CROSS-PLATFORM FEJLESZTÉS XAMARIN.FORMS HASZNÁLATÁVAL

Segédlet a Kliens oldali technológiák c. tárgyhoz

Czeglédi Viktor

2018.

Utoljára frissítve: Erdős Szilvia (2020)

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

A labor során megismerjük a Xamarin.Forms technológiát, mely segítségével könnyedén tudunk cross-platform kódot fejleszteni UWP, Android, iOS, MacOS környezetekre. A technológia különlegessége, hogy a felület leírása is közös kódbázissal történik.

Előfeltételek

A labor elvégzéséhez szükséges eszközök:

- Microsoft Visual Studio
- Xamarin SDK (Mobile Development with .NET option for Xamarin)

Amit érdemes átnézned

- Kliens oldali technológiák előadások követése

Lebonyolítás

A gyakorlat anyagát távolléti oktatás esetén önállóan, jelenléti oktatás esetében számítógépes laborban, gyakorlatvezetői útmutatással kell megoldani.

Beadás

Amennyiben elkészültél a megoldással, távolítsd el belőle a fordítási binárisokat és fordítási segédmappákat („bin" mappa, „obj" mappa, „Visual Studio" mappa, esetleges nuget/node csomagok), majd az így elkészült anyagot egy zip fájlba becsomagolva töltsd fel a Moodle rendszerbe. Amennyiben a zip fájl mérete 10 MB fölött van, valószínűleg nem töröltél ki belőle mindent a korábbi felsorolásból. Jegyzőkönyv készítése nem szükséges, azonban amennyiben a beadott kódoddal kapcsolatban kérdések merülnek fel, megkérhetünk arra, hogy működésének egyes részleteit utólag is magyarázd el.

Összefoglalás

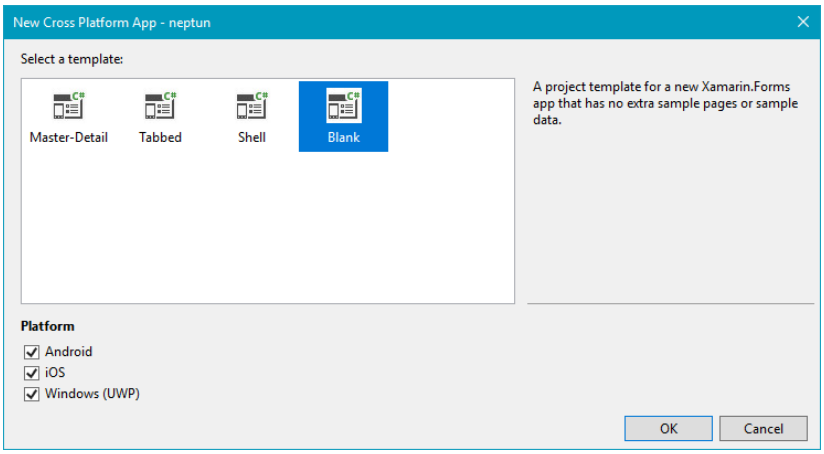
A Xamarin.Forms célja olyan platform biztosítása, amelynek segítségével egy közös kódbázisban tudunk fejleszteni a népszerű platformokra. Az üzleti logikát C#-ban írhatjuk, a felület leírót pedig XAML-ben. Ezek után a Forms a futás során leképzi ezt a leírót platform specifikus vezérlőkre, így biztosítva a natív élményt.

Feladatok

1. Feladat – Új projekt létrehozása, Forms projekt felépítése

Az első feladatban kialakítjuk a környezetet.

1. Hozzunk létre egy Xamarin Forms projektet Visual Studioban! Készítsünk egy új projektet és válasszuk a **Mobile App (Xamarin.Forms)** sablont. **A projekt neve a Neptun kódunk legyen**!
2. A projekt létrehozása után a felugró ablakban válasszuk a Blank App-ot, és a Platformok kiválasztása során ügyeljünk arra, hogy a Windows (UWP) mindenképp ki legyen pipálva.



3. *(A Visual Studio korábbi verzióiban File -> New Project -> C# -> Cross-platform -> Cross Platform App sablont használjuk. A felugró ablakban a template maradjon Blank App, a UI technology maradjon Xamarin.Forms de a Code Sharing Strategy-t állítsuk át Portable Class Library-re.)*
4. A megfelelő sablonok használatához szükséges a **Mobile Development with .NET** Xamarinra opció telepítése Visual Studio Installerben. Így, ha a templetet nem találjuk, ellenőrizzük, hogy biztosan fel van-e telepítve.

Vizsgáljuk meg a projekt felépítését.

5. Van egy közös PCL projektünk, ez lényegében egy DLL-re fordul, amit minden platform specifikus projekt bereferál. Ide kerülnek a közös C# és XAML fájljaink. Ezen kívül van még néhány platform specifikus projekt, ezeknek a feladata a Xamarin.Forms alkalmazás elindítása. Ha meg akarjuk nézni ez hogyan is történik, nézzük meg az UWP projekt App.xaml.cs fájlját. A fájl szinte teljesen megfelel egy szokásos UWP alkalmazás sablonnak, kivéve az OnLaunched függvénybe lévő Xamarin.Forms.Forms.Init(e); sort, ami a Forms inicializálásáért felel. Ezután az alkalmazásunk elnavigál a platform specifikus projektben lévő kezdő oldalra, UWP esetén a MainPage.xaml-re. Ez valójában csak egy belépési pont a közös kódrészekbe, ahogy látható is a MainPage.xaml.cs konstruktorában. Itt valójában elindul a közös projektbe lévő App osztály. Az App osztály konstruktora pedig be is állítja az alkalmazás valódi kezdőoldalát.
6. Tetszés szerint vizsgáljuk meg a többi projektet is, hogy az egyes platformokon hogyan indul el az alkalmazás.
7. Ezután nézzük meg a közös projektbe lévő MainPage.xaml fájlt. Láthatjuk, hogy ez a fájl nagyon hasonlít az UWP laboron tanult XAML fájlokra. Ez azért van, mert a XAML egy általános nyelv, tehát az, hogy ebben az UWP platformnak megfelelő felület leírót készítünk, vagy Forms felületet írunk le, nem befolyásolja az alapjait. Az elemek példányosítása, a Property-k beállítása teljesen hasonlóképpen történik. Amibe különbséget vehetünk észre, az a felületet leíró elemek neve és tulajdonságai. Ez azért van, mert a Forms egy saját objektum modellt épített fel, ami platform függetlenül képes leírni egy felületet. Itt Page-ek, Layout-ok és View-k szerepelnek a felületen, amik majd leképződnek a megfelelő natív vezérlőkre.
8. Fontos, hogy a labor során csak az UWP alkalmazást fogjuk futtatni, de mivel minden kódrész, amit írunk közös, ezért egy megfelelő fejlesztő környezettel rendelkező gépen a Startup project átállításával azonnal futtathatnánk is az elkészült alkalmazást akár Androidon akár iOS-en, és egy azonos felépítésű, de natív felületet kapnánk minden platformon. Kivéve persze a specifikusan megadott részeket.

2. Feladat – ToDo alkalmazás létrehozása

Első lépésként írjuk át a közös projektben lévő App.xaml.cs-ben a MainPage példányosítását az alábbi sorra, hogy később a beépített navigáció segítségével tudjunk navigálni az oldalak között.

```
MainPage = new NavigationPage(new MainPage());
```

Elsőként vegyük fel az adatmodellünket, ehhez a közös projektbe vegyünk fel egy Models mappát majd abba egy TodoItem osztályt.

```
public class TodoItem
{
    public string Title { get; set; }

    public string Description { get; set; }
}
```

Ezután készítsük el az alkalmazásunk logikáját. Lesz egy Singleton tárolónk, ami a teendőket fogja tárolni. Ehhez hozzunk létre a közös Portable projektbe egy Services mappát, abba pedig egy TodoService osztályt az alábbi módon.

```
public class TodoService
{
    public static TodoService Instance { get; } = new TodoService();

    private Dictionary<int, TodoItem> items = new Dictionary<int, TodoItem>
    {
        {0, new TodoItem {Description = "Nagybevásrálást elintézni", Title = "Bevásárlás"}},
        {1, new TodoItem {Description = "Lemenni a tóhoz és horgászni egy jót", Title = "Horgászat"}},
    };

    protected TodoService() { }

    public void AddItem(string title, string description)
    {
        items.Add(items.Keys.Max() + 1, new TodoItem { Title = title, Description = description});
    }

    public TodoItem GetItem(int id)
    {
        return items[id];
    }

    public List<TodoItem> GetAll()
    {
        return items.Values.ToList();
    }
}
```

Xamarin.Forms-ba szintén az MVVM mintát követik az alkalmazások, ezért hozzuk létre a kezdő oldal viewmodelljét. Ehhez készítsünk egy ViewModels mappát a közös projektbe, majd abba egy ViewModelBase osztályt, mint közös ősosztályt a ViewModelleknek.

Ez az osztály implementálni fogja az INotifyPropertyChanged interfészt, és tartalmazni fog egy INavigation property-t, amin keresztül fogunk tudni navigálni a ViewModellekből is. Ezenkívül van egy OnAppearing metódusa, amivel majd a megfelelő View jelezni tudja a ViewModellnek, hogy meg fog jelenni, tehát érdemes frissítenie az adatát.

```
public abstract class ViewModelBase : INotifyPropertyChanged
{
    public INavigation Navigation { get; set; }

    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public virtual void OnAppearing()
    {
    }

    public ViewModelBase(INavigation navigation)
    {
        Navigation = navigation;
    }
}
```

Ezekután készítsük el a MainPage-hez tartozó MainViewModel-t, melyet hozzunk létre szintén a ViewModels mappába. A benne található AddItemPage függvényt később fogjuk implementálni.

```
public class MainViewModel : ViewModelBase
{
    public List<TodoItem> Items => TodoService.Instance.GetAll();

    public ICommand AddItemCommand { get; }

    public MainViewModel(INavigation navigation) : base(navigation)
    {
        AddItemCommand = new Command(AddItemCommandExecute);
    }

    private async void AddItemCommandExecute()
    {
        await Navigation.PushAsync(new AddItemPage(), true);
    }

    public override void OnAppearing()
    {
        base.OnAppearing();
        OnPropertyChanged(nameof(Items));
    }
}
```

Most, hogy megvan a ViewModel, írjuk át a MainPage.xaml-t, hogy megjelenjen az adat. Ehhez első lépésként szedjünk le egy szimpatikus hozzáadás ikon-t. (Érdemes a Segoe MDL2 iconok közül választani a https://docs.microsoft.com/en-us/windows/uwp/design/style/segoe-ui-symbol-font oldalról, de

bármi hasonló is megfelelő. A fenti oldalon az ikon képére jobb klikkel kattintva le tudjuk menteni képként.) Ezt az ikont mentsük le az UWP projekt Assets mappájába *add.png* néven. Így majd az UWP platformon el fogjuk érni a megfelelő ikont. Ezután írjuk át a MainPage.xaml-t a közös projektben az alábbi kódra. Ne felejtsük el az `x:Class="neptun.MainPage"` helyen a névteret módosítani.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="neptun.MainPage"
             Title="Awesome ToDo app">

    <Grid HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Label VerticalOptions="Start"  HeightRequest="150"
                   HorizontalOptions="FillAndExpand"
                   HorizontalTextAlignment="Center"
                   VerticalTextAlignment="Center"
                   FontSize="20"
                   FontAttributes="Bold"
                   TextColor="White"
                   BackgroundColor="Blue"
                   Text="Összes teendő"/>
        <ListView Grid.Row="1" ItemsSource="{Binding Items}">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <TextCell Text="{Binding Title}" Detail="{Binding Description}"
                              TextColor="Blue"
                              DetailColor="Gray"/>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </Grid>

    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Hozzáadás" Command="{Binding AddItemCommand}">
            <ToolbarItem.Icon>
                        <FileImageSource File="Assets/add.png"/>
            </ToolbarItem.Icon>
        </ToolbarItem>
    </ContentPage.ToolbarItems>
</ContentPage>
```

Figyeljük meg hogy itt egy ContentPage van, amibe Grid-be rendezzük el a felületet. A felső sora egy fejléc. Alatta pedig egy lista található, ami ez elemeket jeleníti meg. Itt érdemes összevetni a beállítási lehetőségeket az UWP-ből ismert lehetőségekkel. Fontos még a ToolbarItem.Icon tartalma. Itt egy OnPlatform elemet használunk, aminek a segítségével különböző platformokra különböző beállításokat adhatunk meg. Esetünkben a kép elérése platformonként máshol lenne, ezét csak a Windows-t adjuk meg.

Ha ez megvan, a MainPage.xaml.cs-ben a közös projektben állítsuk be a BindingContext-et, ami az adatkötés kontextusát adja meg. Illetve hívjuk meg a ViewModel megfelelő eseményét.

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
        this.BindingContext = new MainViewModel(Navigation);
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();
        (this.BindingContext as MainViewModel).OnAppearing();
    }
}
```

A MainViewModel AddItemCommandExecute törzsét kikommentezve kipróbálhatjuk, hogy működik-e a projektünk. Az elindításhoz az UWP projektre jobb gombbal kattintva Set as Startup project opciót kiválasztva a megfelelő project fog elindulni. Arra figyeljünk, hogy a Platform x86 legyen, ne Any CPU.

Ezután készítsük el az elem hozzáadása oldalt. Ehhez adjunk hozzá a projekthez egy AddItemViewModel osztályt a ViewModels mappánkba.

```
public class AddItemViewModel : ViewModelBase
{
    private string title;

    public string Title
    {
        get { return title; }
        set
        {
            title = value;
            OnPropertyChanged();
        }
    }

    private string description;

    public string Description
    {
        get { return description; }
        set
        {
            description = value;
            OnPropertyChanged();
        }
    }

    public ICommand SaveItemCommand { get; }

    public AddItemViewModel(INavigation navigation) : base(navigation)
    {
        SaveItemCommand = new Command(AddItemCommandExecute);
    }

    private async void AddItemCommandExecute()
```

```
    {
        TodoService.Instance.AddItem(Title, Description);
        await Navigation.PopAsync(true);
    }
}
```

Majd adjunk hozzá a közös projekthez egy Forms Blank Content Page Xaml (régi nevén Forms Xaml Page) elemet. Ehhez jobb klikk a projekten, majd Add → New Item → Xamarin.Forms menü segítségével keressük ki a Content Page elemet. A neve AddItemPage.xaml legyen, a tartalma pedig az alábbi XAML. Szintén ne felejtsük el a `x:Class="neptun.AddItemPage"` alatt a névteret frissíteni.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="neptun.AddItemPage"
             Title="Awesome ToDo app">

    <Grid HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Label VerticalOptions="Start"
               HeightRequest="150"
               Grid.ColumnSpan="2"
               HorizontalOptions="FillAndExpand"
               HorizontalTextAlignment="Center"
               VerticalTextAlignment="Center"
               FontSize="20"
               FontAttributes="Bold"
               TextColor="White"
               BackgroundColor="Blue"
               Text="Új elem hozzáadása"/>
        <Label VerticalOptions="Center" Margin="8" Grid.Row="1" Grid.Column="0" Text="Név:
"/>
        <Editor Margin="8" Grid.Row="1" Grid.Column="1" Text="{Binding Title, Mode=TwoWay}"/>
        <Label    VerticalOptions="Center"    Margin="8"    Grid.Row="2"    Grid.Column="0"
Text="Leírás: "/>
        <Editor   Margin="8"   Grid.Row="2"   Grid.Column="1"   Text="{Binding   Description,
Mode=TwoWay}"/>
    </Grid>

    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Mentés" Command="{Binding SaveItemCommand}">
            <ToolbarItem.Icon>
                        <FileImageSource File="Assets/save.png"/>
            </ToolbarItem.Icon>
        </ToolbarItem>
    </ContentPage.ToolbarItems>
</ContentPage>
```

Szintén hiányzik a mentés ikon, ehhez az add.png-hez hasonló módon mentsünk le egy save ikont save.png néven az Assets mappájába az UWP projektnek.

Utolsó lépésként szintén az AddItemPage.xaml.cs-be példányosítsunk egy ViewModel-t.

```
public AddItemPage()
{
    InitializeComponent();
    this.BindingContext = new AddItemViewModel(Navigation);
}
```

Ha korábban kommenteztük a MainViewModel AddItemCommandExecute metódusának törzsét, akkor most kommentezzük ki, hiszen már létrehoztuk az ezt használó függvényt.

Most már működik az alkalmazásunk, próbáljuk is ki. Adjunk hozzá néhány elemet a listához.

3. Platform specifikus renderer írása

Láthatjuk, hogy a beépített szövegdoboz UWP platformon egy vastag keretet tartalmaz alapból a részletező oldalon. Ha szeretnénk, hogy jobban nézzen ki, és egy vékonyabb vonal legyen körülötte, akkor UWP-be a BorderThickness tulajdonságot kellene átállítani. Viszont a Forms-ban egy Editor elemünk van, és mivel ez egy általánosított felület leíró nyelv, így természetesen nem tartalmazza a platform specifikus részeket. A keretrendszer működése során a Forms-os elem és a natív elem közti leképzést úgynevezett Renderer-ek végzik. Lehetőségünk van arra, hogy adott elemekhez saját Renderer-t írjunk, így platform specifikus beállításokat is elérve.

Ehhez első lépésként érdemes létre hozni egy saját vezérlőt, hogy ne rendszer szinten az összes Editor elem Renderer-ét cseréljük le. Ehhez hozzunk létre egy Renderers mappát a közös projektbe. Ebbe pedig egy CustomEditor osztályt.

```
public class CustomEditor : Editor
{
}
```

Erre az osztályra tényleg csak azért van szükség, hogy ne egy beépített rendszer elemet cseréljünk le.

Ezután az UWP projektbe is hozzunk létre egy Renderers mappát, amibe rakjunk egy UwpEditorRenderer osztályt. Ez az osztály fogja beállítani a platform specifikus részeket. A kód az alábbi módon néz ki, ha másolunk, ne felejtsük a névtér javítását.

```
[assembly: ExportRenderer(typeof(CustomEditor), typeof(UwpEditorRenderer))]
namespace neptun.UWP.Renderers //névteret javítani
{
    public class UwpEditorRenderer : EditorRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<Editor> e)
        {
            base.OnElementChanged(e);

            if (Control != null)
            {
                Control.BorderThickness = new Windows.UI.Xaml.Thickness(1);
            }
```

```
        }
    }
}
```

Ez az osztály az Editor elem beépített Rendereréből az EditorRenderer-ből származik. Elsőként nézzük meg az OnElementChanged metódust. Ez hívódik meg egy elem megváltozásánál. Itt paraméterbe megkapjuk a régi és az új Forms beli Editor elemet, bár erre most nincs szükségünk. Az ősosztály Control propertyjén keresztül pedig elérjük a platform specifikus vezérlőt. Így, ha ez nem null, akkor be tudjuk állítani a BorderThickness propertyjét tetszőleges értékre. Az utolsó fontos rész a névtér fölött lévő globális attributum. Ebben megadjuk, hogy a CustomEditor vezérlőnk renderere a most létrehozott osztály legyen. Mivel a többi platformon ez nem lesz beállítva, ezért ott a CustomEditor ősének az Editor osztálynak a gyári renderere fog továbbra is működni. Utolsó lépésként cseréljük le a két Editor elemünket az AddItemPage.xaml-ben. (Névtérre itt is figyeljünk)

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:renderers="clr-namespace:neptun.Renderers;assembly=Neptun"
             x:Class="Neptun.AddItemPage"
             Title="Awesome ToDo app">

    <Grid HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Label VerticalOptions="Start"
               HeightRequest="150"
               Grid.ColumnSpan="2"
               HorizontalOptions="FillAndExpand"
               HorizontalTextAlignment="Center"
               VerticalTextAlignment="Center"
               FontSize="20"
               FontAttributes="Bold"
               TextColor="White"
               BackgroundColor="CornflowerBlue"
               Text="Új elem hozzáadása"/>
        <Label VerticalOptions="Center" Margin="8" Grid.Row="1" Grid.Column="0" Text="Név:
"/>
        <renderers:CustomEditor Margin="8" Grid.Row="1" Grid.Column="1" Text="{Binding
Title, Mode=TwoWay}"/>
        <Label VerticalOptions="Center" Margin="8" Grid.Row="2" Grid.Column="0"
Text="Leírás: "/>
        <renderers:CustomEditor Margin="8" Grid.Row="2" Grid.Column="1" Text="{Binding
Description, Mode=TwoWay}"/>
    </Grid>

    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Mentés" Command="{Binding SaveItemCommand}">
```

```
            <ToolbarItem.Icon>
                <OnPlatform x:TypeArguments="FileImageSource">
                    <On Platform="Windows">
                        <FileImageSource File="Assets/save.png"/>
                    </On>
                </OnPlatform>
            </ToolbarItem.Icon>
        </ToolbarItem>
    </ContentPage.ToolbarItems>
</ContentPage>
```

Ezután elindítva az alkalmazást láthatjuk, hogy az elem körvonala megváltozott az új elem hozzáadása oldalon.

4. Önálló feladat

Az alkalmazásban készre lehessen jelölni a ToDo elemeket egy kapcsoló segítségével, illetve ezt a készre jelölést már a felvétel oldalon is el lehessen végezni.

Segítség: A MainPage-en a TextCell helyett egy egyedi cella sablon kell, így ezt cseréljük le egy ViewCell elemre, amibe már tetszőleges felületet rakhatunk össze hasonlóan az UWP platform DataTemplate-jéhez. (Lényegében ez is az csak kötelezően egy ViewCell az első gyereke és ő tartalmazza a felület leírót.)