Working With Prism 4.0 (Hello World Sample with MVVM)

Posted by Abhishek Sur on Sunday, June 5, 2011
Labels: .NET, beyondrelational, C#, CodeProject, design pattern, Patterns, Prism, Unity, WPF, XAML 21 Comments

Modularity is one of the primary concern when working with a big projects. Most of us think of how we can implement our application that could be reusable across more than one applications. Patterns and Practices Team puts forward the notion of modularity with the help of Unity and Prism which most importantly focus on WPF and Silverlight applications. Being a WPF developer, it would be nice to take this apart and explain you a bit of how you can implement your application using Prism.

Before you begin, I must tell you, this is the most basic article that guides you step by step how you can write your first Prism based application and what are the advantages of building such kind of application. I will take this further in my next posts to make more concrete samples. So if you know the basics of how you can work with Prism, I would recommend you to read my next posts.

Also I assume you have some knowledge of Unity, WPF and MVVM to get you through with Prism.

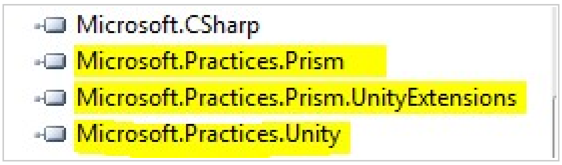
Where do I find Prism?

Prism is a framework introduced by Patterns and Practices Team which is available from here. After you install your bits, you will get a folder named Prism on your local drive. After you are done, lets start coding.

Starting Our Application

Lets create a WPF Application now using Prism. To do that,

1. *Start Visual Studio and Create WPF Application.*
2. *A MainWindow.xaml will be automatically created for you. Add some assemblies in the project you have just created.*



Microsoft.CSharp
Microsoft.Practices.Prism
Microsoft.Practices.Prism.UnityExtensions
Microsoft.Practices.Unity

3. *You will find these dlls inside Bin directory under Prism. Remember to choose Desktop for WPF application.*
4. *Once all the dlls successfully added we need to create a Shell first.*

What is a Shell ?

As Prism supports resolution of modules, you need a container where all the modules would appear. You can think of a Shell as your Window where all your UserControls can appear. Lets say I use MainWindow.xaml for our Shell.

But that is not all, to make your window act like a UnityContainer, you need to change the design like below :

```
<Window x:Class="SimplePrism.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx
/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx
/2006/xaml"
        xmlns:cal="http://www.codeplex.com/prism"
        Title="MainWindow" Height="350" Width="525">
    <ItemsControl Name="MainRegion"
cal:RegionManager.RegionName="MainRegion"/>
</Window>
```

You need to get rid of the Grid and specify an ItemsControl and specify the RegionName for the RegionManager which we will use later.

Note : You can customize this window according to your wish, and the design will be common to all the modules. ItemsControl will act as a Container to the Shell.

Registering Types

After you have defined your Shell, you need to register some of the interfaces that will be called automatically from the container. Rather than doing this, we will use some existing Bootstraper that will do these for us. The Bootstrapper actually runs the Bootstrapping sequence to Resolve the classes needed to be configured before working with UnityContainer. The interfaces that it needs to Register are IServiceLocator, IModuleManager, IRegionManager, IModuleInitializer, IEventAggregator etc. I will discuss them sometime later.

As we are going to use Unity for modularity, we use UnityBootstrapper abstract class.

Create a class in your project and derive it from UnityBootstrapper. We need to override the abstract method CreateShell at least to work with it. Lets do it now :

```
public class MyOwnBootStraper : UnityBootstrapper
{
    protected override DependencyObject CreateShell()
    {
        return this.Container.Resolve<MainWindow>();
    }

    protected override void InitializeShell()
    {
        base.InitializeShell();

        App.Current.MainWindow = (Window)this.Shell;
        App.Current.MainWindow.Show();
    }

    protected override void ConfigureModuleCatalog()
    {
        base.ConfigureModuleCatalog();

        ModuleCatalog moduleCatalog =
(ModuleCatalog)this.ModuleCatalog;
        //ToDo : Add modules that you need. You can also
use Configuration for this.
    }
}
```

You can see we pass MainWindow to resolve the Shell inside CreateShell where the MainWindow is the name of the Window in the project that acts as a container for the Unity. Other than that, we also need to ConfigureModuleCatalog. Here I didn't add any ModuleCatalog, and will do as we go on add a new Module for the application. And Finally the InitializeShell is used to Show the Window that is referred to the Shell (viz, MainWindow).

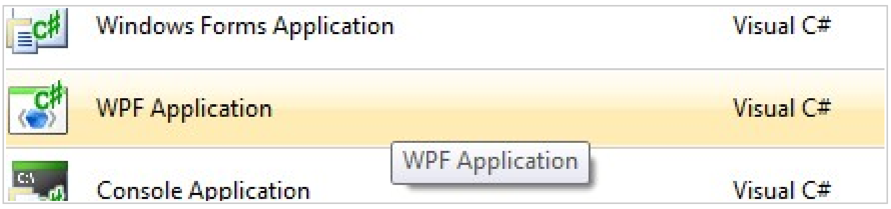
Finally from App.xaml we need to Run the Bootstrapper.

```
public partial class App : Application
    {
        protected override void
OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);
            MyOwnBootStraper bootstrapper = new
MyOwnBootStraper();
            bootstrapper.Run();
        }
    }
```

Hmm. Looks like we are done with the application. Lets try to run it. You will see the blank window appear on the screen.

Creating the Module

Now as you are done with the container, lets try to add a module which would be loaded into the Window. To do this, lets add a new project to the solution. You can add a Class Library if you wish, but in that case you need to add dlls that you need for Presentation layers. For the time being I am choosing another WPF application for the module.



We call it SimpleModule. For this project we need to add **Microsoft.Practices.Prism.dll.**

Delete the App.Xaml and MainWindow.Xaml and add a new UserControl for your application.

Before we start designing the module, lets add a ViewModel for the application. For that we use a ViewModelBase class which derives INotifyPropertyChanged event used to update the UI. The class is named as SimpleViewModel which looks like :

```
namespace SimpleModule.ViewModel
{
    public class SimpleViewModel : ViewModelBase
    {
        private string _message;
        public string Message
        {
            get { return this._message; }
            set
            {
                this.OnPropertyChanging("Message");
                this._message = value;
                this.OnPropertyChanging("Message");
//ensures the UI gets updated.
            }
        }

        private ICommand _SendToViewModel;
        public ICommand SendToViewModel
        {
            get
            {
                this._SendToViewModel =
this._SendToViewModel ?? new
DelegateCommand(this.OnSendToViewModel);
                return this._SendToViewModel;
            }
        }

        /// <summary>
        /// Called when Button SendToViewModel is
clicked
        /// </summary>
        private void OnSendToViewModel()
        {
            string message = this.Message;

            if (!string.IsNullOrWhiteSpace(message))
            {
                MessageBox.Show("You passed the message
" + message, "Welcome!", MessageBoxButton.OK);
                this.Message = string.Empty;
            }
        }
    }
}
```

The class is very basic. It has one Property called Message which takes a Message from the UI. As mentioned, for any property that you use from UI you need to call INotifyPropertyChanged event to update the UI. The ICommand interface actually fired when the Button on the page is clicked. hence it shows up a messagebox and clears the text.

Let me put the ViewModelBase as well here to make it more clear :

```
public class ViewModelBase : INotifyPropertyChanged,
INotifyPropertyChanging
    {

        public ViewModelBase()
        {

        }




        //Here you need to have your DataLayer Manager
object or BizLogic to access database

        //private DataManager _manager;
        //public DataManager Manager
        //{
        //    get
        //    {
        //        this._manager = this._manager ?? new
DataManager();
        //        return this._manager;
        //    }
        //}

        #region INotifyPropertyChanged Members

        private void OnPropertyChanged(string
propertyName)
        {
            if(this.PropertyChanged != null)
                this.PropertyChanged(this, new
PropertyChangedEventArgs(propertyName);
        }
        public event PropertyChangedEventHandler
PropertyChanged;

        #endregion

        #region INotifyPropertyChanging Members

        public void OnPropertyChanging(string
propertyName)
        {
            if(this.PropertyChanging != null)
                this.PropertyChanging(this, new
PropertyChangingEventArgs(propertyName);
        }

        public event PropertyChangingEventHandler
PropertyChanging;

        #endregion
    }
```

Even though INotifyPropertyChanging is not actually needed in this scenario, but it is better to use it together.

Note : From your ViewModelBase you should connect to Database Layer or BizLogic Layer if you have any so that data is available here. We have also added a class DelegateCommand which acts as a CommandPattern derived from ICommand interface for Button click. It just fires the Action that we pass.

Once you are done with the ViewModels, lets add some xaml for our UserControl so that it could be loaded on the Container we have created.

```
<UserControl x:Class="SimpleModuleLibrary.SampleView"
             xmlns="http://schemas.microsoft.com/winfx
/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx
/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org
/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com
/expression/blend/2008"
             xmlns:model="clr-
namespace:SimpleModuleLibrary.ViewModel"
             mc:Ignorable="d">
    <UserControl.Resources>
        <model:SimpleViewModel x:Key="SimpleModel" />
    </UserControl.Resources>
    <Grid DataContext="{StaticResource SimpleModel}">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <TextBlock Text="Write anything : " />
        <TextBox Text="{Binding Message, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"
                 Grid.Row="0"
                 Grid.Column="1"/>

        <Button Grid.Row="1" Grid.ColumnSpan="2"
                Content="Send Your Message"
                Command="{Binding SendToViewModel}">
</Button>
    </Grid>
</UserControl>
```

Hmm, So basically I create one textbox and one Button, which invokes the respective PropertyChanged on the ViewModel.

Now after you have created the View in the Library, you need to register the View as Module. You should do it once for each Module (which may contain more than one view in turn).

```
public class SimpleModule : IModule
{
    private readonly IRegionViewRegistry
regionViewRegistry;

    public SimpleModule(IRegionViewRegistry registry)
    {
        this.regionViewRegistry = registry;
    }

    public void Initialize()
    {

        //We need to register our Module in MainRegion.

regionViewRegistry.RegisterViewWithRegion("MainRegion",
typeof(SampleView));
    }
}
```

The IModule interface will tell the application to Register the View as a Module which could be loaded into MainRegion. If you have multiple Regions available, you can do it from here.

Hey, I think the application is ready to be deployed. One thing that remains, is to add the Module in the ModuleCatalog. Lets do it now :

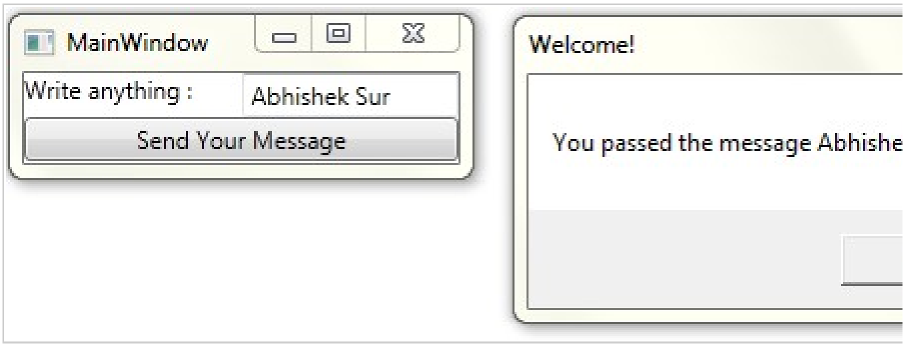
```
protected override void ConfigureModuleCatalog()
{
    base.ConfigureModuleCatalog();

    ModuleCatalog moduleCatalog =
(ModuleCatalog)this.ModuleCatalog;
    //ToDo : Add modules that you need. You can also use
Configuration for this.


moduleCatalog.AddModule(typeof(SimpleModuleLibrary.Simpl
eModule));
}
```

Basically the ModuleCatalog will hold all the modules that you need to register for the whole application.

Please Note : Before you run the application change the Build Action of XAML to Page from ApplicationDefination.

Running the App :

So we have finished building our first application. Run the application you will see something like this :



You passed the message Abhishe

The MessageBox appears on the screen which shows the message you pass through the TextBox.

Please Note : Rather than Registering the Modules directly from the Bootstrapper, you can also use Configuration to do the same. We will try to cover it on my next post.

You can Download Sample Code from here .

Read more about Prism.

I am sorry for not getting too deep in this post, you will find them in some of my next posts.

Thank you for reading.

SHARE

Like 9 people like this. Be the first of your friends.

Read Disclaimer Notice