PRISM 4.0 TRAINING KIT

Hands-On Lab

Using MEF as the Dependency Injection Container

Lab version: 1.0.0

Last updated: 11/19/2010

* 1. 

# Contents

[Overview 3](#_Toc277931904)

[Exercise 1 - Loading Modules Using MEF Catalogs 4](#_Toc277931905)

[Task 1 – Creating the Application Bootstrapper 4](#_Toc277931906)

[Task 2 – Loading Modules with Type Catalogs 5](#_Toc277931907)

[Task 3 – Adding a Region to the Shell 6](#_Toc277931908)

[Task 4 – Showing a view in the Shell 7](#_Toc277931909)

[Exercise 2 - Registering views using the ViewExport Attribute 9](#_Toc277931910)

[Task 1 – Creating a view inside the shell project 9](#_Toc277931911)

[Task 2 – Decorating a view with the ViewExport Attribute 10](#_Toc277931912)

[Exercise 3 - Using Remote Module Loading 12](#_Toc277931913)

[Task 1 – Creating the new ModuleB 13](#_Toc277931914)

[Task 2 – Creating a XAML ModuleCatalog and Updating BootStrapper to Remotely Load ModuleB 15](#_Toc277931915)

[Exercise 4 - Monitoring Download Progress of Modules 18](#_Toc277931916)

[Task 1 – Configuring a Module to be Loaded on Demand 18](#_Toc277931917)

[Task 2 – Adding a Download Progress Indicator 19](#_Toc277931918)

Overview

The Prism Library provides two options for dependency injection containers: Unity or MEF. Prism is extensible, thereby allowing other containers to be used instead with a little bit of work. Both Unity and MEF provide the same basic functionality for dependency injection, even though they work very differently. Some of the capabilities provided by both containers include the following:

* + They both register types with the container.
  + They both register instances with the container.
  + They both imperatively create instances of registered types.
  + They both inject instances of registered types into constructors.
  + They both inject instances of registered types into properties.
  + They both have declarative attributes for marking types and dependencies that need to be managed.
  + They both resolve dependencies in an object graph.
  1. MEF provides several capabilities that Unity does not:
  + It discovers assemblies in a directory.
  + It uses XAP file download and assembly discovery.
  + It recomposes properties and collections as new types are discovered.
  + It automatically exports derived types.
  + It is deployed with the .NET Framework.

Exercise 1 - Loading Modules Using MEF Catalogs

Task 1 – Creating the Application Bootstrapper

* 1. In this task, you will create a bootstrapper for your Prism application. Over the following tasks, you will add functionality to this bootstrapper class.
  2. Open the **MEFCodeModuleLoadingBegin.sln** solution located under the **\MEF\Exercise 1\Begin\** directory of this training kit**.**
  3. Add references to the following assemblies to the **Shell** project. The assemblies are located under the **Lib** folder of this lab:
     1. Microsoft.Practices.Prism.dll
     2. Microsoft.Practices.Prism.MefExtensions.dll
     3. Microsoft.Practices.ServiceLocation.dll
     4. System.ComponentModel.Composition.dll
  4. Add a new class to the **Shell** project, named **WorkshopBootstrapper**.
  5. Add the following using statement at the top of the new class.
     1. C#
     2. using Microsoft.Practices.Prism.MefExtensions;
  6. Update the class’ signature to inherit from the **MefBootstrapper** class.
     1. C#
     2. public class WorkshopBootstrapper : **MefBootstrapper**
     3. {
     4. }
     5. **Note:** From the Prism documentation: “*A bootstrapper is a class that is responsible for the initialization of an application built using the Prism Library. By using a bootstrapper, you have more control of how the Prism Library components are wired up to your application.”* More information about this can be found [here](http://msdn.microsoft.com/en-us/library/gg430868%28PandP.40%29.aspx).
  7. Provide a fake implementation for the abstract **CreateShell** method, as shown in the following code.
     1. C#
     2. protected override DependencyObject CreateShell()
     3. {
     4. return null;
     5. }
     6. **Note:** Before the end of this exercise you will provide a real implementation for this method.
  8. In Visual Studio, save all changes.

Task 2 – Loading Modules with Type Catalogs

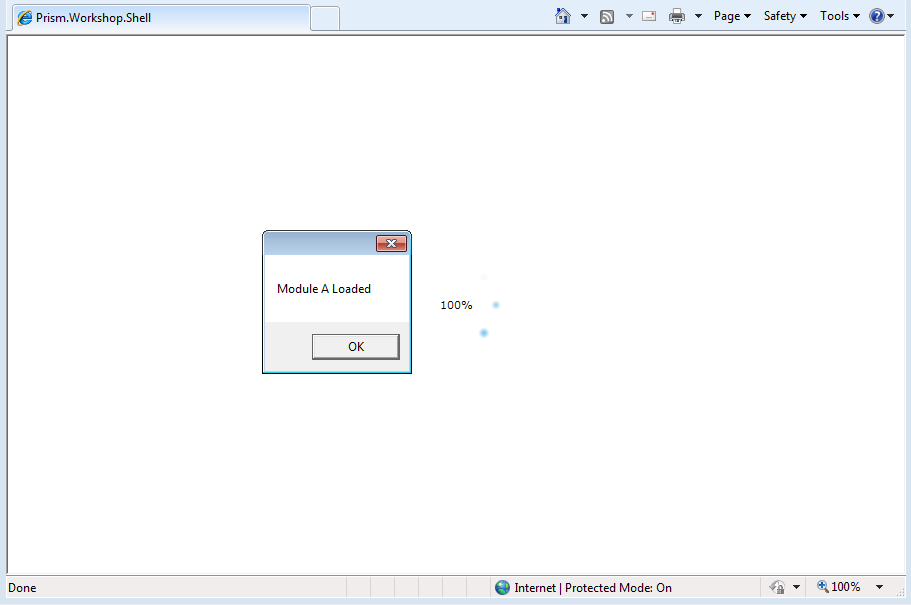
* 1. In this task you will create a class that implements the IModule interface (which is used to initialize modules).
  2. Add a new class to the **Shell** project, named **ModuleA**.
  3. Add the following using statements to the **ModuleA.cs** file.
     1. C#
     2. using Microsoft.Practices.Prism.MefExtensions.Modularity;
     3. using Microsoft.Practices.Prism.Modularity;
  4. Decorate the **ModuleA** class with the **ModuleExport** attribute and make it implement the **IModule** interface, as shown in the following code.
     1. C#
     2. [ModuleExport(typeof(ModuleA))]
     3. public class ModuleA : IModule
     4. {
     5. }
     6. **Note:** The **ModuleExport** attribute is used to make classes that implement the **IModule** interface automatically discoverable by MEF.
  5. Implement the interface’s Initialize method to display a message showing that the module has been loaded.
     1. C#
     2. public void Initialize()
     3. {
     4. MessageBox.Show("Module A Loaded");
     5. }
  6. Open the **WorkshopBootstrapper.cs** file.
  7. Add the following using statement at the top of the class.
     1. C#
     2. using System.ComponentModel.Composition.Hosting;
  8. Override the **ConfigureAggregateCatalog** method, to add **ModuleA** as a type recognizable by MEF.
     1. C#
     2. protected override void ConfigureAggregateCatalog()
     3. {
     4. // Modules are added / registered in the aggregate catalog

// More info: http://msdn.microsoft.com/en-us/library/ff921163(PandP.40).aspx

base.ConfigureAggregateCatalog();

* + 1. this.AggregateCatalog.Catalogs.Add(new TypeCatalog(new[] { typeof(ModuleA) }));

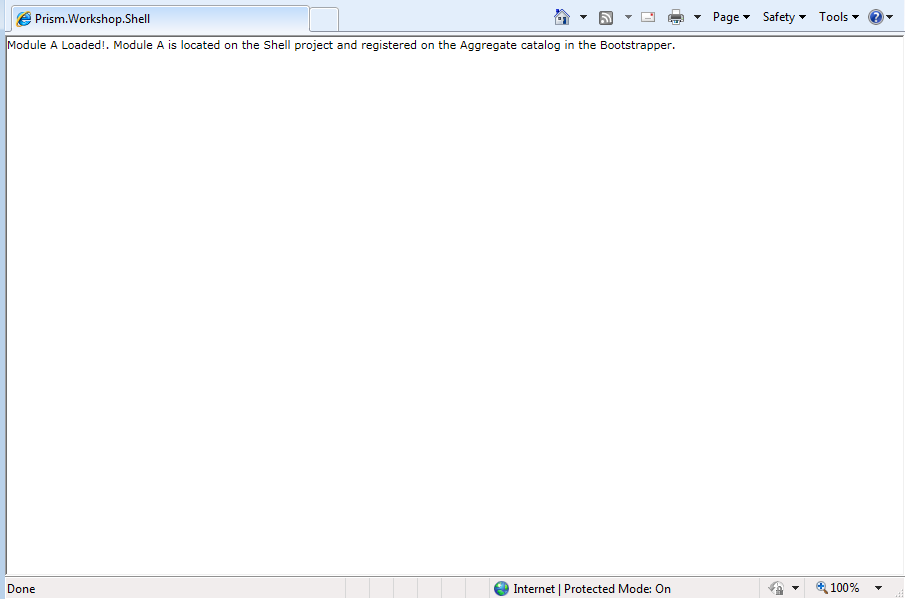
}

* 1. Open the code behind file for **App.xaml**, and update the code for the **Application\_Startup** method as shown in the following code.
     1. C#
     2. private void Application\_Startup(object sender, StartupEventArgs e)
     3. {
     4. new WorkshopBootstrapper().Run();
     5. }
  2. In Visual Studio, press **F5** to run the application. The message showing that the module has been loaded will appear, as shown in the following figure.
     1. 
     2. Message showing that ModuleA has been loaded
  3. Click **OK** and close the browser window.

Task 3 – Adding a Region to the Shell

* 1. In this task you will add a control to the Shell and mark it as a region.
  2. Open the **Shell.xaml** file.
  3. 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"
  4. Add an **ItemsControl** to **Shell.xaml** (as a child to the Grid) and mark it as a **Region** named **MainRegion**. To do this, add the **RegionManager**.**RegionName** attached property to the control, as seen in the following code.
     1. C#
     2. <Grid x:Name="LayoutRoot" Background="White">
     3. **<ItemsControl regions:RegionManager.RegionName="MainRegion"/>**
     4. </Grid>

Task 4 – Showing a view in the Shell

* 1. In this task, you will display the Shell in the browser and add a view to its region.
  2. Open the code behind file for the **Shell**.
  3. Add the following using statement at the top of the file.
     1. C#
     2. using System.ComponentModel.Composition;
  4. Decorate the Shell class with the Export attribute, to make it discoverable for MEF.
     1. C#
     2. **[Export]**
     3. public partial class Shell : UserControl
     4. {
     5. public Shell()
     6. {
     7. InitializeComponent();
     8. }
     9. }
  5. Open the **WorkshopBootstrapper.cs** file.
  6. Update the **ConfigureAggregateCatalog** to add the Shell as a type recognizable by MEF. This is shown in the following code.
     1. C#
     2. protected override void ConfigureAggregateCatalog()
     3. {
     4. // Modules are added / registered in the aggregate catalog
     5. // More info: http://msdn.microsoft.com/en-us/library/ff921163(PandP.40).aspx
     6. base.ConfigureAggregateCatalog();
     7. // Add the shell to the catalog so that it can be retrieved afterwards in the CreateShell method.
     8. **this.AggregateCatalog.Catalogs.Add(new TypeCatalog(new[] { typeof(Shell), typeof(ModuleA) }));**
     9. }
     10. **Note:** The **ConfigureAggregateCatalog** method gives you programmatic control over the registrations to be added to the **AggregateCatalog**.
  7. Update the **CreateShell** method to return as instance of the Shell class.
     1. C#
     2. protected override DependencyObject CreateShell()
     3. {
     4. //Return an instance of the Shell, retrieving it from the MEF container.
     5. //As the shell has no dependencies (imports) it could just be created with "new Shell()", but it is recommended to always export the shell accounting for future updates to shell dependencies.
     6. **return this.Container.GetExportedValue<Shell>();**
     7. }
  8. Override the **InitializeShell** method to set the **Shell** as the application’s **RootVisual** element.
     1. C#
     2. protected override void InitializeShell()
     3. {
     4. base.InitializeShell();
     5. // Set the shell as the RootVisual (startup window in this case) of the application.
     6. Application.Current.RootVisual = (UIElement)this.Shell;
     7. }
  9. Open the **ModuleA.cs** file.
  10. Add the following using statements at the top of the file.
      1. C#
      2. using System.ComponentModel.Composition;
      3. using Microsoft.Practices.Prism.Regions;
  11. Add an **ImportingConstructor** that imports an instance of the **RegionManager** and stores it in a field.
      1. C#
      2. private readonly IRegionManager regionManager;
      3. [ImportingConstructor]
      4. public ModuleA(IRegionManager regionManager)
      5. {
      6. this.regionManager = regionManager;
      7. }
  12. Update the **Initialize** method to show a **TextBox** in the region declared in the **Shell**.
      1. C#
      2. public void Initialize()
      3. {
      4. TextBlock textBlock = new TextBlock();
      5. textBlock.Text = "Module A Loaded!. Module A is located on the Shell project and registered on the Aggregate catalog in the Bootstrapper.";
      6. this.regionManager.Regions["MainRegion"].Add(textBlock);
      7. }
  13. In Visual Studio, press **F5** to run the application.
  14. Notice that the **TextBox** has been added to the **Region**.
      1. 
      2. The TextBox has been added to the Region from the Module

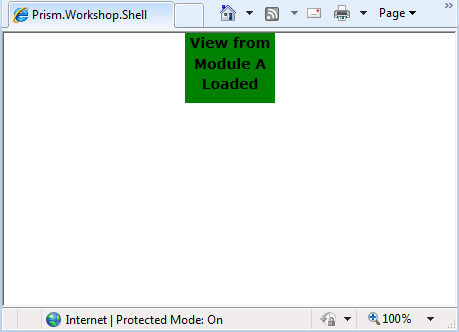
Exercise 2 - Registering views using the ViewExport Attribute

Task 1 – Creating a view inside the shell project

In this task you will create a view, which you will later register in a region through the ViewExport attribute.

* 1. Open the **MEFViewExportAttribute.sln** solution located under the **\MEF\Exercise 3\Begin\** folder of this training kit**.**
  2. In the **Shell** project, create a folder named **Views**.
  3. Add a new **UserControl** named **ModuleAView** to the **Shell** project, inside the **Views** folder.
  4. Add a TextBlock to the **ModuleAView.xaml** file, to display a message indicating that the view has been loaded.
     1. XAML
     2. <Grid x:Name="LayoutRoot" Background="White">
     3. **<Border HorizontalAlignment="Center" VerticalAlignment="Center" Width="90" Height="70" Background="Green">**
     4. **<TextBlock TextAlignment="Center" TextWrapping="Wrap" FontWeight="Bold" FontSize="14">View from Module A Loaded</TextBlock>**
     5. **</Border>**
     6. </Grid>

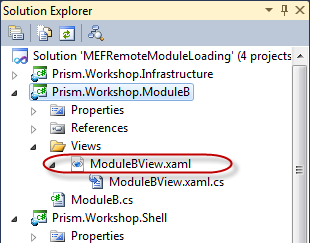
Task 2 – Decorating a view with the ViewExport Attribute

* 1. In the **Prism.Workshop.Shell** project create a folder named **Infrastructure**.
  2. Add the following classes from the Assets folder inside the **Infrastructure** folder: **AutoPopulateExportedViewsBehavior.cs, IViewRegionRegistration.cs** and **ViewExportAttribute.cs**. Those classes make it possible to register a view to a region through the use of a ViewExport attribute.
  3. Add the following using statements to the **WorkshopBootstrapper.cs** file.
     1. C#
     2. **using Prism.Workshop.Shell.Infrastructure;**
     3. **using Prism.Workshop.Shell.Views;**
     4. **using Microsoft.Practices.Prism.Regions;**
  4. In the **ConfigureAggregateCatalog** method delete the code that was used to add the **Shell** and **ModuleA** to the AggregateCatalog.
     1. C#
     2. // Add the shell to the catalog so that it can be retrieved afterwards in the CreateShell method.
     3. this.AggregateCatalog.Catalogs.Add(new TypeCatalog(new[] { typeof(Shell) }));
     4. // Add Module A to the catalog
     5. this.AggregateCatalog.Catalogs.Add(new TypeCatalog(new[] { typeof(ModuleA) }));
  5. Update the **ConfigureAggregateCatalog** to add the **Shell**, the **ModuleAView** and the **AutoPopulateExportedViewsBehavior** as types recognizable by MEF.
     1. C#
     2. // Add neccesary classes to the catalog
     3. this.AggregateCatalog.Catalogs.Add(new TypeCatalog(new[] {
     4. typeof(Shell), // Add the shell to the catalog so that it can be retrieved afterwards in the CreateShell method.
     5. typeof(AutoPopulateExportedViewsBehavior), // Region behavior for finding viewExports
     6. typeof(ModuleAView) // View of Module A.
     7. }));
     8. **Note:** Alternatively, you could add the whole **Prism.Workshop.Shell** assembly to the AggregateCatalog, by adding the following code to the **ConfigureAggregateCatalog** method.
     9. this.AggregateCatalog.Catalogs.Add(new AssemblyCatalog(typeof(Shell).Assembly));
  6. Override the **ConfigureDefaultRegionBehaviors** method to add the **AutoPopulateExportedViewsBehavior**, as shown in the following code.
     1. C#
     2. protected override IRegionBehaviorFactory ConfigureDefaultRegionBehaviors()
     3. {
     4. // Add custom region behaviors.
     5. // More info: http://msdn.microsoft.com/en-us/library/gg430866(v=PandP.40).aspx
     6. var factory = base.ConfigureDefaultRegionBehaviors();
     7. // behavior that registers all views decorated with the ViewExport attribute
     8. factory.AddIfMissing("AutoPopulateExportedViewsBehavior", typeof(AutoPopulateExportedViewsBehavior));
     9. return factory;
     10. }
  7. Add the following using statements to the **ModuleAView.cs** file.
     1. C#
     2. **using Prism.Workshop.Shell.Infrastructure;**
     3. **using System.ComponentModel.Composition;**
  8. Decorate the **ModuleAView** class with the following attributes.
     1. C#
     2. **[PartCreationPolicy(CreationPolicy.NonShared)] // creates a new instance of the view each time it is imported**
     3. **[ViewExport(RegionName = "MainRegion")] // registers the view with the MainRegion. More info: http://msdn.microsoft.com/en-us/library/ff921074(v=PandP.40).aspx**
     4. public partial class ModuleAView : UserControl
     5. {
     6. public ModuleAView()
     7. {
     8. InitializeComponent();
     9. }
     10. }
     11. **Note:** By default, MEF exports are created as singletons. That is, each time the part is imported, the same instance is returned. By setting the PartCreationPolicy to NonShared, a new instance is returned each time the part is imported.
  9. Delete the **ModuleA.cs** file, as it is no longer necessary.
  10. In Visual Studio, press **F5** to run the application.
  11. Notice that the **ModuleAView** has been added to the MainRegion inside the Shell.
      1. 
      2. The ModuleAView has been added to the MainRegion

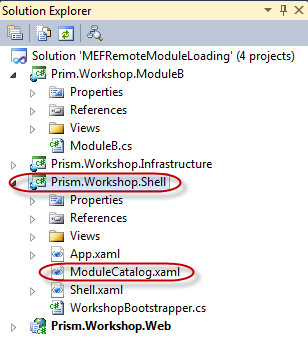
Exercise 3 - Using Remote Module Loading

* 1. In this exercise you will create a new module (**ModuleB**) on a separated project, and you will configure the bootstrapper in order to load the module remotely. The new module will add an additional view to the **MainRegion**, but in contrast with the view created on the previous exercises, the shell project will not have a reference to the **ModuleB** (or its views).
  2. You will observe that the necessary classes to support the **ViewExport** attributes, which on the previous exercises were located on the **Shell** project, now are in a separated **Infrastructure** project. This will allow view’s on any module to make use of the **ViewExport** attribute to register view on regions.

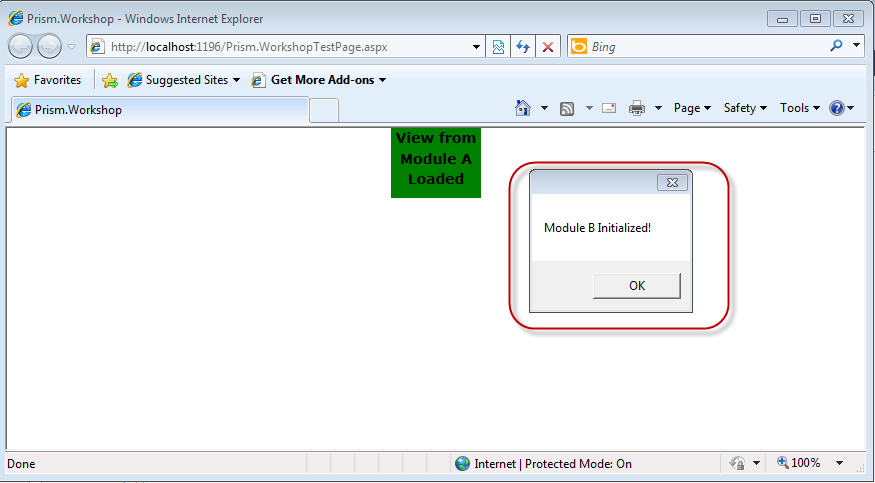
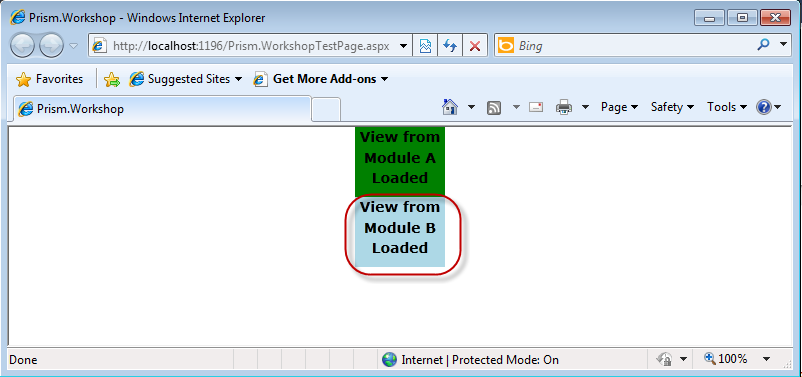
Task 1 – Creating the new ModuleB

* 1. In this task you will create a new module on a separated project. This project will contain the module class (ModuleB.cs), which will show a message box when it is initialized, and a view that will shows a simple label with its name.
  2. Open the **MEFRemoteModuleLoading.sln** solution located under the **\MEF\Exercise 3\Begin\** folder of this training kit**.**
  3. On the **MEFRemoteModuleLoading** solution, create a new Silverlight Application project named **Prism.Workshop.ModuleB.** On the **New Silverlight Application** dialog, uncheck the **Add a test page that references this application** and make sure the project is hosted on **the Prism.Workshop.Web** project. This project will contain all ModuleB’s assets.
  4. On the **Prism.Workshop.ModuleB** project, delete the App.xaml and MainPage.xaml files.
  5. On the **Prism.Workshop.ModuleB** project, add a new class named **ModuleB**. The ModuleB.cs file will open.
  6. On the **Prism.Workshop.ModuleB** project, add the following references:
     1. Microsoft.Practices.Prism.dll
     2. Microsoft.Practices.Prism.MefExtensions.dll
     3. System.ComponentModel.Composition.dll
     4. **Note:** **Microsoft.Practices.Prism.dll** and **Microsoft.Practices.Prism.MefExtensions.dll** are Prism libraries that can be found on the **Lib** folder on this Training Kit. **System.ComponentModel.Composition.dll** is part of .Net libraries.
  7. On the Properties for the Microsoft.Practices.Prism.MefExtensions.dll reference, set the **Copy Local** option to **False**.
     1. **Note:** If this step is not performed, the MEF libraries will be included (and imported by MEF) twice: one for the module and one for the Shell), causing an exception to be thrown. For more information you can read [Chapter 4: Modular Application Development](http://msdn.microsoft.com/en-us/library/gg405479(v=PandP.40).aspx).
  8. In the ModuleB.cs file, add the following using statements:
     1. C#
     2. **using System.Windows;**
     3. **using Microsoft.Practices.Prism.MefExtensions.Modularity;**
     4. **using Microsoft.Practices.Prism.Modularity;**
  9. Update the **ModuleB** class signature to make the **ModuleB** inherit from the **IModule** interface, as shown below:
     1. C#
     2. public class ModuleB **: IModule**
  10. Implement the **Initialize** method on the **ModuleB** class. This will show a message box when the module is initialized.
      1. C#
      2. public class ModuleB : IModule
      3. {
      4. **public void Initialize()**
      5. **{**
      6. **MessageBox.Show("Module B Initialized!");**
      7. **}**
      8. }
  11. Decorate the **ModuleB** class with the **ModuleExport** attribute, so that it can be discovered as a Prism module.
      1. C#
      2. **[ModuleExport(typeof(ModuleB))]**
      3. public class ModuleB : IModule
  12. To create the ModuleB’s view, create a new folder named **Views**, and create a new UserControl named **ModuleBView.xaml** inside**.**
      1. 
      2. ModuleB solution
  13. Update the ModuleBView XAML as follows, to render a rectangle with a simple label inside.
      1. XAML
      2. <UserControl x:Class="Prism.Workshop.ModuleB.Views.ModuleBView"
      3. xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      4. xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      5. xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      6. xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      7. mc:Ignorable="d"
      8. d:DesignHeight="300" d:DesignWidth="400">
      10. **<Grid x:Name="LayoutRoot" Background="White">**
      11. **<Border HorizontalAlignment="Center" VerticalAlignment="Center" Width="90" Height="70" Background="LightBlue">**
      12. **<TextBlock TextAlignment="Center" TextWrapping="Wrap" FontWeight="Bold" FontSize="14">View from Module B Loaded</TextBlock>**
      13. **</Border>**
      14. **</Grid>**
      15. </UserControl>
  14. Add the following reference to the **Prim.Workshop.ModuleB** project, to enable the use of the **ViewExport** attribute
      1. Prism.Workshop.Infrastructure
  15. On the ModuleBView.cs file, add the following using statements
      1. C#
      2. using System.ComponentModel.Composition;
      3. using Prism.Workshop.Infrastructure;
  16. On the ModuleBView.cs file, decorate the **ModuleBView** class to register the view on the **MainRegion**.
      1. C#
      2. **[PartCreationPolicy(CreationPolicy.NonShared)] // creates a non-singleton instance of the view**
      3. **[ViewExport (RegionName = "MainRegion")] // registers the view with the MainRegion. More info: http://msdn.microsoft.com/en-us/library/ff921074(v=PandP.40).aspx**
      4. public partial class ModuleBView : UserControl

Task 2 – Creating a XAML ModuleCatalog and Updating BootStrapper to Remotely Load ModuleB

* 1. In this task you will add a XAML module catalog to the shell project. The XAML module catalog will describe the modules you want to be loaded in your application (in the case of this lab, only ModuleB should be loaded)
  2. In the **Prism.Workshop.Shell** project, create a ModulesCatalog.xaml file.
     1. 

ModuleCatalog file

* 1. Update the ModulesCatalog.xaml file as follows, to declare how ModuleB will be loaded.
     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. **<!--ModuleGroup could be removed-->**
     7. **<Modularity:ModuleInfoGroup InitializationMode="WhenAvailable">**
     8. **<Modularity:ModuleInfo Ref="Prism.Workshop.ModuleB.xap" ModuleName="ModuleB" ModuleType="Prism.Workshop.ModuleB.ModuleB, Prism.Workshop.ModuleB, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />**
     9. **</Modularity:ModuleInfoGroup>**
     10. </Modularity:ModuleCatalog>
  2. On the WorkshopBootstraper.cs file, add the following using statements:
     1. C#
     2. using Microsoft.Practices.Prism.Modularity;
     3. using System;
  3. Override the **CreateModuleCatalog** of the base **MefBootstrapper** class, adding the following code to the **WorkshopBootstraper** class. This code creates a ModuleCatalog based on the XAML catalog defined previously.
     1. C#
     2. **protected override IModuleCatalog CreateModuleCatalog()**
     3. **{**
     4. **// Create ModuleCatalog to configure modules via configuration files.**
     5. **return Microsoft.Practices.Prism.Modularity.ModuleCatalog.CreateFromXaml(new Uri("/Prism.Workshop.Shell;component/ModulesCatalog.xaml", UriKind.Relative));**
     6. **}**
  4. Set the **Prism.Workshop.Web** project as the startup project.
  5. Set the **Prism.WorkshopTestPage.aspx** located on the **Prism.Workshop.Web** project, as the startup page.
  6. Press **F5** to run the application.
  7. Verify that when ModuleB is loaded a message box appears on the screen.
     1. 
     2. ModuleB loaded
  8. Verify that the view from ModuleB is added on the MainRegion.
     1. 
     2. View from ModuleB loaded

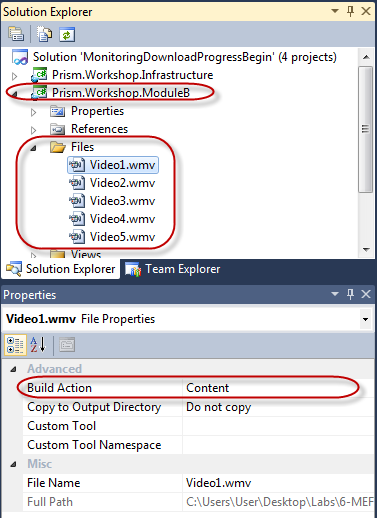
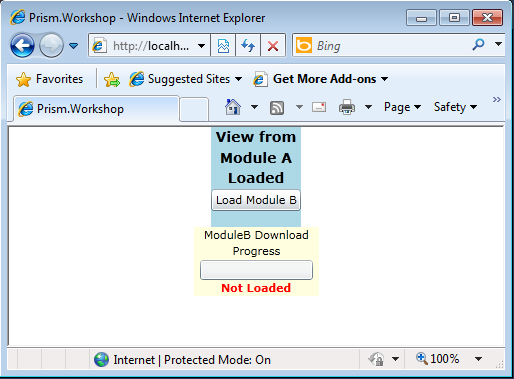
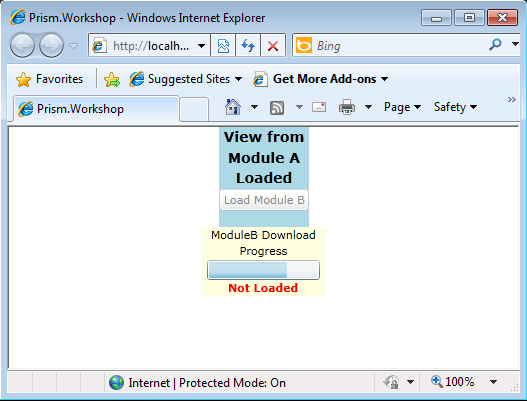
Exercise 4 - Monitoring Download Progress of Modules

Task 1 – Configuring a Module to be Loaded on Demand

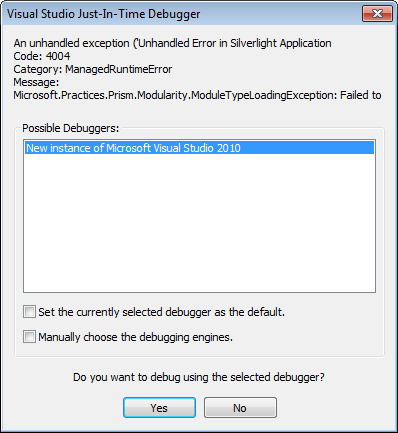
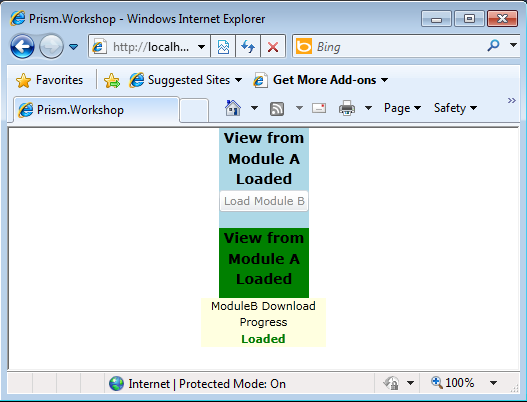
In this task you will configure a module that was previously made to be loaded when available to be loaded and initialized on demand. Afterwards you will add the necessary code to the ModuleAView to trigger the module loading using the Prism API.

* 1. Open the **MonitoringDownloadProgressBegin.sln** solution located under the **\MEF\Exercise 4\Begin\** directory of this training kit**.**
  2. Open the **ModulesCatalog.xaml** file inside the **Prism.Workshop.Shell** project.
  3. Modify the **InitializationMode** of the **ModuleGroup** containing **ModuleB** to **OnDemand**, as shown in the following code.
     1. C#
     2. <Modularity:ModuleInfoGroup **InitializationMode="OnDemand"**> <!--ModuleGroup could be removed-->
     3. <Modularity:ModuleInfo Ref="Prism.Workshop.ModuleB.xap" ModuleName="ModuleB" ModuleType="Prism.Workshop.ModuleB.ModuleB, Prism.Workshop.ModuleB, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
     4. </Modularity:ModuleInfoGroup>
  4. Open the **ModuleAView.xaml** file located inside the **Views** folder. Add a button to load ModuleB, as shown in the following code.
     1. C#
     2. <Border HorizontalAlignment="Center" VerticalAlignment="Center" Width="90" Height="100" Background="LightBlue">
     3. **<StackPanel>**
     4. <TextBlock TextAlignment="Center" TextWrapping="Wrap" FontWeight="Bold" FontSize="14">View from Module A Loaded</TextBlock>
     5. **<Button x:Name="LoadButton" Content="Load Module B" Click="Button\_Click"/>**
     6. **</StackPanel>**
     7. </Border>
  5. In the code behind for the **ModuleAView** view, add the following using statement.
     1. C#
     2. **using Microsoft.Practices.Prism.Modularity;**
  6. Define a field to hold an instance of the ModuleManager.
     1. C#
     2. **private readonly IModuleManager moduleManager;**
  7. Decorate the constructor of **ModuleAView** with the **ImportingConstructor** attribute and assign the Module Manager to the field you’ve previously defined, as shown in the following code.
     1. C#
     2. **[ImportingConstructor]**
     3. public ModuleAView(**IModuleManager moduleManager**)
     4. {
     5. **this.moduleManager = moduleManager;**
     6. InitializeComponent();
     7. }
  8. In the handler for the **Click** event for the button you’ve created, disable the button and load ModuleB.
     1. C#
     2. private void Button\_Click(object sender, System.Windows.RoutedEventArgs e)
     3. {
     4. this.LoadButton.IsEnabled = false;
     5. this.moduleManager.LoadModule("ModuleB");
     6. }

Task 2 – Adding a Download Progress Indicator

* 1. On the Shell.cs file, add the following using statements
     1. C#
     2. **using System;**
     3. **using System.Windows;**
     4. **using System.Windows.Media;**
     5. **using Microsoft.Practices.Prism.Modularity;**
  2. Add the following private field to the **Shell** class
     1. C#
     2. **private readonly IModuleManager moduleManager;**
  3. Decorate the **Shell** class constructor with the **ImportingConstructor** attribute, and add an **IModuleManager** parameter
     1. C#
     2. **[ImportingConstructor]**
     3. public Shell(**IModuleManager moduleManager**)
  4. Update the Shell constructor implementation as follows. These will keep an instance of the ModuleManager and listen to the **LoadModuleCompleted** and **ModuleDownloadProgressChanged** events.
     1. C#
     2. [ImportingConstructor]
     3. public Shell(IModuleManager moduleManager)
     4. {
     5. **this.moduleManager = moduleManager;**
     6. InitializeComponent();
     7. **this.moduleManager.LoadModuleCompleted += LoadModuleCompleted;**
     8. **this.moduleManager.ModuleDownloadProgressChanged += ModuleDownloadProgressChanged;**
     9. }
  5. Implement the methods that will handle the **LoadModuleCompleted** and **ModuleDownloadProgressChanged** events using the following code
     1. C#
     2. **void ModuleDownloadProgressChanged(object sender, ModuleDownloadProgressChangedEventArgs e)**
     3. **{**
     4. **this.ModuleBProgressBar.Value = Convert.ToDouble(e.ProgressPercentage);**
     5. **}**
     6. **void LoadModuleCompleted(object sender, LoadModuleCompletedEventArgs e)**
     7. **{**
     8. **this.ModuleBInitTextBlock.Text = "Loaded";**
     9. **this.ModuleBInitTextBlock.Foreground = new SolidColorBrush(Colors.Green);**
     10. **this.ModuleBProgressBar.Visibility = Visibility.Collapsed;**
     11. **}**
  6. In the ModuleB.cs file on the **Prism.Workshop.ModuleB** project, remove the line that shows the message box when the module is loaded.
     1. C#
     2. [ModuleExport(typeof(ModuleB))]
     3. public class ModuleB : IModule
     4. {
     5. public void Initialize()
     6. {
     7. **~~MessageBox.Show("Module B Initialized!");~~**
     8. }
     9. }
  7. To verify the download progress for ModuleB, as you are running the lab locally, you will have to add a lot of content to ModuleB XAP file. This way, it will take longer to download and you will be able to see the progress. To do this, you can for example add several files to the ModuleB and set the **Build** **Action** property to **Content**.
     1. **Note:** To see the download progress while running the lab locally, you should add ~200MB of content.
     2. 
     3. Adding content to ModuleB
  8. Set the **Prism.Workshop.Web** project as the startup project.
  9. Set the **Prism.WorkshopTestPage.aspx** located on the **Prism.Workshop.Web** project, as the startup page.
  10. Press **F5** to run the application. The application will start and the view of ModuleA will be added to the Main Region.
      1. 
      2. ModuleA added to the MainRegion. ModuleB not loaded
  11. Click **Load Module B**, to start downloading ModuleB. Verify that the progress in shown on the progress bar at the bottom.
      1. 

ModuleB load progress

* + 1. **Note:** While loading ModuleB you might experience the following exception. This happens because you’re loading a large amount of files locally. If this happens, try again. If you continue experiencing this error, you could try hosting the module in a remote location with fewer files.
    2. 
    3. Exception when loading ModuleB
  1. Verify that when ModuleB completes downloading, a view is added to the MainRegion
     1. 
     2. ModuleA added to the MainRegion. ModuleB not loaded