



**Budapesti Műszaki és Gazdaságtudományi Egyetem**  
Villamosmérnöki és Informatikai Kar  
Automatizálási és Alkalmazott Informatikai Tanszék

Bui Quanganh Krisztián

# **INGATLAN HIRDETŐ PORTÁL MEGVALÓSÍTÁSA**

KONZULENS

**Benedek Zoltán**

BUDAPEST, 2023

# Tartalomjegyzék

<b>Összefoglaló .....</b>	<b>5</b>
<b>Abstract.....</b>	<b>6</b>
<b>1. Bevezetés .....</b>	<b>7</b>
1.1 Témaválasztás .....	7
1.2 Technológiai háttér .....	7
1.3 A szakdolgozat felépítése .....	8
<b>2. Felhasznált technológiák .....</b>	<b>9</b>
2.1 .NET.....	9
2.2 ASP.NET Core.....	10
2.3 ASP.NET Identity .....	10
2.4 Entity Framework Core .....	11
2.5 SignalR.....	11
2.6 Quartz.NET.....	13
2.7 TypeScript.....	13
2.8 React .....	14
2.9 Chakra UI.....	15
2.10 Tervezési minták.....	15
2.10.1 Domain Driven Design .....	15
2.10.2 Repository Pattern.....	16
<b>3. Funcionális követelmények .....</b>	<b>18</b>
3.1 Böngészés .....	18
3.2 Regisztrálás és bejelentkezés .....	19
3.3 Hirdetés létrehozása és kezelése .....	19
3.4 Előfizetés .....	19
3.5 Üzenetek küldése .....	20
3.6 További követelmények.....	20
<b>4. Architektúrák.....</b>	<b>21</b>
4.1 Rendszer architektúra.....	21
4.2 Szerveroldali architektúra .....	22
4.2.1 Domain réteg.....	22
4.2.2 Web API .....	23
4.2.3 Adatelérési réteg .....	24
4.3 Adatbázis felépítése .....	24

4.3.1 AspNetUsers .....	25
4.3.2 Ads .....	25
4.3.3 Messages .....	26
4.3.4 SubscriptionTiers .....	26
4.3.5 Subscriptions.....	26
4.4 Kliensalkalmazás .....	26
<b>5. Megvalósítás .....</b>	<b>28</b>
5.1 Hirdetések böngészése .....	28
5.2 Regisztrálás .....	35
5.3 Bejelentkezés .....	38
5.4 Hirdetések készítése és menedzselése .....	40
5.5 Előfizetések.....	43
5.6 Üzenetküldés.....	46
<b>6. Összefoglalás.....</b>	<b>51</b>
6.1 Továbbfejlesztési lehetőségek .....	51
<b>7. Irodalomjegyzék.....</b>	<b>53</b>

# HALLGATÓI NYILATKOZAT

Alulírott **Bui Quanganh Krisztián**, 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, 2023. 12. 05.

.....  
Bui Quanganh Krisztián

# Összefoglaló

Az ingatlanpiac folyamatosan változik és fejlődik, és egyre többen választják az online platformokat a lakások vásárlásához és eladásához. Ebben az új digitális korszakban az ingatlan hirdető portálok váltak az elsődleges eszközökké az ingatlanok értékesítéséhez és bérbeadásához. Ezen portálok nemcsak a lakáshirdetések gyűjtését teszik lehetővé, hanem számos kényelmi funkciót is nyújtanak az eladók és a vevők számára. Azonban az ilyen alkalmazások fejlesztése és működtetése számos kihívással jár, amelyeket meg kell oldani ahhoz, hogy hatékony és sikeres legyen a platform.

A szakdolgozatomban egy single-page webalkalmazást készítettem, amely lehetővé teszi azt, hogy a felhasználók meghirdethessék saját ingatlanukat, kereshessenek saját igényeiknek megfelelő lakásokat, és megkönnyíti a kommunikálást a hirdető és az érdeklődő között egy valós idejű üzenetváltás funkcióval. A felhasználóknak van lehetőségük előfizetésért fizetni is, amellyel hirdetésüket feltűnőbbé tehetik, így több ember látja az ő hirdetésüket. Az elkészítés során törekedtem egy felhasználóbarát megjelenést készíteni. A szakdolgozatban részletesen ismertetni fogom az alkalmazás tervezésének és fejlesztésének technikai részleteit is.

Az ingatlanhirdető portál megvalósításához egy többretegű webalkalmazást hoztam létre, amely egy React keretrendszerrel elkészített webkliensből és egy ASP.NET Core-t használó kiszolgáló szerverből áll, melyek REST API és SignalR könyvtár segítségével kommunikálnak egymással. Az alkalmazás implementálása közben több olyan könyvtárat és technológiát használtam, amelyek nem fordultak elő az egyetemi képzésem során, de népszerűek fejlesztők között, és szakdolgozatom célja volt ezeknek az eszközöknek a megismerése és elsajátítása.

# Abstract

The real estate market is constantly changing and developing, and more and more people are choosing online platforms to buy and sell homes. In this new digital era, real estate advertising portals have become the primary means of selling and renting real estate. These portals not only enable the collection of housing advertisements, but also provide many convenient functions for sellers and buyers. However, the development and operation of applications comes with many challenges that must be solved in order for the platform to be efficient and successful.

In my thesis, I created a single-page web application that allows users to advertise their own property, search for apartments that meet their needs, and facilitates communication between the advertiser and the interested party with a real-time message exchange function. Users also have the option to pay for a subscription to make their ad more prominent, so more people see their ad. During development, I tried to create a user-friendly interface. In the thesis, I will also describe in detail the technical details of the design and development of the application.

To implement the real estate advertising portal, I created a multi-layered web application, which consists of a web client made with the React framework and a server using ASP.NET Core, which communicate with each other using the REST API and the SignalR library. While implementing the application, I used several libraries and technologies that did not occur during my university education, but are popular among developers, and the aim of my thesis was to get to know and master these tools.

# 1. Bevezetés

Az ingatlanhirdetések módszerei drasztikusan változtak az elmúlt évtizedek során a technológiai fejlődések miatt. Régen a kiadó vagy eladó lakásokról ingatlanújságokból vagy hirdetőtáblákon keresztül tájékozódhattunk, illetve ingatlanközvetítőktől. Ezek a módszerek valamennyire korlátozottak voltak a potenciális bérlők és vásárlók elérésében, ezek a hirdetési módszerek általában a helyi közönséget érintették.

Manapság az internet, a számítógépek és a mobiltelefonok elterjedése jelentősen megváltoztatta az ingatlanhirdetéseket. Az online hirdetőoldalak segítségével bárhol és bárhol tudunk hirdetéseket böngészni elektronikus eszközeink segítségével, továbbá a kommunikálást is megkönnyítik.

## 1.1 Témaválasztás

A webalkalmazásfejlesztés egy izgalmas és dinamikus terület, amely folyamatosan változik és fejlődik. Az új technológiák és trendek folyamatosan megjelennek, és az alkalmazásoknak alkalmazkodniuk kell ezekhez a változásokhoz. A szakdolgozatom célja az volt, hogy mélyebben megértsem a webalkalmazásfejlesztés folyamatát, és hogy gyakorlati tapasztalatokat szerezzek ezen a területen.

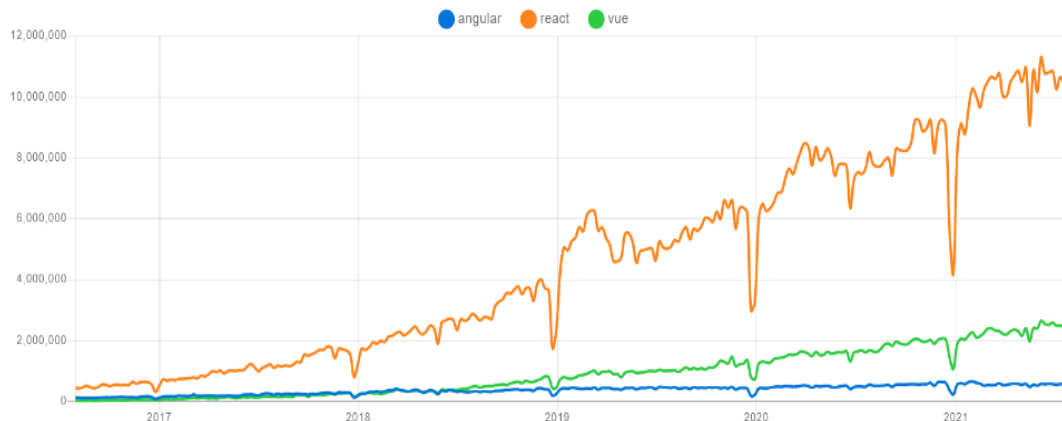
## 1.2 Technológiai háttér

Az utóbbi évtizedekben a webalkalmazások egyre nagyobb népszerűségnek örvendeznek. Már a legtöbb asztali alkalmazásnak megtalálhatjuk webalkalmazásbeli megfelelőjét, például a Microsoft termékei közül létezik a Word-nek, az Excel-nek és a többi Office 365 szolgáltatásnak böngészőből futtatható változata közel ugyanakkora (, ha nem ugyanakkora) funkcionalitással. Manapság az emberek rendszeresen használnak ilyen szolgáltatásokat, könnyedén el tudják érni őket telefonjaikon és asztali gépükön egyaránt.

Ezeknek az alkalmazások elkészítéséhez komplex UI keretrendszereket fejlesztettek ki a cégek, amelyeket nyilvánosan elérhetővé tettek. A legnépszerűbb a Meta (régebben Facebook) által fejlesztett React keretrendszer, és az Angular, amely a Google terméke. A React egy rugalmas és könnyen megtanulható könyvtár, de kevesebb alapkompont tartalmaz, míg az Angular több beépített funkcióval kínál, de szigorúbb

konvenciókkal rendelkezik. Mivel az Angular keretrendszer működését BSc tanulmányaim során már megismerhettem, így a React használata mellett döntöttem.

Downloads in past 5 Years ▾



Ábra 1: Frontend keretrendszerek népszerűsége letöltések szerint [1]

### 1.3 A szakdolgozat felépítése

A szakdolgozatom 2. fejezetében az alkalmazásom fejlesztése során felhasznált technológiákat és könyvtárakat fogom bemutatni.

A 3. fejezetben a webalkalmazáshoz kapcsolódó funkcionális és nem funkcionális követelményeket fogom részletezni.

A 4. fejezetben a portál architektúrális felépítését fogom ismertetni. Ki fogok térni a használt tervezési mintákra, és a rétegek felelősségeire.

Az 5. fejezetben az elkészült webalkalmazás funkcióit mutatom be, leírom a fejlesztés menetét, a kliens és a kiszolgáló szerver megvalósításának részleteit.

A 6. fejezetben összefoglalom a fejlesztés során szerzett tapasztalataimat, és leírom a továbbfejlesztési lehetőségeket.



## 2. Felhasznált technológiák

### 2.1 .NET

A .NET [2] (korábban ismert nevén .NET Core) egy ingyenes és nyíltforráskódú szoftverkeretrendszer, melyt 2016-ban mutatott be a Microsoft. A .NET Framework jelentősen átdolgozott cross-platform utódja, így futtatható Windows, Linus és MacOS rendszereken egyaránt.

A .NET alkalmazások magja a robosztus és felügyelt közös nyelvi futtatókörnyezet (Common Language Runtime - CLR) [3]. Többféle szolgáltatást nyújt, ilyen a memóriakezelés a szemétygyűjtő segítségével, a kivételkezelés, a típusellenőrzés és a szálkezelés. A CLR biztosítja továbbá a különböző programozási nyelvek együttműködését, ezért egy alkalmazás könnyedén felhasználhatja a különböző .NET által támogatott nyelveken írt könyvtárakat.

A .NET környezet nyelvfüggetlen, C#, Visual Basic, illetve az F# mellett számos objektumorientált programozási nyelvet támogat, illetve a különböző nyelveken készült osztályok egymásból örökölhetnek. Az alkalmazás fordításakor első sorban IL (Intermediate Language) kód készül, mely több architektúrára és operációs rendszerre fordítható. A platformtól függően natív kód áll elő, ami Just-in-time vagy Ahead-of-time módon fut le.



Ábra 2: Kód fordítása .NET környezetben. [4]

A keretrendszer moduláris felépítésű és csak alapvető osztályokat tartalmaz, így a környezet kis méretű marad, az alkalmazás pedig csak a használt könyvtárakat tartalmazza. További komponenseket a NuGet csomagkezelő rendszeren keresztül tudunk hozzáadni.

## 2.2 ASP.NET Core

Az ASP.NET Core [5] egy ingyenes, nyíltforráskódú keretrendszer, melyet modern web alapú alkalmazások készítésére fejlesztettek ki. A keretrendszer segítségével készíthetünk szerver oldalon renderelt multi-page alkalmazásokat (Razor pages), single-page alkalmazásokat (React, Angular) vagy REST (Representational State Transfer) API (Application Programming Interface) szolgáltatást, amelyet használhatunk tetszőleges kliensoldali technológia alkalmazásához.

Az ASP.NET támogatja a függőséginjektálás (Dependency injection) tervezési mintát. A minta lehetővé teszi azt, hogy egy osztály és azon függőségei között laza csatolás jöjjön létre azáltal, hogy az osztály nem függ az implementációtól. Az osztály a függőség szolgáltatásaihoz egy interfészen keresztül tud hozzájutni. Ezeket a Program.cs fájlban tudjuk rögzíteni, és a keretrendszer az osztály konstruktorán keresztül adja át a megfelelő függőségeket. Az injektált függőségeknek többféle életciklust [6] tudunk beállítani: Transient esetén minden használatkor új objektum jön létre, ami többszálú alkalmazásban előny lehet; a Scoped objektum egy kérésen belül ugyanaz marad; a Singleton objektum esetén a kérések ugyanazt az egy objektumot használják.

## 2.3 ASP.NET Identity

Az ASP.NET Identity [7] a Microsoft által biztosított felhasználókezelő rendszer, amely biztonságos hitelesítési és autorizációs funkciók kiépítésére szolgál az ASP.NET alkalmazásokban. Az ASP.NET Core keretrendszer része, amelyet a regisztráció, a bejelentkezés, a jelszókezelés és a szerepek kezelésére terveztek.

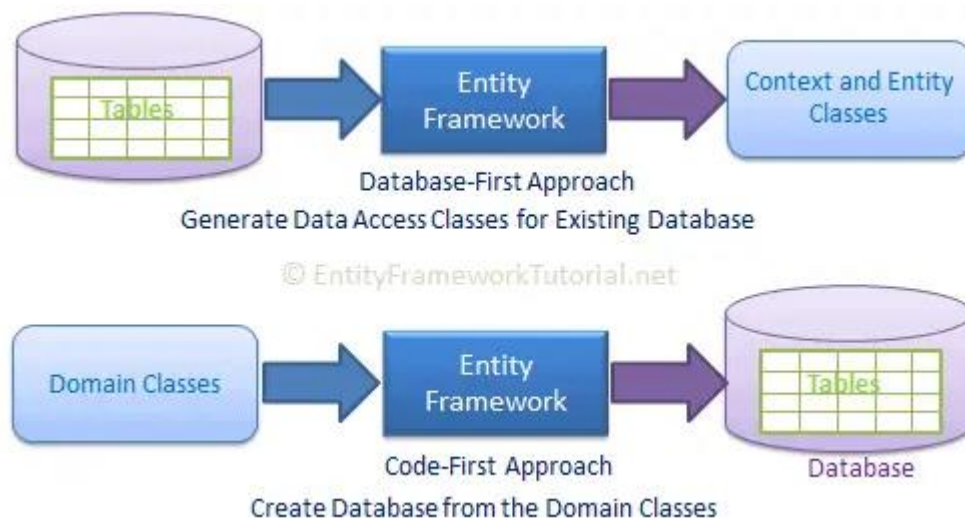
Az ASP.NET Identity több hitelesítési mechanizmust is támogat. Használhatunk külső bejelentkezés szolgáltatásokat, például Facebook, Google vagy Microsoft fiókokat, vagy eltárolhatjuk a bejelentkezési adatokat az Identity rendszer segítségével. Használhatunk cookie-alapú és token-alapú hitelesítést JSON Web Token (JWT) segítségével.

A hitelesítés és autorizáció egyik legfontosabb része a biztonság. Mivel én ebben a területben nincs sok tapasztalatom, a felhasználókezelő saját implementálása helyett egy előre megírt és jól dokumentált megoldás használata mellett döntöttem. Az ASP.NET Identity és az Entity Framework Core szorosan integrálódik egymással, ezért ennek a könyvtár használatát választottam.

## 2.4 Entity Framework Core

Az Entity Framework Core (EF Core) [8] egy Microsoft által létrehozott objektum-relációs leképező könyvtár. A szoftver platformfüggetlen, használhatjuk Windows, MacOS és Linux operációs rendszeren, továbbá kompatibilis a legtöbb adatbáziskezelő rendszerrel, például a Microsoft SQL Server-rel, a MySQL-lel és a PostgreSQL-lel. Ennek használatának segítségével függetleníteni tudjuk alkalmazásainkat az adatbázismotortól.

Az EF Core egyik fő funkciója a code-first leképezési módszer. A fejlesztőnek van lehetősége arra, hogy elsődlegesen az alkalmazás objektummodelljét definiálja majd a keretrendszer segítségével hozza létre a kapcsolódó adatbázis sémát. A könyvtár egy migrációs rendszert is tartalmaz, ami az adatsémában való változtatások lebonyolítását segíti az adatok elvesztése nélkül.



Ábra 3: A database-first és a code-first megközelítés közötti különbség [9]

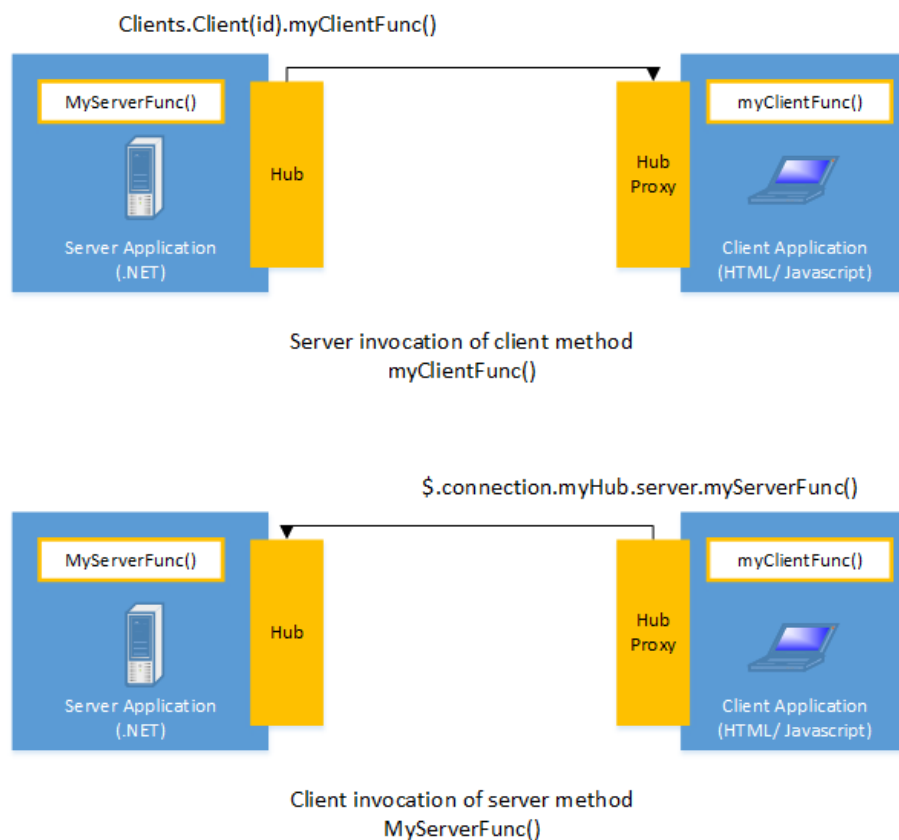
Az EF Core és a LINQ (Language Integrated Query) rendszer együttműködésével könnyen tudunk olvasható adatbázis lekérdezéseket írni C# szintaxis használatával. A LINQ típusbiztonságot is biztosít, és olyan objektumokon is használhatjuk, amik implementálják az IEnumerable és az IQueryable interfészeket.

## 2.5 SignalR

A SignalR [10] egy ingyenes és nyíltforráskódú könyvtár ASP.NET keretrendszerhez, melyet a Microsoft fejlesztett ki, és lehetővé teszi a valós idejű kommunikációt a szerver és a kliens között. Ez a technológia különösen hasznos azoknál

az alkalmazásoknál, amelyek olyan funkciókat igényelnek, mint az élő csevegés és a valós idejű értesítések. A könyvtár tartalmaz szerveroldali és kliensoldali JavaScript komponenseket egyaránt.

A könyvtár Hub-okat használ a kliens és a szerver közötti kommunikációhoz. A hub egy olyan interfészt nyújt a fejlesztőnek, ami lehetővé teszi azt, hogy a kliens a szerveroldali funkciókat tudjon meghívni paraméterekkel, és ugyanúgy fordítva, ezzel biztosítva a kétirányú kommunikációt. A SignalR eseménykezelést (pl. csatlakozás és lecsatlakozás), hibakezelést, felhasználó autentikációt és autorizációt is nyújt.



**Ábra 4: A kliens és szerver közötti függvényhívás. [11]**

A SignalR többféle módon tudja kezelni a valós idejű kommunikációt: WebSockets, szerver által küldött események (Server-Sent Events), vagy Long Polling segítségével. A rendszer automatikusan kiválasztja a körülményeknek legjobban megfelelő kommunikációs módszert a kliens képességei és a szerver konfigurációját figyelembe véve.

## 2.6 Quartz.NET

A Quartz.NET [12] egy nyíltforráskódú feladat ütemező keretrendszer, melyet a .NET keretrendszerre fejlesztettek ki. A könyvtár segítségével könnyedén lehet olyan feladatokat definiálni, amelyeket bizonyos időközönként vagy eseményekkor kell elvégezni. A keretrendszer erősen konfigurálható.

Az egyik funkciója könyvtárnak a Clustering, a rendszer több szerveren is képes futtatni a feladatütemezőt, amely nagy rendelkezésre állást biztosít. A Clustering figyel arra, hogy egy feladatot csak egy szerveren futtasson, ezzel megakadályozva azok ismétlődését. A keretrendszer támogatja a bővítmények létrehozását is, lehetővé teszi különböző eseményekhez hallgatók hozzáadását, amelyek lehetővé teszik testre szabott műveletek vagy monitorozás hozzáadását.

## 2.7 TypeScript

A TypeScript [13] egy magasszintű, objektum-orientált, böngészőkben használt, nyíltforráskódú programozási nyelv, ami a JavaScript kiterjesztése. A TypeScript fordító JavaScript kódot hoz létre, ezért minden JavaScript-et támogató környezetben futtatható, és minden JavaScript kód szintaktikailag érvényes TypeScript kód is. A nyelv funkciói jelentősen javítja a kód karbantarthatóságát és olvashatóságát.



Ábra 5: A fordító TypeScript kódból JavaScript kódot készít. [14]

A TypeScript statikus típusosságot biztosít, így lehetővé teszi a típusellenőrzést fordítási időben. Ez jelentősen megkönnyíti a fejlesztést, mivel a legtöbb futási idejű hiba kiderül fordításkor, továbbá valódi IntelliSense támogatást is nyújt. A nyelv segítségével saját osztályokat és interfészeket is létrehozhatunk, melyeknek adattagokat és függvényeket adhatunk. Strukturális típusosság jellemzi, vagyis egy objektum

strukturálisan kompatibilis egy másik objektummal, ha tartalmazza a publikusan elérhető tagváltozóit és függvényeit.

Alapértelmezetten a változók felvehetik a null és undefined értékeket, melyek további futási idejű hibához vezethetnek. A fordító `--strictNullChecks` flag beállítása esetén a fordító típushibát dob, ha egy típusos változónak az előbb említett értékek egyikét adjuk meg.

A TypeScript modulok használatát is biztosítja, ezek segítségével összefüggő osztályokat, függvényeket, változókat tudunk összefűzni egy logikai fájlba. Alapesetben egy modul tartalma csak exportálás után láthatóak kívülről, és minden használni kívánt elem importálás után használhatóak.

## 2.8 React

A React [15] egy ingyenes, nyíltforráskódú, JavaScript keretrendszer, melyet a Meta (korábban Facebook) fejlesztett ki. A könyvtár segítségével felhasználói felületet tudunk létrehozni. A felület leírását deklaratív módon tudjuk leírni, a keretrendszer automatikusan frissíti és újrarendereli a komponenseket a fájlok változtatásakor. Ez jelentősen felgyorsíthatja a fejlesztést és megkönnyíti a hibakeresést. A keretrendszerhez mellékeltek típus fájlokat is, ezért TypeScript projektben is könnyedén használható.

A React segítségével komponens alapú felhasználói felületet tudunk létrehozni. A komponenseket kétféleképpen tudjuk leírni: osztálykomponensként vagy függvénykomponensként. Az osztálykomponensek ES6 osztályokként jelennek meg, amelyek öröklök a `React.Component` össztályt és életciklus függvényeket alkalmaznak. Állapotok eltárolásához a 'state' objektumot használják, a külső paramétereket pedig a 'props' objektumon keresztül érjük el. A függvénykomponensek egyszerű függvényekként jelennek meg, a külső paramétereket a függvény paramétereiként kell átadni. Eleinte a függvénykomponenseknek nem volt állapotkezelő rendszere és életciklus függvényeik, de a könyvtár 16.8-as verziójában implementálták a 'hook' függvényeket, melyek segítségével már lehetséges volt. A függvénykomponensek később népszerűbbek lettek az osztálykomponensekhez képest a hook-ok bevezetése óta az egyszerűségük miatt.

A komponensek leírásához a React új szintaxisokat vezetett be, amelyet JSX-nek (JavaScript HTML) neveztek el. A JSX a JavaScript bővítménye, amely segítségével

HTML-szerű jelöléseket tudunk használni a React komponensek leírásához. Az alkalmazás fordításakor a HTML leírás függvény hívásokká alakulnak át, amelyek HTML elemeket hoznak létre, erre egy példa lent látható:

```
const hello1 = <div>Hello world!</div>;  
const hello2 = React.createElement('div', null, 'Hello world!');
```

A React mellé célszerű további könyvtárakat is használni, amelyek kibővítik a keretrendszer funkcionalitását, ilyen például a React Router, a React Hook Forms vagy a React Query.

## 2.9 Chakra UI

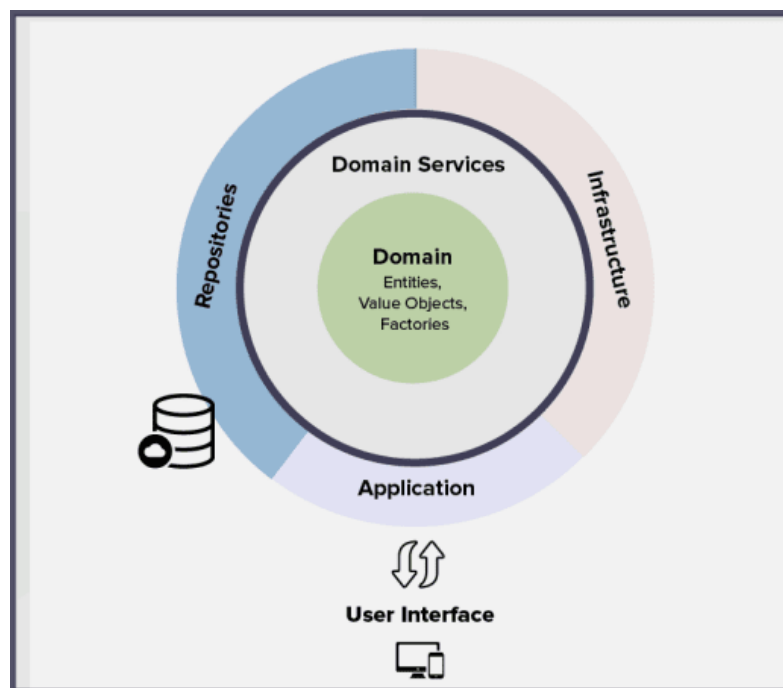
A Chakra UI egy keretrendszer, amely egy rendszert biztosít saját stílusok definiálásához és komponensek testreszabásához. A könyvtár tartalmaz rengeteg előre definiált React komponenseket, amelyeket felhasználhatunk a felhasználói felület leírásához. A Chakra UI reszponzív tervezési funkciókat is nyújt, így könnyen tudunk a képernyőméretekhez alkalmazkodó felületeket készíteni.

A könyvtár komponenseinek stílus prop-okat tudunk megadni, amelyek megfeleltethetők inline CSS szabályoknak. Rövidítéseket is elfogadnak és ezzel gyorsabban meg tudtam tervezni a felületek kinézetét, de a projektben a komponens véglegesítése után a szabályokat áthelyeztem egy külön fájlba könnyebb átláthatóság és újrafelhasználhatóság miatt.

## 2.10 Tervezési minták

### 2.10.1 Domain Driven Design

A Domain Driven Design [16] (DDD, magyarul tartományalapú tervezés) egy szoftverfejlesztési területen használt tervezési irányelv. Az irányelv alapja az, hogy az alkalmazás középpontjában a domain (terület) szerepel, amely a probléma modelljei és szolgáltatásaiból áll, és a körülötte lévő struktúra vagy eszközök alkalmazkodnak hozzá.



Ábra 6: A Domain Driven Design alapján elkészített alkalmazás felépítése [17]

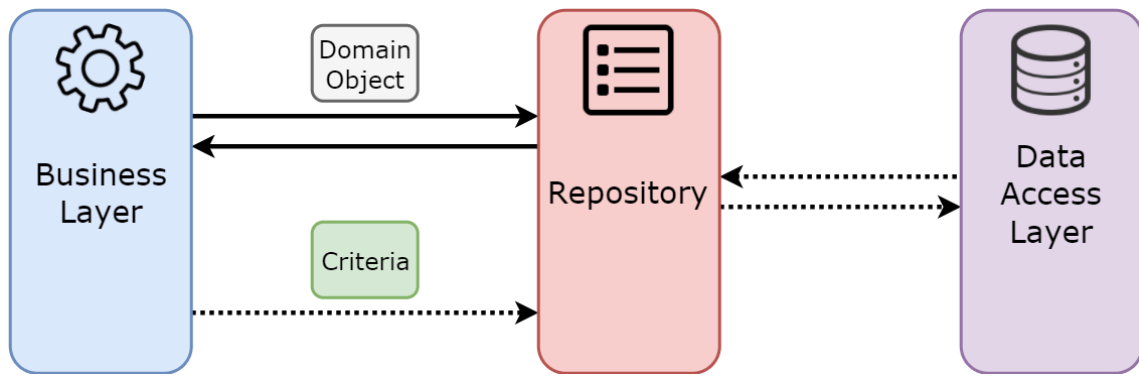
A Domain Driven Design egyik fő karakterisztikája a Bounded Context, azaz az üzleti logika felbontása olyan aldomaineekre, amelyek adott problémákra fókuszálnak. Ezeknek az egységeknek általában saját modelljeik, funkcióik és szolgáltatásaik vannak, és más egységtől függetlenek. A minta használata csökkenti az elemek közötti csatolást és megkönnyíti a párhuzamos fejlesztést, különböző csapatok tudnak a saját elemein dolgozni anélkül, hogy konfliktusokba kerülnének egymással.

Az alkalmazások fejlesztésekor kulcsfontosságú a probléma többretegű megközelítése. A belső rétegek erős összefüggése és a különböző rétegek közötti laza kapcsolatok segítik az alkalmazás karbantarthatóságát és bővíthetőségét. A DDD hasonlít a hagyományos három rétegű architektúrára, azonban az üzleti logikai réteg és az adatelérési réteg függősége megfordul, az üzleti logikai réteg függ az adatelérési rétegtől. Ennek a függőség irányának megfordításához a Repository tervezési mintát használtam.

### 2.10.2 Repository Pattern

A Repository tervezési mintát gyakran használják a szoftverfejlesztésben, az alkalmazás adatelérési rétegében. A minta egy absztrakciós réteget definiál az alkalmazás üzleti logikája és az adatforrás között, amely segítségével laza csatolást érhetünk el a kettő réteg között, az üzleti logikának nem kell ismernie pontosan az adatelérés implementációját, hogy elérje az adatokat.





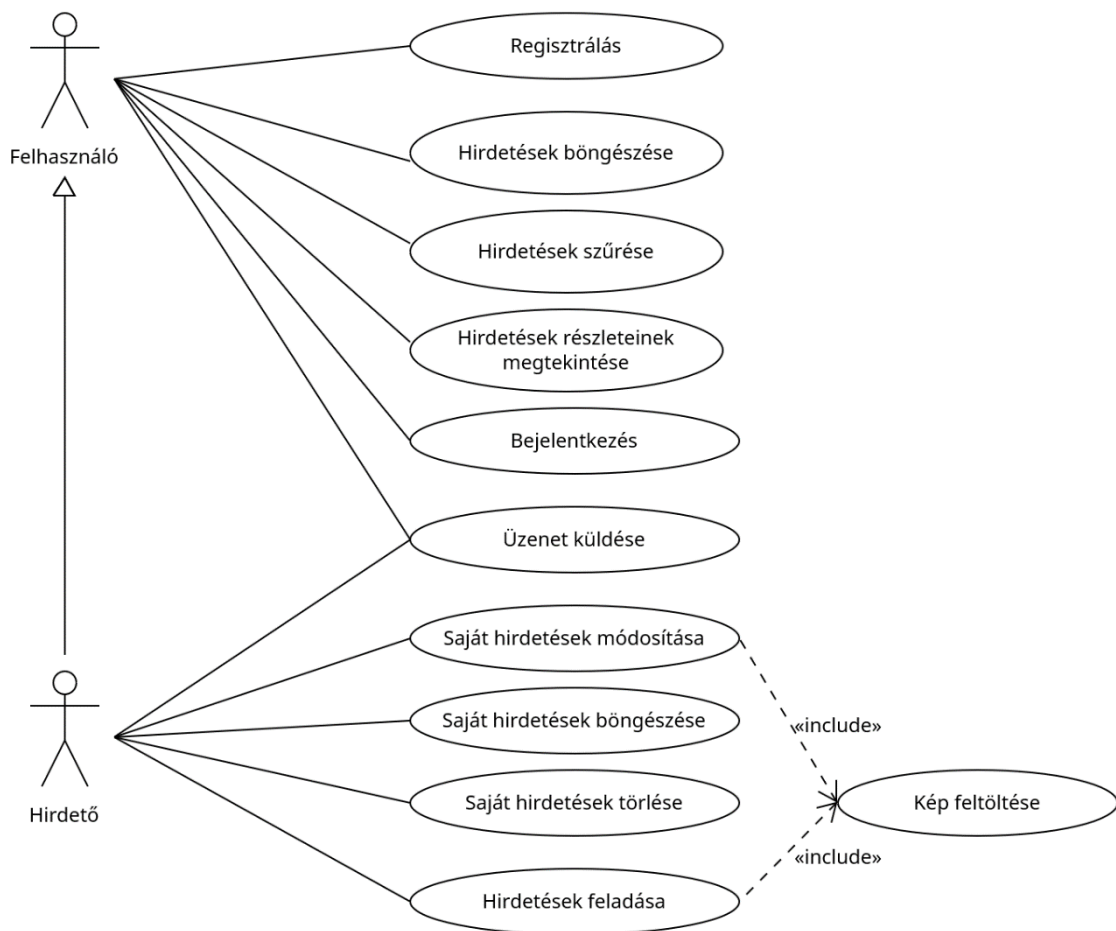
Ábra 7: A Repository tervezési minta általános sémája [18]

A minta egységes interfészeket nyújt, amelyen keresztül az üzleti réteg manipulálni tudja az adatokat. Ezeket az interfészeket a különböző típusú adatforrásoknak implementálniuk kell. Az adatforrások lehetnek különböző gyártók adatbázismotorjait használó relációs adatbázisok, NoSQL adatbázis, fájlrendszer, vagy külső szolgáltatás is, és az egységes interfész miatt könnyedén tudunk váltogatni a megoldások között anélkül, hogy megváltoztatnánk az üzleti logikát.

Az interfészek használata megkönnyíti az alkalmazás részeinek tesztelését is. Mock repository osztályokat használhatunk tényleges adathozzáférés nélkül, amely segítségével függetleníteni tudjuk a tesztelendő részeket, és lehetővé teszik a tesztelést adatbázis nélkül is.

### 3. Funcionális követelmények

Ebben a fejezetben a funkcionális követelményeket fogom felsorolni, amelyek a megvalósítandó alkalmazáshoz nélkülözhetetlenek. A szerepkörök és az elvégezhető műveletekről készült ábra lentebb látható.



Ábra 8: A végrehajtható funkciók felhasználók szempontjából

#### 3.1 Böngészés

Az egyik nélkülözhetetlen funkció az alkalmazásban a létrehozott hirdetések böngészése. A felhasználó bejelentkezés nélkül csak ezt a funkciót éri el. A hirdetések egy listán keresztül tekinthetők meg, melynek szűrési lehetőségeket kell biztosítani. Szűrési feltételnek megadható az ingatlan címe, ára, mérete és a szobák száma. A listanézetben a hirdetések száma korlátozott a könnyebb átláthatóság és navigálás érdekében, lapozással lehet a többi hirdetést megjeleníteni.

Ha egy hirdetés felkeltette a felhasználó érdeklődését, a listában lévő hirdetésre kattintva meg kell jelennie az ingatlan részletes leírásának. Ennek a nézetnek egy külön oldalon kell megjelennie, a részletek között meg kell jelennie az ingatlan adatainak, egy róla készült fotónak és a hirdető elérhetőségeinek. A bejelentkezett felhasználónak van lehetősége felvenni a kapcsolatot a hirdetővel az alkalmazáson keresztül is.

### **3.2 Regisztrálás és bejelentkezés**

Ahhoz, hogy elérje az alkalmazás további funkcióit, a felhasználónak rendelkeznie kell egy saját fiókkal, amit a regisztrációs ablakon tud létrehozni. Regisztráció esetén meg kell adni egy érvényes e-mail címet, felhasználónevet, megfelelő erősségű jelszót és opcionálisan egy telefonszámot. A regisztráció és bejelentkezés után elérhetővé válik a hirdetés készítése és az üzenetek küldése.

### **3.3 Hirdetés létrehozása és kezelése**

A másik fontos funkció az alkalmazásban az ingatlan hirdetések létrehozása. A hirdetés létrehozásakor meg kell adni a hirdetés címét, a lakás címét, méretét, szobák számát, egy róla elkészült képet és az árát. A hirdetőnek van lehetősége kiemelt hirdetést is létrehoznia, ebben az esetben a hirdetés a böngészőlista elején jelenik meg a létrehozási dátumtól függetlenül. A kiemelt hirdetés létrehozásához előfizetés szükséges. A kliensnek és a szervernek validációt kell elvégeznie a kérés feldolgozása előtt. A képek eltárolása az adatbázisban történik, amit Base64 formájú szöveggént tárolunk el. A hirdetőnek van lehetősége a saját hirdetéseit kilistázni egy külön nézetben. Itt eltávolíthatja a már nem aktuális hirdetést vagy módosíthatja a már elavult információkat, a hirdetés kiemeltségét.

### **3.4 Előfizetés**

A felhasználónak van lehetősége előfizetnie különböző csomagokra az alkalmazásban. Az előfizetés előnye az, hogy az előfizető kiemelt hirdetéseket hozhat létre, amelyek a böngészési lista elején fognak megjelenni a létrehozási dátumtól függetlenül. Az előfizetés menete több lépésből áll. Az első lépésben a felhasználónak előfizetéskor ki kell választania a csomagot, amire elő szeretne fizetni. A csomagok a maximum kiemelt hirdetések számában és az előfizetés árában különböznek. A következő lépésben meg kell adni a használandó bankkártyaszámot, biztonsági kódot és

a bankkártya lejáratát. Az alkalmazás nem használ tényleges fizetési rendszert. A harmadik lépésben lehet átnézni a megadott információkat, és a „Subscribe” (Előfizetés) gombra kattintva tudja véglegesíteni a folyamatot. Ha a felhasználónak van előfizetése, akkor meg tudja tekinteni annak adatait.

### **3.5 Üzenetek küldése**

Az üzenetváltás elkezdését az érdeklődőnek kell kezdeményeznie egy ingatlan hirdetés részletes nézetén keresztül. Az üzenet elküldése után egy másik nézet fog elénk tárulni, amin az összes üzenetet el tudjuk olvasni és további üzeneteket is tudunk küldeni. Ha egy felhasználónak vannak olvasatlan üzenetei, akkor a kliens menübarban kijelzi egy számláló segítségével. Ha az üzenetekhez navigál a felhasználó, akkor az összes vele kapcsolatos párbeszédet látni fogja. A párbeszédre kattintva frissülnie kell a menübarban lévő számlálónak, meg kell jelennie az egymásnak elküldött üzeneteknek és azoknak az elküldési idejüknek.

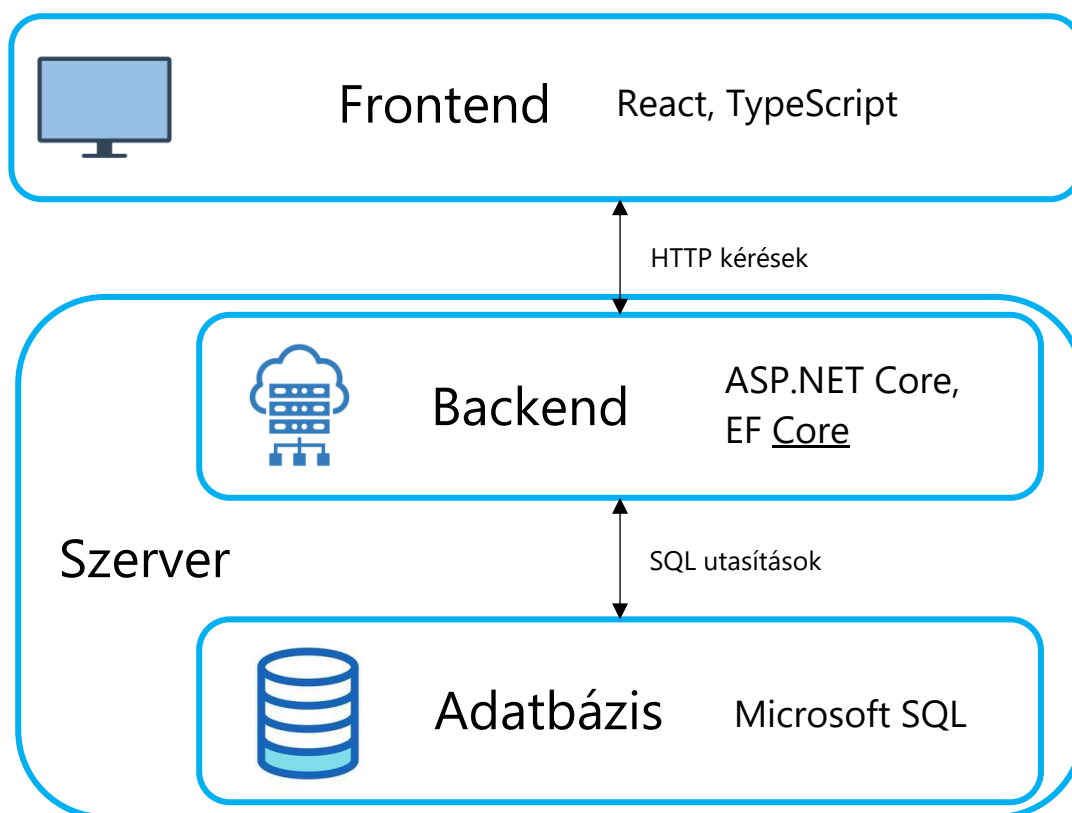
### **3.6 További követelmények**

A felhasználó felületnek letisztultnak, reszponzívnak és intuitívnak kell lennie. A kliensnek a felhasználó legtöbb műveletére reagálnia kell, a hosszabb műveletek esetén aszinkron működést kell biztosítani, hogy ne fagyjon le a felhasználói felület. Az esetleges hibákat a kliensnek jeleznie kell.

## 4. Architektúrák

### 4.1 Rendszer architektúra

Az ingatlan hirdető alkalmazás három fő komponensből áll: az alkalmazást megjelenítő kliensből, amin keresztül a felhasználó tudja használni a szolgáltatást, a kiszolgáló szerverből, amely az alkalmazás logikáját valósítja meg, és az adatbázisból, ahol az adatokat tároljuk el. A felhasználó egy React alkalmazást futtató böngészőn keresztül tartja a kapcsolatot a kiszolgáló szerverrel HTTP kérések segítségével. A szerver egy REST interfészen keresztül szolgálja ki a klienseket, amit az ASP.NET Core technológiával valósít meg, és az Entity Framework Core könyvtár segítségével kommunikál a Microsoft SQL adatbázissal. A valós idejű funkciók megvalósításához SignalR Hub-okat használtam.



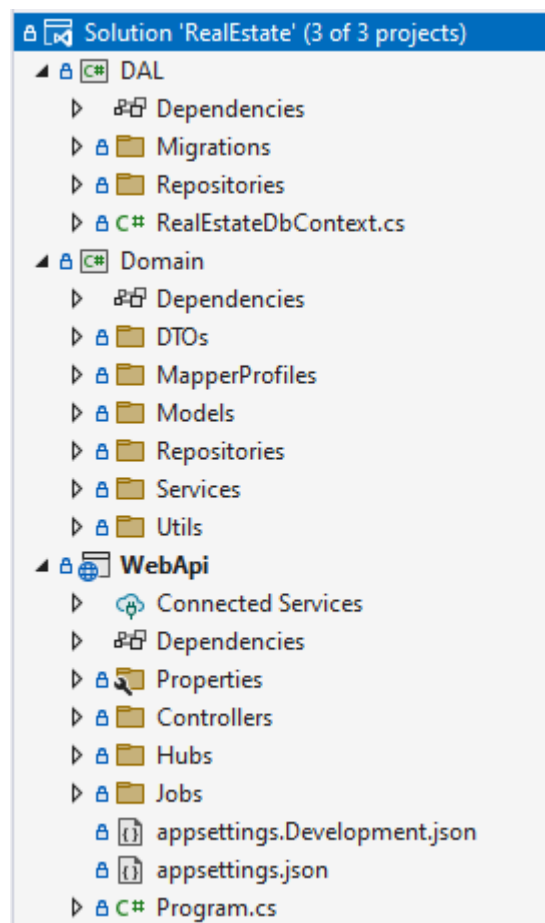
Ábra 9: Az alkalmazás komponensei

Az ábrán a szerver komponensben lévő adatbázis és REST API réteget egybetartozónak jelöltem, de ezeket a részeket lehet külön fizikai gépeken vagy

szolgáltatásokon futtatni megfelelő konfigurációval, ugyanez igaz a frontend architektúrára is.

## 4.2 Szerveroldali architektúra

A szerveroldali komponenst több részre osztottam, amelyeket a Domain Driven Design elv szerint definiáltam. A szerver magjában a Domain réteg áll, ez tartalmazza az üzleti logikát implementáló szolgáltatásokat és a felhasznált entitásosztályokat. A külső erőforrások eléréséért a Data Access Layer (DAL) felelős, amely az adatbázissal kommunikál Entity Framework Core segítségével, és a WebApi projekt teszi lehetővé azt, hogy egy REST API-on és SignalR hub-on keresztül felhasználhassuk a Domain szolgáltatásait.



Ábra 10: A szerveroldali komponens mappaszerkezete

### 4.2.1 Domain réteg

A Domain réteget több bounded context-ekre bontottam, amelyek különböző szolgáltatásokra fókuszálnak (ilyenek a hirdetések kezelése, üzenetek kezelése és

felhasználók kezelése), és egységbe zárják a funkciók megvalósítását. A szolgáltatásoknak el kell érniük az adatot ahhoz, hogy megfelelően tudjanak működni, de fontos az adatbázistól való függetlenség megtartása. A Repository tervezési minta használatával elérhetjük azt, hogy a szolgáltatások absztrakcióktól függhenek és ne az implementációtól, könnyedén le tudjuk cserélni az adatelérés implementációját, valamint a tesztelést is megkönnyítheti. A szolgáltatások általában nem az entitás összes attribútumait adják vissza, mivel lehetnek benne olyan érzékeny adatok, melyeket nem lenne érdemes elérhetővé tenni a felhasználó számára. Ilyen esetekben Data Transfer Object-eket (DTO) használtam az adatok továbbításához, melyekben csak a szükséges mezők láthatóak.

### 4.2.2 Web API

A WebApi nevű projektben valósítottam meg a kiszolgáló szerver, ami az ASP.NET Core könyvtárt használja. Ebben a rétegben találhatóak a REST API-hoz tartozó Controller osztályok és a SignalR-hez szükséges Hub osztályok.

Minden egyes szolgáltatáshoz tartozik Controller osztály, amelyek a Domain-ben létrehozott szolgáltatásokat használják, itt találhatóak a definiált REST interfészek és ezek kezelik a http kéréseket. A Controller-ekben meg lehet adni, hogy mely végponthíváshoz kell autentikálnia magát a felhasználónak a hívása előtt, továbbá az elérési utat, és a paramétereket, amiket vár.

A Hubs mappában találhatjuk azoknak a funkcióknak a kiszolgálóit, amelyeknél valós idejű kommunikálást szeretnénk elérni. Ezeket a végpontokat a Controllerek-hez hasonlóan tudjuk felkonfigurálni, és eseménykezelőket is tudunk definiálni a Hub különböző eseményeihez, például a csatlakozáshoz és a lecsatlakozáshoz.

A periodikusan elvégzendő feladatok a Jobs nevű mappában találhatóak meg. Ezeknek a feladatoknak az implementálásához a Quartz nevű könyvtár segítségét használtam, amivel könnyedén tudunk függvényhívásokat ütemezni.

A Program.cs fájlban a szerver konfigurációját írhatjuk le, a függőséginjektáláshoz szükséges osztályobjektumokat itt tudjuk leírni. Ez a réteg felelős az autentikációért is, ellenőrzi az adott kéréshez tartozó access token érvényességét, és feloldja a hozzátartozó felhasználót. A feladatok ütemtervét is itt kell felkonfigurálni, meg tudjuk adni, hogy mikor és milyen időközönként történjen a végrehajtása.

### 4.2.3 Adatelérési réteg

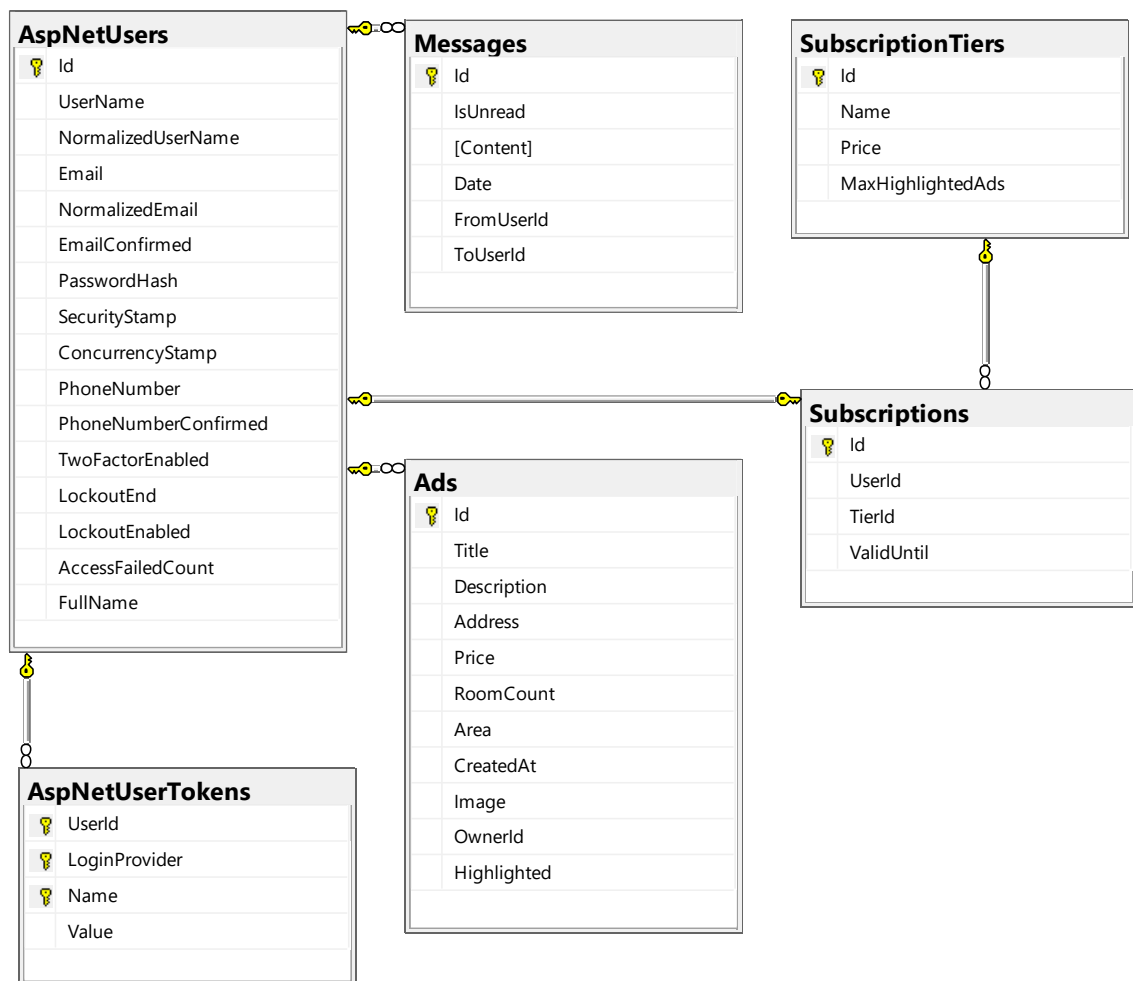
Az adatelérési rétegben az adatbázissal való kommunikációt valósítom meg Entity Framework Core segítségével. A réteg több Repository osztályból áll, amelyek a Domain-ben definiált interfészeket implementálják, ezek kezelik a CRUD műveleteket (Create, Read, Update, Delete) az adatbázisban lévő entitásokon. Az adatbáziskapcsolat a RealEstateDbContext osztályban jön létre, amely a DbContext ösosztályt örökli, DbSet-eken keresztül lehet lekérdezni az entitásokat, amely támogatja a LINQ kifejezéseket is. Az alkalmazás jelenleg Microsoft SQL adatbázismotort használ, de az Entity Framework Core több, más cégek által létrehozott adatbázis-kezelő rendszert támogat, így könnyedén le tudjuk cserélni.

## 4.3 Adatbázis felépítése

Az alkalmazásom fejlesztése során Microsoft SQL relációsadatbázis-kezelő rendszert használtam. A tervezéskor Code first megközelítést alkalmaztam az Entity Framework Core segítségével, a táblák szerkezete követi a Domain rétegben definiált modellek szerkezetét. Az adatbázis sémája a 8. ábrán látható.

Az ASP.NET Core Identity több felhasználóval kapcsolatos táblát is generált, de azok szerepe elhanyagolható, ezért nem szerepelnek. Az ábrán lévő táblákat tekintsük át.





Ábra 11: Az adatbázis sémája

### 4.3.1 AspNetUsers

A felhasználó fontos adatait tartalmazó táblázat. Az adatok között szerepel a felhasználó e-mail címe, teljes neve, felhasználóneve, jelszava és opcionálisan a telefonszáma. A további, fel nem sorolt attribútumok az ASP.NET Core Identity működéséhez szükségesek. A jelszó hashelt formában érhető el.

### 4.3.2 Ads

Ebben a táblázatban az ingatlanhirdetések adatait tárolom el. Szerepel benne a hirdetés címe, leírása, létrehozási ideje, a tulajdonos idegen kulcsa, az ingatlan címe, ára, mérete, kiemelt-e, és a róla készült kép Base64 formátumban.

### 4.3.3 Messages

A felhasználók közötti üzenetváltásokat tárolom ebben a táblában. Egy sor a küldött üzenet tartalmát, idejét, olvasottságát, feladóját és címzettét tárolja el.

### 4.3.4 SubscriptionTiers

Ez a táblázat tartalmazza az összes olyan előfizetési csomagot, amire elő tud fizetni egy felhasználó. Egy sor megmondja az előfizetésről, hogy mi a neve, az ára, és azt, hogy hány hirdetést tud kiemelni.

### 4.3.5 Subscriptions

Az összes felhasználóhoz kötött előfizetést tartalmazza ez a táblázat. Tartalmazza az előfizetéshez tartozó felhasználót, az előfizetési csomagot, amire előfizetett, és azt, hogy mikor fog lejárni. A felhasználók és az előfizetések között egy-egy kapcsolat van, ha a felhasználó megpróbál többször előfizetni, akkor a rendszer hibával jelzi.

## 4.4 Kliensalkalmazás

A kliensalkalmazás megvalósításához React keretrendszert használtam. A kliens megírása közben arra törekedtem, hogy a projekt mappaszerkezete átlátható és jól strukturált legyen, külön mappát hoztam létre a REST API-t hívó függvényeknek; a komponenseknek; a modelleknek, amelyek a http kéréseknél kerülnek elő; a context-eknek és a hook-oknak; és az adott oldal struktúráját leíró fájloknak.

A valós idejű kommunikáláshoz SignalR könyvtárt használtam. Mivel a Hub-hoz tartozó kapcsolatot több komponensben is kell használni, a hosszú, több komponensen keresztül áthidaló paraméterláncolás (prop drilling) helyett Context-et használok. Context definiálásával a szülő komponens könnyedén meg tudja osztani objektumokat a fában alatta lévő komponenseknek.

A komponenseket is több részre osztottam: a features mappában funkcióhalmazonként vannak rendezve az elemek, mivel ezek egy adott oldal részét képezik és nehéz őket általánosan újra felhasználni, a components mappában pedig könnyen újrahasználható komponensek helyezkednek el.

```

✓ src
  > api
  > components
  > context
  ✓ features
    > Ad
    > Message
    ✓ User
      > LoginModal
      > MessageList
      > Profile
    ✓ RegisterModal
      TS index.ts
      TS RegisterModal.tsx
      # style.css
      TS types.ts
      > ShowListings
      > Subscription
  > hooks
  > layouts
  > model
  > pages
  > util
  TS App.tsx
  # main.css

```

**Ábra 12: A kliens mappaszerkezete**

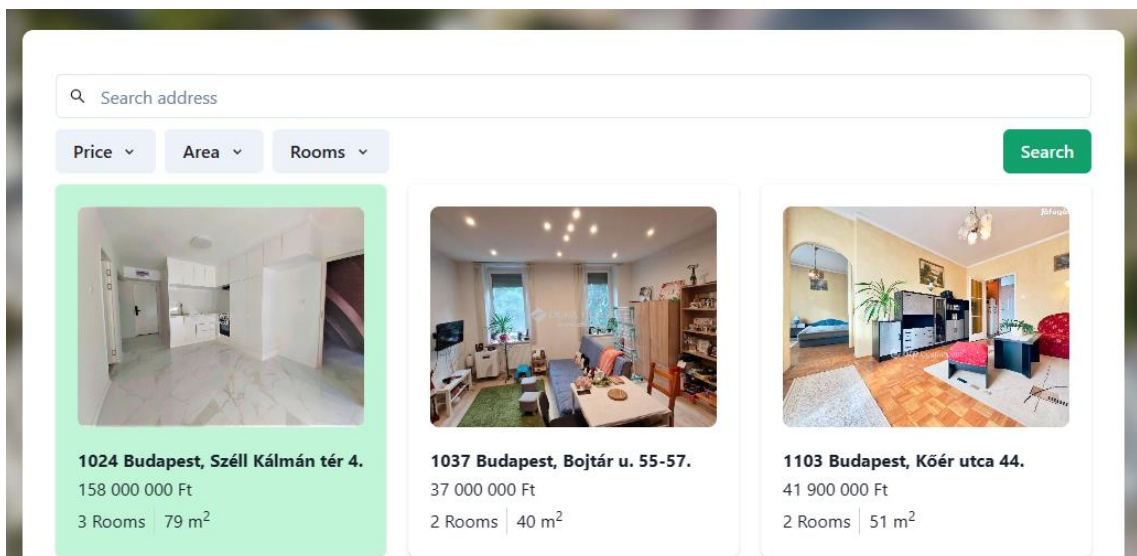
Egy UI komponenshez tartozó mappán belül több fájl is található: az egyik fájl tartalmazza a felület deklaratív leírását és működését, a `style.css`-ben a komponens megformázása található, és a `types.ts` fájl az erősen hozzáköthető típusdefiníciókat tartalmazza.

## 5. Megvalósítás

Ebben a fejezetben az alkalmazást fogom bemutatni képekkel illusztrálva, továbbá részletezni fogom fontosabb funkciók implementációs megoldásait.

### 5.1 Hirdetések böngészése

A hirdetések böngészése az ingatlanhirdető alkalmazás egyik legalapvetőbb funkciója. A felhasználók a felületen kereshetnek ingatlanokat, továbbá szűrhetnek a hirdetések részleteire is. A szűrőben megadhatjuk az ingatlan címét, az ingatlan árát és méretét, és a szobák számát. Az ingatlanhirdetések száma elérheti a százazres nagyságrendet is, ezért egy végponthívásban lekérhető találatok száma korlátozott, amit szerveroldali lapozással értem el. A felhasználó a weboldal alján tud lapozni, és a már lekért eredményeket a kliens eltárolja egy gyorsítótárban. A hirdetésekben megjelennek az ingatlanok fontosabb adatai, kiemelt hirdetések esetén az elem egy feltűnőbb zöld hátteret vesz fel, és a feladás időpontjától függetlenül a lista elején jelenik meg.



Ábra 13: A szűrő és az eredmények megjelenítése

A hirdetéseket egy szerveroldali végponthívás segítségével kérem le. Ezt a végpontot a `getAds` függvény segítségével hívom meg, amely az `axios` könyvtárt használja. A végponthívás kódrészlete alább található:

```
const adApi = axios.create({
  baseURL: 'https://localhost:7202',
});
```

```
export const getAds = async (options?: GetAdsOptions) => {
  const response = await adApi.get<AdList>('/api/ad', {
    params: options,
  });
  return response.data;
};
```

Az adApi konstans egy axios példányt tárol, amelynek megadtam a szerver elérési útját. A szűrőfeltételek az options paraméterben találhatóak, ezeket query paraméterekként adom át a szervernek. A végpontokon keresztül lekérdezett értékek kezeléséhez és eltárolásához a React Query könyvtárat használom.

```
const { isLoading, isError, error, data, isPreviousData } = useQuery<
  AdListDto,
  AxiosError
>({
  queryKey: ['ads', pageIndex, adFilter],
  queryFn: () => getAds({ pageIndex, ...adFilter }),
  keepPreviousData: true,
});
```

A useQuery visszatérési értéke egy lekérdezést reprezentál, amely automatikusan frissül bizonyos események hatására. A frissítés konfigurálható. A lekérdezés paramétereként megadhatunk egy kulcslistát, amellyel tudunk referálni rá. Ha a listába egy változót adunk meg, akkor annak változása esetén a lekérdezés frissül. A queryFn paraméternek egy függvényt adtam meg, ezen keresztül történik az adat lekérése a megadott szűrési feltételeknek megfelelően.

A szűrő implementálásához egy komponenst készítettem, amelyen egy űrlap található. A komponens vár egy kötelező onSubmit függvényparamétert, amely a szűrő űrlapon lévő keresés gomb megnyomása után hívja meg. Típusokat definiáltam a komponensnek átadható paramétereknek (props) és az űrlap által visszaadott objektumnak.

```
export type AdFilterProps = {
  onSubmit: SubmitHandler<AdFilterFormInput>;
};

export type AdFilterFormInput = {
  address: string;
  minPrice?: number;
  maxPrice?: number;
  minArea?: number;
  maxArea?: number;
  minRoomCount?: number;
```

```
maxRoomCount?: number;
};
```

Az űrlap kezeléséhez a React Hook Form könyvtárat használtam, amely a megadott értékek validálását megkönnyíti. A useForm hook függvény egy objektumot ad vissza, amely egy űrlapot reprezentál. Az objektum register függvényével tudjuk felkonfigurálni az űrlap bemeneteit, a handleSubmit meghívásával pedig validáljuk és kezeljük a beküldés eseményt.

```
const {register, handleSubmit, watch, formState: { errors }} =
  useForm<AdFilterFormInput>({
    resolver: yupResolver(adFilterSchema),
    mode: 'onChange',
  });

<form onSubmit={handleSubmit(onSubmit)}>
  /*.../*
  <FormControl isValid={!errors.minArea}>
    <NumberInput>
      <NumberInputField
        size={5}
        {...register('minArea')}
        placeholder="Min"
      />
    </NumberInput>
  </FormControl>
  /*.../*
</form>
```

A validációs sémák leírásához a yup könyvtárat használtam. A séma jól konfigurálható, a legtöbb esetben a bemenet típusát, az opcionálisát és a bemenet nevét határozom meg. Validálás esetén, ha az egyik feltétel nem felel meg, akkor hibaüzenettel tér vissza, amelyet meg tudunk változtatni.

```
const adFilterSchema = yup.object<AdFilterFormInput>({
  address: yup.string().nullable(),
  minPrice: yup
    .number()
    .label('Minimum price')
    .positive()
    .integer()
    .optional()
    .transform((_, val) => (Number(val) ? Number(val) : undefined)),
  /*...*/
});
```

A szűrő validálása és megerősítése után meghívódik a paraméterként átadott `onSubmit` függvény, amely a hirdetésbongészó komponensben található. A módszer beállítja az `adFilter` state-t és megváltoztatja a query paramétereket a webkliens URL mezőjében. Mivel a lekérdezés kulcslistája tartalmazza az `adFilter` állapotot, emiatt az adatok automatikusan frissülni fognak a szűrőnek megfelelően. Ha a felhasználó egy URL-re kattintva jut el erre a nézetre, akkor a komponens az első kirajzoláskor kiolvassa az összes query paramétert, és beállítja a szűrőfeltételeket.

A hirdetés listázásáért felelős végponthívást az `AdController` osztály kezeli. Az osztály függőséginjektálás segítségével kap egy `AdService` példányt, amelyen keresztül hirdetésekkel kapcsolatos műveleteket tud végezni. A böngésző nézet a `GetAds` függvényt hívja meg, melyet az „api/ad” címen keresztül tud elérni GET eljárással. Egy GET eljárást a `HttpGet` attribútummal lehet jelölni.

```
[HttpGet]
[ProducesResponseType(StatusCodes.Status200OK)]
public AdListDTO GetAds([FromQuery] GetAdsParameters parameters)
{
    return _adService.GetAds(parameters);
}
```

A query paraméterek lekérdezéséhez a `FromQuery` attribútumot adtam meg a `parameters` nevű paraméternek, ezen keresztül érhetőek el a szűrések. Ezeknek a paramétereknek egy osztályt definiáltam, így a szűrés könnyebben átadható más eljárásnak. Ennek a kódrészlete lent látható:

```
public class GetAdsParameters
{
    public string? Username { get; init; }
    public string? Address { get; init; }
    public int PageIndex { get; init; } = 1;
    public int PageSize { get; init; } = 12;
    public int MinPrice { get; init; } = 0;
    public int MaxPrice { get; init; } = int.MaxValue;
    public int MinArea { get; init; } = 0;
    public int MaxArea { get; init; } = int.MaxValue;
    public int MinRoomCount { get; init; } = 0;
    public int MaxRoomCount { get; init; } = int.MaxValue;
}
```

Mivel az összes szűrési feltétel opcionális, minden változónak van alapértelmezett értéke. Az eljárás meghívja a szolgáltatás `GetAds` függvényét, amely a függőséginjektálás segítségével megkapott, `IAdRepository` interfészt implementáló objektumot használja.

Mivel az AdService osztály egy IAdRepository interfészt implementáló objektumot vár, emiatt az adatelérést megvalósító implementációt könnyedén le tudjuk cserélni. Ez hasznos lehet tesztelés, vagy egy másik adatbázismotorra való átállás esetén.

Az adatok eléréséhez Entity Framework Core ORM könyvtárat használtam. Az adatbázissal történő kommunikálást a RealEstateDbContext osztályon keresztül tudjuk megtenni, amely a DbContext ösztályból származik le. Ebben az osztályban írjuk le az adatbázisban elérhető táblákat, és komplexebb relációs jellemzőket is tudunk definiálni az OnModelCreating függvény felüldefiniálásával. Én az adatbázis leképezéséhez a code-first megközelítést választottam a DDD tervezési minta alapján, erre kiváló eszközöket biztosít a könyvtár.

Az AdRepository osztályban találhatóak az adatok eléréséhez szükséges lekérdezésdefiniciók. A böngészőhöz szükséges lekérdezést ilyen módon írtam le:

```
public IEnumerable<AdDTO> GetAds(GetAdsParameters parameters)
{
    var query = _context.Ads
        .Where(ad => parameters.MinPrice <= ad.Price && ad.Price <=
parameters.MaxPrice)
        .Where(ad => parameters.MinArea <= ad.Area && ad.Area <=
parameters.MaxArea)
        .Where(ad => parameters.MinRoomCount <= ad.RoomCount &&
ad.RoomCount <= parameters.MaxRoomCount);

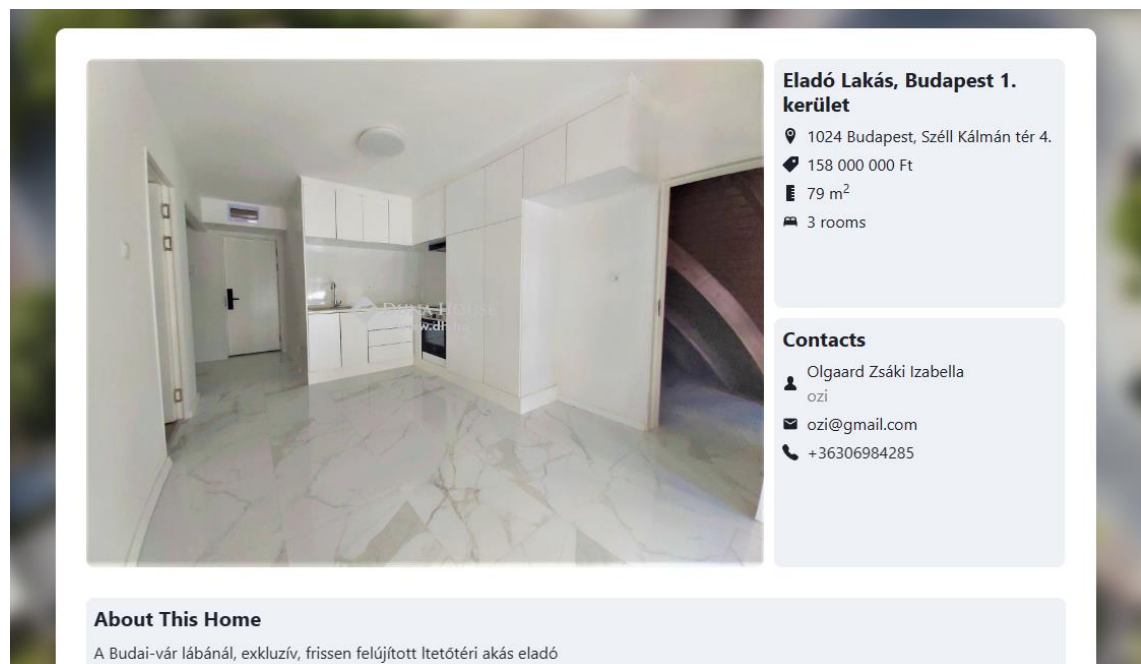
    if (!string.IsNullOrEmpty(parameters.UserName))
    {
        query = query.Where(ad => ad.Owner.UserName ==
parameters.UserName);
    }
    if (!string.IsNullOrEmpty(parameters.Address))
    {
        query = query.Where(ad =>
ad.Address.Contains(parameters.Address));
    }

    return query
        .OrderByDescending(ad => ad.Highlighted)
        .ThenByDescending(ad => ad.CreatedAt)
        .Skip((parameters.PageIndex - 1) * parameters.PageSize)
        .Take(parameters.PageSize)
        .Select(a => _mapper.Map<AdDTO>(a))
        .ToArray();
}
```



A lekérdezések leírásához LINQ kifejezéseket használtam. A Where kifejezéssel tudunk feltételek alapján szűrni, a Skip és a Take segítségével pedig a serveroldali lapozást valósítottam meg. A hirdetésekön kívül a továbblapozhatóságot is átadja végpont, hogy a kliens le tudja tiltani a következő oldalra való lapozást.

A hirdetésekre kattintva a részletes nézetre térünk át. Ezen a nézeten az alap információkon kívül megjelennek a hirdető adatai, és egy hosszabb, felhasználó által írt leírás.



Ábra 14: A hirdetés részletes nézete

A részletes nézetre való átirányításhoz a React Router könyvtárat használtam. Ennek segítségével állítom be azt, hogy bizonyos elérési utakon mely komponenseket kell megjeleníteni. Ez a konfiguráció az App.tsx fájlban található meg.

```
function App() {  
  return (  
    <>  
    <Routes>  
      <Route element={<Layout />}>  
        <Route index element={<Navigate to="/home" />} />  
        <Route path="home" element={<HomePage />} />  
        <Route path="browse" element={<AdList />} />  
        {/* ... */}  
        <Route path="ad/:adId" element={<AdPage />} />  
        <Route path="*" element={<NotFound />} />  
      </Route>  
    </Routes>  
  )  
}
```

```

        </>
    );
}

```

Az elérési utakat Routes tag-ek közé kell elhelyezni. A nézeteim egy Layout nevű komponensen belül jelennek meg, amelyen a menüfejléc és a lábléc helyezkedik el. Amikor egy hirdetés részletes nézetére térünk át, az URL felveszi az „ad/:adId” formátumot, ahol az :adId helyén a hirdetés azonosítója jelenik meg. A useParams hook segítségével tudja a komponens, hogy melyik hirdetés információit kell a kiszolgáló szervertől lekérni.

Amikor a kliens elkéri a hirdetés adatait, az AdRepository-ban egy rekord lekérésével olyan érzékeny információk is megjelennek (például jelszó hash), amelyeket nem feltétlenül szeretnénk megosztani adott felhasználókkal. Ennek kiküszöbölésére olyan DTO-kra képeztem le az objektumokat, amelyben a nem megosztani kívánt adatok hiányoznak. A hirdetés esetében a felhasználó osztálynak kell DTO osztályt létrehozni, amelyben csak a megosztható információk találhatók meg. Ennek a kódja itt látható:

```

public class UserDTO
{
    public string FullName { get; set; }
    public string Username { get; set; }
    public string Email { get; set; }
    public string PhoneNumber { get; set; }
}

```

Az objektumok leképezésének megkönnyítéséhez az AutoMapper könyvtárat használtam, melyet a Program.cs fájlban tudunk felkonfigurálni, és függőséginjektáláson keresztül tudunk használni.

```

public AdWithOwnerDTO? GetAdById(int adId)
{
    var dbAd = _adRepository.GetAdById(adId);
    if (dbAd == null)
    {
        return null;
    }
    return _mapper.Map<AdWithOwnerDTO>(dbAd);
}

```

A végponthívás useQuery és axios segítségével történik meg, a hirdetésböngészéshez hasonlóan. Ha az adott azonosítóval nem létezik hirdetés, akkor a szerver egy 404 Not Found válasszal tér vissza, és a webkliens az alábbi nézetre irányít át:



Ábra 15: A hirdetés nem található.

## 5.2 Regisztrálás

A felhasználóknak lehetőségük van fiókot készíteniük, melynek használatával több funkciót érhetnek el. A felhasználói felületen a menüsorban található „Sign Up” gombra kattintva egy felugró ablak jelenik meg, melyet a ChakraUI komponenseivel valósítottak meg.

A registration form titled 'Create your account' with a close button (X) in the top right corner. The form contains five input fields: 'Username' (with the text 'teszt.elek'), 'Email address' (highlighted with a red border and a red error message 'Email is a required field' below it), 'Name' (highlighted with a red border and a red error message 'Name is a required field' below it), 'Password' (with a blue border, a password mask '.....', and a toggle icon), and 'Phone number'. At the bottom right, there are two buttons: a green 'Sign up' button and a light blue 'Cancel' button.

Ábra 16: A regisztrációs ablak megjelenése

A felhasználóknak a regisztráláshoz meg kell adniuk egy egyedi felhasználónevet, email címet, teljes nevüket, és egy jelszót. Opcionálisan telefonszámot is megadhatnak, mely alternatív kommunikációs módot biztosít az érdeklődőknek. A megadott adatokat yup segítségével validáltam. Mivel a könyvtár nem képes a telefonszám érvényességének

ellenőrzésére, emiatt Regular Expression-t használtam [19], és a yup test függvényével ellenőriztem.

```
const phoneRegExp = /^((\+[1-9]{1,4}[ -]?)|(\([0-9]{2,3}\)[ -]?)|([0-9]{2,4})[ -]?)*)?[0-9]{3,4}[ -]?[0-9]{3,4}$/;

const registerSchema = yup.object<RegisterFormInput>({
  /*...*/
  phoneNumber: yup
    .string()
    .label('Phone number')
    .test(
      'phoneNumber',
      () => 'Phone number is not valid.',
      (value, testContext) => !value || phoneRegExp.test(value),
    ),
});
```

Helytelen bemenet esetén a felhasználóknak szembeűnő visszajelzést kell küldeni. A ChakraUI komponenseivel egyszerűen tudunk stílusos hibáüzeneteket elhelyezni. A FormControl isValid paraméter beállításával tudjuk az Input komponensnek jelezni az érvénytelen bemenetet, és a hibáüzenetek megjelenítését is kezeli.

```
<FormControl className="register-form-control" variant="floating"
  isValid={!errors.email}>
  <Input placeholder=" " {...register('email')} />
  <FormLabel>Email address</FormLabel>
  <FormErrorMessage>{errors.email?.message}</FormErrorMessage>
</FormControl>
```

A regisztrációs kérést POST eljárásaként küldöm el a szervernek axios segítségével. A végponthívás állapotát useQuery hook helyett useMutation-nel kezelem. Ezt a hook-ot általában CRUD műveletek esetén használják, vagy olyan műveletek elvégzésekor, amelyeknek szerveroldali hatása van. A végponthívást a „Sign Up” gomb megnyomásával indíthatjuk el, amely hatására a mutate függvény meghívódik.

```
const { mutate, isLoading, isError, error } = useMutation<
  unknown,
  AxiosError,
  SignUpProps
>({
  mutationFn: (data: SignUpProps) => userSignUp(data),
  onSuccess: () => {
    onClose();
    successOnOpen();
  }
});
```

```

    },
  });

const onSubmit: SubmitHandler<RegisterFormInput> = (
  data: RegisterFormInput,
) => {
  mutate(data);
};

<ModalContent as="form" onSubmit={handleSubmit(onSubmit)}>

```

A megadott adatokat szerveroldalon is ellenőrzöm a Controller osztály ModelState.IsValid property-jén keresztül. Az egyes adattagokra elhelyeztem a Required attribútumot, illetve a DataType attribútumot megfelelő paraméterezéssel.

```

public class SignUpDTO
{
    [Required]
    public string UserName { get; set; }
    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }
    [Required]
    public string FullName { get; set; }
    [Required]
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
    [DataType(DataType.PhoneNumber)]
    public string PhoneNumber { get; set; }
}

```

Az ellenőrzés után az UserRepository-ban létrehozom a felhasználót. Mivel a felhasználói fiókok érzékeny adatokat tartalmaznak, a felhasználók kezelését az ASP.NET Core Identity könyvtárra bízom. Ennek a szolgáltatásnak a használatához a WebApi-t megfelelően fel kell konfigurálni. A használt konfiguráció itt látható:

```

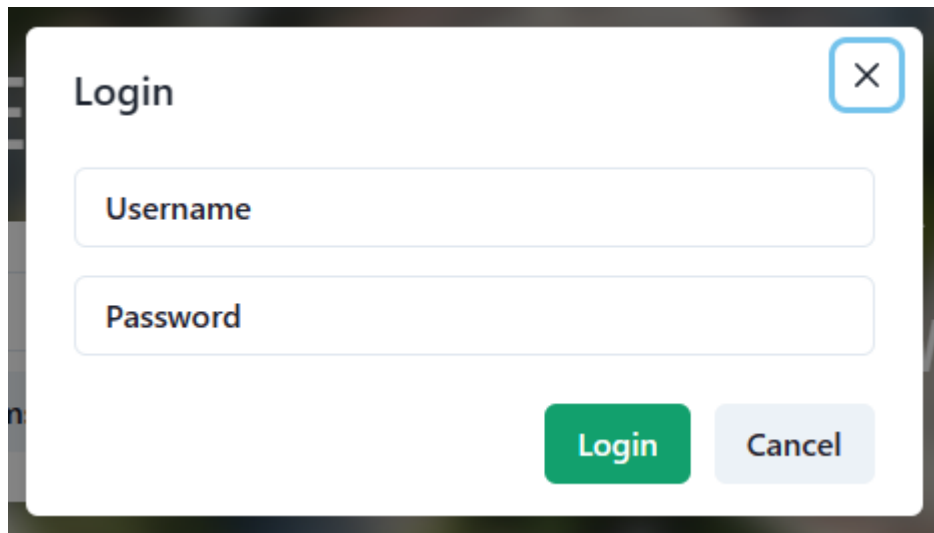
builder.Services.AddIdentity<User, IdentityRole>(options =>
options.User.RequireUniqueEmail = true)
    .AddEntityFrameworkStores<RealEstateDbContext>()
    .AddDefaultTokenProviders();

```

Felhasználói fiók készítése során az Identity modul ellenőrzi a jelszó erősségét. Ha valamilyen feltételnek nem tesz eleget, akkor a Controller osztály visszajelzést küld a kliensnek. A jelszavakat a PBKDF2 szabvány szerint tárolja el, amely mellé a HMAC-SHA512 hash algoritmust és 128 bites salt-ot használ [20].

## 5.3 Bejelentkezés

A felhasználó regisztrálás után bejelentkezhet a megfelelő felhasználónév és jelszó kombinációval. A menüsorban a „Sign in” gombra kattintva megjelenik a felugró ablak, amelyen keresztül be lehet jelentkezni. Ennek az ablaknak a működése többnyire megegyezik a regisztrációs ablak működésével.



Ábra 17: Bejelentkezés megjelenése

A szerveroldalon a felhasználó azonosítása a JSON Web Token [21] szabvány alapján történik. A szabvány segítségével biztonságosan tud a webkliens és a szerver információkat küldeni egymásnak, és kis mérete miatt alkalmas http környezetben való használatra is. A szerver a felhasználónév és a jelszó páros ellenőrzése után generál egy JWT token-t. Ez a token tartalmazza a felhasználónevet, amely az azonosításhoz szükséges, és digitálisan alá van írva egy titkos kulcs segítségével. Mivel a kulcsot csak a szerver ismeri, így egy harmadik fél nem képes olyan token-eket generálni, amelyeket a szerver elfogad annak ismerete nélkül. Ezzel a módszerrel tudjuk biztonságosan azonosítani a felhasználót érzékeny műveletek elvégzése előtt. A bejelentkezés implementációja szerveroldalon lent látható:

```
public async Task<TokenDTO> LoginAsync(LoginDTO loginDTO)
{
    var result = await _userRepository.LoginAsync(loginDTO);

    if (!result.Succeeded)
    {
        throw new AuthenticationException(result.ToString());
    }
}
```

```

var authClaims = new List<Claim>()
{
    new Claim(ClaimTypes.Name, loginDTO.UserName),
    new
Claim(Microsoft.IdentityModel.JsonWebTokens.JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString()),
};
var authSignInKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["JWT:Secret"]
));

const int expiresIn = 1440; // minutes
var token = new JwtSecurityToken(
    issuer: _configuration["JWT:ValidIssuer"],
    audience: _configuration["JWT:ValidAudience"],
    claims: authClaims,
    expires: DateTime.Now.AddMinutes(expiresIn),
    signingCredentials: new SigningCredentials(authSignInKey,
SecurityAlgorithms.HmacSha256Signature)
);

var accessToken = new JwtSecurityTokenHandler().WriteToken(token);

return new TokenDTO {
    AccessToken = accessToken,
    ExpiresIn = expiresIn,
    UserName = loginDTO.UserName,
};
}

```

A kliens a végpont meghívása után megkapja a JWT token-t, amelyet el kell tárolnia későbbi használatra, ehhez a React Auth Kit könyvtárat használtam. Mivel az autentikációnak a kliens legtöbb komponensére hatása van, ezért az AuthProvider komponenst a DOM fa gyökerében kell elhelyezni, hogy mindenhol elérhető legyen a token. A könyvtár konfigurációja ezen a komponensen keresztül történik meg.

```

<React.StrictMode>
  <AuthProvider
    authType="cookie"
    authName="_auth"
    cookieDomain={window.location.hostname}
    cookieSecure={false}
  >
    <MessageHubProvider>
      <QueryClientProvider client={queryClient}>
        <ChakraProvider theme={theme}>
          <BrowserRouter>

```

```

        <App />
      </BrowserRouter>
    </ChakraProvider>
  </QueryClientProvider>
</MessageHubProvider>
</AuthProvider>
</React.StrictMode>

```

A token eltárolása közvetlenül a végponthívás után történik a useSignIn hook segítségével. Kijelentkezés esetén a useSignOut hook-ot tudjuk használni, amely kitörli a JWT token-t a tárhelyből. Ha a végponthívás autentikációt igényel, a token-t a kérés Authorization fejlécében kell elhelyezni.

```

export const createAd = async (data: AdProps, accessToken: string) => {
  const response = await adApi.post<Ad>('/api/ad', data, {
    withCredentials: true,
    headers: {
      Authorization: accessToken,
    },
  });
  return response.data;
};

```

A felhasználó bejelentkezés után készíthet és módosíthat hirdetéseket, illetve küldhet üzeneteket hirdetőknak vagy érdeklődőknek.

## 5.4 Hirdetések készítése és menedzselése

A hirdetések készítését a felhasználó a menübáron keresztül tudja elérni a profil ikonra kattintva. A nézeten egy űrlap jelenik meg, amelyen megadhatjuk az ingatlan adatait, a kitűzött árat, és az arról készített képet. Ha minden szükséges adat ki van töltve, akkor a „Submit” gombra kattintva létrejön a hirdetés, és a kliens átnavigál a részletes nézetre.

A felhasználónak van lehetősége kiemelt hirdetést is készítenie a „Highlighted” kapcsoló beállításával. Ehhez előfizetéssel kell rendelkeznie, illetve ellenőriznem kell, hogy elérte-e már a maximum kiemelt hirdetések számát. Ezt az „api/ad/highlight” végponthívással tudom megtenni. Ha nem tesz eleget semelyik feltételnek sem, akkor a gomb nem használható. Egy szöveges indoklás is megjelenik, ha a gombra teszi az egerét a felhasználó, amely tájékoztatja őt.



## Create advertisement

Title

Description

Address

Price

Room count

Area

 m<sup>2</sup>

☐ Highlighted

Image







Ábra 18: Űrlap a hirdetés elkészítéséhez



Az adatok validálását az előző funkciókhoz hasonlóan valósítottam meg, frontend-en a yup könyvtár használatával, backend-en pedig annotációkkal, a ModelState ellenőrzésével, illetve az adatbázisban lévő információk segítségével.

A hirdetés módosításához a felhasználónak át kell navigálnia arra a nézetre, ahol az összes hirdetését láthatja. Ezt a menübáron lévő profil ikonra kattintva tudja megtenni a megfelelő menügombra kattintva.

Ezen a nézeten egy táblázatot láthatunk, amelyen megjelenik a hirdetéshez tartozó cím, az ingatlan címe, a hozzátartozó kép és a létrehozásának dátuma. A táblázat jobboldalán láthatjuk a gombokat, amelyeken keresztül tudjuk módosítani, illetve törölni a hirdetést.

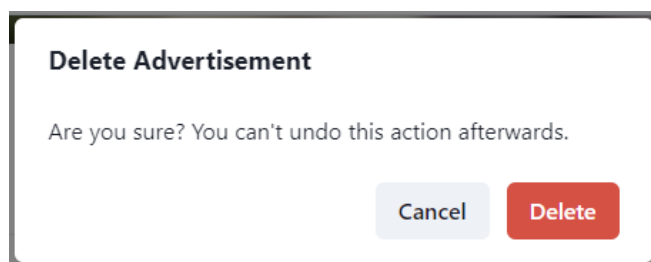
## Your listings

LISTING	CREATION DATE	ACTIONS
 Eladó Lakás, Budapest 1. kerület 1024 Budapest, Széll Kálmán tér 4.	6/5/2023	 
 Eladó Lakás, Budapest 3. kerület 1037 Budapest, Bojtár u. 55-57.	6/5/2023	 

 1 

Ábra 19: A felhasználó hirdetései

A törlés gombra kattintva egy felugró ablak jelenik meg, ahol meg kell erősíteni a műveletet. Megerősítés után meghívódik az `AdController DeleteById` végpontja. Ezután a kliens invalidálja az ingatlanlista query-jét, mely hatására az adatok frissülnek, és a törölt hirdetés eltűnik.



Ábra 20: Felugró ablak a törlés megerősítéséhez.

A módosításra kattintva áttérünk egy másik nézetre, melyen ugyanaz az űrlap jelenik meg, mint a hirdetések készítése nézeten. Az űrlapnak létrehoztam egy `AdForm` komponenst, amelynek meg lehet adni egy végpontot hívó függvényt, illetve egy hirdetés objektumot. Ezt a hirdetés objektumot módosítás esetén adom meg, hogy beállítsam az űrlap alapértelmezett értékeit. A „Submit” gombra kattintva a komponens meghívja a paraméterben átadott függvényt.

A hirdetés képét az „Upload” gombra kattintva tudja megadni a felhasználó. Mivel a Chakra UI nem tartalmaz olyan komponenst, amellyel fájlt lehet feltölteni, emiatt egy saját komponenst kellett létrehoznom. A kép kiválasztása után megjelenik a fájl neve. Végponthívás előtt a képet átkódolom Base64 formátumba, és a szervernek szöveges formátumban továbbítom és tárolom el az adatbázisban. A kép átkonvertálásához tartozó kód részlet lent látható:

```

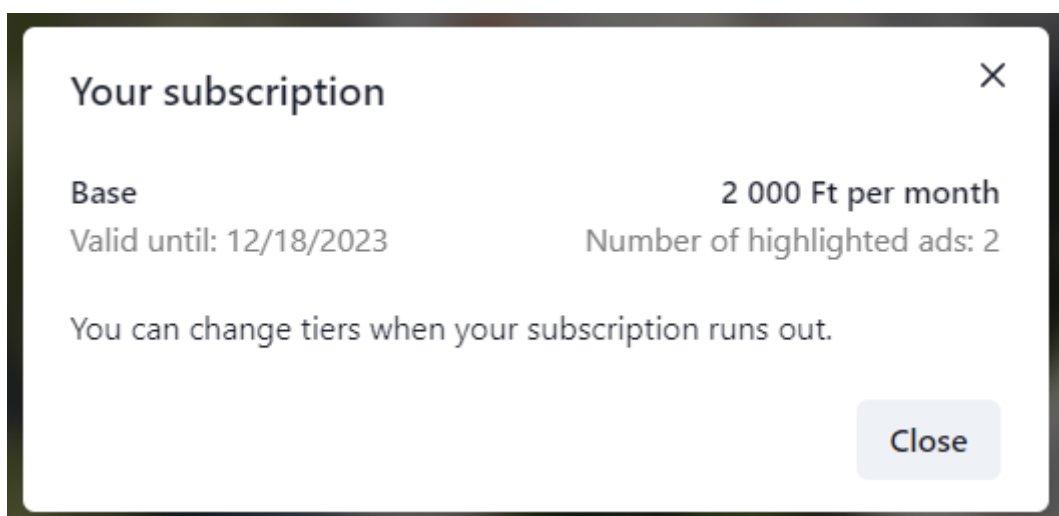
const createAdMutationFn = useCallback(
  async (data: AdFormInput) => {
    const imageBase64 = await toBase64(data.image[0]);
    return createAd(
      { ...data, image: imageBase64, createdAt: new Date(Date.now()) },
      authHeader(),
    );
  },
  [authHeader],
);
export const toBase64 = (file: File): Promise<string> =>
  new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.readAsDataURL(file);
    reader.onload = () => resolve(reader.result as string);
    reader.onerror = (error) => reject(error);
  });

```

A kép eltárolását többféleképpen is meg lehet oldani. Ilyen például a fájlrendszerben való tárolás, amely gyorsabb, de én az adatbázisban tárolom az egyszerűség miatt.

## 5.5 Előfizetések

A felhasználóknak lehetőségük van előfizetés vásárlására, melyet a menübárban lévő profil ikonra kattintva tudnak elérni. A gombra kattintva egy felugró ablak jelenik meg, mely tartalma függ attól, hogy rendelkezik-e előfizetéssel a felhasználó. Meglévő előfizetés esetén a kifizetett csomag adatai jelennek meg, valamint az előfizetés lejárat ideje.



Ábra 21: Az előfizetés adatai.

Ha a felhasználó nem rendelkezik előfizetéssel, akkor egy listán keresztül tudja kiválasztani a megvásárlandó csomagot. Ezeket a lehetőségeket végponthívás segítségével kérdezi le a kliens a szervertől. A csomagokat az adatbázisban tárolom el névvel, árral és a maximum kiemelhető hirdetések számával együtt. Ha új előfizetéstípust szeretnénk felvenni, akkor azt könnyedén meg tudjuk tenni egy új sor felvételével az adatbázisban.

The screenshot shows a modal window titled "Buy a new subscription!" with a close button (X) in the top right corner. Below the title is a progress bar with three steps: 1. Tiers (Select your tier), 2. Payment (Enter payment method), and 3. Overview (Check your details). Step 1 is currently active, indicated by a green circle with a checkmark. Below the progress bar is a dropdown menu with "Base" selected. At the bottom right is a green "Next" button.

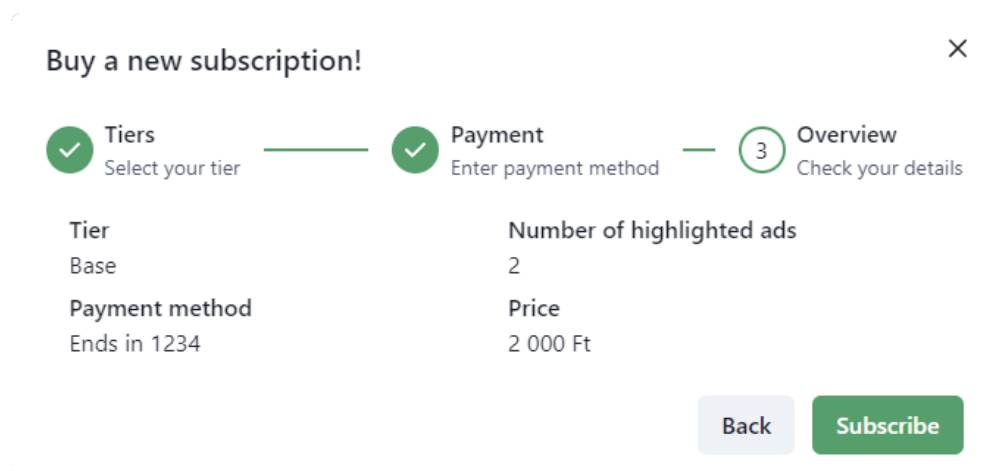
Ábra 22: Csomag kiválasztása.

A következő lépésben a felhasználónak meg kell adnia a bankkártya adatait. Ezen az űrlapon az előző komponensekhez hasonlóan a yup könyvtár segítségével validáltam az adatokat. A bankkártya adatai nem kerülnek ténylegesen feldolgozásra, mivel ehhez egy külső szolgáltatásra lenne szükség.

The screenshot shows the same modal window, now at step 2: Payment (Enter payment method). Step 1 is now marked as completed with a green checkmark. The form contains three input fields: a card number field with "5100 1234 5678 1234", a masked cardholder name field with a key icon and three dots, and an expiration date field with "01/25". At the bottom right are "Back" and "Next" buttons.

Ábra 23: Bankkártya adatok kitöltése.

A következő ablakon egy áttekintés jelenik meg, amelyen leellenőrizheti a megadott adatokat a felhasználó. A „Subscribe” gomb megnyomása után egy végponthívás történik, amivel létrejön az előfizetés. Ezután a felhasználó szabadon készíthet kiemelt hirdetéseket.



Ábra 24: Az előfizetés adatainak áttekintése

A felhasználók a vásárlástól számított egy hónap után elveszítik előfizetésüket. Az előfizetések lejáratának ellenőrzéséhez a Quartz.NET könyvtárat használtam, amelynek segítségével periodikusan el tudom végezni ezt a feladatot. A Quartz.NET konfigurációját a Program.cs fájlban adtam meg, melynek kódja lent látható:

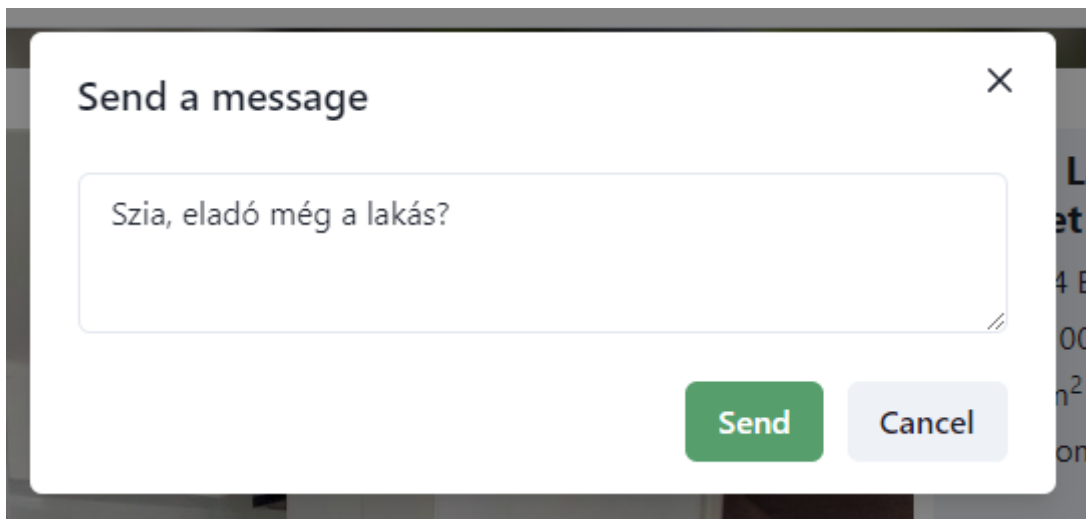
```
builder.Services.AddQuartz(q =>
{
    JobKey jobKey = new JobKey(nameof(CheckSubscriptionsJob));
    q.AddJob<CheckSubscriptionsJob>(opts => opts.WithIdentity(jobKey));

    q.AddTrigger(opts => opts
        .ForJob(jobKey)
        // Run job on startup
        .StartNow());
    q.AddTrigger(opts => opts
        .ForJob(jobKey)
        .WithIdentity($"{nameof(CheckSubscriptionsJob)}-trigger")
        // Run job every day
        .WithCronSchedule("0 0 0 ? * *"))
    );
});
builder.Services.AddQuartzHostedService(q => q.WaitForJobsToComplete =
true);
```

A feladat minden nap 00:00-kor indul el, illetve a szerver minden egyes elindításakor. Előfordulhat az, hogy az előfizetés akkor jár le, amikor a szerver nem fut, ezért szükséges az utóbbi ütemezés. A feladatot a CheckSubscriptionJob osztály Execute függvényében írom le, amely implementálja az IJob interfészt.

## 5.6 Üzenetküldés

Ha egy felhasználó érdeklődni szeretne egy ingatlan iránt, az alkalmazáson keresztül üzenetet küldhet a hirdetőnek. Az üzenet kezdeményezését a hirdetés részletes nézetén keresztül tudjuk elindítani a „Message advertiser” gombra kattintva. Egy felugró ablak jelenik meg, melyben a felhasználó megírhatja üzenetét, és a „Send” gombot megnyomva elküldésre kerül. Ezután a kliens átirányítja arra a nézetre, ahol az összes párbeszédet láthatja.



Ábra 25: Üzenet kezdeményezése.

Az üzenet küldését a SignalR modul használatával valósítottam meg. A WebApi rétegben létrehoztam egy MessageHub osztályt, mely az üzenetváltással kapcsolatos műveleteket kezeli. A Controller-ekhez hasonlóan függvényeket definiáltam az osztályban, amelyeket a kliens meg tud hívni. A Hub eseményeket is tud küldeni, melyekre a frontend fel tud iratkozni. Az üzenetküldés megvalósítását lent láthatjuk:

```
public MessageDTO Send(string recipient, string message)
{
    string userName = GetUserName();
    MessageDTO messageDto = _messageService.CreateMessage(
        recipient, userName, message
    );

    if (ConnectedUsers.ContainsKey(recipient))
    {
        Clients.Clients(ConnectedUsers[recipient]).SendAsync(
            NEW_MESSAGE, messageDto
        );
    }
}
```

```

        int numberOfNewMessages =
            _messageService.GetNewMessageCount(recipient);
        Clients.Clients(ConnectedUsers[recipient]).SendAsync(
            NEW_MESSAGE_COUNT_CHANGED, numberOfNewMessages
        );
    }

    return messageDto;
}

```

Az üzenet létrehozása után a szerver elküldi az eseményt a címzettnek, továbbá értesíti őt az olvasatlan üzenetek számának megváltozásáról. Mivel egy felhasználó több helyen is be lehet jelentkezve, a Hub-ban el kell tárolnom a felhasználóhoz tartozó kapcsolatazonosítókat, hogy minden kliensnek el tudjam küldeni az eseményt. Ezt egy Dictionary segítségével teszem meg, melyben a felhasználóhoz tartozó összes kapcsolatot eltárolom. Az azonosítókat az OnConnectedAsync függvény kibővítésével tárolom el, a kliens lecsatlakozása esetén pedig törlöm.

A SignalR hub-ok használatához meg kell adni a megfelelő konfigurációt a Program.cs fájlban. Ebben a fájlban fel kell tüntetni az összes hub-ot, amelyeket használni szeretnénk, illetve az elérési útját, melyen keresztül el tudjuk érni.

```

builder.Services.AddSignalR();
/*...*/
app.MapHub<MessageHub>("chathub");

```

Kliensoldalon a SignalR kapcsolat létrehozásához készítettem egy MessageHubContext-et és egy MessageHubProvider komponenst, hogy mindenhol elérhető legyen. A MessageHubProvider-t az AuthProvider komponenshez hasonlóan a DOM fa gyökerébe helyeztem.

```

export const MessageHubContext = createContext<MessageHub | undefined>(
    undefined,
);

const conn = new HubConnectionBuilder()
    .withAutomaticReconnect()
    .withUrl('https://localhost:7202/chathub', {
        accessTokenFactory: () => {
            return getCookie('_auth') || '';
        },
    })
    .build();

```

```

export const MessageHubProvider = ({
  children,
}: ChildrenType): ReactElement => {
  const connection = useRef(conn);
  const isAuthenticated = useIsAuthenticated();

  /* ... */

  return (
    <MessageHubContext.Provider value={{ invoke: invoke, on: on, off: off
  }}>
      {children}
    </MessageHubContext.Provider>
  );
};

```

Az üzenetküldéssel kapcsolatos műveleteknek létrehoztam egy useMessageHub hook-ot, melyen keresztül lehet küldeni üzenetet, meg lehet jelölni üzeneteket olvasottként, és fel lehet iratkozni az új üzenet eseményre. A hook felhasználja a MessageHubContext-et, melyen keresztül hozzáférhet a kapcsolathoz.

Az üzeneteket a menübárban lévő „Messages” gombra kattintva érhetjük el. A gombon látható az olvasatlan üzenetek száma is. A nézeten láthatjuk, hogy a felhasználók nevét, az új üzenetek számát, és az utolsó válasz óta eltelt időt.

## Messages

CONTACT	NEW	LAST MESSAGE
Olgaard Zsáki Izabella ozi	5	a few seconds ago
Kovács Sándor kovacssandor	0	12 days ago

< 1 >

Ábra 26: A felhasználó párbeszédei.

Egy felhasználóra kattintva megtekinthetjük az egymásnak elküldött üzeneteket. Az üzenetek listája valós időben frissül. Új üzenet küldése esetén a címzett kliens a



szervertől egy eseményt kap, melynek hatására megjelenik az új szöveg egy szövegbuborékban és frissül az új üzenetek száma a menübarban. A felhasználó használhat Markdown szintaxist is mellyel szabadon megformázhatja üzenetét. A műveletekhez és az események kezeléséhez a useMessageHub függvényeit használtam.

```
export default function useMessageHub() {
  const connection = useContext(MessageHubContext);
  if (!connection) {
    throw new Error('MessageHubProvider is missing.');
```

```
  }

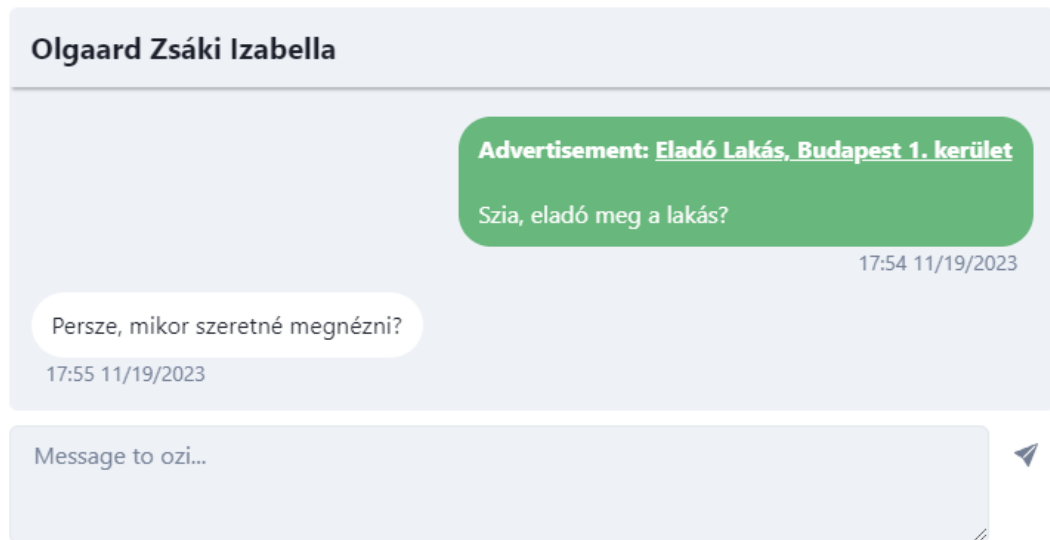
  const sendMessage = async (message: string, userName: string) => {
    if (!connection) return;
    return connection.invoke('send', userName, message);
  };

  const addOnNewMessageHandler = (fn: (message: Message) => void) => {
    if (!connection) return;
    connection.off('NewMessage');
    connection.on('NewMessage', fn);
  };

  const removeOnNewMessageHandler = (fn: (message: Message) => void) => {
    if (!connection) return;
    connection.off('NewMessage');
  };

  const markMessagesAsRead = async (userName: string) => {
    if (!connection) return;
    return await connection.invoke('markMessagesWithUserAsRead',
    userName);
  };

  return {
    sendMessage: sendMessage,
    addOnNewMessageHandler: addOnNewMessageHandler,
    removeOnNewMessageHandler: removeOnNewMessageHandler,
    markMessagesAsRead: markMessagesAsRead,
  };
}
```



**Ábra 27: Üzenetek megjelenítése.**

Az üzenetek olvasottsága az ablak állapota alapján változik. Ha az ablak fókuszba kerül, akkor a nézet meghívja a `useMessageHub` hook `markMessagesAsRead` függvényét és a szerveroldalon lefut a `MarkMessagesWithUserAsRead` metódus.

## 6. Összefoglalás

A szakdolgozatom elkészítése során rengeteg olyan technológiát ismerhettem meg, melyekkel nem találkoztam egyetemi alapképzésem során. Az alkalmazás elkészítése előtt nem volt sok tapasztalatom böngésző alapú kliensek készítésével kapcsolatban, de a dolgozat lehetőséget adott arra, hogy jobban megismerjem a webes technológiákat. Úgy gondolom, hogy sok tudásra tettem szert a projekt elkészítése során, melyet a piacon is fel tudok használni, illetve könnyebben átlátom a bonyolultabb architektúrájú alkalmazásokat is. Fejlesztés során nem tapasztaltam különösebb nehézségeket, amit a megvalósítás előtt elvégzett alapos tervezésnek köszönhetem.

A megvalósítás során a React keretrendszerrel ismerkedtem meg, melyet az Angular könyvtárral lehetne összehasonlítani. Az utóbbi technológiát szakmai gyakorlatom, illetve alapképzésem során volt lehetőségem használni. Véleményem szerint az Angular egy teljesebb megoldást nyújt, míg a React erősen támaszkodik külső könyvtárak használatára. Angular-ben könnyebben tudtam jól strukturált kódot készíteni, emiatt én szívesebben használnám jövőbeli projektjeimben.

A kiszolgáló szerver fejlesztésekor célom volt egy könnyen bővíthető architektúrát létrehozni a Domain Driven Design elv szerint. Ennek előnyeit meg is tapasztalhattam a valós idejű üzenetváltás implementálásakor, melyben a meglévő szolgáltatás függvényeit újra tudtam használni. Az alkalmazás elkészítése során továbbfejleszthettem a .NET keretrendszerrel kapcsolatos tudásomat.

### 6.1 Továbbfejlesztési lehetőségek

Az alkalmazást több olyan funkciókkal bővíthető ki, melyek javítják a felhasználói élményt. A hirdetések részletes nézete kibővíthető egy térképpel, mely megmutatja, hogy hol helyezkedik el az ingatlan. Ennek megvalósításához az OpenStreetMap API-t használnám, mely egy ingyenes szolgáltatás a népszerűbb Google Térképpel ellentétben. Jelenleg a hirdetés leírása egyszerű szöveggként jelenik meg, de egy formázási lehetőség implementálásával (például Markdown) a hirdető komplexebb ismertetőt tudnának írni.

Az előfizetések vásárlásához, a bankkártyás tranzakciók kezeléséhez egy külső fizetési szolgáltatásra van szükség. Több választási lehetőség is van, ilyen a PayPal,

amely egy külföldi megoldás, vagy a Magyarországon fejlesztett SimplePay, illetve Barion.

Mivel bárki készíthet hirdetést, emiatt előfordulhatnak rosszindulatú felhasználók, akik át akarják verni az érdeklődőket. Ezek kiszűréséhez egy adminisztrátori felületet hoznék létre, melyhez magasabb jogú szerepkörökkel rendelkező fiókok segítségével lehetne hozzáférni, és képes lenne hirdetések menedzselésére, illetve felhasználók kitiltására. A moderátorok munkáját egy jelentés funkcióval is meg tudjuk könnyíteni.

Az előbb felsorolt funkciókon kívül a további fejlesztésekkel lehet még bővíteni az alkalmazást:

- Több kép feltöltése egy képhez
- Hirdetések ajánlása, személyre szabott javaslatok
- Felhasználó értesítése email-ben új üzenet esetén

## 7. Irodalomjegyzék

- [1] Existek, „Top Front-End Frameworks in 2023,” [Online]. Available: <https://existek.com/blog/top-front-end-frameworks-2021/>. [Hozzáférés dátuma: 27 10 2023].
- [2] Microsoft, „What is .NET? Introduction and overview,” Microsoft, 15. 03. 2023.. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/core/introduction>. [Hozzáférés dátuma: 13. 05. 2023.].
- [3] Microsoft, „Common Language Runtime (CLR) overview,” Microsoft, 25. 04. 2023.. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/standard/clr>. [Hozzáférés dátuma: 13. 05. 2023.].
- [4] NDepend, „.NET Decompilers Compared: A Comprehensive Guide (2023),” [Online]. Available: <https://blog.ndepend.com/in-the-jungle-of-net-decompilers/>. [Hozzáférés dátuma: 13 05 2023].
- [5] Microsoft, „Overview of ASP.NET Core,” Microsoft, 15. 11. 2022.. [Online]. Available: <https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-7.0>. [Hozzáférés dátuma: 13. 05. 2023.].
- [6] Wade, „.NET Core Dependency Injection Lifetimes Explained,” .NET Core Tutorials, [Online]. Available: <https://dotnetcoretutorials.com/net-core-dependency-injection-lifetimes-explained/>. [Hozzáférés dátuma: 13. 05. 2023.].
- [7] R. Anderson, „Introduction to Identity on ASP.NET Core,” Microsoft, 01. 12. 2022. [Online]. Available: <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-7.0&tabs=visual-studio>. [Hozzáférés dátuma: 14. 05. 2023.].
- [8] Microsoft, „Entity Framework Core,” Microsoft, 25. 05. 2021.. [Online]. Available: <https://learn.microsoft.com/en-us/ef/core/>. [Hozzáférés dátuma: 14. 05. 2023.].

- [9] N. I. Solutions, „Better Entity Framework Core in .net,” [Online]. Available: <https://n9-it.com/blog/Better-Entity-Framework-Core-in-.net>. [Hozzáférés dátuma: 14 05 2023].
- [10] P. Fletcher, „Introduction to SignalR,” Microsoft, 10 09 2020. [Online]. Available: <https://learn.microsoft.com/en-us/aspnet/signalr/overview/getting-started/introduction-to-signalr>. [Hozzáférés dátuma: 23 10 2023].
- [11] Internetexception, „WebSocket libraries comparison,” [Online]. Available: <https://internetexception.com/2014/02/21/websocket-libraries-comparison/>. [Hozzáférés dátuma: 23 10 2023].
- [12] M. Lahma, „Quartz.NET,” [Online]. Available: <https://www.quartz-scheduler.net/>. ]
- [13] Microsoft, „TypeScript,” Microsoft, [Online]. Available: <https://www.typescriptlang.org/>. [Hozzáférés dátuma: 15. 05. 2023.].
- [14] Y. çidem, „Typescript overview,” 13 02 2023. [Online]. Available: <https://medium.com/front-end-weekly/typescript-overview-e5f66a836847>. [Hozzáférés dátuma: 15 05 2023].
- [15] Meta, „React,” Meta, [Online]. Available: <https://react.dev/>. [Hozzáférés dátuma: 15. 05. 2023.]. ]
- [16] „Domain-driven design,” [Online]. Available: [https://en.wikipedia.org/wiki/Domain-driven\\_design](https://en.wikipedia.org/wiki/Domain-driven_design). [Hozzáférés dátuma: 28 10 2023].
- [17] knóldus, „3D’s of Architecture & Development: Domain Driven Design,” 31 12 2023. [Online]. Available: <https://blog.knoldus.com/3ds-of-architecture-development-domain-driven-design/>. [Hozzáférés dátuma: 28 10 2023].
- [18] J. Monceda, „NET6 Multitenant with MediatR Design Pattern Starter Kit,” 16 04 2023. [Online]. Available: <https://joever-monceda.medium.com/net6-multitenant-with-mediatr-design-pattern-starter-kit-33b87fc58c7e>. [Hozzáférés dátuma: 28 10 2023].

- [19 „Phone validation regex,” 22 08 2013. [Online]. Available:  
] <https://stackoverflow.com/questions/18375929/validate-phone-number-using-javascript>. [Hozzáférés dátuma: 10 02 2023].
- [20 dotnet, „PasswordHasher.cs source code,” [Online]. Available:  
] <https://github.com/dotnet/aspnetcore/blob/main/src/Identity/Extensions.Core/src/PasswordHasher.cs>. [Hozzáférés dátuma: 11 11 2023].
- [21 auth0, „Introduction to JSON Web Tokens,” auth0, [Online]. Available:  
] <https://jwt.io/introduction>. [Hozzáférés dátuma: 12 11 2023].