Chapter 2

Getting Started with ASP.NET MVC

Getting Started with ASP.NET MVC

Objectives

After completing this unit you will be able to:

- **Understand how ASP.NET MVC is used within Visual Studio.**
- **Create several versions of a simple ASP.NET MVC application.**
- **Understand how Views are rendered.**
- **Use the Razor view engine in ASP.NET MVC 5.**
- **Understand how dynamic output works.**
- **Pass input data to an MVC application in a query string.**

An ASP.NET MVC 5 Testbed

- **This courses uses the following software:**
  - Visual Studio 2019. The course was tested using the free Visual Studio Community 2019.
  - During installation you should install the following workloads: .NET desktop development and ASP.NET and web development.
  - This includes bundled ASP.NET MVC 5.
  - SQL Server Express 2016 LocalDB, which also comes bundled with Visual Studio 2019. But you need to explicitly choose to install it.
- **Recommended operating system is Windows 7 SP1, which is what was used in developing this course.**
- **If you want to practice deployment on IIS, you should also have IIS installed.**
  - See Appendix B.

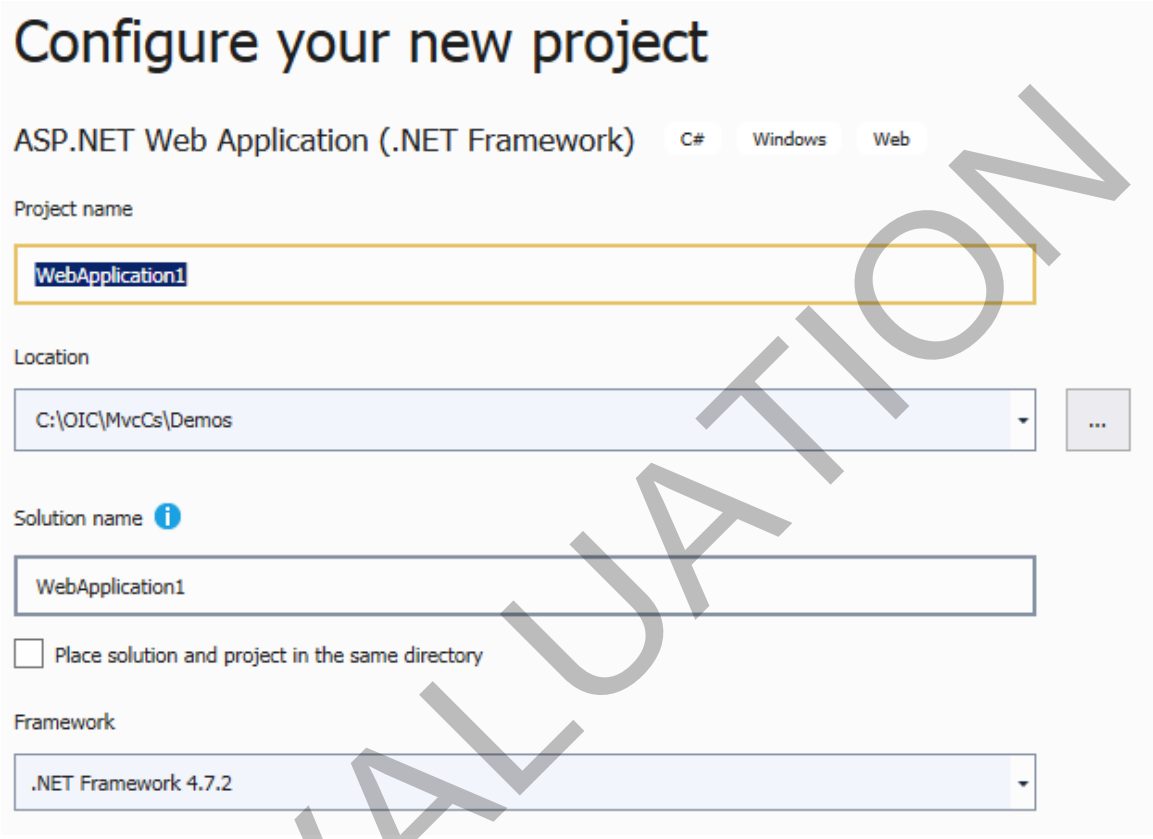
Visual Studio ASP.NET MVC Demo

- **Let's use Visual Studio to create an ASP.NET MVC 5 Web Application project.**

1. From the start window choose Create a new project.
2. In the next window choose ASP.NET Web Application (.NET Framework). You may filter by C# and Web project type. Then click Next.

Configure Your Project

3. Browse to the **C:\OIC\MvCs\Demos** folder, and leave the Project name as WebApplication1. Leave the Framework as .NET Framework 4.7.2.

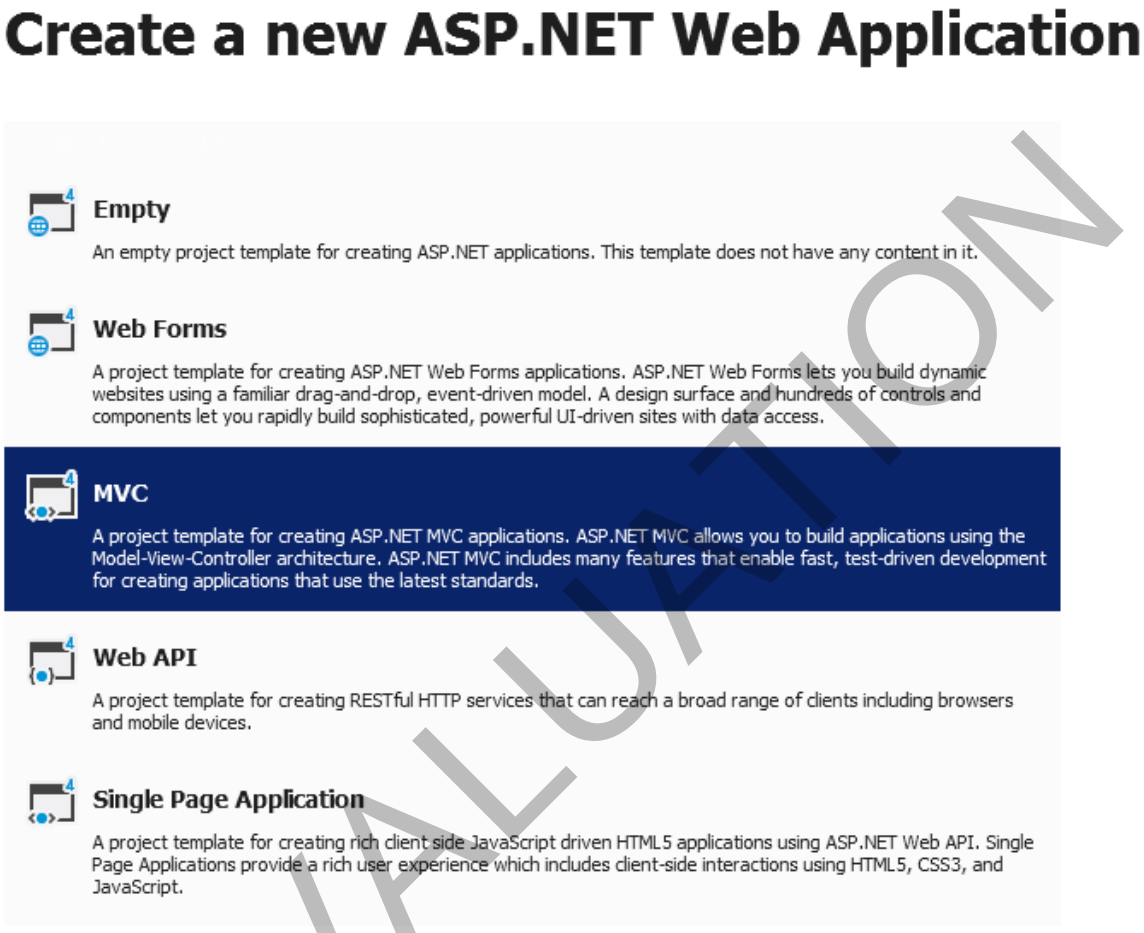


4. Click Create.

   – You will be able to choose on the next window whether to create a Web Forms, MVC, or Web API project.

ASP.NET MVC Demo (Cont'd)

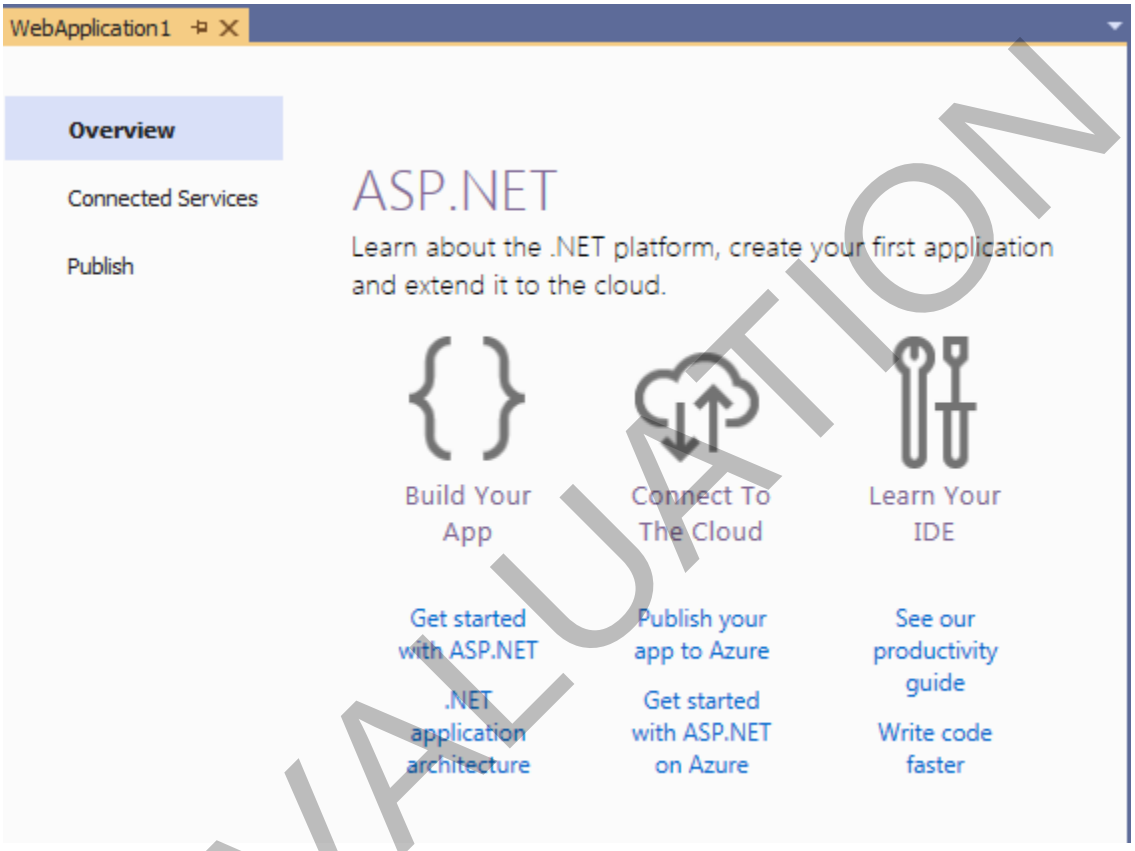
5. Choose MVC, and leave the checkboxes on right of window at their default settings.



6. Click Create.

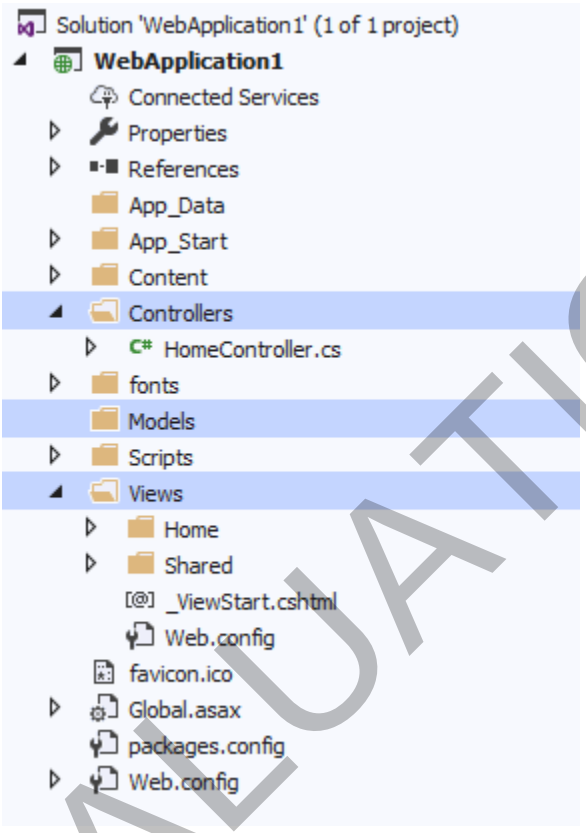
ASP.NET Documentation Page

- **You will see an ASP.NET documentation page displayed.**
  - There are links to various resources.



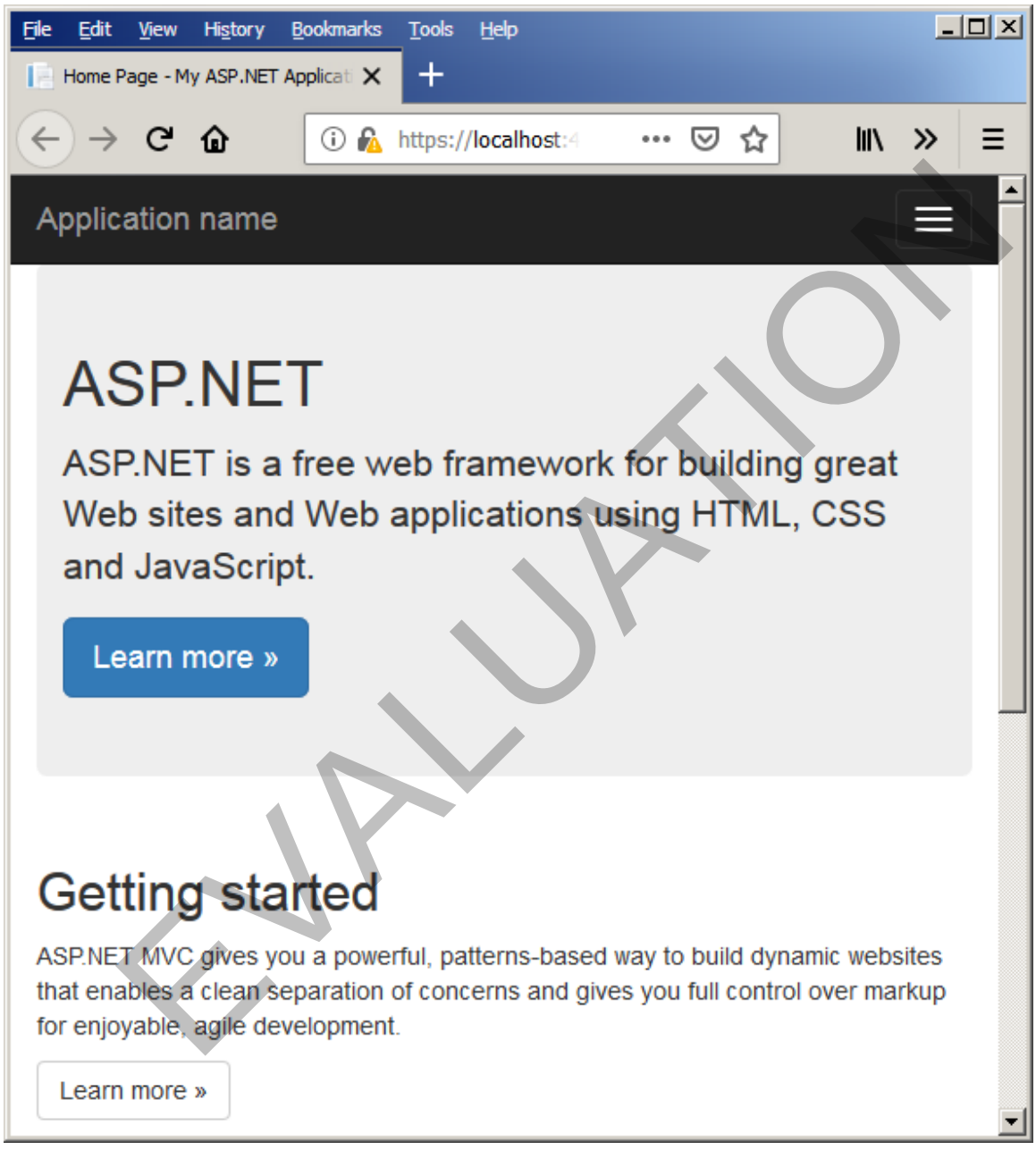
Starter Application

- **Notice that there are separate folders for Controllers, Models and Views.**



Starter Application (Cont'd)

- **Build and run this starter application[1]:**

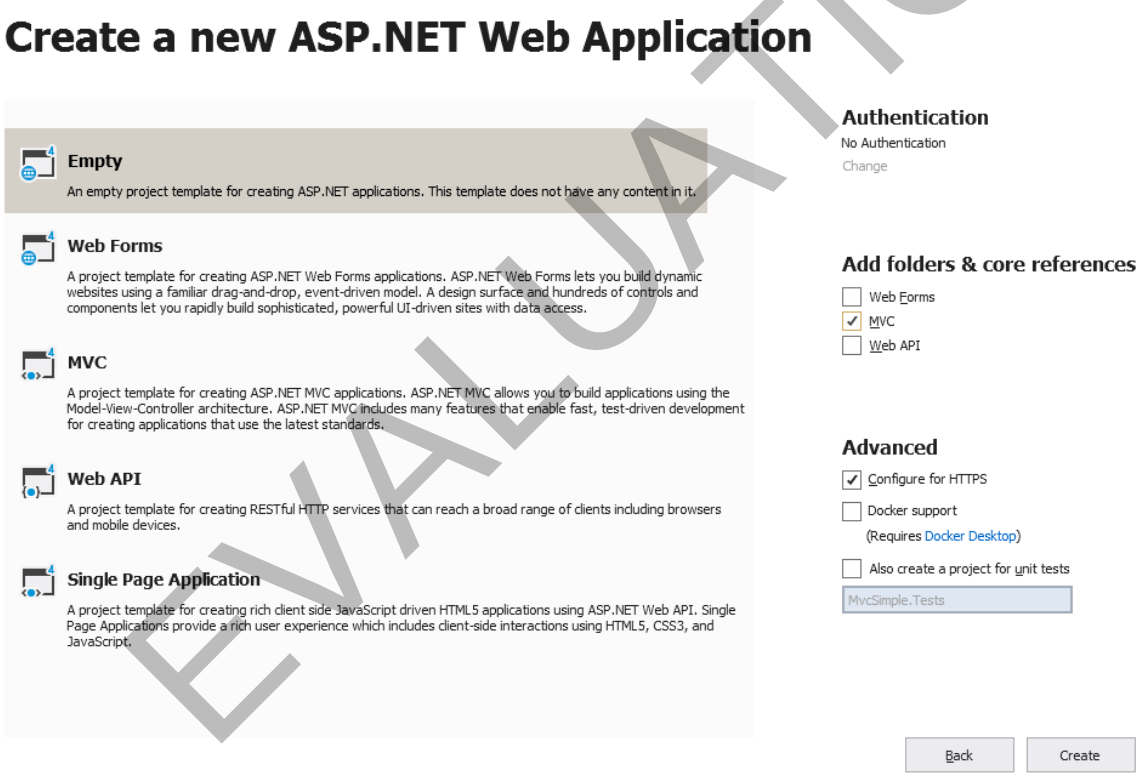


[1] Visual Studio will automatically start your default browser to run the application. In our screenshots you will sometimes see Firefox and sometimes Internet Explorer. You may see a security warning. Accept the risk and proceed.

Simple App with Controller Only

- **To start learning how ASP.NET MVC works, let's create a simple app with only a controller.**

1. Create a new ASP.NET Web Application project with the name **MvcSimple** in the **Demos** folder.
2. This time choose the Empty project template.
3. Check MVC in "Add folders and core references". Note that the same project can include any combination of Web Forms, MVC and Web API.

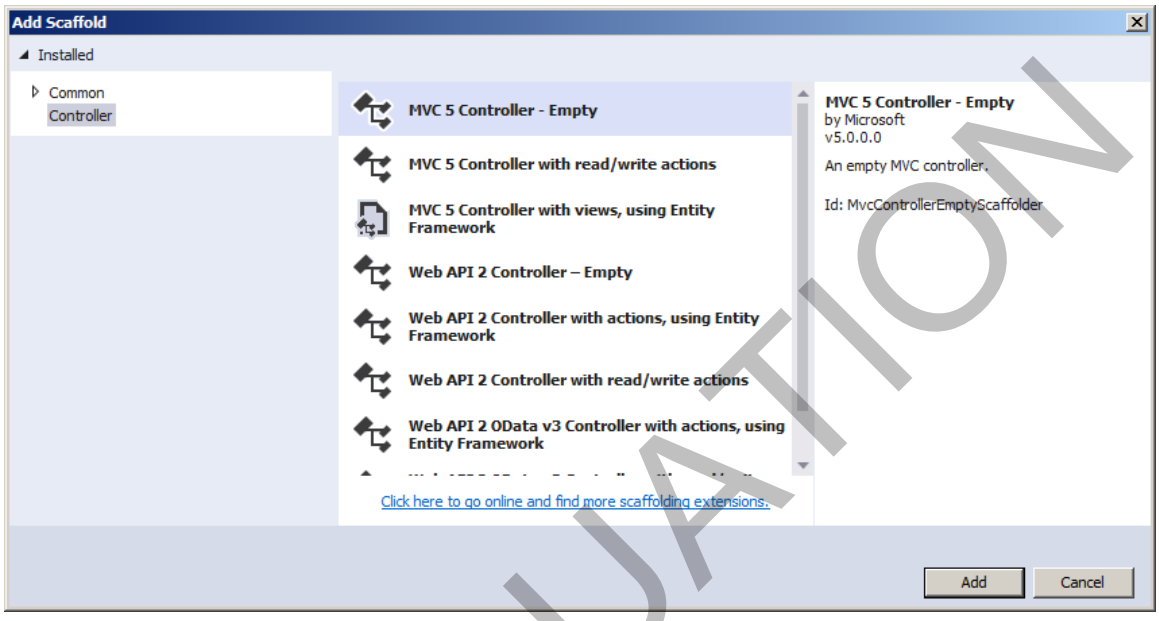


4. Click Create.

Demo: Controller Only (Cont'd)

5. Right-click over the Controllers folder and choose Add | Controller from the context menu.

6. Choose MVC 5 Controller - Empty for the scaffold.



7. Click Add.

8. Provide the name **HomeController**

9. Click Add.

Demo: Controller Only (Cont'd)

10. Examine the generated code **HomeController.cs**.

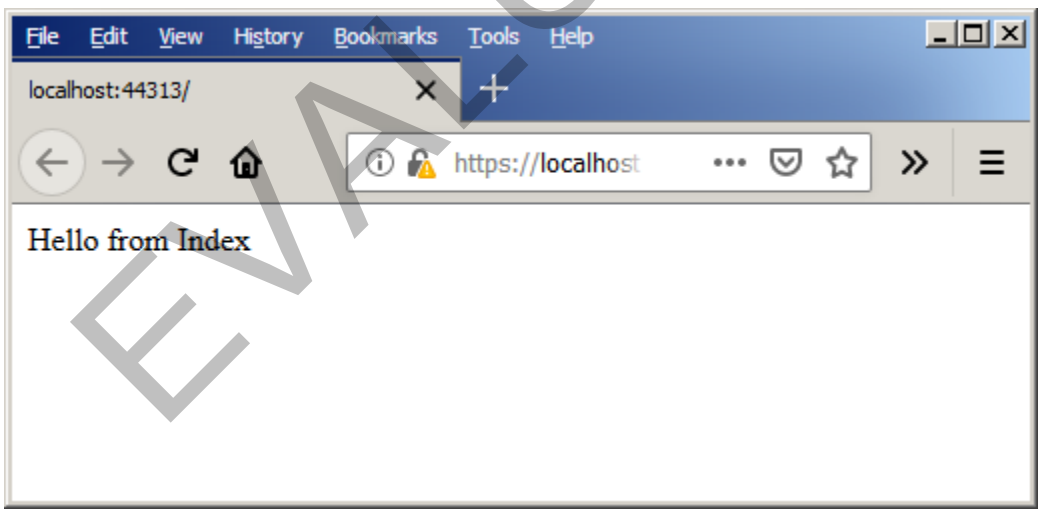
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcSimple.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/
        public ActionResult Index()
        {
            return View();
        }

    }
}
```

Demo: Controller Only (Cont'd)

11. Replace the code for the **Index()** method by the following. Also, provide a similar **Foo()** method.

```
public class HomeController : Controller
{
   // GET: /Home/
   public string Index()
   {
      return "Hello from Index";
   }

   public string Foo()
   {
      return "Hello from Foo";
   }
}
```

12. Build and run. If your browser warns about security, accept the risk and continue.



13. Examine the URL Visual Studio used to invoke the application. (The port number varies.)

```
https://localhost:44313/
```

Demo: Controller Only (Cont'd)

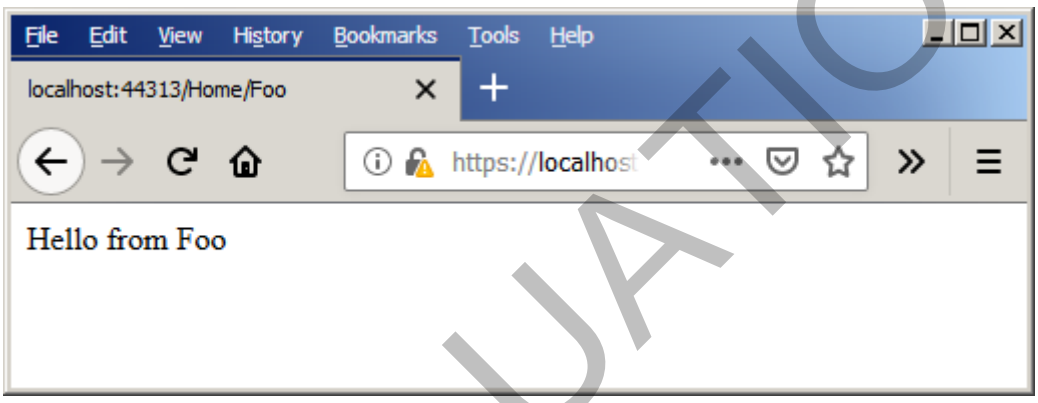
14. Now try using these URLs[2]. You should get the same result.

```
http://localhost:44313/Home/
http://localhost:44313/Home/Index/
```

15. Now try this URL.

```
http://localhost:44313/Home/Foo/
```

You will see the second method **Foo()** invoked:



16. Finally, let's add a second controller **SecondController.cs**.

17. Provide the following code for the **Index()** method of the second controller.

```
public class SecondController : Controller
{
    // GET: /Second/
    public string Index()
    {
        return "Hello from second controller";
    }
}
```

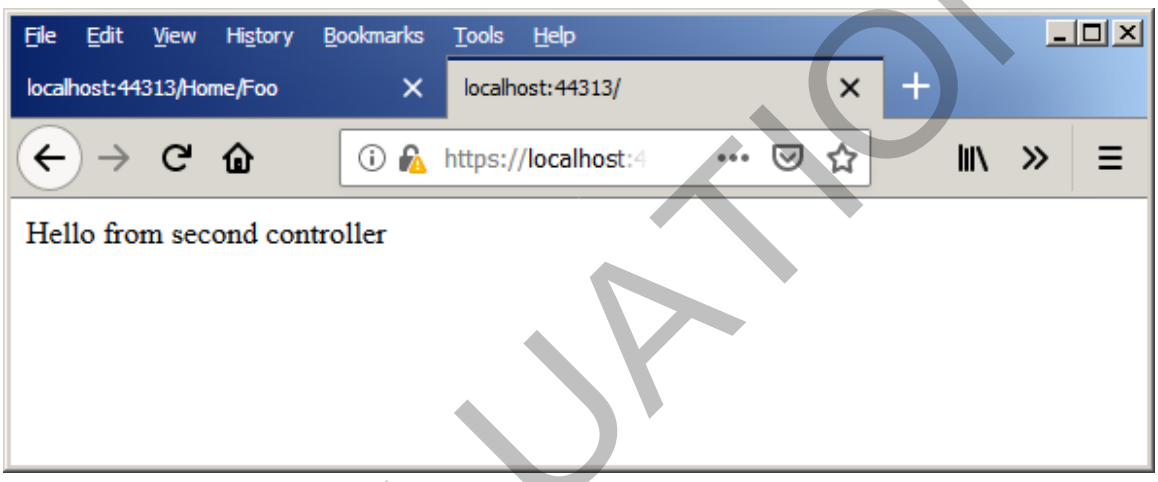
[2] The trailing forward slash in these URLs is optional.

Demo: Controller Only (Cont'd)

18. You can invoke this second controller using either of these URLs:

```
http://localhost:44313/Second/
http://localhost:44313/Second/Index/
```

In either case we get the following result. The program at this point is saved in **MvcSimple\Controller** in the chapter folder[3].



[3] You should open all the ASP.NET MVC examples as projects, not web sites.

Action Methods and Routing

- **Every public method in a controller is an *action method*.**
  - This means that the method can be invoked by some URL.
- **The ASP.NET MVC routing mechanism determines how each URL is mapped onto particular controllers and actions.**
- **The default routing is specified in the file *RouteConfig.cs*, contained in the *App_Start* folder.**

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

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

- **If desired, additional route maps can be set up here.**

Action Method Return Type

- **An action method normally returns a result of type *ActionResult.***
  - An action method can return any type, such as **string**, **int**, and so on, but then the return value is wrapped in an **ActionResult**.
- **The most common action of an action method is to call the *View()* helper method, which returns a result of type *ViewResult,* which derives from *ActionResult.***
- **The table shows some of the important action result types, which all derive from *ActionResult.***

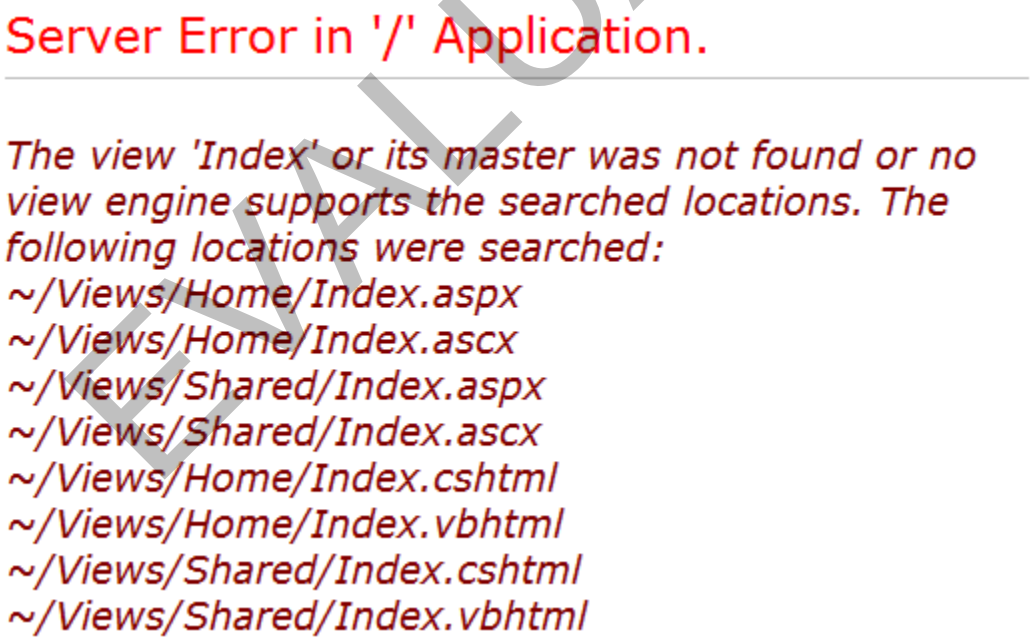
Action Result	Helper Method	Description
ViewResult	View()	Renders a view as a Web page, typically HTML
RedirectResult	Redirect()	Redirects to another action method using its URL
JsonResult	Json()	Returns a serialized Json object
FileResult	File()	Returns binary data to write to the response

Rendering a View

- **Our primitive controllers simply returned a text string to the browser.**
- **Normally, you will want an HTML page returned. This is done by rendering a *view*.**
  - The controller will return a **ViewResult** using the helper method **View()**.

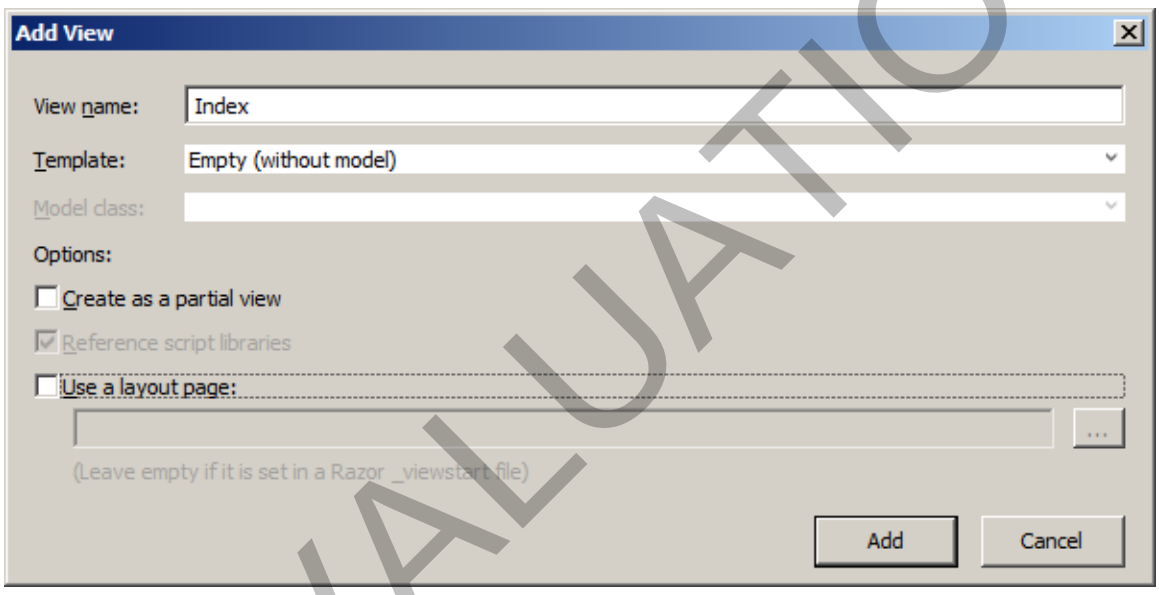
```
public ViewResult Index()
{
   return View();
}
```

- **Try doing this in the *MvcSimple* program. Build and run. It compiles but you get a runtime error.**



Creating a View in Visual Studio

- **The error message is quite informative!**
  - Let us create an appropriate file **Index.cshtml** in the folder **Views/Home**.
- **In Visual Studio you can create a view by right-clicking in the action method. Choose Add View.**
  - Clear the check box for layout page and click Add

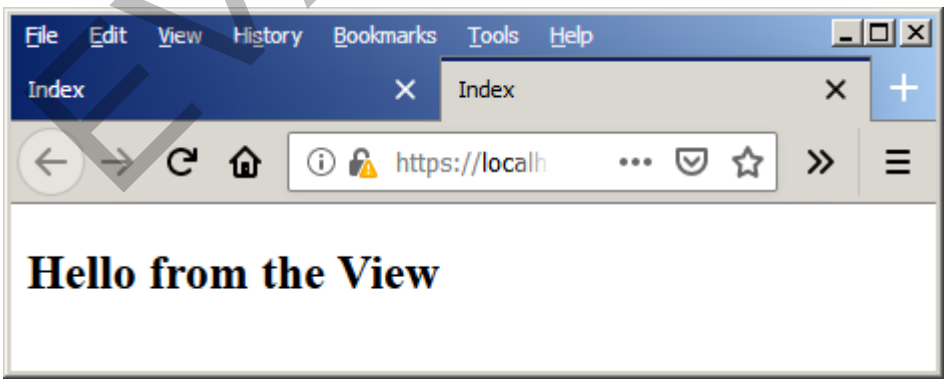


The View Web Page

- **A file *Index.cshtml* is created in the *Views\Home* folder.**
  - Edit this file to display a welcome message from the view. To make it stand out, use H2 format.

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport"
          content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <h2>Hello from the View</h2>
</body>
</html>
```

  - Build and run.



Dynamic Output

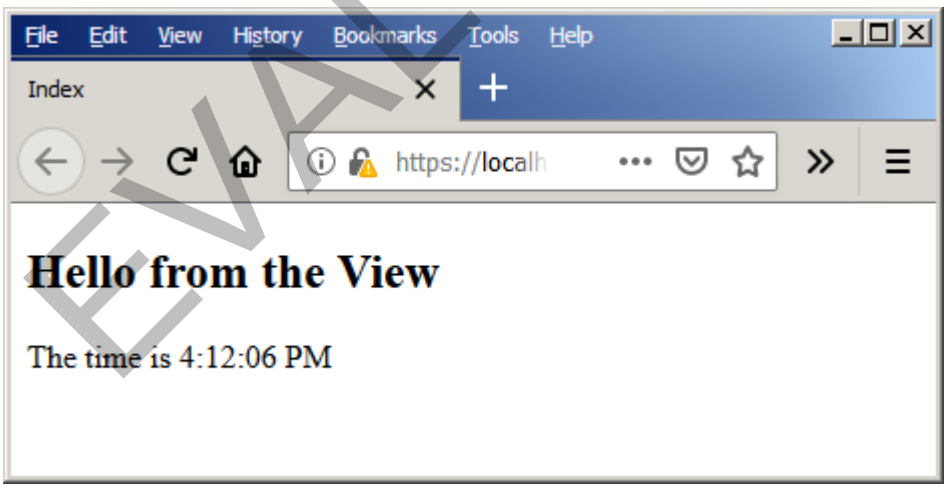
- ***ViewBag* is a dynamic type that can be used for passing data from the controller to the view, enabling the rendering of dynamic output.**
- **This code in the controller stores the current time.**

```
public ViewResult Index()
{
   ViewBag.Time =
      DateTime.Now.ToLongTimeString();
   return View();
}
```

- **This markup in the view page displays the data.**

```
<h2>Hello from the View</h2>
The time is @ViewBag.Time
```

- **Here is a run:**



  - The program is saved in **MvcSimple\View**.

Razor View Engine

- **From the beginning ASP.NET MVC has supported "view engines", which are pluggable components that implement different syntax options for view templates.**
- **In ASP.NET MVC 1 and 2 the default view engine is the Web Forms (or ASPX) view engine.**
- **In ASP.NET MVC 3 and 4 the default view engine is Razor.**
  - In creating a view, Visual Studio allowed you to choose whether to use ASPX or Razor.
- **Razor template syntax is much more concise than ASPX template syntax.**
  - You use @ in place of <%= ... %>
  - The Razor parser makes use of syntactic knowledge of C# code (in a .cshtml file) or of VB code (in a .vbhtml file).
- **In ASP.NET MVC 5 the Razor view engine is used automatically, and we will employ it in our examples.**

Embedded Scripts

- **Razor makes it easy to use embedded C# script in an HTML page. Simply enclose it with *@{}*.**

```
@{
     int day = 0;
     int gifts = 0;
     int total = 0;
     while (day < 12)
     {
         day += 1;
         gifts += day;
         total += gifts;
    }
 }
```

- **You can convert an object to a string and display it in HTML simply by using the @ symbol in front of it.**

```
<p>Total number of gifts = @total</p>
```

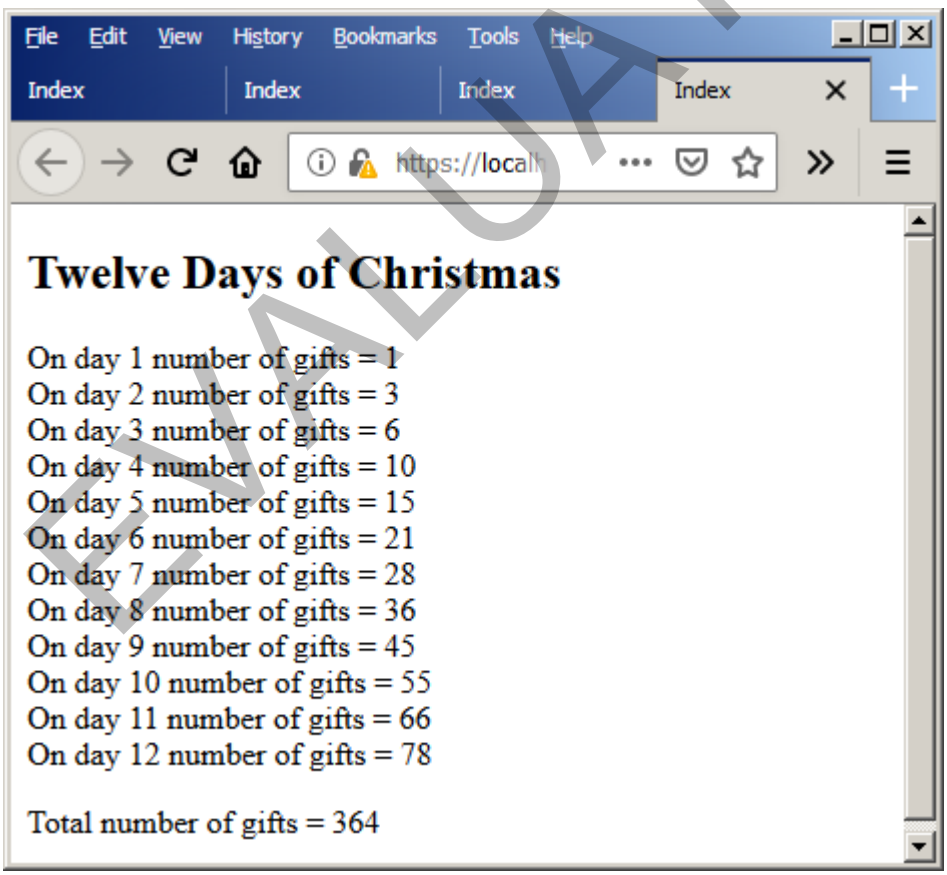
- **Inside an embedded script you can simply use HTML elements, giving you great flexibility in output.**
  - You can use literal text by prefacing it with **@:**.

```
@{
  ...
  while (day < 12)
  {
      day += 1;
      gifts += day;
      total += gifts;
      @:On day @day number of gifts = @gifts <br />
 }
}
```

Embedded Script Example

- **See *MvcSimple\Script.***

```
@{
  int day = 0;
  int gifts = 0;
  int total = 0;
  while (day < 12)
  {
      day += 1;
      gifts += day;
      total += gifts;
      @:On day @day number of gifts = @gifts <br />
 }
}
```



Using a Model with ViewBag

- **Our next version of the program uses a model along with the ViewBag.**
  - See **MvcSimple\ModelViewBag** in the chapter folder.
- **The model contains a class defining a *Person*.**
  - See the file **Person.cs** in the **Models** folder of the project.
  - There are public properties **Name** and **Age**.
  - Unless otherwise assigned, **Name** is "John" and **Age** is 33.

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

namespace MvcSimple.Models
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
        public Person()
        {
            Name = "John";
            Age = 33;
        }
    }
}
```

Controller Using Model and ViewBag

- **The controller instantiates a *Person* object and passes it in *ViewBag*.**
  - Note that we need to import the **MvcSimple.Models** namespace.

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

namespace MvcSimple.Controllers
{
   public class HomeController : Controller
   {
      // GET: /Home/
      public ViewResult Index()
      {
         ViewBag.person = new Person();
         return View();
      }
   }
}
```

View Using Model and ViewBag

- **The view displays the output using appropriate script.**
  - Again we need to import the **MvcSimple.Models** namespace.

```
@{
    Layout = null;
}
@using MvcSimple.Models;
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-
width" />
    <title>Index</title>
</head>
<body>
    @{ Person p = ViewBag.person;}
    <h2>Using model data:</h2>
    <p>Name = @p.Name</p>
    <p>Age = @p.Age </p>
</body>
```

  - The output:

Using Model Directly

- **You may pass a single model object to a view through the use of an overloaded constructor of the View() method.**
  - For an example see **MvcSimple\Model**.
- **To see how this works, first rewrite the controller.**

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

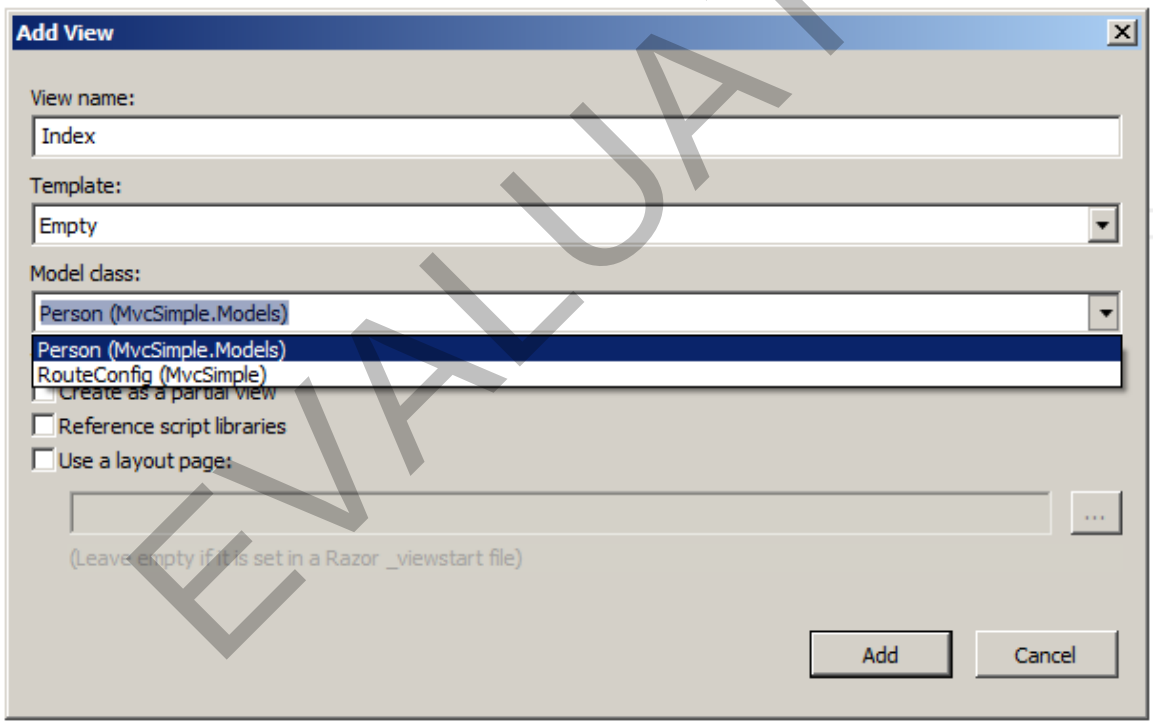
  - The parameter to the overload of the **View()** method is a model object.
- **Next, rewrite the view page.**

```
@model MvcSimple.Models.Person

...

<body>
    <h2>Using model data:</h2>
    <p>Name = @Model.Name</p>
    <p>Age = @Model.Age </p>
</body>
</html>
```

  - The **Person** object is passed as a parameter to the view, and the model object can be accessed through the variable **Model**.
  - We no longer need the script code.

A View Using Model in Visual Studio

- **To create a view using a Model in Visual Studio, right-click inside an action method and choose Add View from the context menu.**
- **You may create a view tied to the model by selecting a model from the dropdown.**
  - You should build the application first in order that the dropdown be populated.
  - Select the Empty template, rather than the Empty (without model) template.



  - You can demonstrate this for yourself by deleting the view in the **MvcSimple\Model** example.

View Created by Visual Studio

- **Here is the view page created by Visual Studio:**

```
@model MvcSimple.Models.Person

@{
    Layout = null;
}

<!DOCTYPE html>

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

Passing Parameters in Query String

- **In MVC applications you will typically need to handle input data in one manner or another.**
- **A simple way to pass input data is through the query string on the URL that invokes the application.**
- **For an example, see the *MvcHello* application in the chapter folder.**
  - Pass the name in the query string, for example:

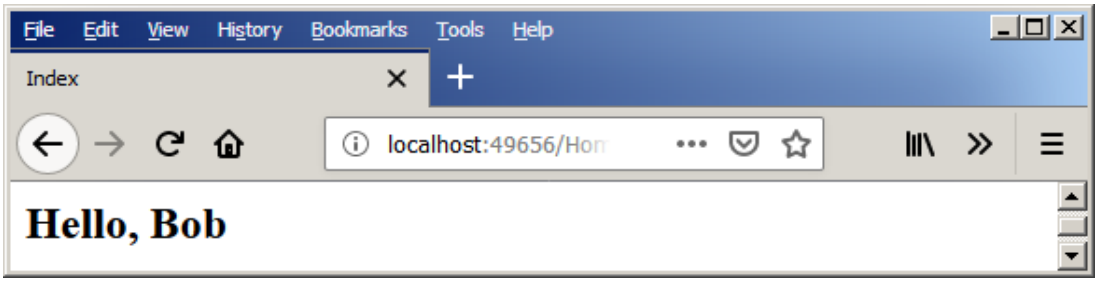
```
/Home/Index?name=Bob
```

  - The Index action method in the home controller takes name as a parameter, which is stored in the ViewBag.

```
// GET: /Home/Index?name=x
public ActionResult Index(string name)
{
    ViewBag.Name = name;
    return View();
}
```

  - The view displays a greeting using the name.

```
<body>
    <h2>Hello, @ViewBag.Name</h2>
</body>
```



Lab 2

Contact Manager Application

In this lab you will implement an ASP.NET MVC application that creates a contact and displays it on the page. The contact can be changed by passing the first and last names in the query string. The model persists the contact in a flat file.

Detailed instructions are contained in the Lab 2 write-up at the end of the chapter.

Suggested time: 30 minutes

Summary

- **You can begin creating an ASP.NET MVC application with the controller, which handles various URL requests.**
- **From an action method of a controller you can create a view using Visual Studio.**
- **ASP.NET MVC 5 uses the Razor view engine.**
- **You can pass data from the controller to the view by using the *ViewBag*.**
- **By creating a model you can encapsulate the business data and logic.**
- **You can pass data to an MVC application in query string.**

Lab 2

Contact Manager Application

Introduction

In this lab you will implement an ASP.NET MVC application that creates a contact and displays it on the page. The contact can be changed by passing the first and last names in the query string. The model persists the contact in a flat file.

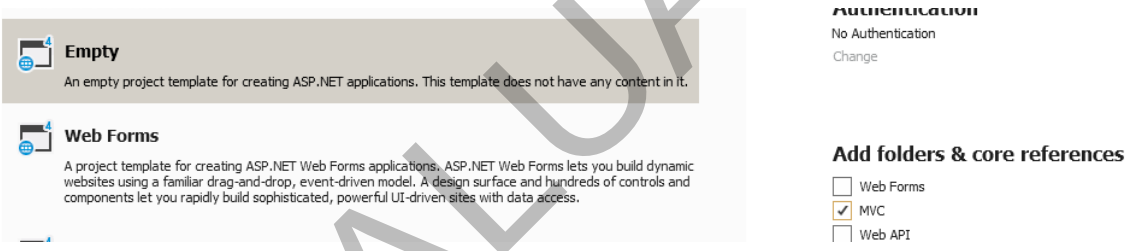
Suggested Time: 30 minutes

Root Directory: **C:\OIC\MvcCs**

Directories:	**Labs\Lab2**	(do your work here)
	Labs\Lab2\Contact.cs	(code for model)
	Chap02\MvcContact	(solution)

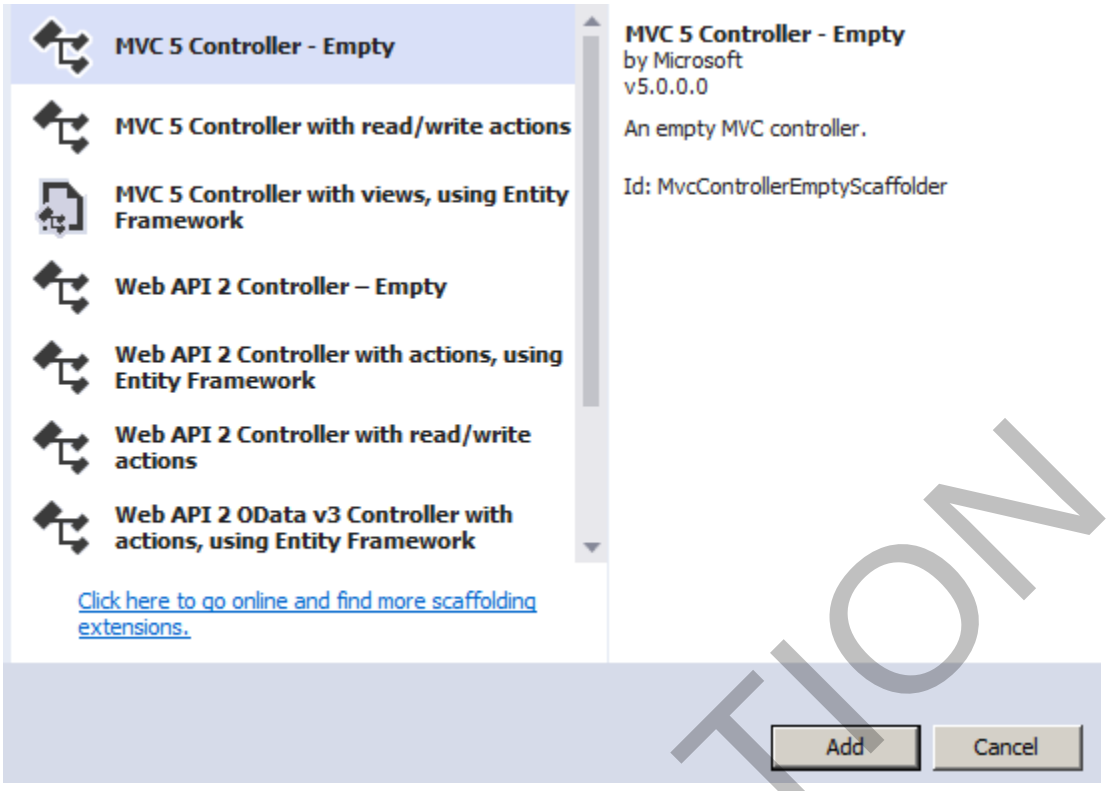
Instructions

1. Create a new ASP.NET Empty Web Application **MvcContact** in the working directory. Add folders and core references for MVC.

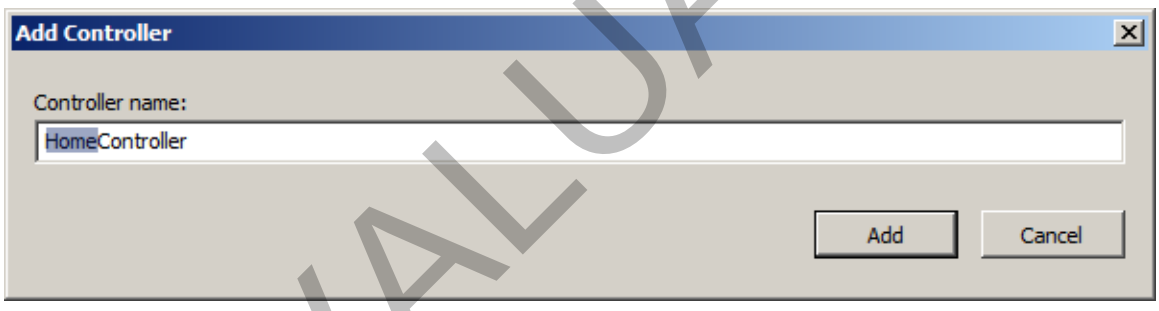


2. Copy the file **Contact.cs** defining a model class to the **Models** folder and add it to your new project. Examine the code. There are public properties **FirstName** and **LastName** and public static methods to read and write the contact to the flat file **contact.txt** in the **\OIC\Data** folder. A constructor initializes the contact to what is read in from the file.

3. Right-click over the Controllers folder and choose Add | Controller from the context menu. Select the MVC 5 Controller – Empty template and click Add.



4. Assign name HomeController and click Add.



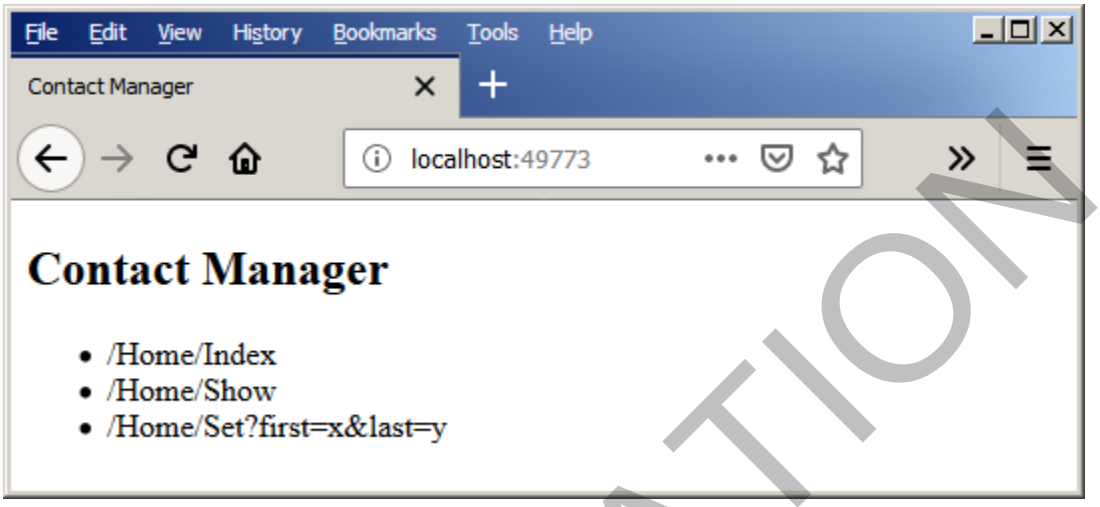
5. Add a view corresponding to the **Index()** action method. Use the suggested name Index and the Empty (without model) template. Do not use a layout page.

6. Make the title of the view “Contact Manager”. Provide HTML for a little help page consisting of an unordered list showing three URLs for invoking the application, corresponding to action methods Index, Show and Set. The latter takes a query string specifying first and last names.

```
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Contact Manager</title>
</head>
<body>
    <h2>Contact Manager</h2>
```

```
    <ul>
        <li>/Home/Index</li>
        <li>/Home/Show</li>
        <li>/Home/Set?first=x,last=y</li>
    </ul>
</body>
</html>
```

7. Build and run the application. You should see the help page displayed.



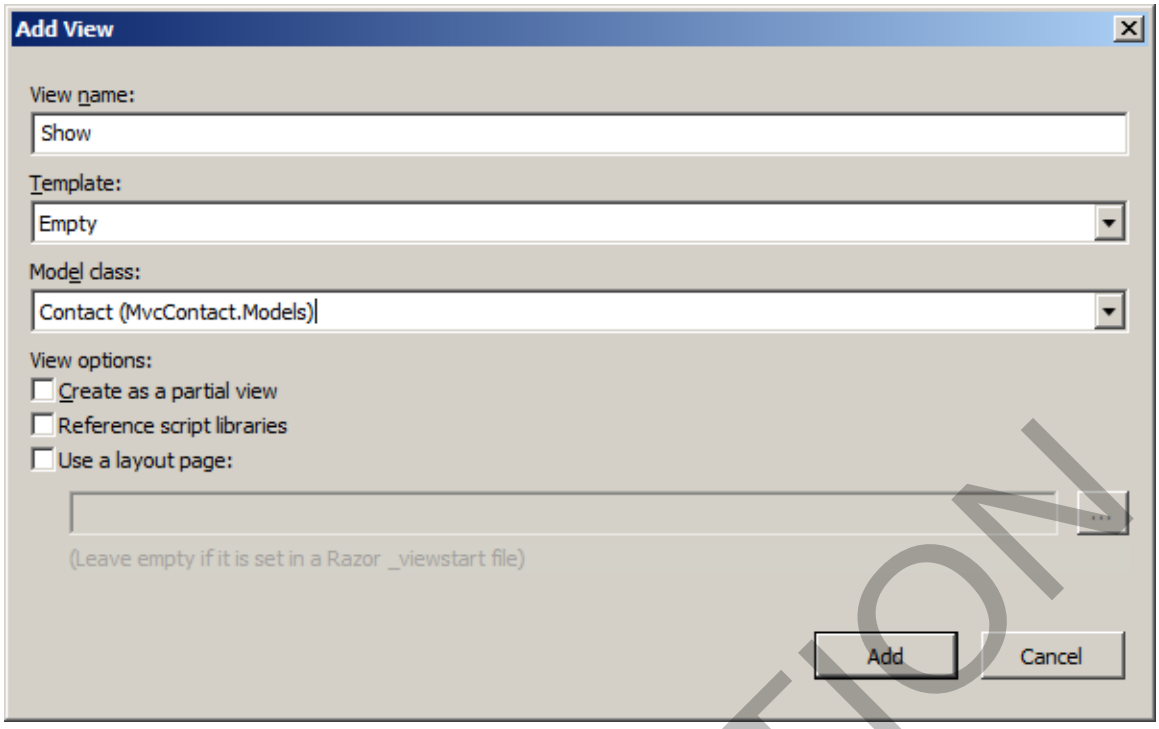
8. Provide a **Show()** action method. Use an override of **View()** that takes the name of the view as the first argument and an object as the second object. Use a new **Contact** as the object.

```
// GET: /Home/Show
public ActionResult Show()
{
    return View("Show", new Contact());
}
```

9. Import the namespace **MvcContact.Models** so that you can access the **Contact** class.

10. Build the project to make sure you get a clean compile and so that you can use the model when you create the view.

11. Right-click inside this new action method to add a view. Accept the suggested name Show. From the dropdown for Model class select the Contact class. See screen capture on the following page.



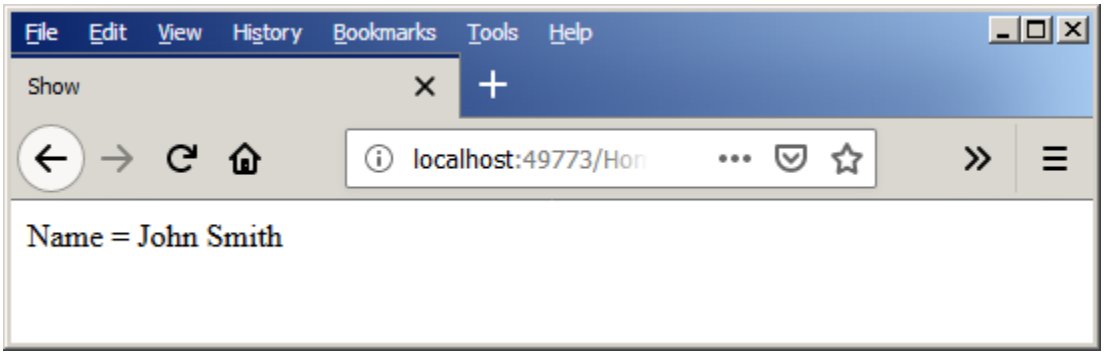
12. The scaffolding will have placed a **@model** directive at the top of the **.cshtml** file. Provide Razor HTML code to display the first and last names separated by a space.

```
@model MvcContact.Models.Contact

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Show</title>
</head>
<body>
    <div>
        Name = @Model.FirstName @Model.LastName
    </div>
</body>
</html>
```

13. Build and run. You should initially see the help page. Then modify the URL in the browser to invoke Show:

```
/Home/Show
```

You should see the default name that is stored in the file.



14. Next provide a third controller action method **Set()** which takes as parameters strings for the first and last name. As a comment, show the query string by which the parameters will be passed in the URL. The code should write the contact to the flat file and store the first and last names in the ViewBag.

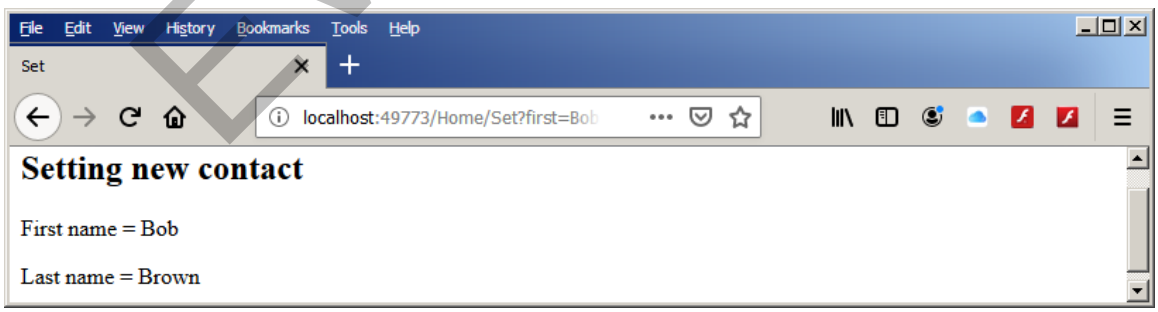
```
// GET: /Home/Set?first=x,last=y
public ActionResult Set(string first, string last)
{
   Contact.WriteContact(first, last);
   ViewBag.First = first;
   ViewBag.Last = last;
   return View();
}
```

15. Add a corresponding view, in which you display the parameters as stored in the ViewBag.

```
<body>
    <h2>Setting new contact</h2>
    <p>First name = @ViewBag.First</p>
    <p>Last name = @ViewBag.Last</p>
</body>
```

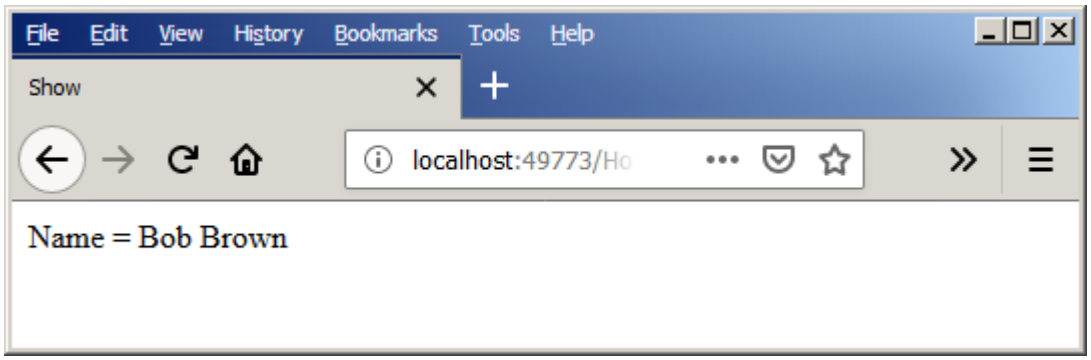
16. Build and run. Invoke Set, providing first and last names in the query string, for example:

```
/Home/Set?first=Bob&last=Brown
```



17. Finally, invoke Show again. You should see the new contact displayed.

```
/Home/Show
```



18. As a final adjustment to the project, set the project properties so that you will always display the home page when you run the application, not whatever view happens to be open in the editor. You can do this by setting the Start Action to a specific page, which will be Home/Index.