



**CODE
PROJECT**
For those who code

Dynamic Modules with Prism and the Modern UI for WPF Toolkit



Rubén Hinojosa Chapel, 23 Mar 2016

[MS-PL](#)



5.00 (4 votes)

Prototype for a plugin architecture based on the Prism Library and the Modern UI for WPF (MUI) toolkit



Download source - 892.9 KB

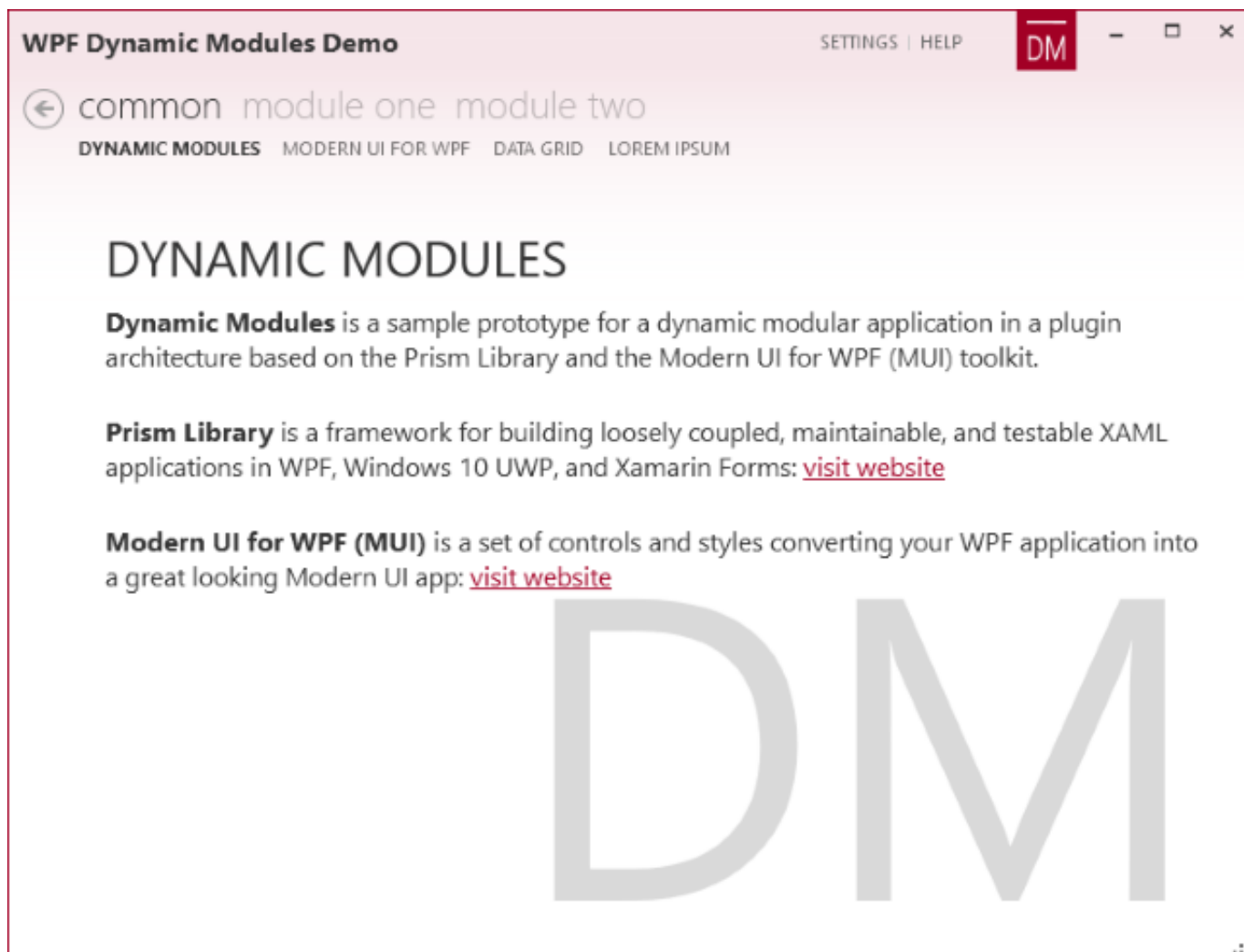


View source on GitHub

Introduction

A little while ago, a colleague of mine told me about a problem he had to solve. A customer asked him to develop a desktop application that presents a different set of features in accordance to which of the enterprise offices is running the software. On the one hand, I remembered how to do that from a past project where I was involved. On the other hand, there is an open source project for creating WPF applications with a modern look & feel, which I am following since a couple of years because I think it's really great.

I wondered if there is a way to solve the problem by using Prism and the open source MUI library for creating a plugin architecture, and came up with a prototype solution which I am presenting here.



Dynamic Modules is a sample prototype for a WPF modular application based on the [Prism Library](#) and the [Modern UI for WPF \(MUI\)](#) toolkit. It is a proof of concept for creating metro-styled, modern UI WPF applications in a plugin architecture.

Background and Requirements

The article assumes that the reader has at least a basic background on Windows Presentation Foundation (WPF), the Prism Library and the Unity IoC container. Visual Studio 2015 is needed to compile the project.

Architecture

The central ideas for the proposed plugin architecture are:

- Put into a directory the desired project modules (or put them all and run a filter on loading time).
- Dynamically load the project modules from the *modules* folder.
- Each module exposes an entry point for an option in the main menu.
- Dynamically build the main menu from the loaded modules.
- The first option in the main menu is fixed and common for every user.
- A core module with shared services, repositories, DTOs, data model definitions, etc. is statically loaded. It can be referenced by any solution project.

Dynamic modules are copied to a directory as part of a post-build step. These modules are not referenced in the startup project and are discovered by examining the assemblies in a directory. The module projects have the following post-build step in order to copy themselves into that directory:

```
xcopy "$(TargetDir)$(TargetFileName)" "$(TargetDir)modules\" /y
```

The solution is built into "..\bin\" folder.

Understanding the Code

If you check out the source code for `MainWindow.xaml` in the [MUI demo project](#), you will see how the main menu is statically built:

```
<mui:ModernWindow.MenuLinkGroups>
  <mui:LinkGroup DisplayName="Welcome">
    <mui:LinkGroup.Links>
      <mui:Link DisplayName="Introduction"
        Source="/Pages/Introduction.xaml" />
    </mui:LinkGroup.Links>
  </mui:LinkGroup>
  <mui:LinkGroup DisplayName="Layout">
    <mui:LinkGroup.Links>
      <mui:Link DisplayName="Wireframe"
        Source="/Pages/LayoutWireframe.xaml" />
      <mui:Link DisplayName="Basic"
        Source="/Pages/LayoutBasic.xaml" />
      <mui:Link DisplayName="Split"
        Source="/Pages/LayoutSplit.xaml" />
      <mui:Link DisplayName="List"
        Source="/Pages/LayoutList.xaml" />
      <mui:Link DisplayName="Tab"
        Source="/Pages/LayoutTab.xaml" />
    </mui:LinkGroup.Links>
  </mui:LinkGroup>
  <mui:LinkGroup DisplayName="Controls">
    <mui:LinkGroup.Links>
      <mui:Link DisplayName="Styles"
        Source="/Pages/ControlsStyles.xaml" />
      <mui:Link DisplayName="Modern controls"
        Source="/Pages/ControlsModern.xaml" />
    </mui:LinkGroup.Links>
  </mui:LinkGroup>
  ...
  ...
  ...
</mui:ModernWindow.MenuLinkGroups>
</mui:ModernWindow>
```

The main menu of the main window is a dependency property of the `ModernWindow` class, named `MenuLinkGroups`. It returns an instance of the `LinkGroupCollection` class, which inherits from `ObservableCollection<LinkGroup>`. That is, the main menu is an observable collection of link groups. Each `LinkGroup` represents an entry point to the menu. So, if each dynamic module has a way to export a `LinkGroup` instance, all we have to do is add it to the observable collection of link groups. That way comes under the form of an interface contract.

```
public interface ILinkGroupService
{
    LinkGroup GetLinkGroup();
}
```

The core module defines the `ILinkGroupService` interface. It states that if a module wants to plug an option onto the main menu, it can do so by implementing the `GetLinkGroup()` method, which in fact returns a `LinkGroup` instance. An implementation of the `ILinkGroupService` interface and the `GetLinkGroup()` method will look like:

```
public class LinkGroupService : ILinkGroupService
{
    public LinkGroup GetLinkGroup()
    {
        LinkGroup linkGroup = new LinkGroup
        {
```

```

        DisplayName = "Module One"
    };

    linkGroup.Links.Add(new Link
    {
        DisplayName = "Module One",
        Source = new Uri
        ($"/DM.ModuleOne;component/Views/{nameof(MainView)}.xaml", UriKind.Relative)
    });

    return linkGroup;
}
}

```

Now, we need to be able to dynamically load the modules, create an instance of the **ILinkGroupService** interface for each module, and insert the exported option in the main menu. That is the function of the code implemented in the **ConfigureModuleCatalog()** method of the **Bootstrapper** class.

```

protected override IModuleCatalog CreateModuleCatalog()
{
    return new DirectoryModuleCatalog() { ModulePath = MODULES_PATH };
}

protected override void ConfigureModuleCatalog()
{
    var directoryCatalog = (DirectoryModuleCatalog)ModuleCatalog;
    directoryCatalog.Initialize();

    linkGroupCollection = new LinkGroupCollection();
    var typeFilter = new TypeFilter(InterfaceFilter);

    foreach (var module in directoryCatalog.Items)
    {
        var mi = (ModuleInfo)module;
        var asm = Assembly.LoadFrom(mi.Ref);

        foreach (Type t in asm.GetTypes())
        {
            var myInterfaces = t.FindInterfaces(typeFilter,
            typeof(ILinkGroupService).ToString());

            if (myInterfaces.Length > 0)
            {
                // We found the type that implements the ILinkGroupService interface
                var linkGroupService = (ILinkGroupService)asm.CreateInstance(t.FullName);
                var linkGroup = linkGroupService.GetLinkGroup();
                linkGroupCollection.Add(linkGroup);
            }
        }
    }

    var moduleCatalog = (ModuleCatalog)ModuleCatalog;
    moduleCatalog.AddModule(typeof(Core.CoreModule));
}

```

We first create **Bootstrapper.ModuleCatalog** as a **DirectoryModuleCatalog** and initialize the module catalog. Then iterate over the discovered modules. For each one, look for a type that implements the **ILinkGroupService** interface. If such a type is found, then create an instance and call its **GetLinkGroup()** method. The returned **LinkGroup** instance is then inserted into a collection which is passed to the Shell when it is created:

```

protected override DependencyObject CreateShell()
{
    Shell shell = Container.Resolve<shell>();

    if (linkGroupCollection != null)
    {
        shell.AddLinkGroups(linkGroupCollection);
    }
}

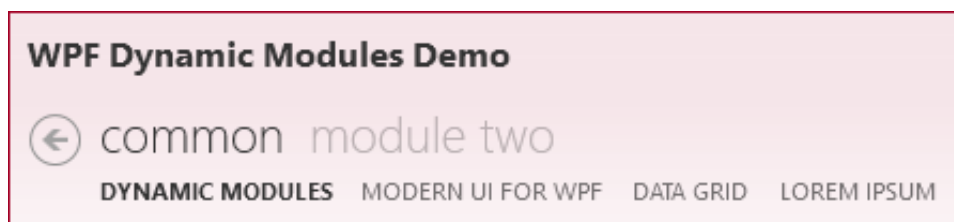
```

```
}  
  
    return shell;  
}
```

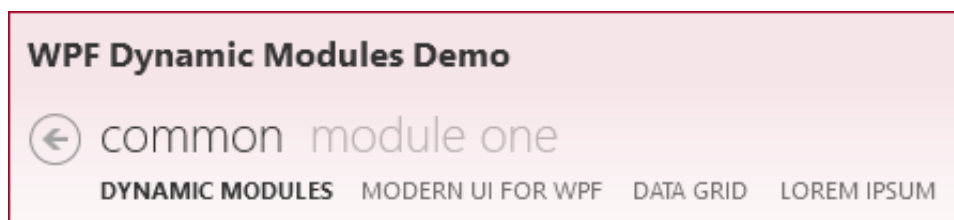
The `Shell.AddLinkGroups()` method is defined as:

```
public void AddLinkGroups(LinkGroupCollection linkGroupCollection)  
{  
    CreateMenuLinkGroup();  
  
    foreach (LinkGroup linkGroup in linkGroupCollection)  
    {  
        this.MenuLinkGroups.Add(linkGroup);  
    }  
}
```

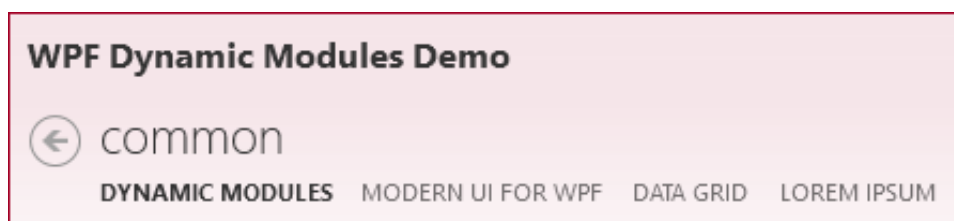
Where the `CreateMenuLinkGroup()` method creates the `static` common option of the main menu, and the `foreach` loop creates the dynamic ones. And that is all, folk. If some module, for instance `ModuleOne`, is removed from the `modules` folder, the main menu will look like:



Alternatively, if `ModuleTwo` is removed, the main menu will look like:



And obviously, if there is no module in the `modules` folder, only the static common option is accessible:



Conclusion

Prism Library offers a set of resources for creating modularized WPF applications. The Modern UI for WPF (MUI) toolkit offers a set of resources for creating nice and good looking UI for WPF applications. This article presents a way to mix both worlds for creating a plugin architecture. A related subject on authorization, that is, dynamically load the modules according to the user name or his role, so that user can only access authorized areas or features of the application, is out of the scope of the prototype project.

Surely, there is an alternative or even better approaches to doing such a mix so, please, feel free to comment and leave your ideas, suggestions and opinions. They are welcome!

Links of Interest

You may find complementary information at:

- [Modern UI for WPF \(MUI\) Demo project](#)
- [Modern UI for WPF Templates \(Visual Studio Extension\)](#)
- [Prism Samples WPF / Modularity With Unity](#)
- [On Prism's ViewModelLocator convention*](#)

*Currently, there is no need to "implement" the **IView** interface in the view's code-behind.

History

- 22nd March, 2016: Version 1.0 - Initial article submitted

License

This article, along with any associated source code and files, is licensed under [The Microsoft Public License \(Ms-PL\)](#)

Share

About the Author



Rubén Hinojosa Chapel

 Software Developer (Senior)
Spain 

Microsoft Development Consultant. XAML Developer. Collaborates with companies from different sectors in the development or maintenance of their computer systems, helping them in identifying their needs and proposing technical solutions that provide value and solve their problems.

You may also be interested in...



[Prism for Silverlight/MEF in Easy Samples. Part 3 - Communication between the](#)



[Java for Bluetooth LE applications](#)

Modules



Prism WPF standalone and interaction modules in VS2012



WPF: Showing you how to use PRISM in a very unlike PRISM way



The Past, Present, and Future of IoT



Make your website mobile friendly using Bootstrap

Comments and Discussions



1 message has been posted for this article Visit <http://www.codeproject.com/Articles/1087362/Dynamic-Modules-with-Prism-and-the-Modern-UI-for-W> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Terms of Use](#) | Mobile Web02 | 2.8.160311.1 | Last Updated 23 Mar 2016



Seleccionar idioma ▼

Article Copyright 2016 by Rubén Hinojosa Chapel
Everything else Copyright © [CodeProject](#), 1999-2016