SZAKDOLGOZAT

JAKAB BENEDEK

SOPRON

2022

Soproni Egyetem

Faipari Mérnöki és Kreatívipari Kar

Informatikai és Matematikai Intézet

Hirdetési újság külső és belső webes rendszerének készítése React.js alapokon Firebase backend-del

Témavezető:

Szabó Tamás

mesteroktató

A szakdolgozat készítője:

Jakab Benedek

IV.évf. gazdaságinformatikus
BSc hallgató

Tartalomjegyzék

SOPRONI EGYETEM | FAIPARI MÉRNÖKI ÉS KREATÍVIPARI KAR
9400 Sopron, Bajcsy-Zsilinszky u. 4.

NYILATKOZAT

Alulírott **Jakab Benedek** (KZXLAC) jelen nyilatkozat aláírásával kijelentem, hogy a Hirdetési újság külső és belső webes rendszerének készítése React.js alapokon Firebase backend-del című

házi dolgozat;

diplomadolgozat;

szakdolgozat/diplomamunka

(a továbbiakban: dolgozat) **önálló munkám**, a dolgozat készítése során betartottam *a szerzői jogról szóló 1999. évi LXXVI. tv.* szabályait, különösen a hivatkozások és idézések tekintetében.

Hivatkozások és idézések szabályai:
Az 1999. évi LXXVI. tv. a szerzői jogról 34. § (1) és 36. § (1) első két mondata.)

Kijelentem továbbá, hogy a dolgozat készítése során az önálló munka kitétel tekintetében a konzulenst illetve a feladatot kiadó oktatót **nem tévesztettem meg.**

Jelen nyilatkozat aláírásával tudomásul veszem, hogy amennyiben bizonyítható, hogy a dolgozatot **nem magam készítettem,** vagy a dolgozattal kapcsolatban szerzői jogsértés ténye merül fel, a Soproni Egyetem **megtagadja a dolgozat befogadását és ellenem fegyelmi eljárást indíthat.**

A dolgozat befogadásának megtagadása és a fegyelmi eljárás indítása nem érinti a szerzői jogsértés miatti egyéb (polgári jogi, szabálysértési jogi, büntetőjogi) jogkövetkezményeket.

Sopron, 2022. 12. 01.

hallgató

Kivonat

Dolgozatomban egy Burgenland területén működő hirdetőújság online felületének elkészítését mutatom be. Munkám során a Meta (korábban Facebook) által fejlesztett React.js felhasználói felület könyvtárat használtam, melynek működési elvéről, szervezési gyakorlatairól lesz szó. A felület fejlesztésében sok másik nyílt forráskódú könyvtár vált még a hasznomra, hiszen az applikáció többnyelvű és fizetési lehetőség is implementálásra került benne. A fizetési lehetőség megvalósítására a PayPal szolgáltatását választottam. A külső könyvtárak integrálására a Node Package Manager-t használtam, amely a világ legnagyobb szoftver nyilvántartója. Backendnek a Firebase Backend-as-a-Service szolgáltatását választottam. A Firebase rendkívül felgyorsította a fejlesztés menetét, hiszen nem volt szükség kód szinten egyedi backend írására, de mégis testreszabható megoldást biztosított. Ezen szolgáltatás biztosítja az adatbázist is, melyet részletesen tárgyalok.

Természetesen a dolgozatban bemutatom a fejlesztett webapplikáció funkcióit, használatának módját és képességeit. Az applikáció, amellett, hogy rendelkezik egy kidolgozott külső, bejelentkezés nélkül elérhető felülettel, több különböző szerepkörű felhasználó kezelésére képes. Az egyes szerepköröket is részletesen tárgyalom.

Abstract

In my thesis, I present the creation of the online interface of an advertising newspaper operating in Burgenland. During my work, I used the React.js user interface library developed by Meta (formerly Facebook). The library’s operating principle and organizational practices will be discussed. In the development of the interface, many other open source libraries were useful to me, since the application was multilingual and a payment option was also implemented in it. I chose the service of PayPal to implement the payment option. To integrate external libraries, I used the Node Package Manager, which is the world's largest software registry. I chose Firebase's Backend-as-a-Service as the backend. Firebase greatly accelerated the development process, since there was no need to write a unique backend at the code level, but it still provided a customizable solution. This service also provides the database, which I discuss in detail.

Of course, in the thesis I will present the functions of the developed web application, how to use it and its capabilities. The application, in addition to having an elaborate external interface accessible without logging in, is capable of managing several users with different roles. I also discuss the individual roles in detail.

1. Bevezetés

Az újságok az időszámításunk előtti második évszázad óta az életünk részét képezik.[1] Mára elmondható, hogy az online hirdetés kapja a nagyobb hangsúlyt, mégsem elenyésző a papír alapú szórólapok, apróhirdetési újságok száma. Munkám végterméke e kettő megjelenési formát ötvözi. Koncepció szintjén egy web alkalmazás fejlesztéséről beszélünk, ahol a hirdetni kívánó fél helyet vásárolhat magának az online és papír formában egyaránt megjelenő újságban.

A feladat frontendjének elkészítésére a React.js könyvtárat választottam, mivel egyetemi tanulmányaim alatt ezzel a technológiával foglalkoztam és jelen dolgozat írása közben a munkám során is ezt használom. A backendet egy BaaS (Backend-as-a-Service) platform a Google által fejlesztett Firebase adja. Szintén tanulmányaim során ismertem meg és munkám során is használom, viszont a választásnak több oka is van. Elsősorban a fejlesztés sebességének nagymértékű felgyorsítása, akár az alap-funkciókra és -beállításokra, akár a backend későbbi szerkesztésére gondolunk. Másodsorban biztonsági szempontból, a Google tapasztalt mérnökei által biztosított szolgáltatás biztosan erősebb IT-védelemmel rendelkezik, mint, amit egy pályakezdő informatikus ki tud alakítani. Természetesen kiemelten fontos, hogy magam is megfelelően konfiguráljam a rendelkezésre álló biztonsági beállításokat. Harmadrészt a Firebase egy all-in-one felületet ad üzemeltetési szempontból is, hiszen akár technikai- vagy marketing-analitikára, akár Functions (egyénileg, kód szinten írható backend függvények) kimeneteinek jegyzésére, CRON job futtatásra, vagy hosztolásra, felhasználó vagy fájl kezelésre gondolunk, minden egyetlen felhasználóbarát felületen érhető el. Mivel a szolgáltatás élőben elérhető, ezért domain névről is gondoskodni kellett. Ehhez a debreceni székhelyű DiMa.hu Kft. szolgáltatásait vettem igénybe. A DiMa egy magyar cég, amely tárhely és domain szolgáltatással foglalkozik.

1.1 Specifikáció

A követelményeket egy használati eset diagramban ábrázoltam.

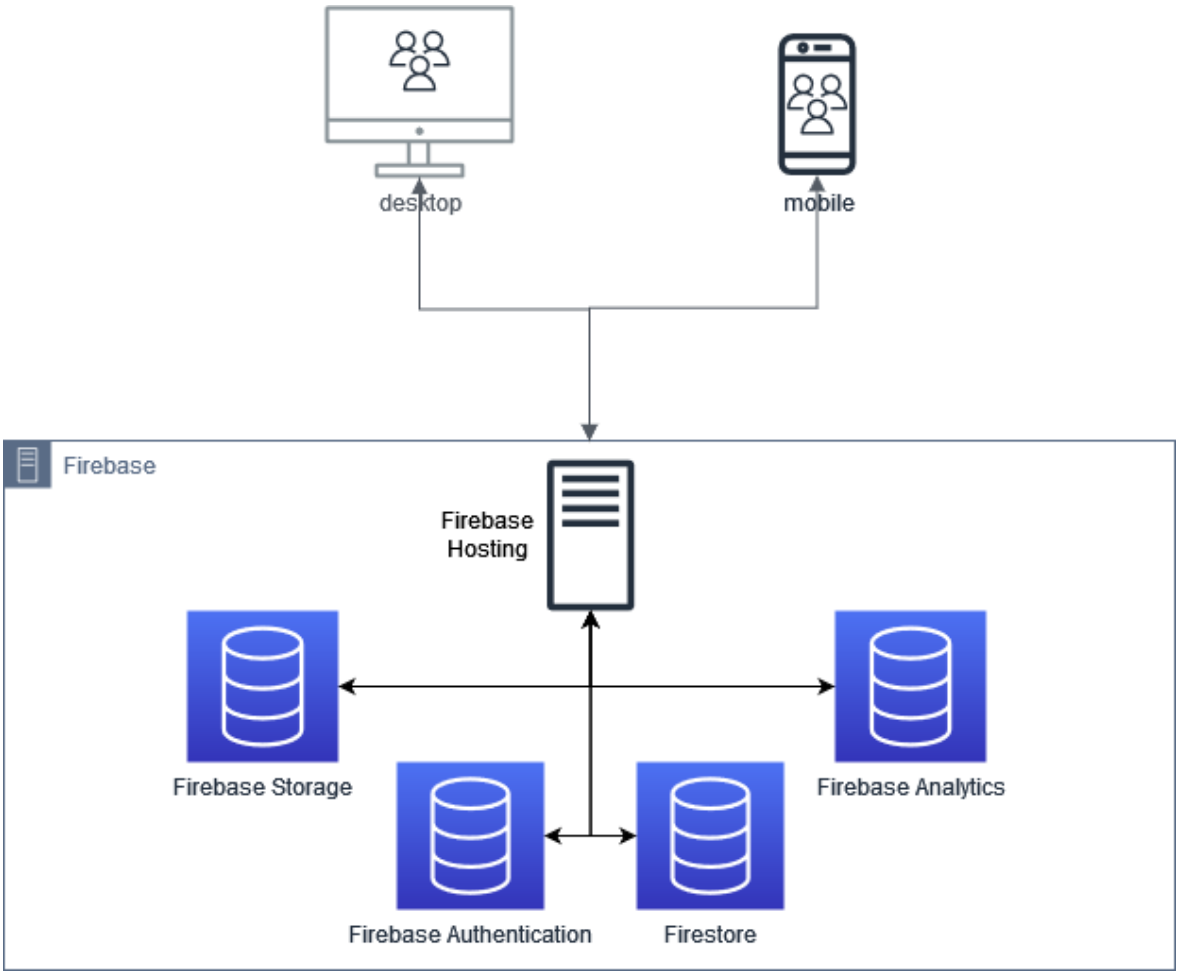
Használati eset diagram

A diagram alapján sok minden egyértelmű, egyetlen lépés van, ami hosszabb kifejtésre szorul.

A hirdetés vásárlása során a hirdetőnek meg kell adnia melyik lapszámban szeretne megjelenni. Meg kell adnia a hirdetés elhelyezésének módját (főoldal, hátoldal, egyedi pozíció, automatikus). Meg kell adnia a hirdetés méretét (egész oldal, ½ oldal, ¼ oldal, ⅛ oldal). Opcionálisan megadhatja a weboldalának webcímét, illetve, ha üzletkötő segítségével talált a hirdetőújságra, akkor annak nevét. Szintén opcionális, hogy fordítói vagy grafikusi segítséget kérjen. Ha kér

grafikusi segítséget, akkor fel kell töltenie pár képet, amiből szeretné, hogy a grafikus összeállítsa a hirdetést. Ha kér fordítói segítséget, akkor meg kell adnia magyarul a hirdetés szövegét, amit szeretne németül megjeleníteni. Ha nem kér segítséget, akkor fel kell töltenie a hirdetés képét. Ezen lépéseket követően fizetnie kell.

1.2 Architektúra



A webalkalmazás architektúrája

Architektúráját tekintve az alkalmazás egyszerű. Egy frontend-backend kapcsolatot kellett kiépíteni. A backend szolgáltatásként, integráltan biztosít adatbázisokat így nem kellett külön adatbázis szolgáltatót igénybe vennem. Az asztal és mobil ikonok csak a felület reszponzivitására utalnak nem fejlesztettem két külön felületet. A reszponzivitás jelentése, hogy minden kliens eszközön, az adott eszközre optimalizálva jelenik meg a weboldal. Esetünkben ez a felület

átrendeződését, extra animációkat, funkciókat jelent. Napjainkban minden webalkalmazással szemben elvárás, hogy reszponzív legyen.

2. Firebase használata az alkalmazásban

2.1 Áttekintés

A Firebase a Google által mobil- és webalkalmazások, játékok backendjének készítéséhez kifejlesztett platform. Eredetileg egy független vállalat volt, amelyet 2011-ben alapítottak. 2014-ben a Google megvásárolta a platformot, és mára ez az alkalmazásfejlesztés egyik zászlóshajója. Napjainkban web-, mobilapplikációk, illetve Unity Engine alapú játékok fejlesztéséhez használatos.[2]

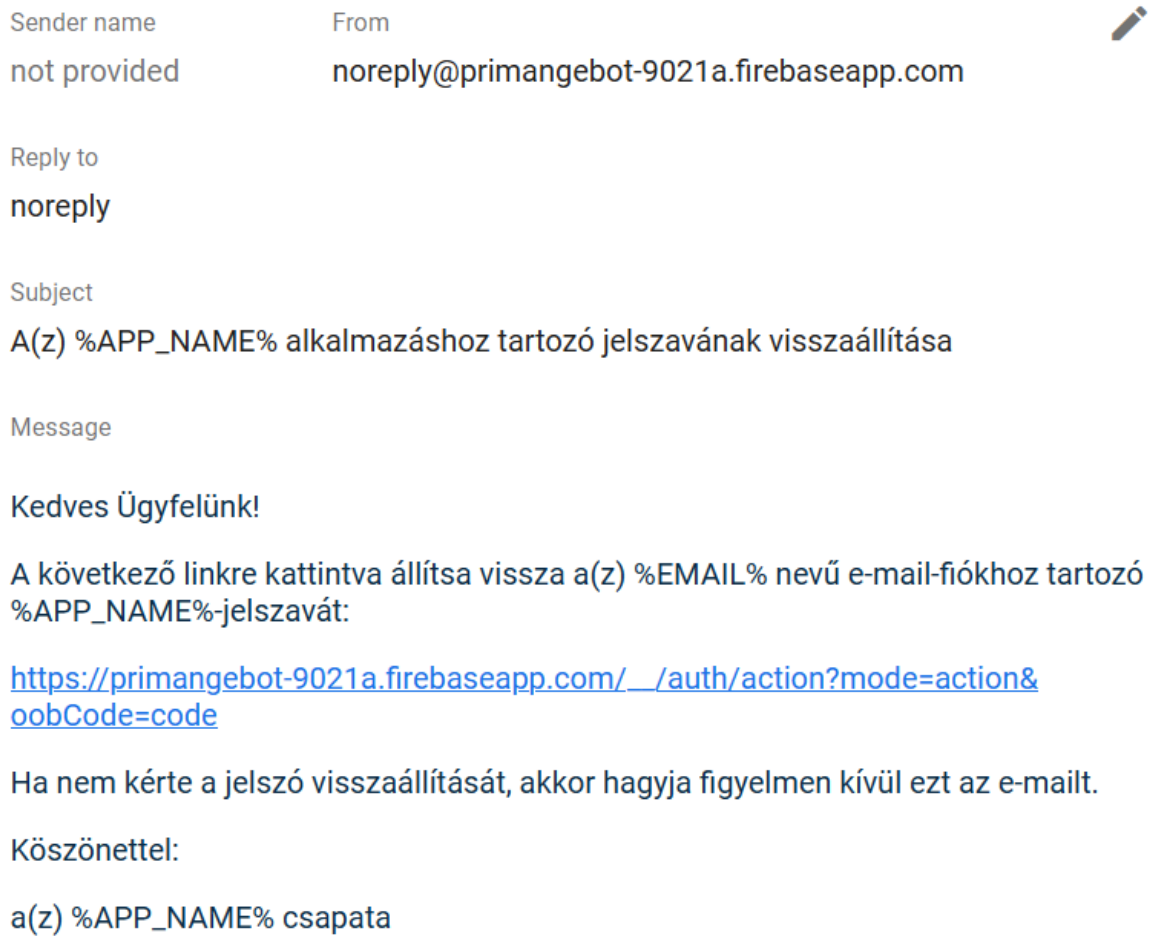
Természetesen, mivel szolgáltatás alapú és ingyen nem érné meg a Google-nek kifejleszteni egy ilyen rendszert és üzemeltetni, érdemes szót ejteni az árakról. Két pricing plan-el rendelkezik a Spark és a Blaze plan-ekkel. Itt megjegyezném az elnevezés kreatív mivoltját, miszerint a Firebase vagyis “tűzbázisnál” az ingyenes verzió a “szikra” és a fizetős verzió a “láng” elnevezést kapta, szellemes. Esetünkben az ingyenes verzió (Spark Plan) egyelőre elég is. ez magában foglalja az Analytics használatát, 1 GB tárhelyet a Firestore-ban, 10GB havi adatforgalommal. 10 GB tárhelyet hosztolási célokra, napi 360 MB adatforgalommal. A Firebase Storage-ben, ahol a képeket tárolom 5 GB tárhelyet biztosít, 1 GB napi letöltési korláttal.

2.2 Munkám során használt szolgáltatásai

2.2.1 Authentication

A Firebase biztosít egy biztonságos különböző megoldásokkal kompatibilis autentikációs rendszert. Amellett, hogy a kezdőképernyőjén részletes információkkal együtt egy könnyen átlátható listát kapunk a felhasználóinkról, lehetőséget biztosít felhasználók manuális felvitelére is, amely megoldással élek is a Projekt tulajdonos szerepkörű felhasználónak létrehozásakor. Azért így hozom létre a kulcs felhasználót, mivel minderre a saját alkalmazásomon belül lehetőséget biztosítani komoly biztonsági rés lehetőségét vetné fel. Bár jelenleg több mint 10 különböző sign-in providert támogat (például: Google, Facebook, Play Games),

jelen rendszer ebből csupán egyet, a klasszikus email-jelszó autentikációt használja. Mindemellett lehetőséget ad az autentikációval kapcsolatos emailek (mint új jelszó kérés) küldésére és szerkesztésére is, amire egy példa a következő képen látható.



A jelszó visszaállítást biztosító email mintája.

2.2.2 Firestore database

Ez a beépített NoSQL alapú adatbázis szolgáltatás, tulajdonképpen szinte minden az alkalmazásban szereplő adatot itt tárolok. Lehetőség van az adatbázisszerver fizikai helyének megválasztására, jelen esetben az „eur3 (europe-west)” szerveren Frankfurtban tárolódnak az adatok, hogy geológiailag a lehető legközelebb legyen Ausztria területéhez. Természetesen minden -a szolgáltatás által tárolt adatért- felelősséggel tartozom, úgyhogy felmerül a hozzáférések kérdése, de a Firestore ennek kezelésére biztosít egy kompakt, könnyen használható megoldást. A backend kezelőfelületén elérhető egy „Rules” menüpont,

ahol az adatbázisra vonatkozóan pár sor kódból be lehet állítani írási – olvasási jogosultságokat.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read: if true
      allow write: if request.auth != null
    }
  }
}
```

Az applikáció hozzáférési konfigurációja

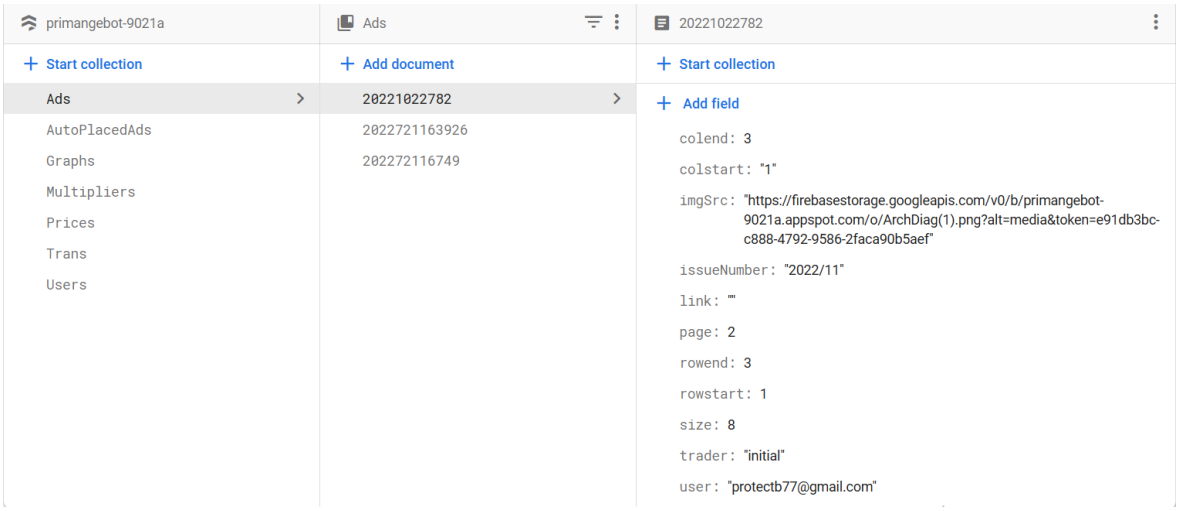
Ahogy az ábrán is látható összesen 9 sor kód elegendő ahhoz, hogy beállítsuk, hogy csak bejelentkezett felhasználók rendelkezzenek írási joggal. Fontos megjegyezni, hogy ekkor még nem bejelentkezett felhasználók is képesek az összes adat olvasására és a bejelentkezett felhasználók sem csak a saját maguk által létrehozott információ felett rendelkeznek írási joggal, hanem bárkié felett. Mivel NoSQL adatbázisról beszélünk célszerű lenne, ha collection vagy akár azon belüli dokumentum szinten be tudnánk állítani jogosultságokat, természetesen erre is van lehetőség. Annak érdekében, hogy megbizonyosodhassunk a konfiguráció hibátlan működéséről, a Firestore biztosít számunkra egy Rules Playgroundot, ahol pár kattintással szimulálhatunk írási-olvasási tevékenységet különböző konfigurációkban és megnézhetjük ennek eredményét így kiszűrve, ha egy általunk tiltott tevékenység még mindig bekövetkezhet vagy épp ellenkezőleg, megtiltunk olyan interakciókat, amelyeket nem akartunk. Ez azért jó, mert így saját frontend fejlesztése nélkül is képesek vagyunk tesztelni az adatbázis hozzáférési szabályait.

2.2.2.1 Adatszerkezet

Ha adatbázisról beszélünk fontos ismerni a típusát. Esetünkben egy NoSQL adatbázisról beszélünk vagyis nem tábláink és azokon belül soraink vannak, hanem collection-ök és dokumentumok. A collection a dokumentumok gyűjtőhelye, ami indexelhető. Egy dokumentumot egy JSON Object-hez tudok a leginkább hasonlítani, annyi eltéréssel, hogy egy dokumentumnak magának is van ID-ja. Egy

dokumentum kulcs - érték pár alapú.

A legbonyolultabb dokumentum az alkalmazásban egy hirdetésé.



A Firestore kezelési felületének része, szemléltetve egy hirdetés adatszerkezetét

A dokumentum ID-ja nem más, mint egy saját timestamp, amely a dokumentum létrejöttének idejét tartalmazza –év, hónap, nap, óra, perc, másodperc formában–, pl: 20221022782. Rendelkezik egy imgSrc kulccsal, amely egy linket tartalmaz. Ezt a linket a hirdetés képének feltöltésekor kapjuk, úgy, hogy, amikor feltöltjük a képet a Storage-ba, visszakérjük a downloadURL-jét és ezt az URL tároljuk el. Ezenfelül rendelkezik egy issueNumber kulccsal, amely string-ként eltárolja a hirdetés lapszámát. rendelkezik továbbá egy link kulccsal, ahol eltárolhatjuk a hirdető honlapjának címét is, mivel egy extra olyan funkcióval rendelkezik az újság online kiadása, hogy, ha megtetszik egy hirdetés elég rákattintani és máris a hirdető weboldalán találjuk magunkat. A nyomtatott sajtó esetén ezt a webcímet nem jelenítjük meg. Rendelkezik egy page kulccsal, amelyben az lapszámon belüli oldalszámát tároljuk. A hirdetés méretét a size kulcs alatt találjuk, ennek értéke mindig 1/[oldalon elfoglalt terület]. Például 1/(1/8) vagyis 8 egy egy nyolcad oldalt elfoglaló hirdetés esetén. Rendelkezik egy trader kulccsal, amely akárcsak a link opcionális. Itt tároljuk el az esetleges üzletkötő nevét. Különbség e két kulcs között, hogy a linknek az alapértelmezett értéke egy üres string, míg a “trader”-nek “initial”. Az “initial” szövegre az olvashatóbb kód írás végett van szükség, mivel az üzletkötőt kiválasztó dropdown listán csak akkor jelenik meg az üzletkötő választására felszólító szöveg, ha a “trader” értéke “initial”.

Természetesen ahhoz, hogy be tudjuk azonosítani a felhasználót, akihez tartozik, annak az email címét eltároljuk egy user kulcs alatt. Az oldalon belüli pozíció meghatározására 4 kulcsot vezettem be: rowStart, rowEnd, colStart, colEnd. Ezek hivatottak a CSS által meghatározott gridben elfoglalt helyet reprezentálni, olyan módon, hogy a két “start” inkluzív a sor kezdetét illetően, a két “end” viszont exkluzív. Például jobb felső sarokban elhelyezkedő ⅛ méretű hirdetés esetén: rowstart - 1, rowend - 2, colstart - 2, colend - 3.

Azon hirdetések esetén, amelyek helye automata elhelyezés szerint lesz meghatározva külön collection-t hoztam létre. Az ebben található dokumentumok szerkezete egyezik a normál hirdetésekével, leszámítva, hogy nincs benne meghatározva a rowstart, rowend, colstart, colend kvartett.

Amennyiben a hirdetéshez grafikusi segítségre van szükség külön collection tárolja az adott hirdetéshez (egyező ID-val) tartozó képeket, amelyből a grafikusnak dolgoznia kell. Ugyanez igaz a fordításokra is.

Az árak collection-je négy dokumentumot tartalmaz, mindegyik egy hirdetésméretet jelöl. Minden hirdetés mérethez két kulcs tartozik: egy huf és egy eur, amelyek a jelölt pénznemben tartalmazzák az árat.

A felhasználók tárolása egyszerű, minden felhasználónak van egy dokumentuma, amely ID-ja a felhasználó email címe és az egyetlen kulcsa a role vagyis a szerepe.

2.2.3 Storage

A Storage a Firebasen belül egy fájl feltöltésre alkalmas felhőtárhely. Az alkalmazás a hirdetések képeinek tárolására használja. Amikor a feltöltés megvalósul, a feltöltött képnek lekérem a publikusan elérhető downloadURL-jét, amelyet eltárolok a hirdetés dokumentumában.

2.2.4 Hosting

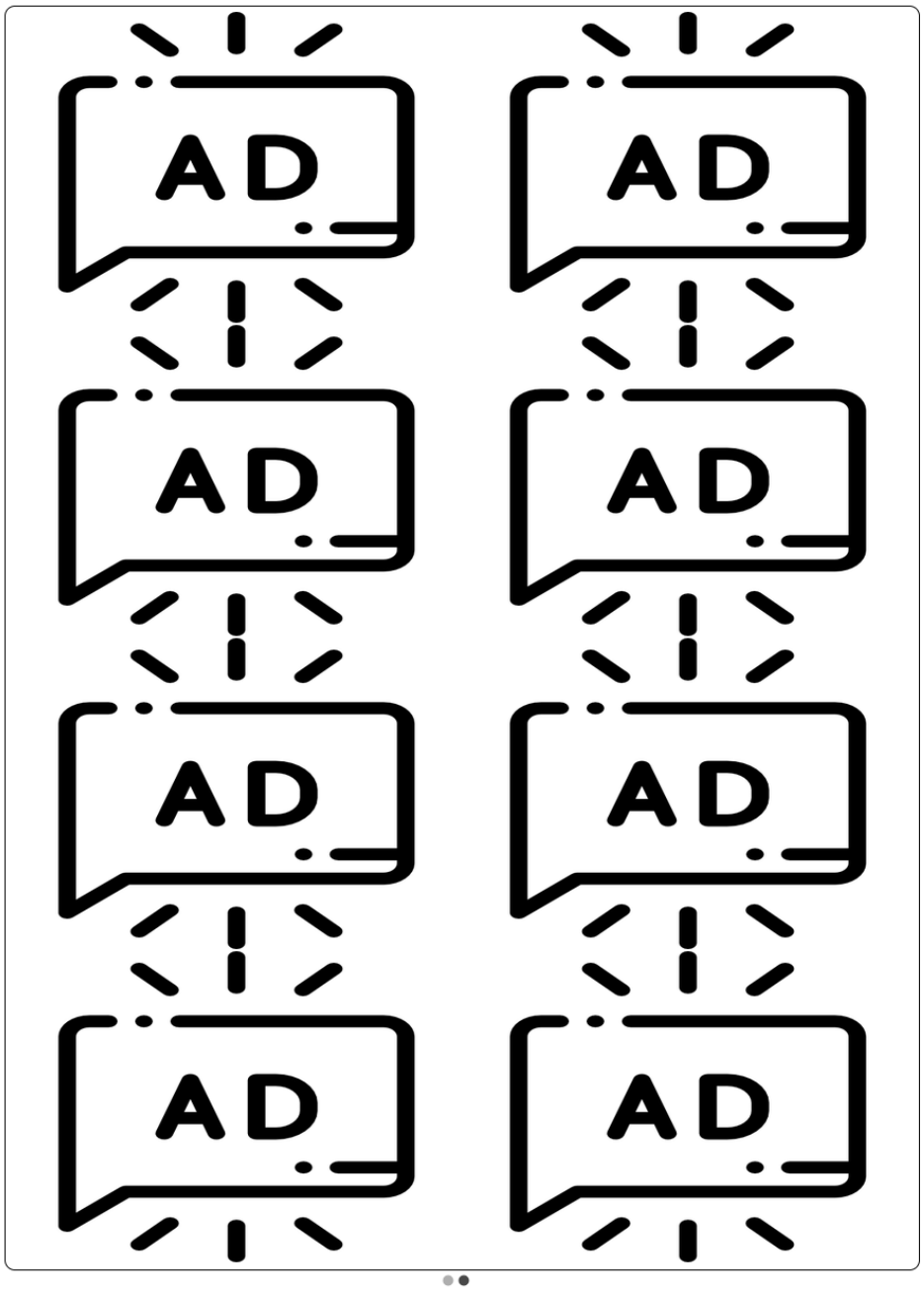
Ahogy a bevezetésben is írtam a Firebase kétségtelen előnye, hogy hosztolást is biztosít. Jelen esetben is ezt veszem igénybe, a DiMa.hu Kft.-től vásárolt domain webhelyre irányítása nagyon könnyű volt.

- Amikor a Firebase konzol felületén kiválasztom a Hosting menüpontot lehetőségem van a weboldal domain-eit módosítani.
- Ha rákattintok az “Add custom domain” gombra feldob egy felugró ablakot. Itt meg kell adnom az új domain nevet.
- Utána kapok egy google-site-verification kódot.
- A DNS szolgáltatónál az új domainhez létre kell hoznom egy új TXT rekordot, melynek tartalma a google-site-verification kód.
- Ezt követően a “Verify” gombra kell kattintanom és az új domain már életbe is lép.

3. A webapplikáció oldalai

3.1 Külső rendszer

3.1.1 Főoldal

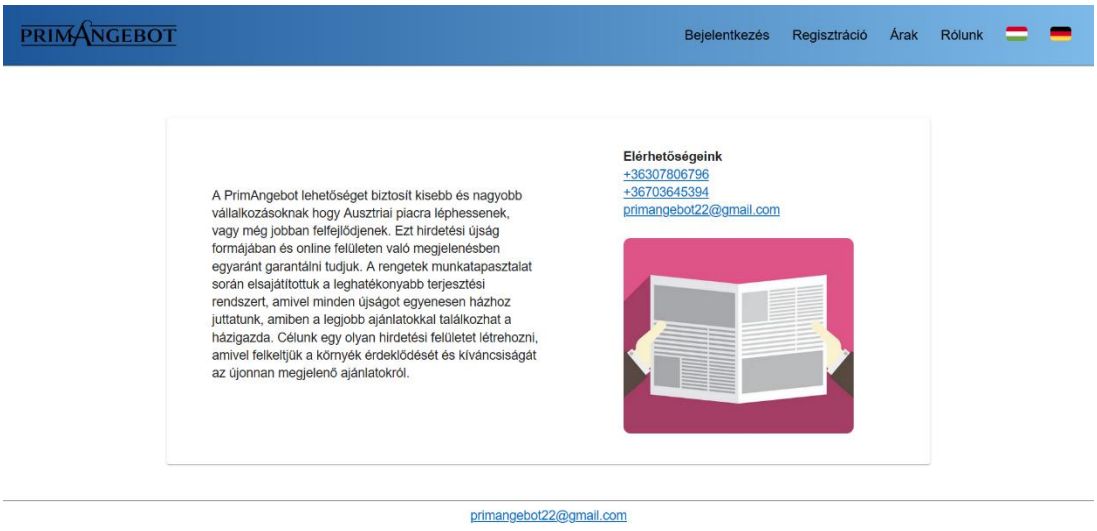


A főoldal

A weboldal nyitólapja maga az újság digitális formája. Ez egy 2x4-es grid, amely az aktuális lapszám főoldalának hirdetéseit tartalmazza. Amennyiben az adott hónapban valamilyen okból nem jelenik meg az oldal fallback-el az előző lapszámra. Természetesen nem csak a főoldal tekinthető meg itt, hanem az oldal jobb és bal oldalán található nyilakkal navigálhatunk az adott lapszám oldalai között. Mivel jó eséllyel az oldal felhasználói önmaguktól nem fognak “végigkattogtatni” a hirdetéseken, ezért az újság oldalai automatikusan cserélődnek, amikor az újság végére érünk elölről kezdődik. Ezen felül a megjelenített újság alatt pagination vezérlők kerültek elhelyezésre. A felhasználó ennek segítségével kiválaszthatja, hogy az újság mely’ oldalát szeretné megnézni.

3.1.2 Rólunk

Amennyiben egy cégnek van egy weboldala, szinte kötelező, hogy legyen egy „rólunk” oldal, amely bemutatja a céget és annak termékeit, szolgáltatásait. Jelen esetben ez is két nyelven érhető el, tartalmazza emellett az elérhetőségeket és egy képet is, hogy stílusos maradjon az oldal. A következő képeken megfigyelhető az oldal többnyelvűsége.



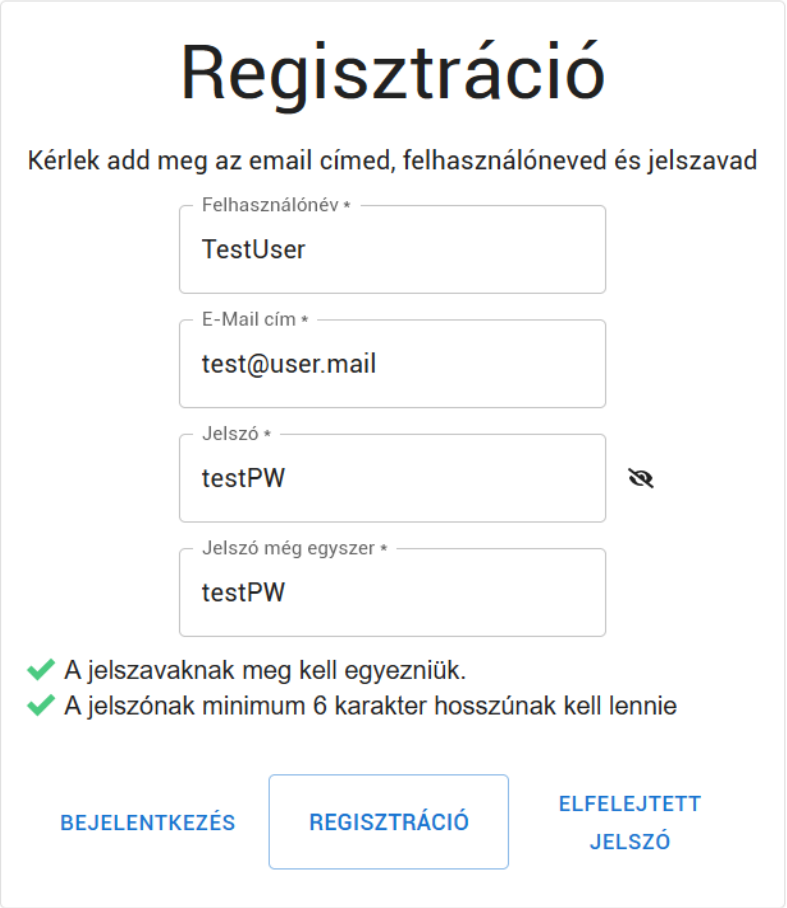
Rólunk oldal magyarul asztali gép nézetben

Rólunk oldal németül mobil nézetben

3.1.3 Regisztráció

Mivel a leendő ügyfeleknek felhasználóval kell rendelkezniük az oldalon, felületet kell biztosítani a regisztrációhoz. Különösebb megkötések a jelszót illetően nincsenek, viszont, felmerült bennem az ötlet, hogy egy esetleges elgépelés ne okozzon problémát a jelszót kétszer kell megadni és nem lehet beilleszteni vágólapról. UX szempontból ez egy elég kellemetlen funkció, hiszen, ha egy hosszú és bonyolult jelszóval rendelkezik a felhasználó, akkor sok ideig tart kétszer begépelni, úgyhogy végül elvetettem ezt a megoldást. Egy kis szem ikont is elhelyeztem a jelszó input mellett, hogy a felhasználó láthassa mit gépelt be. Ezzel a funkcióval kapcsolatban felmerült egy kompatibilitási probléma, mégpedig, hogy a Microsoft Edge alapértelmezetten hozzárendel egy ilyen ikont a „password” típusú inputokhoz, úgyhogy mielőtt az ikon kirajzolásra kerülne a “react-device-detect” csomag *isEdge* értéke ellenőrzésre kerül, és csak akkor renderelődik ki, ha ez az érték hamis. A jelszóval szemben támasztott követelmények megjelenítése a react-password-checklist csomag segítségével történik. A kártya, a beviteli mezők és gombok stílusa a @mui/material csomagból származik. Az elrendezést eleve

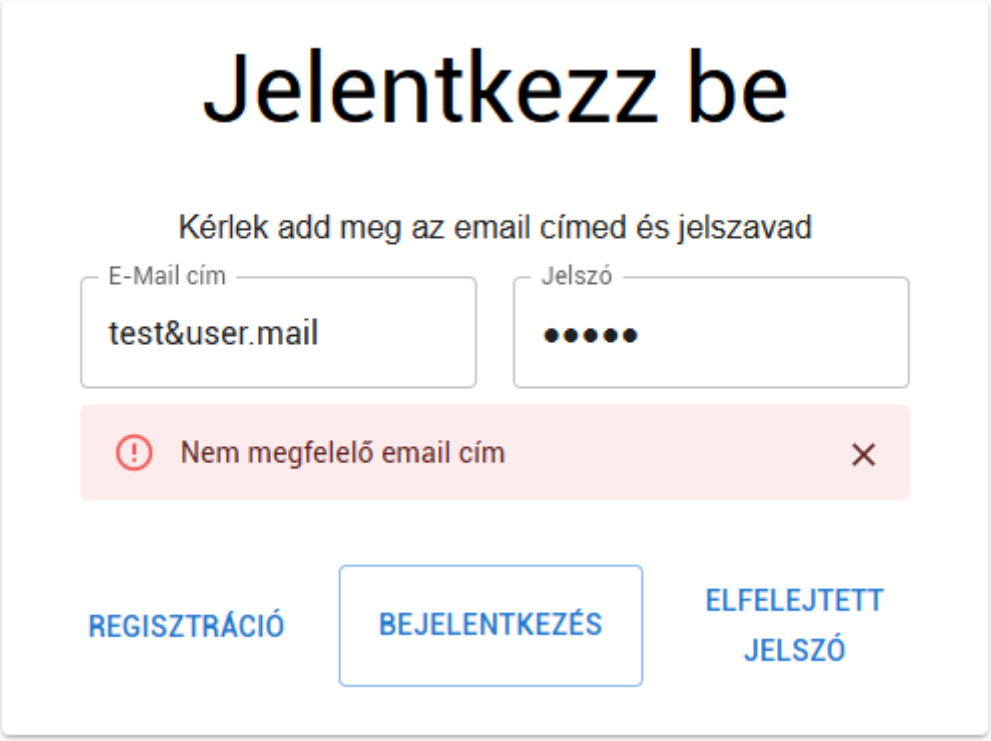
oszloposnak definiáltam, így nem kellett sokat foglalkozni a reszponzivitással.



Regisztrációs felület

3.1.4 Bejelentkező felület

Természetesen a regisztrációt követően be kell tudnia jelentkezni a felhasználónak, ehhez a következő képen látható felület biztosít lehetőséget. Amennyiben a bejelentkezés valamilyen hibába ütközik adekvát hibaüzenet jelenik meg. Az alábbi példában a felhasználó a „@” karakter helyett „&” karaktert használt az email cím megadásakor, vagyis az email cím nem megfelelő.



Bejelentkező felület hibaüzenettel

A hibaüzenetnek készítettem egy saját komponenst. A komponens paraméterek alapján rajzolódik ki.

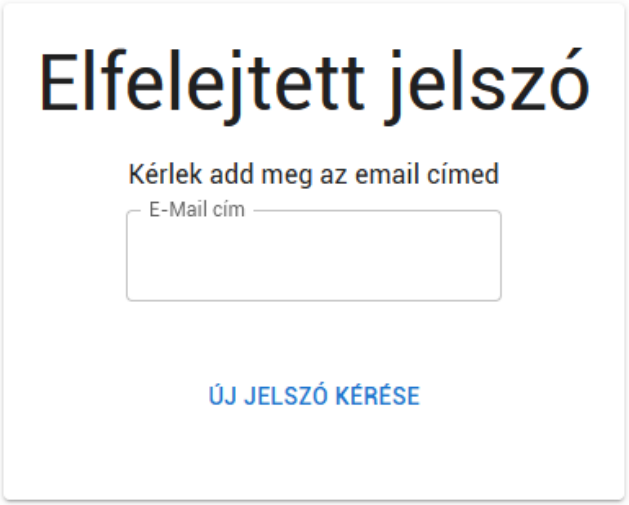
- Az első paraméter, hogy egyáltalán megjelenjen-e az üzenet.
- A második paraméter, hogy milyen típusú üzenet. Ez a komponens nem csak hibaüzenetek, hanem sikerességet jelző üzenetek kirajzolására is képes.
- A harmadik paraméter maga az üzenet, amit ki kell írnia.

A felugró üzenet két másodperc után eltűnik, de látható rajta egy “X” gomb, amellyel bezárhatjuk. Ha valakinek nem lenne egyértelmű a gomb jelentése, az egeret a gomb fölé mozgatva egy “Close” szöveget kapunk, természetesen az aktuálisan beállított nyelvre lefordítva.

3.1.5 Elfelejtett jelszó

Napjainkban rengeteg különböző alkalmazást használunk. Ezeknek nagy részéhez azonosítanunk kell magunkat – szinte minden esetben egy jelszóval–. Sokat lehetne írni a helyes jelszó készítési, használati stratégiákról, de azt nyugodtan merem állítani, hogy egyazon jelszó használata az összes platformon

kimondottan kerülendő. Ha nem használunk jelszókezelőt, akkor viszont a nagyszámú különböző jelszó megjegyzése nehéz lehet. Ekkor válhat hasznunkra egy Elfelejtett jelszó opció, amikor az email címünk megadását követően egy emailt kapunk, amelyben található egy link, ahol beállíthatunk magunknak egy új jelszót. Esetükben mindez a Firebase által már implementált funkció, de testreszabható, hiszen be tudjuk állítani az új jelszó kérő email szövegét.



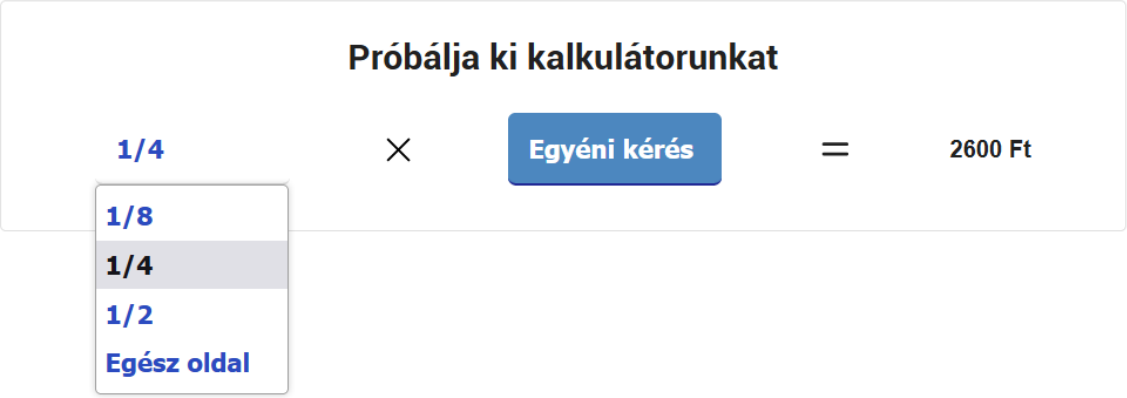
Új jelszó kérésére alkalmas felület

3.1.6 Áraink

Amikor valaki igénybe szeretne venni egy szolgáltatást mindenekelőtt tisztában szeretne lenni azzal, hogy mennyibe fog neki kerülni. Mivel a mi esetünkben az árak dinamikusan beállított paramétereknek megfelelően számítódnak, hasznos a felhasználóknak adni egy felületet, ahol egy becslést tehetnek a költségeikre. Ehhez készítettem egy kalkulátort. Két dropdown listából kiválaszthatjuk:

- a hirdetés méretét
- elhelyezkedésének megadási módját az újságban. Ez lehet főoldal, hátoldal, automatikus elhelyezés és egyedi választás.

Eredményül az aktuális árak alapján a hirdetés díját kapjuk.



Kalkulátor

3.2 Belső rendszer felhasználói szerepkörök szerint

A belső rendszer a bejelentkezett felhasználó szerepköre alapján különböző funkciókat biztosít.

Ezt úgy valósítottam meg, hogy bejelentkezést követően egy Higher-Order komponenst töltök be. A Higher-Order Component röviden HOC. Míg egy normál komponens átalakítja a prop-okat a felhasználói felület egyes elemeivé, a HOC egy másik komponenssé alakítja át.

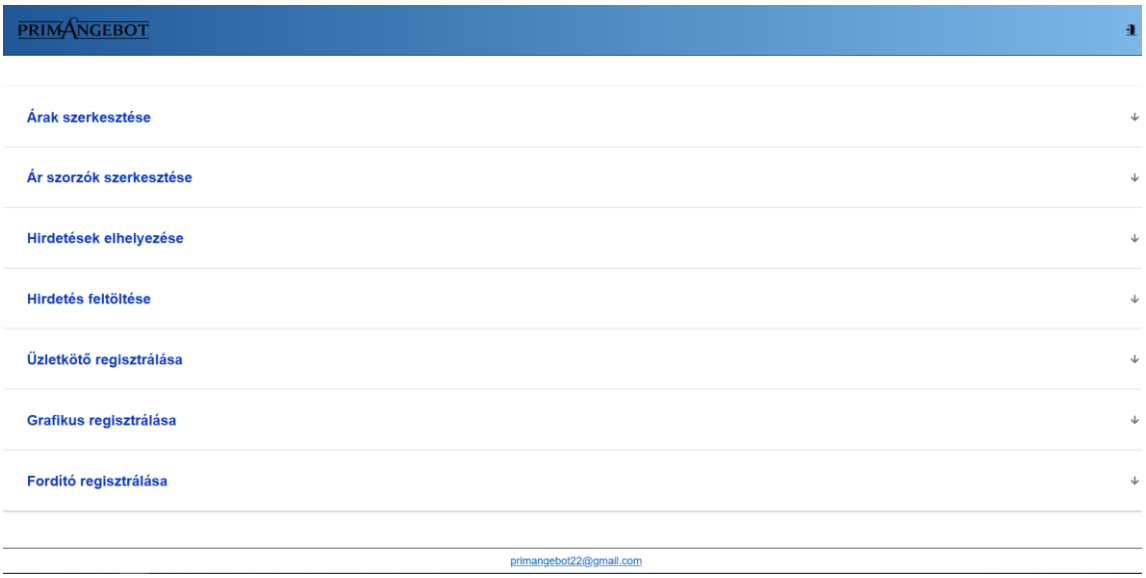
Ez a HOC a bejelentkezett felhasználó emailcíme alapján megkeresi a Firestore-ban hozzá tartozó dokumentumot és az ebben lévő *role* kulcs értéke alapján *switch-case* segítségével visszaadja a *role*-nak megfelelő komponenst.

```jsx
const [role, setRole] = useState();

useEffect(() => {
  async function GetRole() {
    const userDoc = await getDoc(doc(db, "Users", auth.currentUser.email));
    setRole(userDoc.data().role);
  }
  GetRole();
}, []);

switch (role) {
  case "ProjectOwner":
    return <Admin />;
  case "Trader":
    return <Trader />;
  case "Customer":
    return <Customer />;
  case "GraphDesigner":
    return <GraphDesigner />;
  case "Translator":
    return <Translator />;
  default:
    return <CircularProgress />;
}
```

A felhasználó szerepköre szerint kirajzoló HOC

3.2.1 Projekt tulajdonos

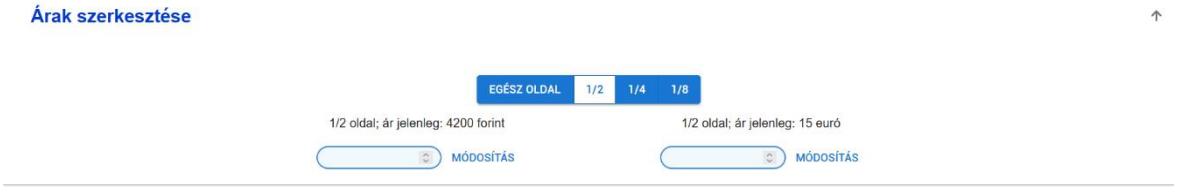
Ezen felhasználó számára a belső felület lényegében egyetlen oldalból áll, ahol a különböző menüpontokat Accordion-okként jelenítem meg.



Projekt tulajdonos főoldal nézete

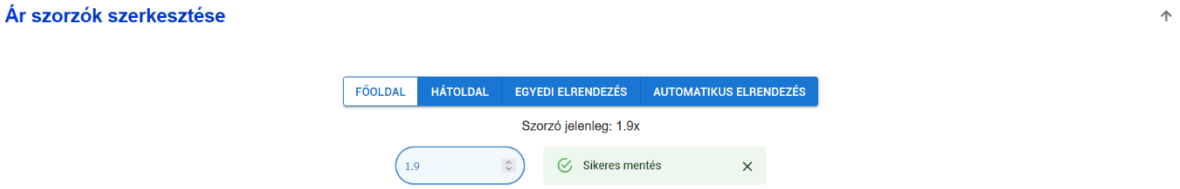
Az első menüpont az árak szerkesztése. Külön biztosítok lehetőséget a forint és euróban meghatározott ár módosítására, mivel automata konverzió esetén

törtszámokkal kellene számolni vagy kerekítést alkalmazni, amelyek marketing szempontból nem „szép” számokat eredményeznek. Az emberek jobban szeretik kerek számokban látni a fizetendő összegeket.



Árak szerkesztése

Az elhelyezés módjának megválasztása marketing szempontból fontos, hiszen az újságnak zárt állapotában vagy a fő- vagy a hátoldalát látjuk, tehát egy ezeken a helyeken elhelyezkedő hirdetésre többször pillant rá a potenciális vevő. Éppen ezért azokon a helyeken drágább hirdetni. A hirdetést vásárlók ezek mellett kérhetik, hogy ők választhassák ki a reklám –újságban való helyét–, hiszen, példaként, ha mondjuk minden hónapban vesz hirdetést (mert mondjuk havi akciókat hirdet), a vevői könnyebben megtalálhatják a hirdetést, ha mindig a második oldal bal felső sarkában helyezkedik el. Természetesen van lehetőség úgynevezett Automatikus elhelyezésre, ahol a Projekt tulajdonos az újság megjelenése előtt elhelyezi a szabad helyekre a beérkezett ilyen hirdetéseket. Az árat szorzók segítségével határozzuk meg (lsd.: Kalkulátor a külső rendszerben), amelyeket a projekt tulajdonos szerkesztheti.



Ár szorzók szerkesztése sikeres változtatást követően

A Projekt tulajdonosnak lehetősége van feltölteni hirdetést. Teljesen azonos módon tudja megadni a hirdetéshez szükséges adatokat, mint, amikor egy ügyfél hirdetést vásárol.

A Projekt tulajdonosnak lehetősége van Üzletkötőt, Grafikust és Fordítót regisztrálni. Ehhez meg kell adnia a felhasználó nevét, annak email címét és a saját jelszavát. Fontos, hogy a sajátját ekkor elkérjük, hogy ne lehessen egy

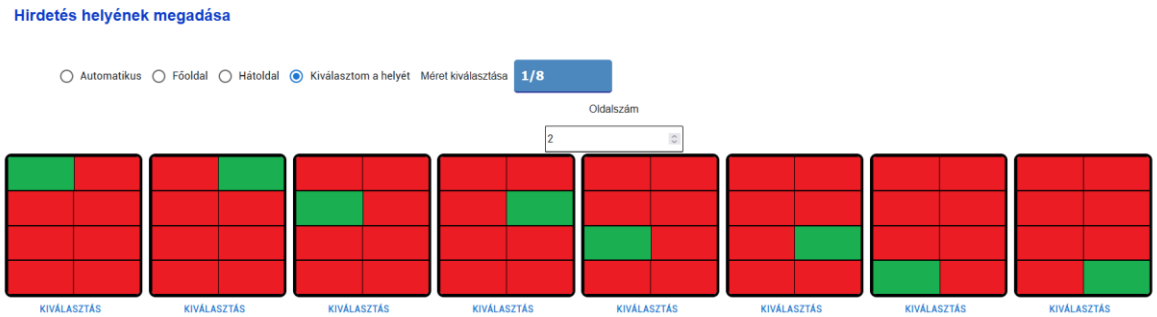
bejelentkezve hagyott gépen új felhasználót illetéktelenül regisztrálni. Feltűnhetett, hogy a regisztráció során nem állítottunk be jelszót az új felhasználónak. Ez azért van, mert ilyenkor automatikusan egy új jelszó kérő emailt küldünk az új felhasználónak. Addig ameddig nem állított be új jelszót, nem tud bejelentkezni az új felhasználó.

3.2.2 Ügyfél

Az ügyfelet bejelentkezésekor két viszonylag nagy méretű gomb fogadja, melyekkel meg tudja tekinteni eddig vásárolt hirdetéseit vagy hirdetést tud venni.

3.2.2.1 Hirdetés vétel

A hirdetésvásárlás folyamatán Accordionok vezetik végig a felhasználót. Minden lépés csak az előtte lévő lépés befejezte után következik.
Elsőként a lapszámot választhatja meg az ügyfél.
Ezt követően az elhelyezés módját, a fentebb már tárgyalt főoldal, hátoldal, automata, egyedi opciók közül. Meg kell adnia továbbá egy dropdown lista segítségével a hirdetés méretét.
Amennyiben az egyedi elhelyezést választja a vásárló, úgy a kiválasztott méret és oldalszám függvényében megjelenik egy egyedi elhelyező rendszer.



Elhelyező rendszer

A zöld téglalap jelöli a helyet, amelyet kiválasztunk a hirdetésnek. A “Kiválasztás” gombra kattintva az adott opció kerete zöld színű lesz jelezve, hogy kiválasztásra került az opció. Itt tartozom magamnak azzal a megjegyzéssel, hogy ezen mindenképpen változtatni kellene, mert web akadálymentesítési szempontból csak színekkel jelölni bármilyen változást hiba. Színtévesztők számára kimondottan

megnehezíti az oldal használatát, ezzel potenciális vásárlókat elveszítve. Mivel a webapplikáció webshop-nak minősül, ezért 2025. június 28-tól akár perbe fogható is lenne, mivel egy Európai Uniós törvény értelmében az említett dátumtól kezdve minden webshop-nak akadálymentesnek kell lennie.

Az elhelyező komponens implementációja a következő:

Számolás: Először is mindig a kiválasztott lapszám és oldalszám szerint lekérjük az adatbázisból az eddig meglévő hirdetéseket. Ezt követően minden egyes elfoglalható helyre megnézzük, hogy szabad-e. Ismerve, hogy a 2x4-es grid-en melyik helyek vannak szabadon, ellenőrzést végzünk, hogy a kiválasztott méretű hirdetés hova fér el.

Kirajzolás: Minden lehetséges elhelyezést kirajzolunk. Egy adott elhelyezés kirajzolásához mindig annyit piros téglalapot rajzolunk ki, amennyi szükséges a méret alapján. Az ⅛ oldal méretű hirdetésnél hetet, az ¼ oldal méretűnél hatot, ½ oldal méretűnél négyet. Ezeken belül a lehetséges helyre helyezzük a colstart, colend, rowstart, rowend segítségével a hirdetést jelző zöld téglalapot.

Ezt követően megadhatja weboldalának címét, amely az online hirdetési felületen tölt be funkciót. Ha a hirdetésre kattintanak megnyílik a hirdető weboldala egy új ablakban.

Amennyiben üzletkötő segítette az itteni vásárláshoz, megadhatja egy dropdown segítségével az üzletkötő nevét. A hirdető megadhatja, hogy szüksége van-e grafikusi munkára.

Amennyiben igen, úgy fel kell töltenie a képeket, amelyeket szeretne, ha rajta lennének a hirdetésen, de legalább egy cég logót el kell küldenie.

Amennyiben szeretne fordítási segítséget kérni, erre is van lehetőség, ekkor meg kell adni magyarul a szöveget, amelyet szeretne lefordíttatni. A grafikus és a fordító díja fix összegű.

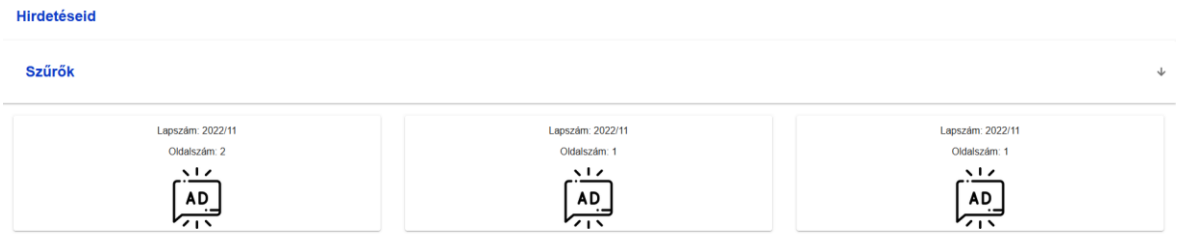
A folyamat végén kiírjuk a nyelvnek megfelelő pénznemben a fizetendő összeget és megjelenik egy PayPal fizetés ikon, amelyre kattintva a bejelentkezési adatainak megadását követően fizethet.

PayPal gombok a weboldalon

A fizetés jelenleg a PayPal Sandbox módjában valósul meg vagyis valódi fizetésre nincs lehetőség. Ennek integrációjához segítségemre volt a PayPal dokumentációja. Alapvetően létezik ennek könnyű integrációjára egy react-paypal-checkout névre hallgató npm csomag, ellenben az checkout.JS-t használ, amelyet a PayPal már nem támogat a Sandbox rendszerekben, úgyhogy az eredeti PayPal Javascript SDK-t kellett használnom. Ennek integrálása React alapú rendszerekbe finoman szólva sem illett a Clean kód elvekbe. useEffect segítségével egy futásidőben definiált script taggel tudtam egy előre definiált “paypalhere” id-val rendelkező <*div*>-be elhelyezni a gombokat. De legalább sikerült működésre bírnom. Egy újabb próbálkozással a @paypal/react-paypal-js csomag került integrálásra. Teljesen egyértelmű volt számomra, hogy sokkal olvashatóbb kódot eredményez a használata, így ez lett a végleges megoldás. Bővebben a használatáról a Külső könyvtárak fejezetben írok.

3.2.2.2 Hirdetések megtekintése

Természetesen mindenki szeret megbizonyosodni vásárlásának sikerességéről, éppen ezért szükségesnek érzem egy olyan felület létrehozását, ahol a hirdető ügyfél ellenőrizheti hirdetésének létezését. A felület egyszerű, nem rendelkezik sok funkcióval. Alapértelmezetten listázza az összes hirdetését az ügyfélnek. Megjelenítem a hirdetések lapszámát, a lapszámon belül az oldalszámot és magának a hirdetésnek a képét, hogy a felhasználó könnyen be tudja azonosítani melyik hirdetést látja éppen. Mindemellett, feltételezve, hogy az ügyfél több hirdetéssel is rendelkezik biztosítok egy “Szűrők” menüpontot, ahol jelenleg egy szűrési lehetőség van, lapszám szerinti.



Hirdetések megjelenítése az ügyfél számára

3.2.3 Üzletkötő

Az üzletkötő szerepe egyszerű. Lényegében “házaló”-król van szó, akik felveszik a kapcsolatot a Burgenland és környéki cégek vezetőivel és igyekeznek meggyőzni őket, hogy hirdessenek a hirdetőújságban. Az üzletkötők kifizetése jutalékos alapon működik. Az üzletkötők szerepe elengedhetetlen az újság indulásának kezdeti fázisaiban, hogy az első ügyfeleket megszerezhessük.

3.2.4 Grafikus

A grafikus feladata abban merül ki, hogy azon megrendelések esetén, ahol az ügyfél a hirdetés összeállításához grafikus segítségét kérte, összeállítson egy hirdetési képet, egyeztessen a vásárlóval emailben, hogy tetszik-e neki az összeállított hirdetés és ezt követően elküldje a Projekt tulajdonosnak a képet, hogy hirdetésként megjelenhessen.

3.2.5 Fordító

A fordító szerepe, hogy azon megrendelések esetén, ahol az ügyfél a hirdetés összeállításához fordító segítségét kérte, lefordítsa németre a megrendelő által elküldött szöveget, egyeztessen a vásárlóval emailben, hogy megfelel-e számára a szöveg fordítása, majd a lefordított szöveget elküldje a grafikusnak.

4. Frontend technológiák

4.1 A webapplikáció motorja: a React.JS

A React (más néven React.js vagy ReactJS) egy ingyenes és nyílt forráskódú kliens oldali JavaScript-könyvtár, amely felhasználói felületek UI-komponenseken alapuló létrehozására szolgál. A Meta (korábban Facebook) és az egyéni fejlesztőkből és cégekből álló közösség tartja karban. A React alapul szolgálhat egyoldalas, mobil vagy szerver által renderelt alkalmazások fejlesztéséhez is. A React azonban csak az állapotkezeléssel és az állapot DOM-nak való megjelenítésével foglalkozik, így ezen webalkalmazások létrehozásához általában további külső könyvtárak használatára van szükség a routing, valamint bizonyos kliensoldali funkciók megvalósításához.[3]

A React könyvtár használatakor az úgynevezett JSX nyelvet használjuk. Ennek lényege, hogy egyszerre használhatunk HTML és Javascript elemeket. Mi több stíluslapokat is definiálhatunk JSON formában. Minden komponensnek van egy render() metódusa, amely akkor kerül meghívásra, amikor megjelenítjük az adott komponenst. A render függvényben más Javascript függvények meghívása nem szerepelhet, csak állapotok aktuális értékeit tudjuk megjeleníteni. Mindemellett természetesen Javascript események bekövetkezésekor lefutó függvények meghívása lehetséges arrow függvények segítségével. Egy ilyen arrow függvény formája: *onClick={()=>{doSomething()}}*. Ha így definiálnánk: *onClick={doSomething()}*, úgy minden renderelésnél lefuttatná a függvényt, amely újabb rendereléseket idézne elő és egy végtelen ciklusba kerül a komponens az első kirajzolás alkalmával. A JSX nyelven íródott forráskód fájlokat szokás .jsx kiterjesztéssel ellátni, de nem szükséges, mivel .js kiterjesztéssel is lefordíthatóak.

A React.JS alapja a komponens alapú felépítés. Projekt tervezés szempontjából érdemes, mindig kis méretű, újra felhasználható egységekben gondolkozni. A könyvtár különlegessége, hogy bármely interakció, vagy adatmódosulás esetén csak azt a komponenst rendereli újra, amelyben a változás bekövetkezett, ezáltal kevésbé terheli a kliens oldalt. Minden komponens csak a

saját állapotait ismeri, viszont a felépítésben “lefelé” át lehet adni változókat, funkciókat. Ennek a megoldásnak a használata kényelmetlen lehet, ha sok komponensen keresztül kell átadni paramétereket, ennek megoldására fejlesztették ki a React Context koncepciót, melyet magam is használtam a webalkalmazás elkészítése során.[4]

4.2 useState

Természetesen JSX nyelven lehetséges normál Javascript változók létrehozása és kezelése, de javasolt a useState használata. A useState használatával lehetséges, hogy újrarenderelést idézzünk elő egy érték megváltozásakor. Egy *var*, *let* vagy *const*-al kapcsolatos változás esetén nem történik renderelés. Amikor useState-et használunk deklaráljuk a változót, illetve egy hozzá tartozó setter függvényt. Az érték megváltoztatása kizárólag a setter függvény meghívásával lehetséges.

4.3 useEffect

A React egyik fő előnye abban rejlik, hogy mindig csak azt a komponenst rendereli újra, amelyben állapotváltozás ment végbe. Ezzel kapcsolatban fogalmazódott meg a useEffect koncepciója, ami lényegében egy függvény futtatását teszi lehetővé állapotváltozáskor. A useEffectnek opcionálisan megadható egy dependencia tömb.

- Fontos, hogy ez csak opcionális, amennyiben nem adunk meg ilyet, abban az esetben minden render() függvényhíváskor lefut a benne meghatározott függvény.
- Ha üres tömböt adunk meg dependencia tömbként, akkor csak a komponens első kirenderelődésekor fut le a benne meghatározott függvény.
- A dependencia tömbben megadhatunk állapotokat. Ebben az esetben csak akkor fut le a benne meghatározott függvény, amikor az az adott állapot változik.

Gyakorlati haszna ennek jelen alkalmazásban például a kalkulátor működési

elve. Amikor változtatjuk a kiválasztott méretet, egy aszinkron függvény segítségével lekérjük a Firestoreból az adott mérethez tartozó árat a kiválasztott nyelv függvényében, majd beállítjuk a méret szerinti árat erre az értékre. Ugyanez történik, amikor az elhelyezés módját változtatjuk, csak abban az esetben nem konkrét árat állítunk be, hanem egy szorzót. Ez két külön useEffect. Ahhoz, hogy az eredményt kivetíthessük a felhasználónak nincs szüksége egy harmadikra, bár meg lehetne oldani annak segítségével is, hanem elegendő csak a kivetítés pillanatában összeszorozni a két értéket.

4.4 React Context

A React könyvtárról szóló bevezetésben már felmerült a paraméterek átadásának kérdése és az, hogy ez kellemetlen lehet több komponensen keresztüli átadások esetén. Ennek megoldása a React Context. A Context működési elve, hogy önmagában egy komponens, amely állapotokkal rendelkezik. A Provider-e segítségével ezeket az állapotokat a providerének gyermek komponensei közvetlenül képesek elérni. Egy React alkalmazáson belül lehetséges több context használata is. Egy context aszinkron módon is képes adatokat lekérni. Mindezen információk azért hasznosak, hogy tudjuk mikor kell contextet használni. Nem érdemes egy külső-belső rendszerre bontható webalkalmazásban az egész alkalmazásra kiterjeszteni. Ez nem csak a backend terhelése miatt lehet gond, mivel felesleges, nem használt adatokat kér le róla, de komoly biztonsági kockázatot is jelenthet, ha már a külső rendszert látó klienshez bizalmas adatok kerülnek egy context által. Fontos megjegyeznem, hogy nem érdemes minden komponensek közti paraméter átadásra conexteket használni, mert végül túl bonyolulttá válhat egy-egy context és bár lehet több contextet használni mégsem javasolt túl sokat, mert megnehezítheti a routing implementálását.

4.5 Error Boundary

Az error boundary-k hasonlóan a context-ek providerei-hez wrapperként használhatók komponensek körül. Definiáltam neki egy hasError állapotot, amely alapértelmezett értéke false. Ha bármely gyermek komponensében hiba következik

be, amely nem kerül kezelésre a komponensen belül, akkor ennek a szülő komponensnek a getDerivedStateFromError() függvénye kerül meghívásra. Ebben fejlesztési környezetben kiírhatjuk a konzolra az error object-et, de ami még fontosabb, hogy a hasError állapotot true-ra állíthatjuk. Az általam írt error boundary render() metódusa jelen webapplikáció esetén egyszerű. Amennyiben a hasError false renderelje ki a gyermek komponenseket, viszont ha a hasError értéke true, akkor kirenderel egy gombot, amelyre ha rákattintunk betölti a főoldalt.

```
      render() {
          if (!this.state.hasError) {
            return this.props.children;
          }
          return (
            <div>
              <Button
                onClick={() => {
                  window.location.replace(window.location.origin +
"/");

                }}
              >
                Vissza a főoldalra
              </Button>
            </div>
          );
        }
```

Az ErrorBoundary render() metódusa

4.2 Külső könyvtárak

4.2.1 react-router-dom

A JavaScript keretrendszer JavaScript kódkönyvtárak gyűjteménye, amelyek előre megírt kódot biztosítanak a webfejlesztőknek a rutin programozási feladatokhoz. A keretrendszerek meghatározott kontextusú struktúrák, és segítenek webalkalmazások létrehozásában ebben a kontextusban.

Míg a JS-keretrendszer egy teljes eszközkészlet, amely segít a webhely vagy az alkalmazás kialakításában és rendszerezésében, a JS-könyvtár ezzel szemben olyan előre megírt kódrészletek gyűjteménye, amelyek kevésbé az alkalmazás alakításáról, hanem inkább igény szerint használható funkciók biztosításáról szól.

A React nem keretrendszer! Ezzel legelőször akkor szembesülünk, amikor onepager helyett egy több oldalból álló weboldalt szeretnénk készíteni. Beépített megoldással nem rendelkezik a react arra, hogy megoldjon több lap közti navigálást. *Switch-case* és *if* elágazások segítségével lehetséges az implementálása, de sok lap esetén átláthatatlan kódot eredményez. Ekkor jön a képbe a react-router-dom. A működése illik a React alapelvéhez. Amikor például a /login-ról a /about-ra navigálunk, akkor a fejléc és lábléc nem renderelődik újra, a router csak a Login komponens helyett kirajzolja az About komponenst. Így nem kell újratöltődnie az oldalnak, ami nagyságrendekkel jobb felhasználói élményt biztosít, gyorsabbá válik az oldal használata, ezáltal a felhasználók nagyobb eséllyel vásárolnak hirdetést itt.

Magam ennél a csomagnál éreztem először az informatika gyors változásainak szelét. Amikor a Webes rendszerek fejlesztése 1. tárgyat hallgattam még az 5.x.x verzió volt életben a saját használati módjával. Amikor elkezdtem React fejlesztéssel munka szinten foglalkozni nem sokkal később, már a 6.x.x verzió volt életben és a használatát tekintve szinte semmiben nem egyezett az 5.x.x verzióval.

A webapplikáció fejlesztése során a következőképpen használtam:

- Elsőként az egész applikációt wrappelni kell egy BrowserRouter-rel.
- Ezt követően ebbe a BrowserRouter-be beleraktam a fejlécet, egy Routes komponenst és a láblécet. A router csak a Routes-on belül végez módosításokat, így értem el, hogy a fejléc és lábléc ne frissüljön másik oldalra navigáláskor. A Routes komponens csak Route komponenseket tartalmazhat.
- Egy Routenak meg kell adni egy “exact path”-t például “/login”, ha a bejelentkező felületet szeretnénk elérni.
- Meg kell adni továbbá egy element prop-ot, amely a bejelentkező

felület esetén a Login komponens.

Az előző bekezdésben említettem egy fontos szabályt mégpedig: “A Routes komponens csak Route komponenseket tartalmazhat.”. Jelen web applikációnk rendelkezik felhasználókezeléssel, ebből következően vannak bizonyos oldalak, amiket csak bejelentkezett felhasználók esetén szeretnénk, ha elérhető lenne. Ehhez hasznos lenne írni egy PrivateRoute komponenst, amely megvalósítja a nem bejelentkezett látogatók korlátozását. A komponens működése egyszerű, prop-ként kapnia kell egy komponenst, amit szeretnénk, ha kirenderelne, amikor van bejelentkezett felhasználó. Ha nincs ilyen bejelentkezett felhasználó, akkor rendereljen ki egy Navigate komponens, amely a “/” oldalra irányítja az oldalt. Ez a PrivateRoute beilleszthető a Routes komponensbe? Nem, mert nem egy Route komponens! Így workaround-ot kell alkalmazni, ami kimerül abban, hogy egy normál Route komponenst használunk, element-ként megkap egy PrivateRoute-ot, amelynek komponensként odaadjuk a kirenderelni kívánt oldalt. Ez szintén jó példa a Higher-Order komponensek használatára.

4.2.2 @mui/material

A Material UI egy nyílt forráskódú React komponenskönyvtár, amely megvalósítja a Google Material Design rendszerét. Az előre elkészített komponensek átfogó gyűjteményét tartalmazza, amelyek azonnal használatra készek. A Material UI kialakítása gyönyörű, és testreszabási lehetőségeket kínál, amelyek megkönnyítik a saját egyedi tervezési rendszerének megvalósítását. Több mint 2500 open source contributor fektetett számtalan órát ezekbe az összetevőkbe. Szervezetek ezrei bíznak meg benne: A Material UI rendelkezik a legnagyobb felhasználói felület közösséggel a React ökoszisztémában. Majdnem ugyanannyi idős, mint maga a React – története egészen 2014-ig nyúlik vissza.[5]

A webapplikáció készítésekor nagy segítségemre volt ez a design könyvtár. Összességében nyugodt szívvel merem állítani, hogy a fejlesztési folyamatot nagyon fel tudja gyorsítani egy ilyen könyvtár használata. Természetesen nem biztosít egyedi megjelenést egy weboldalnak, de a fejlesztő válláról sok terhet le tud venni azzal, hogy nem kell apró, pixel szintű design problémákkal foglalkoznia és

az idejének meghatározó részét a programlogika írásával tudja tölteni.

Eredetileg nem csak a Material UI könyvtárát használtam, hanem több másikat is egyszerre. Ez hasznos lehet, ha két könyvtár stílusjegyeit szeretném összeolvasztani. Ellenben okozhat gondot a kódkiegészítés használatakor és az esetleges azonos osztálynevek használata is problémákat jelenthet. Mivel számomra szinte megmagyarázhatatlan CSS problémákba ütköztem, ezért áttértem az egy stíluskönyvtár használatra, amely megoldotta ezeket a problémákat.

4.2.3 @paypal/react-paypal-js

Miért érdemes a react-paypal-js-t használni?

A PayPal-lal integráló fejlesztőknek hozzá kell adniuk a JS SDK <script>-et egy webhelyhez, majd a szkript betöltése után olyan komponenseket kell előállítaniuk, mint a PayPal gombok. Ez az architektúra nagyszerűen működik egyszerű webhelyeken, de kihívást jelenthet egyoldalas alkalmazások készítésénél. Egyoldalas alkalmazás alatt értem, hogy az előbbiekben taglalt Router segítségével váltunk a komponensek között és nem töltjük újra az egész oldalt, ha új oldalra akarunk navigálni.

A React fejlesztők az komponensekben gondolkoznak, nem pedig a külső szkriptek index.html fájlból történő betöltésében. A PayPal gombok megvalósítási részleteinek egyetlen React komponensbe való absztrahálása anti-pattern, mert szorosan összekapcsolja a szkriptek betöltését a rendereléssel. Ez akkor is problémás, ha több különböző PayPal-összetevőt kell renderelni, amelyek ugyanazokat a globális szkriptparamétereket használják.

A react-paypal-js megoldást kínál a fejlesztőknek a JS SDK betöltésével kapcsolatos bonyolultságok elvonatkoztatására. Alapértelmezés szerint érvényesíti a bevált gyakorlatokat, így a vásárlók a lehető legjobb felhasználói élményt kapják.[6]

Használatát tekintve létrehoztam egy külön komponenst a PayPal gombok

számára. Ebben a komponensben használtam az úgynevezett *usePayPalScriptReducer()*-t, amelyből hasznos információkat nyerhetek ki az aktuális tranzakcióval kapcsolatban. Én ebből az *isPending* változót használom arra, hogy, amikor a tranzakció egy felugró ablakban folyamatban van, kirajzoljak egy animációt, amely a felugró ablakban futó tranzakcióra való várakozást jelzi. Szintén használtam a *PayPalButtons* komponenst. Ennek megadhatunk egy createOrder függvényt, amelynek van egy actions paramétere és visszatérési értéke egy actions.order.create felparaméterezve a megfelelő pénzem és pénzmennyiség értékével. Szintén adhatunk neki egy onApprove függvényt, ahol pedig kezelhetjük mi történjen a fizetés sikeres bekövetkezése után. Ahhoz, hogy az általam létrehozott komponenst használhassam egy *PayPalScriptProvider*-be kell csomagolnom. Ennek a provider-nek meg kell adnom egy options prop-ot, ami egy JSON, amely tartalmazza a client-ID-t, hogy milyen beépített PayPal komponenseket használok benne (esetünkben “buttons”) és az aktuális pénznemet.

4.2.4 firebase

Az eddigiekben felmerülhetett a kérdés, hogy a frontend és backend kapcsolata miként valósul meg. A Firebase biztosít egy firebase névre hallgató csomagot, amely könnyedén megvalósítja a kapcsolat kialakítását. A frontend-backend kapcsolat kialakítását és vezérlését egy külön firebase.js fájlban valósítom meg. Fontos, hogy ez nem egy React komponens, ez egy egyszerű .js fájl.

A firebase működése ezen csomag használata esetén nem igényli, hogy külön címzett API-kkal kommunikáljak, helyette “firebase-app”-ot kell létrehozni. A csomag tartalmaz egy initializeApp() függvényt, amely paraméterül kap egy firebaseConfigot. Ezt a konfig gyakorlatilag egy JSON, amelyet a Firebase webes felületéről tudunk letölteni a projektünkhöz. Tartalmazza az API Key-t, auth domaint, a Firebase-en belüli projectID-t és a storage bucket azonosítóját. Mivel ugyanazon projekten belül lehet több applikációnk is például: Android, iOS, Web, Unity, ezért egy appID-t is tartalmaz, hogy be tudja azonosítani, melyik applikációt használjuk éppen.

Többféle gyakorlat is létezik a firebase kezelésére az alkalmazáson belül. Én egy központi helyen, a firebase.js-ben szoktam kezelni a Firebase-el kapcsolatos

“standard” függvényeket. Standard alatt az authentikációs függvényekre, fájl feltöltésre, bármilyen olyan függvényre, amit több helyen is használhatok, gondolok. Ennek előnye, hogy az esetek többségében több projekt között ezek a függvények megegyeznek, elegendő átmásolnom egyik projektemből a másikba, bekonfigurálni a firebaseConfigot és máris kulcsrakész a Firebase kezelése. Ezek a függvények a firebase csomag beépített függvényeit használják, de tudnak kezelni egyedi projekt specifikus paramétereket, több beépített függvény együttes hívását és kidolgozott hibakezeléssel rendelkeznek.

Több beépített függvény együttes hívására a legjobb példa az ügyfél regisztráció. Ez a függvény 3 paramétert kap: emailcím, jelszó, név. Egy try-catch-en belül zajlik minden művelet. Amennyiben hiba következik be a hibaüzenetet egy window.alert() formájában megjelenítem. elsőként a createUserWithEmailAndPassword() függvénnyel létrehozom a felhasználót a Firebase Authentication-ön belül. Ennek a függvénynek 3 paramétert kell adni: az auth modult, amit egy getAuth(), firebase/auth-ból exportált függvénnyel kapok meg. Ezen felül az email cím és jelszó párosra van szükség és a felhasználónk már létre is lett hozva. A következő függvény az updateProfile(), amellyel a felhasználó displayName-ét írom felül a megadott névre. Ez a függvény 2 paramétert kér, az auth.currentUser-t, amely a jelenleg bejelentkezett felhasználó objektuma és egy JSON-t amely tartalmaz egy displayName kulcsot, amelynek a név értékét adjuk. Felmerülhet a kérdés, hogy ki a jelenleg bejelentkezett felhasználó? A createUserWithEmailAndPassword() különlegessége, hogy a felhasználó létrehozását követően azonnal be is jelentkezteti. Mindezt követően meghívunk egy setDoc() függvényt. A setDoc(), ahogy neve is takarja már nem a Firebase Authentication szolgáltatással lesz kapcsolatos, hanem beállít egy dokumentumot a Firestore NoSQL adatbázisban. A setDoc() 2 paramétert vár. Egy doc() függvénnyel generált DocumentReference objektumot és egy JSON-t. A JSON gyakorlatilag a dokumentum szerkezetét kell tartalmazza, esetünkben ez csak egy role kulcs és, mivel ügyfelet hoztunk létre az értéke “Customer”. A doc() függvény 3 paramétert vár. Az adatbázist, amit a getFirestore(), firebase/firestore-ból exportált függvénnyel kapok meg. A collection nevét, esetünkben “Users” és a dokumentum nevét, a mi esetünkben a felhasználó email címe.

4.2.5 i18next és segédcsomagjai

Az alkalmazás többnyelvű. Ennek kezelésére az i18next csomagot hívtam segítségül. Az i18next egy nagyon népszerű nemzetköziesítési keretrendszer böngészőkhöz vagy bármely más javascript környezethez (például: Node.js, Deno).[7] Az i18n elnevezés az internationalization szót takarja, rövidítve, hogy az első “i” és utolsó “n” betű között 18 karakter szerepel.

Használata jelen esetünkben összesen 4 csomag telepítését eredményezte, ezek: i18next, react-i18next, i18next-browser-languagedetector, i18next-http-backend.

```
i18n
  .use(initReactI18next)
  .use(LanguageDetector)
  .use(HttpApi)
  .init({
    detection: {
      order: ["cookie", "htmlTag"],
      caches: ["cookie"],
    },
    backend: {
      loadPath: "/assets/locales/{{lng}}/translation.json",
    },
    react: {
      useSuspense: false,
    },
    fallbackLng: "hu",
  });
```

Az i18n konfigurációja

Az i18n-t az i18next csomagból importáljuk. Inicializálónak megadjuk az initReactI18next-et, amelyet a react-18next-ből kapunk. Ezt követően használjuk a LanguageDetector-t és a HttpApi-t majd következik az inicializálás. Elsőnek meghatározzuk, hogy a LanguageDetector honnan próbálja megszerezni a betöltendő nyelvre vonatkozó információt. Elsőként egy i18next névre hallgató sütit

keres ennek értéke “de” vagy “hu” lehet, mivel német és magyar nyelven érhető el az oldal. Amennyiben ez a süti még nem létezik, vagyis az adott böngészőben először nyitották meg az oldalt, úgy a <html> tag “lang” attribútumának értékét veszi alapul, ami “de”. Amint ezt az értéket megtalálja felülírja az i18next sütit az éppen aktuális beállításra. Amikor nyelvet változtatunk a fejlécben található gombok segítségével ez a süti szintén felülíródik.

A backend kulcs a HttpApi-hoz kapcsolódik. Gondoljunk bele abba, hogy, ha az alkalmazásunk rengeteg nyelvet támogat, rengeteg string van benne, akkor a fordítási fájlok igencsak figyelemre méltó méretet tudnak felvenni. Ha mindezt betöltjük kliens oldalon, amikor valaki megnyitja a weboldalunkat, feleslegesen nagy sávszélességet használunk, lelassítva ezzel a weboldal betöltését, rontva a felhasználói élményt. Az i18next-http-backend egy egyszerű i18next háttérprogram, amely a Node.js-ben, a böngészőben és a Deno-ban is használható. Az erőforrásokat egy háttérkiszolgálóról tölti be az XMLHttpRequest vagy a fetch API használatával.[8] A lényege, hogy mindig csak a szükséges fordítási fájlokat tölti be a kliensre. Így elkerülhető, hogy akár húsz nyelv esetén mind a huszat betöltse. Jó eséllyel két nyelv betöltése lesz a maximum, aminek meg kell történnie, ha az alapértelmezett nyelv nem a felhasználó anyanyelve. Nem mellesleg, mivel a Firebase Spark plan-je 360 MB adatforgalmat engedélyez naponta, ezzel a megoldással jelentősen javítjuk az esélyeinket, hogy ne lépjük ezt túl. Visszatérve a backend kulcsra: ez a fordítási fájlok betöltési útvonalát takarja. A projekt public mappájába, ami elérhető nyilvánosan, létrehoztam egy assets mappát, ezen belül egy locales mappát, majd egy “de” és egy “hu” mappát és ezeken belül egy-egy translation.json-t. A fordítások JSON alapon vannak tárolva kulcs érték párokkal. Például a Rólunk oldal szövege az “AboutText” kulcsot kapta és minden nyelv translation.json fájljában meg van határozva egy “AboutText” kulcs, amely mögött az adott nyelven írjuk le az információkat.

4.2.6 react-icons

A React-projektekbe egyszerűen beilleszthetjük a népszerű ikonokat az ES6-importálást használó react-icons csomaggal, amely lehetővé teszi, hogy csak azokat az ikonokat importáljuk be, amelyeket a projekt használ.[9]

Ez a csomag rendkívül hasznos, mivel összesítve tartalmazza számos ikon tervezéssel foglalkozó szervezet ikonjait. Ilyen szervezetek közé tartozik a Font Awesome, az Ant Design, Grommet, Tabler, hogy csak azokat említsem, akik ikonjait használtam. Bár több stílus került egyszerre felhasználásra, de ezzel nincs gond, ameddig nem inkonzisztens miatta a felhasználói felület. Egyébként célszerű azonos designon belül maradni, de nem minden készítő alkotott minden helyzetre ikont, így megeshet, hogy emiatt kell másik ikon szolgáltatót használni. A csomag minden ikont .svg formátumban tárol, így pixelesedés miatt nem kell aggódni. Ha nem akarunk vagy nem tudunk ikonokat rajzolni javaslom ennek a könyvtárnak a használatát.

4.2.7 react-password-checklist

React komponens, amely megjeleníti a jelszóerősségi szabályok sikerességét vagy sikertelenségét, amely a felhasználó gépelésekor frissül.[10]

Munkáim során többször is használtam már ezt a csomagot, nagyon gyorsan meg lehet valósítani vele, jó minőségben a jelszó validációt. Be lehet konfigurálni, hogy milyen jelszót fogadunk el. Megadhatjuk, hogy minimum milyen hosszú legyen, tartalmazzon speciális karaktert, számot, valamint kis és nagybetűt. Amikor a feltételek teljesülnek egy zöld pipával, ha nem teljesülnek piros “X”-el jelzi. Mindezekhez tartozik egy leírás, mint például “A jelszónak minimum 6 karakter hosszúnak kell lennie”, ezeket is testre tudjuk szabni. A kritériumokot leíró szövegeket könnyedén tudjuk lokalizálni az i18next csomag segítségével.

5. Fejlesztési eszközök

5.1 Visual Studio Code

A Visual Studio Code (rövidítve: VSCode vagy VS Code) ingyenes, nyílt forráskódú kódszerkesztő, melyet a Microsoft fejleszt Windows, Linux és OS X operációs rendszerekhez. Támogatja a hibakeresőket, valamint beépített Git támogatással rendelkezik, továbbá képes az intelligens kódkiegészítésre az IntelliSense segítségével. A VSCode-ban a felhasználók megváltoztathatják a kinézetet (témát), megváltoztathatják a szerkesztő gyorsbillentyű-kiosztását, az alapértelmezett beállításokat és még sok egyebet. Támogatja a kiegészítőket, melyek segítségével további funkciók, testreszabási lehetőségek érhetőek el.

A VSCode az Electron nevű keretrendszeren alapszik, amellyel asztali környezetben futtatható Node.js alkalmazások fejleszthetőek. A Visual Studio Code a Visual Studio Online szerkesztőn alapszik (fejlesztési neve: "Monaco").[11]

A Visual Studio Code támogatja a git integrációt ezzel megkönnyítette számomra a verziókövetés kivitelezését.

Támogat extension-öket, a fejlesztés során hasznomra vált az ES7+ React/Redux/React-Native snippets, amely lényegében kódkiegészítést biztosított. Például egy új komponens létrehozásánál a “rafce” kódból generált egy komponenst, amely importálta a React-ot, default exportja volt és egy -ben kiírta a komponens nevét. Hasznos például egy egyszerű useState() megírásakor is, mivel, amikor legenerálja a snippet-et automatikusan fókuszba helyezi a változó nevét és a setter függvény nevén belül is a változó nevét. Ezeket egyszerre szerkeszthetem, majd egy Tab megnyomására az állapot alapértelmezett értékét is azonnal megadhatom.

5.2 GitHub

A GitHub, Inc. egy egyesült államokbeli nemzetközi vállalat, amely a Git

segítségével szoftverfejlesztési verziókövetés-szolgáltatást nyújt. 2018-ban a Microsoft leányvállalata lett 7,5 milliárd dollárért. Saját funkcióin felül a Git elosztott verziókövetését és forráskódkezelését (SCM) teszi elérhetővé. Hozzáférés-kezelést és számos együttműködési funkciót nyújt, mint például hibakövetés, szolgáltatáslekérés, feladatkezelés, valamint wikiket minden projekthez.[12]

A verziókövetésre való igény szinte egyidős a programozással. Nemcsak egy jó backup lehetőséget biztosít adatvesztés esetére, de visszaállításra is kitűnően használható. Gondoljunk bele, hogy ki akartunk javítani egy hibát, de anélkül, hogy észrevettük volna egy sokkal nagyobb hibát okoztunk. Ez mondjuk egy fizetési rendszernél megengedhetetlen és ilyenkor célszerű visszaállni a régi, kevésbé hibás rendszerre, amíg meg nem oldjuk a hiba elhárítását.

A git lehetőséget nyújt branchek létrehozására, bár esetemben erre nem igazán volt szükség, mivel egyedül dolgoztam a funkciókon. Ennek ellenére nagyon hasznos lenne olyan esetben, ha valamely később tárgyalt továbbfejlesztési lehetőséget megvalósítanám, úgy, hogy már hosztolva van a weboldal, hiszen úgy még véletlenül nem idézhetek elő hibát az eredeti kódban. Hasznos lehet arra, hogy ameddig egy új feature-t fejlesztek és bejön egy bugról egy értesítés, akkor a bugot, gyorsan ki tudom javítani anélkül, hogy összefonódna az új feature kiadásával. Szintén jó arra is, hogy, ha az új funkció máshol kihosztolom User Acceptance Test lehetőséget biztosítok mondjuk a Projekt tulajdonos számára.

Igyekeztem magam számára is hasznos commit üzeneteket írni, a konzultációs alkalmakkor ezek alapján tudtam beszámolni az elkészült újításokról és a kijavított hibákról.

6. Továbbfejlesztési lehetőségek

A webapplikációban rengeteg kiaknázatlan lehetőség van. Felsorolásszerűen tárgyalok néhány időközben megfogalmazódott ötletet.

Az üzletkötők jutalékos rendszerben kapnak fizetést. Szükséges lehet ehhez egy felület, ahol minden üzletkötőnek egyénileg beállítható, hogy %-át kapja az ő segítségével kötött üzletek árának. Az üzletkötőkkel kapcsolatban létre kellene hozni egy olyan felületet a Projekt tulajdonos számára, amelyen kaphat egy áttekintést arról, hogy mely üzletkötői teljesítettek a legjobban, melyiküknek, hány szerzett ügyfele volt, ki hozta a legmagasabb profitot.

Az előbbi felület logikáján kiindulva egy konkrét költségtervező felület is kiépítésre kerülhetne, ahol nyomtatási költség, egyéb felmerülő költség, szállítási költség szinten lebontva láthatná a Projekt tulajdonos, hogy hol faraghatna lejjebb a költségein.

Lehetne használni a Google Analytics szolgáltatást, hogy információt kapjunk arról, hogy hányan látogatják az oldalunkat, mely térségekből. Következtetéseket vonhatunk le, hogy esetleg mely területekre lenne érdemes még újságot kiszállítani.

Szükséges lehet a grafikus és fordító díjának szerkesztése. Sőt, ha már ezen két szerepkörnél tartunk, ezek kommunikációját az ügyféllel meg kellene könnyíteni azáltal, hogy nem külön emailben kommunikálnak az ügyféllel, hanem egy teljes üzenetküldő rendszert implementáhatnánk.

Hogy bíztatóbbak és kedvesebbek legyünk a vásárló számára, beépíthetünk üdvözlő, “köszönjük a vásárlást” és ehhez hasonló emaileket. Mindehhez csak egy saját SMTP szerverre van szükség, amelyet a DiMa.hu Kft. képes biztosítani és Firebase Functions funkciók segítségével, nodemailer használatával képesek vagyunk emailek küldésére. Promóciós emaileket ugyanezen módon lehetne beintegrálni a rendszerbe. Szintén a promóciós emailekkel kapcsolatban lehetne

egy külön felületet biztosítani a projekt tulajdonos számára, ahol ezeket az emaileket tudja szerkeszteni, újakat tud összeállítani. A működési elve az lenne, hogy az általa összeállított emaileket Firestore-ba lementjük és a promóciós emailt küldő CRON job, pedig innen emelné ki az email tartalmát.

A vásárlás meneténél hasznos lenne, ha egy vásárlás alkalmával több hirdetést is lehetne venni, hiszen lehet, hogy a hirdető minden tavaszi hónapban egy akció keretén belül más-más hirdetéseket jelenítene meg, viszont szeretné egyszerre megvásárolni. Hasonlóan ehhez lehetne olyan opció, hogy az adott hirdetés több lapszámban is megjelentessük. A példánál maradva, lehet, hogy egy kertészeti bolt egész tavasszal szeretne a címlapon maradni. Az ilyen kombinált vásárlásokhoz lehetne bizonyos akciókat kötni, ezeknek az akcióknak is lehetne konfigurálási felületet biztosítani a Projekt tulajdonos számára.

A bejelentkezési módokat tekintve hasznos lenne, ha nem csak email - jelszó kombinációval lehetne regisztrálni, hanem akár a Google, Facebook fiókunkkal is. Bevezethetnénk megerősítő emaileket, ahogy 2 faktoros azonosítást is.

A Projekt tulajdonos részére lehetne biztosítani egy “Új felhasználó regisztrálása” Accordion-t, ahol a Projekt tulajdonos maga választhatja ki az új felhasználó szerepkörét (Üzletkötő, Grafikus, Fordító). Ezzel egy Accordionba be tudjuk tömöríteni a jelenlegi 3 Accordiont.

Mivel a projekt jelenleg manuálisan kerül buildelésre és manuálisan frissítem a hosztolt oldalt, jó lenne egy automatizált CI/CD folyamat kialakítása.

A vásárlásnál az egyedi elhelyezés opciót választva a több felugró lehetőség helyett egy olyan 2x4-es gridet kellene kirajzolni a felhasználónak, ahol az adott mezőre kattintással tudja kiválasztani a hirdetésének helyét.

7. Irodalomjegyzék

1: Római Birodalom - Acta Urbis

https://romaikor.hu/kislexikon/kislexikon_(kozigazgatas_es_kormanyzas)/cikk/acta__urbis

Látogatási idő: 2022. 10. 20.

2: Firebase Documentation

https://firebase.google.com/docs

Letöltés dátuma: 2022. 08. 12.

3: React (Javascript Library)

https://en.wikipedia.org/wiki/React_(JavaScript_library)

Letöltés dátuma: 2022. 07. 28.

4: React – A JavaScript library for building user interfaces

https://reactjs.org/

Letöltés dátuma: 2022. 10. 13.

5: Material UI - Overview

https://mui.com/material-ui/getting-started/overview/

Letöltés dátuma: 2022. 11. 22.

6: @paypal/react-paypal-js - npm

https://www.npmjs.com/package/@paypal/react-paypal-js

Letöltés dátuma: 2022. 09. 12.

7: i18next - npm

https://www.npmjs.com/package/i18next

Letöltés dátuma: 2022. 11. 22.

8: i18next-http-backend - npm

https://www.npmjs.com/package/i18next-http-backend

Letöltés dátuma: 2022: 10. 15.

9: react-icons - npm

https://www.npmjs.com/package/react-icons

Letöltés dátuma: 2022: 10. 18.

10: sators/react-password-checklist: A React Component to display the success or failure of password strength rules

https://github.com/sators/react-password-checklist#readme

Letöltés dátuma: 2022: 11. 21.

11: Visual Studio Code – Wikipédia
https://hu.wikipedia.org/wiki/Visual_Studio_Code
Letöltés dátuma: 2022: 09. 29.
12: GitHub – Wikipédia
https://hu.wikipedia.org/wiki/GitHub
Letöltés dátuma: 2022: 11. 29.