Diary of a Windows developer
Windows, mobile and much more

The MVVM pattern – The practice

Posted on January 4, 2016 by qmatteoq

Let's continue our journey to learn the MVVM pattern, applied to Universal Windows apps development. After we've learned the basic concepts in the previous post, now it's time to start writing some code. As already mentioned in the previous post, we're going to leverage MVVM Light as toolkit to help us implementing the pattern: since it's the most flexible and simple to use, it will be easier to understand and apply the basic concepts we've learned so far.

The project

The goal of the MVVM pattern is to help developers to organize their code in a better way. As such, the first step is to define the structure of a project in a way that, also from a logical point of view, it can follow the principles of the pattern. Consequently, usually the first thing to do is to create a set of folders where to place our classes and files, like:

- **Models,** where to store all our basic entities.
- **ViewModels**, where to store the classes that will connect the Model with the View.
- **Views**, where to store the Views, which are the XAML pages in case of a Universal Windows app.

In a typical MVVM project, however, you will end up to have more folders: one of the assets, one for the services, one for the helper classes, etc.

The second step is to install in the project the library we have chosen to help us implementing the pattern. In our case, we chose MVVM Light, so we can leverage NuGet to install it. We will find two different versions:

- The complete package (http://www.nuget.org/packages/MvvmLight/) which, other than the libraries, it adds alse some documentation and a series of default classes (like a ViewModel, a ViewModelLocator, etc.)
- Just the libraries (http://www.nuget.org/packages/MvvmLightLibs/) which will add just the required DLLs.

For the moment, my suggestion is to use the second package: this way, we will do everything from scratch, giving us the chance to better learn the basic concepts. In the future, you're free to use the first package if you want to save some time.

As a sample project, we're going to create a very simple Universal Windows app that it's likely you would be able to develop in no time using code behind: a Hello World app with a TextBox, where to insert your name, and a Button that, when it's pressed, will display a hello message followed by your name.

Linking the View and ViewModel

The first step is to identify the three components of our application: Model, View and ViewModel.

- The **model** isn't necessary in this application, since we don't need to manipulate any entity.
- The **view** will be made by a single page, which will show to the user the TextBox where to insert his name and the Button to show the message.

- The **ViewModel** will be a class, which will handle the interaction in the view: it will retrieve the name filled by the user, it will compose the hello message and it will display it in the page.

Let's start to add the various components: let's create a **Views** folder and add, inside it, the only page of our application. As default behavior, when you create a new Universal Windows app the template will create a default page for you, called **MainPage.xaml**. You can move it to the **Views** folder or delete it and create a new one in Solution Explorer by right clicking on the folder and choosing **Add –> New item –> Blank page.**

Now let's create the **ViewModel** that will be connected to this page. As explained in the previous post, the ViewModel is just a simple class: create a **ViewModels** folder, right click on it and choose **Add –> New Item –> Class**.

Now we need to connect the View and the ViewModel, by leveraging the **DataContext** property. In the previous post, in fact, we've learned that the ViewModel class is set as **DataContext** of the page; this way, all the controls in the XAML page will be able to access to all the properties and commands declared in the ViewModel. There are various ways to achieve our goal, let's see them.

Declare the ViewModels as resources

Let's say that you have created a ViewModel called **MainViewModel.** You'll be able to declare it as a global resource in the **App.xaml** file this way:

```
<Application
    x:Class="MVVMSample.MVVMLight.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:viewModel="using:MVVMSample.MVVMLight.ViewModel">

    <Application.Resources>
        <viewModel:MainViewModel x:Key="MainViewModel" />
    </Application.Resources>

</Application>
```

The first step is to declare, as attribute of the **Application** class, the namespace which contains the ViewModel (in this sample, it's **MVVMSample.MVVMLight.ViewModel**). Then, in the **Resources** collection of the **Application** class, we declare a new resource which type is **MainViewModel** and we associate it to a key with the same name. Now we can use this key and the **StaticResource** keyword to connect the **DataContext** of the page to the the resource, like in the following sample:

```
<Page
    x:Class="MVVMSample.MVVMLight.Views.MainPage"
    DataContext="{Binding Source={StaticResource MainViewModel}}"
    mc:Ignorable="d">

    <!-- your page content -->

</Page>
```

The ViewModelLocator approach

Another frequently used approach is to leverage a class called **ViewModelLocator**, which has the responsability of dispatching the ViewModels to the various pages. Instead of registering all the ViewModels as global resources of the application, like in the previous approach, we register just the **ViewModelLocator**. All the ViewModels will be exposed, as properties, by the locator, which will be leveraged by the **DataContext** property of the page.

This is a sample definition of a **ViewModelLocator** class:

```
public class ViewModelLocator
{
    public ViewModelLocator()
    {
    }

    public MainViewModel Main
    {
        get
        {
            return new MainViewModel();
        }
    }
}
```

Or, as an alternative, you can simplify the code by leveraging one of the new C# 6 .0 features:

```
public class ViewModelLocator
{
    public ViewModelLocator()
    {
    }

    public MainViewModel Main => new MainViewModel();
}
```

After you've added the **ViewModelLocator** class as global resource, you'll be able to use it to connect the **Main** property to the **DataContext** property of the page, like in the following sample:

```
<Page
    x:Class="MVVMSample.MVVMLight.Views.MainPage"
    DataContext="{Binding Source={StaticResource Locator}, Path=Main}"
    mc:Ignorable="d">

    <!-- your page content -->

</Page>
```

The syntax is very similar to the one we've seen with the first approach; the main difference is that, since the **ViewModelLocator** class can expose multiple properties, we need to specify with the **Path** attribute which one we want to use.

The ViewModelLocator's approach adds a new class to maintain, but it gives you the flexibilty to handle the ViewModel's creation in case we need to pass some parameters to the class' constructor. In the next post, when we'll introduce the dependeny injection's concept, it will be easier for you to understand the advantages in using the **ViewModelLocator** approach.

Let's setup the ViewModel

No matter which approach we decided to use in the previous step, now we have a View (the page that will show the form to the user) connected to a ViewModel (the class that will handle the user's interactions).

Let's start to populate the ViewModel and to define the properties that we need to reach our goal. In the previous post we've learned that ViewModels need to leverage the **INotifyPropertyChanged** interface; otherwise, every time we change the value of a property in the ViewModel, the View won't be able to detect it and the user won't see any change in the user interface.

To make easier to implement this interface, MVVM Light offers a base class which we can leverage in our ViewModels using inheritance, like in the following sample:

```
public class MainViewModel : ViewModelBase
```

```
{

}
```

This class gives us access to a method called **Set()**, which we can use when we define our propeties to dispatch the notifications to the user interface when the value changes. Let's see a sample related to our scenario. Our ViewModel has to be able to collect the name that the user has filled into a **TextBox** control in the View. Consequently, we need a property in our ViewModel to store this value. Here is how it looks like thanks to the MVVM Light support:

```
private string _name;

public string Name
{
    get
    {
        return _name;
    }

    set
    {
        Set(ref _name, value);
    }
}
```

The code is very similar to the one we've seen in the previous post when we introduced the concept of the **INotifyPropertyChanged** interface. The only difference is that, thanks to the **Set()** method, we can achieve two goals at the same time: storing the value in the **Name** property and dispatching a notification to the binding channel that the value has changed.

Now that we have learned how to create properties, let's create another one to store the message that we will display to the user after he has pressed the button.

```
private string _message;

public string Message
{
    get
    {
        return _message;
    }

    set
    {
        Set(ref _message, value);
    }
}
```

In the end, we need to handle the interaction with the user. When he presses the **Button** in the view, we have to display a hello message. From a logical point of view, this means:

1. Retrieving the value of the **Name** property.
2. Leveraging the string interpolation APIs to prepare the message (something like "Hello Matteo").
3. Assign the result to the **Message** property.

In the previous post we've learned that, in the MVVM pattern, you use **commands** to handle the user interactions in a ViewModel. As such, we're going to use another one of the classes offered by MVVM Light, which is **RelayCommand**. Thanks to this class, instead of having to create a new class that implements the **ICommand** interface for each command, we can just declare a new one with the following code:

```
private RelayCommand _sayHello;
```

```
public RelayCommand SayHello
{
    get
    {
        if (_sayHello == null)
        {
            _sayHello = new RelayCommand(() =>
            {
                Message = $"Hello {Name}";
            });
        }

        return _sayHello;
    }
}
```

When you create a new **RelayCommand** object you have to pass, as parameter, an **Action**, which defines the code that you want to execute when the command is invoked. The previous sample declares an **Action** using an anonymous method. It means that, instead of defining a new method in the class with a specific name, we define it inline in the property definition, without assigning a name.

When the command is invoked, we use the new C# 6.0 feature to perform string interpolation to get the value of the property **Name** and add the prefix "Hello". The result is stored into the **Message** property.

Create the view

Now that the ViewModel is ready, we can move on and create the View. The previous step should have helped you to understand one of the biggest benefits of the MVVM pattern: we've been able to define the ViewModel to handle the logic and the user interaction without writing a single line of code in the XAML file. With the code behind approach, it would have been impossible; for example, if we wanted to retrieve the name filled in the **TextBox** property, we should have first added the control in the page and assign to it a name using the **x:Name** property, so that we would have been able to access to it from code behind. Or if we wanted to define the method to execute when the button is pressed, we would have needed to add the **Button** control in the page and subscribe to the **Click** event.

From a user interface point of view, the XAML we need to write for our application is more or less the same we would have created for a code behind app. The UI, in fact, doesn't have any connection with the logic, so the way we create the layout doesn't change when we use the MVVM pattern. The main difference is that, in a MVVM app, we will use binding a lot more, since it's the way we can connect the controls with the properties in the ViewModel. Here is how our View looks like:

```
<Page
    x:Class="MVVMSample.MVVMLight.Views.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    DataContext="{Binding Source={StaticResource Locator}, Path=Main}"
    mc:Ignorable="d">

    <Grid>
        <StackPanel Margin="12, 30, 0, 0">
            <StackPanel Orientation="Horizontal" Margin="0, 0, 0, 30">
                <TextBox Text="{Binding Path=Name, Mode=TwoWay, UpdateSourceTrigg
                <Button Command="{Binding Path=SayHello}" Content="Click me!" />
            </StackPanel>

            <TextBlock Text="{Binding Path=Message}" Style="{StaticResource Heade
        </StackPanel>
    </Grid>
```

```
20
21  </Page>
```

We have added to the page three controls:

1. A **TextBox**, where the user can fill his name. We have connected it to the **Name** property of the ViewModel. There are two features to higlight:
   A. We have set the **Mode** attribute to **TwoWay**. It's required since, for this scenario, we don't just need that when the property in the ViewModel changes the control displays the updated value, but also the opposite. When the user fills some text in the control, we need to store its value in the ViewModel.
   B. We have set the **UpdateSourceTrigger** attribute to **PropertyChanged**. This way, we make sure that, every time the text changes (which means every time the user adds or remove a char in the TextBox), the **Name** property will automatically update to store the changed value. Without this attribute, the value of the property would be updated only when the **TextBox** control loses the focus.
2. A **Button**, that the user will click to see the hello message. We have connected it to the **SayHello** command in the ViewModel.
3. A **TextBlock**, where the user will see the hello message. We have connected it to the **Message** property of the ViewModel.

If we have done everything properly and we launch the app, we should see the application behaving as we described at the beginning of this post: after filling your name and pressing the button, you will see the hello message.

Let's improve the user interaction

Our application has a flaw: if the user presses the button without writing anything in the **TextBox**, he would see just the "Hello" prefix. We want to avoid this behaviro by disabling the button if the **TextBox** control is empty. To achieve our goal we can leverage one of the features offered by the **ICommand** interface: we can define when a command should be enabled or not. For this scenario the **RelayCommand** class allows a second parameter during the initialization, which is a function that returns a boolean value. Let's see the sample:

```
1   private RelayCommand _sayHello;
2
3   public RelayCommand SayHello
4   {
5       get
6       {
7           if (_sayHello == null)
8           {
9               _sayHello = new RelayCommand(() =>
10              {
11                  Message = $"Hello {Name}";
12              },
13
14              () => !string.IsNullOrEmpty(Name));
15          }
16
17          return _sayHello;
18      }
19  }
```

We've changed the initialization of the **SayHello** command to add, as a second parameter, a boolean value: specifically, we check if the **Name** property is null or empty (in this case, we return **false**, otherwise **true**). Thanks to this change, now the **Button** control connected to this command will be automatically disabled (also from a visual point of view) if the **Name** property is empty. However, there's a catch: if we try the application as it

is, we would notice that the **Button** will be disabled by default when the app starts. It's the expected behavior, since when the app is launched the **TextBox** control is emtpy by default. However, if we start writing something in the **TextBox**, the **Button** will continue to be disabled. The reason is that the ViewModel isn't able to automatically determine when the execution of the **SayHello** command needs to be evaluated again. We need to do it manually every time we do something that may change the value of the boolean function; in our case, it happens when we change the value of the **Name** property, so we need to change the property definition like in the following sample:

```
1  private string _name;
2
3  public string Name
4  {
5      get
6      {
7          return _name;
8      }
9
10     set
11     {
12         Set(ref _name, value);
13         SayHello.RaiseCanExecuteChanged();
14     }
15 }
```

Other than just using the **Set()** method to store the property and to send the notification to the user interface, we invoke the **RaiseCanExecuteChanged()** method of the **SayHello** command. This way, the boolean condition will be evaluated again: if the **Name** property contains some text, than the command will be enabled; otherwise, if it should become empty again, the command will be disabled.

In the next post

In this post we've finally started to apply with a real project the concepts we've learned in the first post and we have started to write our first application based on the MVVM pattern. In the next posts we'll see some advanced scenario, like dependency injection, messages or how to handle secondary events. Meanwhile, you can play with the sample app we've created in this post, which is published on my GitHub repository https://github.com/qmatteoq/UWP-MVVMSamples. Together with the sample created in this post, you will find also the same sample created using Caliburn Micro as MVVM framework instead of MVVM Light. Happy coding!

Introduction to MVVM – The series

1. Introduction
2. The practice
3. Dependency Injection
4. Advanced scenarios
5. Services, helpers and templates
6. Design time data

Related

The MVVM pattern – Introduction
December 28, 2015
In "Universal Apps"

Xamarin Forms for Windows Phone devs – Using the MVVM pattern
January 20, 2015

Prism and Universal Windows app – The basic concepts
December 1

In "wpdev" In "Universal Apps"

This entry was posted in Universal Apps, UWP, wpdev and tagged MVVM, Universal Windows Platform, Windows 10. Bookmark the permalink.

Diary of a Windows developer
Proudly powered by WordPress.