PRISM 4.0 TRAINING KIT

Hands-On Lab

Bootstrapper

Lab version: 1.0.0

Last updated: 11/19/2010

* 1. 

# Contents

[Overview 3](#_Toc277956447)

[Exercise 1: Customizing the container’s configuration 4](#_Toc277956448)

[Task 1 - Moving the UIMessagesService to the Infrastructure Project 4](#_Toc277956449)

[Exercise 2: Using a custom logger 8](#_Toc277956450)

[Task 1 - Creating the custom logger 9](#_Toc277956451)

[Task 2 – Retrieving the Logger Output 12](#_Toc277956452)

[Task 3 – Verifying the log file output 13](#_Toc277956453)

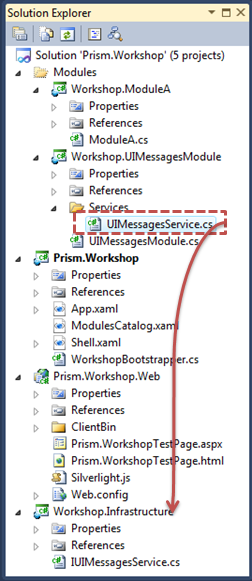
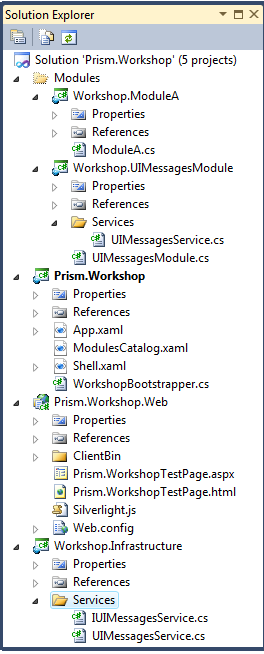
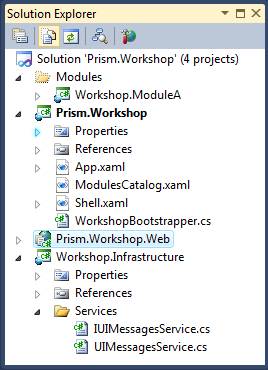
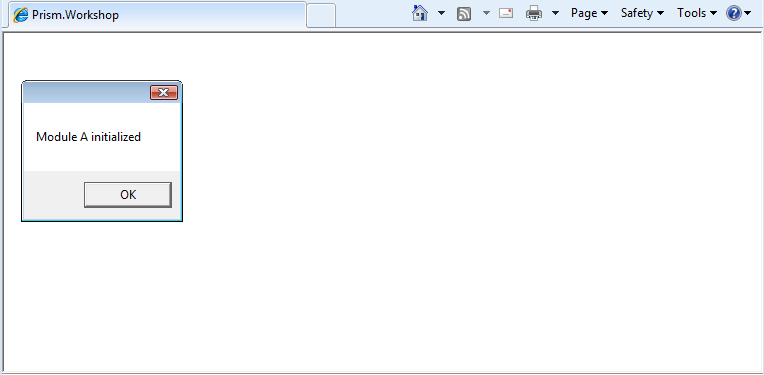
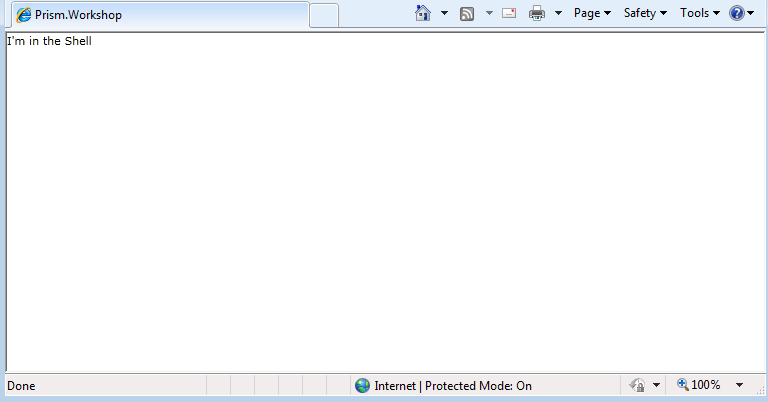
Overview

* 1. 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.
  2. The Prism Library includes a default abstract **Bootstrapper** base class that can be specialized for use with any container. Many of the methods on the bootstrapper classes are virtual methods. You can override these methods as appropriate in your own custom bootstrapper implementation.
  3. The Prism Library provides some additional base classes, derived from **Bootstrapper**, that have default implementations that are appropriate for most applications. The only stages left for your application bootstrapper to implement are creating and initializing the shell.

Exercise 1: Customizing the container’s configuration

* 1. This exercise will guide you through the steps required to customize the container’s configuration in your application’s **Bootstrapper**. You will use an XML configuration file to achieve this. For more information, please refer to [this section](http://msdn.microsoft.com/en-us/library/ff660917%28PandP.20%29.aspx) from the Unity documentation.

Task 1 - Moving the UIMessagesService to the Infrastructure Project

* 1. Open the **Prism.Workshop** solution, located in the **Exercise 1\Begin** folder.
  2. Move the **UIMessagesService.cs** file to the **Workshop.Infrastructure** project.
     1. 
     2. Figure 1
     3. Moving the message service to the infrastructure project.
  3. Place both, the **UIMessagesService.cs** and **IUIMessagesService.cs** files inside a folder named **Services**. The resulting Solution Explorer is shown in the following picture.
     1. 
  4. Update the namespaces of the **UIMessagesService** class and **IUIMessagesService** located in the interface Infrastructure project with the one shown in the following code.
     1. C#
     2. namespace Workshop.Infrastructure.Services
     3. {
     4. …
     5. }
  5. Open **ModuleA** class in the **Workshop.ModuleA** project.
  6. Replace the following using statement
     1. C#
     2. using Workshop.Infrastructure;
     3. with this one:
     4. C#
     5. using Workshop.Infrastructure.Services;
  7. Open the **ModulesCatalog.xaml** file in the Shell project.
  8. Remove the following entry, which configures the **UIMessagesModule** to be loaded. This section is shown in the following code.
     1. C#
     2. <Modularity:ModuleInfo Ref="Workshop.UIMessagesModule.xap" ModuleName="UIMessagesModule" ModuleType="Workshop.UIMessagesModule.UIMessagesModule, Workshop.UIMessagesModule, Version=1.0.0.0" />
  9. Remove **ModuleA**’s dependency to the **UIMessagesModule.** This requires the deletion of the **DependsOn** section for **ModuleA**, which is shown in the following snippet.
     1. C#
     2. <Modularity:ModuleInfo.DependsOn>
     3. <sys:String>UIMessagesModule</sys:String>
     4. </Modularity:ModuleInfo.DependsOn>
  10. Remove the **Workshop.UIMessagesModule** project from the solution. The resulting **Solution Explorer** is the following:
      1. 
  11. Add a reference to the **Workshop.Infrastructure** project in the **Shell** (**Prism.Workshop**) project
  12. Open the **WorkshopBootrstraper** class file, located in the **Shell** project.
  13. Add the following using statement:
      1. C#
      2. using Workshop.Infrastructure.Services;
  14. Override the configure container method to register the service as part of the **Bootstrapper**’s configuration. The following code shows the implementation for the **ConfigureContainer** method override.
      1. C#
      2. protected override void ConfigureContainer()
      3. {
      4. RegisterTypeIfMissing(typeof(IUIMessagesService), typeof(UIMessagesService), true);
      5. base.ConfigureContainer();
      6. }
  15. Set the **Prism.Workshop.Web** as the application’s startup project.
  16. Set **Prism.WorkshopTestPage.aspx** from the aforementioned project as the start page.
  17. Rebuild and run the solution. The outcome should be the following.
      1. 
      2. The module initialization message box is shown first.
      3. 
      4. The TextBlock is shown in the Shell afterwards.

Exercise 2: Using a custom logger

* 1. This exercise explains how to provide a custom logger for your application’s **Bootstrapper**. The logger will use XML format and will be stored in IsolatedStorage.

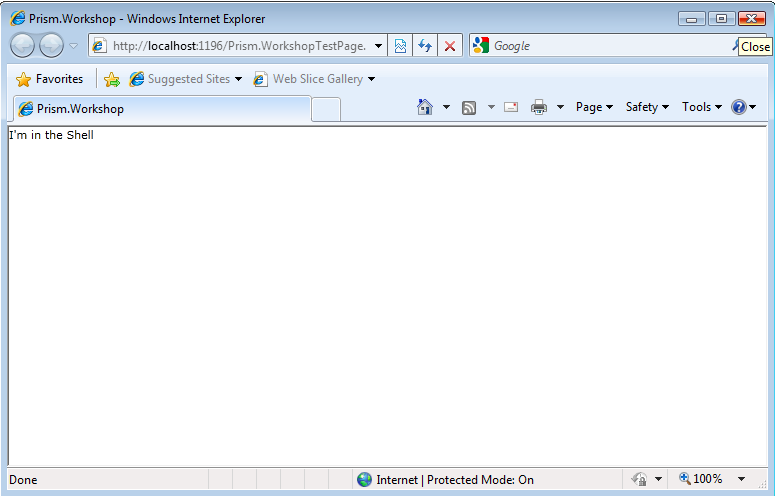
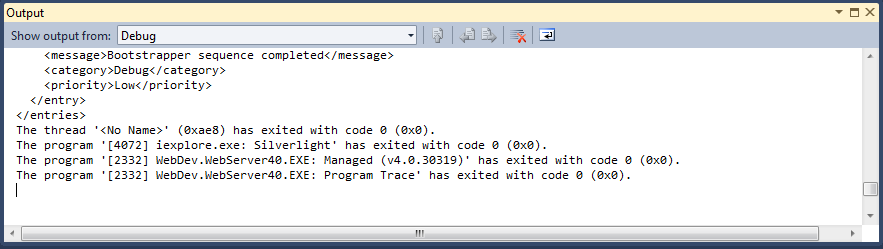
Task 1 - Creating the custom logger

* 1. Open the **Prism**.**Workshop** solution, located in the **Exercise 2\Begin** folder.
  2. Add a new class, named **Constants**, to the **Shell** project. This class will hold the output name to save the log file. Make the class static.
  3. Update the class declaration to keep the name of the log file. The following code shows what the outcome should be.
     1. C#
     2. namespace Workshop.Infrastructure
     3. {
     4. public class Constants
     5. {
     6. public const string LogOutputName = "log.xml";
     7. }
     8. }
  4. Add a reference to **System.Xml.Linq.dll** in the **Shell** project.
  5. Add a new class, named **XmlLogger**, to the project.
  6. Add the following using statements to the new class.
     1. C#
     2. using Microsoft.Practices.Prism.Logging;
     3. using System.Xml.Linq;
     4. using System.IO.IsolatedStorage;
     5. using System.IO;
     6. using Workshop.Infrastructure;
  7. Implement the **ILoggerFacade** interface. This interface has a single method, **Log**.
     1. C#
     2. public class XmlLogger : ILoggerFacade
     3. {
     4. public void Log(string message, Category category, Priority priority)
     5. {
     6. }
     7. }
  8. Provide a constructor for the class that instantiates a private **XDocument** field. As shown in the following code, this constructor will generate the initial structure for the log file.
     1. C#
     2. private XDocument document;
     3. public XmlLogger()
     4. {
     5. document = new XDocument(new XDeclaration("1.0", "utf-8", "yes"), new XElement("entries"));
     6. }
  9. Modify the **Log** method to add a new entry each time it is called. The following code shows how to achieve this.
     1. C#
     2. public void Log(string message, Category category, Priority priority)
     3. {
     4. document.Element("entries").Add
     5. (new XElement
     6. ("entry",
     7. new XElement("message", message),
     8. new XElement("category", category),
     9. new XElement("priority", priority)
     10. )
     11. );
     12. }
  10. Create a private method to save the log file in Isolated Storage. The private method will be executed each time logged is called. The following snippet shows the necessary code to save to the log file to Isolated Storage.
      1. C#
      2. private void SaveLogFile()
      3. {
      4. using (IsolatedStorageFile isoStore = IsolatedStorageFile.GetUserStoreForApplication())
      5. {
      6. using (IsolatedStorageFileStream isoStream =
      7. new IsolatedStorageFileStream(Constants.LogOutputName, FileMode.Create, isoStore))
      8. {
      9. this.document.Save(isoStream);
      10. }
      11. }
      12. }
  11. Call the **SaveLogFile** method after adding the entry in the **Log** method. The following code shows the **Log** method outcome.
      1. C#
      2. public void Log(string message, Category category, Priority priority)
      3. {
      4. document.Element("entries").Add
      5. (new XElement
      6. ("entry",
      7. new XElement("message", message),
      8. new XElement("category", category),
      9. new XElement("priority", priority)
      10. )
      11. );
      12. this.SaveLogFile();
      13. }
  12. Open the **WorkshopBootstrapper** class, located in the Shell project.
  13. Add the following using statement
      1. C#
      2. using Microsoft.Practices.Prism.Logging;
  14. Override the **CreateLogger method to return a new instance of the XmlLogger.** The resulting **Bootstrapper** is shown in the following code.
      1. C#
      2. public class WorkshopBootstrapper : UnityBootstrapper
      3. {
      4. protected override DependencyObject CreateShell()
      5. {
      6. Shell shell = new Shell();
      7. Application.Current.RootVisual = shell;
      8. return shell;
      9. }
      10. protected override IModuleCatalog CreateModuleCatalog()
      11. {
      12. return Microsoft.Practices.Prism.Modularity.ModuleCatalog.CreateFromXaml(
      13. new Uri("/Prism.Workshop;component/ModulesCatalog.xaml", UriKind.Relative));
      14. }
      15. protected override void ConfigureContainer()
      16. {
      17. RegisterTypeIfMissing(typeof(IUIMessagesService), typeof(UIMessagesService), true);
      18. base.ConfigureContainer();
      19. }
      20. protected override ILoggerFacade CreateLogger()
      21. {
      22. return new XmlLogger();
      23. }
      24. }
      25. }
  15. The next time the application is run, the **XmlLogger** will be used. The next task will help you retrieve the Logger outcome of the log file by reading it from Isolated Storage, and displaying it in the **Debug** window.

Task 2 – Retrieving the Logger Output

* 1. Open the **XmlLogger** class inside the Shell project.
  2. Add a new method to read the content of the Log file from the Isolated Storage, as shown in the following code snippet.
     1. C#
     2. public static string ReadLogFile()
     3. {
     4. string contents;
     5. using (IsolatedStorageFile isoStore = IsolatedStorageFile.GetUserStoreForApplication())
     6. {
     7. using (StreamReader reader =
     8. new StreamReader(isoStore.OpenFile(Constants.LogOutputName,
     9. FileMode.Open, FileAccess.Read)))
     10. {
     11. contents = reader.ReadToEnd();
     12. }
     13. }
     14. return contents;
     15. }
  3. Open the **App.xaml.cs** file in the Shell project.
  4. Add the following using statement to the file.
     1. C#
     2. using System.Diagnostics;
  5. Update the **Application\_Exit** method to retrieve the log file content from the Isolated Storage, and display them in the Debug window. The following code shows how to achieve this.
     1. C#
     2. private void Application\_Exit(object sender, EventArgs e)
     3. {
     4. Debug.WriteLine("LOGGER OUTPUT");
     5. Debug.WriteLine(XmlLogger.ReadLogFile());
     6. }
  6. Having finished the previous task, the output of the log file will be shown in the Debug window once you exit the application.

Task 3 – Verifying the log file output

* 1. Rebuild and run the solution.
  2. Close the application by clicking the browser’s **Close** button.
     1. 
  3. Open the **Output** window in Visual Studio. To do this, open the **View** menu and click **Output** (alternatively you can press **CTRL+W, O**). If necessary, select the option to Show output from **Debug**.
     1. 
  4. Maximize the **Output** window. You should be able to see the content of the log file, as shown in the following screenshot.
     1. 