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

Berta Balázs

FOGLALÁSI RENDSZER MEGVALÓSÍTÁSA .NET CORE ÉS ANGULAR KÖRNYEZETBEN

KONZULENS

Benedek Zoltán

BUDAPEST, 2019

Tartalomjegyzék

Összefoglaló **5**

Abstract **6**

1. Bevezetés **7**

2. Technológiák bemutatása **9**

2.1 JavaScript 9

2.1.1 TypeScript 10

2.2 Angular (Angular 2+) 10

2.2.1 Felépítés 10

2.2.2 Modulok 10

2.2.3 Komponensek 12

2.2.4 Sablon szintaxis és adatkötés 12

2.2.5 Szolgáltatások és függőség injektálás 13

2.2.6 Működés 14

2.3 Entity Framework Core 14

2.4 ASP.NET Core 15

2.4.1 Modell-nézet-vezérlő minta 16

2.4.2 Web API 16

2.4.3 Single-page application Middleware 17

2.4.4 SignalR 19

2.5 AutoMapper 21

2.6 Skálázható Vektor Grafika (SVG) 21

2.7 Bootstrap 22

2.8 xUnit.net 23

3. Tervezés **25**

3.1 Alkalmazás tervének bemutatása 25

3.2 Szerepkörök bemutatása 25

3.2.1 Normál felhasználó 26

3.2.2 Ügyintéző felhasználó 26

3.3 Nem-funkcionális követelmények 27

3.4 Architektúra bemutatása 27

3.4.1 Projektek 29

3.4.2 Adatbázis terv 30
4. Megvalósítás 32
4.1 Adatelérési réteg (DAL) 32
4.2 Üzleti logikai réteg (BLL) 34
4.3 Ügyintézői portál 36
4.3.1 Felépítés 36
4.3.2 Felhasználókezelés 38
4.3.3 Működés 38
4.4 Foglaló portál 43
4.4.1 Felépítés 44
4.4.2 Működés 47
4.5 Tesztelés 59
5. Összefoglalás 61
5.1 Elvégzett munka 61
5.2 Munkám értékelése, tapasztalatok 61
5.3 Továbbfejlesztési lehetőségek 62
6. Hivatkozások 63
7. Utolsó simítások 65

HALLGATÓI NYILATKOZAT

Alulírott **Berta Balázs**, szigorló hallgató kijelentem, hogy ezt a diplomatervet 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(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2019. 11. 21.

...…….
Berta Balázs

Összefoglaló

Ide jön a ½-1 oldalas magyar nyelvű összefoglaló, melynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

Abstract

Ide jön a ½-1 oldalas angol nyelvű összefoglaló, amelynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

1. Bevezetés

A modern, 21. századi társadalomban a mindennapi élet széles területein általánossá vált az online számítógépes szoftverek alkalmazása, mind a munkavégzés, mind a szórakozás terén. Napjainkban egyre gyakoribb az igény olyan szoftverek megvalósítására, ahol az alkalmazás és a felhasználók között azonnali, élő kommunikáció van. Ennek köszönhetően a munkavégzés gyorsabb és hatékonyabb lehet, azonnali visszajelzést kapunk az általunk végzett cselekményre.

A diplomatervem során egy foglalási rendszer megvalósítását tűztem ki célul. Ez magában foglalja mind az adatok online kezelését, adminisztrációs feladatok ellátását egy szűkebb felhasználói kör számára, akik valamilyen szolgáltatást nyújtanak. Ezen felül az átlagos felhasználók számára egy egyszerű és kényelmes foglalási webportált.

A diplomatervi témaválasztásom szakterületét illetően egyértelműen befolyásolt az érdeklődés a kliens oldali technológiák, illetve az élő kommunikáció megvalósítására szolgáló eszközök iránt, és inspirált a tanulmányaim során megszerzett ismeretanyag.

A foglalási rendszer felületéhez egy olyan interaktív webalkalmazást készítettem el, amely összeköti az adminisztrációs oldalon kezelhető adatokat a felhasználókkal. Ezen felül a foglalás során valós időben lehet megtekinteni a mások által kiválasztott, illetve már lefoglalt helyeket.

A dolgozatom első részében bemutatom az általam felhasznált technológiákat és az elkészült szoftver architektúráját. Komplex webalkalmazás lévén számos külső eszközt felhasználtam a fejlesztés során, egyszerűsítve a munkámat. A dolgozatom második részében az általam megvalósított szoftver, illetve az elvégzett munka bemutatása történik.

A munkámat a jelenleg használt kliens oldali technológiák megismerésével kezdtem, és ezek közül választottam azt, amelyik számomra legmegfelelőbbnek tűnt az alkalmazás megvalósítására. Választásom az Angular keretrendszerre [1] esett, mivel kimagasló a teljesítménye a webes platformon, ezen felül részletesen kidolgozott dokumentációval és példákkal rendelkezik. Felhasználtam még a .NET Standard [2] keretrendszerre épülő Entity Framework Core [3] technológiát is, ami az adatok tárolását segíti relációs adatbázisban.

A szerverem megvalósításához az ASP.NET Coret választottam, mivel ez egy cross-platform[1] webfejlesztési keretrendszer, ami jól integrálódik az Entity Frameworkhöz és népszerű a fejlesztők között. Ezen felül az élő kommunikációhoz használt SignalR [4] technológia is a részét képezi.

A követelmények részletezése után ábrák segítségével szemléltetem a rendszereim architektúráját és az adatbázis tervét. Ezt követően bemutatom az alkalmazásom működését a legfontosabb használati eseteket megvalósító implementációk részletezésével. Az elvégzett munka részét képezik a szoftverhez készült egység tesztek is.

Zárásként értékelem, hogy milyen tapasztalatokat sikerült szereznem a dolgozat megírása során és bővebben kitérek arra, hogyan lehetne tovább fejleszteni az elkészült alkalmazásomat.

[1] A keresztplatformos szoftverfejlesztés egy olyan folyamat, amikor egy alkalmazás egyszerre több platformra készül el.

2. Technológiák bemutatása

Dolgozatom e fejezetében szeretném részletesebben bemutatni az általam felhasznált technológiákat. A mai rendszerek komplexitásai miatt szükségszerű, hogy ne mindent a legkisebb elemektől kezdjünk el felépíteni, hanem felhasználjuk a már létező megoldásokat, eszközöket. A webfejlesztés a szoftverfejlesztés egy nagy ága, melyhez már számos segédeszköz született, melyeket modulszerűen felhasználva tudunk új szoftvereket létrehozni.

2.1 JavaScript

A JavaScript (röviden JS), egy magas szintű, interpretált[2] programozási nyelv, mely megfelel az ECMAScript[3] specifikációjának. A HTML[4] és CSS[5] mellett a webfejlesztés egyik magja. Célja gazdag és interaktív alkalmazások készítése.

Támogatja az esemény alapú, funkcionális, illetve az imperatív programozási stílusokat. Eszközöket nyújt alap típusokkal való munkára, úgymint szövegek, tömbök, dátumok, reguláris kifejezések[6] és a DOM[7] kezelésére is ad megoldásokat. Azonban a nyelv önmagában nem tartalmaz I/O, hálózati, tárolási, illetve grafikai eszközöket, ezekhez a host eszköz beépített szolgáltatásait használja fel. A JavaScript programozási nyelv dinamikusan típusos.

A webfejlesztés egyik nagy kihívása volt, hogy a különböző fajta böngészők, máshogy oldottak meg egyes funkciókat a saját megvalósításaikban. Mára minden modern böngésző támogatja a JavaScript beépített interpreterrel.

---

[2] Az interpreter (program) futtatókörnyezetként viselkedik, saját maga hajtja vére a kódból kiolvasott parancsoknak megfelelő műveleteket.

[3] Szabványosított programozási nyelv meghatározás.

[4] Hypertext Markup Language, leíró nyelv webalkalmazások készítéséhez.

[5] Cascading Style Sheets stíluslap nyelv, dokumentumok megjelenésének leírására.

[6] Karakter sorozatok, keresési minták meghatározására.

[7] Document Object Model, elemek fa struktúrában való kezelése.

2.1.1 TypeScript

A TypeScript egy nyílt forráskódú programozási nyelv, melyet a Microsoft hozott létre és tart karban. Ez egy szigorú szintaktikai részhalmaza a JavaScriptnek és statikus típusosságot ad a nyelvnek [5].

Az ebben készített alkalmazások visszafordulnak JavaScriptre. A visszafordításra több lehetőség is van, amik közül a TypeScript Checker használata ajánlott.

A statikus típusosságot leíró fájlokon keresztül éri el, amiben típus információk vannak megadva a létező JavaScript könyvtárakról, hasonló, mint a C++ header állományok.

Fő előnyei a JavaScripthez képest: típus annotáció / fordítás idejű típus ellenőrzés, típus következtetés, interfészek, felsorolás típusok (Enum), generikus típusok, névterek, async/ await programozási minta.

2.2 Angular (Angular 2+)

Az Angular egy HTML és TypeScript alapú nyílt forráskódú alkalmazásfejlesztési keretrendszer. A korábbi AnglarJS teljesen újragondolt változata.

Ez a keretrendszer főként webfejlesztésre szolgált, de mára már cross-platform lett, mely segítségével progresszív web- (PWA), natív mobil- és asztali alkalmazások egyaránt készíthetők.

2.2.1 Felépítés

Az alap építő kövei az *NgModule*-ok, melyek komponensek egy gyűjteményét biztosítják. Az alkalmazásoknak mindig kell, hogy legyen egy gyökér egysége (modulja), ami betölti a komponenseket, szolgáltatásokat nyújt, illetve másik modulokat tölt be, melyek valamilyen segéd funkciót nyújtanak.

A komponensek a kijelző egy kisebb részét (*view*) irányító elemek. Az alkalmazás logika alapján módosíthatja a felületet. Felhasználja a rendszer szolgáltatás osztályait, amelyek valamilyen speciális funkcióval szolgálnak.

2.2.2 Modulok

Az *NgModule*-ok az összetartozó kódrészletek tárolói. Tartalmazhatnak komponenseket, szolgáltatás *providereket* és más modulok kód fájljait, amik

felhasználásra kerülnek, ezzel elérhetővé téve a funkcionalitásukat. Minden Angular alkalmazásnak van egy gyökér eleme, ami a név konvenció alapján *AppModule* névre hallgat és az *app.module.ts* állomány tartalmazza. Az alkalmazás az *NgModule* betöltésével indul.

A *@NgModule* dekorátorral megjelölt TypeScript osztályokból lesznek a modulok. Ez a dekorátor konfigurációs metaadatokat tartalmaz, amelyek közül a legfontosabbak:

- declarations: komponensek, direktívás, *pipe*-ok[8], amik a modul részei,
- exports: deklarációk részhalmaza, amik szükségesek, hogy a komponensek láthatók és használhatók legyenek a többi modulban,
- imports: más modulok, melyek osztályai szükségesek a saját komponenseink sablonjaihoz,
- providers: a modul által nyújtott szolgáltatások, melyek az egész alkalmazásban elérhetők,
- bootstrap: a fő alkalmazás komponens.

Egy példa egy egyszerű modulra:

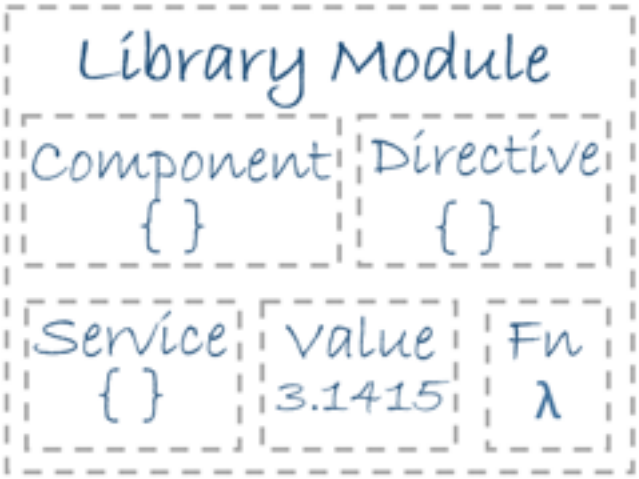
```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
    imports: [ BrowserModule ],
    providers: [ Logger ],
    declarations: [ AppComponent ],
    exports: [ AppComponent ],
    bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Számos könyvtárat szolgáltat az Angular keretrendszer, melyekre osztálykönyvtárként is lehet gondolni. Mindegyik neve *@angular* előtaggal kezdődik. Telepíteni őket a node package manager[9] (npm) segítségével lehet. Felhasználásukhoz a

[8] Adat transzformációt végző eszköz a HTML sablonban

[9] Csomagkezelő rendszer JavaScript programozáshoz.

fő modulban szükséges importálni őket úgy, hogy a *@NgModule imports* tulajdonságához hozzáadjuk a könyvtárból a releváns osztályokat.



1. ábra: Angular osztálykönyvtár felépítése [6]

2.2.3 Komponensek

A képernyő egy darabját/részletét vezérlik, ezt nevezik *view*-nak. A komponensek TypeScript osztályok, a *@Component* dekorátorral kell őket ellátni. Ebben a meta adatban szükséges megadni a *selector* tulajdonságot: ez meghatározza, hogy melyik CSS selector helyére kell létrehozni a komponenst. A *templateUrl* a komponenshez tartozó HTML darab címét tartalmazza. A *providers* a felhasználni kívánt szolgáltatások listája.

2.2.4 Sablon szintaxis és adatkötés

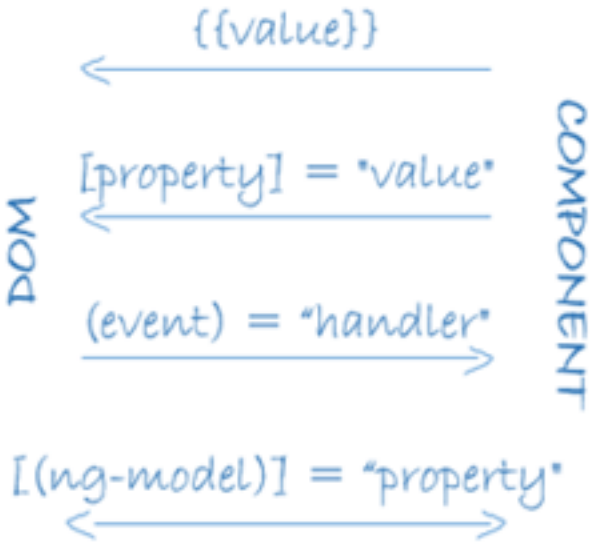
A sablonok, más néven *template*-k, egyszerű HTML részletek. Kiegészítik az Angular szintaxisok: a direktívák (**ngIf, *ngFor*) és az interpolációk (beágyazott kifejezések, jelölése:*{{}}*). A sablonok teremtik meg a kapcsolatot a felhasználók és a komponensek között. Ezt segíti az adatkötés (data binding): összekapcsolja az alkalmazás adatait és a DOM-ot.

Egy egyszerű példa egy sablonra adatkötéssel:

```
<ul>
    <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
        {{hero.name}}
    </li>
</ul>
```

Ebben a sablon részletben több Angular szintaktika is megjelenik. A **ngFor* direktíva a lista minden elemén végighalad. A *{{hero.name}}*, illetve a *(click)* már az adatkötés részei.

A keretrendszer használata nélkül, a fejlesztők felelőssége lenne a felhasználók által bevitt értékek hozzárendelése a szoftver adataihoz. Erre nyújt megoldást az Angular a kétirányú adatkötéssel. Ez a mechanizmus köti a sablon egy darabját a komponens egy darabjához. A *{{}}* jelzés köti össze az adatot a két oldallal.



2. ábra: Adatkötés fajtái [2]

A *{{value}}* megjeleníti a benne lévő értéket. A *[property]* továbbadja az értéket egy másik komponensnek. A *(event)* egy eseménykezelőt köt a felhasználó által kiváltott eseményre és az igazi kétirányú adatkötést pedig az *[(ng-model)]* szolgáltatja.

2.2.5 Szolgáltatások és függőség injektálás

A szolgáltatások egy széles kategória, amely magában foglal minden olyan adatot, funkciót, amire az alkalmazásnak szüksége van. Ezek általában osztályok egy szűkebb, jól definiált céllal, melyek specifikusak egy adott területre.

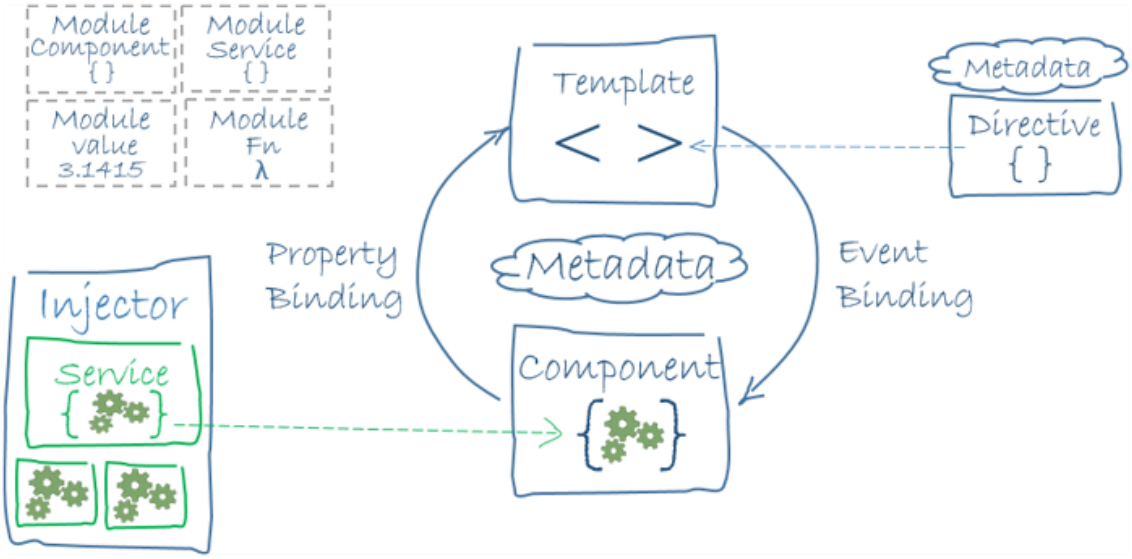
Az Angular megkülönbözteti a szolgáltatásokat a komponensektől, ezzel növelve a modularitást és az újrahasznosíthatóságát a megírt kódnak. Ezt úgy éri el, hogy a nézet specifikus funkciókat szétválasztja az egyéb fajta számításoktól.

Ideális esetben a komponensek feladata csupán a felhasználók interakcióinak kezelése és semmi több. A szolgáltatásoknak kell elvégezni olyan tevékenységeket, mint például adatok letöltése a szerverről, bevitt értékek vizsgálata és ellenőrzése, naplózás.

A függőség injektálás (Dependency injection, DI) beépítetten elérhető az Angular keretrendszerben és biztosítja a szolgáltatásokat az új komponenseknek. A szolgáltatásokat a *@Injectable()* dekorátorral lehet létrehozni és ez biztosítja, hogy a keretrendszer tudja injektálni, mint függőség. Az alkalmazás létrehoz egy alkalmazás-szintű injektort a betöltés folyamat alatt (*bootstrapping*), ami felelős az injektálásokért.

2.2.6 Működés

Az alapelemek bemutatása után szeretném részletesen demonstrálni, hogyan működik egy Angular alkalmazás. Ehhez nyújt segítséget a következő ábra:



3. ábra: Angular alkalmazás felépítés [7]

Elsőként az alkalmazás gyökér modulja töltődik be. Ez létrehozza a különböző szolgáltatásokat és betölti a függőségeket. Ezt követően a *bootstrap* tulajdonságban megadott komponenst létrehozza az alkalmazás. Ennek legenerálódik a sablonja, amit betölt a DOM-ba. Ez a sablon tartalmazhat adatkötést, illetve direktívákat.

2.3 Entity Framework Core

Az Entity Framework Core (EF Core) egy bővíthetőbb és a platform független verziója a nagy Entity Framework (EF) adat elérési technológiának. Ez egy object-relational mapper (ORM), ami lehetővé teszi, hogy az adatbázis elemeit .NET objektumként használjuk. Ezzel kiváltható sok alacsony szintű adatelérési kódolás, ami egyébként szükséges lenne. Az Entity Framework Core sok fajta adatbázis motort

támogat. Ezek közül én az MSSQL adatbázis-kezelő rendszert használtam a szerveremhez, illetve InMemory adatbázist a teszteléshez [8].

A nagy Entity Framework három modell készítési fajtát támogat alkalmazások készítéséhez úgymint a Code First, Model First és Database Firs. Az EF Core két fő utat biztosít a séma létrehozáshoz. Az egyik az említett Code First, ahol a kódban elkészített osztályokból migráció segítségével generálódik az adatbázis, a másik pedig a Reverse Engineering, ahol az adatbázis alapján jönnek létre az entitás osztályok.

Az entitások között több fajta kapcsolat lehet, ezek megadására több lehetőséget is biztosít az Entity Framework Core. Az első és legegyszerűbb megoldás a navigációs tulajdonság alapú leképezés. Ezek a kapcsolatok úgy működnek, mint az adatbázisban a külső kulcsok. Ezen felül létezik az annotációs megoldás, ahol az entitás osztályainkat annotáljuk fel a megfelelő attribútumokkal. A harmadik megoldási lehetőség az ún. Fluent API, ahol az adatbázis kontextusunk létrehozásában kell definiálni a megfelelő relációkat, tábla leképezéseket.

2.4 ASP.NET Core

Az ASP.NET Core egy keresztplatformos, nagyteljesítményű és nyílt forráskódú keretrendszer, modern, felhő alapú alkalmazások készítésére. Ez a nagy ASP.NET újra tervezése szerkezeti változásokkal, amiknek köszönhetően modulárisabb lett a rendszer. Számos előnnyel rendelkezik: beépített függőség injektálás, nagyteljesítményű, moduláris HTTP kérés csővezeték, IIS és ön-hosztolásra is képes saját folyamatban (process), több operációs rendszeren is futhat (Windows, macOS, Linux).

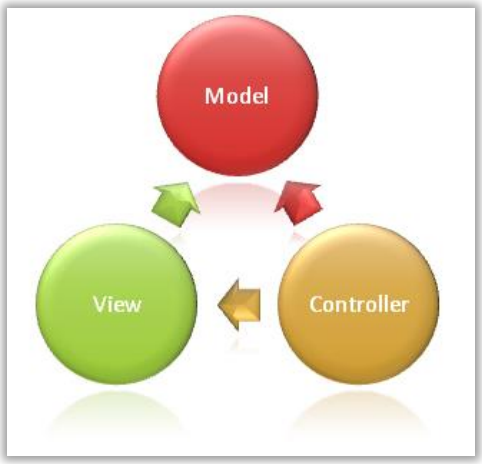
Az ASP.NET Core részei NuGet csomagokon[10] keresztül érhetők el, ezzel is lehetővé téve az optimalizálást, mivel elég csak a szükséges csomagra hivatkozni.

Az ASP.NET Core MVC sok szolgáltatást nyújt a fejlesztőknek, ezek közé tartozik az út kezelés (routing), modellkötés (model binding), modell validáció, függőség injektálás, szűrők stb. Ezek közül számos komponenst felhasználtam az alkalmazásom elkészítése során.

[10] Ez egy csomagkezelő rendszer .NET-es projektek számára.

2.4.1 Modell-nézet-vezérlő minta

Az modell-nézet-vezérlő (model-view-controller, MVC) minta három komponensből épül fel: modell, nézet és vezérlő. Segítségével elérhetjük a szerepkörök különválasztását. A modellek felelőssége az alkalmazás logika megvalósítása. A vezérlők kezelik a felhasználók kéréseit. Modellekkel dolgozik és kiválasztja, milyen nézeteket kell szolgáltatni. A tartalom megjelenítéséért a nézetek a felelősek.



4. ábra: MVC komponensei és hivatkozásaik egymásra [9]

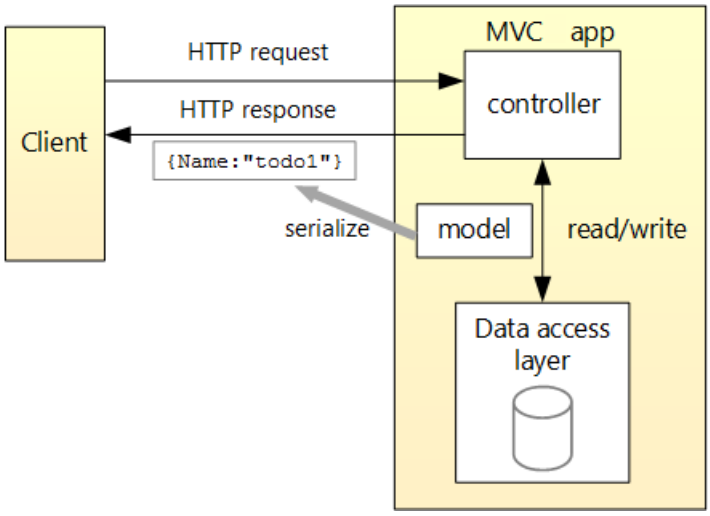
2.4.2 Web API

Az ASP.NET Core támogatja a RESTful szolgáltatások (WebApi) létrehozását C# nyelven. A kérések feldolgozásához a vezérlő (*Controller*) osztályokat használja. A létrehozott alkalmazásban a HTTP igék segítségével történik a kommunikáció. Egy egyszerű példa az igék szemléltetésére:

Ige	**Leírás**	**Kérés törzse**
GET /api/Elements	Visszaadja az összes Element listáját	-
GET /api/Elements/{Id}	Visszaadja az Id-val rendelkező Elementet	-
POST /api/Elements	Új Element létrehozása	Element adatai
PUT /api/Elements/{Id}	Id-val rendelkező Element frissítése	Element adatai

DELETE /api/Elements/{Id}	Id-val rendelkező Element törlése	-

A WebApi alkalmazás működése a következő: A kliens HTTP kérést indít a szerver felé (GET, POST, PUT, DELETE), az adatokat JSON[11] vagy XML[12] formátumba sorosítva. Ezt a *Controller* osztály megkapja, kicsomagolja, majd a kapott objektumot feldolgozza (írhat, olvashat az adatbázisból). A kérés alapján a választ sorosítva visszaküldi.



5. ábra: WebAPI kommunikáció

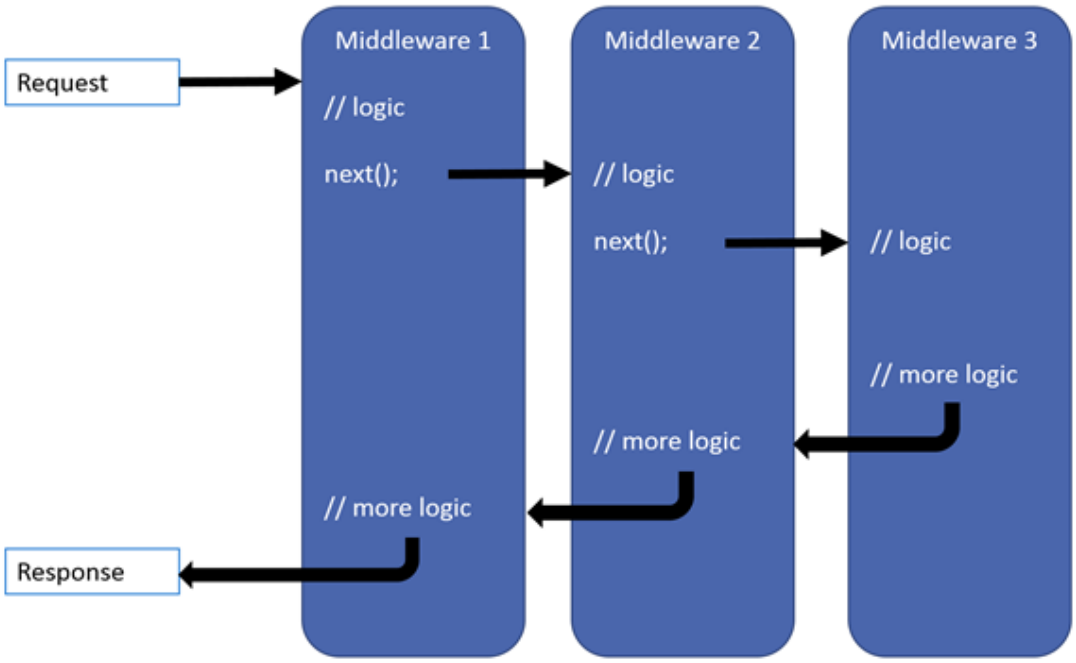
2.4.3 Single-page application Middleware

A single-page-application (SPA) egy web alkalmazás, amely kezeli a felhasználók interakcióit és ez által dinamikusan újra írja a jelenlegi munkaablakot, ahelyett, hogy az egész oldalt újra letöltené a szerverről.

A Middleware az alkalmazáshoz befutó kérések kiszolgálásához beépülő szoftverkomponens. Feldolgozza a hozzá beérkező kérést, majd eldöntheti, hogy továbbítja azt a következő komponensnek vagy nem.

[11] JavaScript Object Notation, egy kis méretű, szöveg alapú szabvány adatcserére

[12] Extensible Markup Language, leíró nyelv



6. ábra: Middleware pipline [10]

A SPA Middleware egy megoldás arra, hogy a SPA-k jobb támogatást kapjanak a Microsoft ökoszisztémában. Ez a middleware egy nagyon gyakorlatias megközelítést nyújt. A SPA egy alkönyvtárban helyezkedik el a projektben és itt fut közvetlenül és elszigetelve.

A *Startup Configure* függvényben szükséges beregisztrálni a middlewaret, hogy beépüljön a feldolgozási láncba:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseSpa(spa =>
    {
        spa.Options.SourcePath = "ClientApp";

        if (env.IsDevelopment())
        {
            spa.UseAngularCliServer(npmScript: "start");
        }
    });
}
```

2.4.4 SignalR

Az ASP.NET Core által nyújtott osztály könyvtár, ami a szerver számára lehetőséget nyújt aszinkron értesítés küldésére a kliensalkalmazás felé. Ennek köszönhetően valós idejű web alkalmazások készíthetők. Tartalmaz szerver és kliens oldali JavaScript komponenseket.

Ez a valós idejű funkcionalitás azt jelenti, hogy a szerver oldali kód tartalmat küld a csatlakozott klienseknek, valós időben. Ez abban különbözik a sima webes alkalmazásoktól, hogy nem csak a felhasználó valamilyen műveletére történhet változás az oldalon, hanem a szerver értesítheti eseményekről.

2.4.4.1 Régebbi megoldások valós idejű web alkalmazásokhoz

Az osztálykönyvtár létezése előtt is több lehetőség volt a valós idejű alkalmazás létrehozására, ám ezek nem voltak a legjobb, illetve a legszebb megoldások. Ezeket szeretném bemutatni előnyeik és hátrányaik szemléltetésével.

A hagyományos polling egy olyan módszer, amikor a szervert folyamatosan, adott időközönként lekérdezzük az új változásokról Ajax segítségével. Ebben segít a JavaScript *setInterval* függvénye. Előnye, hogy nagyon egyszerű és minden böngésző támogatja, míg hátránya, hogy nem azonnal jelennek meg a friss adatok (várni kell a következő frissítésig), sok felesleges kommunikáció, ha nincs változás. Veszélyei közé tartozhat, hogy ha a kérés tovább tart, mint a frissítési ciklus, akkor a kérések egymásra csúszhatnak.

Long-polling esetén a kliens tartja nyitva a kapcsolatot hosszabb ideig. Ha van változás, majd a szerver visszaküldi (akár streamen keresztül is). Ha nem történt változás az időkorlát lejárata bontja a kapcsolatot. Előnyei közé tartozik az egyszerű megvalósítás, böngésző támogatottság, azonnal friss adatok. Hátránya a bonyolult szerver oldali implementáció, illetve a szerver terhelése (erőforrások tartása a kérés alatt).

A WebSocket egy protokoll [11], ami két-oldali kommunikációt biztosít a kliens és a szerver között (TCP kommunikáció). Előnye, hogy full duplex és nem HTTP alapú, így a velejáró fejlécek overheadjei lekerülnek. Régebben hátrányai közé tartozott, hogy csak az újabb böngészők támogatták.

A SignalR a WebSocket protokollt használja alapértelmezetten, ha a böngésző támogatja azt. Ha nem, akkor több fallback[13] mechanizmust támogat, hogy minden eszközön működjön.

2.4.4.2 Megvalósítás

A könyvtár használatához több lépés tartozik. A kliens oldali kódhoz hozzá kell adni a *@aspnet/signalr* osztálykönyvtárt, amihez TypeScript típus definíciók is tartoznak. Szerver oldalon az alkalmazást kell bekonfigurálni, hogy hozzáadja a SignalR middlewaret:

```
public void ConfigureServices(IServiceCollection services)
{
…
    services.AddSignalR();
    …
}
```

Ezen felül szükséges beállítani a címet, amin keresztül elérhető:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseSignalR(routes =>
       {
           routes.MapHub<BookingHub>("/bookingHub");
       });
}
```

Itt már megjelenik a fő komponense a SingalR-nek, ami nem más, mint a *Hub*. Ez szolgál magas szintű csővezetékként, amely kezelni a kliens-szerver kommunikációt. Benne kezelődnek a kapcsolatok, csoportok és az üzenetek.

A *Hub* tartalmaz egy *Context* tulajdonságot, amiben a kapcsolat adatai érhetők el (*User, ConnectionId, Items*). Ezen felül a csatlakozott felhasználókat a *Clients* tulajdonságon keresztül érhetjük el. Ezeknek a *SendMessage, SendMessageToCaller, SendMessageToGroups* metódusokon keresztül küldhetünk üzeneteket.

[13] Olyan lehetőség, amit akkor választ, ha a használni kívánt mechanizmus nem elérhető.

2.5 AutoMapper

Egyszerű és kicsi osztálykönyvtár objektumok értékeinek leképezésére. Ez egy objektum-objektum leképező, ami segít a rétegek közötti adatmegosztásban. Egyszerű megközelítést alkalmaz, van egy forrás objektum és egy cél objektum.

```
var esDto = _mapper.Map<EventSchedule, EventScheduleDTO>(eventSchedule);
```

Első sorban azokat az értékeket másolja át, ahol az egyes tulajdonságok nevei megegyeznek. Ha komplexebb logikára van szükség, ahhoz külön konfigurációt kell létrehozni, illetve lehetőség van globálisan is beállítani a leképezéseket.

```
Mapper.Initialize(cfg =>
{
    cfg.CreateMap<Service, ServiceViewModel>()
        .ForMember(dest => dest.Image, opt => opt.Ignore());
    cfg.CreateMap<ServiceViewModel, Service>()
        .ForMember(dest => dest.Image, opt => opt.Ignore())
        .ForMember(dest => dest.Id, opt => opt.Ignore());
      ...
});
```

2.6 Skálázható Vektor Grafika (SVG)

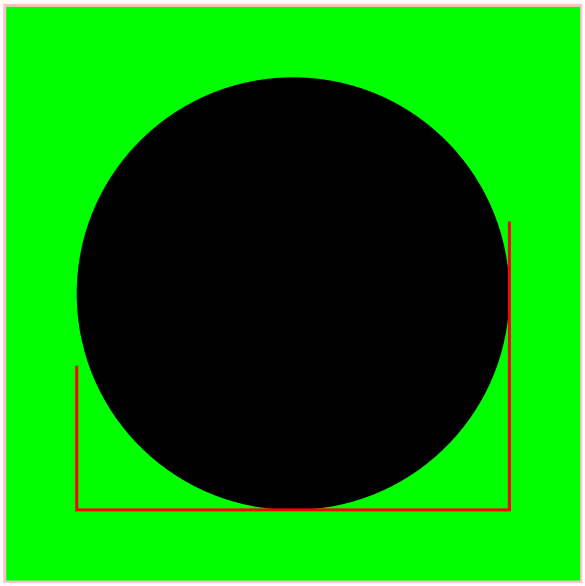
Az SVG egy XML alapú vektoros kép formátum kétdimenziós grafikákhoz. Támogatják az interaktivitást és az animációkat. Az SVG specifikáció egy szabvány, amit a Word Wide Web Consortium fejleszt 1999 óta.

Mivel az SVG képek XML fájlokban vannak definiálva, ezért kereshetők, indexelhetők, szkriptelhetők és tömöríthetők. Létrehozhatók egyszerű szövegszerkesztővel, de léteznek hozzá rajzoló szoftverek is.

Minden modern böngésző támogatja az SVG képek megjelenítését. Az SVG dinamikus és interaktív rajzokat is lehetővé tesz. Objektummodellje (DOM) tartalmazza a teljes XML DOM-ot, így a JavaScript használatával egyszerű és hatékony vektorgrafikus animációt tesz lehetővé. Gazdag eseménykezelő készlettel rendelkezik, mint az *onmouseover* vagy az *onclick*, melyeket bármelyik grafikus elemhez hozzá lehet rendelni.

Egy egyszerű SVG kép leírása:

```
<?xml version="1.0" encoding="UTF-8" ?>
<svg width="391" height="391" viewBox="-70.5 -70.5 391 391"
xmlns="http://www.w3.org/2000/svg">
  <rect x="25" y="25" width="200" height="200" fill="lime" stroke-width="4"
stroke="pink" />
  <circle cx="125" cy="125" r="75" fill="black" />
  <polyline points="50,150 50,200 200,200 200,100" stroke="red" stroke-
width="4" fill="none" />
</svg>
```



7. ábra Példa SVG képként

A vektorgrafikus képek geometria primitívekből épülnek fel, ilyenek például a pontok, egyenesek, görbék. Előnyei a raszter grafikával szemben, hogy tetszőlegesen nagy nagyítás sem torzítja a képet, a vonalvastagságnak nem kell a nagyítással arányosan nőni, az alakzatok méretei tárolhatók és így később könnyebb őket megváltoztatni.

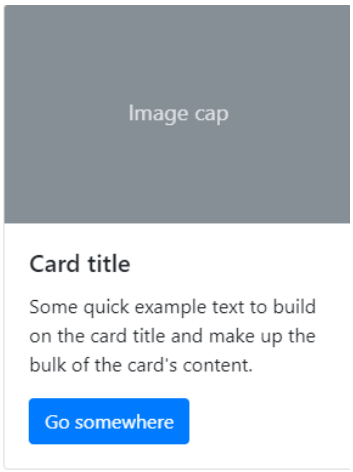
2.7 Bootstrap

A Bootstrap egy front-end oldali komponens könyvtár esztétikus, reszponzív web alkalmazások készítéséhez. Az eszköztár HTML, CSS és JavaScript komponensekből épül fel. Telepítése egyszerű, a hozzá tartozó stíluslapokat és JavaScript állományokat kell behivatkozni *link* és *script* tagekben.

Az egyik talán legfontosabb eleme az elrendezési (Layout) komponens. Ez az úgynevezett rács rendszer (grid system). Ennek az alapjai az oszlopok, mivel a grid

tizenkettő oszlopra bontja fel a kijelzőt. Ezeket az oszlopokat kell a *.row* osztállyal ellátott elemekbe helyezni. Az oszlopok méretét a *.col* után idézőjellel megadott szám jelenti. Ezek elé lehet különböző beépített képernyő méreteket megadni, ezzel testre szabva, hogy mekkora legyen az elem az egyes kijelzőkön.

Ezen felül számos beépített komponenst nyújt, amiket egyszerű CSS osztályokkal lehet használni. Ilyenek például a kártyák, amik egy képből, címből és egy szövegtörzsből állnak.



8. ábra: Bootstrap kártya

A hozzá tartozó HTML kódrészlet a következő:

```
<div class="card" style="width: 18rem;">
  <img src="..." class="card-img-top" alt="...">
  <div class="card-body">
    <h5 class="card-title">Card title</h5>
    <p class="card-text">Some quick example text to build on the card title and make up the bulk of the card's content.</p>
    <a href="#" class="btn btn-primary">Go somewhere</a>
  </div>
</div>
```

2.8 xUnit.net

Az xUnit.net egy ingyenes, nyílt forráskódú unittesztelő eszköz .NET keretrendszerhez. A xUnit tesztek létrehozásához egy külön xUnit teszt projektet kell

létrehozni. Az egység tesztek olyan függvények, melyek *[Fact]* vagy *[Theory]* attribútummal vannak ellátva.

A tények (Facts) olyan tesztek, amik mindig igazak. Nem változó körülményeket tesztelnek. Az elméletek (Theories) olyan tesztek, amelyek csak adott adatsorra igazak. Ezért a *[Theory]* attribútummal ellátott függvényeknek kell mindig legyen *[InlineData()]* attribútuma is. Ahány paramétere van az *InlineData*-nak, annyi kell legyen a függvénynek is, különben fordítás idejű hibát kapunk. Ezt szemlélteti a következő példa.

```
[Theory]
[InlineData(3)]
[InlineData(5)]
[InlineData(6)]
public void MyFirstTheory(int value)
{
    Assert.True(IsOdd(value));
}
```

Az egység tesztek három lépésből épülnek fel (*Arrange, Act, Assert*). Az *Arrange* részben szükséges a környezet felállítása, változók paraméterezése. Az *Act* részben végre kell hajtani a lépést, amit tesztelünk. Az *Assert* részben pedig vizsgáljuk, hogy a kapott eredmény megegyezik azzal, amit elvártunk.

3. Tervezés

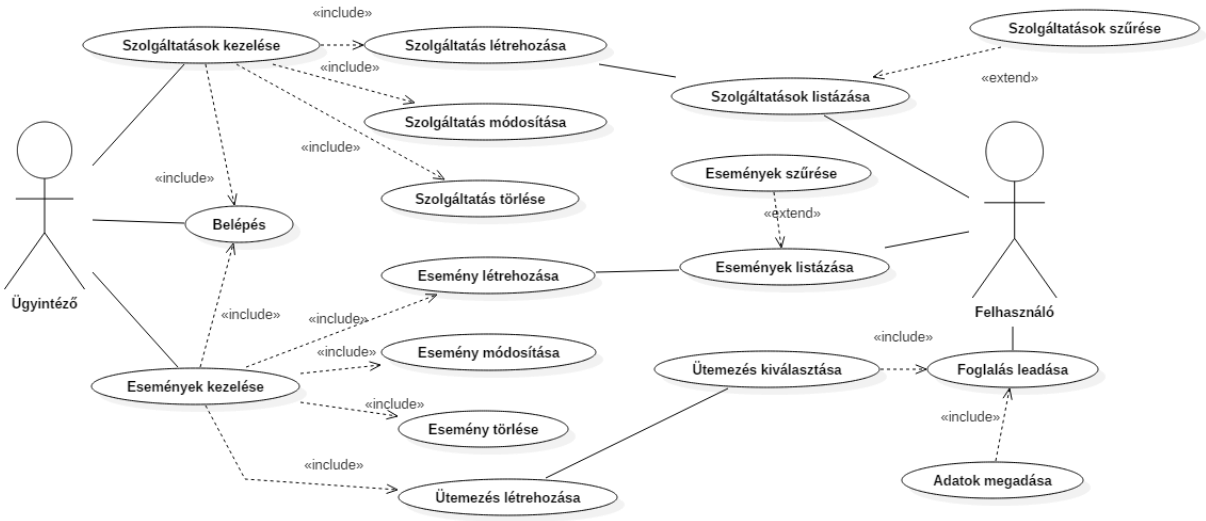
A fejezetben az elkészített alkalmazásom mutatom be, ezen felül annak követelményeit és az architektúra felépítését.

3.1 Alkalmazás tervének bemutatása

Az első lépés a foglalási oldal elkészítése volt, ahol az élő kommunikáció megvalósításán volt a hangsúly. Lehetőséget kell adni a felhasználók számára a rendszerben lévő események és szolgáltatások szűrésére, megtekintésére. Az ezekhez szükséges adatokat egy ügyintéző oldalról lehessen feltölteni. A felhasználók a foglalásokat valós időben lássák és legyen lehetőségük a kiválasztott helyeiket adott ideig zárolni. Az ügyintéző portálon az adatok kezelése (események, szolgáltatások létrehozása/ törlése) legyen azonosításhoz kötött.

3.2 Szerepkörök bemutatása

A rendszerben két fajta szerepkör van, amiket két külön portál kezel. Az egyik a normál felhasználó, aki a publikus oldalt használhatja, illetve az ügyintéző felhasználó, aki a belső adminisztrációs oldalt éri el. Az alábbi use-case diagramm segítségével szeretném bemutatni az egyes szerepkörökhöz kapcsolódó funkcionális követelményeket.



9. ábra: Rendszer use-case diagrammja

3.2.1 Normál felhasználó

A normál felhasználó a publikus portált éri el és azon hajthat végre műveleteket. Az ehhez kapcsolódó funkciók a következők:

Funkció	Leírás	Részletek
Szolgáltatások listázása	A rendszerben lévő szolgáltatások listázása és szűrése.	A rendszerben lévő szolgáltatások megjelenítése listás formában, ezen felül az adatok szűrése típus, név és város alapján.
Események listázása	A rendszerben lévő események listázása és szűrése.	A rendszerben lévő események megjelenítése listás formában, ezen felül az adatok szűrése név, szolgáltatás és kezdési időpont alapján.
Foglalás leadása	Kiválasztott esemény egy időpontjára foglalás leadása.	A rendszerben lévő eseményekhez kapcsolódó ütemezés kiválasztása. Ezt követően egy vagy több pozíció megjelölése, amit le szeretne foglalni. Adatok megadása a foglalás adminisztrálásához.

3.2.2 Ügyintéző felhasználó

Az ügyintéző felhasználó a belső adminisztrációs oldalt éri el és azon hajthat végre műveleteket. Az ehhez kapcsolódó funkciók a következők:

Funkció	Leírás	Részletek
Belépés	Felhasználónév és jelszóval történő bejelentkezés a portálra.	Minden funkció használatához bejelentkezés szükséges.
Szolgáltatások kezelése	A rendszerben lévő szolgáltatások kezelése.	A funkció magában foglalja a szolgáltatáshoz kapcsolódó további műveleteket:

		• Adatok listázása • Létrehozás • Módosítás • Törlés (ha nincs még kapcsolódó entitás)
Események kezelése	A rendszerben lévő események kezelése.	A funkció magában foglalja az eseményhez kapcsolódó további műveleteket: • Adatok listázása • Létrehozás • Módosítás • Törlés (ha nincs még kapcsolódó entitás) • Ütemezések létrehozása

3.3 Nem-funkcionális követelmények

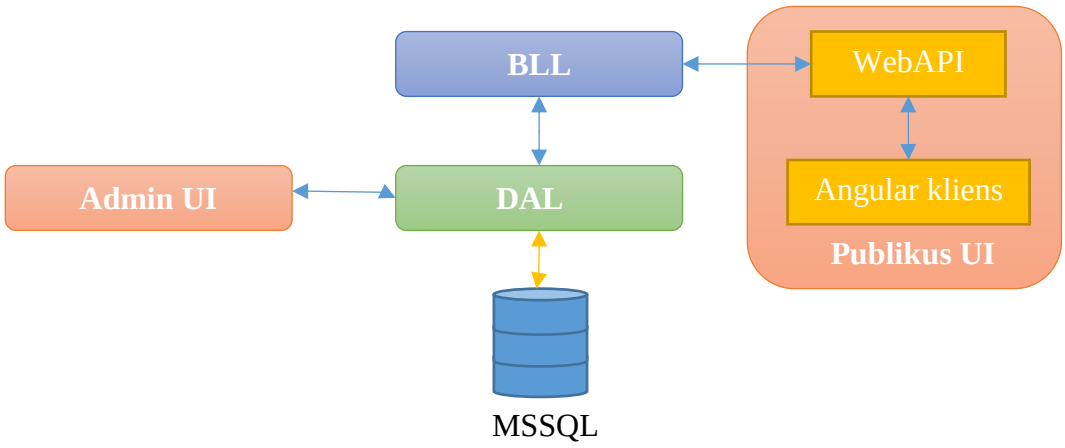
Az elkészült rendszernek számos nem-funkcionális követelménynek is meg kell felelnie:

- Biztonság: az érzékeny részeket védeni kell az illetéktelen hozzáféréstől.
- Karbantarthatóság: az elkészült kód legyen könnyen karbantartható.
- Megbízhatóság: az esetleges hibák legyenek kezelve.
- Egyszerűség: a fejlesztés során törekedni kell az egyszerű megoldásokra.
- Felhasználói élmény: törekedni kell a felhasználói élmény maximalizálására.

3.4 Architektúra bemutatása

A rendszerem két webportálból épül fel, egy publikusból a foglalások lebonyolításához és egy belső adminisztrációsból. Ezek megvalósítására többrétegű architektúrát választottam, ahol a megjelenítés, az adatkezelés és az üzleti logika különálló folyamatot alkotnak. Ennek előnyei, hogy a rétegek külön tervezhetők és

fejleszthetők, akár a teljes implementáció kicserélhető, ha az interfész változatlan marad. Rendszerem közös architektúráját az alábbi ábra szemlélteti.



10. ábra: Rendszer közös architektúra

A legalsó réteg az adat elérési réteg (Data Access Layer, DAL), ami biztosítja az adatok elérését külső forrásból. Relációs adatbázisnak a Microsoft SQL Servert választottam, mert már többször is használtam és egyszerű vele a fejlesztés Windows operációs rendszer alatt. Az adatbázis sémát az Entity Framework Core Code First megoldásával készítettem el.

Az üzleti logikai réteg (Business Logic Layer, BLL) tartalmazza a szolgáltatásokat, amelyek meghatározzák az üzleti folyamatokat, szabályokat azáltal, hogy szabályozza az adatokon végezhető műveleteket. Ezt az én architektúrámban csak a publikus UI réteben használtam fel. Ez abból adódik, hogy az adminisztrációs oldal kisebb szerepet kapott a diplomamunkám során, mert főleg csak CRUD[14] műveletekből áll.

A rendszerem két megjelenítési réteggel (User Interface, UI) rendelkezik. Az egyik a publikus oldal kiszolgálásáért felelős, a másik pedig az adminisztrációs munkák

[14] Mozaik szó a létrehozás, lekérdezés, módosítás, törlés. (Create, Read, Update, Delete)

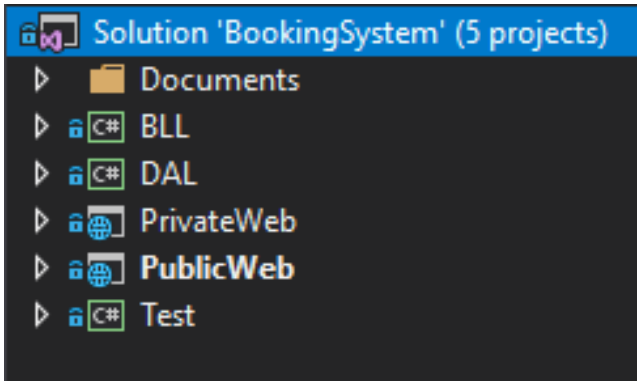
elvégzését segíti. A diplomamunkám egyik feladata volt egy kliensoldali technológia megismerése és felhasználása volt, ezért választottam ketté a rendszeremet.

A publikus oldal felhasználói felületét Angular keretrendszerrel valósítottam meg. Mivel így elveszítettem azt az előnyt, amit a Razor nyújt (View-Controller kapcsolat), ezáltal itt egy WebAPI is elkészült, amin keresztül folyik a kommunikáció a kliens és a szerver között. Ezen felül az élő kommunikáció is itt valósul meg, amihez a SignalR technológiát használtam fel.

Az adminisztrációs oldal felhasználói felületét az ASP.NET Core MVC segítségével hoztam létre. Itt nem használtam fel az üzleti logikai réteget, mivel csak plusz munka lett volna a sok CRUD műveletet leképezni és a munkámban ez nem kapott nagy szerepet. Természetesen a továbbfejlesztési célok között nagy hangsúlyt kap ennek a megvalósítása.

3.4.1 Projektek

Az elkészült alkalmazásom egy Visual Studio Soultionből [12] áll, öt projektet tartalmaz, ebből három darab .NET Core osztálykönyvtár és két darab ASP.NET Core projekt.



11. ábra: Az alkalmazás projektjei

A DAL projekt tartalmazza az entitás osztályokat, az adatbázis konfigurációt, a generált migrációs állományokat [13], illetve szolgáltatásokat, amik az adatokkal való feltöltést segítik.

A BLL projektben kaptak helyet az üzleti logikát megvalósító szolgáltatás osztályok, az üzleti logikai entitások, saját kivétel osztályok. Hivatkozik a DAL osztálykönyvtárra.

A PrivateWeb projekt az adminisztrációs oldalt megvalósító ASP.NET Core webalkalmazás. Tartalmazza a *Controller* osztályokat, a felhasználói felületeket és a modell osztályokat. Hivatkozik a DAL osztálykönyvtárra.

A PublicWeb projekt a publikus oldalt megvalósító ASP.NET Core webalkalmazás. Tartalmazza a kliensalkalmazást egy külön mappában, a modell osztályokat, az élő kommunikációhoz megvalósított komponenseket és a WebAPI *Controller* osztályait. Hivatkozik a BLL és DAL osztálykönyvtárakra.

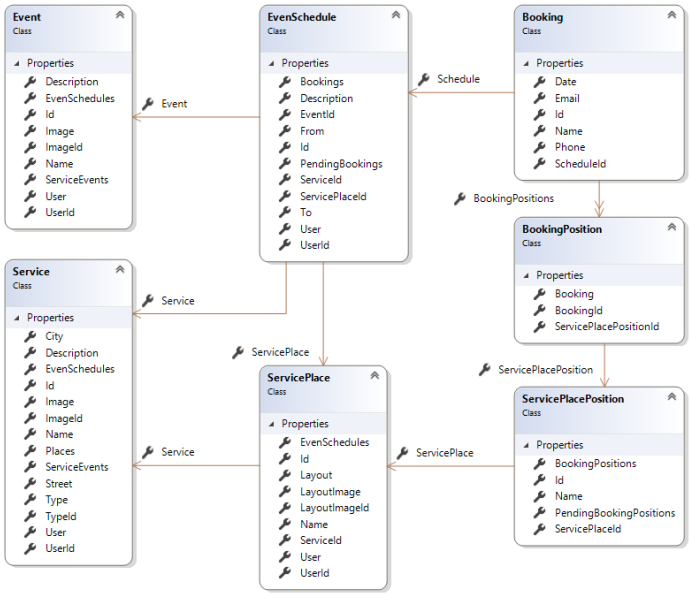
A Test projekt egy xUnit teszt projekt, ahol a rendszerhez elkészült egység tesztjeim vannak. Hivatkozik a BLL, DAL osztálykönyvtárakra és a PublicWeb alkalmazásra.

3.4.2 Adatbázis terv

Az architektúra részeként szükségesnek érzem az alkalmazás alapjául szolgáló adatmodell bemutatását. A sémára nagy hatással lévő tervezői döntések és a hozzájuk kapcsolódó entitások részletezésével kezdem.

3.4.2.1 Tervezői döntések

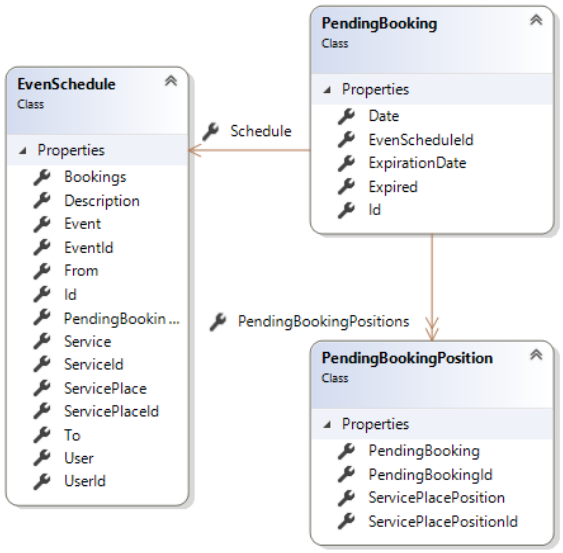
Foglalási rendszer lévén fontos volt megalkotni egyfajta „foglalási modellt”. Az én esetemben ez úgy valósult meg, hogy a felhasználó egy esemény egy ütemezését kiválasztva foglalhat az adott szolgáltató helyeire. Például moziban egy vetítés egy székére való jegyvásárlás. Az ehhez kapcsolódó entitásaimat az alábbi ábra mutatja:



12. ábra: Foglaláshoz kapcsolódó entitások

Egy másfajta modell lett volna a „folytonos foglalhatóság” megoldása. Például egy étterem asztalának foglalása adott órától adott óráig. Ez a modell távolabb állt az elvárásoktól, ezért választottam a másikat.

A foglalások tétele előtt szükséges egyfajta előfeltétel, a foglalási szándék rögzítése. Ez azért van, hogy a felhasználók valós időben lássák, melyik helyek elérhetőek még. A foglalási szándék két dolgot foglal magában: az időintervallumot, ameddig érvényes a szándék, illetve a pozíciókat, amelyekre majd a sikeres foglalás szól. A következő entitások kapcsolódnak hozzá:



13. ábra: Foglalási szándék entitásai

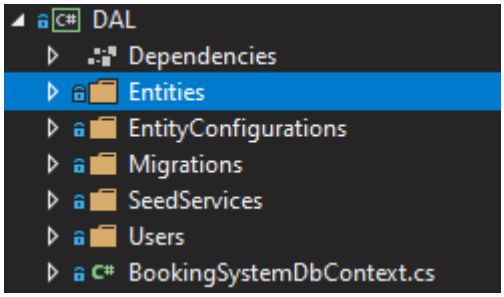
A *Booking* és *PendingBooking* entitások közti különbség, hogy az első tárolja a felhasználó adatait is. Mivel a foglalási szándékok csak bizonyos ideig érvényesek, ezért valamikor törölni kell őket, hogy ne foglalják a helyet. Erre a dolgozatomban nem adtam megoldást, csak manuálisan lehetséges a törlés. Megoldás lehet erre az admin oldalon egy funkció, ami eltávolítja ezeket az elemeket, vagy egy háttérszolgáltatás implementálása.

4. Megvalósítás

Az elkészült rendszerem két webportálból áll. A publikus részen lehet a foglalásokat leadni, a belső felület pedig az ügyintézői feladatok elvégzésére szolgál. Az általam elvégzett munkát a közös rétegek bemutatásával kezdem, majd ezt követően részletesen bemutatom a két portált.

4.1 Adatelérési réteg (DAL)

Az adatelérési rétegem felelős az adatok perzisztens tárolásáért. Az adatbázis EF Core Code First megoldásával hoztam létre. A réteg felépítése a következő:



14. ábra: DAL felépítése

Az Entities mappa tartalmazza az entitás osztályokat, melyek az adatbázis sémát alkotják. A konfigurációt annotációk segítségével végeztem el. Ezt szemléltetem a *ServicePlace* entitásomon.

```
public Guid Id { get; set; }
[Required]
[StringLength(200)]
public string Name { get; set; }
[Required]
public Guid ServiceId { get; set; }
public Service Service { get; set; }
```

A *Name* tulajdonság maximális hossza a *StringLength* attribútumnak köszönhetően 200 karakter. Ez adatbázis szinten úgy képződött le, hogy az oszlop *nvarchar(200)* típusú lett, illetve a *Required* miatt nem enged *null* értéket. Ha a kódban ezek sérülnek a mentés során, akkor *DbEntityValidationException* kivétel képződik.

További konfigurációkat a fluent API [14] segítségével végeztem el. A *DbContext OnModelCreating* függvényében van lehetőség ezt használni. Összetett kulcsok, illetve kapcsoló táblák beállításához vettem igénybe.

```
builder.Entity<BookingPosition>()
    .HasKey(x => new { x.BookingId, x.ServicePlacePositionId });

builder.Entity<BookingPosition>()
   .HasOne(bc => bc.Booking)
   .WithMany(b => b.BookingPositions)
   .HasForeignKey(bc => bc.BookingId)
   .OnDelete(DeleteBehavior.ClientSetNull);

builder.Entity<BookingPosition>()
  .HasOne(bc => bc.ServicePlacePosition)
  .WithMany(b => b.BookingPositions)
  .HasForeignKey(bc => bc.ServicePlacePositionId)
  .OnDelete(DeleteBehavior.ClientSetNull);
```

Ebben a kódrészletben a *BookingPosition* entitást konfigurálom be. Ebből képződik egy kapcsoló tábla, ami összeköti a *Booking* és a *ServicePlacePosition* entitásokat. Nagy adatbázis esetén ezek a konfigurációk nagyon hosszúak lehetnek, átláthatatlanná válnak. Erre megoldást szolgáltat az *IEntityTypeConfiguration<TEntity>* interfész. A *Configure* függvényében végezhető el az entitásra vonatkozó beállítás.

```
public class UserEntityConfiguration : IEntityTypeConfiguration<User>
{
    private readonly ISeedService _seedService;

    public UserEntityConfiguration(ISeedService seedService)
        => _seedService = seedService;

    public void Configure(EntityTypeBuilder<User> builder)
        => builder.HasData(_seedService.Users.Values.ToArray());
}
```

Az interfészt megvalósító osztályokat be lehet regisztrálni *OnModelCreating* függvényben és ott elvégzi a beállításokat.

```
builder.ApplyConfiguration(new UserEntityConfiguration(_seedService));
```

A Migrations mappa a Code First által generált migrációkat tartalmazza. Új migrációt a *Package Manager Console Add-Migration [Name]* parancsával lehet létrehozni.

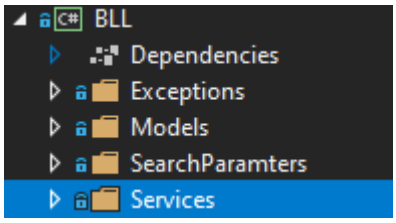
A *SeedService* osztályom az ősadatokkal való feltöltésért felelős. Példányosítja az ISeedService interfészem. Szerepe az adatbázis *OnModelCreating* függvényében betölteni az adatokat. Erre a *ModelBuilder* osztály *HasData* függvényét használja. Két szerepkör van a rendszeremben. Az **Administrator** az ügyintéző felhasználókat jelképezi, a **SystemAdmin** pedig a rendszerszintű adminisztrátorokat. Az ősadatok tartalmaznak mindkét szerepkörhöz egy felhasználót.

```
public interface ISeedService
{
    IDictionary<string, IdentityRole> Roles { get; }
    IDictionary<string, User> Users { get; }
    IList<IdentityUserRole<string>> UserRoles { get; }
    Dictionary<string, Image> Images { get; }
    Dictionary<string, ServiceType> ServiceTypes { get; }
    Dictionary<string, Service> Services { get; }
    Dictionary<string, Event> Events { get; }
    Dictionary<string, ServicePlace> ServicePlaces { get; }
    IList<ServiceEvent> ServiceEvents { get; }
    IList<ServicePlacePosition> ServicePlacePositions { get; }
    IList<EvenSchedule> EvenSchedules { get; }
}
```

A *SeedService*-n kívül még egy osztály segíti az adatok betöltését. A képek betöltését nem sikerült a *HasData* metódus segítségével betölteni, mivel azok *byte[]* típusúak. Ezért kellett a képek migrációjához egy külön *IImageService* interfész létrehoznom és implementálnom. Egy adatbázis kontextuson keresztül betölti a képeket, amelyekhez a rekordok már léteznek.

4.2 Üzleti logikai réteg (BLL)

Az üzleti logikai rétegem felépítése a következő:



15. ábra: BLL felépítése

A réteg alapját a szolgáltatások képzik. Ezek felelősek az adatok kezeléséért. Minden szolgáltatáshoz egy interfész tartozik, amit meg kell valósítania. Ez azért szükséges, hogyha változik az implementáció alatta (például adatbázis csere vagy EF helyett ADO.NET), akkor a feletette lévő réteget ne kelljen módosítani.

```
public interface IBookingService
{
    PendingBooking CreatePendingBooking(PendingBooking
        pendingBooking);

    Task<PendingBooking> CreatePendingBookingAsync(PendingBooking
        pendingBooking);

    Booking CreateBooking(Booking booking);

    Task<Booking> CreateBookingAsync(Booking booking);

    Task<PendingBooking> GetPendingBookingByClientIdAsync(Guid id);
}
```

Az interfészek metódusai szinkron, illetve aszinkron típusúak. A szolgáltatásoknak szükségük van egy *DbContext*-re, amit az ASP.NET Core függőség injektálással ad át. A műveleteket ezen a kontextuson keresztül végzik.

A Models mappa tartalmazza az üzleti logikai entitásokat. A réteghez tartalmaznak külön kivételek is, mint például a *BookingException*. Ez a foglalási kísérlet során keletkezhet.

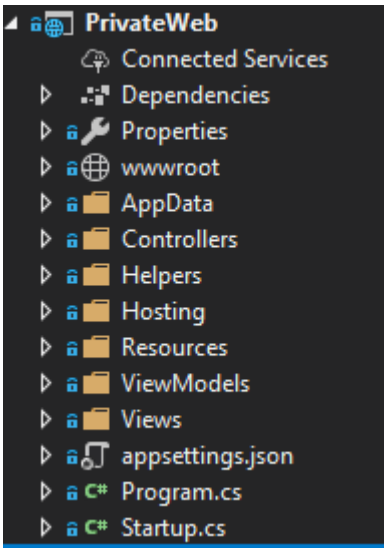
A listás megjelenítéseknél a szűrést a *SearchParameter* postfixel rendelkező osztályok segítik.

4.3 Ügyintézői portál

A belső webportál egy ASP.NET Core MVC alkalmazás. Az ügyintézői szerepkörhöz kapcsolódó funkcionalitásokat valósítja meg. Az alkalmazást az adatelérési és a belső UI alkotják.

4.3.1 Felépítés

Ebben a részben az ASP.NET Core MVC alkalmazásom felépítését szeretném bemutatni. A kérések a Controller osztályokhoz futnak be, melyek az MVC minta vezérlői. A modelleket a ViewModels osztályaim képviselek, a nézetek a Razor Pages-el készített .cshtml állományok.



16. ábra: Ügyintézői alkalmazás projektje

Az alkalmazás a *Program* osztály *Main* metódusával indul. Az ASP.NET Core által nyújtott *IWebHost* interfészhez létrehoztam egy extension method-ot[15], ami a futás idejű migrációt valósítja meg. Így nem kell a *Package Manager Console*-on kiadni az *Update-Database* parancsot és a képek betöltését sem kell külön elvégezni.

```
public async static Task<IWebHost> MigrateDatabase<TContext>(this IWebHost
host) where TContext : DbContext
{
```

[15] Kiegészítő metódus leszármazás nélkül.

```
    using (var scope = host.Services.CreateScope())
    {
        var serviceProvider = scope.ServiceProvider;
        var context = serviceProvider.GetRequiredService<TContext>();
        context.Database.Migrate();
        var imageSeedSevice =
              serviceProvider.GetRequiredService<IImageSeedService>();
        await imageSeedSevice.SeedImagesAsync();
    }
    return host;
}
```

A *Main* metódusom még kiegészül az AutoMapper nevű komponens bekonfigurálásával. Ehhez egy külön *MapperHelper* statikus osztályt valósítottam meg. A bekonfigurált *Mapper* osztály leképezi az adatbázis entitásaimat a webes *ViewModel* osztályokra.

```
public static void InitializeMapper()
{
    Mapper.Initialize(cfg =>
    {
        cfg.CreateMap<Service, ServiceViewModel>()
            .ForMember(dest => dest.Image, opt => opt.Ignore());
        cfg.CreateMap<ServiceViewModel, Service>()
            .ForMember(dest => dest.Image, opt => opt.Ignore())
            .ForMember(dest => dest.Id, opt => opt.Ignore());

        cfg.CreateMap<Event, EventViewModel>()
            .ForMember(dest => dest.Image, opt => opt.Ignore());
        cfg.CreateMap<EventViewModel, Event>()
            .ForMember(dest => dest.Image, opt => opt.Ignore())
            .ForMember(dest => dest.Id, opt => opt.Ignore());

        ...
    });
}
```

A *ViewModel* osztályok az adatbázis entitások webes leképezései, adatok megjelenítésére és egyéb feladatok elvégzésére szolgálnak. Ilyen például a

LoginViewModel, ami a bejelentkezési folyamat fontos eleme. Az osztály tulajdonságai *Display* attribútommal vannak ellátva, hogy a felületen a megfelelő label generálódjon hozzá.

Fontos még, hogy a *ViewModel* osztályok a megfelelő validációs attribútumokat megkapják, hogy a nézetekben is megjelenjenek a hibaüzenetek a hibásan megadott adatokra. Ezeknek jelentős szerepe van még a *Conroller* osztályban is, mivel ajax hívással bármilyen adat küldhető, ezért szükséges az adatokat szerver oldalon is validálni. Ezt a *ControllerBase ModelState* entitásának az *IsValid* tulajdonságával lehet megtenni.

4.3.2 Felhasználókezelés

A felhasználók kezeléséhez ASP.NET Core Indentity-t használtam. Az adatbázisban való tárolás miatt a kontextusom az *IdentityDbContext<User>* osztályból származik. Így létrejönnek a megfelelő táblák, ami az Identity komponenshez szükségesek.

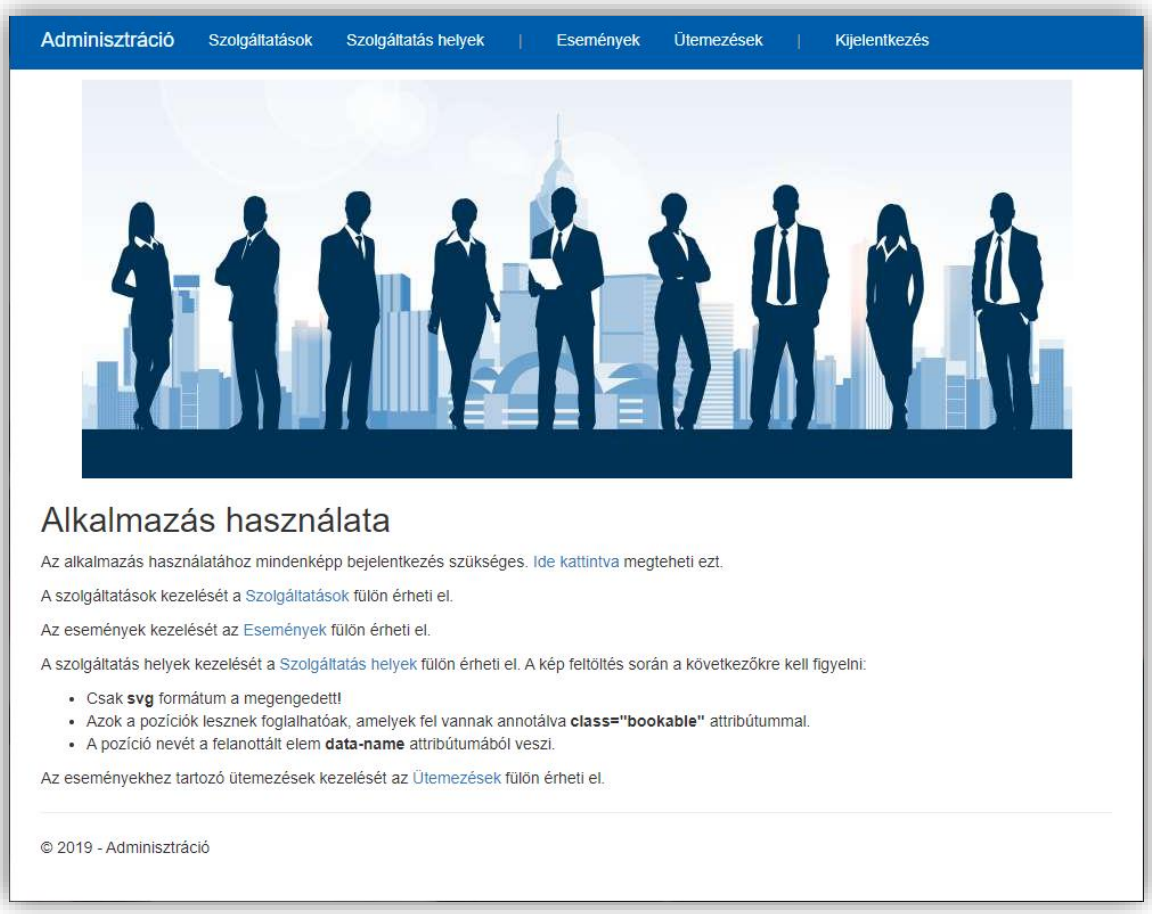
Az alkalmazásban minden művelethez bejelentkezés szükséges. A szerepkörök szétválasztását az *Authorize* attribútummal oldottam meg. A példámban az eseményekhez tartozó vezérlőt, csak az autentikált ügyintézők vehetik igénybe.

```
[Authorize(Roles = Roles.Administrator)]
public class EventsController : BaseController
{
       ...
}
```

4.3.3 Működés

Az elkészült alkalmazás csak bejelentkezés után használható. A bejelentkezett felhasználót a főoldal várja, ahonnan a menü sávon keresztül navigálhat tovább a kívánt oldalakra.

Minden fő entitás külön menüpontont kapott. Ezekhez tartozik egy listázó oldal, ahol a felhasználó által létrehozott elemek vannak. Itt van lehetőség új adatok felvételére. Minden elemhez van szerkesztési funkció, illetve törölhetők is.



17. ábra: Ügyintézői főoldal

Mivel az alkalmazás főleg CRUD műveletek elvégzésére szolgál, ezért nem mutatom be az összes entitáshoz kapcsolódó műveletet. A működést a szolgáltatás entitásomon keresztül szemléltetem.

4.3.3.1 Szolgáltatások kezelése

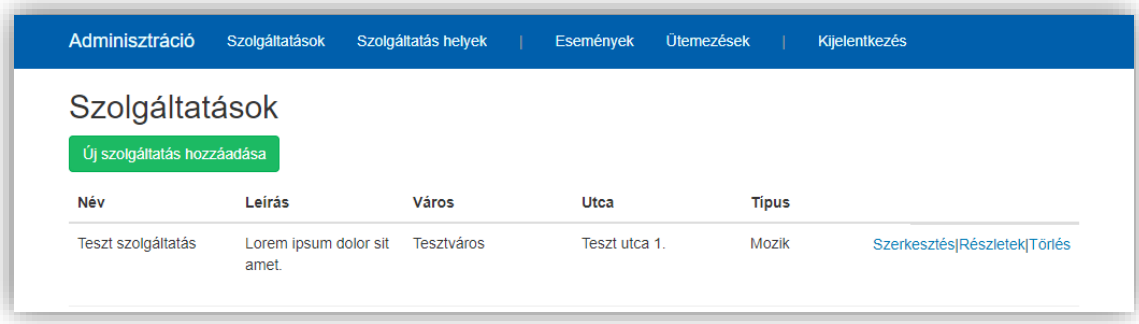
A szolgáltatások oldalra navigálva a *ServicesController Index()* metódusa fut le. Az oldalon a felhasználóhoz kapcsolódó szolgáltatások listája fog megjelenni. Minden adatbázis hívás try-catch-be van csomagolva lekezelve az esetleges hibákat. A hibás ágakat többnyire úgy kezeltem, hogy a *Controller* osztály *TempData* tulajdonságában helyeztem el egy hibaüzenetet és ezt jelenítem meg a felhasználónak. Azért választottam ezt, mert így a navigáció során sem veszik el az üzenet (ellentétben a *ViewBaggel*).

```
public async Task<IActionResult> Index()
{
    try
    {
```

```
        var bookingSystemDbContext = _context
            .Services
            .Include(s => s.Type)
            .Where(x => x.UserId == GetCurrentUserId());
        return View(await bookingSystemDbContext.ToListAsync());

    }
    catch (Exception e)
    {
        _logger.LogError(e.Message, CommonC.ErrorLoad);
        TempData["ErrorMessage"] = CommonC.ErrorLoad;
    }
    return View();
}
```

A működéshez szükséges szolgáltatásokat a keretrendszer szolgáltatja függőség injektálással. Ezek közé tartozik egy *UserManager* osztály, ami az Indentity része, egy naplózó szolgáltatás és az adatbázis kontextus.

Az oldal megnyitása után táblázatos formában megjelennek a szolgáltatások adatai. Itt van lehetőség új elem felvételére, illetve a már létrehozott elemek módosítására, törlésére.



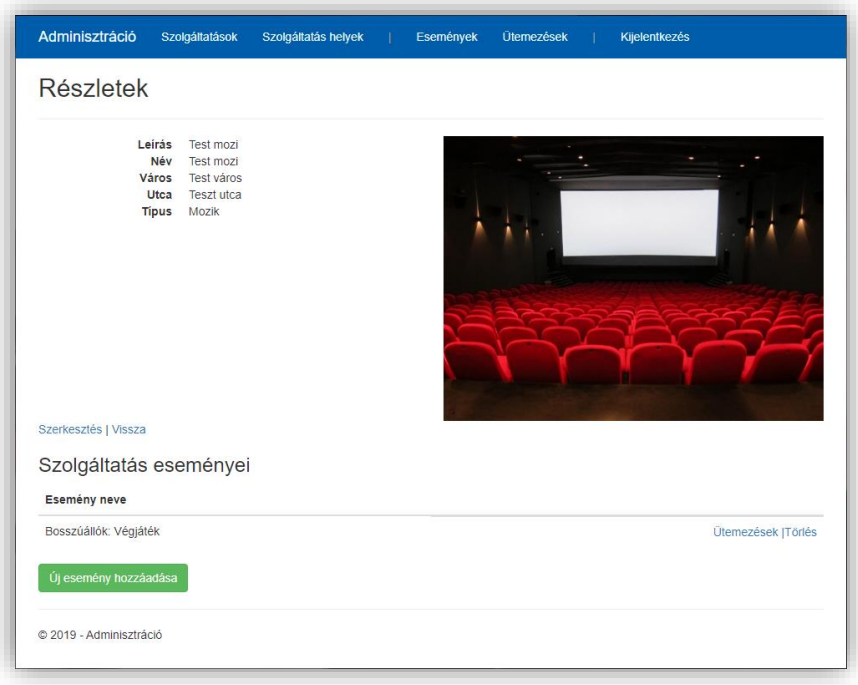
18. ábra: Szolgáltatás menüpont részlet

Minden művelethez két *Action* tartozik. Az egyik névkonvenció alapján megkeresi a generálandó nézetet, feltölti az adatlistákat és átnavigál az új oldalra. A párja, a POST művelet elvégzi a módosításokat az adatokon és visszanavigál a fő menüpontra sikeres akció esetén.

```
3 references | 0 requests | 0 exceptions
public async Task<IActionResult> Index()...

0 references | 0 requests | 0 exceptions
public IActionResult Create()...

[HttpPost]
[ValidateAntiForgeryToken]
0 references | 0 requests | 0 exceptions
public async Task<IActionResult> Create(ServiceViewModel serviceViewModel)...

0 references | 0 requests | 0 exceptions
public async Task<IActionResult> Edit(Guid? id)...

[HttpPost]
[ValidateAntiForgeryToken]
0 references | 0 requests | 0 exceptions
public async Task<IActionResult> Edit(Guid id, ServiceViewModel serviceViewModel)...

0 references | 0 requests | 0 exceptions
public async Task<IActionResult> Details(Guid? id)...


0 references | 0 requests | 0 exceptions
public async Task<IActionResult> Delete(Guid? id)...

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
0 references | 0 requests | 0 exceptions
public async Task<IActionResult> DeleteConfirmed(Guid id)...
```

19. ábra: CRUD műveletek

A rendszerben egy esemény nem szolgáltatáshoz kötött, hanem külön entitásként él. Ez azért van így, hogy egy szolgáltatónak egy eseményt csak egyszer kelljen felvenni és ezt hozzákötni a szolgáltatásaihoz (ha több is van). Az összekapcsolást a szolgáltatás részletező felületén keresztül érheti el.



20. ábra: Szolgáltatás részletező oldal

4.3.3.2 Szolgáltatás helyek létrehozása

A szolgáltatás helyek kis mértékben eltérnek a többi entitástól. Fontos volt számomra, hogy a foglalási oldal esztétikusan nézzen ki, illetve univerzális legyen. Erre kellett egy megoldást találnom.

Egy szolgáltatás hely entitás két dologból épül fel. Az egyik a helyiség foglalható egységei, illetve a helyiséget leíró kép (foglalható egységek összessége, rendezett formában). Azt kellett tehát meghatároznom, hogy milyen leképezést valósítok meg a képhez és a pozíciókhoz.

Azért esett a választásom az svg formátumra, mert:

1. bármilyen kép elkészíthető ebben a formában,
2. vektoros grafika (jól skálázódik, nagy kijelzőn is),
3. XML alapú (feldolgozásnál ez fontos volt),
4. támogatja az animációkat és az eseménykezelőket.

Az első két pont ügyintézői és felhasználói szempontból fontos. A harmadik és a negyedik pont fejlesztői oldalról voltak hasznosak.

A kép és a foglalható egységek közötti leképzésem a következő lett: minden olyan svg elem (circle, rect, ellipse, stb.), aminek van *class* attribútuma, és tartalmazza a **bookbable** kifejezést, abból létrejön egy pozíció. A hely neve a **data-name** tulajdonság értéke lesz.

A szolgáltatás hely svg állományát feldolgozó metódusom a következő:

```
private byte[] ProcessLayoutImage(Stream imageStream, Guid servicePlaceId)
{
    var xdoc = XDocument.Load(imageStream);
    var bookablePositions = xdoc
        .Descendants()
        .Where(x => x
            .Attributes()
            .Any(y => y.Name.LocalName.Equals("class")
                      && y.Value.Contains(BOOKABLE)));
    foreach (var bookablePosition in bookablePositions)
    {
        var layoutPosition = new ServicePlacePosition
```

```
            {
                  Id = Guid.NewGuid(),
                  Name = bookablePosition.Attribute(DATA_NAME).Value,
                  ServicePlaceId = servicePlaceId
            };
            var idAttribute = new XAttribute(DATA_POSITION_ID,
                                layoutPosition.Id.ToString().ToLower());
            bookablePosition.Add(idAttribute);
            _context.Add(layoutPosition);
      }
      var settings = new XmlWriterSettings
                  {
                     OmitXmlDeclaration = true,
                     Encoding = Encoding.UTF8
                   };
        using (var memoryStream = new MemoryStream())
        using (var xmlWriter = XmlWriter.Create(memoryStream, settings))
        {
            xdoc.WriteTo(xmlWriter);
            xmlWriter.Flush();
            return memoryStream.ToArray();
        }
}
```

A metódus bemeneteként kapott kép állományt XML dokumentumként betöltöm. Ezt követően kiválasztom azokat a gyerekeket, akiknek megvan a **bookable** osztályuk. Minden ilyen gyerekből lesz egy *ServicePlacePostion* entitás, amit hozzáadok a kontextushoz. A feldolgozandó elemet kiegészítem egy új attribútummal (*data-position-id*), aminek az értéke a hozzákapcsolódó pozíció azonosítója lesz. Ez azért szükséges, hogy a kliens oldalon az eseménykezelő függvényben azonosítani tudjam. A kiegészített képből és a felvett pozíciókból jön létre az új szolgáltatás hely a rendszerben.

4.4 Foglaló portál

A foglaló portál egy ASP.NET Core szerverből és egy Angular kliensből felépülő webalkalmazás. A normál felhasználói szerepkörhöz kapcsolódó funkcionalitásokat valósítja meg. Az alkalmazás bemutatását először a felépítés bemutatásával kezdem,

külön kitérve a szerver és kliens oldalra. Ezt követően részletesen szemléltetem a működést a felhasználói felületekkel és a hozzájuk kapcsolódó kódrészletekkel.

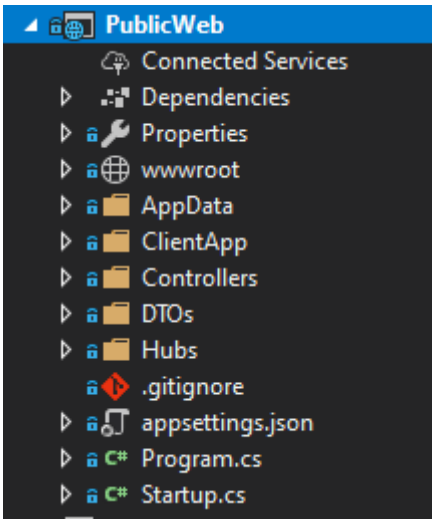
4.4.1 Felépítés

Az alkalmazás váza egy Visual Studioba épített Angular projekt sablon. Ez egyenértékű azzal, mintha egy ASP.NET Core projektet készítenék, ami backendként viselkedik és egy Angular CLI projektet, ami a felhasználói felület. Ez a sablon típus azt a kényelmet nyújtja, hogy egyszerre hosztolja két alkalmazást, így egyként lehet buildelni és publishelni.

Az ASP.NET Core alkalmazás felelős az adatelérésért és az élő kommunikáció lebonyolításáért. Az Angular app a ClientApp nevű almappában tartózkodik, a célja pedig minden UI feladat megvalósítása.

4.4.1.1 Szerver felépítése

A szerverem felépítése nagyon egyszerű. Egy ASP.NET Core webalkalmazás, ami hosztolja az Angularos kliens alkalmazást. Szolgáltatja az adatok elérését a WebApi kontroller osztályokon keresztül és kezeli az élő kommunikációt.



21. ábra: Szerver felépítése

4.4.1.2 Élő kommunikáció

Az élő kommunikáció megvalósításához a SignalR komponenst választottam. A könyvtára alapját a *Hubok* szolgáltatják. Az alkalmazásomban egy *Hubot* készítettem, amit a foglalási oldalon használtam. A SignalR API lehetővé teszi, hogy a szerverről

meghívjuk a kliens oldali kódot. Ezen felül definiálhatunk metódusokat, amiket a kliensek hívhatnak meg.

Az én *Hub* osztályomban két függvényt nyújt. A *SendPendingBooking* metódus az összes csatlakozott kliensnek elküldi az újonnan létrejövő zárolási kérelmeket. A *SendBooking*, a párja, ami a foglalásokat továbbítja a kliens oldalra.

```
public class BookingHub : Hub<IBookingHub>
{
     public async Task SendPendingBooking(PendingBookingDTO pBooking)
     {
         await Clients.All.RecieveNewPendingBooking(pBooking);
     }
     public async Task SendBooking(BookingDTO booking)
     {
         await Clients.All.RecieveNewBooking(booking);
     }
}
```

Ahhoz, hogy típusosak legyenek a hívások létrehoztam egy interfészt, az *IBookingHub*-ot. Ha a *BookingHub* osztály nem a *Hub<IBookingHub>* osztályból származna, akkor a függvények törzse az alábbi módon nézne ki:

```
     await Clients.All.SendAsync("RecieveNewBooking", booking);
```

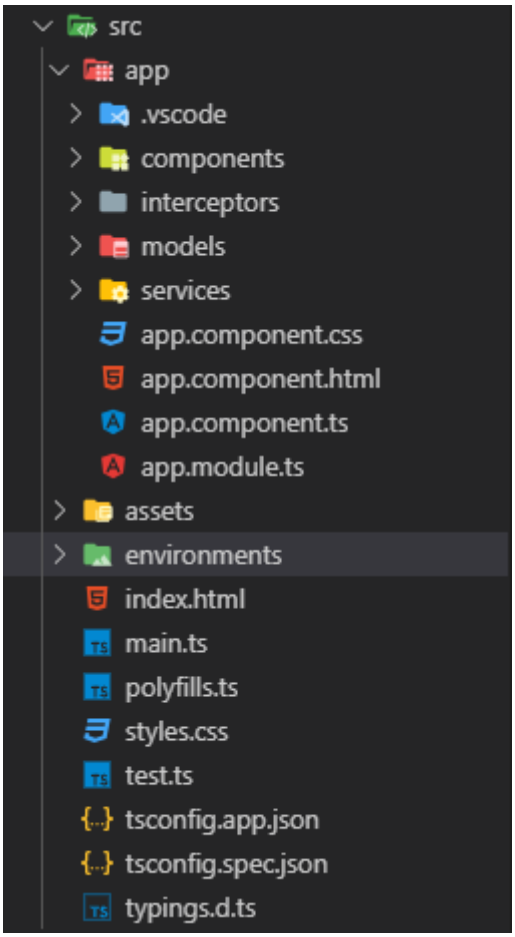
Az interfészemen definiált függvényekre kell a kliens oldalon feliratkozni, mert ilyen típusú eseményeket kaphatnak.

```
connection.on('RecieveNewPendingBooking', (pBooking: PendingBooking) => {
      ...
});
```

A *Controller* osztályokból is meghívhatók a *Hubok* metódusait, egy *IHubContext*-en keresztül. Ennek az eléréséhez a *Startup* osztály *Configure* függvényében kell beregisztrálni egy ilyen elemet, ezután a keretrendszer függőség injektálással már tudja szolgáltatni.

4.4.1.3 Kliens alkalmazás felépítése

A diplomamunkám része volt az Angular kliens oldali technológia megismerése. Törekedtem a platformnak megfelelő felépítést alkalmazni az alkalmazásom elkészítése során.



22. ábra: Kliens felépítése

Az elkészült program egy Single-Page Application, a gyökér eleme az *AppModule* nevű modul. Az oldal vázát az index.html adja, ahol az *app-root* tag helyére jön létre az alkalmazás. Az oldalak közötti navigációt a beépített *RouterModule* biztosítja. A modul importálásánál meg kell adni, hogy melyik komponens, melyik útvonalra példányosodjon. Ha a felhasználó megcímzi ezt az url-t, akkor a *router-outlet* tag helyére létrejön a beregisztált komponens. Az munkám fő része az app mappa tartalma, itt helyezkednek el a megvalósított komponenseim, szolgáltatásaim és a modell osztályok / interfészek.

A WepApi-val való kommunikálás során számos dto-t kellett létrehoznom, amik TypeScriptes osztályokra, illetve interfészekre képződtek le. Ezek közül szeretném

kiemelni a *JsonResult<T>* , ami egy generikus osztály. A hálózati hívásaim fontos eleme. Három tulajdonságból épül fel: a *Message* a szerver üzenete, a *Success* a hívás sikerességét jelzi és egy *Result*, amibe a generikus adat helyezhető el.

A szolgáltatásaim főként a WebApi kommunikáció lebonyolítását segítik. Azért fontos ezeket a kódrészleteket kiszervezni, mert így a kód modulárisabb lesz. Minden szolgáltatás el van látva a *@Injectable()* dekorátorral, így tudja a rendszer függőség injektálással nyújtani a többi osztálynak.

A hálózati hívások lebonyolításához az Angular *HttpClient* osztályát használtam. Ez elfedi és egyszerűsíti a *XMLHttpRequest* objektum használatát. A szolgáltatásaimnak továbbá szükségük van még egy címre is, amin a szervert elérik. Ezt is a függőség injektálással kapják meg. A hívások visszatérési értéke egy *Observable<T>*, ami az aszinkronitást segíti. Ez a RxJs könyvtár része. Az alábbi kódrészlet az *EventService* szolgáltatás osztályom törzsét, illetve egy függvényét mutatja be.

```
@Injectable()
export class EventService {
  httpClient: HttpClient;
  url: string;
  ...
  getEvent(id: string): Observable<JsonResult<Event>> {
    return this.httpClient
            .get<JsonResult<Event>>(this.url + `api/Event/${id}`);
  }
}
```

A components mappában helyezkednek el az elkészített komponenseim. Mindegyiknek külön almappája van, melyben négy állomány van. A *.component.ts* kiterjesztésű fájl tartalmazza magát a komponens osztályt. Ezeket a @Component dekorátor határozza meg. A komponenshez tartozó többi állomány: a sablon, a stíluslap és a karma teszt.

4.4.2 Működés

Az alkalmazást futtatva először elindul az ASP.NET Core szerver, majd ez hosztolja a kliens alkalmazást a ClientApp mappából. Ez elszigetelten fut és a felhasználók kérései mind az Angular klienshez futnak be. A működést az elkészült komponenseim részletezésével szeretném bemutatni.

4.4.2.1 AppComponent (`app-root`)

Az *AppModule* főmodulomban megadott *bootstrap* komponens az *AppComponent*. Az indulás során ez lesz az első komponens, ami példányosodni fog. A sablonja nyújtja az oldalak vázát. Tartalmazza a *NavMenuComponent* komponenst, a *LoaderComponent* komponenst és a *RouterOutlet* direktívát. Az utóbbi a *@angular/router* része. A routing során ide töltődnek be dinamikusan a komponensek.

4.4.2.2 NavMenuComponent (`app-nav-menu`)

Az alkalmazás menüjét megvalósító komponens. A sablonja a bootstrap könyvtár segítségével épül fel. A navigáláshoz a *[routerLink]* direktívát használtam fel.

```
<li class="nav-item" [routerLinkActive]='["link-active"]'>
     <a class="nav-link" [routerLink]='["/event-list"]'>Események</a>
</li>
```

4.4.2.3 LoaderComponent (`app-loader`)

Az alkalmazásomban számos adat a WebApi-n keresztül érhető el. Amíg a letöltés folyamatban van, egy töltőképernyőt kell mutatni. Ennek a megvalósítására egy beavatkozót (interceptor), egy szolgáltatást és egy komponenst hoztam létre.

A komponens tartalmaz egy *Subject<boolean>* típusú tulajdonságot. Az értéke alapján mutatja, illetve rejti el a töltőképernyőt. A tulajdonság egy eseménykezelőként működik, ami a szolgáltatás osztályhoz van kötve.

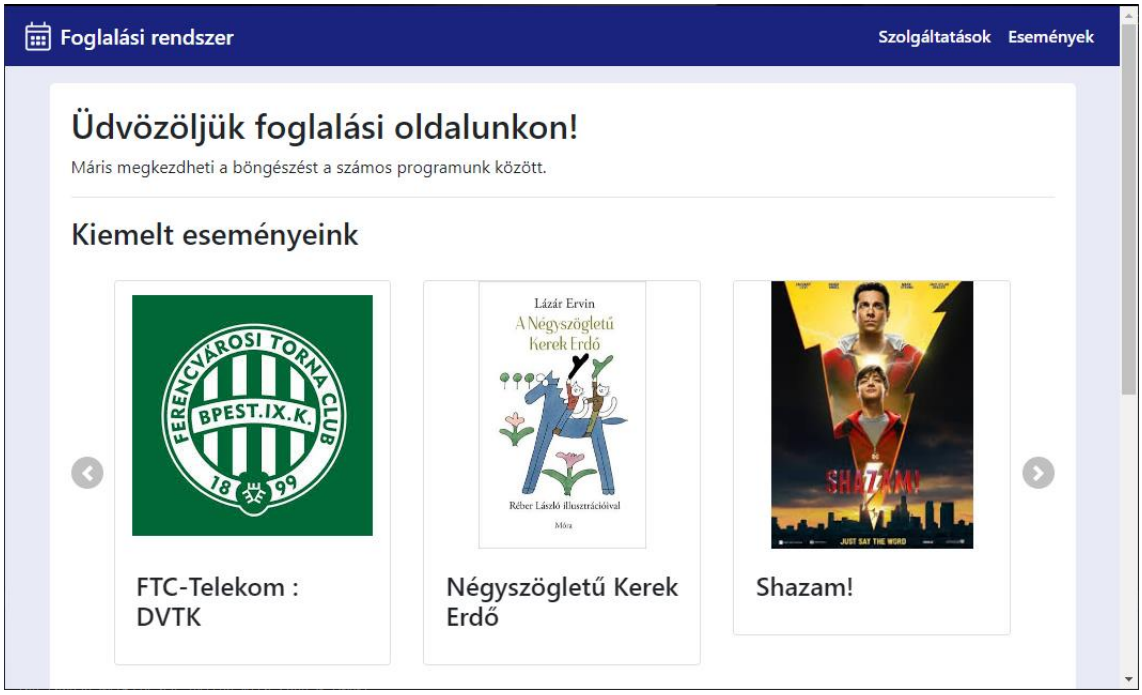
A *LoaderInterceptor* osztályom implementálja *HttpInterceptor* interfészt. Az *intercept* metódusa beékelődik minden hálózati híváshoz. Amikor egy kérés elindul a szolgáltatás *show* függvényét hívom. Ez a *Subject* objektumon egy *next()* metódust hív, amit lekezel a komponens.

4.4.2.4 HomeComponent (`app-home`)

Ez a komponens az alkalmazás nyitóoldalát valósítja meg, implementálja az *OnInit* interfészt. Ez az interfész az Angular része, az életciklusok kezelésére szolgál. Az *ngOnInit* metódusa akkor fut le, amikor az összes adatkötés inicializálódott.

A *HomeComponent* betöltés után letölti a szerverről a top öt eseményt és a top öt szolgáltatást. Amíg az adatok nem töltődtek le a szerverről, addig egy placeholder szöveg látszik. Ezt az Angular *ngIf direktívájával valósítottam meg.

```
    <ngb-carousel *ngIf="services; else serviceLoadingBlock">
```



23. ábra: Főoldal

Amint az adatok a szerverről megérkeztek, azokat a felületen meg is jeleníti. A szolgáltatásra kattintva a hozzátartozó események listájára navigál az alkalmazás.

```
<div class="col-md-4" [routerLink]="[ '/event-list', services[0].id ]">
    <div class="card grow">
        <img class="card-img-top" src="/api/image/{{services[0].image.id}}">
        <div class="card-body">
            <h4 class="card-title">{{services[0].name}}</h4>
        </div>
    </div>
</div>
```

A képeket mind adatbázisban tárolom, a hozzájuk tartozó entitásom az *Image*. Készítettem egy API hívást, ahol egy azonosító alapján le lehet kérdezni őket. Ez látszik az *image* tag forrás attribútumában. A vezérlő url címe a /api/image és *{{services[0].image.id}}* a szolgáltatáshoz tartozó kép azonosítója. Ha nincs megadva, akkor egy alapértelmezett képet adok vissza, amit az AppData mappából olvasok fel. Ha megvan adva az azonosító, akkor az *IImageService* szolgáltatáson keresztül a *GetImageAsync* metódussal felolvasom az entitás az adatbázisból. Mindkét esetben egy *FileContentResult* a visszatérési érték.

4.4.2.5 ServiceListPageComponent (`app-service-list-page`)

A rendszeremben a két legfőbb entitás az a *Service* és az *Event*. Az első reprezentálj a szolgáltatásokat (például egy mozi), míg a második a különféle eseményeket (például egy színházi előadás). A továbbiakban szeretném bemutatni a hozzájuk tartozó oldalak működését.

A szolgáltatások főmenüpontra kattintva a *RouterModule* betölti a *ServiceListPageComponent* komponensem. Az osztály implementálja az *OnInit* interfészt és az *ngOnInit* életciklus függvényben betölti az adatokat. Első lépésben a szolgáltatás típusok kerülnek letöltésére, amik a szűréshez szolgálnak (sport, mozi, stb.). Ez egy rendszer táblából jön, aminek a neve *ServiceTypes.* Új típusok esetén könnyen bővíthető a rendszer fejlesztés nélkül is.

A keresési mező feltöltése után lekérdezem a szolgáltatások listáját is. Jelenleg az összes adat letöltődik, a lapozás kliens oldalon van megvalósítva az *ngx-pagination* komponens segítségével. Ez egy könnyen konfigurálható Angular megoldás listák kezelésére.

A keresés megvalósításához a saját *ServiceSearchParamter* interfészemet használom. Ebből van egy példány kötve a felületi kereső mezökhöz. Amint a felhasználó rákattint a keresés gombra lefut az *onSearch* metódusom a komponens osztályban. Ez a *ServiceService* szolgáltatáson keresztül a *getServices(searchParamter: ServiceSearchParamter)* hívással lekérdezi a szerverről az adatokat. Az esetleges hibák kezelésére az *Observable* objektum *error* callback metódusa szolgál, a sikeres ág kezelésére pedig a *next* callback metódus. A szerver oldalon is keletkezhetnek hibák, például megszakad az adatbázis kapcsolat, stb.. Erre a megoldásom a *JsonResult* osztályom, aminek a *Success* tulajdonsága határozza meg a hívás sikerességét. Az *onSearch* metódusom kódja a következő:

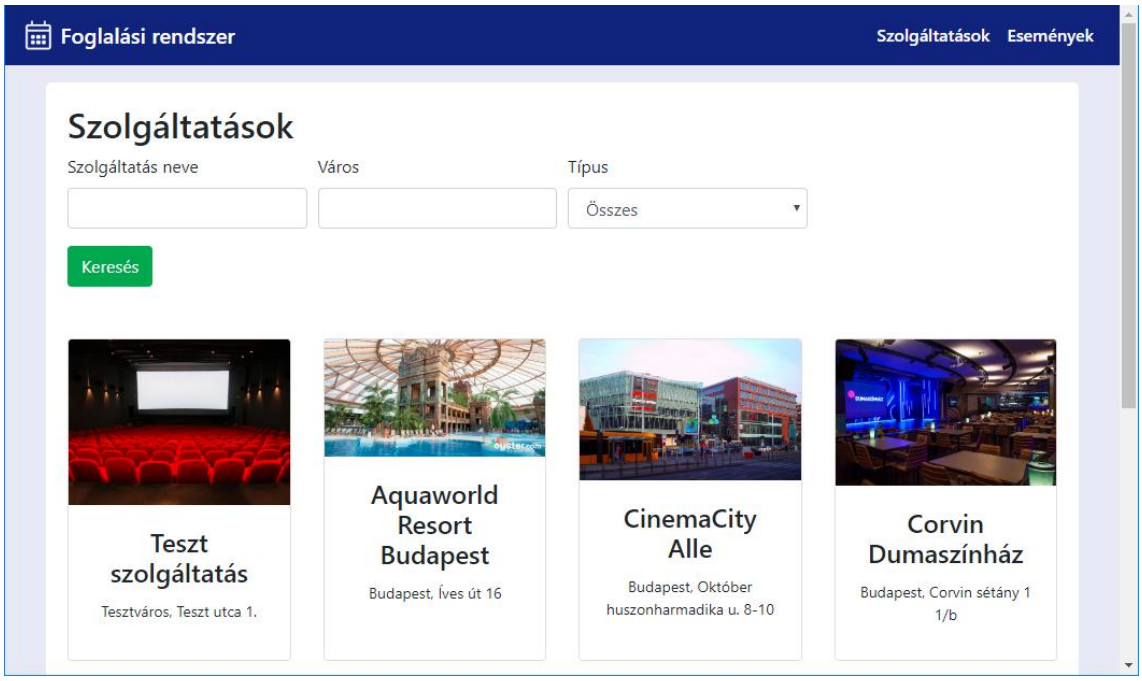
```
onSearch() {
  this.serviceService
      .getServices(this.serviceSearchParameter)
       .subscribe(result => {
         if (!result.success) {
           this.htmlHelper.showErrorMessage(result.message);
         }
         this.services = result.result;
      }, error => {
```

```
        this.htmlHelper
            .showErrorMessage('Hiba a szolgáltatások betöltése során.');
  });
}
```

Az adatok letöltése a *ServiceController GetServices* metódusán keresztül megy. Ez a /api/Service címen érhető el. A metódus a *ServiceService* BLL osztályon keresztül lekérdezi a szolgáltatásokat és a *Mapper* osztály *Map* hívásával átalakítja őket egy DTO osztállyá. Az *Ok* függvény JSON formátummá alakítja az adatokat.

```
public async Task<IActionResult> GetServices(ServiceSearchParameter
searchParameter)
{
    var result = new JsonResult<List<ServiceDTO>>();
    try
    {
        result.Result =
            (await _service.GetServicesAsync(searchParameter))
            .Select(x => _mapper.Map<Service, ServiceDTO>(x))
            .ToList();
        result.Success = true;
    }
    catch
    {
        result.Message =
            "Ismeretlen hiba a szolgáltatások letöltése során.";
    }
    return Ok(result);
}
```

A megjelenítés itt is kártyás formában történik, mint a főoldalon. Egy elemre kattintva a hozzá tartozó események oldalára navigál az alkalmazás.



24. ábra: Szolgáltatások oldal

Az események oldalára két féle képpen is el lehet jutni. Az egyik a főmenüponton keresztül, ekkor az összes olyan esemény megjelenik, amelyik már legalább egy szolgáltatáshoz hozzá van kötve. A másik irány, amikor egy szolgáltatáson keresztül navigál ide az alkalmazás. Ilyenkor a szolgáltatáshoz tartozó események listája jelenik meg.

4.4.2.6 EventListPageComponent (`app-event-list-page`)

Az események oldal működése nagyon hasonló a szolgáltatások oldaléhoz. A menüponthoz tartozó komponens az *EventListPageComponent*, ami szintén implementálja az *OnInit* interfészt. Az eltérés ott van, hogy a szolgáltatásra való szűrést az url-ben lévő azonosító alapján teszi meg. Ez *RouterModule ActivatedRoute* objektumán keresztülérhető el.

```
this.searchParameter.ServiceId = this.route.snapshot.paramMap.get('id');
```

Az adatok letöltése után itt is kártyás formában jelennek meg az események. Az ehhez kapcsolódó sablonom a következő:

```
<div class="row text-center mt-5" *ngIf="events.length > 0; else noItemBlock">
    <div class="col-lg-3  mb-3" *ngFor="let myevent of events | paginate: {
itemsPerPage: 8, currentPage: page }">
        <div class="card grow" [routerLink]="[ '/event', myevent.id ]">
```

```
            <img  class="card-img-top"  src="/api/image/{{myevent.image.id}}"
alt="Esemény kép">
            <div class="card-body">
                <h4 class="card-title">{{myevent.name}}</h4>
            </div>
        </div>
    </div>
</div>
<div *ngIf="events.length > 0">
    <pagination-controls (pageChange)="page = $event" previousLabel="Előző"
nextLabel="Következő" class="pager">
    </pagination-controls>
</div>
```

A szerver oldalon a hívás a következő módon működik. A POST törzsében lévő keresési paraméterekkel meghívja a szolgáltatás interfész *GetEventsAsync* metódusát. Az implementációban az adatbázis kontextus *Events* tulajdonságán keresztül LINQ segítségével leszűröm a megfelelő adatokat.

```
if (searchParameter.ServiceId.HasValue)
    q = q.Where(x => x.ServiceEvents
            .Any(y => y.ServiceId == searchParameter.ServiceId));
if (!string.IsNullOrEmpty(searchParameter.Name))
    q = q.Where(x => x.Name.Contains(searchParameter.Name));
if (searchParameter.BeginDate.HasValue)
    q = q.Where(x => x.EvenSchedules
            .Any(y => y.From >= searchParameter.BeginDate));

return await q.ToListAsync();
```

A szolgáltatás által visszaadott adatokat átalakítom DTO osztállyá és ezt küldöm vissza a kliens oldalnak.

4.4.2.7 EventPageComponent (`app-event`)

Ez a komponens egy esemény részletező felületét nyitja meg. Az esemény listázó oldalról érhető el. Az url-ben átadott azonosító alapján letölti az esemény adatait a szerverről. Amennyiben a szolgáltatás azonosítója is adott, akkor egyből betölti a hozzá tartozó ütemezéseket is táblázatos formában.

Az egyes időpontok a szerveren *DateTime* formátumban tárolódnak. A megfelelő formában történő megjelenítéshez az Angular *date* pipe-ot[16] használtam.

```
<td>
   {{dateGroup.date | date:'yyyy.MM.dd'}}
 </td>
```

Az oldal fő célja, hogy a felhasználó ki tudja választani, hogy melyik szolgáltatásban szeretne foglalni, illetve annak melyik időpontjára. A megjelenítésre táblázatos formát választottam. Az időpontok naponként csoportosítva jelennek meg egy sorban. Egy elemet kiválasztva átnavigál az alkalmazás a foglalási nézetre.

25. ábra: Esemény részletező oldal

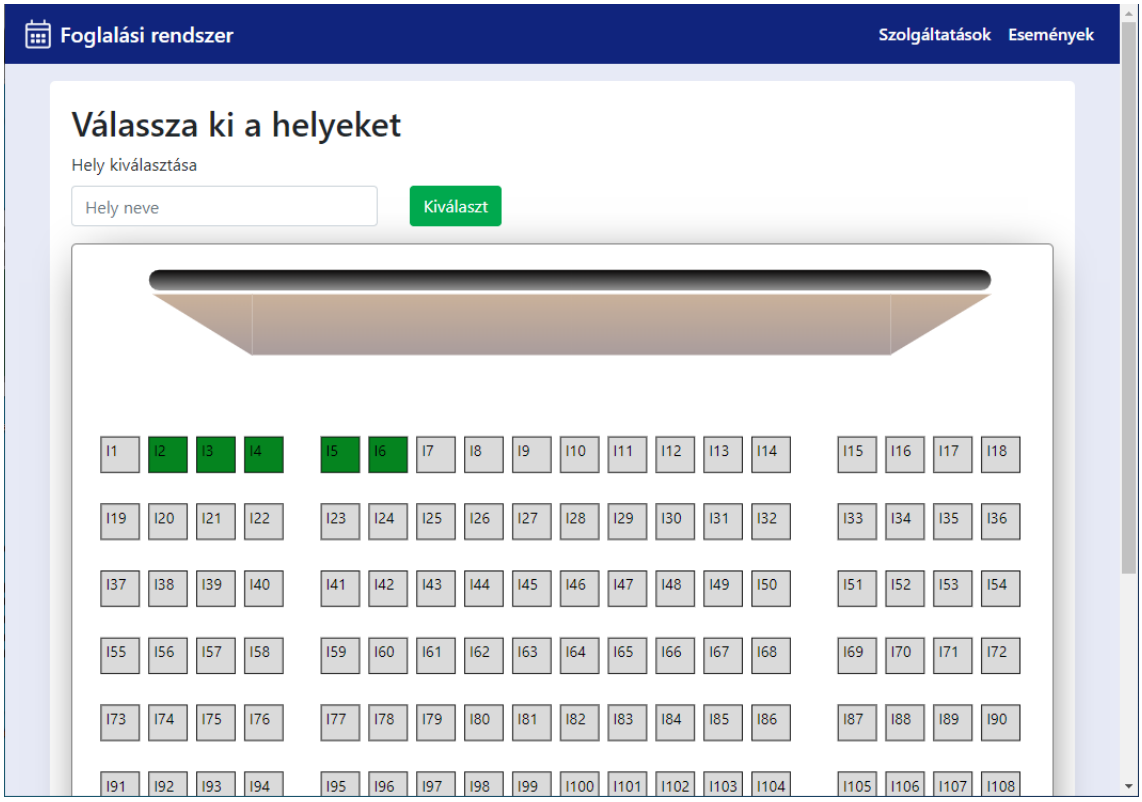
4.4.2.8 EventScheduleComponent (`app-event-schedule`)

Ez a komponens az *EventSchedule* entitáshoz készült. Feladata a zárolások és foglalások megjelenítése valós időben, továbbá pozíciók választása. Az oldalon megjelenik a szolgáltatás hely rajza, amit az ügyintéző portálon töltöttek fel.

A komponens implementálja az *AfterViewInit, OnInit, OnDestroy* interfészeket. Az oldal a betöltés után a hivatkozott időpontot lekérdezi a szerverről, a hozzá kapcsolódó foglalásokkal és zárolásokkal.

[16] Transzformációt végző objektum, megjelenítéshez [17].

```
ngAfterViewInit() {
  const id = this.route.snapshot.paramMap.get('id');
  this.scheduleService.getSchedule(id).subscribe(result => {
    if (!result.success) {
      this.htmlHelper.showErrorMessage(result.message);
    }
    this.schedule = result.result;
    const coloredSVG =
           this.colorSVGElements(this.schedule.servicePlace.layoutImage);
    this.servicePlaceSvg = this.sanitizer
      .bypassSecurityTrustHtml(coloredSVG);
  }, error => {
    this.htmlHelper.showErrorMessage('...');
  });
}
```



26. ábra: Foglalási oldal

Mivel a szolgáltatás hely képe egy svg, ezt html darabként töltöm be az oldalba. A kapott kép még nem tökéletes, nincsenek előre megjelölve rajta a foglalások, ezért ezeket dinamikusan kell feldolgozni. Erre szolgál a *colorSVGElements* metódusom. Ez egy olyan függvény, ami folyamatosan újrarajzolja a foglalási képet. A metódus első

lépésként az svg állomány méreteit állítja úgy, hogy azok reszponzívak legyenek. Ezt követően feldolgozom a foglalásokat és a zárolásokat. Mivel a zárolási kérelmek fix ideig érvényesek, ezért a lejáratuk után már nem szabad őket kirajzolni. A feldolgozáshoz JQuery osztály könyvtárat használtam. A foglalt elemeket a *booked* osztállyal, a zárolt elemeket a *pending* osztállyal, illetve a foglalható elemeket *bookable* osztállyal láttam el.

Ezen az oldalon valósítottam meg az élő kommunikációt, ezért a SignalR komponenst ide kötöttem be. Az *ngOnInit* életciklus függvényben csatlakozok az elkészült *Hubomhoz,* mert ilyenkor már a kép betöltődött az oldalra. A kapcsolatot úgy építem fel, hogy beállítom rajta az automatikus újra csatlakozást. Amikor elkezdtem az implementálást, akkor még kézzel kellett ezt megvalósítani. Ekkor a SignalR a *@aspnet/signalr* csomag része volt, most pedig a *@microsoft/signalr*-é, ezért át kellett alakítsam ezt a részt az új könyvtárra. Ebben már meg van valósítva az automatikus újra csatlakozás.

```
const connection = new signalR.HubConnectionBuilder()
  .withUrl(`${this.baseUrl}bookingHub`)
  .withAutomaticReconnect()
  .build();
connection.start().catch(err => this.htmlHelper.showErrorMessage('...'));
const eventScheduleId = this.id;
connection.on('RecieveNewPendingBooking', (pendingBooking: PendingBooking)
=> {
  if (pendingBooking.eventScheduleId != eventScheduleId) return;
  this.schedule.pendingBookings.push(pendingBooking);
  const colored = this.colorSVGElements($(`#svg-holder`));
  $(`#svg-holder`).html($(colored)[0].innerHTML);
});
connection.on('RecieveNewBooking', (booking: Booking) => {
  if (booking.scheduleId != eventScheduleId) return;
  this.schedule.bookings.push(booking);
  $(`#svg-holder`).html(this.colorSVGElements($(`#svg-holder`)));
});
```

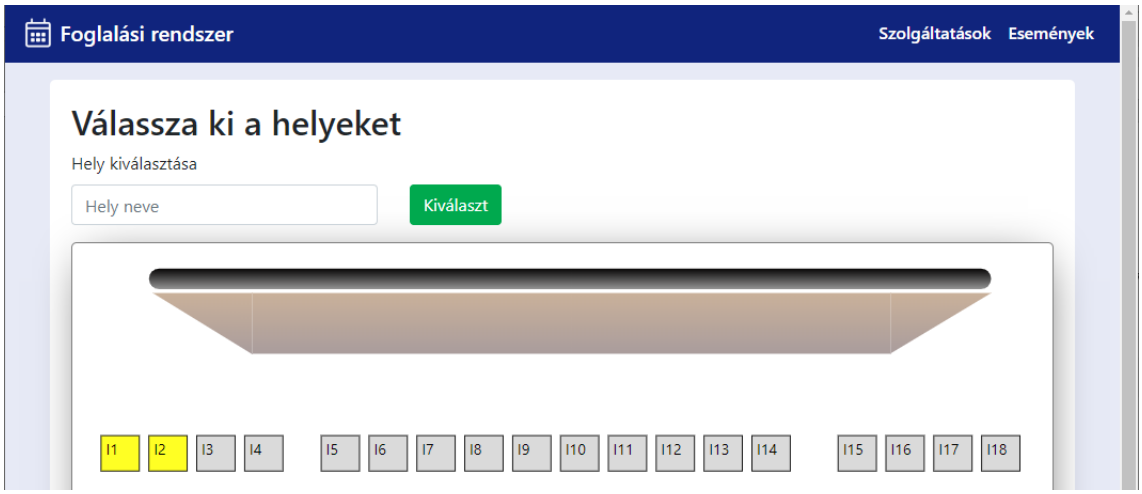
A kapcsolat létrehozása után feliratkozok a két eseményre, ami a foglaláshoz kapcsolódik. Az első az új zárolási kérelmet dolgozza fel, ami a kiválasztott időponthoz

tartozik. Ha jön egy új, akkor berakja a listába, és újra rajzolja a képet. A másodikban az új foglalás feldolgozása történik az előzőhöz hasonló módon.

Új foglalás létrehozásához ki kell választani a helyeket. Olyan pozíciók választhatók, amik el vannak látva a *bookable* osztállyal. A foglalt helyek a *booked* osztályt, a zároltak pedig a *pending* osztályt kapták. Ezek arra szolgálnak, hogy megjelöljék a felhasználóknak a szabad pozíciókat. Két módon lehet helyeket választani: a beviteli mezőbe meg lehet adni a pozíció nevét, vagy rákattint a pozícióra. Ezt követően lefut az eseménykezelőm, ami ellátja az elemet az *active* osztállyal, vagy leveszi róla, ha rajta van.

```
private createSelection(positionId: string) {
  if (this.containsBooking(positionId)) {
    return;
  }
  if (this.containsPending(positionId)) {
    return;
  }
  const containedElement = this.selectedPositions
    .find(x => x.id == positionId);
  if (containedElement) {
    this.selectedPositions =
      this.selectedPositions
        .filter(x => x.id != positionId);
    $(`#svg-holder [data-position-id="${positionId}"]`)
      .removeClass('active');
  } else {
    const position =
      this.schedule.servicePlace.layout
        .find(x => x.id == positionId);
    this.selectedPositions.push(position);
    $(`#svg-holder [data-position-id="${positionId}"]`)
      .addClass('active');
  }
}
```

Miután a helyek ki lettek választva, és a zárolási kérelmet a szerver is rögzítette, átnavigálok az adatok megadására szolgáló komponensre. A többi felhasználónak a következő módon jelenik meg.



27. ábra: Zárolt pozíciók

A komponensből való navigáláshoz a *RouterModule ActivatedRoute* osztályát használtam. Paraméterként a zárolási kérelem azonosítóját adom át.

```
this.router.navigateByUrl('/booking', { state: res.result });
```

4.4.2.9 BookingComponent (`app-booking`)

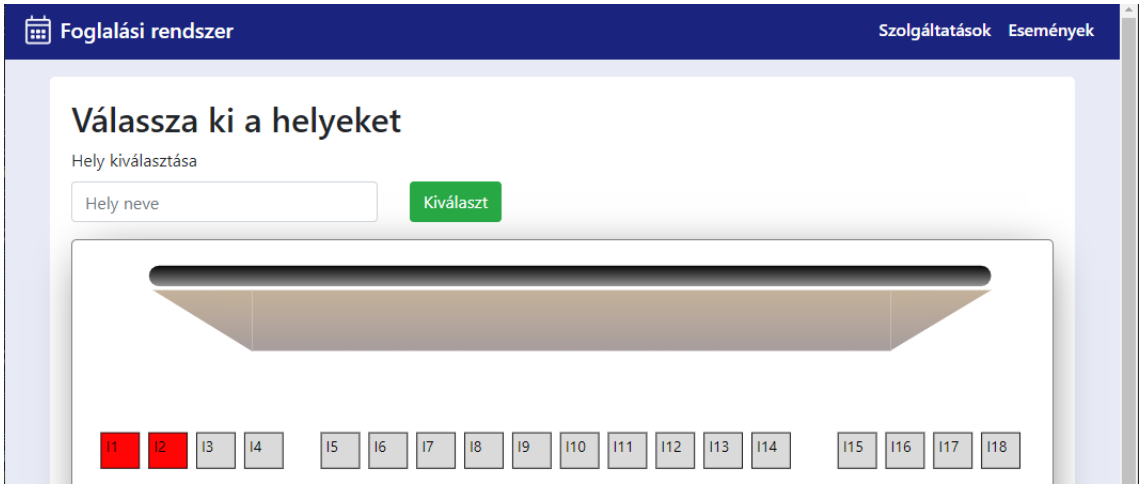
A komponens feladata a zárolási kérelemből és a bevitt adatokból egy foglalás létrehozása. A sablonjában egy *form* található, amiben a felhasználónak meg kell adni az adatait (adminisztráció céljából).

A megadásra egy rögzített intervallum van, ez a zárolási idő. Amennyiben ez lejár, visszaléptetem a foglaló oldalra, és a kiválasztott helyek felszabadulnak mindenkinek. Az idő monitorozásához a *setInterval* metódust használtam. A megadott callback függvényben kiszámolja, mennyi idő van vissza és adatkötéssel megjeleníti a felületen.

Ha a felhasználó helyesen átadata az adatokat, akkor rögzítésre kerül a foglalása. Ezt követően a *HubContexten* keresztül a kliensek megkapják az új létrejött foglalást és megpróbálok a megadott e-mail címre egy üzentet küldeni. Egy a felhasználó számára olvasható 8 karakteres azonosítót generálok és ezt adom vissza a kliens oldalnak. Ez kerül megjelenítésre, ezáltal könnyen hivatkozható, ha a foglalást le szeretnék mondani.

28. ábra: Sikeres foglalás

Miután egy sikeres foglalást adtunk le, a többi felhasználónak az alábbi módon jelenik meg.



29. ábra: Foglalt helyek megjelölése

4.5 Tesztelés

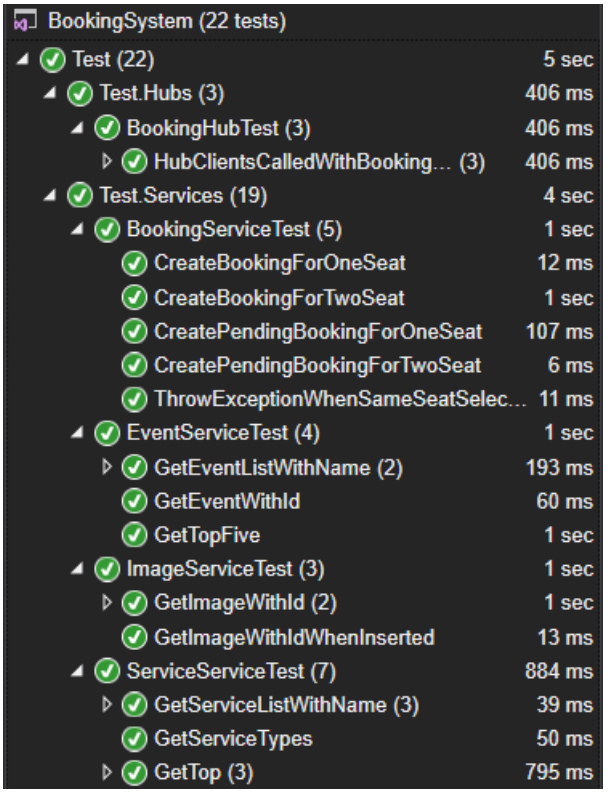
Az elkészült alkalmazásomat két módon teszteltem. Az első a manuális út volt, minden új funkció bevezetése után vizsgáltam annak működését. A másik mód unit tesztek készítése volt, melyek egy xUnit teszt projektbe kerültek.

Két fő csoport volt, amiket tesztelni kellett. Az elsőbe tartoznak az elkészült szolgáltatás osztályok. A korábban elkészített *SeedService* osztályomban lévő minta adatok segítségével végeztem a tesztelést. Ahogy korábban említettem, a szolgáltatás osztályaim egy darab adatbázis kontextust várnak a konstruktorukban. Ezt a keretrendszer szolgáltatta, de a tesztek során erre nem volt lehetőség, ezért az Entity Framework Core InMemory adatbázisát használtam fel.

```
 [Fact]
public async Task GetTopFive()
{
```

```
    // Arrange
    using (var context = DbContextHelper.GetInMemoryDbContext())
    {
        var service = new EventService(context);
        int expectedCount = 5;

        // Act
        var events = await service.GetTopAsync(5);

        // Assert
        Assert.True(events.Count == expectedCount);
    }
}
```

A tesztjeim a következő módon épültek fel: mindegyik három részből áll, melyek az Arrange (értékek, adatok beállítása), Act (tesztelt metódus meghívása), Assert (viselkedés ellenőrzése). Két módon lehet unit tesztet létrehozni. Az egyik, ha metódust a *Fact* attribútummal látjuk el, a másik, amikor a *Theory* attribútumot kapja, de ilyenkor kötelező adatokat megadni.



30. ábra: Lefutott unit tesztek

5. Összefoglalás

5.1 Elvégzett munka

A diplomamunkám első részeként az Angular keretrendszerrel kellett megismerkedjek, illetve kipróbálni hogyan működik a SignalR technológiával. A hivatalos dokumentációkból egész egyszerűen sikerült összekapcsolnom őket.

Miután megismertem a keretrendszert, elkezdtem az alkalmazás megtervezését. Elsőként az adatmodellt terveztem meg, amire a program épít. Ezt követően elkészült a klienshez egy rövid felhasználói felület vázlat, ami alapján a nézeteim készültek. Az Angular alkalmazást párhozamosan fejlesztett a szerver oldali ASP.NET Core alkalmazással.

Elsőként a DAL réteget valósítottam meg, majd a webalkalmazáson kezdtem el dolgozni. A BLL réteg párhuzamosan fejlődött az új funkciók létrehozásával. A legnagyobb hangsúlyt az élő-kommunikáció kapta, ezért a foglalási oldalt valósítottam meg először. A SignalR bekötése nem okozott nagyobb kihívást, viszont a fejlesztés során a kliens oldali könyvtár mást csomagba kerül, így át kellett alakítanom.

Miután a publikus portál elkészült, az ügyintéző webalkalmazás megvalósítása volt a cél. Ez egy ASP.NET Core MVC portál, ami ugyan azt az adatbázist használja, mint a publikus portál. A tesztek elkészítése a két alkalmazás megvalósítása után következett.

5.2 Munkám értékelése, tapasztalatok

A programommal szembeni elvárásokat teljesítettem, törekedtem a tisztakód elveit betartani és szép architektúrát megvalósítani. Törekedtem az Angular alkalmazáshoz tartozó előírásokat betartani, a komponenseket és a szolgáltatásokat szétválasztani. Számos részt már máshogy valósítottam volna meg a megszerzett tapasztalatok alapján.

Az Angularos alkalmazásom az ötös verzióval indult el, mert, amikor a projektet kezdtem, akkor a Visual Studio sablonja még csak ezt tartalmazta. Hogy naprakész legyen a webalkalmazás, ezért felfejlesztettem a nyolcas verzióra, ami számomra nem volt

teljesen egyszerű. Számos helyen kellett javítanom a kódomat, főleg a Bootstrap miatt, mert az épített az újabb verziókra.

5.3 Továbbfejlesztési lehetőségek

Az elkészült rendszerem számos továbbfejlesztési lehetőséget rejt még magában. Az egyik legfontosabb egy sokkal jobb UX kialakítása lenne, akár egy designer segítésével. A felületeket én alakítottam ki a saját terveim alapján, de törekedtem a kényelmes és barátságos felhasználói élmény megvalósítására.

Egy másik fontos továbbfejlesztés az ügyintézői oldal bővítése. A megvalósított alkalmazásban fel lehet venni az entitásokat, amelyek a publikus oldalon megjelennek, de az implementáció és a design elég kezdetleges. Ezt bizonyítja, hogy az architektúrája csak kétrétegű lett, a felület egyből az adatelérési réteget hívja.

Ahogy kifejtettem egy fajta foglalási modellt valósítottam meg. Ezt lehetne bővíteni egy másikkal, ami lehetne egy „folyamatos eseményre” foglalás. Ez modellezné az éttermek működését, ahol az asztalfoglalásos rendszer működik.

6. **Hivatkozások**

[1] Angular, „Angular,” [Online]. Available: https://angular.io/. [Hozzáférés dátuma: 28 09 2019].

[2] Microsoft, „.NET Standard,” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/net-standard. [Hozzáférés dátuma: 28 09 2019].

[3] Microsoft, „EF Core,” [Online]. Available: https://docs.microsoft.com/en-us/ef/core/. [Hozzáférés dátuma: 28 09 2019].

[4] Microsoft, „Real-time ASP.NET with SignalR,” Microsoft, [Online]. Available: https://dotnet.microsoft.com/apps/aspnet/signalr. [Hozzáférés dátuma: 28 09 2019].

[5] Microsoft, „TypeScript,” Microsoft, [Online]. Available: http://www.typescriptlang.org/. [Hozzáférés dátuma: 28 09 2019].

[6] Angular, „Angular - Introduction to modules,” 28 09 2019. [Online]. Available: https://angular.io/guide/architecture-modules#introduction-to-modules.

[7] Angular, „Architecture overview,” Angular, [Online]. Available: https://angular.io/guide/architecture. [Hozzáférés dátuma: 28 09 2019].

[8] Microsoft, „Database Providers,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/ef/core/providers/. [Hozzáférés dátuma: 28 09 2019].

[9] Microsoft, „Overview of ASP.NET Core MVC,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.0. [Hozzáférés dátuma: 28 09 2019].

[10] Microsoft, „ASP.NET Core Middleware,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.0. [Hozzáférés dátuma: 28 09 2019].

[11] IETF, „The WebSocket Protocol,” [Online]. Available: https://tools.ietf.org/html/rfc6455. [Hozzáférés dátuma: 28 09 2019].

[12] Microsoft, „Solutions and projects in Visual Studio,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/ide/solutions-and-projects-in-visual-studio?view=vs-2019. [Hozzáférés dátuma: 12 10 2019].

[13] Microsoft, „Migrations,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/. [Hozzáférés dátuma: 13 10 2019].

[14] Microsoft, „Creating and configuring a model,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/ef/core/modeling/. [Hozzáférés dátuma: 23 10 2019].

[15] Angular, „Angular - Components,” Angular, 28 09 2019. [Online]. Available: https://angular.io/guide/architecture-components.

[16] Microsoft, „.NET Standard,” 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/net-standard.

[17] Angular, „Angular - Pipes,” 20 11 2019. [Online]. Available: https://angular.io/guide/pipes.

7. Utolsó simítások

Miután elkészültünk a dokumentációval, ne felejtsük el a következő lépéseket:

- *Kereszthivatkozások frissítése:* miután kijelöltük a teljes szöveget (Ctrl+A), nyomjuk meg az F9 billentyűt, és a Word frissíti az összes kereszthivatkozást. Ilyenkor ellenőrizzük, hogy nem jelent-e meg valahol a "Hiba! A könyvjelző nem létezik." szöveg.
- *Dokumentum tulajdonságok megadása:* a dokumentumhoz tartozó meta adatok kitöltése (szerző, cím, kulcsszavak stb.). Erre való a Dokumentum tulajdonságai panel, mely a Fájl / Információ / Tulajdonságok / Dokumentumpanel megjelenítése úton érhető el.
- *Kinézet ellenőrzése PDF-ben:* a legjobb teszt a végén, ha PDF-et készítünk a dokumentumból, és azt leellenőrizzük.