Diary of a Windows developer
Windows, mobile and much more

Template10: a new template to create Universal Windows apps – MVVM

Posted on October 5, 2015 by qmatteoq

The Model-View-ViewModel pattern (from now on, just MVVM) is, without any doubt, the most widely used pattern when it comes to develop XAML based applications. In this post, we won't learn all the inner details of the pattern: I've already talked about it many times in this blog, when I presented some of the most popular MVVM frameworks like Caliburn Micro and Prism. The goal of the MVVM pattern is to improve testability and maintainability of an application, since it helps the developer in reaching one of the key goals that drives any complex development project: keeping the different layers separated, so that the code that handles the user interface doesn't mix with the one that handles the business logic (like retrieving the data from the cloud or from a database).

The MVVM pattern is heavily used in the Universal Windows app world and, generally speaking, also in all the XAML based technologies, since it relies on the basic XAML feature like binding, dependency properties, the **INotifyPropertyChanged** interface, etc. When you use this pattern, you typically split the project into three components:

1. The **model**, which are the entities and the services that are used to handle the raw data of the application, without dependencies on how they are presented.
2. The **view**, which is the user interface presented to the user. In Universal Windows apps, the Views are identified with the XAML pages.
3. The **ViewModel**, which is the connection between the view and the model. ViewModel are simple classes that takes care of retrieving the data handled by the model and of preparing them to be presented by the View using binding.

The MVVM pattern gives to developers a lot of benefits but, however, it requires some time to be configured into a new project: the problem is that when you create a new Universal Windows app there are a lot of scenarios (navigation, the application's lifecycle, contracts, etc.) which are simple to implement when you use code-behind, but that need to be reviewed when you use the MVVM pattern, since most of the code is written in a ViewModel, which is an independent class. A good example of this problem is the page lifecycle management: typically, in a Universal Windows app you use the **OnNavigatedTo()** and **OnNavigatedFrom()** events to handle the navigation from one page to another. However, these events are exposed only by the code behind class, since it inherits from the **Page** class; you can't use them in a ViewModel, which doesn't have any dependency from it.

For this reason, many developers have created libraries and frameworks which makes the developer's life easier, by providing all the needed tools. The most popular ones are MVVM Light by Laurent Bugnion, Caliburn Micro by Rob Esieneberg and Prism, originally developed by the Microsoft Pattern & Practice division and now turned in to a community project. Every framework has its own pros and cons: MVVM Light is the simplest and most flexible one, but it lacks any helper to handle the typical scenarios of a Universal Windows app. Caliburn Micro and Prism, on the other side, are more complex to master and a little bit "rigid", but they offer many tools to handle common scenarios like navigation, page lifecycle, etc.

Template10 doesn't act as another MVVM framework (even if, as we're going to see, it offers some classes to implement it without having to rely on another library), but it helps developers to make the implementation of the MVVM pattern easier. Let's see in details which are these facilities: it's important to remember that what we have

described in the previous posts (bootstrapper, extended splash screen, the new controls, etc.) can be applied also on MVVM based projects.

The basic tools to implement the pattern

Even if Template10 doesn't behave as a complete alternative to the existing libraries, it offers any way the basic tools required to implement it in the proper way. You aren't forced to use them: you can choose, for example, to add MVVM Light to your project and to use its helpers in replacement of the ones that I'm going to describe.

The first important helper offered by Template10 is the **ViewModelBase** class, which is the base class which your ViewModels should inherit from. Other than offering a set of useful features that we're going to see later in this post, it offers the basic tools to implement the pattern. The most important one is the **Set()** method, which can be used to propagate the changes to the ViewModel properties to the user interface, by implementing the **INotifyPropertyChanged** interface. Here is how you can use it to define a property in the ViewModel:

```
public class MainViewModel : Template10.Mvvm.ViewModelBase
{
    public MainViewModel()
    {

    }

    private string _name;

    public string Name

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

The **Set()** method, invoked inside the property's setter, allows to send a notification to the view that the value has changed, other than just effectively changing the value of the property. This way, if in the View you have one or more controls connected to this property through binding, they will update their visual state to reflect the changes. For example, here is how you can connect a **TextBlock** control placed in the XAML page to the property we've just declared:

```
<TextBlock Text="{Binding Path=Name}" />
```

If you already have some experience with MVVM Light, this approach will be familiar to you: also this toolkit offers a method called **Set()**, which has the same purpose.

Another important feature that Template10 implements is the **DelegateCommand** class, which can be used to easily create commands. Which is the purpose of commands? When it comes to handle the user interaction in a traditional application created using the code-behind approach, you rely on **event handlers:** they're methods that are connected to an event that is raised by a control, like pressing a button or selecting an item from a list. These event handlers have a tight dependency with the view and, as such, they can be declared only in the code-behind class. In a MVVM application, instead, most of the logic is handled by the ViewModel and, consequently, we would need to handle the user interaction in a ViewModel. This is exactly the purpose of commands: a way to handle an event using a property, so that we can connect it to a control using binding. This way, we can "break" the dependency that traditional event handlers have with the view and we can declare commands in any class, including ViewModels.

A command is defined by:

1. The actions to perform when the command is invoked.
2. The condition that needs to be satisfied to enable the command. This feature is very useful, because it makes sure that the control connected to the command automatically changes his visual status based on the command status. For example, if a command is connected to a **Button** control and, during the execution, the command is disabled, also the button will be disabled: it will be displayed in grey and the user won't be able to tap on it.

The Windows Runtime offers a base class to implements commands, called **ICommand**: however, it's just an interface, the concrete implementation is up to the developer, who would need to create a class for every command. To reduce the footprint and speed up the development, Template10 offers a class called **DelegateCommand**, which implements this interface and that can be initialized simply by passing to the constructor the two required information. Here is an example:

```
private DelegateCommand _setNameCommand;

public DelegateCommand SetNameCommand
{
    get
    {
        if (_setNameCommand == null)
        {
            _setNameCommand = new DelegateCommand(() =>
            {
                Result = $"Hello {Name}";
            }, () => !string.IsNullOrEmpty(Name));

        }

        return _setNameCommand;

    }
}
```

The first parameter is an **Action** (in this case, it's defined using an anonymous method), which defines the operations to execute when the command is invoked. The second optional parameter, instead, is a function that should return a boolean value and that is used to determine if the command is enabled or not. In the sample, the command is enabled only in case the content of the **Name** property isn't empty.

Once you have defined a command, you can connect it to a XAML control using the property with the same name, which is exposed by all the controls that support the user interaction (like the **Button** control). The following sample shows how to use the **Command** property offered by the **Button** control to connect the **DelegateCommand** property we've previously defined.

```
<Button Content="Click me" Command="{Binding Path=SetNameCommand}" />
```

Also in this case, if you have previous experience with MVVM Light, the approach will be familiar: the toolkit, in fact, offers a class called **RelayCommand** for the same purpose and which works in the same way.

Navigation and page's lifecycle

We already mentioned this scenario multiple times in this post: one of the key scenarios to handle in a Universal Windows app is the page's lifecycle. Frequently, the navigation events are used to handle basic operations like data loading. As such, having access to these kind of events also in a ViewModel is critical, since usually all the logic is there.

For this purpose, the **ViewModelBase** class offers a set of additional features to make the developer's life easier. The first feature is the implementation of an interface called **INavigable**, which allows to access to the navigation

events directly in a ViewModel, like in the following sample:

```
public class DetailViewModel : ViewModelBase
{
    public override void OnNavigatedTo(object parameter, NavigationMode mode, IDi
    {
        if (parameter != null)
        {
            int id = (int) parameter;
            //load data
        }
    }

    public override Task OnNavigatedFromAsync(IDictionary<string, object> state,
    {
        return base.OnNavigatedFromAsync(state, suspending);
    }
}
```

As you can see, thanks to this base class you get a 1:1 mapping with the same methods that are available in the code-behind class. You get also useful information about the navigation, like parameters that have been passed from another page or the **NavigationMode**, which can be useful to distinguish the different navigation scenarios (for example, you may decide to load the application's data only when the **NavigationMode** parameter is set to **New**, which means that it's a new navigation).

Another navigation feature offered by the **ViewModelBase** class is the **NavigationService**, which is a helper that can be used to trigger the navigation from one page to another directly in the ViewModel. Again, this is a scenario that, without this helper, would be supported only in the code-behind class: navigation is handled, in fact, by the **Frame** class, which can be accessed only in code-behind. The **NavigationService** class is pretty straightforward to use, since it mimics the behavior of the **Frame** one: to navigate from one page to another you can use the **Navigate()** method, passing as information the page's type and, optionally, a parameter.

```
NavigationService.Navigate(typeof(DetailPage), person?.Id);
```

As you can see in the documentation, the **Navigate()** method accepts as parameter a generic **object**: however, it isn't a good practice to pass complex objects from one page to another, but it's better to stick with simple types (a string, a number, etc.). This way, we make sure not to break the state management system implemented by Template10 (that we're going to see later), which requires that parameters needs to be serializable.

Handle the page's state

One of the most important scenarios to handle in a Universal Windows app is the application's lifecycle. When an application isn't in foreground anymore, it's suspended: the process is frozen in memory (so that the state is preserved) but all the threads that can use CPU, battery or network are terminated. When the application is resumed, we don't have to do anything special: since the state was preserved in memory, the app will be restored exactly as it was. However, the operating system can decide to terminate a suspended application in case the resources are running low. In this case, since the process is completely killed, when the app is resumed it won't automatically be restored to the previous state. As developers, we need to handle this scenario: since the termination by the OS is completely transparent to the user, he will expect to find the application as he left it. It's our duty to properly save and restore the page's state. Let's say, for example, that our application includes a page with some fields that the user needs to fill before moving on (like a registration form). Once the application is suspended and then resumed, he would expect to find the data he already filled already there and not to fill the entire form from scratch.

When you use the MVVM pattern, handling this scenario introduces some challenges:

1. Since we don't know for sure if the suspended application will be terminated or not, we need to save the page's state every time a new navigation is performed. When you use the MVVM pattern without relying on other libraries (like Template10), this requirement is hard to handle because we don't have access to the navigation events in a ViewModel.
2. When the application is suspended, we need to save in the local storage the application's state, like the content of the registration form or the last visited page. Then, when the application is resumed, we need to understand if it was terminated and, consequently, restore the state we've previously saved. Unfortunately, the basic Windows 10 template doesn't offer any tool or API to make easier for the developer to support this scenario.

Template10 makes the whole process a lot easier. When we have seen the sample code about handling the page's lifecycle, you may have noticed that the **OnNavigatedTo()** and **OnNavigatedFrom()** events expose a parameter called **state**. It's a dictionary, which can contain a list of key-value pairs, that are automatically serialized and deserialized in the local storage when the app is suspended or resumed. When the application is suspended, you're going to use the **OnNavigatedFrom()** method to add in the dictionary all the information we need to recover the application's state. When the page is loaded, instead, in the **OnNavigatedTo()** event, we'll retrieve the previously saved info and we'll assign them to the proper properties in the ViewModel.

Let's see a real sample, by defining a page with a **TextBox** control. The user can insert any text in it, which is saved into a ViewModel property, like the following one

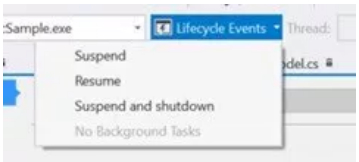
```
private string _name;

public string Name
{

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

The property is connected, as usual, to the **TextBox** control using binding:

```
<TextBlock Text="{Binding Path=Name, Mode=TwoWay}" />
```

You can notice that the binding has been configured using the **TwoWay** mode: this way, every time the user will write something in the **TextBox**, the **Name** property in the ViewModel will be automatically updated with the new value.

Now you can try to suspend the application and to open it back, by using the dropdown menu called **Lifecycle events** in Visual Studio, that allows to simulate the different stages of the application's lifecycle. It's important to remind, in fact, that when the debugger is attached, the app **is not suspended** when you put in background, like instead happens during the regular usage. If we write some text in the **TextBox** control, then we suspend the app and we open it back, we will find everything as we've left, since the app was just suspended and the process kept in memory. However, if we try to do the same but using the **Suspend and shutdown** option, you will notice that the content of the **TextBox** control will be gone: the process has been indeed terminated and, since we didn't save the content of the **TextBox**, the data was lost during the restore process.



To avoid this issue we can use the dictionary I've previously mentioned: in the **OnNavigatedFrom()** method of the ViewModel we can understand if the app is being suspended and, eventually, save in the collection the information we need to restore the state.

```
public class DetailViewModel : ViewModelBase
{

    private string _name;

    public string Name
    {
        get { return _name; }
        set { Set(ref _name, value); }
    }

    public override Task OnNavigatedFromAsync(IDictionary<string, object> state,
    {
        if (suspending)
        {
            state.Add("Name", Name);
        }

        return base.OnNavigatedFromAsync(state, suspending);
    }
}
```

By using the boolean parameter called **suspending** we understand if the app is being suspended: if that's the case, we can save into the dictionary called **state** the value of the **Name** property, by matching it to a key with the same name. This way, during the suspension, Template10 will automatically take care of serializing the content of this dictionary into the local storage in a text file. Now it will be easier for you to understand one of the key requirements of this implementation: we can store into the dictionary only simple data, that can be serialized into XML or JSON file. Saving, for example, an image stream into the dictionary wouldn't work.

The next step is to handle the page loading: in case the **state** dictionary contains some data in the **OnNavigatedTo()** method, then it means that the app is being resumed from a suspension and we need to retrieve the. Also in this case most of the work will be done by Template10: the bootstrapper will take care of loading the data that was previously serialized and it will automatically redirect the user to the last visited page of the application. This way, we can simply access to the **state** parameter in the **OnNavigatedTo()** method and use the data to populate back the properties in our ViewModel, like in the following sample:

```
public class DetailViewModel : ViewModelBase
{

    private string _name;

    public string Name
    {
        get { return _name; }
        set { Set(ref _name, value); }
    }

    public override void OnNavigatedTo(object parameter, NavigationMode mode, IDi
    {
        if (state.Any())
        {
            Name = state["Name"].ToString();
            state.Clear();
        }

    }

    public override Task OnNavigatedFromAsync(IDictionary<str
```



```
te,
```

```
23      {
24          if (suspending)
25          {
26              state.Add("Name", Name);
27          }
28
29          return base.OnNavigatedFromAsync(state, suspending);
30      }
31  }
```

Inside the **OnNavigatedTo()** method we check if there is any data inside the dictionary: only in this case we retrieve the value identified by the **Name** key and we assign it to the proper ViewModel's property. The next step is to clear the content of the collection (using the **Clear()** method), to avoid that the data is retrieved again also when we're simply navigating from one page to another.

Now we can try again to suspend the app using the **Suspend and shutdown** option offered by Visual Studio: now the application will behave as expected and the content of the **TextBox** control will still be there.

Important! It's crucial to remind that this approach has to be used to save the page's state, not the application data. Data generated by the user has to be saved as soon as possible, to minimize data loss in case of unexpected errors. During the suspension, you have between 5 and 10 seconds to complete all the pending operations, which may not be enough to save all the data handled by your application. The only data we need to save during suspension are the one that makes easier for the developer to create the illusion that the app has never been closed, even if it was silently terminated by the operating system.

Access to the UI thread

Sometimes you may have the requirement, inside a ViewModel, to perform some operations on background threads. In this case, if we try to directly access to one of the UI controls (for example, because we need to change the value of a property which is connected to a control in the View), we ill get an exception: the reason is that we're trying to access to the UI thread from a secondary one.

The Windows Runtime offers a class called **Dispatcher** to solve this problem: it's a sort of "mailman", which is able to dispatch some operations to the UI thread, regardless of the thread where the execution is happening. Also in this case, we're talking about a class which can be easily accessed from a code-behind class, but the same doesn't apply to an independent class like a ViewModel. Template10 includes a helper that makes easier to access to the dispatcher also from a third party class, like a ViewModel, by exposing a property called **Dispatcher** as part of the **ViewModelBase** infrastructure:

```
await Dispatcher.DispatchAsync(() =>
{
    //do something on the UI thread
});
```

Using it is very easy: just call the **DispatchAsync()** method of the **Dispatcher** class by passing, as parameter, an **Action** with the operations that you want to perform on the UI thread.

Wrapping up

Template10 can be really helpful when you're working with the MVVM pattern in a Universal Windows app: you can focus on getting things done, rather than trying to reinvent the wheel every time to handle the fundamentals of Universal Windows app development. However, Template10 is still growing and the number of services that will make easier to integrate Windows features into a ViewModel will continue to grow!

For the moment, we've finished our journey to discover the Template10 features, but I'm sure we'll have the chance to talk about it again in the future. Don't forget that you can find all the information about the project, in addition to many samples, on the official GitHub repository https://github.com/Windows-XAML/Template10

Related

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

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

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

This entry was posted in Universal Apps, wpdev and tagged Template10, Windows 10, wpdev. Bookmark the permalink.

5 Responses to *Template10: a new template to create Universal Windows apps – MVVM*

Diogenes *says:*
March 9, 2016 at 9:04 pm

Please make some new posts about Template10…

Reply

qmatteoq *says:*
March 9, 2016 at 9:22 pm

Hello Diogenes, which kind of content about Template10 would you like to see covered?

Reply

endguns *says:*
May 30, 2016 at 10:03 pm

Something about why a BackButton would not appear on one page but does on all the others. It is not the first page. I basically don't know where to go to find any help when Template10 does not act the way it is advertised. Maybe you could talk about troubleshooting Template10 or something like that.

Reply

qmatteoq *says:*
August 22, 2016 at 9:10 am

The official GitHub repository is a great place where to report issues: https://github.com/Windows-XAML/Template10

Reply