Eseményvezérelt alkalmazások: 13. gyakorlat

A munkafüzet a reaktív programozást demonstrálja az Avalonia UI keretrendszerben.

1 Reaktív programozás Avalonia UI keretrendszerben ^{KM}

Az asztali grafikus alkalmazások felületi eseményei és állapotváltozásai is kezelhetőek reaktív módon, .NET keretrendszerben ezt a ReactiveUI könyvtár teszi lehetővé, amely Windows Forms, Windows Presentation Foundation és Avalonia UI felületű alkalmazásokkal is könnyen integrálható 1-1 NuGet csomagnak a projekthez adásával. A *ReactiveUI* keretrendszer az *Rx.NET* osztálykönyvtárra épül.

MVVM architektúra esetén a nézetmodell megfigyelhető tulajdonságainak (`INotifyPropertyChanged` interfész) változása egy megfigyelhető felsorolóként is értelmezhető. Hasonló szemlélettel a parancsok (`ICommand` interfész) végrehajthatósága (`CanExecute`) is tekinthető egy logikai értékeket felsoroló objektumnak, amely változásairól eseményeket küld (`CanExecuteChanged`), azaz megfigyelhető.

Feladat: Készítsük el a webes képletöltő alkalmazásunkat a reaktív programozás paradigmáját alkalmazva, a *ReactiveUI* keretrendszert használva.

1.1 Projekt létrehozása

Készítsünk Visual Studioban egy új *Avalonia C# Project*-et, a *solution* és a *projekt* neve legyen *ImageDownloader.Avalonia*. Válasszuk ki most is a *Desktop* és *Android* platformokat támogatásra, alkalmazott tervezési mintaként (*design pattern*) azonban most a *ReactiveUI*-t jelöljük ki. Megfigyelhetjük, hogy a platformfüggetlen *ImageDownloader.Avalonia* projekthez a `CommunityToolkit.Mvvm` helyett most az `Avalonia.ReactiveUI` NuGet csomag került hozzáadásra.

A modell nem változik, ezért a *solution*-be átmásolhatjuk a *8. gyakorlaton* elkészített platformfüggetlen *Class Library* projektet *ImageDownloader.Model* néven. Adjuk hozzá az *ImageDownloader.Avalonia* projektünkhöz a létrehozott *ImageDownloader.Model* projektet függőségként (*"Add Project Reference"*).

1.2 Nézetmodell

Figyeljük meg, hogy a generált `ViewModelBase` osztályunk most a *ReactiveUI* `ReactiveObject` típusából származik le. Ez az ősosztály megvalósítja a `INotifyPropertyChanged` interfészt és a reaktív programozáshoz (később használandó) eszközöket biztosít számunkra.

1.2.1 `ImageViewModel`

A *10. gyakorlatról* emeljük át a `ImageViewModel` osztályt, majd igazítsuk a *ReactiveUI* keretrendszerhez. A `RelayCommand` osztály helyett most a `ReactiveCommand<T1, T2>` osztályt használhatjuk a parancsainkhoz, amely `T1` típusú paramétert vár és `T2` típusú eredményt ad eredményül. Ha nem akarunk paramétert vagy eredményt megadni, akkor tekintettel arra, hogy a `void` nem használható generikus típusparaméterként, a `Rx.NET` speciális `Unit` típusát használhatjuk. Ennek megfelelően:

- a `SaveImageCommand` és a `CloseCommand` típusa legyen `ReactiveCommand<Unit, Unit>`;
- definiálni a `ReactiveCommand.Create()` eljárással tudjuk őket, pl.:

```
SaveImageCommand = ReactiveCommand.Create(() =>
{
```

```
    SaveImage?.Invoke(this, Image);
});
```

1.2.2 `MainViewModel`

A *10. gyakorlatról* emeljük át a `MainViewModel` osztályt is, majd igazítsuk ezt is a *ReactiveUI* keretrendszerhez:

- A `Progress` és az `IsDownloading` megfigyelhető tulajdonságokhoz használjuk a `ReactiveObject`-ből örökölt `RaiseAndSetIfChanged()` metódust, amellyel egyszerre beállíthatjuk a kapcsolt adattagot és kiválthatjuk a `PropertyChanged` eseményt, pl.:

```
private float _progress;

public float Progress
{
    get => _progress;
    set => this.RaiseAndSetIfChanged(ref _progress, value);
}
```

- Az `IsDownloading` változásának esetén a `DownloadButtonLabel` tulajdonságra is ki kell váltatnunk `PropertyChanged` eseményt. Erre ReactiveUI használata esetén a `ReactiveObject`-ből örökölt `RaisePropertyChanged()` metódust használhatjuk az `IsDownloading` *setter* ágában.
- Az `ImageSelectCommand` típusa legyen `ReactiveCommand<Bitmap, Unit>`, hiszen ez a parancs paramétert vár, de visszatérési értéke nincsen. Definiálásához használjuk a generikus `ReactiveCommand.Create<Bitmap, Unit>()` gyártó műveletet.
- A `DownloadCommand` típusa hasonló módon legyen `ReactiveCommand<string, Unit>`. Definiálásakor a `ReactiveCommand.CreateFromTask<string, Unit>()` eljárást használjuk, így aszinkron tevékenység is végrehajtható a törzsében (a `Download` metódus aszinkron):

```
DownloadCommand = ReactiveCommand.CreateFromTask<string, Unit>(async param =>
{
    await Download(param);
    return Unit.Default;
});
```

1.3 Alkalmazás futtatása

A nézetek változatlan formában felhasználhatóak a *10. gyakorlaton* elkészített megoldásból. Futassuk az alkalmazást!

Vegyük észre, hogy a képek letöltése után a *"Letöltés megszakítása"* gomb nem engedélyezett, így a funkció nem használható. Ennek oka, hogy alapértelemezetten a `ReactiveCommand`-hoz kapcsolt `CanExecute` olyan logikai értékeket sorol fel, hogy a parancs párhuzamosan többször nem hajtható végre. Míg az *MVVM Toolkit* esetében ezt a `RelayCommand` attribútum `AllowConcurrentExecutions=true` argumentumával oldottuk meg, a `ReactiveCommand` esetében más megközelítést alkalmazunk: megadjuk, hogy a `CanExecute` egyetlen `true` értéket soroljon fel, így mindig végrehajtható lesz a parancs:

```
DownloadCommand = ReactiveCommand.CreateFromTask<string, Unit>(async param =>
{
    await Download(param);
    return Unit.Default;
}, Observable.Return(true));
```

2 Képek szűrése a *ReactiveUI* használatával ^{EM}

Egészítsük ki a webes képletöltő alkalmazás funkcionalitást egy szűrési lehetőséggel, amellyel a letöltött képek helyettesítő szövegében (`alt` attribútum) lehet keresni.

2.1 Modell

A modell két osztályát (`WebImage` és `WebPage`) egészítsük ki az új funkcionalitás támogatásával:

- a `WebImage` osztályt egészítsük ki egy publikus `string AltText` tulajdonsággal (legyen publikusan írható is);
- a `WebPage` osztályban egy új `WebImage` elkészítésekor állítsuk be annak helyettesítő szövegét (ha van):

```
var image = await WebImage.DownloadAsync(imageUrl);
if (node.Attributes.Contains("alt"))
{
    image.AltText = node.Attributes["alt"].Value;
}
```

2.2 Nézetmodell

Egészítsük ki a `ImageViewModel` nézetmodellt 2 új tulajdonsággal (`AltText`, `IsEnabled`), a következő módon:

- A `string AltText` a kép helyettesítő szövegét tárolja (nem kell az állapotváltozásait jeleznie, mivel nem fogjuk módosítani). A konstruktor várja paraméterül az `AltText` értékét.
- A `bool IsEnabled` adja meg, hogy a szűrőkifejezés alapján a kép kiválasztott-e. Legyen megfigyelhető, ehhez használjuk a `ReactiveObject`-ből örökölt `RaiseAndSetIfChanged()` metódust a *setter* ágban, a már korábban látottakhoz hasonlóan.

A `MainViewModel` nézetmodellben implementáljuk a letöltött képek közötti szűrést, a reaktív paradigma alkalmazásával.

- Először is adjunk az osztályhoz egy `string SearchContent` megfigyelhető tulajdonságot (és egy kapcsolt `_searchContent` adattagot), amelyet adatkötéssel a felület kereső mezőjének tartalmához tudunk majd kötni.
- Adjunk az osztályhoz egy `IObservable<string> _searchContentObservable` privát adattagot is, amely a `SearchContent`-nak a felhasználói input hatására változó tartalmát fogja megfigyelhető módon felsorolni.
- Az osztály konstruktorában definiáljuk a `_searchContentObservable`-t a következő módon:
  1. A `SearchContent` változásait szeretnénk felsorolni, ehhez használhatjuk a `ReactiveObject` ősosztályhoz tartozó `WhenAnyValue()` eljárást:
     ```
     this.WhenAnyValue<MainViewModel, string>(x => x.SearchContent);
     ```
  2. Csak akkor állítson elő a felsorolónk új elemet, ha 0.5 másodperce nem történt billentyűleütés, azaz állapotváltozás a `SearchContent` értékében. Ehhez használjuk az *Rx.NET* `Throttle(TimeSpan.FromSeconds(0.5))` szűrőjét.
  3. Csak akkor állítson elő a felsorolónk új elemet, ha a keresőmező tartalma a *whitespace*-eket levágva megváltozott. Ehhez használjuk a `Select()` transzformátort és a `DistinctUntilChanged()` szűrőt.
  4. Az aszinkron feldolgozás miatt fontos, hogy a megfigyelők a *UI threaden* kerüljenek majd végrehajtára, hiszen majd a felületen megjelenített képek vezérlőihez hozzá kell férniük. Ezt grafikus keretrendszertől független módon a `ObserveOn(RxApp.MainThreadScheduler)`hozzáadásával érhetjük el.

- Az `Images`-ben a továbbiakban nem csak a képet (`Bitmap`), hanem a képre vonatkozó teljes nézetmodellt (`ImageViewModel`) tárolni fogjuk, így a típusát is ennek megfelelően változtassuk `ObservableCollection<ImageViewModel>`-re.
- Az `OnImageLoaded` eseménykezelőben az új `ImageViewModel` létrehozása után állítsuk be, hogy amennyiben az adott kép helyettesítő szövege tartalmazza a megadott keresőkifejezést (vagy az üres), akkor engedélyezett a kép, egyébként nem.

```
private void OnImageLoaded(object? sender, WebImage webImage)
{
    if (!IsSupportedExtension(webImage.Url.LocalPath))
        return;

    var bitmap = new Bitmap(new MemoryStream(webImage.Data));
    var imageViewModel = new ImageViewModel(bitmap, webImage.AltText);
    _searchContentObservable
        .Select(s => imageViewModel.AltText.Contains(s) || s == string.Empty)
        .Subscribe(s => imageViewModel.IsEnabled = s);

    Images.Add(imageViewModel);
}
```

2.3 Nézet

A `MainView` nézetet egészítsük ki még egy szövegdobozzal, amellyel a kereső kifejezést a felhasználóm megadhatja. Ennek `Text` tulajdonságát kössük a nézetmodell `SearchContent` tulajdonságához.

A képeket megjelenítő `ItemsControl` vezérlő `ItemTemplate`-jében a gomb (`Button`) vezérlő `IsEnabled` tulajdonságát kössük az aktuális `ImageViewModel`-nek az `IsEnabled` tulajdonságához. Így csak azok a képek lesznek kattintásra megnyithatóak, amelyekre illeszkedik a kereső kifejezés.

A kép (`Image`) vezérlő forrását (`Source`) most már nem a teljes nézetmodellhez, hanem annak `Image` tulajdonságához kössük. Ugyanitt a megjelenített képhez vegyünk fel egy `IsHighlighted` stílus osztályt, amelyet szintén az aktuális `ImageViewModel`-nek az `IsEnabled` tulajdonságához kössünk. Ezen stílussal módosítsuk azon képek áttetszőségét 10%-ra, amelyekre nem illeszkedett a keresés:

```
<Image Stretch="Fill" Source="{Binding Image}"
       Classes.IsHighlighted="{Binding IsEnabled}">
  <Image.Styles>
      <Style Selector="Image">
          <Setter Property="Opacity" Value="0.1" />
      </Style>
      <Style Selector="Image.IsHighlighted">
          <Setter Property="Opacity" Value="1" />
      </Style>
  </Image.Styles>
</Image>
```

3 Reaktív forráskód generátorok ^{OP}

Ahogyan az *MVVM Toolkit* esetében is rendelkezésünkre álltak olyan attribútumok, amelyeket adattagok és metódusok fölé illesztve a megfigyelhető tulajdonságok és a parancsok kódgenerálása automatizálható volt; erre a *ReactiveUI* esetén is lehetőségünk van, a `ReactiveUI.SourceGenerators` NuGet csomag projektünkhöz adásával.

Használatával az adattagok a `Reactive` attribútummal ellátva generálhatunk olyan *property*-t, amely változása esetén az adattagot módosítja, és a `PropertyChanged` eseményt is kiváltja magára. Pl.:

```
[Reactive]
private float _progress;

// A Progress property generálva lesz.
```

Metódusokat a `ReactiveCommand` attribútummal ellátva a nevükhöz fűzött `Command` szuffixszel ellátott nevű parancs kerül generálásra, amely végrehajtásakor az annotált metódust fogja végrehajtani. Pl.:

```
[ReactiveCommand]
private void ImageSelect(Bitmap param)
{
    ImageSelected?.Invoke(this, param);
}

// Az ImageSelectedCommand generálva lesz.
```

A `ReactiveCommand` attribútum használatakor a generikus paraméterekre nincs szükség, azt az annotált metódus szignatúrájából dedukálja.

Feladat: használjuk ki a reaktív forráskód generátorok képességeit a programunkban!