Diary of a Windows developer
Windows, mobile and much more

The MVVM pattern – Services, helpers and templates

Posted on January 14, 2016 by qmatteoq

In this last post of the series about MVVM we're going to introduce some concepts and libraries that can make your life easier when you develop a Universal Windows app leveraging the MVVM pattern.

Services, services and services

In one of the previous posts we have created a sample app to display a list of news retrieved from a RSS feed. While developing the app we introduced the concept of service: a class that takes care of performing some operations and passing the results to the ViewModel. Services can be useful also to reach another important goal of the MVVM pattern: avoiding writing platform specific code directly in the ViewModel, to make it easier to share them with other platforms or applications. As usual, I prefer to explain concepts with real examples, so let's start with a new one.

Let's say that you're developing an awesome application that needs to display a dialog to the user. By applying the knowledge we've learned in the previous posts, you'll probably end up to create a command like the following one:

```
private RelayCommand _showDialogCommand;

public RelayCommand ShowDialogCommand
{
    get
    {
        if (_showDialogCommand == null)
        {
            _showDialogCommand = new RelayCommand(async () =>
            {
                MessageDialog dialog = new MessageDialog("Hello world");
                await dialog.ShowAsync();
            });
        }

        return _showDialogCommand;
    }
}
```

The command shows the dialog using a specific API of the Universal Windows Platform, which is the **MessageDialog** class. Now let's say that your customer asks you to port this amazing app to another platform, like Android or iOS, using Xamarin, the cross-platform technology that allows to create apps for all the major mobile platforms using C# and the framework .NET. In this scenario, your ViewModel has a problem: you can't reuse it as it is in Android or iOS, because they use a different API to display a message dialog. Moving platform specific code in a service is the best way to solve this problem: the goal is to change our ViewModel so that it just describes the operation to do (displaying a dialog) without actually implementing it.

Let's start by creating an interface, which describes the operations to perform:

```
public interface IDialogService
{
    Task ShowDialogAsync(string message);
}
```

This interface will be implemented by a real class, which will be different for each platform. For example, the implementation for the Universal Windows Platform will be like the following one:

```
public class DialogService: IDialogService
{
    public async Task ShowDialogAsync(string message)
    {
        MessageDialog dialog = new MessageDialog(message);
        await dialog.ShowAsync();
    }
}
```

On Xamarin Android, instead, you will leverage the **AlertDialog** class, which is specific from Android:

```
public class DialogService : IDialogService
{
    public async Task ShowDialogAsync(string message)
    {
        AlertDialog.Builder alert = new AlertDialog.Builder(Application.Context);

        alert.SetTitle(message);

        alert.SetPositiveButton("Ok", (senderAlert, args) =>
        {

        });

        alert.SetNegativeButton("Cancel", (senderAlert, args) =>
        {
        });
        alert.Show();

        await Task.Yield();
    }
}
```

Now that we have moved the platform specific APIs in a service, we can leverage the dependency injection approach (which I've described in a previous post) to use, in the ViewModel, the interface instead of the real class. This way, our command will just describe the operation to perform, demanding to the **DialogService** class to effectively execute it. With this new approach, the ViewModel will add a dependency to the **IDialogService** class in the constructor, like in the following sample:

```
public class MainViewModel : ViewModelBase
{
    private readonly IDialogService _dialogService;

    public MainViewModel(IDialogService dialogService)
    {
        _dialogService = dialogService;
    }
}
```

Then we can change our command in the following way:

```
private RelayCommand _showDialogCommand;

public RelayCommand ShowDialogCommand
{
    get
    {
        if (_showDialogCommand == null)
        {
            _showDialogCommand = new RelayCommand(async () =>
            {
                await _dialogService.ShowDialogAsync("Hello world");
            });
```

```
            }

            return _showDialogCommand;
        }
    }
```

By using the dependency injection approach, the Android application will register in the container the **DialogService** implementation which uses the Android APIs; vice versa, the Windows 10 application will register, instead, the implementation which uses the UWP APIs. However,now our ViewModel can be shared as it is between the two versions of the application, without having to change it. We can move the ViewModel, for example, in a Portable Class Library, which we can share across the Windows, Xamarin Android, Xamarin iOS, WPF, etc. versions of the application.

To help developers in moving the platform specific code into services, there are many libraries which offer a set of services ready to be used in your applications. One of the best ones, which plays well with MVVM Light, is Cimbalino Toolkit (http://cimbalino.org/), which is specific for the Windows world. Other than many converters, behaviors and helpers classes, it includes a wide set of services to handle storage, settings, network access, etc. All the services are provided with their own interfaces, so that it's easy to use them with the dependency inection's approach.

If you want to know more about reusing your ViewModels on different platforms, I strongly suggest you to read this article from Laurent Bugnion, the creator of MVVM Light itself. The tutorial will help you to learn how you can reuse your binding knowledge also on platforms like Android and iOS, which doesn't support binding out of the box.

Implementing the INotifyPropertyChanged interface in an easy way

In the second post of the series we've learned that, to properly support the **INotifyPropertyChanged** interface, we need to change the way we define the properties in the ViewModel. We can't use the standard get / set syntax, but in the setter we need to call a method that dispatches the notification to the binding channel that the value has changed. This approach makes the code more "noisy", since the simple definition of a property requires many lines of code.

Please welcome **Fody**, a library that is able to change the code you wrote at build time. **Fody** offers many addons and one of them is called **Fody.PropertyChanged.** Its purpose is to automatically turn every standard property into a property that, under the hood, implements the **INotifyPropertyChanged** interface. All you have to do is to decorate your class (like a ViewModel) with the **[ImplementPropertyChanged]** attribute.

For example, the following code:

```
[ImplementPropertyChanged]
public class MainViewModel : ViewModelBase
{
    public string Message { get; set; }
}
```

is converted, during the compilation, into this:

```
public class MainViewModel: ViewModelBase, INotifyPropertyChanged
{
    private string _message;

    public string Message
    {
```

```
        get { return _message; }
        set
        {
            _message = value;
            OnPropertyChanged("Message)
        }
    }
}
```

This way, you can simplify the code you need to write in your ViewModel and make it less verbose. To use this special attribute, you have to:

1. Install the package called **Fody.PropertyChanged** from NuGet (https://www.nuget.org/packages/PropertyChanged.Fody/)
2. To properly work, Fody requires a special XML file in the root of the project, which describes which is the addon to apply at compile time. The name of the file is **FodyWeavers.xml** and the content shoudl look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<Weavers>
  <PropertyChanged />
</Weavers>
```

That's all!

MVVM and Template10

Template10 is a new template, specific for Universal Windows apps development, which has been created to make the developer's life easier, by providing a cleaner and simpler way to handle the initialization of an app, new controls, MVVM helpers, etc. Template10 is a great starting point also for MVVM apps, since it offers a set of classes that will help you to solve some of the platform specific challenges that may arise during the development, like handling the navigation or the page's lifecycle. I won't dig into this topic in this post, since I've already talked about it in another post on this blog. If you're planning to create a Universal Windows app with the MVVM pattern, I strongly suggest you to give it a read and evaluate it.

That's all folks!

We've reached the end of our learning path. I hope you found these series useful to understand the power of the MVVM pattern and that, after reading it, you will try to develop your next application using it. As usual, remember that you can find the samples used as a reference in these posts on GitHub: https://github.com/qmatteoq/UWP-MVVMSamples 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

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

The MVVM pattern – The practice
January 4, 2016
In "Universal Apps"

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

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