ASP.NET Core MVC: megjelenítés

A második gyakorlaton folytatjuk a TodoList fejlesztését. Az eddigi listaelem modellt kiegészítjük egy adatbázisban tárolt képpel, illetve a webes felületen megjelenítjük a listákat és a hozzájuk tartozó elemeket. Az elemeket lehetőségünk lesz név vagy határidő szerint rendezni.

A munka megkezdése előtt állítsuk vissza `Program.cs` fájl eredeti tartalmát!

1 Modell réteg

1.1 Szolgáltatás-interface

Az előző gyakorlaton készített `TodoListService` osztályát is emeljük át a projektbe! A listákra vonatkozó műveletek közül az összes lista lekérésére (`GetLists`) és az azonosító alapján való lekérésre (`GetListById`) lesz szükség. Új metódusként definiáljuk egy lista részletek lekérést `GetListDetails(int id)`, illetve listaelem azonosító alapján történő lekérését (`GetItem(int id)`).

Az osztálynak számára hozzunk létre egy `ITodoListService` interfészt is, amelyben szerepeljenek a szükséges függvények.

Megjegyzés: Függőségi befecskendezést (*dependency injectiont*) a legtöbb esetben interfészek segítségével szoktunk megvalósítani. Bár ez nem kötelező, az interfészek használatának számos előnye van:

1. Az interfésznek több különböző megvalósítása létezhet.
2. Egyfajta kötelezettséget fogalmaz meg a megvalósítás számára.
3. Egy osztály több interfészt is megvalósíthat.
4. Segíti a tesztelést.
5. Segíti az egyes réteget szétválasztását, párhuzamosan történő fejlesztését.
6. Könnyebbé teszi a kód bővítését, változtatását.

2 Program.cs és az adatbázis-kontextus

Az ASP.NET Core keretrendszerben az ún. *service provider* egy IoC tárolóként (*IoC container*) funkcionál, azaz egy olyan *Inversion of Control* paradigmájú komponens, amely lehetőséget ad szolgáltatások megvalósításának dinamikus (futási idejű) betöltésére. Az IoC tároló egy központi regisztráció, amelyet minden programkomponens elérhet és felhasználhat.

A `Program.cs` állományban a `WebApplicationBuilder` osztályhoz tartozó `IServiceCollection` segítségével végezhetjük a szolgáltatások hozzáadását a tárolóhoz, amelyet a `Services` tulajdonságon keresztül érhetünk el.

2.1 Adatbázis kontextus regisztrálása

Hívjuk meg a `builder.Services.AddDbContext` metódust, paraméterül egy lambda-kifejezést adunk, amelynek törzsében elvégezhetjük azokat a beállításokat amelyeket az előző gyakorlaton a `TodoListDbContext` osztály `OnConfiguring` metódusában állítottunk be. A konfigurációt most a `WebApplicationBuilder` osztály `Configuration` tulajdonságán keresztül érjük el.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<TodoListDbContext>(options =>
{
```

```
    IConfigurationRoot configuration = builder.Configuration;
    DbType dbType = configuration.GetValue<DbType>("DbType");

    switch (dbType)
    {
        case DbType.SqlServer:
            // Need Microsoft.EntityFrameworkCore.SqlServer package for this
            options.UseSqlServer(
                configuration.GetConnectionString("SqlServerConnection"));
            break;

        case DbType.Sqlite:
            // Need Microsoft.EntityFrameworkCore.Sqlite package for this
            options.UseSqlite(
                configuration.GetConnectionString("SqliteConnection"));
            break;
    }

    // Use lazy loading
    // (don't forget the virtual keyword on the navigational properties also)
    options.UseLazyLoadingProxies();
});
```

A `TodoListDbContext` osztályban az `OnConfiguring` metódusra ezután már nem lesz szükség, helyette egy `DbContextOptionsBuilder<TodoListDbContext>` típusú paramétert váró üres konstruktort hozzunk létre, amely meghívja a szülőosztály konstruktorát (`base`). Ilyen módon az adatbázis kontextus konfigurálását kívülről, függőségi befecskendezéssel (*dependency injection*) végezhetjük.

2.2 TodoListService regisztrálása

Az IoC tárolóba regisztráljuk a `TodoListService` típust is, annak interfészével (pl. `AddTransient`)!

```
builder.Services.AddTransient<ITodoListService, TodoListService>();
```

Megjegyzés: Az osztályok regisztrálására az alábbi opcióink vannak:

1. `AddTransient`: *Dependency injection* esetében minden alkalommal új példány jön létre az osztályból.
2. `AddSingleton`: A webalkalmazás elindítását követően legelső alkalommal jön létre az objektum példány, utána minden egyes alkalommal ugyanaz a példány adódik át, akár különböző klienstől származó kérések között is megosztva.
3. `AddScoped`: Az adott kérés (*request*) élettartama alatt ugyanaz az objektum példány adódik át.

2.3 DbInitializer futtatása

A szolgáltatások regisztrálása után a `WebApplicationBuilder` osztály `Build` metódusa segítségével létrehozunk egy a `WebApplication` osztály egy példányát (`app`).

A webalkalmazás futtatása előtt szeretnénk inicializálni az adatbázist. Ezt a `DbInitializer` statikus osztály `Initialize` metódusával tudjuk megtenni, amely paraméterként várja az adatbázis kontextus egy példányát.

Mivel az adatbázis kontextus osztályok alapértelmezés szerint *scoped* szolgáltatásként vannak regisztrálva a tárolóba, ezért a példány lekérése két lépésből áll:

1. Hozzunk létre egy új service scope-t az előbb létrehozott webalkalmazáshoz tartozó `IServiceProvider` `CreateScope` metódusának segítésével.
2. A scope-hoz tartozó `IServiceProvider` `GetRequiredService<TodoListDbContext>` metódusának használatával megkapjuk az adatbázis kontextus egy példányát.

```
using (var serviceScope = app.Services.CreateScope())
using (var context = serviceScope.ServiceProvider.GetRequiredService<TodoListDbContext>())
```

```
{
    DbInitializer.Initialize(context);
}
```

Megjegyzés: az olyan *transient* és *singleton* szolgáltatások esetében, amelyeknek nincs *scoped* függőségük, használhatjuk közvetlenül webalkalmazáshoz tartozó `IServiceProvider`-t, új service scope létrehozása nélkül is.

3 Controller (vezérlő) réteg

3.1 Controller és nézetek létrehozása

A `Controllers` mappára jobbklikkelve az *Add Controller* menüpont alatt adhatunk a projekthez új controller osztályt. Adjuk meg az új controllerhez tartozó entitást és az adatbázis-kontextust. A legegyszerűbb, ha egyből nézetekkel együtt hozzuk létre a controllert, ekkor a `Views` mappában létrejön egy, a controllerünknek megfelelő nevű új mappa, amely az alapértelmezett CRUD műveletek mindegyikéhez tartalmaz egy-egy nézetet, amelyek `.cshtml` kiterjesztést kapnak. Hozzunk létre controllert és nézeteket a `List` entitáshoz!

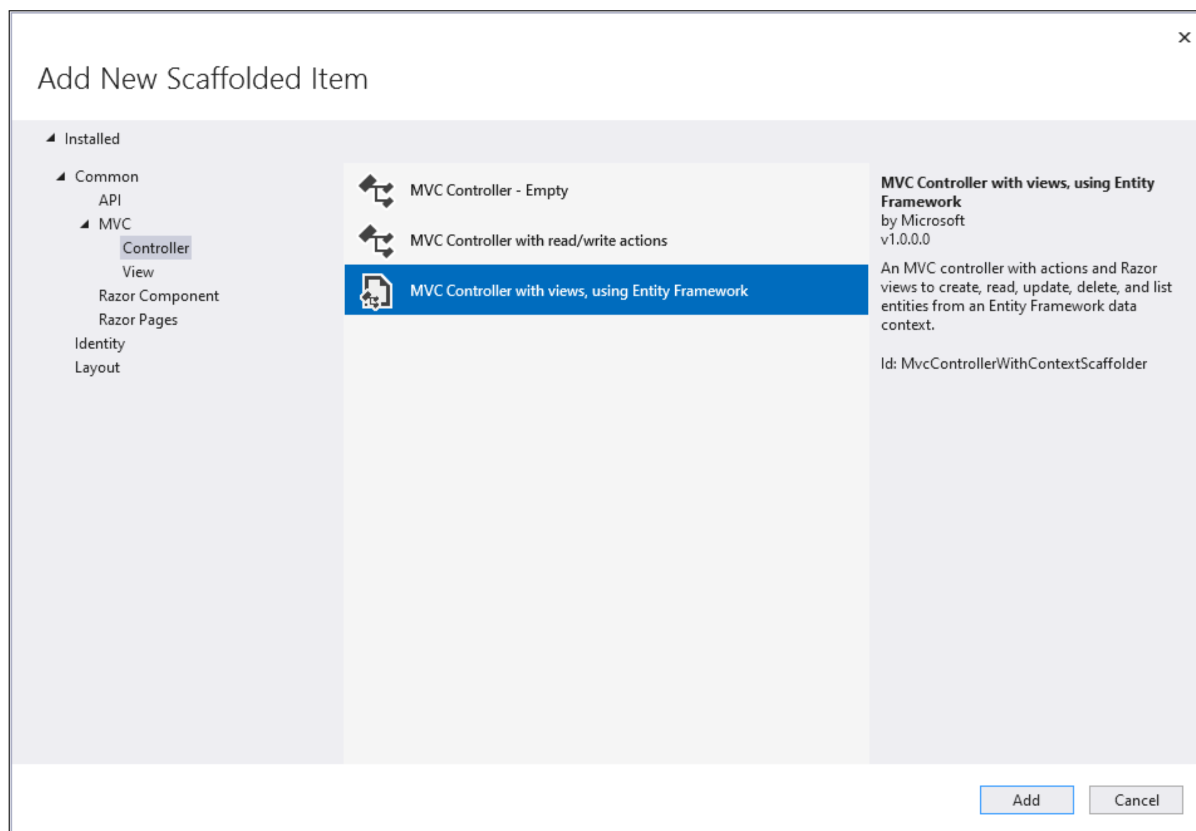

Figure 1: CRUD controller osztály generálása

Miután elkészültünk a controller generálásával, gondoskodjunk róla, hogy a controller `TodoListDbContext` példány helyett egy `TodoListService` objektummal rendelkezzen. Ezt a vezérlő osztály konstruktora átveheti, és az IoC tároló automatikusan befecskendezi majd.

Megjegyzés: az első controller osztály generálásakor a projekthez automatikusan hozzáadásra kerül a `Microsoft.VisualStudio.Web.CodeGeneration.Design` NuGet package, erre a generáláshoz szüksége is van.

3.2 Listaentitások megjelenítése

A generált controllerbeli metódusok közül az `Index` felelős a listák megjelenítéséért, a nézet rétegben pedig az azonos nevű nézet (a controller által generált metódusokhoz automatikusan létrejön egy-egy azonos nevű nézet, ha ezt az opciót választjuk). Az `Index` metódusban adjunk vissza egy nézetet, amely visszaadja a listákat (a service megfelelő metódusának meghívásával).

A nézetben a listákhoz adjunk egy-egy linket (`<a>`), amely a listához tartozó listaelemeket jeleníti meg!

3.3 Listaelemek megjelenítése

A listaelemek megjelenítését az `ListController` osztály `Details` metódusa végzi. A `Details` adja vissza a kapott azonosítónak megfelelő lista nézetét.

A `Details.cshtml` fájl tartalmát írjuk át úgy, hogy az elemeket táblázatos formában mutassa!

3.4 Listaelemek sorba rendezése

A `ListController` osztály `Details` metódusát egészítsük ki az elemek sorba rendezésével! A metódus várjon egy második paramétert, amely a rendezés típusát fogja meghatározni (`sortOrder`, típusa `SortOrder enum`). A `Details`hez tartozó nézetben az listaelemek nevét és határidejét megjelenítő oszlopok fejléceit (`Name` és `Deadline`) tegyük linkké (`<a>`), amelyek a `Details` akciót hívják, és a `sortOrder` paraméterhez egy-egy értéket kötnek, a következő módon: `asp-route-sortOrder="@ViewData["NameSortParam"]"` (a határidő esetében természetesen másik kulcsra lesz szükség, pl. `@ViewData["DeadlineSortParam"]`).

A `ViewData`ban lévő kulcsoknak a controllerben a `sortOrder` aktuális értékének megfelelően adjunk értéket! Ha a `sortOrder` értéke név szerint növekvő rendezés volt (ez lesz az alapértelmezett értéke), váltsuk át név szerint csökkenőre (pl. `NAME_DESC`). Ennek megfelelően váltogassunk a határidő szerinti rendezések között a ViewData másik kulcsánál (pl. `DEADLINE_ASC` és `DEADLINE_DESC`)!

Ezután a `sortOrder` értékétől függően rendezzük a korábban lekért listaobjektum elemeinek sorrendjét.

4 Képkezelés

A listaelemeket reprezentáló modell osztályt (`Item`) egészítsük ki egy új propertyvel: *Kép* (`Image`, típusa bájttömb). Listaelem létrehozásakor lehetőségünk lesz a webes felületen keresztül kiválasztani egy képet, amit beolvasás után bájttömbként tárolunk az adatbázisban. A korábbi `Add-Migration` paranccsal készítsünk új migrációt az adatbázis szerkezetének módosításához, illetve a `DbInitializer`ben az `EnsureCreated` helyett a `Migrate` metódust hívjuk meg!

Az adatbázishoz a `DbInitializer`en keresztül fogunk képeket adni. Hozzunk létre a projektben egy `App_Data` nevű mappát, ebben fogjuk tárolni a képeket. Az `appsettings.json`t egészítsük ki egy új kulcs-érték párral: `"ImageSource": "App_Data"`

A `DbInitializer` statikus osztály `Initialize` metódusa ezentúl várjon egy paramétert, amely megmondja, hogy hol kell keresnie a képeket (`imageDirectory`, típusa `string`)! Híváskor kérje le az `ImageSource` értékét az `appsettings.json`ből (használhatjuk például a `WebApplication` osztály `Configuration` tulajdonságát a konfiguráció elérésére). Az `Initialize` metódusban a listák és listaelemek létrehozását egészítsük ki a képek hozzáadásával:

1. Vizsgáljuk meg, hogy létezik-e az átvett könyvtár a `Directory` osztály `Exists` metódusa segítségével!
2. Ha létezik, a `Path` osztály `Combine` metódusával készítsük el a hozzáadni kívánt képek útvonalát!
3. Ha ez sikerült, és a fájlok léteznek (`File` osztály, `Exists` metódus), az inicializáló listában megadhatjuk az `Image` property értékét. Ez egy bájttömb, tehát a beolvasott képet konvertálni kell. Ezt a `File` osztály `ReadAllBytes` metódusával érhetjük el.

Megjegyzés: a `Directory`, `Path` és `File` osztályok a `System.IO` névtérben találhatóak.

4.1 Képmegjelenítés

1. Az `ItemsControllert` egészítsük ki egy új akcióval, amely a képmegjelenítésről fog gondoskodni (`DisplayImage`, egy azonosítót vár paraméterül, és egy `IActionResult`ot ad vissza).

2. Kérjük le az azonosítóhoz tartozó listaelemet!
3. Adjunk vissza egy `FileResult` típusú eredményt, ehhez használjuk a `Controller` ősosztályból örökölt `File` metódust. Ez argumentumként a listaelemhez tartozó képet és annak MIME type-ját (`image/png`) várja!
4. A `ListsController` osztály `Details` nézetében lévő táblázathoz adjunk egy új oszlopot *Image* fejléccel! Ha a listaelemhez tartozik kép, azt jelenítsük meg (`<img>`), aminek a forrása (`src` attribútum) egy `@Url.Action`, ami az `ItemsController` osztály `DisplayImage` akcióját hívja!
5. A `wwwroot` mappában található `site.css` fájlt egészítsük ki egy `img.item` nevű szelektorral, ami megadja, hogy egy kép maximális szélessége és hosszúsága egyaránt 50 pixel! A képek `class` attribútumának ennek megfelelően legyen `item` az értéke.