

Diary of a Windows developer

Windows, mobile and much more

The MVVM pattern – Dependency injection

Posted on [January 7, 2016](#) by [qmatteoq](#)

In the previous posts we've learned [the basic concepts of the MVVM pattern](#) and [how to apply them in a real application](#). In this post we'll introduce some more advanced concepts, which will be helpful to implement the MVVM pattern in real and complex projects. We will do it by building a sample application slightly more complex than the one we built in the previous post.

Let's create a feed reader

To understand what is the dependency injection mentioned in the title of the post we're going to create an application which will show to the user a list of news retrieved from a RSS feed. As we did [in the previous post](#), the first step is to split the application in the three main components:

- The **model**, which is the base entity that identifies a news from the feed.
- The **view**, which is the XAML page that will display the news using a **ListView** control.
- The **ViewModel**, which will take care of retrieving the news from the RSS feed and pass it to the View.

Since the goal of the MVVM pattern is to separate, as much as possible, the different layers of the application, it's a good practice to avoid including the logic required to parse the RSS feed directly in the ViewModel. A better approach is to handle this task to a specific class (we will call it **RssService**), which will take care of downloading the XML of the RSS feed, turn it into a collection of objects and return it to the ViewModel.

Let's start with the model and define a class, called **FeedItem**, which identifies a single news from the feed:

```
1 public class FeedItem
2 {
3
4     public string Title { get; set; }
5
6     public string Description { get; set; }
7
8     public string Link { get; set; }
9
10    public string Content { get; set; }
11
12    public DateTime PublishDate { get; set; }
13 }
```

Now we can create the service that, by leveraging LINQ to XML (a powerful language to manipulate XML files included into the framework .NET), is able to parse the RSS feed and convert it into a collection of **FeedItem** objects:

```
1 public class RssService: IRssService
2 {
3     public async Task<List<FeedItem>> GetNews(string url)
4     {
5         HttpClient client = new HttpClient();
6         string result = await client.GetStringAsync(url);
7         var xdoc = XDocument.Parse(result);
8
9         return (from item in xdoc.Descendants("item")
10                select new FeedItem
11                {
```

```

12         Title = (string)item.Element("title"),
13         Description = (string)item.Element("description"),
14         Link = (string)item.Element("link"),
15         PublishDate = DateTime.Parse((string)item.Element("pubDate"))
16     }).ToList();
17     }
18 }

```

The **RssService** class has an asynchronous method which uses the **HttpClient** class and its **GetStringAsync()** method to download the content of the RSS feed. Then, by using LINQ to XML, we convert the XML into a collection of **FeedItem** objects: every single node of the XML file (like **title**, **description**, **link**, etc.) is stored into a property of the **FeedItem** object.

Now we have a class that accepts the URL of a RSS feed in input and returns a list of **FeedItem** objects, that we can manipulate inside the ViewModel. Here is how the ViewModel of the page that displays the news looks like:

```

1  public class MainViewModel: ViewModelBase
2  {
3
4      private List<FeedItem> _news;
5
6      public List<FeedItem> News
7      {
8          get { return _news; }
9          set { Set(ref _news, value); }
10     }
11
12     private RelayCommand _loadCommand;
13
14     public RelayCommand LoadCommand
15     {
16
17         get
18         {
19             if (_loadCommand == null)
20             {
21                 _loadCommand = new RelayCommand(async () =>
22                 {
23                     RssService rssService = new RssService();
24                     List<FeedItem> items = await rssService.GetNews("http://wp.qm
25                     News = items;
26                 });
27             }
28
29             return _loadCommand;
30         }
31     }
32 }

```

We have reused the knowledge acquired in the previous posts to:

1. Define a property called **News**, which type is **List<FeedItem>**, where we're going to store the list of news, that we will display in the page.
2. We have defined a command, which we can connect to an event (like the button's click), that creates a new instance of the **RssService** class and call the **GetNews()** method.

In the end, we can create the XAML page:

```

1  <Page
2      x:Class="MVVMLight.Advanced.Views.MainView"
3      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4      DataContext="{Binding Source={StaticResource ViewModelLocator}, Path=Main}"
5      mc:Ignorable="d">

```

```

6
7     <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
8         <ListView ItemsSource="{Binding Path=News}">
9             <ListView.ItemTemplate>
10                 <DataTemplate>
11                     <StackPanel>
12                         <TextBlock Text="{Binding Path=PublishDate}" Style="{
13                         <TextBlock Text="{Binding Path=Title}" Style="{Static
14                     </StackPanel>
15                 </DataTemplate>
16             </ListView.ItemTemplate>
17         </ListView>
18     </Grid>
19
20     <Page.BottomAppBar>
21         <CommandBar>
22             <CommandBar.PrimaryCommands>
23                 <AppBarButton Icon="Refresh" Label="Load" Command="{Binding Path=
24             </CommandBar.PrimaryCommands>
25         </CommandBar>
26     </Page.BottomAppBar>
27 </Page>

```

In the page we added:

1. A **ListView** control, which **ItemsSource** property is connected, using binding, to the **News** property of the ViewModel. When the ViewModel uses the **RssService** method to load into the **News** property the list of news, the control will update its visual layout to display them. Specifically, we have defined an **ItemTemplate** which shows the publishing date of the news and the title.
2. An **AppBarButton** in the application bar, which is connected to the **LoadCommand** property. When the button is pressed, the ViewModel will download the news from the RSS feed using the **RssService** class.

Now that we have our new sample application up & running we can introduce some new concepts that will help us to make it better.

Dependency injection

Let's say that, at some point, we need to replace the **RssService** class with another one that, instead of retrieving the data from a real RSS feed, generates some fake data to display inside the app. There are many reasons to do it: for example, a designer may have the requirement to work on the user interface and he needs to simulate some "borderline" scenarios, like a news with a very long title. Or maybe our application handles more complex data and, instead of a RSS feed, we have a database or a cloud service which can be hard to configure just for test purposes.

In our sample application, it isn't that hard to accomplish this goal. We just need to create a new class (for example, **FakeRssService**) which returns a set of fake **FeedItem** objects, like the following one:

```

1 public class FakeRssService
2 {
3     public Task<List<FeedItem>> GetNews(string url)
4     {
5         List<FeedItem> items = new List<FeedItem>
6         {
7             new FeedItem
8             {
9                 PublishDate = new DateTime(2015, 9, 3),
10                 Title = "Sample news 1"
11             },
12             new FeedItem
13             {

```

```

14         PublishDate = new DateTime(2015, 9, 4),
15         Title = "Sample news 2"
16     },
17     new FeedItem
18     {
19         PublishDate = new DateTime(2015, 9, 5),
20         Title = "Sample news 3"
21     },
22     new FeedItem
23     {
24         PublishDate = new DateTime(2015, 9, 6),
25         Title = "Sample news 4"
26     }
27 };
28
29 return Task.FromResult(items);
30 }
31 }

```

Then we just need to change the code executed when the **LoadCommand** is executed to use this new class as replacement of the old one:

```

1 private RelayCommand _loadCommand;
2
3 public RelayCommand LoadCommand
4 {
5
6     get
7     {
8         if (_loadCommand == null)
9         {
10             _loadCommand = new RelayCommand(async () =>
11             {
12                 FakeRssService rssService = new FakeRssService ();
13                 List<FeedItem> items = await rssService.GetNews("http://wp.qmatteo");
14                 News = new ObservableCollection<FeedItem>(items);
15             });
16         }
17
18         return _loadCommand;
19     }
20 }

```

Up to now, things are quite simple. However, let's say that we're working on a more complex application which, instead of having one single page that uses the **RssService** class, has 30 pages, which means that we have 30 ViewModels that are using the **RssService** class. Consequently, when we need to replace it with the fake one we've just created, we need to edit every single ViewModel and change the used class. You can easily imagine how this approach is time consuming and, most of all, makes easier to introduce errors. It's enough to forget to replace back the original class in one of the ViewModels when we have finished the testing phase to create a big issue.

Let's introduce the **dependency injection** concept, which is a way to handle this scenario in a better way. The problem I have just described arises from the fact that, in the previous approach, the **RssService** class (which we can consider it as a dependency of the ViewModel, since it can't work properly without it) was created at build time. With dependency injection, we change approach and we start to solving these dependencies at runtime, when the application is running.

This is possible thanks to a special class, called **container**, that we can consider like a box: inside it, we're going to register all the ViewModels and all the dependencies that are required by our application. When the application is running and we need a new instance of a ViewModel (simply because the user is navigating to a new page), we won't create it anymore manually, but we will ask to the container to do it for us. The container will check if it's

able to solve all the dependencies (which means that all the services used by the ViewModels are available) and, if the answer is yes, it will return to us a new instance of the ViewModel with all the dependencies already solved and ready to be used. Technically speaking, we say that the dependencies are “injected” into to the ViewModel: this is why this technique is called **dependency injection**.

Why does this approach solve the previous problem, which is avoid to change all the ViewModels if we need to change the implementation of a service? Because the connection between the ViewModel and the service is handled by the container, which is a single instance across the whole application. Every ViewModel won't have anymore to manually create a new instance of the **RssService** class, but it will be automatically included in the ViewModel's constructor. It's the container's job to create a new instance of the **RssService** class and pass it to the ViewModel, so that it can use it. When we need to replace the **RssService** class with the **FakeRssService** one, we just need to change the class registered in the container and, automatically, all the ViewModels will start to use the new one.

To properly support this approach, there are a couple of changes we need to make to our application. Let's see them.

Define an interface

The first problem in our code is that **RssService** is a class and, as such, we don't have a way to easily swap it with another one in the container. To do it, we need something that describes in an abstract way the **RssService** class and the operation that it can perform, so that we can leverage it both in the **RssService** and in the **FakeRssService** classes. This is what interfaces are for: their purpose is to describe a set of properties and methods which, in the end, will be implemented by a real class. The first step, consequently, is to create an interface which describes the operations performed by our RSS service. The same interface will be implemented both by the real service (the **RssService** class) and the fake one (the **FakeRssService** class). After we've done this modify, we'll be able to:

1. Reference the service in the ViewModel using the interface instead of the real class.
2. Register in the container the connection between the interface and the real class that we want to use. At runtime, the container will inject into the ViewModel the implementation we have chosen. If we want to swap it, we just need to register in the container another class for the same interface.

Let's start by creating an interface and call it **IRssService**:

```
1 public interface IRssService
2 {
3     Task<List<FeedItem>> GetNews(string url);
4 }
```

Our service, for the moment, exposes just one asynchronous operation: **GetNews()**, which takes as input the URL of the RSS feed and returns, as output, a collection of **FeedItem** objects.

Now we need to change our classes **RssService** and **FakeRssService** to implement this interface, like in the following example:

```
1 public class RssService : IRssService
2 {
3
4     public async Task<List<FeedItem>> GetNews(string url)
5     {
6         HttpClient client = new HttpClient();
7         string result = await client.GetStringAsync(url);
8         var xdoc = XDocument.Parse(result);
9         return (from item in xdoc.Descendants("item")
```

```

10         select new FeedItem
11         {
12             Title = (string)item.Element("title"),
13             Description = (string)item.Element("description"),
14             Link = (string)item.Element("link"),
15             PublishDate = DateTime.Parse((string)item.Element("pubDate"))
16         }.ToList();
17     }
18 }
19
20 public class FakeRssService : IRssService
21 {
22     public Task<List<FeedItem>> GetNews(string url)
23     {
24         List<FeedItem> items = new List<FeedItem>
25         {
26             new FeedItem
27             {
28                 PublishDate = new DateTime(2015, 9, 3),
29                 Title = "Sample news 1"
30             },
31             new FeedItem
32             {
33                 PublishDate = new DateTime(2015, 9, 4),
34                 Title = "Sample news 2"
35             },
36             new FeedItem
37             {
38                 PublishDate = new DateTime(2015, 9, 5),
39                 Title = "Sample news 3"
40             },
41             new FeedItem
42             {
43                 PublishDate = new DateTime(2015, 9, 6),
44                 Title = "Sample news 4"
45             }
46         };
47
48         return Task.FromResult(items);
49     }
50 }

```

As you can see, both services are using the same interface and they are implementing the same method: the only difference is that the **RssService** class is returning real data, while the **FakeRssService** one is manually creating a set of fake **FeedItem** objects.

The ViewModel

There are a couple of changes we need to apply to our ViewModel:

1. We don't need anymore to manually create a new instance of the **RssService** class in the command, but we need to add it as parameter in the ViewModel's constructor. We need to reference the **IRssService** interface, otherwise we won't be able to easily swap between its different implementations.
2. We need to change the **LoadCommand**, in order to use this new instance of the service.

Here is how our updated ViewModel looks like:

```

1 public class MainViewModel : ViewModelBase
2 {
3     private readonly IRssService _rssService;
4
5     public MainViewModel(IRssService rssService)
6     {
7         _rssService = rssService;
8     }
9 }

```

```

10     private ObservableCollection<FeedItem> _news;
11
12     public ObservableCollection<FeedItem> News
13     {
14         get { return _news; }
15         set { Set(ref _news, value); }
16     }
17
18     private RelayCommand _loadCommand;
19
20     public RelayCommand LoadCommand
21     {
22         get
23         {
24             if (_loadCommand == null)
25             {
26                 _loadCommand = new RelayCommand(async () =>
27                 {
28                     List<FeedItem> items = await _rssService.GetNews("http://wp.c
29                     News = new ObservableCollection<FeedItem>(items);
30                 }));
31             }
32
33             return _loadCommand;
34         }
35     }
36 }

```

The ViewModelLocator

The last step is also the most important one, which is creating the container and registering into it all the ViewModels with their dependencies. Typically this configuration is done when the app starts so, when we use MVVM Light and the ViewModelLocator approach, the best place where to do it is the **ViewModelLocator** itself, since it's the class that takes care of dispatching all the ViewModels.

Here is our new ViewModeloLocator implementation looks like:

```

1     public class ViewModelLocator
2     {
3
4         public ViewModelLocator()
5         {
6             ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);
7
8             SimpleIoc.Default.Register<IRssService, RssService>();
9             SimpleIoc.Default.Register<MainViewModel>();
10        }
11
12        public MainViewModel Main
13        {
14            get { return ServiceLocator.Current.GetInstance<MainViewModel>(); }
15        }
16    }

```

First of all, we need to highlight that this code is just a sample: there are plenty of available libraries to implement the dependency injection approach, like Ninject or LightInject. However, most of the MVVM toolkits and frameworks provide also an infrastructure to handle this scenario and MVVM Light makes no exception, by providing a simple container identified by the **SimpleIoc** class, which is set as default container of the app by using the **SetLocationProvider()** of the **ServiceLocator** class.

The next step is to register all the ViewModels and theri dependencies in the container. The **SimpleIoc** class offers the **Register()** method to achieve this goal, which can be used in two ways:

1. The **Register<T>()** version, which is used when we just need to a new instance of a class which isn't described by an interface. It's the case of our ViewModels since, in our context, we don't need a way to completely swap them.
2. The **Register<T, Y>()** version, which is used, instead, when we need to register a class that is described by an interface. In this case, we specify as T the interface and as Y the implementation we want to use.

In the end, we need to change the way the **ViewModelLocator** class defines the **Main** property: instead of manually creating a new **MainViewModel** object, we ask to the container to give us a new instance of the class, by using the **GetInstance<T>** method (where **T** is the type we need).

After we have done these changes, here is what we'll happen when we'll launch our application again:

1. The application will create a new instance of the **ViewModelLocator** class, which will start the configuration of the container. The **SimpleIoc** class will register the **MainViewModel** and the **RssService** classes (the second one, described by the **IRssService** interface).
2. The application will trigger the navigation to the main page of the app. Consequently, since the **DataContext** property of the **Page** is connected to the **Main** property of the **ViewModelLocator**, the locator will try to create a new instance of the ViewModel.
3. The container defined in the **ViewModelLocator** class will check if there's a class which type is **MainViewModel** registered. The answer is yes, however the class depends from a class that implements the **IRssService** interface. Consequently, the container will start another search and it will check if there's a class connected to the **IRssService** interface registered. Also in this case the answer is yes: the container will return a new instance of the **MainViewModel** class with the **RssService** implementation ready to be used.

If any of the previous conditions isn't satisfied (for example, the container doesn't find a class registered for the **IRssService** interface), we will get an exception, since the container wasn't able to resolve all the dependencies. Now that we have reached the end of the journey, you should be able to understand why the dependency injection is extremely useful for our scenario. As soon as we need to do some tests and we want to use the **FakeRssService** class in replacement of the **RssService** one, we just need to change one line of code in the **ViewModelLocator** class. Instead of:

```
1 | SimpleIoc.Default.Register<IRssService, RssService>();
```

we would need to do :

```
1 | SimpleIoc.Default.Register<IRssService, FakeRssService>();
```

Thanks to this new approach, it doesn't matter if our application has just one ViewModel or 50 ViewModels which are depending from the **IRssService** interface: automatically, the container will take care of injecting the proper real class to any of them.

This approach is extremely useful not just when you're doing tests, but also if you need to perform heavy refactoring. Let's say that, at some point, our application doesn't have to retrieve the news from a RSS feed anymore, but from a REST service published on the cloud. As long as our new service will follow the same structure defined by the **IRssService** interface (in our case, a **GetNews()** method which returns a collection of **FeedItem** objects) we won't have to change anything in the ViewModels. We just need to create a new class which implements the **IRssService** interface and register it into the container, in replacement of the **RssService** one.

In the next post

We're reaching the end of our learning path. In the next post we're going to see some other advances scenarios, like sending messages and handling secondary events. As usual, you can play with the sample project described in this post 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](#)

Share this:



Related

[The MVVM pattern – Services, helpers and templates](#)

January 14, 2016

In "Universal Apps"

[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"

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.