



Microsoft[®] Virtual Labs

Advanced ASP.NET MVC

Microsoft[®]

Table of Contents

Advanced ASP.NET MVC	1
Exercise 1 Creating an Action Filter.....	2
Exercise 2 Implementing a Custom View Engine	4
Exercise 3 Creating a Custom Controller Factory	7
Exercise 4 Using Catch-All Routes	9

Advanced ASP.NET MVC

Objectives

After completing this lab, you will be better able to:

- Extend the ASP.NET MVC framework to create reusable behavior
- Customize how controllers are created
- Create a catch-all route

Scenario

This lab shows you how to extend the ASP.NET MVC framework to create reusable behavior that can be shared across controllers, how to customize how controllers are created, how to implement your own view engine, and how to create a catch-all route.

Estimated Time to Complete This Lab

60 Minutes

Computers used in this Lab



WS03_VisualStudio


The password for the **administrator** account on all computers in this lab is:
pass@word1

Exercise 1

Creating an Action Filter

Scenario

In this exercise, you will create an action filter and view your trace entry on a link.

Tasks	Detailed Steps
<p>Complete the following task on:</p>  <p>WS03_VisualStudio</p> <p>1. Creating an Action Filter</p>	<p>a. If necessary the Logon Credentials are user name administrator and the password is pass@word1 (p a s s w Z E R O r d !)</p> <p>b. In Visual Studio 2008, open the C:\labs\LAB 2 - ASP.NET MVC Advanced\ex01\Before\ActionFilterLab\ActionFilterLab.sln solution.</p> <p>c. Right-click the Filters folder and then click Add Class.</p> <p>d. Name the class “LogActionFilterAttribute.cs” and then click Add.</p> <p>e. Add the following using statement to the class:</p> <pre>using System.Web.Mvc;</pre> <p>f. Modify the class so that it inherits from ActionFilterAttribute.</p> <p>g. Add the following bolded code in order to implement an OnActionExecuting event handler:</p> <pre>public class LogActionFilterAttribute : ActionFilterAttribute { public override void OnActionExecuting(ActionExecutingContext context) { context.HttpContext.Trace.Write("Log Action Filter", "Before " + context.ActionMethod.Name); } }</pre> <p>h. In the Controllers directory, open the HomeController class.</p> <p>i. Add a using statement to import the ActionFilterLab.Filters namespace.</p> <pre>using ActionFilterLab.Filters;</pre> <p>j. Mark the action methods by using the LogActionFilter attribute. The resulting methods will look like the bolded code in the following example:</p> <pre>public class HomeController : Controller { [LogActionFilter] public ActionResult Index() { ViewData["Title"] = "Home Page"; ViewData["Message"] = "Welcome to ASP.NET MVC!"; return RenderView(); } [LogActionFilter] public ActionResult About() { ViewData["Title"] = "About Page";</pre>


Tasks	Detailed Steps
	<pre data-bbox="508 226 911 317"> return RenderView(); } }</pre> <ul style="list-style-type: none"><li data-bbox="508 323 1149 352">k. Press CTRL+F5 to run the Web site without debugging.<li data-bbox="508 363 1393 426">l. Click the About Us link. The About view is displayed, which includes a link to the trace output.<li data-bbox="508 436 899 466">m. Click the link to the trace output.<li data-bbox="508 476 1406 539">n. In the row that contains Home/About in the File column, click the View Details link.<li data-bbox="508 550 1422 613">o. You can now see your trace entry with the category Log Action Filter in the trace output.

Exercise 2

Implementing a Custom View Engine

Scenario

In this exercise you will create an alternative view engine that uses XSLT templates to generate a view.

Tasks	Detailed Steps
<p>Complete the following task on:</p>  <p>WS03_VisualStudio</p> <p>1. Implementing a Custom View Engine</p>	<p>a. In Visual Studio 2008, open the C:\labs\LAB 2 - ASP.NET MVC Advanced\ex02\Before\CustomViewEngineLab\CustomViewEngineLab.sln solution.</p> <p>b. Right-click the CustomViewEngineLab folder, then click Add, and then click Add Class.</p> <p>c. Name the class “XsltViewLocator.cs” and then click Add.</p> <p>d. Add the following using statement to the class:</p> <pre>using System.Web.Mvc;</pre> <p>e. Modify the class so that it inherits from ViewLocator. This enables the class to use fallback logic when locating a view, such as checking the Views/Controller directory before checking the Views/Shared directory.</p> <p>f. Add the following bolded code to implement the class:</p> <pre>public class XsltViewLocator : ViewLocator { public XsltViewLocator() { base.ViewLocationFormats = new string[] { "~/Views/{1}/{0}.xslt", "~/Views/Shared/{0}.xslt"}; base.MasterLocationFormats = new string[] { }; //No xslt master templates... } }</pre> <p>g. Add a class named “XsltViewEngine” to the CustomViewEngineLab project.</p> <p>h. Make sure that the file includes the following using statements at the top:</p> <pre>using System.IO; using System.Xml; using System.Xml.Xsl; using System.Xml.Serialization; using System.Xml.XPath; using System.Web; using System.Web.Mvc; using System.Web.Hosting;</pre> <p>i. Modify the class so that it inherits from IViewEngine.</p> <p>j. Add the following bolded code to implement the RenderView method.</p> <p><i>Note: You must import several namespaces; these will be marked by smart tags.</i></p> <pre>public class XsltViewEngine : IViewEngine {</pre>

Tasks	Detailed Steps
	<pre data-bbox="506 195 1386 1077"> private IViewLocator viewLocator = new XsltViewLocator(); public void RenderView(ViewContext viewContext) { string templatePath = viewLocator.GetViewLocation(viewContext, viewContext.ViewName); templatePath = VirtualPathUtility.ToAbsolute(templatePath); Stream xsltStream = VirtualPathProvider.OpenFile(templatePath); XsltCompiledTransform xslt = new XsltCompiledTransform(); xslt.Load(XmlReader.Create(xsltStream)); MemoryStream stream = new MemoryStream(); XmlWriter writer = XmlWriter.Create(stream); XmlSerializer serializer = new XmlSerializer(viewContext.ViewData.GetType()); serializer.Serialize(stream, viewContext.ViewData); stream.Position = 0; XPathDocument doc = new XPathDocument(stream); xslt.Transform(doc, null, viewContext.HttpContext.Response.Output); } } </pre> <p data-bbox="506 1087 1425 1178">k. The RenderView method of this view engine converts the view data into XML by using an XmlSerializer instance. The method then transforms the data by using the XSLT template that is found by using the custom XsltViewLocator instance.</p> <p data-bbox="506 1188 1260 1220">l. Right-click the Views/Home folder and then click Add New Item.</p> <p data-bbox="506 1230 1393 1287">m. Under Templates, select XSLT File, name the file "Index.xslt", and then click Add.</p> <p data-bbox="506 1297 1321 1329">n. Replace the existing content in the file with the following XML markup:</p> <pre data-bbox="506 1339 1393 1904"> <?xml version="1.0" encoding="utf-8"?> <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:msxsl="urn:schemas-microsoft-com:xslt" exclude- result-prefixes="msxsl" > <xsl:output method="html" indent="yes"/> <xsl:template match="*"> <html> <head> <title>Comic Books</title> </head> <body> <xsl:apply-templates select="//ComicBook" /> </body> </html> </template> </pre>


Tasks	Detailed Steps
	<pre data-bbox="506 193 1398 541"> </html> </xsl:template> <xsl:template match="ComicBook"> <xsl:value-of select="Title"/> Issue #<xsl:value-of select="Issue"/> </xsl:template> </xsl:stylesheet> </pre> <p data-bbox="506 548 1414 604">o. Expand the Controllers folder, then open the HomeController.cs file and add the following using statement:</p> <pre data-bbox="506 617 1398 646">using CustomViewEngineLab;</pre> <p data-bbox="506 653 1382 709">p. Insert the following line of code in the Index method just before the call to the RenderView method:</p> <pre data-bbox="506 722 1398 751">this.ViewEngine = new XsltViewEngine();</pre> <p data-bbox="506 758 922 787">q. Press CTRL+F5 to run the project.</p> <p data-bbox="506 793 1422 823">r. A list of comic books is displayed, which is rendered by using the XSLT template.</p>

Exercise 3

Creating a Custom Controller Factory

Scenario

In this exercise you will create a custom controller factory. You will reuse the behavior of the existing `DefaultControllerFactory` class, but you will wrap the controller instance that it returns with a custom controller that intercepts all requests to the internal controller.

Tasks	Detailed Steps
<p>Complete the following task on:</p>  <p>WS03_VisualStudio</p> <p>1. Creating a Custom Controller Factory</p>	<ol style="list-style-type: none"> a. In Visual Studio 2008, open the <code>C:\labs\LAB 2 - ASP.NET MVC Advanced\ex03\Before\CustomControllerFactoryLab\CustomControllerFactoryLab.sln</code> solution. b. Right-click the Controllers folder and then click Add New Item. c. Under Templates, scroll down to My Templates and select the MVC Controller Class. d. Name the class “InterceptingController.cs” and then click Add. e. Replace the contents of the class with the following bolded code. This code implements a custom controller that intercepts calls to other controllers. <pre data-bbox="506 940 1393 1696"> public class InterceptingController : Controller { private IController controller; public InterceptingController(IController interceptedController) { this.controller = interceptedController; } protected override void HandleUnknownAction(string actionName) { if (String.Equals("about", actionName, StringComparison.InvariantCultureIgnoreCase)) { Response.Write("<h1>Sorry, 'About' actions are no longer allowed.</h1>"); return; } ControllerContext context = new ControllerContext(this.ControllerContext, this.controller); this.controller.Execute(context); } } </pre> f. Although the wrapper could perform many tasks when intercepting actions, the example class simply blocks the About action of any controller and displays a message to that effect. g. Right-click the Controllers folder and then click Add Class. h. Select MVC Controller Class.


Tasks	Detailed Steps
	<p>i. Name the new class "InterceptingControllerFactory.cs" and then click Add.</p> <p>j. Replace the class definition in the file with the following class definition. This is a controller factory class.</p> <pre data-bbox="508 304 1396 646">public class InterceptingControllerFactory : DefaultControllerFactory { protected override IController GetControllerInstance(Type controllerType) { IController controller = base.GetControllerInstance(controllerType); return new InterceptingController(controller); } }</pre> <p>k. The controller factory class inherits from DefaultControllerFactory and overrides the GetControllerInstance method. The code obtains the controller returned by the base.GetControllerInstance method and wraps it with the InterceptingController instance.</p> <p>l. In the Global.asax file, add the following line to the Application_Start method:</p> <pre data-bbox="508 835 1396 892">ControllerBuilder.Current.SetControllerFactory(new InterceptingControllerFactory());</pre> <p>m. This replaces the default controller factory with the custom one you have built.</p> <p>n. Press CTRL+F5 to run the project.</p> <p>o. Click the About Us link to see the intercepting controller in action. You see the message that you defined in step e.</p>

Exercise 4

Using Catch-All Routes

Scenario

In this exercise you will use the catch-all routes feature.

Tasks	Detailed Steps
<p>Complete the following task on:</p>  <p>WS03_VisualStudio</p> <p>1. Using Catch-All Routes</p>	<p>a. In Visual Studio 2008, open the C:\labs\LAB 2 - ASP.NET MVC Advanced\ex04\Before\CatchallRouteLab\CatchallRouteLab.sln solution.</p> <p>b. In the Global.asax file, add the following route to the beginning of the RegisterRoutes method:</p> <pre>routes.MapRoute("catchall", "music/{*catchall}", new { controller = "Home", action = "About" });</pre> <p>c. Notice that the catchall parameter uses an asterisk to indicate that it will match any number of segments. The code also sets the defaults so that URLs that match this route will call the About method of the HomeController class.</p> <p>d. In the Controllers directory, open the HomeController class and replace the About method with the following code:</p> <pre>public ActionResult About(string catchall) { var dictionary = new Dictionary<string, string>(); if (catchall != null) { if (catchall.EndsWith("/")) catchall = catchall.Substring(0, catchall.Length - 1); string[] catchallSegments = catchall.Split('/'); // Ensure an even number of segments. if (catchallSegments.Length % 2 != 0) throw new ArgumentException("Need to have an even # of segments for the catch-all", "catchall"); for (int i = 0; i < catchallSegments.Length; i += 2) { dictionary.Add(catchallSegments[i], catchallSegments[i + 1]); } return RenderView("About", dictionary); } }</pre> <p>e. This code parses the catchall parameter (which will contain the rest of the raw URL) and converts the value to a dictionary.</p> <p>f. In the Views\Home folder, open the About.aspx.cs file and make sure that it inherits from ViewPage<Dictionary<string, string>>, as shown in the following</p>

Tasks	Detailed Steps
	<p>example:</p> <pre data-bbox="505 233 1398 359">public partial class About : ViewPage<Dictionary<string, string>> { }</pre> <p>g. This changes the About view to be a strongly typed ViewPage class.</p> <p>h. In the About.aspx page, add the following code to print out the keys and values of the strongly typed ViewData dictionary:</p> <pre data-bbox="505 474 1398 632"> <% foreach(string key in this.ViewData.Keys) { %> <%= key %>: <%= this.ViewData[key] %> <% } %> </pre> <p>i. Close About.aspx and press CTRL+F5 to run the project.</p> <p>j. You should be on the Home view. Try clicking the About links. You see a list of name/value pairs that correspond to the URL.</p>