WAF - 6. gyakorlat

A hatodik gyakorlat célja, hogy a webszolgáltatást és az ahhoz készített klienst kiegészítsük, mint a weboldal esetében, adat- beviteli es 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.

Bejelentkezés

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, kötelezőnek 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.

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.

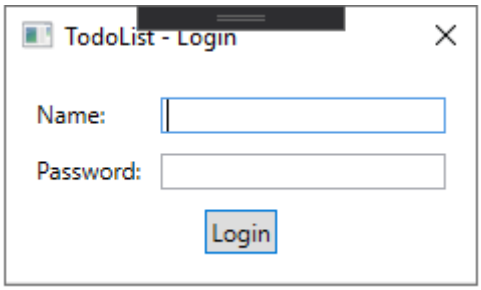

Figure 1: A bejelentkező ablak egy lehetséges kinézete

Adatbevitel és módosítás

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.

Kliens

A szerviz osztályban valósítsuk meg az új végpontokkal kommunikáló metódusokat (*Create*, *Update*, *Delete*). Az adatok módosítására kétféle megoldást nézünk, a listáknál egyből a `DataGrid`-ben, az elemeknél egy külön ablakban legyen erre lehetőség. Ehhez először is hozzunk létre nézetmodelleket ezeknek az osztályoknak*, praktikusan konverziós operátorokkal kiegészítve a DTO-s párjukhoz. Ahhoz, hogy egy módosítandó példány vissza tudjon térni a módosítás elvetése esetén az eredeti állapotba újbóli lekérdezés nélkül, implementáljuk a nézetmodellelen az `IEditableObject` interfészt. A módosítás kezdetén, tároljunk el egy másolatot a példányból, majd elvetés esetén állítsuk azt vissza, illetve hozzunk létre egy logikai tulajdonságot, mellyel le tudjuk kérni, hogy az objektum módosítás alatt van-e.

Listák Állítsuk át a listákat megjelenítendő `DataGrid` megfelelő tulajdonságait úgy, hogy most már ne csak olvasható legyen; lehessen sorokat hozzáadni és törölni, illetve egyszerre csak egy teljes sort lehessen kijelölni. Mivel sokszor

fogjuk használni, hozzunk létre a kijelölt elemnek egy tulajdonságot a nézetmodellben majd kössük rá a `DataGrid SelectedItem` tulajdonságát. Alapbeállítás szerint a `DataGrid` legutolsó – üres – sorába kattintva tudunk új elemet hozzáadni, a kijelölt sort a *delete* gombbal tudjuk törölni, meglévő cella módosítását az *escape* billentyűvel tudjuk elvetni. Az új sort reprezentálandó üres sor egy *placeholder* elemként fog mejelenni a `SelectedItem`-ben ami nem tud a lista nézetmodelljévé konvertálódni, ezért érdemes létrehozni erre egy konvertert ami ilyen esetben *null*-al tér vissza és ezt használni a kötéskor. A `DataGrid` automatikusan meghívja az `IEditableObject` metódusait, így érdemes egy eseményt kiváltanunk egy lista módosításának véglegesítésekor, melyre fel tud majd iratkozni a nézetmodell. A nézetmodellben térjünk át a lista és elemek nézetmodelljeire a DTO-k helyett. A listák feltöltésekor iratkozzunk fel miden lista nézetmodell példány korábban hozáadott eseményére, illetve magára az `ObservableCollection CollectionChanged` eseményére is, mely akkor váltódik ki, – többek között – amikor a listához hozzáadunk vagy törlünk. Kössünk rá a `DataGrid AddingNewItem` eseményére is, mellyel beállíthatjuk az új lista kezdeti értékeit, illetve feliratkozhatunk a módosítás véglegesítésekor kiváltott eseményre is; itt érdemes beállítani egy extremális értéket az azonosítónak, így később megtudjuk majd különböztetni, a módosítandó elemet az létrehozandótól. Ezt használjuk is ki rögtön a módosítás vége eseményben és hozzuk létre vagy módosítsuk az elemet a korábban már megírt szerviz műveletekkel. Így a `CollectionChanged` eseménynél már csak azt az esetet kell lekezelnünk, amikor törlésre kerül egy elem. Itt ha még létre se hoztuk az elemet, akkor persze ne küldjük el ezt a webszolgáltatásnak. Valamint a kiválasztott lista elemeinek lekérdezésekor is vegyük ezt figyelembe.

**Megjegyzés: 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*

Listaelemek A listaelemek szerkesztéséhez hozzunk létre egy *Add*, *Edit* és *Delete* gombsort az elemek alatt. Az elemek nézetrácsában is kössünk egy nézetmodellbeli tulajdonságot az aktuálisan kiválasztott elemre. A *Delete* gomb hatására az aktuálisan kiválasztott elemet töröljük a szerviz osztály segítségével, illetve magából a listából is. Az aktuális elem szerkesztésére hozzunk létre egy új ablakot, melyet az *Add* és *Edit* gombokra nyomva események segítségével jelenítsünk meg. Az *Add* esetében előbb adjunk hozzá egy új elemet – megfelelő kezdeti értékekkel – is az elemek listájához és legyen ez az új elem az aktuálisan kijelölt. Ügyeljünk arra, hogy csak akkor legyenek elérhetőek ezek a gombok, ha ennek van értelme, azaz a kijelölt lista és elem tulajdonságok függvényében. 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 `MiqM.Xceed.Wpf.Toolkit.NETCore` csomagból elérhető `DateTimePickert`. A kép megváltoztatását egy gomb hatására megjelenő fájl

kiválasztó dialógusra bízzuk, melyet az applikáció rétegben eszközöljünk. A legördülő menüben legyen automatikusan kiválasztva az aktuális lista amihez tartozik a kijelölt listaelem, illetve hozzunk létre neki egy külön listát amiből lehet választani, mert ha közvetlenül a listák `ObservableCollection`-jére kötjük rá ott a szerkeszthetőség miatt az új üres sor is megjelenne. A módosítás elvetéséhez használjuk a korábban megírt `IEditableObject` interfészt implementáló metódusait a listaelemnek. Illetve a mentéskor a szerviz osztály segítségével hozzuk létre vagy módosítsuk az elemet.

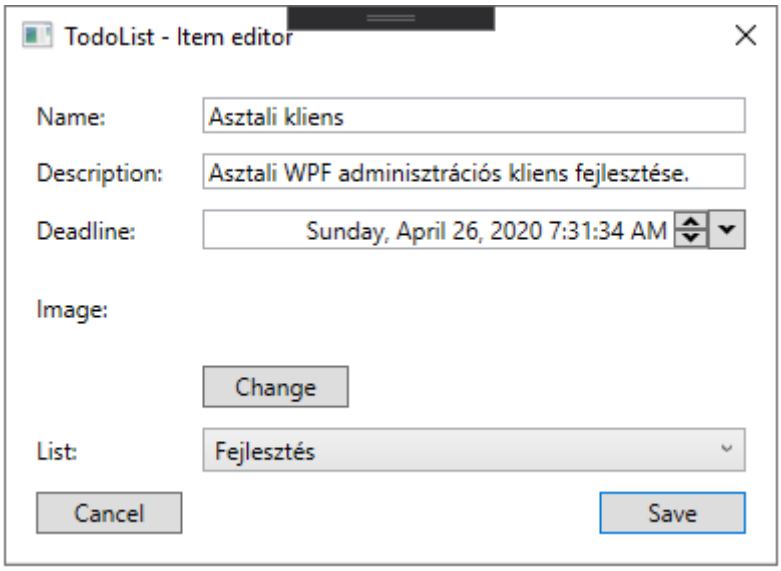

Figure 2: Az elem módosító ablak egy lehetséges kinézete

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.

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

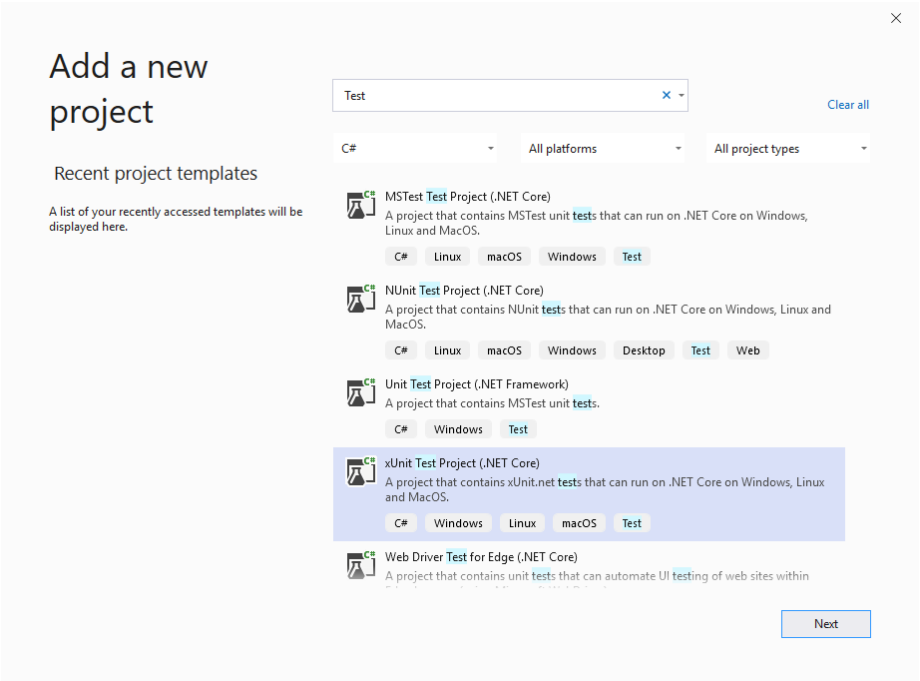

Figure 3: Teszt projekt létrehozása

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
  }
};
```

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 itt találunk: https://docs.microsoft.com/en-us/aspnet/core/test/integration-tests