01. LABOR - ASP.NET CORE

Segédlet a Szoftverfejlesztés laboratórium 2 c. tárgyhoz

Készítette: Gincsai Gábor, 2017
Frissítette: Szabó Gábor, 2018
Erdős Szilvia, 2020

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

1117 Budapest, Magyar tudósok körútja 2.
Q. ép. B. szárny 207. • www.aut.bme.hu
Telefon: 463-2870 • Fax: 463-2871

Bevezetés

Célkitűzés

A labor célja, hogy a hallgatók a webes fejlesztés szerveroldali részbe is betekintést nyerjenek, megismerjék az ASP.NET Core alapjait.

A laboron egy online hirdetési felületet fogunk létrehozni ASP.NET Core-ban, teljesen az alapoktól. A cél, hogy áttekintsük az ASP.NET Core működését, találkozzunk a leggyakrabban felmerült problémákkal, és elkészítsünk egy egyszerűbb webes alkalmazást.

Előfeltételek

A labor elvégzéséhez szükséges eszközök:

- Microsoft Visual Studio 2017/19
  - .NET Core 2.1 SDK
- ASP.NET-Core-kiindulo.zip a tanszéki honlapról

Amit érdemes átnézned

- HTML, CSS, JS alapok

A labor értékelése

A labor első része laborvezető által vezetett, a második rész önálló. A teljes munkát (ideértve a vezetett részt is) a laborvezető a labor végén értékeli. Az értékeléskor a laborvezető kérdéseket tehet fel az elkészült alkalmazással kapcsolatban, amik befolyásolhatják az érdemjegyet. Az értékelést a hallgató a labor végeztével, de legkésőbb a laborvezető által a labor megkezdésekor jelzett időpontban jelzi a laborvezetőnek. Az ellenőrzéshez segítséget nyújt a labor végén összeállított ellenőrző lista.

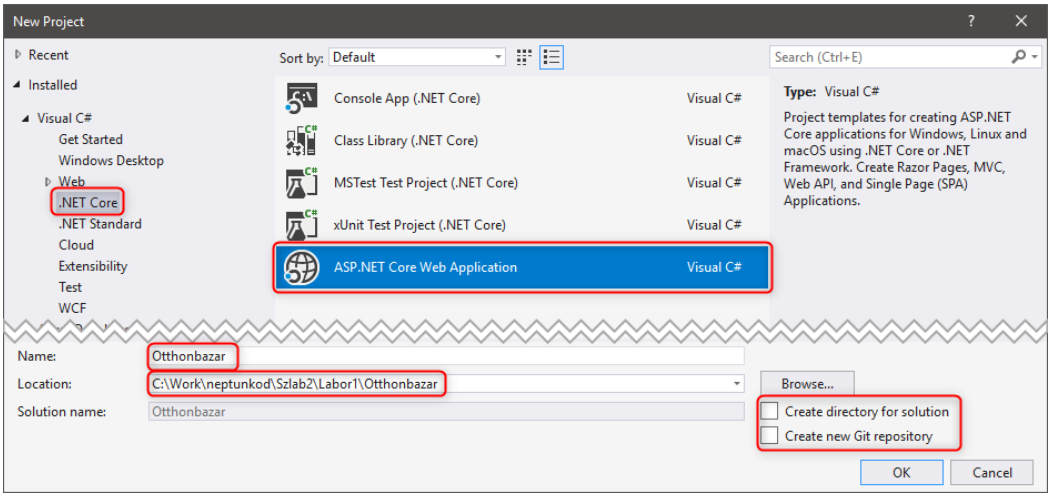
Távolléti formában: a megadott határidőig kell mindenkinek feltölteni a labort, a határidő lejárta után értékelik a laborvezetők a leadott munkákat.

A labor végén a tárgy moodle oldalára (https://edu.vik.bme.hu/) feltöltendő az elkészült forráskód ZIP formátumban. A ZIP-ből törlendők a fordítási artifaktok (bin, obj mappák)!

ASP.NET Core Web Application létrehozása

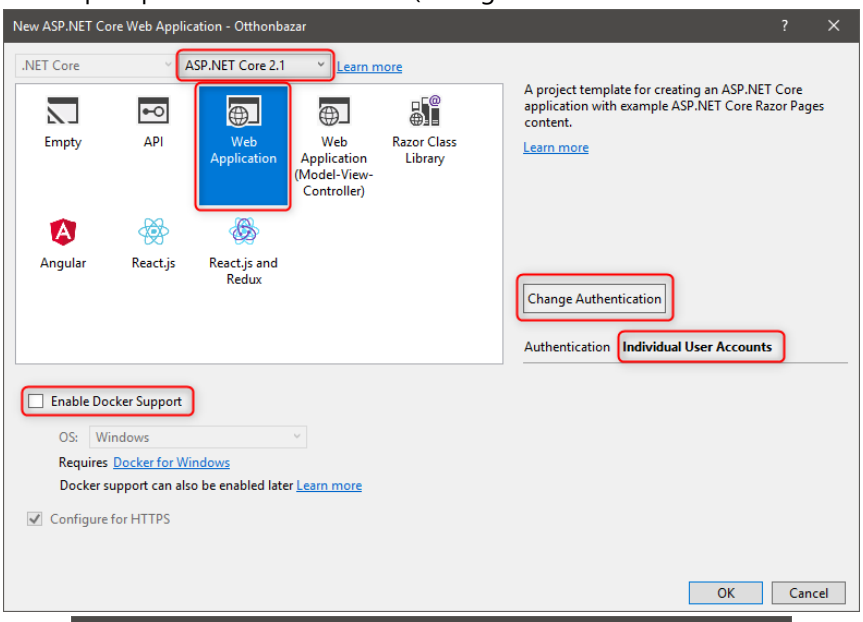
Első lépésként hozzunk létre egy új ASP.NET Core Web Application-t. Ezt a *File → New → Project…* menüpontra kattintva tudjuk megtenni.

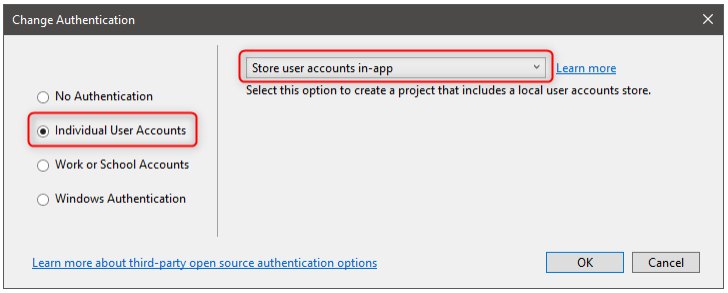
- Válasszuk ki, hogy *.NET Core* alkalmazást, azon belül is *ASP.NET Core Web Application*-t szeretnénk létrehozni.
- A projekt neve legyen *Otthonbazar*, a projekt helye a laborvezető által megadott munkamappa.
- A projekt mellé ne hozzunk létre solution-t, mert jelenleg egy projektből fog állni az alkalmazás.
- Ne inicializáljunk Git repository-t, mert nem fogunk használni forráskódkezelőt.



Ezt követően a következő képernyőn tudjuk megadni, hogy pontosan milyen projektet szeretnénk létrehozni.

- Válasszuk ki a .NET Core → ASP.NET Core 2.1-es projekttípust.
- Válasszuk ki, hogy Web Application típusú legyen a projekt. Ez egy Razor Pages alapú, MVVM-re építő alkalmazásvázat készít nekünk.
- Állítsuk be az űrlap alapú felhasználókezelést (Change Authentication → Individual User Accounts).

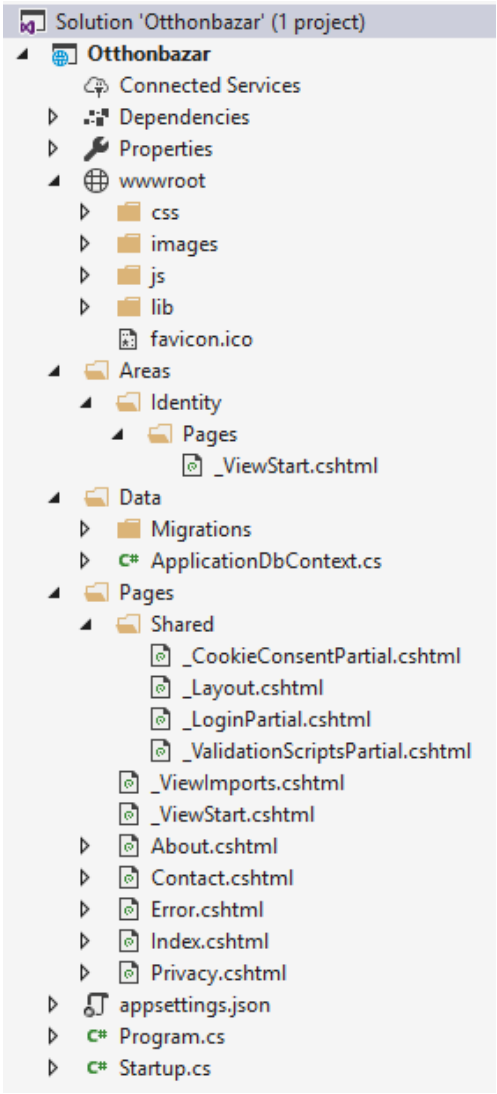




ASP.NET Core projekt felépítése

Röviden tekintsük át a létrehozott projekt felépítését

- *Dependencies*: Itt találhatók a külső függőségei a projektnek (NuGet csomagok, SDK, projektszintű referenciák).
- *wwwroot*: A webszerverre kerülő statikus erőforrások (pl: CSS, képek, JS, és külső JS könyvtárak)
- *Areas/Identity/Pages/_ViewStart.cshtml*: A felhasználókezeléshez használt ASP.NET Core Identity sok hasznos funkciót ad nekünk a felhasználói fiókok kezeléséhez. Ez a fájl a belépési pontunk, ha módosítani szeretnénk a beépített funkcionalitáson.
    - Az Areák az alkalmazás nagy logikai egységekre bontását segítik.
- *Data*: Adatbázis Entity Framework Core (EF Core) Code Firsttel.
- *Pages*: Az alkalmazásunk Razor Pages alapú oldalai. Ezek egy nézetből (Razor CSHTML) és egy PageModelből (az oldalhoz kapcsolódó szerveroldali logikát leíró MVVM osztály) állnak.
- appsettings.json: Az alkalmazás konfigurációs fájlja, itt egyedi és keretrendszer-szintű tulajdonságokat állíthatunk be, amit az alkalmazáson belül kezelhetünk.
- *Program.cs*: Ez a program belépési pontja. Itt állítjuk be a hosztolási környezetet, amit alapértelmezés szerint a *CreateDefaultBuilder* végez el (ez állítja be a Kestrelt mint webszervert, beállítja a content root mappát, betölti a konfigurációkat (pl: appsettings.json), és meghívja a Startup.cs-t).[1]
- *Startup.cs*: A *ConfigureServices* metódusban az alkalmazásban használt szolgáltatásokat konfiguráljuk fel függőséginjektálás (DI) használatára. A *Configure* metódusban az alkalmazás által használt middleware-eket konfiguráljuk fel, ezek egymás után, a beregisztrálás sorrendjében hívódnak meg a HTTP kontextuson.



Az alkalmazásunk az MVC keretrendszerre épít, egy magasabb absztrakciós szinten mi viszont modellek, nézetek és vezérlők helyett Razor Pages-t fogunk használni, ami a funkciók egy halmazára a nézet, a logika és a modell együttesét határozza meg, ezzel modulárisabb, MVVM jellegű felépítést biztosítva.

Amennyiben nincs adminisztrátori jogosultságunk, az alkalmazás nem fog tudni HTTPS protokollon működni lokálisan, mert nem tudjuk telepíteni a fejlesztéshez szükséges tanúsítványt. Ebben az esetben az alábbiakat kell módosítanunk, hogy az alkalmazás HTTP-n kommunikáljon:

- a Properties/launchSettings.json fájlban töröljük ki az alábbi sort: **"sslPort": 443xy**,
- a Startup.cs Configure() metódusában töröljük vagy kommentezzük ki az alábbi sort: **app.UseHttpsRedirection();**

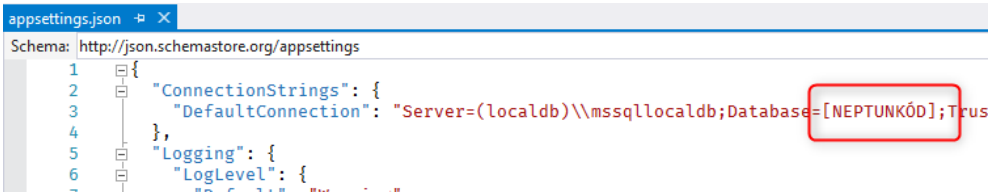
[1] Részletek: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/hosting?tabs=aspnetcore2x

Tegyünk breakpointot (F9) az alábbi helyekre, hogy meg tudjuk vizsgálni, milyen sorrendben ütjük meg őket:

- a **Pages/Index.cshtml.cs** `OnGet()` metódusába,
- a **Pages/Index.cshtml**-be (4. sor),
- a Program.cs `Main()` és `CreateWebHostBuilder()` metódusába,
- a Startup.cs `ConfigureServices()` és `Configure()` metódusába.

Indítsuk el az alkalmazást debug módban (F5), vizsgáljuk meg a breakpointokat, majd kattintgassuk végig, mit is kapunk készen. Vizsgáljuk meg, hogy a böngészőt frissítve milyen breakpointokat ütünk meg ismét!

A generált projekt appsettings.json fájljában írjuk át a connection stringet, hogy a saját neptun kódunk legyen a generált adatbázis neve (szögletes zárójelek nélkül):



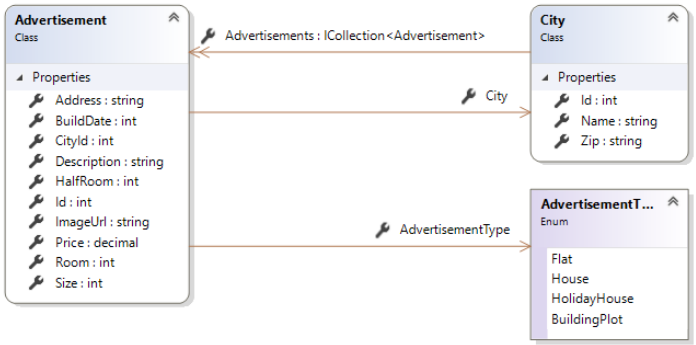
Adatbázis létrehozása EF Code Firsttel

Ne feledjük, hogy a projekt létrehozásakor már megadtuk, hogy a felhasználókat adatbázisban szeretnénk tárolni. Ez azzal járt, hogy az adatbáziskontextusunk, amely az adatbázisunk elérésének központi eleme a DbContexten túl az IdentityDbContext-ből származik, amely a felhasználókezeléssel kapcsolatos fogalmakhoz ad hozzáférést (pl. Users, Roles stb.).

A Data könyvtárba hozzuk létre a City osztályt (usingoljuk a `System.ComponentModel.DataAnnotations` névteret):

```
public class City
{
    public int Id { get; set; }
    [Required, StringLength(4, MinimumLength = 4)]
    public string Zip { get; set; }
    [Required]
    public string Name { get; set; }
    public ICollection<Advertisement> Advertisements { get; set; }
}
```

A fentihez hasonlóan, az alábbi diagram alapján készítsük el az Advertisement osztályt és az AdvertisementType enumot is. Az adatbázist az entitások megfogalmazása utána fogjuk tudni generálni.



A *code first* megközelítés lényege, hogy az adatmodellünket kódban készítjük el (szemben az adatbázis sémájával), és ebből a kódból generálódik majd az adatbázis sémája.

Az EF Core konvenciók alapján azonosítja a pontos sémát, amit generálni fogunk. Konvenció szerint például:

- Az int típusú, Id nevű tulajdonságok adatbázis által generált, mesterséges, elsődleges kulcsok lesznek.
- Az ún. navigation property-k (más saját típusra történő hivatkozás) vagy EntitásId property-k alapján készülnek el a külső kulcsok a sémában. Célszerű mindkettőt megadni, ezért van a hirdetés entitásunkban egy külső kulcs (CityId) és egy navigációs tulajdonság (City) is, ezzel megkönnyítjük az életünket a bonyolultabb DB lekérdezésekkor.
- Minden mező, ami null értéket vehet fel, automatikusan opcionális, a többi automatikusan kötelező.

Ezektől a konvencióktól esetenként el kell térnünk, a string értékek pl. gyakran kötelező mezők kell, hogy legyenek az adatsémánkban is, de mivel null értéket vehetnek fel, ezért automatikusan opcionálisak lesznek. A konvenciókon túl konfigurációt attribútumok formájában (deklaratívan) vagy fluent API-n keresztül (imperatívan) tudunk elvégezni. Jelenleg csak a kötelezőség fontos, ezért az `Advertisement`ben vegyük fel a `[Required]` attribútumot minden tulajdonsághoz az `Id`, `Description`, `City` és `ImageUrl` **kivételével**. Az attribútumot az alkalmazás nézeteinek generálásakor is hasznosítani fogjuk, ekkor nem csak a null, hanem az alapértelmezett értékre is vizsgálni fogunk (pl. int esetén ez a 0).

Ne feledjük using-olni a `System.ComponentModel.DataAnnotations` névteret (Ctrl+. adott kódrészleten).

Egészítsük ki az *ApplicationDbContext.cs* fájlt a két új táblával. Ehhez a *Cities* és *Advertisements* tulajdonságokat kell felvennünk, mind a kettő egy DbSet<T> lesz (megfelelő típusparaméterrel), ez jelzi, hogy ezek az adatbázisban táblaként kezelendők.

Ezen felül, ha kézzel szeretnénk megadni, hogy egy entitás milyen adatbázis táblára mappelődjön, azt az `OnModelCreating`-ben tehetjük meg (vagy használhatunk attribútumos megoldást is). Alapértelmezetten a táblák nevei többesszámosítottak, ezt meg tudjuk változtatni az alábbi módon:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
   // ...

   protected override void OnModelCreating(ModelBuilder builder) {
      base.OnModelCreating(builder);
      // Customize the ASP.NET Identity model and override the defaults if needed.
      // For example, you can rename the ASP.NET Identity table names and more.
      // Add your customizations after calling base.OnModelCreating(builder);

      builder.Entity<City>().ToTable("City");
      builder.Entity<Advertisement>().ToTable("Advertisement");
   }
}
```

Opcionális, mert az entitások navigation property-jeit konvenció szerint a rendszer kezeli, de explicit így tudjuk megadni ugyanitt a kapcsolat létesítését (ez a fluent API alapú modellkonfiguráció, ugyanígy tudnánk attribútumok helyett pl. a kötelezőségi megkötést alkalmazni):

```
builder.Entity<Advertisement>()
    .HasOne(a => a.City)
    .WithMany(c => c.Advertisements)
    .HasForeignKey(a => a.CityId);
```

Ahhoz, hogy az újonnan módosított adatbázist generálja futás közben az alkalmazás, létre kell hozni egy új migrációt. Ezt a *Package Manager Console*-ból tehetjük meg az alábbi utasítással.

```
PM> Add-Migration AdvertisementsAndCities
```

Nézzük meg, hogy milyen kódot generált a Data\Migrations könyvtárba!

Ezt követően generáljuk le magát az adatbázist is, amit az alábbi paranccsal tehetünk meg a Package Manager Console-ból. Ha a gépen nincsen megfelelő jogosultságunk, az adatbázis létrehozása Permission Denied hibával elszáll. Ebben az esetben az adatbázist kézzel hozzuk létre a parancs futtatása előtt az alábbi módon: az SQL Server Object Explorerben (ha nem látható az IDE bal oldalán, a View → SQL Server Object Explorer opcióval tudjuk előhozni) a (localdb)\MSSQLLocalDB adatbázison a Databases elemen jobb klikk, majd New Database…, a felugró ablakban adjuk meg a neptunkódunkat, ezáltal létrejön egy üres adatbázis, és az Update-Database parancs is lefut ezután, ami létrehozza az adatbázis sémáját.

```
PM> Update-Database
```

Vizsgáljuk meg, hogy létrejött-e az adatbázisunk a megfelelő sémával az SQL Server Object Explorer segítségével!

Ha esetleg feltűnik az adatbázis sémában valami hiba vagy hiányosság, akkor módosítani kell a megfelelő táblához tartozó C# osztályokat, majd újabb migrációt generálni (Add-Migration … csak más névvel) és frissíteni az adatbázist (Update-Database).

Ezt követően indítsuk el az alkalmazásunkat és regisztráljunk be egy felhasználót a *neptunkódunk*@aut.bme.hu címmel, ahol a *neptunkódunk* a saját neptunkódunk! A létrejött felhasználót szintén az SQL Server Object Explorer segítségével tudjuk megvizsgálni, navigáljunk az adatbázis táblái között az *AspNetUsers* táblához, és ezen *jobb klikk → View Data* segítségével tudjuk megnézni a létrejött felhasználónkat.

TESZT ADATOK FELVÉTELE

EF Core segítségével van lehetőségünk „ősadatokkal" feltölteni az adatbázist migrációkor. Ehhez a fluent API-n elérhető `HasData<T>(params T[] entities)` metódust tudjuk használni. A nyersadatokat be tudnánk tölteni dinamikus kódból is, viszont ez nem szerencsés, mert ha bármilyen okból kifolyólag nem determinisztikus ennek működése, akkor inkompatibilis migrációkat tudunk véletlenül létrehozni. Ezért érdemes mindig explicit megtennünk az adatok feltöltésének leírását. Erre használjuk az *ASP.NET-Core-kiindulo.zip* fájlban található két SeedConfig fájlt, ezeket a projekt Data/EntityTypeConfigurations mappájába vegyük fel. Ezután az DbContext OnModelCreating metódusában vegyük fel ezeket az ősadatokat:

```
builder.ApplyConfiguration(new CitySeedConfig());
builder.ApplyConfiguration(new AdvertisementSeedConfig());
```

Ezután új migrációt kell létrehoznunk ismét, feltölteni az adatbázist a migráció lefuttatásával, és elkészült az adatmodellünk:

```
PM> Add-Migration SeedAdvertisementsAndCities
PM> Update-Database
```

Ha a migráció létrehozása után nem szeretnénk minden alkalommal lefuttatni az adatbázis frissítését, akkor a webalkalmazás indulásakor kódból is megtehetjük a séma frissítését, a Main metódus módosításával (a hiányzó szimbólumokon Ctrl+. segítségével usingoljuk be a megfelelő névtereket):

```
var webHost = CreateWebHostBuilder(args).Build();
using (var scope = webHost.Services.CreateScope())
{
    scope.ServiceProvider.GetRequiredService<ApplicationDbContext>().Database.Migrate();
}
webHost.Run();
```

A fenti kód a kiszolgáló objektum elkészítése után, de még a szerver elindítása előtt (a .Run() hívás végtelen ciklusban várakozik a beérkező HTTP kérésekre) a DI konténerbe beregisztrált szolgáltatások közül elkéri a DbContext példányunkat, és annak meghívja a Database.Migrate() metódusát. Ezt csak fejlesztési környezetben ajánlott használni.

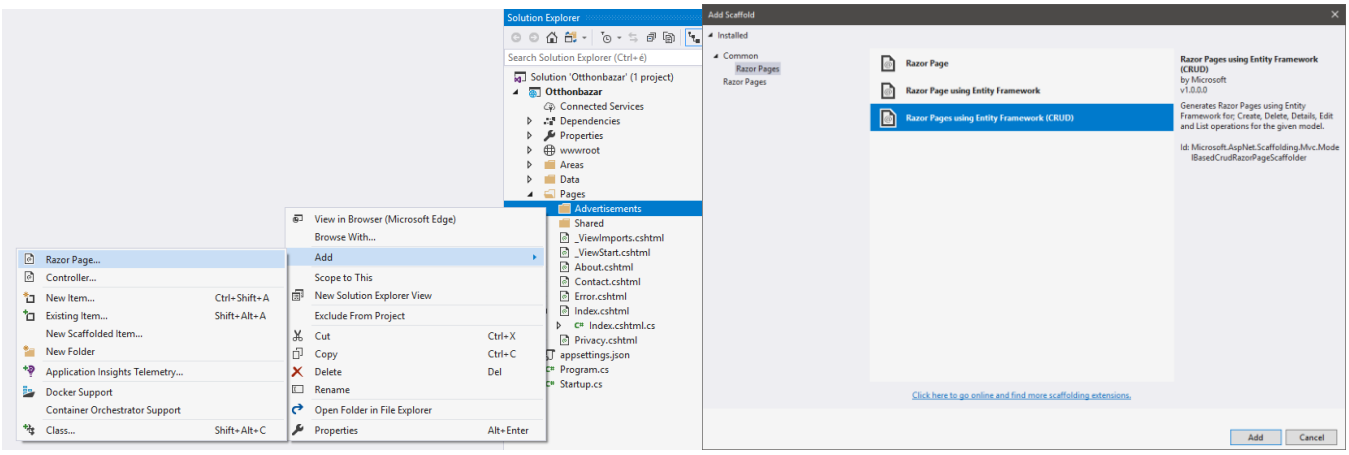
Hirdetések megjelenítése

Az oldalak egységes kinézetét az ASP.NET layout (más nomenklatúrákban *master*) oldalak segítségével biztosítja. Nézzük meg, hogy a fenti Index.cshtml generálásakor milyen layout oldalt használtunk és hogyan került ez beállításra.

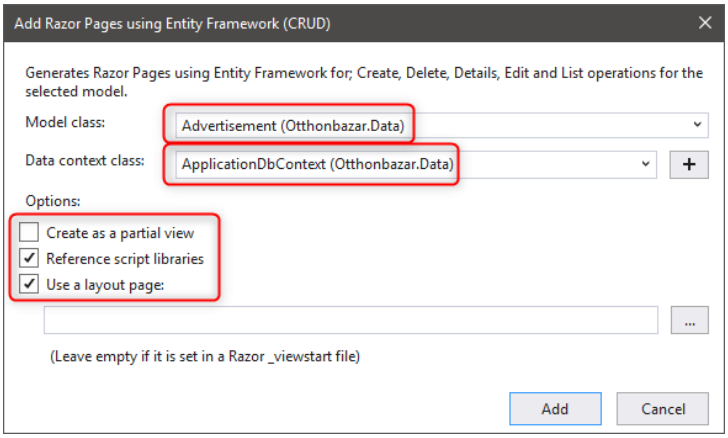
A Pages mappában található ***_ViewStart.cshtml*** adja meg, hogy alapértelmezés szerint az ebben a mappában található view-k milyen elrendezést (*layout*ot, keretet) használjanak (ezt oldalanként felüldefiniálhatjuk a Layout értékének beállításával), jelen esetben a *_Layout.cshtml*-t használják (ezt lehet felüldefiniálni az egyes view-kban). Több _ViewStart.cshtml fájlunk is lehet mappahierarchiában, ekkor ezek egymás után sorra lefuthatnak, így nagyon egyszerű egymásba ágyazott elrendezéseket készíteni. A layout oldal tartalmazza a <*html*> taget, az egyes view-k ebbe az oldalba generálnak bele részeket (az oldalak tartalmi részét), a view-k kódja a `@RenderBody()` részbe fog bekerülni. Ha ezen felül az oldal más részére is szeretnénk kódot illeszteni a view-ból, akkor azt a `@RenderSection()` résszel tudjuk meghatározni egy placeholdert a layoutba téve, amibe majd a view a @section segítségével tudja a tartalmat beleilleszteni. Ezt a részt nagyon elvétve használjuk csak éles környezetben, nem szabad, hogy itt elburjánzzanak a különböző JavaScript kódrészletek!

A feladatunk egy olyan oldal létrehozása, melyen megjelennek az adatbázisban korábban létrehozott hirdetések.

Vegyünk fel egy új mappát a Pages-ben **Advertisements** néven. Ezen belül hozzunk létre egy új Razor Page-et, ami scaffolding használatával gyárt nekünk különböző oldalakat a hirdetések megtekintésére, listázására, szerkesztésére, létrehozására és törlésére:



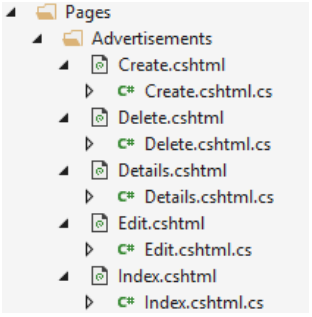
Többféle módon is hozhatunk létre ilyen oldalt: üresen vagy Entity Framework használatával az egyes műveletekre, vagy mindegyikre egyszerre.



Ha az összes műveletre szeretnénk generálni, akkor meg kell adnunk, melyik entitást szeretnénk használni, melyik adatbáziskontextuson szeretnénk dolgozni, partial oldal-e, referálja-e a JS osztálykönyvtárakat és használjon-e valamilyen layoutot.

Tipp: Érdemes elgondolkozni azon, hogy a Viewk milyen modellt kapjanak meg. Egyszerű esetekben könnyedség lehet, ha egy az egyben megkapják az adatbázis táblákhoz tartozó osztályokat modellként, azonban, ha már egy kicsit bonyolultabb felületről beszélünk, akkor érdemes DTO-kat bevezetni, ami összefogják az egy oldal / oldalrészhez tartozó tulajdonságokat egy osztályba. Az egyszerűség kedvéért nem vezetünk be egy ilyen köztes réteget, de ezt éles környezetekben mindig illik megtennünk.

Nézzük meg, milyen oldalak jöttek létre:



Az elkészült nézetek mind Razor Pages alapúak, tehát tartozik hozzájuk egy PageModelből származó objektum. Ez az objektum maga a nézet modellje és vezérlője is egyben, így egy egységnyi MVC komponensnek fogható fel.

Önálló feladatok

Indítsuk el az alkalmazást és nézzük meg, hogy milyen template-ek jöttek létre.

Az Index művelet listázza egy táblázatban az összes hirdetést.

- Az OnGet metódus meghívásakor az adatbáziskontextusból lekérdezi a page az összes hirdetést, és ezt eltárolja a modell Advertisement tulajdonságában.
- A nézetben a hirdetésekhez ún. HTML helperekkel generálódnak a konkrét megjelenítendő adatok: a fejlécekhez a HtmlHelper DisplayNameFor() metódusát, az értékekhez a DisplayFor() metódusát használja a generált kód. A bejárás a klasszikus Razor szintaxissal történik, a HTML-ben generáláshoz használhatunk bármilyen C# funkcionalitást, pl. a foreach-et.
- A táblázat előtt egy, a táblázat utolsó sorában 3 HTML link található a szerkesztő, részletes és törlés oldalakra. Itt vastaggal vannak szedve nem HTML-specifikus attribútumok, amelyek konvenció szerint asp- előtaggal kezdődnek, ezek az ún. TagHelperek. Ezek valójában az adott HTML elemen dolgozva szerver oldali kódot futtatnak, a mi esetünkben az asp-page és asp-route-id attribútumok az <a> elem href attribútumát fogják megfelelően kitölteni, ezt a böngészőben meg tudjuk majd vizsgálni.

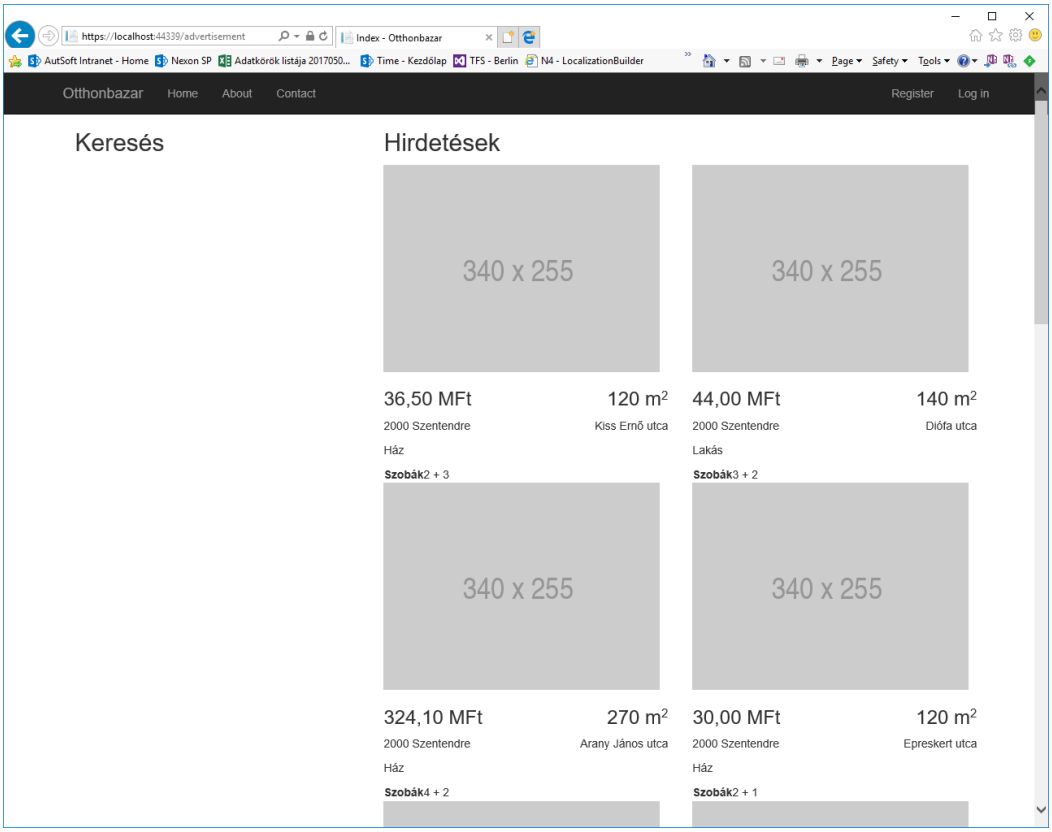
Indítsuk el az alkalmazást, vizsgáljuk meg a **/Advertisements** oldalt és teszteljük a működést!

- A navigáció és a CRUD műveletek minden oldalon működnek, vizsgáljuk meg az URL-eket, hol érhetők el az egyes oldalak!
- Vizsgáljuk meg az így generált HTML-t (a böngészőben, F12 segítségével)!
- Nézzük meg, hogyan működik a CityId kitöltése és az ehhez kapcsolódó mögöttes modell feltöltése!
- Meg tudjuk javítani az AdvertisementType megjelenítését is, ehhez használjuk a `Html.GetEnumSelectList<AdvertisementType>()` kódrészletet!
- Ahhoz, hogy a listázó felület jöjjön be, amikor a gyökér URL-re navigálunk, az `Advertisements/Index.cshtml`-ben módosítsuk a page direktívát az oldal tetején erre: `@page "/"`
    - Ha megpróbáljuk elindítani az alkalmazást, két Page is a gyökér URL-re próbál reagálni: az eredeti Index és az Advertisements/Index, előbbi konvenció szerint, utóbbi a fenti konfiguráció miatt. Oldjuk fel az ütközést azáltal, hogy töröljük a Pages/Index page-et! Ezután a navigációs sávban a megfelelő asp-page TagHelpereket módosítanunk kell /Index-ről /Advertisements/Index-re.
- Razor kódot (.cshtml fájlt) szerkeszthetünk az alkalmazás futása közben is, a fordítandó fájlokat (.cs) viszont csak az alkalmazást leállítva lehet.

Hirdetéslista megjelenítésének testreszabása

Az automatikusan legenerált megjelenés egy adminisztrátori felületen lehet, hogy megfelelő, de mivel jelen alkalmazásnak ez lesz a kezdő oldala is, ezért mindenképpen szebbé kell tennünk.

A cél, hogy az alábbi oldalelrendezést valósítsuk meg. Ehhez a generált Viewból induljunk ki és Bootstrap segítségével formázzuk meg az oldalt. A Bootstrap a generált oldalhoz már linkelve van (a konkrét módszert a Pages/Shared/_Layout.cshtml-ben nézhetjük meg). Az alább látható elrendezés 40 sor HTML kóddal megvalósítható.



Kiinduláshoz az alábbi kódrészletet is használhatjuk:

```
<div class="container">
    <div class="row">
        <section class="col-lg-4">
            <h2>Keresés</h2>
        </section>
        <section class="col-lg-8">
            <h2>Hirdetések</h2>
            @foreach (...)
            {
                <article class="..."></article>
            }
        </section>
    </div>
</div>
```

Tippek a megvalósításhoz:

- Adott méretű helyőrző képet a https://via.placeholder.com/340x255-lel tudunk generálni
- Használjuk a Bootstrapben lévő `row` és `col-*` osztályokat.
- Ha listát szeretnénk megjeleníteni akkor egy `@foreach` segítségével tudunk a Model elemein végiglépkedni és a ciklusban egy sablon alapján tudjuk generálni a HTML kódot.
- Jobbra igazításhoz használjuk a `pull-right` CSS osztályt, a képen pedig az `img-responsive` classt.
- A szobák megjelenítésénél használjuk a `<dl><dd>Szobák</dd><dt> x + y </dt></dl>` tageket.

Ezt követően egészítsük ki úgy, hogy a képre kattintva a részletes oldalra navigáljon az alkalmazás.

Ha egy hiperhivatkozást szeretnénk készíteni az az <a> taggel tudjuk megtenni, és mivel egy Actionre szeretnénk átirányítani ezért nem a href attribútumát adjuk meg, hanem kihasználva a Razorban a `asp-page` és `asp-route-*` TagHelpereket, az alábbi módon adunk meg egy olyan hivatkozást, amit a részletek oldalra navigál úgy, hogy az URL-be beleteszi a hirdetés ID-ját is. Erre látunk példát a listázó oldalon is.

```
<a asp-page="Details" asp-route-id="@advertisement.Id">
```

Egyúttal oldjuk meg azt is, hogy ha az adatbázisban az `ImageUrl` értéke nem üres, akkor a kép jelenjen meg, ha pedig üres, akkor a helyőrző kép. Ehhez először másoljuk át a `wwwroot/images` könyvtárba a *ASP.NET-Core-kiindulo.zip*-ben található képeket.

Enum lokalizációja

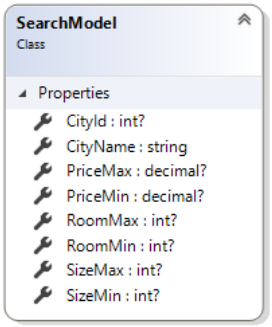
Az épület típusa jelenleg egy enum, aminek a szövege megjelenik, de mivel a kódban angolul vettük fel az értékeket ezért angolul jelenik meg. Lokalizáljuk az enum értékeket a `[Display(Name = "")]` segítségével (az egyes enum értékek fölé kell tenni). Ne feledjük usingolni a `System.ComponentModel.DataAnnotations` névteret. Az értékek: *Lakás*, *Ház*, *Nyaraló*, *Építési telek*. A megjelenítéshez használjuk a nézetben elérhető HtmlHelper objektumot (@Html.Display...)!

Keresés

A bal oldali részen kihagyott helyre egy keresőt fogunk implementálni.

SearchModel létrehozása

Hozzuk létre a `Models` mappát, benne pedig egy `SearchModel` nevű modellt, amit a kereséshez használunk majd. Ez fogja tartalmazni a keresés során megadott paramétereket. Keresni lehet majd városra, árra, szobaszámra és alapterületre. Ehhez az alábbi osztályt implementáljuk.



Vegyük fel a szükséges `[Display(Name = "")]` attribútumokat az egyes property-k elé, hogy a felületen a szövegdobozok előtt a címkék helyesen jelenjenek meg. Figyelj arra, hogy minden mező opcionális, így az értéktípusoknál használd a beépített `Nullable<T>` megfelelőket (pl. `int?`).

Az Advertisements/Index.cshtml.cs IndexModel Razor Page-be vegyünk fel egy ilyen tulajdonságot, amit a beérkező HTTP kérésből adatkötni fogunk a modellhez:

```
[BindProperty(SupportsGet = true)]
public SearchModel Search { get; set; }
```

Ezután a „Keresés” címmel ellátott szekcióhoz vegyük fel az alábbi logika alapján a keresési feltételeket leíró blokkot:

- Input-groupok létrehozásához segítség: http://bootstrapdocs.com/v3.0.3/docs/components/#input-groups-basic
- A min-max inputoknál 5-5 oszlop szélesek legyenek a szövegdobozok és egy 2 oszlop széles span-ba kerüljön a kötőjel (- vagy –).
- Tipp: a form-control-ok köré hozz létre befoglaló div-et, és annak állítsd be, hány oszlop széles legyen! Használd a col-lg-{n} osztályokat!

Példa az ár inputra:

```
<form class="form-horizontal">
    <div class="form-group row">
        <label class="col-xs-12">Ár</label>
        <div class="col-xs-5">
            <div class="input-group">
                <input asp-for="Search.PriceMin" class="form-control" />
                <span class="input-group-addon">M Ft</span>
            </div>
        </div>
        <span class="col-xs-2 text-center">&ndash;</span>
        <div class="col-xs-5">
            <div class="input-group">
                <input asp-for="Search.PriceMax" class="form-control" />
                <span class="input-group-addon">M Ft</span>
            </div>
        </div>
    </div>
    <div class="form-group row">
        <div class="col-xs-3">
            <input type="submit" value="Keresés" class="btn btn-default" />
        </div>
    </div>
</form>
```

Teszteljük, hogy hogyan működik a kereső! A Keresés gombra kattintva URL-be bekerülnek a szűrési feltételek, mert a teljes form tartalmát elküldjük. Az URL-ből a Page-ben a BindProperty-vel ellátott tulajdonság értéke ezekhez adatköt, megtörénik az objektum feltöltése a szűrési feltételekkel. Ezután a nézet renderelésekor újra kiolvassuk az ezekben található értékeket, ezért ki tudjuk tölteni az így elküldött szűrési feltételeket! Már csak az van hátra, hogy a keresést ténylegesen megvalósítsuk.

TÉNYLEGES KERESÉS MEGVALÓSÍTÁSA

Már csak annyi dolgunk van hátra, hogy megvalósítsuk a tényleges szűrést a beérkező szűrési feltételek alapján. Az alább látható ár szerinti szűrésnek megfelelően végezd el a további szűréseket is (**méret**, **szobaszám**, **város neve alapján részleges találat**)!

```
public async Task OnGetAsync()
{
    IQueryable<Advertisement> advertisements = _context.Advertisement.Include(a => a.City);

    if (Search.PriceMin != null)
        advertisements = advertisements.Where(a => a.Price >= Search.PriceMin.Value);

    if (Search.PriceMax != null)
        advertisements = advertisements.Where(a => a.Price <= Search.PriceMax.Value);

    Advertisement = await advertisements.ToListAsync();
}
```

ÚJ HIRDETÉS HOZZÁADÁSA

A bejelentkezett felhasználók számára tegyük lehetővé, hogy új hirdetést adjanak fel.

LINK AZ ÚJ HIRDETÉS FELADÁSÁRA

A _Layout.cshtml-ben a Home, About és Contact linkeket vegyük le és helyette legyen egy Hirdetések és egy Hirdetés feladása link, melyből az utóbbi csak belépett felhasználók számára látható. Fontos, hogy azáltal, hogy egy funkció nem elérhető, még továbbra is validálnunk kell a módosítás pillanatában, hogy a felhasználó jogosult-e a műveletre! Tehát csak az nem elegendő, hogy a linket „elrejtjük” a nem belépett felhasználók elől, amikor menteni próbálnak, újra meg kell vizsgálnunk!

```
<div class="navbar-collapse collapse">
   <ul class="nav navbar-nav">
      <li><a asp-page="/Advertisements/Index">Hirdetések</a></li>
      @if (Context.User.Identity.IsAuthenticated)
      {
         <li><a asp-page="/Advertisements/Create">Hirdetés feladása</a></li>
      }
      <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
      <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
      <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
   </ul>
   <partial name="_LoginPartial"/>
</div>
```

HIRDETÉS FELADÁSA DESIGN

Alakítsuk át az oldalt úgy, hogy a címe „Hirdetés feladása” legyen, és két hasábban jelenjenek meg az adatok.

- A bal oldali hasáb 7, a jobb oldali pedig 5 egység széles legyen.
- Az egyes elemeknél a label 3, a textbox pedig 4 egység szélesek legyenek.
- Figyeljünk rá, hogy az egyes szövegdobozok előtt magyar szövegek jelenjenek meg, erre használjuk a `[Display( Name = „”]` attribútumot, illetve a `Required` attribútumokat is egészítsük ki úgy, hogy a hibaüzenet szövege is magyar legyen, `[Required( ErrorMessage= „”]`. Mivel itt egy `Advertisement` jelenik meg, ezért azt az osztályt kell felattribútálni. Mivel az adatbázis szempontjából csak a `Required`

attribútum a fontos, az `ErrorMessage` már nem számít, ezért az adatbázis nem fog változni, azt nem kell újragenerálni.

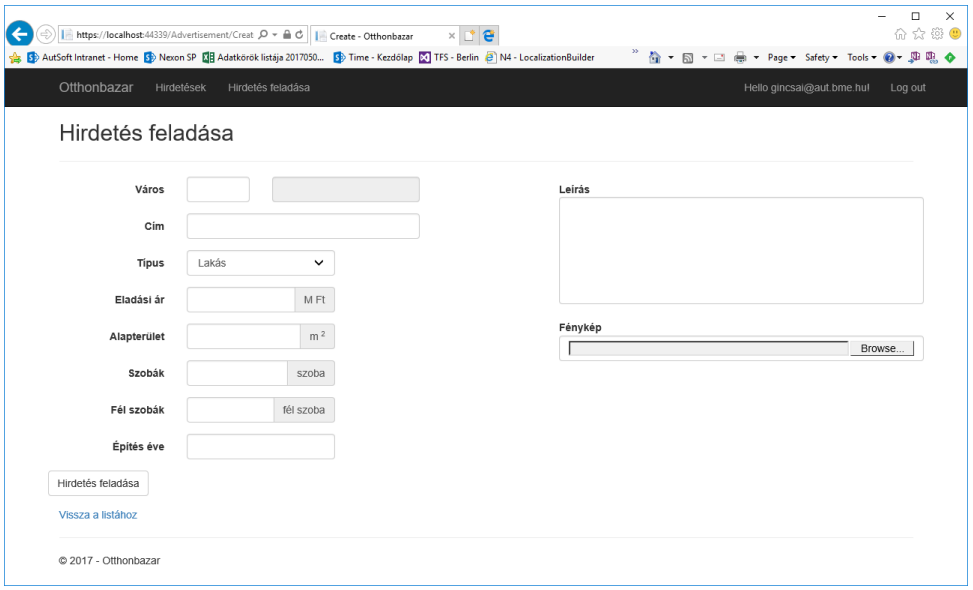
- A cím esetében a Razor kód az alábbiak szerint fog alakulni. Ez alapján már a többi kód is elkészíthető.

```
<div class="row form-group">
   <label asp-for="Address" class="col-lg-3 control-label"></label>
   <div class="col-lg-6">
       <input asp-for="Address" class="form-control" />
   </div>
   <span asp-validation-for="Address" class="text-danger"></span>
</div>
```

- Ahhoz, hogy a típusok is megjelenjenek helyesen, a generált kódot ki kell egészíteni, hogy tudja a View, hogy milyen enum értékeit kell a legördülő listában megjeleníteni. Ezt az alábbi módon tehetjük meg. Vagy usingoljuk az Otthonbazar.Data névteret, vagy a FQN-et adjuk meg.

```
<select asp-for="AdvertisementType" class="form-control"
        asp-items="Html.GetEnumSelectList<Otthonbazar.Data.AdvertisementType>()">
</select>
```

- Az elkészült oldalnak így kell kinéznie



Irányítószám alapján város automatikus kitalálása

A cél, hogy ha a felhasználó megadja az irányítószámot, akkor az alapján automatikusan töltsük ki a város nevét. Ehhez egy aszinkron HTTP kommunikációt használó végpontra lesz szükség, ami az irányítószám alapján visszaadja a város nevét. Ezt az API-t kliens oldalról AJAX segítségével akkor kell meghívni, ha a felhasználó kikattint az irányítószám szövegdobozból.

- Adjunk hozzá a `Create` page-hez egy új metódust OnGetZip névvel, ami aszinkron módon (a kliens szempontjából aszinkron, a szerveren blokkolva) visszaadja az adatbázisból azt a várost, aminek az irányítószámát megadtuk.

```
public ActionResult OnGetZip(int zip) => new JsonResult(_context.Cities.FirstOrDefault(c => c.Zip
== zip.ToString()));
```

- Ezt követően a `Create.cshtml`-ben a `Scripts` szekcióban készítsük el a document ready eseménykezelőt, és ebben iratkozzunk fel az `City_Zip` ID-val rendelkező szövegdoboz `blur` eseményére.
- Az eseménykezelőben hívjuk meg a /Advertisements/Create?handler=Zip&zip={...} URL-t a $.ajax segítségével. Ha sikeres a hívás akkor a `City_Name` ID-jú szövegdobozba másoljuk be a visszakapott város nevét.
- Ha a kattintáskor (blur) üres a szövegdoboz értéke, akkor ne hívjuk meg az AJAX kérést.

```
@section Scripts {
   @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}

<script>
$(document).ready(function () {
   $("#City_Zip").on("blur", function (event, b) {
      // Ha nincs megadva irányítószám, akkor nem hívjuk meg az ajaxot.
      if ($("#City_Zip").val() === "")
         return;
      $.get("/Advertisements/Create?handler=Zip&zip=" + event.currentTarget.value).then(function
(data) {
         $("#City_Name").val(data.name);
      });
   })
});
</script>
}
```

- Próbáljuk ki az alkalmazást.

Új hirdetés elmentése adatbázisba

A `Create` page POST-os actionjében/handlerében implementáljuk az adatbázisba mentést.

- Először is csak akkor menthet a felhasználó, ha be van lépve. Ezt a `[Authorize]` attribútummal tudjuk kikényszeríteni az action előtt, vagy az actionön belül a felhasználó objektumra vizsgálva.
- Először ellenőrizni kell, hogy az adatok érvényesek-e, amit a `ModelState.IsValid` segítségével lehet megtenni.
- Először le kell kérdezni azt a várost, aminek az irányítószámát megkaptunk az adatbázisból, és a model-ben a City tulajdonságot erre kell beállítani.
- Ezt követően elmenthetjük a DB-be a módosításokat.
- Ellenőrizzük a listáró oldalon, hogy létrejött-e a hirdetés

OPCIONÁLIS FELADAT: KÉP FELTÖLTÉSE

Ahhoz, hogy képet is tudjunk feltölteni a hirdetéshez, az alábbi módosításokat kell elvégezni a kódon.

- A Create.cshtml-ben a form `enctype`-ja legyen `multipart/form-data` (csak így lehet fájlt is felküldeni a formmal együtt).

```
<form asp-action="Create" class="form-horizontal" enctype="multipart/form-data">
```

- Az `AdvertisementController Create` actionjében második paraméterként várjunk egy `IFormFile`-t, amiben majd a kliens által küldött fájlt adatait fogjuk megkapni.

```
public async Task<IActionResult> Create(Advertisement advertisement, IFormFile uploadImage)
```

- Ezt követően mielőtt az adatbázishoz nyúlnánk mentsük le a képet a fájlrendszerbe (persze csak akkor, ha tényleg töltöttek fel képet).
    - Ahhoz, hogy ezt meg tudjuk tenni, kellene tudni, hogy mi a webalkalmazás gyökér könyvtárának a fizikai elérési útvonala, amihez képest meg tudjuk majd adni, hogy hova mentse le a fájlt. Ehhez a konstruktorban is eltárolhatnánk a `IHostingEnvironment` típusú paramétert, amit a DI fel tud oldani, de esetileg ezt kérhetjük a DI-tól az action paramétereként is a `[FromServices]` attribútummal.

```
public async Task<IActionResult> Create(Advertisement advertisement, IFormFile uploadImage ,
[FromServices]IHostingEnvironment hostingEnvironment) {
   if (ModelState.IsValid) {
      // File mentése
      if(uploadImage != null && uploadImage.Length > 0 ) {
         var webRoot = hostingEnvironment.WebRootPath;
         advertisement.ImageUrl = $"image-{DateTime.UtcNow.ToString("yyyyMMdd-HHmmss")}.jpg";

         // wwwroot/images könyvtárba fogja menteni.
         var filePath = Path.Combine(webRoot, "images", advertisement.ImageUrl);

         using (var stream = new FileStream(filePath, FileMode.Create)) {
            await uploadImage.CopyToAsync(stream);
         }
      }

      // Város felolvasása irányítószám alapján.
      var city = _context.Cities.Single(c => c.Zip == advertisement.City.Zip);
      advertisement.City = city;

      // Hirdetés adatainak mentése DB-be.
      _context.Add(advertisement);
      await _context.SaveChangesAsync();

      return RedirectToAction(nameof(Index));
   }

   return View(advertisement);
}
```

FELTÖLTÖTT KÉPHEZ ELŐNÉZET GENERÁLÁSA

- A `Create.cshtml`-ben az uploadImage után tegyünk oda egy középre igazított img taget, aminek az src-je kezdetben a `https://via.placeholder.com/340x255` URL-re van állítva. Az `img-responsive` Bootstrap osztállyal a képünk mindig a megfelelő méretű lesz (max. 100% x 100% helyet foglal majd el).

```
<div class="text-center">
      <img id="image" class="img-responsive" src="https://via.placeholder.com/460x250" />
</div>
```

- Vegyük fel a már meglevő JavaScriptünkhöz ennek a kezelőjét is:

```
$(function () {
   $("#City_Zip").on("blur", function (event, b) { })

   $("input[name='uploadImage']").change(function (event) {
      var input = event.target;
      if (input.files && input.files[0]) {
         var reader = new FileReader();

         reader.onload = function (e) {
            $('#image').attr('src', e.target.result);
         };

         reader.readAsDataURL(input.files[0]);
      }
   });
});
```

A LABOR ÉRTÉKELÉSE

A labor során végzett munka értékeléseként az alábbiakat szükséges megvizsgálni:

<table>
<tr><td>Adatbázis:
- Létrejött-e az adatbázis a saját neptunkóddal (SQL Server Object Explorerből látható)
- Létrejött-e a megfelelő adatséma (Advertisement, City és AspNet*** táblák, a két saját tábla nem többesszámosított nevű), közöttük létrejött a külső kulcs kényszer és megfelelő kötelezőségi megkötések
- Az adatbázisban létrejöttek-e az ősfeltöltéshez szükséges adatok (városok és hirdetések)
- Létrejött-e a teszt felhasználó saját neptunkóddal</td><td rowspan="2">Vezetett rész, 2-es érdemjegy</td></tr>
<tr><td>Razor Pages:
- Létrejöttek-e az automatikus generált Razor Pages oldalak a hirdetések listázásához, megtekintéséhez, szerkesztéséhez, létrehozásához és törléséhez
- A gyökér URL nem az Index Page-re, hanem az Advertisments/Index Page-re navigál
- A hirdetések egyedi listázó oldalon jelennek meg
- A hirdetéshez a hozzá tartozó kép, annak hiányában helyőrző jelenik meg a listázó oldalon
- A hirdetés típusát reprezentáló enum a felületen lokalizálva jelenik meg
- A képre kattintva a hirdetés részletes oldalára navigál az oldal</td><td>+1 érdemjegy</td></tr>
<tr><td>Keresés:
- A kereséshez szükséges modell elkészült, minden tulajdonságának értéke nullázható
- A keresés mezői láthatók a felületen, keresés után a beírt érték az oldal újratöltődése után is megmarad
- A keresés képes szűrni az összes felületen látható bemeneti paraméterre</td><td>+1 érdemjegy</td></tr>
<tr><td rowspan="2">Hirdetés létrehozása:
- Hirdetés létrehozható, a validációs hibák láthatók az oldalon
- A létrehozás oldal az elvárt design alapján készült
- Nem bejelentkezett felhasználó nem tud hirdetést létrehozni (nem csak odanavigálni)
- A hirdetéshez lehetséges képet is feltölteni, ami a listázásnál megjelenik (opcionális)
- A hirdetés képének feltöltésekor a kép előnézete is látható (opcionális)</td><td>+1 érdemjegy</td></tr>
<tr><td>+2 imsc pont[2]</td></tr>
</table>

[2] A két opcionális feladatért 1-1 imsc pont adható, de csak abban az esetben, ha az összes többi feladat teljesítve lett.