PRISM 4.0 TRAINING KIT

Hands-On Lab

Modularity

Lab version: 1.0.0

Last updated: 11/19/2010

* 1. 

# Contents

[Overview 3](#_Toc277956283)

[Exercise 1: Creating and Loading a Module 4](#_Toc277956284)

[Task 1 - Creating a Module 4](#_Toc277956285)

[Task 2 - Loading a module 5](#_Toc277956286)

[Exercise 2: Loading a module from a separate project 7](#_Toc277956287)

[Task 1 - Creating and Loading a Module 7](#_Toc277956288)

[Exercise 3: Loading Modules Remotely 11](#_Toc277956289)

[Task 1 - Populating the ModulesCatalog through XAML 11](#_Toc277956290)

[Task 2 - Loading ModuleA Remotely 13](#_Toc277956291)

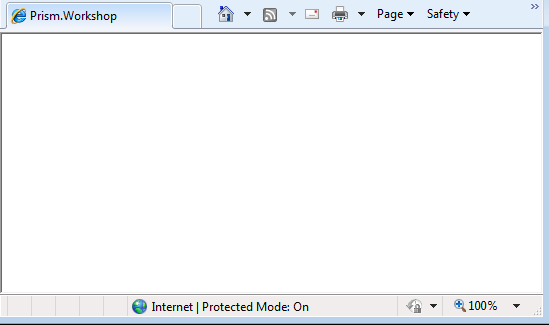
Overview

A modular application is an application that is divided into a set of functional units (named modules) that can be integrated into a larger application. A client module encapsulates a portion of the application's overall functionality and typically represents a set of related concerns. It can include a collection of related components, such as application features, including user interface and business logic, or pieces of application infrastructure, such as application-level services for logging or authenticating users. Modules are independent of one another but can communicate with each other in a loosely coupled fashion. Modular applications can make it easier for you to develop, test, deploy, and extend your application.

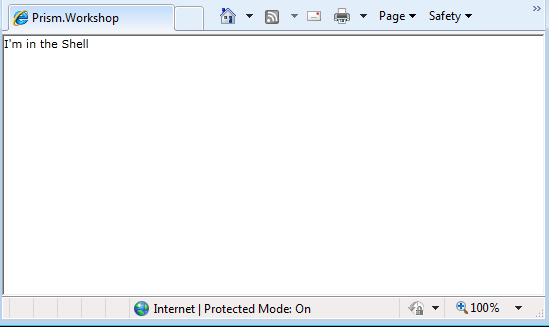
Exercise 1: Creating and Loading a Module

* 1. This exercise will guide you through the creation and loading of a module.

Task 1 - Creating a Module

* 1. Open the **Prism**.**Workshop** solution, located in the **Exercise 1\Begin** folder.
  2. Create a new class, named **ModuleA**, in the **Shell** project. This class should implement the **IModule** interface. This is shown in the following code.
     1. C#
     2. using Microsoft.Practices.Prism.Modularity;
     3. namespace Prism.Workshop
     4. {
     5. public class ModuleA : IModule
     6. {
     7. public void Initialize()
     8. {
     9. }
     10. }
     11. }
  3. Add **ModuleA** to the **ModulesCatalog** via code. To do this, override the **CreateModuleCatalog** template method in the **Bootstrapper** class and return a new instance of the **ModuleCatalog** class. Finally add the module using the **AddModule** method, passing the type of the module as a parameter, as seen in the following code.
     1. C#
     2. protected override IModulecatalog CreateModuleCatalog()
     3. {
     4. return new ModuleCatalog().AddModule(typeof(ModuleA));
     5. }
  4. Compile and run the solution. The Shell should be empty.
     1. Application loaded
     2. 
     3. **Note:** After providing a valid **ModuleCatalog** instance, the Initialize method in **ModuleA** should get called. You can verify that this is happening by placing a breakpoint anywhere in that method.

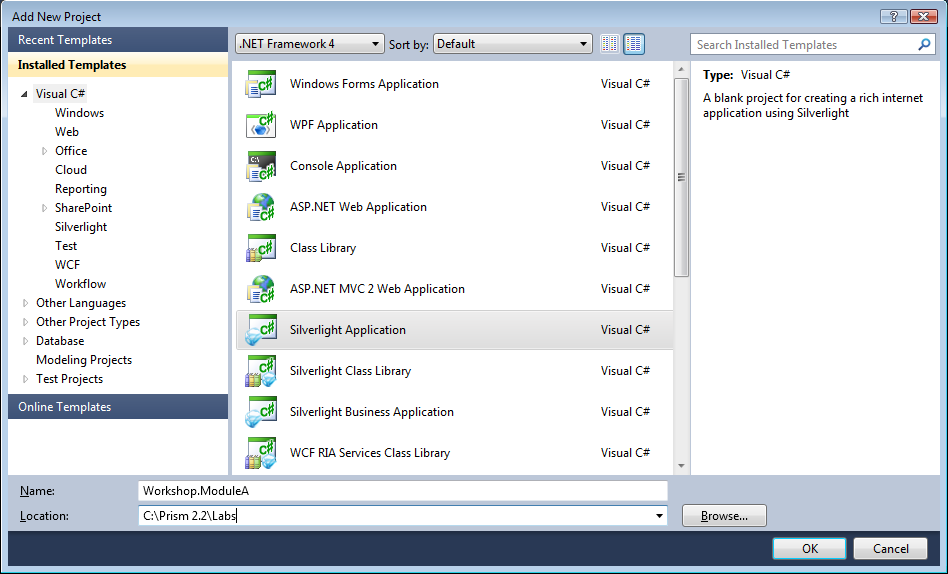
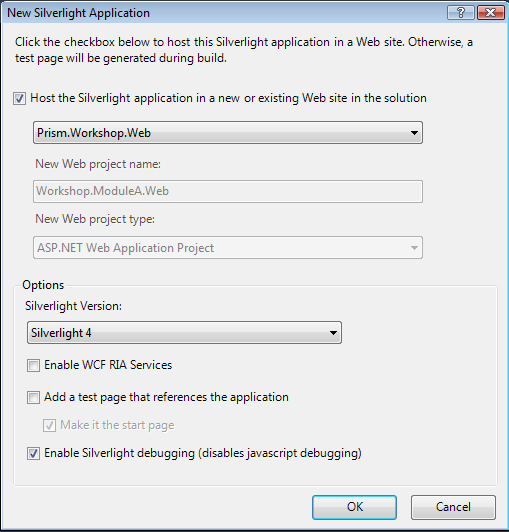
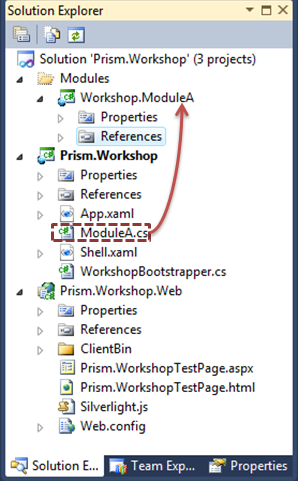
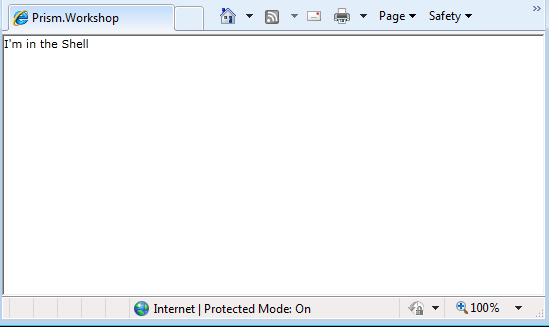
Task 2 - Loading a module

* 1. In this task, we are going to display something in the browser.
  2. Include the **Microsoft.Practices.Prism.Regions** namespace in the **Shell.xaml** file. This is shown in the following code.
     1. XAML
     2. xmlns:regions="clr-namespace:Microsoft.Practices.Prism.Regions;assembly=Microsoft.Practices.Prism"
  3. Add a **ContentControl** to **Shell.xaml** and mark it as a **Region** named **MainRegion**. To do this, add the **RegionManager**.**RegionName** attached property to the content control, as seen in the following code.
     1. XAML
     2. <ContentControl regions:RegionManager.RegionName="MainRegion"/>
  4. Implement a constructor in **ModuleA** that receives an **IRegionManager** instance and assigns it to a private field.
     1. C#
     2. private IRegionManager regionManager;
     3. public ModuleA(IRegionManager regionManager)
     4. {
     5. this.regionManager = regionManager;
     6. }
  5. Create a **TextBlock** control in the **Initialize** method in **ModuleA**. Set the **Text** property to the **"I'm in the Shell"**string and add the **TextBlock** to the **MainRegion**. This is shown in the following code.
     1. C#
     2. public void Initialize()
     3. {
     4. TextBlock textBlock = new TextBlock();
     5. textBlock.Text = "I’m in the Shell";
     6. this.regionManager.Regions["MainRegion"].Add(textBlock);
     7. }
  6. Compile and run the solution. You should see the **"I’m in the Shell"**text being displayed as shown in the following figure.
     1. Textblock from ModuleA being displayed in the Shell
     2. 

Exercise 2: Loading a module from a separate project

* 1. This exercise will show you how to load a module placed in a project other than the Shell through in-code population of the **ModuleCatalog**.

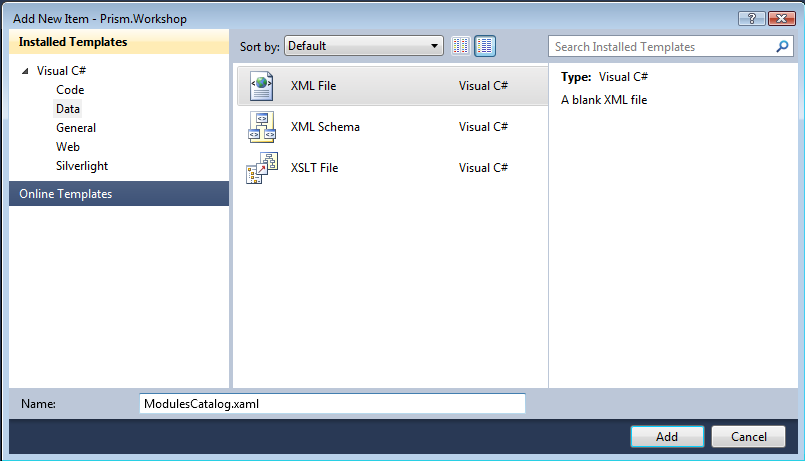
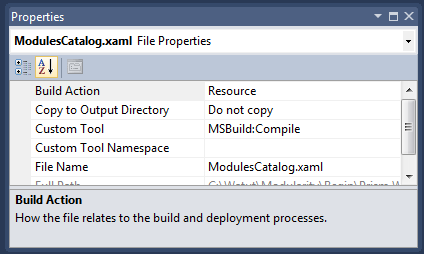
Task 1 - Creating and Loading a Module

* 1. Open the **Prism**.**Workshop** solution located in the **Exercise 2\Begin** folder.
  2. Add a solution folder named **Modules** to the solution.
  3. Add a new Silverlight Application project, named **Workshop.ModuleA**,to the new solution folder.
     1. Clear the **Add a test page that references the application** check box, and then click **OK**.
     2. **Note:** The reason for adding a Silverlight Application (and not a class Library) is to enable Remote Module Loading in the future. For a further explanation you can check [this article](http://msdn.microsoft.com/en-us/library/ff647246.aspx) from the Prism documentation.
     3. New Project Dialog. Creating a Silverlight Application.
     4. 
     5. 
  4. Delete the **MainPage.xaml** and the **App.xaml** files created by the project template.
  5. Add a reference to the **Microsoft.Practices.Prism.dll** assembly from the new project.
  6. Move **ModuleA.cs** from the **Prism.WorkshopProject** to the newly created project.
     1. 
  7. Rename the namespace of **ModuleA** to **Workshop.ModuleA**. The outcome is shown in the following code.
     1. C#
     2. namespace Workshop.ModuleA
     3. {
     4. public class ModuleA : IModule
     5. {
     6. …
     7. }
     8. }
  8. Add a reference in the Shell project to the **Workshop.ModuleA** project.
  9. Open the **WorkshopBootstrapper** class, located in the Shell project.
  10. Update the **Bootstrapper** to use the Module class in the new project. This only requires adding the following using directive.
      1. C#
      2. using Workshop.ModuleA;
  11. Compile and run the application.
      1. 

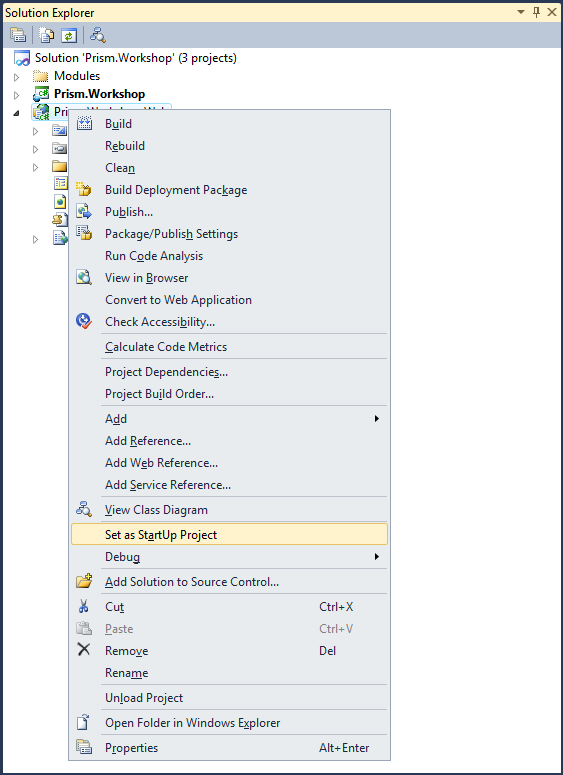
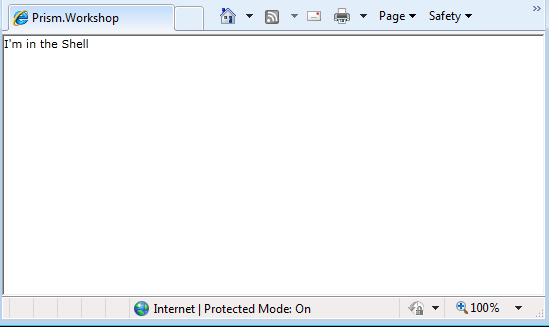
Exercise 3: Loading Modules Remotely

* 1. This exercise will lead you to creating a solution that downloads modules remotely, reducing initial download and startup time of your application.

Task 1 - Populating the ModulesCatalog through XAML

* 1. Open the **Prism**.**Workshop** solution located in the **Exercise 3\Begin** folder.
  2. Add a new file to the **Shell** project named **ModulesCatalog.xaml** using the XML file template. Make sure its Build Action is set to "Resource".
     1. Dialog to create the ModulesCatalog.xaml
     2. 
     3. ModulesCatalog.xaml properties
     4. 
  3. Open the **ModulesCatalog**.**xaml** file.
  4. Update the **ModulesCatalog.xaml** with the following root element.
     1. XAML
     2. <Modularity:ModuleCatalog xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     3. xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     4. xmlns:sys="clr-namespace:System;assembly=mscorlib"
     5. xmlns:Modularity="clr-namespace:Microsoft.Practices.Prism.Modularity;assembly=Microsoft.Practices.Prism">
     6. </Modularity:ModuleCatalog>
  5. Add a child element for **ModuleA** to be loaded. This is shown in the following code.
     1. XAML
     2. <Modularity:ModuleInfo Ref="Workshop.ModuleA.xap" ModuleName="ModuleA" ModuleType="Workshop.ModuleA.ModuleA, Workshop.ModuleA, Version=1.0.0.0" />
  6. Add the following using statement in the **WorkshopBootstrapper**.
     1. C#
     2. using System;
  7. Update the **CreateModuleCatalog** template method in the **WorkshopBootstrapper** to populate the **ModuleCatalog** from XAML. This can be seen in the following code.
     1. C#
     2. protected override IModuleCatalog CreateModuleCatalog()
     3. {
     4. return Microsoft.Practices.Prism.Modularity.ModuleCatalog.CreateFromXaml(
     5. new Uri("/Prism.Workshop;component/ModulesCatalog.xaml", UriKind.Relative));
     6. }
  8. Compile and run the solution.
  9. Currently, the solution is loading **ModuleA** from XAML, but it is not being done remotely. The next procedure will guide you to achieve this.

Task 2 - Loading ModuleA Remotely

* 1. Remove the following using statement from the **WorkshopBootstrapper**.
     1. C#
     2. using Workshop.ModuleA;
  2. Remove the reference to **Workshop**.**ModuleA** in the Shell project.
  3. Delete the solution’s **bin** and **obj** folders to make sure there are no binaries remaining.
  4. Set the Web project as the application’s startup project.
     1. Setting the Web project as the startup project for the application.
     2. 
  5. Compile and run the solution.
     1. 
  6. **Note:** You might notice that **ModuleA** takes a bit longer to load. That is because it is being loaded remotely.