

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Drexler Nándor

**Ingatlan hirdető portál megvalósítása React és .NET Core alapokon**

Dr. Ekler Péter

BUDAPEST, 2020

Tartalomjegyzék

[Összefoglaló 5](#_Toc55748316)

[Abstract 6](#_Toc55748317)

[1 Bevezetés 7](#_Toc55748318)

[1.1 Felhasznált technológiák jelentősége 7](#_Toc55748319)

[1.2 Szakdolgozat tartalma 9](#_Toc55748320)

[2 Feladatspecifikáció 11](#_Toc55748321)

[2.1 Feladat részletes leírása 11](#_Toc55748322)

[2.2 Használati esetek 12](#_Toc55748323)

[2.2.1 Hirdetések keresése 12](#_Toc55748324)

[2.2.2 Hirdetés feladása, módosítása 13](#_Toc55748325)

[2.2.3 A hirdető adatlapja és értékelése 15](#_Toc55748326)

[2.2.4 Use Case diagram 16](#_Toc55748327)

[3 Felhasznált technológiák 17](#_Toc55748328)

[3.1 MongoDb 17](#_Toc55748329)

[3.2 ASP .NET Core 19](#_Toc55748330)

[3.2.1 MongoDb .NET Driver 20](#_Toc55748331)

[3.2.2 ASP .NET Core Identity 20](#_Toc55748332)

[3.2.3 ASP .NET Core Identity MongoDbCore 21](#_Toc55748333)

[3.3 React 21](#_Toc55748334)

[3.3.1 JSX 22](#_Toc55748335)

[3.3.2 React Hooks 23](#_Toc55748336)

[3.3.3 React Router 23](#_Toc55748337)

[3.4 Bootstrap 24](#_Toc55748338)

[3.4.1 Reszponzív weboldalak 25](#_Toc55748339)

[3.4.2 Bootstrap Grid System 25](#_Toc55748340)

[3.5 Google Maps 26](#_Toc55748341)

[3.5.1 Google Maps React 27](#_Toc55748342)

[4 Felsőszintű architektúra 29](#_Toc55748343)

[4.1 Háromrétegű architektúra 29](#_Toc55748344)

[4.1.1 Rétegek 29](#_Toc55748345)

[4.2 Szerver oldali komponensek 30](#_Toc55748346)

[4.3 Kliens oldali komponensek 31](#_Toc55748347)

[5 Megvalósítás részletes bemutatása 33](#_Toc55748348)

[5.1 Szerver oldal 33](#_Toc55748349)

[5.1.1 Modell 33](#_Toc55748350)

[6 Irodalomjegyzék 34](#_Toc55748351)

[Függelék 37](#_Toc55748352)

Hallgatói nyilatkozat

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

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

Kelt: Budapest, 2020. 10. 08

...…………………………………………….

Drexler Nándor

Összefoglaló

Az ingatlanvásárlás mindenki életebén egy meghatározó mérföldkő. Ahány ember annyi különböző elképzelés a megfelelő otthonról, nem beszélve a jelentős összegekről, akár hitelekről, amelyek a vásárlót hosszú évekre röghöz kötik. Ugyanakkor az eladói oldalon is jelentős kihívás megtalálni a megfelelő vevőt, aki értékeli az ingatlan adottságai, belefektetett munkát és költségeket, ennek következtében, alkudozások helyett hajlandó kifizetni egy a tervezett eladási árnak megfelelő összeget. Manapság már általában nem (csak) az újságból vagy ingatlanirodában tájékozódnak az emberek, hanem az interneten, így az ingatlanreferensek sem mehetnek el szótlanul az internetes hirdetőportálok mellet.

A „RealState” webalkalmazás a fent említett kihívásokban próbál segíteni mind a vásárlóknak, mind az eladóknak. Részletes keresési lehetőséget kínál, ahol nem csak adatok alapján kereshetünk, hanem elhelyezkedés alapján is. Miután a vásárlók nem csak puszta adatok alapján hozhatók lázba, ezért kedvcsináló képeket tölthetünk fel a hirdetésekhez. Illetve, abban az esetben, ha nagyon elburjánzik az ingatlanok kínálata, nem valószínű, hogy minden oldalt végig lapoznak az érdeklődők. Virtuális kreditekért cserébe kiemelhetjük, még több érdeklődőt megtalálva ezzel. Közvetlenül a portálon írhatunk üzenetet a hirdetőknek, megkönnyítve ezzel a kapcsolatfelvételt.

Egy jól használható hirdetőportál hamar népszerűségre tehet szert a felhasználók körében. A felhasználó elégedettség itt nem csak a konkrét felhasználói élményben mérhető. Másszóval nem elegendő, hogy gyorsan betöltsön az oldalam, illetve intuitív felületek legyenek. Vevői szemszögből meghatározó lehet, hogy mennyi idő alatt találtam nekem megfelelő ingatlant, illetve eladó szemszögből, hogy milyen áron, mennyi idő alatt tudtam eladni az ingatlanom. Ezen úgy lehet segíteni, hogy amellett, hogy az alkalmazás kedvező felhasználói élményt ad, fontos a folyamatos fejlesztés, az új funkciók hozzáadása, hogy az aktuális trendeket követve megtartsuk hosszútávon a felhasználóbázisunkat és mindig legyen megfelelő mennyiségű hirdetés az oldalon.

Abstract

Purchasing a real estate is remarkable achievement in everyone’s life. Every single person has different desires of a comfortable home. The enormous costs makes people take loans, forcing them to stay in their chosen apartment for decades. Sellers are facing the challange of finding clients who actually acknowledges the features of the apartment and also the invested money and time to build or renovate the certain estate. Therefore the client leaves the haggling and offers a reasonable price for the estate. Nowadays people favor browsing estates on the internet instead of looking for them in the newspaper or going to real estate agency. Real estate agents have to take the online advertisement possibilities into account.

The „RealState” web application offers help for the challanges mentioned above, both the client and seller side. It provides a various search features, not only for certain properties of the estates, but allowing to show them by their location on the map. Since most clients cannot be attracted by promising detailed information, sellers have the ability to upload spectacular pictures, which cheers up the client’s interest. In most cases, while clients are browsing on the internet, they won’t check all the pages of their search’s results due to lack of time. In exchange of virtual credits, we can highlight our advertisement in order to make it reach more people. Finally, we can contact the seller by sending a message directly on the page.

A well-designed, easy to use real estate portal can soon achieve a high popularity among users. Here, the user’s satisfaction does not only consist of good user experience. In other words a fast, responsive and intuitive web page is not enough. From the buyers point of view the time required to find a real estate which appeals to them is really significant. This amount of time can be reduced by certain features, however, these features are only effective if we maintain the necessary amount of active advertisements on the portal and also keep our users in the long run. This requires constans development, along popular trends and techniques.

# Bevezetés

Egyetemi tanulmányaim során kifejezetten tetszettek a webfejlesztéssel kapcsolatos oktatóanyagok és feladatok, valamint a munkaerőpiacon is elég keresettek a webes technológiákhoz értő szakemberek. Ezen két tény segített meghozni a döntés, hogy valamilyen webes témával és abból is a korszerű és gyakran alkalmazott technológiákkal szeretnék foglalkozni.

Egyetemi pályafutásom alatt, több különböző albérletben laktam. Mindegyiknek megvoltak a maga előnyei és hátrányai. Például egy a legfelső emeleten elhelyezkedő lakásban általában elég meleg van nyáron, cserébe a nincs felső szomszéd, aki lehetetlen időpontokban hangosan sétáljon felettünk. Egy másik érdekes igény lehet, ha például fontos, hogy legyen a lakásunkhoz közel zöld terület, iskola, óvoda vagy gyógyszertár, akkor egy egészen másfajta keresési módot kell alkalmaznunk, amit a legtöbb már létező ingatlan hirdető portál nem támogat. Itt fogant meg először az ötlet, hogy egy ilyen ingatlanhirdető portált én magam is megpróbáljak implementálni.

## Felhasznált technológiák jelentősége

A mai világban egyre nagyobb hangsúlyt kapnak a vékonykliens alkalmazások. Ez nem is csoda, hiszen mivel minden modern böngészőt futtató eszköz képes ezek futtatására. A legújabb *HTML (HyperText Markup Language)* és CSS (*Cascading Style Sheets*) szabványok támogatják a különböző kijelző méreteket és felbontásokat, így nem csak asztali számítógépen, hanem mobil és tablet eszközökön is kényelmes, intuitív a megjelenő felület. A *JavaScriptnek* köszönhetően a weboldalak innentől kezdve nem statikus dokumentumok, hanem felhasználó interakciókat kezelő dinamikus felületek. Elméletben ez remekül hangzik, azonban manapság ritkán fejlesztünk így. A gyakorlatban sokkal hatékonyabb a fejlesztés és karbantarthatóbb a kód, ha van egy keretrendszer, ami a fent említett három különböző elemet összefogja. Ilyen keretrendszerek például az Angular, Vue.js vagy amit jelen alkalmazáshoz használtam a React.js. Ez utóbbi a manapság legelterjedtebb ilyen keretrendszer. Az 1. ábrán betekintést nyerünk az elmúlt félévben, hogy alakult a legnépszerűbb keretrendszerek letöltéseinek száma.



. ábra: Legnépszerűbb *JavaScript* keretrendszerek letöltésének száma [1]

A szép felület azonban nem elég a hosszútávon fenntartható működéshez. A bővíthetőség, biztonság és a sebesség is meghatározó tényezők ma már. Ezen felül egy ilyen alkalmazásnál fel kell készülni a módosításokra is. A változó trendekhez, elvárásokhoz minél gyorsabban célszerű alkalmazkodni, illetve folyamatosan fenntartani a fejlesztést, a funkciók bővítését. A biztonságot elsősorban szerver oldali validációk sokaságával, a gyorsaságot pedig a backend teljesítménye mellett, megbízható adatbáziskezelővel lehet garantálni. Adatbázisok tekintetében még mindig a relációs adatbázisok a legelterjedtebbek, azonban számottevő a séma nélküli dokumentum alapú (másnéven *NoSQL*) adatbázisok. Fő előnyük a relációs adatbázisokkal szemben, hogy általában nincs meghatározott séma, ezáltal nincs szükség séma migrációra például egy új funkció hozzáadása esetén és jóval egyszerűbb a már élesben működő alkalmazás továbbfejlesztése. Ezen előnye miatt döntöttem a *NoSQL* vonal és azon belül is a *MongoDb* mellett.



. ábra: Relációs és dokumentum alapú adatbáziskezelők piaci részesedése (2019) [2]

A szerver oldali validációkat általában a backend üzleti logikája végzi, ezzel megőrzi az adatbázis konzisztenciáját, illetve meggátolja az illetéktelen hozzáféréseket. Még egy alapvető és fontos követelmény, hogy titkosított legyen a kapcsolat (*https*) a webszerver és a kliens között, ezáltal elkerülhető, hogy a felhasználók érzékeny adatait, például jelszavakat, e-mail címeket, bankkártya adatokat egy harmadik fél is megismerjen. Korábbi tapasztalataim alapján az *Asp.Net Core* backend mellett sok érv szólt. A legfőbb érv mellette mégis az volt, hogy könnyen felkonfigurálható az előző fejezetben említett MongoDb adatbáziskezelőre.

## Szakdolgozat tartalma

A dokumentum további fejezetiben a következő témák találhatóak:

* A 2. fejezetben a feladatspecifikációról fogok írni, melyben a feladat részletes leírásán túl a program funkcióit foglalom össze és Use Caseket szemléltetem diagramokon.
* A 3. fejezetben részletezem az irodalomkutatást, a megvalósításhoz felhasznált technológiákat, illetve más lehetséges alternatív megoldásokat is, valamint az indoklást egy adott technológia választása mellett.
* A 4. fejezetben betekintést adok a felsőszintű architektúrába, szemléltetem a rendszer felépítését, különböző komponenseit.
* Ezután az 5. fejezetben a részletes megvalósítás következik. Osztálydiagramon, szekvencia diagramokon és egyéb UML-es eszközökkel szemléltetem az alkalmazás működésének elvét.
* A 6. fejezetre maradnak a tesztelések, valamint a felhasználói leírás, képernyőképekkel.
* Végül a 7. fejezetben összefoglalom a munkám eredményét és felvetek néhány továbbfejlesztési lehetőséget.

# Feladatspecifikáció

## Feladat részletes leírása

A feladat egy ingatlan hirdető webalkalmazás architektúrájának kidolgozása, majd implementációja. A portálon legyen lehetőség hirdetések listázására, keresésésre, rendezésére többféle szempont alapján. Ezen szempontok lehetnek az ingatlan címe (település, kerület), az ingatlan ára minimum és maximum értékkel intervallum keresés, az ingatlan típusa (családi ház, lakás, ikerház, stb. ), valamint a hirdetés leírásában található kulcsszavak alapján. A felhasználók tudjanak feladni hirdetéseket, azokat szerkeszteni illetve törölni. A hirdetésekhez képek töltehetők fel, illetve virtuális kreditért cserébe kiemelhetőek, előresorolhatóak. Legyen lehetőség megadni a hirdetett ingatlanok pontos elhelyezkedését a térképen és lehessen ezen elhelyezkedés alapján is keresni ingatlanokat. Azon felhasználóknak amelyeknek van aktív hirdetésük, egyenként láthassák, hogy melyik hirdetést hányan tekintették meg. A felhsználók tudjanak üzenetet küldeni egymásnak, a könnyebb kapcsolatfelvétel érdekében.

A feladat részét képezi, bizonyos nem funckionális követelmények feltárása, illetve implementálása. A weboldalon nem lehetnek statikus újratöltések. Szükséges a különböző kijelzőméretek támogatása. A bejelentkezés és a regisztráció miatt elkerülhetetlen, hogy a backend biztonságos (*https*) kapcsolatot használjon. Az adatok validációját kliens és szerver oldalon is el kell végezni. Továbbá az authorizáció és autentikáció kimaradhatatlan lépés bizonyos funkciók esetében. Például a hirdetést csak az a felhasználó szerkeszthesse vagy törölhesse aki ténylegesen fel is adta korábban.

## Használati esetek

Ez az alfejezet a tervezés legelső fázisát tartalmazza. Felmérjük, hogy a felhasználó milyen interakciókat végezhet az alkalmazásunkkal, ezzel segítve a tervezés későbbi fázisait.

### Hirdetések keresése

Alapvető elvárás, hogy a felhasználók meg tudják tekinteni az aktív hirdetéseket és keresni is tudjanak közöttük, a korábban említett paraméterek alapján. Ezt a funkciót nem szükséges még bejelentkezéshez sem kötni. A rendezési funkció is megtalálható szinte már minden hirdető portálon. Általában az ingatlanok esetében három fő rendezési elvet szoktunk követni a feltöltés ideje, a vételár, vagy az alapterület mérete szerint. Első körben csak a „legújabb”, az „ár szerint növekvő” és „ár szerint csökkenő” rendezéseket terveztem bele, de a lehetőséget mindenképp fenntartom, további rendezési elvek bevezetésére.



3. ábra: Ingatlanok keresése oldal drótváza



. ábra: Ingatlan részletei oldal drótváza

### Hirdetés feladása, módosítása

Hogyha tovább folytatom, gondolatmenetemet hamar nyilvánvalóvá válik, fel is kell adni azokat a hirdetéseket a felhasználóknak. A hirdetés feladását célszerű mindenképp egy előzetes regisztrációhoz, valamint bejelentkezéshez kötni. Az elgépeléseken túl előfordul, hogy menet közben csökkenteni szeretné a felhasználó a meghirdetett irányárat, ha például nagyon csekély az érdeklődés és egyre sürgetőbb lenne az eladás, ezért erre is biztosítani kell a lehetőséget. A hirdetések szerkeszthetőségéhez szorosan kapcsolódik a követelményekben felvázolt képfeltöltés és hirdetés kiemelése funkció, tehát jó eséllyel ezen funkciók is módosítás felületről legyenek elérhetőek.



. ábra: Hirdetés feladása felület drótváz



. ábra: Hirdetés szerkesztés felület drótváza

### A hirdető adatlapja és értékelése

Megfogalmazódott bennem egy olyan ötlet, hogy ha már regisztráció szükséges a hirdetések feladásához, lehessen értékeléseket leadni a felhasználókra és ezek az értékelések nyilvánosan is megtekinthetőek legyenek. Ez nagyban segíti az új felhasználókat és esetleg felesleges köröket spórolhat meg egy megbízhatatlan eladó esetén. Három fajta értékelés adható le, pozitív, negatív és semleges, mindegyikhez fűzhető egy komment, amelyben a felhasználó kifejezheti elégedettségét, avagy részletezheti mivel volt probléma. Ezen értékelések megtekintését nem tartottam szükségesnek regisztrációhoz kötni, ellenben értékelést írni csak bejelentkezett felhasználó tud.

### Use Case diagram

A feladat felmérése után az alábbi (2. árbán) látható *use case* diagramot készítettem. Bár a funkcionalitást nem fedi le teljes egészeben, a lényeges felhasználói interakciókat tartalmazza. A struktúrát igyekeztem úgy felépíteni, hogy tükrözze az alkalmazás felületének struktúráját. Például, ha a hirdetés feladása felület csak a bejelentkezés után érhető el, a hirdetés módosítása (és annak további funkciói) pedig csak azután, ha már legalább egy hirdetést feladtunk.



. ábra: Use case diagram

# Felhasznált technológiák

Manapság egy webalkalmazás alatt nem a klasszikus értelemben vett weboldalt értjük. A modern weboldalak ma már alapvetően az *SPA (Single Page Application)* megközelítést alkalmazzák, ugyanis így natív érzetű gyors, reszponzív webes felületet kapnak a felhasználók. Ahhoz, hogy ez működőképes legyen mindenképp szükség van egy webszerverre, amely kiszolgálja a kliensek kéréseit, illetve gondoskodik a felhasználók hitelesítéséről és az adatok validációjáról. Miután nagy mennyiségű adatokkal dolgozunk, elengedhetetlen egy megbízható adatbáziskezelő rendszer.

## MongoDb

Mint korábban említtetettem a relációs (*SQL*) adatbáziskezelők mellett jelentős a dokumentum alapú (*NoSQL*) adatbáziskezelők térnyerése. Az *SQL* (*Structured Query Language*) [3] alapvetően a relációs adatbázisok standard lekérdező nyelvét jelenti, azonban az angol nyelvű irodalmakban gyakran relációs adatbázisokat *SQL* adatbázisokként, míg a dokumentum alapú adatbázisokat *NoSQL* adatbázisokként emlegetik, ezért itt is használom ezeket a kifejezéseket. Nem lehet egyértelműen kijelenti, hogy egyik jobb a másiknál ugyanis kettő teljesen eltérő megközelítést alkalmazunk. A relációs adatbázisok esetében általában a vertikális skálázhatóság és a magasfokú konzisztencia van előtérben. Ezen konzisztencia fenntartásáért az *SQL* adatbáziskezelő rendszernek nagy mennyiségű karbantartási feladatot kell végeznie. A *NoSQL* adatbázisokat úgy tervezték, hogy flexibilisek és minél gyorsabbak legyenek. Kevesebb a kényszer, mint a relációsok esetében, amely a konzisztencia többletköltségeit csökkenti. Ez a sebességnövekedés főleg akkor jelentős, ha kulcs-érték párokat tárolunk az adatbázisban. Ugyanakkor a *NoSQL* adatbáziskezelők esetében nem mindig támogatottak az *ACID (atomicity, consistency, isolation, durability)* [4] tranzakciók, amelynek adat inkonzisztencia lehet a következménye.



. ábra: SQL és NoSQL adatbáziskezelő teljesítményének összehasonlítása [5]

Egy ilyen népszerű dokumentum alapú adatbáskezelőt, a *MongoDb*-t választottam. Ez egy nyílt forráskódú, platformfüggetlen adatbázis szoftver, a dokumentumokat egy *BSON (Binary JSON)* formátumban tárolja, amely szintaktikájában nagyon hasonlít a *JSON (JavaScript Object Notation)* formátumhoz. [6] A *MongoDb* támogatja a keresést mező alapján, érték-tartomány alapján, vagy reguláris kifejezéssel, valamint lehetővé teszi, hogy a dokumentum bármelyik mezője alapján indexet készítsünk. A *MongoDb* fájlrendszerként is használható, az úgynevezett *GridFS* olyan problémákra ad megoldást, mint a terhelés elosztás és adat visszanyerő funkciók elosztott rendszerek esetén. [7] Amennyiben grafikus felületen keresztül szeretnénk használni az említett adatbáziskezelőt, akkor a *Robo3t* nevű program lehetőséget ad erre.

A *BSON* objektumoknak kötelező eleme egy „*\_id*” nevezetű mező, amely szerepét tekintve megegyezik a más adatbázisokban is használatos azonosító oszlophoz. Ez egy 24 karakter hosszú globálisan egyedi azonosító, *GUID (Globally Unique Identifier).* A további mezők számossága és értéke rekordonként eltérhet. Egyes mezők beágyazott dokumentumokat tartalmazhatnak. Az alábbi példában egy az alkalmazásunkban szereplő ingatlan látható az adatbázisban tárolt formátumban.



9. ábra: Egy ingatlan tárolása a BSON formátumban

## ASP .NET Core

*Az ASP.NET Core* egy nyílt forráskódú változata az *ASP.NET* keretrendszernek, amely *Windows*, *Linux*, *macOS* és *Docker* környezetekben is képes működni. Eredeti készítője a *Microsoft* és *C#* nyelven írták. Célja, hogy web alkalmazásokat készthessünk *.NET* platformon. Ez egy független újraírt keretrendszer, amely egyesíti a korábban külön működő *ASP.NET MVC*-t és az *ASP.NET Web API*-t egy programozási modellen belül. Jelenleg a 3.1-es a legújabb verziója, én is ezt használtam a megvalósítás során. Ezen verzió által támogatott a legújabb 2019-es *Visual Studio* is. A keretrendszer fejlesztésénél a fő fókusz a magas teljesítmény volt, amely meg is mutatkozik más népszerű keretrendszerek összehasonlításával. [8] [9]



. ábra: Keretrendszerek teljesítményének összehasonlítása, a kiszolgált kérések mennyisége alapján. [10]

### MongoDb .NET Driver

Az *ASP .NET Core* nagyfokú modularitást biztosít, az elérhető *NuGet* csomagok által. Ilyen *NuGet* csomagként érhető el például a *MongoDb Driver* könyvtár, amely segítségével aszinkron módon kommunikálhatunk a *MongoDb* adatbázis szerverrel, használhatók a jól megszokott *LINQ (Language Integrated Query)* kifejezések a lekérdezésekhez. A korábban említett *GridFS* elosztott fájlrendszer is támogatott.

### ASP .NET Core Identity

Az *Identity* egy kész megoldást kínál a felhasználókezelés megvalósítására webalkalmazásokhoz. Ez magában foglalja a felhasználók adatainak kezelését, jelszavak biztonságos tárolását, felhasználói jogokat, hozzáféréseket, *token*-eket, email cím megerősítését, és még sok egyéb mást. Lehetőséget kínál külső profillal való bejelentkezésre többek között *Facebook*, *Google*, *Microsoft Account* és *Twitter* fiókkal is regisztrálhatunk. Jelen alkalmazásunkban süti (*cookie)* alapú azonosítást használunk a bejelentkezést követően.

### ASP .NET Core Identity MongoDbCore

A címben említett *NuGet* csomag alapvetően a korábban említett *ASP .NET Core Identity*-hez készült adapter. Lehetővé teszi, hogy ne csak *Microsoft SQL Server* helyett *MongoDb* alapú adatbázisunkban tároljuk a felhasználók adatait.

## React

A *React* vagy más nevén *React.js* egy nyílt forráskódú, frontend, *JavaScript* könyvtár, amely segítségével a fejlesztők felhasználói felületeket (*User Interface*) vagy UI komponenseket készíthetnek. A *React.js* alapvetően webes felületekhez lett tervezve, de a *React Native* segítségével natív mobilalkalmazásokat is fejleszthetünk. Jelen alkalmazásunkban egy kliens oldali *SPA*-t fogunk megvalósítani melynek alapjául a *React* fog szolgálni. Jelenleg a legnépszerűbb *JavaScript* keretrendszer a piacon. Komponens alapú fejlesztést tesz lehetővé, amellyel csökkenthető a kód duplikáció és újra felhasználhatóak a komponensek. [11] Egy komponensen belül definiálhatunk bemeneti paramétereket (*props*), a komponenshez tartozó állapot változóját (*state*), illetve egy render metódust, amely visszaadja, hogy mit jelenítsen meg a komponens. Ez utóbbinál használható a *JSX (JavaScript XML)* szintakszis (ld.: 3.3.1. fejezet). Jelen alkalmazás frontend részének alapját képezi a *React*. Fontos megemlíteni, hogy bár a *React* alapvetően egy *JavaScript* könyvtár, létezik *Typescript React* változata is, amelyet *TypeScript* nyelven programozhatunk.

A *React* keretrendszer a versenytársainál gyorsabb működést ígér. Az alábbi ábrán láthatjuk, hogy az népszerű keretrendszerek esetében a böngészőnek mennyi időre volt szüksége összesen ugyanazon *JavaScript* kód futtatásához.



. ábra: Népszerű JavaScript keretrendszerek sebességének összehasonlítása [12]

### JSX

A *JSX* egy szintaxis kiterjesztés a *JavaScript* nyelvhez. A megjelenést és a logikát nem választja külön, hanem *HTML* elemeket helyezhetünk a *JavaScript* kódunkba. Miután ezek gyakran összefüggenek a kód átláthatóbb és karbantarthatóbb lesz. A *React* nem követeli meg a használatát, ellenben erősen ajánlott, hogy minél kényelmesebb legyen a fejlesztés. Rendkívül hasonlít a *HTML*-hez így nem sok idő elsajátítani a használatát. [13]

Például, az alábbi függvény paraméterül kap egy user változót, amennyiben a változó inicializálva van, visszaad egy címsort melynek a tartalma „Helló, <felhasználó neve>!”. Ha a változó nincs inicializálva, akkor a visszaadott címsor tartalma „Helló, idegen.” lesz.

function getGreeting(user) {

if (user) {

return <h1>Helló, {formatName(user)}!</h1>;

}

return <h1>Helló, idegen.</h1>;

}

### React Hooks

Mint már korábban említettem a *React* esetében komponensekből építhetjük fel webalkalmazásainkat. A *Hook*-ok (horgok) a komponenseink állapotának a kezelését könnyítik meg a programozó számára. Ezek használata szintén nem kötelező, viszont nagyban megkönnyíti a fejlesztést. Két fontosabb függvényt mutatok be, az első a *useState()*, melynek segítségével egy új állapot változót inicializálhatunk a komponensünkben. A másik a *useEffect()* mely paramétereként egy függvényt és állapot változók tömbjét várja. A logika egyszerű, a megadott függvény le fog futni amikor a komponens kirajzolódik, illetve, ha a megadott állapot változó értéke megváltozik.

Az alábbi példa az alkalmazásom kódjából származik. Bizonyos függvények törzsét eltávolítottam mert lényegtelen a példa szempontjából. Ez a komponens az üzenetküldést hivatott lebonyolítani. Két állapot változónk van, a „*users*” és az „*activeUser*”, előző azon felhasználók listája, akikkel az aktív felhasználó („*activeUser*”) már kapcsolatba lépett. Amennyiben ez az aktív felhasználó értéke megváltozik, akkor újra le kell kérni a kapcsolatba lépett felhasználók listáját az *API*-tól (értelemszerűen már megváltozott, új felhasználó értékével).

export default function Messages({ match }) {

//Állapotváltozók

const [activeUser, setActiveUser] = useState("");

const [users, setUsers] = useState([]);

const getUsers = async () => {…}

useEffect(() => {

//Felhasználók listájának lekérése az API-tól.

getUsers();

}, [activeUser])

return (

<div className="d-flex justify-content-center">…);

}

### React Router

Annak ellenére, hogy *SPA* megközelítést alkalmazunk, nem szeretnénk minden tartalmat kizárólag egy oldalon megjeleníteni, hanem navigációt szeretnénk oldalak között, linkeket szeretnénk küldeni másnak, amely nem a főoldalra mutat, hanem mondjuk jelen alkalmazás esetében például egy konkrét ingatlanra mutat. A *React Router* ezt a problémát hivatott megoldani. A natív érzetet növeli, hogy használhatjuk a böngésző navigációs funkcióit is.

<Router>

<Navbar loggedinState={isLoggedin} username={username} />

<Switch>

<Route exact path="/"><Home /></Route>

<Route path="/ingatlan/:id" exact component={Detail} />

<Route path="/ingatlan/:id/edit" component={EditIngatlan} />

<Route path="/messages" exact component={Messages} />

<Route path="/messages/:user" component={Messages} />

…

</Switch>

</Router>

A fenti példakódban az alkalmazás fő komponenséből (*App Component*) láthatunk egy részletet. A *Navbar* (navigációs sáv) komponensünk, a *Switch* (kapcsoló) komponensen kívül helyezkedik el, ugyanis ezt minden oldalon meg kívánjuk jeleníteni. A *Route* (út) komponensekkel megadható egy cím *path* (út) amely az alap *URL (Uniform Resource Locator)* után következik. Ennek az útnak megfelelően fogja kiválasztani a *Switch,* hogy mely komponens kerüljön megjelenítésre. A kettősponttal kezdődő útelemek értéke változó lehet. Például a „/ingatlan/:id” esetén az id az ingatlan azonosítója, amely alapján majd a kirajzolt komponens le tudja kérni az *API*-tól, az adott ingatlan adatait. Az *exact* kulcsszó esetén pontos egyezésre van szükség. Miután mintaillesztéssel halad sorban a komponenseken, így például, ha az első „/” útnál nem szerepelni az *exact* kulcsszó, akkor bármilyen *URL* esetén a *Home* komponens rajzolódna ki. Ez a probléma persze megoldható úgy is, ha a „/” út komponensét alulra helyezzük, azonban ez egy elegánsabb és átláthatóbb megoldás.

Navigáláshoz használhatjuk a *Link,* valamint a *Redirect* komponenseket. Mindössze a *„to”* paraméterben kell megadnunk a kívánt utat. *Link* esetén a navigáció a taralomra való kattintás után történik, a *Redirect* egyből a kirajzolás után navigál.

<Link to={`/messages/${user.userName}`}>

<span>Send a message!</span>

</Link>

## Bootstrap

A *Bootstrap* egy ingyenes, nyílt forráskódú *CSS* keretrendszer, melynek segítségével reszponzív weboldalak készíthetők. *CSS* és (opcionálisan) *JavaScript* alapú dizájn sablonokat tartalmaz, így többek között űrlapokat, gombokat, navigációt és egyéb felhasználói felület komponenseket. Jelenlegi legújabb stabil verziója a *Bootstrap 4* amely minden modern, népszerű támogat, jelen alkalmazáshoz is ezt használtam. [14]

### Reszponzív weboldalak

Az *RWD (Responsive web design)* (nem összekeverendő a reszponzivitással)egy olyan megközelítése a webalkalmazások tervezésének, ahol a weboldalak kirajzolása nagyban függ az azt futtató eszköz teljesítményétől, illetve a kijelző méretétől, felbontásától (pixelsűrűségétől). A tartalom a dizájn és a teljesítmény is eszközönként eltérő lehet, annak érdekében, hogy jobb legyen a felhasználói élmény. [15]

Tipikusan a legkritikusabb pont az ablak szélessége szokott lenni, ugyanis nagy könnyebbséget jelent a felhasználónak, ha nem kell oldalirányban is görgetni az oldalon, esetleg ránagyítani / kicsinyíteni adott részekre, mobilon vagy tableten való böngészés során. Ahogy csökken a szélesség az elemek általában egymás mellől egymás alá rendeződnek, eltűnnek a navigációs gombok és megjelenik az alábbi ábrán is látható úgynevezett „hamburger ikon”.



. ábra: Az alkalmazásban megjelenő "hamburger ikon" és menü

### Bootstrap Grid System

A *Bootstrap* egy *Flexbox* alapú „rács rendszert” biztosít nekünk a UI komponenseink megfelelő elhelyezéséhez és méretezéséhez. Ezt a *HTML* elemeinkre helyezett *CSS* osztályokkal érhetjük el. Alapértelmezetten mindig 12 db oszlop áll rendelkezésre a tartalmazó (*container*) osztály területén belül. A „col-\*” osztállyal megadhatjuk, hogy az adott elemünk hány oszlop széles legyen. Az alábbi példában két elemet helyezünk el egy sorba, az egyik 8 oszlop széles a másik 4.

<div class="row">

<div class="col-8">col-8</div>

<div class="col-4">col-4</div>

</div>



. ábra: Egy sorban elhelyezkedő 8 és 4 oszlop széles HTML komponens [16]

Természetesen a sokféle kijelző méretek változó komponens méreteket is igényelnek, ezért az előbb taglaltak megadhatók különböző szélességekre. Például, ha azt szeretnénk, hogy egy gomb mobilon nagyobb legyen, hogy könnyebb legyen megnyomni, azonban például PC-n nem kell annyi helyet elfoglaljon, akkor használjuk „col-lg-4” és a „col-md-6” osztályokat. Ezáltal, ha a böngésző szélessége 992 pixel alá esik, akkor 2 oszloppal szélesebb lesz a gombunk. Az alábbi ábrán láthatjuk, hogy mely kategóriák milyen szélesség tartományba esnek.



. ábra: Bootstrap kijelző méret kategóriák [16]

## Google térkép

A térképet igénylő megoldásokhoz a legnépszerűbb online térkép a Google térkép *API*-ját használtam. Ehhez szükség volt egy regisztrációra a *Google Cloud* platform-ra, ugyanis ahhoz, hogy integrálni tudjam a webalkalmazásomba a szolgáltatást elengedhetetlen egy *API* kulcs (*API Key*).

### Google Maps React

A google térkép integrációjához a „*google maps react*” nevezetű *NPM (Node package manager)* könyvtárat használtam. Ezen könyvtárral és egy egyszerű *boilerplate* kód segítségével könnyedén készíthetünk egy térkép komponenst. Lehetőségünk van beállítani a kezdeti pozíciót, a kezdeti közelítés (*zoom*) mértékét, illetve „gombostűket” (*Marker*) is elhelyezhetünk rajta. Az alkalmazásban ilyen *Marker* jelöli az ingatlan elhelyezkedését. Az alábbi kódrészlet egy egyszerű térkép komponenst hoz létre, fix középponttal. Ebből indultam ki és egészítettem ki kellő mennyiségű logikával. [17]

import React, { Component } from 'react';

import { Map, GoogleApiWrapper } from 'google-maps-react';

const mapStyles = {

width: '100%',

height: '100%'

};

export class MapContainer extends Component {

render() {

return (

<Map

google={this.props.google}

zoom={14}

style={mapStyles}

initialCenter={

{

lat: -1.2884,

lng: 36.8233

}

}

/>

);

}

}

export default GoogleApiWrapper({

apiKey: 'YOUR\_GOOGLE\_MAPS\_API\_KEY\_GOES\_HERE'

})(MapContainer);

A kezdeti pozíciót szélességi és hosszúsági koordináták megadásával határozhatjuk meg. Ezen koordinátákat akár a böngészőtől is elkérhetjük, feltéve, hogy a felhasználó engedélyezi a böngészőben az helymeghatározást.

    //Jelenlegi helyzet elkérése a böngészőtől

    const getCurrentLocation = async () => {

        if (navigator && navigator.geolocation) {

            navigator.geolocation.getCurrentPosition(pos => {

                const coords = pos.coords;

                setLongitude(coords.longitude);

                setLatitude(coords.latitude);

            });

        }

    }

# Felsőszintű architektúra

Az alkalmazás a klasszikus webesalkalmazásoknál használatos háromrétegű architektúra szerint épül fel. A három konkrét réget jelen esetben az adatbázis szerver, az alkalmazás szerver és a webszerver.

## Háromrétegű architektúra

Az alkalmazás felépítése a jól ismert háromrétegű architektúrán alapszik. Webalkalmazások körében széleskörben használt ez az architektúra. Általában egy *Frontend* szerverből, egy alkalmazásszerverből és egy adatbázisszerverből épül fel. Minden réteg csak a szomszédéival kommunikál. A háromrétegű architektúra egy szoftvertervezési minta is, a szoftverarchitektúrán felül. A modell legnagyobb előnye, hogy lehetővé teszi az egyes rétegek egymástól függetlenül történő fejlesztését, sőt, akár teljes cseréjét is. Ez biztonságosan megtehető, mert egy réteg módosítása nincs hatással a többi réteg működésére. Egymástól független modulokként tartalmazza a felhasználói felületet, az üzleti logikát és az adatbázist a szükséges hozzáférési műveletekkel. [18]

### Rétegek

Háromrétegű architektúra esetén az alkalmazást az alábbi 3 rétegre bontjuk:

* Megjelenítés: a társ rendszer felé nyújt interfészt (gyakran ez a felhasználói interfész), és az ehhez kapcsolódó eseményeket kezeli le. Ennek a leggyakoribb megvalósítása a weboldal és az előállító logika.
* Üzleti logika: ezen réteg feladata az üzleti folyamatok futtatása, hosszú életű tranzakciók kezelése. Itt kerül sor az adatok szélesebb hatókört, több adattípust átölelő szabályainak kikényszerítésére is.
* Perzisztencia: az adatok tartós tárolásával foglalkozik. Itt történik meg a szűkebb hatókörrel rendelkező adatszabályok kikényszerítése és gyakran az objektum és relációs adatmodellek közötti leképezés is. [18]



. ábra: A háromrétegű architektúra felépítése [19]

## Szerver oldali komponensek

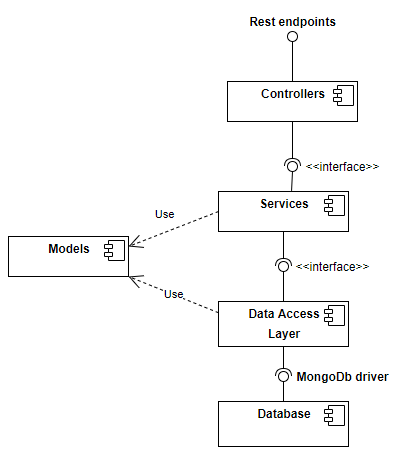
A legfelső réteg a kontrollereket tartalmazza. Egy *REST (Representational State Transfer)* végpont hívás közvetlenül ide fut be. Ebben a rétegben történik az azonosítás (*authentication*) és az engedélyezés (*authorization*). Amennyiben ez sikeres volt, vagy az adott végpont eléréséhez nem volt szükség rá, akkor továbbítja a kérést a megfelelő szolgáltatásnak (*Service*).

A szolgáltatás rétegben helyezkedik el általában az üzleti logika. Jelen esetben a szolgáltatások állítják össze a lekérdezéseket, módosítják az adatokat, adnak hozzá új hirdetést stb. Ezen szolgáltatások regisztrálódnak alkalmazás indulásakor és később más szolgáltatásokba (vagy kontrollerekbe) injektálhatók a függőség injektálás (*Dependency Injection*) segítségével.

A *Models* komponens tartalmazza az adatbázisban szereplő entitásokat, valamint az adatátviteli objektumokat (*DTO - Data Transfer Object*). Az utóbbiak a *Frontend* és a *Backend* közti kommunikációra szolgálnak, míg az előbbiek az adatbázisban *BSON* dokumentumokká képződnek le. Végül az üzleti logika által használt *Enum* típusok is itt találhatók. Ezek mind az adatbázisba, mind a megjelenési rétegbe egy egyszerű számként vannak reprezentálva.

Az adatelérési réteget (*DAL – Data Access Layer*) jelen alkalmazásban elfedi a *MongoDb Driver* könyvtár. A szolgáltatások egy (vagy akár több) *IMongoCollection* interfészt kérnek el az adatbázis entitástól (*IMongoDatabase*), amely segítségével lekérdezhetők, manipulálhatók az adatok. Ezen interfész is a *Models* komponens osztályait használja.

A legalsó réteg az adatbázis szerver, amellyel az alkalmazás a *MongoDb Driver* segítségével kommunikál. Jelen konfigurációban az adatbázis szerver és az alkalmazás szerver ugyanazon a számítógépen üzemelnek, de lehetséges felkonfigurálni úgy, hogy akár belső hálózaton külön számítógépről üzemeljenek.



. ábra: Backend komponens diagram

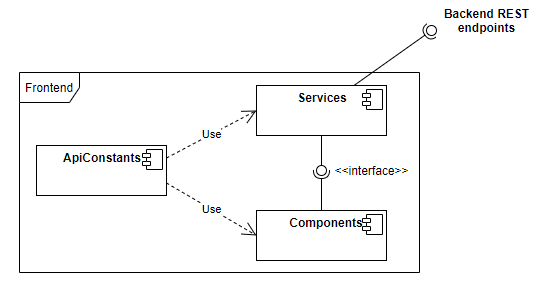
## Kliens oldali komponensek

Jelen esetben a Frontend réteg egy *SPA*, amely kliens oldali *render* megközelítést alkalmazza. Másszóval a weboldal megnyitásakor, letöltődik az egész frontend alkalmazás és ezek utána a szerverrel *REST* végpontok segítségével, csak a feltétlenül szükséges adatokat utaztatják a hálózaton (JSON formátumban). Ennek a módszernek nagy előnye, hogy nincsenek statikus oldal újra töltések, valamint reszponzív, natív élményt ad. Nem elhanyagolható ellenben az a hátrány, hogy jelentősen több erőforrást igényel a kliens eszközétől, mint egy szerver oldali *render* megközelítés, ahol a kész *HTML* fájlt kapja meg a kliens.

Az *ApiConstants* nevezetű komponens neve igen beszédes. Ez osztály egyetlen „index.js” fájlból áll, amely tartalmazza a backend által használt URL címeket, valamint az *enum* típusok JavaScript leképezéseit. Ezeket a többi komponens hivatkozza, így amennyiben módosulnak a backend akkor elegendő itt módosítani.

A *Services* komponensben találhatóak a végpont hívások. Fájl szinten kontrollerek szerinti csoportosításban, minden függvény külön behivatkozható és egy-egy különböző *API* végpontot hív. A végpont hívások aszinkron műveletek, így ezek a függvényeknek *Promise* a visszatérési értékük. Nem végzik el a válasz *JSON* formátumba alakítását, az egy alacsonyabb réteg feladata.

A *Components* tartalmazza azon *React* komponenseket, amelyek a megjelenítést, illetve a frontend logikát tartalmazzák. Ezen logika indítja a végpont hívásokat az előbb említett Services komponensen keresztül. Eleget téve a *JSX* és a *React* útmutatásainak a megjelenés (*HTML* kód) és a logika (*JavaScript* kód) egy fájlban helyezkednek el. Ezen komponenseket igyekeztem a funkcionalitásuk mentén felosztani, így minden újra felhasználható komponens külön fájlba került. Valamint a különböző útvonalon „route” elérhető komponensek kódja is mind külön fájlban kapott helyet.



. ábra: Frontend komponens diagram

# Megvalósítás részletes bemutatása

A megvalósítás részleteit külön a *frontend* és *backend* részekre lebontva fogom bemutatni. A fejezet tartalmazza még az alkalmazás tesztelésének menetét is.

## Szerveroldal

A szerver oldal bemutatását, a legalsó szintjével az adatbázis réteggel kezdem.

### Adatbázis

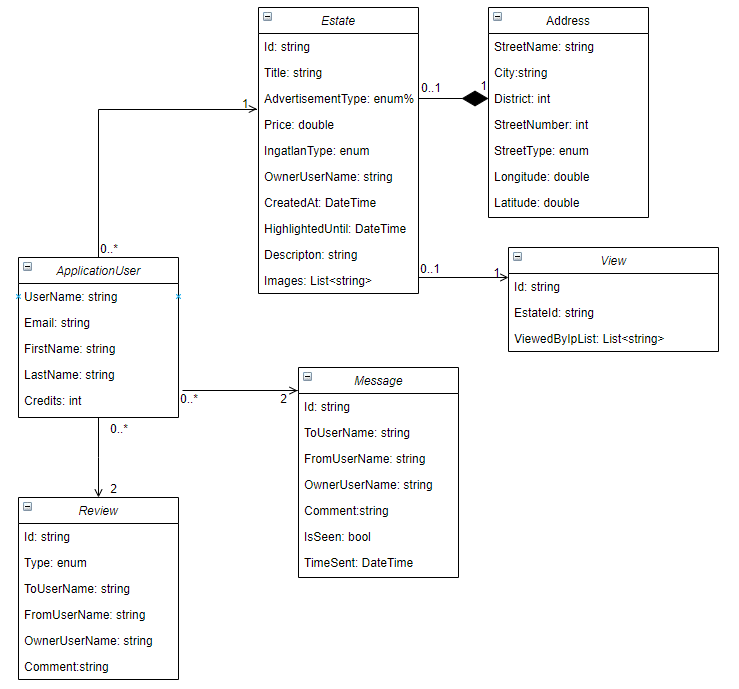
Az adatbázis létrehozásához, karbantartásához az egyik legnépszerűbb dokumentum alapú adatbázis kezelőt a *MongoDb*-t használtam. Ehhez a telepítés után elegendő kiadni az alábbi parancsot egy *PowerShell* ablakban és máris elérhető a szolgáltatás (a *PATH* helyére egy olyan könyvtár útvonalát adjuk meg, ahova dolgozhat az adatbáziskezelő).

mongod --dbpath PATH

### Adatelérés modell

A *MongoDb Driver* használatának segítségével az Adatbázis *BSON* dokumentumai *C#* osztályokká képződnek le és fordítva. Ezen osztályok képzik a modellt az adatelérési rétegben. Az osztályok egymás közötti viszonyát a 18. ábrán láthatjuk.

Egy felhasználóhoz (*ApplicationUser*) tartozhatnak ingatlanok, értékelések, üzenetek nullától véges sok intervallumon. Az ingatlan (*Estate*) az „*OwnerUserName”* tulajdonságán keresztül kapcsolható össze a tulajdonos felhasználóval. Az üzenet (*Message),* illetve az értékelés (*Review*)2db felhasználóhoz köthető, egy feladó (*FromUserName)* és fogadó (*OwnerUserName*). Egy ingatlannak ezen a szinten lehet nulla vagy egy darab címe (*Address*), de a gyakorlatban az üzleti logika kikényszeríti legalább a város megadását, szóval nem lesz cím nélküli ingatlan. Valamint az ingatlanokhoz tartozik egy lista, hogy mely felhasználók tekintették meg őket, feltéve, hogy valaki már legalább egyszer megnyitotta a részletek oldalon. A felhasználók IP címet tároljuk „*ViewedByIpList”* nevű listában, valamint az ingatlan azonosítóját az „*EstateId”* tulajdonságban. Az ingatlanhoz tartozó képeket adatbázis helyett külön mappában fájlként tárolom és csak a fájlok neveit tárolom az ingatlan „*Images*” nevű listájában.



. ábra: Modell osztályok osztálydiagramja

Adatbázis szinten nem valósulnak meg az úgynevezett külső kulcs kényszerek, tehát lehetséges létrehozni olyan üzenetet, amelynek a küldő vagy fogadó felhasználónév mezője üres (null), ellenben a backend üzleti logikája validációt végez és visszautasítja az ilyen jellegű kéréseket, normális körülmények között ilyen nem kerül az adatbázisba.

### Services

Ebben a névtérben található osztályok végzik az adatbázis műveleteket. Az előző fejezetben lévő objektumok szerint különülnek el, így például megtalálható „*EstateService*, „*MessageService*, *ReviewService*” stb. Ezen osztályok alapvetően úgynevezett *CRUD (Create, read, update and delete)* műveletek, vagy azok egy részét valósítják meg. [20]Például az ingatlanok esetében lehetőségünk van lekérdezni, létrehozni, módosítani, vagy törölni bizonyos példányokat. A hosszabb ideig tartó műveleteket (például ingatlanok lekérése) megvalósító függvényeket aszinkron módon valósítottam meg, valamint az „*Async and Await*” mintát követtem, az adatbázis műveletekhez LINQ kifejezéseket használtam. Példa egy ingatlan lekérésére azonosító (*id*) alapján:

public async Task<Estate> Get(string id) =>

await \_ingatlanok.Find<Estate>(house => house.Id == id).FirstOrDefaultAsync();

### Kontrollerek

Az ASP .Net Core webalkalmazásoknál a *Frontend* által küldött kérések a végpontnak megfelelő kontrollerbe futnak be. Paraméterben vesznek át Modell vagy DTO osztályokat, amelyek jöhetnek HTML kérés *Query* paraméterei közül, vagy a törzsből (*Body).* Ezt a „[FromQuery]” és a „[FromBody]” előtaggal van lehetőség jelezni a függvényparaméterek listájában. A kontroller metódusai fölött szereplő [Route] attribútum segítségével a végpont URL címét lehet megadni (amennyiben el kívánunk térni az alapbeállítástól). Jelen esetben több *HttpGet* metódusra volt szükségem, ellenben ezt nem akartam külön kontrollerben elhelyezni mert szorosan kapcsolódik az ingatlanokhoz (például összes város vagy összes kerület lekérdezése a keresés oldalhoz). Amely végpontok aszinkron műveletet hajtanak végre, itt is az „*Async and Await*” minta szerint lettek megvalósítva. Az alábbi kódrészletben az *EstateController* egyik metódusára látunk példát, ahol azon pozíció (földrajzi koordináták) alapján keresünk ingatlanokat. Azon példányokat kérjük el, amelyek egy koordinátától bizonyos távolságon belülre esnek.

[Route("/api/[controller]/bylocation")]

[HttpGet]

public async Task<List<Estate>> GetByLocation([FromQuery] double

longitude, [FromQuery] double latitude, [FromQuery]

double distance) {

return await \_ingatlanService.GetByLocation(longitude,

latitude, distance);

}

Az autentikáció és az authorizáció is ebben a rétegben történik. A kontrollerek függvény deklarációi előtt szereplő [Authorize] attribútummal beállítható, hogy csak bejelentkezett felhasználó érje el az adott végpontot. Jelen esetben azonban szükség van arra is, hogy például egy adott felhasználó csak az általa létrehozott hirdetéseket tudja módosítani vagy törölni. Ezt kézzel kódból ellenőrzöm és amennyiben nem a megfelelő felhasználótól érkezett a kérés 401 (*Unauthorized)* választ küldök vissza.

[HttpPut("{id:length(24)}")]

[Authorize]

public async Task<IActionResult> Update(string id, Estate

ingatlanIn) {

var ingatlan = await \_ingatlanService.Get(id);

if (id != ingatlanIn.Id)

return BadRequest();

if (ingatlan == null) {

return NotFound();

}

if (ingatlan.OwnerUsername != User.Identity.Name) {

return Unauthorized();

}

//…

\_ingatlanService.Update(id, ingatlanIn);

return NoContent();

}

### Autentikáció

Korábban említettem, hogy az autentikációhoz az *ASP .NET Core Identity* könyvtárat, és hozzá a *Mongo Db Core* adaptert használtam. Az adapter lehetővé teszi, hogy *MSSQL* Szerver, helyett. Az *Identity* könyvtár egy süti (*cookie*) alapú azonosítást tesz lehetővé. Ez a süti „*HttpOnly*” beállítással rendelkezik, melynek köszönhetően a böngésző nem engedi JavaScript kódnak a hozzáférését a sütihez. Egy külön *API* végpontot hoztam létre annak érdekében, hogy a *Frontend* értesülni tudjon a bejelentkezés meglétéről, illetve a bejelentkezett felhasználó nevéről. Ezen végpont megvalósítása, az alábbi kódrészletben található.

[Route("/[controller]/isloggedin")]

[HttpGet]

public ActionResult<LoggedStatusDTO> GetLoggedIn() {

LoggedStatusDTO status = new LoggedStatusDTO();

status.IsLoggedIn = User.Identity.IsAuthenticated;

if (status.IsLoggedIn)

status.UserName = User.Identity.Name;

return Ok(status);

}

Természetesen lett volna lehetőség a süti „*HttpOnly*” tulajdonságának kikapcsolására, ez azonban biztonsági kockázatok vetne fel, így jobbnak láttam a fent említett megoldásnál maradni, még ha ez több *API* hívással is jár.

## Kliensoldal

Ebben a fejezetben részletezem a *frontend* komponensek konkrét megvalósítását. A *Services* könyvtár az api hívásokat tartalmazza, ezzel fogok kezdeni.

### Services

Az szerver oldalon található kontrollerenként külön fájlokra bontottam a végpont hívások függvényeit. Ezen függvények átveszik a híváshoz szükséges adatokat (például keresési paraméterek, üzenet törzse stb.), majd visszaadják a hívás végeredményét (*JSON* konverzió nélkül). Egy ilyen függvény péládul az ingatlanok lekérése, amely átveszi a keresési paraméter objektumot (*queryobject*) meghívja vele az *API* végpontot, majd visszatér a hívás eredményével (a rétegen belüli többi függvény is nagyon hasonló, ezért itt csak ezt az egyet mutatom be).

export const getEstates = async (queryobject) => {

return await fetch(`${ApiCallItem}/?${serialize(queryobject)}`, {

method: 'GET',

credentials: 'include',

headers: {

'Content-Type': 'application/json'

},

});

}

Miután a http hívás (*fetch*) aszinkron hívás, ezért az „*await”* kulcsszó szerepel minden a komponensből hivatkozott függvény hívása esetén. A függvényt az alábbi sor segítségével hivatkozom be a különböző komponensekbe.

import {getEstates} from '../Services/EstateService'

### Components

Minden komponenst külön fájlban helyeztem el. A fájlok nevei megegyeznek a tartalmazott komponens nevével (a kiterjesztés minden esetben .js). A kliensoldali alkalmazás indulásakor az index.js fájl töltődik be először, amely importálja a szükséges CSS fájlokat, és kirendereli az App komponenst.

Az App komponensen belül található a Router (útválasztó) komponens. Az oldal tetején a navigációs sáv, mindig megtalálható, azonban az oldal törzsét a Router komponens rendereli az aktuális URL alapján. A következő fejezetben néhány érdekesebb komponenst fogok bemutatni.

#### Ingatlanok keresése

A *Search* komponens az ingatlanok keresése funkciót valósítja meg. Jelen esetben város (Budapest esetén kerület), minimum és maximum ár, hirdetés típus, ingatlan típus, valamint a leírásban található kulcsszavak alapján van lehetőséget szűrni. A városok és kerületek között csak azok választhatóak, amelyekhez tartozik legalább egy darab hirdetés. Ezen városokat, kerületeket az *API*-tól kérjük el külön végponton.

A fenti űrlapon nincs külön beadás gomb, a keresés egyből megtörténik, amit megváltoztatjuk valamelyik értéket. Ehhez a *React Hook*-ot használtam. Minden keresési értékhez egy külön állapotváltozót hoztam létre, amelyet a „useEffect” függvényhez függőségként hozzárendeltem.

const [city, setCity] = useState("");

const [district, setDistrict] = useState(0);

const [types, setTypes] = useState([]);

const [priceFrom, setPriceFrom] = useState();

const [priceTo, setPriceTo] = useState();

//…

useEffect(() => {

if (city || district || priceFrom || priceTo || adType) {

searchestates();

}

else {

getCitiesFromApi();

getDistrictsFromApi();

searchestates();

}

}, [city, district, priceFrom, priceTo, adType])

A városokat és kerületeket elég egyszer induláskor elkérni az *API*-tól.

#### Ingatlanok listázása

Ez a komponens az ingatlanok listázásáért és rendezéséért felelős. Nem önállóan jelenik meg a weblapon, hanem más komponensek használják fel, például a fönt említett *Search* komponens. Az *EstateList* egy keresési paramétereket tartalmazó objektumot vesz át és amikor ez az objektum megváltozik meghívja az *API*-t. Értelemszerűen ilyenkor megjelenített hirdetések listája is frissül. Az így kapott ingatlanokat szortírozza kiemelt és nem kiemelt listákba. Ezután a lapozási logika (*Pagination*) szerint kiválogatja a megjelenítendő ingatlanokat, illetve azt is beállítja hány lapozható oldalt kell megjeleníteni.

//…

const json = await response.json();

let highList = [];

let normalList = [];

json.forEach(estate => {

if (estate.isHighlighted) {

highList.push(estate);

}

else {

normalList.push(estate);

}

});

setHighlightlist(highList);

setList(normalList);

let pagelist = [];

pagelist.push(1);

let pagenumber = 1;

for (let i = 0; i < json.length; i++) {

if (i >= pagesize \* pagenumber) {

pagenumber++;

pagelist.push(pagenumber);

}

}

initListToDisplay(currentPage, json);

setPageNumbers(pagelist);

//…

//Lapozás

const initListToDisplay = (pagenumber, list) => {

setCurrentPage(pagenumber);

let newList = [];

list.map((estate) => {

if ((pagenumber - 1) \* pagesize < (list.indexOf(estate) + 1) &&

(list.indexOf(estate) + 1) <= (pagenumber \* pagesize))

newList.push(estate);

});

setListToDisplay(newList);

}

A rendezésnek három típusát implementáltam, ár szerint növekvő, ár szerint csökkenő, valamit időrendben csökkenő (legújabb elöl). A rendezés a kiemelt hirdetések listáját is rendezi.

#### C*hat, UserList és Messages*

A megvalósítás alapvetően hasonlít a *Search* és az *EstateList* kapcsolatára. A *Chat* komponens átveszi az ellenoldali felhasználó nevét és (minden alkalommal amikor megváltozik ez az érték) lekéri az általa, valamint a neki küldött üzeneteket, valamint kilistázza azokat. A *Message* komponens lekéri az szervertől azon felhasználók neveit, akikkel a bejelentkezett felhasználó váltott már üzenetet, majd átadja a *UserList* komponensnek. Ez a komponens csupán kirajzolja a felhasználók neveit egy-egy link formájában. Amikor rákattintunk valamelyik felhasználó nevére, akkor az URL megváltozik, „/messages/felhasználónév” lesz. A „refresh” függvényre azért volt szükség, hogy a *Messages* értesüljön az URL megváltozásáról, és a *Chat* újra rajzolódjon, a kiválasztott felhasználóval való üzenetváltásunkat mutassa.

#### Űrlapok

Az alkalmazásban három űrlap található, a bejelentkezés, a regisztráció, valamint az új hirdetés feladása. Ezen űrlapok mindegyike rendelkezik egy alapszintű kliensoldali validációs logikával. Például nem engedi beadni az űrlapot, ha valamely mező nem megfelelő (ellenben például a regisztrációnál a felhasználónév egyediségére csak *backend* oldalon történik validáció). Ezeket a visszajelzéseket részletesen a következő fejezetben mutatom be.

const handleSubmit = async (e) => {

e.preventDefault();

setLoading(true);

let user = {

email: email,

password: password,

remember: checkbox

}

const ok = await signIn(user);

if (ok) {

setFailed(false);

}

else

setFailed(true);

setLoading(false);

}

Az űrlapok beadására minden esetben külön függvényt hoztam létre, erre egy példa a fönti kódrészlet, ahol a bejelentkezés űrlap elküldését láthatjuk. Fontos az elején a *preventDefault* függvény hívása, ez ugyanis megakadályozza az alap működést, és nem tölti újra az oldalt. Ehelyett egy *API* végpontot hív és elküldi a felhasználó belépési adatait (email, jelszó, belépve maradok).

#### Google térkép

A Google térkép integrációjához a már korábban említett *Google Maps React* nevű könyvtárat használtam. Három különböző térkép komponens került az alkalmazásba, *SearchMap*, *DetailMap*, illetve az *EditMap.* A *SearchMap* és a *DetailMap* is csupán megjelenít ingatlanokat a térképen gombostű (marker) formájában. A különbség közöttük mindössze annyi, hogy a *SearchMap* kezdeti pozíciója a jelenlegi pozíciónk, amelyet a böngészőtől kér el. A *DetailMap* miután az ingatlan részletek oldalon látható, ezért a kérdéses ingatlan pozícióját helyezi a középpontba. Amennyiben ráklikkelünk egy gombostűre, akkor az alkalmazás átnavigál az adott ingatlanhoz tartozó részletek oldalra. A *DetailMap* kódja alább látható.

export function MapContainer({ estates, google, center }) {

let history = useHistory();

const onMarkerClick = (id) => {

history.push(`/estate/${id}`)

};

return (

<Map

google={google}

zoom={16}

style={mapStyles}

initialCenter={center}>

{estates.map((estate) => {

return (

<Marker

onClick={e => {onMarkerClick(estate.id)}}

key={estate.id + "marker"}

position={{

lat: estate.address.latitude,

lng: estate.address.longitude

}}

>

</Marker>

);

})}

</Map >

);

}

Az EditMap komponenst a hirdetés szerkesztése felületen használtam. Jellegzetessége, hogy a gombostű, amely az ingatlanunk pozícióját jelöli mozgatható. (Amennyiben nincs még megadva pozíció, akkor a böngészőtől elkért jelenlegi pozícióra helyezi a gombostűt). Amikor az befejezzük a mozgatást (felengedjük az egérgombot), akkor egy API hívás történik, amely felülírja az ingatlan koordinátáit, a gombostű koordinátáival. A megvalósításból néhány kódrészlet rögtön a fejezet után látható.

const onDragEnd = (coord) => {

const { latLng } = coord;

const lat = latLng.lat();

const lng = latLng.lng();

updateLocation(lng, lat);

}

return(

<Map

google={google}

zoom={14}

style={mapStyles}

initialCenter={center}>

<Marker

onClick={onMarkerClick}

name={"Your estate's location"}

draggable={true}

onDragend={(t, map, coord) => onDragEnd(coord)}

/>

<InfoWindow

marker={activeMarker}

visible={showingInfoWindow}

onClose={onClose}

>

<div>

<h4>Your estate's location</h4>

</div>

</InfoWindow>

</Map>

)

## Tesztelés

Az alkalmazás tesztelése manuális teszteléssel történt. A teszteseteket a használati esetetek alapján terveztem meg. A *backend* viszonylag egyszerű üzleti logikát valósít meg, valamint a manuális tesztelésnél végrehajtott tesztesetek is kellő kódlefedettséget tartalmaznak, így egység tesztek nem készültek.

### Tesztesetek

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | S. Martin, „The Top JavaScript Frameworks For Front-End Development in 2020,” [Online]. Available: https://www.freecodecamp.org/news/complete-guide-for-front-end-developers-javascript-frameworks-2019/. [Hozzáférés dátuma: 14 október 2020]. |
| [2] | ScaleGrid, „2019 Database Trends – SQL vs. NoSQL, Top Databases, Single vs. Multiple Database Use,” 4 március 2019. [Online]. Available: https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use/. [Hozzáférés dátuma: 15 október 2020]. |
| [3] | Wikipedia, „SQL,” [Online]. Available: https://hu.wikipedia.org/wiki/SQL. [Hozzáférés dátuma: 17 10 2020]. |
| [4] | Wikipedia, „ACID,” [Online]. Available: https://en.wikipedia.org/wiki/ACID. [Hozzáférés dátuma: 17 10 2020]. |
| [5] | Z. Y. Ruihan Wang, „SQL vs NoSQL: A Performance Comparison,” [Online]. Available: https://www.cs.rochester.edu/courses/261/fall2017/termpaper/submissions/06/Paper.pdf. [Hozzáférés dátuma: 17 10 2020]. |
| [6] | MongoDb, „JSON and BSON,” [Online]. Available: https://www.mongodb.com/json-and-bson. [Hozzáférés dátuma: 15 10 2020]. |
| [7] | Wikipédia, „MongoDB,” [Online]. Available: https://hu.wikipedia.org/wiki/MongoDB. [Hozzáférés dátuma: 15 10 2020]. |
| [8] | Microsoft, „What is ASP.NET Core?,” [Online]. Available: https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core. [Hozzáférés dátuma: 17 10 2020]. |
| [9] | Wikipedia, „ASP.NET Core,” [Online]. [Hozzáférés dátuma: 17 10 2020]. |
| [10] | M. WATSON, „Top 13 ASP.NET Core Features You Need to Know,” 31 October 2017. [Online]. Available: https://stackify.com/asp-net-core-features/. [Hozzáférés dátuma: 17 10 2020]. |
| [11] | Wikipedia, „React (web framework),” [Online]. Available: https://en.wikipedia.org/wiki/React\_(web\_framework). [Hozzáférés dátuma: 18 10 2020]. |
| [12] | S. Peyrott, „More Benchmarks: Virtual DOM vs Angular 1 & 2 vs Others,” 07 Január 2016. [Online]. Available: https://auth0.com/blog/more-benchmarks-virtual-dom-vs-angular-12-vs-mithril-js-vs-the-rest/#The-Results. [Hozzáférés dátuma: 17 október 2020]. |
| [13] | React, „JSX bemutatása,” [Online]. Available: https://hu.reactjs.org/docs/introducing-jsx.html. [Hozzáférés dátuma: 18 10 2020]. |
| [14] | Wikipedia, „Bootstrap (front-end framework),” [Online]. Available: https://en.wikipedia.org/wiki/Bootstrap\_(front-end\_framework)#Bootstrap\_4. [Hozzáférés dátuma: 20 10 2020]. |
| [15] | Wikipedia, „Responsive web design,” [Online]. Available: https://en.wikipedia.org/wiki/Responsive\_web\_design#Mobile\_first,\_unobtrusive\_JavaScript,\_and\_progressive\_enhancement. [Hozzáférés dátuma: 20 10 2020]. |
| [16] | Bootstrap, „Grid system,” [Online]. Available: https://getbootstrap.com/docs/4.0/layout/grid/. [Hozzáférés dátuma: 26 10 2020]. |
| [17] | R. Njeri, „How to Integrate the Google Maps API into React Applications,” 11 9 2020. [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-integrate-the-google-maps-api-into-react-applications. [Hozzáférés dátuma: 27 10 2020]. |
| [18] | Wikipedia, „Többrétegű architechtúra,” [Online]. Available: https://hu.wikipedia.org/wiki/T%C3%B6bbr%C3%A9teg%C5%B1\_architecht%C3%BAra. [Hozzáférés dátuma: 30 10 2020]. |
| [19] | F. Normén, „Using Web Services in a 3-tier architecture,” [Online]. Available: https://weblogs.asp.net/fredriknormen/using-web-services-in-a-3-tier-architecture. [Hozzáférés dátuma: 30 10 2020]. |
| [20] | Wikipedia, „Create, read, update and delete,” [Online]. Available: https://en.wikipedia.org/wiki/Create,\_read,\_update\_and\_delete. [Hozzáférés dátuma: 10 11 2020]. |

Függelék