Diary of a Windows developer
Windows, mobile and much more

The MVVM pattern – Design time data

Posted on January 18, 2016 by qmatteoq

Ok, I've broken my promise. I've said that the previous post would have been the last one about MVVM, but I've changed my mind 😊 I realized, in fact, that I didn't talk about one of the most interesting pros of working with the MVVM pattern: design time data.

Design time data

In one of the posts of the series we've talked about dependency injection, which is an easy way to swap the implementation of a service used by ViewModel. There are various reasons to do that: to make refactoring easier, to replace the data source in a quick way, etc. There's another scenario for which dependency injection can be very useful: helping designers to define the user interface of the application. If this requirement is simple to satisfy when it comes to static data (like the header of a page), which are rendered in real time by the Visual Studio designer, things are a little bit more tricky with dynamic data. It's very likely that most of the data displayed in the application isn't static, but it's loaded when the application is running: from a REST service, from a local database, etc. By default, all these scenarios don't work when the XAML page is displayed in the designer, since the application isn't effectively running. To solve this problem, the XAML has introduced the concept of **design time data**: it's a way to define a data source that is displayed only when the XAML page is displayed in Visual Studio or in Blend in design mode, even if the app isn't effectively running.

The MVVM pattern makes easier to support this scenario, thanks to the separation of layers provided by the pattern and the dependency injection approach: it's enough to swap in the dependency container the service which provides the real data of the app with a fake one, which creates a set of static sample data.

However, compared to the sample we've seen in the post about the dependency injection, there are a couple of changes to do. Specifically, we need to detect when the XAML page is being displayed in the designer rather than at runtime and load the data from the correct data source. To do this, we use again one of the features offered by the MVVM Light toolkit, which is a property offered by the **ViewModelBase** class that tells us if a class is being used by the designer or by the running app.

Let's see in details the changes we need to do. We're going to use the same sample we've seen in the post about dependency injection, which can be found also on my GitHub repository https://github.com/qmatteoq/UWP-MVVMSamples. The app is very simple: it displays a list of news, retrieved from a RSS feed. In the old post we implemented an interface, called **IRssService**, which offers a method with the following signature:

```
public interface IRssService
{
    Task<List<FeedItem>> GetNews(string url);
}
```

Then, the interface is implemented by two classes: one called **RssService**, which provides the real data from a real RSS feed, and one called **FakeRssService**, which provides instead fake static data.

```
public class RssService : IRssService
{
    public async Task<List<FeedItem>> GetNews(string url)
    {
        HttpClient client = new HttpClient();
```

Privacy & Cookies Policy

```
        string result = await client.GetStringAsync(url);
        var xdoc = XDocument.Parse(result);
        return (from item in xdoc.Descendants("item")
                select new FeedItem
                {
                    Title = (string)item.Element("title"),
                    Description = (string)item.Element("description"),
                    Link = (string)item.Element("link"),
                    PublishDate = DateTime.Parse((string)item.Element("pubDate"))
                }).ToList();
    }
}

public class FakeRssService : IRssService
{
    public Task<List<FeedItem>> GetNews(string url)
    {
        List<FeedItem> items = new List<FeedItem>
        {
            new FeedItem
            {
                PublishDate = new DateTime(2015, 9, 3),
                Title = "Sample news 1"
            },
            new FeedItem
            {
                PublishDate = new DateTime(2015, 9, 4),
                Title = "Sample news 2"
            },
            new FeedItem
            {
                PublishDate = new DateTime(2015, 9, 5),
                Title = "Sample news 3"
            },
            new FeedItem
            {
                PublishDate = new DateTime(2015, 9, 6),
                Title = "Sample news 4"
            }
        };

        return Task.FromResult(items);
    }
}
```

Before starting to explore the changes we need to do in the application to support design data, I would like to highlight a possible solution to handle asynchronous operations. One of the challenges in creating fake data comes from the fact that, typically, real services use asynchronous methods (since they retrieve data from a source which may take some time to be processed). The **RssService** is a good example: since the **GetNews()** method is asynchronous, it has to return a **Task<T>** object, so that it can be properly called by our ViewModel using the await keyword. However, it's very unlikely that the fake service needs to use asynchronous methods: it just returns static data. The problem is that, since both services implement the same interface, we can't have one service that returns **Task** operations while the other one plain objects. A workaround, as you can see from the sample code, is to use the **FromResult()** method of the **Task** class. Its purpose is to to encapsulate into a **Task** object a simple response. In this case, since the **GetNews()** method returns a **Task<List<FeedItem>>** response, we create a fake **List<FeedItem>** collection and we pass it to the **Task.FromResult()** method. This way, even if the method isn't asynchronous, it will behave like if it is, so we can keep the same signature defined by the interface.

The ViewModelLocator

The first change we have to do is in the **ViewModelLocator.** In our sample app we have the following code, which registers into the dependency container the **IRssService** interface with the **RssService** implementation:

```
public class ViewModelLocator
```

```
{
    public ViewModelLocator()
    {
        ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);


        SimpleIoc.Default.Register<IRssService, RssService>();
        SimpleIoc.Default.Register<MainViewModel>();
    }

    public MainViewModel Main => ServiceLocator.Current.GetInstance<MainViewModel
}
```

We need to change the code so that, based on the way the app is being rendered, the proper service is used. We can ues the **IsInDesignModeStatic** property offered by the **ViewModelBase** class to detect if the app is running or if it's being rendered by the designer:

```
public class ViewModelLocator
{

    public ViewModelLocator()
    {
        ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);

        if (ViewModelBase.IsInDesignModeStatic)
        {
            SimpleIoc.Default.Register<IRssService, FakeRssService>();
        }
        else
        {
            SimpleIoc.Default.Register<IRssService, RssService>();
        }

        SimpleIoc.Default.Register<MainViewModel>();
    }

    public MainViewModel Main => ServiceLocator.Current.GetInstance<MainViewModel
}
```

In case the app is being rendered in the designer, we connect the **IRssService** interface with the **FakeRssService** class, which returns fake data. In case the app is running, instead, we connect the **IRssService** interface with the **RssService** class.

The ViewModel

To properly support design time data we need also to change a bit the ViewModel. The reason is that, when the app is rendered by the designer, isn't really running; the designer takes care of initializing all the required classes (like the ViewModelLocator or the different ViewModels), but it doesn't execute all the page events. As such, since typically the application loads the data leveraging events like **OnNavigatedTo()** or **Loaded**, we will never see them in the designer. Our sample app is a good example of this scenario: in our ViewModel we have a **RelayCommand** called **LoadCommand**, which takes care of retrieving the data from the **RssService**:

```
private RelayCommand _loadCommand;

public RelayCommand LoadCommand
{
    get
    {
        if (_loadCommand == null)
        {
```

Privacy & Cookies Policy

```
            _loadCommand = new RelayCommand(async () =>
            {
                List<FeedItem> items = await _rssService.GetNews("http://wp.qmatt
                News = new ObservableCollection<FeedItem>(items);
            });
        }

        return _loadCommand;
    }
}
```

By using the Behaviors SDK described in this post, we have connected this command to the **Loaded** event of the page:

```
<Page
    x:Class="MVVMLight.Advanced.Views.MainView"
    xmlns:interactivity="using:Microsoft.Xaml.Interactivity"
    xmlns:core="using:Microsoft.Xaml.Interactions.Core"
    DataContext="{Binding Source={StaticResource ViewModelLocator}, Path=Main}"
    mc:Ignorable="d">

    <interactivity:Interaction.Behaviors>
        <core:EventTriggerBehavior EventName="Loaded">
            <core:InvokeCommandAction Command="{Binding Path=LoadCommand}" />
        </core:EventTriggerBehavior>
    </interactivity:Interaction.Behaviors>

    <!-- page content here -->

</Page>
```

However, when the page is being rendered in the designer, the **LoadCommand** command is never invoked, since the **Loaded** event of the page is never launched. As such, we have to retrieve the data from our service also in the ViewModel's constructor which, instead, is executed by the designer when it creates an instance of our ViewModel. However, we need to do it only when the ViewModel is being rendered by the designer: when the app is running normally, it's correct to leave the data loading operation to the command. To achieve this goal we leverage the **IsInDesign** property, which is part of the **ViewModelBase** class we're already using as base class for our ViewModel:

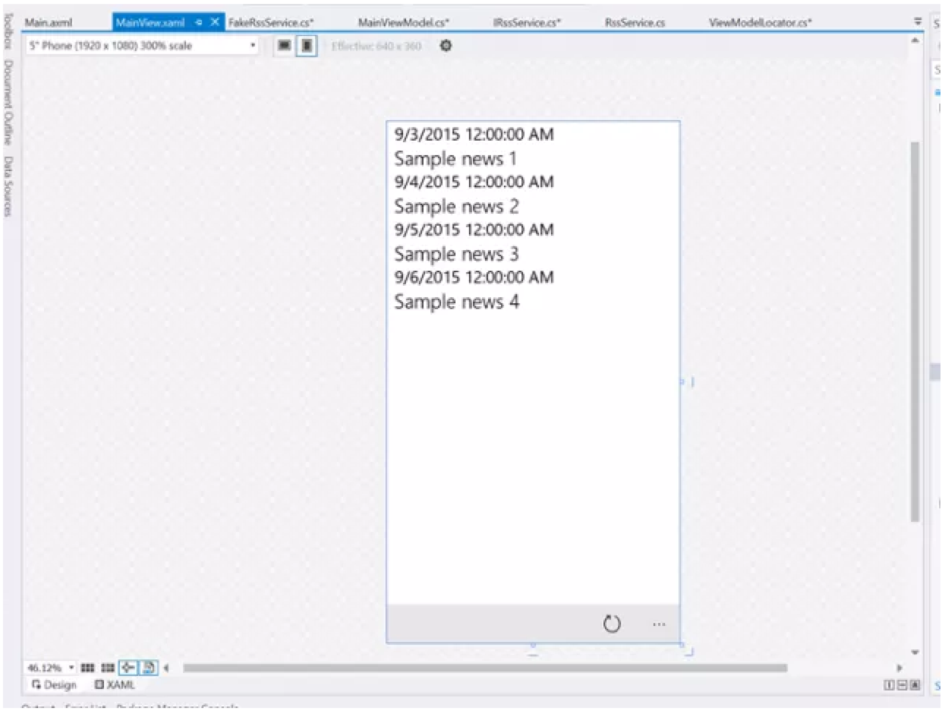
```
public MainViewModel(IRssService rssService)
{
    _rssService = rssService;
    if (IsInDesignMode)
    {
        var task = _rssService.GetNews("abc");
        task.Wait();
        List<FeedItem> items = task.Result;
        News = new ObservableCollection<FeedItem>(items);
    }
}
```

Only if the app is running in design mode, we retrieve the data from the service and we populate the **News** property, which is the collection connected to the **ListView** control in the page. Since the **GetNews()** method is asynchronous and you can't call asynchronous methods using the async / await pattern in the constructor, you need first to call the **Wait()** method on the **Task** and then access to the **Result** property to get the list of **FeedItem** objects. In a real application this approach would lead to a synchronous call, which would block the UI thread. However, since our **FakeRssService** isn't really asynchronous, it won't have any side effect.

This sample shows you also the reason why, in case you wanted to make things simpler, you can't just call the **GetNews()** method in the constructor also when the application is running: since we' can't properly use the async / await pattern, we would end up with unpredictable behaviors. As such, it's correct to continue calling the data loading methods in the page events that are triggered when the page is being loaded or navigated: since they're simple methods or event handlers, they can be used with the async and await keywords.

And we're done!

Now the job is done. If we launch the application, we should continue to normally see the data coming from the real RSS feed. However, if we open the **MainPage.xaml** page in the Visual Studio designer or in Blend, we should see something like this:



```
9/3/2015 12:00:00 AM
Sample news 1
9/4/2015 12:00:00 AM
Sample news 2
9/5/2015 12:00:00 AM
Sample news 3
9/6/2015 12:00:00 AM
Sample news 4
```

The designer has created an instance of our ViewModel, which received from the dependency container a **FakeRssService** instance. Since the ViewModel is running in design mode, it will excecute the code we wrote in the constructor, which will retrieve the fake data. Nice, isn't it? Thanks to this implementation, we can easily see how our collection of news will look like and, if it doesn't satisfy us, easily change the **DataTemplate** we have defined in the **ItemTemplate** property.

As usual, you can find the sample code used in this blog post on my GitHub repository: https://github.com/qmatteoq/UWP-MVVMSamples Specifically, you'll find it in the project called **MVVMLight.Advanced.** 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

Privacy & Cookies Policy

Share this:

Twitter | Facebook 1 | LinkedIn | Google | Pinterest | Tumblr | Pocket | Email | Print

Related

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

First steps with Caliburn Micro in Windows Phone 8 – The theory
February 4, 2013
In "Windows Phone"

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

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.