Hands-on lab

Lab: ViewModels and Commands (MVVM)

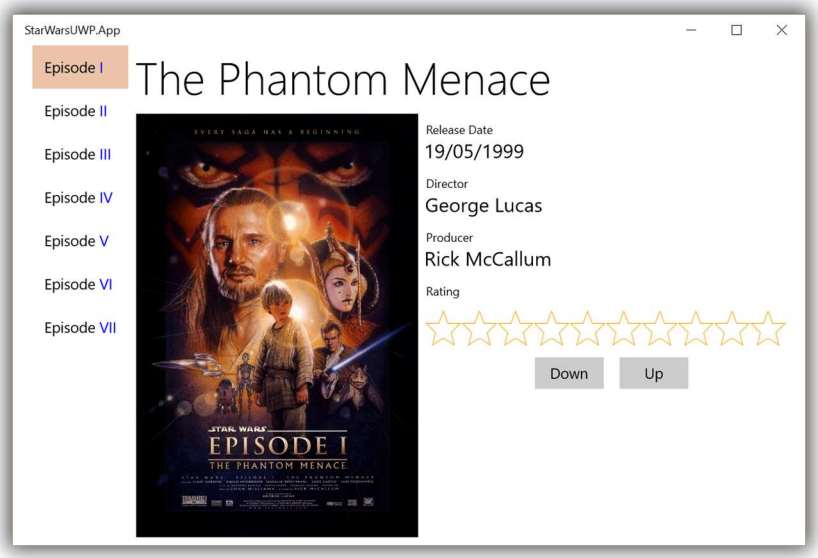
January 2016

Exercise 1: ViewModels

In this exercise, we are going to refactor the solution of the previous module (Exercise 6, Lab Databinding) in order to use proper MVVM.

Perform the following steps:

1. In the StarWarsUWP.App project, create two folders: View and ViewModel.
2. In the folder ViewModel, create a class StarWarsMoviesViewModel, which implements INotifyPropertyChanged. Create two properties: SWMovies (which is an observable collection of all movies, sorted by id) and SelectedSWMovie (which holds the current selected movie). Use the repository interface and class from the previous exercise to load the movies into the viewmodel. Select the first movie (The Phantom Menace) as the selected movie.
3. In the folder View, create a Page called StarWarsMainView. Modify App.xaml.cs in order to start this class instead of the previous one (MainPage). Copy all the XAML code, starting from the <Grid>-element and below and remove all event handlers: Click and SelectionChanged etc.
4. Now delete the MaingPage class (we don't need it anymore) and recompile your project. It should run, but there will be no data.
5. Write the necessary databindings to the viewmodel.
6. Instantiate the viewmodel from XAML and run your program (see picture below). You should be able to navigate between movies, but the buttons won't work yet.



Exercise 2: Commands

In this exercise, we are going to implement the rating functionality again.

Steps:

1. Create a folder “Utility” in the App project and write the following class.

```
using System;
using System.Windows.Input;

namespace StarWarsUWP.App.Utility
{
    public class CustomCommand : ICommand
    {
        private Action<object> execute;
        private Predicate<object> canExecute;
        public event EventHandler CanExecuteChanged;

        public CustomCommand(Action<object> execute,
                             Predicate<object> canExecute)
        {
            this.execute = execute;
            this.canExecute = canExecute;
        }

        public bool CanExecute(object parameter)
        {
            bool b = canExecute == null ? true : canExecute(parameter);
            return b;
        }


        public void RaiseCanExecuteChanged()
        {
            if (CanExecuteChanged != null)
                CanExecuteChanged(this, EventArgs.Empty);
        }

        public void Execute(object parameter)
        {
            execute(parameter);
        }
    }
}
```

Note: this is ***not*** the same class as the one you see in the Pluralsight course (MVVM) which is written for WPF. The class above is adapted for UWP. More specifically: you need to raise the CanExecuteChanged event by yourself (which re-evaluates the command bindings).

More info:
http://stackoverflow.com/questions/33029841/icommand-not-pushing-changes-to-the-ui-in-windows-uwp

2. Write a **RateUpCommand** and **RateDownCommand** in the StarWarsMoviesViewModel. Create and instantiate these commands in a **LoadCommands** method and make sure the rating functionality works again.

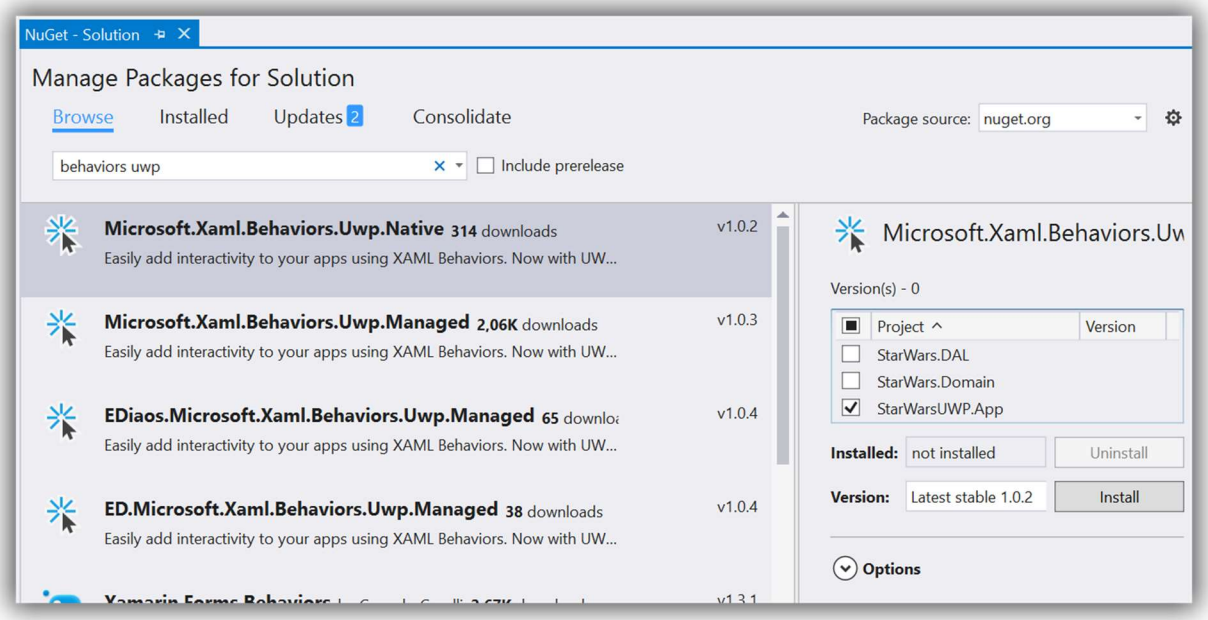
3. Now extend the functionality by means of two properties on the viewmodel: **CanRateUp** evaluaties to true if the current rating is below 10. Similarly, **CanRateDown** is true if the current rating is > 0. Use these properties on the IsEnabled property of the buttons, so that you can only click them if it makes sense.

Exercise 3: Behaviors

In this exercise, we are going to replace the Command properties of the buttons with behaviors and use behaviors to load data in the Loaded event of the page.

1. There are no built-in behaviors defined for UWP, we have to install them via Nuget. In the NuGet package manager do a search on “behaviors uwp” and install Microsoft.Behaviors.Uwp.Native into the StarWarsUWP.App project.

More info:
https://blogs.windows.com/buildingapps/2015/11/30/xaml-behaviors-open-source-and-on-uwp/



2. In the StarWarsMainView.xaml, add the following namespace declarations.

```
xmlns:Interactivity="using:Microsoft.Xaml.Interactivity"
xmlns:Interactions="using:Microsoft.Xaml.Interactions.Core"
```

3. Replace the Command binding of the Up-Button with the following XAML:

```
<Button x:Name="UpButton" Width="70"
        Margin="8,0,0,0"
        IsEnabled="{Binding CanRateUp, Mode=OneWay}"
        Content="Up">
        <Interactivity:Interaction.Behaviors>
            <Interactions:EventTriggerBehavior EventName="Click">
                <Interactions:InvokeCommandAction
                          Command="{Binding RateUpCommand}" />
            </Interactions:EventTriggerBehavior>
        </Interactivity:Interaction.Behaviors>
</Button>
```

4. Do the same for the DownButton

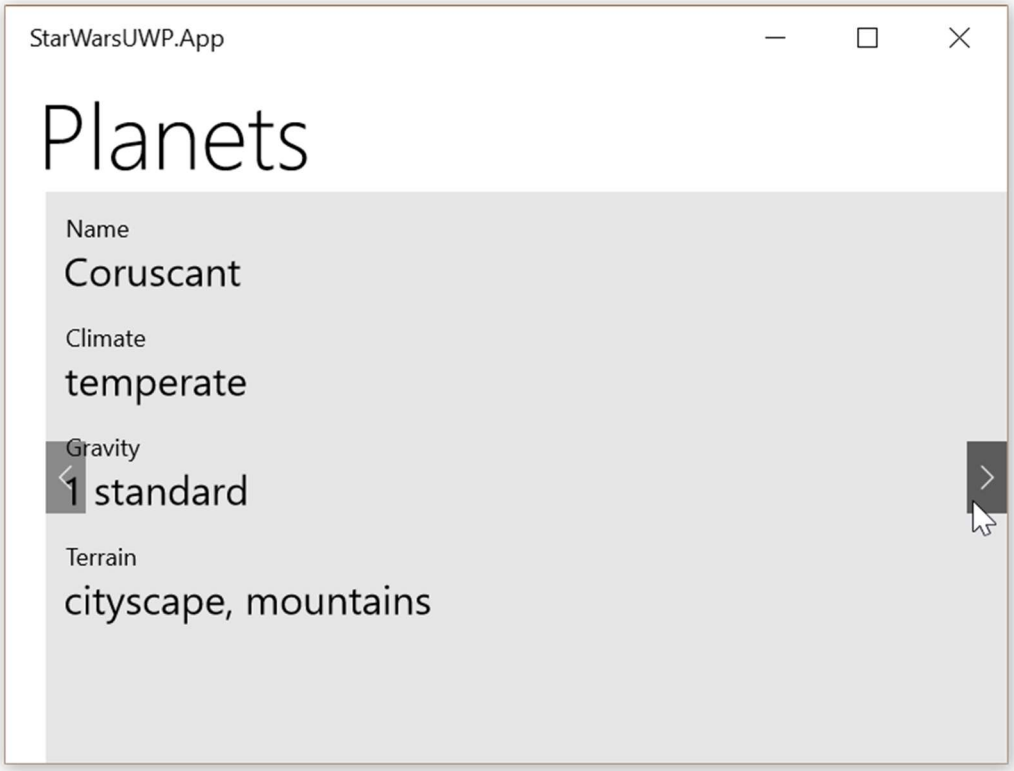
5. Now wire the Loaded-event of the page with a Command that loads the data. You have to remove this code from the ViewModel constructor and place it in a command.

Exercise 4: Navigation

In this exercise, we will introduce a Messenger and NavigationService to go to a new Page with extra information about the planets in the selected movie.

Steps:

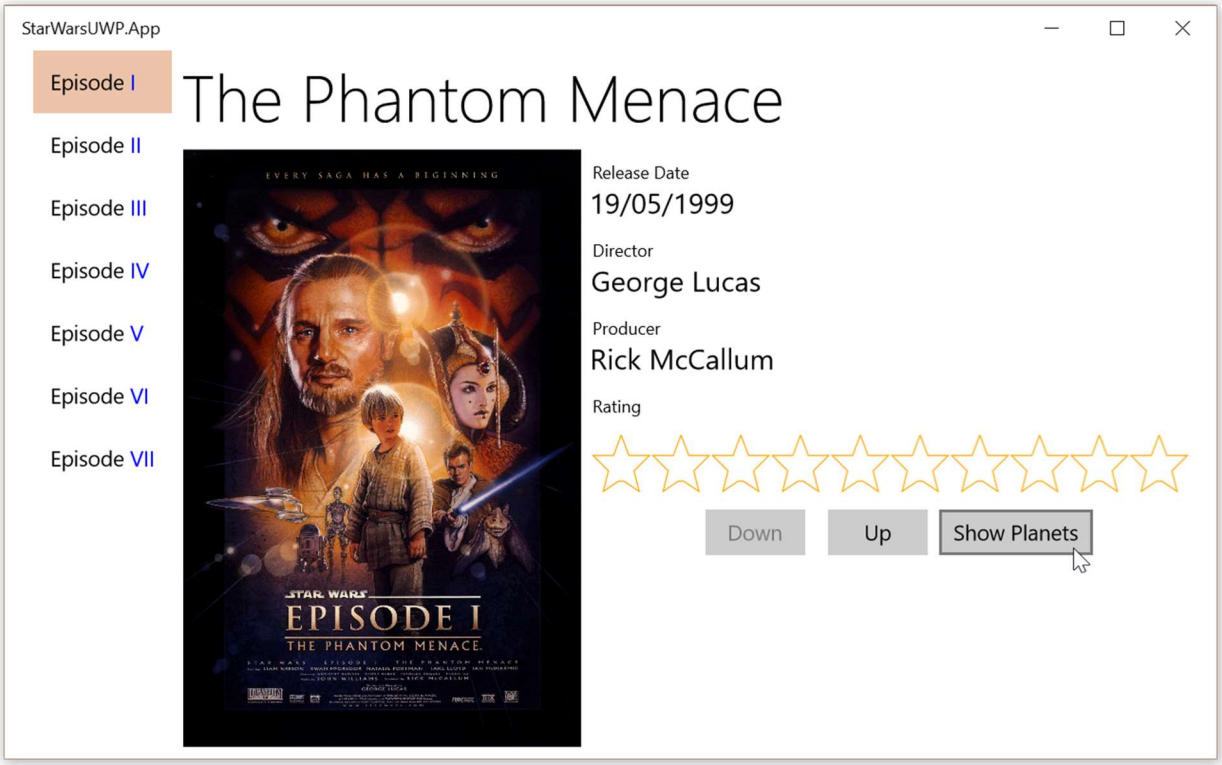
1. Write a PlanetsView in order to show a list of planets using a FlipView control (see screenshot).



2. Write a PlanetsViewModel with a collection of planets and a SelectedMovie property.
3. In the Utility folder, copy the Messenger class from the Pluralsight course (MVVM). Now add a command to the StarWarsMoviesViewModel called ShowPlanetsCommand. This command will send the SelectedPlanet via the Messenger to the PlanetsViewModel.

   > Note: you don't navigate from a viewmodel to a view! You simply send messages to viewmodels.

4. Now place a button on the first screen, so that you can navigate to the PlanetsView:



5. Finally write a NavigationService, with the following code. Place this in a folder called Services:

```
using StarWarsUWP.App.View;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace StarWarsUWP.App.Services
{
    public class NavigationService
    {
        public void NavigateTo(string key)
        {
            if (key == "Planets")
            {
                PlanetsView view = new PlanetsView();
                Frame rootFrame = Window.Current.Content as Frame;
                rootFrame.Navigate(typeof(PlanetsView));
            }
        }
    }
}
```

> Note: we use some sort of key to decouple the view name from the viewmodel. In a framework like MVVM Light, you have to configure this service, e.g. using an IoC container.

6. Now connect everything into a working application!

> More info on navigation in UWP
> https://msdn.microsoft.com/en-us/library/windows/apps/mt465735.aspx

Exercise 5: Testing

Use the techniques of the final module of the Pluralsight course on MVVM to test your ViewModels. You will need to introduce some interfaces (e.g. for the NavigationService) as well as a ViewModelLocator.

> Note: for UWP, referencing the ViewModelLocator from XAML may not work. Use the code-behind file as explained below.

In App.xaml.cs create a property for the ViewModelLocator:

```
private ViewModelLocator vmlocator;

/// <summary>
/// Initializes the singleton application object.  This is the first line of
authored code
/// executed, and as such is the logical equivalent of main() or WinMain().
/// </summary>
public App()
{
    Microsoft.ApplicationInsights.WindowsAppInitializer.InitializeAsync(
        Microsoft.ApplicationInsights.WindowsCollectors.Metadata |
        Microsoft.ApplicationInsights.WindowsCollectors.Session);
    this.InitializeComponent();
    this.Suspending += OnSuspending;

    vmlocator = new ViewModelLocator();
}

public ViewModelLocator ViewModelLocator
{
    get { return vmlocator; }
}
```

Now in de View codebehind, lookup the view model:

```
public StarWarsMainView()
{
    this.InitializeComponent();
    var vm = (App.Current as App).ViewModelLocator;
    this.DataContext = vm.StarWarsMoviesViewModel;
}
```

Exercise 6: MVVM Light framework (optional)

Rewrite the entire application using an MVVM framework. A framework provides for the helper classes you wrote yourself (CustomCommand, INotifyPropertyChanged impl, Messenger, NavigationServices, etc.) and provides for production quality implementation. In general it is recommended to use a good framework like MVVM Light instead of writing everything yourself.

Useful resources to study:

http://blog.galasoft.ch/posts/2015/03/using-mvvmlight-with-windows-10-universal-applications/

http://stackoverflow.com/questions/32186295/mvvm-light-cant-work-in-windows-10-universal-app

http://www.spikie.be/blog/post/2014/06/30/.aspx

http://www.spikie.be/blog/category/mvvm-light.aspx

http://www.spikie.be/blog/post/2013/04/12/10-things-you-might-have-missed-about-MVVM-Light.aspx

https://app.pluralsight.com/library/courses/mvvm-light-toolkit-fundamentals/table-of-contents (5de module)

http://wp.qmatteoq.com/the-mvvm-pattern-the-practice/