5. LABOR – XAML 2

Segédlet a Szoftverfejlesztés laboratórium 2 c. tárgyhoz

Kidolgozta: Tóth Tibor (2018)
Átdolgozta: Szücs Cintia Lia (2021)

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

1117 Budapest, Magyar tudósok körútja 2.
Q. ép. B. szárny 207. • www.aut.bme.hu
Telefon: 463-2870 • Fax: 463-2871

BEVEZETÉS

CÉLKITŰZÉS

Az labor célja, hogy a hallgatók gyakorolják a Kliens oldali technológiák c. tárgyban tanult XAML ismereteket azon belül is az MVVM tervezési minta használatát.

ELŐFELTÉTELEK

A labor elvégzéséhez szükséges eszközök:

- min. Microsoft Visual Studio 2019
  - Universal Windows Platform v16299 SDK
  - (Opcionálisan Hyper-V + Windows 10 Mobile v16299 emulátor. Ez nem minden laborban van feltelepítve)
    - Boot során a **with Hypervisor** opciót kell választani
- Kiinduló projekt, ami az előző labor végi állapotot tartalmazza (TodoPrism - Kiindulo.zip)

AMIT ÉRDEMES ÁTNÉZNED

- Univerzális Windows Platform jellemzői
- Kliens oldali technológiák jegyzet - XAML technológiák, MVVM tervezési minta
- Kliens oldali technológiák labor – 2 XAML labor anyaga, MVVM tervezési minta használata

A LABOR ÉRTÉKELÉSE

A teljes munkát a laborvezető a labor végén értékeli. Az értékeléskor a laborvezető kérdéseket tehet fel az elkészült alkalmazással kapcsolatban, amik befolyásolhatják az érdemjegyet. Az eredményről a hallgató a labor végeztével, de legkésőbb a laborvezető által jelzett időpontban értesül. Az ellenőrzéshez segítséget nyújt a labor végén összeállított ellenőrző lista.

A labor végén a Moodl-be (https://edu.vik.bme.hu/) a Szoftverfejlesztés laboratórium című tárgy megfelelő laboralkalmához feltöltendő az elkészült forráskód ZIP formátumban. A ZIP-ből törlendők a fordítási artifaktok és külső függőségek (bin, obj, packages mappák).

CÉLKITŰZÉS

Ezen a laboron az előző labor során elkészített alkalmazásban vezetjük be az MVVM tervezési mintát, és vesszük végig a leggyakrabban előforduló problémákat.

Töltsük le a kiinduló projektet, és próbáljuk meg futtatni!

MVVM MINTA ÁTTEKINTÉSE

Az MVVM minta célja, hogy a kliens alkalmazások architekturális felépítése, egységes, jól karbantartható mintát kövessen, különválasztva a különböző felelősségi köröket. A View osztályok (esetünkben a XAML vezérlők, oldalak és code-behind) feladata elsődlegesen csak a megjelenítés. A ViewModel osztályok a nézetek absztrakciói, tartalmazzák a nézetek állapotát, és a különböző üzleti műveleteket anélkül, hogy a nézetet bármilyen módon is ismernék. A View és a ViewModel között a csatolás laza az adatkötések, konverterek és kommandok használata miatt. A modell esetünkben

az üzleti objektumoknak felel meg, de gyakran ide soroljuk még a különböző üzleti logikákat újrafelhasználható módon megvalósító szolgáltatás osztályokat is.

A minta nagy előnye a nézet és az üzleti logika különválasztása, és a tesztelhetőség növekedése. Gyakran használják párban a Dependency Injection és az Inversion of Control tervezési mintákkal.

MVVM KÖNYVTÁR INTEGRÁLÁSA

Az MVVM mintát nagyon ritkán implementáljuk le kézi módszerekkel, helyette inkább valamilyen előre elkészített osztálykönyvtárat használunk, amelyek gyakran segítenek általános alkalmazáséletciklussal kapcsolatos feladatokat is egyszerűen megoldani. Ilyen könyvtár UWP platformra például a Template10, Prism, MVVMLight, Caliburn.Micro stb. Mi most a Prism-et fogjuk használni alkalmazásunkban.

ALKALMAZÁS ŐSOSZTÁLY

Ahogy említettük, az MVVM könyvtárak gyakran nem csak ennek a mintának a támogatását tűzik ki célul, hanem például az alkalmazás életciklusának egyszerűbb kezelését is, így tesz a Prism is. Ahhoz, hogy ezeket a funkciókat elérjük az `App` osztályunknak az `Application` osztály helyett egy speciális osztályból kell leszármazni (`PrismUnityApplication`), ami mellesleg közvetetten az `Application` osztályból származik.

Az `App.xaml`-ben tekintsük meg az `App` osztály ősét (gyökér XML elem). A Prism alkalmazásával ez a `PrismUnityApplication` osztály:

```
<prismUnity:PrismUnityApplication
    x:Class="TodoPrism.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:prismUnity="using:Prism.Unity.Windows"
    RequestedTheme="Light">
```

Ez az `App.xaml.cs`-ben is megjelenik:

```
sealed partial class App : PrismUnityApplication
```

Próbáljuk ki! Szinte ugyan úgy kell működnie az alkalmazásnak, mint eddig.

Az egyetlen különbség, hogy a részletes oldalon nem jelenik meg a korábban már látott beépített visszagomb. Ennek beállítását az `App.xaml.cs`-ben tudjuk megtenni a `DeviceGestureService` segítségével. Ehhez az `OnCreateDeviceGestureService` metódus felüldefiniálására van szükség, itt tudjuk elérni a `GestureService` példányt. A feladatunk csupán annyi, hogy ennek a `UseTitleBarBackButton` tulajdonságát átállítsuk `true`-ra. Ennek következtében a későbbiekben, mikor a navigáláshoz a Prism-es navigationService adta lehetőségeket fogjuk használni, akkor a részletes nézet oldalán meg fog jelenni a beépített visszagomb.

```
protected override IDeviceGestureService OnCreateDeviceGestureService()
        {
            var service = base.OnCreateDeviceGestureService();
            service.UseTitleBarBackButton = true;
            return service;
        }
```

MVVM MINTA A FŐOLDALON

Az MVVM minta szerint a nézet állapota és az ehhez kapcsolódó logika a ViewModel osztályokba kerül elhelyezésre, ami lazán (adatkötéssel) tart kapcsolatot a nézettel.

Készítsük el a `MainPage`-hez tartozó ViewModel osztályt. Vegyünk fel egy új mappát a projektbe `ViewModels` néven, majd ebbe egy új osztályt `MainViewModel` néven, amely származzon le a `ViewModelBase` osztályból.

Ide helyezzük át a memóriában lévő `TodoItem` listánkat a `MainPage.xaml.cs`-ből a `MainPageViewModel`-be, és vegyük le róla a `static` módosítót.

```
public class MainViewModel : ViewModelBase
{
    public ObservableCollection<TodoItem> Todos { get; set; } = new ObservableCollection<TodoItem>()
    {
        // Tartalom
    }
}
```

Az adatkötések létrehozása során a tipikus felállás az szokott lenni, hogy az oldal (vagy vezérlő) `DataContext` tulajdonságában a hozzá tartozó `ViewModel` objektum található, és ennek a tulajdonságaihoz kötünk a felületen. Szerencsénkre a Prism ezt az összepárosítást automatikusan elvégzi az elnevezési konvenciók mentén. Miszerint a `MainPage` osztályunk `DataContext` tulajdonságát a `MainViewModel` példányra állítja. Ehhez töröljük a `MainPage.xaml`.cs konstruktorából a `DataContext` beállítására vonatkozó sort.

Az adatkötéseink így rendben vannak, mert a `Binding` a `DataContext Todos` tulajdonságához szeretne kötni, ami esetünkben a `MainViewModel Todos` tulajdonsága lesz.

Próbáljuk ki! (Ehhez még ki kell kommentezni a régi tulajdonságra történő hivatkozást a `TodoDetailsPage.xaml.cs`-ben lévő `Save_Click` metódusból.)

```
private void Save_Click(object sender, RoutedEventArgs e)
{
    //MainPage.Todos.Add(Todo);

    Frame.GoBack();
}
```

WEBSZOLGÁLTATÁS

A memóriában lévő lista helyett használjunk valamilyen életszerűbb példát és használjunk egy REST-es webszolgáltatást. Ezt a https://bmetodoservice.azurewebsites.net oldalon érhetjük el. A `/swagger` oldalon találhatunk egy interaktív dokumentációt a különböző szolgáltatás kérésekről. Próbáljuk ki a GET-es kéréseket, nézzük meg milyen adattal tér vissza, pont olyannal, mint az alkalmazásunkban lévő modell.

TODOSERVICE

Készítsünk egy szolgáltatás osztályt, ami egy webes kérést intéz egy `HttpClient` objektumon keresztül a szolgáltatás felé, és lekéri a teendők listáját.

Hozzunk létre egy `Services` mappát a projektbe, majd ebbe egy `TodoService` osztályt. Ide a már a kliens oldali tárgy laborjából ismerős minimalista webes kérést implementáljuk le.

```
public class TodoService
{
    private async Task<T> GetAsync<T>(Uri uri)
    {
        using (var client = new HttpClient())
        {
            var response = await client.GetAsync(uri);
            var json = await response.Content.ReadAsStringAsync();
            T result = JsonConvert.DeserializeObject<T>(json);
            return result;
        }
    }
}
```

Aszinkronitás

Ismételjük át mit is jelent az aszinkronitás:

Alapvetően a metódusok szinkron hívódnak meg, tehát a hívó szál addig várakozik, amíg a metódus vissza nem tér. UI szálról indított szinkron metódusok megakasztják a felhasználói felületet, addig nem tud más függvény futni, így más eseménykezelők sem, vagyis az app nem képes reagálni a felhasználói inputokra. A hosszan tartó műveleteket futtassunk háttér szálon, és aszinkron várakozzunk rájuk.

Hogy ne kelljen bonyolult szálkezeléssel a program írása során törődnünk, a C# nyelv támogatást ad az aszinkronitás nyelvi szintű kezelésére az `async`, `await` kulcsszavakon és a `Task` osztályon keresztül. A `Task` osztály fogja össze a háttérben végzett műveletet (mit kell futtatni, aktuális állapot, hiba stb.). Ezt a `Task`-ot az `await` kulcsszóval lehet aszinkron módon bevárni, ilyenkor a program futása nem fog blokkolva várakozni a hívó szálon. Az `await` művelet végeztével a futás visszatér a hívóhoz, és az eredeti szálon folytatódnak tovább az azt követő műveletek.

Esetünkben a `HttpClient` osztály műveletei aszinkron módon vannak megvalósítva: `Task`-kal térnem vissza, amit be tudunk várni az `await` kulcsszóval. Viszont `await`-et csak `async` módosítóval ellátott függvényekben lehet használni, amitől ez a metódus is aszinkron lesz, így, ha be szeretnénk várni ennek a lefutását kívülről, akkor itt is `Task` objektummal kell visszatérjünk, ami esetünkben tartalmazza majd az eredménylistát is. Vegyük észre, hogy az `async` függvényekben a `return` utasítás igazából a `Task` eredményét kell, hogy visszaadja, és nem magát a `Task` objektumot.

Teendők listájának lekérése a TodoService-ben

Használjuk a segédfüggvényünket egy új metódusban, ami lekéri a teendők listáját.

```
private readonly Uri _serverUrl = new Uri("https://bmetodoservice.azurewebsites.net");

public async Task<List<TodoItem>> GetTodosAsync()
{
    return await GetAsync<List<TodoItem>>(new Uri(_serverUrl, "api/Todo"));
}
```

Teendől listájának lekérése a MainViewModel-ben

Kérjük le a `MainViewModel`-ben ezt a listát a szolgáltatásunkon keresztül. Mivel a konstruktorban nem lehet aszinkron műveleteket végrehajtani az `await` kulcsszó segítségével, más helyet kell keressünk neki. Az eddigi tapasztalataink során az oldal `OnNavigatedTo` metódusa jó választásnak tűnik, viszont az csak a Page objektumnak

van. Szeretnénk, ha a `ViewModel`-ek is le tudnák kezelni az oldal életciklus eseményeit. Szerencsére a Prism gondolt erre és kínál nekünk egy `ViewModelBase` ősosztályt, amiben át vannak vezetve az oldal életciklus eseményei.

Definiáljuk felül az `OnNavigastedTo` metódusát, amiben pedig kérjük le a teendők listáját a `TodoService` osztályon keresztül aszinkron módon. Ezzel párhuzamosan a korábbi beégetett értékeket tartalmazó listánkat törölhetjük.

```
public class MainViewModel : ViewModelBase
{
    public ObservableCollection<TodoItem> Todos { get; set; }

    public override async void OnNavigatedTo(NavigatedToEventArgs e,
       Dictionary<string, object> viewModelState)
    {

        var todos = await new TodoService().GetTodosAsync();
        Todos = new ObservableCollection<TodoItem>(todos);

        base.OnNavigatedTo(e, viewModelState);
    }}
```

Próbáljuk ki az adatok lekérését! **Mit tapasztalunk és mi lehet ennek az oka?**

Próbáljuk ki úgy, hogy breakpointot teszünk a `TodoService` hívás után és ellenőrizzük, hogy megérkezett-e a kívánt adat.

Ezek alapján azt tapasztaljuk, hogy ugyan a hívás során nincs probléma, a teendőket tartalmazó lista tökéletesen megérkezik, a felületen a lista mégsem jelenik meg. Gyanítható, hogy ebből kifolyólag a probléma oka az adatkötés környékén keresendő. Hiába frissül a `Todos` tulajdonság, erről a változásról a felület nem értesül. Ennek megoldására tegyük ezt a tulajdonságot változásértesítőssé. A megoldás során használjuk a Prism.MVVM.BindableBase által nyújtott lehetőségeket.

Vegyük fel a tulajdonsághoz tartozó privát belső változót, és töltsük fel a `get` és a `set` törzsét. A `get` esetén minden a megszokott módon történik, ám a `set` ágban a szokásos `_todos = value;` helyett használjuk a `Prism.MVVM.BindableBase SetProperty` hívását. Ez első paramétereként a tulajdonsághoz tartozó privát változónk referenciáját, míg második paramétereként magát a beállítandó értéket várja.

```
private ObservableCollection<TodoItem> _todos;

public ObservableCollection<TodoItem> Todos
{
    get { return _todos; }
    set { SetProperty(ref _todos, value); }
}
```

Próbáljuk ki!

Ezzel elkészültünk az alkalmazás lényegi részével. A labor ezen pontjáig eljutva **elégséges** érdemjegy kapható.

ÖNÁLLÓ FELADATOK

1. HOZZÁADÁS GOMB COMMAND

A főoldalon a hozzáadás gomb eseménykezelője ne a xaml.cs, code-behind fájlban legyen, hanem kerüljön át a ViewModelbe. Ehhez használjuk a XAML-ben alkalmazott Command mintát (`ICommand` interfész):

- ViewModel-ben hozzunk létre egy `ICommand` interfészt implementáló objektumot tartalmazó publikus tulajdonságot (tipikusan csak gettere van)
  - Tipp: Prism `DelegateCommand` osztály
- A Command végrehajtásakor az futtasson egy privát, ViewModel-ben lévő metódust, ami a navigációt végzi el
  - Tipp: A navigációhoz `NavigationService`-re lesz szükségünk, amit dependency injection segítségével injektálhatunk a `ViewModel`-be.
  - Tipp: Navigációra lehet példát találni az `App.xaml.cs` fájlban
- A nézeten a gomb használja ezt a Commandot adatkötésen keresztül, töröljük a `Click` eseménykezelőt.

Eddig eljutva **közepes** osztályzat szerezhető.

2. TODODETAILSPAGEVIEWMODEL, NAVIGÁCIÓ

Készítsük el a `TodoDetailsPage`-hez tartozó ViewModel osztályt is. Követelmények:

- Támogassa az adatkötést és a navigációt is, mint a `MainViewModel`. (A Prism által nyújtott lehetőségeket kihasználva.)
- Működjenek a jelenlegi adatkötések. Ne felejtsd el a xaml.cs-ből eltávolítani a felesleges részeket. *Mentéssel még nem kell foglalkozni.*
- Navigáció során **ne** a teljes `TodoItem` objektumot adjuk át, hanem csak az elem `Id`-ját, új teendő esetén `null`-t.
- A helyes működéshez a `NavigationService`-t kell használni, amiből egy példány most a `ViewModel`-ben található. A listában az elemre történő kattintást most nem szükséges Command alapúra átírni, elég, ha az eseménykezelő függvény csak áthív a VM-be.
  - Tipp: A code-behind fájlban a VM kiolvasható a `DataContext` tulajdonságból.
- Készíts el egy webes kérést a `TodoService`-be, ami egy adott `Id`-val rendelkező teendőt ad vissza (lásd swagger: `GET api/Todo/{id}`)
  - ***Ha lusta vagy megvalósítani a kérést, akkor előbb oldd meg a 4. önálló feladatot, ezzel időt spórolva majd a későbbi feladatokra is*** ☺
- Ha nem `null` a navigáció során kapott paraméter, akkor kérjük le a teendő elemeit a szerverről aszinkron módon

Eddig eljutva **jó** osztályzat szerezhető.

Megj.: Egy életszerű példában azért szoktuk lekérni külön a részletes oldalon újra az adatokat, mert a listában használt modellben általában az adatoknak csak egy kis része található meg. Esetünkben ez a kettő most megegyezik.

---

3. SWAGGER

Használjuk ki a swagger webszolgáltatást leíró képességét kódgenerálásra. A `TodoService` kézi megírása helyett lehetőség van a swagger leíró állományból kliens oldali C# kliens osztály és modell generálására.

- Generálás:
  - Projekten jobb gomb Add / REST API Client
  - Swagger URL-be beszúrni a swagger JSON leíró állományára mutató linket (swagger oldalról)
  - Namespace megválasztása: pl.: `TodoPrism.TodoServiceApi`
  - Osztálynév megválasztása: pl.: `TodoServiceApiClient`
- Töröljük ki a régi `TodoItem` és `Priority` modell osztályainkat
- A `TodoService` osztályt legyen továbbra is a fő szolgáltatásunk
  - Származzon a generált kliens osztályból
  - Az ős konstruktorában adjuk meg a szerver címét és egy HttpClient objektumot
  - töröljük az eddigi metódusokat

```
public class TodoService : TodoServiceApiClient
    {
        public TodoService() : base("https://bmetodoservice.azurewebsites.net", new HttpClient())
        {

        }
    } }
```

- Javítsuk ki a maradék fordítási hibákat (tipikusan névterek)
- **Próbáljuk ki!**

4. MENTÉS

Valósítsuk meg a mentést.

- Adatkötött, Command alapú művelet legyen a ViewModel-ben.
  - Létrehozás esetén a POST api/Todo REST-es kérést használd
  - Módosítás esetén a PUT api/Todo/{id} REST-es kérést használd
  - ***Tipp.: a 4. feladat sokat segíthet***
- A webes kérések aszinkron módon legyenek megvalósítva
- Navigáljunk vissza, ha lefutott a mentés

Eddig eljutva **jeles** osztályzat szerezhető.

5. IMSC FELADATOK

Valósítsuk meg a törlést.

A részletes oldalról történő törlés megvalósítása 1 IMSC pontot ér, míg amennyiben a listából **is** lehetőség van az elem törlésére, akkor 2 IMSC pontot szerezhetsz.