## Developing Provider-hosted Add-ins using MVC

**Lab Time**: 60 minutes

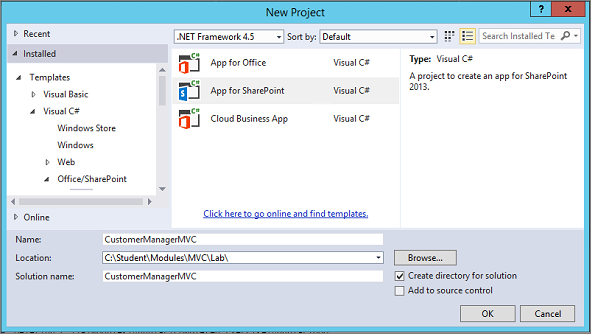
**Lab Folder**: C:\Student\Modules\MVC\Lab

**Lab Overview**: In this lab you will create a Provider-Hosted Add-in using MVC5. You will investigate the capabilities of the Model-View-Controller pattern, and create a simple RESTful service that wraps a database.

### Exercise 1: Creating a Provider-Hosted Add-in that uses the MVC Framework

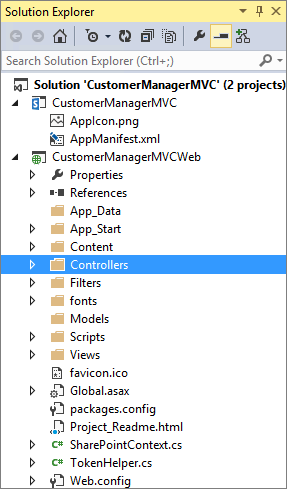
In this exercise you create a new Provider-Hosted Add-in based on the ASP.NET MVC Framework.

1. Launch **Visual Studio 2015**.
2. Create the new solution in Visual Studio 2015:
   1. In Visual Studio select **File 🡪 New 🡪 Project**.
   2. In the **New Project** dialog:
      1. Select **Templates 🡪 Visual C# 🡪 Office/SharePoint 🡪 Apps**.
      2. Click **App for SharePoint**
      3. Name the new project **CustomerManagerMVC**.
      4. Add the new project into the folder at **C:\Student\Modules\MVC\Lab**.
      5. Click **OK** to create the project which will lead Visual Studio to display the **New App for SharePoint** wizard.



* 1. In the **New App for SharePoint** wizard:
     1. Enter the URL for your Office 365 developer site.
     2. Select **Provider-Hosted** as the hosting model.
     3. Click **Next**.

1. Specify the web project type as **ASP.NET MVC Web Application** and click **Next**.
   1. Click **Finish** to complete the wizard and create the new Provider-Hosted Add-in project.
2. When the New app for SharePoint wizard completes it work, you should see a new Visual Studio solution which contains two projects. The top project name **CustomerManagerMVC** is the app project while the bottom project named **CustomerManagerMVCWeb** is an ASP.NET project that can be used to implement the remote web.



1. In the **CustomerManagerMVC** project open the **AppManifest.xml** file by double clicking IT.
   1. Set the **Title** to **Customer Manager MVC**
   2. Save and close the **AppManifest.xml** file.
2. Examine the HomeController.cs file in your CustomerManagerMVCWeb project
   1. In this project expand the Controllers folder and double click on the file named **HomeController.cs** to open the C# source file in a Code View window.
   2. Examine the Controller class that Visual Studio has generated for you. You should be able to see that there is a class named **HomeController** that inherits from the **Controller** class. The class contains an action method named **Index**.
   3. An important aside; note the use of the **[SharePointContextFilter]** annotation on the **Index()** method. This is defined in the **CustomerManagerMVCWeb** project 🡪 **Filters** 🡪 **SharePointContextFilterAttribute.cs** file which makes use of the very handy **SharePointContext.cs** code file (located in the root of the CustomerManagerMVCWeb project) to automagically handle user checks (i.e. this handles the **OAuth** calls, app access token validation, and also handles the storing of this info into the user session. Every call to a Controller View painted with this attribute will pull what is stored in the user session with the **SPHostUrl** and automatically redirect the user to the login page if needed.

public class HomeController : Controller {

[SharePointContextFilter]

public ActionResult Index() {

User spUser = null;

var spContext = SharePointContextProvider.Current.GetSharePointContext(HttpContext);

using (var clientContext = spContext.CreateUserClientContextForSPHost()) {

if (clientContext != null) {

spUser = clientContext.Web.CurrentUser;

clientContext.Load(spUser, user => user.Title);

clientContext.ExecuteQuery();

ViewBag.UserName = spUser.Title;

}

}

...

}

* 1. Modify the implementation of the **Index** method to match the following code listing.

public ActionResult Index()

{

// create named properties in ViewBag

ViewBag.welcomeMessage = "Hello from the ASP.NET MVC Framework";

ViewBag.hostWebUrl = Request.QueryString["SPHostUrl"];

// return default view for this action

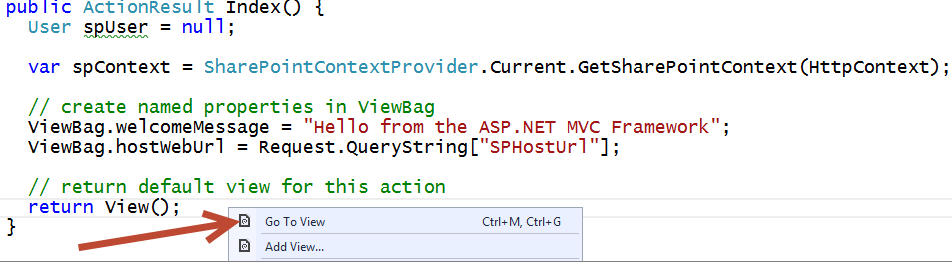
return View();

}

* 1. **Save** your changes to **HomeController.cs**.

The ViewBag object is used to hold arbitrary data that you wish to pass from a controller to the view. In more advanced designs the ViewBag is replaced with a ViewModel object that is strongly typed.

1. Examine the default view for the **Index** action method.
   1. Right-click the white area inside the **Index** method and select **go to View**.



* 1. The view for the **Index** action named **Index.cshtml** should be open inside a Code View window.
  2. Examine the view code that Visual Studio has added to **Index.cshtml**.

@{

ViewBag.Title = "Home Page";

}

<div>

…

</div>

An MVC View like this renders a partial page and generally works in concert with a master view (typically named **\_Layout.cshtml**). This relationship in the MVC framework is similar to the relationship between a content page and a master page in the Web Forms framework,

* 1. Delete all the existing content inside **Index.cshtml** file and replace it with the code shown in the following code listing.

@{ ViewBag.Title = "Customer Manager MVC"; }

<div>

<a href="@ViewBag.hostWebUrl">Host Web</a>

</div>

<h2>App Start Page</h2>

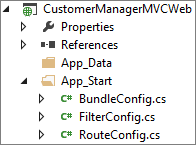
<p>@ViewBag.welcomeMessage</p>

* 1. **Save** your work.

You have just created a simple controller and view which means you are almost ready to test your app in the Visual Studio debugger. However, you still need to take a few more steps to ensure that the start page of the SharePoint app points to a valid URL within the routing scheme for the MVC app.

The MVC Framework uses “routes” to direct HTTP calls to “controller” methods. The basic form of the route is the base URL followed by the controller name followed by the method name followed by any parameters. {controller}/{method}/{parameter}. In the next step, you will ensure that the properly configured to accept requests based on a URL that matches the root URL of the site by examining/configuring the MVC app's default controller.

1. Examine the **HomeController** for the default controller to the MVC app's routing scheme.
   1. In the Solution Explorer, expand the **App\_Start** folder inside the **CustomerManagerMVCWeb** project and locate the C# source file named **RouteConfig.cs**.



* 1. Double-click on **RouteConfig.cs** to open this source file in a Code View window. You should see a RouteConfig class that contains a method named **RegisterRoutes**. Inside the implementation of the **RegisterRoutes** method, there is a call to the **MapRoute** method. You should be able to see that that code generated by Visual Studio configures a controller named **"Home"** as the default controller for the MVC app's top-level routing scheme.

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

* 1. Close **RouteConfig.cs**.

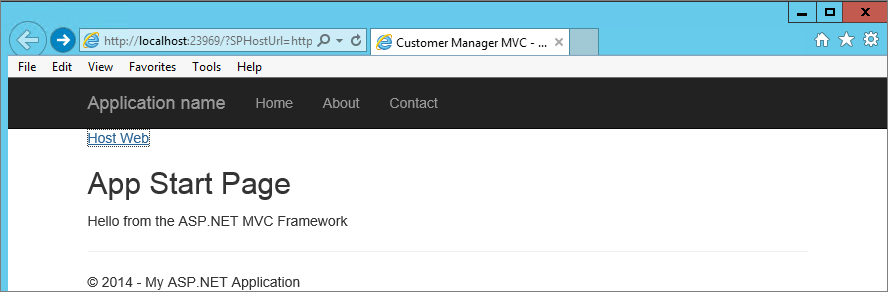
Note: If you encounter a situation where you name the controller something other than **"Home"** you will need to alter the default routing scheme in **RouteConfig.cs**. As you can see it is easy to reconfigure the default controller in scenarios where you need this flexibility.

1. Configure the start page in the SharePoint app.
   1. In the Solution Explorer, expand the **CustomerManagerMVC** project node.
   2. Locate and double-click **appManifest.xml** to open the app manifest file in the app manifest designer.
   3. Check and if needed, change the **Start page** to **CustomerManagerMVCWeb/**.



* 1. Save and close **appManifest.xml**.

1. Test your work by running the app in the Visual Studio debugger.
   1. Press **F5** to begin debugging.
   2. If you are prompted with a SharePoint page that asks whether you to trust the app, click **Trust it**.
   3. Once the app has been installed, you should be redirected to the app's start page. Verify that the start page for the app displays properly. The start page for your app should look like the page in the following screenshot.



* 1. Click on the **Host Web** link and verify that you can click on it to navigate back to the host web.
  2. When you are done, close the browser window to end the debugging session and return to Visual Studio.

Now you will take a different approach in your MVC app for reading the **SPHostUrl** query string parameter. More specifically, you will leverage the ability of the MVC Framework to automatically map incoming query string parameters sent over HTTP to named parameters you define inside your action method using C#.

1. Leverage the MVC Framework's ability to read query string parameters when handling requests from SharePoint.
   1. Currently, there is code in the **HomeController** class which explicitly reads the **SPHostUrl** query string property value using the **QueryString** collection property of the ASP.NET **Request** object.

ViewBag.hostWebUrl = Request.QueryString["SPHostUrl"];

* 1. Begin by modifying the parameter list for the **Index** method to define a single string parameter named **SPHostUrl**.

public ActionResult Index(string SPHostUrl)

{

// ...

}

* 1. Next, modify the code in the **Index** method that assigns a value to **hostWebUrl** property of the **ViewBag** object. Now, you can just use the **SPHostUrl** parameter defined by the **Index** method.

public ActionResult Index(string SPHostUrl)

{

// ...

// Delete this -> ViewBag.hostWebUrl = Request.QueryString["SPHostUrl"];

ViewBag.hostWebUrl = SPHostUrl;

// ...

}

1. Add this same line to the other two methods in the HomeController class, as shown below, (i.e. About and Contact) as we will need this in a bit.

public ActionResult About(string SPHostUrl) {

ViewBag.Message = "Your application description page.";

ViewBag.hostWebUrl = SPHostUrl;

return View();

}

public ActionResult Contact(string SPHostUrl) {

ViewBag.Message = "Your contact page.";

ViewBag.hostWebUrl = SPHostUrl;

return View();

}

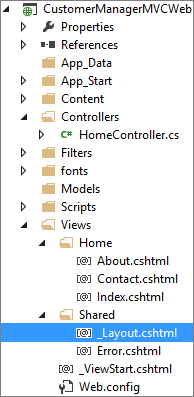
1. Test your work by running the app in the Visual Studio debugger.
   1. Press **F5** to begin debugging.
   2. Once the app has been installed, you should be redirected to the app's start page.
   3. Verify that hyperlink which links to the host web is still working properly.
   4. When you are done, close the browser window to end the debugging session and return to Visual Studio.

### Exercise 2: Creating a Multipage User Interface in an MVC App

Over the next few steps, you will examine an **About** page and a **Contact** page in the MVC app as well as examine the links for a navigation scheme which allows the user to move from page to page. When you want to add a new page to an MVC app, it typically involves adding a new action method to a controller class and then adding a view to show the results of that action method.

When you are working in a multi-page app, it makes sense to define a common layout for all the app's pages at one time. If you were using the ASP.NET Web Forms framework, you would use a master page to accomplish this goal. With the MVC framework, you use shared views to achieve the same goal. Before you add any more pages to the MVC app, you will first work on the primary shared view file for an MVC app which is named **\_Layout.cshtml**.

1. Modify the shared view file for the **CustomerManagerMVCWeb** project.
   1. In Solution Explorer, locate the shared view file named **\_Layout.cshtml** which can be found inside the **Views** folder inside a nested child folder named **Shared**.



* 1. Double click on **\_Layout.cshtml** to open this view in a Code View window inside Visual Studio. The following listing shows what Visual Studio provides in this file by default when you create an MVC Provider Hosted App.

<!DOCTYPE html>

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>@ViewBag.Title - My ASP.NET Application</title>

@Styles.Render("~/Content/css")

@Scripts.Render("~/bundles/modernizr")

</head>

<body>

<div class="navbar navbar-inverse navbar-fixed-top">

<div class="container">

<div class="navbar-header">

<button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

@Html.ActionLink("Application name", "Index", "Home", null, new { @class = "navbar-brand" })

</div>

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav">

<li>@Html.ActionLink("Home", "Index", "Home")</li>

<li>@Html.ActionLink("About", "About", "Home")</li>

<li>@Html.ActionLink("Contact", "Contact", "Home")</li>

</ul>

</div>

</div>

</div>

<div class="container body-content">

@RenderBody()

<hr />

<footer>

<p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>

</footer>

</div>

@Scripts.Render("~/bundles/jquery")

@Scripts.Render("~/bundles/bootstrap")

@Scripts.Render("~/bundles/spcontext")

@RenderSection("scripts", required: false)

</body>

</html></html>

* 1. This file is the primary shared view container that is used to display the other views in this application. Examine this file and locate the **<body>** section.

Note that the Application’s home page is referenced by the @Html.ActionLink in the navigation bar in two manners (First as **Application Name**, and then again as **Home**). The two additional pages are Referenced in subsequent @Html.ActionLink entries (About and Contact). In each Case you are passing in the Text you would like to display first, the actionName (or page you would like to display) second, and third the routeValues (i.e. the object that contains all the parameters for the given routes [which in our case is the same for all pages “Home”]). The reason you are doing this is to ensure that the navigation links are contained in a manner that allows for their use (in a location relative format) anywhere the App might be deployed.

* 1. In the <body> section locate the **@Html.ActionLink(“Application Name”,**…. code

<body>

<div class="navbar navbar-inverse navbar-fixed-top">

<div class="container">

<div class="navbar-header">

<button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

@Html.ActionLink("Application name", "Index", "Home", null, new { @class = "navbar-brand" })

</div>

…

* 1. We already have a link to the home page for our Application (**Home**) so we do not need this one as well… let’s replace it with a link to the Host Web as shown below; i.e. Delete **@Html.ActionLink(“Application name”…)** and replace it with:

<a href="@ViewBag.hostWebUrl" class="navbar-brand">Host Web</a>

* 1. Back on the **Index.cshtml** file we can now delete **the <div> <a href=”@ViewBag.hostWebUrl”>Host Web</a> </div>** html as this is redundant. (Note: this file should still be open, but if not you can open it from the **CustomerManagerMVCWeb 🡪 Views 🡪 Home** folder).
  2. Be sure to save your change to the **Index.cshtml** file and then return to the **\_Layout.cshtml** file for the next step.
  3. In the **\_Layouts.cshtml** modify the **body** section to match the following code listing (Note: you only need to modify the items bolded in black). Make sure you match the casing for HTML element the IDs (e.g. **topHeader** and **contentBody**) because they will be referenced by a CSS file which uses this casing.

<body>

//...

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav">

<li>@Html.ActionLink("Home", "Index", "Home")</li>

<li>@Html.ActionLink("About", "About", "Home")</li>

<li>@Html.ActionLink("Contact", "Contact", "Home")</li>

</ul>

</div>

</div>

</div>

<header id="topHeader">

<h2>Customer Manager MVC App</h2>

</header>

<div class="container body-content" id="contentBody">

@RenderBody()

<hr />

<footer>

<p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>

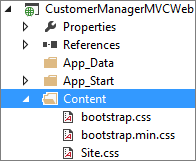
</footer>

</div>

//...

</body>

1. Improve the user interface by adding CSS styles and a background image to your shared page layout.
   1. In the **CustomerManagerMVCWeb** project, expand the **Content** folder. You should be able to see that this contains a CSS file named **Site.css**.



Note that Visual Studio and the MVC Framework have already configured the shared view named **\_Layout.cshtml** to link to **Site.css**. Any styles you add to **Site.css** will affect all pages that use the shared view.

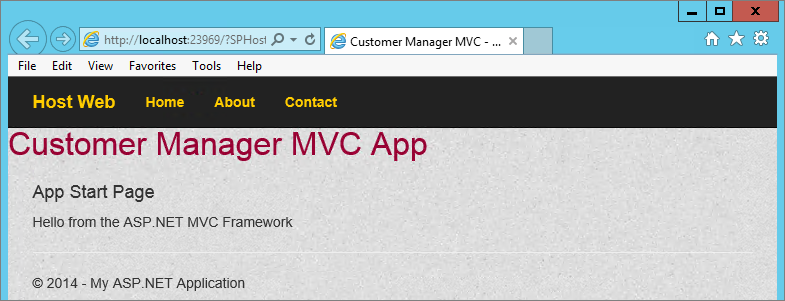
* 1. Inside the Content folder, create a child folder name **img**.
     1. Right click on the **Content** folder and select **Add 🡪 New Folder** from the context menu, name the folder **img**
  2. In the Windows Explorer, look inside the folder at **C:\Student\Modules\MVC\Lab\StarterFiles** and locate the images file named **background.png**.

Return to Visual Studio and add the **background.png** image file into the **img** folder.  
(Note: the easiest way to add a file to Visual Studio is by dragging it from the source folder in Explorer View into the Solution Explorer destination folder in Visual Studio).



* 1. Back in the Windows Explorer, look inside the folder at **C:\Student\Modules\MVC\Lab\StarterFiles** and locate a text file named **Site.css.additions.txt**. Open this text file in **Notepad.exe** and copy its content to the Windows clipboard.
  2. Return to Visual Studio and open the CSS file named **Site.css**. Place your cursor at the end of Site.css and paste the contents of the Windows clipboard. Your shared view should now be displayed with the extra CSS styles from **Site.css.additions.txt**.
  3. Save and close the Site.css file and Close the Site.css.additions.txt file.
  4. In the same **Content** folder in the project, open the bootstrap.cs file and Find (Ctrl+f) the following text “max-width: 1170px”
     1. Modify this to read “max-width: **1250px**”
     2. Save and close the bootstrap.css file.  
        (Note: this change is needed at the end of the final lab to lay out the data table correctly)

1. Test your work by running the app in the Visual Studio debugger.
   1. Using the **Visual Studio Build** Menu select **Rebuild Solution** to ensure that your changes are picked up by the compiler.
   2. Press **F5** to begin debugging.
   3. If you are prompted with a SharePoint page that asks whether you to trust the app, click **Trust it**.
   4. Once the app has been installed, you should be redirected to the app's start page. Verify that the start page for the app displays properly. The start page should now include the user interface elements and CSS styling that have been applied to the shared view. When finished close the Internet Explorer window to return to Visual Studio.



1. Examine the **About** and **Contact** pages in the MVC app.
   1. Open the C# source file with the **HomeController** class named **HomeController.cs**.  
      (Note: this should still be open from earlier, if not open it from the **CustomerManagerMVCWeb** project **Controllers** folder)
   2. Examine the action methods in the **HomeController** class named **About** and **Contact** which matches the following code listing.

public ActionResult About() {

ViewBag.Message = "Your application description page.";

return View();

}

public ActionResult Contact() {

ViewBag.Message = "Your contact page.";

return View();

}

* 1. Right-click the white area inside the **About** method and select **Go To View**.
  2. Examine the contents of **About.cshtml**. Modify the **<p>** tag so that it matches the code below.

@{

ViewBag.Title = "About”;

}

<h2>@ViewBag.Title </h2>

<h3>@ViewBag.Message</h3>

<p>This app is currently being written and debugged by yours truly.</p>

* 1. Save and close the About.cshtml file.

1. Return to the **HomeController.cs** file and right-click the white area inside the **Contact** method and select **Go To View**.
   1. Examine and modify the contents of **Contact.cshtml** to match the following code listing.

@{

ViewBag.Title = "Contact the App Vendor";

}

<h2>@ViewBag.Title.</h2>

<h3>@ViewBag.Message</h3>

<address>

One Microsoft Way<br />

Redmond, WA 98052-6399<br />

<abbr title="Phone">P:</abbr>

425.555.0100

</address>

<address>

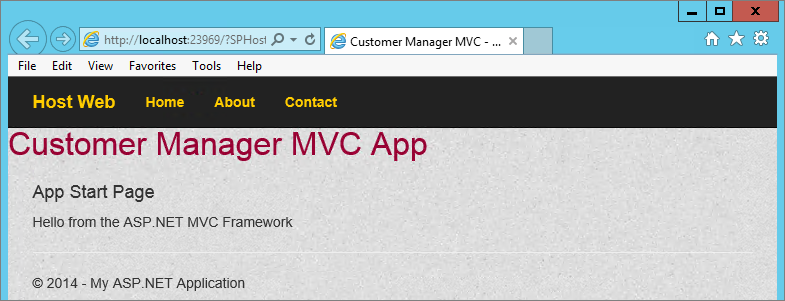
<strong>Support:</strong> <a href="mailto:Support@example.com">Support@example.com</a><br />

<strong>Marketing:</strong> <a href="mailto:Marketing@example.com">Marketing@example.com</a>

</address>

* 1. Save and close the Contact.cshtml file.

1. Test your work by running the app in the Visual Studio debugger.
   1. Press **F5** to begin debugging.
   2. Once the app has been installed, you should be redirected to the app's start page. Test the links for the three pages to ensure you can navigate between all three pages.



### Exercise 3: Tracking SharePoint State using an ASP.NET Session Object

You will now examine how host web URL’s are tracked across requests. When SharePoint redirects a user to the app start page, your code is reading the value of the query string parameters and then writing it into an ASP.NET Session object. This type of state management design can result in greater maintainability because you don't have to worry about customizing the action links in your views to propagate the **SPHostUrl** query string parameters. It also has the side effect of producing cleaner-looking URLs.

1. Return to Visual Studio and examine how we are currently tracking the HostURL by examining the **HomeController.cs** class (Should still be open). You will note that we previously configured all three Methods (**Index**, **About**, and **Contact**) to take a string **SPHostURrl** and store this in the **ViewBag.hostWebUrl**. This is one way to ensure that this property is available between page views. A downside of using this is that with every additional view you add to your App you will need to remember to modify each corresponding HomeController method (that is used to return the associated view) to add in the calls to the ViewBag. In MVC5 there is another better way to access this information that does not require the use of the ViewBag, so let’s remove these method calls in preparation for using the “improved method”.
   1. For each of the three methods (**Index, About, Contact** in the **HomeController.cs** file, remove the **string SPHostUrl** input parameter and also remove the **ViewBag.hostWebUrl=SPHostUrl;** code. When finished your **HomeController** class should appear as follows:

public class HomeController : Controller {

[SharePointContextFilter]

public ActionResult Index() {

// create named properties in ViewBag

ViewBag.welcomeMessage = "Hello from the ASP.NET MVC Framework";

// return default view for this action

return View();

}

public ActionResult About() {

ViewBag.Message = "Your application description page.";

return View();

}

public ActionResult Contact() {

ViewBag.Message = "Your contact page.";

return View();

}

}

1. In the **CustomerManagerMVCWeb** project open the **spcontext.js** file located in the **Scripts** folder.
   1. Examine the contents of this file. This is used to automatically append the SPHost Url parameter to every link on a page.
   2. Close the **spcontext.js** file
2. This **spcontext** bundle is added by default to the **\_Layout.cshtml** partial view file (recall that this is akin to a master page in that all other partial views will be rendered inside of this, meaning that all partial views will have access to this same JavaScript).
   1. Examine this file for the @Scripts.Render code.

Note that the spcontext.js library is used to automatically append the SPHostUrl parameter to every link on a page. In MVC5, the library is packaged in its own bundle, and is added by default to the \_Layout.cshtml partial view. The only thing to be careful about is if you are using a custom Chrome control bundle (i.e. @Scripts.Render (“~bundles/spchrome) the spcontext library loads after the chrome control so that it can process any links you have defined in the Chrome Control options.

1. In Visual Studio double-click on the **SharePointContext.cs** file in the **CustomerManagerMVCWeb** project to open this file in Code View.
2. Examine the code in this utility class named **SharePointContext.cs**.
   1. Note the overloaded **GetSPHostUrl** methods for retrieving the SharePoint host URL from the QueryString of the specified HTTP request as shown below.

public static Uri GetSPHostUrl(HttpRequestBase httpRequest) {

if (httpRequest == null) {

throw new ArgumentNullException("httpRequest");

}

string spHostUrlString = TokenHelper.EnsureTrailingSlash(httpRequest.QueryString[SPHostUrlKey]);

Uri spHostUrl;

if (Uri.TryCreate(spHostUrlString, UriKind.Absolute, out spHostUrl) &&

(spHostUrl.Scheme == Uri.UriSchemeHttp || spHostUrl.Scheme == Uri.UriSchemeHttps)) {

return spHostUrl;

}

return null;

}

/// <summary>

/// Gets the SharePoint host url from QueryString of the specified HTTP request.

/// </summary>

/// <param name="httpRequest">The specified HTTP request.</param>

/// <returns>The SharePoint host url. Returns <c>null</c> if the HTTP request doesn't …</returns>

public static Uri GetSPHostUrl(HttpRequest httpRequest) {

return GetSPHostUrl(new HttpRequestWrapper(httpRequest));

}

/// <summary>

/// The SharePoint host url.

/// </summary>

public Uri SPHostUrl {

get { return this.spHostUrl; }

}

Note that the classes defined in this file (e.g. **SharePointContext** class) are being defined in the same namespace as the App (i.e. **CustomerManagerMVCWeb**) This makes it easy to access the members of these classes such as the **GetSPHostUrl** method in the **SharePointContext** class from a Razor view.

1. Now that we have seen how the Query String for the Host Web URL is generated for each link on a page and persisted throughout the application (through the use of the **spcontext.js** bundle), and we have seen how we might access this information in our application (through the use of the SharePointContext class, let’s put this knowledge to use in the \_Layout.cshtml file by updating how we are generating the link back to the Host Web in the navigation bar.
   1. Open the shared view file **\_Layouts.cshtml** in a Code View window.
   2. Inside the **<div class=”navbar-header”>**  element locate the **<a>** tag which provides the link to the host web.

<a href="@ViewBag.hostWebUrl" class="navbar-brand">Host Web</a>...

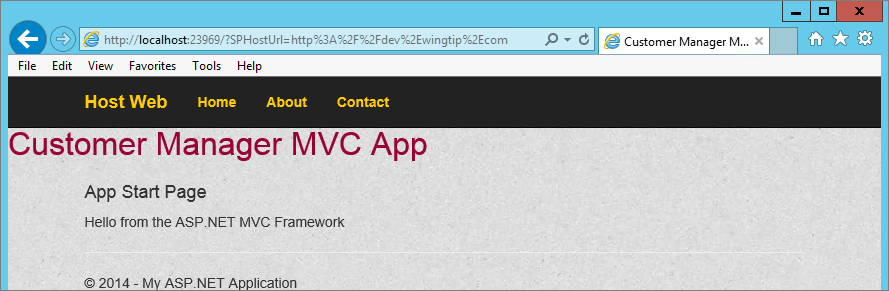
* 1. Update this link determine the host web URL using a call to **SharePointContext.GetHostUrl** as shown below:

<a href="@Html.Raw(SharePointContext.GetSPHostUrl(HttpContext.Current.Request).AbsoluteUri)"

class="navbar-brand">Host Web</a>

* 1. Save your changes and close **\_Layouts.cshtml**.

1. Test your work by running the app in the Visual Studio debugger.
   1. Press **F5** to begin debugging.
   2. Once the app has been installed, you should be redirected to the app's start page.
   3. Verify that the links for **Home**, **About** and **Contact** still work correctly.
   4. Also verify that after you move from page to page that the **Host Web** link still works correctly. If the host web link still works, it means that you are successfully tracking the host web URL across requests using an ASP.NET Session variable.



You have seen it's relatively simple to use the inbuilt functionality to track query string parameters that the SharePoint host passes to your app's start page.