

# Developing Provider-hosted Add-ins with MVC5

**Lab Time:** 60 minutes

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

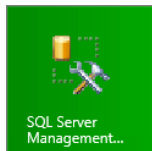
**Lab Overview:** In this lab you will create a provider-hosted app using MVC5. You will investigate the capabilities of the Model-View-Controller pattern, and create a simple RESTful service that wraps a database.

## Exercise 1: Create and Populate the Wingtip CRM Database in SQL Server

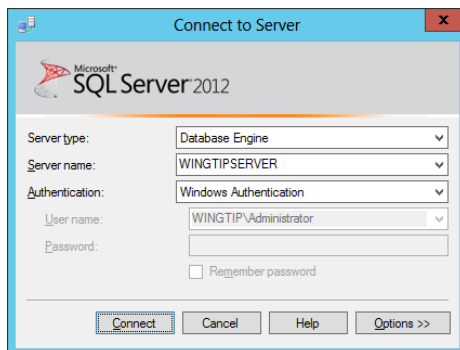
In this exercise you will open the solution for the lab and examine the contents.

(Note: you will only need to do this if you have not already run this script for a previous exercise in your course)

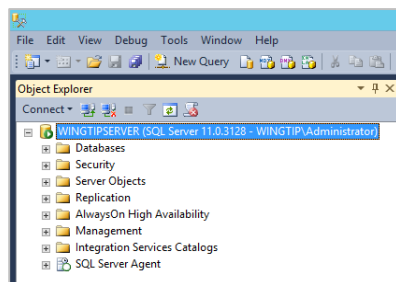
1. Launch SQL Server Management Studio and connect to the default SQL Server instance on **WingtipServer**.
  - a) Press the Windows key to display the Window Start page.
  - b) On the Windows Start page, locate and click the **SQL Server Management Studio** tile to launch this application.



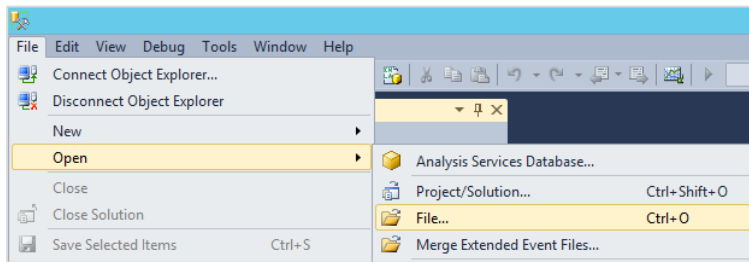
- c) When you are prompted with the **Connect to Server** dialog, select **WingtipServer** as the server name and click **Connect**.



- d) Once SQL Server Management Studio has started, you should be able to see a tree view which shows database objects in the Object Explorer.



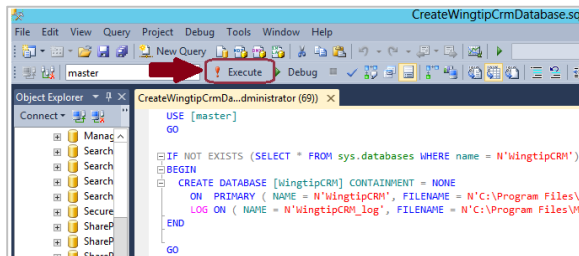
2. Execute the SQL script named. **CreateWingtipCrmDatabase.sql** to create the **WingtipCRM** database.
  - a) From the **File** menu in SQL Server Management Studio, select the menu command **File** ☐ **Open** ☐ **File...**



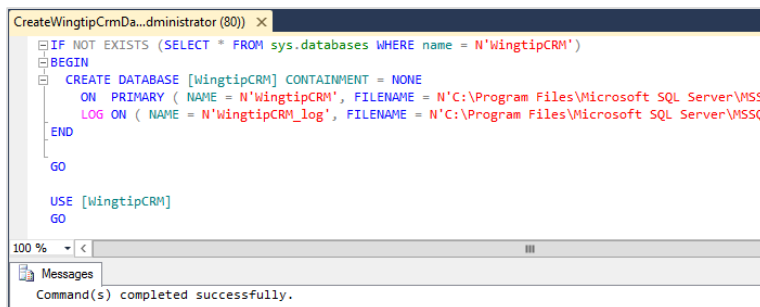
- b) When prompted by the **Open File** dialog, select the script named **CreateWingtipCrmDatabase.sql** at the following path and then click Open.

**C:\Student\Setup\CreatewingtipCrmDatabase.sql**

- c) Once the script named **CreateWingtipCrmDatabase.sql** is open, take a moment to examine what's inside. You should be able to see this script carries out the following tasks.
- Create a new SQL Server database named **WingtipCRM**.
  - Create a new table named **Customers** in the **WingtipCRM** database.
  - Add SQL Login for the Windows account **BUILTIN\IIS\_IUSRS** and configure permissions to the **WingtipCRM** database.
  - Add SQL Login for domain account **WINGTIP\Domain Users** and configure permissions to the **WingtipCRM** database.
  - Add sixteen samples records into the **Customers** table.
- d) Make sure that window displaying **CreateWingtipCrmDatabase.sql** is the active window. Then click the **Execute** button in the toolbar with the exclamation point to execute the script.

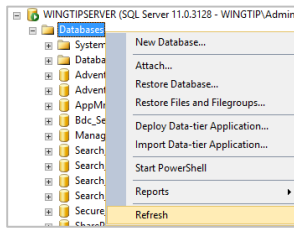


- e) The script should execute without any errors and display **Command(s) completed successfully** in the Messages window.

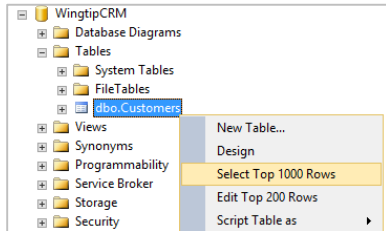


3. Verify that the database has been properly created.

- a) In the **Object Explorer** in SQL Server Management Studio, locate the **Database** node in the tree view control on the left-hand side of screen. Right-click the **Database** node and then click the **Refresh** menu command to display the database that has just been created.



- b) After the collection of databases has been refreshed, look down and locate the new database named **WingtipCRM**.
- c) Expand the **WingtipCRM** node and then the **Tables** node and verify you can see a table named **dbo.Customers**.
- d) Right-click on the **dbo.Customers** node and then click the menu command with the caption of **Select Top 1000 Rows**.



- e) Verify that you can see a set of customer records that have been added to the **Customers** table.

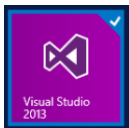
ID	FirstName	LastName	Company	WorkPhone	HomePhone	EmailAddress
1	Quincy	Nelson	Berthic Petroleum	1(340)608-7748	1(340)517-3737	Quincy.Nelson@BerthicPetroleum.com
2	Jude	Mason	Cyberdyne Systems	1(203)408-0466	1(203)411-0071	Jude.Mason@CyberdyneSystems.com
3	Sid	Stout	Rooxon	1(518)258-6571	1(518)376-8576	Sid.Stout@Rooxon.com
4	Gilberto	Gillespie	Shirra Electric Power Company	1(270)510-1720	1(270)795-7810	Gilberto.Gillespie@ShirraElectricPowerCompany.com
5	Diane	Strickland	Izon	1(407)413-4851	1(407)523-5411	Diane.Strickland@Izon.com
6	Jacqueline	Zimmerman	Zorg Industries	1(844)234-0550	1(844)764-3522	Jacqueline.Zimmerman@ZorgIndustries.com
7	Naomi	Schroeder	ComTron	1(204)355-6648	1(204)356-2831	Naomi.Schroeder@ComTron.com
8	Lynne	Stephens	Trade Federation	1(407)787-7308	1(407)732-1700	Lynne.Stephens@TradeFederation.com
9	Luther	Sullivan	Metacortex	1(323)755-3404	1(323)684-7814	Luther.Sullivan@Metacortex.com
10	Rose	Parsons	Hanso Foundation	1(802)357-5583	1(802)727-0246	Rose.Parsons@HansoFoundation.com
11	Bridgette	Meadows	Brown Streak Railroad	1(250)468-4824	1(250)403-3653	Bridgette.Meadows@BrownStreakRailroad.com
12	Merle	Black	Volée Airlines	1(248)240-1267	1(248)221-0302	Merle.Black@VoléeAirlines.com
13	Berta	Wilkinson	Doublemeat Palace	1(270)830-5347	1(270)338-3401	Berta.Wilkinson@DoublemeatPalace.com
14	Brandi	Bates	Duff Beer	1(808)660-1110	1(808)833-4310	Brandi.Bates@DuffBeer.com
15	Ana	Mathews	WarioWare, Inc.	1(844)663-5428	1(844)782-2117	Ana.Mathews@WarioWare, Inc. com
16	Chet	Lawson	The Crab Shack	1(340)843-4478	1(340)523-1010	Chet.Lawson@TheCrabShack.com

You have now created the **WingtipCRM** database that you will use as a sample database for the remainder of this lab as well as many of the labs that come afterward. At this point you can close the SQL Server Management Studio or you can keep it open if you want to monitor changes to the data over the course of the following exercises.

## Exercise 2: Creating a Provider-Hosted App that uses the MVC Framework

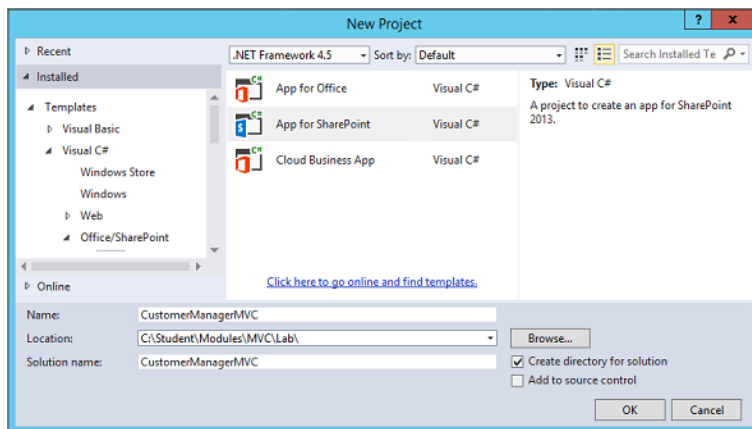
In this exercise you create a new provider-hosted app based on the ASP.NET MVC Framework.

4. Launch **Visual Studio 2013** as administrator:
  - a) **Windows** Keyboard Key → Right click on the **Visual Studio 2013** tile and select **Run as administrator**.

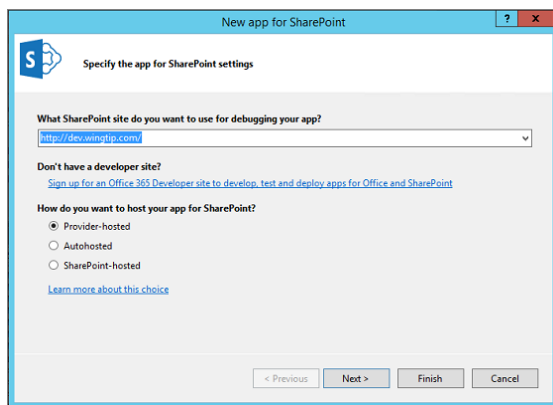


5. Create the new solution in Visual Studio 2013:
  - a) In Visual Studio select **File → New → Project**.
  - b) In the **New Project** dialog:
    - i) Select **Templates → Visual C# → Office/SharePoint → Apps**.
    - ii) Click **App for SharePoint**
    - iii) Name the new project **CustomerManagerMVC**.

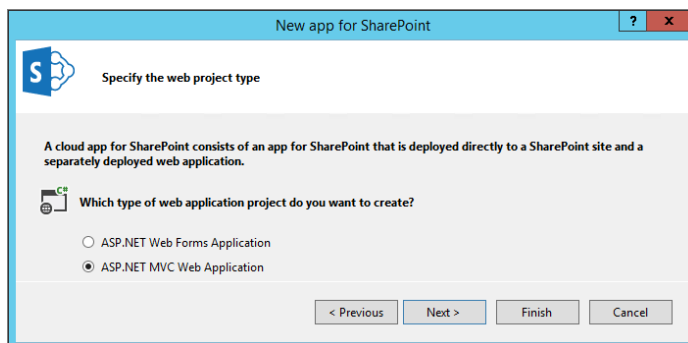
- iv) Add the new project into the folder at **C:\Student\Modules\MVC\Lab**.
- v) Click **OK** to create the project which will lead Visual Studio to display the **New App for SharePoint** wizard.



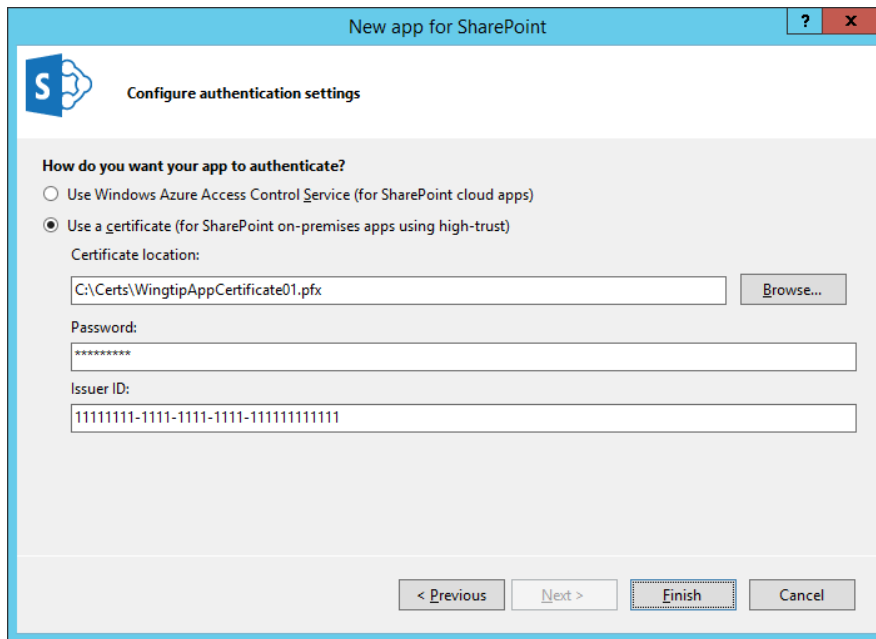
- c) In the **New App for SharePoint** wizard:
  - i) Select an appropriate site for to host the new app such as the developer site at <http://dev.wingtip.com>.
  - ii) Select **Provider-Hosted** as the hosting model.
  - iii) Click **Next**.



- 6. Specify the web project type as **ASP.NET MVC Web Application** and click **Next**.

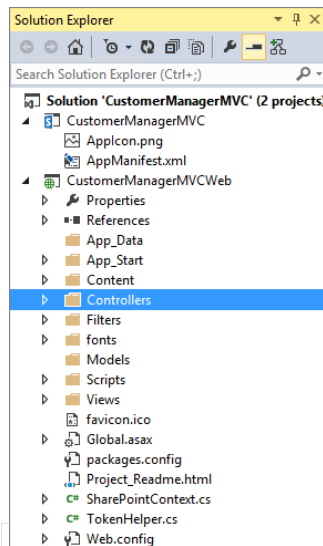


- a) You should now be at the Configure **authentication settings** page. Fill this page out as follows.
  - i) Select **Use a certificate (for SharePoint on-premise apps using high-trust)**
  - ii) **Certificate location:** C:\Certs\WingtipAppCertificate01.pfx
  - iii) **Password:** Password1
  - iv) **Issuer ID:** 11111111-1111-1111-1111-111111111111

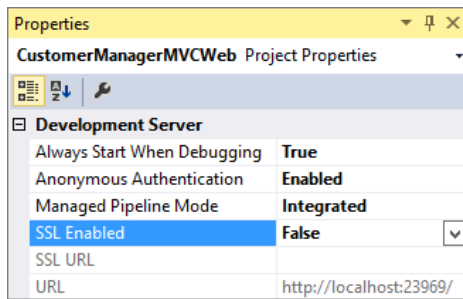


b) Click **Finish** to complete the wizard and create the new provider-hosted app project.

7. When the New app for SharePoint wizard completes its 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.



8. In the **CustomerManagerMVC** project open the **AppManifest.xml** file by double clicking it.
  - a) Set the **Title** to **Customer Manager MVC**
  - b) Save and close the **AppManifest.xml** file.
9. Configure the **CustomerManagerMVCWeb** project to disable the use of SSL.
  - a) Select the project **CustomerManagerMVCWeb** in the **Solution Explorer** tool window.
  - b) Look in the **Properties** tool window. Look at the **SSL Enabled** property and make sure it is set to **False**.



10. Examine the HomeController.cs file in your CustomerManagerMVCWeb project

- 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.
- 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**.
- 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;
            }
        }
    }
}
```

- 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();
}
```

- 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.

11. Examine the default view for the **Index** action method.

- Right-click the white area inside the **Index** method and select **go to View**.

```


public ActionResult Index() {
    User spUser = null;

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

    // 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();
}

```



- b) The view for the **Index** action named **Index.cshtml** should be open inside a Code View window.
- c) 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,

- d) 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>

```

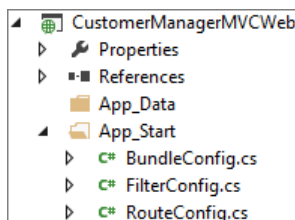
- e) **Save** your work.

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.

## 12. Examine the **HomeController** for the default controller to the MVC app's routing scheme.

- a) In the Solution Explorer, expand the **App\_Start** folder inside the **CustomerManagerMVCWeb** project and locate the C# source file named **RouteConfig.cs**.



- b) 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 }
};
```

- c) 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.

13. Configure the start page in the SharePoint app.

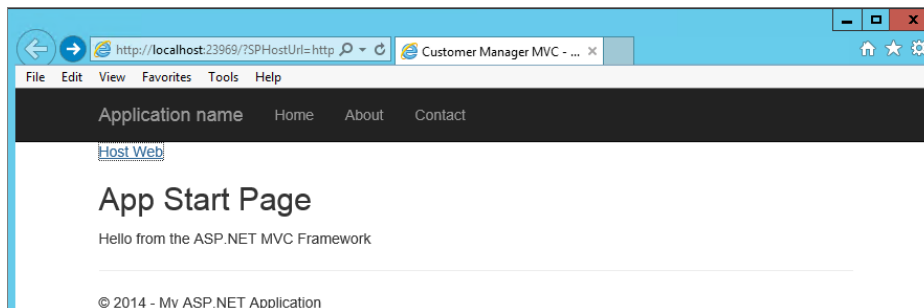
- In the Solution Explorer, expand the **CustomerManagerMVC** project node.
- Locate and double-click **appManifest.xml** to open the app manifest file in the app manifest designer.
- Check and if needed, change the **Start page** to **CustomerManagerMVCWeb/**.

The screenshot shows the App Manifest Designer interface. The 'Start page' field is set to 'CustomerManagerMVCWeb/' and the 'Query string' field is set to '{StandardTokens}'.

- d) Save and close **appManifest.xml**.

14. Test your work by running the app in the Visual Studio debugger.

- Press **F5** to begin debugging.
- If you are prompted with a SharePoint page that asks whether you to trust the app, click **Trust it**.
- 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.



- Click on the **Host Web** link and verify that you can click on it to navigate back to the host web.
- 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#.

15. Leverage the MVC Framework's ability to read query string parameters when handling requests from SharePoint.

- 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"];
```

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

```
public ActionResult Index(string SPHostUrl)
{
    // ...
}
```

- 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;
    // ...
}
```

16. 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();
}
```

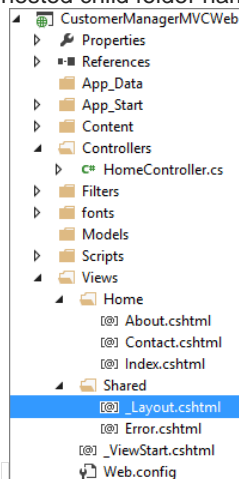
17. Test your work by running the app in the Visual Studio debugger.
- Press **F5** to begin debugging.
  - Once the app has been installed, you should be redirected to the app's start page.
  - Verify that hyperlink which links to the host web is still working properly.
  - When you are done, close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

### Exercise 3: 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**.

- Modify the shared view file for the **CustomerManagerMVCWeb** project.
  - 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**.



- 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>

```

- c) 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.

- d) 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>
    </div>
  </div>
  ...

```

- e) 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:

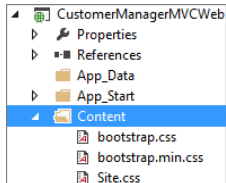
```
<a href="@ViewBag.HostWebUrl" class="navbar-brand">Host web</a>
```

- f) 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).
- g) Be sure to save your change to the **Index.cshtml** file and then return to the **\_Layout.cshtml** file for the next step.
- h) 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>
<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>
```

2. Improve the user interface by adding CSS styles and a background image to your shared page layout.

- a) In the **CustomerManagerMVCWeb** project, expand the **Content** folder. You should be able to see that this contains a CSS file named **Site.css**.

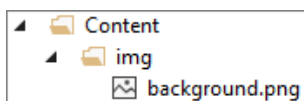


Note that Visual Studio and the MVC Framework have already configured the shared view named **\_Layout.cshtml** to link to **Site.css**. Any styles you add to **Site.css** will affect all pages that use the shared view.

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

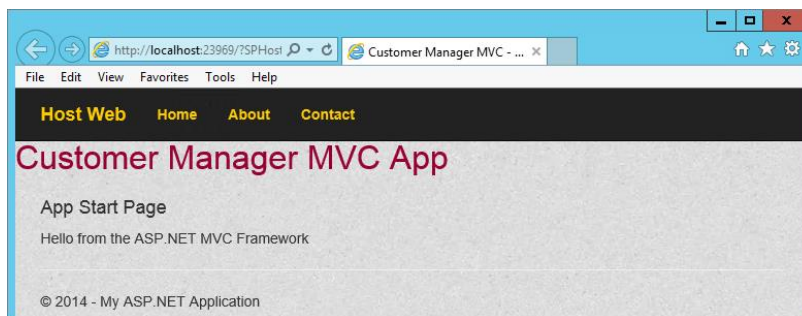
Return to Visual Studio and add the **background.png** image file into the **img** folder.

(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).



- d) 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.

- e) 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**.
- f) Save and close the Site.css file and Close the Site.css.additions.txt file.
- g) In the same **Content** folder in the project, open the bootstrap.cs file and Find (Ctrl+f) the following text "max-width: 1170px"
  - i) Modify this to read "max-width: **1250px**"
  - ii) 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)
3. Test your work by running the app in the Visual Studio debugger.
  - a) Using the **Visual Studio Build Menu** select **Rebuild Solution** to ensure that your changes are picked up by the compiler.
  - b) Press **F5** to begin debugging.
  - c) If you are prompted with a SharePoint page that asks whether you to trust the app, click **Trust it**.
  - d) 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.



4. Examine the **About** and **Contact** pages in the MVC app.
  - a) 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)
  - b) 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();
}
```

- c) Right-click the white area inside the **About** method and select **Go To View**.
- d) 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>
```

- e) Save and close the About.cshtml file.
5. Return to the **HomeController.cs** file and right-click the white area inside the **Contact** method and select **Go To View**.
  - a) Examine and modify the contents of **Contact.cshtml** to match the following code listing.

```
@{
    ViewBag.Title = "Contact the App Vendor";
```

```

}

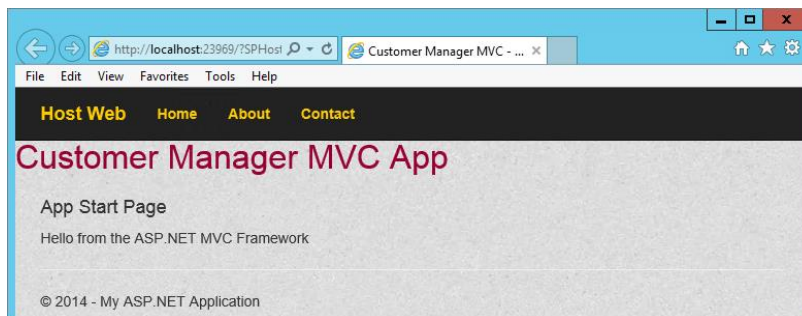
<h2>@ViewBag.Title.</h2>
<h3>@ViewBag.Message</h3>

<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>

```

- b) Save and close the Contact.cshtml file.
6. Test your work by running the app in the Visual Studio debugger.
  - a) Press **F5** to begin debugging.
  - b) 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 4: 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 **SPHostUrl** 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".
  - a) 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();
    }
}

```

2. In the **CustomerManagerMVCWeb** project open the **spcontext.js** file located in the **Scripts** folder.
  - a) Examine the contents of this file. This is used to automatically append the SPHost Url parameter to every link on a page.
  - b) Close the **spcontext.js** file
3. 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).
  - a) 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.

4. In Visual Studio double-click on the **SharePointContext.cs** file in the **CustomerManagerMVCWeb** project to open this file in Code View.
5. Examine the code in this utility class named **SharePointContext.cs**.
  - a) 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 <null> 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.

6. 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.

- a) Open the shared view file `_Layouts.cshtml` in a Code View window.
- b) 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>...
```

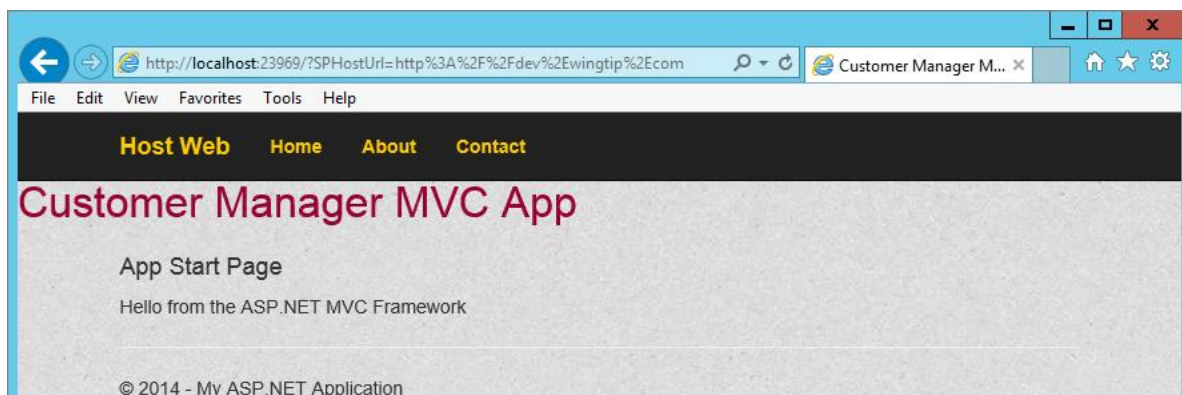
- c) Update this link determine the host web URL using a call to `SharePointContext.GetHostUrl` as shown below:

```
<a href="@Html.Raw(SharePointContext.GetSPHostUrl(HttpContext.Current.Request).AbsoluteUri)"
class="navbar-brand">Host Web</a>
```

- d) Save your changes and close `_Layouts.cshtml`.

7. Test your work by running the app in the Visual Studio debugger.

- a) Press **F5** to begin debugging.
- b) Once the app has been installed, you should be redirected to the app's start page.
- c) Verify that the links for **Home**, **About** and **Contact** still work correctly.
- d) 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.



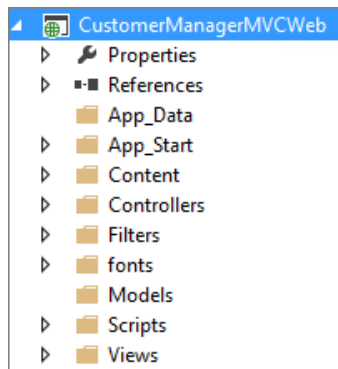
You have seen it's relatively simple to use the inbuilt functionality to track query string parameters that the SharePoint host passes to your app's start page.

## Exercise 5: Accessing a SQL Server Database using a Strongly-typed Controller Class

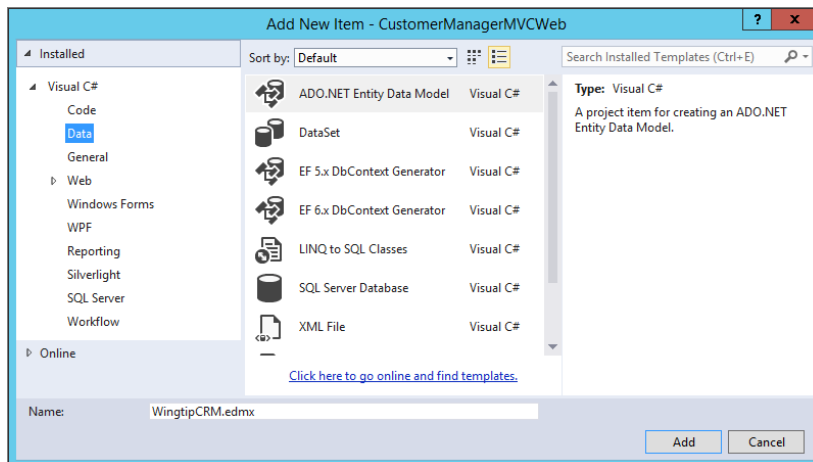
In this exercise, you will continue to work on the Customer Manager MVC app to add CRUD functionality for reading and writing customer records in the SQL Server database named **WingtipCRM**. This is the same SQL Server database you have worked with in earlier labs.

**Note:** This lab assumes you have already completed the previous lab on REST where you created the SQL Server database named **WingtipCRM**. If you have not already completed the REST lab, you will not have the database required to continue with this lab. If this is the case, you should stop your work in this lab and go back to the REST lab and complete **Exercise 1: Create and Populate the Wingtip CRM Database in SQL Server**. Once you have completed that exercise and created the **WingtipCRM** database, you can continue with this lab.

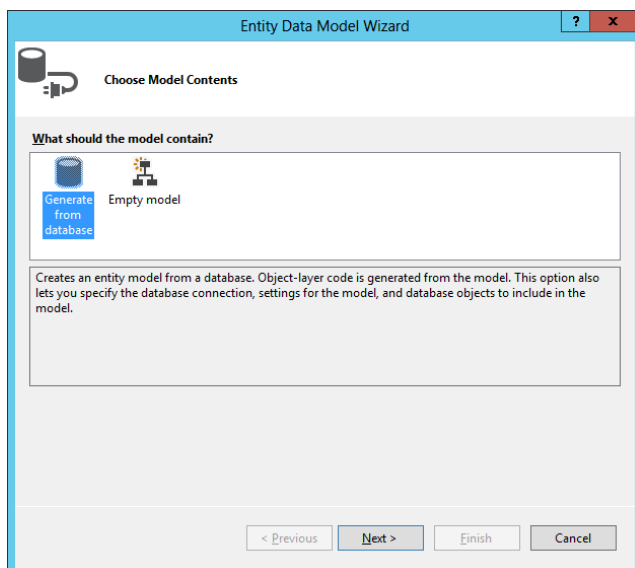
1. Add an ADO.NET Entity Model for the **WingtipCRM** Database.
  - a) Using the Solution Explorer, examine the top level folders of the **CustomerManagerMVCWeb** project. You should be able to see that there is already a folder named **Models**.



- b) Right-click on the **Models** folder and click **Add → New Item**.
- c) In the left side of the **Add New Item** dialog, select **Installed → Visual C# → Data**.
- d) Select the project item template named **ADO.NET Entity Data Model**.
- e) Enter a **Name** of **WingtipCRM.edmx**.
- f) Click the **Add** button on the bottom right to begin the process of creating the new project item.

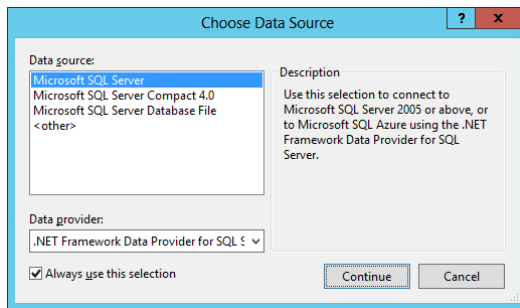


- g) When you are prompted by **Choose Model Contents** page of the **Entity Data Model Wizard** dialog, select the option **Generate from database** and click **Next**.

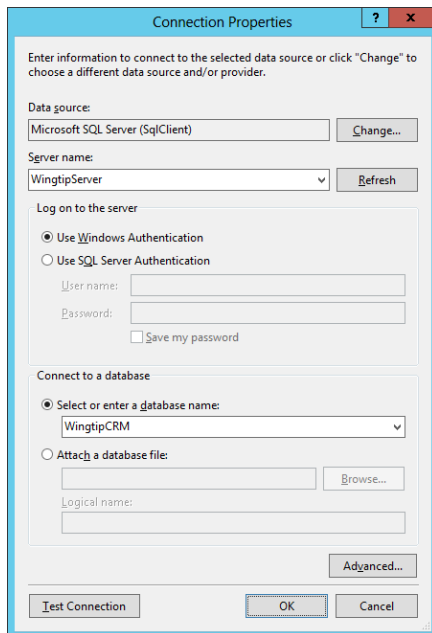




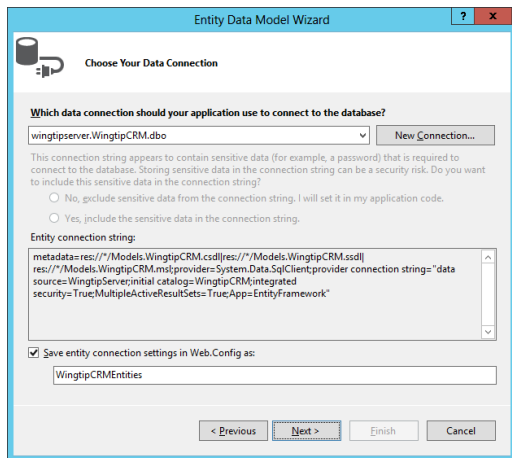
- h) On the **Choose Your Data Connection** page of the **Entity Data Model Wizard**, you are prompted to select or create a new connection. Click the **New Connection** button to display the **Connection Properties** dialog.
- i) If you are prompted by the **Choose Data Source** dialog, select **Microsoft SQL Server** and then click **Continue**.



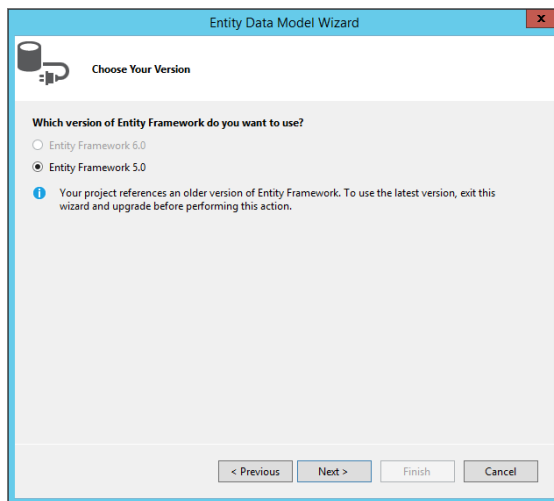
- j) Next, you should see the **Connection Properties** dialog. Follow these steps to fill in the required information.
- Enter a **Server name** of **WingtipServer**.
  - Enter a database name of **WingtipCRM**.
  - Once the **Connection Properties** dialog looks like the following screenshot, click the **OK** button to save the connection information and return to the **Choose Your Data Connection** page of the **Entity Data Model Wizard**.



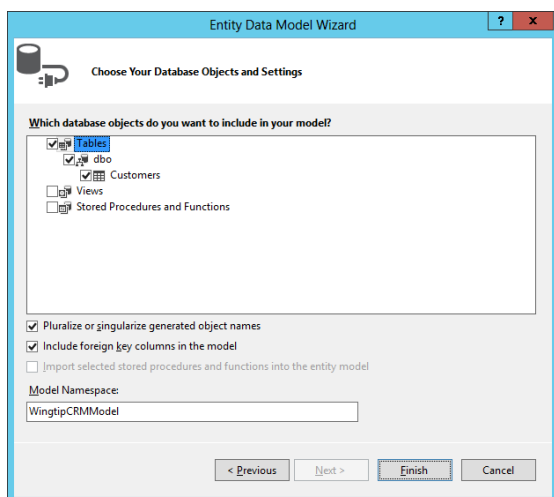
- k) On the **Choose Your Data Connection** page of the **Entity Data Model Wizard**, leave the bottom of the page with the default settings as shown in the following screenshot so that the connection information is stored in **web.config** file under the name of **WingtipCRMEntities**. Click **Next** to continue to the **Choose Your Database Objects and Settings** page.



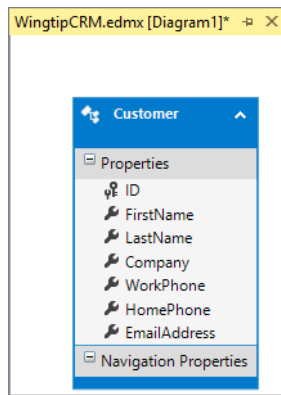
- l) On the **Choose Your Version** page select **Entity Framework 5.0** and click **Next**



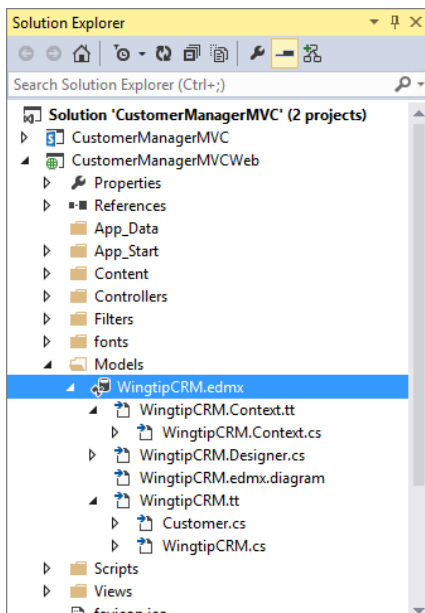
- m) On the **Choose Your Database Objects and Settings** page, expand the **Tables** node and select the **Customers** table. Leave the other settings on the page with their default settings as shown in the following screenshot. Click the **Finish** button to complete the task of create the entity data model.



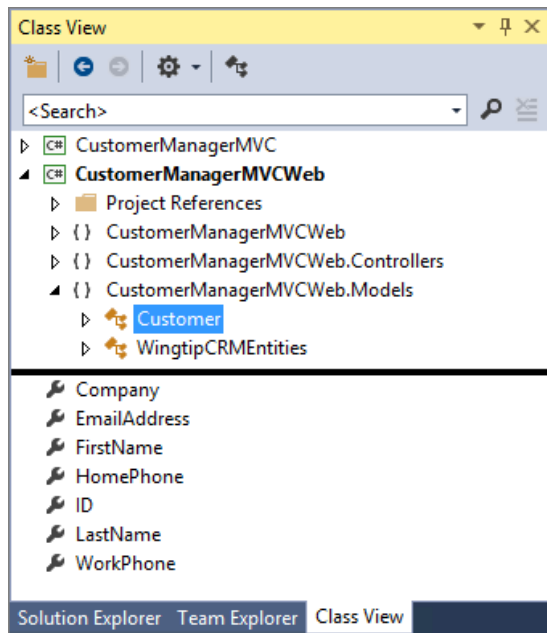
- n) If Visual Studio prompts you whether it is OK to update the **web.config** file, select **Yes**.
- o) Once the Entity Data Model Wizard has finished its work, it will then display a Visual Studio designer that provides a visual representation of the new entity data model which contains a single entity named **Customer**.



- p) After you have examined the **Customer** entity and its properties in the visual designer, close **WingtipCRM.edmx**.
2. Use the Solution Explorer to examine the files that were added to the **CustomerManagerMVCWeb** project.
- a) You should be able to see several files have been added the project to support the new entity data model.



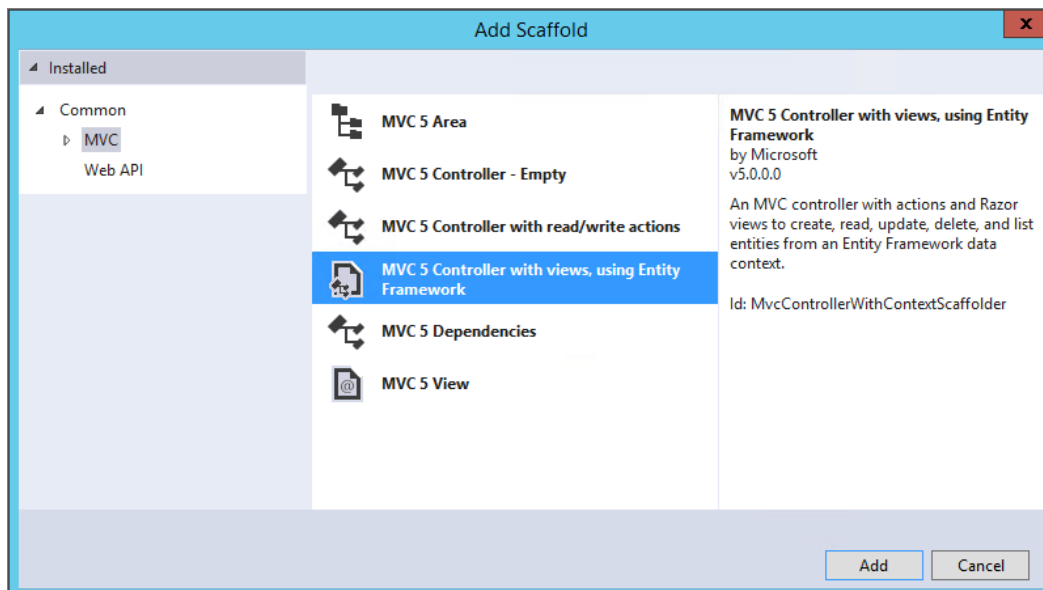
3. Use Class View in the Solution Explorer to see what C# classes have been created.
- a) Switch from the Solution Explorer over to Class View.
- b) Expand the node with the **CustomerManagerMVCWeb.Models** namespace.
- c) Verify that there is an entity model class named **Customer** which is associated with the **Customers** table in SQL Server.
- d) Verify that there is second class named **WingtipCRMEntities**.



4. Run the app project in the Visual Studio debugger once to build out all the code generated by the Entity Framework. This step is required so that later steps in this lab work correctly.
  - a) Press the **F5** key to begin a debugging session.
  - b) The Customer Manager app should install and then you should be directed to the app's start page.
  - c) Close the browser.
  - d) Return to Visual Studio and ensure that the debugging session has ended.

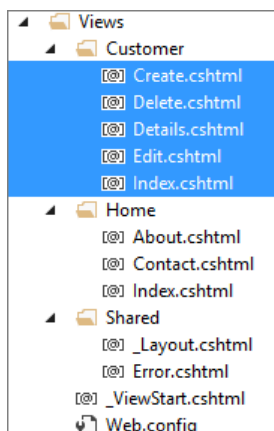
Next, you will use Visual Studio and the MVC framework to generate a strongly-typed controller class that will make it relatively easy to add basic CRUD functionality to the Customer Manager MVC app.

5. Create a strongly-typed controller class.
  - a) In the **Solution Explorer**, expand the **CustomerManagerMVCWeb** project node.
  - b) Right-click the **Controllers** folder and select **Add → New Scaffolded Item**.
    - i) Select **Installed → MVC → Controller → MVC 5 Controller with views, using Entity Framework**
    - ii) Click **Add**



- iii) **Controller name:** CustomerController
- iv) **Model class:** Customer (CustomerManagerMVCWeb.Models)
- v) **Data context class:** WingtipCRMEntities (CustomerManagerMVCWeb.Models)
- vi) **Views:** All 3 items checked (Generate views, Reference script libraries, Use a layout page)
- vii) Click the **Add** button.

- c) When you add the controller named **CustomerController**, Visual Studio creates a file named **CustomerController.cs** and opens this C# source file a Code View window.
  - d) Examine the **CustomerController** class that Visual Studio has generated for you. What you will find in this controller class is very different from the **HomeController** class you generated earlier as an **Empty MVC controller**. You will see that there are many action methods that have been added and implemented by Visual Studio.  
Note: You do not need to modify anything in the **CustomerController** class. It will work with just the code that has been generated by Visual Studio. However, as mentioned above, you should take a little time reviewing the code in each of the action methods in the **CustomerController** class to get an idea of how the code is written.
  - e) When you are done, close **CustomerController.cs** without saving any changes.
6. Modify a view that has been generated for the **CustomerController**.
- a) In Solution Explorer, expand the project node for the **CustomerManagerMVCWeb** project.
  - b) Expand the top-level **Views** folder and then expand the child folder named **Customer**.
  - c) Within the Customer folder, you should be able to see that five views that have been created for the **CustomerController**.



- d) Double-click on the view file named **Index.cshtml** inside the **Customer** folder to open it in a Code View window.
- e) As you can see, there is quite a bit of Razor code on the page.
- f) Make the following three changes to **Index.cshtml**.
  - i) Modify the value assigned to **ViewBag.Title** to **Customer List**.
  - ii) Change the text in the **<h2>** element from **Index** to **Customer List**.
  - iii) Change the first parameter passed to the **@Html.Action** method from **Create New** to **Create New Customer**.

```
@model IEnumerable<CustomerManagerMVCWeb.Models.Customer>

@{
    ViewBag.Title = "Customer List";
}

<h2>Customer List</h2>

<p>
    @Html.ActionLink("Create New Customer", "Create")
</p>
```

- g) Save your changes and close **Index.cshtml**.

At this point, you could open up the other view files for the **CustomerController** such as **Edit.cshtml** and **Details.cshtml** and make similar changes such as changing **Edit** to **Edit Customer** and changing **Details** to **Customer Details**.

7. Modify the navigation links in the shared view to incorporate the **CustomerController** in addition to the **HomeController**.
  - a) Locate the **\_Layout.cshtml** file in the **Shared** folder inside the **Views** folder.
  - b) Double-click on **\_Layout.cshtml** to open the view file in a Code View window.
  - c) Locate the **nav navbar-nav** element with the id of **topnav**. The contents inside should match the following code listing.

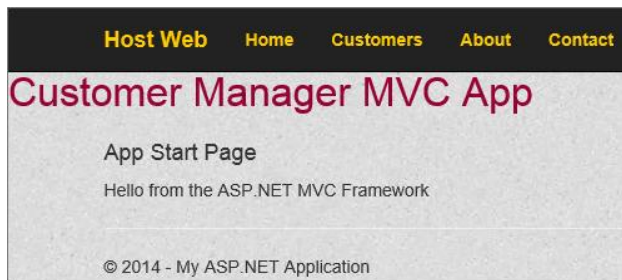
```
<ul class="nav navbar-nav">
    <li>@Html.ActionLink("Home", "Index", "Home")</li>
    <li>@Html.ActionLink("About", "About", "Home")</li>
    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
```

At this point, you are introducing a second controller. Therefore, you must modify the calls to **@Url.Action** to pass a second string parameter with the controller name

- d) Insert a new line in between the **<li>** tag elements for **Home** and **About** and add a new **<li>** tag which links to the **Index** action of the controller named **Customer** as shown in the following code listing.

```
<ul class="nav navbar-nav">
    <li>@Html.ActionLink("Home", "Index", "Home")</li>
    <li>@Html.ActionLink("Customers", "Index", "Customer")</li>
    <li>@Html.ActionLink("About", "About", "Home")</li>
    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
```

- e) When you are done, save your changes and close **\_Layouts.cshtml**.
8. Test your work by running the app in the Visual Studio debugger.
    - a) Press **F5** to begin debugging.
    - b) Once the app has been installed, you should be redirected to the app's start page.
    - c) Verify that your app is showing the same links as before along with the new link for **Customers**.



- d) Verify that the links for **Home**, **About** and **Contact** still work correctly.
- e) Click on the link for **Customers**. You should see a list of customers that matches the screenshot below.

FirstName	LastName	Company	WorkPhone	HomePhone	EmailAddress	
Quincy	Adams	Benthic Petroleum	1(340)608-7748	1(340)517-3737	Quincy.Nelson@BenthicPetroleum.com	Edit   Details   Delete
Sid	Stout	Roxxon	1(518)258-6571	1(518)376-8576	Sid.Stout@Roxxon.com	Edit   Details   Delete
Gilberto	Gillespie	Shinra Electric Power Company	1(270)510-1720	1(270)755-7810	Gilberto.Gillespie@ShinraElectricPowerCompany.com	Edit   Details   Delete
Diane	Strickland	Izon	1(407)413-4851	1(407)523-5411	Diane.Strickland@Izon.com	Edit   Details   Delete
Jacqueline	Zimmerman	Zorg Industries	1(844)234-0550	1(844)764-3522	Jacqueline.Zimmerman@ZorgIndustries.com	Edit   Details   Delete
Naomi	Schroeder	ComTron	1(204)355-6648	1(204)356-2831	Naomi.Schroeder@ComTron.com	Edit   Details   Delete
Lynne	Stephens	Trade Federation	1(407)787-7308	1(407)732-1700	Lynne.Stephens@TradeFederation.com	Edit   Details   Delete
Luther	Sullivan	Metacortex	1(323)755-3404	1(323)684-7814	Luther.Sullivan@Metacortex.com	Edit   Details   Delete
Rose	Parsons	Hanso Foundation	1(802)357-5583	1(802)727-0246	Rose.Parsons@HansoFoundation.com	Edit   Details   Delete
Bridgette	Meadows	Brown Streak Railroad	1(250)468-4824	1(250)403-3653	Bridgette.Meadows@BrownStreakRailroad.com	Edit   Details   Delete
Merle	Ferguson	Volée Airlines	1(248)240-1267	1(248)221-0302	Merle.Ferguson@VoléeAirlines.com	Edit   Details   Delete

- f) Try the following operations to better understand the user interface that has been created for you.
  - i) Create a new customer.
  - ii) View the details of an existing customer.
  - iii) Edit an existing customer.
  - iv) Delete an existing customer.
- g) When you have finished your testing, close the browser to stop the current debugging session and return to Visual Studio.

You have now completed the MVC lab successfully.