ASP.NET Core WebAPI + WPF: megjelenítés

Az ötödik gyakorlat célja, hogy egy webszolgáltatást és egy azt feldolgozó klienst nyújtsunk a korábban elkészített adatbázishoz. A WPF alapú kliensben szeretnénk a meglévő adatokat megjeleníteni, illetve követni a weboldalon keresztül eszközölt változtatásokat.

Amennyiben a skeleton projektből indulsz ki, folytasd a *WebApi* fejezetnél a munkafüzetet!

1 Refaktorálás

Mivel a webszolgáltatás és a weboldal is ugyanazokat az entitás modelleket használná, ezért érdemes egy külön projektet létrehozni a perzisztencia rétegnek, így elkerüljük a kódismétlést. Az új projekt *Class Library* legyen, így a perzisztencia kódredundancia nélkül újrafelhasználható lesz a korábbi weboldal és az új webszolgáltatás projektben is. *Target Frameworknek* `.NET 6.0`-t állítsunk be.

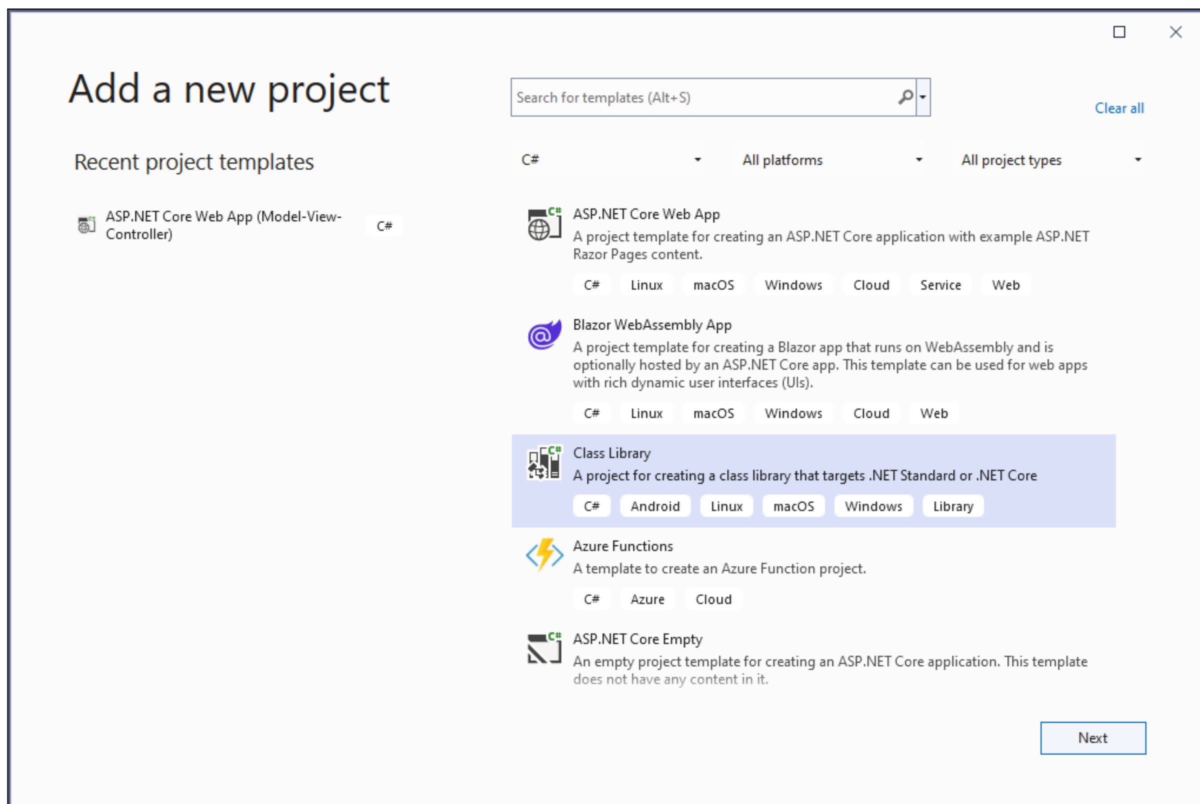

Figure 1: Projekt létrehozása a perzisztencia rétegnek

Emeljük át ebbe a projektbe a perzisztencia réteghez tartozó osztályokat:

- `ApplicationUser`
- `DbInitializer`
- `DbType`
- `List`

- `Item`
- `TodoListDbContext`
- `Migrations` mappa
- `Services` mappa

Adjuk hozzá az új projekthez az alábbi használt *NuGet* csomagokat, illetve írjuk át a névtereket az új struktúrát tükrözve.

- `Microsoft.EntityFrameworkCore`
- `Microsoft.AspNetCore.Identity.EntityFrameworkCore`
- Attól függően, hogy mit használtunk a migrációhoz: `Microsoft.EntityFrameworkCore.SqlServer` vagy `Microsoft.EntityFrameworkCore.Sqlite`

Adjuk hozzá a weboldal projekthez függőségként az új projektünket a *Build dependencies* → *Project Dependencies...* által.

Mivel szeretnénk, hogy a webszolgáltatás és a weboldal is ugyanazt az adatbázist érje el, ezért ha *Sqlite*-ot használunk, módosítsuk az adatbázis fájl útvonalát relatív egy mappával feljebb.

Miután javítottunk minden névtérelérést, ellenőrizzük, hogy ugyanúgy működik-e a weboldalunk, mint eddig.

2 Webszolgáltatás - WebApi

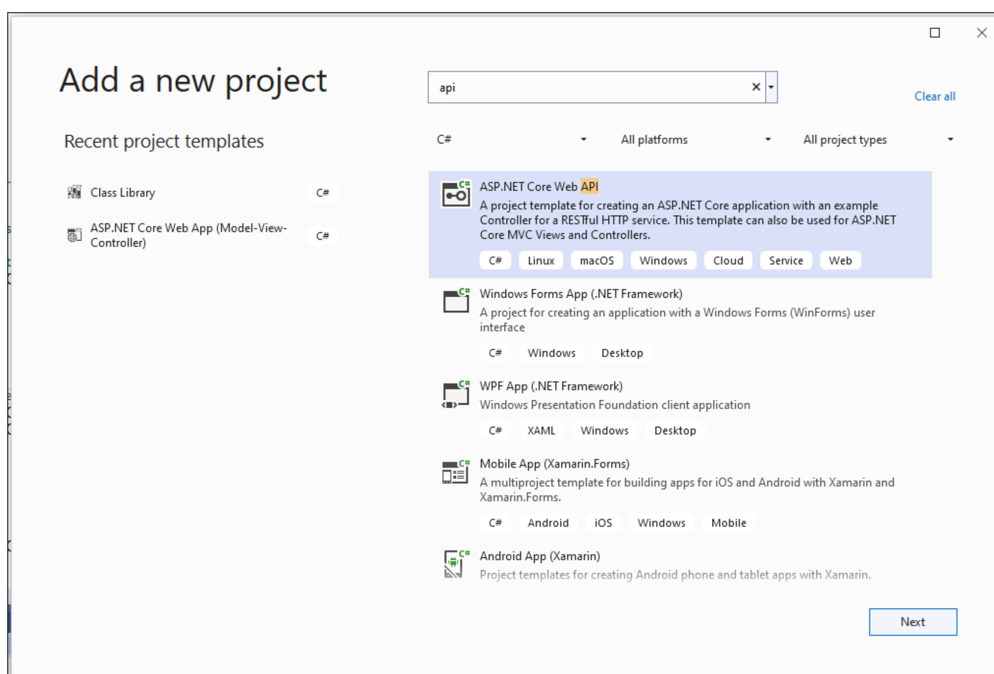

Figure 2: Projekt létrehozása a webszolgáltatásnak

Hozzunk létre egy új projektet a webszolgáltatásnak az *ASP.NET Core Web Application* → *API* template segítségével.

Miután tanulmányoztuk a projektben létrejött példakódot, töröljük azt. Ehhez a projekthez is adjuk hozzá függőségként a perzisztencia réteget a korábban ismertetett módon, illetve állítsuk be az adatbázis elérést a weboldalhoz hasonlóan a `Program.cs` és az `appsettings.json`-ban.

2.1 Vezérlők

Két végpontot szeretnék a webszolgáltatással biztosítani. Egy `/lists`-et, mellyel az összes listát, illetve egy `/items`-et, mellyel a (route-)paraméterül megadott azonosítójú listához tartozó összes elemet tudjuk lekérni. Ezekhez hozzunk létre kontrollereket a weboldalnál mar ismertetett módon, MVC helyett

API kontrollerből kiindulva, ügyelve arra, hogy az adatbázist ne közvetlenül a kontextuson, hanem a `TodoListService` osztályon keresztül érjük el.

A webszolgáltatásoknál különösen fontosak a válaszul küldött *hibakódok*, ezért ügyeljünk arra, hogy ha egy nem létező azonosítót kapunk az elemlekérdező végpontnál, akkor egy ennek megfelelő *404 Not Found* státusz kóddal térjunk vissza, pl. `return NotFound()`.

2.2 Manuális tesztelés

Mielőtt megírnánk a klienst, teszteljük le a webszolgáltatást egy erre alkalmas eszközzel (*Postman*, *httpprepl*). Tegyük indulóvá az új projektet, a tulajdonságaiban pedig jegyezzük fel az alkalmazásunk portját, illetve kapcsoljuk ki a *Launch browsert*, mivel nem szeretnénk böngészőt indítani a projekt indulásakor. Amennyiben nem írtuk át, a végpontjaink el vannak látva egy *api* prefixszel. Figyeljük meg, hogy az alapbeállítás szerint a végpontok JSON kódolásban küldik el az objektumokat. Ha megpróbálunk lekérni egy listához tartozó elemeket, azt tapasztalhatjuk, hogy a JSON szerializáló önmagára hivatkozó kört talált, ezért hibát dob. Ez az entitás modellünkben szereplő navigációs tulajdonságok miatt van. A szerializáló egy elemről visszajut az azt tartalmazó listába, ami szintén tartalmazza az összes elemet, így egy végtelen ciklust generál.

2.3 Adatátviteli objektumok

A kliens és a szerver közti kommunikáció általában úgynevezett adatátviteli objektumokkal (*Data Transfer Object - DTO*) történik. Ezek az adott kérésre szabott objektumok, melyek csak az átküldendő információt tartalmazzák. Esetünkben az entitásmodelljeink a navigációs tulajdonságokat kihagyva megfelelnek erre a célra. Hozzunk létre nekik megfeleltethető DTO-kat a perzisztencia rétegben, és a továbbiakban ezeket használjuk a szerver és a kliens közti kommunikációra az entitásmodellek helyett.

2.3.1 AutoMapper

A típusok közti megfeleltetést végezhetjük manuálisan (pl. konverziós operátorokkal vagy a tulajdonságok értékadásával), vagy automatizálhatjuk pl. az `AutoMapper` nevű *NuGet* csomag segítségével, amely lehetőséget ad a típusok közötti konvertálások automatikus lefuttatására a megadott beállítások alapján. Utóbbi használatához adjuk hozzá a WebApi projekthez az `AutoMapper` csomagot. A mappert regisztráljuk az IoC konténerben szolgáltatásként, ehhez szükségünk lesz az `AutoMapper.Extensions.Microsoft.DependencyInjection` nevű kiegészítő csomagra is.

```
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
```

Annak érdekében, hogy az `AutoMapper` felismerje a típusok közti megfeleltetéseket, hozzunk létre egy-egy profil osztályt az entitás típusok és a DTO típusok közti leképezésekhez:

```
public class ListProfile : Profile
{
    public ListProfile()
    {
        CreateMap<List, ListDto>();
    }
}
```

A listák és a listaelemek esetében ennél többre nincs szükségünk, mivel a különböző típusok tulajdonságai azonos névvel rendelkeznek. Készítsünk egy, a fentihez hasonló profil osztályt a listaelemekhez is. Ügyeljünk rá, hogy a profil osztályaink a `AutoMapper.Profile` osztályból származzanak, az *AutoMapper* csak így fogja őket felismerni.

> *Megjegyzés:* amennyiben nincs megadva külön konfiguráció, akkor az azonos nevű adattagok konverziója automatikusan megtörténik.

3 Asztali alkalmazás - WPF

A korábbi félévekből már ismerősnek kell lennie a keretrendszernek, így itt kevésbe lesz részletes a leírás. Hozzunk létre egy új projektet a kliensnek.

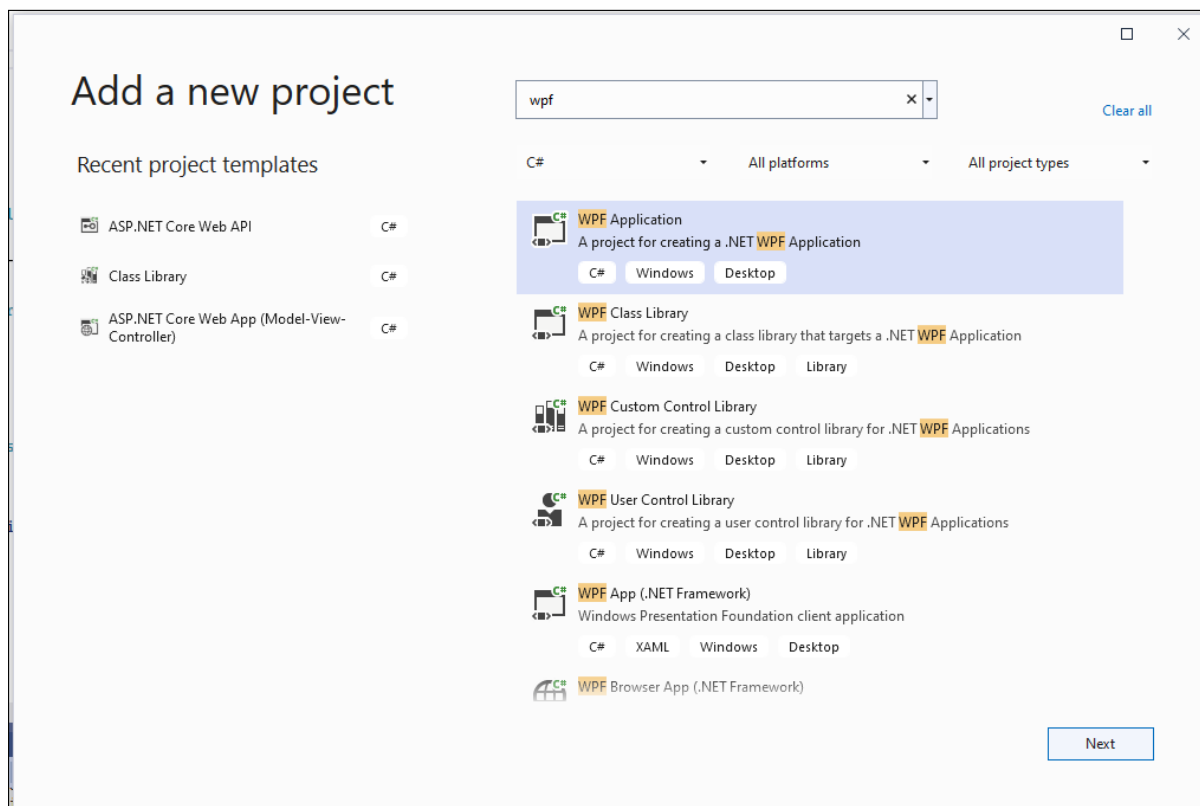

Figure 3: Projekt létrehozása a kliensnek

MVVM architektúrában fogunk dolgozni, ezért hozzunk létre három mappát ezen rétegeknek (*Model*, *View*, *ViewModel*). Mivel a DTO-k a perzisztencia réteg projektjében kaptak helyet, így állítsuk be azt a projekt függőségeként. Megjegyzendő, hogy ezen osztályoknak akár saját projektet is létrehozhatnánk, csökkentve így a perzisztencia réteghez való szoros függést. A korábban feljegyzett elérési útvonalat a webszolgáltatáshoz helyezzük el egy új konfigurációs fájlba. *Add* → *New item* → *Application Configuration File*.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="baseAddress" value="http://localhost:{port}/"/>
  </appSettings>
</configuration>
```

3.1 Modell réteg

Hozzunk létre egy szerviz osztályt, ami a webszolgáltatással történő kommunikációért felel. A kérések elvégzésére a `HttpClient` osztályra lesz szükségünk, állítsuk be a majd később konstruktoron keresztül átadott elérési útvonalat a `BaseAddress` tulajdonságban. Valósítsuk meg a két végpontot lekérdező metódust. A `HttpClient` objektum `GetAsync(string endpoint)` metódusával tudunk *GET* kéréseket intézni. A válasz `HttpResponseMessage` tartalmazza a kapott státuszkódot és a tartalmat JSON stringként, amit még deszerializálnunk kell. Ehhez adjuk hozzá a projekthez a `Microsoft.AspNet.WebApi.Client` csomagot, ami kiegészíti a `HttpContent`-et egy ezt elősegítő `ReadAsAsync` metódussal. Ha nem várt státuszkódot kapunk, jelezzük ezt a hívó félnek egy saját kivételtípus dobásával.

3.2 Nézet réteg

A főablakban szeretnénk megjeleníteni egy, a listákat tartalmazó `ListBox` elemet, illetve egy `DataGrid`ben jelenítsük meg az aktuálisan kiválasztott listához tartozó elemeket, azok részleteivel együtt.

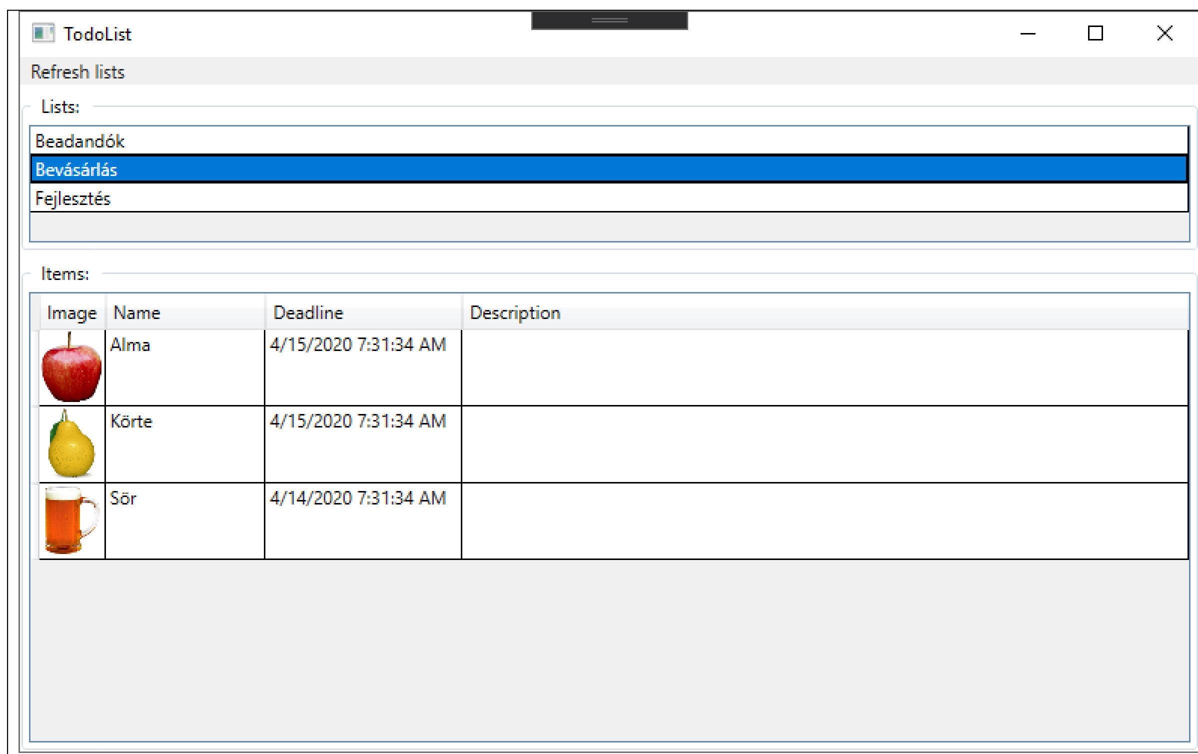

Figure 4: A főablak egy lehetséges kinézete

> *Megjegyzés:* Amennyiben szeretnénk használni egy esemény és parancs összeköttetését elősegítő `Interaction.Triggers`-t, adjuk hozzá a projekthez a `Microsoft.Xaml.Behaviors.Wpf` csomagot.

3.3 Nézetmodell réteg

Használjuk a korábbi félévekben megismert `DelegateCommand`, `ViewModelBase` és `MessageEventArgs` osztályokat. Hozzunk létre a nézetünkhöz egy nézetmodell osztályt. A korábban megírt szerviz osztály segítségével már az ablak megjelenítésekor kérjük le a listákat, melyekkel töltsük fel a nézethez kötött ehhez tartozó `ObservableCollection`-t. Esetleg tegyük lehetővé a listák frissítését egy fájl menüben. Kezeljük az esetlegesen kapott kivételeket egy `MessageBox` megjelenítésével, mellyel informáljuk a felhasználót az aktuális hibáról.

3.4 Alkalmazás réteg

A konfigurációs fájlból kérjük le a webszolgáltatás elérési útvonalát: `ConfigurationManager.AppSettings["baseAddress"]`, majd ezzel konstruáljuk meg a szerviz osztályunkat, azzal pedig a főablakhoz tartozó nézetmodellt.

Állítsuk be mindkét, vagy akár mindhárom projektet induló projektnek a solution tulajdonságainál, majd teszteljük működésüket.