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

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. 09. 30.

...…….
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 rendszer kialakítását, amit szabadon böngészhetnek és használhatnak.

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 felhasználóval, ezen felül a foglalásokat a valós időben megtekinthetik.

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 részletesen bemutatom az alkalmazásom működését és a legfontosabb használati eseteket, hogyan haladnak végig a különböző komponenseken. 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, úgy, mint 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 határozzák meg a nézeteket, melyeket az alkalmazás logika és adatok alapján módosíthat. Szolgáltatásokat használ fel, melyek valamilyen speciális funkcióval szolgálnak, ami nem kapcsolódik a felülethez (például adatok hálózati letöltése). Kapcsolatot az úgy nevezett biztosítók (*provider*) nyújtanak, betöltve őket a megfelelő helyre, ezzel téve a kódot modulárissá, újra használhatóvá és hatékonyabbá.

2.2.2 Modulok

Az *NgModule*-ok az összetartozó kódrészletek tárolói. Tartalmazhat 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.

Az modulokat a *@NgModule()* annotációval ellátott osztály határozza meg. Ez a dekorátor egy funkció, ami felvesz egy egyszerű meta adat objektumot, ami leírja a modult. A legfontosabb tulajdonságai:

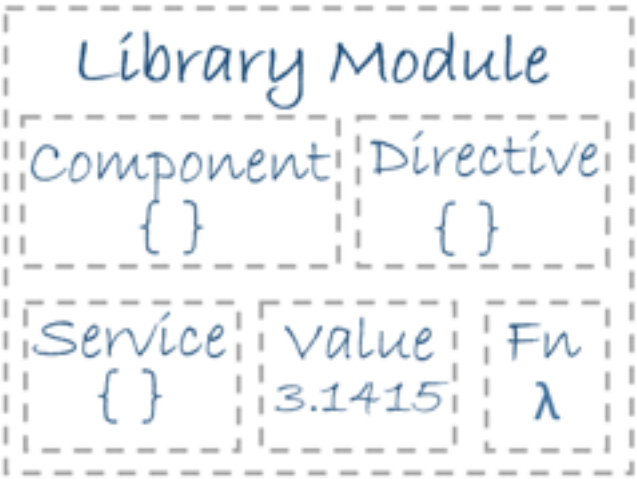
- declarations: komponensek, direktívás, *pipe*-ok[8], amik a modul részei,
- exports: deklarációk részhalmaza, amik szükségesek, hogy a komponensek láthatók és használhatók legyenek a többi modulban,
- imports: más modulok, melyek osztályai szükségesek a saját komponenseink sablonjaihoz,
- providers: 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 { }
```

[8] Adat transzformációt végző eszköz a HTML sablonban

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 és importálni kell őket.



1. ábra: Angular osztálykönyvtár felépítése [6]

2.2.3 Komponensek

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

2.2.4 Sablon szintaxis és adatkötés

A sablonok, más néven *template*-k, egyszerű HTML részletek, kivéve, hogy kiegészítik az Angular szintaxisok, melyek kibővülnek a komponensek adataival és az alkalmazásunk logikájával. Erre szolgál az adatkötés, ami összeköti az alkalmazás adatait és a DOM-ot.

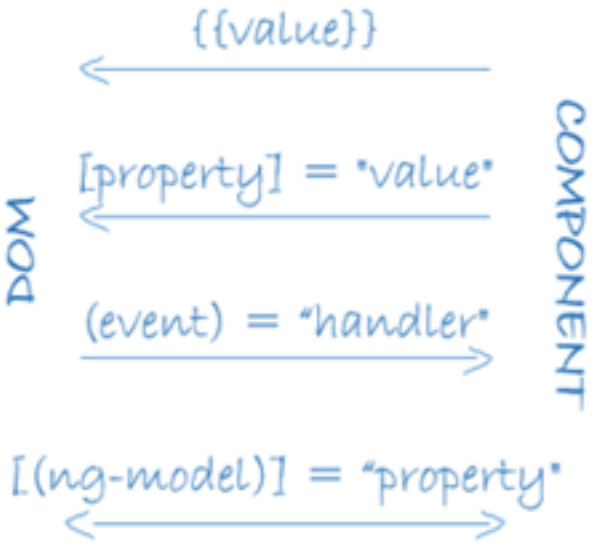
Egy egyszerű példa egy sablonra adatkötéssel:

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

[9] Csomagkezelő rendszer JavaScript programozáshoz.

Ebben a sablon részletbe több Angular szintaktika is megjelenik. A **ngFor* egy direktíva, ami egy 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. Azzal, hogy a komponensek nézet specifikus funkcionalitását szétválasztja az egyéb fajta számításoktól, soványabbá és hatékonyabbá teszi a komponenseket.

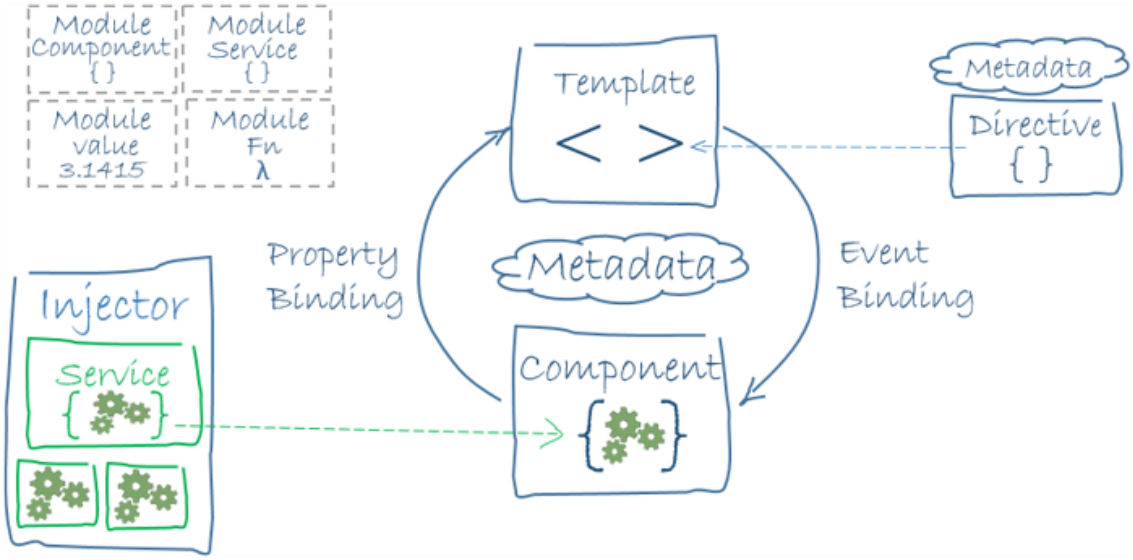
Ideális esetben a komponensek feladata csupán a felhasználók interakcióinak kezelése és semmi több. Feladata megbízni a szolgáltatásokat bizonyos

tevékenységekkel, például adatok biztosítása 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 könnyebb, 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 háromfajta fejlesztési szemléletet támogat alkalmazások készítéséhez úgy, mint a Code First, Model First és Database First megközelítés. 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, internetes alkalmazások készítésére. Ez a nagy ASP.NET újra tervezése szerkezeti változásokkal, amiknek köszönhetően soványabb és modulárisabb lett a rendszer. Számos előnnyel rendelkezik: beépített függőség injektálás, nagyteljesítményű, moduláris HTTP kérés csővezeték, IIS és ön-hosztolásra is képes saját folyamatban (process), fejleszthető több operációs rendszeren is (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.

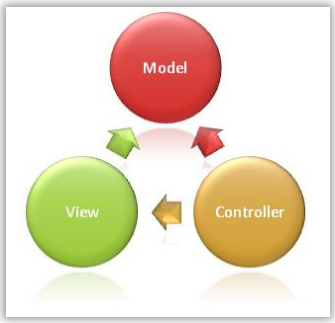
Az ASP.NET Core MVC sok szolgáltatást nyújt a fejlesztőknek, ezek közé tartozik az út kezelés (routing), modellkötés (model binding), modell validáció, függőség

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

injektálás, szűrők stb. Ezek közül számos komponenst felhasználtam az alkalmazásom elkészítése során.

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 felhasználó kérései a vezérlőhöz vannak irányítva, ami felelős a modellel való munkáért. A vezérlő választja ki azt a nézetet, ami megjelenik az ügyfélnek és biztosítja az adatot, ami szükséges a felületen való megjelenítéshez.



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

2.4.2 Web API

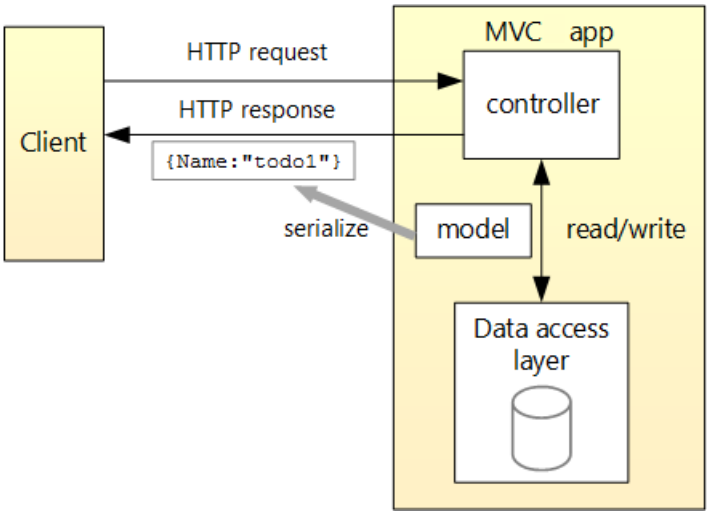
A HTTP ma már nem csak HTML oldalak kiszolgálására létezik. Ez egy erőteljes platform Web API készítésére, felhasználva az igéit (GET, POST, PUT stb.) és néhány egyszerű elgondolást, mint például az URI és a fejlécek.

Az ASP.NET Core Web API egy komponens halmaz, ami leegyszerűsíti a HTTP programozást. 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, kisorosítja, majd a kapott objektumot adatbázisba írja illetve lekérdez az adatbázisból. A kérés alapján a választ sorosítva visszaküldi.



5. ábra WebAPI kommunikáció

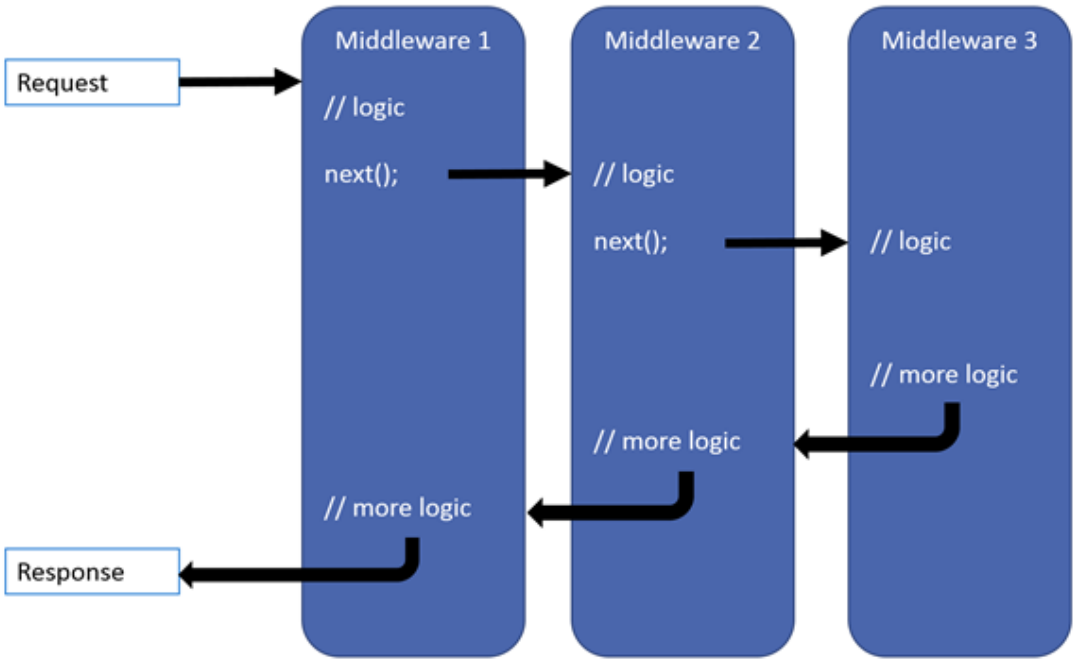
2.4.3 Single-page application Middleware

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

A Middleware egy olyan szoftverkomponens, ami beépül az alkalmazáshoz befutó kérések kiszolgálásához. 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 praktikus megközelítést nyújt, a SPA egy alkönyvtárban helyezkedik el a projektben és itt működik 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 kliens alkalmazás 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.

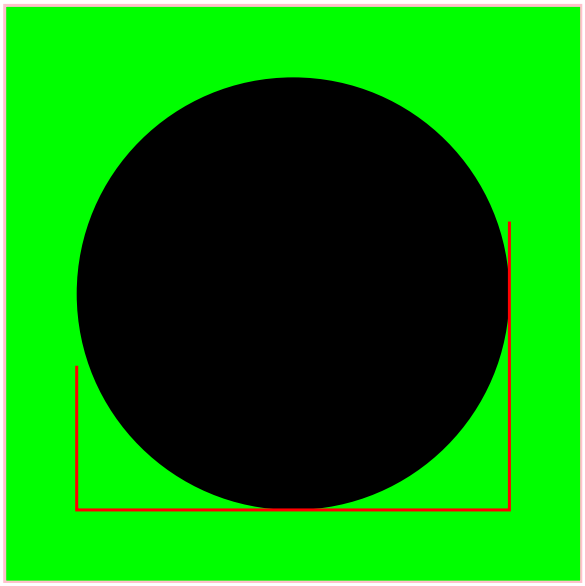
Mivel az SVG képek XML fájlokban vannak definiálva, ezért kereshetők, indexelhetők, szkriptelhetők és tömöríthetők. Létrehozhatók egyszerű szövegszerkesztővel, de léteznek hozzá rajzoló szoftverek is.

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

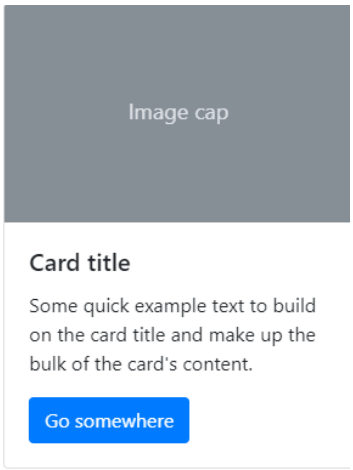
A vektorgrafikus képek geometria primitívekből épülnek fel, ilyenek például a pontok, egyenesek, görbék. Előnyei a raszter grafikával szemben, hogy tetszőlegesen nagy nagyítás sem torzítja a képet, a vonalvastagságnak nem kell a nagyítással arányosan nőni, az alakzatok méretei tárolhatók és így később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

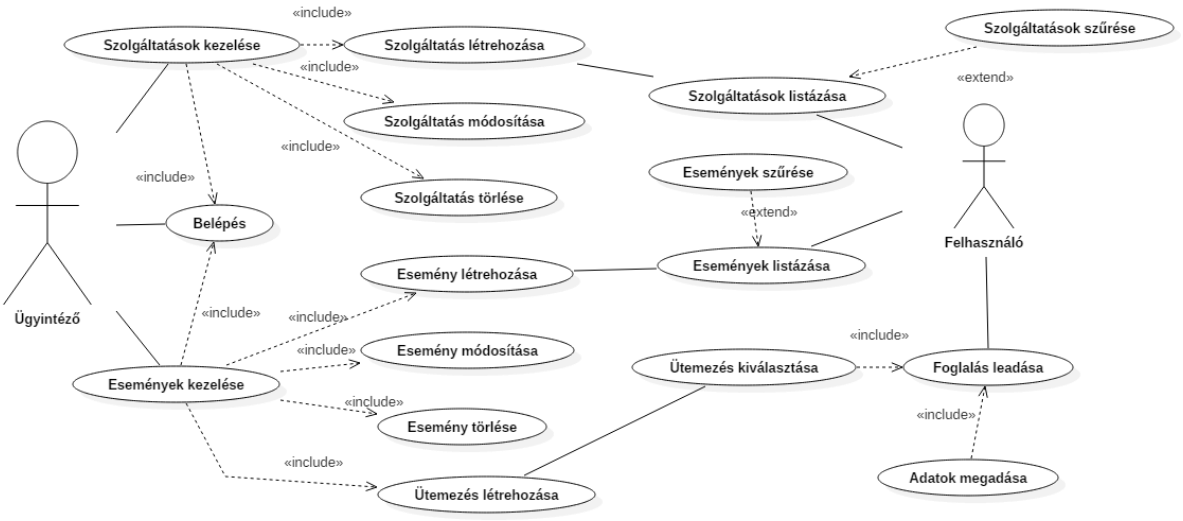
Az xUnit.net egy ingyenes, nyílt forráskódú unittesztelő eszköz .NET keretrendszerhez. A xUnit tesztek létrehozásához egy külön xUnit teszt projektet kell létrehozni. Az egység tesztek olyan függvények, melyek *[Fact]* vagy *[Theory]* attribútummal vannak ellátva.

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. Követelmények



4. Architektúra

5. Megvalósítás

6. Értékelés

7. Irodalomjegyzék

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

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

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

[4] 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].

[5] 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].

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

8. 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.