Chapter 2

Getting Started with ASP.NET Core MVC

Getting Started with ASP.NET Core MVC

Objectives

After completing this unit you will be able to:

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

An ASP.NET Core MVC Testbed

- **This courses uses the following software:**
  - Visual Studio 2019. The course was developed using the free Visual Studio Community 2019.
  - .NET Core 3.0, which is bundled with updated releases of Visual Studio 2019. (It was not available at the time of the initial release of Visual Studio 2019.)
  - SQL Server 2016 LocalDB, which comes bundled with Visual Studio 2019.
- **Required operating system is Windows 7 SP1 or higher.**

Visual Studio ASP.NET MVC Demo

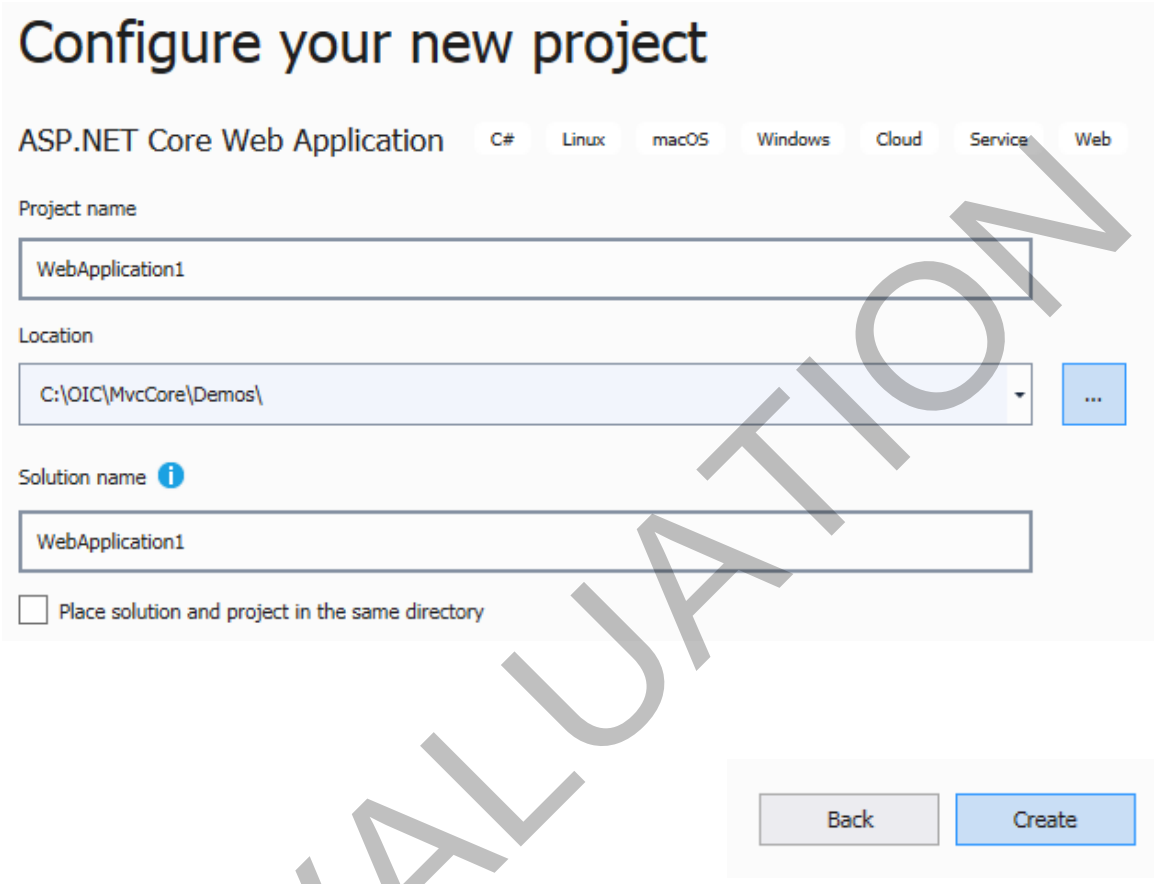
- **Let's use Visual Studio to create an ASP.NET Core MVC Web Application project.**

1. From the opening screen choose Create a new project.
2. From among the Web templates choose ASP.NET Core Web Application.

3. Click Next.

ASP.NET MVC Demo (Cont'd)

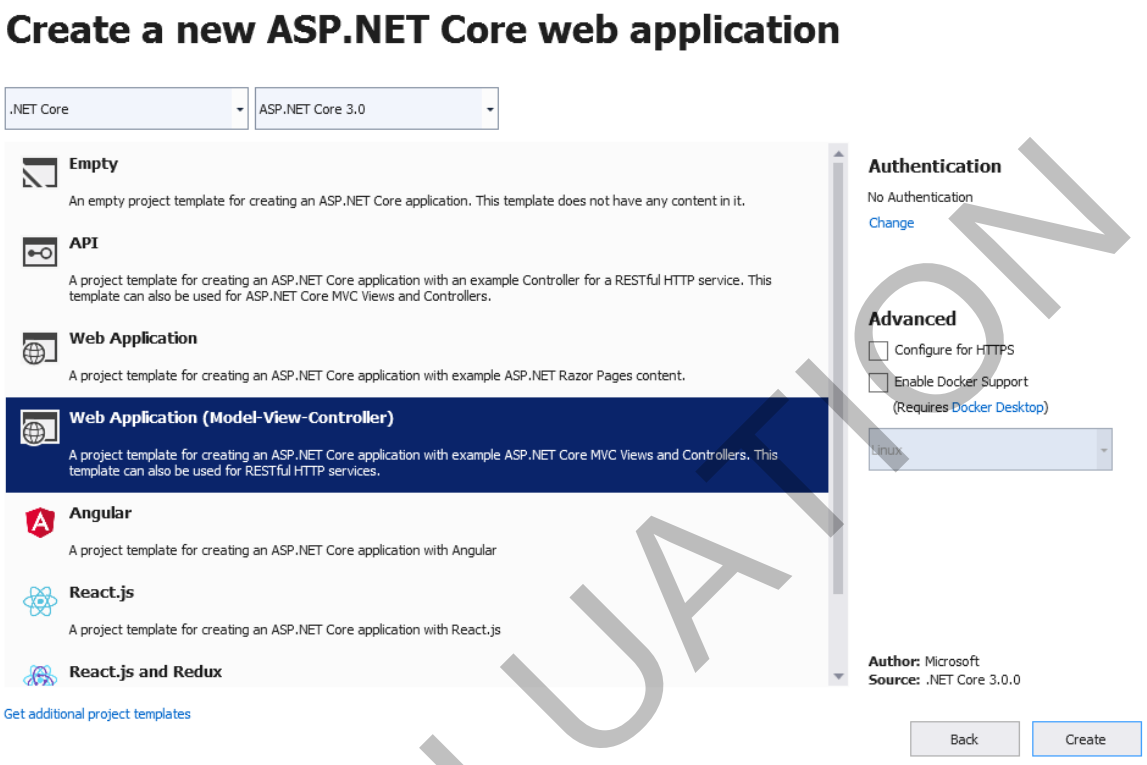
4. Browse to the **C:\OIC\MvcCore\Demos** folder for Location, and leave the Project name as WebApplication1.



5. Click Create.

ASP.NET MVC Demo (Cont'd)

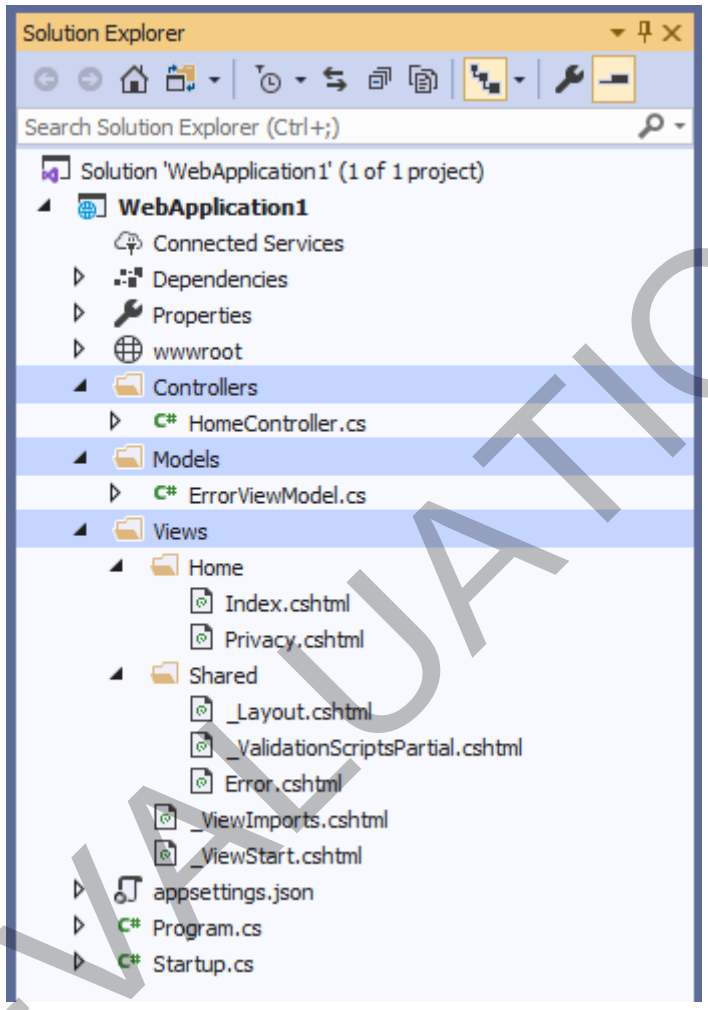
6. Choose Web Application (Model – View – Controller) and clear the Configure for HTTPS checkbox.



7. Click Create.

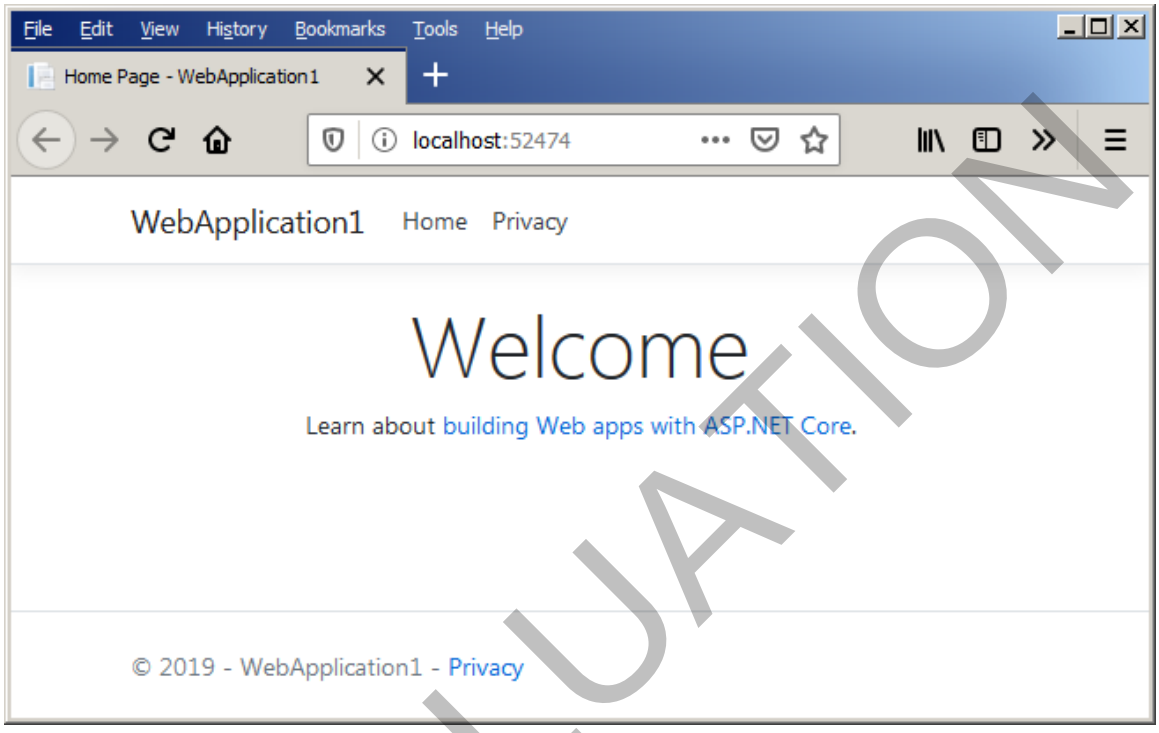
Starter Application

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



Starter Application (Cont'd)

- **Build and run this starter application.**
  - It will start in your default browser.



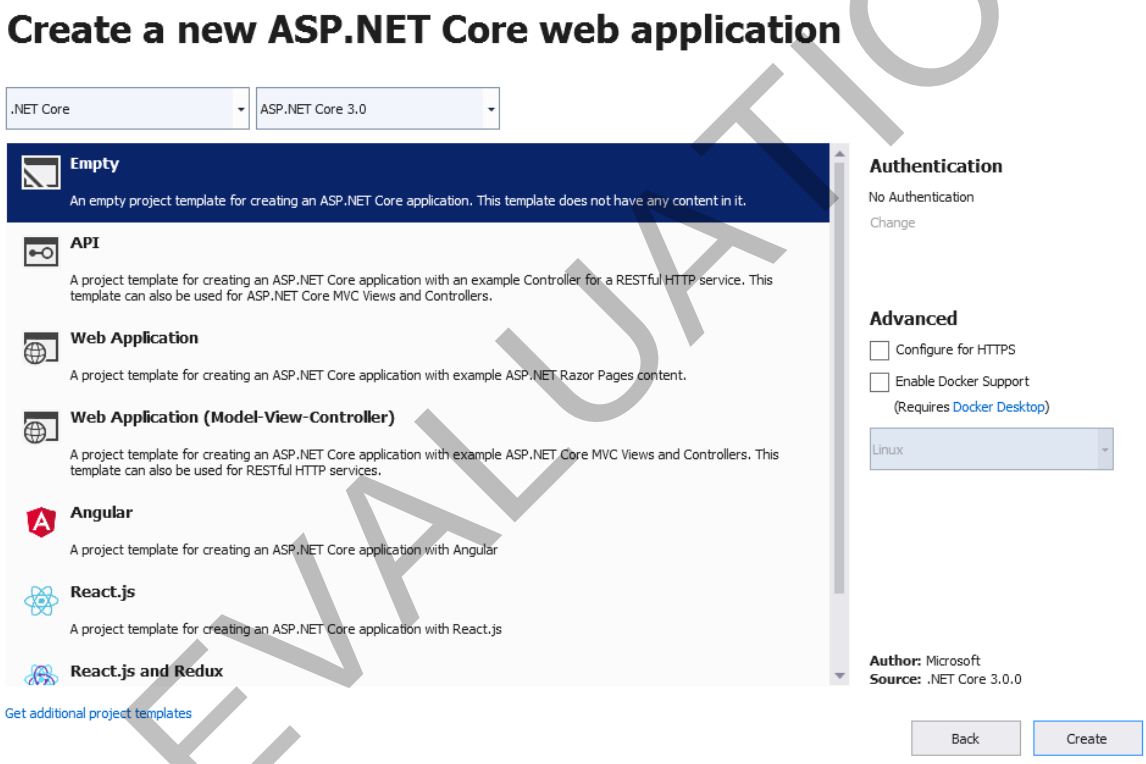
  - This starter application, built out of the box, actually has many features.
  - In particular, through use of the Bootstrap framework, it has responsive design and will display well on mobile devices as well as on the desktop.
  - To test you could deploy to Azure. See Chapter 9.

Simple App with Controller Only

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

1. Create a new project within Visual Studio from the menu File | New | Project ...

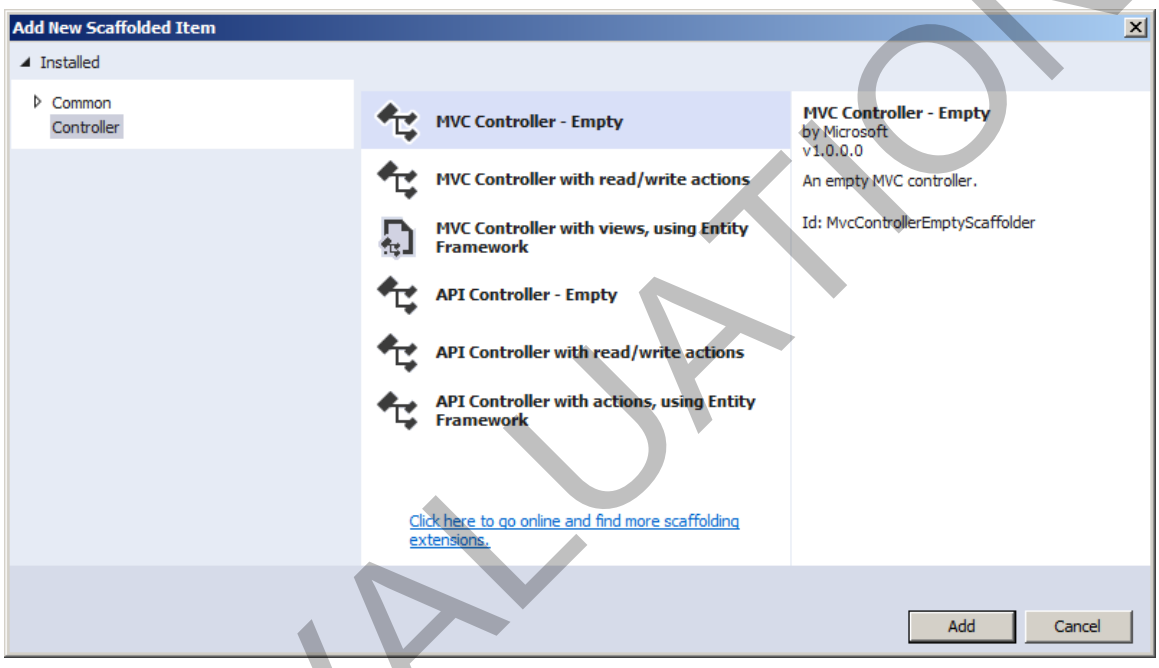
2. Proceed as before to create a new ASP.NET Core web application project with the name **MvcSimple** in the **Demos** folder. This time choose the Empty project template.



3. Click Create.

Demo: Controller Only (Cont'd)

4. Add a Controllers folder to the project.
5. Build the project.
6. Right-click over the Controllers folder and choose Add | Controller ... from the context menu.
7. Choose MVC Controller – Empty.



8. Click Add.
9. Specify **HomeController** as the name of the new controller.

10. Click Add.

Demo: Controller Only (Cont'd)

11. This will take a while, loading NuGet packages.

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

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

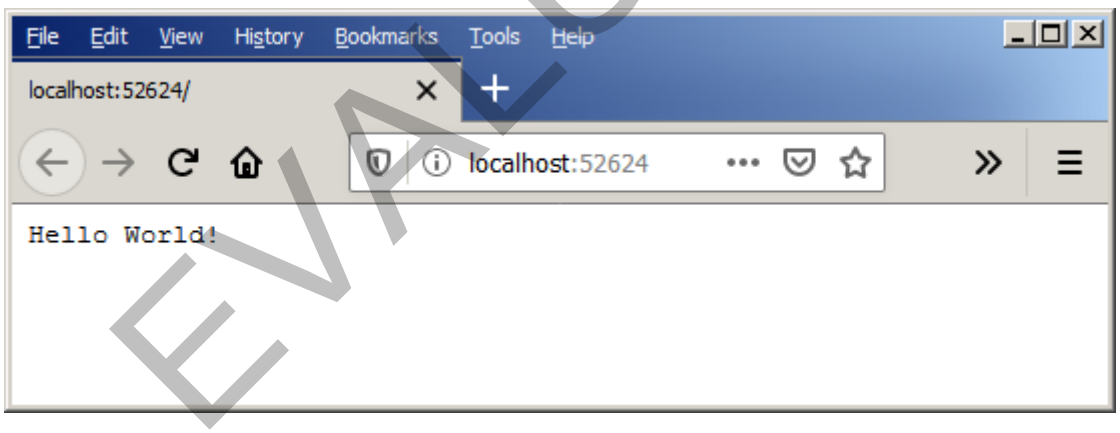
13. You may wish to remove the **using** statements for unused namespaces, shown not bolded.

Demo: Controller Only (Cont'd)

14. 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";
    }

    // GET: /Home/Foo
    public string Foo()
    {
        return "Hello from Foo";
    }
}
```

15. Build and run.



16. Bummer! We were expecting "Hello from Index". Where do you suppose the "Hello World!" came from?

Startup.cs

17. Examine the file Startup.cs.

```
public class Startup
{
  ...
  public void ConfigureServices(
     IServiceCollection services)
  {
  }

  ...
  public void Configure(IApplicationBuilder app,
     IWebHostingEnvironment env)
  {
      if (env.IsDevelopment())
      {
          app.UseDeveloperExceptionPage();
      }

      app.UseRouting();

      app.UseEndpoints(endpoints =>
      {
          endpoints.MapGet("/", async context =>
          {
              await
context.Response.WriteAsync("Hello World!");
          });
      });   }
}
```

18. Examine the file **Startup.cs** in the **WebApplication1** example. This will give us a clue for fixing our new example.

Edit Startup.cs

19. Add the highlighted code to the **ConfigureServices()** and **Configure()** methods.

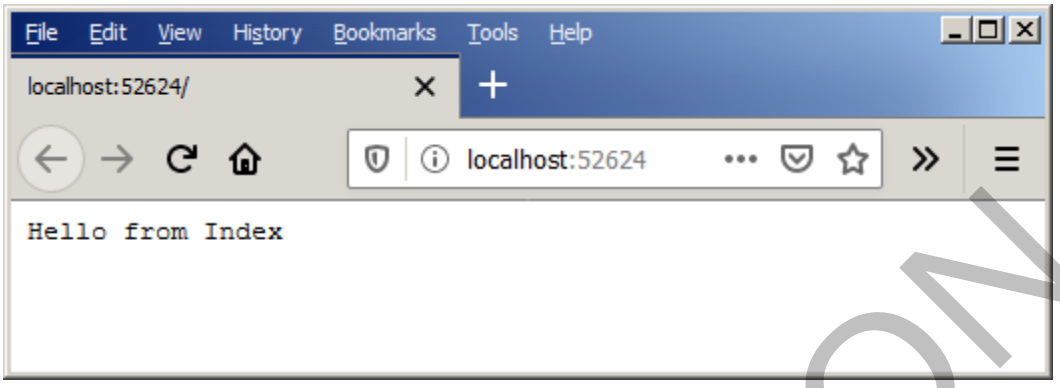
```
public void ConfigureServices(
                      IServiceCollection services)
{
   services.AddControllersWithViews();
}

public void Configure(IApplicationBuilder app,
                      IWebHostEnvironment env)
{
   if (env.IsDevelopment())
   {
       app.UseDeveloperExceptionPage();
   }
   app.UseStaticFiles();

   app.UseRouting();

   app.UseEndpoints(endpoints =>
   {
       endpoints.MapControllerRoute(
           name: "default",
           pattern:
"{controller=Home}/{action=Index}/{id?}");
   });
}
```

– Although not needed in this example, we also copied from **WebApplication1** this line:

```
app.UseStaticFiles();
```

Demo: Controller Only (Cont'd)

20. Build and run. Success!



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

```
http://localhost:52624/
```

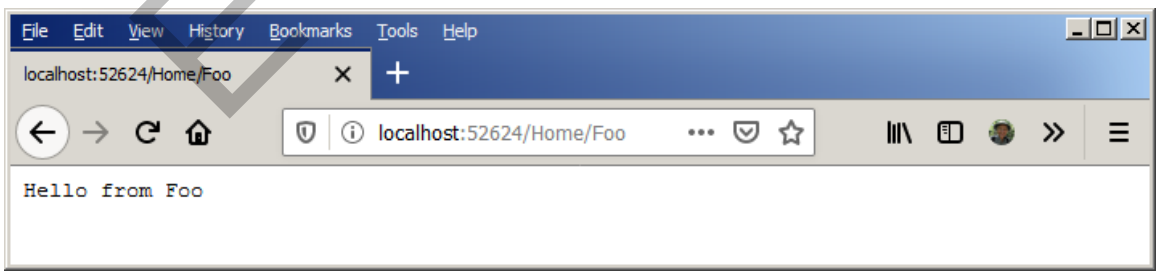
22. Now try using these URLs[1]. You should get the same result.

```
http://localhost:52624/Home/
http://localhost:52624/Home/Index/
```

23. Now try this URL.

```
http://localhost:52624/Home/Foo
```

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



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

Demo: Controller Only (Cont'd)

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

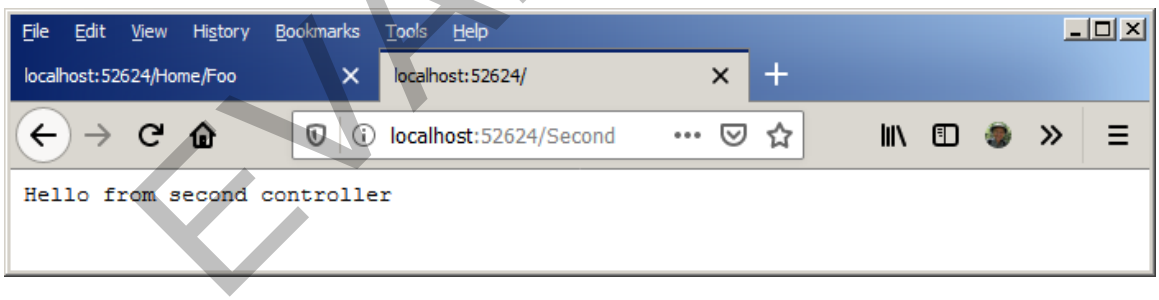
25. Provide the following code for the **Index()** method of the second controller.

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

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

```
http://localhost:52624/Second
http://localhost:52013/Second/Index
```

In either case we get the following result. The program at this point is saved in **MvcSimple\Controller** in the chapter folder[2].



[2] You should open all the ASP.NET Core 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 *Configure()* method in the *Startup.cs* file.**

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern:
"{controller=Home}/{action=Index}/{id?}");
});
```

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

Action Method Return Type

- **An action method normally returns a result of type *IActionResult.***
  - An action method can return any type, such as **string**, **int**, and so on, but then the return value is wrapped in an **ActionResult**, which implements the interface **IActionResult**.
- **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*.**

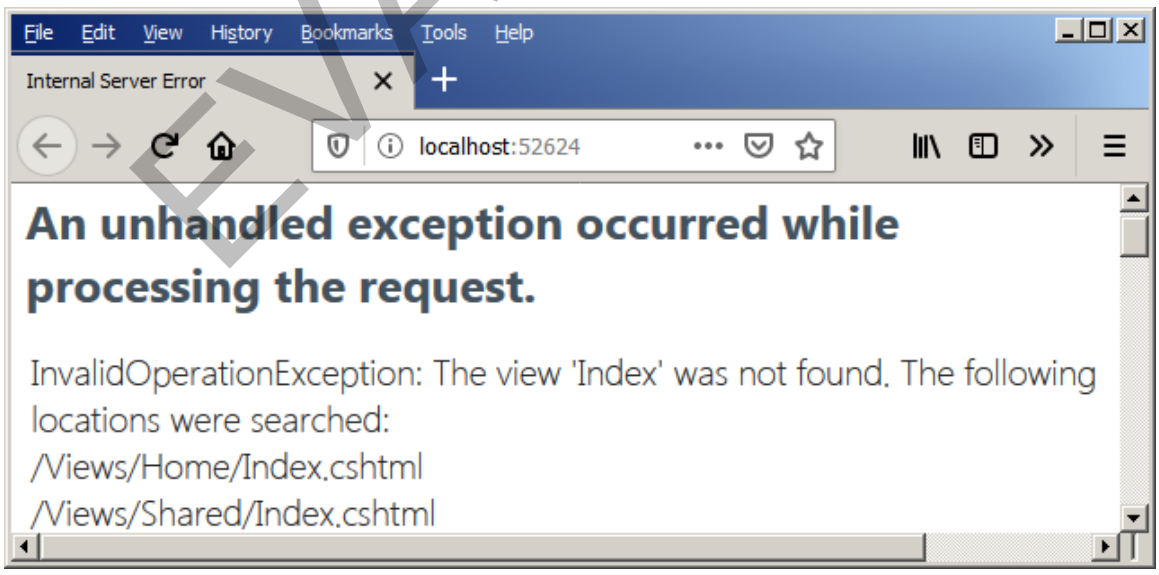
Action Result	Helper Method	Description
ViewResult	View()	Renders a view as a Web page, typically HTML
RedirectResult	Redirect()	Redirects to another action method using its URL
JsonResult	Json()	Returns a serialized Json object
FileResult	File()	Returns binary data to write to the response

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();
}
```

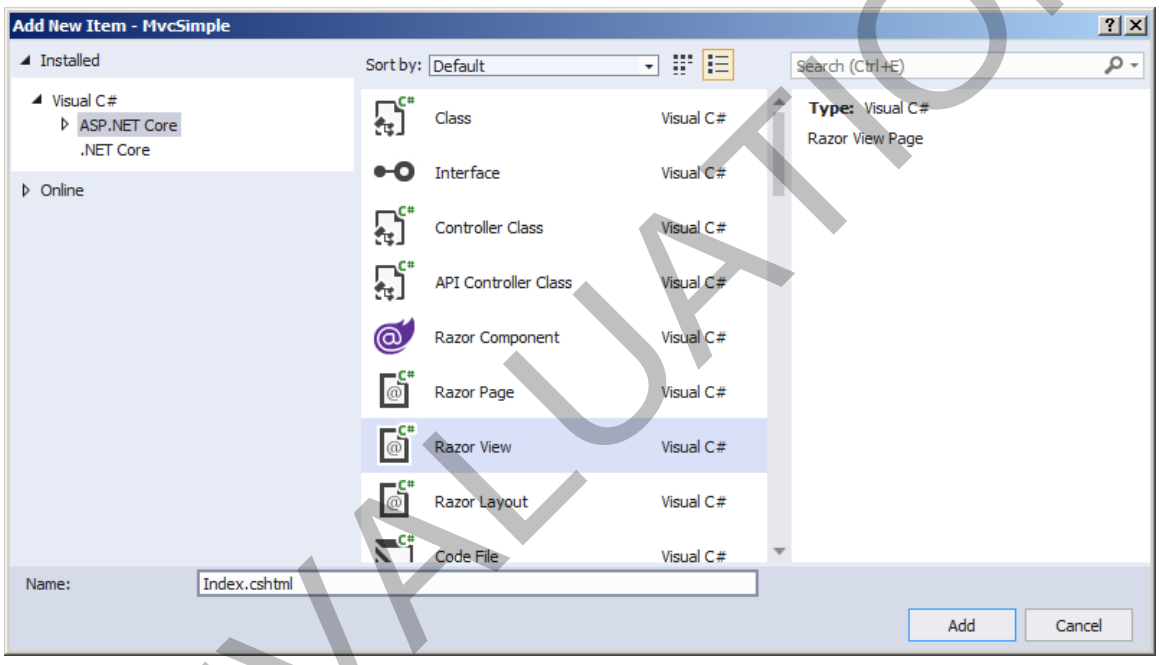
- **Try doing this in the *MvcSimple* program.**
  - Delete the second controller and simplify the home controller to have the one method shown.
- **Build and run. It compiles but you get a runtime error.**



Creating a View

- **The error message is quite informative!**
  - Let us create an appropriate file **Index.cshtml** in a folder **Views/Home**.

1. Add a new folder **Views** and under that a folder **Home**.
2. Right-click over Home and choose Add | New Item from the context menu.



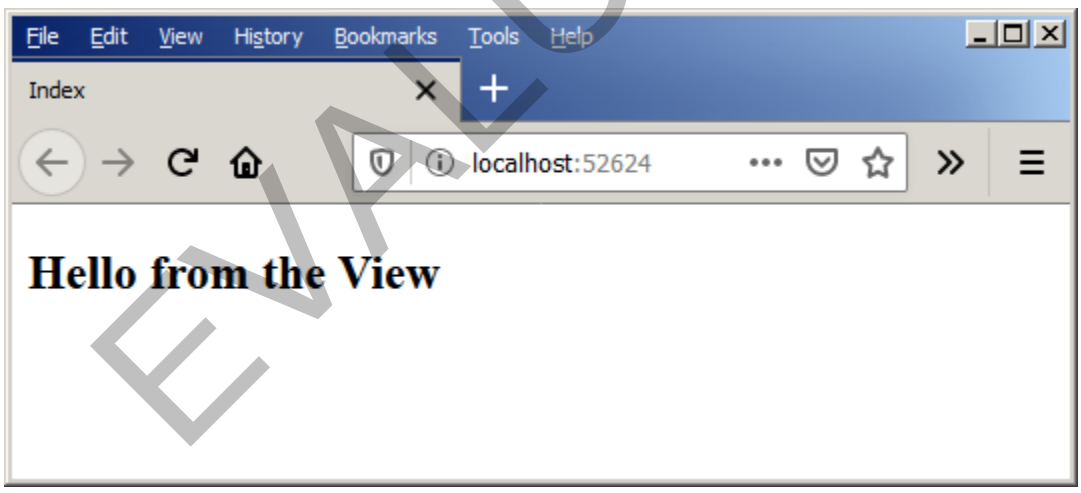
3. Select Razor View and click Add.

The View Web Page

- **A file *Index.cshtml* is created in the *Views\Home* folder.**
  - Add to this file HTML to display a welcome message from the view. To make it stand out, we used <h2> format.

```
<!DOCTYPE html>

<html>
<head>
    <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.**

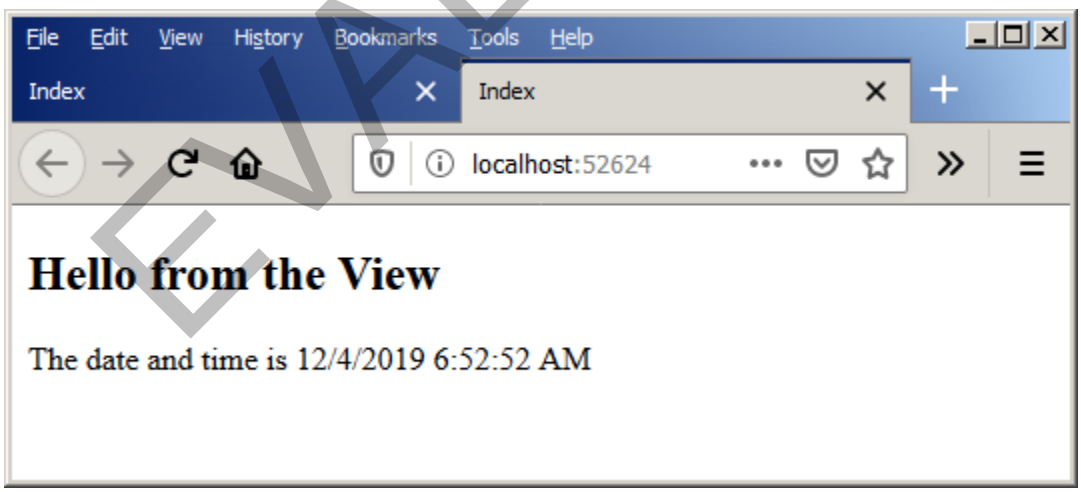
```
public ViewResult Index()
{
    ViewBag.Time = DateTime.Now.ToLocalTime();
    return View();
}
```

  - We need to import the **System** namespace.

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

```
<h2>Hello from the View</h2>
The date and 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 1.0 and 2.0 the default view engine is the Web Forms (or ASPX) view engine.**
- **In ASP.NET MVC 3.0 and 4.0 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 5.0 and higher and in ASP.NET Core MVC, the Razor view engine is used automatically, and we will employ it in our examples.**
  - In its first release, ASP.NET Core MVC only supported C#, but in ASP.NET Core 3.0 Visual Basic and F# are also supported.

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 />
 }
}
```

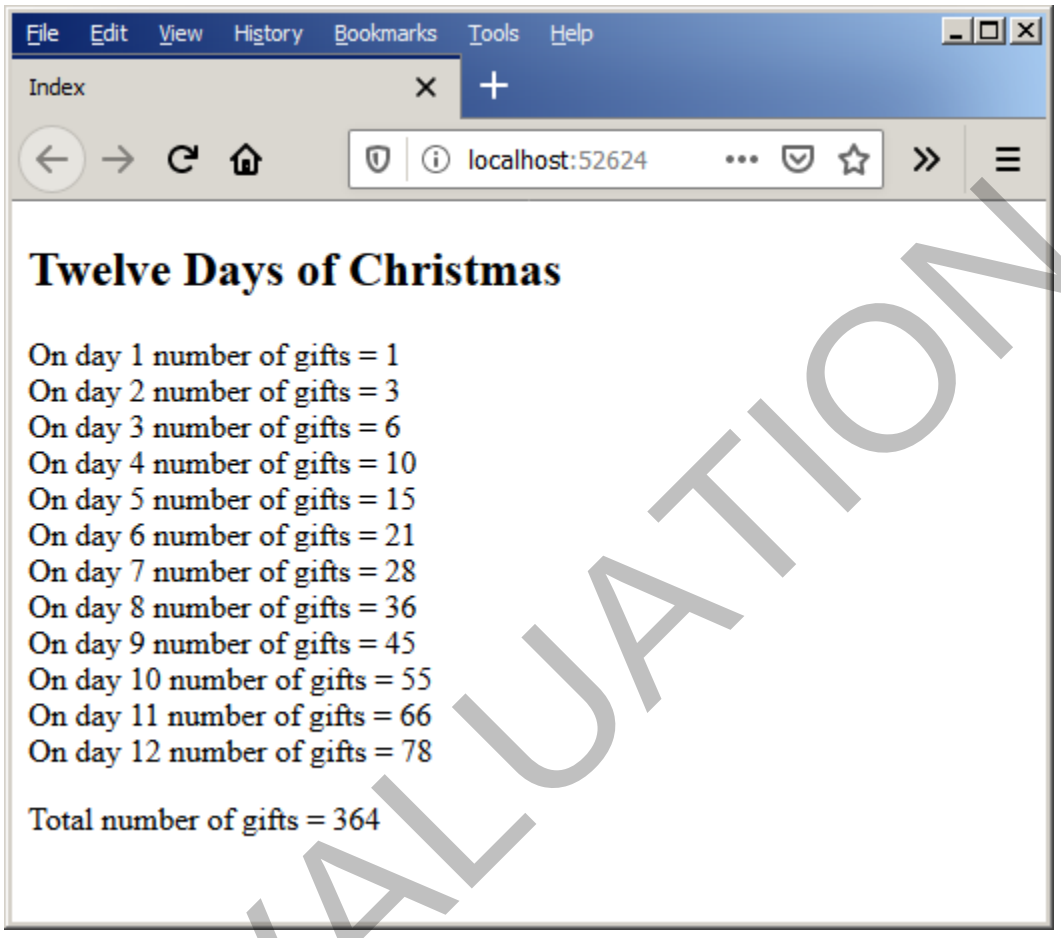
Embedded Script Example

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

```
<!DOCTYPE html>

<html>
<head>
    <title>Index</title>
</head>
<body>
 <h2>Twelve Days of Christmas</h2>
 @{
     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/>
     }
     <p>Total number of gifts = @total</p>
 }
</body>
</html>
```

Embedded Script Output

- **Build and run.**



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.

```
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.

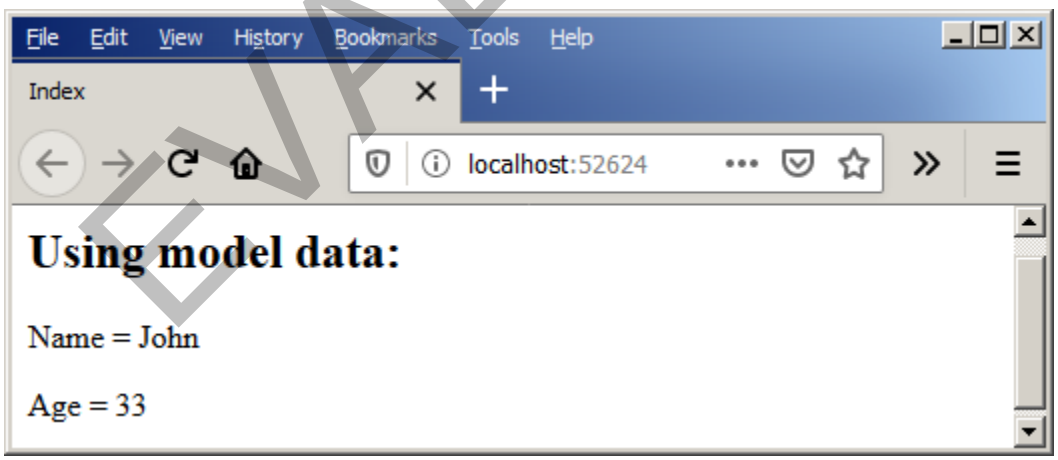
```
using Microsoft.AspNetCore.Mvc;
using MvcSimple.Models;

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.

```
@using MvcSimple.Models;
<!DOCTYPE html>

<html>
<head>
    <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>
</html>
```

  - 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.**

```
@using MvcSimple.Models;

...

<body>
    <h2>Using model data directly:</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.

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:

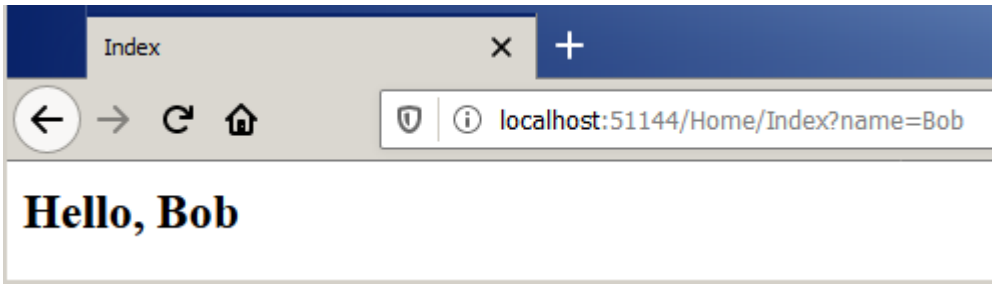
```
/Home/Index?name=Bob
```

  - 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 IActionResult 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 Core 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: 60 minutes

Summary

- **You can begin creating an ASP.NET Core 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 Core MVC 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 a query string.**

Lab 2

Contact Manager Application

Introduction

In this lab you will implement an ASP.NET Core 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: 60 minutes

Root Directory: **C:\OIC\MvcCore**

Directories:	**Labs\Lab2**	(do your work here)
	Labs\Lab2\Contact.cs	(enhanced code for model)
	Chap02\MvcContact\Step1	(solution to Part 1)
	Chap02\MvcContact\Step2	(solution to Part 2)

Data File: **C:\OIC\Data\Contact.txt**

Part 1. Create Starter ASP.NET Core MVC Application

In this part you will create a simple ASP.NET Core MVC application with a controller, view and model that displays information for a single contact.

1. Create a new ASP.NET Core Empty Web Application **MvcContact** in the working directory.

2. Edit **Startup.cs** as done in the **MvcSimple** example..

```
public void ConfigureServices(
                      IServiceCollection services)
{
   services.AddControllersWithViews();
}

public void Configure(IApplicationBuilder app,
                      IWebHostEnvironment env)
{
   if (env.IsDevelopment())
   {
       app.UseDeveloperExceptionPage();
   }
   app.UseStaticFiles();

   app.UseRouting();

   app.UseEndpoints(endpoints =>
```

```
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Or, you may simply copy **Startup.cs** from the **MvcSimple** example and change the namespace to **MvcContact**.

3. Add new folders **Controllers**, **Views** and **Views\Home** to your project.

4. Add a new item, an MVC Controller class, **HomeController**. Examine the starter code. Provide the comment.

```
public class HomeController : Controller
{
    // GET: /Home/
    public IActionResult Index()
    {
        return View();
    }
}
```

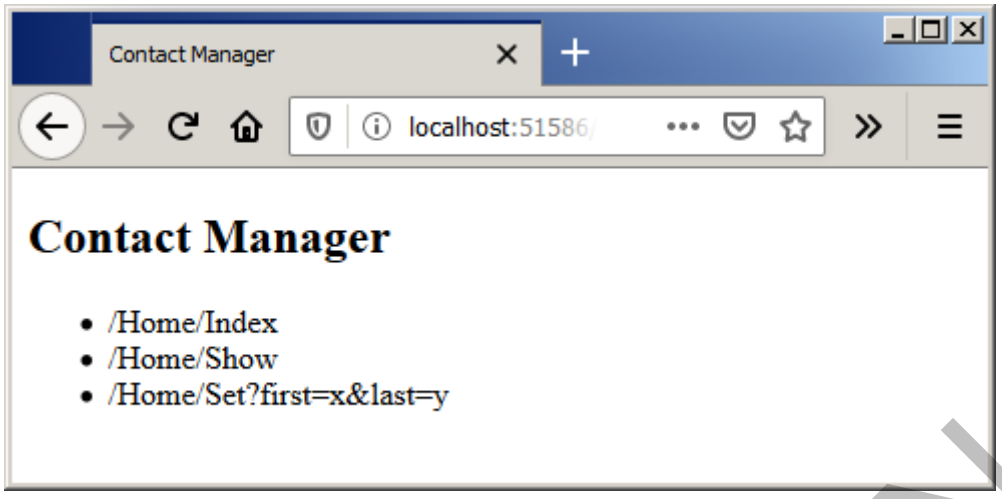
5. Right-click over the **Views\Home** folder and add a new item, a Razor View **Index.cshtml**.

6. Replace the contents of that file with this simple HTML. It provides a documentation page for various URLs that we will be able to submit in the final application.

```
<!DOCTYPE html>

<html>
<head>
    <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>
```

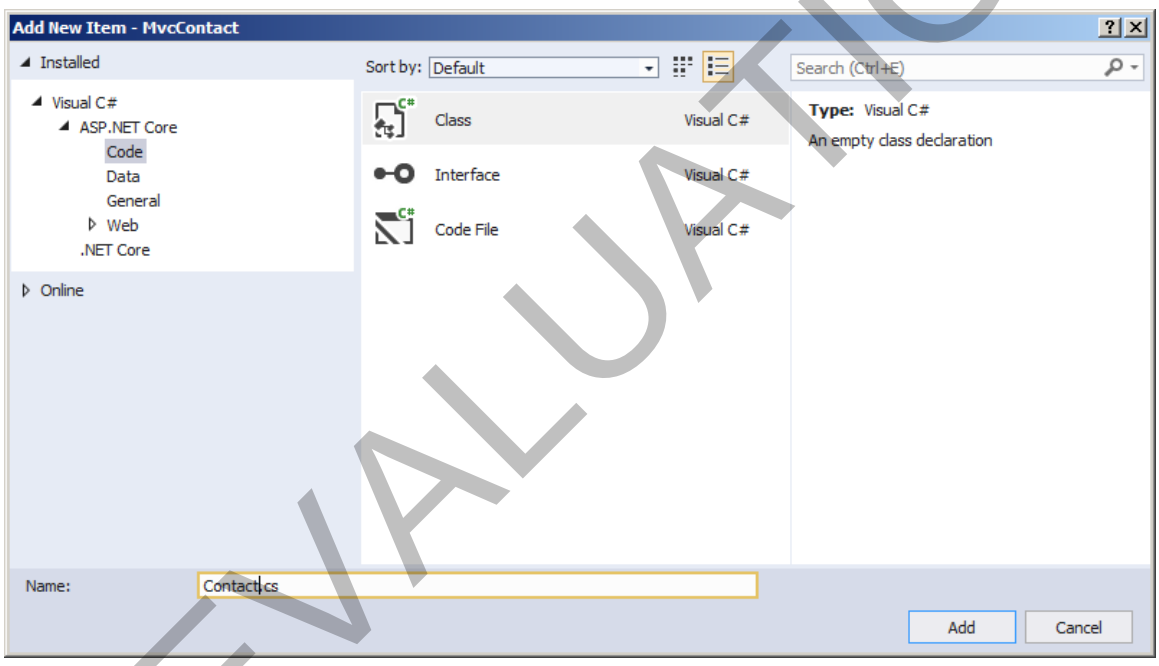
7. Build and run. You should get output like that shown below. For example, try this alternate URL for the index page (your port number will be different).

```
http://localhost:51586/Home/Index
```



8. Next we will add a simple Model. Add a new folder **Models** and right-click over it.

9. Choose Add | Class from the context menu. Name your class file **Contact.cs**.



10. Click Add.

11. Provide the following code defining two properties **FirstName** and **LastName** along with a constructor that will initialize the contact to have first name “John” and last name “Smith”.

```
namespace MvcContact.Models
{
    public class Contact
    {
        public string FirstName { get; set; }
```

```
        public string LastName { get; set; }
        public Contact()
        {
            FirstName = "John";
            LastName = "Smith";
        }
    }
}
```

12. In the home controller 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 ViewResult Show()
        {
            return View("Show", new Contact());
        }
```

13. Import the **MvcContact.Models** namespace .

```
using MvcContact.Models;
```

14. In Solution Explorer right-click over the **Views/Home** folder and add a new Razor View. Name it **Show.cshtml**.

15. Replace the contents of the page with the following Razor code.

```
@model MvcContact.Models.Contact

<!DOCTYPE html>

<html>
<head>
    <title>Show</title>
</head>
<body>
    Name = @Model.FirstName @Model.LastName
</body>
</html>
```

16. Build and run. Use a URL like the following (your port number will be different).

```
http://localhost:51586/Home/Show
```

17. You should see the hardcoded model data displayed. This completes Part 1.

Part 2. Read and Write Contact Data in a File

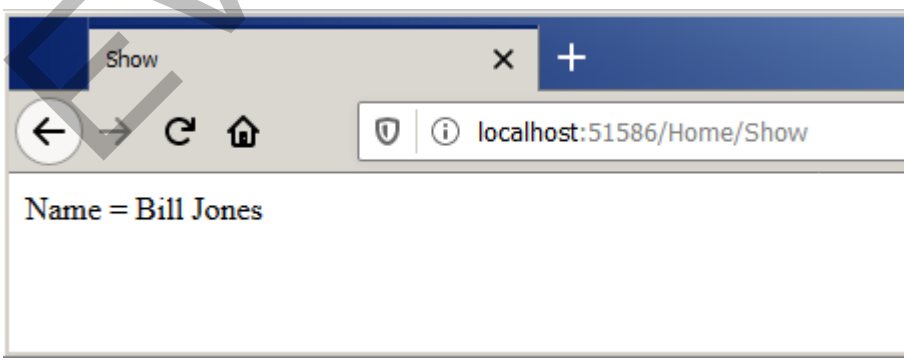
In this part you will enhance your application to read the contact data from a file. You will also add a new controller action method and corresponding view to enable you to write contact data to the file.

1. Examine the File **Contact.cs** in the **Lab2** folder. It defines an enhanced **Contact** class in which the initial data is read from a file rather than hardcoded. There is also a method.

```
using System.IO;
namespace MvcContact.Models
{
    public class Contact
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public Contact()
        {
            string[] names = ReadContact();
            FirstName = names[0];
            LastName = names[1];
        }
        public static string[] ReadContact()
        {
            // Read from file contact.txt
            string contact =
                File.ReadAllText(@"C:\OIC\Data\Contact.txt");
            char[] seps = {' '};
            return contact.Split(seps);
        }
        public static void WriteContact(string first, string last)
        {
            File.WriteAllText(
                @"C:\OIC\Data\contact.txt", first + " " + last);
        }
    }
}
```

2. Copy this version of **Contact.cs** into the Models folder of your project, overwriting the previous version.

3. Rebuild your solution and run it. In the browser use the URL for invoking the Show action method. Now you will see different starting contact data, because it is read in from a file that has different data in it.



4. 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.

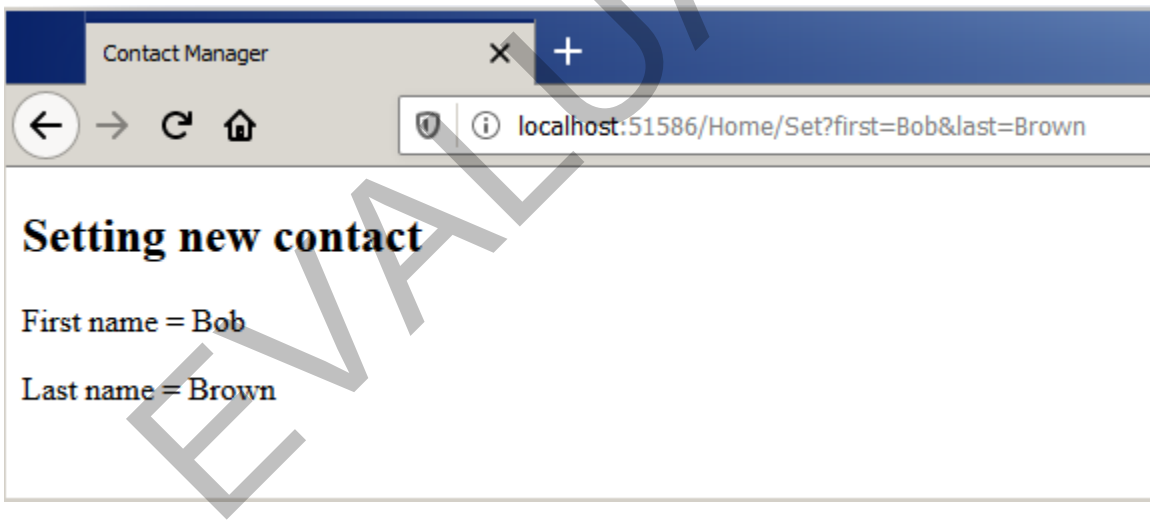
```
// GET: /Home/Set?first=x&last=y
public ActionResult Set(string first, string last)
{
   Contact.WriteContact(first, last);
   ViewBag.First = first;
   ViewBag.Last = last;
   return View();
}
```

5. 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>
```

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

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



7. Finally, invoke Show again. You should see the new contact displayed. This completes Part 2.