ASP.NET Core WebAPI + WPF: adatbevitel, tesztelés

A hatodik gyakorlat célja, hogy a webszolgáltatást és az ahhoz készített klienst kiegészítsük, mint a weboldal esetében, adat- beviteli és módosító lehetőségekkel. Ehhez az autentikációt is implementálnunk kell, hiszen nem szeretnénk, hogy bejelentkezés nélkül módosítani lehessen. Az egyszerűség kedvéért, a kliensben, az eddig megvalósításra került megjelenítési funkciók is egy bejelentkező ablak után legyenek csak elérhetőek.

1 Bejelentkezés

1.1 Webszolgáltatás

Az autentikációhoz hozzunk létre egy új vezérlőt az üres `API Controller` sablonból, melyben mint a weboldalnál, implementáljunk egy bejelentkezés es kijelentkezés akciót. A bejelentkezéskor átküldendő információknak hozzunk létre egy DTO-t a perzisztencia rétegben, mely tartalmazza a felhasználónevet és jelszót, kötelezőként annotálva. Mivel a DTO paramétert nem egy GET kérés *query string*-jében szeretnénk átadni, hanem egy POST kérés *body*-jában ezért ezt jelezvén annotáljuk a paramétert `[FromBody]` attribútummal. Amennyiben nem sikerült a bejelentkezés térjünk vissza *Unauthorized* státuszkóddal. A `Startup`-ban konfiguráljuk az identity keretrendszert és adjuk hozzá az autentikációs *middleware*-t az alkalmazásunkhoz a már ismertetett módon.

Megjegyzés: Vegyük észre, hogy a sablon annotálta a kontrollert az `[ApiController]` attribútummal, ennek hatására az akciók automatikusan `BadRequest`-el térnek vissza ha a kapott modell nem megy át a validációs szabályokon. Így ezt felesleges manuálisan ellenőriznünk.

1.2 Kliens

Hozzunk létre egy új bejelentkező ablakot és egy hozzátartozó nézetmodellt, melyben helyezzük el a bejelentkezéshez szükséges elemeket. A kezdetben megjelenő ablakunk is ez legyen. Implementáljunk egy a bejelentkező végponttal kommunikáló metódust a szerviz osztályban. A felépített DTO elküldéséhez használjuk a `PostAsJsonAsync` metódust, mely ezt elhelyezi a POST kérés törzsében. Érdemes elhelyezni a nézetmodellben egy *flag*-et ami azt jelzi, hogy a kérés folyamatban van, erre rákötve a login parancs *CanExecute* részét, elkerülhetjük, hogy a felhasználó a kérés végrehajtása közben többször is rákattintson a gombra így esetleg többször is elküldje a kérést. Kezeljük le a sikertelen bejelentkezéskor kapott *Unauthorized* státuszkódot is. Legyen lehetőség kijelentkezni is. Ehhez helyezzünk el egy menüpontot a főablak menüjében, majd valósítsuk meg a kijelentkező végpontot meghívó metódust a szerviz osztályban, a kettőt pedig kössük össze. Az ablakok közti váltást az applikáció rétegben implementáljuk, események segítségével.

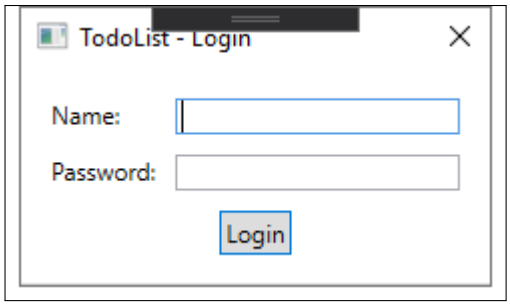

Figure 1: A bejelentkező ablak egy lehetséges kinézete

2 Adatbevitel és módosítás

2.1 Webszolgáltatás

A kontrollerek korábban legenerált CRUD akcióit amelyet kikommenteztünk vagy kitöröltünk, most tegyük vissza(Get, Put, Post, Delete), majd írjuk át, hogy a *GET*-hez hasonlóan a szerviz osztályt és a DTO-kat használják. Mivel azt szeretnénk, hogy a módosító műveletek csak bejelentkezett felhasználók számára legyenek elérhetőek, ezért annotáljuk fel ezeket az akciókat a korábban már ismertetett `[Authorize]` attribútummal. A *POST* akció egy új entitás példány létrehozását teszi lehetővé, ennek megfelelően egy *201 - Created* státuszkóddal tér vissza, ami konvencionálisan tartalmazza az újonnan létrehozott erőforrást, beleértve a kapott azonosítót is. Ennek megvalósítása érdekében módosítsunk a korábban erre implementált szerviz metóduson, hogy ne egy *bool*-al térjen vissza hanem magával az entitással a beillesztés után, illetve *null*-al hiba esetén. Így már könnyedén visszatérhetünk a megfelelő objektummal.

2.2 Kliens

A szerviz osztályban valósítsuk meg az új végpontokkal kommunikáló metódusokat (*Create*, *Update*, *Delete*). A listaelemek módosítását az egyszerűség kedvéért külön ablakban fogjuk lehetővé tenni. Hozzunk létre nézetmodelleket a listáknak és az elemeknek, praktikusan konverziós operátorokkal kiegészítve a DTO-s párjukhoz. Ahhoz, hogy a nézet is értesüljön arról, ha kódból módosítunk egy tulajdonságon, akkor a megszokott módon származtassunk a `ViewModelBase`-ből és hívjuk meg a megörökölt `OnPropertyChanged` metódust minden tulajdonság *setter*-ében_. A nézetmodellben térjünk át a lista és elemek nézetmodelljeire a DTO-k helyett.

2.2.1 Listák

A listák szerkesztéséhez hozzunk létre egy *Add*, *Edit* és *Delete* gombsort az elemek alatt, illetve egy szövegdobozt a lista nevének szerkesztéséhez. Mivel sokszor fogjuk használni, hozzunk létre a kijelölt elemnek egy tulajdonságot a nézetmodellben majd kössük rá a `DataGrid SelectedItem` tulajdonságát. A *Delete* gomb hatására az aktuálisan kiválasztott elemet töröljük a szerviz osztály segítségével, illetve magából a listából is. Az *Add* gombbal lehessen új listát létrehozni kezdeti értékekkel, melyet adjunk hozzá a `ObservableCollection`-hoz is az *Api* hívás után, így szinkronba hozva a lokális állapotot. Az aktuálisan kiválasztott lista nevéhez hozzunk létre egy új tulajdonságot a nézetmodellben, melyet frissitsünk a kiválasztás változása esetén, illetve az *Edit* gomn hatására erre frissüljön a lista neve.Ügyeljünk arra, hogy csak akkor legyenek elérhetőek ezek a gombok, ha van kiválasztva lista.

2.2.2 Listaelemek

A listaelemek szerkesztéséhez is hozzunk létre egy gombsort. Az elemek nézetrácsában is kössünk egy nézetmodellbeli tulajdonságot az aktuálisan kiválasztott elemre. Az *Add* és *Delete* gombbal járjunk el úgy mint a listák esetében, azonban az *Edit* hatására jelenítsünk meg egy új ablakot, melyben az aktuálisan kiválasztott elem egy másolatát tudjuk módosítani, így ha elvetnénk a módosításokat, akkor ne vesszen

el az eredeti állapota lokálisan sem. Az új ablaknak az egyszerűség kedvéért ne legyen külön nézetmodellje, hanem használja a főablakét. Helyezzünk el benne megfelelő vezérlőket egy elem szerkesztésére, illetve a mentéshez és elvetéshez szükséges gombokat. A határidő kiválasztására, esetleg használhatjuk a `DotNetProjects.Extended.Wpf.Toolkit` csomagból elérhető `DateTimePickert`. A kép megváltoztatását egy gomb hatására megjelenő fájl kiválasztó dialógusra bízzuk, melyet az alkalmazás rétegben eszközöljünk. A legördülő menüben legyen automatikusan kiválasztva az aktuális lista amihez tartozik a kijelölt listaelem.

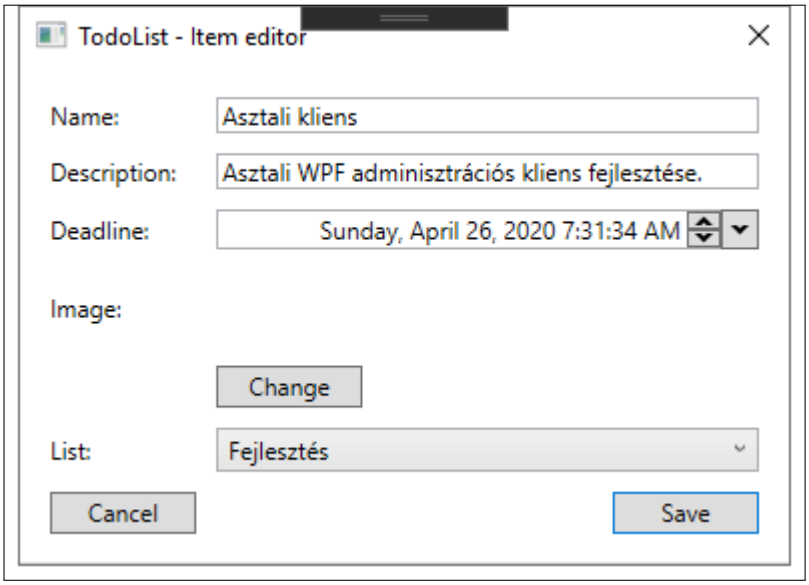

Figure 2: Az elem módosító ablak egy lehetséges kinézete

3 Tesztelés

A teszteléshez többfajta keretrendszer is használható itt az *xUnit* kerül bemutatásra, de a többi is nagyon hasonló. Hozzunk létre egy új `xUnit Test Project`-et a webszolgáltatásunk teszteléséhez. Adjuk hozzá függőségnek a perzisztencia és webszolgáltatás projektet.

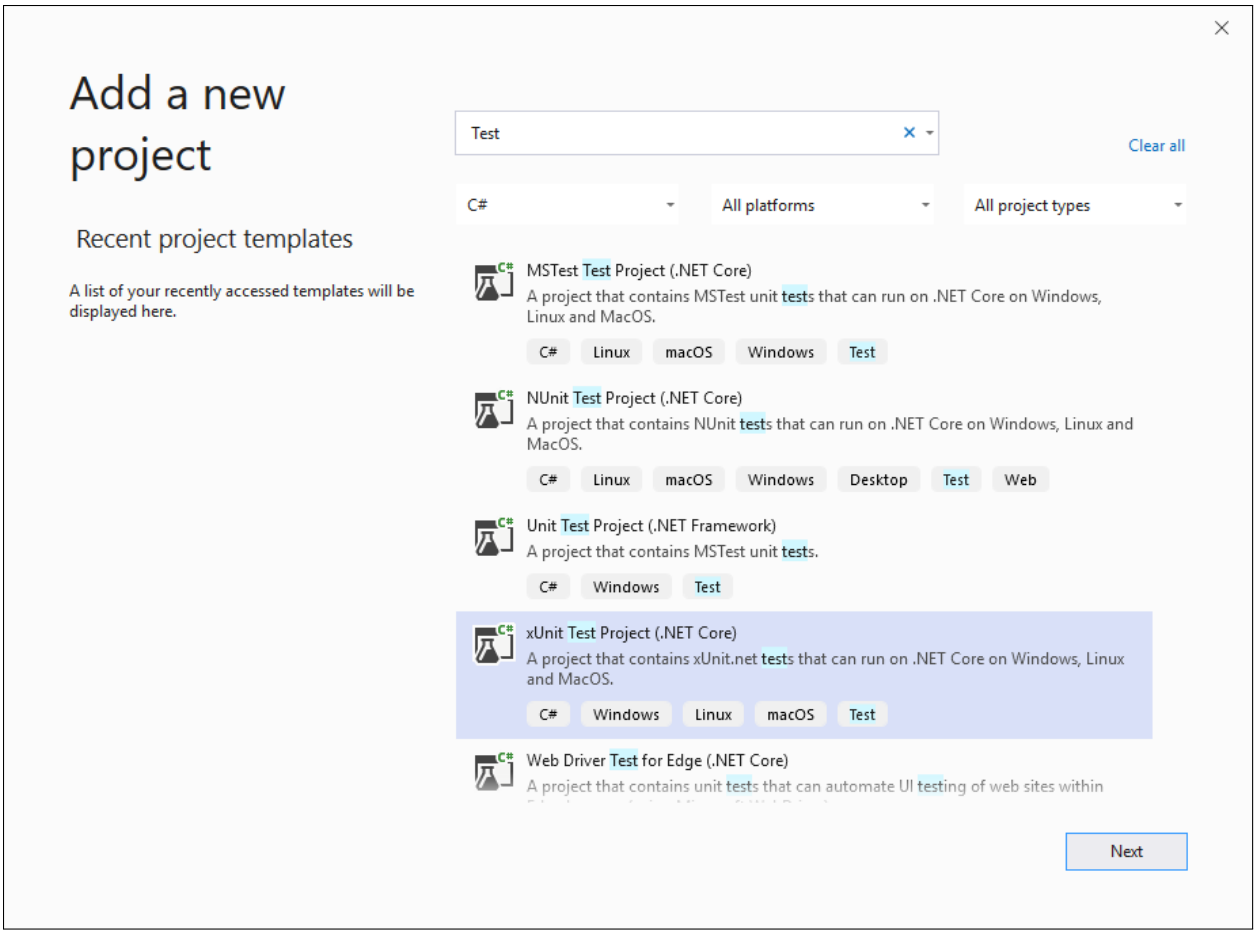

Figure 3: Teszt projekt létrehozása

A listák vezérlőjét teszteljük úgy, hogy az adatbázist a memóriában hozzuk létre, majd ezzel példányosítjuk magát a kontrollert és meghívjuk közvetlenül a tesztelendő akciót. Ez némiképp több, mint egy egység teszt, mivel a tényleges egység teszthez minden függőséget helyettesítenünk kéne egy teszt megvalósítással, de azért még nem teljes integrációs teszt sem. Ahhoz hogy tudjunk a memóriában létrehozni adatbázist adjuk hozzá a projekthez az ehhez szükséges `Microsoft.EntityFrameworkCore.InMemory` csomagot. Az *xUnit* keretrendszerben a tesztelő osztályban minden `[Fact]` attribútummal ellátott metódus egy tesztesetet jelöl. Az osztály konstruktora meghívódik minden teszteset indulása elött, illetve ha a tesztosztály megvalósítja az `IDisposable` interfészt akkor a `Dispose` metódus is meghívódik minden teszteset lefutása után. Mivel szeretnénk, hogy a teszteseteink egymástól függetlenek legyenek ezekben a metódusokban hozzuk létre, illetve töröljük az adatbázisunkat. A teszt adatbázist feltöltése során ügyeljünk rá, hogy explicit adjunk az elemeknek azonosítót, így könnyebb lesz a tesztelés. Lehetőségünk van parametrizált tesztesetre is a `[[Theory](Theory)]` attribútum segítségével, ebben az esetben a konkrét paramétereket is meg kell adnunk pl. `[InlineData(...)]` attribútum segítségével. Vegyük észre, hogy ezzel a tesztelési móddal ki van kerülve az `[Authorize]` attribútum, igy bejelentkezés nélkül is megtudjuk hívni az adott akciót ez viszont azt is jelenti, hogy ha az akció használja az `HttpContext.User`-t akkor ezt külön fel kell töltenünk az alábbi módon, természetesen a *testName* és *testId* értékeket egy a teszt adatbázisunkban létező felhasználó alapján helyettesítsük be:

```
var claimsIdentity = new ClaimsIdentity(new List<Claim>
{
  new Claim(ClaimTypes.Name, "testName"),
```

```
  new Claim(ClaimTypes.NameIdentifier, "testId"),
});
var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
_controller.ControllerContext = new ControllerContext
{
  HttpContext = new DefaultHttpContext
  {
    User = claimsPrincipal
  }
};
```

A memóriabeli adatbázishoz a felhasználók seed-eléséhez továbbra is a `UserManager` osztályt használjuk, azonban itt nincs mitől lekérnünk mint a `Startup` osztályban ezért kénytelenek vagyunk mi példányosítani, pl.:

```
var userManager = new UserManager<ApplicationUser>(
  new UserStore<ApplicationUser>(_context), null,
  new PasswordHasher<ApplicationUser>(), null, null, null, null, null, null);
```

Az akciók által visszaadott `IActionResult Result` tulajdonságában ellenőrizhetjük a visszatért nem *Ok* státuszkód a típus alapján, illetve a `Value` tulajdonságában az objektumot ha van olyan. Ha az akciónak *404*-el kell visszatérnie akkor így ellenőrihetjük ezt: `Assert.IsAssignableFrom<NotFoundResult>(result.Result);` Ha pedig egy `ListDto`-val akkor: `Assert.IsAssignableFrom<ListDto>(result.Value)`

A Teszteseteket a *View* → *Test Explorer* segítségével futtassuk le.

Lehetőségünk van integrációs tesztelésre is a `TestServer` segítségével, ehhez bővebb információt a vonatkozó előadás és a kapcsolódó Microsoft dokumentáció adhat.