

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Papp Kristóf

hirdetési portál megvalósítása .net Core és React alapokon

Konzulens

Benedek Zoltán

BUDAPEST, 2019

Tartalomjegyzék

[Összefoglaló 5](#_Toc21513196)

[Abstract 6](#_Toc21513197)

[1 Bevezetés 7](#_Toc21513198)

[1.1 Témaválasztás 7](#_Toc21513199)

[1.2 Technológiai háttér 7](#_Toc21513200)

[1.3 Hasonló megvalósítások 8](#_Toc21513201)

[1.4 A szakdolgozat felépítése 8](#_Toc21513202)

[2 Felhasznált technológiák 9](#_Toc21513203)

[2.1 .NET Core 9](#_Toc21513204)

[2.2 C# nyelvi elemek és aszinkronitás 10](#_Toc21513205)

[2.3 ASP.NET Core 11](#_Toc21513206)

[2.4 ASP.NET Core Identity 12](#_Toc21513207)

[2.5 Entity Framework Core és LINQ 12](#_Toc21513208)

[2.6 TypeScript 13](#_Toc21513209)

[2.7 React 15](#_Toc21513210)

[2.8 React router 18](#_Toc21513211)

[2.9 Redux 19](#_Toc21513212)

[2.10 Axios 20](#_Toc21513213)

[2.11 Bootstrap 4 21](#_Toc21513214)

[2.12 Architekturális és tervezési minták 21](#_Toc21513215)

[2.12.1 Domain Driven Design 21](#_Toc21513216)

[2.12.2 Repository 23](#_Toc21513217)

[3 Követelmények 25](#_Toc21513218)

[3.1 Use Case diagramm 25](#_Toc21513219)

[3.2 Böngészési funkciók 26](#_Toc21513220)

[3.3 Hirdetői funkciók 26](#_Toc21513221)

[3.4 Adminisztrátori funkciók 27](#_Toc21513222)

[4 Architektúrák 28](#_Toc21513223)

[4.1 Rendszer architektúra 28](#_Toc21513224)

[4.2 Az adatbázis felépítése 29](#_Toc21513225)

[4.3 Backend architektúra 32](#_Toc21513226)

[4.4 Frontend architektúra 36](#_Toc21513227)

[5 Megvalósítás 38](#_Toc21513228)

[5.1 Bejelentkezés 38](#_Toc21513229)

[5.2 Hirdetés feladása 45](#_Toc21513230)

[5.3 Előfizetés módosítása 50](#_Toc21513231)

[5.4 Egyéb funkciók 53](#_Toc21513232)

[5.4.1 Regisztráció 53](#_Toc21513233)

[5.4.2 Hirdetések listázása és keresése 53](#_Toc21513234)

[5.4.3 Hirdetések módosítása 54](#_Toc21513235)

[5.4.4 Felhasználói beállítások 55](#_Toc21513236)

[5.4.5 Adminisztrátori funkciók 55](#_Toc21513237)

[5.5 Biztonság 55](#_Toc21513238)

[5.6 Tesztelés 57](#_Toc21513239)

[6 Összefoglalás 59](#_Toc21513240)

[6.1 Tapasztalatok 59](#_Toc21513241)

[6.2 Továbbfejlesztési lehetőségek 59](#_Toc21513242)

[7 Irodalomjegyzék 61](#_Toc21513243)

[Függelék 64](#_Toc21513244)

Hallgatói nyilatkozat

Alulírott **Papp Kristóf** 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 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, 2019. 05. 15

...…………………………………………….

Papp Kristóf

Összefoglaló

Napjainkban egyre többen intézik a vásárlásaikat, banki ügyeiket vagy utazásuk szervezését az interneten, ezért minden korábbinál elterjedtebbé válnak a webes vékonykliens alkalmazások. A hagyományos asztali alkalmazásokkal ellentétben a webes alkalmazásokat nem szükséges telepíteni, használatuk egyszerűbb, gyorsabb és kényelmesebb. A Single-page alkalmazások a böngészőben futnak, a logika egy részét kliens oldalon valósítják meg, így használatuk során nincs szükség az oldal újra töltésére. Ezáltal lehetőség van kliens oldali cachelésre is, így gyorsabbak és jobb felhasználói élményt nyújtanak, mint a szerver oldalon renderelt weblapok.

Szakdolgozatom során egy többrétegű, online hirdetési keretrendszert készítettem, melyben a felhasználók (előfizetési modelljüktől függően) hirdetéseket jeleníthetnek meg vagy böngészhetnek más hirdetők ajánlatai között. A feladat megvalósításánál fontos volt a nagyfokú konfigurálhatóság, a könnyű bővíthetőség és a felhasználóbarát megjelenés. Az alkalmazásban lehetőség van a hirdetések létrehozására különböző szintű kiemelésére, képek feltöltésére és az ajánlatok közti részletes keresésre is. A felhasználók különböző szerepkörökbe tartozhatnak, és a megrendelő által szabadon konfigurálható előfizetési modellekre iratkozhatnak fel.

A projekt során több olyan technológiával is megismerkedtem, melyek a BSc képzés során csak később, vagy egyáltalán nem fordulnak elő. Ezek közé tartozik a webes felület építéséhez használt React keretrendszer, a központosított állapotkezelésért felelős Redux és a TypeScript nyelv is. A feladatom részét képezte ezeknek a technológiáknak a megismerése, majd pedig a szakdolgozatomban való felhasználása is.

Abstract

Nowadays, more and more people are shopping, banking or organizing their travel online, and thin client applications are becoming more widespread than ever before. Unlike traditional desktop applications, web applications do not need to be installed, they are easier, faster and more convenient to use. Single-page applications run in the browser, implementing some of the logic on a client page, so they do not need to reload the page. This also enables client-side caching, which is faster and provides a better user experience than web pages rendered on the server side.

In my thesis, I created a multi-layered online advertising framework where users (depending on their subscription model) can display ads or browse offers from other advertisers. Highly configurable, easy-to-expand and user-friendly design was important to accomplish this task. The app also has the ability to create ads at different levels of highlighting, image uploads, and advanced search for offers. Users have different roles and subscribe to configurable subscription models.

During the project I became acquainted with several technologies that did not occur during the BSC training. These include the React framework used to build the web interface, Redux for centralized state management, and the TypeScript language. Part of my job was to get familiar with these technologies and to use them in my thesis.

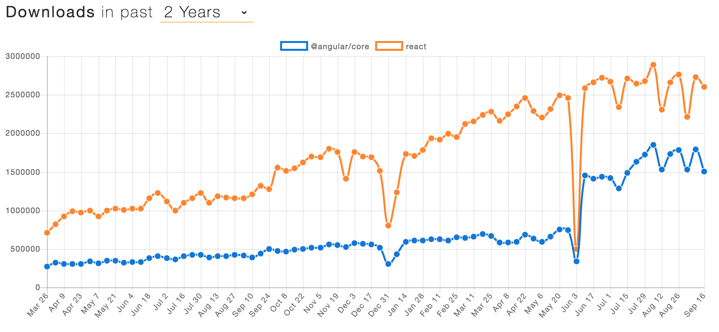
# Bevezetés

## Témaválasztás

Már több ismerősömben felmerült az igény, hogy használt autóját, megunt bútorait vagy kiadó lakását az interneten hirdesse meg. Azonban a magas díjak és a bonyolult regisztrációs folyamat miatt legtöbben elvetették ezt az ötletet. Mivel világviszonylatban is nagymértékben nő a rendszeres internetfelhasználók és az internetes hirdetések száma, ezért szakdolgozatom témájául egy webes hirdetési portál fejlesztését választottam.

## Technológiai háttér

Az elmúlt években egyre elterjedtebbé vált a webes alkalmazások használata, melyet a felhasználók egyaránt elérnek asztali gépükről és okostelefonjukról is. A webes kliensoldali technológiák az informatika egyik legdinamikusabban fejlődő területének számítanak. Majdnem minden évben jelenik meg újabb verzió valamelyik UI keretrendszerből. Mivel a BSC képzés során már megismertem az Angular használatát, szerettem volna egy másik webes frontend keretrendszert is megismerni. Mivel a React az Angular-nál modernebb és jelenleg több aktív fejlesztővel is rendelkezik, így arra esett a választásom.



. ábra React és Angular letöltések száma [1]

## Hasonló megvalósítások

A hazai piacon több internetes hirdetési portál is megtalálható. Ezek között előfordulnak általános jellegűek, mint a Vatera vagy egy termékcsoportra specializáltak, mint az Ingatlan.com vagy a Használtautó.hu. Utóbbiak esetén az áttérés egy másik hirdetendő termékcsoportra igen költséges lehet, elképzelhető, hogy a teljes alkalmazás újraírását igényelné. Mivel nem találtam olyan terméket, mely más-más konfigurációkkal más-más termékek hirdetésére lenne alkalmas, így eldöntöttem, hogy egy ilyen hirdetési portál fejlesztésébe fogok bele.

## A szakdolgozat felépítése

A dolgozat 2. fejezetében bemutatom a megismert és az alkalmazás implementációja során felhasznált technológiákat és architekturális mintákat.

A 3. fejezetben ismertetem az alkalmazással szemben támasztott részletes követelményeket és a felhasználói szerepköröket.

A 4. fejezetben leírom az alkalmazás architekturális felépítését, bemutatom az egyes szinteken felhasznált tervezési irányelveket és mintákat.

A 5. fejezetben bemutatom az elkészült alkalmazás funkcióit, valamint ismertetem a frontend és a backend megvalósítását. Leírom a tesztelés folyamatát és bemutatom az alkalmazás biztonsága érdekében tett implementációs lépéseket.

A 6. fejezetben összefoglalom az elvégzett munkával kapcsolatos tapasztalataimat és kitérek az alkalmazás továbbfejlesztési lehetőségeire is.

A 7. fejezetben összegyűjtöttem a munkám során felhasznált források elérhetőségeit.

# Felhasznált technológiák

## .NET Core

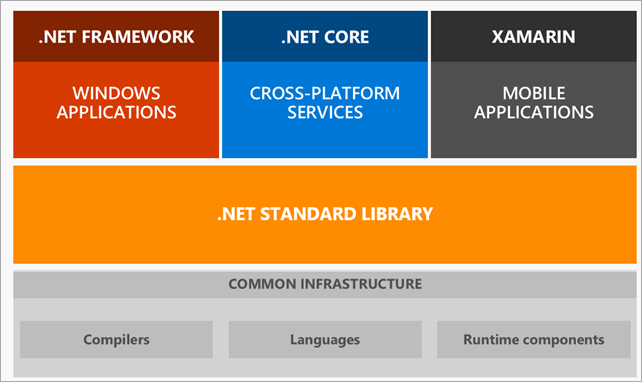
A Microsoft által 2016-ban bemutatott .NET Core [1] egy ingyenesen hozzáférhető és nyílt forráskódú, általános célú szoftverfejlesztési keretrendszer. A teljes .NET Frameworkhöz képest jelentősen átdolgozott .NET Core multiplatform szoftverfejlesztést tesz lehetővé, egyaránt léteznek implementáció Windows, Linux és macOS operációs rendszerekre. A .NET nyelvfüggetlen, a C#, VB és F# mellett több mint 30 különböző nyelvet támogat. A különböző nyelvekről különböző platformokra történő fordítás egyszerűsítéséhez a .NET Framework és a .NET Core esetében is két lépésben történik a kód fordítása. A fordítás során először egy köztes IL (Common Intermediate Language) kód keletkezik, majd a program végrehajtásakor Just-in-time módon áll elő az IL kódból a gépi kód.

Képtalálatok a következőre: .net compile

. ábra Fordítási folyamat .NET környezetben [2]

A .NET Frameworkkel ellentétben a .NET Core moduláris felépítésű, így csak az alapvető osztályokat tartalmazza, további komponensek Nuget csomagok formájában tölthetőek le. A .NET Core a legdinamikusabban fejlődő .NET platform, legújabb, 3.0-s verziója, amely már Windows Forms és WPF alkalmazásokat is támogat, 2019-ben jelent meg. A moduláris felépítés több előnnyel is jár. Lerövidül a fordítási idő, a kész alkalmazások pedig gyorsabban indulhatnak és kevesebb tárhelyet igényelnek.

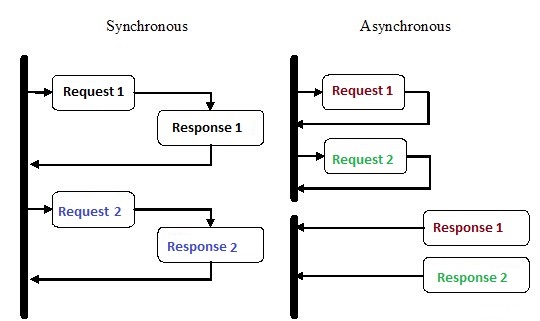
A .NET Framework, .NET Core és a Xamarin a .NET Standard szabvány különböző megvalósításai. A három platform közti könnyű átjárhatóságot a közös osztálykönyvtár (.NET Standard Library), a közös nyelvek támogatása (a Common Language Infrastructure) és a közös fordítók biztosítják.



. ábra A különböző .NET platformok [3]

## C# nyelvi elemek és aszinkronitás

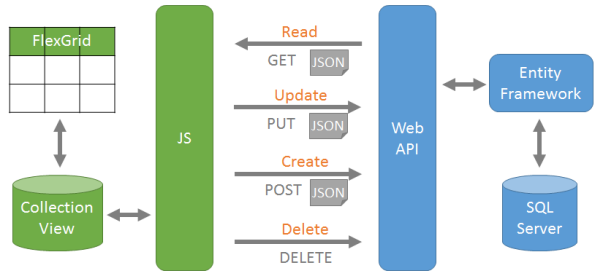
A C# egy erősen és statikusan típusos objektumorientált programozási nyelv. Az erős típusosság jelentősen csökkenti a futási idejű hibák előfordulását, hiszen a típusinformációkból a compiler már fordítási időben is ki tud szűrni bizonyos hibákat. A C# felügyelt nyelv, így nincs szükség manuális memóriakezelésre, az objektumokat a szemétgyűjtő algoritmus (Garbage collector) szabadítja fel. A C# nyelv 5.0-s verziójával megjelent async és await kulcsszavak lehetővé teszik az aszinkron programozási modell használatát. Ezáltal szerverek esetében hatékonyabban szolgálhatók ki a beérkező kérések, kliens oldali szoftverek esetén pedig biztosítható a UI reszponzívitása.



. ábra Szinkron és aszinkron műveletek összehasonlítása [4]

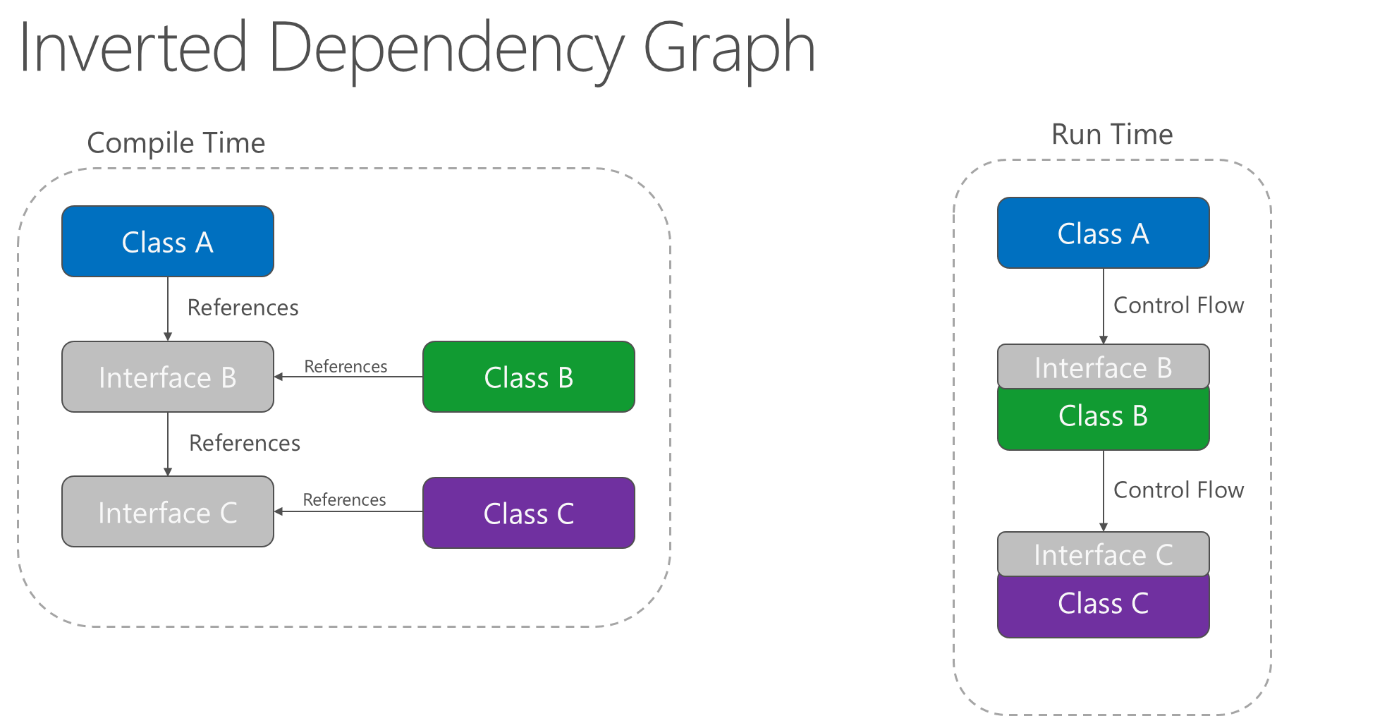
## ASP.NET Core

Az ASP.NET Core [5] egy modern, főleg szerver oldali, internetes alkalmazások fejlesztését támogató keretrendszer. Egyaránt támogatja a szerver oldalon renderelt weboldalak megjelenítését (Razor Pages), Single-page webalkalmazások (Blazor, Angular, React) készítését vagy bármely kliens kiszolgálását REST (Representational State Transfer) alapú WebAPI-n keresztül. Az ASP.NET egyaránt rendelkezik .NET Framework és multiplatform .NET Core implementációval.



. ábra Alapvető REST kérések [6]

Az ASP.NET Core egyik fontos szolgáltatása a Függőség injektálás [7] (Dependency Injection). A DI konténer segítségével osztályaink függőségeit nem kell a konstruktorukban bedrótozni, hanem egy központi helyen adhatóak meg. A konkrét osztályok helyett a kódunkban interfészekre hivatkozunk, a konkrét implementáció a legtöbb komponens számára rejtve marad, ezáltal nem is függenek tőle. A Függőség injektálással lazábban csatolt és könnyebben tesztelhető alkalmazásokat kapunk. Tipikusan függőség injektálást használnak adatbázis kontextusok példányosítására. ASP.NET esetében a függőségek feloldását a Startup osztály ConfigureServices függvényében adhatjuk meg. Az injektált objektumok életciklusukat tekintve három félék lehetnek. A Transient szolgáltatásokból minden kéréskor új példány keletkezik, a Scoped szolgáltatások kliensenként vagy kapcsolatonként azonos példányt biztosítanak, Singleton esetben pedig a teljes alkalmazás futása alatt egy példánnyal dolgozhatunk.



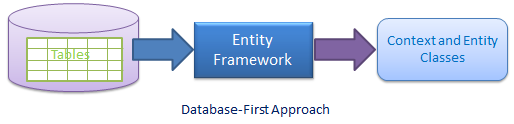
. ábra Függőségek csökkentése DI segítségével [8]

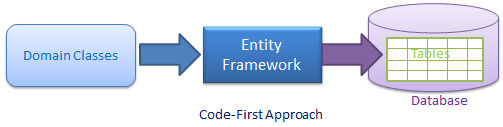
## ASP.NET Core Identity

Az ASP.NET Core egyik kiterjesztéseként az Identity felhasználók menedzselését, bejelentkezések nyilvántartását és sessionok kezelését segíti. Beépítve tartalmaz szerepköröket, felhasználói hitelesítést és konfigurálható jelszó erősséget. Az ASP.NET felhasználói adatbázisa a beépített elemek mellett további, a felhasználókról tárolni kívánt információkkal is kiegészíthető. Az Identity konfigurálható lokális SQL adatbázis használatához vagy online tárolóhoz, mint az Azure Table Storage.

## Entity Framework Core és LINQ

Az Entity Framework egy Objektum-relációs keretrendszer (ORM), amely az üzleti objektumok és az adatbázis rekordok közötti leképzés megvalósításáért felel. Használatával függetleníthetjük az alkalmazásunkat a tényleges adatbázis motortól. Az EF szinte minden nagyobb adatbázis meghajtóval kompatibilis, van hozzá MSSQL, MySQL, SQLite és Oracle adatbázis támogatás is. Az adatbázisbeli rekordok és a programbeli objektumok közötti megfeleltetés létrehozására Entity Framework Core esetén két mód áll rendelkezésünkre: Database first esetén meglévő adatbázis sémához generálhatjuk le C# (vagy más .NET nyelvű) domain osztályainkat. Manapság azonban a Code first megoldás a legelterjedtebb, ahol C# programkódban definiáljuk az üzleti logikai entitásainkat, majd az EF ehhez készíti el a relációs adatbázist.





. ábra A modellezés iránya a Database First és a Code First megközelítéseknél [9]

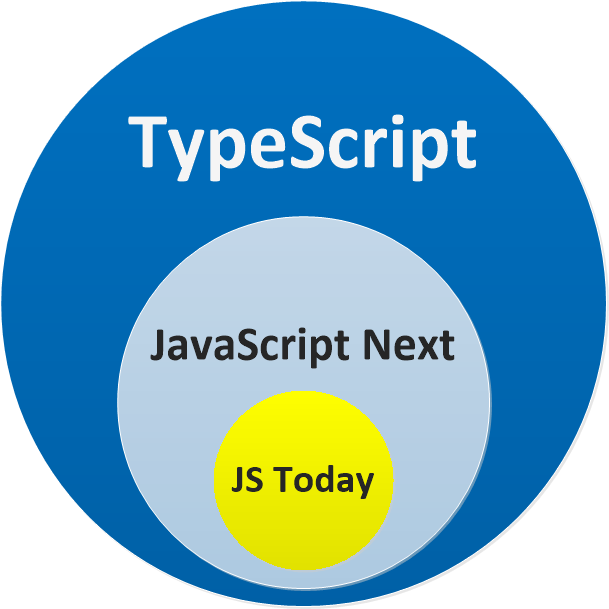
EF Core esetén is lehetőségünk van Lazy loading támogatásra, melynek lényege, hogy a betöltött objektumhoz kapcsolódó entitások csak akkor töltődnek be, ha azokra ténylegesen szükségünk van, és a lekérdezés végrehajtásakor ezt explicit jelezzük. Ez jelentősen meggyorsíthatja a lekérdezések végrehajtását. Az Entity Framework lehetőséget ad Seedelt teszt adatok adatbázisba való feltöltésére, amivel jelentősen meggyorsíthatja a fejlesztési folyamatot.

A LINQ (Language Integrated Query) segítségével deklaratív módon fogalmazhatunk meg adatbázis lekérdezéseket. A hagyományos szöveges lekérdezés megfogalmazáshoz képest a LINQ nagy előnye, hogy a fordító felismeri a kulcsszavakat, így elgépelés esetén fordítási idejű hibákat jelez (a nehezebben detektálható futás idejű hibák helyett) és automatikus kiegészítési javaslatokat (IntelliSense) ad a fejlesztőnek. A LINQ két különböző szintaxist is támogat: az objektumorientált függvényhívásokat és a szöveges, SQL szerű lekérdezés megfogalmazást. Adatbázisok mellett a LINQ bármilyen kollekción (IEnumerable) és XML dokumentumokon is használható lekérdezések megfogalmazására.

## TypeScript

A Microsoft által fejlesztett TypeScript programozási nyelv a JavaScript típusos szuperszetje. Minden JavaScript kód egyben TypeScript is, azonban a TypeScript statikus, fordítási idejű típusinformációkat is tartalmaz. A TypeScript követi az objektumorientált paradigmákat, lehetőség van interfészek deklarációjára, leszármazásra és generikus típusok létrehozására is. A statikus típusosság révén a kódszerkesztő valódi IntelliSense támogatást nyújt és a legtöbb hiba még az alkalmazás futtatása előtt észlelhető. TypeScript használatával jelentősen javul a kód karbantarthatósága és újrahasznosíthatósága.

A TypeScript fordító (TypeScript Compiler, vagy röviden TSC) standard JavaScript kódot állít elő TypeScript fájljainkból. A fordítás nagyon részletesen konfigurálható, megadható, hogy a kimenet mely ECMAScript verzióval legyen kompatibilis, illetve a fordító milyen hibákra figyelmeztessen. A TypeScript használatának egyik nagy előnye, hogy a még JavaScriptbe be nem került, illetve a még kevés böngésző által támogatott funkciókat is problémamentesen használhatjuk, hiszen a fordítás kimenetének kompatibilitása csak egy beállításon múlik.



. ábra TypeScript és JavaScript funkciók [9]

A fordítás eredményeként előállított JavaScript nem tartalmaz típusinformációkat (hiszen nincs benne rá nyelvi lehetőség), így ezek külön .d.ts kiterjesztésű fájlokba kerülnek és más TypeScript projekteknél felhasználhatóak. Ezek a fájlok nem tartalmaznak futtatható kódot, csak típusdeklarációkat. Meglévő JavaScript osztálykönyvtárakat mi magunk is kiegészíthetünk típusinformációkkal, ehhez csak a .d.ts fájlt kell megírnunk hozzá, és a könyvtárat máris úgy használhatjuk, mintha eredetileg is TypeScriptben írták volna.

Az objektum orientált nyelveknél megszokott explicit interfész megvalósítás helyett TypeScriptben strukturális interfészeket definiálhatunk. A strukturális interfész egy típus publikusan elérhető tagváltozóinak, függvényeinek halmaza és osztályok implicit valósíthatják meg. TypeScriptben lehetőség van haladó típusok (advanced types) létrehozására. A metszet típusok (intersection type) két vagy több osztály publikus interfészét is megvalósítják. Így például a **Person & Serializable** típus egyszerre személy és sorosítható is. Az unió típusok (union type) egy adott típus halmazból egy interfészt valósítanak meg. A **number | string** egyaránt lehet szöveg vagy szám, de más típusú értéket nem vehet fel. Az unió típusok switch szerkezetekben típus alapján szétbonthatók, és ekkor az egyes switch ágakon már a szűkebb típussal dolgozhatunk. Ezt használja ki a Redux (alább) is. TypeScriptben lehetőség van **any** típus megadására is, amely a JavaScript dinamikus típusnak felel meg, és futás időben bármilyen típust felvehet.

Ha a fordítón beállítjuk a --strictNullChecks flag értékét, akkor TypeScript típusai alapból nem vehetnek fel null és undefined értékeket. Ezáltal több gyakori programozói hibát is elkerülhetünk: nem fogunk inicializálatlan objektumra hivatkozni és elhagyhatjuk a null ellenőrzéseket. Ha mégis nullable típust szeretnénk létrehozni, akkor alkalmazhatjuk az unió típusokat. A TypeScript változók láthatósága megegyezik a JavaScripttel, a var változók az egész függvényben látszanak, a let változók pedig block scope-ot használnak [10].

TypeScript kódunk összetartozó osztályait és függvényeit modulokba szervezhetjük. Egy modul általában egy fájlnak felel meg, a modulból az **export** kulcsszóval publikálhatjuk típusainkat, melyeket aztán más modulokban az **import** segítségével használhatunk fel.

## React

A React Facebook által tervezett JavaScript könyvtár webes felhasználói felületek építéséhez. Használatával deklaratív módon fogalmazhatunk meg HTML komponenseket, melyeket a React állapotváltozás esetén hatékonyan frissít és renderel. A deklaratív nézetek használata megkönnyíti a fejlesztést és a hibakeresést. A React komponens orientált, így az egyes felületi elemek egységbe zárva tartalmazzák állapotukat és állapotváltási logikájukat, ezáltal lazább csatolás érhető el az egyes komponensek között, ami nagyobb újrahasznosíthatóságot jelent.

A React modern Single-page alkalmazások alapjául szolgál, melyek nagy előnye a szerver oldali rendereléssel szemben, hogy nem kell minden módosításnál a teljes weboldalt újra letölteni, ezáltal kevesebb hálózati és szerver oldali erőforrást használnak. A jó skálázhatóságnak köszönhetően a React alkalmas olyan összetett webalkalmazások fejlesztésére is, mint a Facebook, az Instagram vagy a Netflix.

A komponensek frissítéséről a React gondoskodik, amely a felület másolatát egy virtuális DOM-ban tartja nyilván, és csak azokat a részeket frissíti, amelyek tényleg megváltoztak az oldalon. Minden komponens rendelkezik saját életciklus függvényekkel, melyeket a keretrendszer hív meg például a komponens megjelenítésekor vagy állapotának frissülésekor. Egy adott komponens megjelenését pedig a render() metódusában írhatjuk le. JSX (JavaScript XML) lehetővé teszi, hogy JavaScript kódban HTML-hez hasonló szintaxissal fogalmazzuk meg a felület strukturált leírását, ezáltal a kódunk sokkal átláthatóbbá válik. A JSX kódba sztenderd JavaScript kifejezéseket is ágyazhatunk kapcsos zárójelek között. A JSX kód átláthatóbb felületleírását a következő kódrészletek [11] mutatják.

JSX szintaxis használatával:

<div>

Hello {this.props.name}

</div>

JSX nélkül, standard JavaScript-ben:

React.createElement("div", null, "Hello ", this.props.name);

Az alábbi kódrészlet [11] egy egyszerű számláló komponens felépítését szemlélteti TypeScript nyelven. A React importálása után definiáljuk a Props (ezt kell átadni a komponens példányosításakor) és a State (ez a komponens belső állapota) interfészeket. Ezeken jól látszik a TypeScript statikus típusossága. Az osztály előtti export kulcsszó jelzi, hogy a komponenst más fájlokból, modulokból is fel szeretnénk használni. A Timer komponens a konstruktorában inicializálja a belső állapotát. A komponens megjelenésekor lefut a componentDidMount() életciklus függvény, amely elindítja a számlálást. A tick() függvény a setState() metóduson keresztül módosítja az állapotot, hogy a React értesüljön a módosításról és optimalizálva újra tudja rajzolni a komponenst. A vizuális megjelenést a render() függvényben adjuk meg JSX tagek formájában.

import React from "react";

interface IProps {

interval: number;

}

interface IState {

ticks: number;

interval: number

}

export class Timer extends React.Component<IProps, IState> {

constructor(props: Readonly<IProps>) {

super(props);

this.state = {

ticks: 0,

interval: props.interval

};

}

tick() {

this.setState(state => ({

ticks: state.ticks + 1

}));

}

componentDidMount() {

setInterval(() => this.tick(), this.state.interval);

}

render() {

return (

<div>

<h1>

Seconds: {this.state.ticks}

</h1>

</div>);

}

}

A kódrészlet a fenti komponens felhasználására mutat példát. A Timer-t használat előtt importálni kell, a példányosításakor pedig (a fenti IProps-ban definiált) interval értéket kell megadnunk. A ReactDOM.render() megkeresi a „root” elemet az előre elkészített HTML fájlban és lecseréli az alkalmazásunkra, am jelen esetben egy számlálót tartalmaz.

import { Timer } from "./Clock";

const App: React.SFC = props => {

return (

<Timer interval={1000} />

);

};

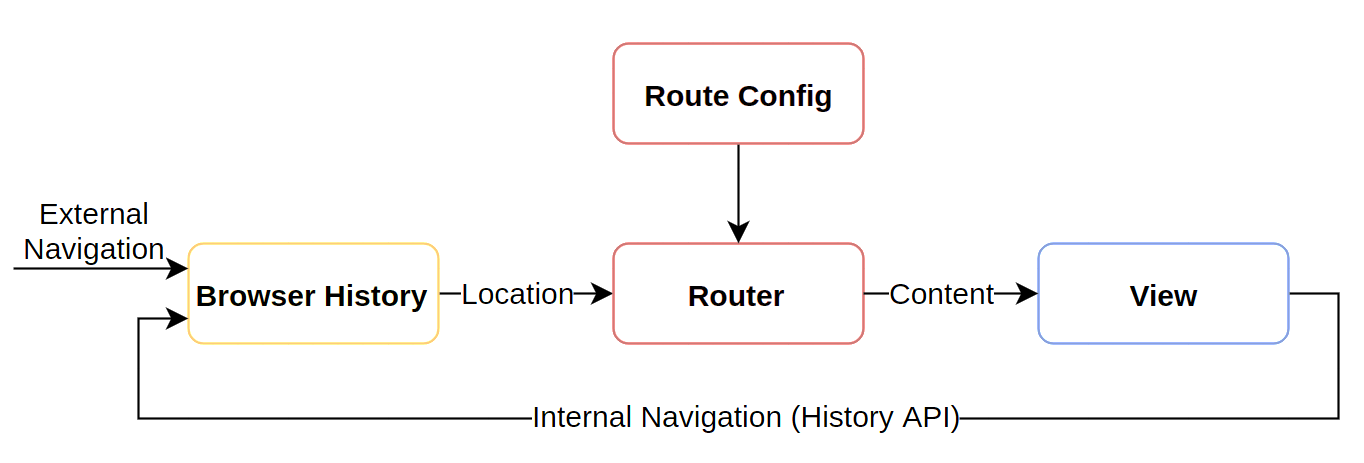
ReactDOM.render(<App />, document.getElementById("root"));

A React komponensek és a JSX kód JavaScriptté történő fordítását a Babel végzi el. A komponensekből függvények vagy osztályok keletkeznek (az ECMAScript verziótól függően). Fontos, hogy a React komponensek immutable objektumok, azaz nem módosíthatóak. Az egyetlen mód egy komponens megváltoztatására egy új példány renderelése.

A React önmagában csak a felület leírását biztosítja, komplexebb alkalmazások fejlesztése esetén a további funkciókhoz más osztálykönyvtárak használata is célszerű. Ilyen például a navigációt biztosító React router vagy az állapotkezelésért felelős Redux.

## React router

Mivel Single-page weboldalak egyetlen HTML fájlból állnak, ezért a hiperhivatkozások helyett más megoldás szükséges az oldalon belüli navigáció megvalósításához. A React router URL részeként tárolja, hogy az alkalmazás mely részét kell éppen megjeleníteni. A React router képes átirányításokat kezelni és az előzményeknek hála a visszafelé navigációra is lehetőséget nyújt.



. ábra A React router képes az oldalon belüli előzmények kezelésére [12]

A React router gyakorlati működését az alábbi kódrészlet szemlélteti. A komponens nem rendelkezik belső állapottal, így egy egyszerű függvény valósítja meg. A navigációhoz relatív útvonalú Link-eket használunk, amik szintén React komponensek, melyek a react-router-dom modulban vannak definiálva. Az egyes URL-ek esetén megjelenítendő komponenseket Route-okként adjuk meg. Alapértelmezetten a React router minden, az éppen aktuális URL-re (akár részben) illeszkedő komponenst (egymás alatt) megjelenít, de megadható, hogy csak pontos egyezés esetén jelenjen meg az adott oldal. A Home komponens megjelenítése pontos egyezést kíván (csak egy / legyen a base URL végén), különben a másik komponensek megjelenítésekor is illeszkedne a Home útvonala, így az is megjelenne. Ezt az „exact” kulcsszó jelzi az alábbi kódrészletben.

import { BrowserRouter as Router, Route, Link } from "react-router-dom";

function BasicRouter() {

return (

<Router>

<div>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/about">About</Link>

</li>

<li>

<Link to="/topics">Topics</Link>

</li>

</ul>

<Route exact path="/" component={Home} />

<Route path="/about" component={About} />

<Route path="/topics" component={Topics} />

</div>

</Router>

);

}

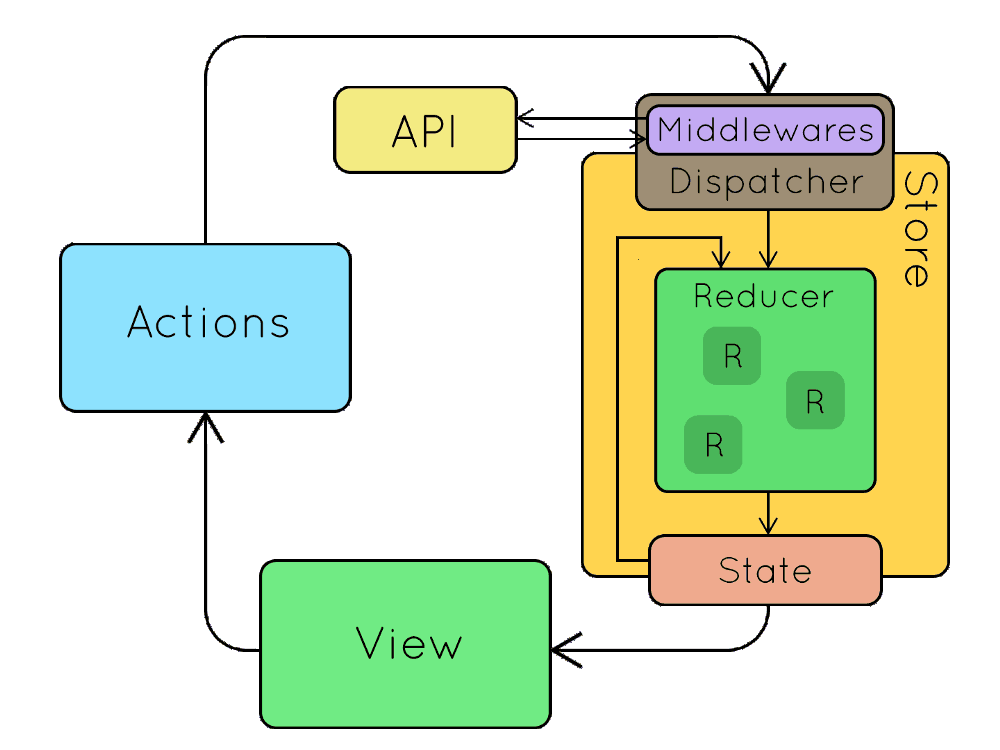
## Redux

A Redux egy központosított állapot tároló, melyet Single-page webes alkalmazásokhoz fejlesztettek ki. Habár legtöbben React mellett használják, léteznek implementációi Angular, Ember, jQuery és vanilla JavaScript-hez is.

A Redux a megjelenítés (presentation) és a tárolók (container) különválasztásával kiszámíthatóbbá teszi az alkalmazásunk viselkedését különböző környezetekben, ezáltal megkönnyíti a tesztelést és hibakeresést. Redux használatával kiemelhető a komponensekből az állapot és egy központi tárolóba (Store) vonható össze. Az állapot központosításával elkerülhető a kódduplikálás olyan komponensek között, melyeknek ugyanarra az állapotra van szükségük.

Ez az állapot a komponensek számára csak olvasható bemenő adatként (úgynevezett „props” -on keresztül) érhető el. Az állapot frissítésére action-öket használhatunk. Az actionök végrehajtásakor az új állapotot a Reducer állítja elő. A Reducerek tiszta függvények (pure function) azaz ugyanarra a bemenetre mindig ugyanazt a kimenetet adják. A megjelenítésért felelős React komponensek számára a Redux használata transzparens, azaz nem függenek tőle.

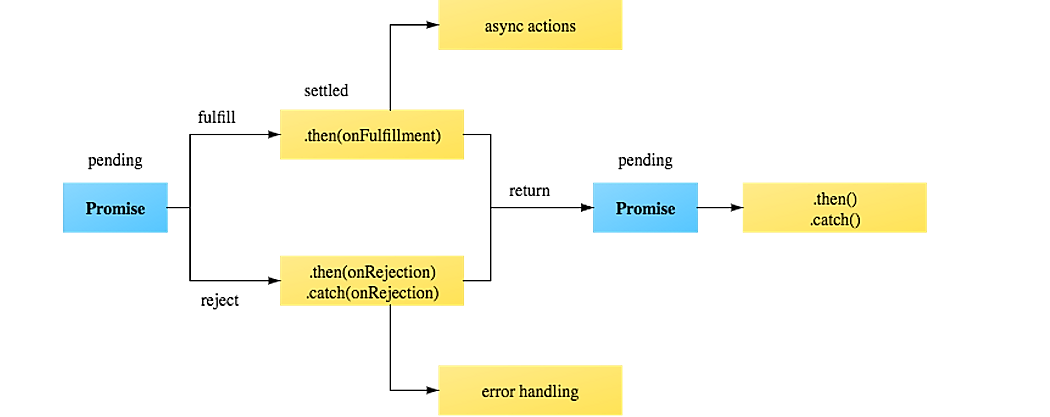
Webes környezetben különösen fontos az aszinkron adatelérés, a felhasználói élmény miatt. Az aszinkron kérésekre adott válaszok feldolgozását egy Middleware modul végzi (pl.: Thunk). Az állapot frissülésekor a Redux a „props” -on keresztül értesíti a komponenseket, akik frissítik a megjelenésüket.



. ábra Aszinkron állapotfrissítés Reduxban [13]

## Axios

Az Axios [14] egy Promise alapú, aszinkron HTTP kliens, mellyel JSON (JavaScript Object Notation) alapú kéréseket küldhetünk a böngészőnkből vagy Node.js-ből. Az Axios támogatja a REST fontosabb üzenet típusait, küldhetünk vele GET, POST, PUT és DELETE kéréseket is. Az Axios beépítve tartalmaz hibakezelést, értelmezi a HTTP státusz kódokat és lekezelhető JavaScript (vagy TypeScript) kivételekké fordítja őket. Az Axios a válaszok JSON-ból történő konverzióját is automatikusan megoldja. Az aszinkron műveletek eredményének elérésére egyaránt használható callback függvényhívás (a .then() metódus) vagy az ES2017-től elérhető async-await kulcsszavak.



. ábra A JavaScript Promise állapotai [15]

## Bootstrap 4

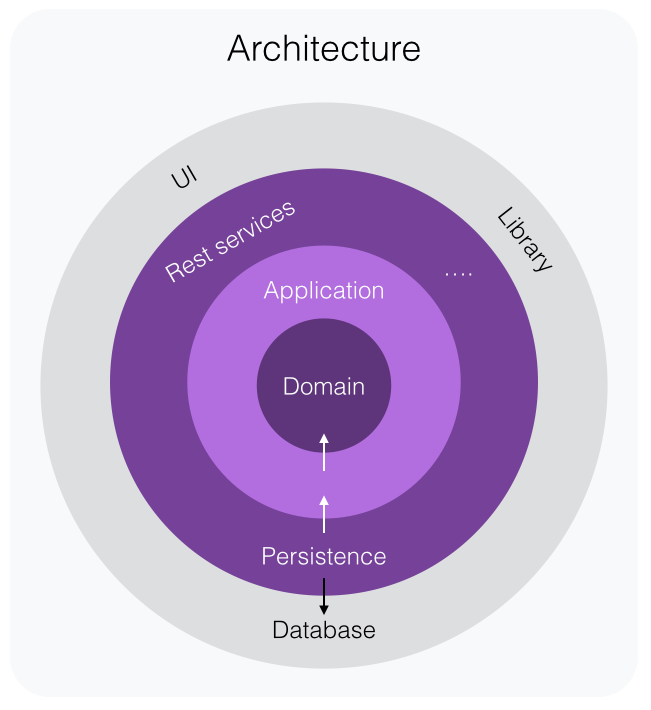
A Bootstrap az egyik leggyakrabban használt CSS (Cascading Style Sheets) keretrendszer, amivel igen egyszerűen hozható létre reszponzív layout. Használatával gyors prototípusokat és flexibilis weboldalakat készíthetünk előre definiált stílusok, színek és formázások felhasználásával. A Bootstrap a mobile first elvet követi, melynek értelmében az elrendezést először mobil képméretre adjuk meg, a szélesebb töréspontoknál csak az eltéréseket írjuk le.

A Bootstrap 4-es [16] verziójának újításai a Flexbox alapú elrendezés, a SASS (Syntactically Awesome StyleSheets) CSS preprocesszor alkalmazása és egy újabb töréspont megjelenése.

## Architekturális és tervezési minták

### Domain Driven Design

A Domain Driven Design (vagy magyarra fordítva fogalomtér-vezérelt tervezés) egy tervezési irányelv, melynek segítségével összetett üzleti logikával rendelkező és könnyen bővíthető alkalmazásokat készíthetünk. A DDD az alkalmazás modell és üzleti logikai osztályaira helyezi a hangsúlyt, hiszen ezek adják az alkalmazás platform és infrastruktúra független magját.

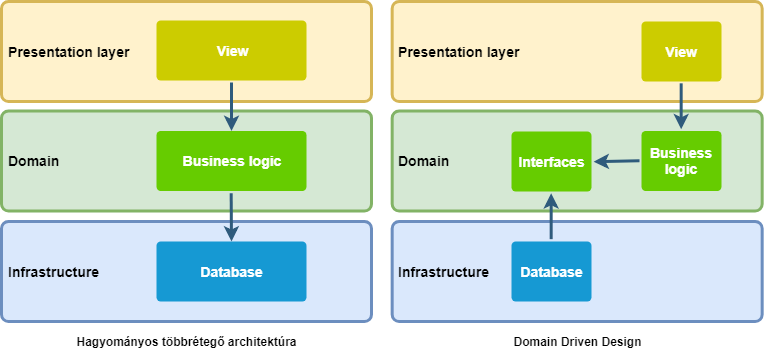


. ábra Egy DDD alkalmazás felépítése [19]

Az irányelv célja a hagyományos többrétegű architektúra DAL (Data Access Layer) -> BLL (Business Logic Layer) függőségének megfordítása, így az adatbázis vagy más infrastrukturális elemek az üzleti logika megváltoztatása nélkül lecserélhetők.

A DDD egyik fontos fogalma a Bounded Context [17], amely a Domain-t kisebb problémákra bontja, ezáltal könnyebben érthetővé és karbantarthatóvá teszi. A különböző Bounded Context-ek minimális mértékben függenek egymástól és egy jól körülhatárolt funkció megvalósításáért felelősek. A Domain különálló részekre bontása segíti a különálló fejlesztői csapatok által történő párhuzamos fejlesztést úgy, hogy az egyes csapatok egy-egy Bounded Context-en dolgozhatnak. Ilyen lehet például a felhasználókezelés, az előfizetések vagy a megrendelések kezelése.

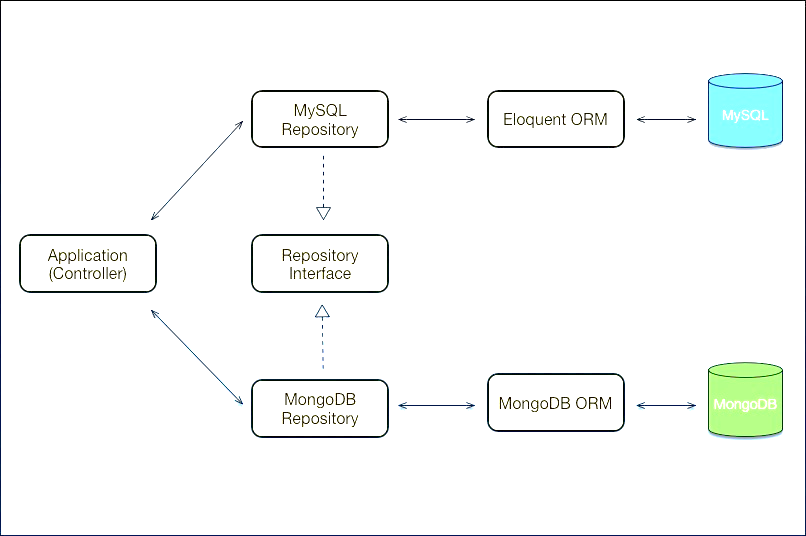
Nagyobb alkalmazások fejlesztésekor elengedhetetlen a probléma több rétegre bontása. A rétegeken belüli erős kohézió és a különböző rétegek közötti gyenge csatolás elősegíti az alkalmazás karbantarthatóságát és bővíthetőségét. Az egyes rétegeknek szolgáltatásokat nyújtanak a fölöttük elhelyezkedő réteg számára és használják az alattuk lévő réteg szolgáltatásait. Ideális esetben minden réteg csak a közvetlenül alatta lévő rétegtől függ. A Domain Driven Design gyakorlati alkalmazásakor is rétegeket vezetünk be alkalmazásunk architektúrájába, a hagyományos megközelítéssel ellentétben azonban nem az üzleti logika függ az adatbázistól, hanem éppen fordítva. Kevésbé komplex alkalmazásoknál a Domain entitások és a Service-k egy üzleti logika projektbe vonhatók össze, ahogy ezt a következő ábra is mutatja.



. ábra A hagyományos és DDD tervezés rétegeinek összehasonlítása

### Repository

A Repository [18] tervezési minta hatalmas népszerűségre tett szert, mióta 2004-ben a Domain Driven Design részeként bemutatták. A minta egy egységes interfészt nyújt a különböző típusú adatelérésekhez, használatával az üzleti logika független maradhat attól, hogy az adatforrást relációs adatbázis, NoSQL adatbázis, a fájlrendszer vagy akár egy távoli API biztosítja. A Repository interfészek használata az infrastruktúrától függetlenné teszi a tesztelést is, alkalmazásunk tesztelése során használhatunk egy mockolt memóriabeli adatbázist (vagy akár csak egy listát), míg az éles környezetben MSSQL-t.



. ábra Repository használata több adatbázis motorral [19]

Mivel az alkalmazásunk független marad az adatelérés konkrét implementációjától, tovább csökken a csatolás a rétegek között és agy új adatelérés bevezetése nem kíván módosítást az üzleti logikában. A Repository implementációjára több lehetőségünk is van. Definiálhatunk külön Repository interfészt minden üzleti entitás számára, ezáltal mindig csak a ténylegesen használt metódusokat kell felüldefiniálni. A másik lehetőség, hogy generikus interfészt használhatunk, ekkor a programkódunk egységesebb lesz, de ez a megközelítés több használatlan kódhoz vezethet.

Az alábbi kódrészlet [18] egy generikus Repository interfészt mutat be.

public interface IRepository<T> where T : EntityBase

{

T GetById(int id);

IEnumerable<T> List();

IEnumerable<T> List(Expression<Func<T, bool>> predicate);

void Add(T entity);

void Delete(T entity);

void Edit(T entity);

}

public abstract class EntityBase

{

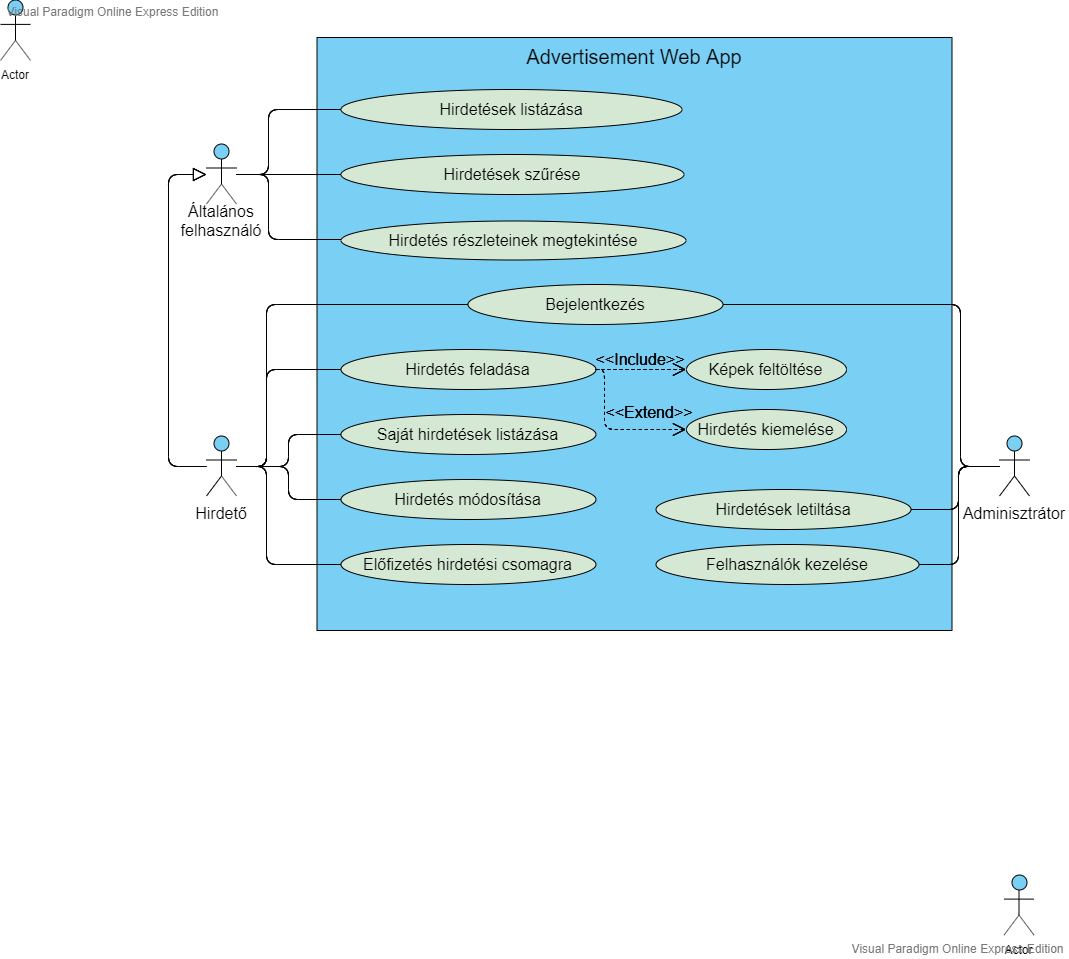
public int Id { get; protected set; }

}

# Követelmények

A szakdolgozatom során egy online hirdetési keretrendszert valósítok meg. Az alkalmazás fejlesztésekor fontos szempont a nagyfokú konfigurálhatóság és a könnyű bővíthetőség, így az egy hirdetéshez felvehető mezők ne legyenek a kódba beégetve, hanem adatbázistáblákból legyenek kiolvashatók. Így ezen mezők listája akár telepítés után is bővíthető lesz, illetve a különböző telepítéseknél különböző hirdetési mezők adhatók meg. Az alkalmazásban legyen lehetőség felhasználók kezelésére, új felhasználó létrehozására, jelszó módosítására vagy meglévő felhasználó törlésére. Az alkalmazás bizonyos funkciói legyenek bejelentkezéshez kötöttek, ezen belül is különüljön el az egyszerű hirdetői és az adminisztrátori feladatkör. Az alkalmazás webes felülete használjon reszponzív elemeket, a megjelenése a lehető legtöbb képernyőmérethez igazodjon. A gyors visszajelzések érdekében a kliens oldal is valósítson meg egyszerűbb validációkat és az esetleges gyakori hibaüzenetek (pl.: sikertelen bejelentkezés) is jelenjenek meg a UI-on.

## Use Case diagramm



. ábra Az alkalmazás főbb funkcióinak use case diagrammja

Az alábbiakban külön pontokba szedtem az egyes felhasználó típusok által látható tartalmakat és elvégezhető műveleteket.

## Böngészési funkciók

A legalapvetőbb funkció a már meglévő hirdetések böngészése. A felhasználók bejelentkezés nélkül is legyenek képesek megtekinteni a mások által feltöltött hirdetéseket. A gyors áttekintés érdekében a hirdetések legyenek egy lista nézetben megtekinthetőek, melyen a felhasználó több (az adatbázisban konfigurálható) hirdetési mező alapján szűréseket végezhet. Az alapvető szűrési paraméterek legyenek a hirdetés címe, ára és kategóriája, de ezen kívül lehessen részletesebb szűréseket végezni az adatbázistáblákban tárolt metaadatok alapján is. A hirdetések listája legyen rendezhető olyan alapvető hirdetési adatok szerint, mint a hirdetés címe, ára vagy a feltöltés dátuma. A kiemelt hirdetések a rendezéskor is a lista elején jelenjenek meg. Az alkalmazás támogassa a növekvő és csökkenő sorrendet is. Mivel egy valós hirdetési alkalmazásban a hirdetések száma elérheti a milliós nagyságrendet, ezért fontos, hogy a listázáskor ne töltődjön be az adatbázisban összes tárolt hirdetés, hanem azok szerver oldali lapozással legyenek elérhetőek.

Ha a felhasználó érdeklődik az egyik hirdetés iránt, akkor legyen képes annak részleteit is megtekinteni. Lehessen egy adott hirdetés részleteihez navigálni a listás nézetben az adott hirdetés kiválasztásával. Ekkor a hirdetés részletei külön oldalon jelenjenek meg. A részleteknél szerepeljenek a hirdetés részletes adatai, a hirdetés kategóriája, a hirdetéshez feltöltött képek és a hirdető elérhetőségei is.

## Hirdetői funkciók

Az alkalmazásban regisztráció, majd bejelentkezés után legyen lehetőség hirdetés feladására. Hirdetés feladásakor az alkalmazás végezzen validációt a feltöltött adatokon és az esetleges hibákat vagy hiányzó értékeket jelezze a felhasználónak. Az egyszerűbb validációkhoz az alkalmazás használjon kliens oldali logikát is. Hirdetésekhez legyen lehetőség akár több kép feltöltésére is, melyeket majd a hirdetés részleteinél tekinthetünk meg. A szerver a képeket fájlrendszerben tárolja, felhasználók szerinti mappastruktúrában. A feltöltésnél az esetleges fájlnév ütközéseket a program átnevezéssel automatikusan oldja fel. A hirdetés törlésekor a hozzá tartozó képek is törlődjenek a szerverről. A hirdetőknek (plusz előfizetés fejében) legyen lehetőségük hirdetéseik kiemelésére. Az alkalmazás támogasson több kiemelési lehetőséget is. A hirdetések listázásakor a kiemelt hirdetések előbb jelenjenek meg és vizuálisan is különüljenek el a normál hirdetésektől. Legyen lehetőség a hirdetések főoldalon történő megjelenítésére is. Hirdetés létrehozása után a hirdetők saját hirdetéseik listáját is tekinthessék meg, ha kell, törölhessenek közülük vagy egyesével módosíthassák a saját hirdetéseiket.

A hirdetők különböző (az adatbázisban szabadon konfigurálható) előfizetési csomagok közül választhassanak, melyek tartalmazzák az általuk feladható hirdetések maximális számát és az elérhető hirdetés kiemeléseket. Az egyes előfizetési csomagokhoz tartozzon ár és időtartam is. A hirdetők legyenek képesek dinamikusan váltani az egyes csomagok között, és a kisebb csomagra váltásokat is kezelje a rendszer. A regisztrált felhasználóknak legyen lehetőségük a jelszavuk módosítására és a felhasználói fiókuk törlésére is. Az előfizetéssel rendelkező felhasználók tekinthessék meg az előfizetésük részleteit, köztük a lejárati dátumot és az előfizetésből jelenleg felhasznált hirdetés kiemelések számát is.

## Adminisztrátori funkciók

A hirdetési rendszer kezelését egy, a telepítéskor létrehozott adminisztrátor felhasználó kezelje. Az adminisztrátor, természetesen bejelentkezés után, legyen képes áttekinteni a regisztrált felhasználók listáját és a hozzájuk tartozó előfizetéseket. Az adminisztrátornak legyen lehetősége a felhasználók listázására, keresésére és az egyes felhasználók letiltására. Az így letiltott felhasználók nem jelentkezhetnek be, nem hozhatnak létre új hirdetést és eddigi hirdetéseik is el lesznek rejtve a listázásokkor. Az adminisztrátor egyes hirdetéseket is tilthasson le. Ilyenkor az adott hirdetés nem lesz látható a felhasználók számára, a hirdetés tulajdonosát és az adminisztrátort kivéve. A hirdető ekkor is legyen képes a hirdetés szerkesztésére és ha elmúlt a letiltás oka, akkor az adminisztrátor újra engedélyezhesse a hirdetés megjelenítését. Amennyiben az adminisztrátor újra engedélyezi egy kiemelés megjelenítését, az a lehetőség szerint a felhasználó által megadott kiemelési szintnek megfelelően jelenjen meg.

# Architektúrák

Az alkalmazás architektúráját több szinten fogom bemutatni. A tervezés során törekedtem az újrafelhasználhatóságra, és a komponensek közti laza csatolásra. Az alkalmazás egyes szintjein más-más architekturális mintákat és tervezési elveket követtem.

## Rendszer architektúra

A hirdetési alkalmazás két fő komponensből - egy webes kliensből és egy azt kiszolgáló szerverből - épül fel. A szerveren belül elkülönül a backend üzleti logikai és az adatelérési réteg, így az alkalmazás magas szintű felépítését egy klasszikus három rétegű architektúra határozza meg. A webes kliens és a szerver futása fizikailag is elkülönül egymástól. A React technológiával megvalósított kliens a felhasználók böngészőjében, míg a .Net szerver egy erre dedikált központi gépen vagy gépeken fut. Jelenleg az adatbázis fizikailag nem különül el a backend logikát futtató komponensektől, de logikailag a két egység egyértelműen külön van választva, így később az adatbázismotor akár külön gépen is futhat.

Kliens

**Http kérések**

**SQL utasítások**

Szerver

. ábra Az alkalmazás főbb architekturális komponensei

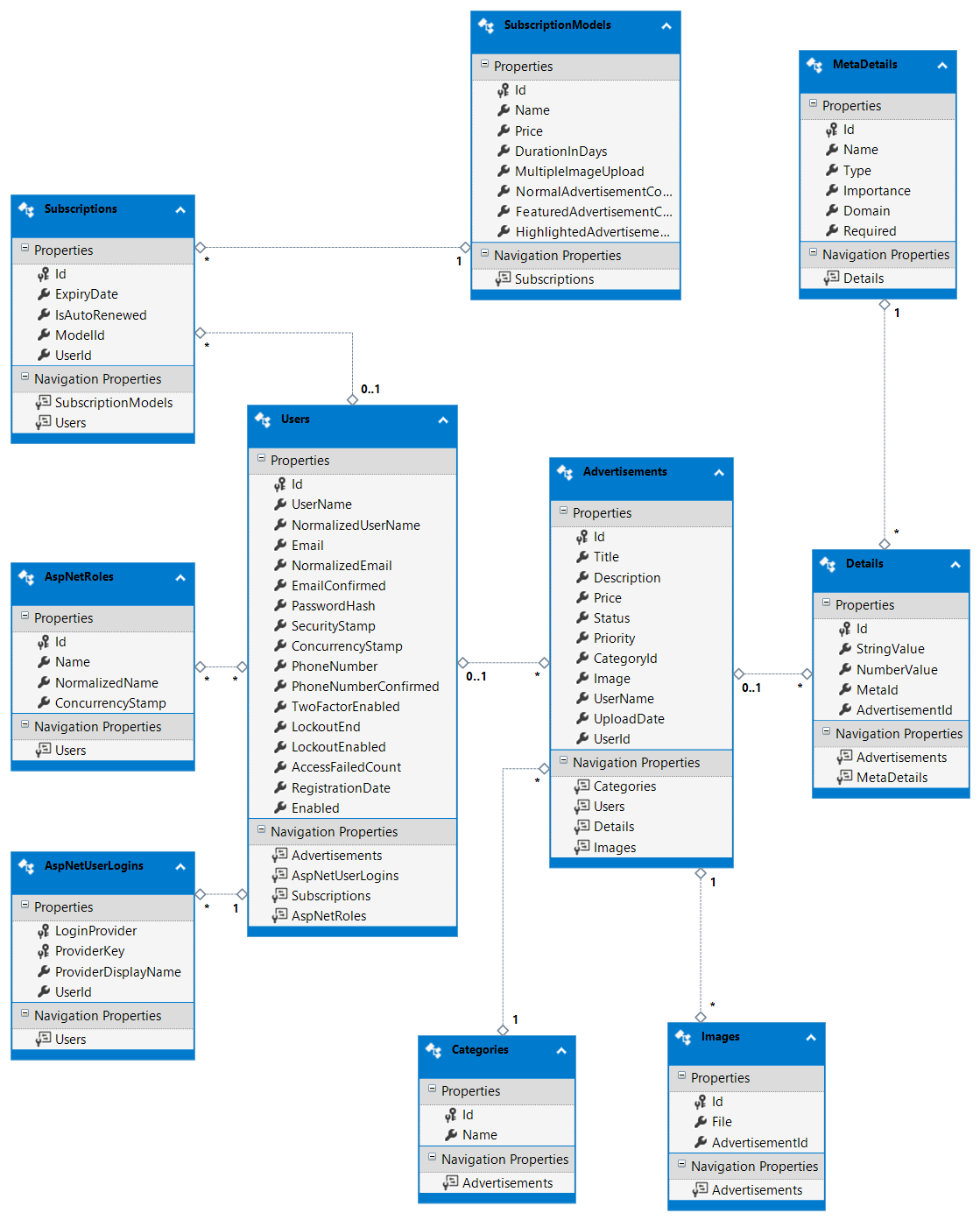
A kliens és a szerver elkészítéséhez eltérő technológiákat alkalmaztam. A frontend egy webes single-page alkalmazás. Ennek megfelelően a weboldal váza egy html fájlt tartalmaz, melynek tartalma a kliens oldalon futási időben generálódik a böngésző által futtatott JavaScript kód segítségével. A kliens elkészítéséhez a TypeScript nyelvet és a React osztálykönyvtárt választottam. A React az Angularhoz képest kevesebb beépített szolgáltatást tartalmaz, így további könyvtárak felhasználására volt szükségem a Routing és az állapottárolás megvalósításához. Előbbihez a React-routert, míg utóbbihoz a Redux-ot használtam fel. Az alkalmazás szerver oldalának fejlesztéséhez platformfüggetlensége és dinamikus fejlődése miatt a .Net Core keretrendszert választottam. A backend API-ját az ASP.Net Core (2.1-es verziójának) segítségével implementáltam. Ez a könyvtár biztosította a kontroller osztályokat, és a felhasználókezelésem alapját képező Identity modult is.

A kliens és a szerver közötti hálózati kommunikációt http kérések és válaszok segítségével oldottam meg. A kliensről indított kéréseket a szerver egy REST API-n [20] keresztül fogadja, majd a kérés feldolgozása után a választ JSON formátumba sorosítva küldi el a kliensnek. A platformfüggetlen REST API lehetővé teszi, hogy a szerver később más (pl.: mobilos vagy desktop) kliens alkalmazásokkal is együttműködjön. Az erőforrások köré szervezett API tervezése során felhasználtam az egyes http igék jelentését és az erőforrásokra egyedi azonosító vagy keresési paraméterek segítségével lehet hivatkozni.

Az adatok perzisztens tárolásáért felelős relációs adatbázist egy ORM osztálykönyvtáron, az Entity Framework Core-on keresztül érem el a backend szolgáltatásokból. Az adatbázismotor jelenleg egy Microsoft SQL Server. Ez azonban könnyen módosítható, sőt az Entity Framework miatt más gyártók adatbázismotorjaira is lecserélhető. Entity Framework Core fordítja a programkódban kiadott LINQ lekérdezéseket az adatbázis számára értelmezhető SQL utasításokká.

## Az adatbázis felépítése

Az adatbázis tervezésekor a Code first megközelítést alkalmaztam. A kezdetben definiált entitás osztályokat az alkalmazás fejlesztése közben folyamatosan finomítottam, bővítettem. Az adatbázis végső felépítése az alábbi ábrán látható.



. ábra Az adatbázis felépítése

A felhasználókezeléssel kapcsolatos táblákat az ASP.Net Identity definiálta. Ezekből a fontosabbak a következők:

* **Users:** Az alkalmazás felhasználóit tartalmazó tábla. A felhasználónéven és a hash-elt jelszón kívül tartalmazza a felhasználó elérhetőségeit (melyeket a hirdetéseknél is megjelenítünk), a regisztráció dátumát, és azt, hogy az adott felhasználó le van-e tiltva.
* **Roles:** A felhasználói szerepköröket tartalmazó tábla. Az alkalmazás jelenleg két szerepkört, a User-t és az Administrator-t különbözteti meg. Az eltérő jogkörük az alkalmazás biztonság aspektusából is fontos.
* **UserRoles:** A felhasználókat a szerepköreikhez rendelő kapcsolótábla.

A hirdetések fix adattagjait egy táblában tárolom, míg a hirdetéshez tartozó extra adatok egy külön táblában, félig strukturált módon kerültek tárolásra. Ezt az alábbi táblák írják le:

* **Advertisements:** A hirdetések fix adattagjait (cím, ár, leírás, státusz, prioritás és a feltöltés dátuma) tartalmazó tábla. Ebben jelenik meg továbbá a hirdetés fő képének elérési útvonala, amely alapján a kép a hirdetések listázásánál is látható.
* **Details:** Minden hirdetéshez több részlet tartozhat, melyek vagy egy szöveges vagy egy szám értéket tárolnak. Ezen értékeket tartalmazza a Details tábla. Minden részlethez tartozik ezen felül egy meta részlet is, amely megadja a részlet típusát, nevét, fontosságát, valamint opcionálisan a mértékegységét és az értékkészletét.
* **MetaDetails:** A részletekhez tartozó metaadatokat tartalmazó tábla. Egy metaadathoz több részlet is tartozhat.
* **Categories:** A hirdetések kategóriái lényegében egy listában tárolva.
* **Images:** A hirdetésekhez tartozó képek elérési útvonalát string-ként tartalmazó tábla.

Az alkalmazásban megkülönböztetem az előfizetéseket (Subscription), amelyek egy konkrét felhasználóhoz vannak rendelve, valamint az előfizetési modelleket (Subscription Model), amelyek egy elérhető hirdetési csomag részleteit tartalmazzák. Ezek az információk az alábbi táblákban kerültek tárolásra:

* **Subscriptions:** Tartalmazza az előfizetés lejárati dátumát és idegen kulcsokat az előfizetési modellre és a felhasználóra.
* **SubscriptionModels:** Egy előfizetési modell rendelkezik névvel, árral és időtartammal. Modellenként megadható, hogy a felhasználó tölthet-e fel hirdetésenként több képet, illetve, hogy hány hirdetést jeleníthet meg kiemelési szintenként.

## Backend architektúra

A backend is alapvetően rétegekből épül fel, de eltér a hagyományos három rétegű modelltől [21], mivel az architektúra tervezése során a Domain Driven Design alapelveit követtem. Ennek értelmében az alkalmazásom középpontjában az üzleti logika szolgáltatásai (Services) és a model osztályok állnak. Mivel ezek képezik az alkalmazás értékét, így fontos ezek infrastruktúrától és adatbázistól való függetlenségének megőrzése. Az alábbi ábrán is jól látható, hogy az alkalmazásom Domain komponense független a konkrét adatbázistól és a hálózati kommunikációt megvalósító API rétegtől is.

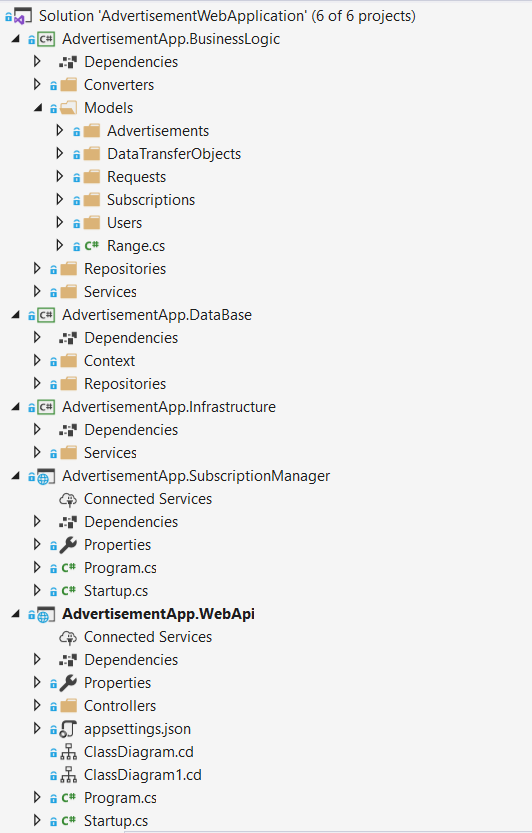
. ábra A backend architektúra elemei és függőségeik

A modellek és szolgáltatások megvalósítása a Business Logic projektbe került. A projekten belül a model osztályokat további névterekbe és mappákba szerveztem, ezáltal is követve a DDD Bounded Context[[1]](#footnote-1) tervezési elvét. A Domain kisebb részekre bontása (jelen esetben ezek a felhasználókezelés, a hirdetések és az előfizetések) segíti a könnyebb megérthetőséget és a jobb újrahasznosíthatóságot. Az alkalmazás tervezése során igyekeztem az egyes Bounded Context-ek közti függőségek minimalizálására, így azok külön szolgáltatás osztálybaba kerültek, és az adatelérésük is külön Repository-n keresztül valósul meg. Szintén a Business Logic rétegben definiálom az adatelérés interfészeit (Repositories) és a klienstől érkező kérések felépítését (Requests). A szolgáltatások csak a Repository interfésztől függenek, a konkrét megvalósítás transzparens a számukra. A referenciát a konkrét Repository osztályokra az ASP.Net által támogatott Dependency Injection konténer biztosítja. Ezáltal az alkalmazás üzleti logikáját nem szükséges módosítani, ha később más adatbázismotor használatát vezetjük be.

A modelleken belül a hálózaton átvitt adatokat csomagoló osztályoknak külön DTO[[2]](#footnote-2) objektumokat definiáltam. A DTO-k használatával tovább csökkenthető a kliens és a szerver függősége: egy módosítás az adatbázisban nem feltétlenül vonja mag után egy kliens oldali módosítás igényét. Szintén a DTO objektumok bevezetése teszi lehetővé, hogy a kliens mindig csak a számára szükséges adattagokat kapja meg a szervertől. Ezért külön DTO tartozik egy entitás részletes megjelenítéséhez és a listás megjelenítéséhez, mivel utóbbihoz kevesebb adat is elegendő.

Az alkalmazás talán legfontosabb elemei a szolgáltatások. Külön szolgáltatást definiáltam minden Bounded Context számára. Ezek végzik a kérések tényleges feldolgozását, a Repository-kon keresztül betöltött adatok transzformálását, összekapcsolását és DTO objektumokba történő konvertálását. Az AdvertisementService felelős a hirdetések létrehozásáért, listázásáért és kezeléséért, a UserService végzi a felhasználók autentikációját, a felhasználó részletes adatainak lekérdezését és a JWT [22] tokenek előállítását. A SubscriptionService kezeli a felhasználók előfizetéseit, lekérdezhetők tőle az előfizetési modellek, és képes eldönteni, hogy egy feltöltendő hirdetés megfelel-e a felhasználó előfizetésének. Ez a mappa tartalmazza még az ImageService-t, amely a képek tárolásáért és lekéréséért felelős, a MetaDetailService-t, amely a hirdetéshez rendelhető mezőket kezeli és a DataSeeder-t, amely demo adattal tölti fel az adatbázist.

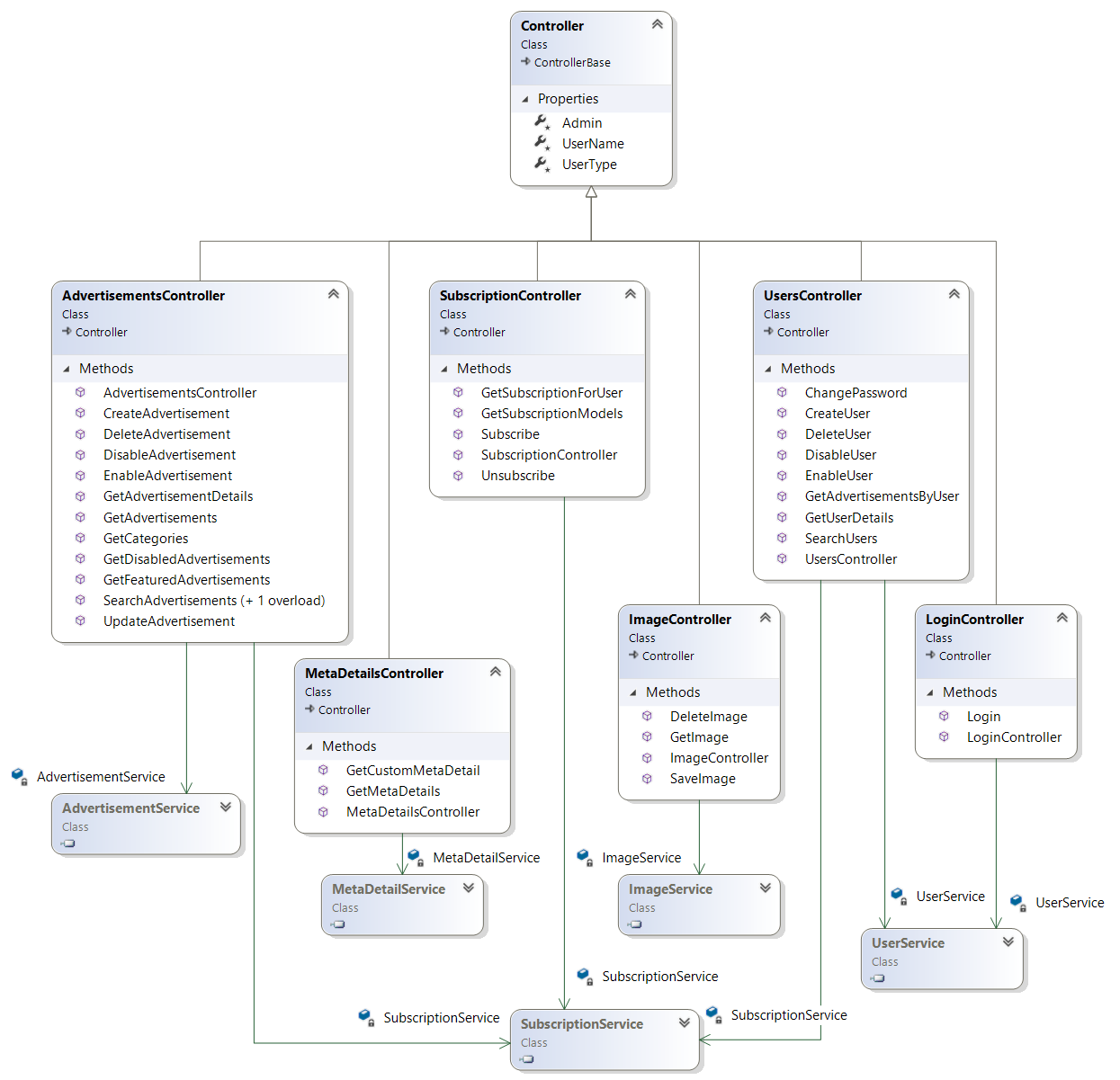
A Solution-ben szereplő összes többi projekt a Business Logic-tól függ. Ezek felépítése az alábbi ábrán látható.



. ábra A Visual Studio Solution felépítése

A DataBase projekt az adatbáziskezeléssel kapcsolatos kódrészleteket tartalmazza. Itt található a DB Context definiálása és a fentebb említett adatelérési interfészek megvalósítása is. Külön Repository tartozik a hirdetések kezeléséhez, a hirdetések metaadataihoz, a felhasználókhoz, az előfizetésekhez és a fájlkezeléshez. Utóbbi ugyan szorosan nem az adatbázis része, de a képek fájlrendszerben történő tárolását is infrastruktúra-független módon szerettem volna megvalósítani, így ott is a Repository tervezési mintát alkalmaztam. Ezáltal a hirdetésekhez tartozó képek később külön szerveren vagy akár felhőszolgáltatásban is tárolhatóak. Az adatbázisban az üzleti logikai modelleket tárolom, mivel egyelőre a projekt bonyolultsága nem követelte meg a külön entitás osztályok definiálását.

A WebAPI nevű projekt valósítja meg a REST API-t. A projekt ASP.Net Core Controller osztályokat tartalmaz, melyek az üzleti logika szolgáltatásait használják. A Controllerek számára a közös viselkedés kiszervezésére egy ősosztályt vezettem be, amely az olyan, több Controllerben is felmerülő műveleteket valósítja meg, mint a felhasználó azonosítása, vagy a felhasználó típusának meghatározása. A Controller osztályok és a velük kapcsolatban álló Service-ket az alábbi osztálydiagramm mutatja be.



. ábra A Controller és Service osztályok kapcsolata

A diagrammon látható, hogy a SubscriptionService-t több Controller is használja, mivel például hirdetés feladásakor is ellenőrizni kell, hogy a felhasználó rendelkezik-e a megfelelő csomaggal.

A SubscriptionManager projekt lényegében egy konzolos alkalmazás, amely indításakor ellenőrzi a lejárt előfizetéseket, és ennek megfelelően módosítja a hozzájuk tartozó hirdetések megjelenését, hogy a felhasználók ne léphessék túl a csomagjukban foglalt hirdetésszámot. Ehhez a SubscriptionManager a SubscriptionService szolgáltatásait használja fel. A projekt éles környezetben háttérfolyamatként futtatható, például napi rendszerességgel.

Az Infrastructure projektbe kerültek az egyéb olyan szolgáltatások, melyek szorosan nem részei az alkalmazás domain-nek, de kiegészítő szolgáltatásokat nyújtanak számára. Az ImageEditor osztály, a hirdetésekhez feltöltött képeket egységes oldalarányúra méretezi át, a MapperInitializer a DTO-k és a model osztályok közti automatikus konverziót konfigurálja, a DemoPaymentService pedig egy dummy kiterjesztési lehetőséget nyújt külső fizetési szolgáltatások későbbi implementációjához. Szintén itt került implementálásra egy CustomPasswordHasher algoritmus, melyet a tesztelés és hibakeresés során használtam.

## Frontend architektúra

A kliens oldalon is jól strukturált kódra törekedtem, melyet a mappaszerkezettel alakítottam ki. Külön mappába került az állapotkezelés, a model osztályok, a http kommunikációért felelős függvények, a felületi komponensek, valamint az ezekből felépülő oldalak.

A http mappa tartalmazza a hálózati kommunikációért felelős osztályokat, melyeket igyekeztem a backend struktúrájához igazítani. Egy modulba kerültek egy API Controller-rel kommunikáló DataLoader függvények. Később ez a kódszervezés előnyt jelenthet egy microservice [23] architektúrára történő áttérésnél.

. ábra A kliens oldali mappastruktúra

A kialakult architektúrát nagyban befolyásolta a felhasznált Redux technológia is. A felület legtöbb komponense a szülő objektumától kap adatokat, és az eseményeit is a szülőjének publikálja. A Redux Store-ral csak a teljes oldalas (page) komponensek állnak kapcsolatban. A kliens oldali alkalmazás központi állapotát a Store két külön részében tárolom, melyek a felhasználókért felelős UserState és a hirdetéseket tároló AdvertisementsState. Az ezekhez tartozó action-ök és reducer-ek definícióját szintén külön mappákba szerveztem. A nem teljes oldalért felelős felületi elemeket (components) a felhasználói szerepkör alapjás strukturáltam. Külön mappába kerültek a hirdetéseket böngésző felhasználókhoz (User), a hirdetőkhöz (Advertiser) és az adminisztrátorhoz (Admin) tartozó komponensek. A közös (Common) mappába olyan komponensek kerültek, melyek nem köthetők egyetlen felhasználói körhöz. Ilyen például a navigációs sáv, vagy a bejelentkezésért felelős komponens.

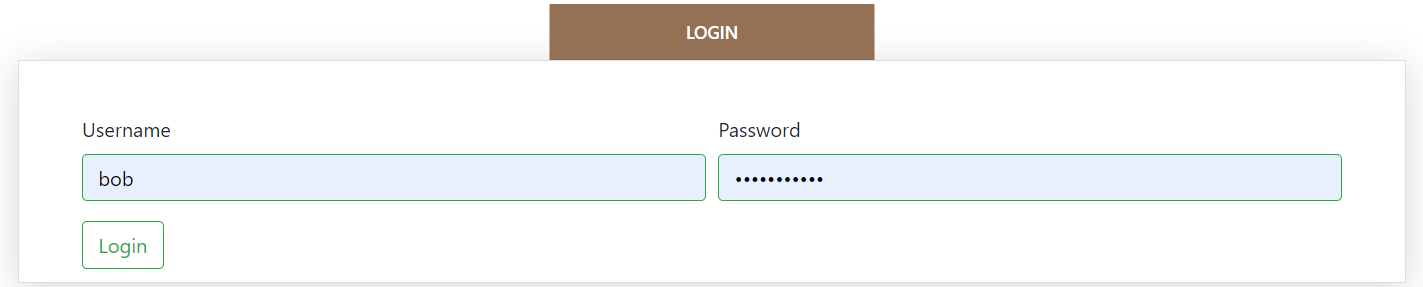
A mappákon belül az egyes komponensek külön-külön fájlba kerültek. Egy fájl a komponensen kívül tartalmazza az állapotának (ISate) és szülőjétől kapott adatoknak (IProps) az interfészét is.

# Megvalósítás

Az alkalmazás részletes implementációját néhány kiemelt use-case-n keresztül fogom bemutatni.

## Bejelentkezés

A hirdetési rendszer egyik legalapvetőbb funkciója a regisztrált felhasználók bejelentkezése. A felhasználók a webes felületen felhasználónevük és jelszavuk megadásával tudnak belépni. Ezen mezők megjelenítéséért és az input kezeléséért a Login komponens felelős.



. ábra A Login komponens megjelenése

Mint majdnem minden TypeScript fájl, a Login kódja is a más fájlokból importált típusok megadásával kezdődik. A Login a React által definiált Component osztályból származik le, az egységes megjelenítéshez pedig a react-bootstrap könyvtár beviteli mezőit használom. Az ErrorModel egy hiba típusát és leírását tartalmazza, a szervernek küldött LoginRequest hordozza a felhasználónevet és jelszót, a ReactInputEvent pedig a könnyebb átláthatóság kedvéért általam definiált type alias[[3]](#footnote-3).

import React, { Component } from "react";

import Form from "react-bootstrap/Form";

import Button from "react-bootstrap/Button";

import Col from "react-bootstrap/Col";

import Row from "react-bootstrap/Row";

import Container from "react-bootstrap/Container";

import { ErrorModel } from "../../../models/common/ErrorModel";

import { LoginRequest } from "../../../models/user/UserRequests";

import { ReactInputEvent } from "../../../models/common/ReactInputEvent";

A komponens a szülőjétől egy opcionális „error” bemenetet[[4]](#footnote-4) vár el, és egy „onLoginClicked” eseményt publikál. Utóbbi a bejelentkezés gomb megnyomását jelzi. Bejelentkezési hiba esetén a komponens formázott hibaüzenetet is képes megjeleníteni, melyet az error prop-on keresztül kap meg. A komponens bemeneteit az alábbi interfész mutatja be.

interface IProps {

onLoginClicked: (request: LoginRequest) => void;

error?: ErrorModel;

}

A Login komponens saját állapottal is rendelkezik, melyet az IState típusban definiálok.

interface IState {

name?: string;

password?: string;

}

A komponens leszármazik a megfelelő template paraméterekkel ellátott Component osztályból, és a konstruktorában inicializálja belső állapotát.

class Login extends Component<IProps, IState> {

public constructor(props: IProps) {

super(props);

this.state = {

name: "",

password: ""

};

}

…

A komponens megjelenését annak render() függvényében adom meg az alábbi módon, JSX szintaxis használatával.

render() {

return (

<Container>

<Row>

<Col>

<div className="labeled-form">

<div className="form-title">

<p>Login</p>

</div>

<Form validated={true}>

<Form.Row>

<Col sm md="6">

<Form.Group>

<Form.Label>Username</Form.Label>

<Form.Control

type="text"

placeholder="Enter username"

value={this.state.name}

onChange={(e: ReactInputEvent) =>

this.setState({ name: e.currentTarget.value })

}

className="mr-sm-2 m-0"

required

/>

</Form.Group>

</Col>

<Col sm md="6">

<Form.Group>

<Form.Label>Password</Form.Label>

<Form.Control

value={this.state.password}

onChange={(e: ReactInputEvent) =>

this.setState({ password: e.currentTarget.value })

}

className="mr-sm-2 m-0"

type="password"

placeholder="Enter password"

minLength={6}

required

/>

</Form.Group>

</Col>

</Form.Row>

<Form.Row>{this.renderError()}</Form.Row>

<Button

variant="outline-success"

onClick={() =>

this.props.onLoginClicked({

userName: this.state.name!,

password: this.state.password!

})

}

>

Login

</Button>

</Form>

</div>

</Col>

</Row>

</Container>

);

}

}

A felhasználónév és a jelszó is kötelezően kitöltendő, ezért egy validált form-on belül vettem fel a két input mezőt és a bejelentkezést elküldő gombot. Az eseménykezelőkhöz inline lambda függvényeket rendelek, melyek a belső állapot adott mezőjét állítják. Ez alól kivétel a Login gomb, melynek megnyomásakor a szülőjét értesíti a „props.onLoginClicked” függvényen keresztül, így ennek implementálása a szülő komponens feladata lesz.

Az esetleges bejelentkezési hiba megjelenítéséhez az osztály rendelkezik még egy privát láthatóságú „renderError” függvénnyel is. Ennek implementációja az alábbi:

private renderError() {

var error = this.props.error;

if (error && error.type == "Login")

return (

<div className="alert alert-danger" role="alert">

<strong>{error.code + " "}</strong>

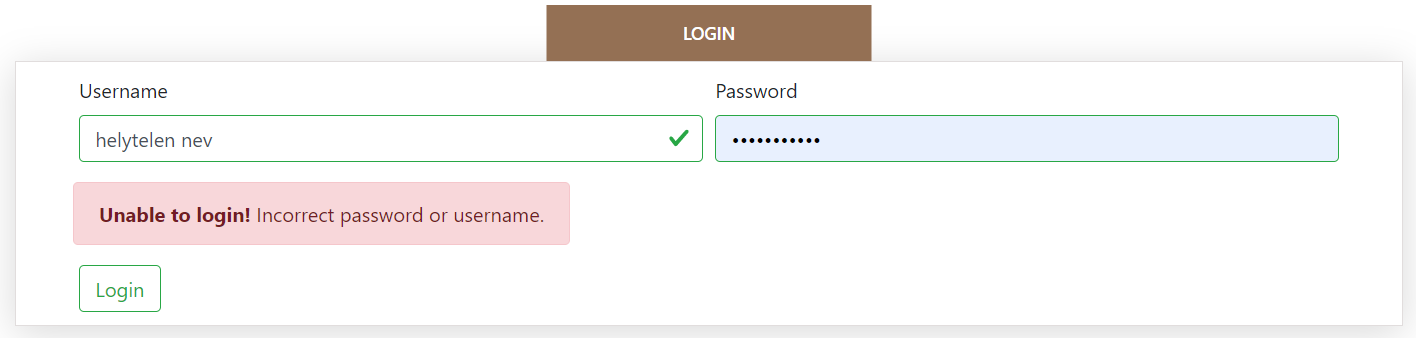
{error.description}

</div>

);

}

Ha létezik az „error” változó, és bejelentkezési hiba történt, akkor kiírjuk a hiba kódját és leírását. A felületen a hiba az alábbi módon jelenik meg:



. ábra Hibaüzenet sikertelen bejelentkezés esetén

A Login több oldalon is megjelenik, jelen példában a MyAdvertisementsPage bejelentkezéshez tartozó részeit fogom bemutatni. A saját hirdetések csak bejelentkezés után érhetők el, így az oldalon alapból csak a Login komponens jelenik meg. A komponens megjelenése megváltozik, ha a felhasználó bejelentkezik. (Ekkor a hirdető saját hirdetéseit listázza). Ennek megállapítására használom a Session változót, amely többek között tartalmazza a felhasználónevet, a felhasználó típusát és a bejelentkezéskor kapott JWT tokent. A komponens bemeneteit a Redux Store-ból kapja, szintén a „props” változón keresztül. Ennek definíciója a következő:

interface IProps extends RouteComponentProps {

…

loginUser: typeof loginUser;

session: Session;

}

A render függvényben történik a Login komponens kirajzolása.

<Login

onLoginClicked={this.handleLogin}

error={this.props.session && this.props.session.error}

/>

Az alábbi függvény pedig a Login komponensnek átadott „handleLogin” és a MyAdvertisementsPage „props”-on keresztül publikált „loginUser” függvényeinek összekötése.

private handleLogin = (request: LoginRequest) => {

this.props.loginUser(request);

};

Annak érdekében, hogy a komponens a Redux Store-ból kapjon adatokat, még további összekötések szükségesek.

//A propok hozzákötése a globális állapothoz

const mapStateToProps = (store: IApplicationState) => {

return {

session: store.users.session

};

};

//Függvények hozzákötése az actionökhöz

const mapDispatchToProps = (dispatch: any) => {

return {

loginUser: (request: LoginRequest) => dispatch(loginUser(request))

};

};

export default connect(

mapStateToProps,

mapDispatchToProps

)(MyAdvertisementsPage);

A fenti hívások érik el, hogy ha a felhasználó lenyomja a Login gombot, az alkalmazás egy loginUser() hívást hajt végre, melynek implementációja a UserActions fájlban található.

export const loginUser: ActionCreator<

ThunkAction<Promise<any>, IUsersState, null, IUsersLoginAction>

> = (request: LoginRequest) => {

return async (dispatch: Dispatch) => {

const session = await loginUserFromAPI(request);

dispatch({

session,

type: UserActionTypes.LOGIN

});

};

};

A függvény egy aszinkron API hívás után egy UserActionTypes.LOGIN típusú action-t hoz létre, melyet a UserReducer-ben kell majd lekezelni. A Thunk middleware-re az aszinkronitás miatt van szükség.

Az API hívást a http mappában található UserDataLoader adja ki. A bejelentkezéshez POST kérést használok, a felhasználónév és jelszó párosát egy LoginRequest objektumba ágyazva, body paraméterben küldöm el a szervernek. A kódrészleten látható, hogy az axios támogatja a http kérésekre érkező típusos válaszokat, így a kódban a válasz objektumot kasztolás nélkül Session-ként kezelhetem. A későbbi kérések során egy Header mezőben továbbítom a bejelentkezéskor kapott tokent. A token tárolása látszik a „.then” ágon. Hiba esetén egy ErrorModel objektumot hozok lére, és azt adom vissza a hívó félnek.

export const loginUser = async (request: LoginRequest): Promise<Session> => {

return await axios

.post<Session>(baseUrl + "login", request, {

headers: headers

})

.then(response => {

axios.defaults.headers.common["Authorization"] =

"Bearer " + response.data.token;

return response.data;

})

.catch(error => {

let errorModel: ErrorModel;

if (error.response == undefined)

errorModel = {

type: "Login",

code: "Unable to login!",

description: "Can't connect to server."

};

else

errorModel = {

type: "Login",

code: "Unable to login!",

description: error.response.data

};

let session = new Session();

session.error = errorModel;

return session;

});

};

A bejelentkezési kérés a szerveren a LoginController-hez érkezik be. A POST kérés és a [HttpPost] attribútum miatt a Login függvény hívódik meg.

[HttpPost]

public async Task<IActionResult> Login([FromBody] LoginRequest request)

{

(Session session, string message) = await UserService.Login(request);

if (session != null)

return Ok(session);

else

return BadRequest(message);

}

A kontroller a UserService-nek továbbítja a kérést, amely sikeres bejelentkezés esetén generál egy JWT tokent (erről részletesen a biztonség fejezetben írok), és elkészíti a Session objektumot, mely a kérés visszatérési értéke lesz. Sikertelen bejelentkezés esetén a függvény egy hibaüzenetet ad vissza, melyet majd a kliens is megjelenít.

public async Task<(Session session, string message)> Login(LoginRequest request)

{

var result = await UserRepository.SignIn(request.UserName, request.Password);

//Ha sikeres a bejelentkezés

if (result.Succeeded)

{

var type = await GetUserRole(request.UserName);

var token = GenerateJwtToken(request.UserName, type);

var session = new Session(request.UserName, true, token.ToString(), type);

return (session, "Succesfull.");

}

//Ha sikertelen

else

{

return (null, "Incorrect password or username.");

}

}

A bejelentkezés sikerességét a UserRepository dönti el. Erre a UserService az IUserRepository interfészen keresztül hivatkozik. Az alkalmazás felhasználóit az ASP.Net Core Identity keretrendszer által definiált Users táblában tárolom. Ebből keressük ki az adott nevű felhasználót, majd ha nincs letiltva, akkor ellenőrizzük a jelszó helyességét.

public async Task<SignInResult> SignIn(string name, string password)

{

var user = await UserManager.FindByNameAsync(name);

//Ha nincs ilyen felhasználó vagy le van tiltva

if (user == null || !user.Enabled)

return SignInResult.Failed;

return await SignInManager.CheckPasswordSignInAsync(user, password, false);

}

A kliens oldalra visszaküldött Session objektum az alábbi adatokat tartalmazza:

export class Session {

status: UserStatus = UserStatus.New;

userType: UserType = UserType.User;

username: string ="";

success: boolean =false;

token: string ="";

error?: ErrorModel;

}

Ez kerül bele a Login action-be. Az action kezelését az úgynevezett Reducer végzi. Ebben a action típusa alapján más és más műveletek vannak definiálva egy switch case szerkezetben. A bejelentkezéshez az alábbi művelet tartozik.

switch (action.type) {

case UserActionTypes.LOGIN: {

if (action.session.success) {

action.session.status = UserStatus.LoggedIn;

return {

...state,

session: action.session

};

} else {

session = new Session();

session.error = action.session.error;

return {

...state,

session: session

};

}

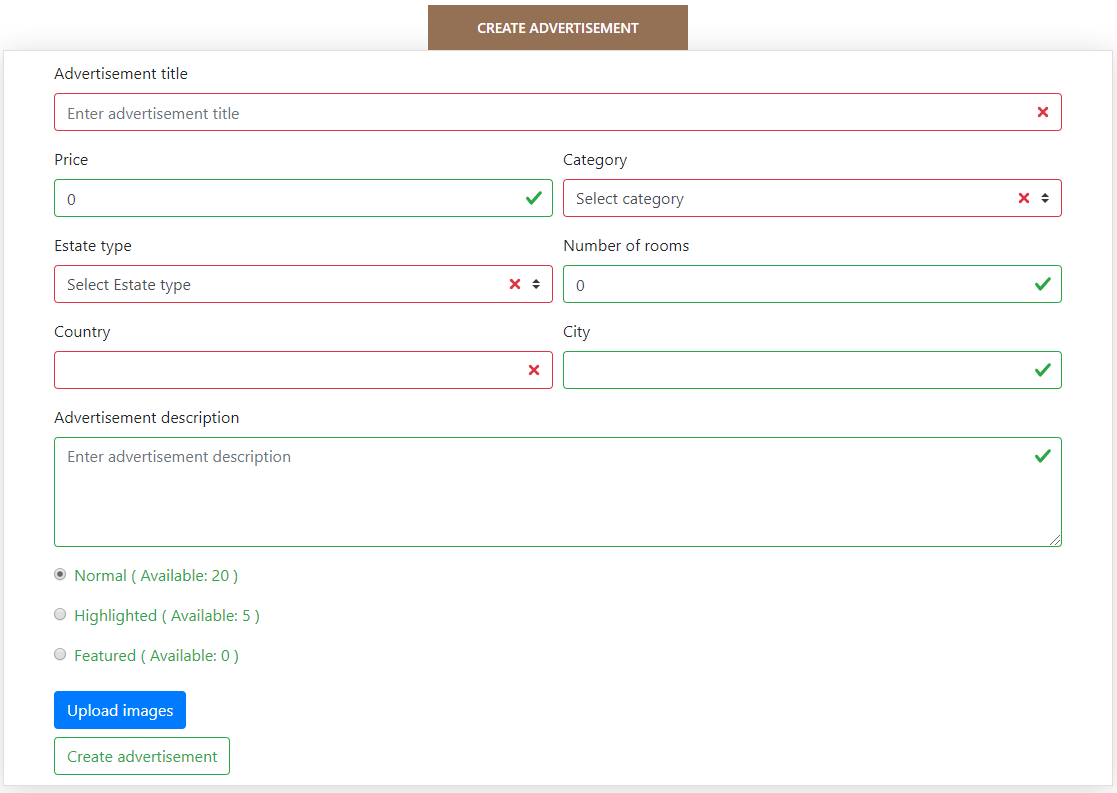
}

…

Siker esetén beállítja a felhasználó állapotát „bejelentkezett”-re, míg hiba esetén a szervertől kapott error kerül beállításra. Ezután a Reducer mindkét esetben módosítja a Store állapotát az új session adataival. A Store-ban történt állapotváltozás a komponens fa gyökerétől kezdve a felületen is érvényre jut. A React automatikusan újrarajzol minden komponenst, melynek „props” bemenetei módosultak. Újrarajzolásra kerül a bejelentkezési kérést eredetileg elindító MyAdvertisementsPage is, mely a megváltozott állapot miatt már a felhasználó saját hirdetéseit kezdi el betölteni, majd megjeleníteni.

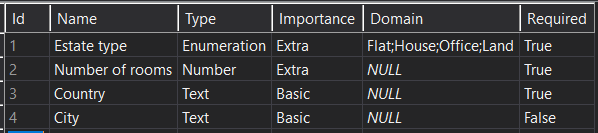
## Hirdetés feladása

A hirdetések feladása egy form-on keresztül történik, melynek kezeléséért a CreateAdvertisement komponens felelős. A form dinamikus mezőket is tartalmaz, melyeket a kliens az adatbázisban található metaadatok alapján rajzol ki. Ilyen mező az alábbi ábrán az Estate type, a Number of rooms, a Country és a City.



. ábra Hirdetés feladására szolgáló form dinamikus mezőkkel

A dinamikus mezők változó típusúak, lehetnek szöveges, szám vagy legördülő menüből választható értékek is. Az adatbázisban konfigurálható az is, hogy mely mezők kitöltése kötelező. Az Importance oszlop a keresésnél dönti el, hogy az adott mező az alap vagy a részletes keresésnél jelenjen meg, a Domain pedig listás értékeknél a választási lehetőségeket tartalmazza pontosvesszővel elválasztva.



. ábra A MetaDetails tábla, a fenti mezőkhöz tartozó értékekkel

A CreateAdvertisement komponens megjelenésekor kezdeményezi (Redux action-okon keresztül) a form megjelenítéséhes szükséges meta adatok és a lehezséges hirdetési kategóriák listájának betöltését. Szintén szükség van a felhasználó előfizetési adataira, hogy megjelenítsem, hogy milyen típusú kiemelésből hánnyal rendelkezik még a felhasználó. Ezeknek az adatoknak a betöltését a componentDidMount kezdeményezi.

public componentDidMount() {

this.props.getDetails();

this.props.getCategories();

this.props.getUserDetails(this.props.session.username);

}

A mezők renderelése a típusuk alapján történik. Az alkalmazás a szerverről letöltött meta detail-ek listáját a „props”-ból eltárolja a „state.advertisement” változó „details” tagjába. Az alábbi függvény ezen a listán megy végig, és minden részlethez a típusa alapján megfelelő mezőt teszi a „fields” tömbbe. A függvényt a CreateAdvertisement render() függvénye minden kirajzoláskor meghívja.

private renderCustomFields = () => {

let fields = [];

var details = this.state.advertisement.details;

for (let i = 0; i <= details.length - 1; i++) {

switch (details[i].type) {

case 0:

case 1:

fields.push(

<Col sm md="6" key={i}>

<Form.Group>

<Form.Label>{details[i].name}</Form.Label>

<input

type={details[i].type == 0 ? "text" : "number"}

className="form-control mr-sm-2 m-0"

onChange={e => this.handleChange(e, i)}

required={details[i].required}

/>

</Form.Group>

</Col>

);

break;

case 2:

fields.push(

<Col sm md="6" key={i}>

<Form.Group>

<Form.Label>{details[i].name}</Form.Label>

<select

required={details[i].required}

className="custom-select mr-sm-2 m-0"

onChange={e => this.handleChange(e, i)}>

<option value="">Select {details[i].name}</option>

{generateOptions(details[i].possibleValues)}

</select>

</Form.Group>

</Col>

);

break;

}

}

return fields;

};

A generateOptions() egy segédfüggvény, amely a kapott string tömb elemeiből <option> tag-eket generál. A különböző típusú mezőkhöz közös eseménykezelő tartozik, melynél kihasználom a TypeScript összetett típusait. A tömb megfelelő elemét a metaadat egyedi azonosítója alapján címzem meg.

private handleChange(

e: React.ChangeEvent<HTMLInputElement | HTMLSelectElement>,

id: number

) {

var advertisement = { ...this.state.advertisement };

if (+e.target.value) {

advertisement.details[id].numberValue = +e.target.value;

} else {

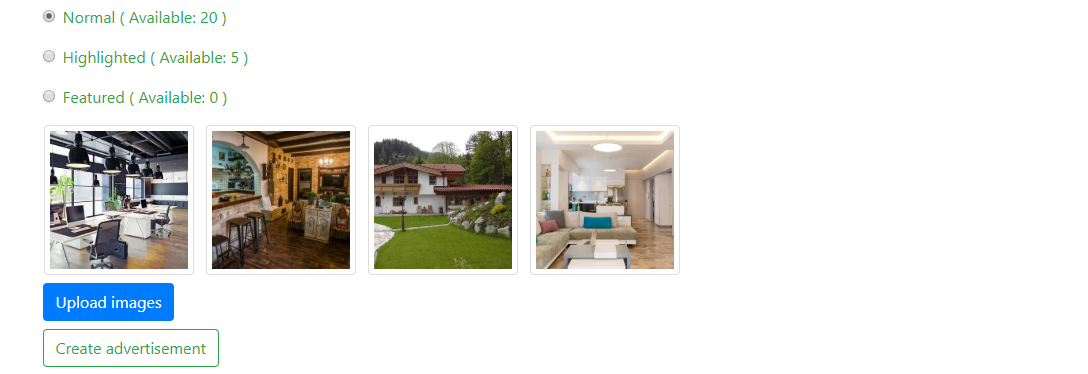
advertisement.details[id].stringValue = e.target.value;

}

this.setState({ advertisement });

}

A hirdetés feladásakor választhatunk kiemelési szintet, ahol az alkalmazás azt is jelzi, hogy az adott típusból még hány hirdetést engedélyez az előfizetésünk.



. ábra Kiemelés választása és képek feltöltése

A képek kiválasztásáról egy külső komponens, a React-image-picker [24] gondoskodik. A „Create Advertisement” gombra való kattintás és a validáció után a kliens a képeket egy IFormFile interfészű objektumba csomagolva az ImageController-nek, a hirdetés szöveges adatait pedig az AdvertisementController-nek küldi el.

Az ImageController az ImageService-nek, majd az az ImageRepository-nak adja át az feltöltött képet. Az ImageRepository azonos méretre vágja majd a felhasználó nevével azonosított mappába menti a képet. Névütközés esetén az alkalmazás egy számot fűz a fájlnév végére.

public async Task<string> SaveAsync(IFormFile file, string userName)

{

string fileName = Path.GetFileNameWithoutExtension(file.FileName);

string extension = Path.GetExtension(file.FileName);

var fullPath = path + userName + "/" + fileName + extension;

try

{

int i = 0;

while (File.Exists(fullPath))

{

i++;

fileName += i;

fullPath = path + userName + "/" + fileName + extension;

}

using (var stream = new FileStream(fullPath, FileMode.Create))

{

await file.CopyToAsync(stream);

}

ImageEditor.Crop(fullPath, 1420, 800);

}

catch (Exception e) { Debug.WriteLine("A fájl fogadása sikertelen: " + e.StackTrace); }

return fileName + extension;

}

A hirdetés többi adtának mentését az AdvertisementService és AdvertisementRepository végzi. Ennek része a szerver oldali validáció, felhasználó beállítása, a hirdetés létrehozási dátumának elmentése és az előfizetési csomag ellenőrzése is.

public async Task<AdvertisementDetailedDTO> CreateAdvertisement(string userName, CreateAdvertisementRequest request)

{

var advertisement = request.FromDTO();

…

bool success =

SubscriptionService.CheckSubscription(user, advertisement);

bool valid = await ValidateAsync(advertisement);

if (success && valid)

{

advertisement.Status = advertisement.Priority.ToStatus();

await AdvertisementRepository.CreateAsync(advertisement);

return advertisement.ToDetailedDTO();

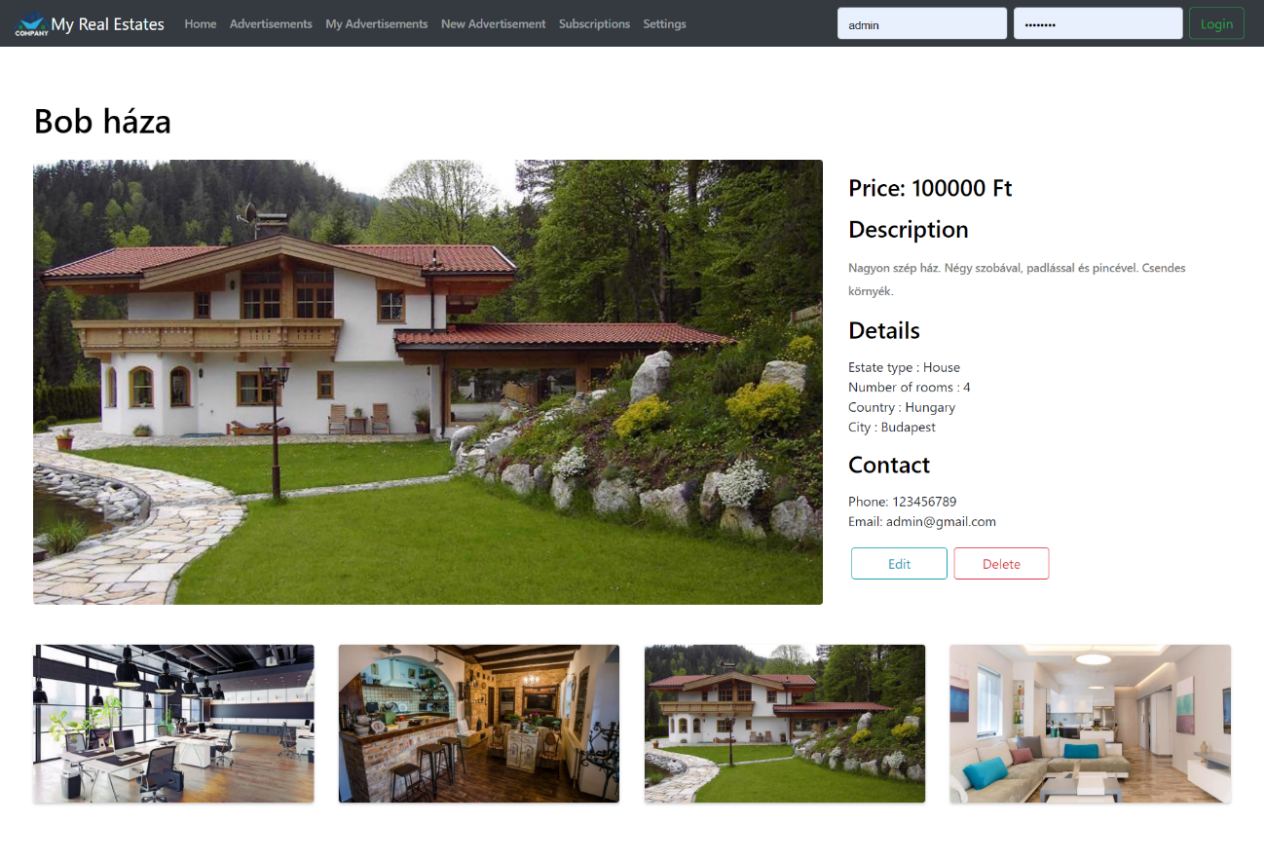
}

return null;

}

A hirdetés fix adattagjai az Advertisements táblába, a meta részletek a MetaDetails táblába, a képek pedig az Images táblába kerülnek.

A hirdetés feltöltése után a szerver visszaküldi az elkészült hirdetés adatait, melyek most már az Id-t is tartalmazzák. Ezután a kliens automatikusan a hirdetés részletes nézetéhez navigál.



. ábra Egy hirdetés részletes nézete

## Előfizetés módosítása

A felhasználók a Subscription oldalra navigálva láthatják jelenlegi előfizetésük részleteit és választhatnak az elérhető előfizetési modellek közül. A jelenlegi előfizetés adatait a SubscriptionDetails komponens jeleníti meg, míg a lehetséges előfizetéseket a SubscriptionTable rendereli az alábbi módon. A .map() függvény a content tömb (ez tartalmazza az előfizetési modelleket) minden eleméhez egy SubscriptionCard-ot rendel. A listás megjelenítéskor minden elemnek rendelkeznie kell egy egyedi azonosítóval, ezt a komponens id-jában adom át. A session változóra azért van szükség, mert a előfizetéseket mutató kártyákon a „Subscribe” gomb csak bejelentkezett felhasználók esetén jelenik meg. A komponens az onSubscribeClicked eseménykezelő meghívását a saját „props”-án keresztül továbbítja, így az esemény tényleges kezelése a SubscriptionPage-re hárul.

<div className="subscriptions">

{this.props.content.map(s => (

<SubscriptionCard

session={this.props.session}

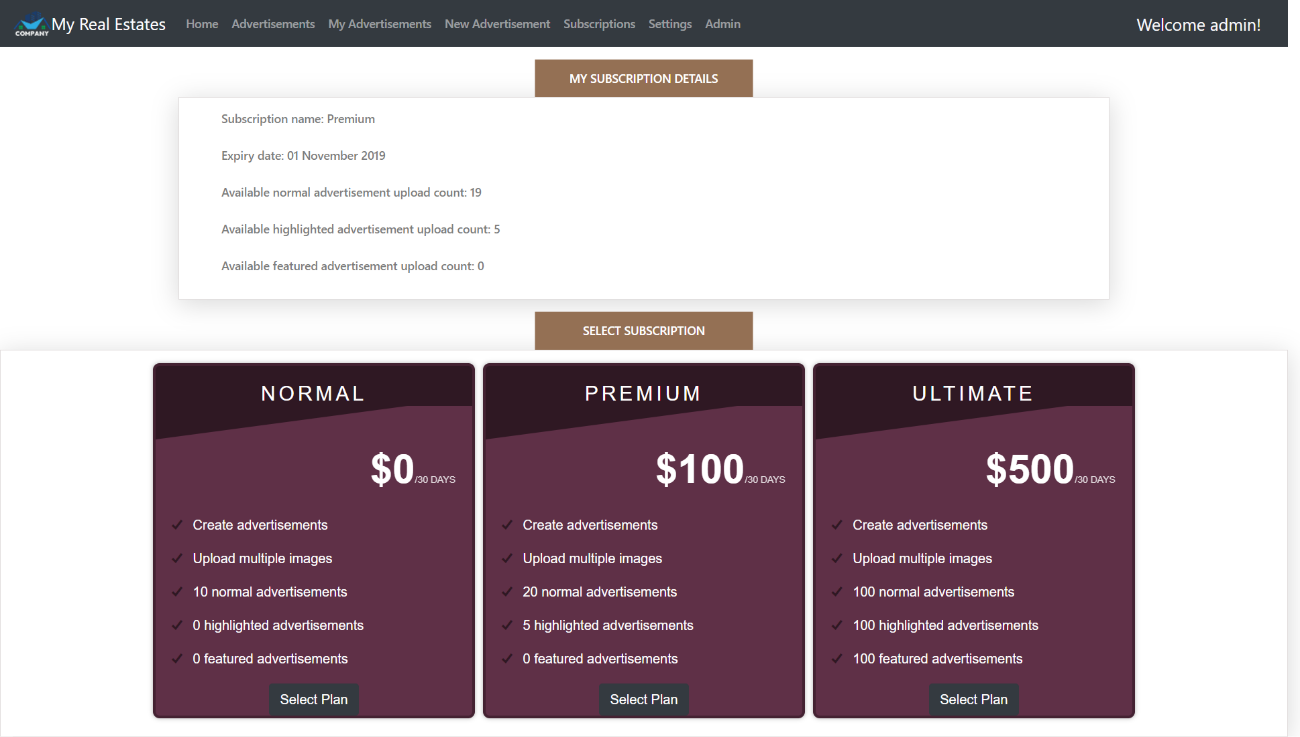
key={s.id}

subscription={s}

onSubscribeClicked={this.props.onSubscribeClicked}/>

))}

</div>



. ábra Előfizetések oldal megjelenése

A felhasználó jelenlegi előfizetésének részleteit a kliens SubscriptionController GetSubscriptionForUser() metódusától kérdezi le egy GET kéréssel. Előfizetés váltásakor a SubscriptionPage komponens aszinkron módon egy SubscriptionRequest-et küld a szervernek, majd újra betölti a megváltozott előfizetési adatokat.

subscriptionSelected = async (selected: SubscriptionModel) => {

if (this.props.session.status == UserStatus.LoggedIn) {

var request = { subscription: selected };

await sendSubscription(request);

}

await this.props.getUserDetails(this.props.session.username);

};

A POST kérés következtében a szerveren meghívódik a Subscribe() függvény. Ez a korábbiakhoz hasonlóan a JWT token alapján állapítja meg a kéréshez tartozó felhasználónevet, majd a SubscriptionService-hez továbbítja azt. Ez a későbbiekben ellenőrizheti a fizetés sikerességét (jelenleg egy dummy implementáció található meg), majd a SubscriptionRepository-n keresztül elmenti az új előfizetést az adatbázisba. Ezután a felhasználó összes eddigi hirdetésének kiemelését felül kell vizsgálni, hiszen, ha egy kisebb csomagra váltott, elképzelhető, hogy több hirdetése lenne kiemelve, mint amit az új előfizetése megenged. Az UpdateSubscriptionAsync() által meghívott UpdateAdvertisementsStatus() a felhasználó összes hirdetésén végigmegy, és közben nyilvántartja az eddigi kiemelések számát. Ha egy hirdetés kiemelése meghaladná a megengedett mennyiséget, akkor egyel alacsonyabb kiemelési szinten jelenik meg. A függvény végén az UpdateStatusAsync() hívás menti az adatbázisba a módosításokat.

private async Task UpdateAdvertisementsStatus(SubscriptionModel model,

IEnumerable<Advertisement> advertisements)

{

int featured = 0;

int highlighted = 0;

int normal = 0;

foreach (var adv in advertisements){

switch (adv.Priority){

case Priority.Featured:

// Csökkenő sorrendben próbáljuk a kiemeléseket

if (featured < model.FeaturedAdvertisementCount){

adv.Status = Status.Featured;

featured++;

break;

}

if (highlighted < model.HighlightedAdvertisementCount){

adv.Status = Status.Highlighted;

highlighted++;

break;

}

if (normal < model.NormalAdvertisementCount){

adv.Status = Status.Displayed;

normal++;

break;

}

adv.Status = Status.Hidden;

break;

case Priority.Highlighted:

…

// Az előzőhöz hasonlóan csökkenő sorrendben próbáljuk a kiemeléseket,

most a „Highlighted”-tól kezdve

case Priority.Normal:

…

// Ha „Normal” hirdetésként sem jelenhet meg, akkor elrejtjük

...}

}

// Változtatások mentése az adatbázisba

await AdvertisementRepository.UpdateStatusAsync(advertisements);

}

A kliens oldalon a sikeres feliratkozás után a felhasználó hirdetésének adatai automatikusan módosulnak. Ehhez a SubscriptionPage komponens az aszinkron feliratkozás megvárása (await) után egy újabb kérést indít a felhasználó előfizetési részleteinek lekéréséhez az alábbi módon.

subscriptionSelected = async (selected: SubscriptionModel) => {

if (this.props.session.status == UserStatus.LoggedIn) {

var request = { subscription: selected };

await sendSubscription(request);

}

await this.props.getUserDetails(this.props.session.username);

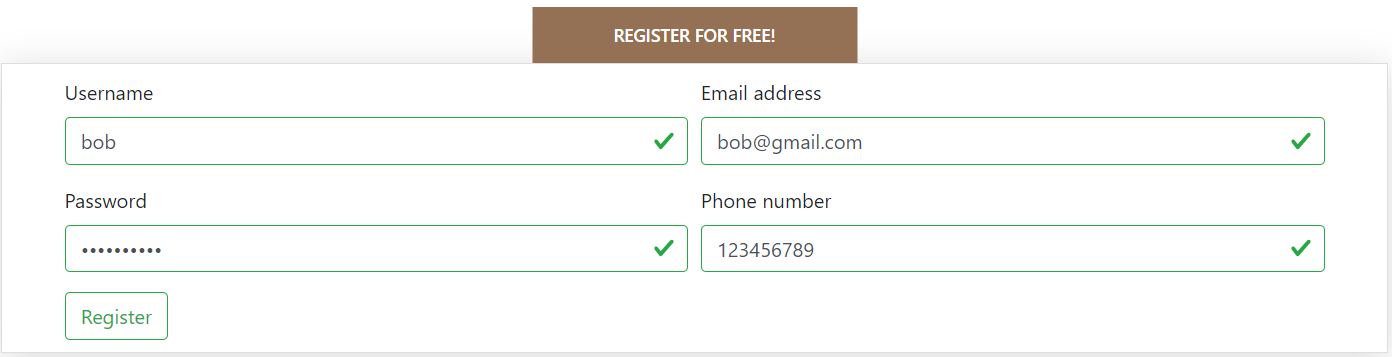
};

## További funkciók

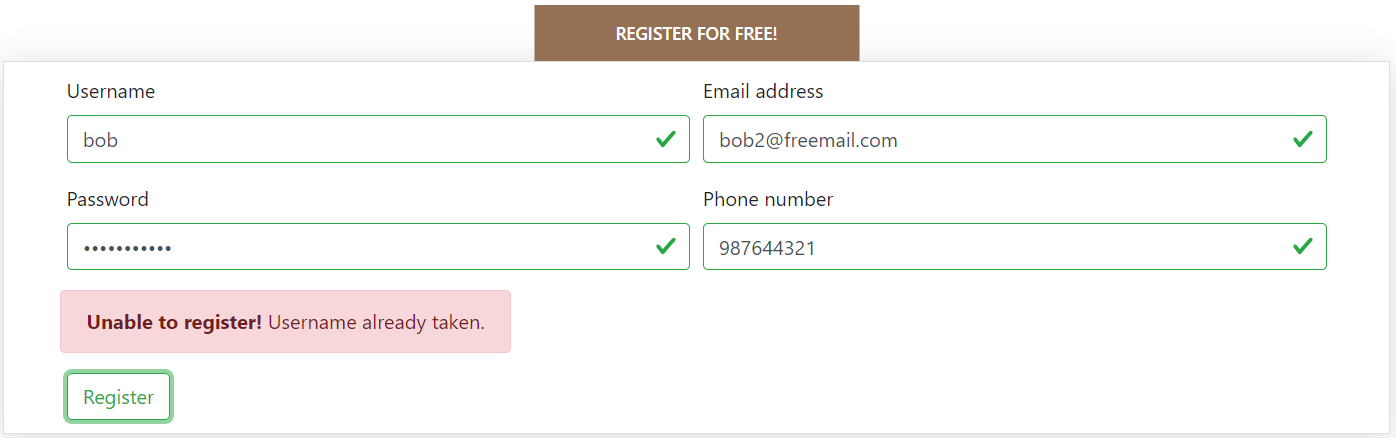
A további funkciókat röviden, képernyőképekkel illusztrálva fogom bemutatni.

### Regisztráció

A regisztrációkor felhasználónevet, jelszót, e-mail címet és telefonszámot kell megadnunk. Utóbbi kettő a feltöltött hirdetésekhez tartozó elérhetőségeknél is megjelenik. A form validációt is használ, kliens oldalon jelzi például, ha az e-mail cím vagy a jelszó nem megfelelő formátumú, szerver oldalon pedig a felhasználónév ütközésekor jelez hibát, amely a UI-on is megjelenik.



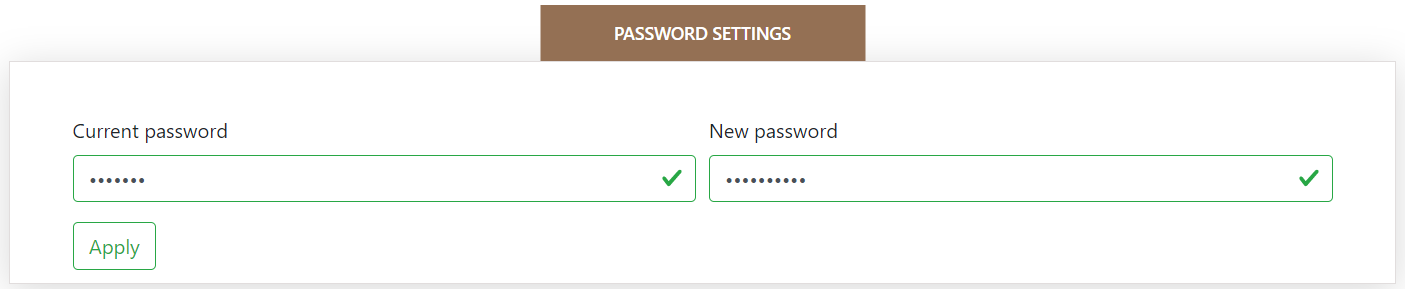
. ábra A regisztrációs form



. ábra Hibaüzenet felhasználónév ütközésekor

### Felhasználói beállítások

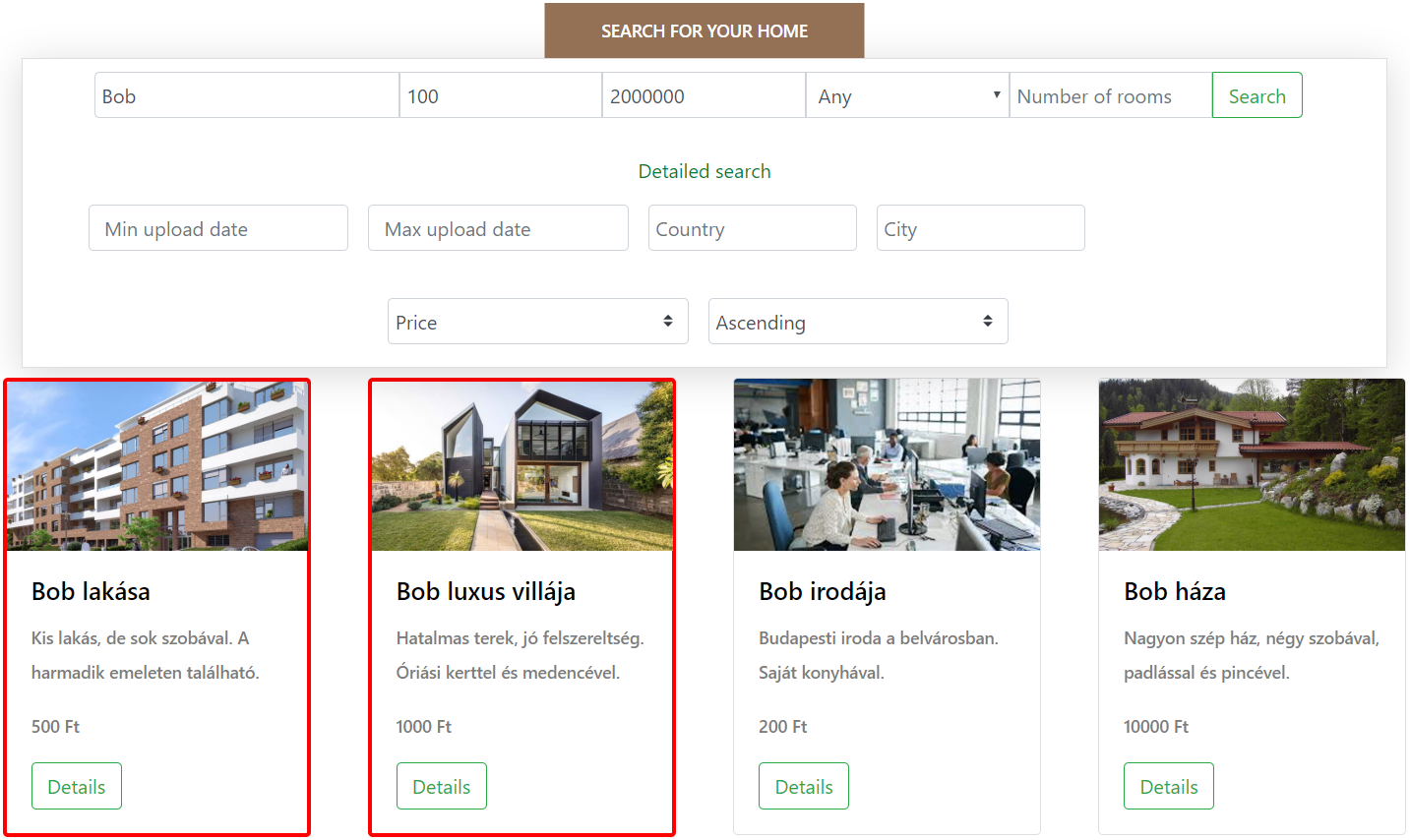
A Settings oldalon lehetőség van a jelszavunk megváltoztatására vagy a felhasználói fiókunk törlésére.



. ábra Lehetőség a jelszó megváltoztatására

### Hirdetések listázása és keresése

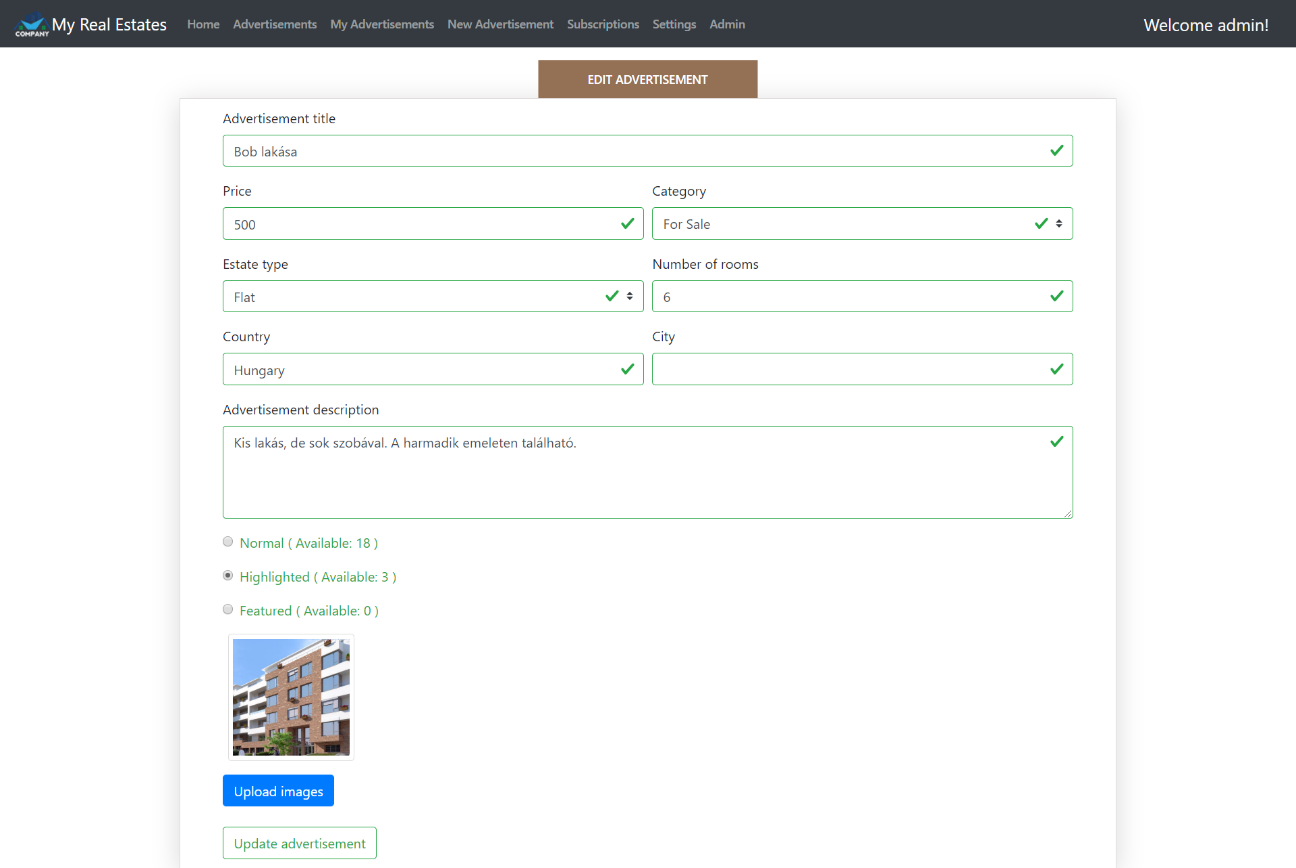
Hirdetések listázásakor részletes keresési paramétereket adhatunk meg. A Detailed search rész lenyitásakor újabb paramétereket állíthatunk be. Listázáskor megadhatjuk a sorbarendezés módját is. A kiemelt hirdetések ilyenkor is a lista elején jelennek meg.



. ábra Példa a hirdetések közti keresésre

### Hirdetések módosítása

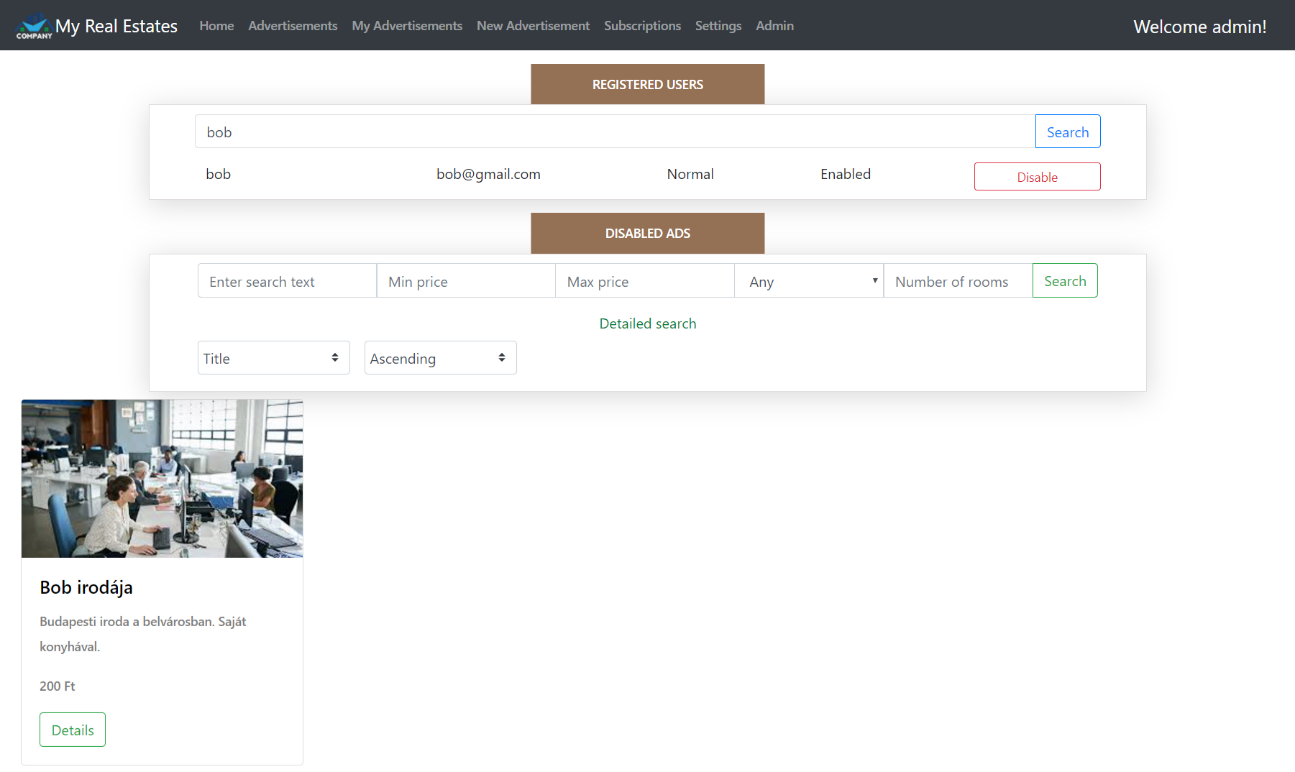
A módosítás egy a hirdetés feladásához hasonló felületen történik. Oda navigálni a saját hirdetéseket tartalmazó oldalról egy hirdetésen az „Edit” gomb megnyomásával lehet.



. ábra Hirdetés módosítása

### Adminisztrátori funkciók

Az admisztrátor listázhatja a felhasználókat és le is tilthatja őket. Az adminisztrátor egyes hirdetéseket is letilthat vagy újra megjelenítheti az eddig letiltottakat. Az alábbi példában a bob nevű felhasználóra kerestünk rá. A letiltott hirdetések között most csak egy szerepel: Bob irodája.



. ábra Az adminisztrátori felület

## Biztonság

Az elkészített hirdetési portál személyes adatokat is tárol, így ezek biztonsága kritikus szempont. A felhasználók azonosítására egy egyedi felhasználónevet, az authorizációhoz pedig egy jelszót használok. Az ASP.Net Core rengeteg lehetőséget biztosít a jelszókövetelmények finomhangolására. Jelen esetben az alkalmazás egy legalább 6 karakterből álló, kis és nagybetűt is tartalmazó jelszót vár el. Ennek konfigurálását a Startup metódusban végzem el.

//Jelszó követelmény beállítások

services.Configure<IdentityOptions>(options =>

{

options.Password.RequireDigit = true;

options.Password.RequiredLength = 6;

options.Password.RequireNonAlphanumeric = false;

options.Password.RequireUppercase = true;

options.Password.RequireLowercase = true;

});

A kliens oldali regisztrációnál is validálom a megadott jelszót, ekkor regex kifejezést használok.

pattern="^(?=.\*\d)(?=.\*[a-z])(?=.\*[A-Z])(?!.\*\s).\*$"

Az adatbázisban a jelszavak hash-ét tárolom el, így még ha valaki hozzáfér is a felhasználókat tartalmazó táblához, akkor sem tud illetéktelenül bejelentkezni semelyik felhasználó nevében.

A csak bejelentkezés után elérhető funkciók védelméről az ASP.Net [Authorize] attribútuma gondoskodik. Az implementáció korai szakaszában a kéréshez tartozó felhasználó nevét path vagy body paraméterben adtam át a szervernek. Ez azonban biztonsági szempontból hátrányosnak bizonyult, hiszen így bármely bejelentkezett felhasználó meg tudott személyesíteni más felhasználókat egy kézzel vagy külső eszközzel összeállított http kéréssel. Ennek elkerülésére a szerver oldalon a kérést küldő felhasználó nevét egy JWT tokenből állapítom meg. A JWT token a szerver privát kulcsával van titkosítva, és dekódolása is csak a privát kulcs ismeretében lehetséges. A token egy adott bejelentkezési session-höz tartozik, és csak korlátozott ideig érvényes. Lejárata után a felhasználónak ismét be kell jelentkeznie. A JWT token generálását a bejelentkezés során a LoginService osztály végzi. Ehhez felhasználja a felhasználó nevét és típusát is.

var claims = new List<Claim>

{

new Claim(JwtRegisteredClaimNames.Sub, name),

new Claim(JwtRegisteredClaimNames.Typ, type.ToString()),

new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),

new Claim(ClaimTypes.NameIdentifier, name)

};

var token = new JwtSecurityToken(issuer, audience, claims, null,

expires, creds);

A tokent a kliens eltárolja és minden kérésnél Header mezőben visszaküldi a szervernek. A szerveren a felhasználónév és a felhasználó típusának token-ből történő megállapítását egy Controller ősosztályba szerveztem ki.

protected string UserName

{

get

{

return User.FindFirst("sub")?.Value;

}

}

protected string UserType

{

get

{

return User.FindFirst("typ")?.Value;

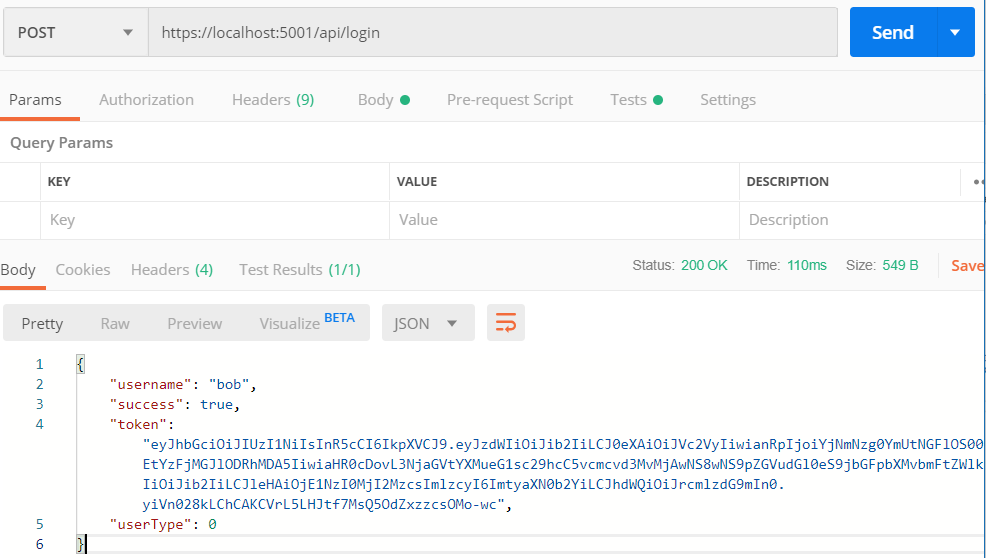
}

}

A kliens szerver kommunikációt https felett valósítom meg. A https a http és a TCP szintek közé egy SSL [24] vagy TLS réteg beiktatásával titkosítja az átvitt üzenetet, így az a hálózaton nem hallgatható le.

## Tesztelés

A fejlesztés korai fázisában a szerver teszteléséhez a Postman alkalmazást használtam. A Postman-ben kézzel állíthatunk össze http kéréseket, melyeknek megadhatunk query paramétereket, authorizációs tokent és body tartalmat is. A Postman elküldi a kérést egy megadott URL-re és formázottan jeleníti meg a válaszüzenetet. Egy bejelentkezési kérésre kapott válasz az alábbi képen látható.



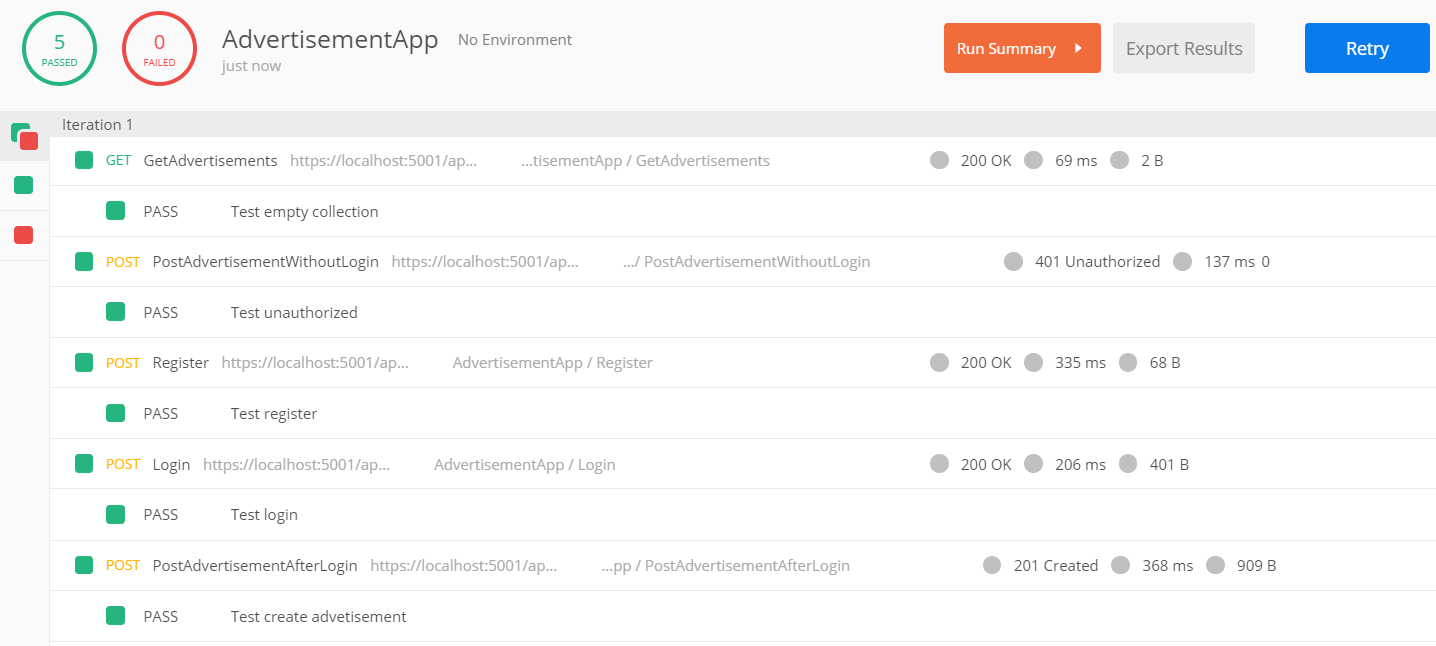
. ábra Sikeres bejelentkezés Postman-ből

A Postman lehetőséget nyújt, hogy http kéréseinket gyűjteményekbe rendezhessük, és ezeket automatikusan egymás után futtathassuk. Ez jelentősen meggyorsítja a tesztelést, ha az alkalmazásunk bejelentkezéskor egy tokent ad vissza, melyet a többi kérésben fel akarunk használni. A Postman ezt a tokent képes egy változóban eltárolni és a későbbi kérések Header mezőjében elküldeni.



. ábra A korábban eltárolt „token” változó felhasználása a header-ben

A Postman egyszerűbb automatizált tesztek futtatására is képes. Ehhez JavaScript nyelven fogalmaztam meg teszteseteket olyan alapvető funkciók tesztelésére, mint a hirdetések listázása, a regisztráció, a bejelentkezés és egy hirdetés feladása. A tesztek részletes leírása a függelék fejezetben található. A tesztek sikeres lefutását az alábbi kép mutatja.



. ábra Automatizált tesztek lefutása

Az elkészült alkalmazás működését a UI-on keresztül is teszteltem. Ehhez szükség esetén kiírattam bizonyos változók értékét a konzolra vagy brakepoint-okat helyeztem el a szerver kódjában. A tesztelés során talált hibákat javítottam.

# Összefoglalás

## Tapasztalatok

A hirdetési webalkalmazás elkészítése során rengeteg új technológiát és architekturális mintát ismertem meg. A projekt elején teljesen új volt számomra a React-Redux keretrendszer, melyet mostanra nagy magabiztossággal használok kliens oldali fejlesztések során. A szakmai gyakorlatom egy része alatt is egy React frontend alkalmazás fejlesztésén dolgoztam, ahol megismertem a React 16.8-as verziójával megjelent Hooks-ot. A React Hooks lehetővé teszi function komponensekben az állapottárolást, mellyel sok, a JavaScript „this” helytelen használatából eredő hibát kiszűr. Tapasztalataim alapján a Hooks nem alkalmas a Redux állapottárolás lecserélésére, inkább kiegészíti azt.

A fejlesztés során jó döntésnek bizonyult a TypeScript nyelv választása. Ez közelebb áll az általam korábban megszokott C# erősen típusos gondolkodásmódjához, és a hibák nagyrészét még fordítási időben felismeri. A típusok használata csupán egyetlen esetben bizonyult hátrányosnak, ugyanis a React <input/> és a React-bootstrap által definiált <Form.Control/> eseménykezelője habár ugyanazt a feladatot látja el, mégis más szignatúrával rendelkezik. Ez megnehezíti a két komponens egységes kezelését.

A szakdolgozatom kitűzött céljait sikerült elérnem, elkészültem az alkalmazás implementációjával. Az újonnan megismert technológiák mellett a .Net Core keretrendszerrel kapcsolatos tudásomat is tovább fejlesztettem és most már sokkal jobban átlátom egy elsőre bonyolultnak tűnő alkalmazás architektúráját is. Úgy gondolom, hogy alkalmazásom továbbfejlesztett változata éles környezetben is megállná a helyét.

## Továbbfejlesztési lehetőségek

Az alkalmazás későbbi továbbfejlesztésére több lehetőség is felmerült. Ezek némelyike technológiai, míg mások funkcionális jellegűek.

A nagy adatmennyiség és a félig strukturált adatformátum miatt teljesítmény szempontjából érdemes lenne valamilyen NoSQL [24] adatbázismotor alkalmazása. Az ilyen adatbázisok gyakran nem táblákban, hanem JSON dokumentumokban tárolják az adatokat. Habár a szakmai gyakorlatom során megismerkedtem a MongoDB [25] adatbáziskezelő használatának és lekérdezési lehetőségeinek alapjaival, az Entity Framework támogatás hiánya miatt mégsem azt választottam az alkalmazás adatainak perzisztens tárolására. Az adatbázis lecserélése az alkalmazás üzleti logikájának szempontjából akár teljesen transzparens módon is történhet, a Repository tervezési minta alkalmazásának köszönhetően. A MongoDB mellett egy másik lehetőség az Elastic Search használata, amely nagy adatbázisokban teszi lehetővé a gyors és skálázható keresést. Egy memóriabeli cache alkalmazása szintén meggyorsíthatja a gyakran használt műveleteket (pl.: a főoldali hirdetések listázása). Erre nyújt megoldást a nyílt forrású Redis [26].

Az alkalmazás számtalan további funkcióval egészíthető ki. Ezek közül a legkiemelkedőbbek:

* Egy külső fizetési szolgáltatás (pl.: PayPal) integrálása.
* Hirdetésfigyelő rendszer megvalósítása.
* Mesterséges intelligencia által javasolt, személyre szabott hirdetések.
* Korábban megtekintett hirdetések eltárolása, megjelenítése.
* Hirdetési helyszínek megadása, keresés közelség alapján

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | „.NET Core,” [Online]. Available: https://en.wikipedia.org/wiki/.NET\_Core. [Hozzáférés dátuma: 01 05 2019]. |
| [2] | T. Y. Todorov, „Understanding .NET Just-In-Time Compilation,” [Online]. Available: https://www.telerik.com/blogs/understanding-net-just-in-time-compilation. [Hozzáférés dátuma: 01 05 2019]. |
| [3] | B. Luijbregts, „.NET Standard Explained: How to Share Code,” [Online]. Available: https://dzone.com/articles/net-standard-explained-how-to-share-code. [Hozzáférés dátuma: 01 05 2019]. |
| [4] | S. Singh, „Understanding Asynchronous Programming with Async and Await in C#,” [Online]. Available: http://blogs.quovantis.com/asynchronous-programming-async-and-await-in-c/. [Hozzáférés dátuma: 01 05 2019]. |
| [5] | R. A. a. S. L. Daniel Roth, „Introduction to ASP.NET Core,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2. [Hozzáférés dátuma: 01 05 2019]. |
| [6] | W. Japan, „Create a Web Application for Performing CRUD Operations Using JavaScript and Web API,” [Online]. Available: https://www.grapecity.com/blogs/create-web-application-performing-database-operations-using-javascript-web-API. [Hozzáférés dátuma: 01 05 2019]. |
| [7] | S. A. a. L. L. Steve Smith, „Dependency injection in ASP.NET Core,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2. [Hozzáférés dátuma: 01 05 2019]. |
| [8] | „Architectural principles - Dependency Inversion,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles#dependency-inversion. [Hozzáférés dátuma: 01 05 2019]. |
| [9] | „Your JavaScript is TypeScript,” [Online]. Available: https://basarat.gitbooks.io/typescript/docs/javascript/recap.html. [Hozzáférés dátuma: 01 05 2019]. |
| [10] | J. Cardillo, „The Difference Between Function and Block Scope in JavaScript,” [Online]. Available: https://medium.com/@josephcardillo/the-difference-between-function-and-block-scope-in-javascript-4296b2322abe. [Hozzáférés dátuma: 01 05 2019]. |
| [11] | „ReactJS,” [Online]. Available: https://reactjs.org/. [Hozzáférés dátuma: 01 05 2019]. |
| [12] | M. Sargent, „An Introduction to the Redux-First Routing Model,” [Online]. Available: https://medium.freecodecamp.org/an-introduction-to-the-redux-first-routing-model-98926ebf53cb. [Hozzáférés dátuma: 01 05 2019]. |
| [13] | M. V. d. Bergh, „React Redux: Building Modern Web Apps with the ArcGIS JS API,” [Online]. Available: https://www.esri.com/arcgis-blog/products/js-API-arcgis/3d-gis/react-redux-building-modern-web-apps-with-the-arcgis-js-API/. [Hozzáférés dátuma: 01 05 2019]. |
| [14] | „Axios - Promise based HTTP client for the browser and node.js,” [Online]. Available: https://github.com/axios/axios. [Hozzáférés dátuma: 01 05 2019]. |
| [15] | T. Mostert, „JavaScript Call-back Functions, Promises and Async/Await,” [Online]. Available: https://www.codementor.io/theresamostert/javascript-call-back-functions-promises-and-async-await-cvhdop4fo. [Hozzáférés dátuma: 01 05 2019]. |
| [16] | A. Purdila, „Bootstrap 4 is Here! What’s New?,” [Online]. Available: https://webdesign.tutsplus.com/articles/bootstrap-4-is-here-whats-new--cms-30442. [Hozzáférés dátuma: 29 04 2019]. |
| [17] | M. Fowler, „Bounded Context,” [Online]. Available: https://martinfowler.com/bliki/BoundedContext.html. [Hozzáférés dátuma: 01 05 2019]. |
| [18] | „Repository Pattern,” [Online]. Available: https://deviq.com/repository-pattern/. [Hozzáférés dátuma: 01 05 2019]. |
| [19] | linko-crypto, „How to use Repository pattern in Laravel 5,” [Online]. Available: https://steemit.com/php/@linko-crypto/how-to-use-repository-pattern-in-laravel-5. [Hozzáférés dátuma: 01 05 2019]. |
| [20] | „API-tervezés,” Microsoft, [Online]. Available: https://docs.microsoft.com/hu-hu/azure/architecture/best-practices/API-design. [Hozzáférés dátuma: 26 09 2019]. |
| [21] | S. Bageri, „Create and Implement 3-Tier Architecture in ASP.Net,” [Online]. Available: https://www.c-sharpcorner.com/UploadFile/4d9083/create-and-implement-3-tier-architecture-in-Asp-Net/. [Hozzáférés dátuma: 2019.09.26]. |
| [22] | „Introduction to JSON Web Tokens,” [Online]. Available: https://jwt.io/introduction/. [Hozzáférés dátuma: 26 09 2019]. |

Függelék

**Az automatizált tesztesetek leírása:**

[1] Az összes hirdetés listázása

GET <https://localhost:5001/api/advertisements>

Kezdetben nincs hirdetés, így egy üres listát várunk el.

pm.test("Test empty collection", function(){

pm.response.to.have.status(200);

var jsonData = pm.response.json();

pm.expect(jsonData.length).eql(0);

})

[2] Hirdetés feladása bejelentkezés nélkül

POST <https://localhost:5001/api/advertisements>

BODY: paraméterként egy példa hirdetés adatait adjuk meg. [[5]](#endnote-1)

Nincs jogosultságunk hirdetés feladására, így „Unauthorized” választ várunk.

pm.test("Test unauthorized", function () {

pm.response.to.have.status(401);

});

[3] Regisztráció

POST <https://localhost:5001/api/users>

BODY

{

"userName": "bob",

"password": "Pass123456",

"email": "bob@gmail.com",

"phone": "123456789"

}

Sikeres regisztrációt várunk el, a megadott adatokkal.

pm.test("Test register", function(){

pm.response.to.have.status(200);

var jsonData = pm.response.json();

pm.expect(jsonData.userName).eql("bob");

pm.expect(jsonData.email).eql("bob@gmail.com");

pm.expect(jsonData.phoneNumber).eql("123456789");

});

[4] Bejelentkezés

POST <https://localhost:5001/api/login>

BODY

{

"userName": "bob",

"password": "Pass123456"

}

Sikeres bejelentkezés után eltároljuk a token-t a további kérésekhez.

pm.test("Test login", function(){

pm.response.to.have.status(200);

var jsonData = pm.response.json();

pm.expect(jsonData.success).eql(true);

pm.expect(jsonData.userType).eql(0);

pm.environment.set("token", jsonData.token);

})

[5] Hirdetés feladása bejelentkezés után, token használatával

POST <https://localhost:5001/api/advertisements>

BODY: paraméterként egy példa hirdetés adatait adjuk meg. i

**A hirdetés feladása sikeres, a felhasználónév megegyezik a bejelentkezett felhasználó típusával és a hirdetés az 1. ID-t kapta.**

**pm.test("Test create advetisement", function(){**

**pm.response.to.have.status(201);**

**var jsonData = pm.response.json();**

**pm.expect(jsonData.id).eql(1);**

**pm.expect(jsonData.userName).eql("admin");**

**pm.expect(jsonData.title).eql("Admin háza");**

**})**

1. A Bounded Context lényege, hogy a Domain-t több kisebb részegységre bontjuk, melyek önmagukban is megérthetőek, újra felhasználhatóak. [↑](#footnote-ref-1)
2. Data Transfer Object, vagy magyarul Adatátviteli objektum. Olyan objektum, amely csak getter és setter függvényekkel rendelkezik és célja a hálózaton keresztüli adatátvitel. [↑](#footnote-ref-2)
3. A TypeScript megengedi, hogy egy típusra több névvel is hivatkozhassunk, így az összetett és template típusok rövidebben jeleníthetőek meg. [↑](#footnote-ref-3)
4. A React komponensek szülőktől kapott bemeneteit props-nak nevezzük. [↑](#footnote-ref-4)
5. Egy példa hirdetés felépítése JSON formátumban.

   {

   "categoryId": 1,

   "description": "Szép ház.",

   "details": [

   {

   "id": 1,

   "name": "Estate type",

   "stringValue": "House",

   "numberValue": 0,

   "type": 2

   },

   {

   "id": 2,

   "name": "Number of rooms",

   "stringValue": "",

   "numberValue": 4,

   "type": 1

   },

   {

   "id": 3,

   "name": "Country",

   "stringValue": "Hungary",

   "numberValue": 0,

   "type": 0

   },

   {

   "id": 4,

   "name": "City",

   "stringValue": "Budapest",

   "numberValue": 0,

   "type": 0

   }

   ],

   "image": "house1.jpg",

   "imageList": [

   "house1.jpg"

   ],

   "price": 500,

   "priority": 0,

   "title": "Admin háza"

   } [↑](#endnote-ref-1)