WEB APPLICATION DEVELOPMENT WITH ASP.NET MVC 5

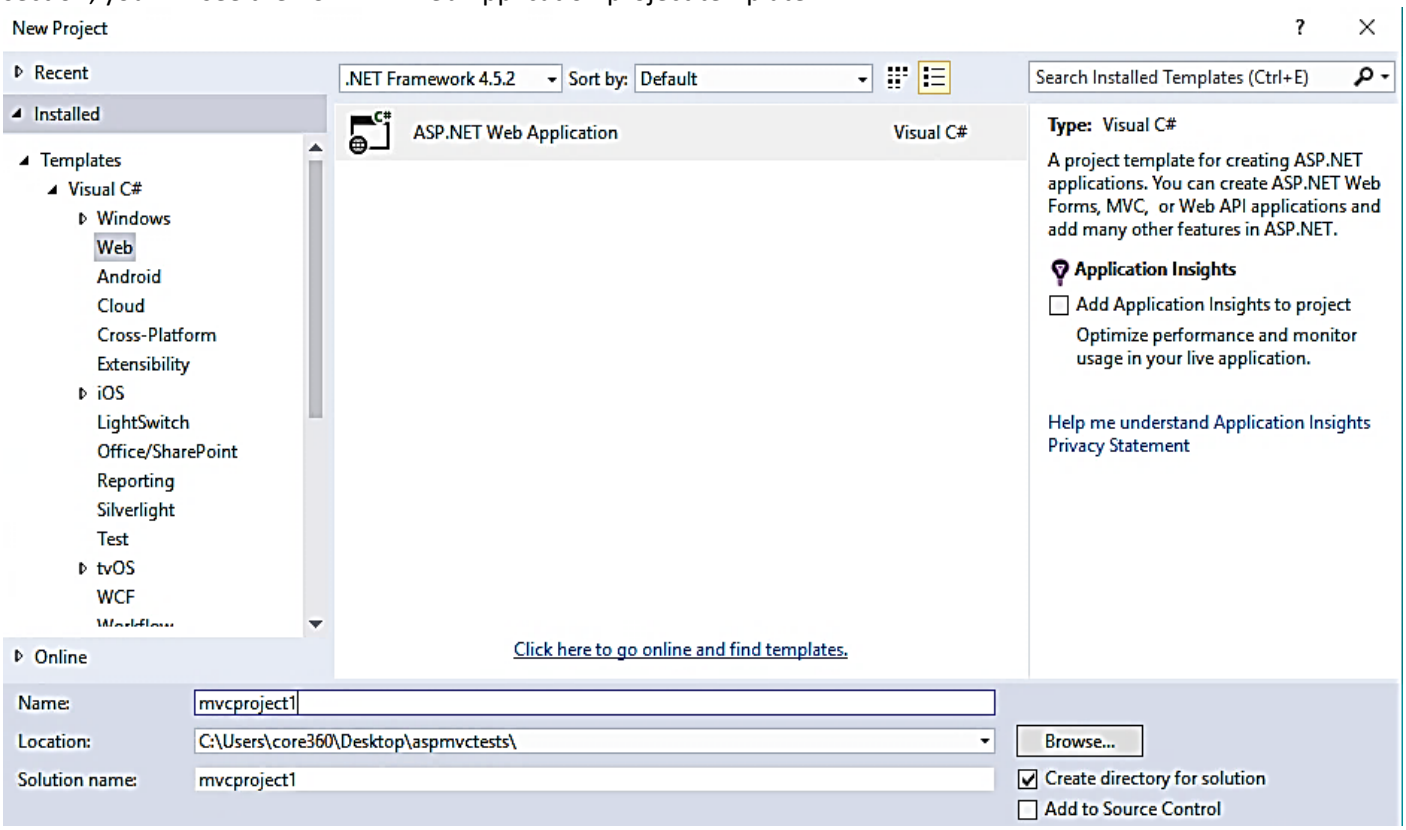
Contents

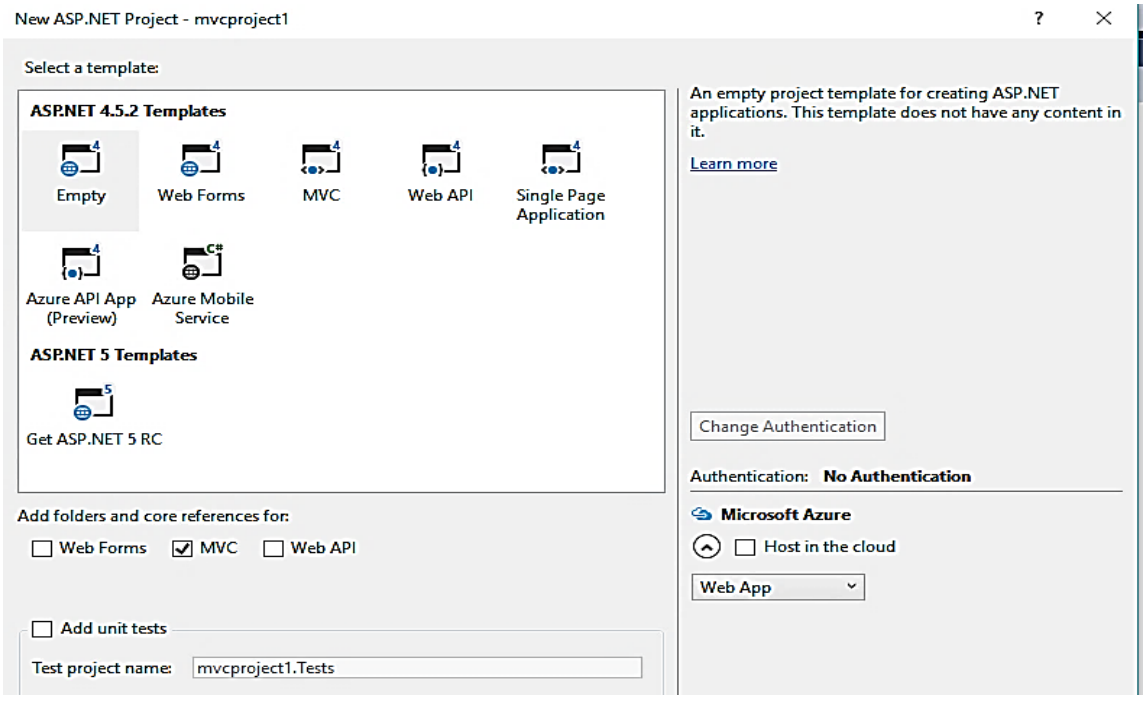
MODULE 1: USING THE ASP.NET MVC ARCHITECTURE

The MVC Architecture

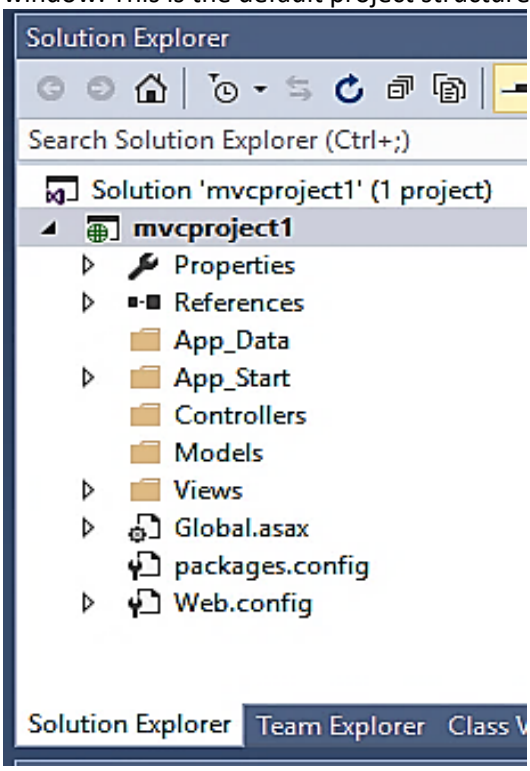
Creating a New ASP.NET MVC Project

Select New Project from the File menu to open the New Project dialog. If you select the Web templates in the Visual C# section, you will see the ASP.NET Web Application project template.

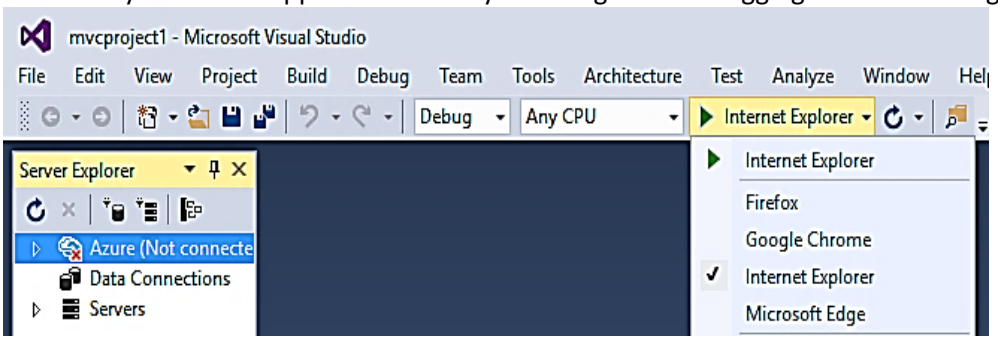




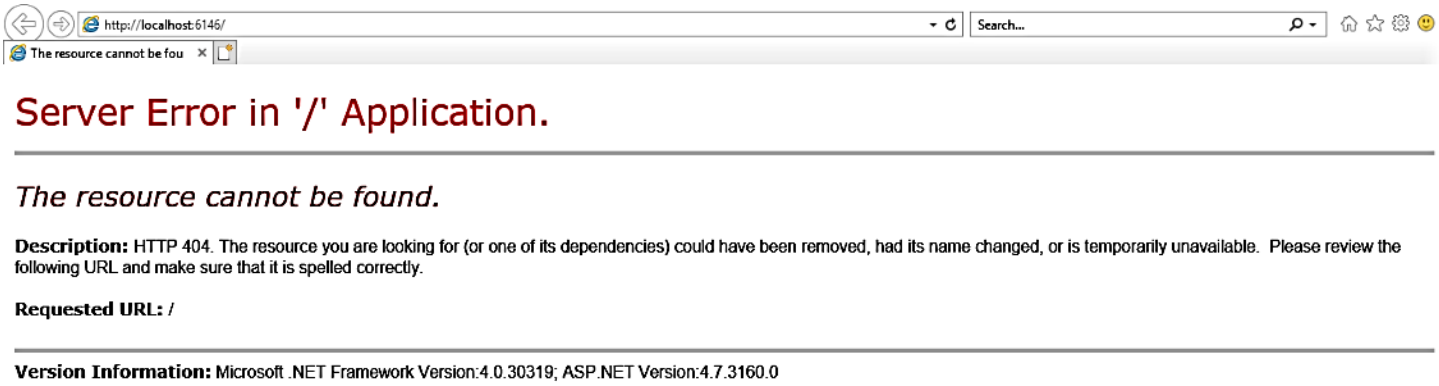
Once Visual Studio creates the project, you will see a number of files and folders displayed in the Solution Explorer window. This is the default project structure for a new MVC project:



You can try to run the application now by selecting Start Debugging from the Debug menu



With the empty project template, the application does not contain anything to run, so the server generates a 404 Not Found Error.

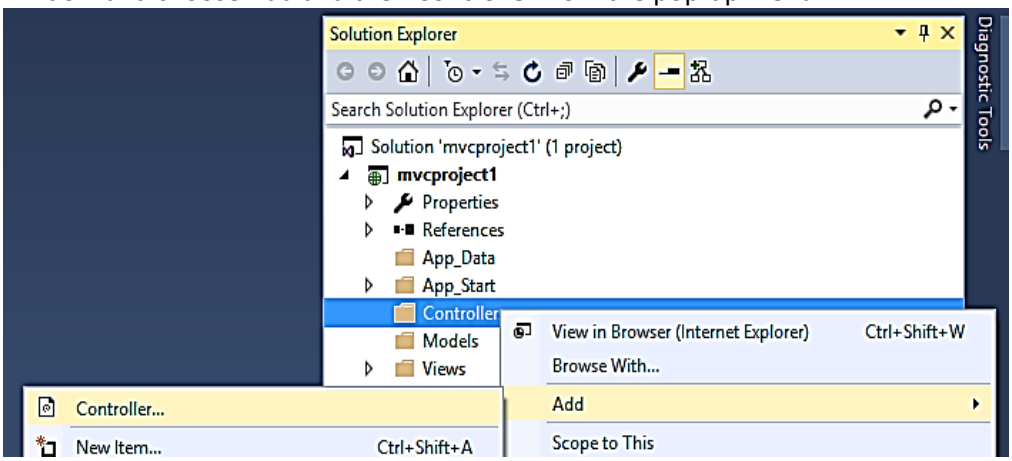


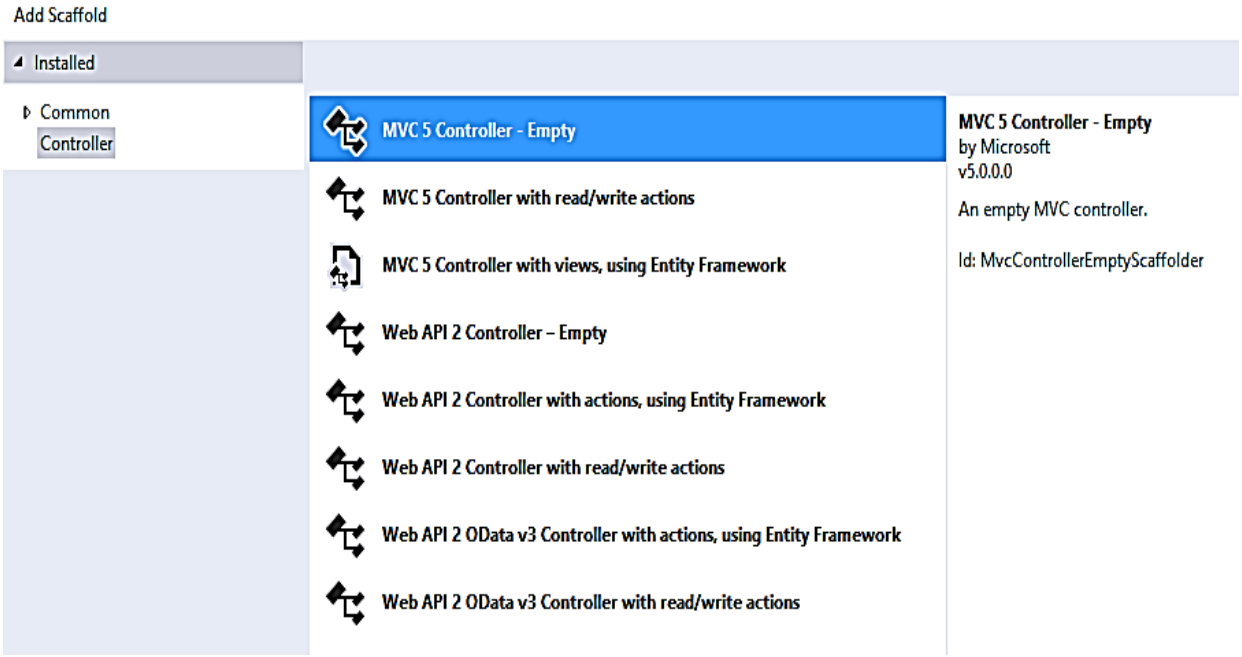
Adding the First Controller

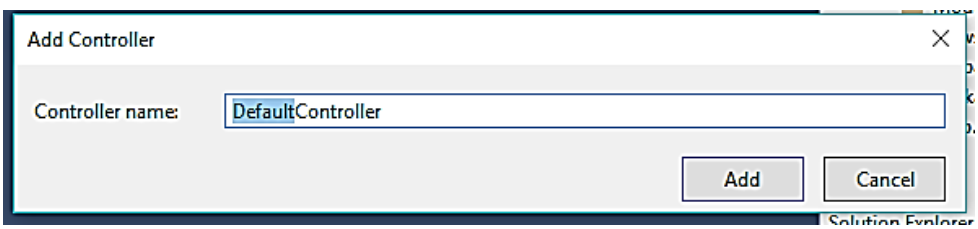
In MVC architecture, incoming requests are handled by controllers. In ASP.NET MVC, controllers are just C# classes (usually inheriting from System.Web.Mvc.Controller, the framework's built-in controller base class).

Each public method in a controller is known as an action method, meaning you can invoke it from the Web via some URL to perform an action. The MVC convention is to put controllers in the Controllers folder, which Visual Studio created when it set up the project.

To add a controller to the project, right-click the Controllers folder in the Visual Studio Solution Explorer window and choose Add and then Controller from the pop-up menu:







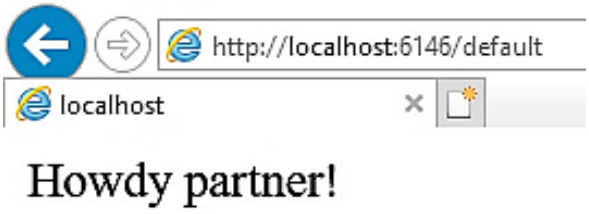
The default code for an empty controller:

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

namespace mvcproject1.Controllers
{
   public class DefaultController : Controller
   {
      // GET: Default
      public ActionResult Index()
      {
         return View();
      }
   }
}
```

Lets try editing it and displaying an output:

```
   public class DefaultController : Controller
   {
      // GET: Default
      public string Index()
      {
         return "Howdy partner!";
      }
   }
```

To see the result, we need to access the controller name through the URL:



Understanding Routes

As well as models, views, and controllers, MVC applications use the ASP.NET routing system, which decides how URLs map to controllers and actions. When Visual Studio creates the MVC project, it adds some default routes to get us started. You can request any of the following URLs, and they will be directed to the Index action on the DefaultController:

```
/
/Default
/Default/Index
```

You can see and edit your routing configuration by opening the RouteConfig.cs file in the App_Start folder.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace mvcproject1
{
   public class RouteConfig
   {
      public static void RegisterRoutes(RouteCollection routes)
      {
         routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

         routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Default", action = "Index", id = UrlParameter.Optional }
         );
      }
   }
}
```

Creating and Rendering a View

Modify Index action method:

```
public ViewResult Index()
      {
         return View();
      }
```

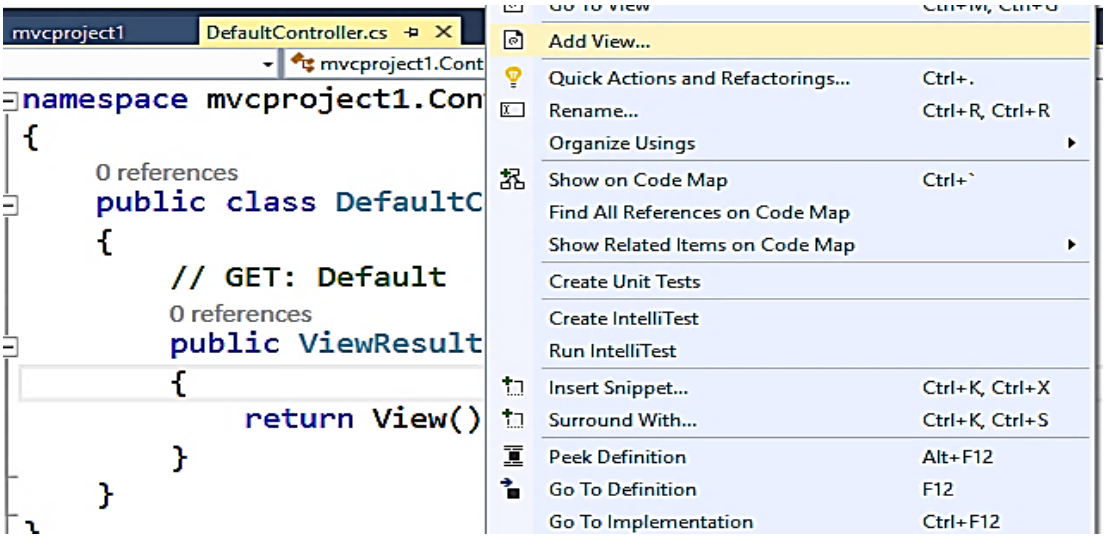
When I return a ViewResult object from an action method, I am instructing MVC to render a view. I create the ViewResult by calling the View method with no parameters. This tells MVC to render the default view for the action. If you run the application at this point, you can see the MVC Framework trying to find a default view to use

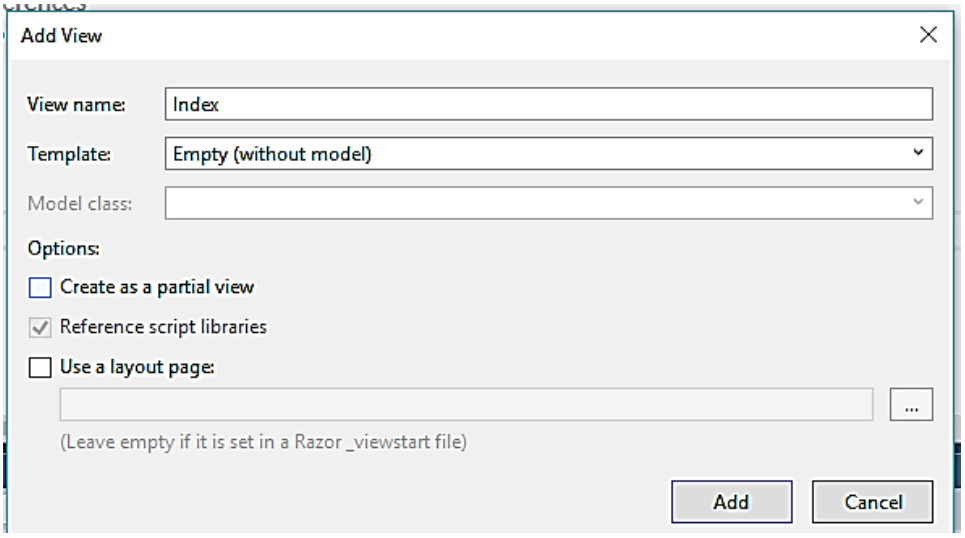
The view 'Index' or its mast...

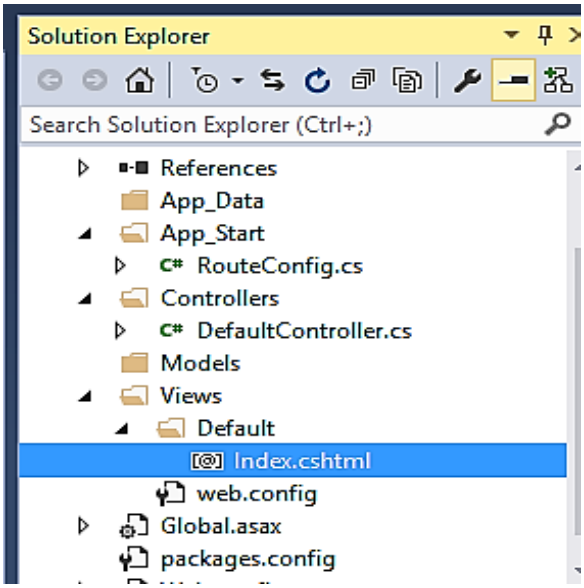
Server Error in '/' Application.

The view 'Index' or its master was not found or no view engine supports the searched locations. The following locations were searched:
~/Views/Default/Index.aspx
~/Views/Default/Index.ascx
~/Views/Shared/Index.aspx
~/Views/Shared/Index.ascx
~/Views/Default/Index.cshtml

The simplest way to create a view is to ask Visual Studio to do it for you. Right-click **anywhere in the definition of the Index action method in code editor window** for the DefaultController.cs file and select Add View from the pop-up menu







The .cshtml file extension denotes a C# view that will be processed by Razor. Early versions of MVC relied on the ASPX view engine, for which view files have the .aspx extension.

The default code of the view:

```
@{
   Layout = null;
}

<!DOCTYPE html>

<html>
<head>
   <meta name="viewport" content="width=device-width" />
   <title>Index</title>
</head>
<body>
   <div>
   </div>
</body>
</html>
```

*try adding and modifying the html of the Index.cshtml page and test the result.

We can return other results from action methods besides strings and ViewResult objects. For example, if we return a **RedirectResult**, the browser will be redirected to another URL. If we return an **HttpUnauthorizedResult**, we force the user to log in. These objects are collectively known as action results, and they are all derived from the **ActionResult** class. The action result system lets us encapsulate and reuse common responses in actions.

Quick examples:

```
public RedirectResult Index()
      {
         return Redirect("https://www.youtube.com");
      }
```

```
public HttpUnauthorizedResult Index()
    {
       return new HttpUnauthorizedResult("unauthorized");
    }
```

Adding Dynamic Output

One way to pass data from the controller to the view is by using the ViewBag object, which is a member of the Controller base class. ViewBag is a dynamic object to which you can assign arbitrary properties, making those values available in whatever view is subsequently rendered.

The DefaultController

```
public ViewResult Index()
    {
       int hour = DateTime.Now.Hour;
       ViewBag.Greeting = hour < 12 ? "good morning!" : "good afternoon!";
       return View();
    }
```

The Index.cshtml page

```
<body>
  <div>
     @ViewBag.Greeting Everyone! I am the Index Page!
  </div>
</body>
</html>
```

Activity: Creating a Simple Data-Entry Application

Setting the Scene

Let us build an events reservation web app with the following pages:

- ✓ A home page that shows information about the event
- ✓ A form that can be used to submit reservation
- ✓ Validation for the reservation form, which will display a thank-you page
- ✓ Reservations e-mailed to the event host when complete

Now:

a. Let us create a new controller for this activity [EventController].
b. Create a view that will be your home page. Add content.

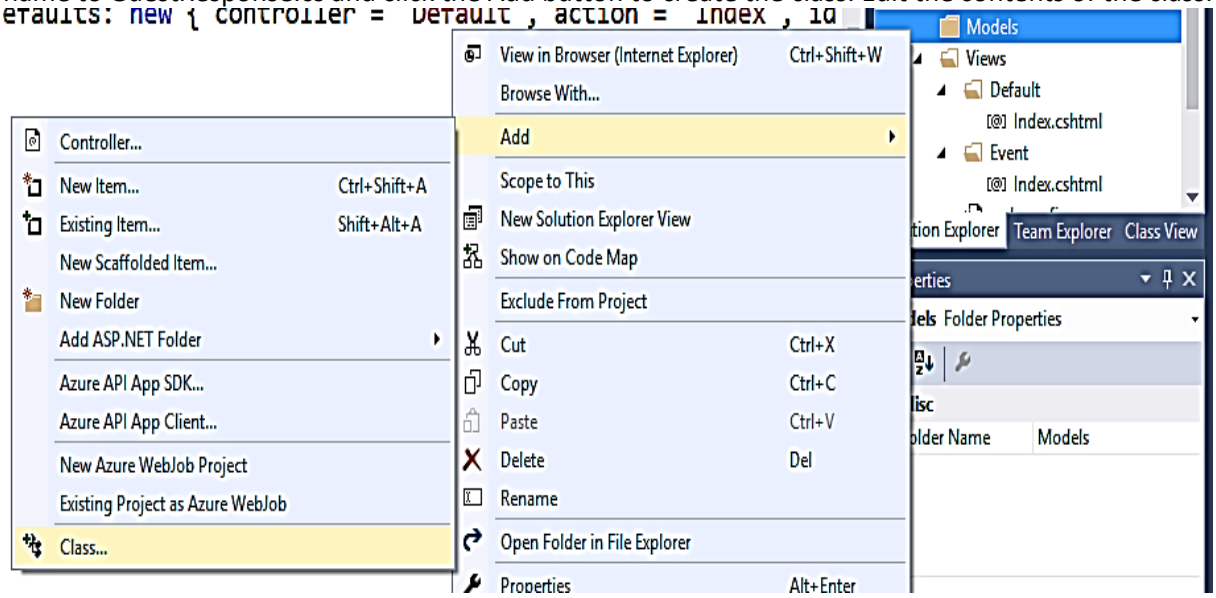
Designing a Data Model

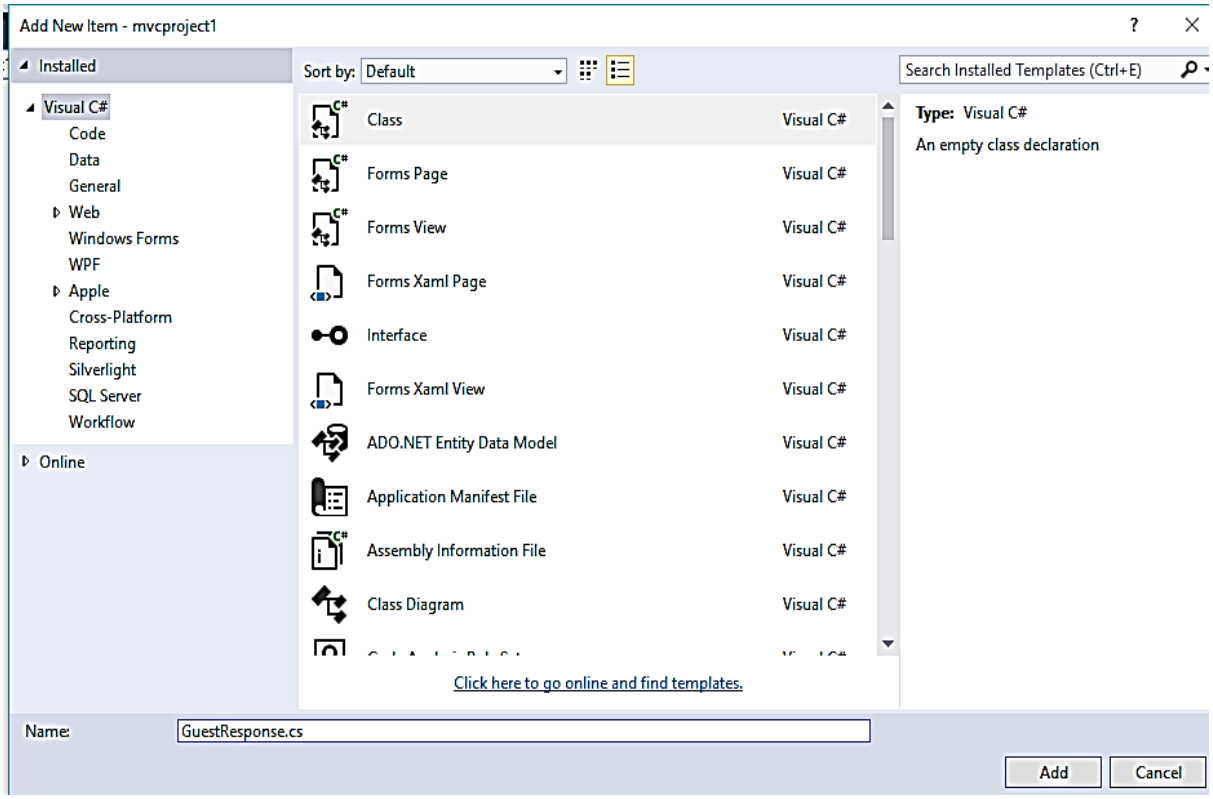
The model is the representation of the real-world objects, processes, and rules that define the subject, known as the domain, of the application. The model, often referred to as a domain model, contains the C# objects (known as domain objects) that make up the universe of the application and the methods that manipulate them. The views and controllers expose the domain to the clients in a consistent manner and a well-designed MVC application starts with a well-designed model, which is then the focal point as controllers and views are added.

Adding a Model Class

The MVC convention is that the classes that make up a model are placed inside the Models folder, which Visual Studio created as part of the initial project setup.

Right-click Models in the Solution Explorer window and select Add followed by Class from the pop-up menus. Set the file name to GuestResponse.cs and click the Add button to create the class. Edit the contents of the class.





GuestResponse.cs

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

namespace mvcproject1.Models
{
   public class GuestResponse
   {
         public string Name { get; set; }
         public string Email { get; set; }
         public string Phone { get; set; }
         public bool? WillAttend { get; set; }
   }
}
```

*the WillAttend property is a nullable bool, which means that it can be true, false, or null.

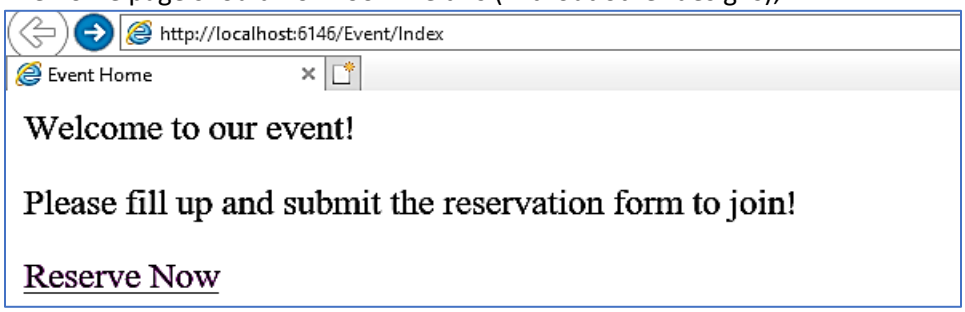
Linking Action Methods

The reservation form page must have a link from the home page. We can easily add an html link to our razor engine powered Index.cshtml like so:

```
@{
   Layout = null;
}

<!DOCTYPE html>
<html>
<head>
   <meta name="viewport" content="width=device-width" />
   <title>Event Home</title>
</head>
<body>
   <div>
      Welcome to our event!<br />
      <p>
         Please fill up and submit the reservation form to join!
      </p>
      @Html.ActionLink("Reserve Now","ReservationForm")
    </div>
</body>
</html>
```

*Html.ActionLink is an HTML helper method. The MVC Framework comes with a collection of built-in helper methods that are convenient for rendering HTML links, text inputs, checkboxes, selections, and other kinds of content. The ActionLink method takes two parameters: the first is the text to display in the link, and the second is the action to perform when the user clicks the link.

The home page should now look like this (without other designs);



Unlike traditional ASP.NET applications, MVC URLs do not correspond to physical files. Each action method has its own URL, and MVC uses the ASP.NET routing system to translate these URLs into actions.

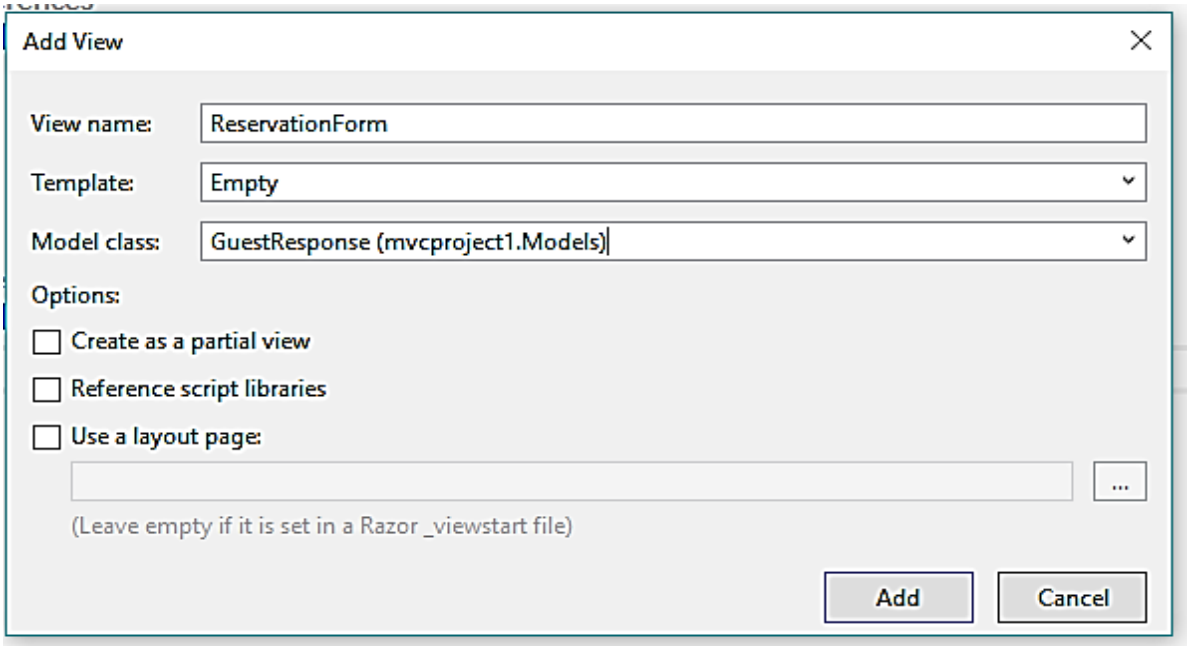
Creating the Action Method

```
…
namespace mvcproject1.Controllers
{
   public class EventController : Controller
   {
      // GET: Event
      public ActionResult Index()
      {
         return View();
      }

      public ActionResult ReservationForm()
      {
         return View();
      }
   }
}
…
```

Adding a Strongly Typed View

Let us add a view for the ReservationForm action method. A strongly typed view is intended to render a specific domain type (a model), and if I specify the type I want to work with (GuestResponse in this case), MVC can create some helpful shortcuts to make it easier.

Right-click the ReservationForm method in the code editor and select Add View from the pop-up menu to open the Add View dialog window. Ensure that the View Name is set as ReservationForm, set Template to Empty and select GuestResponse from the drop-down list for the Model Class field. Leave the View Options boxes unchecked,



It will then create a view with the following initial code. (Observe the first line):

```
@model mvcproject1.Models.GuestResponse
@{
   Layout = null;
}

<!DOCTYPE html>
<html>
<head>
   <meta name="viewport" content="width=device-width" />
   <title>ReservationForm</title>
</head>
<body>
   <div>
   </div>
</body>
</html>
```

Building the Form

We can build out the contents of ReservationForm.cshtml to make it into an HTML form for editing GuestResponse objects:

```
@model mvcproject1.Models.GuestResponse

@{
   Layout = null;
}

<!DOCTYPE html>
<html>
<head>
   <meta name="viewport" content="width=device-width" />
```

```
  <title>ReservationForm</title>
</head>
<body>
  <div>
    @using (Html.BeginForm())
    {
      <p>Your name: @Html.TextBoxFor(x => x.Name) </p>
      <p>Your email: @Html.TextBoxFor(x => x.Email)</p>
      <p>Your phone: @Html.TextBoxFor(x => x.Phone)</p>
      <p>
        Will you attend?
        @Html.DropDownListFor(x => x.WillAttend, new[] {
          new SelectListItem() {Text = "Yes, I'll be there", Value = bool.TrueString},
          new SelectListItem() {Text = "No, I can't come", Value = bool.FalseString}
          }, "Choose an option")
      </p>
      <input type="submit" value="Submit Reservation" />
    }
  </div>
</body>
</html>
```

For each property of the GuestResponse model class, we use an HTML helper method to render a suitable HTML input control. These methods let you select the property that the input element relates to using a lambda expression, like this:

```
...
@Html.TextBoxFor(x => x.Phone)
...
```

The HTML TextBoxFor helper method generates the HTML for an input element, sets the type parameter to text, and sets the id and name attributes to Phone (the name of the selected domain class property) like this:

```
<input id="Phone" name="Phone" type="text" value="" />
```

This handy feature works because the ReservationForm view is strongly typed, and I have told MVC that GuestResponse is the type that I want to render with this view. This provides the HTML helper methods with the information they need to understand which data type I want to read properties from via the @model expression.

If you aren't familiar with C# lambda expressions, an alternative to using lambda expressions is to refer to the name of the model type property as a string, like this:

```
...
@Html.TextBox("Email")
...
```

We find that the lambda expression technique prevents me from mistyping the name of the model type property, because Visual Studio IntelliSense pops up and lets me pick the property automatically.

```
tml.TextBoxFor(x => x.Name) </p>
Html.TextBoxFor(x => x.Email)</p>
Html.TextBoxFor(x => x.)</p>

end?
wnListFor(x => x.WillAt                {
ctListItem() {Text = "Y                there". Value = bool.TrueString},
ctListItem() {Text = "N                }
se an option")
```

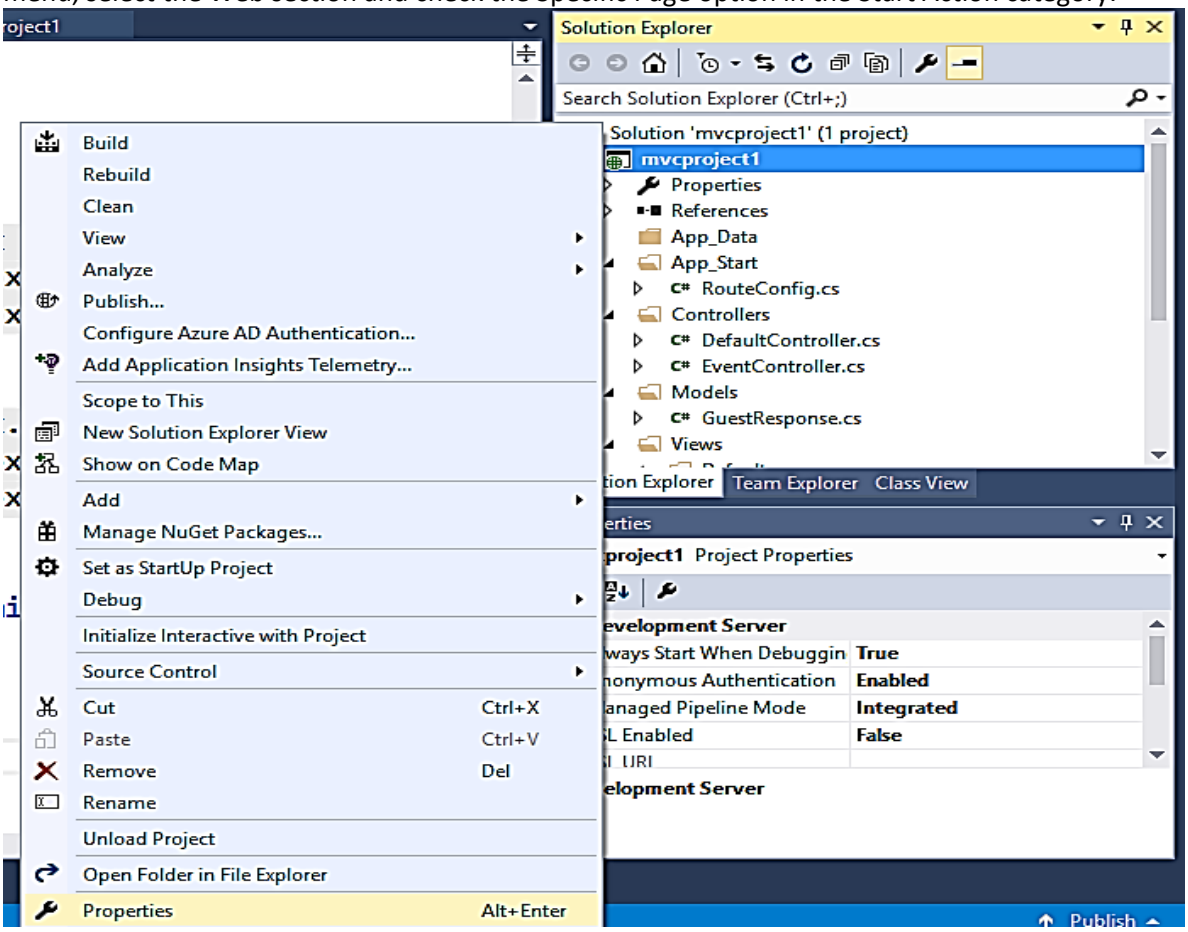
Email
Equals
GetHashCode
GetType
Name
Phone
ToString
WillAttend

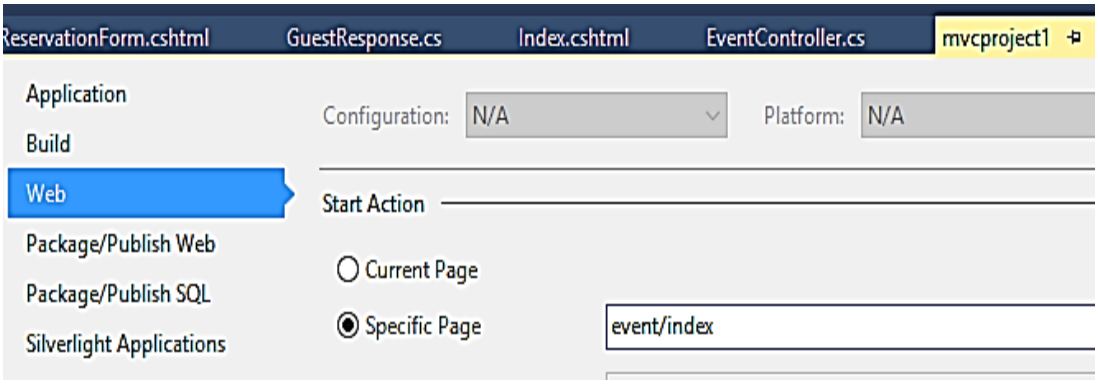
string mvcproject1.Models.GuestResponse.Phone { get; set; }

Setting the Start URL

Visual Studio will, in an effort to be helpful, make the browser request a URL based on the view that is currently being edited. This is a hit-and-miss feature because it doesn't work when you are editing other kinds of file and because you can't just jump in at any point in most complex web apps.

To set a fixed URL for the browser to request, select <yourProjectName> Properties from the Visual Studio Project menu, select the Web section and check the Specific Page option in the Start Action category.





Handling Forms

Clicking the Submit Reservation button just clears any values you have entered into the form. That is because the form posts back to the ReservationForm action method in the Home controller, which just tells MVC to render the view again.

To receive and process submitted form data, I am going to use a clever feature. I will add a second ReservationForm action method in order to create the following:

- A method that responds to HTTP GET requests: A GET request is what a browser issues normally each time someone clicks a link. This version of the action will be responsible for displaying the initial blank form when someone first visits /Home/ReservationForm.
- A method that responds to HTTP POST requests: By default, forms rendered using Html.BeginForm() are submitted by the browser as a POST request. This version of the action will be responsible for receiving submitted data and deciding what to do with it.

Handing GET and POST requests in separate C# methods helps to keep my controller code tidy, since the two methods have different responsibilities. Both action methods are invoked by the same URL, but MVC makes sure that the appropriate method is called, based on whether I am dealing with a GET or POST request.

EventController page

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using mvcproject1.Models;

namespace mvcproject1.Controllers
{
   public class EventController : Controller
   {
      // GET: Event
      public ActionResult Index()
      {
         return View();
      }

      [HttpGet]
      public ActionResult ReservationForm()
```

```
    {
        return View();
    }

    [HttpPost]
    public ActionResult ReservationForm(GuestResponse guestResponse)
    {
        return View("Thanks", guestResponse);
    }
  }
}
```

Rendering Other Views

The **[HttpPost]** overload of the ReservationForm action method also demonstrates how to tell MVC to render a specific view in response to a request, rather than the default view. Here is the relevant statement:

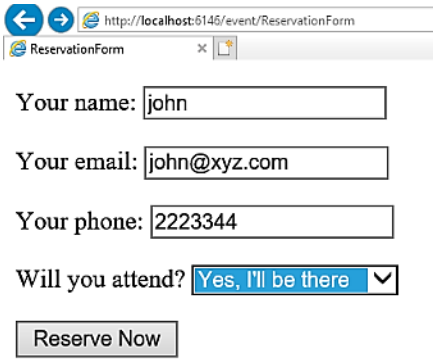
```
...
    [HttpPost]
    public ActionResult ReservationForm(GuestResponse guestResponse)
    {
        return View("Thanks", guestResponse);
    }
...
```

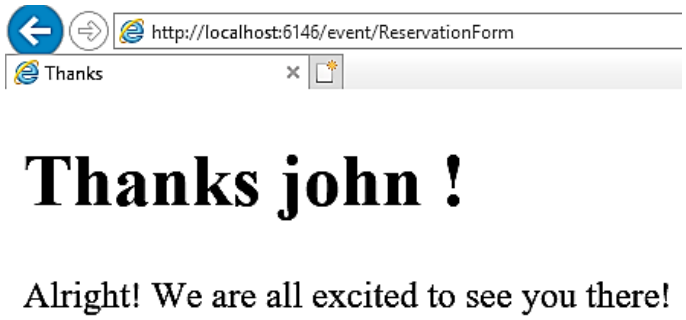
This call to the View method tells MVC to find and render a view called Thanks and to pass the GuestResponse object to the view. To create the view I specified, right-click on any of the EventController methods and select Add View from the pop-up menu and use the Add View dialog to create a strongly typed view called Thanks that uses the GuestResponse model class and that is based on the Empty template. Visual Studio will create the view as Views/Event/Thanks.cshtml.

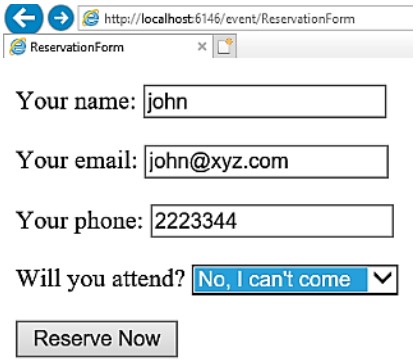
Edit the new view:

```
…
<body>
  <div>
    <h1>Thanks @Model.Name !</h1>
    @if (Model.WillAttend == true)
    {
        @: Alright! We are all excited to see you there!
    }else
    {
        @: Awww, That's too sad. We'll be expecting you next time!
    }
  </div>
</body>
…
```

Test:







Adding Validation

Without validation, users could enter nonsense data or even submit an empty form. In an MVC application, validation is typically applied in the domain model, rather than in the user interface. This means that I am able to define validation criteria in one place and have it take effect anywhere in the application that the model class is used. ASP.NET MVC supports declarative validation rules defined with attributes from the System.ComponentModel.DataAnnotations namespace, meaning that validation constraints are expressed using the standard C# attribute features.

GuestResponse.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace mvcproject1.Models
{
   public class GuestResponse
   {
        [Required(ErrorMessage ="Please enter your name")]
        public string Name { get; set; }

        [Required(ErrorMessage ="Please enter your email address")]
        [RegularExpression(". + \\@. + \\.. +",ErrorMessage ="Please enter a valid email format")]
        public string Email { get; set; }
```

```
        [Required(ErrorMessage ="Please enter your phone number")]
        public string Phone { get; set; }

        [Required(ErrorMessage ="Please specify if you can attend the event")]
        public bool? WillAttend { get; set; }

    }
}
```

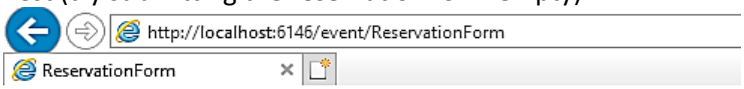
Apply validity conditions on the EventController:

```
[HttpPost]
    public ActionResult ReservationForm(GuestResponse guestResponse)
    {
        if (ModelState.IsValid)
        {
            return View("Thanks", guestResponse);
        }
        else
        {
            //validation error
            return View();
        }

    }
```

Display validation errors summary on the view (ReservationForm.cshtml):

```
…
<body>
    <div>
        @Html.ValidationSummary()
        @using (Html.BeginForm())
        {
            <p>Your name: @Html.TextBoxFor(x => x.Name) </p>
            <p>Your email: @Html.TextBoxFor(x => x.Email)</p>
            <p>Your phone: @Html.TextBoxFor(x => x.Phone)</p>
            <p>
                Will you attend?
                @Html.DropDownListFor(x => x.WillAttend, new[] {
                    new SelectListItem() {Text = "Yes, I'll be there", Value = bool.TrueString},
                    new SelectListItem() {Text = "No, I can't come", Value = bool.FalseString}
                    }, "Choose an option")
            </p>
            <input type="submit" value="Reserve Now" />
        }

    </div>
</body>
…
```

Test (try submitting the reservation form empty)



- Please enter your name
- Please enter your email address
- Please enter your phone number
- Please specify if you can attend the event

Your name:

Your email:

Your phone:

Will you attend? Choose an option

Reserve Now

Highlighting Invalid Fields

The HTML helper methods that create text boxes, drop-downs, and other elements have a handy feature that can be used in conjunction with model binding. The same mechanism that preserves the data that a user entered in a form can also be used to highlight individual fields that failed the validation checks.

When a model class property has failed validation, the HTML helper methods will generate slightly different HTML. As an example, here is the HTML that a call to Html.TextBoxFor(x => x.Name) generates when there is no validation error:

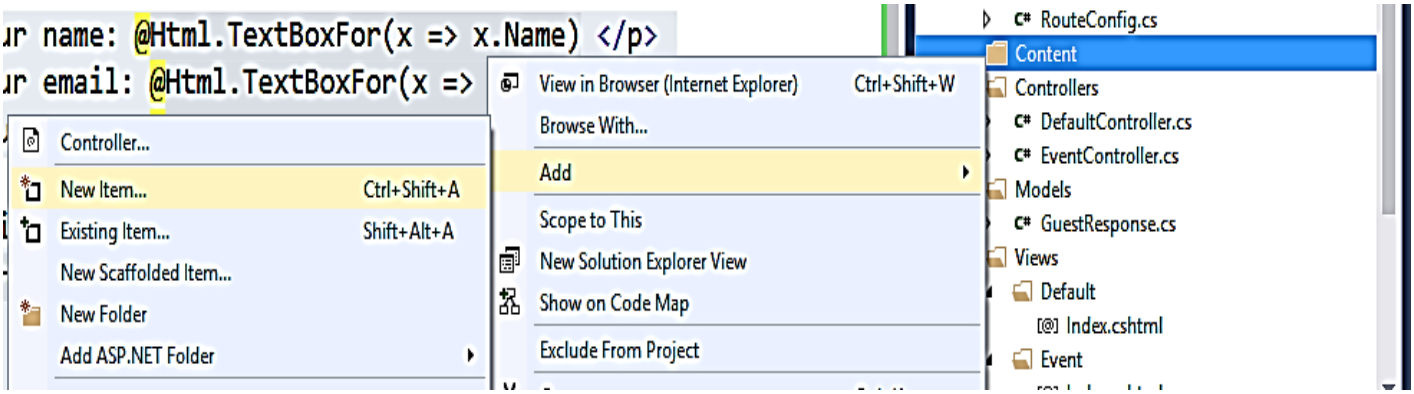
```
<input data-val="true" data-val-required="Please enter your name" id="Name" name="Name" type="text" value="" />
```

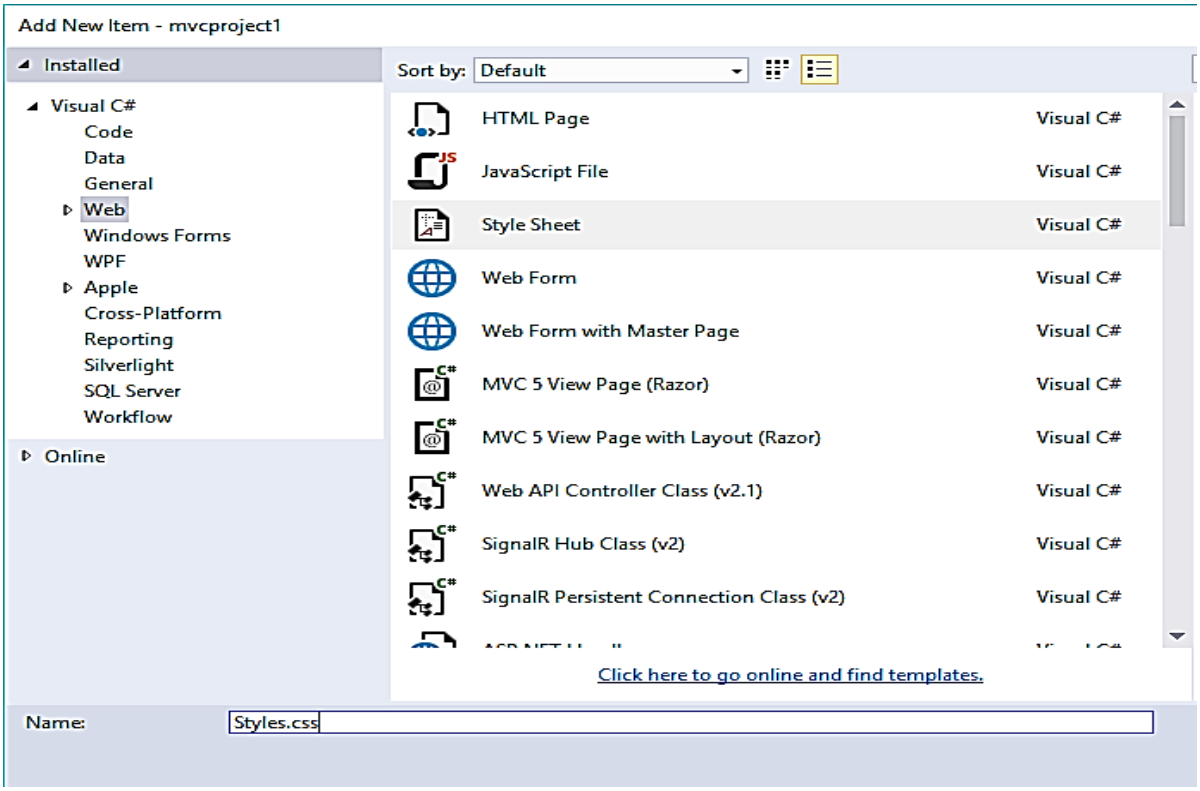
And here is the HTML the same call generates when the user doesn't provide a value (which is a validation error because applied the Required attribute to the Name property in the GuestResponse model class):

```
<input class="input-validation-error" data-val="true" data-val-required="Please enter your name" id="Name"
name="Name" type="text" value="" />
```

The convention in MVC projects is that static content, such as CSS style sheets, is placed into a folder called Content. Create this folder by right-clicking on the mvcproject1 item in the Solution Explorer, selecting Add New Folder from the menu and setting the name to Content.

To create the CSS file, right click on the newly created Content folder, select Add New Item from the menu and choose Style Sheet from the set of item templates. Set the name of the new file to Styles.css.





Add the following rules in the css page:

```
.field-validation-error {color: #f00;}
.field-validation-valid { display: none;}
.input-validation-error { border: 1px solid #f00; background-color: #fee; }
.validation-summary-errors { font-weight: bold; color: #f00;}
.validation-summary-valid { display: none;}
```

Add a reference link on your view (ReservationForm.cshtml) to your css file:

```
<head>
   <meta name="viewport" content="width=device-width" />
   <link rel="stylesheet" type="text/css" href="~/Content/Styles.css" />
   <title>ReservationForm</title>
</head>
```

- **Please enter your name**
- **Please enter your email address**
- **Please enter your phone number**
- **Please specify if you can attend the event**

Your name:

Your email:

Your phone:

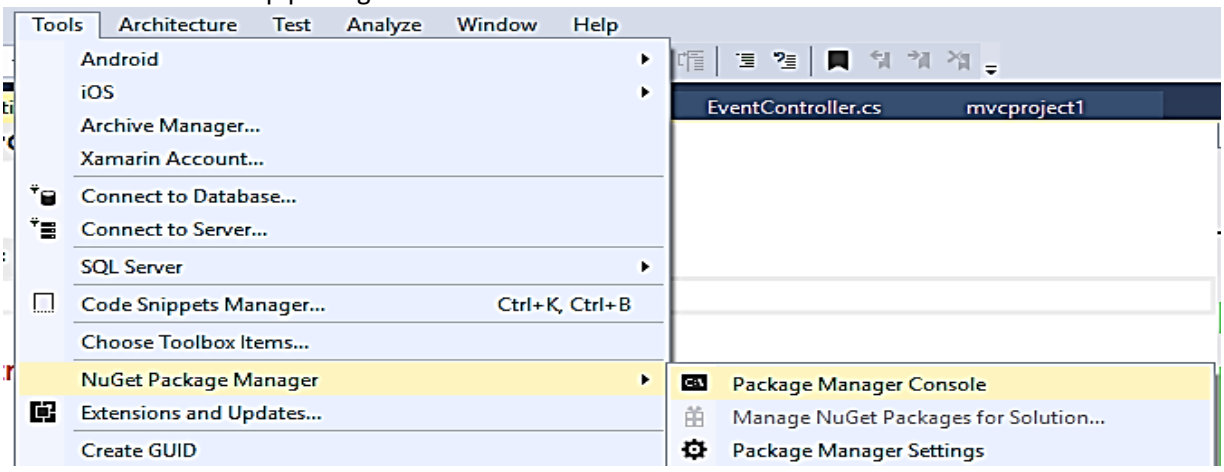
Will you attend? Choose an option

Reserve Now

Styling the Content

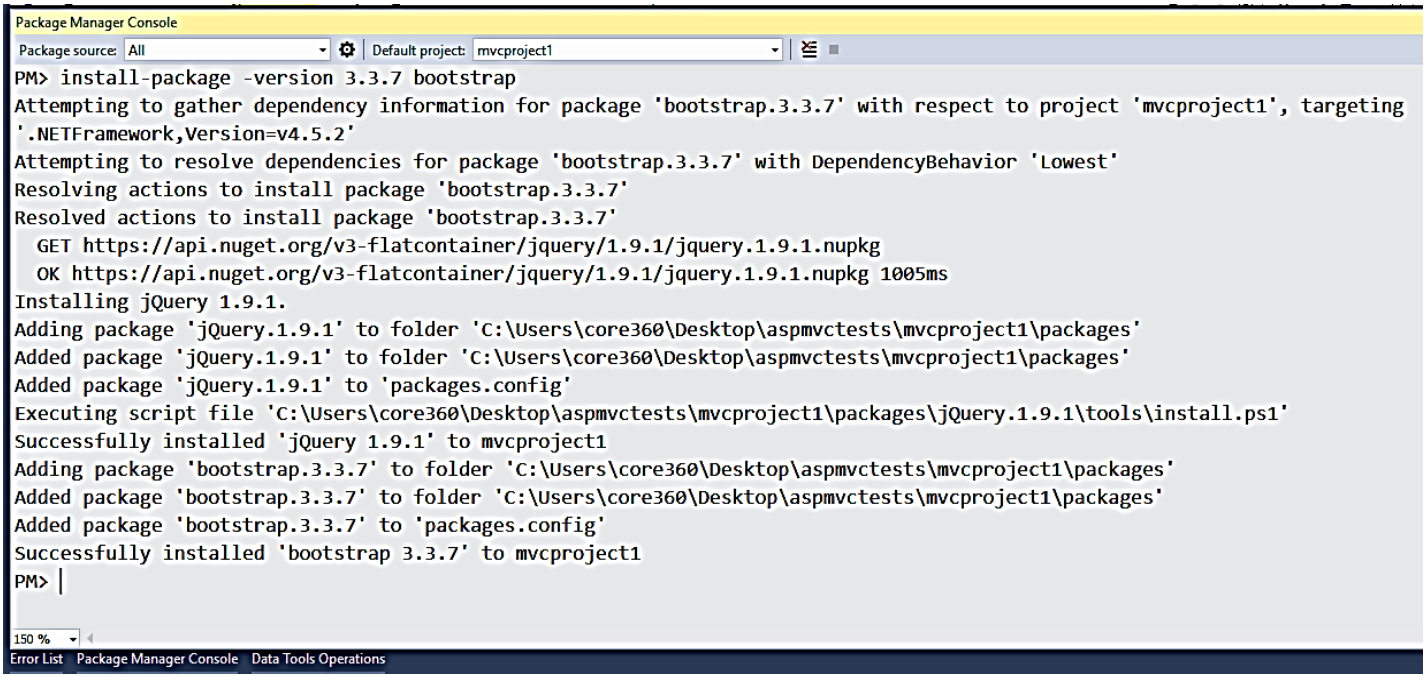
Using NuGet to Install Bootstrap (you might need to launch your VS as administrator)

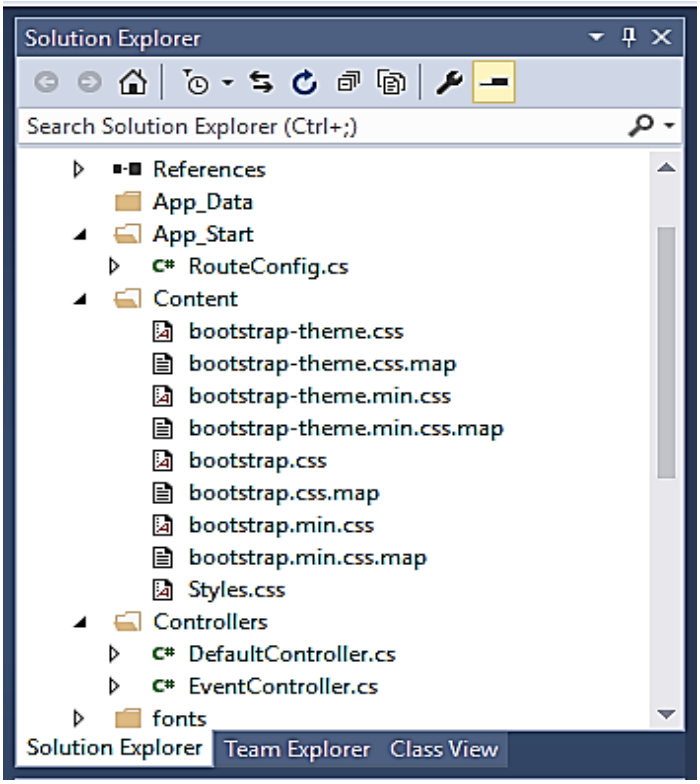
To install the Bootstrap package



Visual Studio will open the NuGet command line. Enter the following command and hit return:

```
Install-Package -version 3.3.7 bootstrap
```

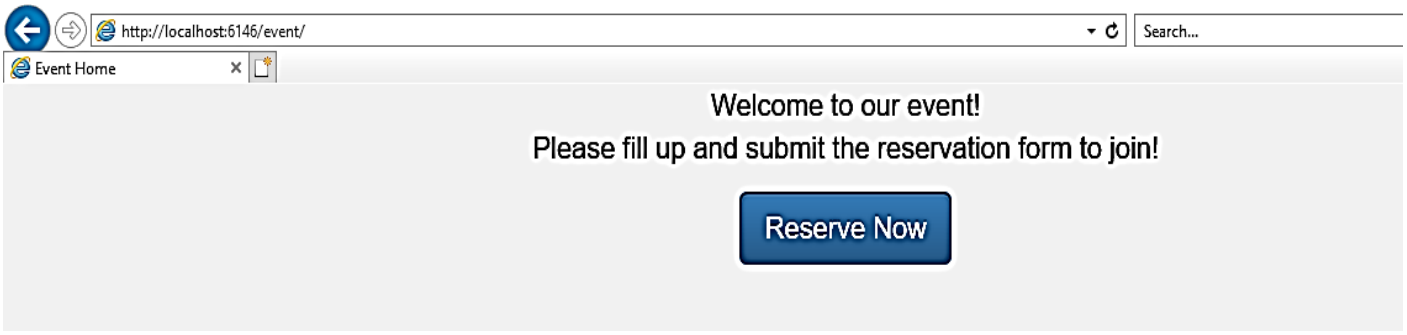




Styling the Index.cshtml

```
@{
   Layout = null;
}

<!DOCTYPE html>
<html>
```

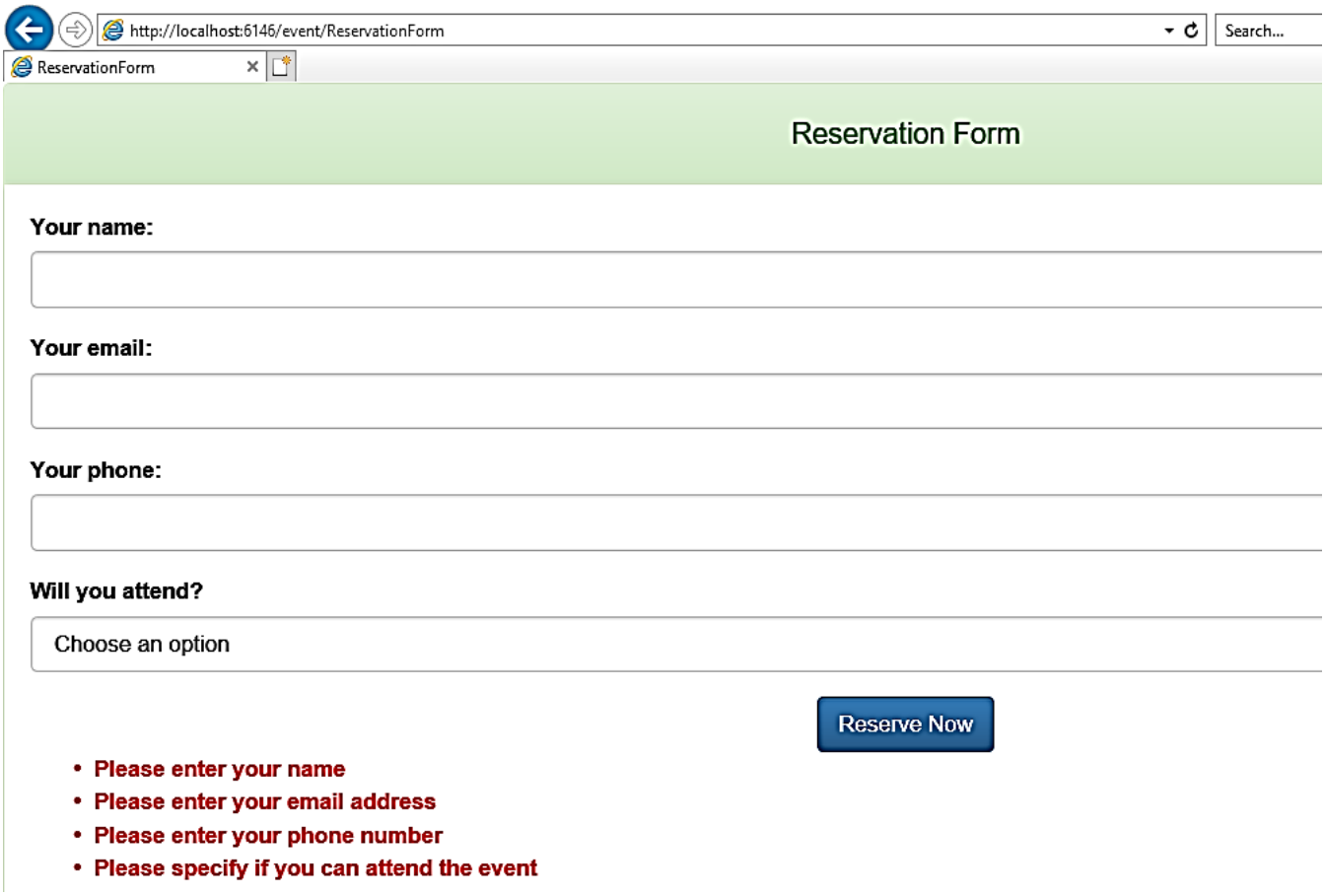
```
<head>
   <meta name="viewport" content="width=device-width" />
   <link href="~/Content/bootstrap.css" rel="stylesheet" />
   <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
   <title>Event Home</title>

   <style>
      .btn a {
         color: white;
         text-decoration: none;
      }

      body {
         background-color: #F1F1F1;
      }
   </style>
</head>
<body>
   <div class="text-center">
      Welcome to our event!<br />
      <p>
         Please fill up and submit the reservation form to join!
      </p>
      <div class="btn btn-primary">
         @Html.ActionLink("Reserve Now", "ReservationForm")
      </div>

    </div>

</body>
</html>
```



Styling the ReservationForm.cshtml

```
@model mvcproject1.Models.GuestResponse

@{
   Layout = null;
}

<!DOCTYPE html>

<html>
<head>
   <meta name="viewport" content="width=device-width" />
   <link href="~/Content/bootstrap.css" rel="stylesheet" />
   <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
   <link href="~/Content/Styles.css"  rel="stylesheet" type="text/css" />
   <title>ReservationForm</title>
</head>
<body>
   <div class="panel panel-success">
      <div class="panel-heading text-center"><h4>Reservation Form</h4></div>
      <div class="panel panel-body">
         @using (Html.BeginForm())
         {
            <div class="form-group">
               <label>Your name: </label>
               @Html.TextBoxFor(x => x.Name, new { @class = "form-control"})
            </div>

            <div class="form-group">
               <label>Your email: </label>
               @Html.TextBoxFor(x => x.Email, new { @class = "form-control" })
            </div>

            <div class="form-group">
               <label>Your phone: </label>
               @Html.TextBoxFor(x => x.Phone, new { @class = "form-control" })
            </div>

            <div class="form-group">
               <label>Will you attend?</label>
               @Html.DropDownListFor(x => x.WillAttend, new[] {
               new SelectListItem() {Text = "Yes, I'll be there", Value = bool.TrueString},
               new SelectListItem() {Text = "No, I can't come", Value = bool.FalseString}
               }, "Choose an option", new { @class = "form-control" })

            </div>

            <div class="text-center">
               <input class="btn btn-primary" type="submit" value="Reserve Now" />
```

```
        </div>
      }
      @Html.ValidationSummary()
    </div>
  </div>
</body>
</html>
```



More ASP.NET Language Features

For the following activities:

a. create a new controller (ex. FeaturesController.cs)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using mvcproject1.Models;

namespace mvcproject1.Controllers
{
  public class FeaturesController : Controller
```

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

b. create a new model (ex. Member.cs)

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

namespace mvcproject1.Models
{
  public class Member
  { }
}
```

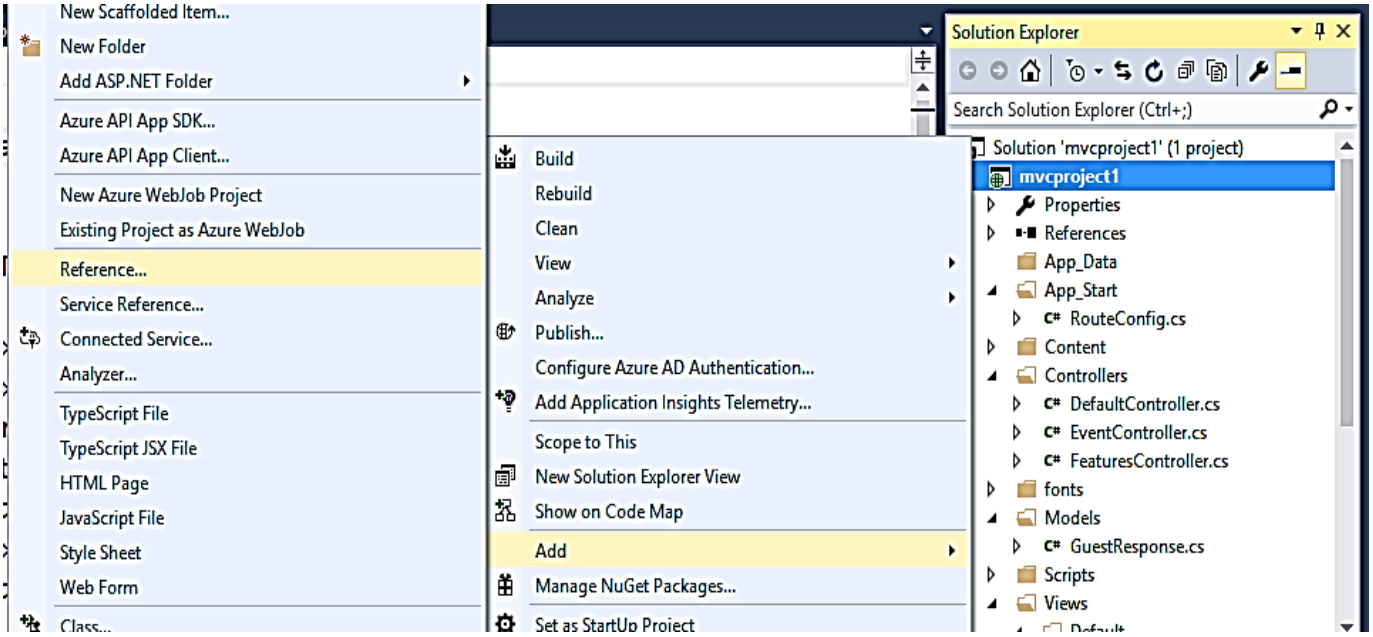
c. create a new view (ex. Output.cshtml)

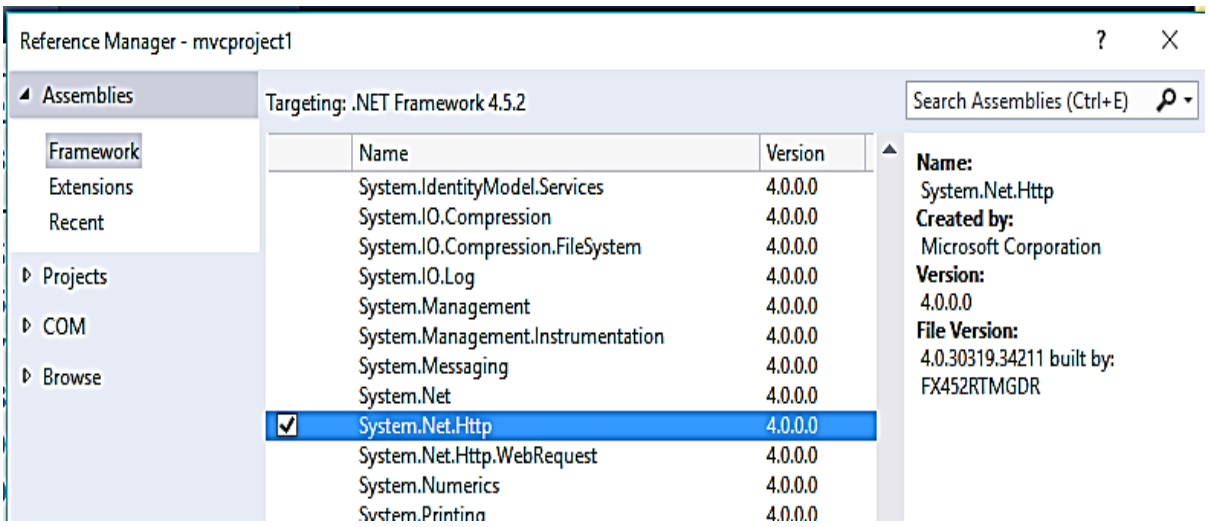
```
@model String

@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Result</title>
</head>
<body>
  <div>
    @Model
  </div>
</body>
</html>
```

Adding the System.Net.Http Assembly

Adding references to other libraries that is not added by default can be done as follows:
Example: Adding the System.Net.Http assembly

Select Add Reference from the Visual Studio Project menu to open the Reference Manager window. Ensure that the Assemblies section is selected on the left-hand side and locate and check the System.Net.Http item.





*we will use this assembly in the succeeding activities.

Using Automatically Implemented Properties

The regular C# property feature lets you expose a piece of data from a class in a way that decouples the data from how it is set and retrieved (aka "setters" and "getters").

For the demonstration, let us use our Model (class) named **Member.cs.**

```
namespace mvcproject1.Models
{
   public class Member
   {   }
}
```

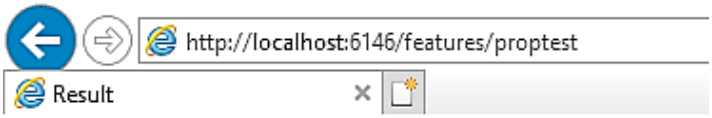
To define properties in the Model class we normally do:

```
namespace mvcproject1.Models
{
   public class Member
   {
      private string name;

      public string Name {
         get { return name; }
         set { name = value; }
      }
   }
}
```

To use (aka "consume") a property in a controller:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using mvcproject1.Models;

namespace mvcproject1.Controllers
{
   public class FeaturesController : Controller
   {
      // GET: Features
      public ActionResult Index()
      {
         return View();
      }

      public ActionResult Proptest()
      {
         //creating a new instance of the model class
         Member member = new Models.Member();

         //setting value to property
         member.Name = "Manny";

         //getting value from property
         string name1 = member.Name;

         //generate view and pass data
         return View("Output", (object)String.Format("Team member 1: {0}", name1));

      }

   }
}
```

Output of the view (Output.cshtml)



Team member 1: Manny

The property getters and setters used in the preceding example is a regular property. The following are examples of automatic inherited properties:

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

namespace mvcproject1.Models
{
   public class Member
   {
      public int MemberID { get; set; }
      public string Name { get; set; }
      public string School { get; set; }
      public int GradeLevel { get; set; }
      public string Category { get; set; }
   }
}
```

Using Object and Collection Initializers

Another tiresome programming task is constructing a new object and then assigning values to the properties. For example (in the FeaturesController.cs):

```
…
public ActionResult CreateMember()
      {
         Member newmember = new Models.Member();
         newmember.MemberID = 1;
         newmember.Name = "Shapi Lezada";
         newmember.School = "Ateneo";
         newmember.GradeLevel = 10;
         newmember.Category = "Swimming";

         return View("Output", (object)String.Format("Name: {0}, School: {1}",newmember.Name,newmember.School));
      }
…
```

You can also use the object initializer feature, which allows me to create and populate the Member instance in a single step as follows

```
…
   public ActionResult CreateMember()
      {
         Member newmember = new Models.Member {
            MemberID = 1, Name = "Shapi Lezada", School = "Ateneo", GradeLevel = 10, Category = "Swimming"
         };


         return View("Output", (object)String.Format("Name: {0}, School: {1}",newmember.Name,newmember.School));
      }
…
```

The same technique can be used to initialize collections and arrays as shown:

```
…
using System.Collections.Generic;
…
…
public ActionResult CreateCollection() {
          string[] stringArray = { "apple", "orange", "plum" };

          List<int> intList = new List<int> { 10, 20, 30, 40 };

          Dictionary<string, int> myDict = new Dictionary<string, int> {
                    { "apple", 10 }, { "orange", 20 }, { "plum", 30 }
          };

          return View("Output", (object)stringArray[1]);
}
…
```

Using Extension Methods

Extension methods are a convenient way of adding methods to classes that you do not own and cannot modify directly.

For this activity:

a. create a Product.cs Model class

```
…
namespace mvcproject1.Models {
  public class Product {
    public int ProductID { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public string Category { set; get; }
  }
}
…
```

b. Create a ShoppingCart class in the Models folder ShoppingCart.cs file and which represents a collection of Product objects.

```
…
using System.Collections.Generic;

namespace mvcproject1.Models {

    public class ShoppingCart {
        public List<Product> Products { get; set; }
    }
}
…
```

This is a simple class that acts as a wrapper around a List of Product objects. If we need to be able to determine the total value of the Product objects in the ShoppingCart class, but we cannot modify the class itself for some reason. I can use an extension method to add the functionality we need.

Create another class (MyExtensionMethods.cs) in the Models folder:

```
…
namespace mvcproject1.Models {

    public static class MyExtensionMethods {
      public static decimal TotalPrices(this ShoppingCart cartParam) {
        decimal total = 0;
        foreach (Product prod in cartParam.Products) {
          total += prod.Price;
        }

        return total;
      }
    }
}
…
```

*The ***this*** keyword in front of the first parameter marks TotalPrices as an extension method. The first parameter tells .NET which class the extension method can be applied to—ShoppingCart in this case. We can refer to the instance of the ShoppingCart that the extension method has been applied to by using the cartParam parameter. Our method enumerates the Products in the ShoppingCart and returns the sum of the Product.Price property.

Then we can use it in our controller:

```
…
public ActionResult UseExtension() {

  // create and populate ShoppingCart
  ShoppingCart cart = new ShoppingCart {
    Products = new List<Product> {
      new Product {Name = "Kayak", Price = 275M},
      new Product {Name = "Lifejacket", Price = 48.95M},
```

```
        new Product {Name = "Soccer ball", Price = 19.50M},
        new Product {Name = "Corner flag", Price = 34.95M}
    }
  };

// get the total value of the products in the cart
decimal cartTotal = cart.TotalPrices();

return View("Output", (object)String.Format("Total: {0:c}", cartTotal));
}
…
```

Creating Filtering Extension Methods

Extension methods can be used to filter collections of objects. An extension method that operates on an IEnumerable<T> and that also returns an IEnumerable<T> can use the ***yield*** keyword to apply selection criteria to items in the source data to produce a reduced set of results.

```
using System.Collections.Generic;

namespace mvcproject1.Models {

public static class MyExtensionMethods
{
  public static decimal TotalPrices(this IEnumerable<Product> productEnum)
  {
      decimal total = 0;
      foreach (Product prod in productEnum) {
        total += prod.Price;
      }
    return total;
  }

  public static IEnumerable<Product> FilterByCategory(this IEnumerable<Product> productEnum, string categoryParam)
  {
    foreach (Product prod in productEnum) {
      if (prod.Category == categoryParam) {
        yield return prod;
      }
    }
  }
}
}
```

This extension method, called FilterByCategory, takes an additional parameter that allows us to inject a filter condition when we call the method. Those Product objects whose Category property matches the parameter are returned in the result IEnumerable<Product> and those that do not match are discarded.

To use the FilterByCategory extension in our Controller:

```
…
public ActionResult UseFilterExtensionMethod()
{

  IEnumerable<Product> products = new ShoppingCart {
    Products = new List<Product> {
      new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
      new Product {Name = "Lifejacket", Category ="Watersports", Price = 48.95M},
      new Product {Name = "Soccer ball", Category ="Soccer", Price = 19.50M},
      new Product {Name = "Corner flag", Category ="Soccer", Price = 34.95M}
    }
  };

  decimal total = 0;
  foreach (Product prod in products.FilterByCategory("Soccer"))
  {
    total += prod.Price;
  }

  return View("Output", (object)String.Format("Total: {0}", total));
}
…
```

When we call the FilterByCategory method on the ShoppingCart, only those Products in the Soccer category are returned. If you start the project and target the UseFilterExtensionMethod action method, you will see the following result, which is the sum of the Soccer product prices:

```
Total: 54.45
```

Using Lambda Expressions

By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator =>, and you put the expression or statement block on the other side. For example, the lambda expression x => x * x specifies a parameter that's named x and returns the value of x squared. You can assign this expression to a delegate type, as the following example shows:

```
delegate int del(int i);
static void Main(string[] args)
{
   del myDelegate = x => x * x;
   int j = myDelegate(5); //j = 25
}
```

We can use a delegate to make our FilterByCategory method more general. That way, the delegate that will be invoked against each Product can filter the objects in any way we choose:

Using a Delegate in an Extension Method in the MyExtensionMethods.cs File

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

namespace mvcproject1.Models
{
  public static class MyExtensionMethods
  {
    public static decimal TotalPrices(this IEnumerable<Product>    productEnum) {
    decimal total = 0;
    foreach (Product prod in productEnum)
    {
      total += prod.Price;
    }
    return total;
  }

  public static IEnumerable<Product> FilterByCategory(this IEnumerable<Product> productEnum, string categoryParam)
  {
    foreach (Product prod in productEnum)
    {
      if (prod.Category == categoryParam)
      {
        yield return prod;
      }
    }
  }

  public static IEnumerable<Product> Filter( this IEnumerable<Product> productEnum, Func<Product, bool>
selectorParam)
  {
    foreach (Product prod in productEnum)
    {
      if (selectorParam(prod))
      {
        yield return prod;
      }
    }
  }
}
}
```

We used a Func as the filtering parameter, which means that we do not need to define the delegate as a type. The delegate takes a Product parameter and returns a bool, which will be true if that Product should be included in the results.

Using the Filtering Extension Method with a Func in the FeaturesController.cs File

```
...
public ActionResult UseFilterExtensionMethod()
{
```

```
  // create and populate ShoppingCart
  IEnumerable<Product> products = new ShoppingCart
  {
    Products = new List<Product>
    {
      new Product {Name = "Kayak", Category = "Watersports", Price =275M},
      new Product {Name = "Lifejacket", Category = "Watersports",Price = 48.95M},
      new Product {Name = "Soccer ball", Category = "Soccer", Price= 19.50M},
      new Product {Name = "Corner flag", Category = "Soccer", Price= 34.95M}
    }
  };

  Func<Product, bool> categoryFilter = delegate(Product prod)
  {
    return prod.Category == "Soccer";
  };

  decimal total = 0;
  foreach (Product prod in products.Filter(categoryFilter))
  {
    total += prod.Price;
  }

  return View("Output", (object)String.Format("Total: {0}", total));
}
…
```

We can now filter the Product objects using any criteria specified in the delegate, but we must define a Func for each kind of filtering that we want, which is not ideal. The less verbose alternative is to use a lambda expression, which is a concise format for expressing a method body in a delegate. We can use it to replace my delegate definition in the action method.

Using a Lambda Expression to Replace a Delegate Definition in the FeaturesController.cs File

```
...
public ActionResult UseFilterExtensionMethod()
{

  // create and populate ShoppingCart
  IEnumerable<Product> products = new ShoppingCart {
    Products = new List<Product> {
      new Product {Name = "Kayak", Category = "Watersports", Price =275M},
      new Product {Name = "Lifejacket", Category = "Watersports",Price = 48.95M},
      new Product {Name = "Soccer ball", Category = "Soccer", Price= 19.50M},
      new Product {Name = "Corner flag", Category = "Soccer", Price= 34.95M}
    }
};

Func<Product, bool> categoryFilter = prod => prod.Category =="Soccer";
```

```
decimal total = 0;

foreach (Product prod in products.Filter(categoryFilter))
{
  total += prod.Price;
}

return View("Output", (object)String.Format("Total: {0}", total));
}
...
```

The parameter is expressed without specifying a type, which will be inferred automatically. The => characters are read aloud as “goes to” and links the parameter to the result of the lambda expression. In our example, a Product parameter called prod goes to a bool result, which will be true if the Category parameter of prod is equal to Soccer. We can make our syntax even tighter by doing away with the Func entirely, as shown:

A Lambda Expression Without a Func in the FeaturesController.cs File

```
...
public ActionResult UseFilterExtensionMethod()
{
  IEnumerable<Product> products = new ShoppingCart
  {
    Products = new List<Product>
    {
    new Product {Name = "Kayak", Category = "Watersports", Price =275M},
    new Product {Name = "Lifejacket", Category = "Watersports",Price = 48.95M},
    new Product {Name = "Soccer ball", Category = "Soccer", Price= 19.50M},
    new Product {Name = "Corner flag", Category = "Soccer", Price= 34.95M}
    }
  };

  decimal total = 0;
  foreach (Product prod in products.Filter (prod => prod.Category =="Soccer"))
  {
    total += prod.Price;
  }

  return View("Output", (object)String.Format("Total: {0}", total));
}
...
```

In this example, we supplied the lambda expression as the parameter to the Filter method. We can combine multiple filters by extending the result part of the lambda expression, as shown below:

Extending the Filtering Expressed by the Lambda Expression in the FeaturesController.cs File

```
...
public ActionResult UseFilterExtensionMethod()
{
  IEnumerable<Product> products = new ShoppingCart
  {
```

```
    Products = new List<Product>
    {
    new Product {Name = "Kayak", Category = "Watersports", Price =275M},
    new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
    new Product {Name = "Soccer ball", Category = "Soccer", Price= 19.50M},
    new Product {Name = "Corner flag", Category = "Soccer", Price= 34.95M}
    }
  };

  decimal total = 0;
  foreach (Product prod in products.Filter(prod => prod.Category == "Soccer" || prod.Price > 20))
  {
    total += prod.Price;
  }

  return View("Output", (object)String.Format("Total: {0}", total));
}
...
```

This revised lambda expression will match Product objects that are in the Soccer category or whose Price property is greater than 20.

Using Automatic Type Inference

The C# var keyword allows you to define a local variable without explicitly specifying the variable type. This is called type inference, or implicit typing.

Using Type Inference

```
..
var myVariable = new Product { Name = "Kayak", Category = "Watersports", Price = 275M };
string name = myVariable.Name; // legal
int count = myVariable.Count; // compiler error
...
```

The type of the variable ***myVariable*** will be inferred from the code. We can test from the statements that follow that the compiler will allow only members of the inferred class—Product in this case—to be called.

Using Anonymous Types

By combining object initializers and type inference, I can create simple data-storage objects without needing to define the corresponding class or struct.

```
...
var myAnonType = new { Name = "MVC", Category = "Pattern" };
...
```

In this example, myAnonType is an anonymously typed object. This does not mean that it is dynamic in the sense that JavaScript variables are dynamic. It just means that the type definition will be created automatically by the compiler. Strong typing is still enforced. You can get and set only the properties that have been defined in the initializer, for example.

The C# compiler generates the class based on the name and type of the parameters in the initializer. Two anonymously typed objects that have the same property names and types will be assigned to the same automatically generated class. This means we can create arrays of anonymously typed objects, as demonstrated below:

Creating an Array of Anonymously Typed Objects in the FeaturesController.cs File

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Web.Mvc;
using mvcproject1.Models;

namespace mvcproject1.Controllers {

public class FeaturesController : Controller {

…
  public ActionResult CreateAnonArray()
  {
    var oddsAndEnds = new[] {
      new { Name = "MVC", Category = "Pattern"},
      new { Name = "Hat", Category = "Clothing"},
      new { Name = "Apple", Category = "Fruit"}
    };
    StringBuilder result = new StringBuilder();

    foreach (var item in oddsAndEnds) {
      result.Append(item.Name).Append("");
    }

    return View("Output", (object)result.ToString());
    }
  }
}
```

Notice that we use var to declare the variable array because we do not have a type to specify. Even though we have not defined a class for any of these objects, we can still enumerate the contents of the array and read the value of the Name property from each of them. This is important, because without this feature, we would not be able to create arrays of anonymously typed objects at all. Or, rather, we could create the arrays, but we would not be able to do anything useful with them.

You will see the following results if you run the example and target the action method:

```
MVC Hat Apple
```

Performing Language Integrated Queries

LINQ is a SQL-like syntax for querying data in classes. Imagine that we have a collection of Product objects, and we want to find the three highest prices and pass them to the View method.

Without LINQ, we would end up with something similar to the example below which shows the FindProducts action method we added to the FeaturesController.

Querying Without LINQ in the FeaturesController.cs File

```
...
public ViewResult FindProducts() {
Product[] products = {
new Product {Name = "Kayak", Category = "Watersports", Price =275M},
new Product {Name = "Lifejacket", Category = "Watersports", Price= 48.95M},
new Product {Name = "Soccer ball", Category = "Soccer", Price =19.50M},
new Product {Name = "Corner flag", Category = "Soccer", Price =34.95M}
};

// define the array to hold the results
Product[] foundProducts = new Product[3];

// sort the contents of the array
Array.Sort(products, (item1, item2) => {
return Comparer<decimal>.Default.Compare(item1.Price,
item2.Price);
});

// get the first three items in the array as the results
Array.Copy(products, foundProducts, 3);

// create the result
StringBuilder result = new StringBuilder();
foreach (Product p in foundProducts)
{
  result.AppendFormat("Price: {0} ", p.Price);
}

return View("Output", (object)result.ToString());
}
...
```

With LINQ, we can significantly simplify the querying process, as demonstrated below.

Using LINQ to Query Data in the FeaturesController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {

public class HomeController : Controller
{
```

```
public string Index() {
  return "Navigate to a URL to show an example";
}

// ...other action methods omitted for brevity...

public ViewResult FindProducts() {
Product[] products = {
new Product {Name = "Kayak", Category = "Watersports", Price= 275M},
new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
new Product {Name = "Soccer ball", Category = "Soccer", Price= 19.50M},
new Product {Name = "Corner flag", Category = "Soccer", Price= 34.95M}
};

var foundProducts = from match in products
                    orderby match.Price descending
                    select new { match.Name, match.Price };

// create the result
int count = 0;
StringBuilder result = new StringBuilder();


foreach (var p in foundProducts)
{
  result.AppendFormat("Price: {0} ", p.Price);
  if (++count == 3)
  {
    break;
  }
}

return View("Output", (object)result.ToString());
}
}
}
```

Using LINQ Dot Notation in the FeaturesController.cs File

```
...
public ViewResult FindProducts() {
Product[] products = {
new Product {Name = "Kayak", Category = "Watersports", Price =275M},
new Product {Name = "Lifejacket", Category = "Watersports", Price= 48.95M},
new Product {Name = "Soccer ball", Category = "Soccer", Price =19.50M},
new Product {Name = "Corner flag", Category = "Soccer", Price =34.95M}
};

var foundProducts = products.OrderByDescending(e => e.Price)
                            .Take(3)
```

```
                                .Select(e => new { e.Name, e.Price });

StringBuilder result = new StringBuilder();

foreach (var p in foundProducts)
{
  result.AppendFormat("Price: {0} ", p.Price);
}

return View("Output", (object)result.ToString());
}
..
```

Some Useful LINQ Extension Methods

Extension Method	Description	Deferred
All	Returns true if all the items in the source data match the predicate	No
Any	Returns true if at least one of the items in the source data matches the predicate	No
Contains	Returns true if the data source contains a specific item or value	No
Count	Returns the number of items in the data source	No
First	Returns the first item from the data source	No
FirstOrDefault	Returns the first item from the data source or the default value if there are no items	No
Last	Returns the last item in the data source	No
LastOrDefault	Returns the last item in the data source or the default value if there are no items	No
Max Min	Returns the largest or smallest value specified by a lambda expression	No
OrderBy		
OrderByDescending	Sorts the source data based on the value returned by the lambda expression	Yes
Reverse	Reverses the order of the items in the data source	Yes
Select	Projects a result from a query	Yes
SelectMany	Projects each data item into a sequence of items and then concatenates all of those resulting sequences into a single sequence	Yes
Single	Returns the first item from the data source or throws an exception if there are multiple matches	No
SingleOrDefault	Returns the first item from the data source or the default value if there are no items, or throws an exception if there are multiple matches	No
Skip		
SkipWhile	Skips over a specified number of elements, or skips while the predicate matches	Yes
Sum	Totals the values selected by the predicate	No
Take	TakeWhile Selects a specified number of elements from the start of the data source or selects items while the predicate matches	Yes
ToArray		
ToDictionary		
ToList	Converts the data source to an array or other collection type	No
Where	Filters items from the data source that do not match the predicate	Yes

Understanding Deferred LINQ Queries

The preceding table includes a Deferred column. There is an interesting variation in the way that the extension methods are executed in a LINQ query. A query that contains only deferred methods is not executed until the items in the result are enumerated, as demonstrated below, which shows a simple change to the FindProducts action method.

Using Deferred LINQ Extension Methods in a Query in the FeaturesController.cs File

```
...
public ActionResult FindProducts() {
Product[] products = {
new Product {Name = "Kayak", Category = "Watersports", Price =275M},
new Product {Name = "Lifejacket", Category = "Watersports", Price= 48.95M},
new Product {Name = "Soccer ball", Category = "Soccer", Price =19.50M},
new Product {Name = "Corner flag", Category = "Soccer", Price =34.95M}
};

var foundProducts = products.OrderByDescending(e => e.Price)
                            .Take(3)
                            .Select(e => new {
                              e.Name,
                               e.Price
                            });

products[2] = new Product { Name = "Stadium", Price = 79600M };
StringBuilder result = new StringBuilder();

foreach (var p in foundProducts)
{
  result.AppendFormat("Price: {0} ", p.Price);
}

return View("Output", (object)result.ToString());
}
...
```

Between defining the LINQ query and enumerating the results, we changed one of the items in the products array. The output from this example is as follows:

```
Price: 79600 Price: 275 Price: 48.95
```

You can see that the query is not evaluated until the results are enumerated, and so the change I made—introducing *Stadium* into the *Product* array—is reflected in the output.

*One interesting feature that arises from deferred LINQ extension methods is that queries are evaluated from scratch every time the results are enumerated, meaning that you can perform the query repeatedly as the source data for the changes and get results that reflect the current state of the source data.

By contrast, using any of the non-deferred extension methods causes a LINQ query to be performed immediately. The example below shows the SumProducts action method we added to the Features controller.

An Immediately Executed LINQ Query in the FeaturesController.cs File

```
...
public ActionResult SumProducts() {
Product[] products = {
new Product {Name = "Kayak", Category = "Watersports", Price =275M},
new Product {Name = "Lifejacket", Category = "Watersports", Price= 48.95M},
```

```
new Product {Name = "Soccer ball", Category = "Soccer", Price =19.50M},
new Product {Name = "Corner flag", Category = "Soccer", Price =34.95M}
};

var results = products.Sum(e => e.Price);
products[2] = new Product { Name = "Stadium", Price = 79500M };
return View("Output", (object)String.Format("Sum: {0:c}", results));
}
..
```

This example uses the Sum method, which is not deferred, and produces the following result:

```
Sum: $378.40
```

You can see that the Stadium item, with its much higher price, has not been included in the results—this is because the results from the Sum method are evaluated as soon as the method is called, rather than being deferred until the results are used.

Using Async Methods

One of the big recent additions to C# is improvements in the way that asynchronous methods are dealt with. Asynchronous methods go off and do work in the background and notify you when they are complete, allowing your code to take care of other business while the background work is performed. Asynchronous methods are an important tool in removing bottlenecks from code and allow applications to take advantage of multiple processors and processor cores to perform work in parallel.

C# and .NET have excellent support for asynchronous methods, but the code tends to be verbose and developers who are not used to parallel programming often get bogged down by the unusual syntax. As an example:

Simple asynchronous method called GetPageLength, defined in a class called MyAsyncMethods.cs and added to the Models folder.

```
using System.Net.Http;
using System.Threading.Tasks;

namespace mvcproject1.Models {

public class MyAsyncMethods {

public static Task<long?> GetPageLength()
{
  HttpClient client = new HttpClient();
  var httpTask = client.GetAsync("https://www.facebook.com");

  // we could do other things here while we are waiting for the HTTP request to complete

  return httpTask.ContinueWith((Task<HttpResponseMessage> antecedent) => {
    return antecedent.Result.Content.Headers.ContentLength;
  });
}
```

```
}
}
```

*This example requires the System.Net.Http assembly.

This is a simple method that uses a System.Net.Http.HttpClient object to request the contents of the Facebook home page and returns its length. We have highlighted the part of the method that tends to cause confusion, which is an example of a task continuation.

.NET represents work that will be done asynchronously as a Task. Task objects are strongly typed based on the result that the background work produces. So, when we call the HttpClient.GetAsync method, what we get back is a Task<HttpResponseMessage>. This tells me that the request will be performed in the background and that the result of the request will be an HttpResponseMessage object.

The part that most programmers get bogged down with is the continuation, which is the mechanism by which you specify what you want to happen when the background task is complete. In the example, We have used the ContinueWith method to process the HttpResponseMessage object we get from the HttpClient.GetAsync method, which we do using a *lambda expression* that returns the value of a property that returns the length of the content we get from the Facebook Web server.

Notice that we use the return keyword twice:

```
...
return httpTask.ContinueWith((Task<HttpResponseMessage> antecedent) => {
  return antecedent.Result.Content.Headers. ContentLength ;
});
...
```

This is the part that makes heads hurt. The first use of the return keyword specifies that we are returning a Task<HttpResponseMessage> object, which, when the task is complete, will return the length of the ContentLength header. The ContentLength header returns a long? result (a nullable long value) and this means that the result of our GetPageLength method is Task<long?>, like this:

```
...
public static Task<long?> GetPageLength() {
...
```

Do not worry if this does not make sense—you are not alone in your confusion. And this is a simple example—complex asynchronous operations can chain large numbers of tasks together using the ContinueWith method, which creates code that can be hard to read and harder to maintain.

Applying the async and await Keywords

Microsoft introduced two keywords to C# that are specifically intended to simplify using asynchronous methods like HttpClient.getAsync. The keywords are async and await and you can see how I have used them to simplify our example method below.

Using the Async and Await Keywords in the MyAsyncMethods.cs File

```
using System.Net.Http;
using System.Threading.Tasks;
```

```
namespace LanguageFeatures.Models {

public class MyAsyncMethods
{
  public async static Task<long?> GetPageLength()
  {
  HttpClient client = new HttpClient();
  var httpMessage = await client.GetAsync("https://www.facebook.com");

  // we could do other things here while we are waiting for the HTTP request to complete
  return httpMessage.Content.Headers.ContentLength;
  }
}
}
```

We used the await keyword when calling the asynchronous method. This tells the C# compiler that we want to wait for the result of the Task that the GetAsync method returns and then carry on executing other statements in the same method.

Applying the await keyword means we can treat the result from the GetAsync method as though it were a regular method and just assign the HttpResponseMessage object that it returns to a variable. And, even better, we can then use the return keyword in the normal way to produce a result from other method—in this case, the value of the ContentLength property. This is a much more natural technique and it means I do not have to worry about the ContinueWith method and multiple uses of the return keyword.

When you use the await keyword, you must also add the async keyword to the method signature, as we have done in the example. The method result type does not change—our example GetPageLength method still returns a Task<long?>. This is because the await and async are implemented using some clever compiler tricks, meaning that they allow a more natural syntax, but they do not change what is happening in the methods to which they are applied. Someone who is calling our GetPageLength method still has to deal with a Task<long?> result because there is still a background operation that produces a nullable long—although, of course, that programmer can also choose to use the await and async keywords as well.

Using Razor Expressions

A view engine processes ASP.NET content and looks for instructions, typically to insert dynamic content into the output sent to a browser and Razor is the name of the MVC Framework view engine.

Preparing the Example Project

We create a **new** Visual Studio project called **Razor** using the **ASP.NET MVC Web Application template**. Then select the Empty option and check the box to get the initial configuration for an MVC project.

Defining the Model

We am going to start with a model called Product.cs, defined in a class file called Product.cs added to the Models folder.

```
namespace Razor.Models
{
public class Product {
  public int ProductID { get; set; }
  public string Name { get; set; }
  public string Description { get; set; }
```

```
  public decimal Price { get; set; }
  public string Category { set; get; }
}
}
```

Defining the Controller

Right-click the Controllers folder in the Solution Explorer, select Add Controller, select MVC 5 Controller–Empty, click Add and set the name to HomeController. When you click the second Add button, Visual Studio will create the Controllers/HomeController.cs file.

```
using System.Web.Mvc;
using Razor.Models;

namespace Razor.Controllers
{

public class HomeController : Controller
{

  Product myProduct = new Product
  {
    ProductID = 1, Name = "Kayak", Description = "A boat for one person", Category = "Watersports", Price = 275M
  };

public ActionResult Index() {
  return View(myProduct);
}

}
}
```

Creating the View

Right-click on the Index method in the HomeController class and select Add View from the pop-up menu. Ensure that the name of the view is Index, change the Template to Empty and select Product for the model class. Uncheck the View Option boxes and click Add to create the Index.cshtml file in the Views/Home folder.

```
@model Razor.Models.Product
@{
Layout = null;
}
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>Index</title>
</head>
<body>
  <div>
  </div>
</body>
```

```
</html>
```

Working with the Model Object

Let us start with the first line in the view:

```
...
@model Razor.Models.Product
...
```

Razor statements start with the @ character. In this case, the @model statement declares the type of the model object that we will pass to the view from the action method. This allows us to refer to the methods, fields, and properties of the view model object through @Model.

```
@model Razor.Models.Product
@{
Layout = null;
}
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>Index</title>
</head>
<body>
<div>
@Model.Name
</div>
</body>
</html>
```

Note Notice that I declare the view model object type using @model (lower case m) and access the Name property using @Model (upper case M). This is slightly confusing when you start working with Razor.

Working with Layouts

The other Razor expression in the Index.cshtml view file is this one:

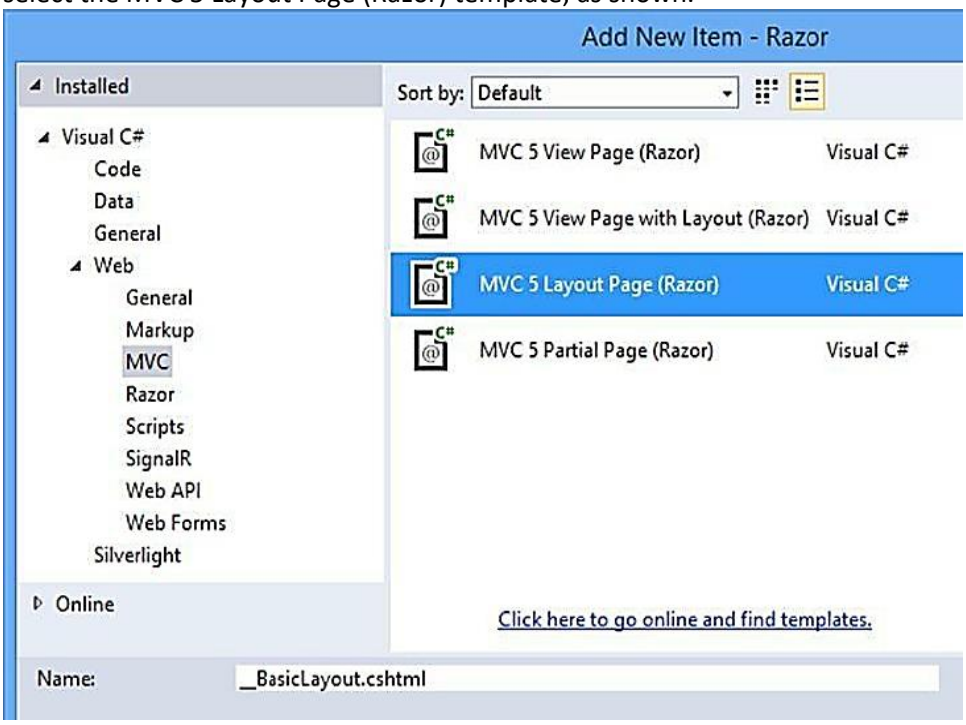
```
...
@{
Layout = null;
}
...
```

This is an example of a Razor code block, which allows me to include C# statements in a view. The code block is opened with @{ and closed with } and the statements it contains are evaluated when the view is rendered. This code block sets the value of the Layout property to null.

The effect of setting the Layout property to null is to tell the MVC framework that the view is self-contained and will render all of the content required for the client. Layouts are effectively templates that contain markup that you use to create consistency across your app—this could be to ensure that the right JavaScript libraries are included in the result or that a common look and feel is used throughout.

Creating the Layout

To create a layout, right-click on the Views folder in the Solution Explorer, click Add New Item from the Add menu and select the MVC 5 Layout Page (Razor) template, as shown.



Files in the Views folder whose names begin with an underscore (_) are not returned to the user, which allows the file name to differentiate between views that you want to render and the files that support them. Layouts, which are support files, are prefixed with an underscore.

The Initial Contents of the _BasicLayout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title> @ViewBag.Title</title>
</head>
<body>
<div>
@RenderBody()
</div>
</body>
</html>
```

Layouts are a specialized form of view and I have highlighted the @ expressions in the listing. The call to the @RenderBody method inserts the contents of the view specified by the action method into the layout markup. The other Razor expression in the layout looks for a property called Title in the ViewBag in order to set the contents of the title element.

The elements in the layout will be applied to any view that uses the layout and this is why layouts are essentially templates.

Adding Elements to the _BasicLayout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>@ViewBag.Title</title>
</head>
<body>
<h1>Product Information</h1>
<div style="padding: 20px; border: solid medium black; font-size:20pt">
@RenderBody()
</div>
<h2>Visit <a href="https://www.facebook.com">Facebook</a></h2>
</body>
</html>
```

We have added a couple of header elements and applied some CSS styles to the div element which contains the @RenderBody expression, just to make it clear what content comes from the layout and what comes from the view.

Applying a Layout

To apply the layout to the view, we just need to set the value of the Layout property. The layout contains the HTML elements that define the structure of the response to the browser, so we can also remove those elements from the view.

Using the Layout Property in the Index.cshtml File

```
@model Razor.Models.Product
@{
ViewBag.Title = "Product Name";
Layout = "~/Views/_BasicLayout.cshtml";
}
Product Name: @Model.Name
```

Using a View Start File

We have to specify the layout file we want in every view. That means that if we need to rename the layout file, we are going to have to find every view that refers to it and make a change, which will be an error-prone process and counter to the general theme of easy maintenance that runs through the MVC framework.

We can resolve this by using a view start file. When it renders a view, the MVC framework will look for a file called _ViewStart.cshtml. The contents of this file will be treated as though they were contained in the view file itself and we can use this feature to automatically set a value for the Layout property.

To create a view start file, add a new layout file to the Views folder and set the name of the file to _ViewStart.cshtml.

The contents of the _ViewStart.cshtml File

```
@{
Layout = "~/Views/_BasicLayout.cshtml";
}
```

Our view start file contains a value for the Layout property, which means that we can remove the corresponding expression from the Index.cshtml file, as shown below.

Updating the Index.cshtml File to Reflect the Use of a View Start File

```
@model Razor.Models.Product
@{
ViewBag.Title = "Product Name";
}
Product Name: @Model.Name
```

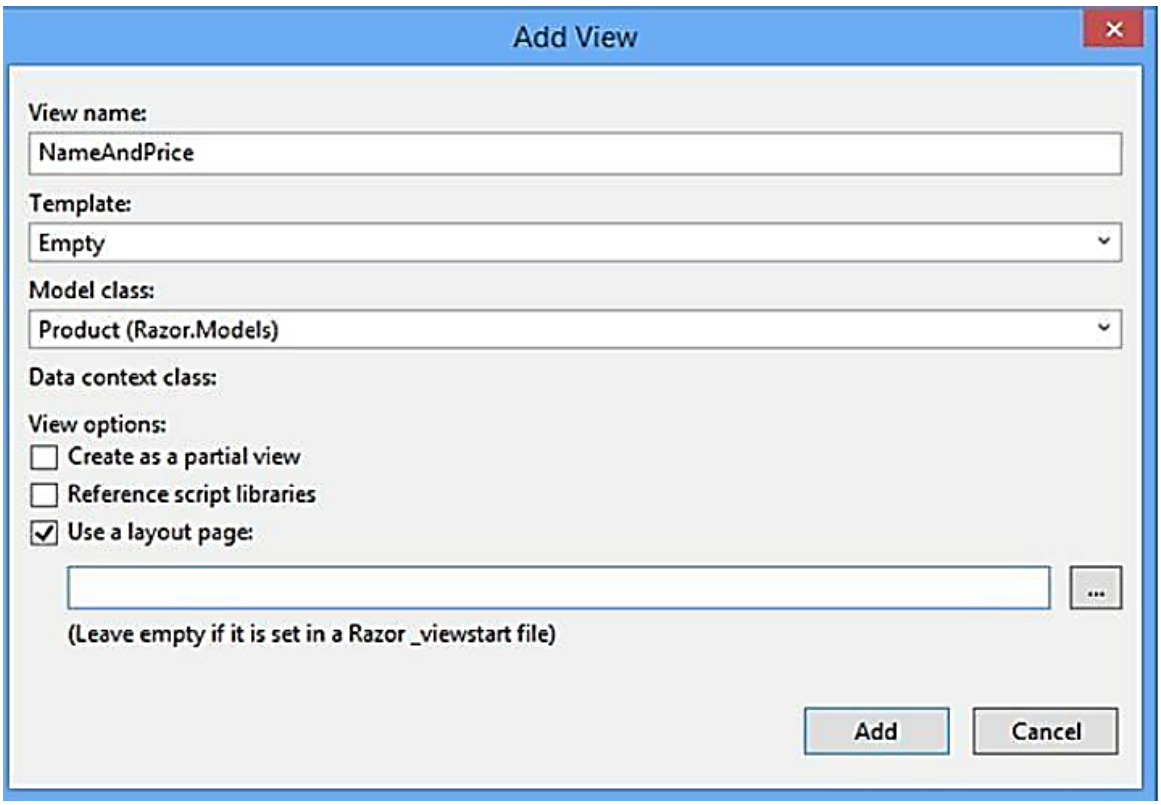
We do not have to specify the use of the view start file. The MVC framework will locate the file and use its contents automatically. The values defined in the view file take precedence, which makes it easy to override the view start file.

*It is important to understand the difference between omitting the Layout property from the view file and setting it to null. If your view is self-contained and you do not want to use a layout, then set the Layout property to null. If you omit the Layout property, then the MVC framework will assume that you do want a layout and that it should use the value it finds in the view start file.

To continue with our activity, add an action to our HomeController.cs

```
public ActionResult NameAndPrice() {
  return View(myProduct);
}
```

Then create a view (NameAndPrice.cshtml). Note that we can also easily set a layout page every time we create a new view like so:



```
@model Razor.Models.Product
@{
  ViewBag.Title = "NameAndPrice";
  Layout = "~/Views/_BasicLayout.cshtml";
}

<h2>NameAndPrice</h2>
The product name is @Model.Name and it costs $@Model.Price
```

Inserting Data Values

The simplest thing you can do with a Razor expression is to insert a data value into the markup. You can do this using the @Model expression to refer to properties and methods defined by the view model object or use the @ViewBag expression to refer to properties you have defined dynamically using the view bag feature.

Add an action method to our HomeController.cs

```
…
public ActionResult DemoExpression() {
  ViewBag.ProductCount = 1;
  ViewBag.ExpressShip = true;
  ViewBag.ApplyDiscount = false;
  ViewBag.Supplier = null;
  return View(myProduct);
}
…
```

The Contents of the DemoExpression.cshtml File

```
@model Razor.Models.Product

@{
  ViewBag.Title = "DemoExpression";
}
<table>
<thead>
<tr><th>Property</th><th>Value</th></tr>
</thead>
<tbody>
<tr><td>Name</td><td>@Model.Name</td></tr>
<tr><td>Price</td><td>@Model.Price</td></tr>
<tr><td>Stock Level</td><td>@ViewBag.ProductCount</td></tr>
</tbody>
</table>
```

Setting Attribute Values

You can also use Razor expressions to set the value of element attributes.

Using a Razor Expression to Set an Attribute Value in the DemoExpression.cshtml File

```
@model Razor.Models.Product
@{
  ViewBag.Title = "DemoExpression";
```

```
  Layout = "~/Views/_BasicLayout.cshtml";
}
<table>
<thead>
<tr><th>Property</th><th>Value</th></tr>
</thead>
<tbody>
<tr><td>Name</td><td>@Model.Name</td></tr>
<tr><td>Price</td><td>@Model.Price</td></tr>
<tr><td>Stock Level</td><td>@ViewBag.ProductCount</td></tr>
</tbody>
</table>
<div data-discount="@ViewBag.ApplyDiscount" dataexpress="@
ViewBag.ExpressShip"
data-supplier="@ViewBag.Supplier">
The containing element has data attributes
</div>
Discount:<input type="checkbox" checked="@ViewBag.ApplyDiscount" />
Express:<input type="checkbox" checked="@ViewBag.ExpressShip" />
Supplier:<input type="checkbox" checked="@ViewBag.Supplier" />
```

Using Conditional Statements

Razor is able to process conditional statements, which means that we can tailor the output from a view based on values in the view data. This kind of technique is at the heart of Razor, and allows you to create complex and fluid layouts that are still reasonably simple to read and maintain.

Using a Conditional Razor Statement in the DemoExpression.cshtml File

```
@model Razor.Models.Product
@{
ViewBag.Title = "DemoExpression";
Layout = "~/Views/_BasicLayout.cshtml";
}
<table>
<thead>
<tr><th>Property</th><th>Value</th></tr>
</thead>
<tbody>
<tr><td>Name</td><td>@Model.Name</td></tr>
<tr><td>Price</td><td>@Model.Price</td></tr>
<tr>
<td>Stock Level</td>
<td>
@switch ((int)ViewBag.ProductCount)
{
case 0:
  @: Out of Stock
  break;
case 1:
  <b>Low Stock (@ViewBag.ProductCount)</b>
  break;
```

```
default:
  @ViewBag.ProductCount
  break;
}
</td>
</tr>
</tbody>
</table>
```

To start a conditional statement, you place an @ character in front of the C# conditional keyword, which is switch in this example. You terminate the code block with a close brace character (}) just as you would with a regular C# code block.

*Notice that I had to cast the value of the ViewBag.ProductCount property to an int in order to use it with a switch statement. This is required because the Razor switch expression cannot evaluate a dynamic property–you must cast to a specific type so that it knows how to perform comparisons.

Inside of the Razor code block, you can include HTML elements and data values into the view output just by defining the HTML and Razor expressions, like this:

```
...
<b>Low Stock (@ViewBag.ProductCount)</b>
...
Or like this:
...
@ViewBag.ProductCount
...
```

We do not have to put the elements or expressions in quotes or denote them in any special way—the Razor engine will interpret these as output to be processed. However, if you want to insert literal text into the view when it is not contained in an HTML element, then you need to give Razor a helping hand and prefix the line like this:

```
...
@: Out of Stock
...
```

The @: characters prevent Razor from interpreting this as a C# statement, which is the default behavior when it encounters text.

Using an if Statement in a Razor View in the DemoExpression.cshtml File

```
@model Razor.Models.Product

@{
ViewBag.Title = "DemoExpression";
Layout = "~/Views/_BasicLayout.cshtml";
}

<table>
<thead>
<tr><th>Property</th><th>Value</th></tr>
</thead>
<tbody>
```

```
<tr><td>Name</td><td>@Model.Name</td></tr>
<tr><td>Price</td><td>@Model.Price</td></tr>
<tr>
<td>Stock Level</td>
<td>
@if (ViewBag.ProductCount == 0)
{
  @:Out of Stock
}
else if (ViewBag.ProductCount == 1)
{
  <b>Low Stock (@ViewBag.ProductCount)</b>
}
else
{
  @ViewBag.ProductCount
}
</td>
</tr>
</tbody>
</table>
```

Enumerating Arrays and Collections

When writing an MVC application, you will often want to enumerate the contents of an array or some other kind of collection of objects and generate content that details each one. To demonstrate how this is done, let's defined a new action method in the Home controller called DemoArray:

The DemoArray Action Method in the HomeController.cs File

```
using System.Web.Mvc;
using Razor.Models;

namespace Razor.Controllers
{
public class HomeController : Controller
{
  Product myProduct = new Product
  {
    ProductID = 1, Name = "Kayak", Description = "A boat for one person", Category = "Watersports", Price = 275M
  };

// ...other action methods omitted for brevity...
public ActionResult DemoArray() {
  Product[] array = {
  new Product {Name = "Kayak", Price = 275M},
  new Product {Name = "Lifejacket", Price = 48.95M},
  new Product {Name = "Soccer ball", Price = 19.50M},
  new Product {Name = "Corner flag", Price = 34.95M}
  };

  return View(array);
```

```
}
}
}
```

This action method creates a Product[] object that contains simple data values and passes them to the View method so that the data is rendered using the default view. The Visual Studio scaffold feature won't let you specify an array as a model type. To create a view for an action method that passes an array, the best approach is to create a view without a model and then manually add the @model expression after the file has been created.

The Contents of the DemoArray.cshtml File

```
@model Razor.Models.Product[]
@{
  ViewBag.Title = "DemoArray";
  Layout = "~/Views/_BasicLayout.cshtml";
}

@if (Model.Length > 0) {
  <table>
  <thead><tr><th>Product</th><th>Price</th></tr></thead>
  <tbody>
  @foreach (Razor.Models.Product p in Model)
  {
    <tr>
    <td>@p.Name</td>
    <td>$@p.Price</td>
    </tr>
  }
  </tbody>
  </table>
}
else
{
  <h2>No product data</h2>
}
```

We use an @if statement to vary the content based on the length of the array that we receive as the view model and a @foreach expression to enumerate the contents of the array and generate a row in an HTML table for each of them. You can see how these expressions match their C# counterpart. You can also see how we created a local variable called p in the foreach loop and then referred to its properties using the Razor expressions @p.Name and @p.Price.

The result is that we generate an h2 element if the array is empty and produce one row per array item in an HTML table otherwise. Because my data is static in this example, you will always see the same result,

Dealing with Namespaces

Notice that we had to refer to the Product class by its fully qualified name in the foreach loop in the last example, like this:

```
...
@foreach (Razor.Models.Product p in Model) {
...
```

This can become annoying in a complex view, where you will have many references to view model and other classes. We can tidy up our views by applying the @using expression to bring a namespace into context for a view, just like we would in a regular C# class.

Applying the @using Expression in the DemoArray.cshtml File

```
@using Razor.Models
@model Product[]
@{
  ViewBag.Title = "DemoArray";
  Layout = "~/Views/_BasicLayout.cshtml";
}

@if (Model.Length > 0)
{
  <table>
  <thead><tr><th>Product</th><th>Price</th></tr></thead>
  <tbody>
  @foreach (Product p in Model)
  {
    <tr>
    <td>@p.Name</td>
    <td>$@p.Price</td>
    </tr>
  }

  </tbody>
  </table>
}
else
{
  <h2>No product data</h2>
}
```

A view can contain multiple @using expressions. We have used the @using expression to import the Razor.Models namespace, which means that we can remove the namespace from the @model expression and from within the foreach loop.

More Tools for the MVC Project

Preparing the Example Project

a. create a simple example project, called EssentialTools (**ASP.NET MVC Web Application template**, selected the Empty option and checked the box to add the basic MVC project content).

Creating the Model Classes

Add a class file to the Models project folder called Product.cs

```
…
namespace EssentialTools.Models
{
```

```
public class Product
{
  public int ProductID { get; set; }
  public string Name { get; set; }
  public string Description { get; set; }
  public decimal Price { get; set; }
  public string Category { set; get; }
}
}
```

Also add a class that will calculate the total price of a collection of Product objects. Let's add a new class file to the Models folder called LinqValueCalculator.

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

namespace EssentialTools.Models
{
  public class LinqValueCalculator
  {
    public decimal ValueProducts(IEnumerable<Product> products)
    {
      return products.Sum(p => p.Price);
    }
  }
}
```

The LinqValueCalculator class defines a single method called ValueProducts, which uses the LINQ Sum method to add together the value of the Price property of each Product object in an enumerable passed to the method.

Our final model class is ShoppingCart and it represents a collection of Product objects and uses a LinqValueCalculator to determine the total value.

The contents of the ShoppingCart.cs File

```
…
using System.Collections.Generic;

namespace EssentialTools.Models
{
  public class ShoppingCart
  {
    private LinqValueCalculator calc;

    public ShoppingCart(LinqValueCalculator calcParam)
    {
      calc = calcParam;
     }

     public IEnumerable<Product> Products { get; set; }
```

```
    public decimal CalculateProductTotal()
    {
      return calc.ValueProducts(Products);
    }
  }
}
```

Adding the Controller

Added a new controller to the Controllers folder called HomeController. The Index action method creates an array of Product objects and uses a LinqValueCalculator object to produce the total value, which we then pass to the View method. We do not specify a view when we call the View method, so the MVC Framework will select the default view associated with the action method (the Views/Home/Index.cshtml file).

The Contents of the HomeController.cs File

```
using System.Web.Mvc;
using System.Linq;
using EssentialTools.Models;

namespace EssentialTools.Controllers
{
  public class HomeController : Controller
  {
    private Product[] products =
    {
    new Product {Name = "Kayak", Category = "Watersports", Price =275M},
    new Product {Name = "Lifejacket", Category = "Watersports",Price = 48.95M},
    new Product {Name = "Soccer ball", Category = "Soccer", Price= 19.50M},
    new Product {Name = "Corner flag", Category = "Soccer", Price= 34.95M}
    };

    public ActionResult Index()
    {
      LinqValueCalculator calc = new LinqValueCalculator();
      ShoppingCart cart = new ShoppingCart(calc) { Products=products };
      decimal totalValue = cart.CalculateProductTotal();
      return View(totalValue);
    }
  }
}
```

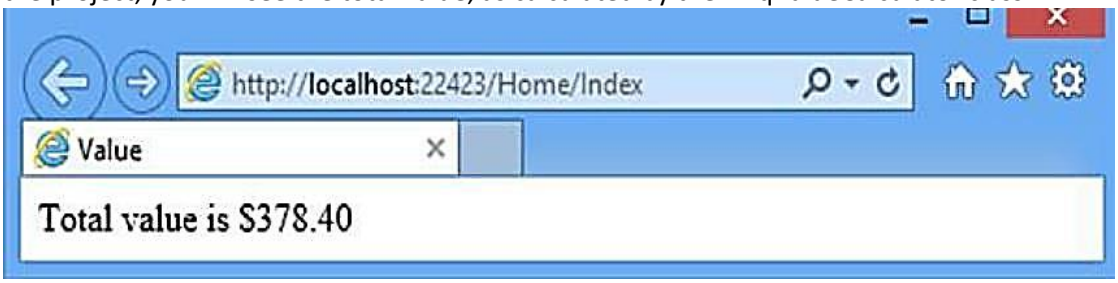
Adding the View

The Contents of the Index.cshtml File

```
@model decimal
@{
Layout = null;
}
<!DOCTYPE html>
```

```
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>Value</title>
</head>
<body>
<div>
Total value is $@Model
</div>
</body>
</html>
```

This view uses the @Model expression to display the value of the decimal passed from the action method. If you start the project, you will see the total value, as calculated by the LinqValueCalculator class.



Unit Testing with Visual Studio

We will use the built-in unit test support that comes with Visual Studio, but there are other .NET unit-test packages available. The most popular is probably NUnit, but all of the test packages do much the same thing. The reason we have selected the Visual Studio test tools is that I like the integration with the rest of the IDE.

To demonstrate the Visual Studio unit-test support, we add a new implementation of the IDiscountHelper interface to the example project. Create a new file in the Models folder called MinimumDiscountHelper.cs .

```
using System;

namespace EssentialTools.Models
{
  public class MinimumDiscountHelper : IDiscountHelper
  {
    public decimal ApplyDiscount(decimal totalParam)
    {
      throw new NotImplementedException();
    }
  }
}
```

Our objective in this example is to make the MinimumDiscountHelper demonstrate the following behaviors:

- ✓ If the total is greater than $100, the discount will be 10 percent.
- ✓ If the total is between $10 and $100 inclusive, the discount will be $5.
- ✓ No discount will be applied on totals less than $10.

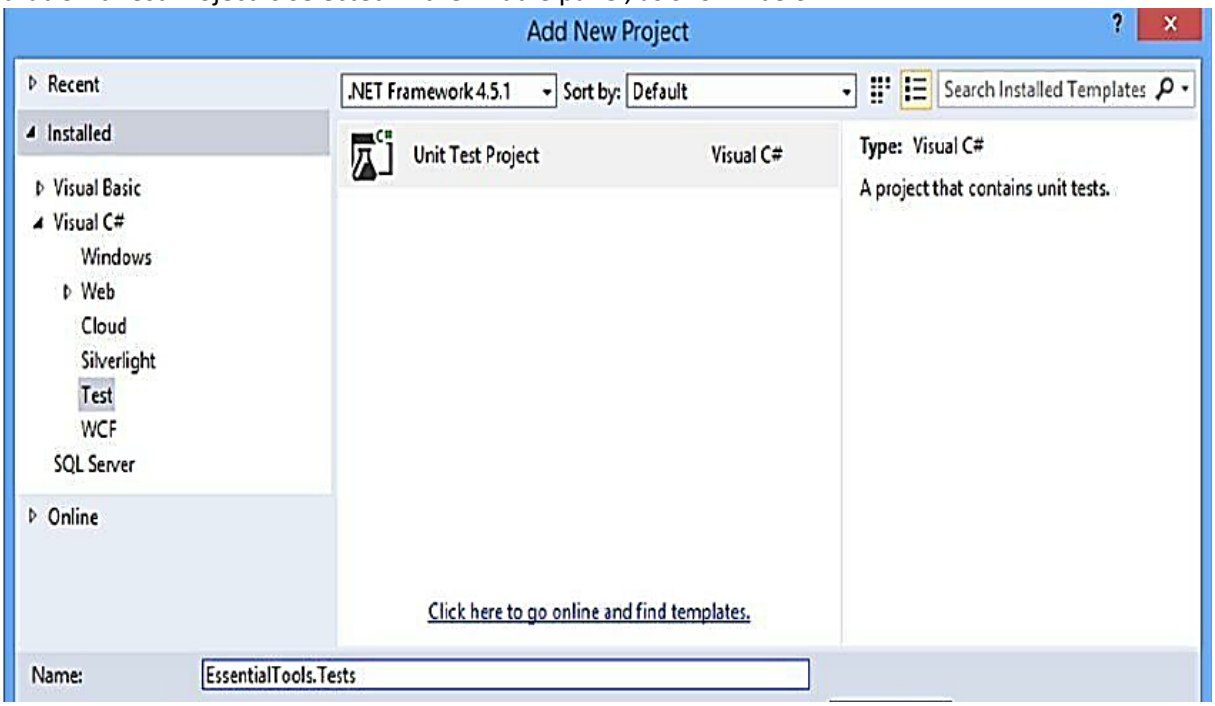
- ✓ An ArgumentOutOfRangeException will be thrown for negative totals.

The MinimumDiscountHelper class does not implement any of these behaviors yet. We are going to follow the Test Driven Development (TDD) approach of writing the unit tests and only then implement the code.

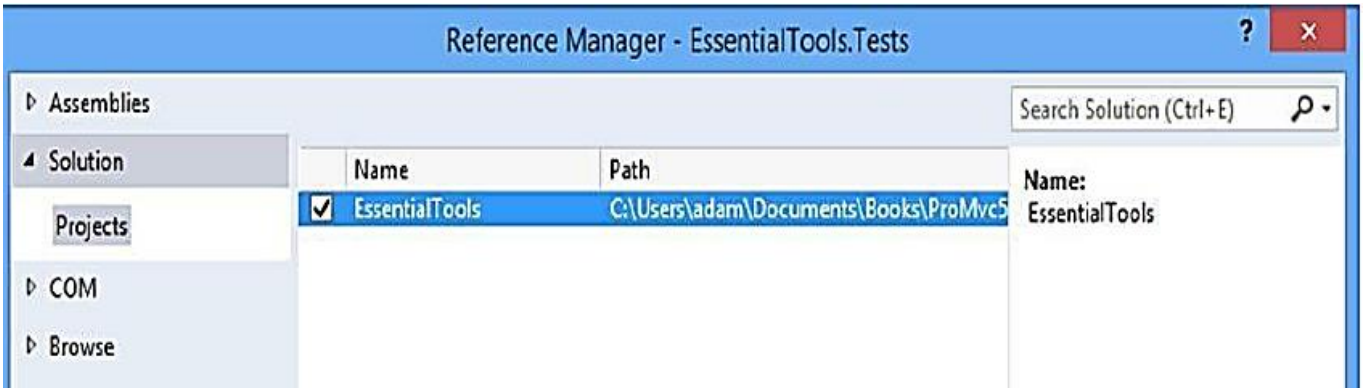
Creating the Unit Test Project
The first step is to create the unit test project by right-clicking the top-level item in the Solution and selecting Add New Project from the pop-up menu.

Tip You can choose to create a test project when you create a new MVC project: there is an Add Unit Tests option on the dialog where you choose the initial content for the project.

This will open the Add New Project dialog. Select Test from the Visual C# templates section in the left panel and ensure that Unit Test Project is selected in the middle panel, as shown below



Set the project name to EssentialTools.Tests and click the OK button to create the new project, which Visual Studio will add to the current solution alongside the MVC application project.
We need to give the test project a reference to the application project so that it can access the classes and perform tests upon them. Right-click the References item for the EssentialTools.Tests project in the Solution Explorer, and then select Add Reference from the pop-up menu. Click Solution in the left panel and check the box next to the EssentialTools item, as shown below:



Creating the Unit Tests

We will add our unit tests to the UnitTest1.cs file in the EssentialTools.Tests project. The paid-for Visual Studio editions have some nice features for automatically generating test methods for a class that are not available in the Express edition, but we can still create useful and meaningful tests. To get started, we made the changes as shown:

Adding the Test Methods to the UnitTest1.cs File

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EssentialTools.Models;

namespace EssentialTools.Tests {

  [TestClass]
  public class UnitTest1
  {
    private IDiscountHelper getTestObject()
    {
      return new MinimumDiscountHelper();
    }

  [TestMethod]
  public void Discount_Above_100()
  {
    // arrange
    IDiscountHelper target = getTestObject();
    decimal total = 200;

    // act
    var discountedTotal = target.ApplyDiscount(total);

    // assert
    Assert.AreEqual(total * 0.9M, discountedTotal);
  }
}
}
```

We have added a single unit test. A class that contains tests is annotated with the TestClass attribute and individual tests are methods annotated with the TestMethod attribute. Not all methods in a unit test class have to be unit tests. To

demonstrate this, we have defined the getTestObject method, which we will use to arrange our tests. Because this method does not have a TestMethod attribute, Visual Studio will not treat it as a unit test.

*Notice that we had to add a using statement to import the EssentialTools.Models namespace into the test class. Test classes are just regular C# classes and have no special knowledge about the MVC project. It is the TestClass and TestMethod attributes which add the testing magic to the project.

You can see that we have followed the arrange/act/assert (A/A/A) pattern in the unit test method. There are countless conventions about how to name unit tests, but my advice is simply that you use names that make it clear what the test is checking. Our unit test method is called Discount_Above_100, which is clear and meaningful to me. But all that really matters is that you (and your team) understand whatever naming pattern you settle on, so you adopt a different naming scheme if you do not like mine.

We set up the test method by calling the getTestObject method, which creates an instance of the object we are going to test: the MinimumDiscountHelper class in this case. We also define the total value with which we are going to test. This is the arrange section of the unit test.

For the act section of the test, we call the MinimumDiscountHelper.ApplyDiscount method and assign the result to the discountedTotal variable.

Finally, for the assert section of the test, we use the Assert.AreEqual method to check that the value we got back from the ApplyDiscount method is 90% of the total that we started with. The Assert class has a range of static methods that you can use in your tests. The class is in the Microsoft.VisualStudio.TestTools.UnitTesting namespace along with some additional classes that can be useful for setting up and performing tests.

The Assert class is the one that is used the most and here is the summarized important methods.

Static Assert Methods

Method	Description
AreEqual<T>(T, T) AreEqual<T>(T, T, string)	Asserts that two objects of type T have the same value.
AreNotEqual<T>(T, T) AreNotEqual<T>(T, T, string)	Asserts that two objects of type T do not have the same value.
AreSame<T>(T, T) AreSame<T>(T, T, string)	Asserts that two variables refer to the same object.
AreNotSame<T>(T, T) AreNotSame<T>(T, T, string)	Asserts that two variables refer to different objects.
Fail() Fail(string)	Fails an assertion: no conditions are checked.
Inconclusive() Inconclusive(string)	Indicates that the result of the unit test cannot be definitively established.
IsTrue(bool) IsTrue(bool, string)	Asserts that a bool value is true. Most often used to evaluate an expression that returns a bool result.
IsFalse(bool) IsFalse(bool, string)	Asserts that a bool value is false.
IsNull(object) IsNull(object, string)	Asserts that a variable is not assigned an object reference.
IsNotNull(object) IsNotNull(object, string)	Asserts that a variable is assigned an object reference.
IsInstanceOfType(object, Type) IsInstanceOfType(object, Type, string)	Asserts that an object is of the specified type or is derived from the specified type.
IsNotInstanceOfType(object, Type) IsNotInstanceOfType(object, Type, string)	Asserts that an object is not of the specified type.

Each of the static methods in the Assert class allows you to check some aspect of your unit test and the methods throw an exception if the check fails. All of the assertions have to succeed for the unit test to pass. Each of the methods in the table has an overloaded version that takes a string parameter. The string is included as the message element of the exception if the assertion fails. The AreEqual and AreNotEqual methods have a number of overloads that cater to comparing specific types. For example, there is a version that allows strings to be compared without taking case into account.

*One noteworthy member of the Microsoft.VisualStudio.TestTools.UnitTesting namespace is the ExpectedException attribute. This is an assertion that succeeds only if the unit test throws an exception of the type specified by the ExceptionType parameter. This is a neat way of ensuring that exceptions are thrown without needing to mess around with try...catch blocks in your unit test.

Let's have added further tests to the test project to validate the other behaviors we want for my MinimumDiscountHelper class.

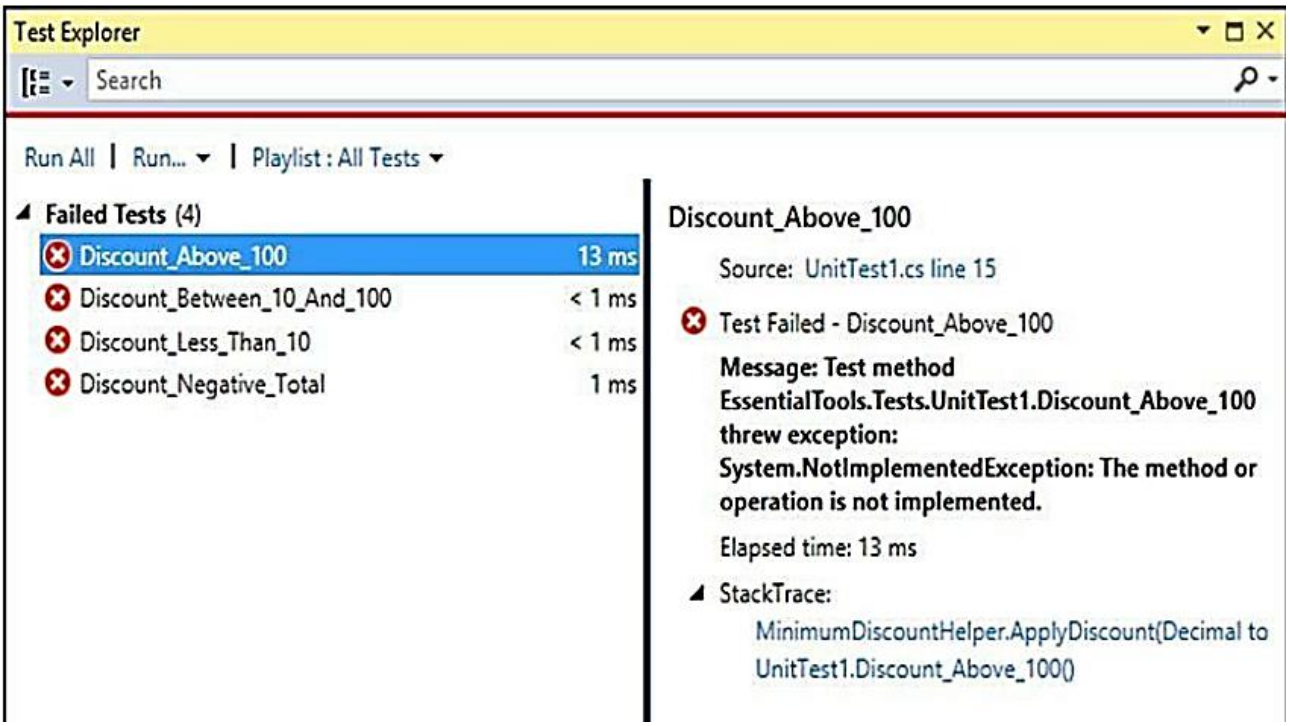
Defining the Remaining Tests in the UnitTest1.cs File

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EssentialTools.Models;

namespace EssentialTools.Tests
{
  [TestClass]
  public class UnitTest1
  {
    private IDiscountHelper getTestObject() {
    return new MinimumDiscountHelper();
  }

  [TestMethod]
  public void Discount_Above_100()
  {
    // arrange
    IDiscountHelper target = getTestObject();
    decimal total = 200;

    // act
    var discountedTotal = target.ApplyDiscount(total);

    // assert
    Assert.AreEqual(total * 0.9M, discountedTotal);
  }

  [TestMethod]
  public void Discount_Between_10_And_100()
  {
    //arrange
    IDiscountHelper target = getTestObject();
```

```
    // act
    decimal TenDollarDiscount = target.ApplyDiscount(10);
    decimal HundredDollarDiscount = target.ApplyDiscount(100);
    decimal FiftyDollarDiscount = target.ApplyDiscount(50);

    // assert
    Assert.AreEqual(5, TenDollarDiscount, "$10 discount is wrong");
    Assert.AreEqual(95, HundredDollarDiscount, "$100 discount is wrong");
    Assert.AreEqual(45, FiftyDollarDiscount, "$50 discount is wrong");
  }

  [TestMethod]
  public void Discount_Less_Than_10()
  {
    //arrange
    IDiscountHelper target = getTestObject();

    // act
    decimal discount5 = target.ApplyDiscount(5);
    decimal discount0 = target.ApplyDiscount(0);

    // assert
    Assert.AreEqual(5, discount5);
    Assert.AreEqual(0, discount0);
  }

  [TestMethod]
  [ExpectedException(typeof(ArgumentOutOfRangeException))]
  public void Discount_Negative_Total()
  {
    //arrange
    IDiscountHelper target = getTestObject();

    // act
    target.ApplyDiscount(-1);
  }
}
}
```

Running the Unit Tests (and Failing)

Visual Studio provides the Test Explorer window for managing and running tests. Select Windows Test Explorer from the Visual Studio Test menu to see the window and click the Run All button near the top-left corner.



You can see the list of tests we defined in the left-hand panel of the Test Explorer window. All of the tests have failed, of course, because we have yet to implement the method we are testing. You can click any of the tests in the window to see details of why it has failed. The Test Explorer window provides a range of different ways to select and filter unit tests and to choose which tests to run. For our simple example project, however, we will just run all of the tests by clicking Run All.

Implementing the Feature

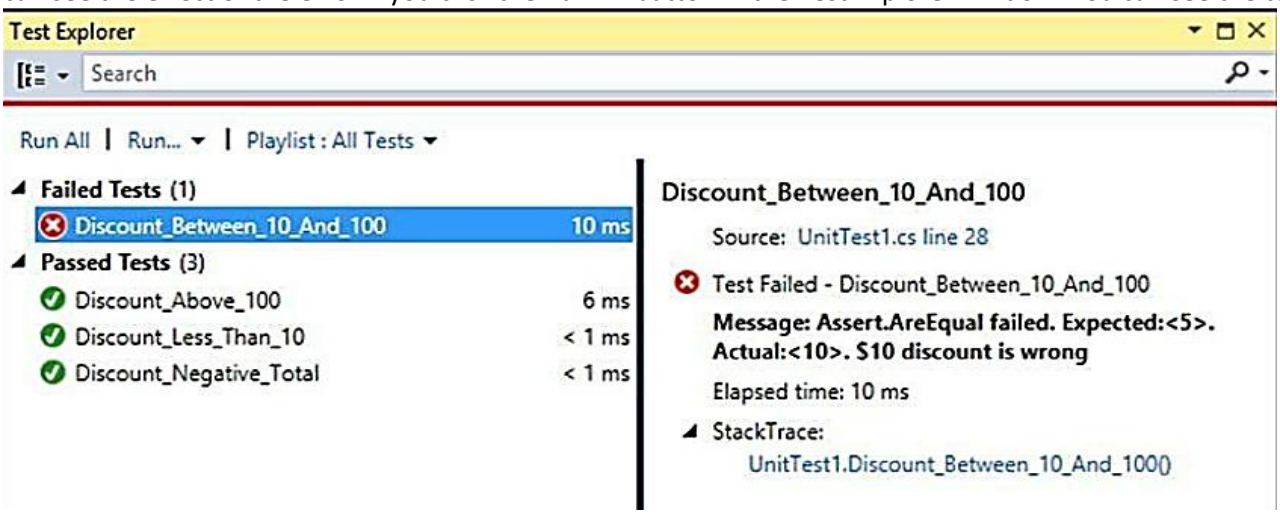
We have reached the point where we can implement the feature, safe in the knowledge that we will be able to check that the code works as expected when we are finished. For all of our preparation, the implementation of the MinimumDiscountHelper class is simple, as shown below:

```
…
using System;

namespace EssentialTools.Models
{
  public class MinimumDiscountHelper : IDiscountHelper
  {
    public decimal ApplyDiscount(decimal totalParam)
    {
      if (totalParam < 0)
      {
        throw new ArgumentOutOfRangeException();
      }
      else if (totalParam > 100)
      {
        return totalParam * 0.9M;
```

```
        }
        else if (totalParam > 10 && totalParam <= 100)
        {
          return totalParam -5;
        }
        else
        {
          return totalParam;
        }
      }
    }
}
```

Testing and Fixing the Code

Let us leave a deliberate error in the code to demonstrate how iterative unit testing with Visual Studio works and you can see the effect of the error if you click the Run All button in the Test Explorer window. You can see the test results.



Visual Studio always tries to promote the most useful information to the top of the Test Explorer window. In this situation, this means that it displays failed tests before passed tests. We can see that we passed three of the unit tests, but we have a problem that the Discount_Between_10_And_100 test method detected. When we click the failed test, we can see that our test expected a result of 5, but actually got a value of 10.

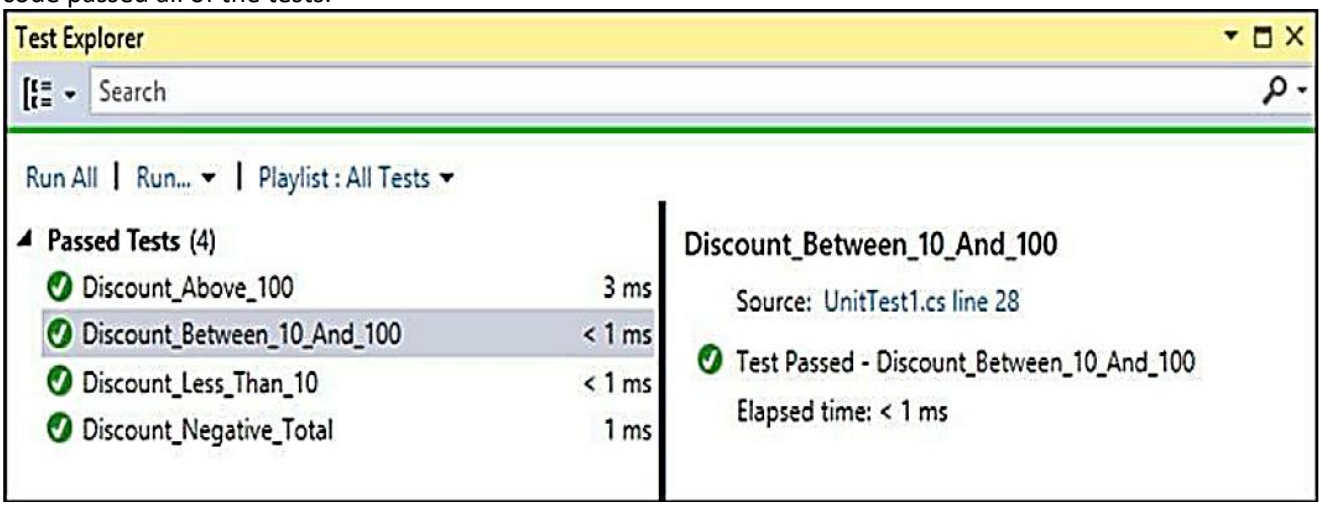
At this point, we return to our code and see that we have not implemented our expected behaviors properly. Specifically, we do not handle the discounts for totals that are 10 or 100 properly. The problem is in this statement from the MinumumDiscountHelper class:

```
...
} else if (totalParam > 10 && totalParam < 100) {
...
```

The specification that we are working to implement sets out the behavior for values which are between $10 and $100 inclusive, but our implementation excludes those values and only checks for values which are greater than $10, excluding totals which are exactly $10.

Can you fix our code?

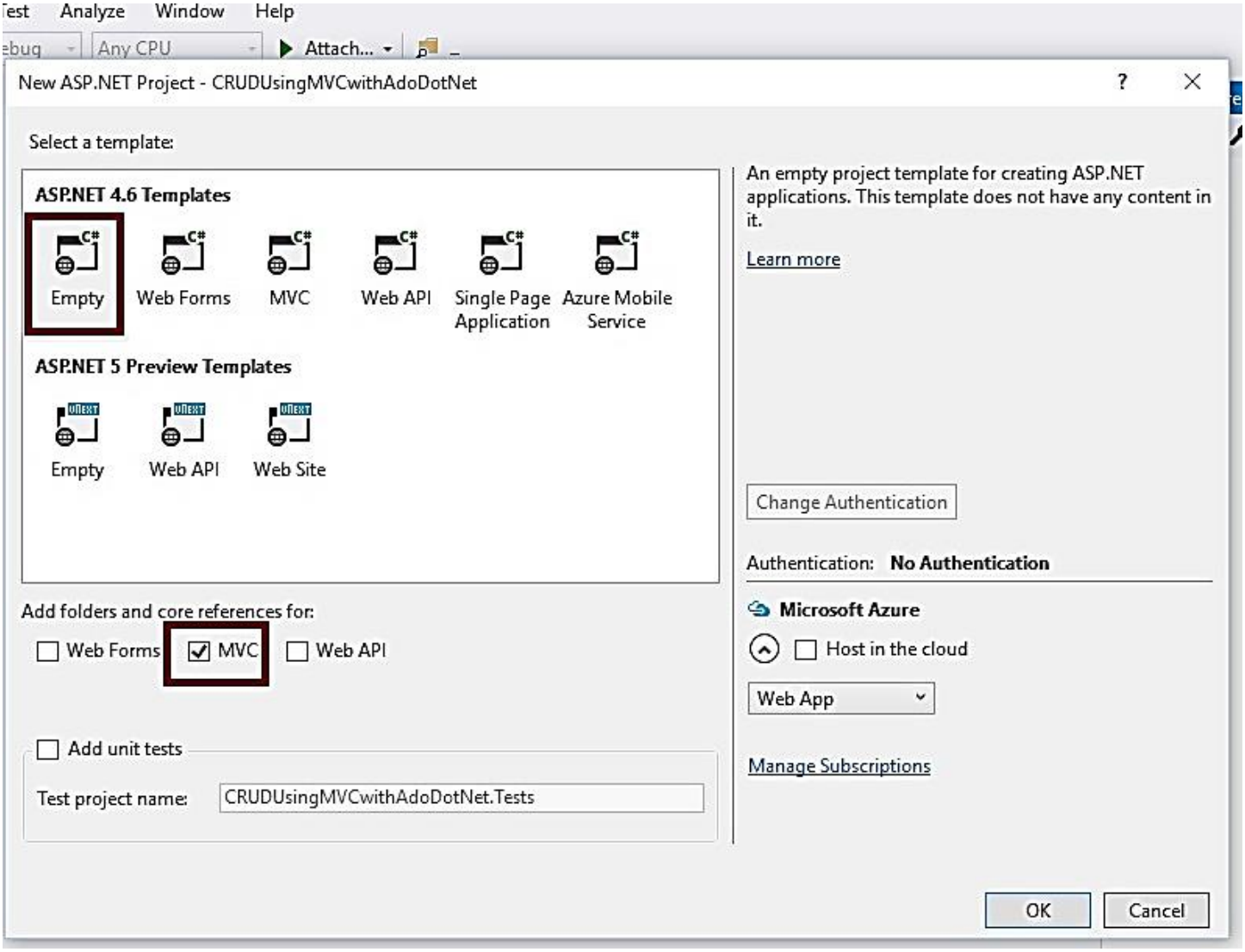
When we click the Run All button in the Test Explorer window after fixing the code, the results should show that our code passed all of the tests.



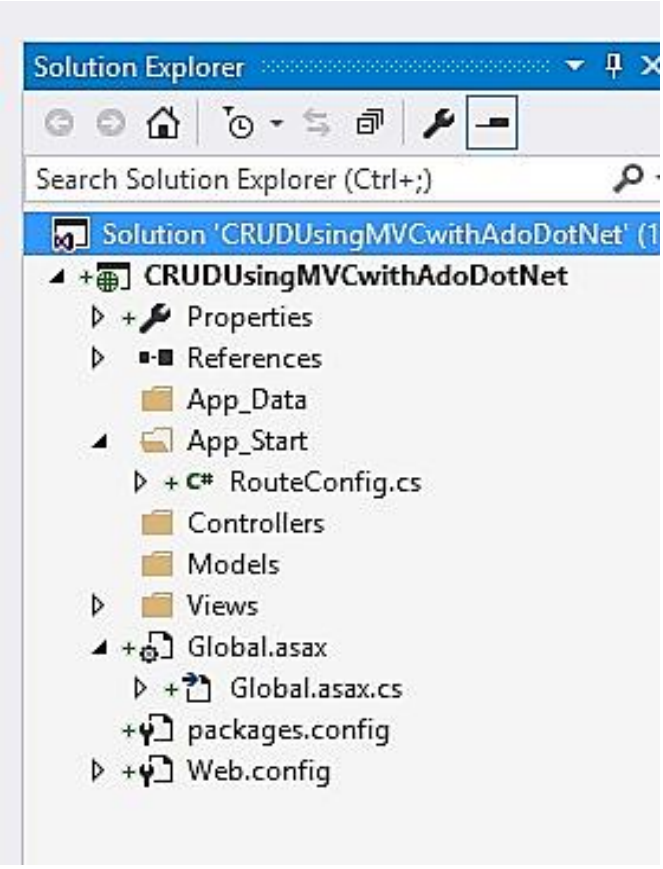
Sample Project 1: Employee Management Site (ASP.NET MVC + ADO.NET)

Step 1: Create an MVC Application

New Project → ASP.NET Web Application Template, then provide the Project a name as you wish :

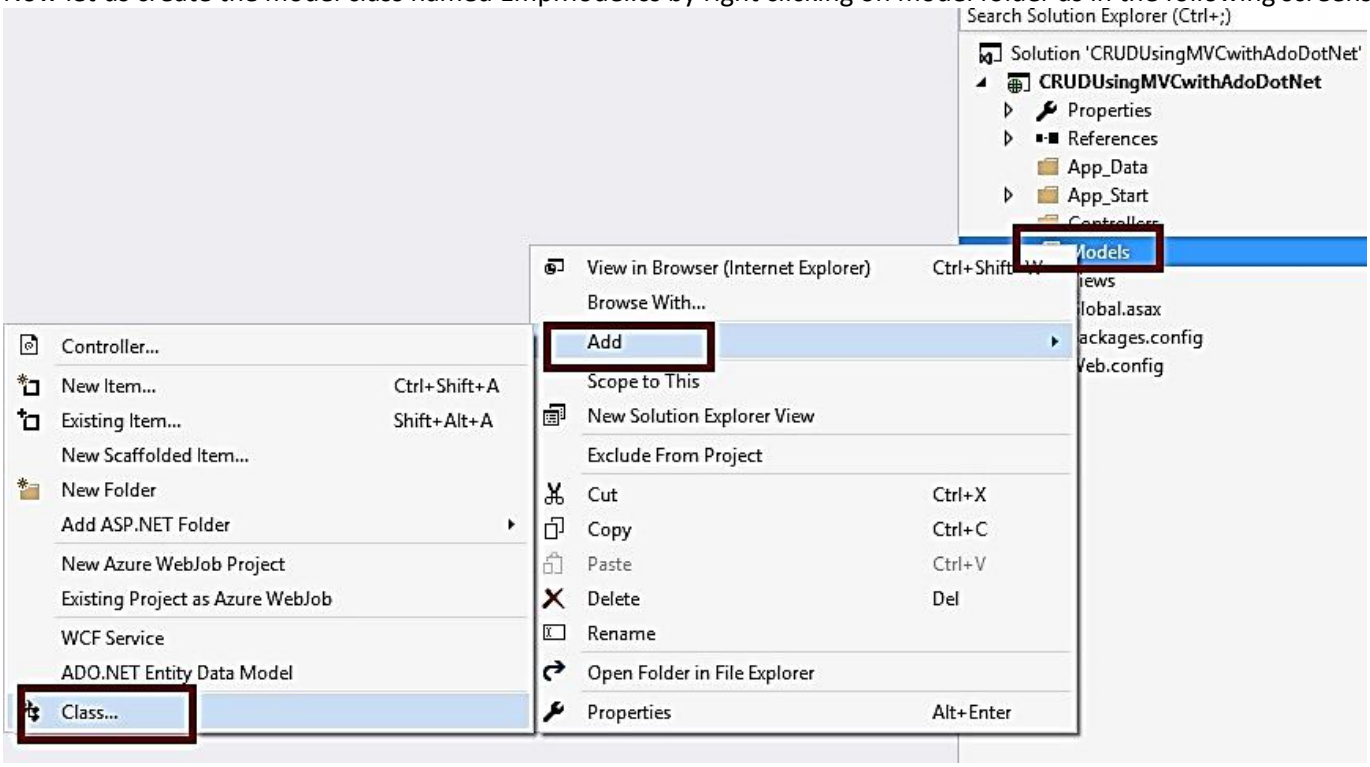


As shown in the preceding screenshot, click on Empty template and check MVC option, then click OK. This will create an empty MVC web application whose Solution Explorer will look like the following:



Step 2: Create Model Class

Now let us create the model class named EmpModel.cs by right clicking on model folder as in the following screenshot:



Note: It is not mandatory that Model class should be in Model folder, it is just for better readability you can create this class anywhere in the solution explorer. This can be done by creating different folder name or without folder name or in a separate class library.

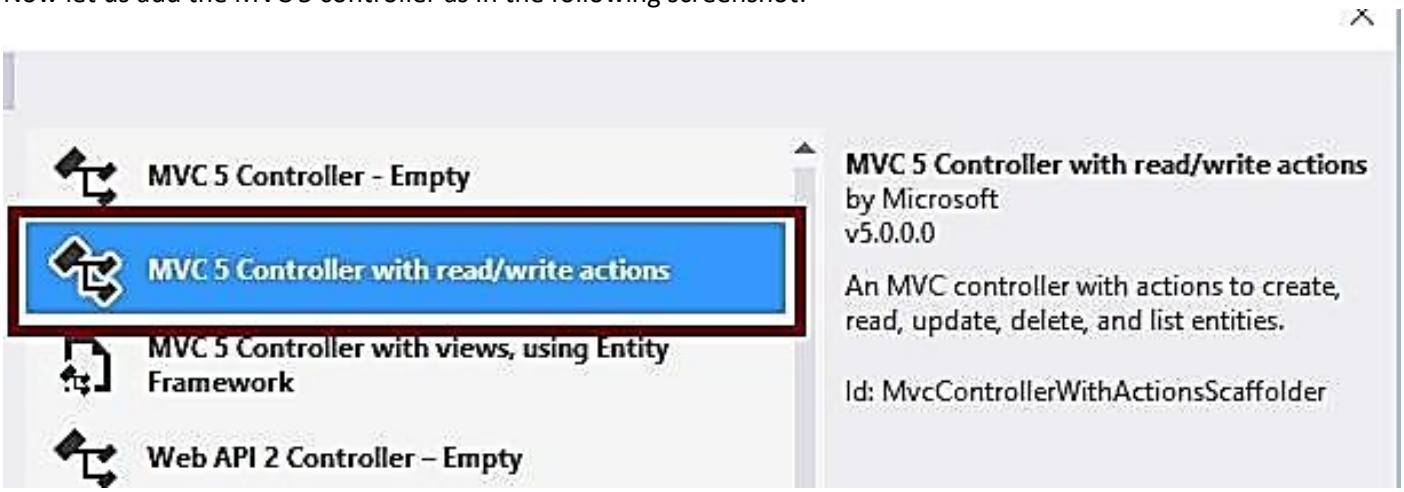
EmpModel.cs class code snippet:

```
…
public class EmpModel
 {
    [Display(Name = "Id")]
    public int Empid { get; set; }

    [Required(ErrorMessage = "First name is required.")]
    public string Name { get; set; }

    [Required(ErrorMessage = "City is required.")]
    public string City { get; set; }

    [Required(ErrorMessage = "Address is required.")]
    public string Address { get; set; }

 }
```

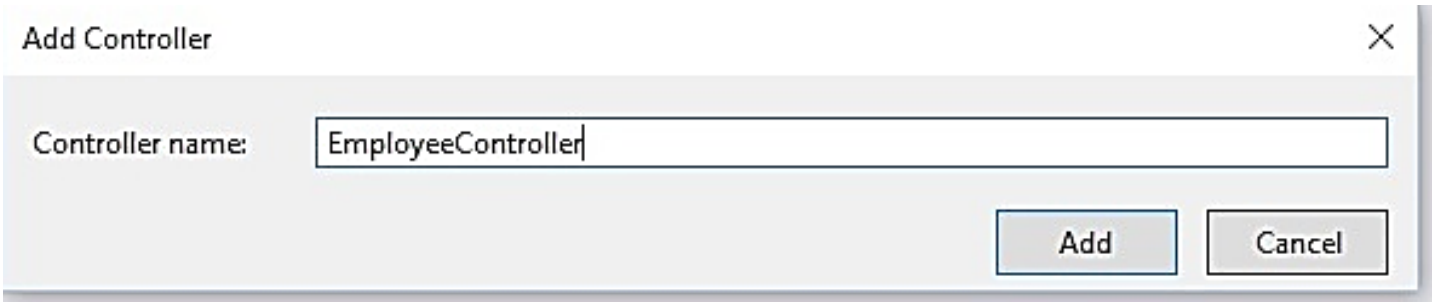
In the above model class we have added some validation on properties with the help of ***DataAnnotations***.

Step 3: Create Controller.

Now let us add the MVC 5 controller as in the following screenshot:



After clicking on Add button it will show the following window. Now specify the Controller name as Employee with suffix Controller as in the following screenshot:



Note: The controller name must be having suffix as 'Controller' after specifying the name of controller.

After clicking on Add button controller is created with by default code that support CRUD operations and later on we can configure it as per our requirements.

Step 4: Create Table and Stored procedures.

Now before creating the views let us create the table name Employee in database according to our model fields to store the details:

VITHAL.RMS - dbo.Employee | SQLQuery1.sql - VITHAL.master (sa (5

Column Name	Data Type	Allow Nulls
Id	int	☐
Name	varchar(50)	☑
City	varchar(50)	☑
Address	varchar(50)	☑

I hope you have created the same table structure as shown above. Now create the stored procedures to insert, update, view and delete the details as in the following code snippet:

To Insert Records

```
Create procedure [dbo].[AddNewEmpDetails]
(
  @Name varchar (50),
  @City varchar (50),
  @Address varchar (50)
)
as
begin
  Insert into Employee values(@Name,@City,@Address)
End
```

To View Added Records

```
Create Procedure [dbo].[GetEmployees]
as
begin
  select *from Employee
End
```

To Update Records

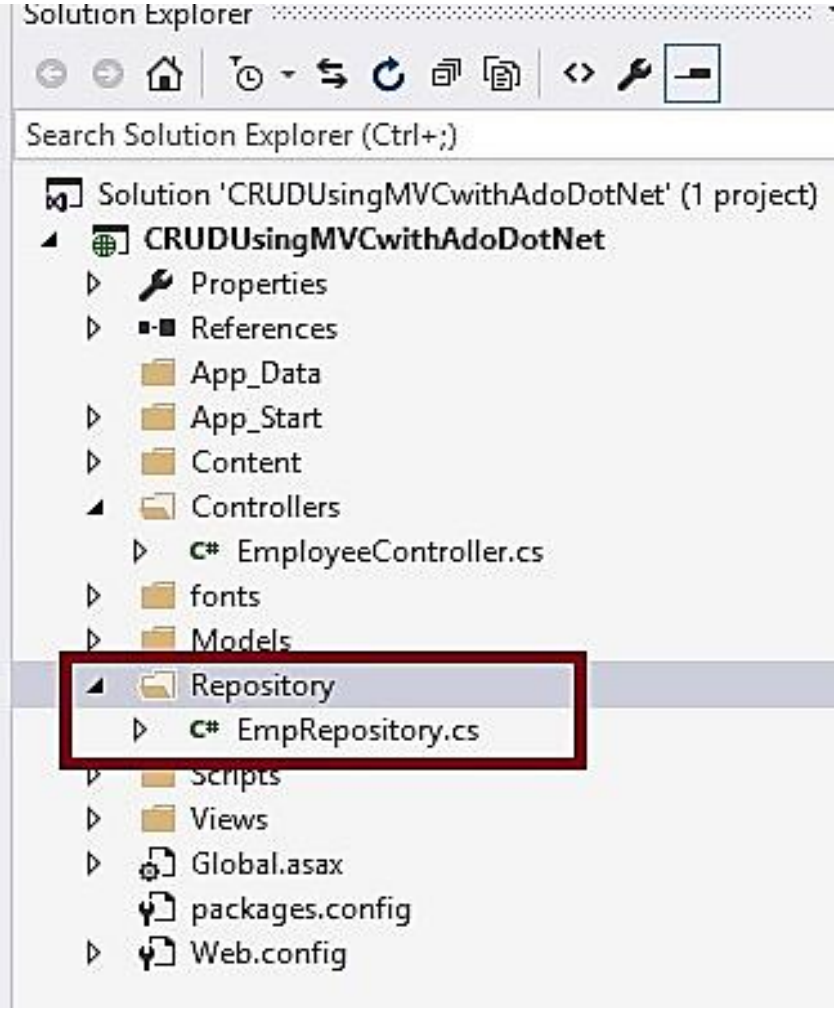
```
Create procedure [dbo].[UpdateEmpDetails]
(
  @EmpId int,
  @Name varchar (50),
  @City varchar (50),
  @Address varchar (50)
)
as
begin
  Update Employee
  set Name=@Name,
  City=@City,
  Address=@Address
  where Id=@EmpId
End
```

To Delete Records

```
Create procedure [dbo].[DeleteEmpById]
(
  @EmpId int
)
as
begin
  Delete from Employee where Id=@EmpId
End
```

Step 5: Create Repository class.

Now create Repository folder and Add EmpRepository.cs class for database related operations, after adding the solution explorer will look like the following screenshot:



Define your connection string in Web.config

```
…
<connectionStrings>
  <add
      name="ConnStringDb1"
      connectionString=@"Data Source=LAPTOP-HTH010Q4\MSSQL2019;Initial Catalog=testdatabase;User
ID=sa;Password=1234567"
      providerName="System.Data.SqlClient" />
</connectionStrings>
…
```

Now create methods in EmpRepository.cs to handle the CRUD operation as in the following screenshot:

EmpRepository.cs

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;
using CRUDUsingMVC.Models;
```

```
using System.Linq;

namespace CRUDUsingMVC.Repository
{
   public class EmpRepository
   {

      private SqlConnection con;
      //To Handle connection related activities
      private void connection()
      {
         string constr = ConfigurationManager.ConnectionStrings["con"].ToString();
         con = new SqlConnection(constr);

      }
      //To Add Employee details
      public bool AddEmployee(EmpModel obj)
      {

         connection();
         SqlCommand com = new SqlCommand("AddNewEmpDetails", con);
         com.CommandType = CommandType.StoredProcedure;
         com.Parameters.AddWithValue("@Name", obj.Name);
         com.Parameters.AddWithValue("@City", obj.City);
         com.Parameters.AddWithValue("@Address", obj.Address);

         con.Open();
         int i = com.ExecuteNonQuery();
         con.Close();
         if (i >= 1)
         {

            return true;

         }
         else
         {

            return false;
         }


      }
      //To view employee details with generic list
      public List<EmpModel> GetAllEmployees()
      {
         connection();
         List<EmpModel> EmpList =new List<EmpModel>();
         SqlCommand com = new SqlCommand("GetEmployees", con);
         com.CommandType = CommandType.StoredProcedure;
```

```
    SqlDataAdapter da = new SqlDataAdapter(com);
    DataTable dt = new DataTable();
    con.Open();
    da.Fill(dt);
    con.Close();

    //Bind EmpModel generic list using LINQ
    EmpList = (from DataRow dr in dt.Rows

               select new EmpModel()
               {
                  Empid = Convert.ToInt32(dr["Id"]),
                  Name = Convert.ToString(dr["Name"]),
                  City = Convert.ToString(dr["City"]),
                  Address = Convert.ToString(dr["Address"])
               }).ToList();


    return EmpList;


}
//To Update Employee details
public bool UpdateEmployee(EmpModel obj)
{

    connection();
    SqlCommand com = new SqlCommand("UpdateEmpDetails", con);

    com.CommandType = CommandType.StoredProcedure;
    com.Parameters.AddWithValue("@EmpId", obj.Empid);
    com.Parameters.AddWithValue("@Name", obj.Name);
    com.Parameters.AddWithValue("@City", obj.City);
    com.Parameters.AddWithValue("@Address", obj.Address);
    con.Open();
    int i = com.ExecuteNonQuery();
    con.Close();
    if (i >= 1)
    {

       return true;

    }
    else
    {

       return false;
    }
```

```
    }
    //To delete Employee details
    public bool DeleteEmployee(int Id)
    {

        connection();
        SqlCommand com = new SqlCommand("DeleteEmpById", con);

        com.CommandType = CommandType.StoredProcedure;
        com.Parameters.AddWithValue("@EmpId", Id);

        con.Open();
        int i = com.ExecuteNonQuery();
        con.Close();
        if (i >= 1)
        {

            return true;

        }
        else
        {

            return false;
        }


    }
  }
}
```

Step 6 : Create Methods into the EmployeeController.cs file.

Now open the EmployeeController.cs and create the following action methods:

```
public class EmployeeController : Controller
  {

    // GET: Employee/GetAllEmpDetails
    public ActionResult GetAllEmpDetails()
    {

        EmpRepository EmpRepo = new EmpRepository();
        ModelState.Clear();
        return View(EmpRepo.GetAllEmployees());
    }
    // GET: Employee/AddEmployee
    public ActionResult AddEmployee()
    {
```

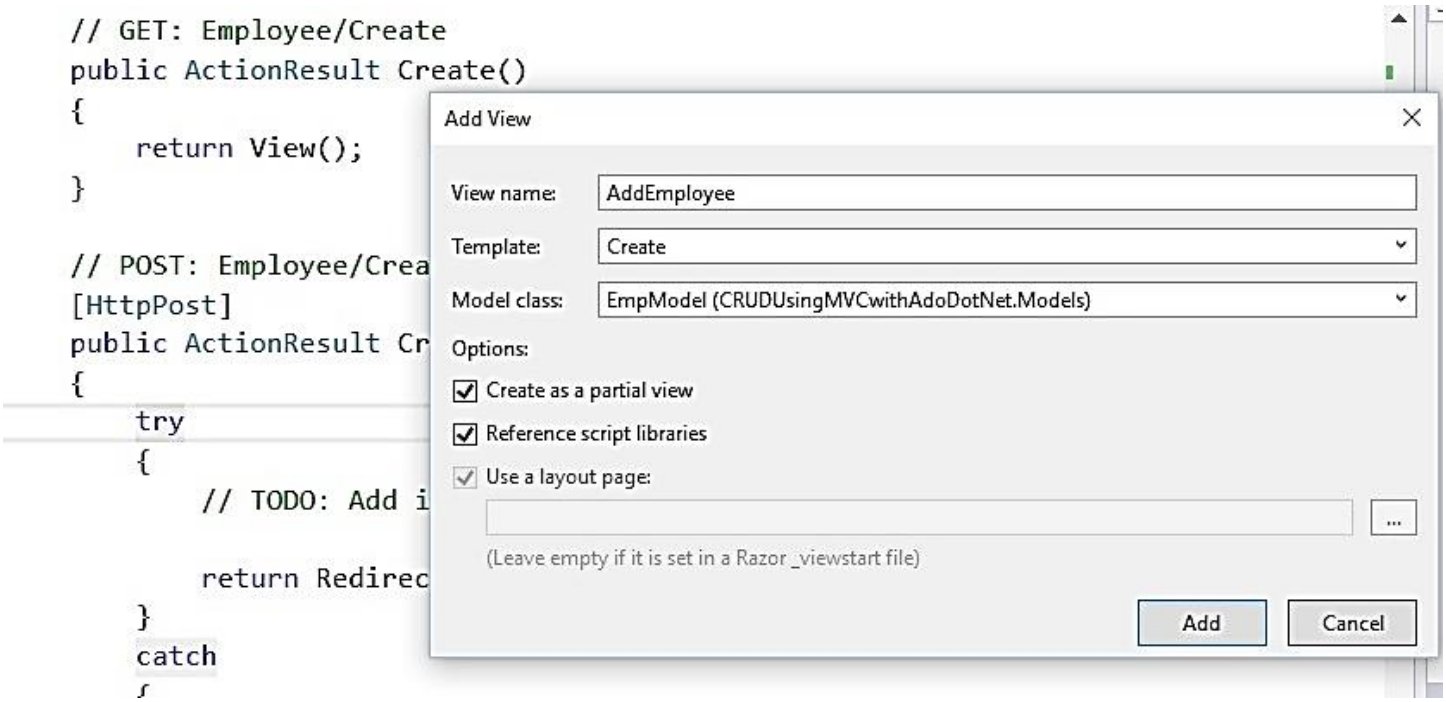
```
    return View();
}

// POST: Employee/AddEmployee
[HttpPost]
public ActionResult AddEmployee(EmpModel Emp)
{
    try
    {
        if (ModelState.IsValid)
        {
            EmpRepository EmpRepo = new EmpRepository();

            if (EmpRepo.AddEmployee(Emp))
            {
                ViewBag.Message = "Employee details added successfully";
            }
        }

        return View();
    }
    catch
    {
        return View();
    }
}

// GET: Employee/EditEmpDetails/5
public ActionResult EditEmpDetails(int id)
{
    EmpRepository EmpRepo = new EmpRepository();



    return View(EmpRepo.GetAllEmployees().Find(Emp => Emp.Empid == id));

}

// POST: Employee/EditEmpDetails/5
[HttpPost]

public ActionResult EditEmpDetails(int id,EmpModel obj)
{
    try
    {
            EmpRepository EmpRepo = new EmpRepository();

            EmpRepo.UpdateEmployee(obj);

            return RedirectToAction("GetAllEmpDetails");
```

```
        }
        catch
        {
            return View();
        }
    }

    // GET: Employee/DeleteEmp/5
    public ActionResult DeleteEmp(int id)
    {
        try
        {
            EmpRepository EmpRepo = new EmpRepository();
            if (EmpRepo.DeleteEmployee(id))
            {
                ViewBag.AlertMsg = "Employee details deleted successfully";

            }
            return RedirectToAction("GetAllEmpDetails");

        }
        catch
        {
            return View();
        }
    }


}
```

Step 7: Create Views.

Create the Partial view to Add the employees

To create the Partial View to add Employees, right click on ActionResult method and then click Add view. Now specify the view name, template name and model class in EmpModel.cs and click on Add button as in the following screenshot:



After clicking on Add button it generates the strongly typed view whose code is given below:

AddEmployee.cshtml

```
@model CRUDUsingMVC.Models.EmpModel
@using (Html.BeginForm())
{

  @Html.AntiForgeryToken()

  <div class="form-horizontal">
    <h4>Add Employee</h4>
    <div>
      @Html.ActionLink("Back to Employee List", "GetAllEmpDetails")
    </div>
    <hr />
    @Html.ValidationSummary(true, "", new { @class = "text-danger" })


    <div class="form-group">
      @Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2" })
      <div class="col-md-10">
        @Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })
      </div>
    </div>

    <div class="form-group">
      @Html.LabelFor(model => model.City, htmlAttributes: new { @class = "control-label col-md-2" })
      <div class="col-md-10">
```

```
            @Html.EditorFor(model => model.City, new { htmlAttributes = new { @class = "form-control" } })
            @Html.ValidationMessageFor(model => model.City, "", new { @class = "text-danger" })
        </div>
    </div>

    <div class="form-group">
        @Html.LabelFor(model => model.Address, htmlAttributes: new { @class = "control-label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.Address, new { htmlAttributes = new { @class = "form-control" } })
            @Html.ValidationMessageFor(model => model.Address, "", new { @class = "text-danger" })
        </div>
    </div>

    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Save" class="btn btn-default" />
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10" style="color:green">
            @ViewBag.Message

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

}

<script src="~/Scripts/jquery-1.10.2.min.js"></script>
<script src="~/Scripts/jquery.validate.min.js"></script>
<script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>
```

To View Added Employees

To view the employee details let us create the partial view named GetAllEmpDetails:
Now click on add button, it will create GetAllEmpDetails.cshtml strongly typed view whose code is given below:

GetAllEmpDetails.CsHtml

```
@model IEnumerable<CRUDUsingMVC.Models.EmpModel>

<p>
   @Html.ActionLink("Add New Employee", "AddEmployee")
</p>

<table class="table">
   <tr>

      <th>
```

```
        @Html.DisplayNameFor(model => model.Name)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.City)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.Address)
      </th>
      <th></th>
    </tr>

    @foreach (var item in Model)
    {
      @Html.HiddenFor(model => item.Empid)
      <tr>

        <td>
          @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.City)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Address)
        </td>
        <td>
          @Html.ActionLink("Edit", "EditEmpDetails", new { id = item.Empid }) |
          @Html.ActionLink("Delete", "DeleteEmp", new { id = item.Empid }, new { onclick = "return confirm('Are sure
wants to delete?');" })
        </td>
      </tr>

    }
</table>
```

To Update Added Employees

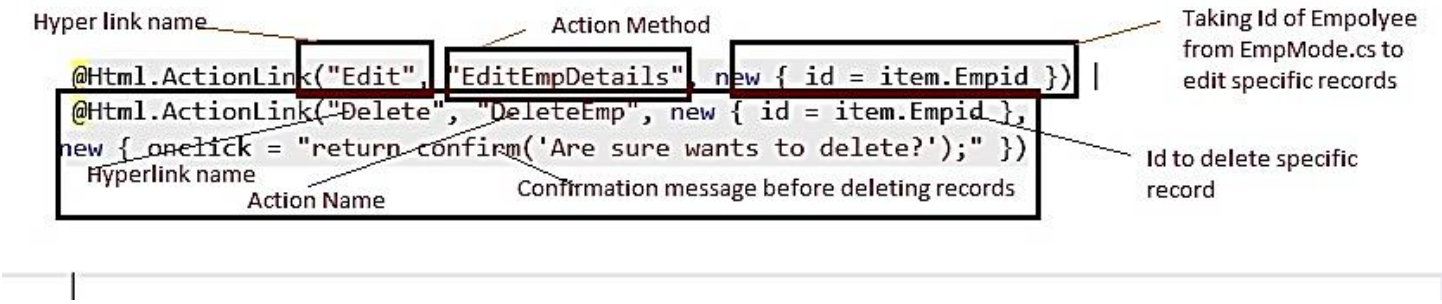
Follow the same procedure and create EditEmpDetails view to edit the employees. After creating the view the code will be like the following:

EditEmpDetails.cshtml

```
@model CRUDUsingMVC.Models.EmpModel
@using (Html.BeginForm())
{
  @Html.AntiForgeryToken()

  <div class="form-horizontal">
    <h4>Update Employee Details</h4>
```

```
    <hr />
    <div>
      @Html.ActionLink("Back to Details", "GetAllEmployees")
    </div>
    <hr />
    @Html.ValidationSummary(true, "", new { @class = "text-danger" })
    @Html.HiddenFor(model => model.Empid)

    <div class="form-group">
      @Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2" })
      <div class="col-md-10">
        @Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })
      </div>
    </div>

    <div class="form-group">
      @Html.LabelFor(model => model.City, htmlAttributes: new { @class = "control-label col-md-2" })
      <div class="col-md-10">
        @Html.EditorFor(model => model.City, new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.City, "", new { @class = "text-danger" })
      </div>
    </div>

    <div class="form-group">
      @Html.LabelFor(model => model.Address, htmlAttributes: new { @class = "control-label col-md-2" })
      <div class="col-md-10">
        @Html.EditorFor(model => model.Address, new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Address, "", new { @class = "text-danger" })
      </div>
    </div>

    <div class="form-group">
      <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Update" class="btn btn-default" />
      </div>
    </div>
  </div>
}
<script src="~/Scripts/jquery-1.10.2.min.js"></script>
<script src="~/Scripts/jquery.validate.min.js"></script>
<script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>
```

Step 8: Configure Action Link to Edit and delete the records



The above ActionLink I have added in GetAllEmpDetails.CsHtml view because from there we will delete and update the records.

Step 9: Configure RouteConfig.cs to set default action

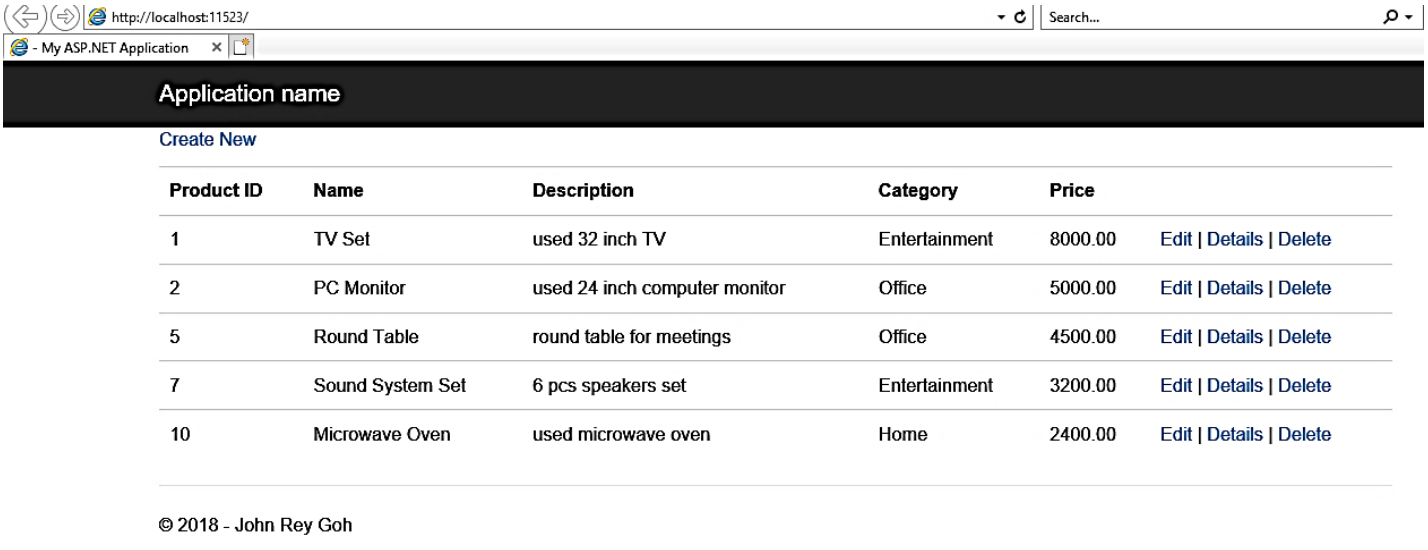
```
public class RouteConfig
{
   public static void RegisterRoutes(RouteCollection routes)
   {
      routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

      routes.MapRoute(
         name: "Default",
         url: "{controller}/{action}/{id}",
         defaults: new { controller = "Employee", action = "AddEmployee", id = UrlParameter.Optional }
      );
   }
}
```

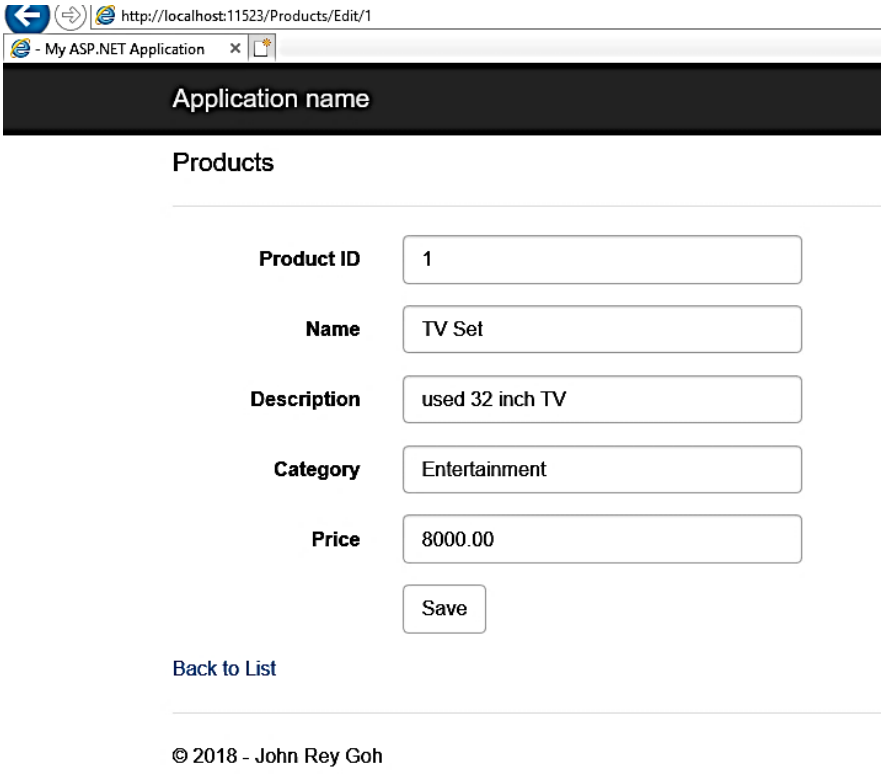
From the above RouteConfig.cs the default action method we have set is AddEmployee. It means that after running the application the AddEmployee view will be executed first.

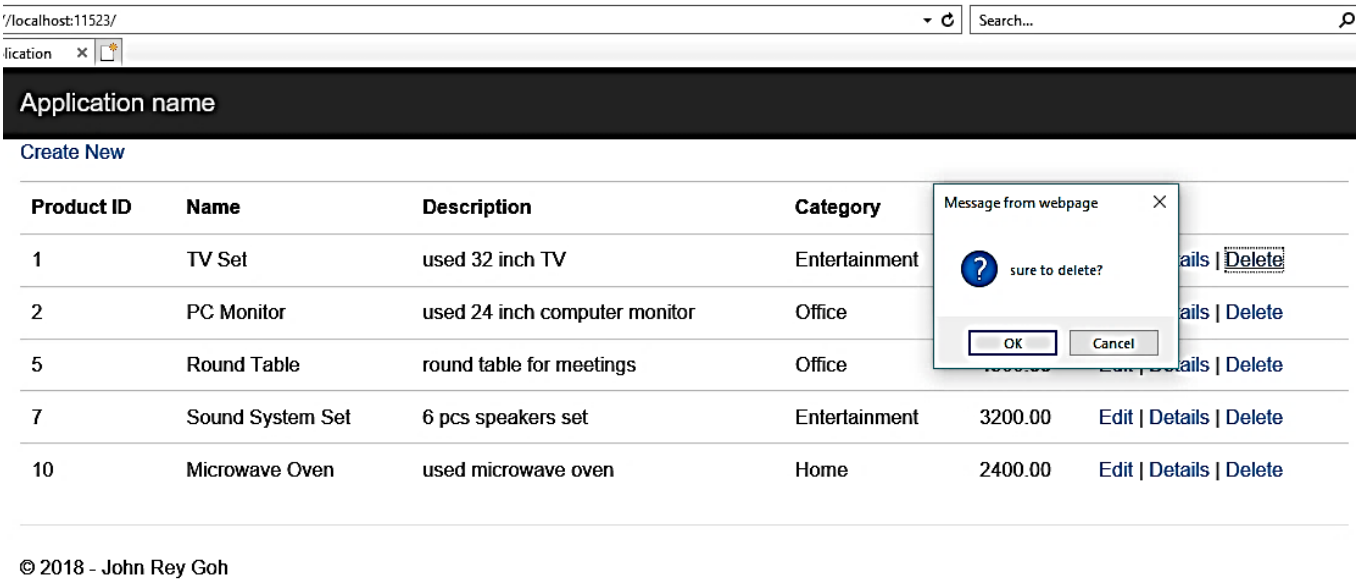
Now after adding the all model, views and controller our solution explorer will be look like as in the following screenshot:

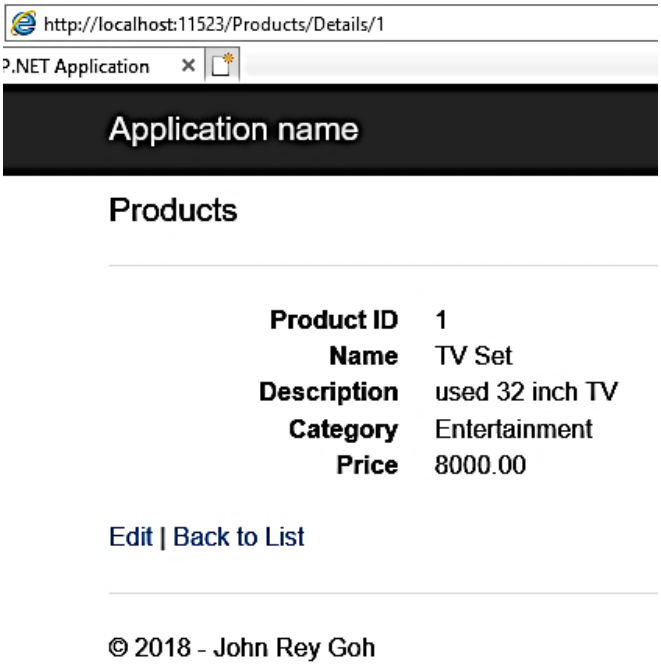
Step 10: Run the application and test.

Note: Configure the database connection in the web.config file depending on your database server location.


http://localhost:11523/
Search...
- My ASP.NET Application
Application name
Create New
Product ID Name Description Category Price
1 TV Set used 32 inch TV Entertainment 8000.00 Edit | Details | Delete
2 PC Monitor used 24 inch computer monitor Office 5000.00 Edit | Details | Delete
5 Round Table round table for meetings Office 4500.00 Edit | Details | Delete
7 Sound System Set 6 pcs speakers set Entertainment 3200.00 Edit | Details | Delete
10 Microwave Oven used microwave oven Home 2400.00 Edit | Details | Delete
© 2018 - John Rey Goh

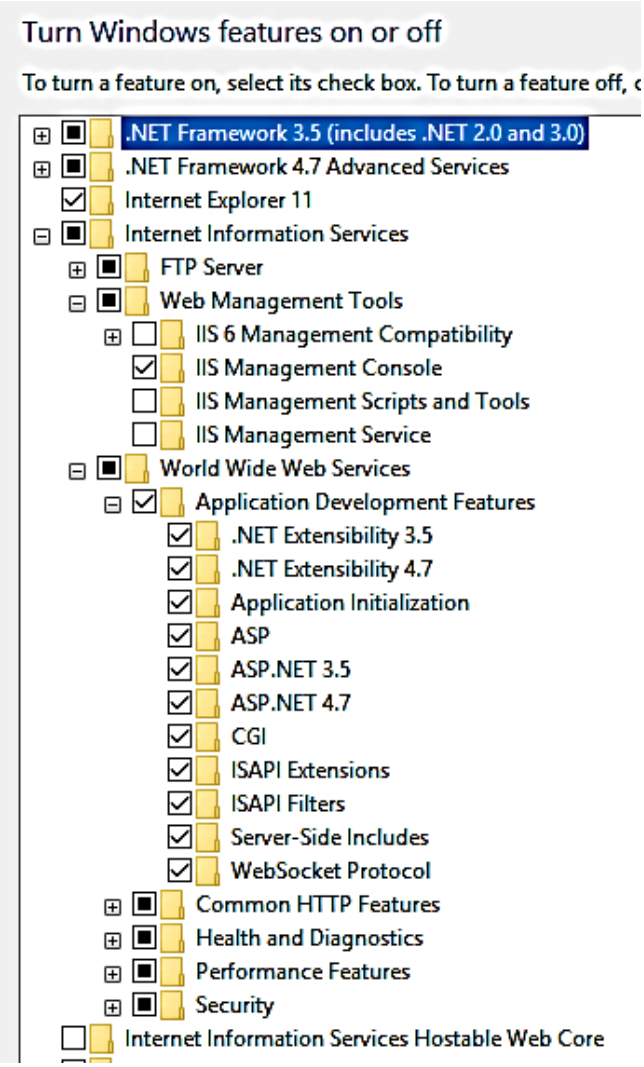

http://localhost:11523/Products/Edit/1
- My ASP.NET Application
Application name
Products
Product ID 1
Name TV Set
Description used 32 inch TV
Category Entertainment
Price 8000.00
Save
Back to List
© 2018 - John Rey Goh



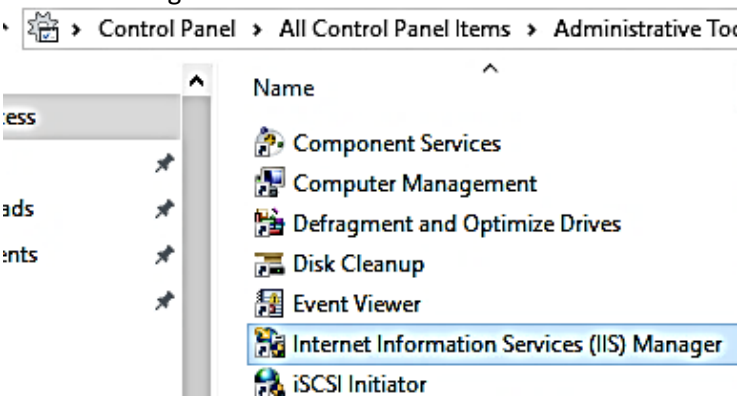


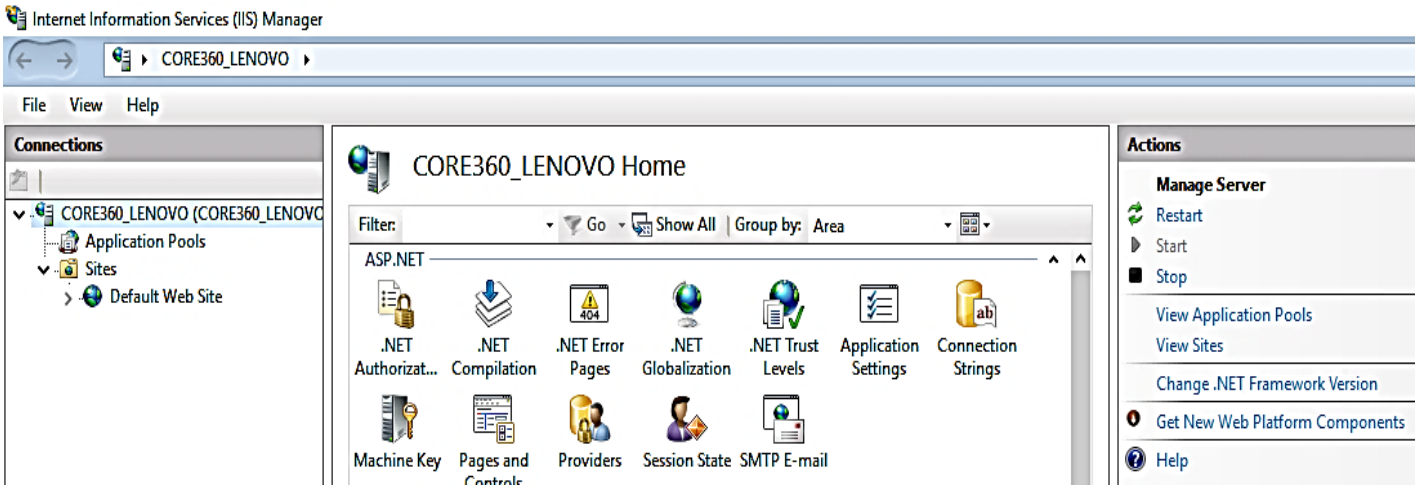
Deploying the Application

1. Plan where to deploy web application (Local IIS, Microsoft Azure, or Third Party hosting). In our activity, we will deploy it on our local IIS.
2. Create a folder where we would publish our application. (ex. C:/local_aspnet).
3. On windows 10, we will enable our IIS feature. Go to control panel→programs and features→turn on or off windows features and navigate to the IIS feature as follows:

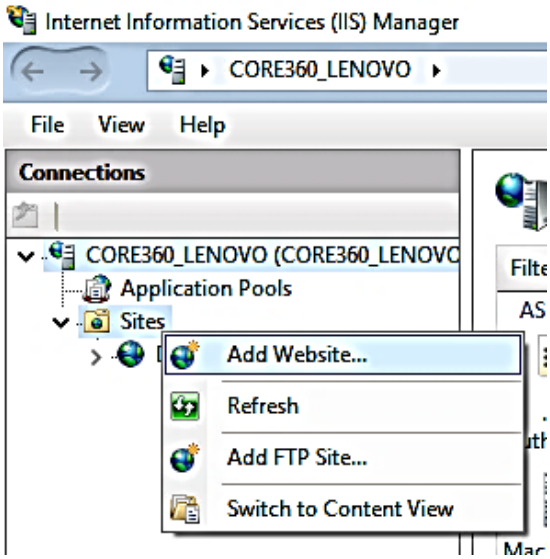


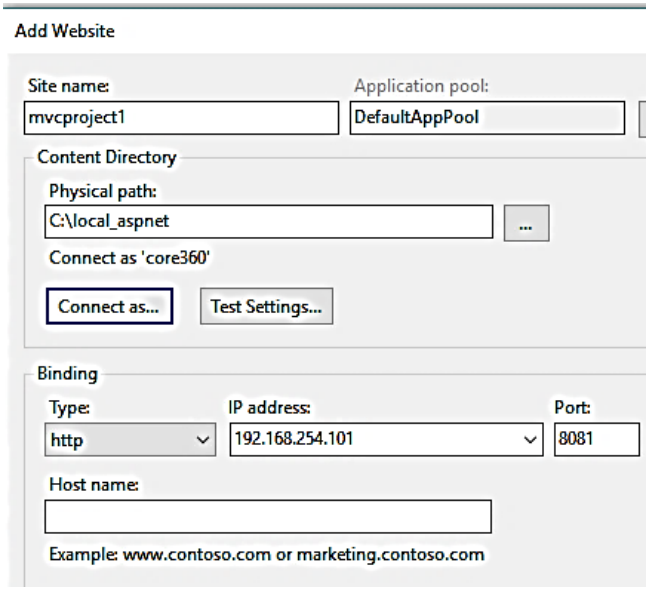
4. After installing the features, your can access IIS Management through Control Panel→Administrative Tools→IIS Manager

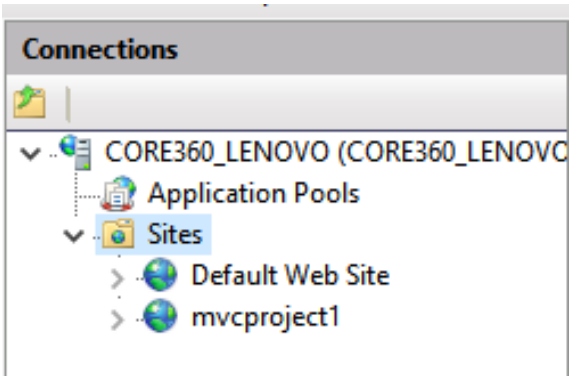




5. In IIS Manager, create a new website under Sites (ex. MVCProject1). Then follow the configurations as shown below:



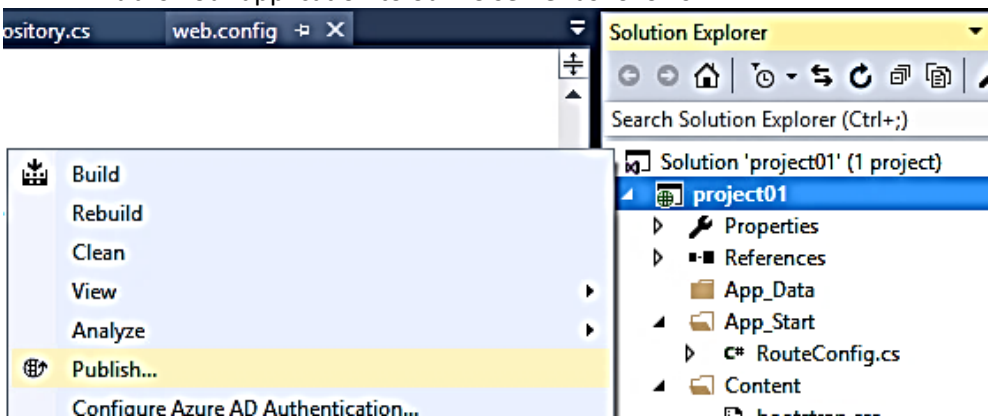


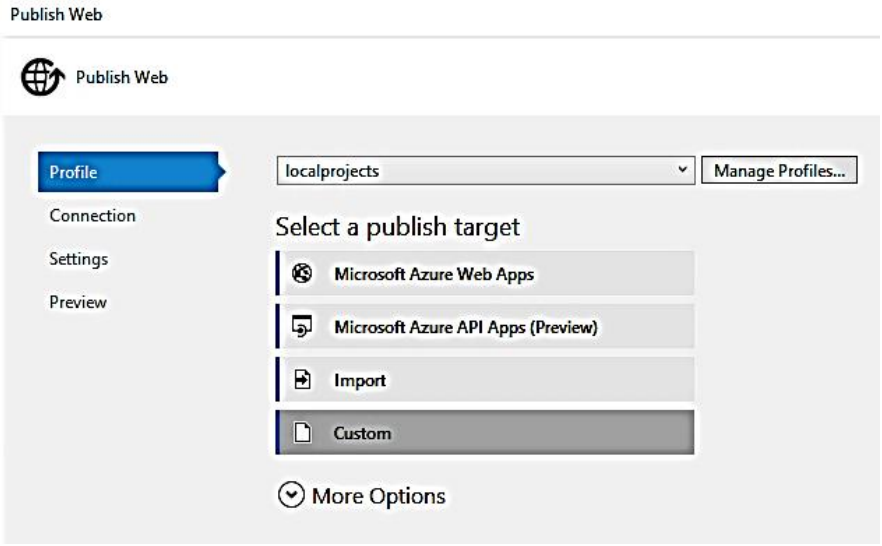


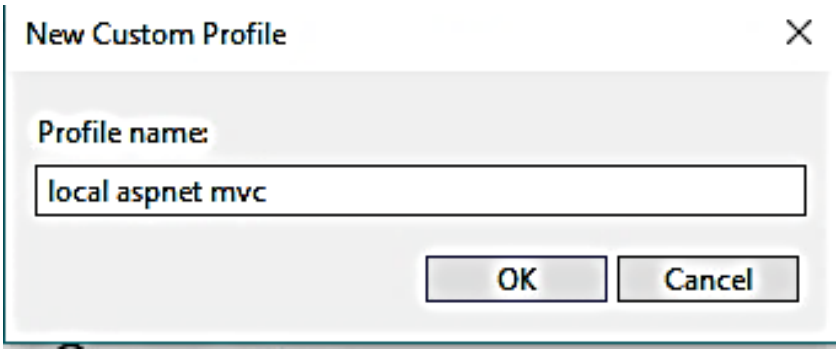
6. In our Project in Visual Studio (launch VS as Administrator to Publish!), (after you have tested your app), add the following configuration in **web.config** (**NOT Web.config!**)

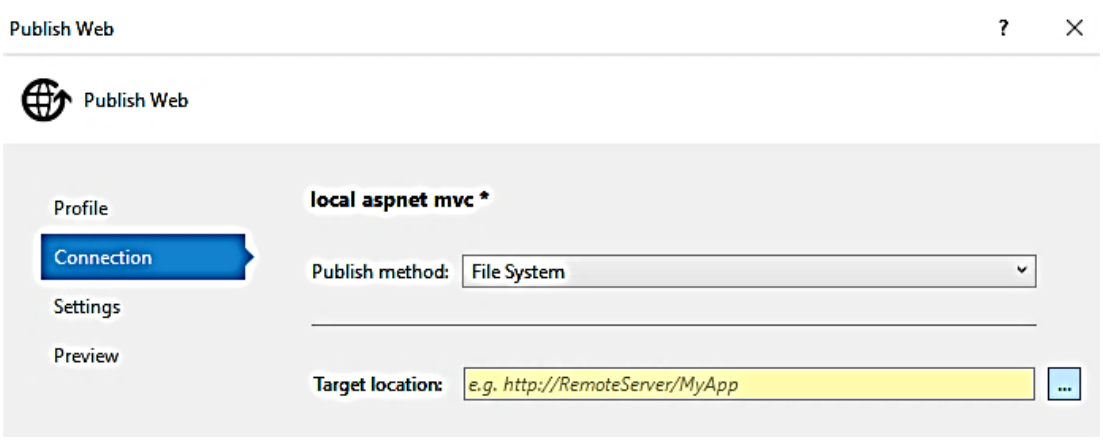
```
<system.webServer>
  <handlers>
    <remove name="BlockViewHandler"/>
    <add name="BlockViewHandler" path="*" verb="*" preCondition="integratedMode"
type="System.Web.HttpNotFoundHandler" />
  </handlers>

  <modules runAllManagedModulesForAllRequests="true" />

 </system.webServer>
```

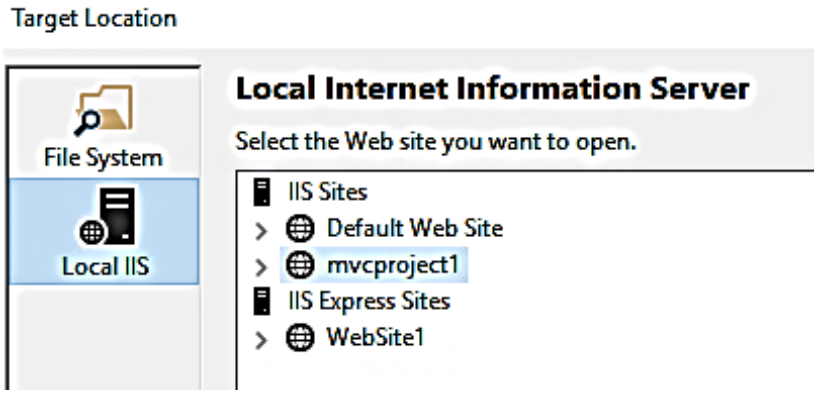
7. Publish our application to our IIS server as follows:

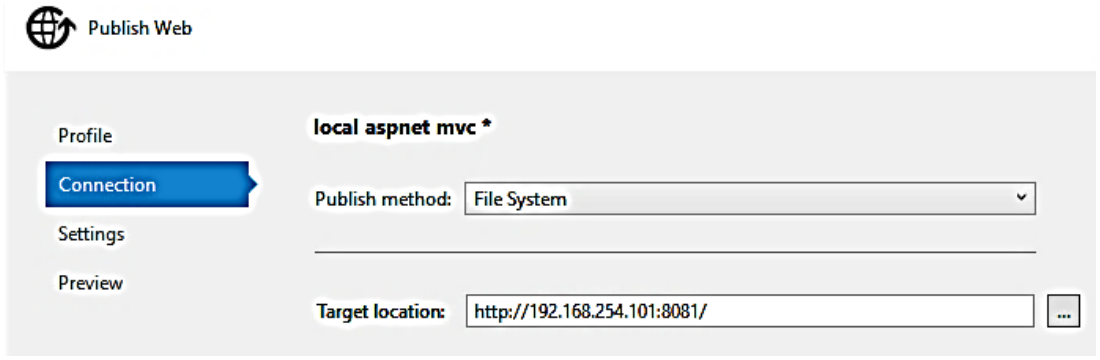


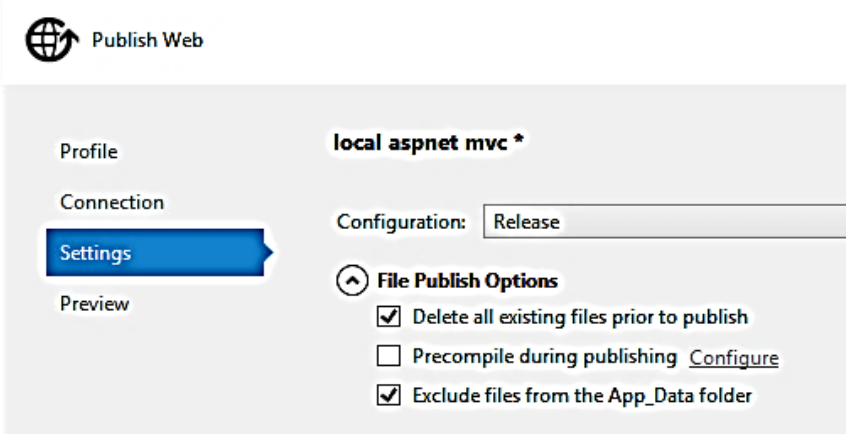


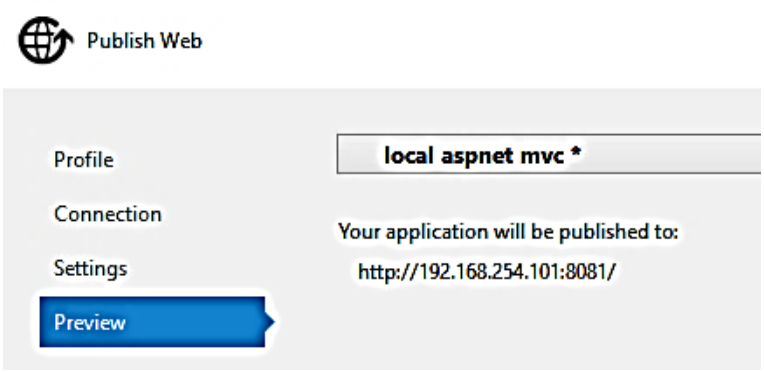

New Custom Profile
Profile name:
local aspnet mvc
OK
Cancel


Publish Web
Publish Web
Profile
Connection
Settings
Preview
local aspnet mvc *
Publish method:
File System
Target location:
e.g. http://RemoteServer/MyApp
...


Target Location
File System
Local IIS
Local Internet Information Server
Select the Web site you want to open.
IIS Sites
Default Web Site
mvcproject1
IIS Express Sites
WebSite1

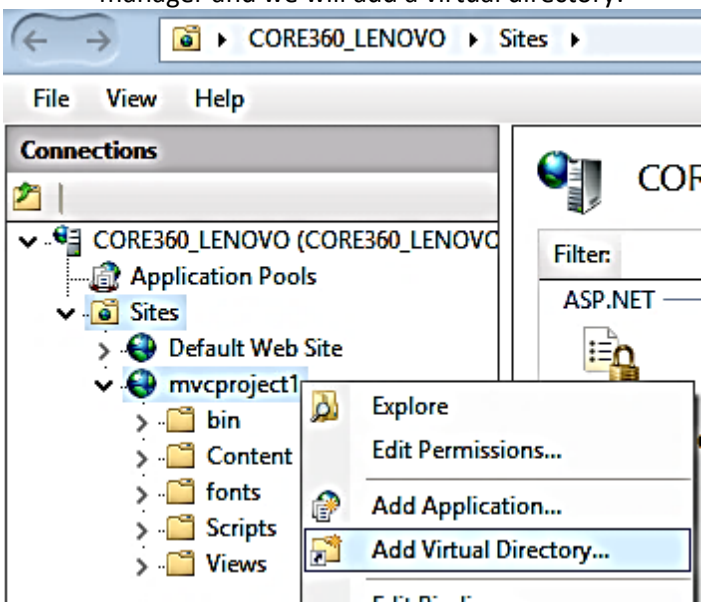

Publish Web
Profile
Connection
Settings
Preview
local aspnet mvc *
Publish method:
File System
Target location:
http://192.168.254.101:8081/
...

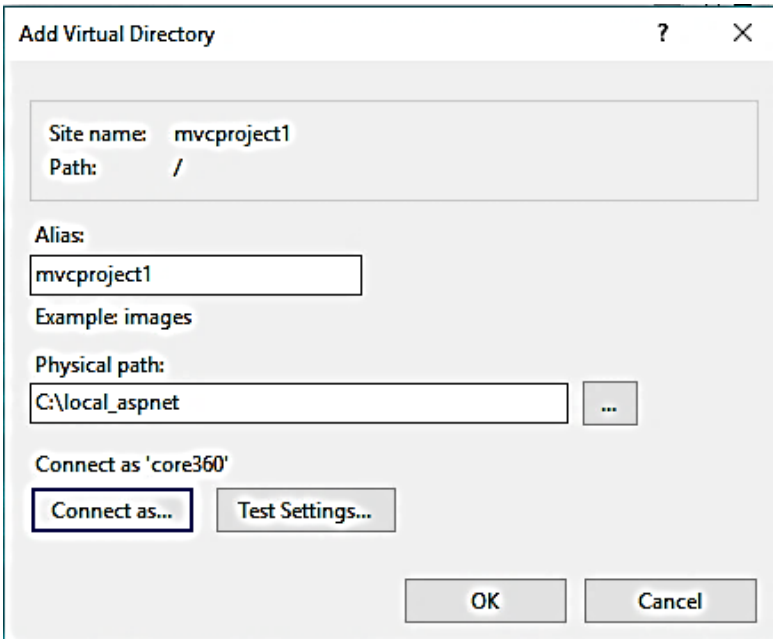




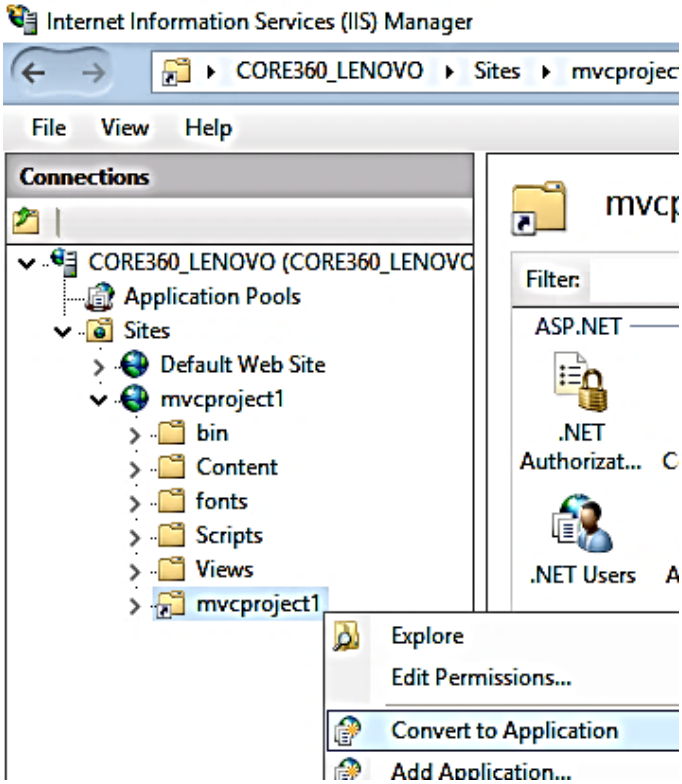
Then click publish.

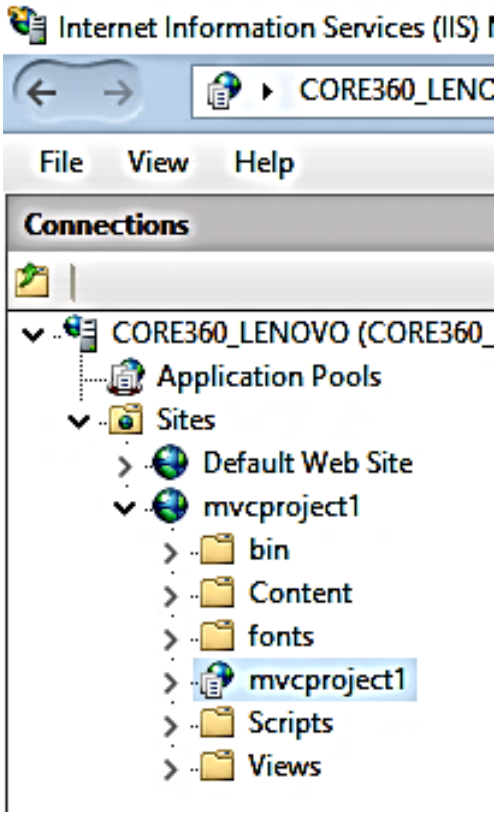
8. Check IIS manager if the publish is successful and the folders have been exported. Now rclick your web site in IIS manager and we will add a virtual directory:



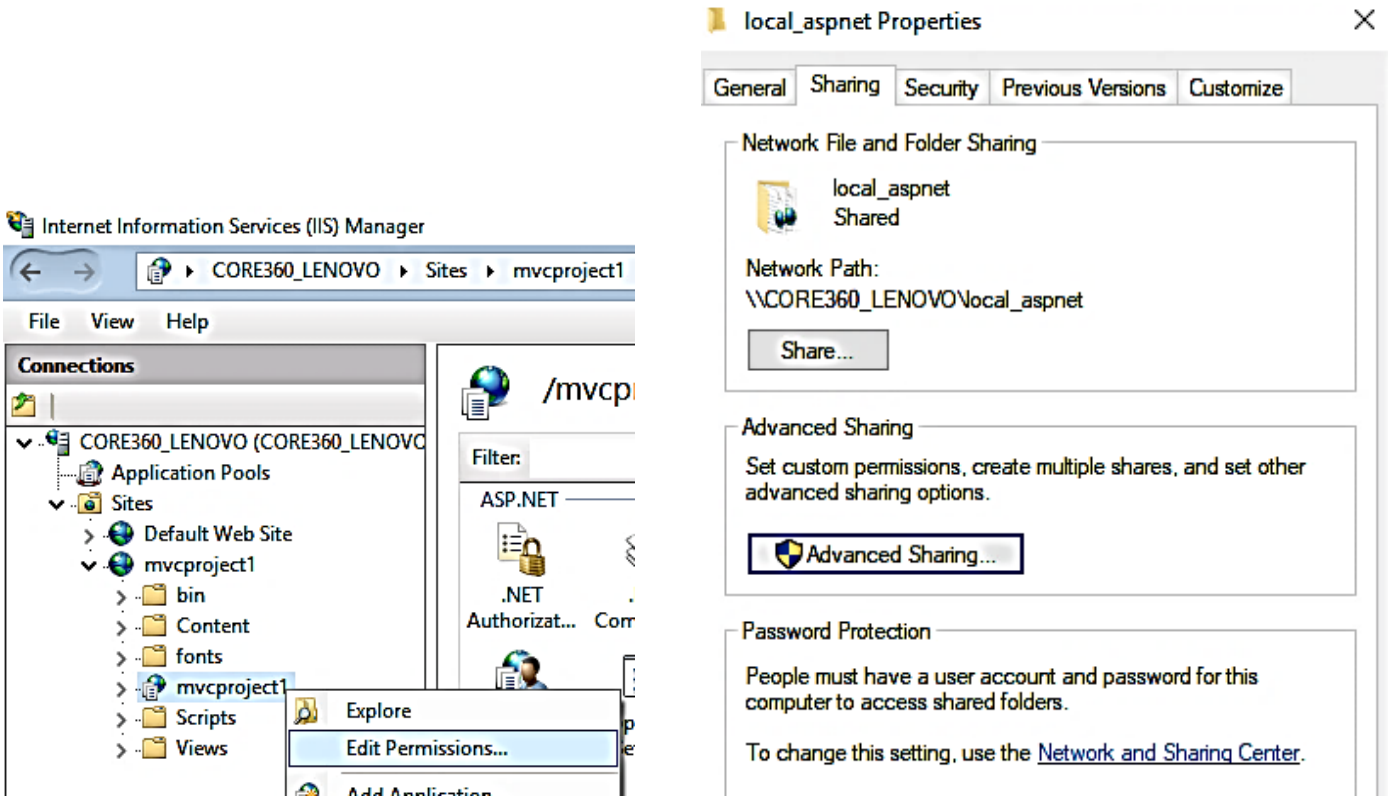


Then convert your virtual directory to an application

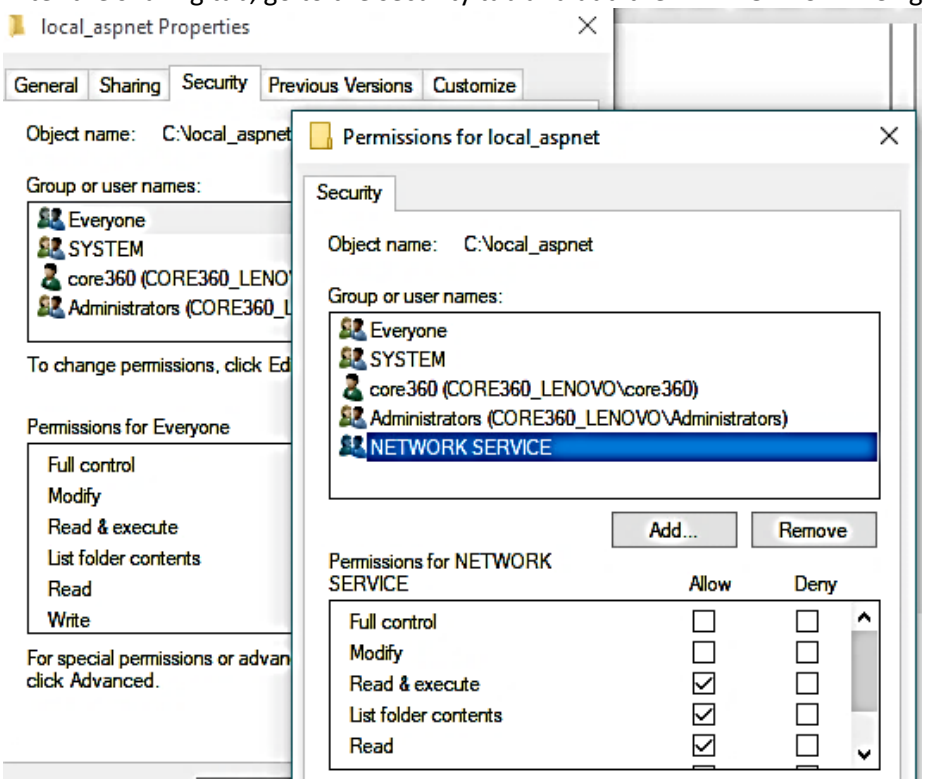




9. Edit permissions for your virtual directory:



After the sharing tab, go to the security tab and add the NETWORK SERVICE group:



10. We can then test our application