__RefHeading___Toc4658_1388287306)

[ALKALMAZÁS INDÍTÁSA 53](#__RefHeading___Toc1822_1730593459)

[10.1. GitHub verziókezelés 53](#__RefHeading___Toc1824_1730593459)

[10.2 Docker 53](#__RefHeading___Toc1876_4059630159)

# 1. BEVEZETÉS

Fiatal korom óta foglalkozom a programozással hobbi szinten több programozási nyelvet is megismertem különböző mélységekben. Eddig még nem volt olyan területe az szakmának amely különösebben megfogott volna. Azonban ezzel a projektek sikerült eldöntenem milyen irányba szeretnék tovább haladni. A fullstack programozók amolyan svájci bicskák akik mind a frontend mind a backend fejlesztésben részt tudnak venni, illetve önállóan is képesek készíteni komplett .

Először az alkalmazás könyvtár architektúrája alapján szerettem volna strukturálni a dokumentáció tartalom jegyzékét azonban így 5-6 szintig is egymásba ágyazódnának a bekezdések, ami miatt kissé áttekinthetetlen lenne. Ehelyett felsorolás jellegűen számoztam.

## 1.1. Alkalmazás célja

Egy fiktív hírportál frontend és backendjét valósítottam meg. Célom ezzel felkészülni, illetve megalapozni a további specializálódási irány vonalamat. Készíteni egy referencia munkát, amelyben összefoglalom tudásom jellegi szintjét, egy későbbi munkahelyéhez.

# 2. IRODALMI ÁTTEKINTÉS

## 2.1. JavaScript

1995 ben megjelenő objektum orientált, interpretált programozási nyelv, amely elsősorban a webfejlesztésben használatos. Kliensoldali szkriptnyelvként fut a böngészőkben, interaktivitást és dinamizmust biztosítva a weboldalak számára. Az aszinkron programozás és eseményvezérelt modell révén hatékonyan kezeli az aszinkron feladatokat.

A folyamatos fejlődés eredményeként az ES6 és későbbi verziók számos modern nyelvi elemet vezettek be továbbá a TypeScript, mint statikus típusellenőrzést támogató kiterjesztése is terjedt. A JavaScript széles körben támogatott, és a fejlesztők a frontendtől a backendig sokféle alkalmazásban használják.

Az interpretált nyelv (interpreted language) olyan programozási nyelvet jelent, amelynek forráskódját nem közvetlenül a gépi kódra fordítják le, hanem egy másik program, az ún. interpreter hajtja végre. Az interpreter a forráskódot soronként olvassa és értelmezi, majd azonnal végrehajtja.

## 2.2. TypeScript

A TypeScriptet fejlesztették ki annak érdekében, hogy kiterjessze a JavaScript nyelvet olyan funkciókkal, amelyek lehetővé teszik a típus biztosabb és strukturáltabb kódírását. A TypeScript forráskódját transzpiler segítségével fordítják le JavaScript forráskóddá. A transzpiler a TypeScript forráskódot átalakítja olyan formára, amelyet a JavaScript motorok értelmezni tudnak.

Ez azt jelenti, hogy a TypeScript a fejlesztői szakaszban nyújt előnyöket (mint a típusellenőrzés), de a futtatási időben JavaScriptként fut.

## 2.3. React

Egy nyílt forrású felhasználói felület készítésére használt könyvtár. Melyet a Facebook fejlesztett ki 2011ben és tett nyílt forrásúvá 2013-ban. A hatékonyan kezeli a DOM manipulációt a virtuális DOM segítségével, ami növeli a teljesítményt és optimalizálja a frissítéseket. Több könyvtárral együtt szokás használni mint a Redux, React Router, Redux Toolkit. Ezeket használtam fel én is a ebben a programban.

## 2.4. Mi az a SPA?

A Single Page Application nem az egész oldalt tölti újra az egyes felhasználói akciók hatására, hanem a DOM fát manipulálva csak annak részei frissülnek. Ezáltal az ilyen alkalmazások sokkal gyorsabbak és nagyobb felhasználói élmény nyújtanak. Hátrányuk hogy a böngészőben engedélyezni kell a futtatáshoz szükséges JavaScriptet. Tovább, hogy jobban kivannak téve a hacker támadásoknak és nem keresőbarátok.

## 2.5. Redux

Egy állapotkezelő könyvtár amelyet elsősorban react alkalmazásokban használnak. Feladata egy konzisztens állapot megőrzése. Ezzel volt a legtöbb problémám.  
Az alkalmazás állapotát egy úgynevezett storeban tárolja amelyet a felhasználói felületről érkező actionok hatására reducereken keresztül változtat meg.

## 2.6. React rooter:

1.ábra felhasználói interakció hatása

Az alkalmazás oldalai közötti navigációra használt könyvtár

## 2.7. Java

A JAVA egy platform független obejtum orientált programozási nyelv . Platform független, mivel a futtató eszközön elegendő csak a JVM (java virtual machine) megléte.

## 2.8. Spring Boot

A Spring Framework-re épül. Spring Framework használatával kapcsolatban nincs komolyabb tapasztalatom rögtön a Spring Bootal kezdtem.

Két fontos fogalmat kell ismernünk első az IoC (Inversion of Controll) egy tervezési minta, amely azt mondja ne az objektum döntse el hogyan kapcsolódik más objektumokhoz. Második a DI (Dependency Injection) ez az IoC egy megvalósítása ezt használja a Spring Boot.

## 2.9. Maven.

Nyílt forrású projektkezelő és buildelő rendszer melynek segítségével könnyen lehet kezelni az alkalmazás függőségeit.

Egy pom.xml fájl tartalmazza az alkalmazás függőségeit és build beállításait, verziókat és egyéb projekthez kapcsolódó információkat.

1. Telepítés után a *mvn -version* és a *java -version* parancsokkal tudjuk ellenőrizni a telepítést.

Fontosabb maven parancsok:

*mvn clean install* ezzel egy lépésben letörli az előző build fájljait a target mappából és újra build-eli a projektet minek során létrejön az alkalmazásunk futtatható .jar kiterjesztése

*mvn spring-boot:run* futtatja a springboot alkalmazást az IDEA-ba integrál Tomcat szerver segítségével amit alapértelmezetten a localhost:8080-as porton érhetünk el .

A **Maven repository** Maven központi tárolója, amely az összes közösségi által fejlesztett és támogatott könyvtárat és függőséget tartalmazza. Az alapértelmezett Maven konfigurációkban azok a függőségek, amelyeket a projektjéhez hozzáad, automatikusan innen kerülnek letöltésre. A fejlesztők saját, vagy vállalatukhoz kötött Repository-ket is létrehozhatnak, ahol a belső fejlesztés során használt saját könyvtárakat, modulokat és függőségeket tárolhatják. Ezek a Repository-k lehetnek privátak és bizonyos jogosultságokhoz kötöttek, így csak az adott vállalat fejlesztői férhetnek hozzájuk.

## 2.10. Mi az API:

Az API (Application Programing Interface), ez az interface lehetőséget nyújt a program számára, hogy más programokkal kommunikáljon.

Felhasználásuk szerint több típusúak lehetnek, mint a

**-web:** API-k amelyek webes protokollokat használnak kommunikációra, például http (GET, POST, PUT, DELETE)

**-library:** API-k egy könyvtár vagy repository kiszolgáló interface

**-hardware:** API-k melyek különböző fizikai eszközök kommunikációját segítik.

## 3.11. Visual Studio Code

VS Code egy Microsoft által fejlesztett eszköz, rengeteg program nyelv támogatásával és plugin-nel, amelyek megkönnyítik a kódolást, ezt használtam a frontend elkészítéshez.

## 2.12. Swager UI / Postman

Ezt a két alkalmazást használtam a backend API endpointok teszteléséhez.

SwaggerUI dependencia ként tudjuk megadni a SpringBoot alkalmazásunk pom.xml fileban ezután, jelen projektemben a [http://localhost:8080/swagger-ui/index.html#/](http://localhost:8080/swagger-ui/index.html" \l "/) oldalon elérhető lesz egy felület mely az alkalmazásban definiált endpointokat felismeri és pár kattintással tudunk különböző requesteket küldeni. Spring security miatt configurálni kell az elérési utat, amíg ezt nem fixáltam a Postmant használtam.

## 2.13. Chrome extensions

Webalkalmazás révn a fejlesztéshez böngésző szükséges, a chrome stabil népszerű böngésző. Remek pluginokal a reactos alkalmazások teszteléséhez én a   
-Ract Developer Tools

-Redux DevTools

Előbbit kevésbé utóbbit folyamatosan a Redux state változásának figyeléskéshez,

## 2.14. Chacra UI

A Reacthez kifejlesztett, felhasználói felület kialakítására használható könyvtár. Könnyen és gyorsan készíthető vele egységesen stilizálható felhasználói felület. Kombinálható más komponens könyvtárakkal, illetve saját készítésűekkel is.

## 2.15. Xampp

Xampal tudunk többek között MySql adatbázist létrehozni a gépünkön. Ennek köszönhetően nem kell távoli szervert használni hanem gyorsabban akár offline módban is lehet tesztelni az alkalmazást.

# 3. A FEJLESZTÉS MENETÉNEK LEÍRÁSA

## 3.1. Frontend

A VS Code terminálból kiadott *npx create-react-app my-app --template @chakra-ui/typescript* paranccsal tudunk új React projektet létrehozni, hogy használja a Chakra UI nevű felhasználói felületi könyvtárat TypeScript sablonnal.

A github repositoryt lehúzva láthatjuk a frontend mappa architektúrája a

**-comonenets:** ebben a tárolom a saját, illetve újrahasznosított komponenseket.

**-pages:** itt tárolom a komplett oldalak

**-store:** a hálózati kommunikációjának, és a redux store action-jainak a leírása

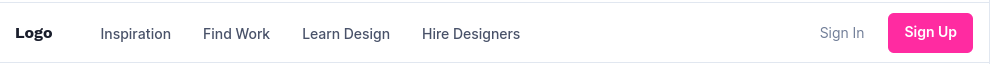
**-theme:** itt tárolom a program komponenseinek dizájn leírását

**-utils:** felhasznált segéd függvények

Ezeken a mappákon végig haladva mutatom be a program működését, felhasznált könyvtárakat, a sarkalatosabb részeit a programnak.

### **3.1.1. Navbar:**

A sablona a chakraUI-t követi és a <https://chakra-templates.dev/navigation/navbar> vettem alapul.



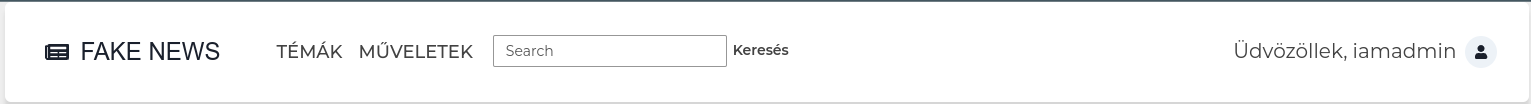
Ezt a kódot módosítottam illetve alakítottam át, hogy a backendről érkező adatokat kezelni tudja

Bejelentkezés után a komponens nézete megváltozik aszerint, hogy milyen

jogosultságai vannak a felhasználónak.

Korai verzióban ezt a böngésző local storageben tároltam és innen olvastam ki az aktuális jogosultságokat. Azonban ezt egy egyszerű változó átírással a böngészőben meg lehetett „hack”-elni a programot.

Ezután váltottam a program global statejében tárolásra. Így bármelyik komponens könnyen elérheti, de illetéktelen nem tud hozzáférni.

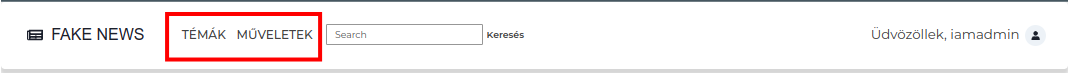


### **3.1.2 Icon**:

Ez egy külön komponens, kattintásra visszaállítja az esetleges keresési feltételeket.

A lekérdezett hírek témáit egy fenntartott -1-re állítja ami a backend oldalon az összes hírt jelenti, továbbá törli a cím alapján való szűrést, és visszanavigál a *HomePage* oldalra .

A két előbbi érték a redux statben van tárolva. Ezeket a redux szabályai szerint csak actionok on keresztül tudjuk módosítani. Ezeket az actionokat mi implementáljuk és a useDispatch() hookon keresztül tudjuk „ellőni” .

A hookok használatára külön szabályok vonatkoznak erre a store résznél részletesebben kitérek.

### **3.1.3 Témák**:

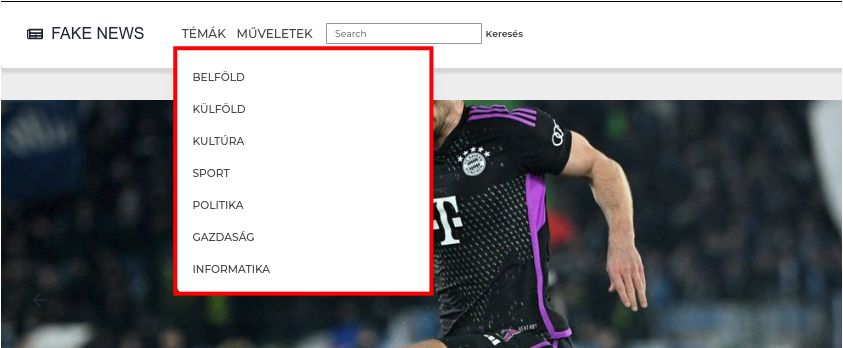
Hírek kategóriáit azaz a backendről lekérdezett TypeDTO-t jeleníti meg

### **3.1.4 Műveletek**:

A navbarban hard kódolva adtam meg a két lehetőséget. Mind két komponens tulajdonképen ugyan az bár a megjelenítendő elemek tulajdonságai mások. Míg a Témák komponens Type típusú elemeket jelenít meg ’id’ és ’title’ attribútumokkal, amelyek megfelelőek voltak a menu elemek value és címke értékeinek addig a Műveletek menu elemei ’title’ és ’path’ attribútumokat tartalmaz. Ezért kiterjesztettem az alap Type típust és hozzáadtam egy opcionális path attribútumot, az új típus neve *NavMenuItem,*  a megjelenítési logikában egy ilyen típusú tömböt, és egy ’lable’ stringet tartalmazó *MenuItem* elemet adok át a komponensnek.



Ez a komponens a ChakraUi ból származó *Popover* amellyel felugró ablakokat tudunk megjeleníteni. Két fő komponense e a *PopoverTrigger,* és a *PopoverContent .*

A *PopoverTrigger* jelenik meg a menu sávban, a beállított *trigger* atributom *hover* értéke miatt amikor fölé kerül a kurzor mutató megjelennek a *PopoverContent* elemek amelyek egy-egy *NavMenuItem* elemet jelent a *children* tömbből

A trigger paraméter *click* értéket is beállíthatunk azonban ekkor gondoskodnunk kell a bezárósáról is

### 3.1.5 Keresési mező

****

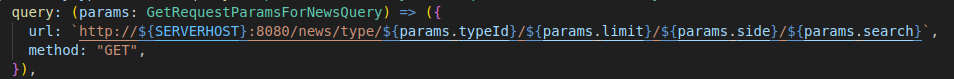
Ezzel az komponenssel tudunk szűrést végezni a hírek között, cím alapján.

A program készítése közben ez a funkcionalitás kissé lemaradt a többitől és csak később tűnt fel, hogy nem is végez valódi szűrést csak az éppen aktuális oldalon megjelenő hírek között. Az elkülönítés és a redux statenek köszönhetően ezt a hibát viszonylag kevés módosítással sikerült kijavítani.

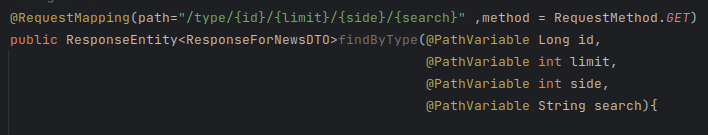
A hírek lekérdezését végző query paraméter listájához hozzáadtam egy string típusú search attribútumot. Mivel korábban már a globális statben volt tárolva az esetleges keresési string így azt a megfelelő helyen letudtam kérdezni. Erről részletesebben a store fejezetben.

Most csak annyit elég tudnunk, hogy egy hook segítségével érjük el azonban fontos, hogy a inicializálási értéke megfelelő legyen ezt minden keresés után alaphelyzetbe állítottam. Jelenlegi verzióban ez az érték undefined. Ezt az értéket veszi a backend üres keresési mezőnek így erre a szóra nem tudunk keresni

A lekérdezési paramétereket az URL be láncolva adtam meg, ebben a sorrendben olvassa ki majd a változókat a backend.



A backend endpointot is kiegészítettem a serach paraméterrel. A RequestMapping annotációval tudjuk megadni a path templatet, továbbá hogy egy GET metódusra figyelünk. *ResponseEntytit* küldünk vissza melynek egy saját *ResponseForNewsDTO* a neve, amely egy logikai értéket és egy hírek listát tartalmaz . Erre azért volt szükség, mert az alkalmazásba implementáltam egy lapozási lehetőséget, oldalanként töltöm le a híreket és jeleznem kellet amikor a keresésnek megfelelő hírek listájának végére értem, ekkor inaktívvá válik a lapozó gomb.

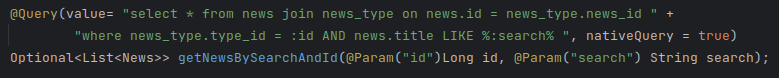
****

A konstruktorban a @PathVariable annotációval sorban megadjuk az URL-ben érkező paraméterek listáját.   
Az idparaméterben adjuk át a hír típusát, a limit-ben az egy oldalon megjeleníteni kívánt híreket a side az aktuális oldalt, a search pedig a keresett karakter sorozat.   
Ez a fajta paraméter átadás ebben az esetben nem a legelőnyösebb. Szerencsésebb lenne magában a requestben küldeni az adatokat, lásd save metódus a NewsContollerben, ahol egy NewsDTO objektumot küldök a szervernek.

Ennek a dolgozatnak a célja minél több módszer bemutatása.

A controlleren keresztül átadjuk a paramétereket a service rétegnek, itt állítom össze a side és limit értékek alapján a vissza küldött hírek tömbjét.

A keresési, illetve szűrési logika figyelembe veszi az éppen aktuálisan kiválasztott témakört és a keresési mezőbe írt karakter sorozatot. Mivel a hírek témakörei egy külön táblában vannak, emiatt több táblás lekérdezést kell használnunk de szerencsére a JPA-ban erre is van lehetőség.

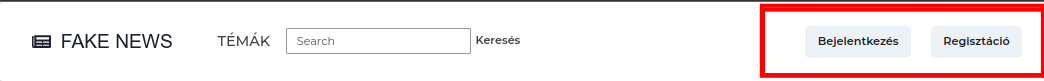
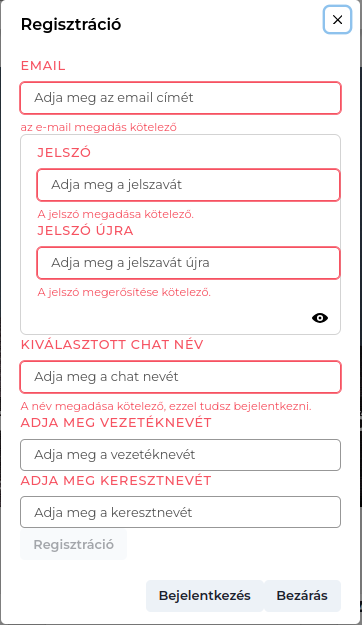
****

A lekérdezésben kiválasztjuk a news-t, amelyet a news\_type-al kötünk össze, azokat a híreket szeretnék megkapni melyek type\_id-a a segédtáblából megegyezik a paraméterben megadott id-val a news- táblából pedig a title field tartalmazza a search karakter sorozatot.

Visszatérési értéke egy news listát tartalmazó optional adatszerkezet, ez azét hasznos mert ha a lekérdezés nem hoz eredményt könnyen tudjuk kezelni.

Másik lehetséges keresés, amikor az összes hír között keresünk témakörtől függetlenül, ilyenkor már nem használjuk a segédtáblát.

### 3.1.6 Regisztráció

A látogatónak, hogy kedvelhessen illetve hozzászólhasson egy hírhez regisztrálnia kell.

A regisztrációs felületet egy ChakraUI ból

származó *Modal* komponens adja. Kifejezetten felugró ablakok megjelenítésére használatos.

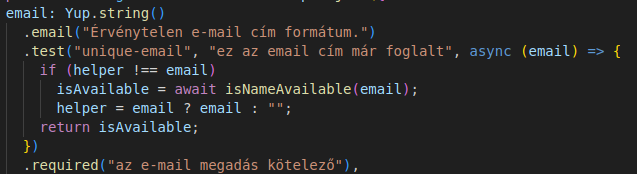
Az *isOpen* paraméterben logikai értéket adhatunk meg nyitva illetve zárva legyen a komponens. Ezt az értéket is a redux globális stateben tárolom.

Az *onClose* paraméterként egy függvényt adhatunk meg amely akkor hívódik amikor a komponenst a felhasználó a bal felső gombra kattintva bezárja vagy kikattint a *Modal*ból.

Többek mellet itt is használtam a *useEffect()* hook *return* függvényétamely a komponens újra rendrelése előtt fut le. Mivel több módon is ellehet hagyni a *Modal-*t, ki kattintás, el navigáció a *Bejelentkezésre*, *Bezárás* gomb, ebben a részben töröltem ki a formhoz felhasznált objektum kulcsait.

A *ModalBody* részében helyeztem el a form tag-et amelyet a *fomik* könyvtárból származó *useFormik* segítségével kezelek.

Ennek segítségével egyszerűen kezelhető a validáció, a submit esemény, és az esetleges hibák.  
A validációt egy úgynevezett validációs séma megadásával kezdtem. A from egyes bemeneti mezőire külön-külön láncolt formában adhatunk meg feltételeket, az egyszerű hossz értéktől egy api kérés *respons*ában érkező érték vizsgálatáig.



Az egyes függvényekben megadhatunk üzeneteket jeleníthetőek meg a frontenden. Az üzenet attól függ, hogy a láncolatban éppen hol akadt el a kiértékelés.

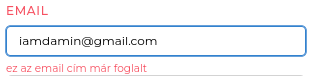
A *useFormban* megadhatjuk a validációs sémát, azonban még a benne megadott függvényeket a fieldekhez kell kötnünk így a korábban megírt függvény többször felhasználható akár más projektekben is.

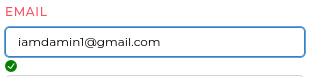
Ehhez két dolog kell a *FormControl* nyitó tagében megadjuk, mint *isInvalid* properti.

A *FormControl*on belüli *FormErrorMessage*ben megjeleníthetjük.



Az email cím vizsgálatára egy előre definiált függvényt használunk, ami a szintaktikai helyességét vizsgálja, ezután az ebben a fileban definiált featch api kérés vizsgálja dinamikusan szabad-e a regisztáció az email címmel.

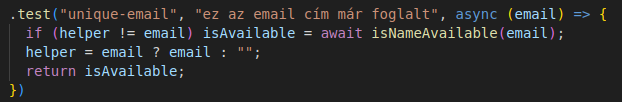




A input mező *onChange* függvényébe állítjuk be *form-*nak átadott objektum field értékét, ami ezáltal minden egyes karakter leütésnél frissül.

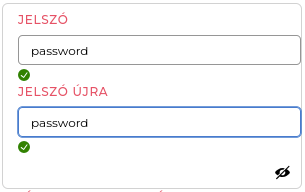


A jelenlegi verzióban a form a *useFormik*ban megadott *initialValues* értéket frissíti, minden egyes input mezőbe leütött karakter hozzá tesz illetve töröl valamilyen kulcs értékből. Ezért renderelődik mindig újra a form, ennek hatására a validációban használt api kérés az email cím field közvetett frissülése miatt mindig újra küldené a requestet. Emiatt ezen a ponton egy kisebb logikát alkalmaztam az email címre



Csak akkor indul kérés ha a változik az *email* cím ezt a változást egy *helper* segédváltozóval figyelem.

A jelszó minimális hosszát 6 karakterre állítottam a megerősítő mezőnek pedig azt a kikötést, hogy egyezzen meg az előzővel. Kiegészítettem egy gombbal ezt a területet ami láttatja, illetve elrejti a begépelt karakter sorozatokat. Ez a gomb egy lokális logikai változó értékét változtatja mindig az ellenkezőjére. Ennek a változónak a függvényében lesz az input mező típusa *password* vagy *text*



Ha miden *field* megfelelően van kitöltve aktivá válik a *Regisztráció*s gomb. Ezt az *errors* tömb hosszával ellenőrizzük.

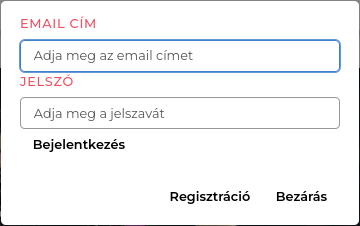
A *useFrom*-al handleSubmit függvényével kezelhetjük a *form* submit eseményét is. Ezt az eseményt én a komponensen belül kezelem, egy ”üres” user objektum kulcsait töltöm fel a *form* fieldjeinek értékével.

A backend-en az adatbázisba regisztrálás közben a megadott felhasználói email címére küld egy üdvözlő üzenetet ez későbbiekben email validációra is alkalmas lehet.

Megjegyzés. A fejlesztés során néhány felhasználót közvetlenül adatbázisban adtam meg. Azonban szerver irányból a JPA valamilyen markerrel megjegyzi az általa utoljára hozzáadott elem indexét, így amikor már alkalmazásból akartam felhasználót regisztrálni, addig hibát dobott amíg elnem érte ez az index az adatbázisban aktuális elemszámot.

### 3.1.7 Bejelentkezés

Szerver oldalon egy felhasználó kezelése közben megkülönböztetünk **autorizációt** és **autentikációt.** Az autoriuáció a felhasználó azonosítása, felhasználói név és jelszó alapján.

****

A bejelentkezési modalban megadott adatok a backend */authentication* endpointjára érkezik.

A backenden a s*pring-boot-starter-security* könyvtárat használom a szerver elérési jogosultságainak beállításához.

Minden szerver felé irányuló kérés előtt lefut egy authorizációs filter, amely megvizsgálja a headerben van-e autorizációs szegmens, ha nincs akkor csak azok az endpointok felé engedi tovább a kérést amelyek a szerver *SecurityConfiguration.java*ezt beállítottuk. Ha van, ellenőrzi az adatbázisból a bejelentkezési adatok helyességét, ezután látja el a hozzá tartozó jogosultságokkal, ezt már autentikációs résznek nevezzük, és engedi tovább a védett endpointokhoz a jogosultság alapján.

Fontos, hogy szerver oldalon az endpointok elérésének beállításakor sorrendet kell tartanunk. Mivel láncolva adjuk meg az endpointok hoz tartozó jogosultságokat, a springsecurity az első illeszkedő template-re abba hagyja a további vizsgálatot és azt használja tovább.



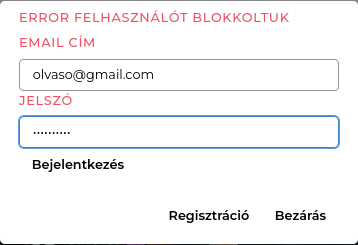
A fenti példa csak a szerver oldalon kiosztott ADMIN joggal rendelkező requesteknek engedi elérni a *news/delete* endpointot, a *news-t* mindenkinek ehhez nem szükséges autorizálni a kérést, bárki láthatja a híreket.

A /\*\*-al jelöljük hogy minden további endpointra vonatkozzon az utasítás.

Fordított sorrend esetén először a /news/\*\* endpoint érékelődne ki amelyre illeszkedik a /news/delete/ így azt bárki elérhetné. Erről bővebben a backend résznél írok.

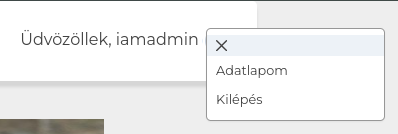
Az esetleges authorizációs hibát a szerveren állítjuk össze és küldjük visszaküldi a frontend felé, amit az inline jelenít meg.

Itt vizsgáljuk az adatokat illetve, hogy blokkolt-e a felhasználó.



Bejelentkezés után a *user* bekerül az alkalmazás *global state*-jebe a továbbiakban innen használjuk fel, nem indítunk fölösleges lekérdezéseket.

Munkám sokrán a Google Chrome böngészőhöz telepített ReduxDevTools nevű kiegészítőt használtam amely nagy segítség a *global state* változásaink követéséhez.

A navbáron megjelennek a felhasználó chatneve és a jogosultságainak megfelelő menügombok minden felhasználó megtudja nézni és szerkeszteni a saját adatlapját

### 3.1.8 Felhasználók listája

Az adatbázisban jelenleg csak néhány felhasznáót regisztráltam szemléltetés céljából, ezen felhasználók mindegyikét lekérdezi a program így egy esetleges több 100 felhasználós adatbázis esetén ezt kezelni kell, vagyis megfelelően át kell alakítani az ezért felelős queryt a kliensoldalon és a szerveren is

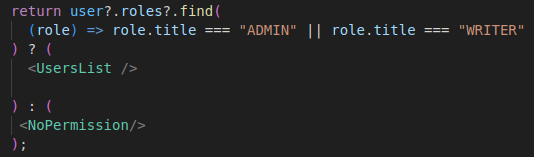
A felhasználók listájának a */users* végponton tekinthetjük meg. Az alkalmazásban a *react-router-dom* könyvtárat használom az oldalak közötti navigációra.



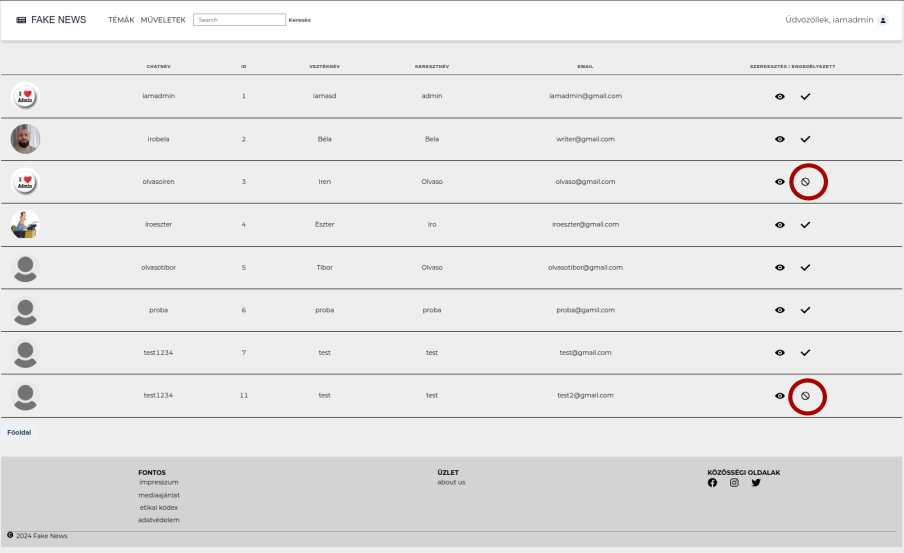
Szerver oldalról a /users API endpoint mindenkinek nyitott, hogy a regisztráció során letudjunk kérdezni adatokat. Ezért a listát kliensoldalról védjük, ami azt jelenti hogy egy provideren keresztül érjük el, ami vizsgálja hogy van-e autentikált felhasználónk jogosultsága látni a listát.

Jogosultság nélkül egy *Error* komponens jelenik meg, ami tájékoztatjuk a felhasználót

***KIKELL cserélni!!***



Listában láthatjuk a felhasználók adatait illetve ikonal jelezzük melyek vannak blokkolva



### 3.1.9 Új cikk hozzáadása

A **MŰVELETEK**menüben találjuk az Új cikk hozáadása menüpontot erre kattintva dispach-elünk egy paraméterek nélküli *showEditor* actiont amely létrehoz és beállít egy üres hírt a slice state-jében. Ezután a /edit path-ra navigálunk a useNavigate() hookon keresztül.

A /edit pathra beállítottam a NewsEditorProvider komponenst, ez a komponenes figyeli autentikált-e a felhasználó és van-e joga elérni a formot. Továbbá a stat-eből lekérdezi a hírt, ebben az esetben a még üres hírt, és tovább adja a *NewsEditor* komponensnek

Ennél a komponensnél is a useFormikot használjuk a form kezelésére. Az alap input fieldek mellet egy kategória választó komponenst használok.

Egy cikk több kategóriába is besorolható. Ez egy legördülő checkbox menu és a mellete megjelenő ikonokból áll. Az ikonra kattintva is kitudunk venni elemet a felsorolásból



A lehetséges kategóriákat adatbázisból töltöm fel, pontosabban a korábban lekérdezett *types* tömbbel amit a globális stateben tárolok így elkerüljük a fölösleges lekérdezést.

A valideates- fileban minimum egy kategóri kiválasztását adjuk meg feltételkén.



Checkbox-al tudjuk egy hír prioritását beállítani, amely hatására a a Carousel sávon is megfog jelenni.

Ha minden field megfelelően van kitöltve aktívvá váli a küldés gomb a Regisztrációs metódus mintájára.

### **3.1.10** **Cikk szerkesztése**

Az egyes hírek bal felső sarkában megjelenő menüből választhatjuk ki a *szerkesztés* lehetőséget

A szerkesztés menüpontra kattintva az aktuális cikk bekerül a global statebe, ezután elnavigálunk a */edit* path-ra amelyet az *EditorProvider* komponenshez kötöttünk.

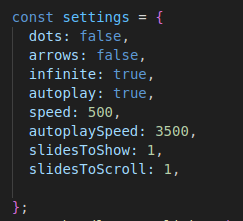
Ebben a providerben kérjük le a hírt a global state-ből és adom tovább *NewsEditor* komponenesnek még ugyan itt állítom össze az oldal-t, navbar footer -komponenseket. Erre azért van szükség, hogy ha szerkesztjük a formot, ne frissüljün az egész ”oldal” a state- és propertik miatt.

A *NewsEditoron* belül a news-t adom meg a form *initialValues* értéke ként, emiatt az értékátadás miatt a news változása után nem frissül a form az új értékekkel, emiatt a useFormik setValues metódusát egy useEffect hookba vezetem ki amely figyeli a news változását és frissíti a form fieldjeit.

### **3.1.11. Carousel komponenes**

A Carousel sávon az aktuálisan betöltött oldalon lévő vező hírek jelennek meg.

Szokásos módon egy provider kezeli az adat átvételt a state-ből és admom át a Carousel komponensnek. Ennek célja hogy ”tiszán” tartsuk a komponenseket.

Ennek hiányában a Carousel-t nem tudnánk különböző programokban felhasználni, mivel közvetlenül kötnénk az alkalmazás state-jéhez.

A *react-slick* könyvtárból importált *Slider* komponenst használtam, ezzel a komponenssel lehetőségünk van további komponensek egy sávban való megjelenítésére. A *Slider-nek* egy objektumban adom át a beállításait így áttekinthetőbb a kód, ez komponens független módszer bármelyik más tag-re alkalmazhatjuk.

A alapértelmezett esetben navigációs gombok jelennek meg a komponensen kívül amelyekkel válthatunk a megjelenített hírek között.

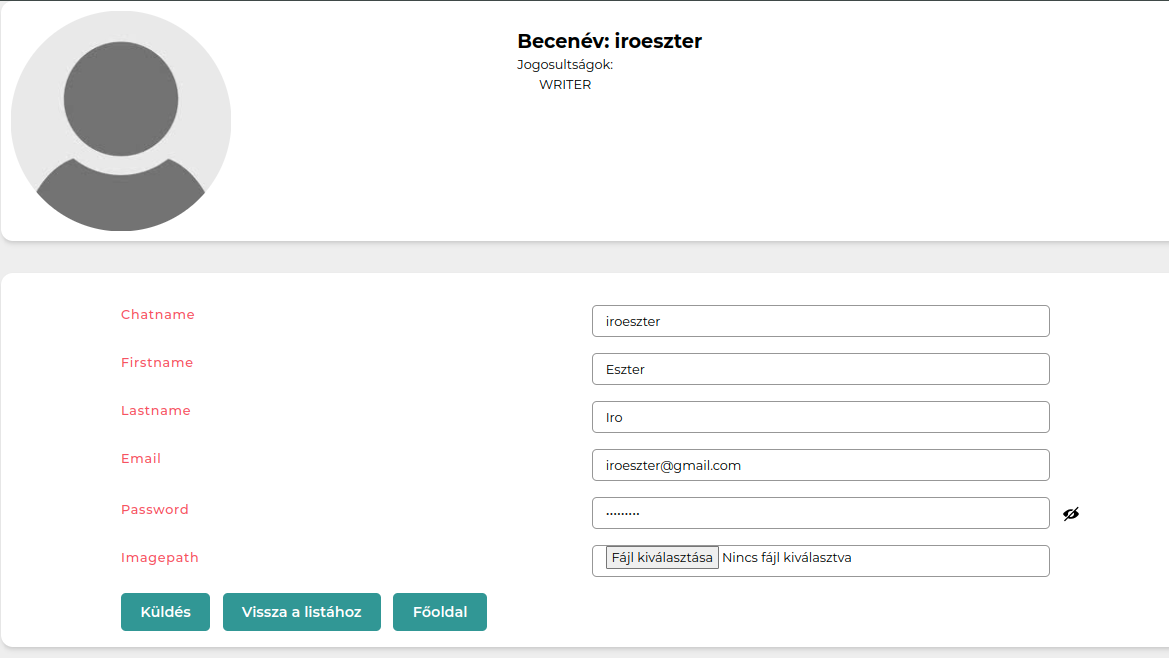
Azt szerettem volna ha a komponensen, illetve azt átfedve jelennek meg ezek a gombok, ehhez le kellet tiltanom az alapértelmezett gombokat. Továbbá egy referenciát kérni a Slider komponensre amelyen keresztül elérem a *slickPrev()*

és a *slickNext()*  metódusokat, amiket fel tudok használni a saját gombok onClick eseményében, így irányíthatom a Slider komponenst.

****

### 3.1.12 Single user page

Itt mindenkinek lehetősége van a saját adatainak a megváltoztatására, továbbá az alapértelmezett profil képet is le tudja cserélni.



Az alap értelmezett kép a szerveren van tárolva és minden regisztráció automatikusan megkapja.



A profilképet a kliensgépről tudjuk kiválasztani.

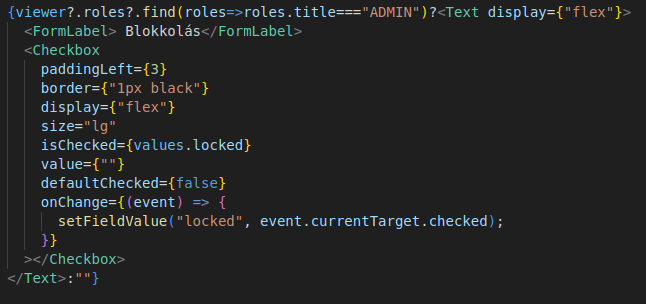
A további profil képek is a szerverre töltődnek fel úgy, hogy a korábbit törlik onnan.

Ez egy react – javascript független Imput komponens amit fájl típussal látunk el így lehetőségünk van böngészni a gazdagép könyvtár szerkezetében.

Ebben a részletben, a kiválasztott képet dinamikusan megjelenítjük az oldalon, majd a metadatokat leválasztva a base64 kódolt képet készítjük elő a küldésre.

Akik WRITER hatáskörrel rendelkeznek megnézhetik mások profilját de nem módosíthatnak abban erre csak az ADMINok képsek, mint arra is hogy töröljenek vagy módosítsanak egy hírt.

ADMIN hatáskörrel a profil oldalon megjelenik egy extra checkbox amely-el blokkolhatja az aktuális felhasználót. A többi ilyen komponenshez hasonlóan, ezt is oly módon rejtem el a UI felületről, hogy végig iterálok az éppen bejelentkezett felhasználó *roles* tulajdonságán és ha egyezést találok ADMIN karakter sorozattal visszaadjuk a komponenes

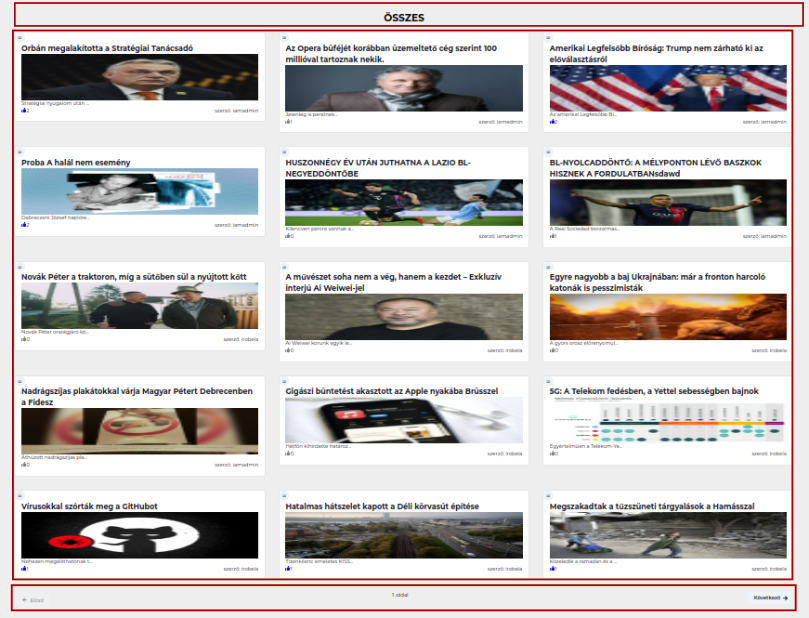
 A küldés gombra kattintva indítjuk el a requestet, ezzel együtt vissza navigálunk a főoldalra.

### 3.1.13 News

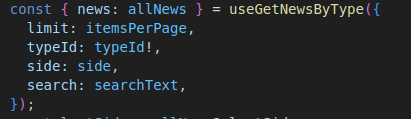
#### **3.1.13.1 NewsListProvider**

Három komponenes megjelenítésért felelős. Az első komponenes jelzi a redux stateben tárolt typeId alapján milyen kategóriájú híreket jelenítek meg.

Második a NewsItemList ez a lekérdezett híreket jeleníti meg, harmadik komponenes a lapozási logikáért felelős.



Továbbá itt kérdezem le a keresési paramétereknek megfelelő hírek adott hosszúságú tömbjét.

****

Jelenlegi verzióban a lekérdezett tömb hossza állandó. de egy ide elhelyezett lenyíló listával, melyből a felhasználó kiválaszthatja mennyi hírt kíván megjeleníteni, még interaktívabbá tehető az alkalmazás. Bármely paraméter változik azt a react észleli és újra futtatja a lekérdezést.

#### **3.1.13.2 NewsList**

Pure komponenes, properi ként egy News elemekt tartalmaző tömböt vár el amit megjelenít NewsItem-ek formájában.

#### **3.1.13.3 NewsListItem**

Az egyes hírek kártyák formájában jelennek meg a fő oldalon. Ezen a kártyán jelenítem meg a hírhez tartozó képet, címet , alcímet, szerzőt, és az aktuális kedvelések számát.

A komponens vázát a chakraUI ból származó Card tag adja. A komponensemnek propertiben adom át a news objektumot, itt nem történik hálózati kommunikáció.

#### **3.1.13.4 NewsDescriptionProvider**

A korábbi verziókban túlságosan kiakartam használni a global stat-et így ennél a komponenesnél a *NewsItem-re* kattintva beállítot hírt kértem le a stateből.

Viszont így ha a felhasználó fríssített valami okból a böngészőjét „elveszett” a hír.

Először az *App.tsx* fileban módosítottam a komponenshez kötött path

templat-et

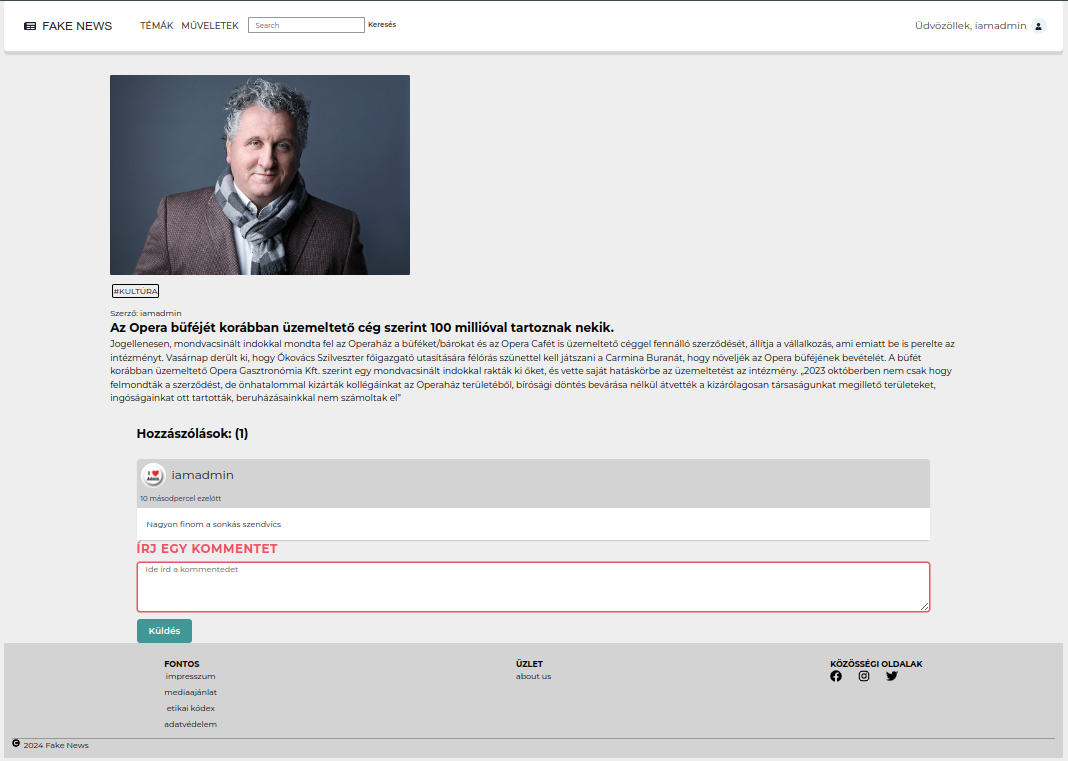
A newsItem-re kattintva már nem a hírt töltöm be a state-be hanem a path-hoz fűzöm a hír id-ját, és erre a dinamikus path-ra navigálok el.

A *react-router-dom ból* BrowserRouter kontextus felimseri a path és a hozzá kötött komponenst betölti a megfelelő helyre.

Sajnos ez a kontextus minden olyan komponenest is újra renerel amely tartalmazz a navigációhoz szükséges useNavigate() hookot is.

Ilyen komponensek a Navbar, Footer, mivel ezek dinamikusak nem helyezhetem a BrowserRouter kontextuson kívülre.

Egyik megoldás lehetne, ha elhagynám a routingot és a regisztációs, illetve bejelentkezési Modal mintájára, az egyes „oldalak” a központi statben tárolt flag-ek függvényében jeleníteném meg. Röviden ki-be kapcsolgatnám a láthatóságukat. Ezzel azonban azt a funkciót veszíteném el hogy az egyes hírekhez tartozó oldal pathokat a felhasználó másolni, vagy menteni, megosztani tudja, mivel ebbel az esetben nem a path-on adnám át az adatot hanem a global statben tárolnám, amely minden frissítésnél vagy ha elhagyja a felhasználó az oldalt eltűnik.

Ezen provider először a path-ból a useParams hookon keresztül kiolvassa a hír id-ját. Ha paraméterek nélkül hívjuk meg a useParams hookot akkor egy kulcs-érték párokat tartalmazó tömböt kapunk vissza a Routingnál megadott template változóneveivel Jelen esetben az *id* változó tartalmazza a hír id-ját a fenti deklarációval konkrétan meg tudom nevezni melyik változóra van szükségem a path-ból.

Ezután az id alapján lekérdezi a szervertől a hírt.

Itt állítom össze az egyes hírek megjelenítéséhez használt komponeneseket.

Három komponenest használok *NewsDecreption* , egy Lista formájában az eddigi kommenteken végig iterálva a *MyComment* kopmonenseket formájában jelenítem meg azokat, végül a *CommentForm* ez a komment beviteléért felelős.

Mivel a szerver lekérdezés asyncron művelet némi késésre kell számítanunk amíg visszakapjuk a respons-t ezért egy useEffekt hookal figyeljük a választ és ez alapján töltöm fel a provider komponens statben tárolt Commentek tömböt

Továbbá ez a komponens végzi még a Komment elküldését.

#### **3.1.13.5 NewsDescription**

A komponens propertiként kapja meg a news objektumot. Megjeleníti a hírhez tartozó képet, alatta hogy mely kategóriákba van sorolva a hír, ezek kattintható komponensek amelyek hatására az főoldalra navigálhat ahol a kategóriának megfelelő híreket fogja látni.

Továbbá megjelenik a szerző és maga a hír is.

### 3.1.14 Comment

#### **3.1.14.1 MyComment**

Ez a komponens jelenití meg a hírhez korábban írt kommenteket. Egyetlen properije van egy komment objektum.

Vázát a ChakraUI ból származó Card komponens adja. Megjelenítem benne a komment írójának profilképét, nevét, magát a komment szövegét és hogy mennyi idővel ezelőtt írta a hozzászólását.  
 Ez utóbbi funkcióra készítettem egy saját komponenst amely a porertiben kapott dátum alapján vissza adja az azóta eltelt másodpercek, percek, órák, napok, hónapok

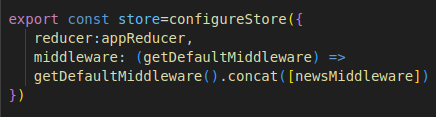
számát, aktuálisan a legnagyobb mérékegységben.

#### **3.1.14.2 CommentForm**

Ezt a beviteli mező külön komponensbe vezettem ki. Korábbi verzióban a meglévő komment megjelenítésével közös komponensben volt ez is. Azonban, mint korábban említem a reakt minden state vagy properti változás hatására rendereli a komponenst így egy beviteli mezőben leütött karakter miatt akár több tíz MyCommen kopmonenes is renderelődni e kellet.

### **3.1.15 Store directory**

#### 3.1.15.1 Store.ts

Ebben fileban koniguráljuk a redux-toolkit állapot tároló komponensét a *storte-t.* A konfiguráláshoz többek között meg kell adnunk egy **reducert.**. A reducer egy fügvény amely kezele alkalmazás állapot változását oly módon, hogy az aktuális állapot és egy action hatására új állapotot ad vissza.   
Ezen alkalmazásomnak az összetettsége miatt a storet több szeletre (slice-re) osztottam és ezek mindegyikéhez külön reducer tartozik, ennek kezelésére használom a combineReducer függvényt amelyel összevonhatom a több reducert egyetlen objektumba és a store definiálásnál ezt adom át.

A egyes sliceok ezeken a reducereken keresztül kapcsolódnak a storba.

A middleware-k olyan funkciók, amelyek megváltoztatják az akciók működését vagy azok hatásait, mielőtt azok eljutnának a reducerekhez.

*GetDefaultMiddleware* :Ezek az alapértelmezett middleware-k az store konfigurálásának részei, és segítenek a szokásos Redux működés fenntartásában, például az akciók kezelésében, az állapot frissítésében stb.

*newsMiddleware:* *Ez az* objektum *a news-api-ts* fileban generálódik automatikusan *a createApi-t* használatakoregyAPIkonfigurálására*, az* automatikusangenerálegymiddleware-t, amely kezelni fogja az API hívásokat. Ez a middleware többek között a következőket teszi:

-Kezeli az API hívások indítását.

-Frissíti az állapotot a különböző hívásállapotok (pl. isLoading, isFetching, error, stb.) alapján.

-Tárolja a válaszadatokat a Redux állapotban.

-Kezeli a cache-elt adatokat és az újrakéréseket.

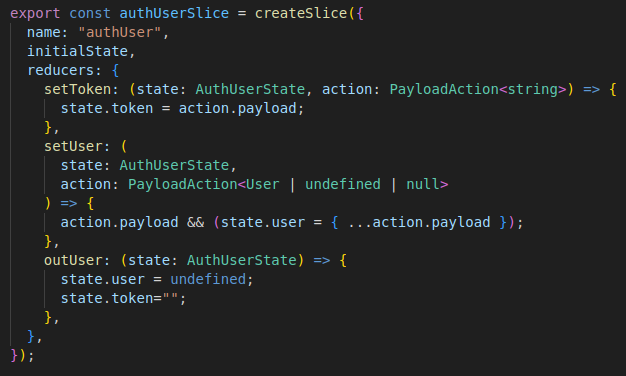
-Kezeli a cache invalidálását és a tag-rendszerű frissítéseket.

#### **3.1.15.2 Slices**

A sliceok ahogy a neve is mutatja a redux store egy-egy szeletére utal. Ennek segítségével tudunk több külön álló tárolót létrehozni. Ezek a szeletek saját state-el rendelkeznek amelyek immutable objektek vagyis csak speciális függvények más néven reducereken keresztül lehet megváltoztatni. A reducer egy úgy nevezett tiszta függvény, amelyek a definiált actionok hatására a state alapján hoznak létre egy új statet amelyet visszaadnak.

Az alábbi példában a bejelentkezett felhasználó tárolására hoztam létre egy slicet.

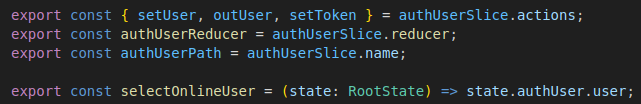
Az actionok beállítják a felhasználóhoz tartozó tokent amelyet a szervertől kap, magát a felhasználót, valamint kijelentkezteti.



A slice state-jét a useSelector hookon keresztül érjük el ehhez exportálnunk kell azokat, *selectOnlineUser*.

Továbbá exortáljuk az actionokat, ezeket a reducer neve alapján ismeri fel a toolkit.

A felhasználó függvényben ezekre a nevekre hívatkozva a useDispach hookon keresztül „lőjük” el az actionokat.

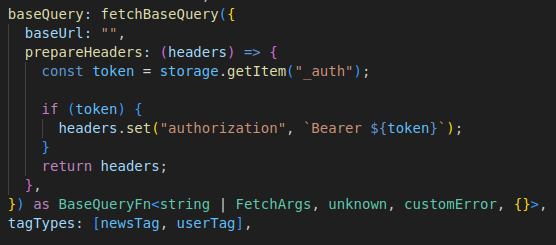


Az *authUserReducer* és *authUserPath-t* a **store.ts** fileban használjuk fel amelyben magát a redux store-t állítom be. Itt egy combineReducer függvénnyel főzöm össze az többi slice-t a reducereken keresztül.

#### **3.1.15.3 News-api.ts**

Az alkalmazás egyik legfontosabb része, itt konfigurálom a hálózati kéréseket, definiálom az alkalmazás API endpointjait.

A konfiguráció lényegesebb részei:



Itt hálózati kérések alapját állítombe,

*-baseUrl:* a szerveroladali kiinduló path-t adhatjuk meg

*-prepareHeaders:* a szerveren történő autentikációhoz szükséges hogy a request header részében azonosítsuk magunkat, ezt a korábban az authorizáció során a szervertől kapott tokennel tehetjük meg, ha van ilyen token a böngésző localstorage-ben akkor ezt elküldjük ha nincs, akkor ez kimarad a headerből és a szerver ennek megfelelően kezeli majd a kérésünket.

A baseQuery-t BaseQueryFn ként inicializáltam, erre a saját *customError* paraméter miatt volt szükség. Ezt az adatszerkezetet én adtam meg a szerverről érkező hibák egyszerűbb kezelésére.

A endpoint definiálásakor megkülönböztetünk *query*ket és *mutation*okat

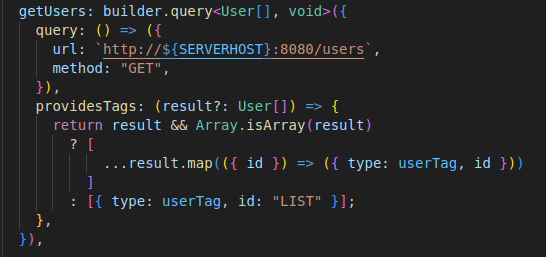
A query egy olyan művelet, amely adatokat kér le a szerverről. Ez tipikusan egy GET kérés, és arra szolgál, hogy a kliens friss adatokat kapjon a szervertől anélkül, hogy bármilyen változást hajtana végre az adatokon, az adatbázisban.

A mutation egy olyan művelet, amely adatokat módosít a szerveren keresztül az adatbázisban. Ez lehet például POST, PUT, PATCH vagy DELETE kérés, amely új adatokat hoz létre, módosít meglévő adatokat, vagy töröl adatokat az adatbázisban.

Ezeket egy-egy példán keresztül mutatom be ezeket.

#### 3.1.15.4 Query

*getUsers*:  
<> karakterek között első paraméterként megadjuk milyen típusú adatot várunk visszatérési érték ként, második paraméterben milyen adatot küldünk.

*url*: a szerver endpoint címe  
SERVERHOST egy .env filban előre definiált környezeti változó

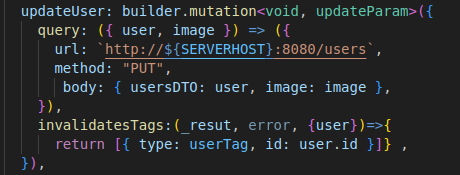
ezt az alkalmazás bármely komponenséből elérhetjük.

*method:* a request fajtája

*providesTags:* megadjuk a böngészőnek hogy a válaszban kapott felhasználókat, illetve bármilyen adatokat, ha sikeres a lekérdezés, milyen címkékkel lássa el, hogy később ezekre a mutaion-ban hívatkozhassunk.

Ezek az adatok a böngésző cach memóriájába kerülnek.

#### 3.1.15.5 Mutation

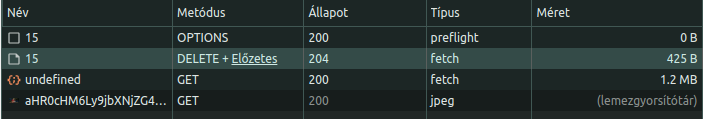
a queryhez hasonlóan megadjuk milyen adatot küldünk és milyet várunk.

A request *body*ban JSON objektum ként küldöm el az adatokat

*invalidatesTags*:  
ebben a paraméterben adjuk meg hogy mely adatokat kell frissítenie, illetve újra lekérdeznie a böngészőnek. Minden olyan query újra lefut amelyhez a provideTags ben megadott tömb tartalmazz az invalidatesTags tömb valamelyik tag-ét. Jelen esetben ez típus és id alapján kulcs érték párokat képezünk, a típusok segítségével különböztetem meg az híreket illetve a felszanálókat, az id pedig a mutation finom hangolására szolgál.   
A { type: userTag, id: 'LIST' }-el, type alatt lévő összes adatra tudunk hivatkozni, ebben az esetben az össze cachben userTag -el ellátot felhasználóra

A híreket admin módban a főoldalról tudjuk törölni egy felugró ablak segítségével. Az alábbi kép a hálózati kommunikációt mutatja a törlés hatására.

Először egy OPTIONS metódusu query kerül kiküldésre ennek biztonsági okai vannak megvizsgálja a szerver fogadja-e a DELETE metódust. Következik a DELETE mutation amely a Tag-elések miatt indikál-ja az adott oldalon lévő összes hír újra letöltését. Negyedik sorban a következő oldalról „becsúszó” hír borítóképe kerül lekérdezésre ez a mi rendszerünktől független.

**

Ebben az esetben indokolt az azonnali cach frissítés, azonban ha éppen egy hírt updatel-ünk, a hírek editor felületén, ez a frissítés nem fut le azonnal, csak amikor visszanavigálunk a főoldalra szüksége van a cach-re az adatok betöltésére.

### 3.1.16 Hookok

A React számos előre definiált hookot kínál nekünk a leggyakrabban használtakat már említettem, useState, useEffect, useCallback. useParams.

A hookok használatát szabályok kötik csak

-funkcionális komponensekben, vagy

-saját hookban használhatóak,

- csak a fügvény legfelső szintjén hasunálhatóak ami azt jelenti, hogy nem használhatóak elágazásokban, ciklusokban

- a saját hookokat mindig a „use” előtaggal kezdjük.

Ezen szabályok betartására a fordító is figyel és fordítási hibát ad

A saját hookokal a kódunk újrafelhasználható, áttekinthető lesz elkerüljük a duplikációt.

Az alkalmazásomban legtöbbször a szerverről érkező adatok feldolgozására készítettem saját hookokat.

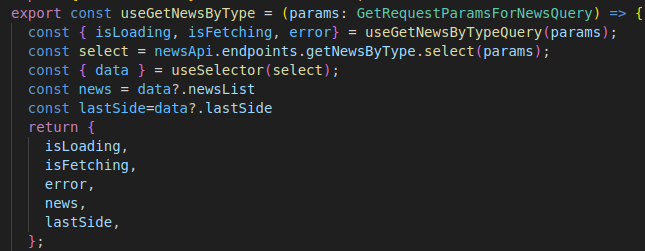
Az alábbi kódrészletben a paraméterekben (oldalszám, típus, keresett szöveg, limit) megadott hírek listáját kérem el a szervertől három lépésben.

Először elindítom lekérdezését de ennek csupán három tulajdonságát veszem át közvetlenül a válaszból.

Második lépésben magát az adatok kérem el a storból a useSelect hook-on keresztül.

Az alkalmazás endpoint-jait a Redux-Toolkit- kezeli ami azt jelenti, hogy a szerverről érkező *data* csomagot a Redux-storba menti el, azért hogy a Redux megfelelően tudja kezelni az alkalmazás állapotát, például ha a *param*éter bármelyik atributuma megváltozna a Redux-Toolkit újra elvégzi a lekérdezést.

Lényegében az tookitre bizzuk az adatok frissen tartását ezzel a módszerrel.



Szerver oldalon olyan adatszerkezetet küldök vissza amely egy tömböt és egy logikai értéket tartalmaz. Ezt még itt ketté bontom hogy a felhasználtó függvényben egyszerűbb legyen használni

Összefoglalva három eljárás eredményét tudjuk tovább adni ennek a hook-nak a segítségével.

### **3.1.17 Utils directory**

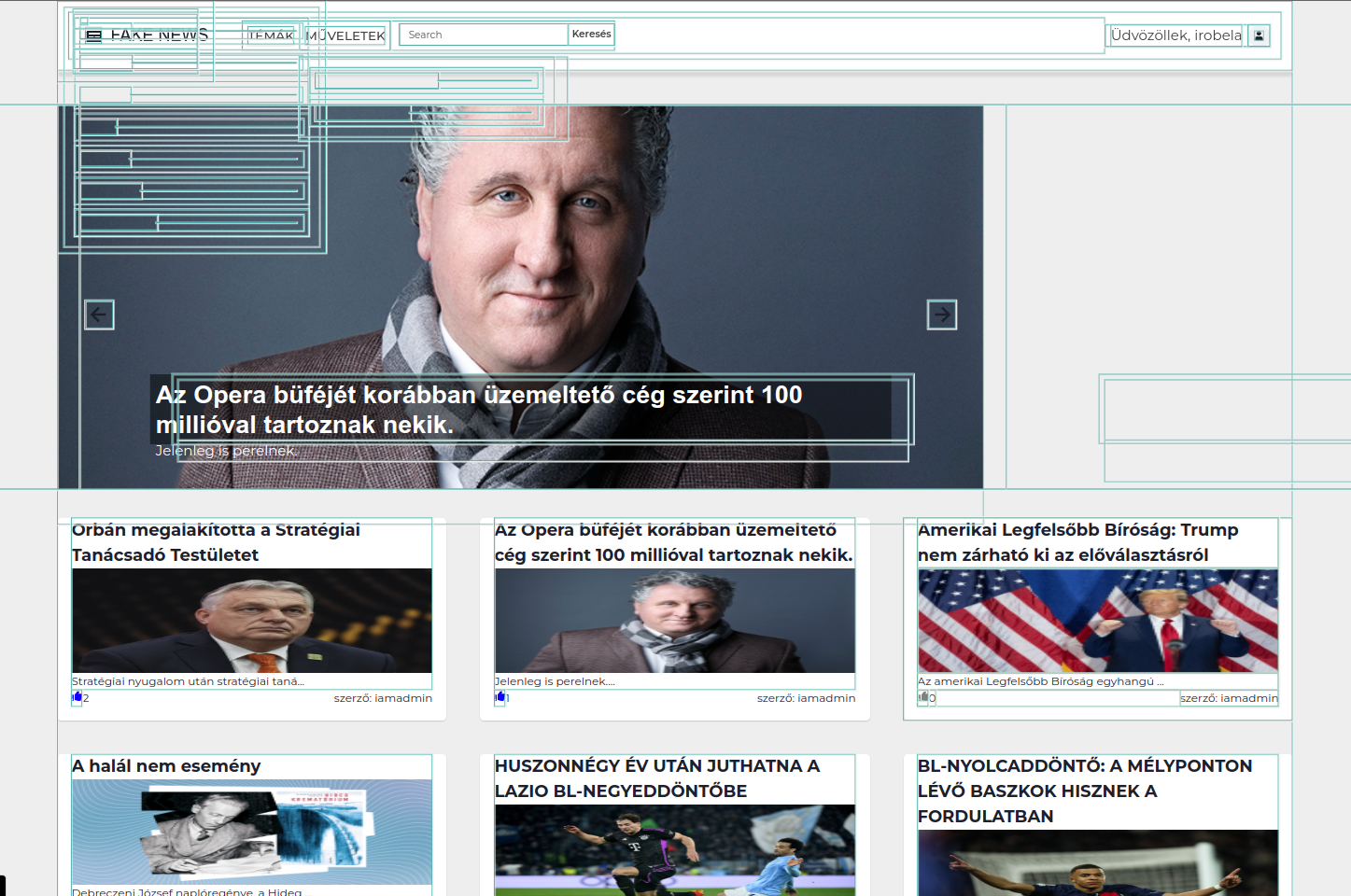
Ebben a mappában tárolom a segéd függvényeket, mint például a *newsFactory*

Melynek feladata a *RawNews* típusu objektumok *News* típusúvá alakítása a két adatszerkezet között az a különbség hogy a *RawNews* szövegként tárolja a hírek *releasedate* attribútumát, emiatt az alkalmazásban bonyolult lenne ennek használata így *News*-á alakítom, ami *Date*-típusban tárolja.  
A vissza alakításért a *serializeNews* függvény felelős.

### **3.1.18 Componensek renderelésére, alias nevek használata**

A react alapelvei közzé tartozik hogy csak a legszükségesebb újrarajzolást (renderelést) végezze az oldalon. A reakt automatikusan végzi a komponens újra renderelését amikor a komponens props vagy state változik.

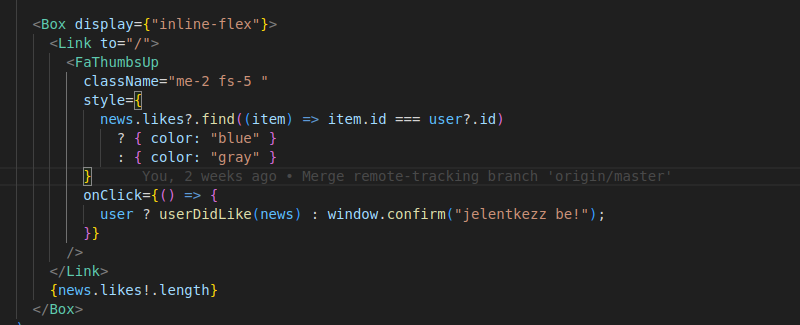
Az alábbi kép (*1.ábra*) egy refaktorálás előtti renderelés látható, amit a „*Like*” gombra kattintás idézett elő.



Egy a böngészőbe telepített React Developer Tools nevű bővítménnyel tudjuk bekapcsolni a Highlight updates-et ami jelzi az oldalon renderelt komponeneseket.

Ebben az esetben két hiba is van a kódban. A *LikeButton* komponenst szerettem volna kattinthatóvá tenni ezért a *Link* taget használtam *react-rooter-dom* könyvtárból amelynek megfelelő a komponensdizája.

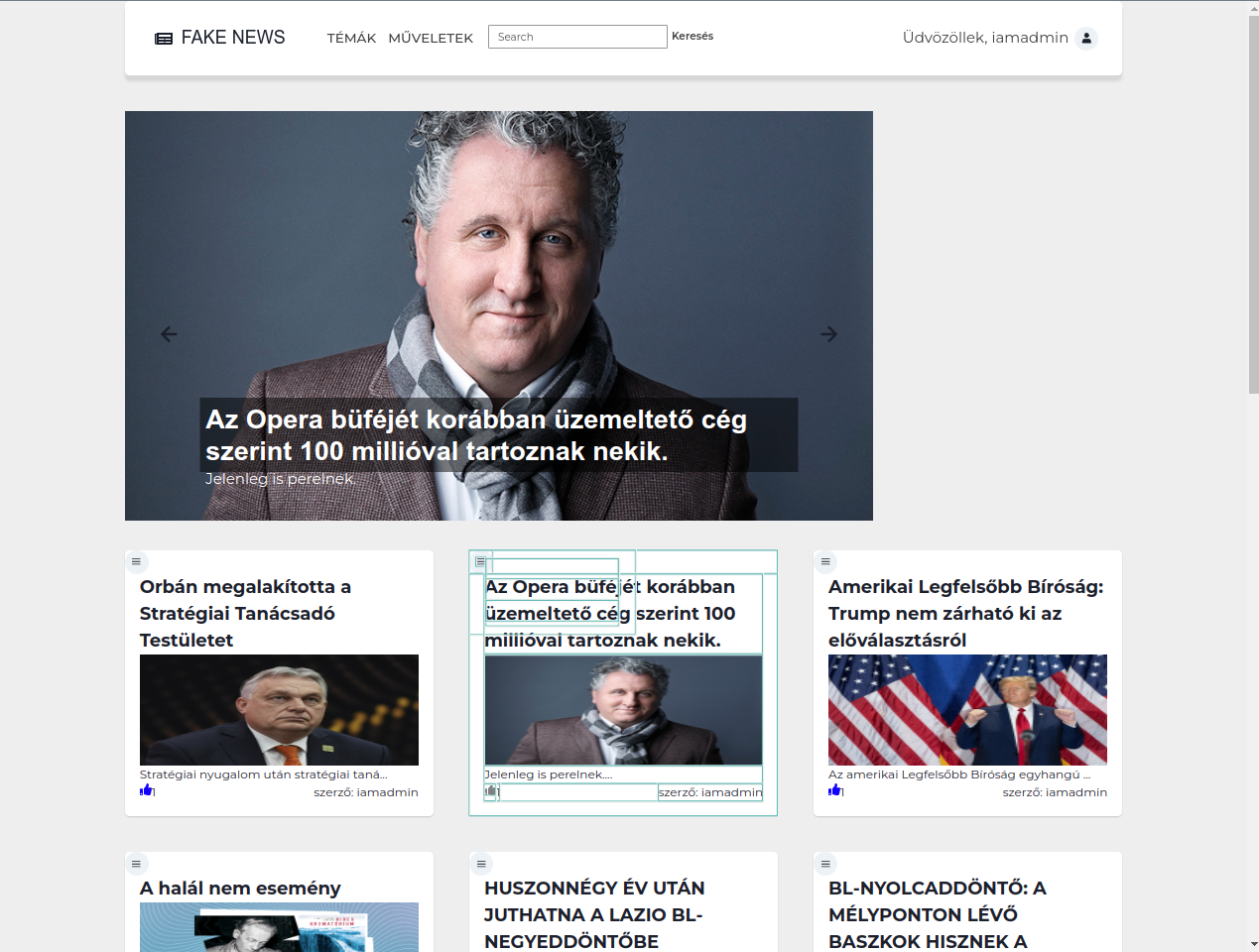
Azonban ennek kötelezően tartalmazni kell egy cél *to=””* propertit, ami alapesetben vissza navigál a gyökér oldalra.



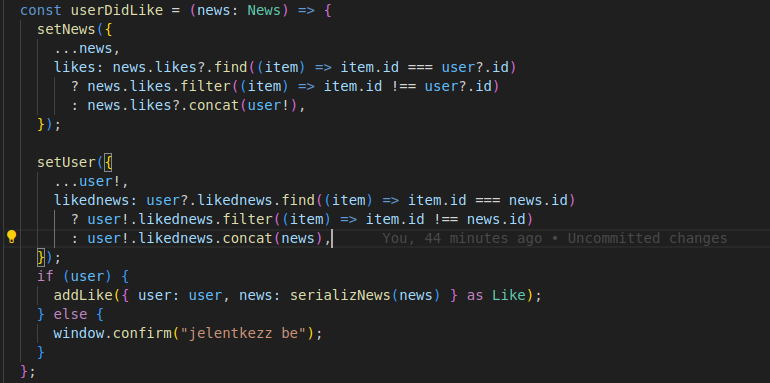
A *@chakra-ui/react* könyvtár tartalmaz *Link tag-*et viszont ebben a szakaszban még a *NewsListItem* komponensben volt a *LikeButton* komponens és szükség volt mind a két típusú *Link* tag-re.  
Ilyen esetben lehetőség van **alias nevet** megadni a tag-nek és azzal hivatkozni rá.



Így elkerüljük az indokolatlan navigációt és renderelést



A másik hiba, hogy közvetlenül a *NewsListItem* nevű komponensen belül valósítottam meg a *LikeButton* gombra kattintás logikáját. Ahoz, hogy ez megfelelően működjön a *NewsListItem-en* belül definiálnom kellet lokális state változókat amelyek megváltozása az egész *NewsListItem* újra rajzolását eredményezte.

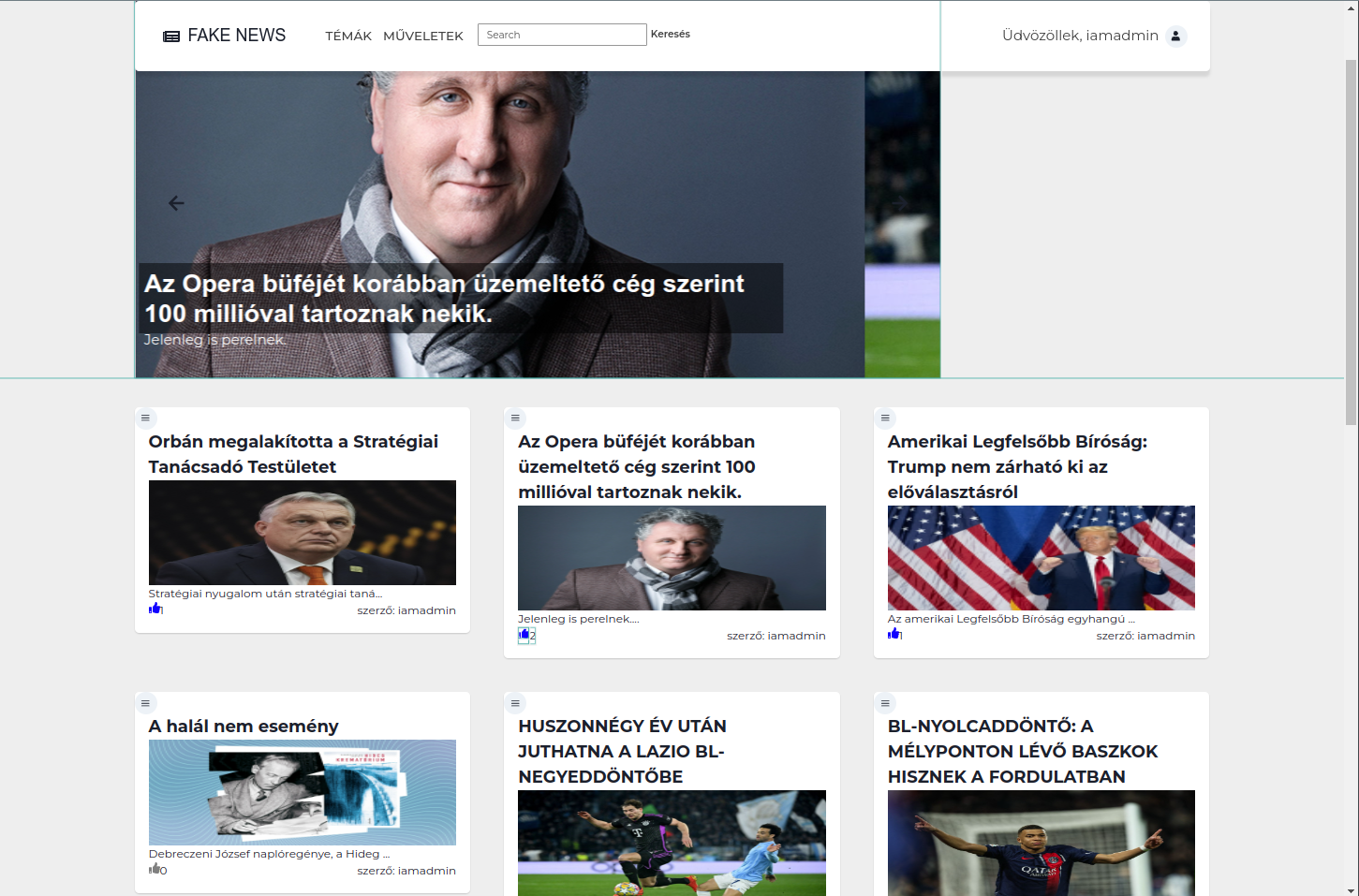


Lokális state változót a *useState()* hook-al definiálhatunk, ezzel tároljuk a komponensek aktuális állapotát és a react ezen keresztül figyeli, mikor indokolt az újra renderelés.

A definícióban megadjuk a változó nevét a függvény nevét amely ezt a változót módosítja, a változó típusát és a kezdő értékét



A *LikeButton* kiszervezése után már megfelelően renderelődik a komponens



Ez komponens felelős továbbá a „like” elküldésért.

Ha éppen nincs bejelentkezett felhasználó egy alert felugró ablakban jelezzük a felhasználónak. Ezt a funkciót később bővíthetjük és egy Modal ban tudjuk dizájnosabban jelezni ezt.



Ezzel a kis logikával azt is megakadályozzuk hogy a *user undefined* értékkel eljusson a *userDidLike* függvényig

Hogy a gombra kattintást dinamikusan kezeljük, folyamatosan figyelnünk kell a global stateban történő user érték változását. Ezt az értéket a *LikeButton* komponens a szülő *NewsListItem-*től properti ként kapja meg.

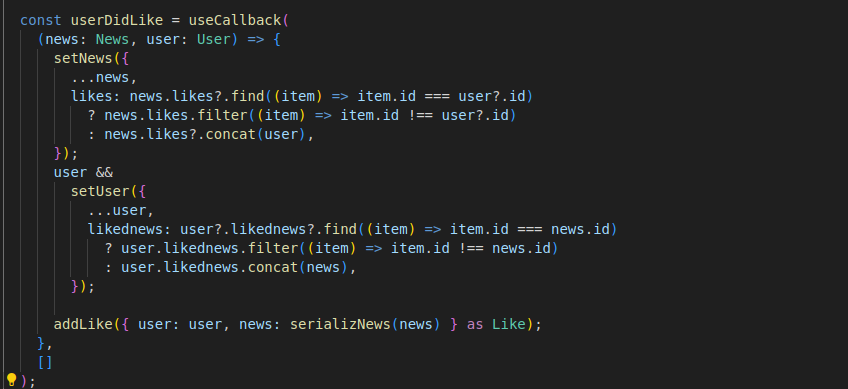


Reactban ha változás történik a globális statben az megváltoztatja a ráhivatkozó változó értékét is a változást egy hook segítségével *useEffect()*-el tudjuk figyelni

Ez a hook két részből áll a függvény törzse, és egy dependencia lista ahol felsoroljuk a figyelni kívánt változókat



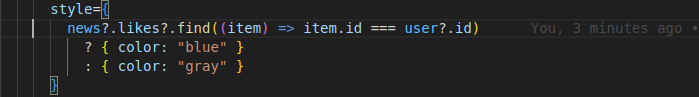
A *LikeButton-*ban a lokális state változók segítségével állítjuk össze a *Like* objektumot amít elküldünk a szervernek és ebben tároljuk az aktuális hír tulajdonságait (likeok száma) így elkerüljük a fölösleges api hívást.



A setNews függvényben előszor betöltjük a korábbi értékeket, majd vizsgáljuk az online user-t hogy a lájkolók között van-e, ha igen a filter függvénnyel végig iterálunk rajta ismét, ez a függvény egy tömböt ad vissza, melynek tartalma minden olyan elem amely id-ja nem egyenlő a user.id-val kvázi töröljük az online usert ebből a listából, ha nem elemég egyszerűen hozzá fűzzük a tömbhoz.

Ezt a logikát használjuk a setUser függvényben is. Végül elküldjük a Like-ot

A lokális statben történ változások újra rendereleik a komponenest így a megfelelő színnel azonnal megjelenik a like aktuális állapota



## **3.2. Backend**

A szerveroldali alkalmazást java nyelven készítettem springboot keretrendszert használva. A springboot backend alkalmazások két típusát különböztetjük meg SSR vagyis Server Side Rendered ebben az esetben a szerver rendereli a html oldalt és küldi vissza a kliensek ebben az esetben a rétegeket MVC modell szerint különítjük el.

Másik típust RESTful API alkalmazásnak nevezzük ebben az esetben a szerver nem felelős a kinézetért csak adatokat szolgáltat a kliensek, ebbe a kategóriába tartozik az általam készített alkalmazás is.

Ebben az esetben a rétegek a

-Modell

-Repository

-Service

-Controller

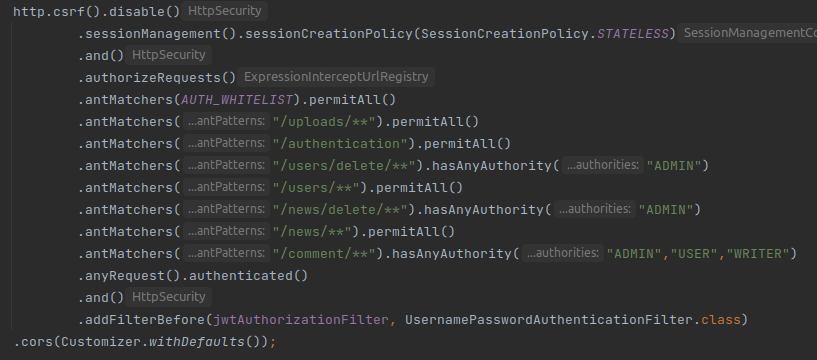
-Exception Handling

-Configuration

### 3.2.1 Configuration

Egy beérkező HTTP kérés a következő utat járja be. Az alkalmazásom biztonságáért a Spring Security, továbbiakban csak security, felel, ez egyike a felhasznált moduloknak, amelyeket a pom.xml fileban adunk meg. Ebben az alkalmazásban a maven végzi ezeknek a moduloknak az integrálását egy központi repositoryból.

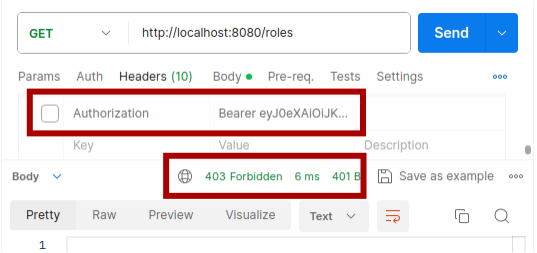
Ennek a modulnak a konfigurálását a *configuration/SecurityConfiguration.java* fileban végeztem.

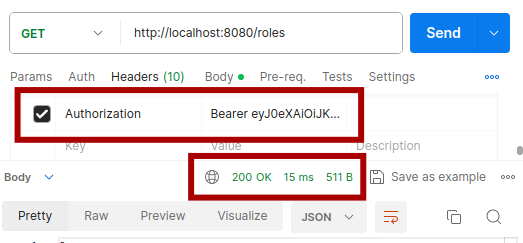


Az *AUTH\_WHITELIST* tömbre a swagger-ui miatt volt szükségem enélkül a security megtagadja a swager hozzáférését az alkalmazáshoz, továbbiakban láncolva sorolom fel a használt endpointokat amelyekre szeretnék valamilyen elérési korlátozást vagy éppen bárkinek elérhetővé tenni .

*anyRequest().authenticated()* minden a felsorolásban nem szereplő endpoint csak hitelesítés után érhető el.  
A *../roles* endpointit a program korai verziójában készítettem a jogosúltságok frontendről történő szerkeztésére, azonban ez a future nem került kialakításra.

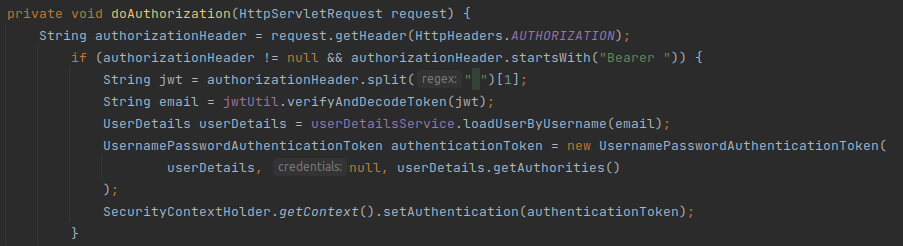
Ezen keresztül betudom mutatni mi történik ha autentikáció nélkül illetve autentikációval próbálok elérni egy a felsorolásban nem szereplő endpointot.





*.addFilterBefore(jwtAuthorizationFilter, UsernamePasswordAuthenticationFilter.class)* továbbá beállítom hogy a *jwtAuthorizationFilter* minden előtt fusson le.

Ez a néhány konfiguráció csak felületesen érinti a SpringSecurity-t és én sem mélyedtem el jobban a működésébe.   
 Ennek a filternek a feladata megvizsgálni hogy a bejövő request volt e már autentikálva vagyis érvényes tokennel rendelkezik. Ha nem volt, megpróbálja elvégezni az autentikációt és ennek függvényében az autorizációt is.



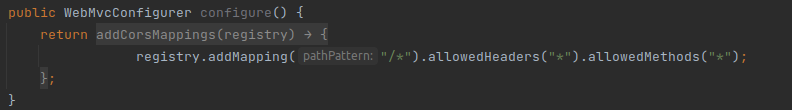
A *request*ből, ha létezik kiolvasom a nyers token-t, a jwtUtil objektum dekódolja ezt és vissza adja a felszanáló email címét.

Következő lépésben használom a *security*ből származó *UserDetailsService* interface egy saját implementációját. Ez a ***../security/UserDetailsServiceImpl.java*** fileban van.

Mivel a *UserDetailsService* interface *loadUserByUsername* metódusa *UserDeatails* objektummal tér vissza, ezt a metódust felülírva tudjuk a saját logikánk által vissza keresni a felszanálót az adatbázisunkból, amelyet felhasználva autentikációs tokent készítünk, ez nem összekeverendő azzal ami a requestben jön, és ezt adjuk hozzá a security kontextushoz. A tokenbe a UserDetails objektum mellet bekerül a jogosultságokat tartalmazó authorities kollekció.

Ezt a kollekciót fogja majd a fentebbi képen használni a jogosultságok ellenőrzésére a SecurityConfig osztály.

Végül a a produkciós környezet miatt beállítom Cross-Origin Resource Sharing-et (cors) azért hogy más domainről is elérhető legyen az alkalmazás.

Ehhez még konfigurálnom kellet a *CrosConfig.java*

Ebben a sorban beállítom hogy bármelyik címről bármilyen HTTP metódus engedélyezve legyen.

Produkciós alkalmazásnál ez nem lenne megfelelő de jelen alkalmazáshoz elégséges. Itt tehetnénk még egy biztonsági intézkedést, például adott címről legyen csak elérhető a DELETE metódus.

A security-n átjutó *request* második lépésben a kontoller rétegbe kerülnek. Ennek a rétegnek „csak” annyi a feladata a feladata hogy fogadja a requsteket és visszaküldeje a megfelelő responst, mindemellet ahogy a neve is mutatja továbbítsa a service réteg megfelelő függvényének. Ezt a réteget elvárás logika mentesen tartanunk minden műveletet, számításokat a service rétegben végzünk.

### **3.2.2 Controller**

A **controller** mappában találjuk a köveztkező osztáyokat:

#### 3.2.2.1AutenticationContoller:

Ez a kontroller kezeli a /*authentication* endpointra érkező requesteket, amikor egy felhasználó beakar jelentkezni még nem rendelkezik tokennel így a *JwtAuthorizationFilter* -ben nem kap jogosultságokat, de nem is kell mert ezt az endpointot mindenki számára elérhetőre állítottuk.



Ha a request bodyban kapott username és password alapján sikerült a felasználót azonosítani visszaküldjük a JwtUtil osztály-al készített token objektumot.

Ezt a tokent konfigurálhatjuk az *application.yml* fileban az fájl és az osztály közötti kapcsolatot a *configuration/JwtConfigurationProperties* osztály végzi.

****

A annotáció hatására a springboot megkeresi a *jwt -*bel kezdődő blokkot pontosabban block-mapping-nek nevezett részt az *application.yml* fileban

#### 3.2.2.2 UsersContoller ,NewsController:

Ugyan azokat a CRUD requesteket kezelik

#### 3.2.2.3.RolesContoller

Jelenlegi verzióban nem használom.

#### 3.2.2.4 CommentContoller

Jelenleg csak a kommenteket csak hozzáadni tudunk. Nem lehet törölni illetve szerkeszteni.

#### 3.2.2.5 ImageController

A felhasználók profil képeit a szerveren tárolom. Ennek oka, hogy szerettem volna ezt a módszert is megismerni és bemutatni. Valamint egy ilyen kis méretű alkalmazásnál ez elégséges.

Ez az endpoint adja vissza a pathban kapott nevű képet. Minden felhasználó a regisztráció során kap egy alapértelmezett profilképet. Jelenlegi verzióban ez az *uploads/image.jpeg.*

-Profilkép feltöltés és frissítés:

A /users/update endpointon keresztül *UserDTOForUpdate* objektumot fogadok ez egy *usersDTO* és egy *Image* objektumot tartalmaz.



A bejövő Image objektum alapján eldönti az algoritmus, hogy van-e nem alapértelmezett kép beállítva profilkép ként, tehát saját feltöltésű kép.

Ha új kép érkezik átadjuk az ImageService-nek a base64 ben kódolt képet, amit így stringként tárolunk, ez a service végzi a szerverre való mentést és a törlést.

### 3.2.3 Dto

Itt tárolom a hálózaton küldött illetve fogadott objektumokat, ezek álltalában szermaztatottjai egy nagyobb kiterjesztésű Entitynek. Eltudjuk dönteni mennyi atributumra van szükségünk a frontenden és csak azokat küldjük, „hagyjuk meg”

### 3.2.4 Entity

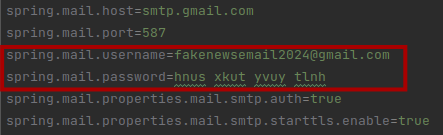
Itt a service/logikai rétegben használt objektumokat tárolom

### 3.2.5 Service

A korábban említett logikai réteg. Külön tárolom a servicek mint interface és azok implementációja, ezt a módszert azért használjuk hogy lehetőség legyen egy szervice több módszerrel megvalósítani.

### 3.2.6 Email küldés.

A *spring-boot-starter-mail* dependency-t használom az emailek küldésére. Az email küldő service a *service/email/EmailService* fileban egyszerű sablon utasítás sorozat. Konfigurálni ezt az *application.poproperties* fileban tudjuk, én gmail-es email címet használtam, ez lesz a username a password értéke pedig nem az email címhez tartozó jelszó, hanem egy úgynevezett alkalmazásjelszó



### **3.2.7 Hibakezelés**

Az *exceptions* mappában hoztam létre saját hibaüzeneteket. Például abban az esetben amikor egy felhasználó blokkolt az adatbázis szerint *UserIsBlockedException-t* dobunk a service rétegből amelyet a *controller/EntityControllerAdvice* figyel. Ebben az osztályban felsorolt hibákra figyel ez a kontoller és amikor a szerveren valamelyik jelentkezik a megfelelő hibaüzenetet küldi vissza responban

### 3.2.8 Repository

Itt valósítjuk meg a kapcsolatot a kapcsolatot az adatbázis kezelő réteggel azzal hogy kiterjesztjük a *JpaRepositoryt* ezzel az egyszerű előre definiált lekérdezések mellet mi is tudunk megadni, erről írtam korábban.

## **3.3 Adatbázis**

### **3.3.1 Java Persistence API (JPA):**

Az alkalmazás backend-je a JPA könyvtárat használja az adatbázishoz való kapcsolódáshoz, műveletek végzéséhez.

Ennek segítségével bármilyen relációs adatbázishoz tudunk kapcsolódni, mivel nem adatbázis specifikus lekérdezéseket írunk.

A fejlesztés során XAMPP-al futtatót lokális MySQL adatbázist használtam.

A kapcsolódáshoz mindössze a projekt *application.properties* file-ban kell megadnunk az adatbázishoz való kapcsolódás URL-jét, felhasználónevet és jelszót.

Az adatbázis tábláit a Modell rétegben implementált osztályok alkotják. Ezeket az osztályokat összefoglaló néven Entytiknek nevezzük. A táblák közötti kapcsolatok kialakításához ezeknek az osztályoknak a megfelelő attribútumait kell a kapcsolatot jelző annotációval ellátnunk: *@ManyToOne, @OneToMany, @ManyToMany* azannotációtmindakétosztályban jelölnünkkell*.*

Az alábbi kódrészlet egy több a többhöz kapcsolatot létesít a *news* és a *typeofnews*táblák között.

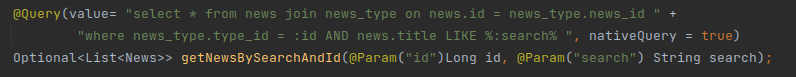
@ManyToMany(mappedBy = "types")  
private List<News> news;

Több a többhöz kapcsolatot a JPA segédtáblák segítségével valósít meg, szabadon megadhatjuk a tábla nevét, valamint a *@JsonBackReference* annotáció a visszahivatkozást és ezáltal a végtelen ciklust akadályozza meg.

@ManyToMany  
@JoinTable(name = "news\_type",  
 joinColumns = @JoinColumn(name = "news\_id"),  
 inverseJoinColumns = @JoinColumn(name = "type\_id"))  
@JsonBackReference(value="newsTypes")  
private Set<TypeOfNews> types;

Az xampp-al futtatott mysql adatbázist a [*http://localhost/phpmyadmin/index.php*](http://localhost/phpmyadmin/index.php) címen érjük el több funkció mellet itt kapunk egy tervezőnézetet is ami láttatja a   
tábláink közötti kapcsolatokat.

A JPA az alapvető CRUD (Create,Read,Update,Delete) műveletek mellet lehetőséget nyújt speciális lekérdezések használatához, amiket a repository rétegben helyezünk el.



## 3.4. További fejlesztési lehetőségek:

### **3.4.1 Email**

A szerveren beállítottam az email küldést ez jelenleg csak egy üdvözlő email-t küld a regisztráció után. De használható lehet a felhasználó validálására egy kód kiküldésével, melyet a szerver oldalon generálunk azt a kliens oldalon keresztül adhatja vissza a felhasználó és a szerveren pedig ellenőrizzük az egyezést.

### 3.4.2 Levelezés

Ahhoz, hogy a felhasználók üzeneteket tudjanak küldeni egymásik felhasználónak, csupán két táblát kell hozzáadnunk a jelenlegi adatbázishoz. Egyik tábla tárolná, az üzeneteket szövegét az egyedi id-vel ehhez jelen esetben egy új osztályt kell bevezetnünk nevezzük Messages-nek, a másik tábla pedig egy segédtábla a küldő, a címzett és az üzenet id-jét. JPA-t használva ez utóbbi automatikusan generálódik, ha a megfelelő annotációkkal láttuk el a Messages osztály attribútumait.

### 3.4.3 User Details kibővítése, értesítések

A Users táblát illetve osztály egy új attribútummal kibővítve tárolnánk hogy az adott felhasználót, ebben az esetben leginkább írót, kik követik és egy újonnan közzétett hír esetén ebben a listában szereplő minden felhasználó kapna email vagy a belső levelezésen keresztül egy üzenetet, miszerint a kedvence posztolt egy új írást.

Tehát az elképzelés lényege hogy nem azt tároljuk hogy az adott felhasználó kiket követ, hanem hogy őt kik követik.

### 3.4.3 Password

Jelenleg a jelszavakat mindenféle titkosítás nélkül tárolja az adatbázis, egy éles alkalmazás esetén ez nagyon nagy biztonsági kockázat.

# 4.Eredmények:

## 4.1 Frontend:

Egy felhasználóknak lehetősége van böngészni a közzétett hírek között, azokra kulcsszavak és/vagy témakör alapján szűrni. Regisztrációt követően képes a cikkeket kedvelni, illetve kommentet fűzni azokhoz. A saját felhasználói adatlapját szerkeszteni hozzá egyéni profilképet beállítani. A regisztrált felhasználóknak három jogosultsága lehet   
úgymint USER, WRITER, ADMIN. Minden regisztrált felhasználó alapból USER, a további jogokat adatbázis oldalról lehet kiosztani. WRITER jogú felhasználó közzétehet új cikket, láthatja más felhasználók adatlapját. ADMIN joggal képes a híreket szerkeszteni, törölni, más felhasználókat blokkolni illetve bele nézhet és szerkesztheti azok adatlapját, gyanítom ez személyiségi jogokat sérthet de az extra funkció miatt így készítettem.

## 4.2 Backend:

Az API szervert Java nyelven készítettem, SpringBoot keretrendszerrel, és a SpringSecurity felelős a biztonságért. Így lehetőség van az egyes endpointok hozzáférését autentikációhoz kötni, de készíthetünk mindenki számára elérhetőt is. A szerver képes kiszolgálni bármilyen nyelven írt másik applikációt illetve más módon indított requestet gondolok itt például a postmanra. Az adatbázissal való kapcsolatot a JPA valósítja.

# 5.Elemzés:

A React-re épül a Facebookon kívül többek között a BBC, Instagram, Imgur és a Netflix is. Ezek közös jellemzője hogy egy időben hatalmas tömegeket kell kiszolgálniuk, illetve különböző eszközökön is elérhetőek. Napjainkban elvárás, hogy a felhasználókat folyamatosan ingerek érjék és ebbe nem fér bele a töltési idő ami ezt megszakítaná. Az én alkalmazásom nyilván összehasoníthatatlan a fentiekkel. Azonban Reactot használva annyi közös hogy ezt a töltési idő minimalizálást két módon valósítja meg első sorban a DOM manipulációval, vagyis a betöltött oldal csak bizonyos részét frissíti (lásd 3.1.18), továbbá lekérdezések, illetve a böngésző cache memória használatának összehangolásával   
(lásd [3.1.15.5](#__RefHeading___Toc2096_3355630758))

# 6.Összefoglalás:

Kétszer vettem fel a webalkalmazás fejlesztés nevű tantárgyat. Először családi okok, munka, és a könnyelműségem miatt nem sikerült időben csatlakoznom a tananyaghoz és már későn mértem fel milyen terjedelmes téma ez.   
Másodjára az utolsó szemeszterben tudtam felvenni de itt már felkészültem és az eltelt időben vissza néztem a korábbi EPAM-os előadásokat. Ennek köszönhetően másodjára sikeresen vizsgáztam igaz a mostanitól egy sokkal kevesebb feature-el rendelkező hírportál készítésével.

A szakdolgozatom témája a Java backend javscript frontend alapú webalkalmazás fejlesztése volt. Ez jelenthetett néhány statikus html oldalból álló weboldalt amelyet javascript hajt meg. Ehelyett folytattam a hírportál fejlesztését. Sokszor felmerült bennem, hogy a kelleténél és a lehetőségeimhez képest többet vállaltam, nem csupán az új typescript nyelv de a React, a Reactal használt Redux könyvtár megértése is nehezítette a készítést. Törekedtem arra hogy a lehető legtisztább kódot tudjam elkészíteni, és a komponensek újra felhasználóak legyenek, azonban ahhoz, hogy az illyes fajta kódolási technikának kialakuljon sok-sok óra kódolás és különböző feladatok kellenek, és persze egy olyan környezet, ahol visszajelzést kaphatok és tanácsokat, úgy gondolom ez csak sok éves munkahelyi tapasztalattal érhető el. Utólag nem bánom hogy ezt a komplexebb megvalósítást választottam, ezzel is bővítettem a tudásomat.

A program elkészítése után a dokumentáció majd nem ugyan annyi időt vett igénybe, sajnos számomra nehézkes az ilyen jellegű írás készítése de Vályi Tanár Úr segített ebben is hogy megtaláljam a megfelelő hangnemet, amit ez úton is köszönök neki.

Bízom benne hogy sikerül minél hamarabb junor fullstack fejlesztő ként elhelyezkednem az itt megszerzett tudásommal.

# 7. IRODALMI JEGYZÉK

# 8. MELLÉKLET

1. ALKALMAZÁS INDÍTÁSA

## 10.1. GitHub verziókezelés

Az alkalmazás verzió követéséhez a GitHub-ot használtam. Ha valaki folytatni szeretné a fejlesztését innen tudja letölteni az alkalmazás jelenlegi állapotának megfelelő forrás fájlokat. Ehhez először is gépünkre telepítenünk kell a Git szoftvert, amelyet a [Git weboldal](https://git-scm.com/)-ról tudunk letölteni, telepítés után a *git –version* paranccsal tudjuk ellenőrizni sikeres volt-e a telepítés.

Ezután klónoznunk kell a git repositoryt, ami lényegében annyit jelent, hogy egy másolatot hoz létere a számítógépünkre.

A JDK letöltésénél fontos hogy a megfelelő verziót töltsük le és állítjuk be a projekt SDK- ként, én a 17-es verziót használtam ezt a pom.xml fileban is tudjuk ellenőrizni.

## **10.2 Docker**

Azonban annak aki csak kiszeretné próbálni az alkalmazást elég a [Docker](https://www.docker.com/products/docker-desktop/) nevű konténerizációs program megléte a gépén. Telepítés után a *docker -v* parancsal tudjuk ellenőrizni sikeres volt-e a telepítés.

A docker nevű mappába töltöttem fel három tömörített állományt melyekbe külön-külön csomagoltam az alkalmazás docker image-it.

Ezek az adatbázis, az image fájlba van csomagolva az adatbázis induló állapota adatokkal feltöltve. Ha ezt a konténert szabályosan állítjuk le munka után akkor a benne végrehajtott változások mentésre kerülnek.

Az alkalmazás frontend-je, az alap operációs rendszer egy alpine linux amelyre telepedik az ngnix, kiszolgálja a böngészőnek az alkalmazást.

A szerver image-ba egy bash filet másolok ami késlelteti a szerver indulását 15 másodperccel.

Több konténer egyszerre indítására használható a *docker composer*

A mappában található a *docker-compose.yml* ebben a fileban tudjuk felparaméterezni az indítani kívánt conténereket. Webalkalmazás lévén sarkalatos pont a hálózati kommunikáció ezt alapesetben egy docker network indításával oldjuk meg és a konténerek ehez a networkhöz kapcsolódnak, azonban a networkön belüli IP címek kiosztása minden indításnál változik, változhat. Ennek okán a docker-compose.yml filban manuálisan adtam meg melyik IP cím tartományt használja a network.   
 Ha ebből a mappából kiadjuk a *docker-compose up* elindul a három kontáner, a szerver 15 másodperces késleltetésének oka itt van mégpedig, hogy megvárjunk amíg felépül az adatbázis szerver.

Email: fakenewsemail2024

pass: FakeNews2025