Developing Web Apps using ASP.NET MVC

Lab Time: 60 minutes

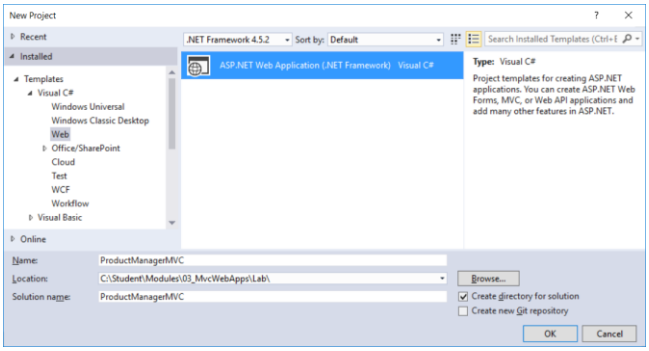
Lab Folder: C:\Student\Modules\03_MvcWebApps\Lab

Lab Overview: In this module you will create a Web application using ASP.NET MVC. This will give you a chance to work with MVC controllers and views. You will model application data using C# classes to create a design which allows you to pass a strongly-typed view model from a controller to a view. You will also learn how to configure a controller class to be an asynchronous controller to provide higher levels of scalability for high-volume applications.

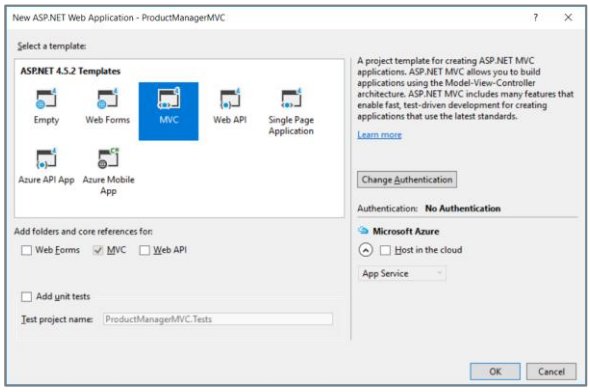
Exercise 1: Create a new MVC Application using Visual Studio 2017

In this exercise you will create a new Visual Studio project using the ASP.NET MVC framework.

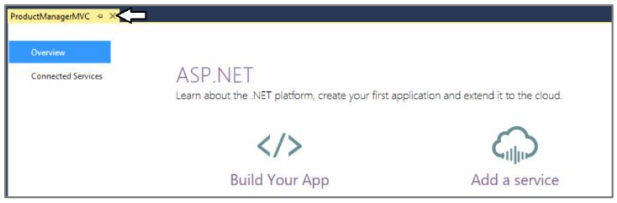
1. Launch **Visual Studio 2017**.
2. Create a new ASP.NET MVC project in Visual Studio 2017.
   a) In Visual Studio select **File > New > Project**.
   b) In the **New Project** dialog:
      i) Select **Installed > Templates > Visual C# > Web**.
      ii) In the platform dropdown menu, make sure the platform is set to **.NET Framework 4.5.2**.
      iii) Select the **ASP.NET Web Application** project template.
      iv) Name the new project **ProductManagerMVC**.
      v) Add the new project into the folder at **C:\Student\Modules\03_MvcWebApps\Lab**.
      vi) Click **OK** to display the **New ASP.Net Web Application** wizard.



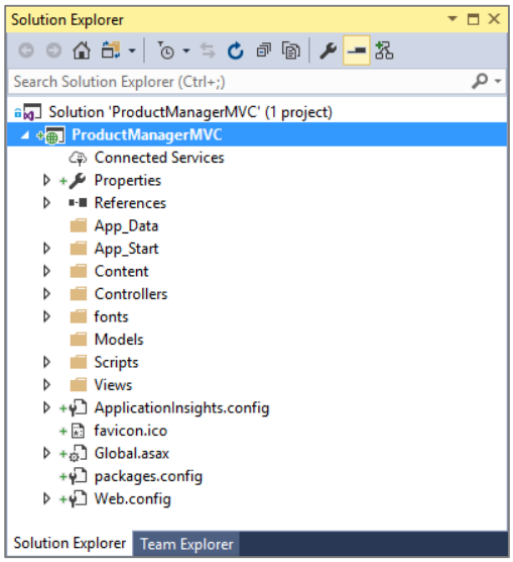
   c) In the **New ASP.Net Web Application** dialog, select the **MVC** template.
   d) Make sure **Authentication** is set to **No Authentication**.
   e) Make sure the **Host in the cloud** checkbox is unchecked.
   f) Click the **OK** button to create the new project.



g) When Visual Studio finishes creating the project, it displays an information page. Close this page by clicking the **x** in the tab.



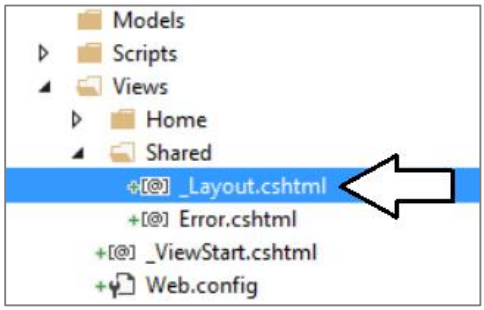
h) Take a minute to familiarize yourself with the structure of the project in the **Solution Explorer**.



3. Modify the shared layouts page named _**Layouts.cshtml**.

a) In Solution Explorer, expand the **Views** folder and then expand the **Shared** folder.

b) Double-click on _**Layouts.cshtml** to open it in an editor window.



c) Delete the entire contents of _**Layouts.cshtml** and replace with the following HTML starter page.

```
<!DOCTYPE html>
<html>

<head>
</head>

<body>
</body>

</html>
```

d) Copy and paste the following HTML code to provide the **head** section

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Product Manager MVC</title>
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" type="text/css" />
    <link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
</head>
```

e) Copy and paste the following HTML code to provide the **body** section of the page.

```
<body>

  <!-- add top nav here -->

  <!-- add main body content here -->

  @Scripts.Render("~/bundles/jquery")
  @Scripts.Render("~/bundles/bootstrap")
  @RenderSection("scripts", required: false)

</body>
</html>
```

f) Copy and paste the following code into the body just below the **add top nav here** comment.

```
<!-- add top nav here -->
<div class="container">
  <div class="row">
    <div class="navbar navbar-default " role="navigation">
      <div class="container-fluid">
        <div class="navbar-header">
          @Html.ActionLink("Product Manager", "Index", "Home", null, new { @class = "navbar-brand" })
        </div>
        <div class="navbar-collapse collapse">
          <ul class="nav navbar-nav">
            <!-- <li> elements with nav links go here -->
          </ul>
        </div>
      </div>
    </div>
  </div>
</div>
```

g) Copy and paste the following code into the body just below the **add main body content here** comment

```
<!-- add main body content here -->
<div class="container">
    <div class="container-fluid">
        @RenderBody()
    </div>
</div>
```

h) Save your changes and close **_Layouts.cshtml**.

4. Modify the **HomeController** class.

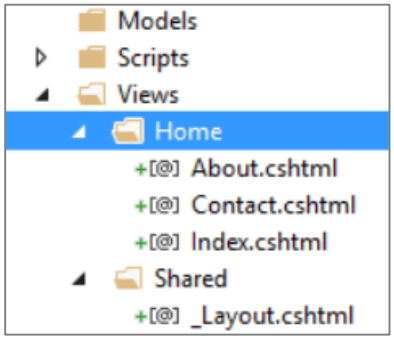
a) In Solution Explorer, expand the **Controllers** folder and then double-click on **HomeController.cs** open it in an editor window.

b) Delete all the existing code inside **HomeController.cs** and replace it with the following code

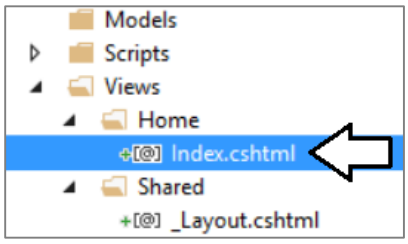
```
using System.Web.Mvc;

namespace ProductManagerMVC.Controllers {
  public class HomeController : Controller {
    public ActionResult Index() {
      return View();
    }
  }
}
```

c) Save and close on **HomeController.cs**.

5. Modify the **Index** view template for the **HomeController** class.
   a) In Solution Explorer, expand the **Views** folder and then expand the **Home** folder.
   b) You should see three views named **About.cshtml**, **Contact.cshtml** and **Index.cshtml**.



   c) Delete the two views named **About.cshtml** and **Contact.cshtml**.
   d) Double-click on the view named **Index.cshtml** to open it in an editor window.



   e) Delete all the existing content inside **Index.cshtml** and replace it with the following HTML code.

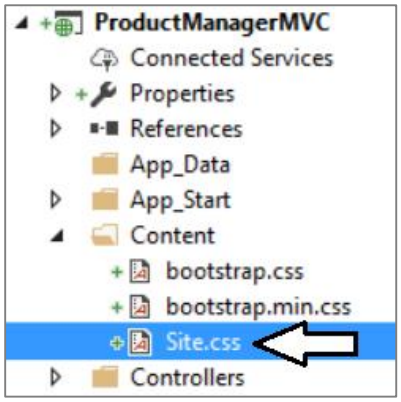
```
<div class="row">
    <div class="jumbotron">
        <h2>Welcome to Product Manager</h2>
    </div>
</div>

<div class="row">
    <div class="col-md-6">
        <div class="panel panel-primary">
            <div class="panel-heading">Add a New Product</div>
            <div class="panel-body"> Click the Add Product link above to add a new product.</div>
        </div>
    </div>
    <div class="col-md-6">
        <div class="panel panel-primary">
            <div class="panel-heading">See the Product Showcase </div>
            <div class="panel-body"> Click the Product Showcase link to see all Wingtip products.</div>
        </div>
    </div>
</div>
```

6. Modify the **Sites.css** file with a set of custom CSS styles.
   a) Using Windows Explorer, locate the snippet file named **Site.css.txt** in the **Students** at the following location.

```
C:\Student\Modules\03_MvcWebApps\Lab\Snippets\Site.css.txt
```

   b) Double click on **Site.css.txt** to open it in Notepad.
   c) Select all the CSS code inside **Site.css.txt** and copy it to the Windows clipboard.
   d) Return to Visual Studio.

e) In Solution Explorer, expand the **Content** folder and then double-click on **Sites.css** open it in an editor window.



f) Delete all the existing content from **Sites.css**.

g) Paste the content of the Windows clipboard into **Sites.css**.

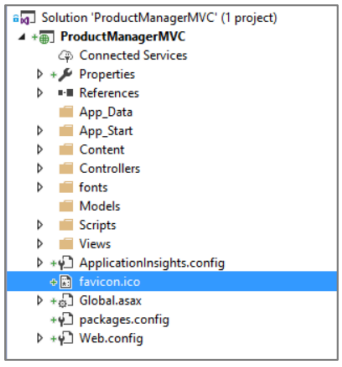
h) Save your changes and close **Sites.css**.

7. Add a **favicon.ico** file to the root folder of the **ProductManagerMVC** project.

a) Using Windows Explorer, locate the file named **favicon.icon** in the **Students** folder at the following location.

```
C:\Student\Modules\03_MvcWebApps\Lab\StarterFiles\favicon.ico
```

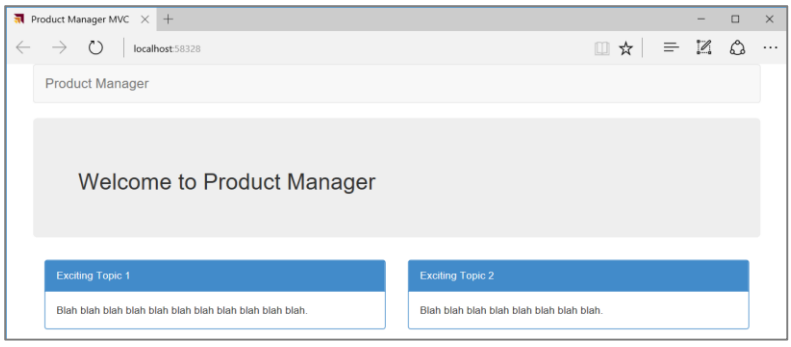
b) Copy the file named **favicon.ico** to the root folder of your project.



8. Test out the **ProductManagerMVC** Project using the Visual Studio Debugger

a) Press the **{F5}** key to start up the project in the Visual Studio debugger.

b) When the project starts, the home page should load in the browser and match the following screenshot.

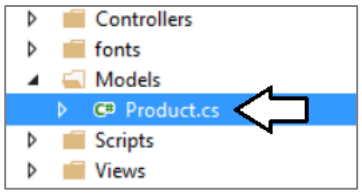


c) Close the browser, return to Visual Studio and stop the debugger.

Exercise 2: Creating a Strongly Typed Controller Class

In this exercise you will add a new C# class named **Product** to design a view model to pass strongly-typed product data you're your controllers into your view templates You will also create an interface named **IProductDataService** which will play a key role in your design to decouple the code in your controllers from any data access code you add to the project.

1. Create the **Product** class.
   a) In **Solution Explorer**, right-click on the **Models** folder and click the **Add > Class…** menu command.
   b) In the **Add New Item** dialog, give the new class file a name of **Product.cs**.



   c) Delete all the existing code inside **Product.cs** and replace it with the following code.

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace ProductManagerMVC.Models {

  public class Product {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Category { get; set; }
    public double ListPrice { get; set; }
    public string Description { get; set; }
    public string ProductImageUrl { get; set; }
  }

}
```

You have just created the **Product** class which contains the required set of properties. Now, you will add attributes to specific properties in the **Product** class so that they behave in a specialized fashion when used in an MVC view model.

   d) Add the **Key** attribute the **Id** property to indicate this property will hold the unique key identifier for each product.

```
[Key]
public int Id { get; set; }
```

   e) Add the **Required** attribute to the **Name** property with the error message **"Please enter product name."**;

```
[Required(ErrorMessage = "Please enter product name.")]
public string Name { get; set; }
```

   f) Add the **Required** attribute to the **Category** property with the error message **"Please enter product category."**;

```
[Required(ErrorMessage = "Please enter product category.")]
public string Category { get; set; }
```

   g) Add the **DisplayName** attribute to the **ListPrice** property with a value of **"List Price"**.
   h) Add the **DisplayFormat** attribute to the **ListPrice** property with a **DataFormatString** parameter value of **"{0:C2}"**
   i) Add the **Range** attribute with a configured range value between $1 and $10,000.
   j) Configure the **Range** attribute with an **ErrorMessage** of **"Please enter product price between $1 and $10,000."**

```
[DisplayName("List Price"), DisplayFormat(DataFormatString = "{0:C2}")]
[Range(1, 10000, ErrorMessage = "Please enter product price between $1 and $10,000.")]
public double ListPrice { get; set; }
```

   k) Add the **DataType** attribute to the **Description** property with a value of **DataType.MultilineText**.

l) Add the **Required** attribute to the **Description** property with the error message **"Please enter product description."**;

```
[DataType(DataType.MultilineText)]
[Required(ErrorMessage = "Please enter product description.")]
public string Description { get; set; }
```

m) Add the **DisplayName** attribute to the **ProductImageUrl** property with a value of **"Product Image"**.

```
[DisplayName("Product Image")]
public string ProductImageUrl { get; set; }
```

n) When you are done apply attributes, your **Product** class definition should match the following code listing.

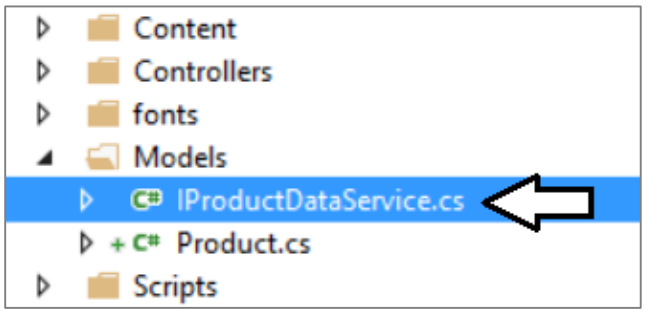
```
public class Product {
  [Key]
  public int Id { get; set; }
  [Required(ErrorMessage = "Please enter product name.")]
  public string Name { get; set; }
  [Required(ErrorMessage = "Please enter product category.")]
  public string Category { get; set; }
  [DisplayName("List Price"), DisplayFormat(DataFormatString = "{0:C2}")]
  [Range(1, 10000, ErrorMessage = "Please enter product price between $1 and $10,000.")]
  public double ListPrice { get; set; }
  [DataType(DataType.MultilineText)]
  [Required(ErrorMessage = "Please enter product description.")]
  public string Description { get; set; }
  [DisplayName("Product Image")]
  public string ProductImageUrl { get; set; }
}
```

o) Save your changes and close **Product.cs**.

2. Create the **IProductDataService** interface.

a) In **Solution Explorer**, right-click on the **Models** folder and click the **Add > Class…** menu command.

b) In the **Add New Item** dialog, give the new class file a name of **IProductDataService.cs**.



c) Delete the contents of **IProductDataService.cs** and replace it with the following code.

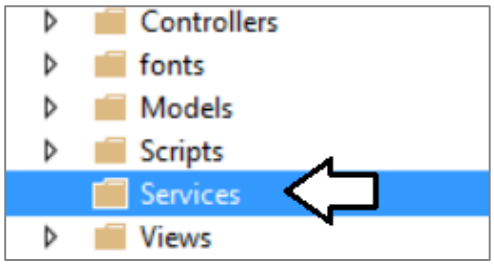
```
using System.Linq;

namespace ProductManagerMVC.Models {

  public interface IProductDataService {
    IQueryable<Product> GetAllProducts();
    Product GetProduct(int id);
    void AddProduct(Product product);
    void DeleteProduct(int id);
    void UpdateProduct(Product product);
  }

}
```

d) Save your changes and close of **IProductDataService.cs**.

Over the next few steps you will create and implement a class named **InMemoryProductDataService**. This class will provide the code to store and manage a list of products using ASP.NET session state. This type of design which relies on ASP.NET session state isn't practical for most real-world applications because the data is never written into persistent storage and there is no type of currency control to deal with multiple operations going on at the same time. However, implementing this class and seeing how it fits into an MVC application as a whole will be a valuable learning experience. Also keep in mind that in later labs you will develop a similar MVC application that reads and writes product data to and from an Azure SQL Server database.

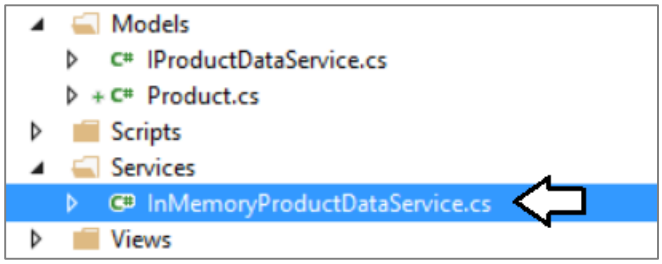
3. Create and implement the **InMemoryProductDataService** class

   a) Using Solution Explorer, create a new top-level folder named **Services**.



   b) Right-click on the **Services** folder and select **Add > Class**.

   c) Name the new class file **InMemoryProductDataService.cs**.

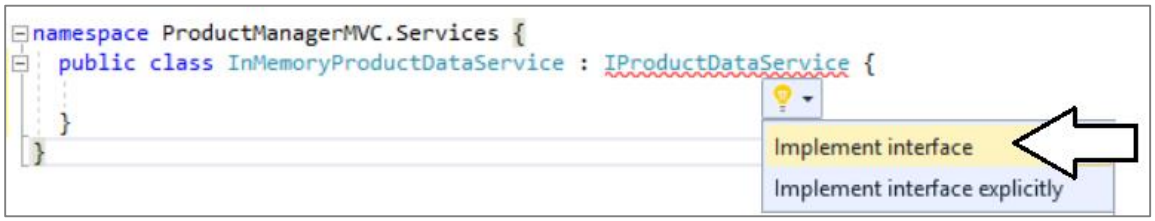


   d) Inside **InMemoryProductDataService.cs**, delete all the existing content and replace it with the following code.

```
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.SessionState;
using ProductManagerMVC.Models;

namespace ProductManagerMVC.Services {

    public class InMemoryProductDataService : IProductDataService {

    }
}
```

At this point, your code will not compile because the **InMemoryProductDataService** class definition does not yet implement all the members of the **IProductDataService** interface.

   e) Using the dropdown menu under **IProductDataService** interface, select the **Implement Interface** command.



f) The **InMemoryProductDataService** class should now contain a skeleton method for each member of **IProductDataService**.

```
namespace ProductManagerMVC.Services {
  public class InMemoryProductDataService : IProductDataService {

    public void AddProduct(Product product) {
      throw new NotImplementedException();
    }

    public void DeleteProduct(int id) {
      throw new NotImplementedException();
    }

    public IQueryable<Product> GetAllProducts() {
      throw new NotImplementedException();
    }

    public Product GetProduct(int id) {
      throw new NotImplementedException();
    }

    public void UpdateProduct(Product product) {
      throw new NotImplementedException();
    }
  }
}
```

In the next step you will copy-and-paste code from a pre-provided text file to create a list of product data. This will give you a set of products to test out your application without forcing you to do too much typing.

g) Using Windows Explorer, locate the file named **ProductListSeedData.cs.txt** in the **Students** folder at the following location.

```
C:\Student\Modules\03_MvcWebApps\Lab\Snippets\ProductListSeedData.cs.txt
```

h) Double-click on **ProductListSeedData.cs.txt** to open this file in Notepad.

i) Copy the contents of **ProductListSeedData.cs.txt** to the Windows clipboard.

j) Return to Visual Studio and place the cursor inside the **InMemoryProductDataService** class at the top.

k) Paste the content of the Windows clipboard at the top of the **InMemoryProductDataService** class.

l) Your class should now contain a new private field named **_productListSeedData** as shown in the following screenshot.

```
namespace ProductManagerMVC.Services {
  public class InMemoryProductDataService : IProductDataService {

    private List<Product> _productListSeedData = new List<Product>() {
            new Product{ Id=1, Name="Batman Action Figure", ListPrice=14.95, Category="Action Figures", Description="A super hero
            new Product{ Id=2, Name="Captain America Action Figure", ListPrice=12.95, Category="Action Figures", Description="A su
            new Product{ Id=3, Name="Easel with Supply Trays", ListPrice=49.95, Category="Arts and Crafts", Description="A serious
            new Product{ Id=4, Name="Crate o' Crayons", ListPrice=14.95, Category="Arts and Crafts", Description="More crayons tha
            new Product{ Id=5, Name="Green Stomper Bully", ListPrice=24.95, Category="Remote Control", Description="A green altern
            new Product{ Id=6, Name="Indy Race Car", ListPrice=19.95, Category="Remote Control", Description="The fastest remote c
            new Product{ Id=7, Name="Twitter Follower Action Figure", ListPrice=1.00, Category="Action Figures", Description="An i
            new Product{ Id=8, Name="Sandpiper Prop Plane", ListPrice=24.95, Category="Remote Control", Description="A simple RC p
            new Product{ Id=9, Name="Etch A Sketch", ListPrice=12.95, Category="Arts and Crafts", Description="A strategic plannin
            new Product{ Id=10, Name="Flying Squirrel", ListPrice=69.95, Category="Remote Control", Description="A stealthy remote
            new Product{ Id=11, Name="FOX News Chopper", ListPrice=29.95, Category="Remote Control", Description="A new chopper wh
            new Product{ Id=12, Name="Godzilla Action Figure", ListPrice=19.95, Category="Action Figures", Description="The classi
            new Product{ Id=13, Name="Perry the Platypus Action Figure", ListPrice=21.95, Category="Action Figures", Description="
            new Product{ Id=14, Name="Seal Team 6 Helicopter", ListPrice=59.95, Category="Remote Control", Description="A serious
            new Product{ Id=15, Name="Crayloa Crayon Set", ListPrice=2.49, Category="Arts and Crafts", Description="A very fun set
    };

    public void AddProduct(Product product) {
      throw new NotImplementedException();
    }
```

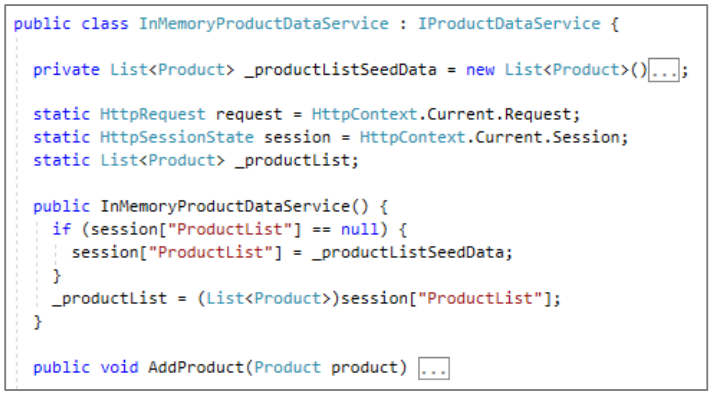
m) Place your cursor just below the **_productListSeedData** field and add the following three static fields.

```
static HttpRequest request = HttpContext.Current.Request;
static HttpSessionState session = HttpContext.Current.Session;
static List<Product> _productList;
```

n) Below the static fields you created in the **InMemoryProductDataService** class, add a constructor using the following code.

```
public InMemoryProductDataService() {
  if (session["ProductList"] == null) {
    session["ProductList"] = _productListSeedData;
  }
  _productList = (List<Product>)session["ProductList"];
}
```

o) Your class now contains a design to initialize an ASP.NET session variable named **ProductList** with sample product data.



```
public class InMemoryProductDataService : IProductDataService {

  private List<Product> _productListSeedData = new List<Product>()...;

  static HttpRequest request = HttpContext.Current.Request;
  static HttpSessionState session = HttpContext.Current.Session;
  static List<Product> _productList;

  public InMemoryProductDataService() {
    if (session["ProductList"] == null) {
      session["ProductList"] = _productListSeedData;
    }
    _productList = (List<Product>)session["ProductList"];
  }

  public void AddProduct(Product product) ...
```

p) Implement the **AddProduct** method with the following code.

```
public void AddProduct(Product product) {
  int nextId = _productList.Max(p => p.Id) + 1;
  product.Id = nextId;
  _productList.Add(product);
  session["ProductList"] = _productList;
}
```

q) Implement the **DeleteProduct** method with the following code.

```
public void DeleteProduct(int id) {
  Product product = _productList.FirstOrDefault(p => p.Id == id);
  _productList.Remove(product);
  session["ProductList"] = _productList;
}
```

r) Implement the **GetAllProducts** method with the following code.

```
public IQueryable<Product> GetAllProducts() {
  return _productList.AsQueryable();
}
```

s) Implement the **GetProduct** method with the following code.

```
public Product GetProduct(int id) {
  return _productList.Find(p => p.Id == id);
}
```

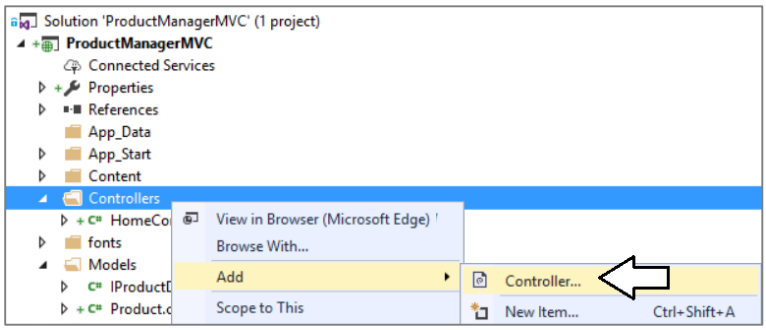
t) Implement the **UpdateProduct** method with the following code

```
public void UpdateProduct(Product product) {
  Product targetProduct = _productList.FirstOrDefault(p => p.Id == product.Id);
  targetProduct.Name = product.Name;
  targetProduct.Category = product.Category;
  targetProduct.ListPrice = product.ListPrice;
  targetProduct.Description = product.Description;
  targetProduct.ProductImageUrl = product.ProductImageUrl;
  session["ProductList"] = _productList;
}
```

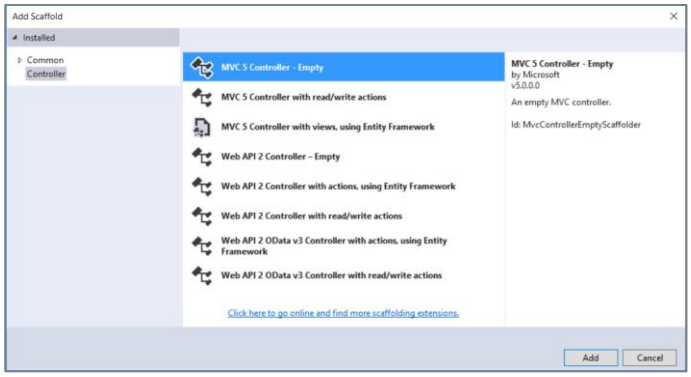
u) Save your changes and close **InMemoryProductDataService.cs**.

4. Create the **ProductsController** class.

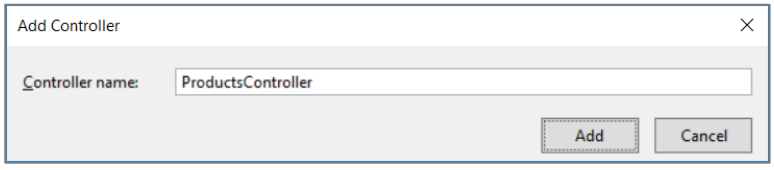
a) Right-click on the **Controllers** folder and click the **Add > Controller** menu command.



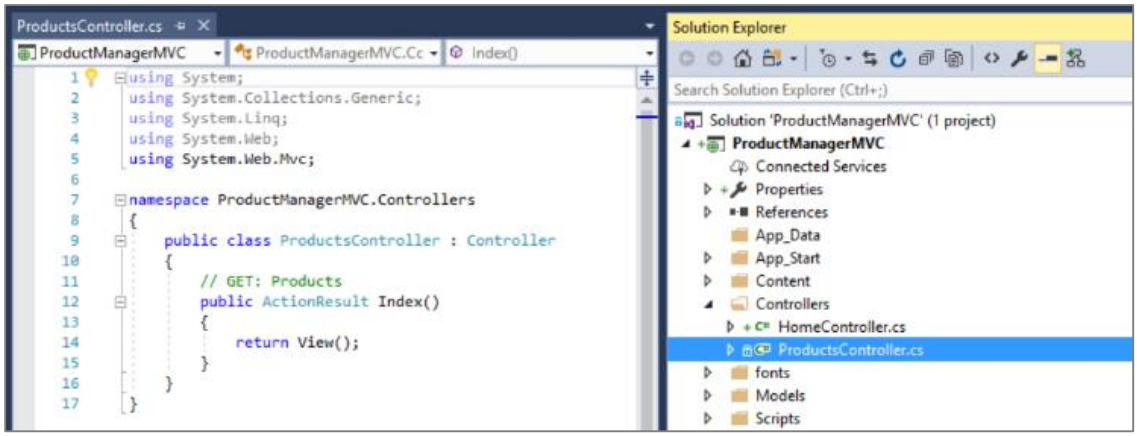
b) In the **Add Scaffold** dialog, select **MVC 5 Controller Empty** and click the **Add** button.



c) In the **Add Controller** dialog, enter a **Controller name** of **ProductsController** and click the **Add** button.



d) There should now be a second controller named **ProductsController** in the **Controllers** folder.



e) Delete all the existing code from **ProductsController.cs**.

f) Add the following **using** statements to the top of **ProductsController.cs**.

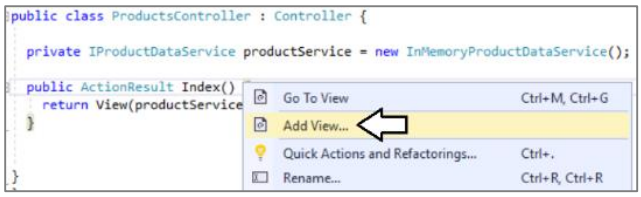
```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Web.Mvc;
using ProductManagerMVC.Models;
using ProductManagerMVC.Services;
```

g) Add the following class definition for the **ProductsController** class.

```
namespace ProductManagerMVC.Controllers {
  public class ProductsController : Controller {

    private IProductDataService productService = new InMemoryProductDataService();

    public ActionResult Index() {
      return View(productService.GetAllProducts());
    }

  }
}
```

5. Add a new view for the **Index** action method of the **ProductsController** class.

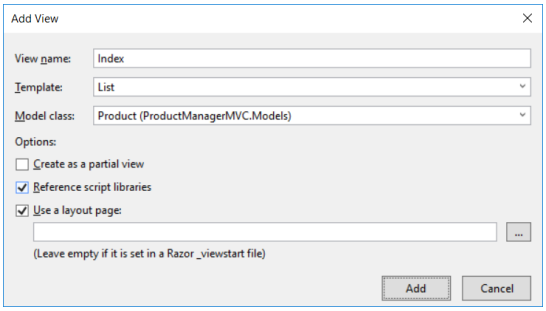
a) Inside the **ProductsController** class definition, right-click on the **Index** method and then click the **Add View…** command.



b) In the **Add View** dialog, set the **Template** property to **List**.

c) Set the **Model class** property to **Product (ProductManagerMVC.Models)**.

d) When the **Add View** dialog matches the following screenshot, click the **Add** button to create the new view.



e) Examine the top of the **Index.cshtml** file after it's been created and see how the **@model** directive defines the view model.

f) Delete the **@{ }** code block at the top which assigns a value to the **Title** property of the **ViewBag** object.

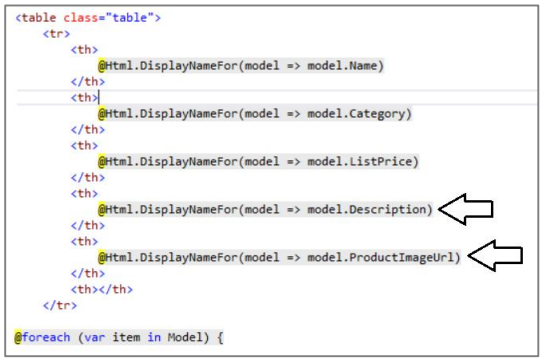
g) Delete **<h2>** tag.

h) Update the first parameter in the call to **@Html.ActionLink** function to pass **"Create Product"** instead of just **"Create"**.

```
<p>
  @Html.ActionLink("Create Product", "Create", "Products", null, new { @class = "btn btn-default" })
</p>
```

i) Move down in the **Index.cshtml** file and examine the code that Visual Studio generated to create a table of products.

j) Locate and delete the two **<th>** elements in the header row for the **Description** property and the **ProductImageUrl** property.



k) Inside the **@foreach** loop below, locate and delete the two **<td>** elements for **Description** and **ProductImageUrl**.

l) The code inside **Index.cshtml** can now be reformatted to match the following screenshot.

```
@model IEnumerable<ProductManagerMVC.Models.Product>

<p>
  @Html.ActionLink("Create Product", "Create", "Products", null, new { @class = "btn btn-default" })
</p>

<table class="table table-bordered table-striped" >
  <tr>
    <th>@Html.DisplayNameFor(model => model.Name)</th>
    <th>@Html.DisplayNameFor(model => model.Category)</th>
    <th>@Html.DisplayNameFor(model => model.ListPrice)</th>
    <th>Actions</th>
  </tr>
@foreach (var item in Model) {
  <tr>
    <td>@Html.DisplayFor(modelItem => item.Name)</td>
    <td>@Html.DisplayFor(modelItem => item.Category)</td>
    <td>@Html.DisplayFor(modelItem => item.ListPrice)</td>
    <td>
      @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
      @Html.ActionLink("Details", "Details", new { id=item.Id }) |
      @Html.ActionLink("Delete", "Delete", new { id=item.Id })
    </td>
  </tr>
}
</table>
```

m) Save your changes to **Index.cshtml**.

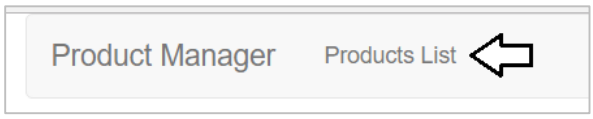
6. Add a navigation link to navigate to the **Index** action of the **Products** controller.

a) Open the shared layouts file named **_Layouts.cshtml** and add a new navigation link as shown in the following code listing.

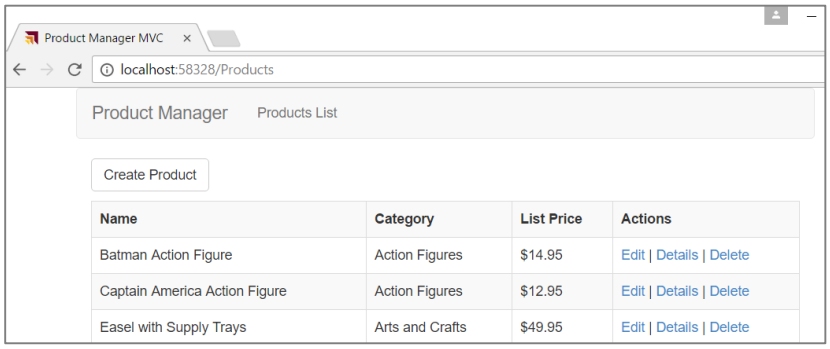
```
<div class="navbar-collapse collapse">
  <ul class="nav navbar-nav">
    <!-- <li> elements with nav links go here -->
    <li>@Html.ActionLink("Product List","Index","Products")</li>
  </ul>
</div>
```

b) Save your changes and close **_Layouts.cshtml**.

7. Test out the **Index** action method of the **Products** controller in the Visual Studio debugger.
   a) Press the **{F5}** key to start a Visual Studio debugging session.
   b) When the application loads, you should see a new navigation link with the caption **Products List**.



   c) Click on the **Products List** navigation link to test the **Index** view of the **Products** controller. This view should render an HTML table with product data as shown in the following screenshot.

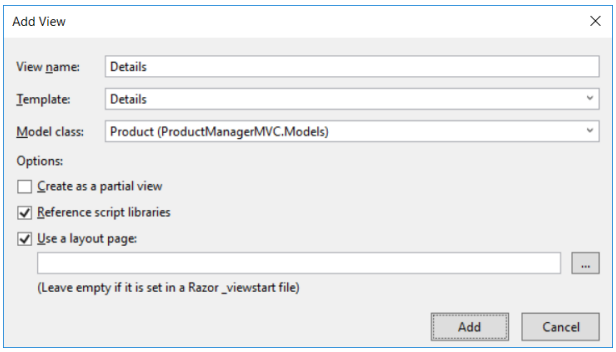


   d) Once you have tested the **Index** view, close the browser window, return to Visual Studio and stop the debugger.

8. Implement the **Details** view of the **Products** controller.
   a) Inside **ProductsContoller.cs** underneath the **Index** method, add the **Details** method using the following code.

```
public ActionResult Details(int? id) {
  if (id == null) {
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
  }
  Product product = productService.GetProduct(id.Value);
  if (product == null) {
    return HttpNotFound();
  }
  return View(product);
}
```

   b) Create a new view for the **Details** method by right-clicking the method name and selecting the **Add View** command.
   c) In the **Add View** dialog, set the **Template** property to **List**.
   d) Set the **Model class** property to **Product (ProductManagerMVC.Models)**.
   e) When the **Add View** dialog matches the following screenshot, click the **Add** button to create the new view.

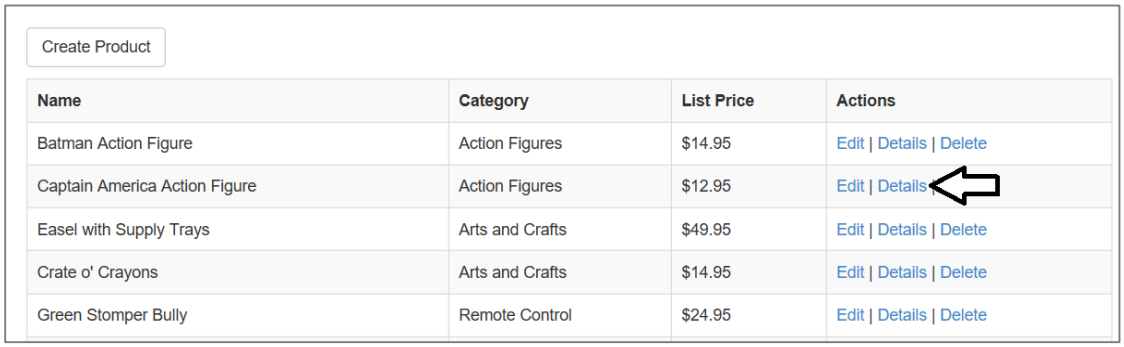


   f) Examine the top of the **Details.cshtml** file after it's been created and see how the **@model** directive defines the view model.

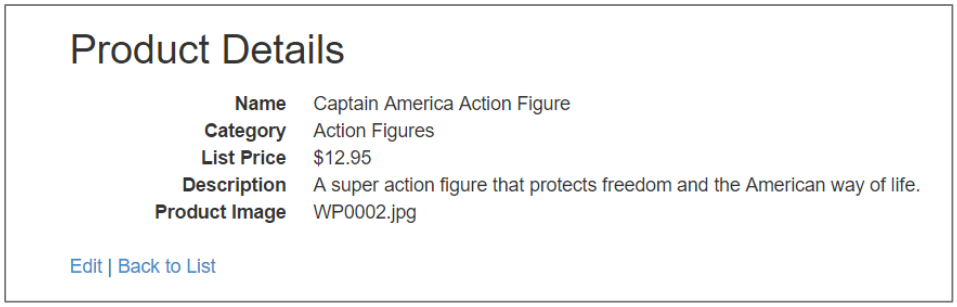
g) Replace with contents of **Details.cshtml** with the following razor view code listing.

```
@model ProductManagerMVC.Models.Product

<h2>Product Details</h2>

<div>
  <dl class="dl-horizontal">

  <dt>@Html.DisplayNameFor(model => model.Name)</dt>
  <dd>@Html.DisplayFor(model => model.Name)</dd>

  <dt>@Html.DisplayNameFor(model => model.Category)</dt>
  <dd>@Html.DisplayFor(model => model.Category)</dd>

  <dt>@Html.DisplayNameFor(model => model.ListPrice)</dt>
  <dd>@Html.DisplayFor(model => model.ListPrice)</dd>

  <dt>@Html.DisplayNameFor(model => model.Description)</dt>
  <dd>@Html.DisplayFor(model => model.Description)</dd>

  <dt>@Html.DisplayNameFor(model => model.ProductImageUrl)</dt>
  <dd>@Html.DisplayFor(model => model.ProductImageUrl)</dd>

</dl>
</div>
<p>
  @Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |
  @Html.ActionLink("Back to List", "Index")
</p>
```

9. Test out the **Details** action method of the **Products** controller in the Visual Studio debugger.
   a) Press the **{F5}** key to start a Visual Studio debugging session.
   b) Navigate to the **Index** view of the **Products** controller.
   c) Click on the **Details** link for one of the products such as the **Captain America Action Figure**.



   d) The Product Details view should match the following screenshot.

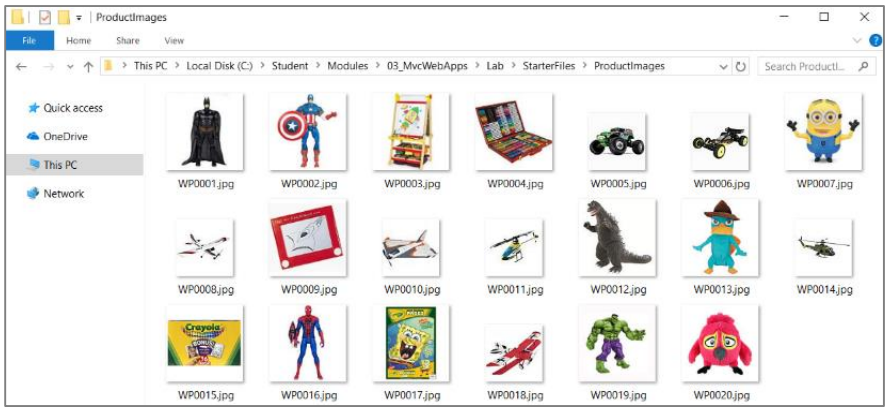


   e) Once you have tested the **Details** view, close the browser window, return to Visual Studio and stop the debugger.

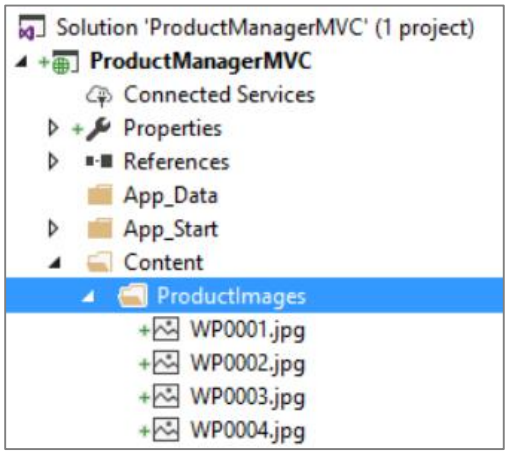
10. Add support to display product images in the **Details** view of the **ProductsController** class.
    a) Using Windows Explorer, locate and inspect the **ProductImages** folder inside the **Students** folder at the following location.

```
C:\Student\Modules\03_MvcWebApps\Lab\StarterFiles\ProductImages\
```

    b) You should see that the **ProductImages** folder contains a set of JPG files with product images.



    c) Copy the **ProductImages** folder and the images inside into the **Content** folder of the **ProductmanagerMVC** project.



    d) Return to the code editor windows for **Details.cshtml** and locate the following two lines of code.

```
<dt>@Html.DisplayNameFor(model => model.ProductImageUrl)</dt>
<dd>@Html.DisplayFor(model => model.ProductImageUrl)</dd>
```

    e) Replace the code inside the **<dd>** tag with the following code.

```
<img src="~/Content/ProductImages/@Model.ProductImageUrl" alt="@Model.Name" width="120" />
```

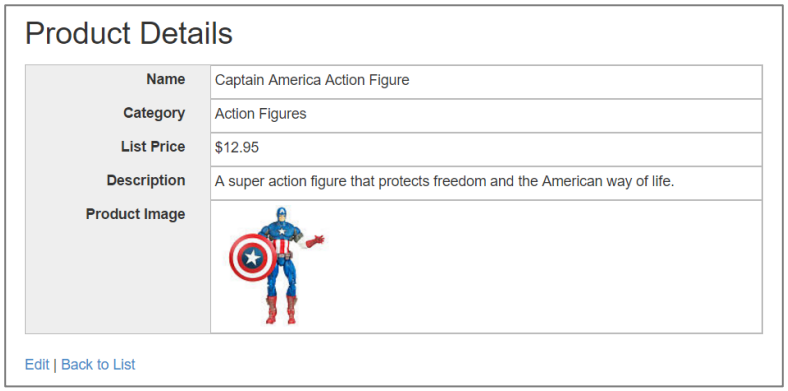
    f) Those two lines should now look like the following code listing.

```
<dt>@Html.DisplayNameFor(model => model.ProductImageUrl)</dt>
<dd><img src="~/Content/ProductImages/@Model.ProductImageUrl" alt="@Model.Name" width="120" /></dd>
```

    g) Save your changes and close **Details.cshtml**.

11. Test out the **Details** action method of the **Products** controller in the Visual Studio debugger.
    a) Press the **{F5}** key to start a Visual Studio debugging session.
    b) Navigate to the **Index** view of the **Products** controller.
    c) Click on the **Details** link for one of the products such as the **Captain America Action Figure**.

d) You should now see a product image in the **Details** view for each product.



e) Once you have tested the **Details** view, close the browser window, return to Visual Studio and stop the debugger.

12. Implement the **Create** action of the **Products** controller.

a) Inside **ProductsContoller.cs** underneath the **Details** method, add two implementations of the **Create** method using the following code.

```
public ActionResult Create() {
  return View();
}

[HttpPost, ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include="Id,Name,Category,ListPrice,Description,ProductImageUrl")]
                           Product product) {
  if (ModelState.IsValid) {
    productService.AddProduct(product);
    return RedirectToAction("Index");
  }
  return View(product);
}
```

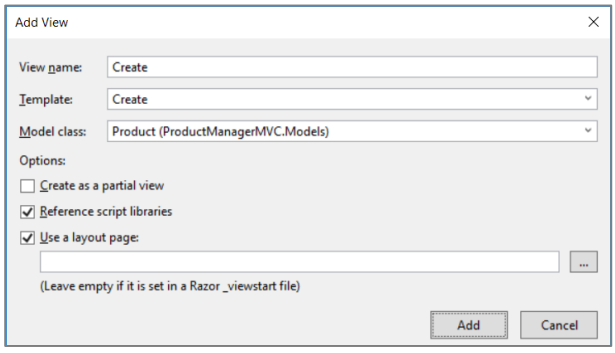
Why are there two implementations of the **Create** action method? The first implementation of **Create** simply returns a view with an HTML form that is used to collect input from the user and to submit an HTML form using an HTTP POST operation. The second implementation of **Create** is the code that executes when the data for a new product is submitted with an HTTP POST request. Note that the second implementation of **Create** is defined using two attributes named **HttpPost** and **ValidateAntiForgeryToken**. The second implementation of **Create** also contains a parameter named **Product** which is defined using the **Bind** attribute to indicate to the MVC framework which product properties in the incoming HTML form should be populated in the **Product** parameter.

a) Create a new view for the **Create** method by right-clicking the top **Create** method and selecting the **Add View** command.

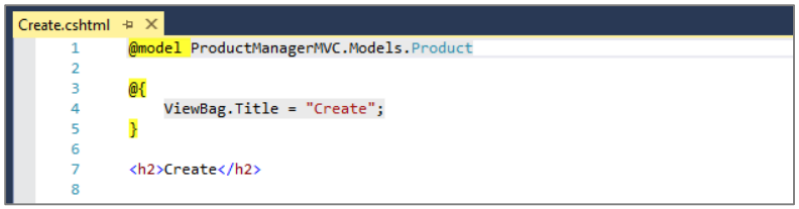
b) In the **Add View** dialog, set the **Template** property to **Create**.

c) Set the **Model class** property to **Product (ProductManagerMVC.Models)**.

d) When the **Add View** dialog matches the following screenshot, click the **Add** button to create the new view.



e) Examine the top of the **Create.cshtml** file after it's been created and see how the **@model** directive defines the view model.



f) Delete the **@{ }** code block at the top which assigns a value to the **Title** property of the **ViewBag** object.

g) Replace the **<h2>** tag with an **<h3>** tag which contains the heading **"Create Product"**.

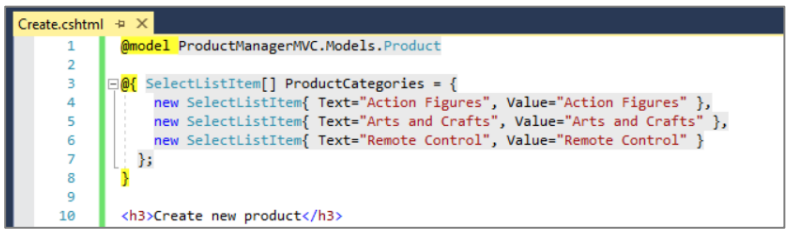
```
@model ProductManagerMVC.Models.Product

<h3>Create Product</h3>

@using (Html.BeginForm()) {
  @Html.AntiForgeryToken()
  <div class="form-horizontal">
```

h) Delete the **<h4>** tag and the **<hr />** tag that have been added underneath the opening **<div class="form-horizontal>** tag.

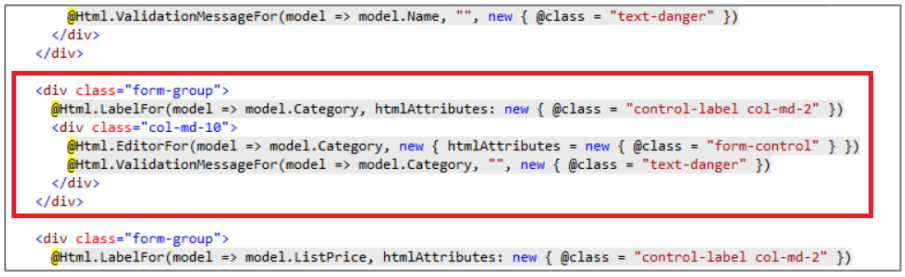
i) Add the following code at the top of **Create.cshtml** under the **@model** directive and above the **<h3>** tag.

```
@{ SelectListItem[] ProductCategories = {
    new SelectListItem{ Text="Action Figures", Value="Action Figures" },
    new SelectListItem{ Text="Arts and Crafts", Value="Arts and Crafts" },
    new SelectListItem{ Text="Remote Control", Value="Remote Control" }
  };
}
```

j) The code at the top of **Create.cshtml** should match the following screenshot.



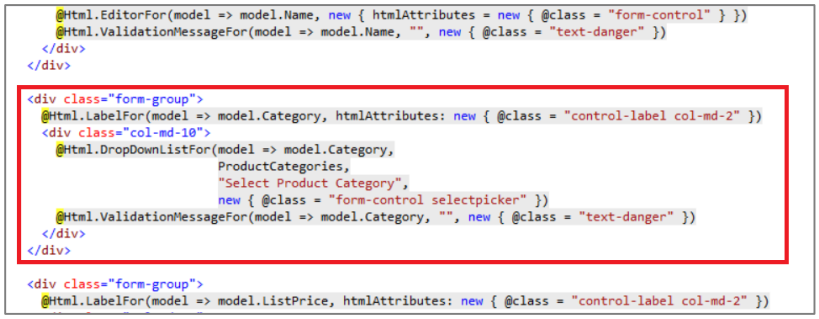
k) Move down inside the code for **Create.cshtml** and find the ** ** element for product category.



l) Replace the code inside that ** ** with the following code to create a dropdown list for selecting a product category.

```
<div class="form-group">
    @Html.LabelFor(model => model.Category, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DropDownListFor(model => model.Category,
                              ProductCategories,
                              "Select Product Category",
                              new { @class = "form-control selectpicker" } )
        @Html.ValidationMessageFor(model => model.Category, "", new { @class = "text-danger" })
    </div>
</div>
```

m) The new HTML code in the ** ** element for the product category section should now match the following screenshot.



n) Move down toward the bottom of **Create.cshtml** and locate the call to **@Html.ActionLink**.

o) Change the text passed to **@Html.ActionLink** to **"Back to Products List"**.

```
<div>
  @Html.ActionLink("Back to Products List", "Index")
</div>
```

p) At the bottom of **Create.cshtml**, you will see a **@section Scripts** section. Leave this code as it is because it is required for the jQuery client-side validation supplied by the MVC framework to work correctly.

```
@section Scripts {
  @Scripts.Render("~/bundles/jqueryval")
}
```

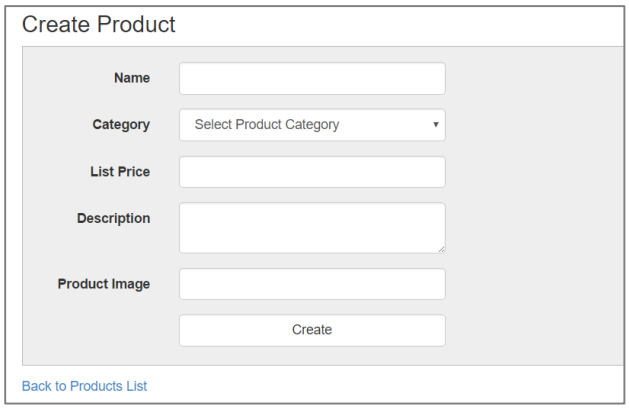
q) Save your changes to **Create.cshtml**.

13. Test out the **Create** action of the **Products** controller.

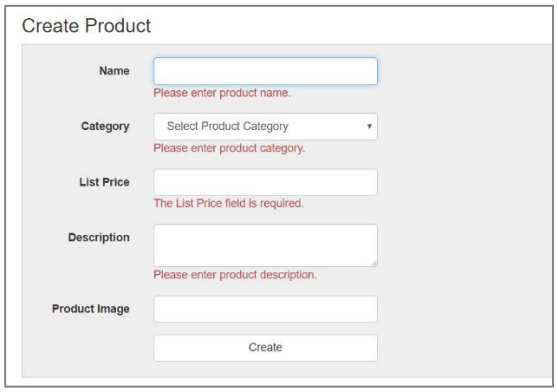
a) Press the **{F5}** key to start a Visual Studio debugging sessions.

b) Navigate to the **Index** view of the **Products** controller and click on the **Create** button.

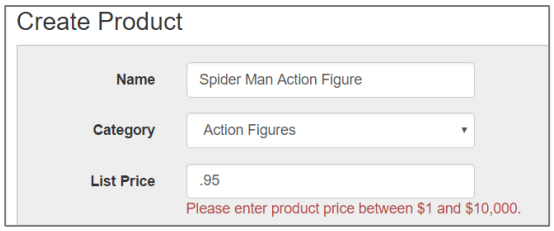
c) You should now see the **Create Product** form as shown in the following screenshot.



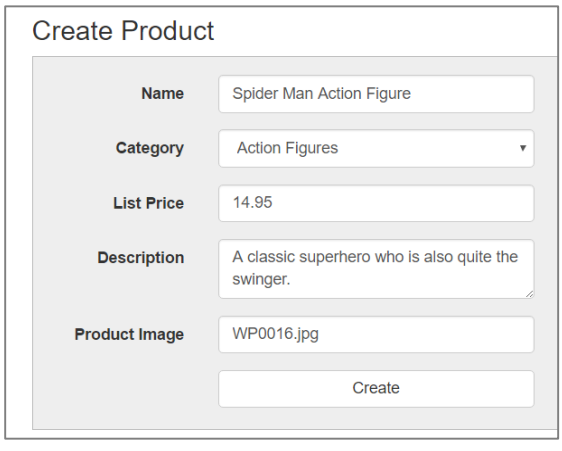
d) Before entering any data into the form, test out what happens if you click the **Create** button and try to create a new product with invalid data. You should see error messages on each input control that requires an entry.



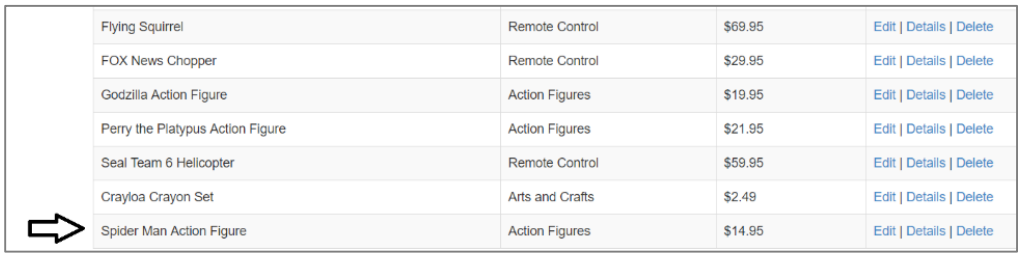
e) Test out what happens when you enter an invalid value of **.95** into the **List Price** textbox. You should see an error message generated from the **Range** attribute you added to the **ListPrice** field in the **Product** class definition.



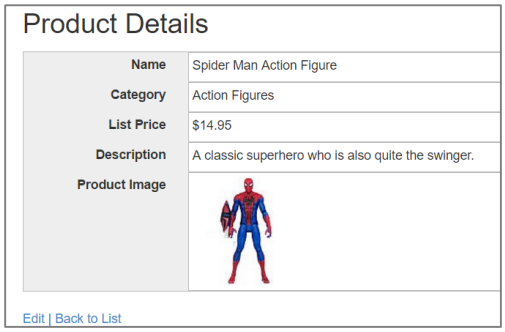
f) Fill out the **Create Product** form using the data shown in the following screenshot and click **Create** to create a new product and make sure to include a **Product Image** value of **WP0016.jpg**.



g) You should be able to verify that the new product has been created by seeing it at the bottom of the products list.



h) Click on the **Details** link to see the details on the new product that has just been created.



The image file named **WP0016.jpg** should display an image of the **Spiderman Action Figure**.

i) Close the browser window, return to Visual Studio and stop the debugger.

14. Implement the **Edit** action of the **Products** controller.

a) Inside **ProductsContoller.cs** underneath **Create**, add two implementations of the **Edit** method using the following code.

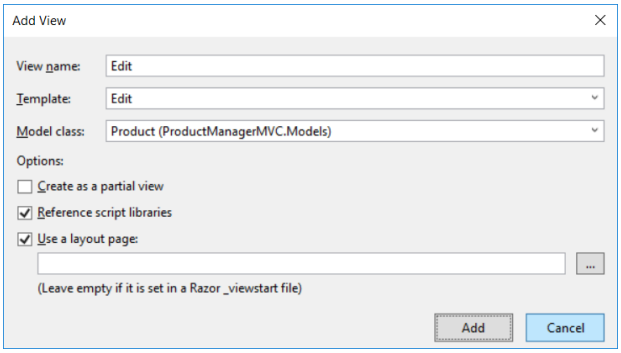
```
public ActionResult Edit(int? id) {
  if (id == null) {
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
  }
  Product product = productService.GetProduct(id.Value);
  if (product == null) {
    return HttpNotFound();
  }
  return View(product);
}

[HttpPost, ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include = "Id,Name,Category,ListPrice,Description,ProductImageUrl")]
                         Product product) {
  if (ModelState.IsValid) {
    productService.UpdateProduct(product);
    return RedirectToAction("Index");
  }
  return View(product);
}
```

b) Create a new view for the **Edit** method by right-clicking the top **Edit** method and selecting the **Add View** command.

c) In the **Add View** dialog, set the **Template** property to **Edit**.

d) Set the **Model class** property to **Product (ProductManagerMVC.Models)**.

e) When the **Add View** dialog matches the following screenshot, click the **Add** button to create the new view.



f) Once the **Edit.cshtml** file has been created, delete its contents.

g) Copy the all the razor view code from **Create.cshtml** and paste it into **Edit.cshtml**.

h) Close **Create.cshtml** and continue to work on **Edit.cshtml**.

i) Update the content of the **<h3>** tag from **"Create Product"** to **"Edit Product"**.

```
<h3>Edit Product</h3>
```

j) Move down to the bottom of on **Edit.cshtml** and locate the **<input type="submit" >** element.

k) Modify the **value** attribute of this **<input>** element to **"Update"**.

```
<input type="submit" value="Update" class="btn btn-default" />
```

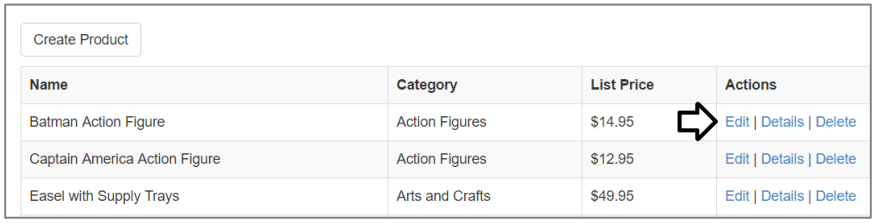
15. Test out the **Edit** action of the **Products** controller.

a) Press the **{F5}** key to start a Visual Studio debugging sessions.

b) Navigate to the **Index** view of the **Products** controller.

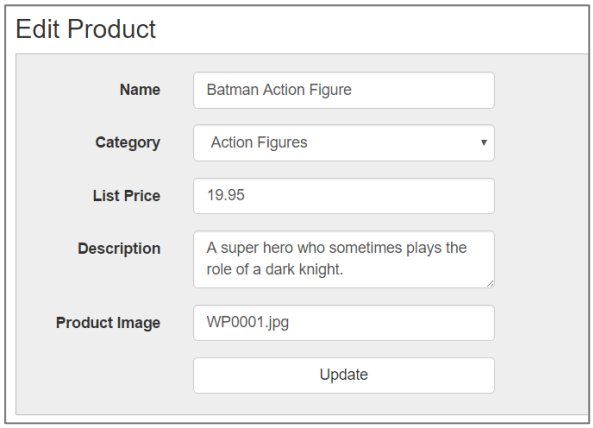
c) Note that the first product named **Batman Action Figure** has a list price of **$14.95**

d) Click the **Edit** link for the product named **Batman Action Figure**.

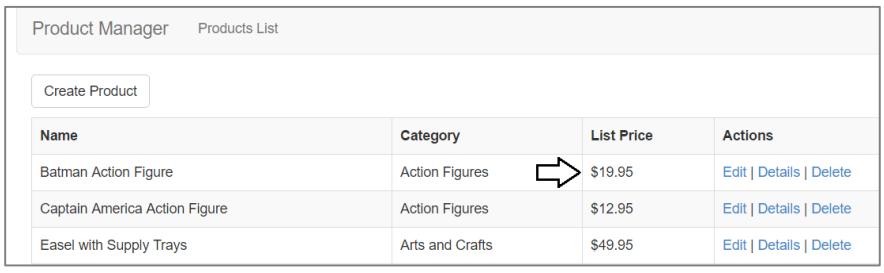


e) You should now see the **Edit Product** form which shows the current state of this product.

f) Update the **List Price** from **$14.95** to **$19.95** and click the **Update** button to save your changes.



g) Verify you can see the change you made to the list price in the product list.

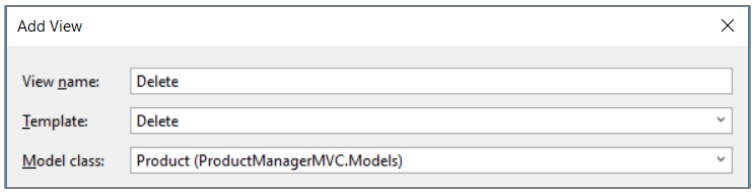


h) Close the browser window, return to Visual Studio and stop the debugger.

16. Implement the **Delete** action of the **Products** controller.
    a) Inside **ProductsContoller.cs** underneath **Edit**, add two implementations of the **Delete** method using the following code.

```
public ActionResult Delete(int? id) {
  if (id == null) {
    return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
  }
  Product product = productService.GetProduct(id.Value);
  if (product == null) {
    return HttpNotFound();
  }
  return View(product);
}

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id) {
  productService.DeleteProduct(id);
  return RedirectToAction("Index");
}
```

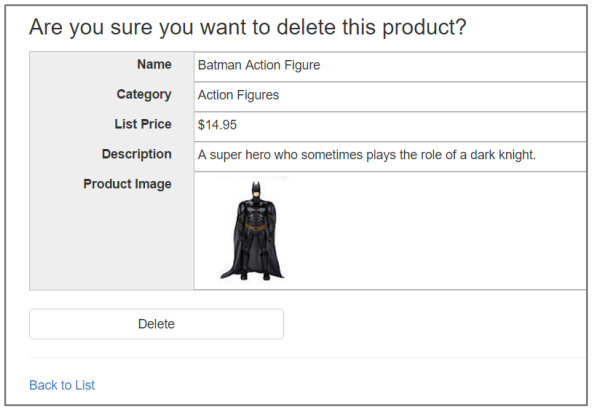
    b) Create a new view for the **Delete** method by right-clicking the top **Delete** method and selecting the **Add View** command.
    c) In the **Add View** dialog, set the **Template** property to **Delete**.
    d) Set the **Model class** property to **Product (ProductManagerMVC.Models)**.
    e) When the **Add View** dialog matches the following screenshot, click the **Add** button to create the new view.



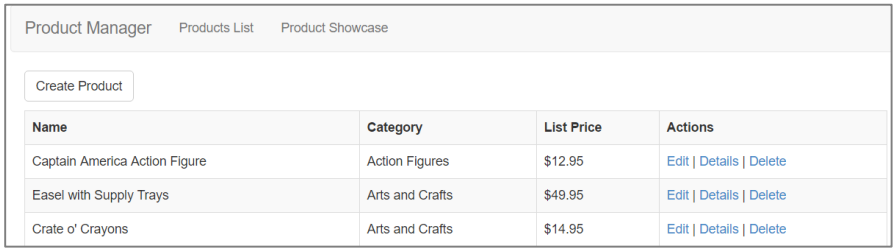
    f) Once Visual Studio has created the **Delete.cshtml** file, delete all the content inside.
    g) Copy and paste the following razor code into **Delete.cshtml**.

```
@model ProductManagerMVC.Models.Product

<h3>Are you sure you want to delete this product?</h3>

<div>
  <dl class="dl-horizontal">
    <dt>@Html.DisplayNameFor(model => model.Name)</dt>
    <dd>@Html.DisplayFor(model => model.Name)</dd>
    <dt>@Html.DisplayNameFor(model => model.Category)</dt>
    <dd>@Html.DisplayFor(model => model.Category)</dd>
    <dt>@Html.DisplayNameFor(model => model.ListPrice)</dt>
    <dd>@Html.DisplayFor(model => model.ListPrice)</dd>
    <dt>@Html.DisplayNameFor(model => model.Description)</dt>
    <dd>@Html.DisplayFor(model => model.Description)</dd>
    <dt>@Html.DisplayNameFor(model => model.ProductImageUrl)</dt>
    <dd><img src="~/Content/ProductImages/@Model.ProductImageUrl" alt="@Model.Name" width="120" /></dd>
  </dl>
</div>

@using (Html.BeginForm()) {
  @Html.AntiForgeryToken()
  <div class="form-actions no-color">
    <input type="submit" value="Delete" class="btn btn-default" />
  </div>
}

<hr/>
@Html.ActionLink("Back to List", "Index")
```

17. Test out the **Delete** action of the **Products** controller in the Visual Studio debugger.
    a) Press the **{F5}** key to start a Visual Studio debugging session.
    b) Navigate to the **Index** view of the **Products** controller.
    c) Click the **Delete** link for the product named **Batman Action Figure**.
    d) You should now see the form that requires to the user to confirm the operation of deleting a product.
    e) Click the **Delete** button to delete the **Batman Action Figure** product.



    f) Verify that the **Batman Action Figure** product has been deleted from the products list.

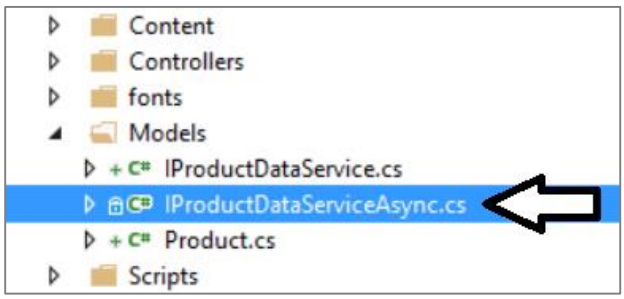


    g) Close the browser, return to Visual Studio and stop the debugger.

Exercise 3: Create an Asynchronous Controller Class

In this exercise, you will begin to work with the asynchronous method execution support in the .NET framework and with asynchronous controller support in ASP.NET MVC. You will begin by creating a new asynchronous version of the **IProductDataService** interface and then you will add some pre-provided code to your project that implements the asynchronous interface. After that, you will modify the action methods of the **ProductsController** class to execute in an asynchronous fashion.

1. Create the **IProductDataServiceAsync** interface.
    a) In **Solution Explorer**, right-click on the **Models** folder and click the **Add > Class…** menu command.
    b) In the **Add New Item** dialog, give the new class file a name of **IProductDataServiceAsync.cs**.



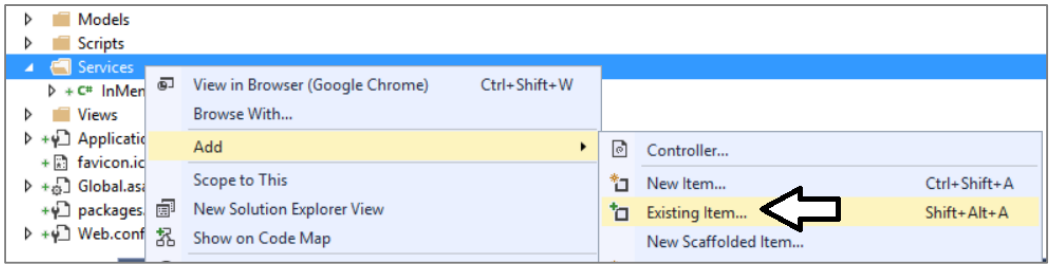
c) Delete the contents of **IProductDataServiceAsync.cs** and replace it with the following code.

```
using System.Linq;
using System.Threading.Tasks;

namespace ProductManagerMVC.Models {

  public interface IProductDataServiceAsync {
    Task<IQueryable<Product>> GetAllProductsAsync();
    Task<Product> GetProductAsync(int id);
    Task AddProductAsync(Product product);
    Task DeleteProductAsync(int id);
    Task UpdateProductAsync(Product product);
  }

}
```

d) Save your changes and close **IProductDataServiceAsync.cs**.

2. Import code into your project to provide an implementation for the **FileBasedProductDataService** class.

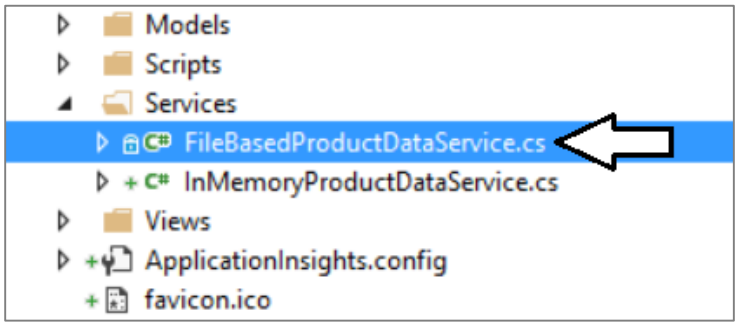
a) Right-click on the **Services** folder and select the **Add > Existing Item…** command.



b) When prompted for a file path, enter the following path to the C# source file named **FileBasedProductDataService.cs**.

```
C:\Student\Modules\03_MvcWebApps\Lab\StarterFiles\FileBasedProductDataService.cs
```

c) The source file named **FileBasedProductDataService.cs** should now be part of your project.



d) Examine the code inside **FileBasedProductDataService.cs**.

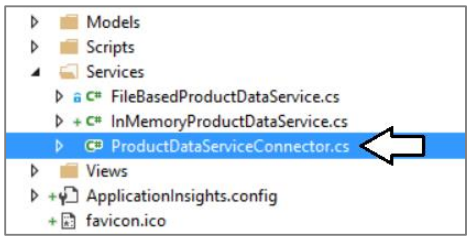
e) You can see that the **FileBasedProductDataService** class implements the **IProductDataServiceAsync** interface.

```
public class FileBasedProductDataService : IProductDataServiceAsync {
  // implemention omitted for brevity
}
```

f) Quickly review the code inside the **FileBasedProductDataService** class. There is no need to modify any of this code, but spend a minute to see how it reads and writes product data to a file named **productList.json** in the **App_Data** folder.

Like the **InMemoryProductDataService** class, the **FileBasedProductDataService** class is best used in proof-of-concept demos that only involve a single user. It contains no logic for dealing with concurrency or synchronizing update operations across multiple user.

3. Create the **ProductDataServiceConnector** class to completely decouple controller classes from data access code.
   a) In **Solution Explorer**, right-click on the **Models** folder and click the **Add > Class…** menu command.
   b) In the **Add New Item** dialog, give the new class file a name of **ProductDataServiceConnector.cs**.



   c) Implement the **ProductDataServiceConnector** class by adding two methods named **GetProductDataService** and **GetAsyncProductDataService** as shown in the following code listing.

```
using ProductManagerMVC.Models;

namespace ProductManagerMVC.Services {

  public class ProductDataServiceConnector {
    public static IProductDataService GetProductDataService() {
      return new InMemoryProductDataService();
    }
    public static IProductDataServiceAsync GetAsyncProductDataService() {
      return new FileBasedProductDataService();
    }
  }

}
```

   d) Save your changes and close **ProductDataServiceConnector.cs**.

4. Update the **ProductsController** class to use the **ProductDataServiceConnector** class.
   a) Open **ProductsController.cs** in a code editor window.
   b) You should see that there is code to initialize the **productService** field by using the C# **new** operator to create a new instance of the **InMemoryProductDataService** class.

```
namespace ProductManagerMVC.Controllers {
  public class ProductsController : Controller {

    private IProductDataService productService = new InMemoryProductDataService();

    public ActionResult Index() {
      return View(productService.GetAllProducts());
    }
...
```

   c) Rewrite the code to initialize the **productService** field by calling **ProductDataServiceConnector.GetProductDataService**.

```
private IProductDataService productService = ProductDataServiceConnector.GetProductDataService();
```

5. Test out the **ProductManagerMVC** project in the Visual Studio Debugger.
   a) Press the **{F5}** key to start a Visual Studio debugging session.
   b) At this point, the application should work just as it did before. The only difference is now the **ProductsController** uses the **ProductDataServiceConnector** class to eliminate dependencies on any specific data service class implementation.

6. Update the actions of the **ProductsController** class to support asynchronous execution.
   a) Update the type of the **productService** field to be **IProductDataServiceAsync** instead of **IProductDataService**.
   b) Initialize the **productService** field by calling **ProductDataServiceConnector.GetAsyncProductDataService**.

```
private IProductDataServiceAsync productService =
        ProductDataServiceConnector.GetAsyncProductDataService();
```

c) After the last change, you should notice that the code in the **ProductsController** class no longer compiles. For example, there is a compile-time error in the **Index** action method in the code that calls **GetAllProducts**.

```
private IProductDataServiceAsync productService =

public ActionResult Index() {
  return View(productService.GetAllProducts());
}
```

d) Add the following **using** statement to the top of **ProductsController.cs**.

```
using System.Threading.Tasks;
```

e) Move down to the **Index** action method and examine the top line which defines the method signature.

```
public ActionResult Index()
```

f) Change the signature of the **Index** method to support asynchronous execution.

```
public async Task<ActionResult> Index()
```

g) Modify the implementation of the **Index** method to call **GetAllProductsAsync** using the **await** operator.

```
public async Task<ActionResult> Index() {
    return View(await productService.GetAllProductsAsync());
}
```

h) Change the signature of the **Details** method to support asynchronous execution.

```
public async Task<ActionResult> Details(int? id)
```

i) Modify the implementation of the **Details** method to call **GetProductAsync** using the **await** operator.

```
Product product = await productService.GetProductAsync(id.Value);
```

j) Leave the signature for the first **Create** method the way it is.

k) Change the signature of the second **Create** method to support asynchronous execution.

```
public async Task<ActionResult> Create
```

l) Modify the implementation of the second **Create** method to call **AddProductAsync** using the **await** operator.

```
await productService.AddProductAsync(product);
```

m) Change the signature of the first **Edit** method to support asynchronous execution.

```
public async Task<ActionResult> Edit(int? id)
```

n) Modify the implementation of the first **Edit** method to call **GetProductAsync** using the **await** operator.

```
Product product = await productService.GetProductAsync(id.Value);
```

o) Change the signature of the second **Edit** method to support asynchronous execution.

```
[HttpPost, ValidateAntiForgeryToken]
public async Task<ActionResult> Edit
```

p) Modify the implementation of the second **Edit** method to call **UpdateProductAsync** using the **await** operator.

```
await productService.UpdateProductAsync(product);
```

q) Change the signature of the **Delete** method to support asynchronous execution.

```
public async Task<ActionResult> Delete(int? id)
```

r) Modify the implementation of the **Delete** method to call **GetProductAsync** using the **await** operator.

```
Product product = await productService.GetProductAsync(id.Value);
```

s) Change the signature of the **DeleteConfirmed** action method to support asynchronous execution.

```
[HttpPost, ActionName("Delete"), ValidateAntiForgeryToken]
public async Task<ActionResult> DeleteConfirmed(int id)
```

t) Modify the implementation of the **DeleteConfirmed** method to call **DeleteProductAsync** using the **await** operator.

```
[HttpPost, ActionName("Delete"), ValidateAntiForgeryToken]
public async Task<ActionResult> DeleteConfirmed(int id) {
  await productService.DeleteProductAsync(id);
  return RedirectToAction("Index");
}
```

u) Save your changes to the **ProductsController** class.

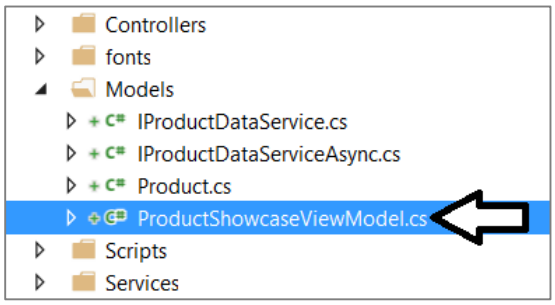
7. Run the **ProductManagerMVC** project in the Visual Studio debugger to test out the new asynchronous controller support.
   a) Press the **{F5}** key to start a Visual Studio debugging sessions.
   b) Test out all the views of the **Products** controller to make sure they all still work as they did before.

The **Products** controller should continue to work just as it did before The only difference is now the action methods of the **Products** controller class are being executed in an asynchronous fashion which makes your code more scalable when run in an Azure web app.

Exercise 4: Create a Controller with a Custom View Model

In this exercise you will create a new controller class named **ProductShowcaseController** to provide an enhanced user interface experience for browsing through the available catalog of products. You will also create a class named **ProductShowcaseViewModel** to pass a strongly-typed object from the **ProductShowcaseController** class to its underling view.

1. Create a new class named **ProductShowcaseViewModel** to serve as a view model class.
   a) In **Solution Explorer**, right-click on the **Models** folder and click the **Add > Class…** menu command.
   b) In the **Add New Item** dialog, give the new class file a name of **ProductShowcaseViewModel.cs**.



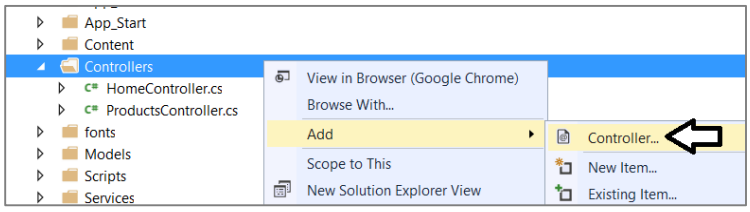
   c) Delete the contents of **ProductShowcaseViewModel.cs** and replace it with the following code.

```
using System.Collections.Generic;
using System.Linq;

namespace ProductManagerMVC.Models {

  public class ProductShowcaseViewModel {
    public IQueryable<Product> Products { get; set; }
    public IEnumerable<string> Categories{ get; set; }
  }

}
```

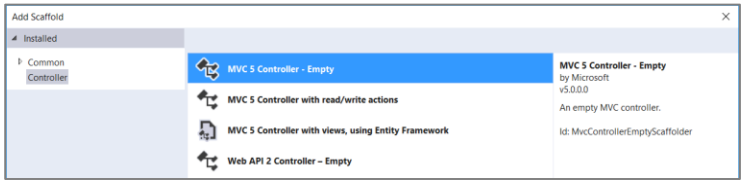
   d) Save your changes and close **ProductShowcaseViewModel.cs**.

2. Create the **ProductShowcaseController** class.

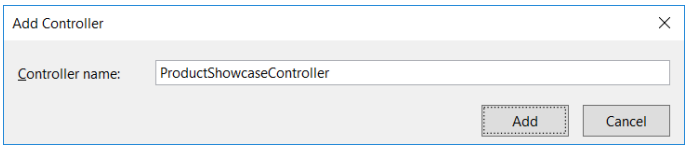
   a) Right-click on the **Controllers** folder and click the **Add > Controller** menu command.



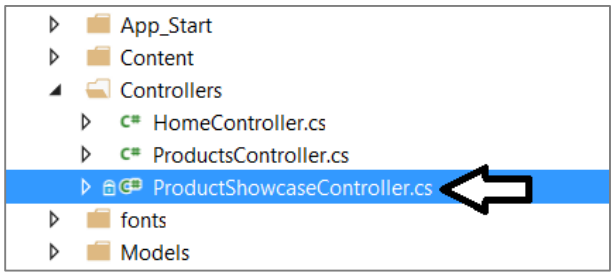
   b) In the **Add Scaffold** dialog, select **MVC 5 Controller Empty** and click the **Add** button.



   c) In the **Add Controller** dialog, enter a **Controller name** of **ProductShowcaseController** and click the **Add** button.



   d) There should now be a third source file named **ProductShowcaseController.cs** in the **Controllers** folder.



   e) Delete the contents of **ProductShowcaseController.cs** and replace it with the following code.

```
using System.Linq;
using System.Web.Mvc;
using System.Threading.Tasks;
using ProductManagerMVC.Models;
using ProductManagerMVC.Services;
using System.Collections.Generic;

namespace ProductManagerMVC.Controllers {

  public class ProductShowcaseController : Controller {

    private IProductDataServiceAsync productService =
                                    ProductDataServiceConnector.GetAsyncProductDataService();

    public async Task<ActionResult> Index(string categoryFilter) {
      // add implementation here
    }
  }
}
```

f) Implement the **Index** method using the following code.

```
public async Task<ActionResult> Index(string categoryFilter) {

  var products = await productService.GetAllProductsAsync();

  // get unique list of product categories
  IEnumerable<string> categories = products.Select(p => p.Category)
                                           .Distinct()
                                           .OrderBy(cat => cat);

  // filter by category if query string parameter named categoryFilter has a value
  if (!string.IsNullOrEmpty(categoryFilter)) {
    products = products.Where(p => p.Category.Equals(categoryFilter));
  }

  // create instance of view model to pass to view
  ProductShowcaseViewModel model = new ProductShowcaseViewModel {
    Products = products,
    Categories = categories
  };

  return View(model);
}
```

g) Create a new view for the **Index** method by right-clicking the method and selecting the **Add View** command.

h) In the **Add View** dialog, click the **Add** button to create the new view.

i) Once Visual Studio has created the **Index.cshtml** view file for the **ProductShowcase** controller, delete all the code inside.

j) Using Windows Explorer, locate the file named **ProductShowcase_Index.cshtml.txt** at the following location.

```
C:\Student\Modules\03_MvcWebApps\Lab\Snippets\ProductShowcase_Index.cshtml.txt
```

k) Double-click on **ProductShowcase_Index.cshtml.txt** to open this file in Notepad.

l) Copy the contents of **ProductShowcase_Index.cshtml.txt** to the Windows clipboard.

m) Return to Visual Studio and paste the content of the Windows clipboard into **Index.cshtml**.

n) Examine the structure of the HTML code inside **Index.cshtml**. where you will find a **sidebar** section and a **main** section.

```
<div class="row">

  <div class="container">
    <div class="row row-offcanvas row-offcanvas-left">

      <!-- sidebar -->
      <div id="sidebar" class="col-xs-6 col-sm-2 sidebar-offcanvas">...</div>

      <!-- main area -->
      <div class="col-xs-12 col-sm-10">...</div>

    </div>
  </div>
</div>
```

o) Examine the razor view code in the **sidebar** section.

```
<!-- sidebar -->
<div class="col-xs-6 col-sm-2 sidebar-offcanvas" id="sidebar" role="navigation">
  <nav id="left-nav" class="navbar" role="navigation">
    <div id="left-nav-title">Filter by Category</div>
    <ul class="nav navbar-nav">
      <li>
        @Html.ActionLink("All Categories",
                "Index",
                "ProductShowcase",
                null,
                new { @class = "nav navbar-link" })
      </li>
      @foreach (var category in Model.Categories) {
        <li>
          @Html.ActionLink(category,
                  "Index",
                  "ProductShowcase",
                  new { categoryFilter = category },
                  new { @class = "nav navbar-link" })
        </li>
      }
    </ul>
  </nav>
</div>
```

p) Examine the razor view code in the main area section. There is a top-level **@foreach** loop that generates an instance of a templated div element for each product in the products list.

```
<!-- main area -->
<div class="col-xs-12 col-sm-10">
  @foreach (var product in Model.Products) {
    string productImageUrl = "../Content/ProductImages/" + product.ProductImageUrl;
    <div class="col-lg-4 col-md-5 col-sm-6">...</div>
  }
</div>
```

q) Examine the razor code inside the ** ** template for displaying product information.

```
<!-- main area -->
<div class="col-xs-12 col-sm-10">
  @foreach (var product in Model.Products) {
    string productImageUrl = "../Content/ProductImages/" + product.ProductImageUrl;
    <div class="col-lg-4 col-md-5 col-sm-6">
      <div class="panel panel-primary productPanel">
        <div class="panel-heading">@product.Name</div>
        <div class="panel-body">
          <img src="@productImageUrl" class="product-image" />
          <p class="text-info">@product.Description</p>
          <div class="" style="clear: both;">
            <table class="table">
              <tr>
                <td>Category:</td>
                <td>@product.Category</td>
              </tr>
              <tr>
                <td>List Price:</td>
                <td>@product.ListPrice.ToString("$0.00")</td>
              </tr>
            </table>
          </div>
        </div>
      </div>
    </div>
  }
</div>
```

r) Save your changes and close **Index.cshtml**.

3. Add a navigation link to the **ProductSowcase** controller.

a) Open the shared layouts file named _**Layouts.cshtml** and add a new navigation link as shown in the following code listing.

```
<ul class="nav navbar-nav">
  <!-- <li> elements with nav links go here -->
  <li>@Html.ActionLink("Products List","Index","Products")</li>
  <li>@Html.ActionLink("Product Showcase", "Index", "ProductShowcase")</li>
</ul>
```

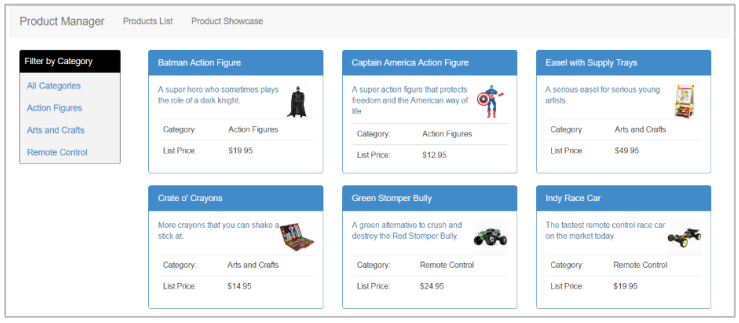
b) Save your changes and close _**Layouts.cshtml**.

4. Test out the **ProductShowcase** controller.

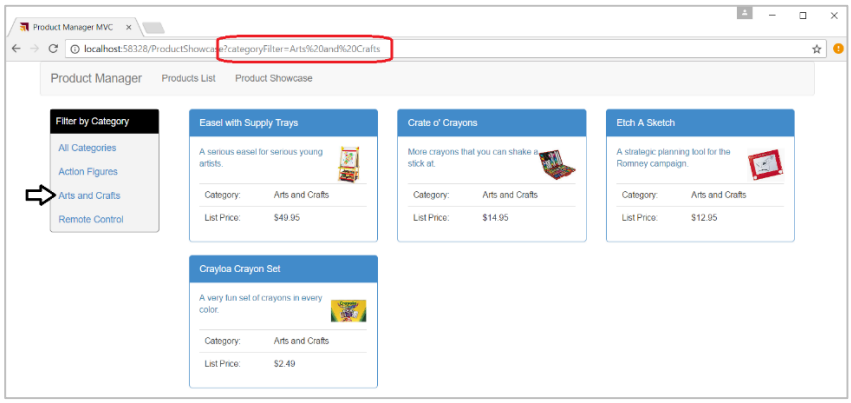
a) Press the **{F5}** key to start a Visual Studio debugging sessions.

b) When the application loads, click the **Product Showcase** navigation link.

c) You should see product data displayed in a flowing layouts that matches the following screenshot.



d) Click on each of the categories in the left navigation menu. When you click on a specific product category, the page should refresh just showing the products in that category.



e) Once you have tested the application, close the browser, return to Visual Studio and stop the debugger.

Congratulations. You have now made it to the end of this lab.

If you have more time and you are up for a challenge, see how long it takes you to deploy the **ProductManagerMVC** project to a new Azure web app with a URL ending with **azurewebsites.net** to make it accessible to anyone on the Internet.