Module 04 Lab 02 Worksheet

Providing a Consistent UI with Layouts

# Overview

|  |  |
| --- | --- |
| In ASP.NET development, a *layout* is used to define shared Web page elements such as scripts and stylesheets which can then be referenced by any view in the app. This keeps us from having to duplicate our code when we set up a new view.  An ASP.NET Web site generally has one layout file, called by default \_Layout.cshtml. This file is a mixture of both Web elements such as HTML, CSS and Javascript, and C# code. The default location for layouts in your project folder is in the Views/Shared folder.  Web apps don't *require* a layout but they can use multiple layouts for different views. There are just a few rules for using a layout with a view:   * *The view must specify which layout it's using by setting the Layout property. For example*   @{  Layout = "\_Layout";  }  You can specify a relative path or a full path.   * *Every layout must include a call to the RenderBody() function. This indicates where the content of the view will be rendered. For example*   <div class="container body-content">  @RenderBody()  <hr />  <footer>  <p>&copy; 2016 - MyWebApp</p>  </footer>  </div>   * *You can also organize where specific page elements are placed by using sections. The view defines a section*:   @section Scripts {  <script type="text/javascript" src="/scripts/main.js"></script>  }  The layout file can then use RenderSection to call it.  @RenderSection("scripts", required: false)  Other than the embedded C#, a layout file looks almost exactly like a normal HTML file.  This is enough to get started with, but you can do quite a bit more with layouts. For example, you can have some code run before a view is rendered. | 0 |
| Let's start by creating the simplest possible layout just so we can get a bead on how they work. We're going to do it in a sneaky way to keep us from having to write a lot of HTML scaffolding.  Create a file called \_Layout.cshtml under the Views/Shared folder.  Now we just have to add two items to turn this into a layout file.   * Open \_Layout.cshtml for editing. * Change the title to Default Layout:   <title>Default Layout</title>  (Later we'll find out how to generate a custom title depending on the view.)   * Add a call to RenderBody() inside the body tag.   <body>  @RenderBody()  </body>  That's it. We now have a very simple layout file.  Run your application and examine your pages. What changed? (5 pts.)  I don’t think anything changed. | 5 |
| A layout isn't much good unless we have a view that uses it. Though we ultimately want all three views to use this layout (at least at first), we'll start with one to make sure it works.   * Open About.cshtml for editing. * At the top of the file, insert the call to the default layout we created earlier.   @{  Layout = "\_Layout";  }   * Save your changes and start up the app with dotnet run. * Open <http://localhost:5000/home/About>   What does the title bar for this page say? (5 pts.) “Default Layout”   * Modify the other two views so that they use our default layout, then test each change as above. | 5 |
| Now that we've got our basic layout working, let's do some more customization with the ViewData property. This lets you define objects in a view and call on them in your layout. We're going to test this by giving each of our pages a custom title that will be rendered with our layout rather the generic Default Layout.   * Open Index.cshtml for editing. * In the section where you call your layout, add a line containing   ViewData["Title"] = "Home Page";  This creates a dictionary object with the contents Home Page and accessed by the key Title.   * Open \_Layout.cshtml for editing. * Modify the title tag to read   <title>@ViewData["Title"]</title>   * Save your changes, restart the app and check the application home page.   What does the title bar say? ( 5 pts.) “Home Page”   * Assuming everything is working, modify About.cshtml and Contact.cshtml to have titles of "About Us" and "Contact Us", respectively. Test the changes.   We now have a way to insert simple custom content into our Web app views.  Before we continue commit your changes to version control with an appropriate message. | 5 |
| We can also use the ViewData property to pass data from a controller.   * Open HomeController.cs for editing. * Modify the Contact() method to add a ViewData entry called Message:   public IActionResult Contact()  {  ViewData["Message"] = "Contact Us";  return View();  }  Do the same for About() and set the Message to "About Us".   * In About.cshtml, modify the line   <h1>This is an empty About page.</h1>  To read  <h1>@ViewData["Message"]</h1>   * In Contact.cshtml, modify the line   <h1>Here is my contact information</h1>  To  <h1>@ViewData["Message"]</h1>   * Restart your app with dotnet run   What does it say on your About page? ( 5 pts.)  Says the new message  What does it say on your Contact page? ( 5 pts.)  Says the new message  Before continuing, commit your changes with an appropriate message. | 10 |
| We will use a third-party Web framework called Bootstrap to help us create a nice layout for our app. Since Bootstrap is not included with ASP.NET Core, it is installed and managed using the bower package manager.  In the Integrated Terminal, run the command  bower list  To list the installed Web packages. Is Bootstrap present?. (5 pts.) | 5 |
| Open \_Layout.cshtml for editing.  In the header section (between the <head> and </head> tags), add the line  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css">  (This should be all on one line.)  (**Note**: "~" is the relative path for wwwroot. Using a full path is considered poor coding practice, particularly if you plan to deploy your code to a different machine. In addition, if we change the location of our static files, our app won't break.)  Start the app if it isn't already running and check your pages. What changed? (5 pts.) | 5 |
| The basic element of a Bootstrap page is the container. Let's add one to our layout and see what happens.   * In \_Layout.cshtml, wrap the RenderBody() call in a container:   <div class="container">  @RenderBody()  </div>  Start your app if necessary and check your pages. What changed? (5 pts.) | 5 |
| Now let's add a navigation bar, one step at a time.   * Just below the <body> tag in \_Layout.cshtml, add the following:   <div class="navbar navbar-default">  <div class="container-fluid">  </div>  </div>  Refresh your application page. What changed? (5 pts.) | 5 |
| Add the following inside of your container-fluid in \_Layout.cshtml:  <div class="navbar-header">  <a class="navbar-brand" href="/">HelloApp</a>    </div>  (Replace HelloApp with the name of your application.)  Refresh your page. What changed? (5 pts.) | 5 |
| Let's add some menu items to our navigation bar. After the navbar-brand line, add the following:  <ul class="nav navbar-nav">  <li><a href="/">Home</a></li>  <li><a href="/home/About">About</a></li>  <li><a href="/home/Contact">Contact</a></li>  </ul>  Refresh the page. What changed? (5 pts.)  Click on the labels. What happens? (5 pts.) | 10 |
| That's okay for now. Let's just clean up our About and Contact pages before we go on to the next part in this series.   * About page - Replace the text "I'll put more in later" with a short description of this app. * Contact page - Replace the line   <p>Call me!</p>  With a street address and email (you can make one up):  <address>  1234 Any Street <br>  Mytown, CO 54321 <br>  USA <br>  Email: <a href="mailto:webmaster@nospam.com">Webmaster</a>  </address>   * Refresh the respective pages and make sure the new content comes up correctly. * Save and commit your project changes then push them out to Azure * The lab is complete. | 0 |
| **Total** |  |

# Summary

This lab introduced you to two ways to easily manage the look-and-feel of your Web application - ASP.NET layout files and the third party Web framework Bootstrap.

Complete this worksheet and submit it to your instructor along with the public URL for your application.