



Java backend javascript frontend alapú webalkalmazás

Hódi Szilárd

Programtervező informatikus BSc

Konzulens: Dr. Vályi Sándor

**2024**

## NYILATKOZAT

Alulírott

Név: HÓDI SZILÁRD ATILIA

Szak: Programtervező Informatikus, B.Sc.

NEPTUN kód: CECX1K

jelen nyilatkozat aláírásával kijelentem, hogy a

Java Backend Java Script frontend alapú  
webalkalmazás

című szakdolgozat önálló munkám eredménye, saját szellemi termékem, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel. A dolgozat készítése során betartottam a szerzői jogról szóló 1999. évi LXXVI. törvény szabályait, valamint a Nyíregyházi Egyetem által előírt, a szakdolgozat készítésére vonatkozó szabályokat.

Kijelentem továbbá, hogy sem a dolgozatot, sem annak bármely részét nem nyújtottam be szakdolgozatként.

Nyíregyháza, 2025. 01. 14.

Hódi Szilárd Attila  
a hallgató aláírása

# NYÍREGYHÁZI EGYETEM

4401 Nyíregyháza  
Sóstói út 31/b

Tel: 42/599-400  
Fax: 42/402-485

Intézményi azonosító: FI 74250

## SZAKDOLGOZATI LAP

BA, BSc képzéshez

Név : Hódi Szilárd Attila

NEPTUN-kód : CECXIK

Alapképzési szak : Programtervező Informatikus

Szakedolgozat címe: Java backend, JavaScript frontend alapú hírportál megvalósítása

A témavezető: Dr. Vályi Sándor

neve



aláírása

A szakfelelős: Dr. Falucska János

neve



aláírása

Intézet, tanszék: Informatika és Matematika Intézet

Nyíregyháza, 20 20 év 1 hónap 31 nap



a hallgató aláírása

Kitöltendő 3 példányban: 1 példányt a szakfelelősnek,  
1 példányt a Tanulmányi előadónak kell leadni  
1 példány a hallgatónál marad

# Tartalomjegyzék

1. BEVEZETÉS.....	6
1.1. Alkalmazás célja .....	6
2. IRODALMI ÁTTEKINTÉS.....	6
2.1. JavaScript .....	6
2.2. TypeScript .....	7
2.3. React.....	7
2.4. Mi az a SPA? .....	7
2.5. Redux .....	7
2.6. React router: .....	8
2.7. Java.....	8
2.8. Spring Boot .....	8
2.9. Maven.....	8
2.10. Mi az API .....	9
2.11. Visual Studio Code.....	9
2.12. Swagger UI / Postman.....	10
2.13. Chrome extensions .....	10
2.14. Chakra UI .....	10
2.15. Xampp.....	10
3. A FEJLESZTÉS MENETÉNEK LEÍRÁSA .....	10
3.1. Frontend .....	10
3.1.1. Navbar: .....	11
3.1.2 Icon: .....	12
3.1.3 Témák: .....	12
3.1.4 Műveletek: .....	12
3.1.5 Keresési mező .....	13
3.1.6 Regisztráció .....	16
3.1.7 Bejelentkezés .....	19
3.1.8 Felhasználók listája.....	21
3.1.9 Új cikk hozzáadása .....	22
3.1.10 Cikk szerkesztése.....	23
3.1.11. Carousel komponenes .....	24
3.1.12 Single user page .....	25
3.1.13 News .....	27
3.1.13.1 NewsListProvider.....	27
3.1.13.2 NewsList .....	28
3.1.13.3 NewsListItem .....	28
3.1.13.4 NewsDescriptionProvider .....	28
3.1.13.5 NewsDescription .....	30
3.1.14 Comment .....	30
3.1.14.1 MyComment .....	30
3.1.14.2 CommentForm .....	30
3.1.15 Store directory .....	30
3.1.15.1 Store.ts.....	30
3.1.15.2 Slices .....	31

3.1.15.3 News-api.ts .....	33
3.1.15.4 Query .....	34
3.1.15.5 Mutation .....	34
3.1.16 Hookok .....	35
3.1.17 Utils directory .....	37
3.1.18 Componensek renderelésére, alias nevek használata .....	37
3.2. Backend.....	42
3.2.1 Configuration.....	42
3.2.2 Controller.....	45
3.2.2.1 AuthenticationController:.....	45
3.2.2.2 UsersController ,NewsController:.....	45
3.2.2.3.RolesController.....	45
3.2.2.4 CommentController.....	46
3.2.2.5 ImageController .....	46
3.2.3 Dto .....	47
3.2.4 Entity .....	47
3.2.5 Service .....	47
3.2.6 Email küldés. ....	47
3.2.7 Hibakezelés.....	48
3.2.8 Repository.....	48
3.3 Adatbázis .....	48
3.3.1 Java Persistence API (JPA): .....	48
3.4. További fejlesztési lehetőségek:.....	50
3.4.1 Email.....	50
3.4.2 Levelezés .....	50
3.4.3 User Details kibővítése, értesítések .....	50
3.4.3 Password.....	51
4.EREDMÉNYEK: .....	51
4.1 Frontend: .....	51
4.2 Backend:.....	51
5. ELEMZÉS .....	51
6. ÖSSZEFOGLALÁS .....	52
7. IRODALMI JEGYZÉK.....	53
8. MELLÉKLET.....	53
10.1. GitHub verziókezelés .....	53
10.2 Docker.....	54

## **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 eldöntennem 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 ágyazandónak 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 script nyelvként fut a böngészőkben, interaktivitást és dinamizmust biztosítva a weboldalak számára. Az aszinkron programozás és esemény vezérelt 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 frontendről 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 kódírást. 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**

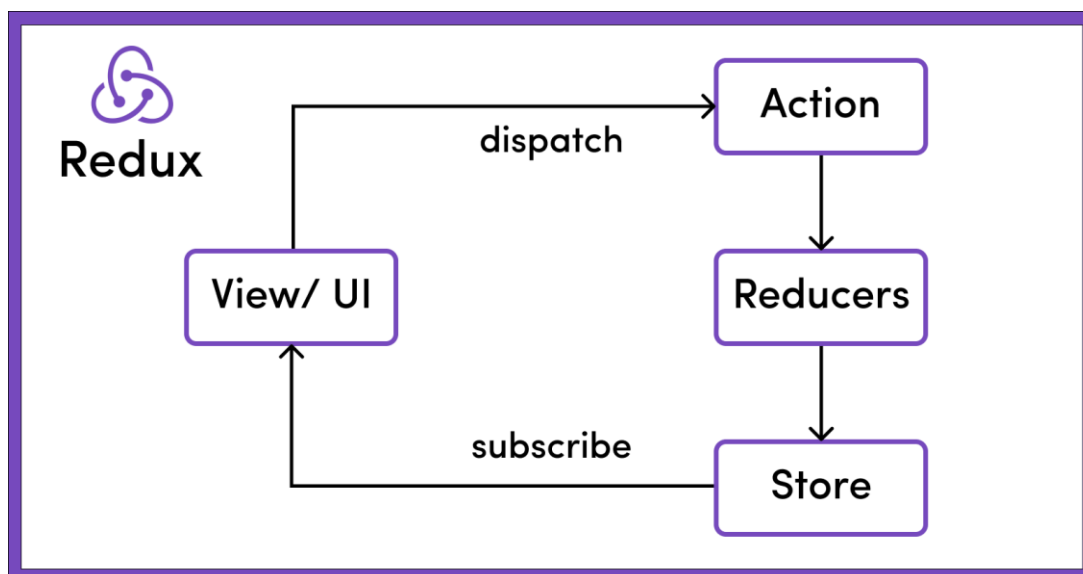
Egy nyílt forrású felhasználói felület készítésére használt könyvtár. Melyet a Facebook fejlesztett ki 2011ben és tett nyílt forrásúvá 2013-ban. A hatékonyan kezeli a DOM manipulációt a virtuális DOM segítségével, ami növeli a teljesítményt és optimalizálja a frissítéseket. Több könyvtárral együtt szokás 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.



1. ábra Redux állapotkezelése

## 2.6. React router:

Az alkalmazás oldalai közötti navigációra használt könyvtár

## 2.7. Java

A JAVA egy platform független objektum 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 Control) 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.

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 projektben a <http://localhost:8080/swagger-ui/index.html#/> oldalon elérhető lesz egy felület mely az alkalmazásban definiált endpointokat felismeri és pár kattintással tudunk különböző requesteket küldeni. Spring security miatt konfigurálni kell az elérési utat, amíg ezt nem fixáltam a Postmant használtam.

## 2.13. Chrome extensions

Webalkalmazás révén a fejlesztéshez böngésző szükséges, a chrome stabil népszerű böngésző. Remek pluginokkal a reactos alkalmazások teszteléséhez én a

- React Developer Tools

- Redux DevTools

Előbbi kevésbé utóbbit folyamatosan a Redux state változásának figyeléséhez,

## 2.14. Chakra UI

A React-hez kifejlesztett, felhasználói felület kialakítására használható könyvtár. Könnyen és gyorsan készíthető vele egységesen stilizálható felhasználói felület. Kombinálható más komponens könyvtárakkal, illetve saját készítésükkel 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

Ezekon 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.



2. ábra Chakra UI navbar sablon

Ezt a kódot módosítottam, illetve alakítottam át, hogy a backendről érkező adatokat kezelni tudja  
Bejelentkezés után a komponens nézete megváltozik aszerint, hogy milyen



3. ábra Módosított navbar ( bejelentkezés nélkül)

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.



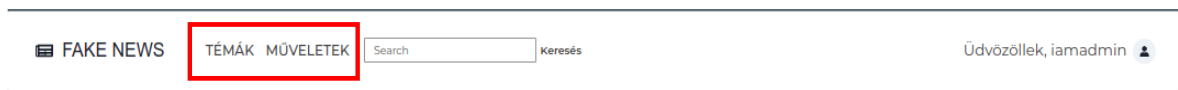
4. ábra Módosított navbar (bejelentkezés után)

### 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 visszavigál 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.



5. ábra Navbar menu

### 3.1.3 Témák:

Hírek kategóriáit azaz a backendről lekérdezett TypeDTO-t jeleníti meg

### 3.1.4 Műveletek:

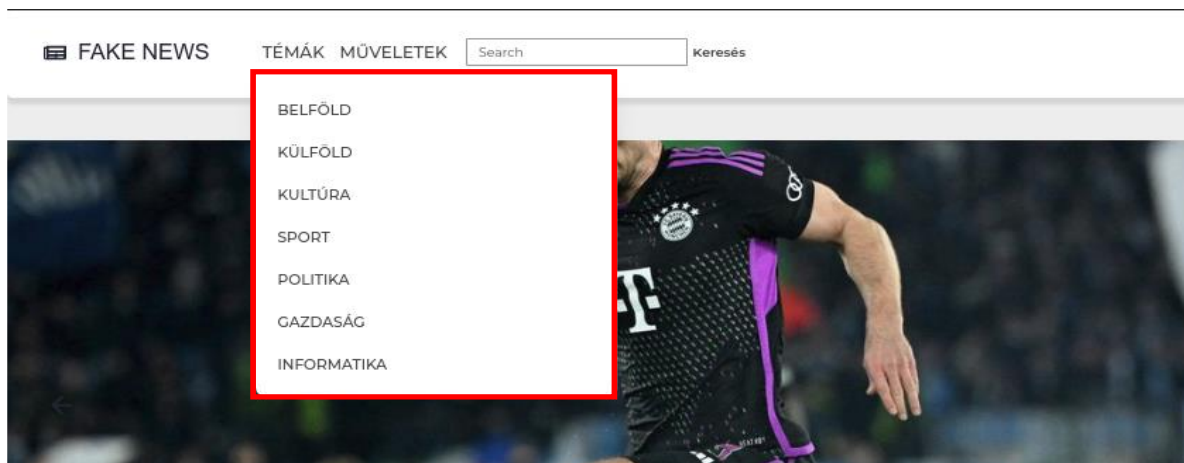
A navbarban hard kódolva adtam meg a két lehetőséget. Mind két komponens tulajdonképpen ugyan az bár a megjelenítendő elemek tulajdonságai mások. Míg a Témák komponens `Type` típusú elemeket jelenít meg `'id'` és `'title'` attribútumokkal, amelyek megfelelőek voltak a menu elemek `value` és `címke` értékeinek 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.

```
interface MenuItem {  
  label: string;  
  children: NavMenuItem[];  
}
```

6. ábra MenuItem interface

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

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



7. ábra Témák menu

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

### 3.1.5 Keresési mező



8. ábra Kereső 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.

```
query: (params: GetRequestParamsForNewsQuery) => ({
  url: `http://${SERVERHOST}:8080/news/type/${params.typeId}/${params.limit}/${params.side}/${params.search}`,
  method: "GET",
}),
```

9. ábra Query template

A backend endpointot is kiegészítettem a search paraméterrel. A RequestMapping annotációval tudjuk megadni a path template-t, továbbá hogy egy GET módszerre figyelünk. *ResponseEntity*-t 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 kellett amikor a keresésnek megfelelő hírek listájának végére értem, ekkor inaktívvá válik a lapozó gomb.

```
@RequestMapping(path="/type/{id}/{limit}/{side}/{search}", method = RequestMethod.GET)
public ResponseEntity<ResponseForNewsDTO>findByType(@PathVariable Long id,
                                                    @PathVariable int limit,
                                                    @PathVariable int side,
                                                    @PathVariable String search){
```

10. ábra Szerver oldali endpoint

A konstruktorban a *@PathVariable* annotációval sorban megadjuk az URL-ben érkező paraméterek listáját.

Az id paraméterben 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 NewsControllerben, 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 visszaküldött 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.

```
@Query(value= "select * from news join news_type on news.id = news_type.news_id " +  
    "where news_type.type_id = :id AND news.title LIKE %:search% ", nativeQuery = true)  
Optional<List<News>> getNewsBySearchAndId(@Param("id")Long id, @Param("search") String search);
```

*11. ábra Két táblás natív SQL lekérdezés*

A lekérdezésben kiválasztjuk a news-t, amelyet a news\_type-al kötünk össze, azokat a híreket szeretnénk megkapni melyek type\_id-a a segédtablából 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 azért hasznos mert ha a lekérdezés nem hoz eredményt könnyen tudjuk kezelni.

```
@Query(value= "select * from news where title LIKE %:search%", nativeQuery = true)  
Optional<List<News>> getNewsBySearch( @Param("search") String search);
```

*12. ábra Natív SQL lekérdezés*

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

### 3.1.6 Regisztráció



13. ábra Navbar regisztráció/bejelentkezés

The image shows a registration modal window titled 'Regisztráció' with a close button (X) in the top right corner. The form contains several input fields and validation messages. The first section is labeled 'EMAIL' and has a text input field with the placeholder 'Adja meg az email címét'. Below it is a red error message: 'az e-mail megadás kötelező'. The next section is labeled 'JELSZÓ' and has two text input fields. The first has the placeholder 'Adja meg a jelszavát' and a red error message below it: 'A jelszó megadása kötelező. JELSZÓ ÚJRA'. The second has the placeholder 'Adja meg a jelszavát újra' and a red error message below it: 'A jelszó megerősítése kötelező.' There is an eye icon to the right of the second password field. The third section is labeled 'KIVÁLASZOTT CHAT NÉV' and has a text input field with the placeholder 'Adja meg a chat nevét'. Below it is a red error message: 'A név megadása kötelező, ezzel tudsz bejelentkezni. ADJA MEG VEZETÉKNÉVÉT'. The fourth section is labeled 'ADJA MEG KERESZTNÉVÉT' and has a text input field with the placeholder 'Adja meg a keresztnévét'. At the bottom of the form, there is a 'Regisztráció' button and two buttons: 'Bejelentkezés' and 'Bezárás'.

14. ábra Regisztrációs modal

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 megnyitva 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 mellett itt is használtam a *useEffect()* hook *return* függvényét amely 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 *formik* 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 form egyes bemeneti mezőire külön-külön láncolt formában adhatunk meg feltételeket, az egyszerű hossz értéktől egy api kérés *respons*ában érkező érték vizsgálatáig.

```
email: Yup.string()
  .email("Érvénytelen e-mail cím formátum.")
  .test("unique-email", "ez az email cím már foglalt", async (email) => {
    if (helper !== email)
      isAvailable = await isNameAvailable(email);
    helper = email ? email : "";
    return isAvailable;
  })
  .required("az e-mail megadás kötelező"),
```

15. ábra Yup email cím validáció

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

A *useForm*ban megadhatjuk a validációs sémát, azonban még a benne megadott függvényeket a mezőkhöz 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 fájlban definiált fetch api kérés vizsgálja dinamikusan szabad-e a regisztráció az email címmel.

The image shows three examples of email form validation error messages. Each example consists of a label 'EMAIL' in red, an input field with a blue border, and an error message in red text below the field.

- Example 1: The input field contains 'iamdamingmail.com'. The error message is 'Érvénytelen e-mail cím formátum.'
- Example 2: The input field contains 'iamdamin@gmail.com'. The error message is 'ez az email cím már foglalt'.
- Example 3: The input field contains 'iamdamin1@gmail.com'. There is no error message, and a green checkmark icon is visible at the bottom left of the form container.

16. ábra Validációs hiba üzenetek

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

```
onChange={(event) =>
  setFieldValue("email", event.target.value)
}
```

17. ábra Email field bemenet átadása a *useFormik*-nak


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.

```
.test("unique-email", "ez az email cím már foglalt", async (email) => {
  if (helper !== email) isAvailable = await isNameAvailable(email);
  helper = email ? email : "";
  return isAvailable;
})
```

18. ábra Email cím ellenőrzés szerverről

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 elrejt 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*



19. ábra Saját password input componens

Ha minden *field* megfelelően van kitöltve aktívá válik a *Regisztráció* gomb. Ezt az *errors* tömb hosszával ellenőrizzük.

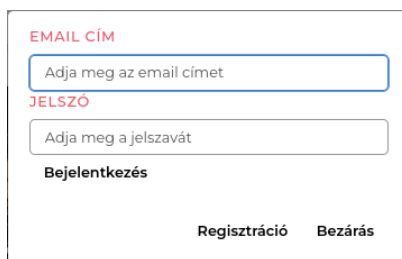
A *useFrom*-al *handleSubmit* függvényével kezelhetjük a *form* submit eseményét is. Ezt az eseményt én a komponensen belül kezelem, egy „üres” user objektum kulcsait töltöm fel a *form* fieldjeinek értékével.

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 autentikáció a felhasználó azonosítása, felhasználói név és jelszó alapján.

A screenshot of a login modal form. It has a white background with a thin grey border. At the top, the text "EMAIL CÍM" is in red. Below it is a text input field with the placeholder "Adja meg az email címet". Underneath that, the text "JELSZÓ" is in red. Below it is another text input field with the placeholder "Adja meg a jelszavát". Below the password field, the text "Bejelentkezés" is in bold. At the bottom right, there are two buttons: "Regisztráció" and "Bezárás".

20. ábra Bejelentkezési modal

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

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

Minden szerver felé irányuló kérés előtt lefut egy autorizációs 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 spring security az első illeszkedő template-re abba hagyja a további vizsgálatot és azt használja tovább.

```
.antMatchers( ...antPatterns: "/news/delete/**").hasAuthority("ADMIN")  
.antMatchers( ...antPatterns: "/news/**").permitAll()
```

21. ábra Endpoint elérésének beállítása

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 érkelődne ki amelyre illeszkedik a */news/delete/* így azt bárki elérhetné. Erről bővebben a backend résznél írok.

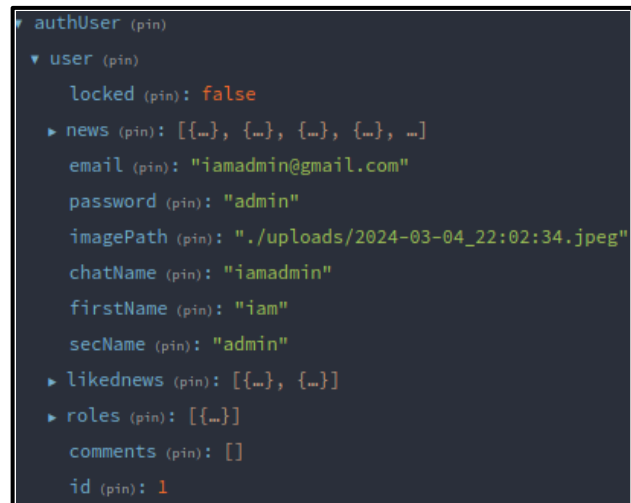
Az esetleges autorizációs hibát a szerveren állítjuk össze és küldjük visszaküldi a frontend felé, amit az inline jelenít meg.

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

22. ábra Szerverről visszakapott hiba üzenet

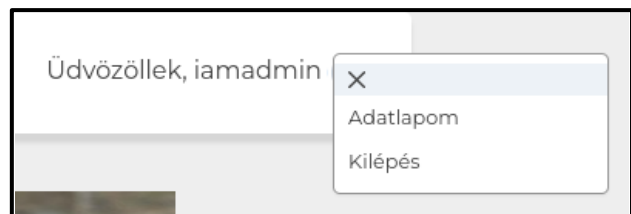
Bejelentkezés után a *user* bekerül az alkalmazás *global state*-jebe a továbbiakban innen használjuk fel, nem indítunk fölösleges lekérdezéseket.

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



23. ábra Redux state megjelenése böngészőn keresztül

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



24. ábra Felhasználói menu

### 3.1.8 Felhasználók listája

Az adatbázisban jelenleg csak néhány felhasználót 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

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

```
return user?.roles?.find(
  (role) => role.title === "ADMIN" || role.title === "WRITER"
) ? (
  <UsersList />
) : (
  <NoPermission/>
);
```

25. ábra Jogosultság vizsgálat

Listában láthatjuk a felhasználók adatait, illetve ikonnal jelezzük melyek vannak blokkolva

FAKE NEWS						TÉMÁK		MŰVELETEK		Keresés		Üdvözlök, iamadmin	

26. ábra Felhasználók listája

### 3.1.9 Új cikk hozzáadása

A **MŰVELETEK** menüben találjuk az Új cikk hozzáadása menüpontot erre kattintva dispatch-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 komponens figyeli autentikált-e a felhasználó és van-e joga elérni a formot. Továbbá a stat-éből

lekérdezi a hírt, ebben az esetben a még üres hírt, és tovább adja a *NewsEditor* komponensnek

Ennél a komponensnél is a *useForm*-ot használjuk a form kezelésére. Az alap input fíldék mellet egy kategória választó komponens használók.

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

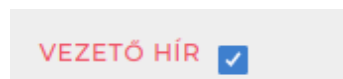


27. ábra Cikk kategóriájának megadása

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

A *validateates-* fileban minimum egy kategória kiválasztását adjuk meg feltételként.

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



Ha minden field megfelelően van kitöltve aktívva 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.



28. ábra Cikk szerkesztés menü

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 frissüljön az egész oldal a state-és propertik miatt.

A *NewsEditoron* belül a *news*-t adom meg a form *initialValues* értékeként, emiatt az értékátadás miatt a *news* változása után nem frissül a form az új értékekkel, emiatt a *useFormik* *setValues* módszerát 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 vezető hírek jelennek meg. Szokásos módon egy provider kezeli az adat átvételt a state-ből és adom át a Carousel komponensnek. Ennek célja, hogy tisztán 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* komponenszt 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.

Az 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 kellett tiltanom az alapértelmezett gombokat. Továbbá egy referenciát kérni a *Slider* komponensre amelyen keresztül elértem a *slickPrev()*

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

```
const settings = {
  dots: false,
  arrows: false,
  infinite: true,
  autoplay: true,
  speed: 500,
  autoplaySpeed: 3500,
  slidesToShow: 1,
  slidesToScroll: 1,
};
```

29. ábra Carousel konfiguráció

```
<Slider {...settings} ref={(slider) => setSlider(slider)}>
```



### 3.1.12 Single user page

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

The screenshot shows a user profile page for a user named 'iroeszter' with the role 'WRITER'. The profile includes a placeholder for a profile picture. Below the profile information, there is a form with labels in red text: 'Chatname', 'Firstname', 'Lastname', 'Email', 'Password', and 'Imagepath'. The form contains input fields for each of these fields. The 'Chatname' field contains 'iroeszter', 'Firstname' contains 'Eszter', 'Lastname' contains 'Iro', 'Email' contains 'iroeszter@gmail.com', and 'Password' contains a masked password. The 'Imagepath' field has a file selection button and the text 'Nincs fájl kiválasztva'. At the bottom of the form, there are three buttons: 'Küldés', 'Vissza a listához', and 'Főoldal'.

30. ábra Felhasználói adatok

Az alap értelmzett 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 korábbiakat törlik onnan.

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

```
<Input
  type="file"
  id="file-input"
  onChange={(event) => {
    const files = event.target.files;
    if (files && files.length > 0) {
      const file = files[0];
      setImage(URL.createObjectURL(file));
      const reader = new FileReader();
      reader.readAsDataURL(file);
      reader.onloadend = () => {
        const userImage = reader.result
          ?.toString()
          .split(",")[1];
        setBase64Image(userImage!);
      };
    }
  }}
/>
```

31. ábra Profilkép kezelése a kliensgépről

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 képesek, 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 rejttem el a UI felületről, hogy végig iterálok az éppen bejelentkezett felhasználó *roles* tulajdonságán és ha egyezést találok ADMIN karakter sorozattal visszaadjuk a komponenes

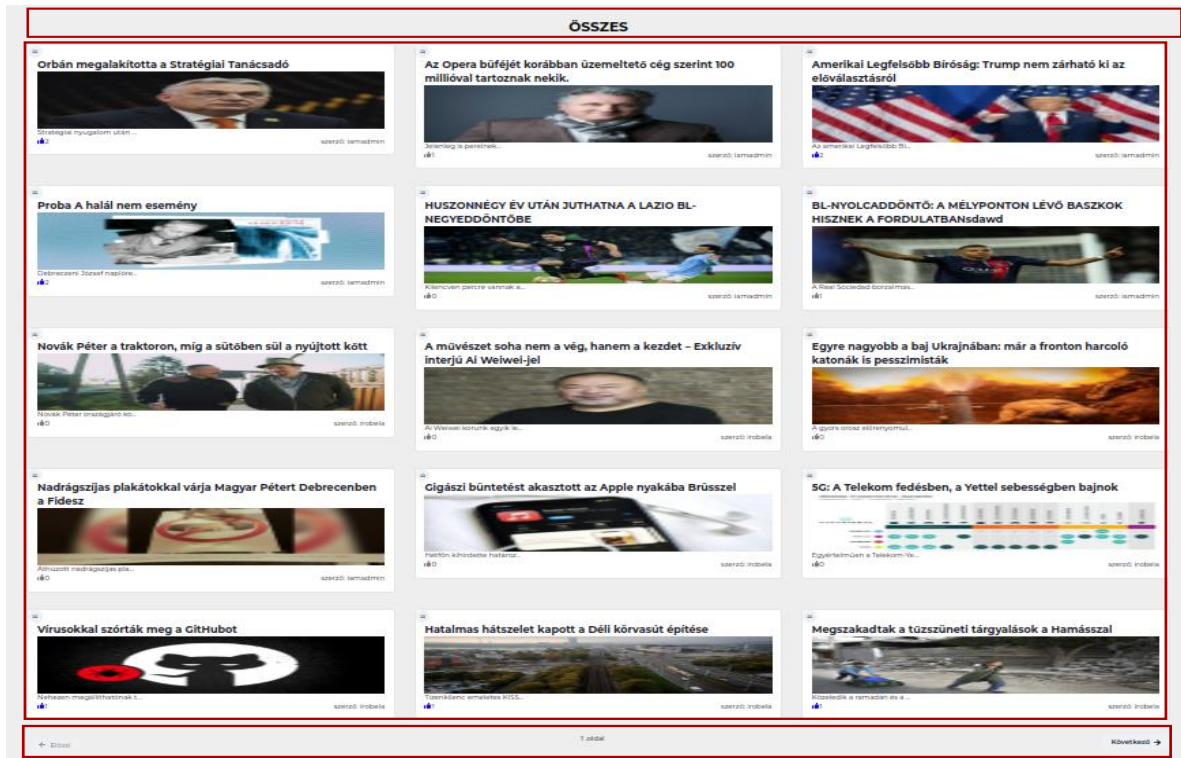
```
{viewer?.roles?.find(roles=>roles.title==="ADMIN")?<Text display={"flex"}>
  <FormLabel> Blokkolás</FormLabel>
  <Checkbox
    paddingLeft={3}
    border={"1px black"}
    display={"flex"}
    size="lg"
    isChecked={values.locked}
    value={" "}
    defaultChecked={false}
    onChange={(event) => {
      setFieldValue("locked", event.currentTarget.checked);
    }}
  ></Checkbox>
</Text>: " "}
```

32. ábra Felhasználó blokkolása adminok számára

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

### 3.1.13 News

#### 3.1.13.1 NewsListProvider



33. ábra NewsListProvider komponenes

Három komponenes megjelenítésért felelős. Az első komponenes jelzi a redux stateben tárolt typeId alapján milyen kategóriájú híreket jelenít 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, hard kódolt 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 k nt egy News elemekt tartalmaz  t mb t v r el amit megjelen t NewsItem-ek form j ban.

### 3.1.13.3 NewsListItem

Az egyes h rek k rty k form j ban jelennek meg a f  oldalon. Ezen a k rty n jelen tem meg a h rhez tartoz  k pet, c met, alc met, szerz t,  s az aktu lis kedvel sek sz m t.

A komponens v z t a chakraUI b l sz rmaz  Card tag adja. A komponensemnek propertiben adom  t a news objektumot, itt nem t rt nik h l zati kommunik ci .

### 3.1.13.4 NewsDescriptionProvider

A kor bbi verzi kban t ls gosan kiakartam haszn lni a global stat-et  gy enn l a komponenesn l a NewsItem-re kattintva be ll tott h rt k rtem le a stateb l.

Viszont  gy, ha a felhaszn l  friss tett valami okb l a b ng sz j t  ,elveszett  a h r.

El sz r az App.tsx fileban m dos tottam a komponenshez k t tt path templat-et

```
<Route path="/news" element={<NewsDescProvider />} />  
<Route path="/news/:id" element={<NewsDescProvider />} />
```

34.  bra Param ter  tad s path-ban

A newsItem-re kattintva m r nem a h rt t lt m be a state-be hanem a path-hoz f z m a h r id-j t,  s erre a dinamikus path-ra navig lok el.

A react-router-dom b l BrowserRouter kontextus felismeri a path  s a hozz  k t tt komponens bet lti a megfelel  helyre.

Sajnos ez a kontextus minden olyan komponenset is  jra renderel amely tartalmaz a navig ci hoz sz ks ges useNavigate() hookot is.

Ilyen komponensek a Navbar, Footer, mivel ezek dinamikusak nem helyezhetem a BrowserRouter kontextuson k v ltre.

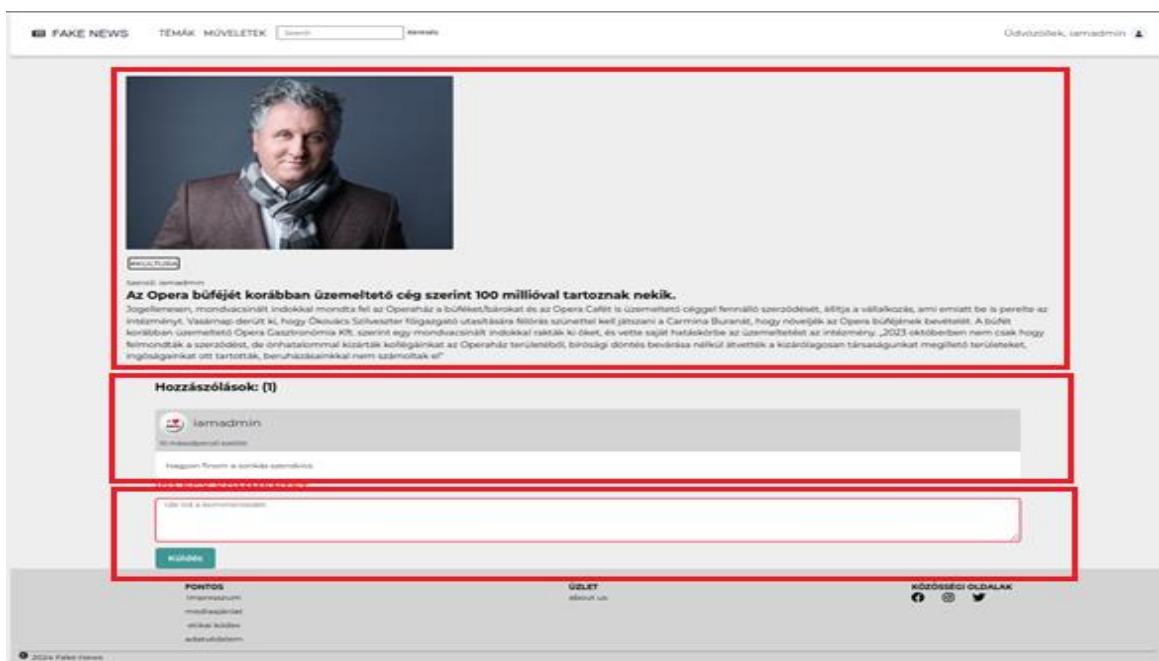
Egyik megold s lehetne, ha elhagyn m a routingot  s a regisztr ci s, illetve bejelentkez si Modal mint j ra, az egyes  ,oldalak  a k zponti statben t r lt flag-ek f ggv ny ben jelen ten m meg. R viden ki-be kapcsolgatn m a l that s gukat. Ezzel azonban azt a funkci t vesz ten m el hogy az egyes h rekhez tartoz  oldal pathokat a felhaszn l 

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.

```
const { id } = useParams<"id">();
```

35. ábra Paraméterben átadott érték használata

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 komponenseket.



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

Mivel a szerver lekérdezés aszinkron 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 jeleníti 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 komponenset, amely a porertiben kapott dátum alapján visszaadja 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 komponenset így egy beviteli mezőben leütött karakter miatt akár több tíz MyCommen komponenes is renderelődni e kellett.

### 3.1.15 Store directory

#### 3.1.15.1 Store.ts

Ebben fileban konfiguráljuk a redux-toolkit állapot tároló komponensét a

```
export const store=configureStore({
  reducer:appReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat([newsMiddleware])
})
```

storte-t. A konfiguráláshoz többek között meg kell adnunk egy reducert.

36. ábra Redux store konfiguráció

A reducer egy függvény 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 használok a combineReducer függvényt amellyel összevonhatom a több reducert egyetlen objektumba és a store definiálásnál ezt adom át.

Az 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* használatakor egy API konfigurálására, az automatikusan generál egy middleware-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 objektok 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.

```

export const authUserSlice = createSlice({
  name: "authUser",
  initialState,
  reducers: {
    setToken: (state: AuthUserState, action: PayloadAction<string>) => {
      state.token = action.payload;
    },
    setUser: (
      state: AuthUserState,
      action: PayloadAction<User | undefined | null>
    ) => {
      action.payload && (state.user = { ...action.payload });
    },
    outUser: (state: AuthUserState) => {
      state.user = undefined;
      state.token = "";
    },
  },
});

```

37. ábra User slice definiálása

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

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

A felhasználó függvényben ezekre a nevekre hivatkozva a useDispatch hookon keresztül „lőjük” el az actionokat.

```

export const { setUser, outUser, setToken } = authUserSlice.actions;
export const authUserReducer = authUserSlice.reducer;
export const authUserPath = authUserSlice.name;

export const selectOnlineUser = (state: RootState) => state.authUser.user;

```

38. ábra Online User lekérdezése stateből

Az *authUserReducer* és *authUserPath*-t a **store.ts** fileban használjuk fel, amelyben magát a redux store-t állítom be. Itt egy combineReducers 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 állítom be,

-*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 autorizáció 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.

```
as BaseQueryFn<string | FetchArgs, unknown, customError, {}>,
```

39. ábra Query kiegészítése saját hibakezeléssel

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.

Az 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 értékként, második paraméterben milyen adatot küldünk.

*url:* a server endpoint címe  
SERVERHOST egy .env filben előre definiált környezeti változó

ezt az alkalmazás bármely komponenséből elérhetjük.

*method:* a request típusa

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

Ezek az adatok a böngésző cach memóriájába kerülnek.

```
baseQuery: fetchBaseQuery({
  baseUrl: "",
  prepareHeaders: (headers) => {
    const token = storage.getItem("_auth");

    if (token) {
      headers.set("authorization", `Bearer ${token}`);
    }
    return headers;
  },
}) as BaseQueryFn<string | FetchArgs, unknown, customError, {}>,
tagTypes: [newsTag, userTag],
```

41. ábra Query konfiguráció

```
getUsers: builder.query<User[], void>({
  query: () => ({
    url: `http://${SERVERHOST}:8080/users`,
    method: "GET",
  }),
  providesTags: (result?: User[]) => {
    return result && Array.isArray(result)
      ? [
          ...result.map(({ id }) => ({ type: userTag, id })),
        ]
      : [{ type: userTag, id: "LIST" }];
  },
}),
```

41. ábra getUser Query definiálása

### 3.1.15.5 Mutation

A queryhez hasonlóan megadjuk milyen adatot küldünk és milyen várunk.

A request *body*ban JSON objektumként küldöm el az adatokat

```
updateUser: builder.mutation<void, updateParam>({
  query: ({ user, image }) => ({
    url: `http://${SERVERHOST}:8080/users`,
    method: "PUT",
    body: { usersDTO: user, image: image },
  }),
  invalidatesTags: (_resut, error, {user})=>{
    return [{ type: userTag, id: user.id }],
  },
}),
```

*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 tartalmaz 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 ellátott 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

Név	Metódus	Állapot	Típus	Méret
□ 15	OPTIONS	200	preflight	0 B
□ 15	DELETE + <u>Előzetes</u>	204	fetch	425 B
🔗 undefined	GET	200	fetch	1.2 MB
✖ aHR0cHM6Ly9jbXNjZG4...	GET	200	jpeg	(lemezgyorsítótár)

42. ábra Hálózati kommunikáció

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 rendszerünkötől 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 visszavigálunk 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 függvény legfelső szintjén használhatóak, 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 hookokkal a kódunk újra felhasználható, 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 useSelector 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 *paraméter* bármelyik attribútuma megváltozna a Redux-Toolkit újra elvégzi a lekérdezést.

```
export const useGetNewsByType = (params: GetRequestParamsForNewsQuery) => {
  const { isLoading, isFetching, error } = useGetNewsByTypeQuery(params);
  const select = newsApi.endpoints.getNewsByType.select(params);
  const { data } = useSelector(select);
  const news = data?.newsList
  const lastSide=data?.lastSide
  return {
    isLoading,
    isFetching,
    error,
    news,
    lastSide,
  };
};
```

#### 43. ábra Hírek lekérdezése hookon keresztül

Lényegében az toolkitre bízunk 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 felhasználó függvényben egyszerűbb legyen használni

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

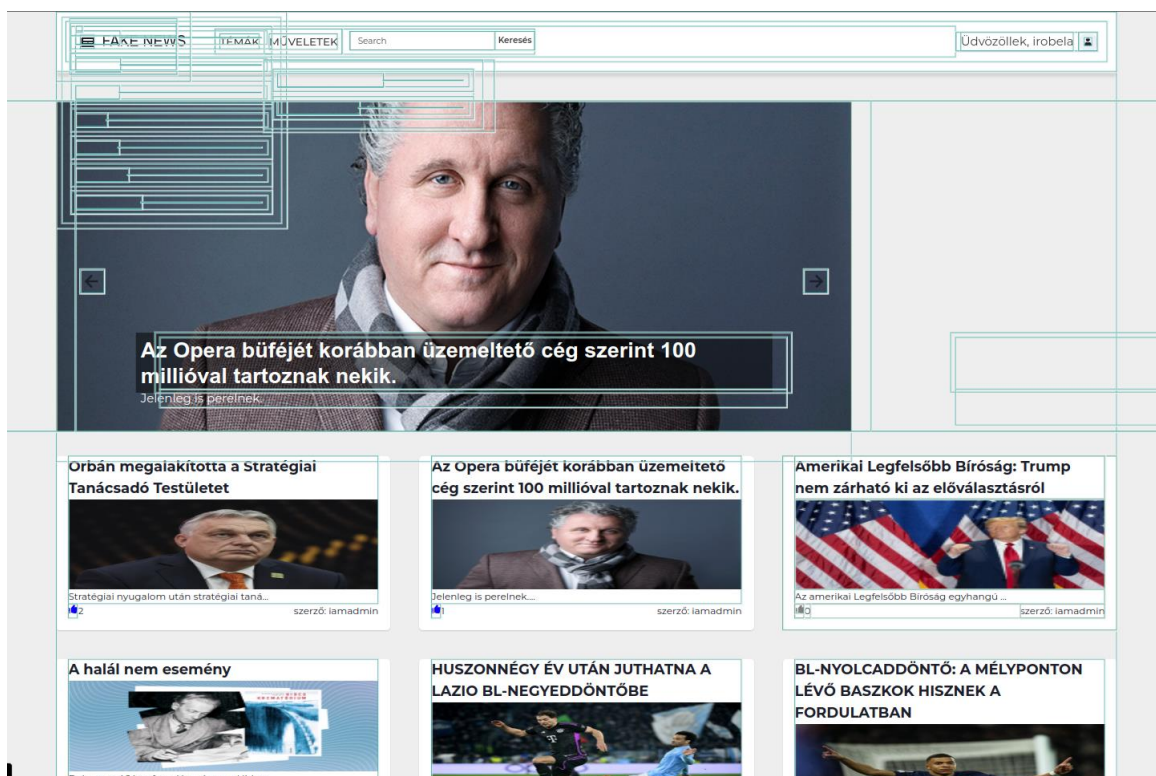
### 3.1.17 Utils directory

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* szöveggént 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 újra rajzolást (renderelést) végezze az oldalon. A react automatikusan végzi a komponens újra renderelését amikor a komponens props vagy state változik.

Az alábbi kép egy refaktorálás előtti renderelés látható, amit a „Like” gombra kattintás idézett elő.



44. ábra Hibás renderelés

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

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

Azonban ennek kötelezően tartalmazni kell egy `to=""` propriet, ami alapesetben vissza navigál a gyöker oldalra. Gyakorlatilag a kattintással újra töltöttem az oldal komponenseit, ami így visszatekintve megmagyarázhatatlan számomra, hogy gondoltam ez így valid megoldás lehet, de itt még csak ismerkedtem a react-el

```
<Box display={"inline-flex"}>
  <Link to="/">
    <FaThumbsUp
      className="me-2 fs-5 "
      style={{
        news.likes?.find((item) => item.id === user?.id)
          ? { color: "blue" }
          : { color: "gray" }
      }}
      You, 2 weeks ago • Merge remote-tracking branch 'origin/master'
      onClick={() => {
        user ? userDidLike(news) : window.confirm("jelentkezz be!");
      }}
    />
  </Link>
  {news.likes!.length}
</Box>
```

45. ábra *LikeButton* definiálása

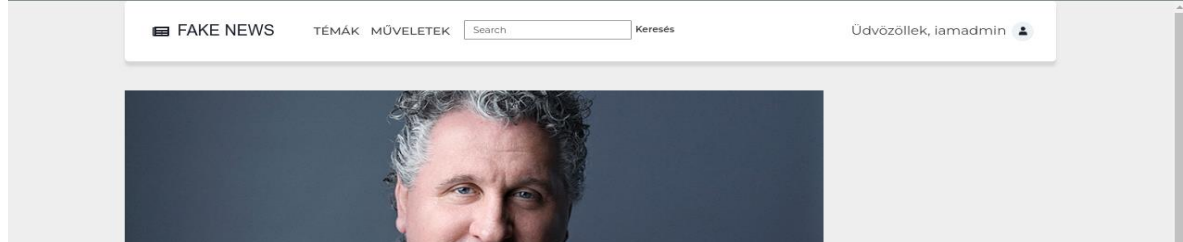
A *@chakra-ui/react* könyvtár tartalmaz egy másik *Link* tag-et viszont ebben a szakaszban még a *NewsListItem* komponensben volt a *LikeButton* komponens és szükség volt mind a két típusú *Link* tag-re Ilyen esetben lehetőség van alias nevet megadni a tag-nek és azzal hivatkozni rá

A másik hiba, hogy közvetlenül a *NewsListItem* nevű komponensen belül valósítottam meg a *LikeButton* gombra kattintás logikáját. Ahhoz, hogy ez megfelelően működjünk a

```
Link as ChakraLink } from "@chakra-ui/react";
```

46. ábra Alias név használata

*NewsListItem*-en belül definiálnom kellett lokális state változókat, amelyek megváltozása az egész *NewsListItem* újra rajzolását eredményezte.



```
const userDidLike = (news: News) => {
  setNews({
    ...news,
    likes: news.likes?.find((item) => item.id === user?.id)
  });

  setUser({
    ...user!,
    likednews: user?.likednews.find((item) => item.id === news.id)
      ? user!.likednews.filter((item) => item.id !== news.id)
      : user!.likednews.concat(news),
  });

  if (user) {
    addLike({ user: user, news: serializNews(news) } as Like);
  } else {
    window.confirm("jelentkezz be");
  }
};
```

47. ábra NewsListItem renderelődés

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

```
const [user, setUser] = useState<User | undefined>(userInState);
```

A `LikeButton` kiszervezése után már megfelelően renderelődik a komponens

Ez komponens felelős továbbá a „like” elküldésért.

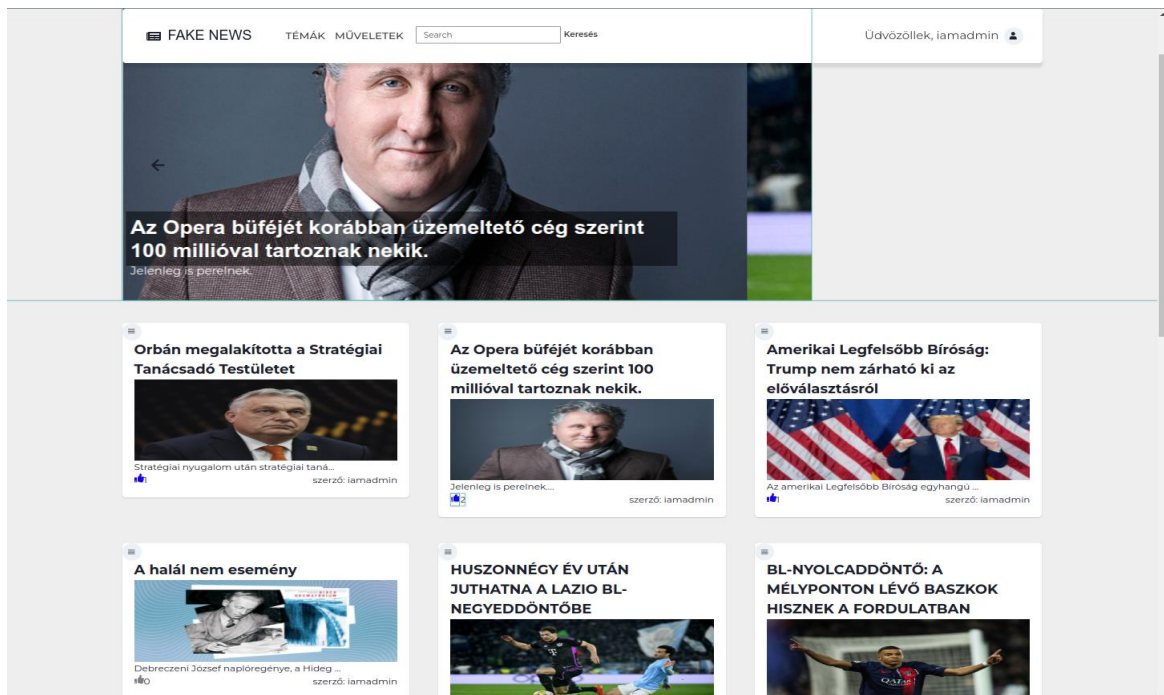
Ha éppen nincs bejelentkezett felhasználó egy alert felugró ablakban jelezzük a felhasználónak. Ezt a funkciót később bővíthetjük és egy Modal ban tudjuk dizájnosabban jelezni ezt.

```
onClick={() => {
  user ? userDidLike(news, user) : window.alert("jelentkezz be!");
}}
```

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



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



48. ábra Elvárt renderelés a komponens kattintás után

```
<LikeButton id={stateId} onLineUser={userInState} />
```

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

```
useEffect(()=>{setUser(onLineUser)},[onLineUser])
```

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



```

const userDidLike = useCallback(
  (news: News, user: User) => {
    setNews({
      ...news,
      likes: news.likes?.find((item) => item.id === user?.id)
        ? news.likes.filter((item) => item.id !== user?.id)
        : news.likes?.concat(user),
    });
    user &&
      setUser({
        ...user,
        likednews: user?.likednews?.find((item) => item.id === news.id)
          ? user.likednews.filter((item) => item.id !== news.id)
          : user.likednews.concat(news),
      });
    addLike({ user: user, news: serializNews(news) } as Like);
  },
  []
);

```

A setNews függvényben először 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 tömbhöz.

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

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

```

style={
  news?.likes?.find((item) => item.id === user?.id)
    ? { color: "blue" }
    : { color: "gray" }
}

```

## 3.2. Backend

A szerveroldali alkalmazást java nyelven készítettem springboot keretrendszerrel 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 biztonságaért 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.

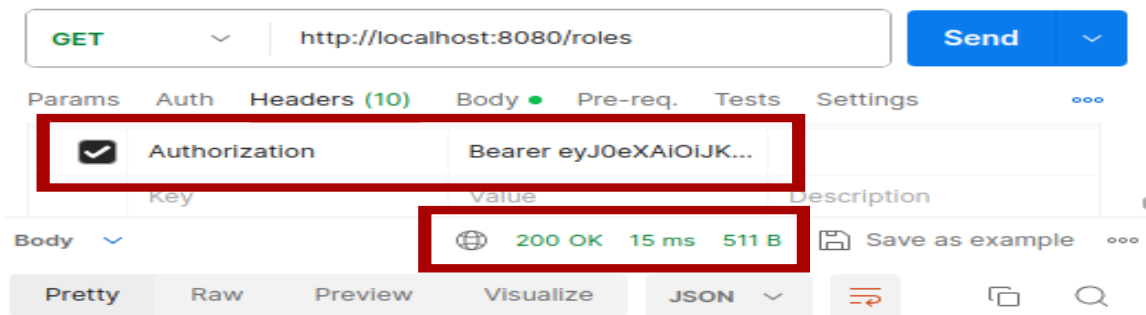
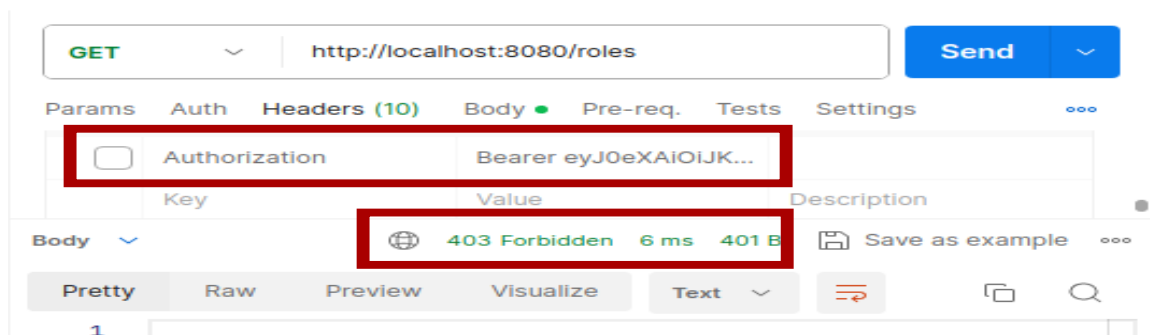
```
http.csrf().disable() HttpSecurity
    .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS) SessionManagementCo
    .and() HttpSecurity
    .authorizeRequests() ExpressionInterceptUrlRegistry
    .antMatchers(AUTH_WHITELIST).permitAll()
    .antMatchers( ...antPatterns: "/uploads/**").permitAll()
    .antMatchers( ...antPatterns: "/authentication").permitAll()
    .antMatchers( ...antPatterns: "/users/delete/**").hasAnyAuthority( ...authorities: "ADMIN")
    .antMatchers( ...antPatterns: "/users/**").permitAll()
    .antMatchers( ...antPatterns: "/news/delete/**").hasAnyAuthority( ...authorities: "ADMIN")
    .antMatchers( ...antPatterns: "/news/**").permitAll()
    .antMatchers( ...antPatterns: "/comment/**").hasAnyAuthority( ...authorities: "ADMIN", "USER", "WRITER")
    .anyRequest().authenticated()
    .and() HttpSecurity
    .addFilterBefore(jwtAuthorizationFilter, UsernamePasswordAuthenticationFilter.class)
    .cors(Customizer.withDefaults());
```

Az `AUTH_WHITELIST` tömbre a swagger-ui miatt volt szükségem enélkül a security megtagadja a swagger 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 jogosultságok 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.

```
private void doAuthorization(HttpServletRequest request) {
    String authorizationHeader = request.getHeader(HttpHeaders.AUTHORIZATION);
    if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {
        String jwt = authorizationHeader.split(" ")[1];
        String email = jwtUtil.verifyAndDecodeToken(jwt);
        UserDetails userDetails = userDetailsService.loadUserByUsername(email);
        UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(
            userDetails, null, userDetails.getAuthorities()
        );
        SecurityContextHolder.getContext().setAuthentication(authenticationToken);
    }
}
```

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

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

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

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

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

Ehhez még konfigurálnom kellett a *CorsConfig.java*

```
public WebMvcConfigurer configure() {
    return addCorsMappings(registry) -> {
        registry.addMapping(pathPattern: "/*").allowedHeaders("*").allowedMethods("*");
    };
}
```

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 kontroller rétegbe kerülnek. Ennek a rétegnek „csak” annyi a feladata hogy fogadja a requesteket és visszaküldeje a megfelelő responst, mindemellett ahogy a neve is mutatja továbbítja 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 következő osztályokat:

#### 3.2.2.1 AuthenticationController:

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

```
.antMatchers( ...antPatterns: "/authentication/**").permitAll()
```

Ha a request bodyban kapott username és password alapján sikerült a felhasználót 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 `@ConfigurationProperties(prefix = "jwt")` annotáció hatására a springboot megkeresi a *jwt* -bel kezdődő blokkot pontosabban block-mapping-nek nevezett részt az *application.yml* fileban

#### 3.2.2.2 UsersController ,NewsController:

Ugyan azokat a CRUD requesteket kezelik

#### 3.2.2.3.RolesController

Jelenlegi verzióban nem használom.

### 3.2.2.4 CommentController

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.

```
@Override
public UsersDTO update(UserDTOForUpdate userDTOForUpdate) {
    UsersDTO usersDTO = userDTOForUpdate.getUsersDTO();
    String image= userDTOForUpdate.getImage();
    // képküldés
    if (userDTOForUpdate.getImage().isEmpty()){
        usersDTO.setImagePath(IMAGE_PATH);
    }else {
        String oldImagePath = usersDTO.getImagePath();
        if (usersDTO.getImagePath().equals(IMAGE_PATH)) {
            usersDTO.setImagePath(imageService.add(image));
        } else {
            imageService.delete(oldImagePath);
            usersDTO.setImagePath(imageService.add(image));
        }
    }
    return update(usersDTO);
}
```

A bejövő Image objektum alapján eldönti az algoritmus, hogy van-e nem alapértelmezett kép beállítva profilképként, 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 általában származtatottjai egy nagyobb kiterjesztésű Entitynek. Eltudjuk dönteni mennyi attribútumra 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 service 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* fájlban egyszerű sablon utasítás sorozat. Konfigurálni ezt az *application.properties* fájlban 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

```
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=fakenewsemail2024@gmail.com
spring.mail.password=hnus xkut yvuy tlnh
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

49. ábra Email fiók konfiguráció

### 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 kontroller 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 *JpaRepository*t ezzel az egyszerű előre definiált lekérdezések mellett 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 futtatott 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 Entyitknek nevezzük. A táblák közötti kapcsolatok kialakításához ezeknek az osztályoknak a megfelelő attribútumait kell a kapcsolatot jelző annotációval ellátnunk: *@ManyToOne*, *@OneToMany*, *@ManyToMany* az annotációt mind a két osztályban jelölnünk kell.

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.

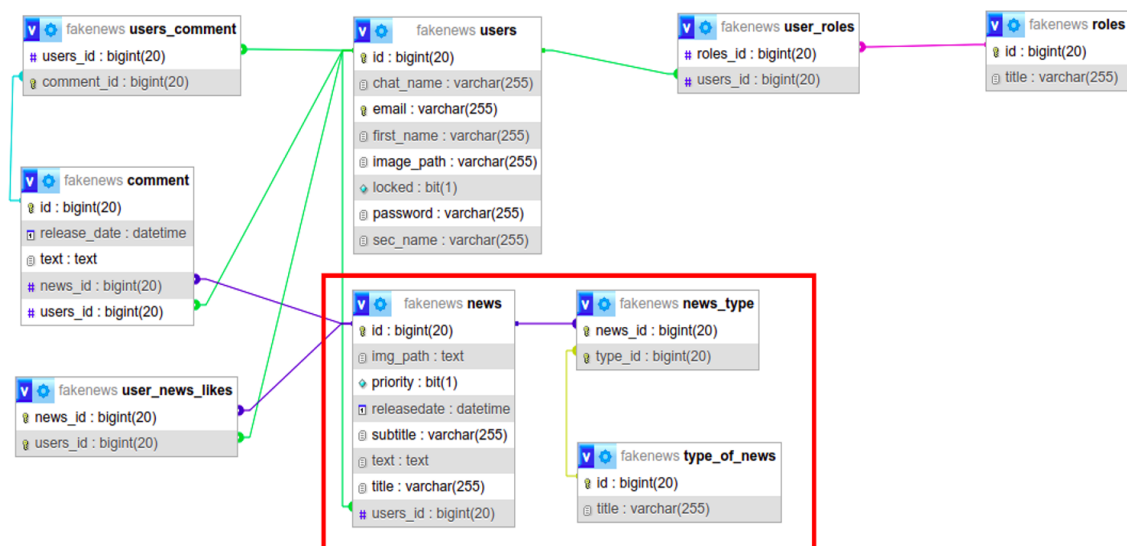


```

@ManyToOne
@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> címen érjük el több funkció mellett itt kapunk egy tervezőnézetet is ami láttatja a tábláink közötti kapcsolatokat.



50. ábra Xampp tervező nézet

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

```
@Query(value= "select * from news join news_type on news.id = news_type.news_id " +  
    "where news_type.type_id = :id AND news.title LIKE %:search% ", nativeQuery = true)  
Optional<List<News>> getNewsBySearchAndId(@Param("id")Long id, @Param("search") String search);
```

51. ábra Natív SQL lekérdezés

### 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 író, 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 alaphoz USER, a további jogokat adatbázis oldalról lehet kiosztani. WRITER jogú felhasználó közzé tehet 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 ériék és ebbe nem fér bele a töltési idő, ami ezt megszakítaná. Az én alkalmazásom nyilván összehasonlíthatatlan 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)

## 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 visszaneztem 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 javascript 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 React-val 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 ilyes 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 pálya kezdő fullstack fejlesztőként elhelyezkednem az itt megszerzett tudásommal.

## 7. IRODALMI JEGYZÉK

## 8. MELLÉKLET

### 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](#)-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 létre 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.

A repository pull után:

1 lépés A repository gyökér könyvtárában lévő fakenews.sql adatbázist importáljuk az xamppba.

2. lépés Tetszőleges fejlesztői környezetben megnyitni a frontend és a backend mappákat. A Git-re a projekteket dependenciák nélkül töltjük fel ezért a futtatás előtt le kell töltenünk az aktuális gépre. Ehhez frontend mappából ki kell adnunk a `npm install` és `npm build` parancsokat majd az

```
npm run build
```

**FONTOS**, hogy az adatbázis kapcsolat megfelelően legyen konfigurálva a backend application.properties file, ha hoston futtatva

```
spring.datasource.url=jdbc:mysql://localhost:3306/fakenews
spring.datasource.username=${MYSQL_USER:root}
spring.datasource.password=${MYSQL_PASSWORD:}
```

52. ábra Host config

Dockert használva

```
spring.datasource.url=jdbc:mysql:// 172.19.0.2:3306/fakenews
spring.datasource.username=${MYSQL_USER:root}
spring.datasource.password=${MYSQL_PASSWORD:rootpassword}
```

53. ábra Docker config

## 10.2 Docker

Azonban annak, aki csak kizsértené próbálni az alkalmazást elég a [Docker](#) nevű konténerizációs program megléte a gépen. Telepítés után a `docker -v` paranccsal tudjuk ellenőrizni sikeres volt-e a telepítés.

1. lépés: Az [alkalmazás](#) v.0.5 branch letöltése után a backend mappába navigálva a parancssorból adjuk ki az `mvn clean install` parancsot, ez letölti a függőségeket és elkészíti a /target mappában az alkalmazás futtatható formáját, majd a `docker build -t backend .`, (-t kapcsoló jelöli hogy elnevezzük az imaget backend néven, a . {pont} jelöli, hogy az aktuális mappában található a leíró Dockerfile) parancsot, ezzel létrehozuk a backendet reprezentáló imaget.

2. lépés: Ezután a FolderForMySql mappából hasonló parancsot `docker build -t database .` ez lesz az adatbázis képe.

3. lépés: a frontend mappából ki adjuk a harmadik kép készítésének parancsát `docker build -t frontend .`

4. lépés: Maradva ebben a mappában a `docker images` paranccsal ellenőrizhetjük, hogy mind a három kép elkészült-e

5. lépés Szintén ebben a mappában helyeztem el a docker-compose.yml filet amely a három konténer indítását végzi el a `docker-compose up` paranccsal.

6. lépés A végül a <http://localhost:3000> oldalon elérhetjük az alkalmazást.

Felhasználók és jelszavak:

[iamadmin@gmail.com](#) - admin1234,

[iobela@gmail.com](#) - iobela1234

[iokalman@gmail.com](#) - iokalman1234

[iroAttila@gmail.com](#) - iroattila

