

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](#) fejeztnél 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 kód ismétlést. Az új projekt *Class Library (.NET Standard)* legyen, így .NET Core és .NET Framework, vagy akár Mono Frameworkre épülő alkalmazásokban is felhasználható lesz.

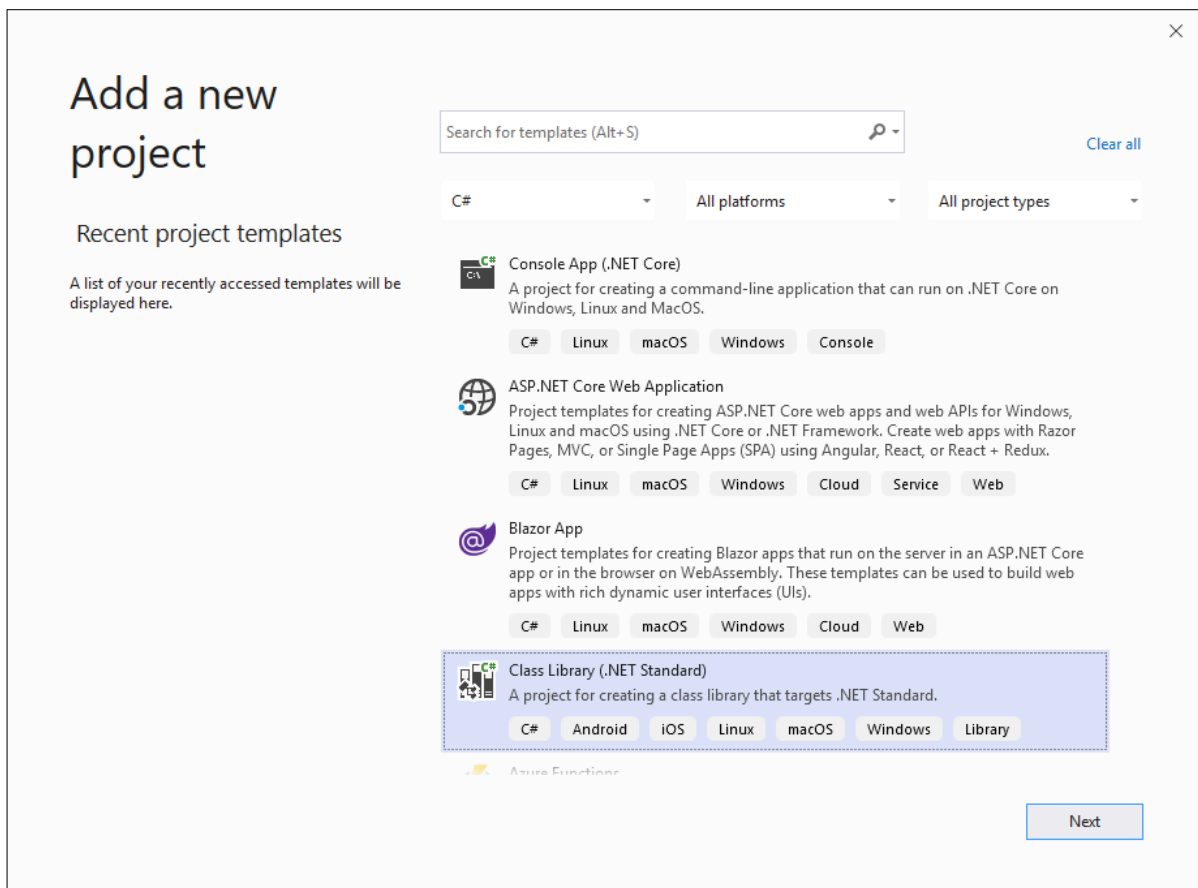


Figure 1: Projekt létrehozása a perzisztencia rétegnek

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*

**Előfordulhat, hogy néhány csomag verzió .NET Standard 2.1-et igényel. A projekt tulajdonságaiban tudjuk átállítani a Target Framework-öt 2.0-ról 2.1-re.*

Adjuk hozzá a weboldal projekthez függőségként az új projektünket a *Dependencies* → *Add Project Reference* 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 névtér elérést, ellenőrizzük, hogy ugyanúgy működik-e a weboldalunk mint eddig.

2 Webszolgáltatás - WebApi

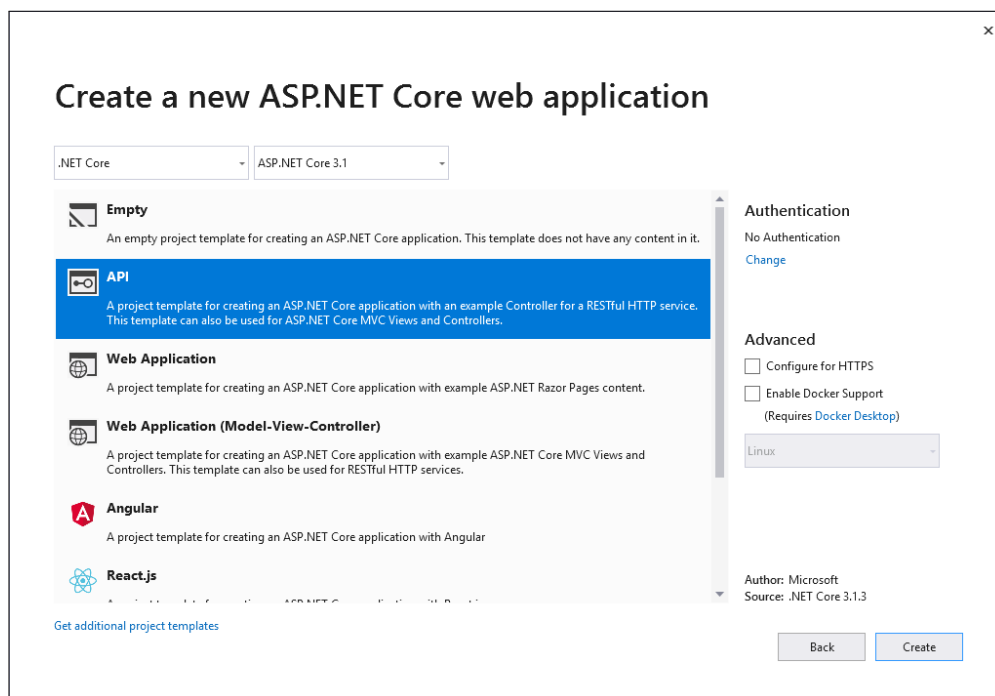


Figure 2: Projekt létrehozása a webszolgáltatásnak

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

Mint ahogy a weboldal esetében is, az egyszerűség kedvéért kapcsoljuk ki a HTTPS-t

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 `Startup.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 elem lekérdező végpontnál akkor egy ennek megfelelő *404 Not Found* státusz kóddal térjünk vissza pl. `return NotFound()`

2.2 Manuális tesztelés

Mielőtt megírnánk a klienst, teszteljük le a Webszolgáltatást egy erre alkalmas eszközzel (*Postman, httprepl*). 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 alap beállítás 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 őket használjuk a szerver és a kliens közti kommunikációra az entitásmodellek helyett.

Megjegyzés: A DTO-kban praktikus konverziós operátorokat implementálni a típusok közti megfeleltetésekre

3 Asztali alkalmazás - WPF

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

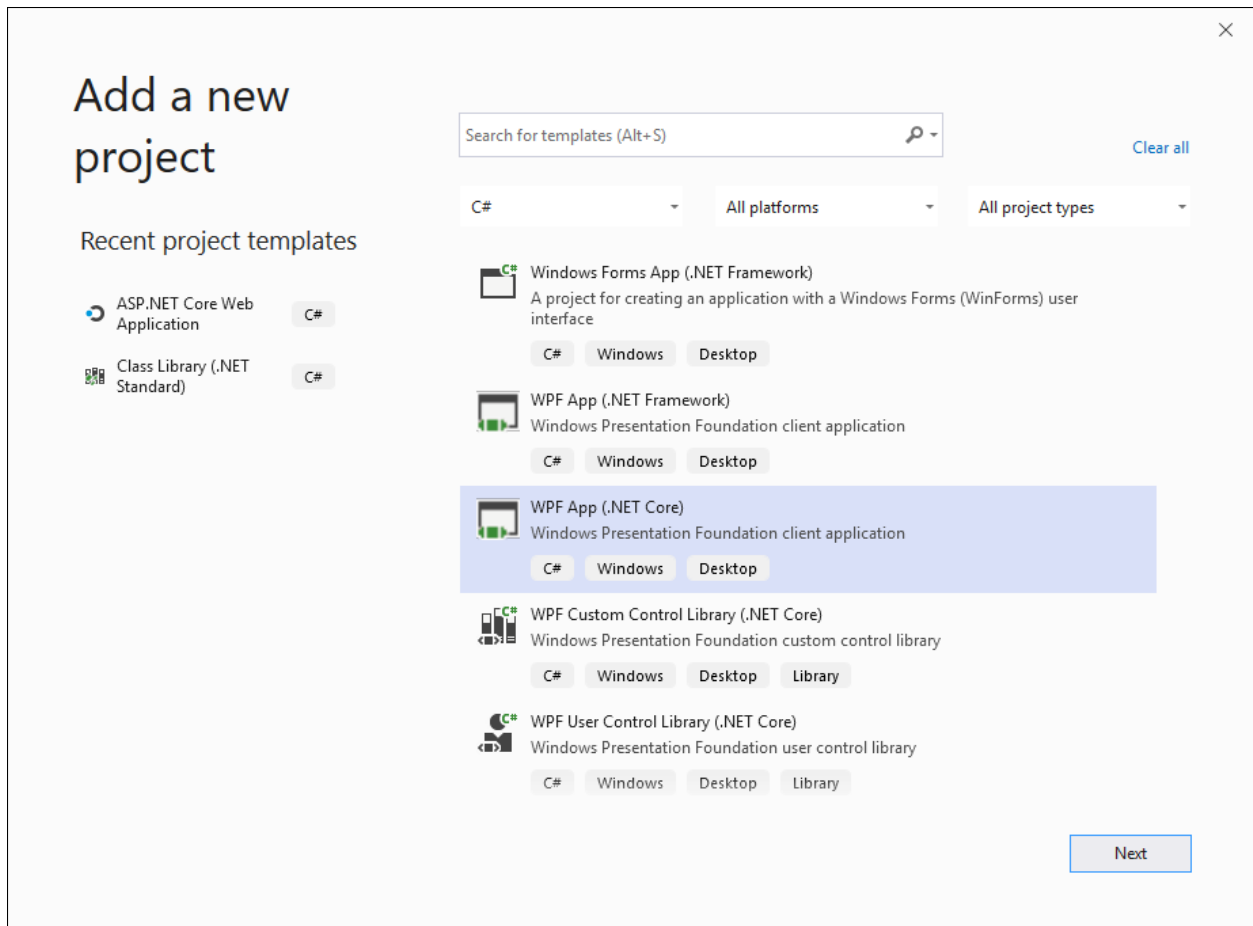


Figure 3: Projekt létrehozása a kliensnek

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 létre hozhatnánk 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*

3.1 Modell réteg

Hozzunk létre egy szervíz 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 kettő végpontot lekérdező metódusokat. A `HttpClient` `GetAsync(string endpoint)` metódusával tudunk *GET* kéréseket intézni. A válasz `HttpResponseMessage` tartalmazza a kapott státuszkódot és a tartalmat JSON stringként amit még deserializálnunk kell. Ehhez adjuk hozzá a projekthez a `Microsoft.AspNet.WebApi.Client` csomagot, ami kiegészíti a `HttpContent`-et egy ezt elősegítő `ReadAsStringAsync` metódussal. Ha nem várt státuszkódot kapunk, jelezzük ezt a hívó félnek egy saját kivétel típus dobásával.

3.2 Nézet réteg

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

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

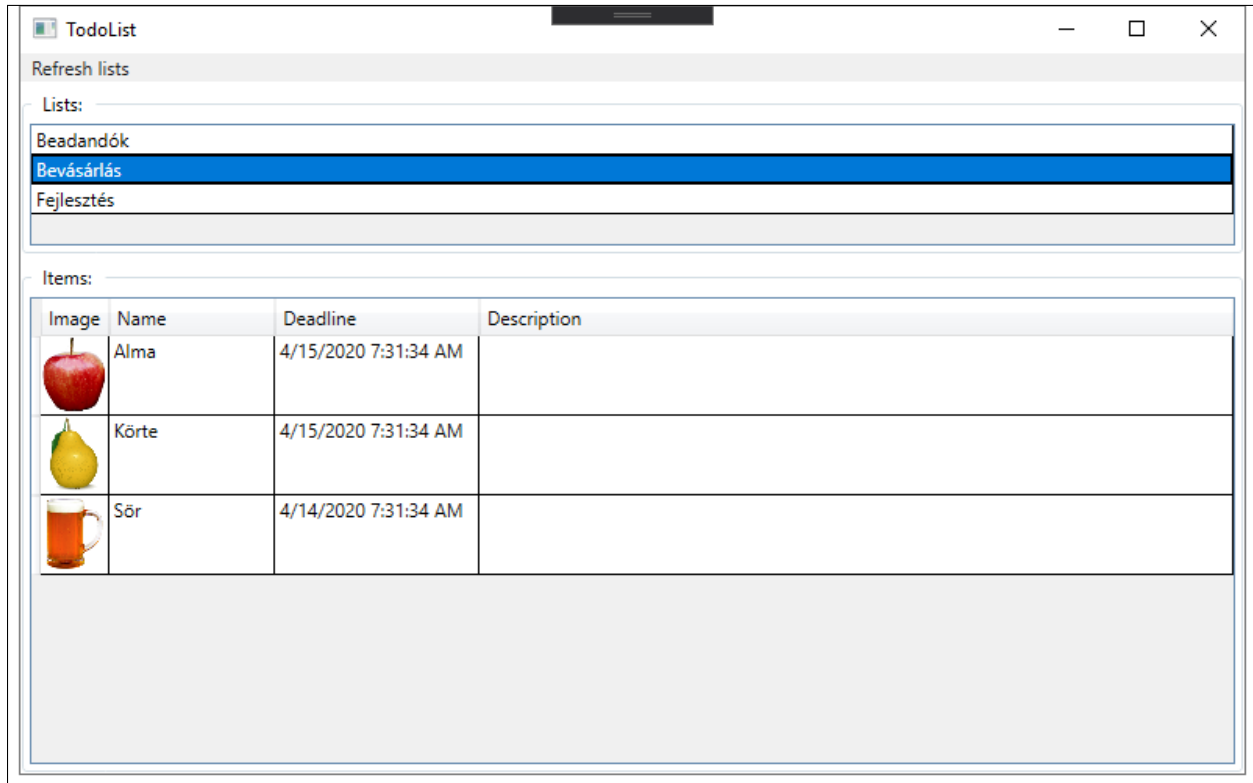


Figure 4: A főablak egy lehetséges kinézete

3.3 Nézetmodell réteg

Használjuk a korábbi félévekben megismert *DelegateCommand*, *ViewModelBase* és *MessageEventArgs* osztályokat. Hozzunk létre a nézetünkhöz egy nézetmodell osztályt. A korábban megírt szervíz osztály segítségével már az ablak megjelenítésekor kérjük le a listákat, melyekkel töltjük fel a nézethez kötött ehhez tartozó *ObservableCollection*-t. Esetleg tegyük lehetővé a listák frissítését egy fájl menüben. Kezeljük az esetlegesen kapott kivételeket egy *MessageBox*-al 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 szervíz osztályunkat, azzal pedig a fő ablakhoz tartozó nézetmodellt.

Állítsuk be mindkét vagy akár mindhárom projektet induló projektnek a solution tulajdonságainal és utána teszteljük le működésüket.