

Java backend Javascript frontend alapú webalkalmazás

Hódi Szilárd

Programtervező informatikus BSc

Konzulens: Dr. Vályi Sándor

**2024**

**Tartalomjegyzék:**

Bevezetés

témaválasztás  
 alkalmazás célja  
Technológiai háttér  
 felhasznált programok

visual studio

intelij idea

github

chrome extension

postman

swagger

xampp

Frontend

1. Navbar 9.

1.1. Icon 9.  
 1.2. Menuk 9.

1.2.1.Téma 9.  
 1.2.2. Műveletek 10.

1.3. Kereséséi mező 11.  
 1.4. Bejelentkezés/Regisztráció 12.  
 1.4.1.Regisztáció 13.

1.4.2.Bejelentkezés 16.

2.User Details 18.

3.Users List 19.

4.Új Cikk hozzáadása 21.  
 5.Carousel komponens 22.

Backend  
 java SpringBoot

Fontosabb megvalósítót műveletek a program elemzés

Tovább fejlesztési lehetőségek

Konzekvencia

**Téma választás.**  
Fiatal korom óta foglalkozom a programozással hobbi szinten több programozási nyelvet is megismertem különböző mélységekben.  
Eddig még nem volt olyan területe az szakmának amely különösöbben megfogott volna. Azonban ezzel a projektek sikerült eldöntenem milyen irányba szeretnék tovább haladni.

A fullstack programozók amolyan svájci bicskák akik mind a frontend minda a backend fejlesztésben részt tudnak venni, illetve önállóan is képesek megvalósítani.

Alkalmazás célja:

Egy fiktív hírportál frontend és backendjét valósítottam meg.  
Célom ezzel felkészülni, illetve megalapozni a további szpecizciódási vonalamat. Készíteni egy referencia munkát amelyben összefoglalom tudásom jelelegi színtjét, egy későbbi álláshoz is.

**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, és a TypeScript, mint statikus típusellenőrzést támogató kiterjesztése is terjedt. JavaScript széles körben támogatott, és a fejlesztők a front-endtől a back-endig 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.

Megírt javascriptkódunkat bárhová, akár hivatkozás ként is elhejezhetjük, azomban felhasználói élmény javítása érdekében ajánlatosabb a body végébe rakni, a böngészó automatikusan felismeri és futtatni fogja.

**TypeScript** egy kiterjesztése a JavaScript-nek TypeScriptet fejlesztettek ki annak érdekében, hogy kiterjessze a JavaScript nyelvet olyan funkciókkal, amelyek lehetővé teszik a típusbiztosabb 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.

**React** egy nyíltforrású felhasználói felület készírésére használt javascript könyvtár. Melyet a Facebook fejlesztett ki 2011ben és tett nyíltforrásuvá 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 feladatomban.

**Mi az a SPA?**

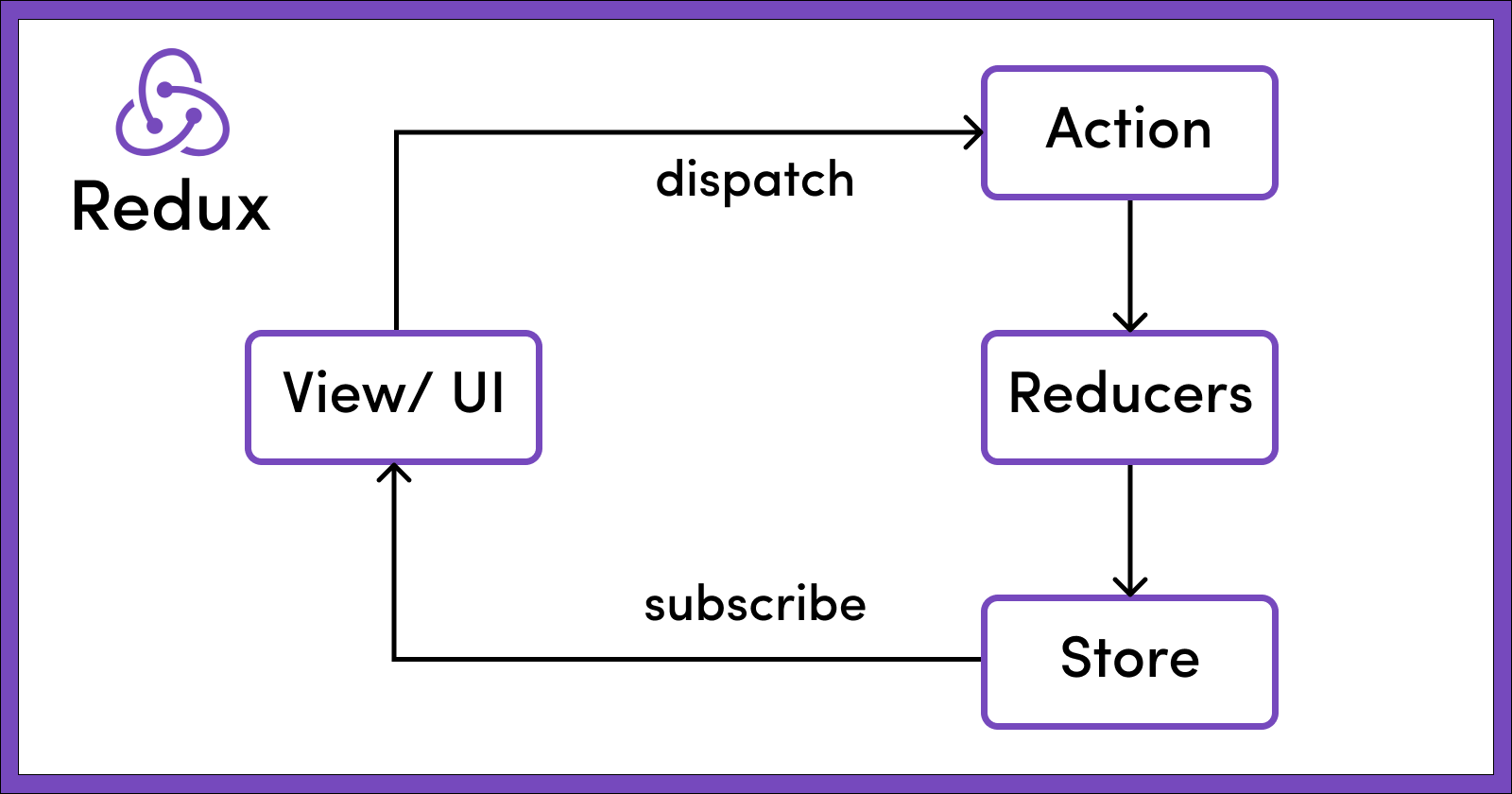
A Single Page Application nem az egész oldalt ölti újra az egyes felhasználói akicók hatására, hanem a DOM fát manipulálva csak annak részei frissülnek.

Ezáltal az ilyen alkamazások sokkal gyorsabbak és jobb felhasználói élmény nyújtanak.

Hátrányuk hogy a böngészőben engedélyezni kell a futtatáshoz szükséges JavaScriptet. Tovább, hogy jobban kivannak téve a hacker támadásoknak és nem keresőbarátok.

**Redux**

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 felhasznóli felületről érkező actionok hatására reducereken keresztül változtat meg. Fontos hogy a state csak egyirányu manipulációra használható.



**React rooter:**

Az alkalmazás oldalai közötti navigációra használt könyvtár

**Java:**

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

**Spring boot:**

A Spring keretrendszerre épülő projek mejnek feladata megkönnyíteni a java projektek készítését.

**Maven.**

Nyílt forrásu projektkezelő és buildelő rendszer mejnek segítségével könnyen lehet kezelni az alkalmazás függőségeit.

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

**A Maven repository**

A 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.

**Mi az API:**

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

Felhasználásuk szerint több típusuak lehetnek, mint a

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

-libary API-k egy könyvtár vagy repository kiszolgáló interface

-hardware API-k melyek külömböző fizikai eszközök kommuikációját segítik.

**Visual Studio Code**

VS Code egy Microsoft álltal fejlesztett eszköz, rengeteg program nyelv támogatásával és pluginnal a mik megkönnyítik a kódolást, ezt használtam a frontend elkészítéhez.

**Intelij IDEA**

JAVA és egyéb JVM alapú programok fejlesztésére használt fejlesztői környezet.

**Swager UI / Postman**

Ezt a két alkalmazást használtam a backend API endpoint jainak teszteléséhez.

SwaggerUI dependencia ként tudjuk megadi 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 kölönbőző requesteket. Spring security miatt configurálni kell az elérési utat, amíg ezt nem fixáltam a Postmant használtam.

**Postman**   
 Álltalános felhasználásra kialakított program.

**chrome extensions**

**w**ebalkalmazás révn a fejlesztéshez böngésző szükséges, a chrome stabil népszerű böngésző. Remek pluginokkal a reaktos alkalmazások teszteléséhez én a   
-Ract Developer Tools

-Redux DevTools

Előbbit kevésbé utóbbit folyamatosan a Redux state változásának figyekléséshez,

**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 készítésüekkel is.

Node.js egy szoftver platform és futtató környezet, amely lehetővé teszi JavaScript kód futtatását a számítógépen vagy szerveren kívül is, nem csak a böngészőben. Ezzel a platformmal lehetőség nyílik szerveroldali JavaScript alkalmazások fejlesztésére is.

**Xampp**

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

**ALKALMAZÁS INDÍTÁSA**

**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 csinál, hogy egy másolatot hoz létere a számítógépünkre a következő paranccsal

*git clone https://github.com/hodiszilard89/news\_web\_app.git*

A klónozott reposytoriban két mappa található a *client\_typescipt* és a *server*

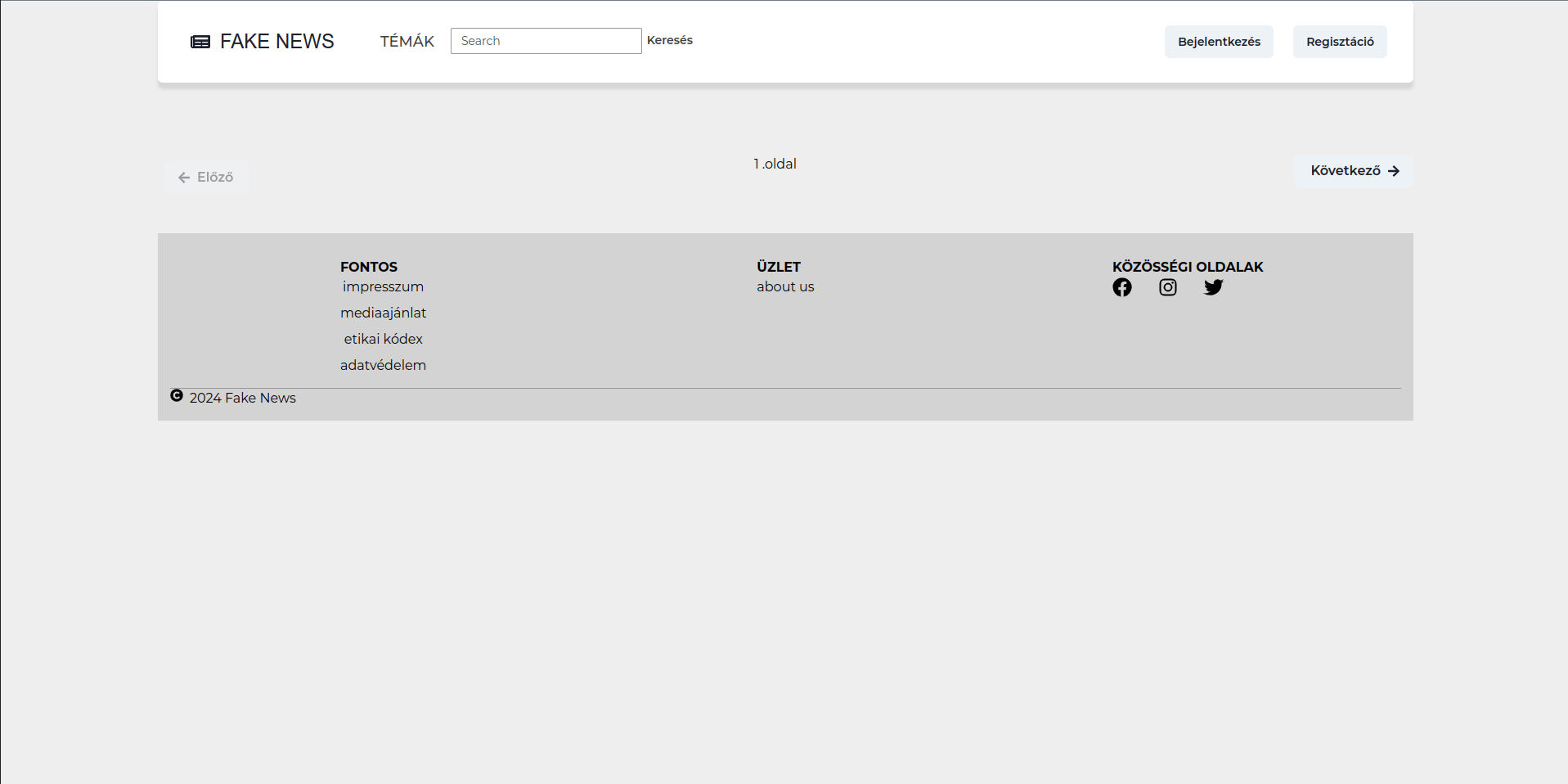
A git nem csak egy online tároló ahol megoszthatjuk illetve tárolhatjuk forrásainkat. Lokális verzió kezelésre is van lehetőségünk.   
Ha új futuret szeretnénk implementálni vagy valamit módosítani kódunkban. Egy új branch nyitásával klónozhatjuk projektünket amit szabadon módosíthatunk, nagyobb mértékben is, a változtatásokat pár kattintással vissza tudjuk állítani.

**Visual Studio Code**

Én a frontend fejlesztéséhez a [VS Code](https://code.visualstudio.com/download)-t használtam, ami egy ingyenesen letölthető fejlesztőkörnyezet, integrál Node Package Manager (npm) , és számos letölthető extension-el.

A program indítása után ki tudjuk tallózni a *client\_typescript* könyvtárat, itt a beépített terminálból ki kell adnunk az *npm run build*  parancsot, amely a *package.json* fájl alapján letölti az alkalmazás szükséges függőségeit és létrehozza a *node\_modules* könyvtárat, többek között ezt a könyvtárat ignoráljuk a Git-re történő feltöltésnél.

Ezek után ha kiadjuk az *npm start* parancsot a [http://localhost:3000](http://localhost:3000/) címen elindul a frontend alkalmazásunk.

****

A frontend oldalról előre hard kódolt komponens feliratokon kívül még nem jelenek meg semmi, mivel még nem kap válaszokat a szerver oldali alkalmazástól.

A Google Chrome böngészőt használtam fejlesztés közben, de más böngészőkben is tudjuk ellenőrizni a hálózati kommunikációt, ami rendkívül nagy segítség fejlesztés közben.

****

Http 500-as hibakóddal tér vissza, ami a szerveroldali hibát jelenti.

**IntelliJ IDEA**

A backend fejlesztéséhez az [IntelliJ IDEA Community Edition](https://www.jetbrains.com/idea/download/?section=linux) fejlesztői környezetet használtam. Az program java nyelven írt Mavennel csomagolt Springboot alkalmazás, ezért ha még nincsenek telepítve a fejlesztéshez szükségünk lesz a [Maven](https://maven.apache.org/download.cgi) projektmenedzsment programra és a [Java Development Kit (JDK](https://www.oracle.com/java/technologies/downloads/" \l "java17) 17) meglétére a gépünkön. 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.

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 .

**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.

**FRONTEND**

A visual studio Code telepítése után a terminálból kiadott

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

A Reakt egy javascript könyvtár ami a komponensek újra felhasználására épít egy szerű megtanulni. Projektemben a reduxal használtam együtt az alkalmazás állapot tárolására.

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 actionon jainak a leírása

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

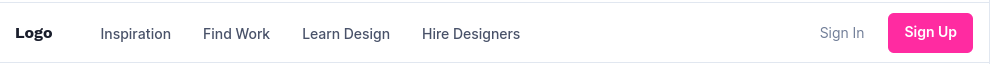
**-utils:** felhasznált segéd függvények

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

**Componenets**

**Navbar:**

A sablona 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

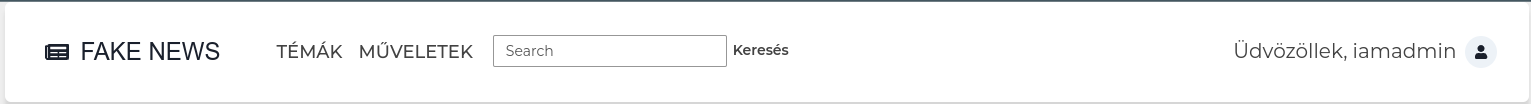


Bejelentkezés után a komponens nézete megváltozik aszerint hogy milyen

jogosultságai vannak a felhasználónak.

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

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



**Icon**:

Ez egy külön komponens, visszaállítja az esetleges keresési feltételeket.

A lekérdezett hírek témájt 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 implementljuk és a useDispatch() hookon keresztül tudjuk „ellőni” .

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

**Menuk**:

**Témák**:

kategória a backendről lekérdezett TypeDTO-t jeleníti meg

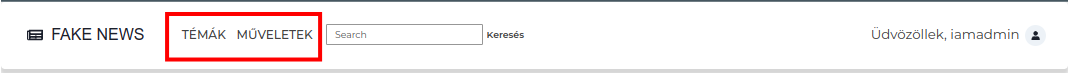
**Műveletek**:

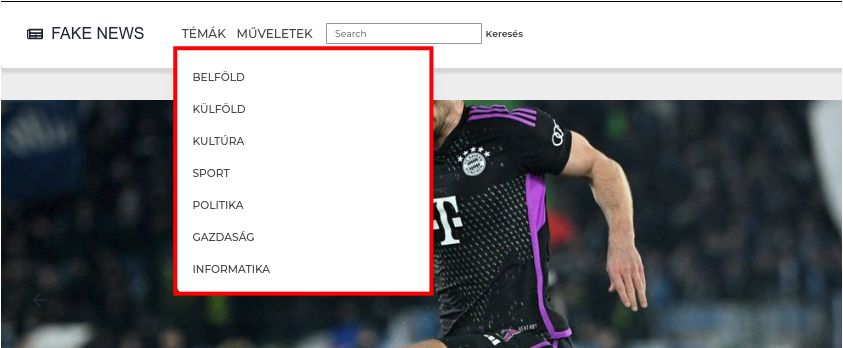
A navbarban hard kódolva adtam meg a két lehetőséget.

Mind két komponens tulajdonképen ugyan az bár a megjelenítendő elemek tulajdonságai mások. Míg a Témák komponens Type típusú elemeket jelenít meg ’id’ és ’tittle’ atributumokkal. Amelyek megfelelőek voltak a menu elemek value és címke értékeinek addig a Műveletek menu elemei title és path atributomokat tartalmaz. Ezért kiterjesztettem az alap Type típust és hozzáadtam egy opcionális path atributumot, 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

**Keresési mező**

****

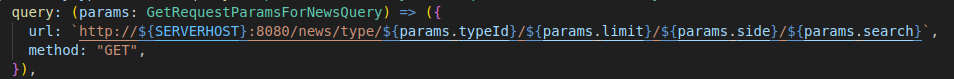
Ezzel az komponenssel tudunk szűrést végezni a hírek között, cím alapján.

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.

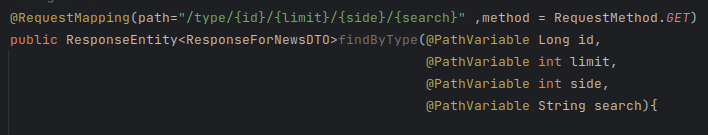
A hírek lekérdezését végző query paraméter listájához hozzáadtam egy *string* típusú *search* attribútumot. Mivel korábban már a globális statben volt tárolva az esetleges keresési string így azt a megfelelő helyen letudtam kérdezni. Erről részletesebben a store fejezetben.

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.



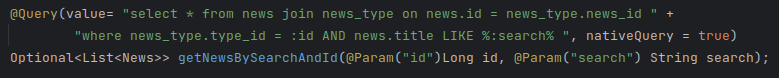
A backend endpointot is kiegészítettem a serach paraméterrel. A RequestMapping annotációval tudjuk megadni a path templatet, továbbá hogy egy GET metódusra figyelünk. *ResponseEntytit* küldünk vissza melynek egy saját *ResponseForNewsDTO* a neve, amely egy logikai értéket és egy hírek listát tartalmaz . 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 *id* paraméterben adjuk át a hír típusát a *limit*-ben az egy oldalon megjeleníteni kívánt híreket a *side* az aktuális oldal 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 *NewsContoler*-ben, ahol egy NewsDTO objektumot küldünk a szervernek.

Ennek a dolgozatnak a célja minél több módszer bemutatása.

A Controler átadjuk a paramétereket a Service rétegnek, itt állítom össze a *side* és *limit* értékek alapján a vissza küldött hír-ek tömbjét. Csupán a forrást kellet módosítanom.   
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* melyet 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.

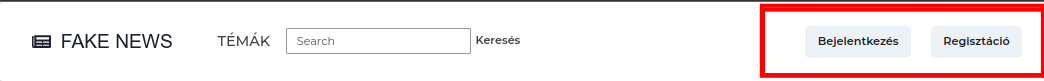
Visszatérési értéke egy *News* listát tartalmazó Optional adatszerkezet, ez azét hasznos mert ha a lekérdezés nem hoz eredményt könnyen tudjuk kezelni.

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

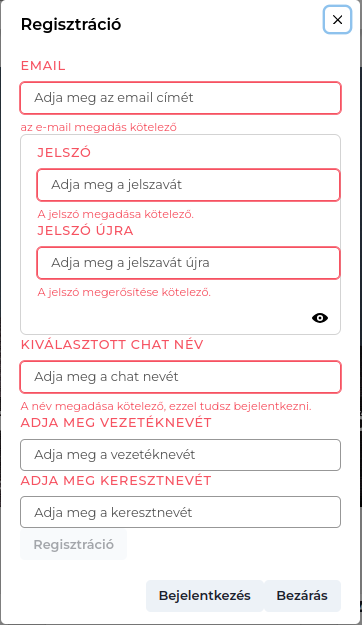


**Bejelentkezés/Regisztráció, Felhasználói adatok**

**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.

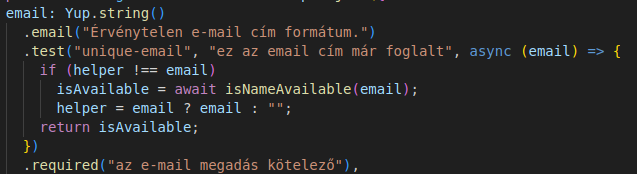
Az *isOpen* paraméterben logikai értéket adhatunk meg nyitva illetve zárva legyen a komponens. Ezt az értéket is a redux globális stateben tárolom.

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.

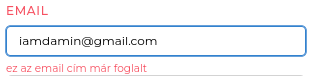
A *useFormban* megadhatjuk a validációs sémát, azonban még a benne megadott függvényeket a fieldekhez kell kötnünk így a korábban megírt függvény többször felhasználható akár más projektekben is.

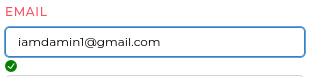
Ehhez két dolog kell a *FormControl* nyitó tagében megadjuk, mint *isInvalid* properti.

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.

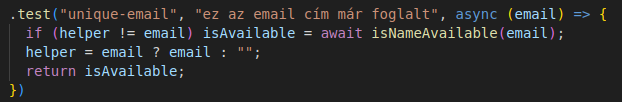




A input mező *onChange* függvényébe állítjuk be *form-*nak átadott objektum field értékét, ami ezáltal minden egyes karakter leütésnél frissül.

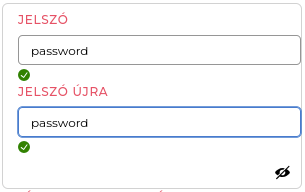


A jelenlegi verzióban a form a *useFormik*ban megadott *initialValues* értéket frissíti, minden egyes input mezőbe leütött karakter hozzá tesz illetve töröl valamilyen kulcs értékből. Ezért renderelődik mindig újra a form, ennek hatására a validációban használt api kérés az email cím field közvetett 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.

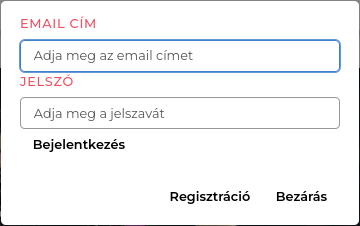
A *useFrom*-al handleSubmit függvényével kezelhetjük a *form* submit eseményét is. Ezt az eseményt én a komponensen belül kezelem, egy ”üres” user objektum kulcsait töltöm fel a *form* fieldjeinek értékével.

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.

**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.



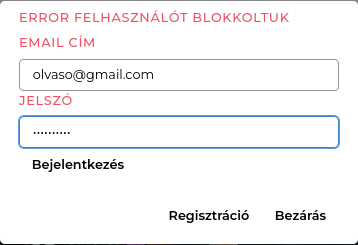
A fenti példa csak a szerver oldalon kiosztott ADMIN joggal rendelkező requesteknek engedi elérni a *news/delete* endpointot, a *news-t* mindenkinek ehhez nem szükséges autorizálni a kérést, bárki láthatja a híreket.

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.

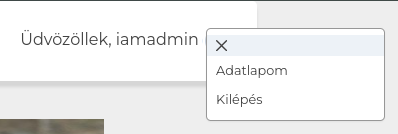
Az esetleges authorizációs hibát a szerveren állítjuk össze és küldjük visszaküldi a frontend felé, amit az inline jelenít meg.

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

**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.

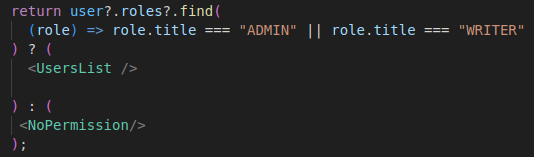
A felhasználók listájának a */users* végponton tekinthetjük meg. Az alkalmazásban a *react-router-dom* könyvtárat használom az oldalak közötti navigációra.



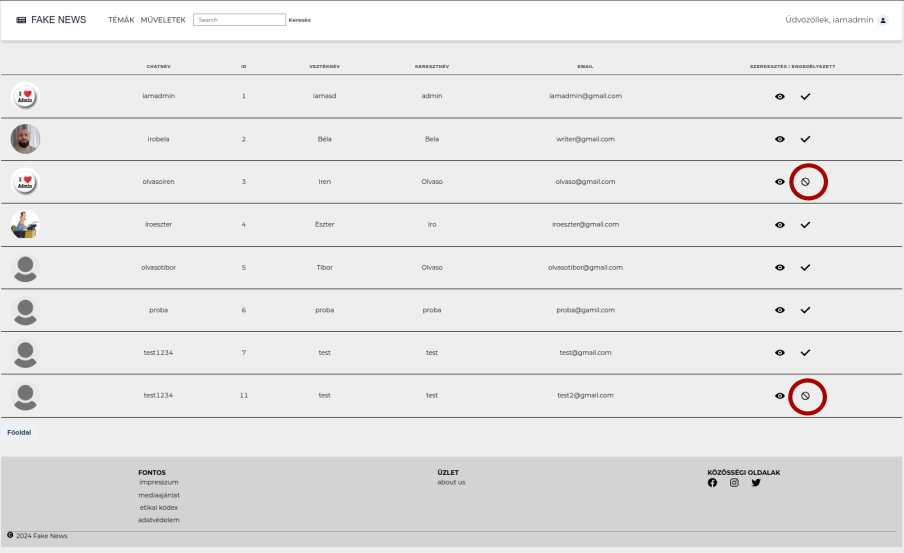
Szerver oldalról a /users API endpoint mindenkinek nyitott, hogy a regisztráció során letudjunk kérdezni adatokat. Ezért a listát kliensoldalról védjük, ami azt jelenti hogy egy provideren keresztül érjük el, ami vizsgálja hogy van-e autentikált felhasználónk jogosultsága látni a listát.

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

***KIKELL cserélni!!***



Listában láthatjuk a felhasználók adatait illetve ikonal jelezzük melyek vannak blokkolva



**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.



**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.

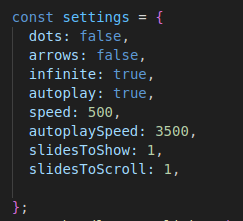
Ebben a providerben kérjük le a hírt a global state-ből és adom tovább *NewsEditor* komponenesnek még ugyan itt állítom össze az oldal-t, navbar footer -komponenseket. Erre azért van szükség, hogy ha szerkesztjük a formot, ne frissüljün az egész ”oldal” a state- és propertik miatt.

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.

**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()*

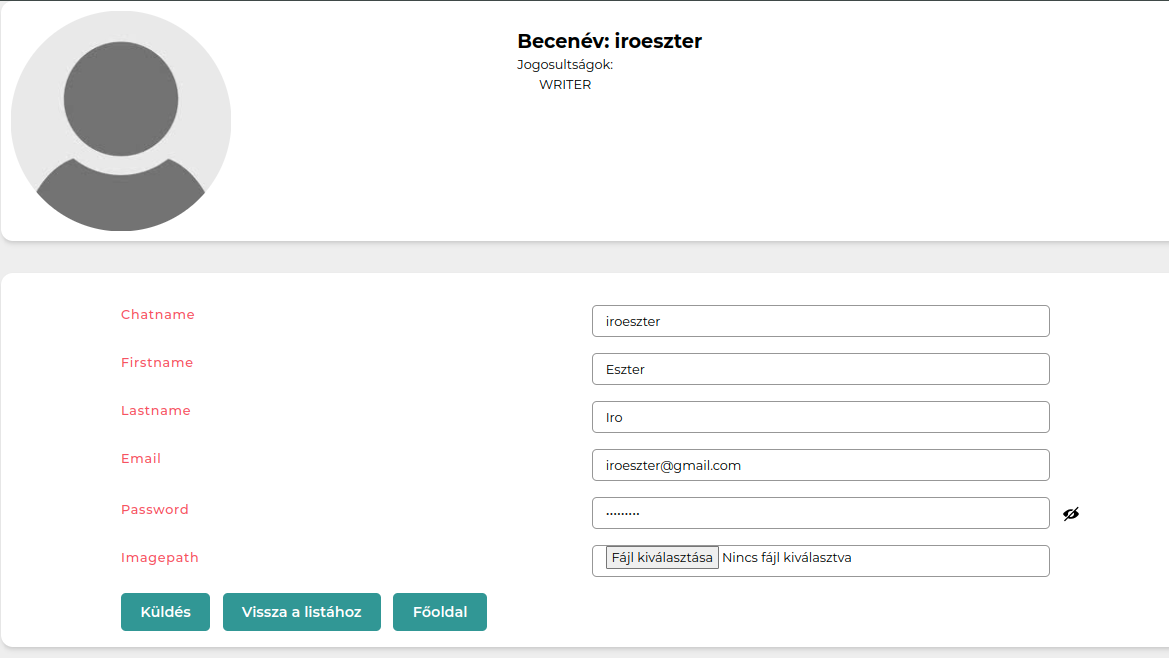
és a *slickNext()*  metódusokat, amiket fel tudok használni a saját gombok onClick eseményében, így irányíthatom a Slider komponenst.

****

**SINGLE USER PAGE DIRECTORY**

**User details**

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 regisztáció automatikusan megkapja.



A profilképet a kliensgépről tudjuk kiválasztani.

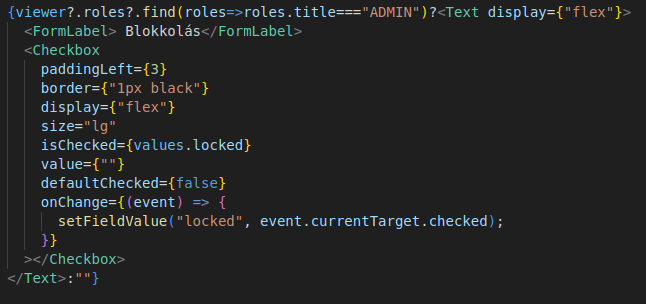
A további profil képek is a szerverre töltődnek fel úgy, hogy a korábbit törlik onnan.

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



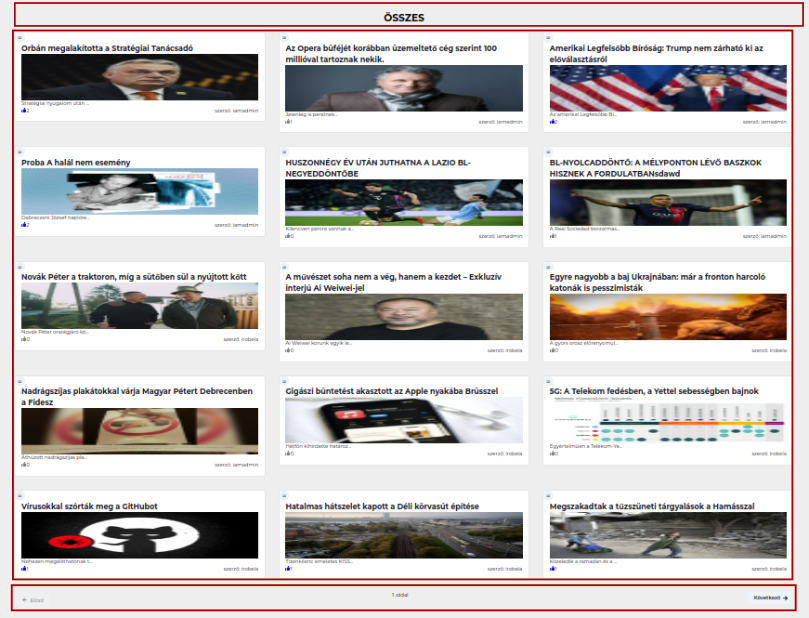
A küldés gombra kattintva indítjuk el a requestet, ezzel együtt vissza navigálunk a főoldalra.

**NEW LIST DIRECTORY**

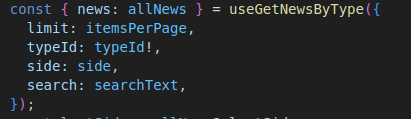
**NewsListProvider**

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.

**NewsList**

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

**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.

**SINGLE NEWS DIRECTORY**

**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

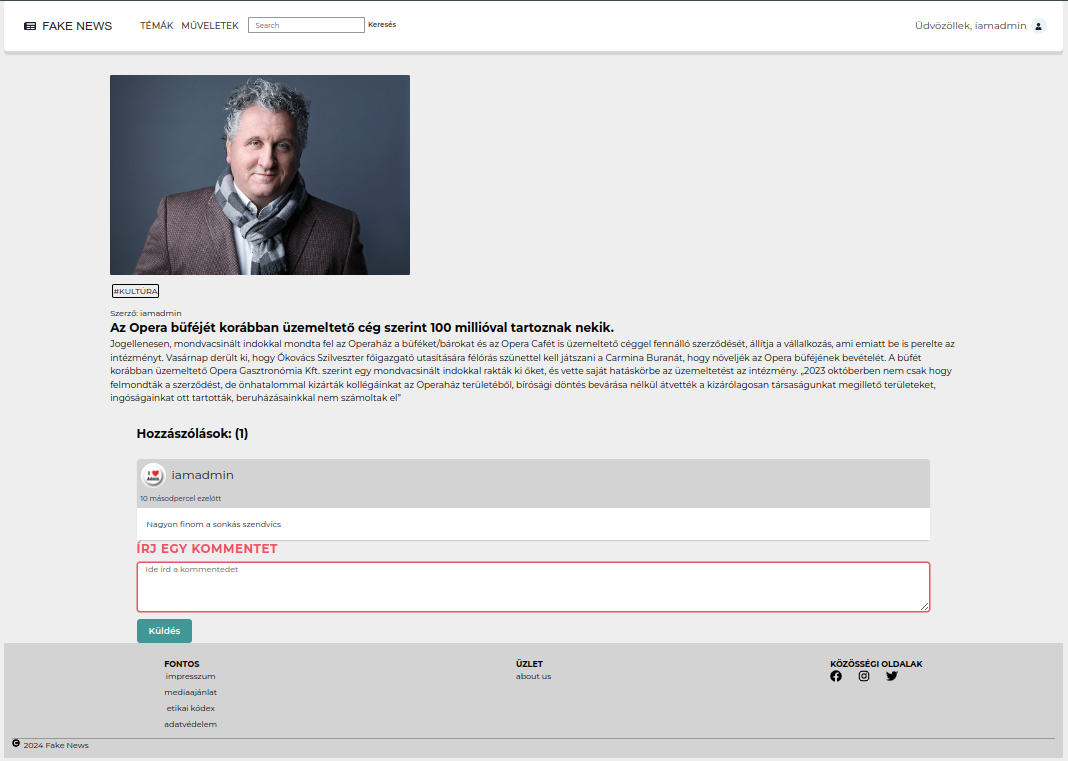
A newsItem-re kattintva már nem a hírt töltöm be a state-be hanem a path-hoz fűzöm a hír id-ját, és erre a dinamikus path-ra navigálok el.

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.

**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.

**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

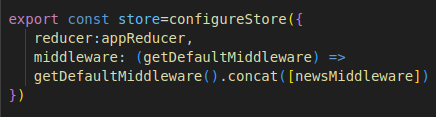
számát, aktuálisan a legnagyobb mérékegységben.

**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.

**STORE DIRECTORY**

**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.

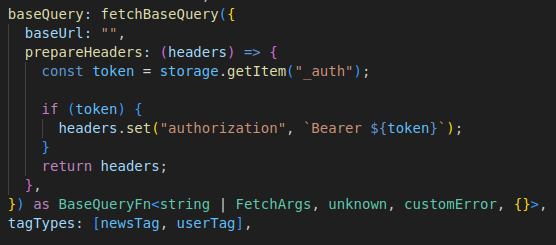
-Kezeli a cache-elt adatokat és az újrakéréseket.

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

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

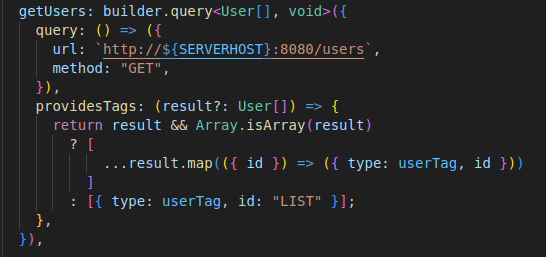
A query egy olyan művelet, amely adatokat kér le a szerverről. Ez tipikusan egy GET kérés, és arra szolgál, hogy a kliens friss adatokat kapjon a szervertől anélkül, hogy bármilyen változást hajtana végre az adatokon.

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

Ezeket egy-egy példán keresztül mutatom be ezeket.

**Query**

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

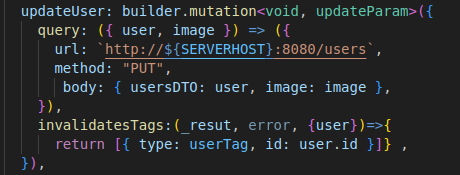
*url*: a szerver endpoint címe  
SERVERHOST egy .env filban előre definiált környezeti változó

ezt az alkalmazás bármely komponenséből elérhetjük.

*method:* a request fajtája

*providesTags:* megadjuk a rendszernek hogy a válaszban kapott felhasználókat milyen címkékkel lássa el, hogy később ezeket frissíteni tudja.

**Mutation**

a queryhez hasonlóan megadjuk milyen adatot küldünk és milyet várunk.   
A request *body*ban JSON objektum ként küldöm el az adatokat

**HOOK**

**UseState** useState hook nem követi le a változó értékének változásait**,** meg kell változtatni

BACKEND

A springboot backend alkalmazások osztályait, illetve komponenseit rétegekbe szervezve strukturáljuk.

Ezek a rétegek a

-Modell

-Repository

-Service

-Controller

-Exception Handling

-Configuration

**ADATBÁZIS**

**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.

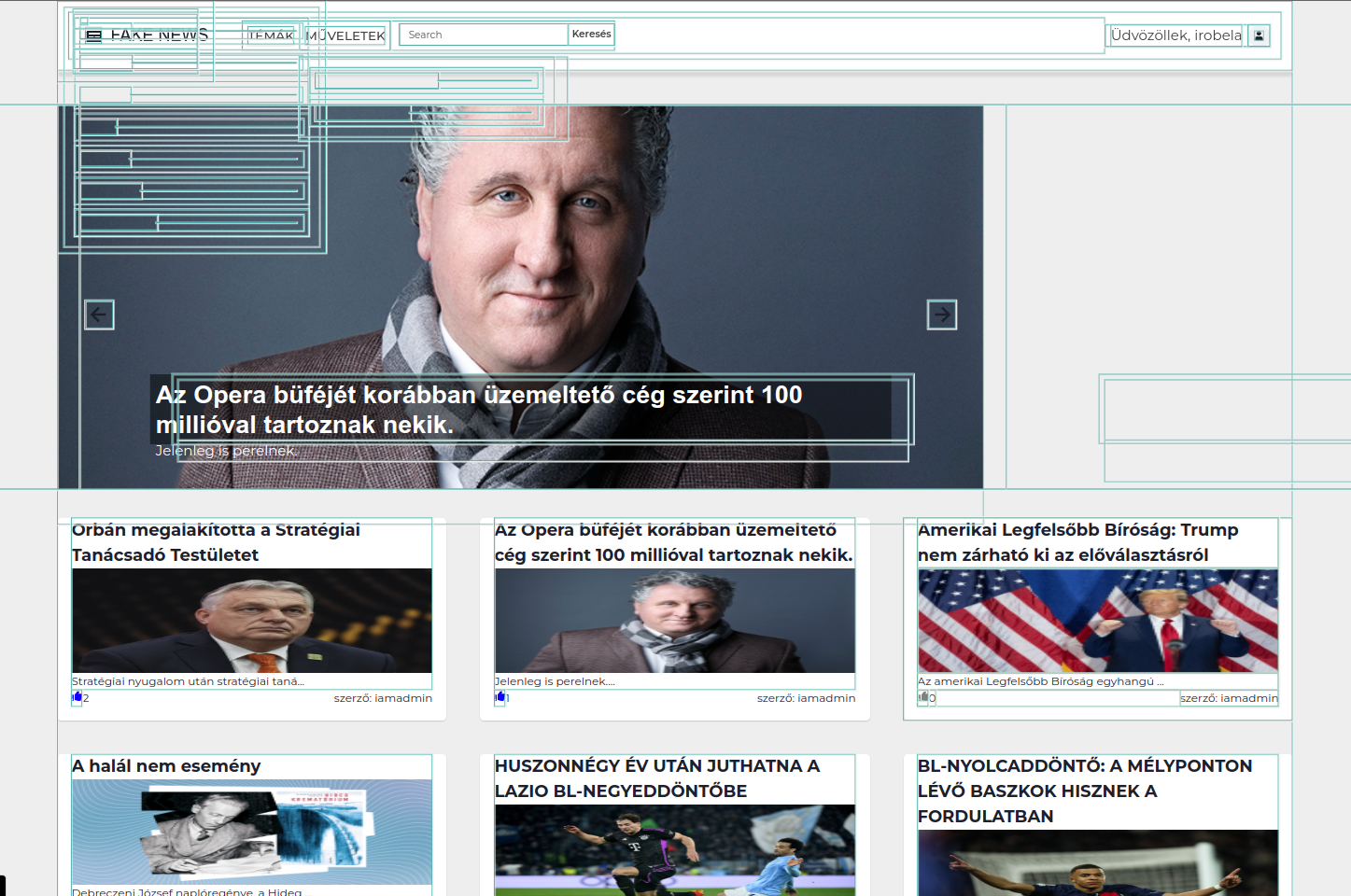
A JPA az alapvető CRUD (Create,Read,Update,Delete) műveletek mellet lehetőséget nyújt speciális lekérdezések használatához, amiket a repository rétegben helyezünk el. Az alábbi lekérdezés a *chatName* alapján keres az adatbázisban, melyet a regisztrációnál használok a *chatName* validálására.

@Query(value= "select \* from users where chat\_name=:chatName", nativeQuery = true)  
Optional<Users> findUserByChatName(@Param("chatName")String username);

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.

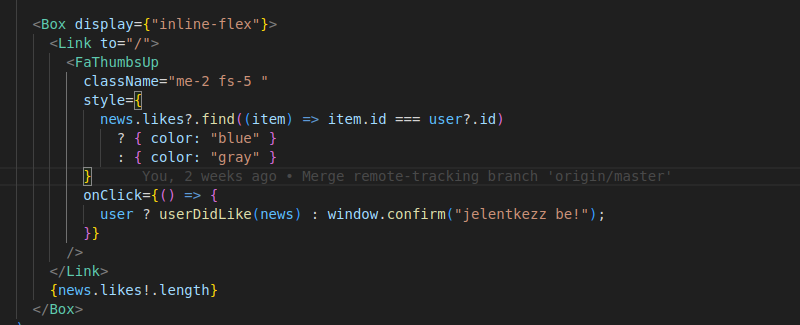
Az alábbi kép (*1.ábra*) egy refaktorálás előtti renderelés látható, amit a „*Like*” gombra kattintás idézett elő.



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.

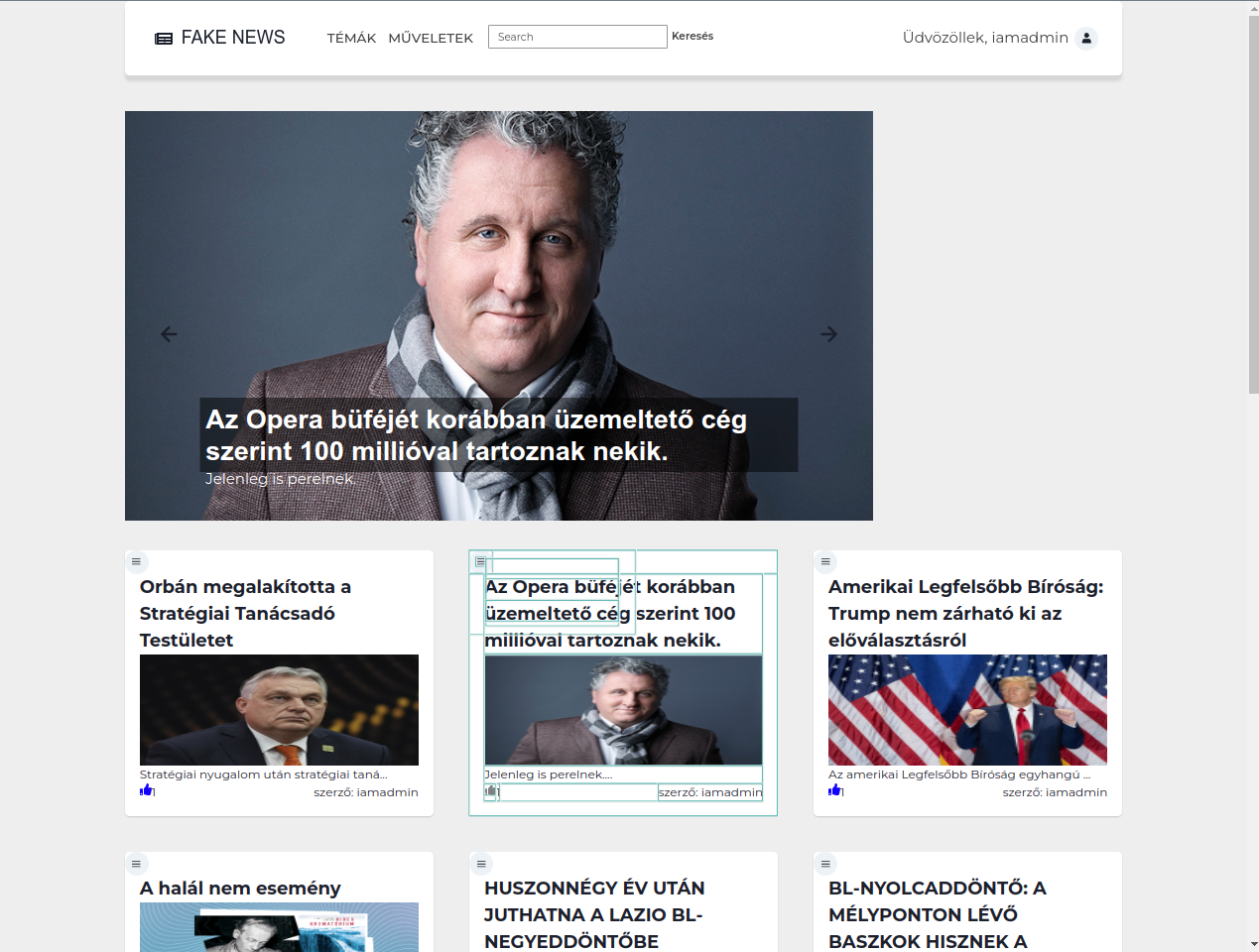
Azonban ennek kötelezően tartalmazni kell egy cél *to=””* propertit, ami alapesetben vissza navigál a gyökér oldalra.



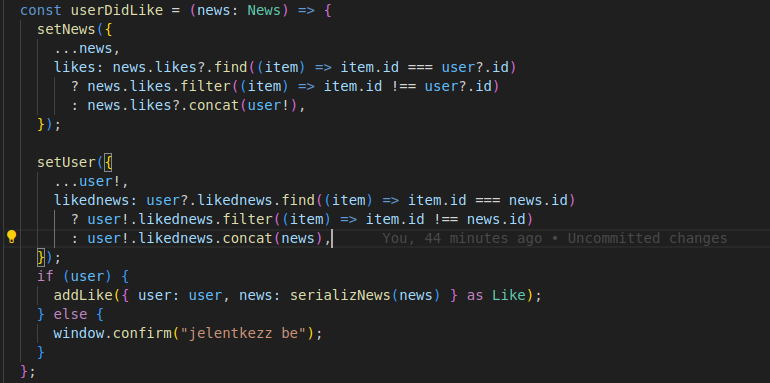
A *@chakra-ui/react* könyvtár tartalmaz *Link tag-*et viszont ebben a szakaszban még a *NewsListItem* komponensben volt a *LikeButton* komponens és szükség volt mind a két típusú *Link* tag-re.  
Ilyen esetben lehetőség van **alias nevet** megadni a tag-nek és azzal hivatkozni rá.



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.

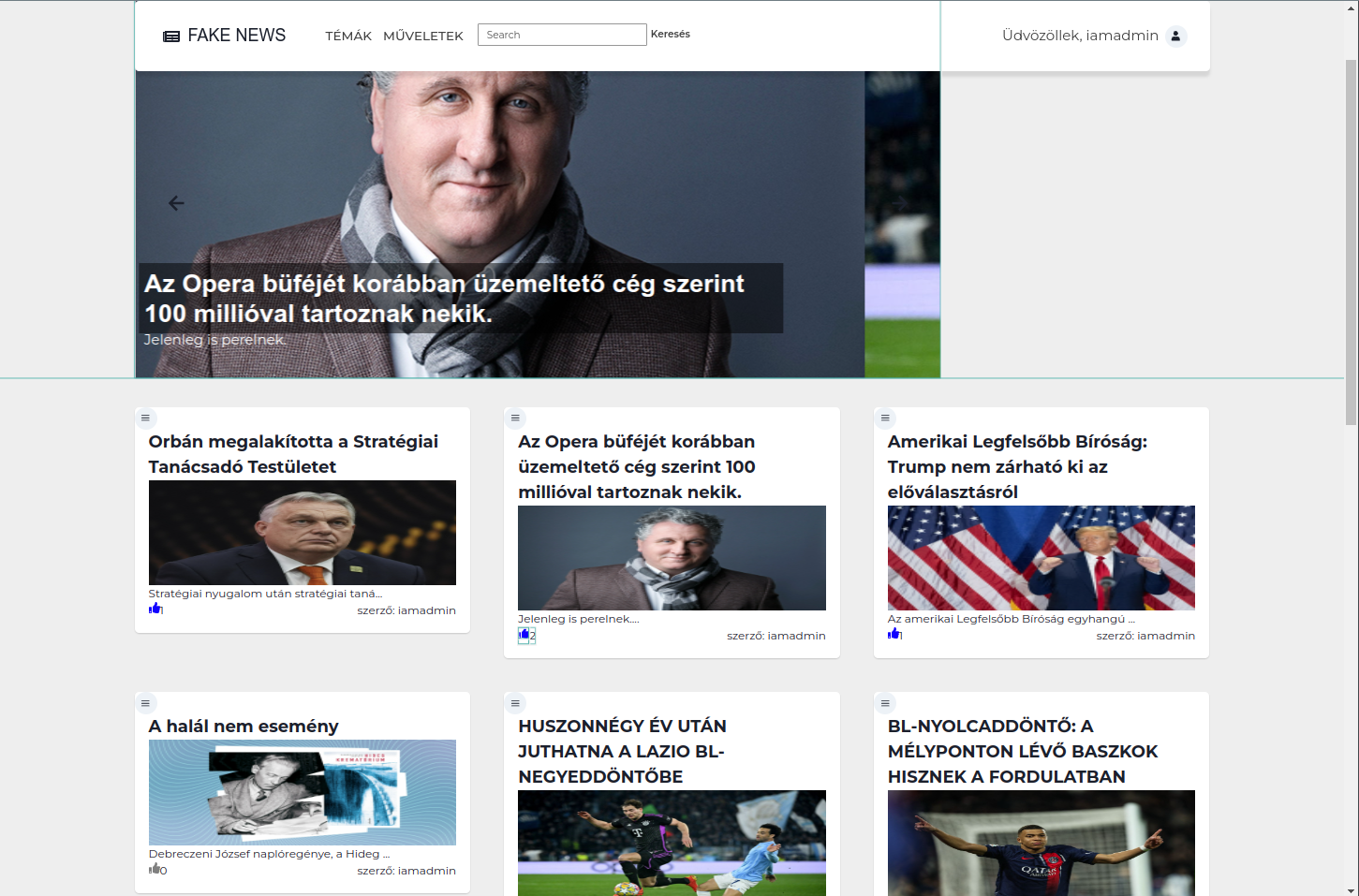


Lokális state változót a *useState()* hook-al definiálhatunk, ezzel tároljuk a komponensek aktuális állapotát és a react ezen keresztül figyeli, mikor indokolt az újra renderelés.

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.

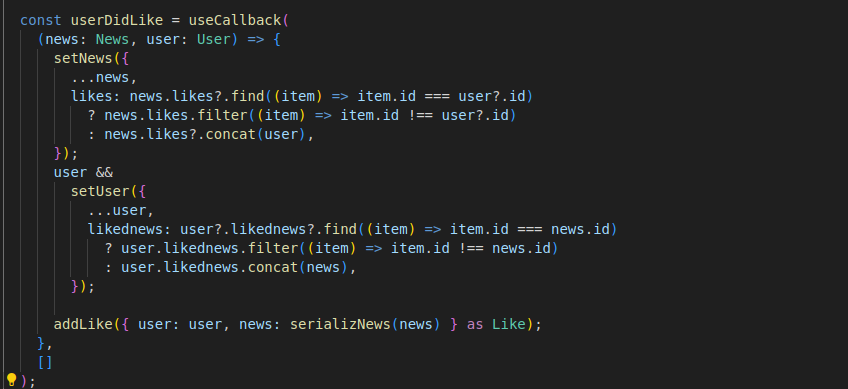


Reactban ha változás történik a globális statben az megváltoztatja a ráhivatkozó változó értékét is a változást egy hook segítségével *useEffect()*-el tudjuk figyelni

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

