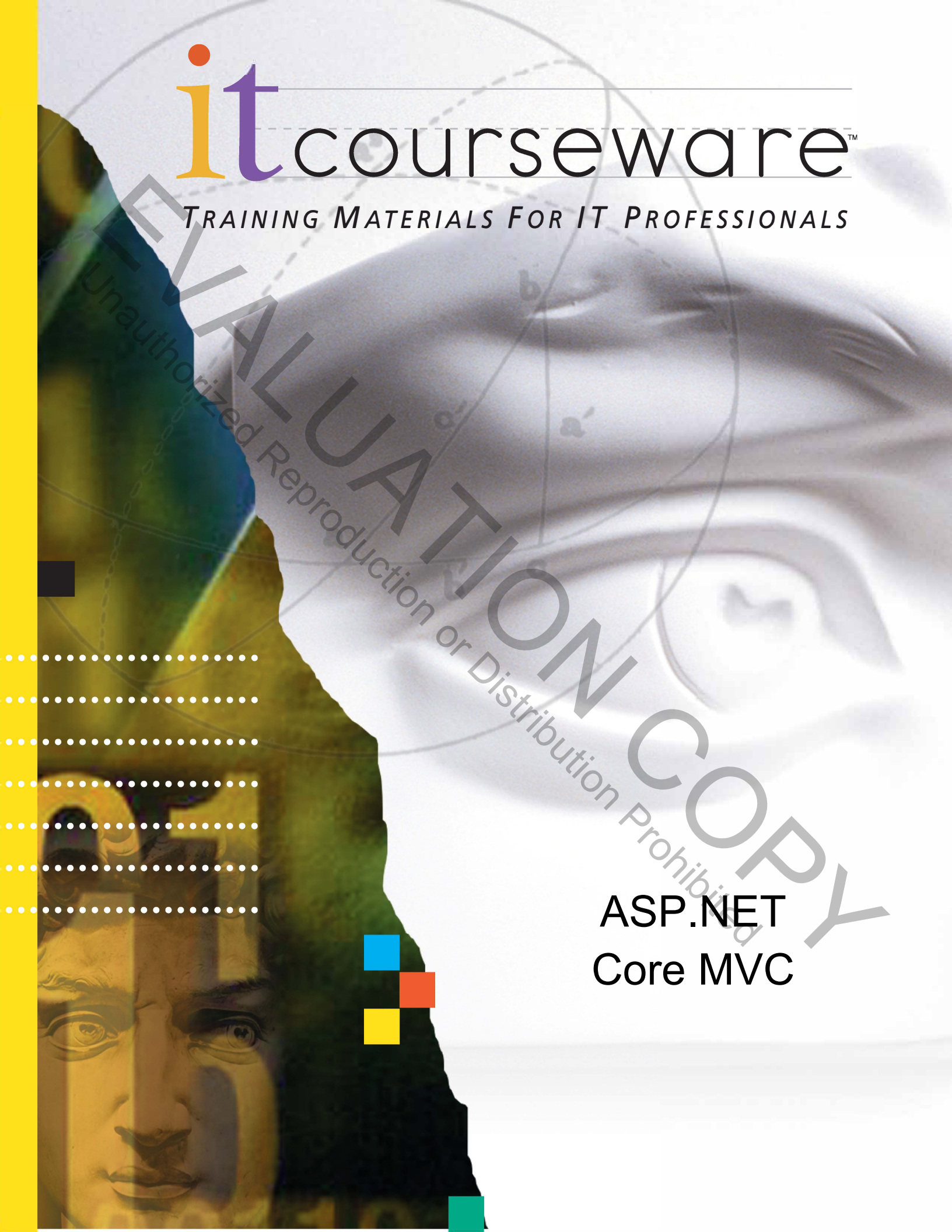

itcourseware™
TRAINING MATERIALS FOR IT PROFESSIONALS
EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited
ASP.NET
Core MVC

ASP.NET Core MVC

Student Guide

Revision 6.0

Object Innovations Course 4043

ASP.NET Core MVC
Rev. 6.0

Student Guide

Authors: Robert Hurlbut and Robert J. Oberg

Object Innovations
877-558-7246
www.objectinnovations.net

Published in the United States of America.

Table of Contents (Overview)

Chapter 1	Introduction to ASP.NET Core MVC
Chapter 2	Getting Started with ASP.NET Core MVC
Chapter 3	ASP.NET MVC Architecture
Chapter 4	The Model
Chapter 5	The Controller
Chapter 6	The View
Chapter 7	Routing
Chapter 8	ASP.NET Core Web API
Chapter 9	ASP.NET Core and Azure
Appendix A	Learning Resources

Directory Structure

- **The course software installs to the root directory *C:\OIC\MvcCore*.**
  - Example programs for each chapter are in named subdirectories of chapter directories **Chap02**, **Chap03**, and so on.
  - The **Labs** directory contains one subdirectory for each lab, named after the lab number. Starter code is frequently supplied, and answers are provided in the chapter directories.
  - The **Demos** directory is provided for doing in-class demonstrations led by the instructor.
  - The file **Links.html** contains link to web sites and pages mentioned in the text along with other useful links.
- **Data files install to the directory *C:\OIC\Data*.**

Table of Contents (Detailed)

Chapter 1: Introduction to ASP.NET Core MVC 1
Review of ASP.NET Web Forms 3
Advantages of ASP.NET Web Forms 4
Disadvantages of ASP.NET Web Forms 5
Model-View-Controller Pattern 6
ASP.NET MVC 7
ASP.NET Core 8
What is .NET Core? 9
Advantages of ASP.NET MVC 10
ASP.NET MVC Considerations 11
Goals of ASP.NET MVC 12
ASP.NET Core 6.0 13
Unit Testing 14
Summary 15

Chapter 2: Getting Started with ASP.NET Core MVC 17
An ASP.NET Core MVC Testbed 19
Visual Studio ASP.NET MVC Demo 20
Starter Application 23
Simple App with Controller Only 25
Edit Program.cs 31
Action Methods and Routing 34
Action Method Return Type 35
Rendering a View 36
Creating a View 37
The View Web Page 38
Dynamic Output 39
Razor View Engine 40
Embedded Scripts 41
Embedded Script Example 42
Using a Model with ViewBag 44
Controller Using Model and ViewBag 45
View Using Model and ViewBag 46
Using Model Directly 47
Passing Parameters in Query String 48
New and Old Project Templates 49
Lab 2 50
Summary 51

Chapter 3: ASP.NET MVC Architecture 59
The Controller in ASP.NET MVC 61
The View in ASP.NET MVC 62
The Model in ASP.NET MVC 63

How MVC Works 64
Using Forms 65
HTML Helper Functions 66
Handling Form Submission 67
Model Binding 68
Greet View 69
Input Validation 70
Nullable Type 71
Checking Model Validity 72
Validation Summary 73
Lab 3 74
Summary 75

Chapter 4: The Model 83
Complex Models 85
MvcBooks Example 86
The View 87
The Model: Book and Category 88
The Model: DB 89
MvcBooks – Step 2 90
Books by Category 91
Books by Category – View 92
Running Step 2 93
Microsoft Technologies for the Model 94
XML Serialization Demo 95
Running the Starter Code 97
XML Serialization: Save 98
Deserialization 102
XML Serialization 103
What Will Not Be Serialized 104
Lab 4 105
XML as a Data Store 106
MvcBooks – Model 107
Save() 108
Restore() 109
AddCategory() 110
MvcBooks – Controller 111
Adding a Category 112
View for Adding a Category 113
Running the Example 114
SmallPub Database 115
Using ADO.NET 117
Model 118
Controller and View 119
Running the Database Example 120
Summary 121

Chapter 5: The Controller 127
Controller Base Class 129
Controller Base Class 130
Action Methods 131
Action Method Example 132
Index() Action Method 133
Info() Action Method 134
Info.cshtml 135
Running the Example 136
Receiving Input 137
Binding Example 138
Non-Nullable Parameters 139
Nullable Parameters 140
Using a Model 141
Action Results 142
Action Result Example 143
Output Demo 144
JavaScript Object Notation 148
Serving Static Files 149
Action Method Attributes 150
Lab 5 151
Filters 152
Asynchronous Controllers 154
Summary 155

Chapter 6: The View 161
View Responsibility 163
A Program with a View 164
View Page 166
Passing Data to the View 167
Dynamic and ExpandoObject 168
Passing Lists to the View 169
HTML Helper Methods 171
Link-Building Helpers 172
Form Helpers 173
Html Helper Example 174
Validation Helpers 176
Templated Helpers 177
Validation in Model 179
Validation in Controller 180
ValidationMessage Helper 181
Running the Example 182
Lab 6 183
Summary 184

Chapter 7: Routing 191

ASP.NET Routing 193
Routing in ASP.NET Core MVC 194
Simple Route Example 195
Math Controller 196
Default Values for URL Parameters 198
Using a Default Route 199
Home Controller 200
Assigning Parameter Values 201
Controller Code 202
View Code 203
Running the Example 204
Properties of Routes 205
Optional Parameter 206
Matching URLs to Route 207
Defaults 208
Multiple Routes Example 209
Attribute Routing 213
Combining Routes 215
Empty String for Route Attribute 216
Token Replacement 217
Summary 218

Chapter 8: ASP.NET Core Web API 219
ASP.NET Core Web API 221
REST 222
Representation, State and Transfer 223
Collections and Elements 224
Web API Demo 225
Strings Controller 230
Project Settings 234
HTTP Testing Tools 235
Using Postman 236
Implementing and Testing POST 239
Lab 8A 242
HTTP Response Codes 243
Testing Improved Code for GET 244
POST Response Code 245
Named Route 246
Testing Improved Code for POST 247
Location Header 248
Response Code for PUT and DELETE 249
Web API Clients 250
HttpClient 252
Initializing HttpClient 253
Issuing a GET Request 254
Issuing a POST Request 255

Lab 8B 256
Summary 257

Chapter 9: ASP.NET Core and Azure 267
What Is Windows Azure? 269
A Windows Azure Testbed 270
Windows Azure Demo 271
Publish to Azure 272
Web Deployment Completed 278
Modifying a Web Application 279
Deploy to Original Site 280
A Deployed Application 281
Lab 9 282
Summary 283

Appendix A Learning Resources 289

Chapter 1

Introduction to ASP.NET Core MVC

Introduction to ASP.NET Core MVC

Objectives

After completing this unit you will be able to:

- **Describe advantages and disadvantages of ASP.NET Web Forms.**
- **Understand the Model-View-Controller (MVC) pattern**
- **Outline the parts of an ASP.NET MVC application.**
- **Describe advantages of ASP.NET MVC and issues to consider in using this technology.**
- **Describe ASP.NET Core 6.0.**
- **Understand the use of unit testing in creating ASP.NET MVC applications.**

Review of ASP.NET Web Forms

- **ASP.NET Web Forms provide a way to build web applications.**
- **You can use compiled, object-oriented languages with ASP.NET, including C# and Visual Basic.**
  - All the power of the .NET Framework is available to you, including the extensive class library.
- **Code and presentation elements can be cleanly separated.**
  - Code can be provided in a separate section of a Web page from user interface elements.
  - The separation can be carried a step further by use of separate "code behind" files.
- **ASP.NET Web Forms comes with an extensive set of server controls that provide significant functionality out of the box.**
- **Server controls transparently handle browser compatibility issues.**
- **Configuration is handled by XML files without need of any registry settings, and deployment can be done simply by copying files.**

Advantages of ASP.NET Web Forms

- **ASP.NET Web Forms continue to be available in classic .NET and have their own advantages:**
  - A rich event model supported in hundreds of server controls facilitates easy development of Web server applications, following a familiar GUI development paradigm.
  - View state makes it easy to manage state information.
  - The model works well for individuals and small teams doing rapid application development.
  - The large number of built-in and third-party components also facilitates rapid application development.
- **In general, Web Forms are quite easy to work with and generally require less code.**
- **Important Note: Web Forms are *not* available in ASP.NET Core.**
  - The brief discussion of Web Forms in this introduction is intended to place in context the newer technology, ASP.NET MVC.
  - For porting Web Forms apps to ASP.NET Core, see:

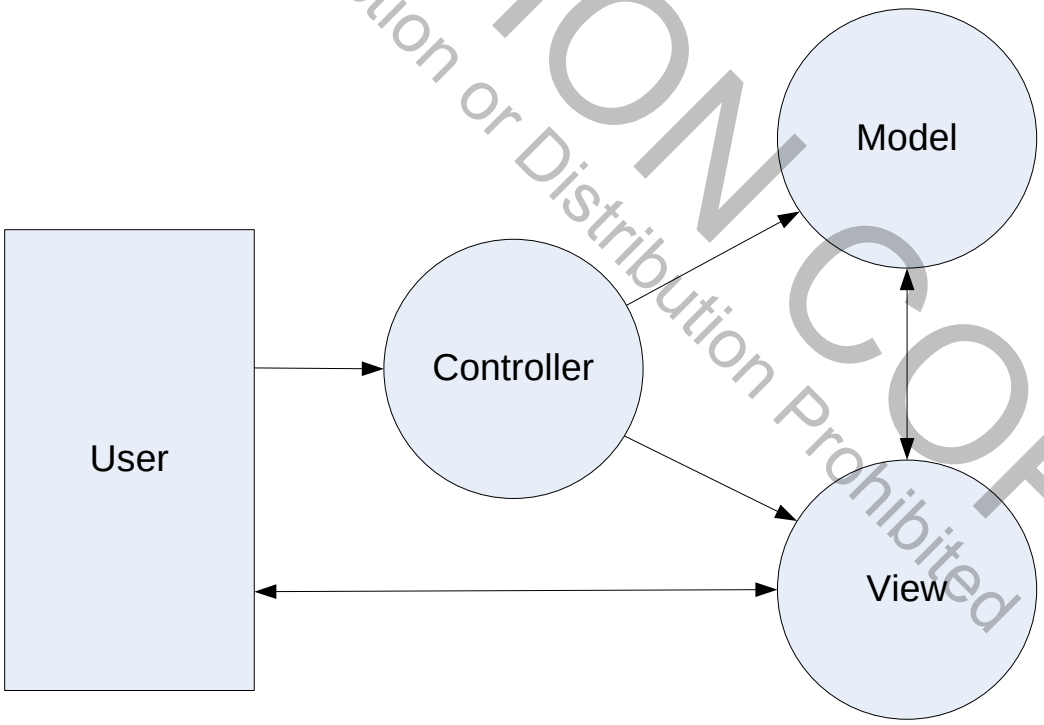
https://docs.microsoft.com/en-us/dotnet/architecture/porting-existing-aspnet-apps/migrate-web-forms

Disadvantages of ASP.NET Web Forms

- **Key disadvantages of ASP.NET Web Forms include**
  - ViewState tends to be large depending on the number of server controls contained on the page, thus increasing the size of the page and the length of the response time from server to browser
  - ASP.NET Web Forms provide tight coupling with the code-behind classes which make automated testing of the back-end code apart from the web pages more difficult
  - Because the code-behind classes are so tightly coupled to the web forms, developers are encouraged to mix presentation code with application logic in the same code-behind classes which can lead to fragile and unintelligible code
  - Limited control of HTML rendered through use of server controls

Model-View-Controller Pattern

- **The Model-View-Controller (MVC) design pattern divides an application into three conceptual components:**
  - A **model** represents the data and operations that are meaningful to the domain of the application. It implements the application logic for the domain.
  - **Views** display a user interface for portions of the model. Typically the UI is created from model data.
  - **Controllers** handle incoming requests, work with the model, and select a view to render a UI back to the user.



ASP.NET MVC

- **ASP.NET MVC is a framework based on ASP.NET for creating Web applications.**
  - It is an alternative to Web Forms in classical .NET.
- **ASP.NET MVC 1.0 was installed on top of .NET 3.5 SP1 and Visual Studio 2008 SP.**
- **ASP.NET MVC 2.0 is integrated into .NET 4.0 and Visual Studio 2010.**
- **ASP.NET MVC 3.0 is a separate download and adds important new features, such as the Razor view engine.**
- **ASP.NET MVC 4.0 is integrated into .NET 4.5 and Visual Studio 2012.**
- **ASP.NET MVC 5.0 is integrated into .NET beginning with .NET 4.5.2 and Visual Studio 2013.**
  - ASP.NET MVC 5 is covered in OI course 4143 .
- **ASP.NET MVC 6.0 is integrated into .NET 6.0 and Visual Studio 2022**

ASP.NET Core

- **ASP.NET Core is a completely new version of ASP.NET that is open-source and cross-platform.**
  - It runs on Windows, Mac and Linux.
- **It runs on.NET Core.**
- **ASP.NET Core is very modular providing flexibility in using what is needed for an application with minimal overhead.**
- **ASP.NET provides a unified framework for building both web UI and web APIs.**
- **Supported web UI include the following besides MVC:**
  - Razor Pages provides a simple model for page-based web UI and is a good replacement for ASP.NET Webforms, which are not supported in .NET Core.
  - Blazor lets you use C# in the browser alongside JavaScript.
- **This course focuses on ASP.NET Core MVC running on .NET 6, with an introduction to Web API.**

What is .NET Core?

- **.NET Core is a general purpose development platform that is cross-platform, supporting:**
  - Windows
  - Mac
  - Linux
- **It is open-source (MIT license) maintained by Microsoft and the .NET community on GitHub.**
- **.NET Core is composed of:**
  - A .NET runtime
  - A set of framework libraries
  - A set of SDK tools and language compilers
  - The app host **dotnet**, which is used to launch .NET Core apps.
- **.NET Core is distributed through NuGet packages.**
  - Packages are relatively fine grained, allowing smaller app size.
  - Metapackages are more coarse-grained, describing a set of packages that are meaningful together.
- **NuGet is the package manager for the Microsoft development platform. It provides a gallery as a central repository.**

Advantages of ASP.NET MVC

- **Key advantages of ASP.NET MVC include:**
  - The MVC pattern promotes **separation of concerns** into input logic (controller), business logic (model) and UI (view). This aids in managing complexity.
  - These components are loosely coupled, promoting parallel development.
  - This loose coupling also facilitates automated testing.
  - Views are created using standard HTML and cascading style sheets, giving the developer a high degree of control over the user interface.
  - There is no view state, reducing the load on the browser in rendering a page.
- **Separation of Concerns:**
  - Each component has one responsibility
  - SRP – Single Responsibility Principle
  - DRY – Don't Repeat Yourself
  - More easily testable
  - Helps with concurrent development

ASP.NET MVC Considerations

- **Key considerations in using ASP.NET MVC include:**
  - Writing View contents the old ASP-like way (though this is now easier with the newer Razor view syntax).
  - Unit testing and Test Driven Development (TDD) are encouraged and used more but also bring a steep learning curve.
  - Need to understand HTML controls and style sheets, but at the same time this allows a designer to work independently of the coders.

Goals of ASP.NET MVC

- **The ASP.NET MVC Framework has the following goals:**
  - Frictionless Testability
  - Tight control over markup
  - User/Search Engine friendly URLs
  - Leverage the benefits of ASP.NET
  - Conventions and Guidance
- **Extensibility**
  - Replace any component of the system
  - Interface-based architecture
  - Very few sealed methods / classes

ASP.NET Core MVC 6.0

- **.NET 6.0 is the next major release of .NET Core.**
- **Microsoft has dropped "Core" from the name to emphasize that this is the main implementation of .NET going forward.**
  - .NET 6.0 supports more types of apps and more platforms than .NET Core or .NET Framework.
- **ASP.NET Core MVC 6.0 is based on .NET 6.0 but retains the name "Core" to avoid confusing it with ASP.NET MVC 6.**
  - Read more in “What’s new in .NET 6.0”.

https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-6

  - See also “What’s new in ASP.NET Core 6.0”.

https://docs.microsoft.com/en-us/aspnet/core/release-notes/aspnetcore-6.0

Unit Testing

- **Unit testing lets you specify the expected behavior of individual classes or other small code units in isolation.**
- **ASP.NET MVC encourages unit testing of the Models and the Controllers of the application to verify expected behaviors.**
- **Separation of concerns makes unit testing of individual components feasible.**
- **There are several choices for unit testing in .NET Core.**
  - The new **dotnet test** supports cross-platform testing.
  - Microsoft's MSTest is integrated into Visual Studio and can be used for testing .NET Core on Windows.
  - The open-source xUnit can be used for testing .NET Core applications.
- **This link provides information on unit testing in .NET Core:**

https://docs.microsoft.com/en-us/dotnet/core/testing/

Summary

- **ASP.NET Web Forms is still used and has both advantages and disadvantages compared to MVC.**
  - But Web Forms is not available in ASP.NET Core.
- **The Model-View-Controller (MVC) pattern is useful in creating applications that have separation of concerns.**
- **ASP.NET Core is open-source and cross platform and unifies MVC and Web API.**
- **Unit testing is helpful and encouraged in developing ASP.NET MVC applications.**

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.**

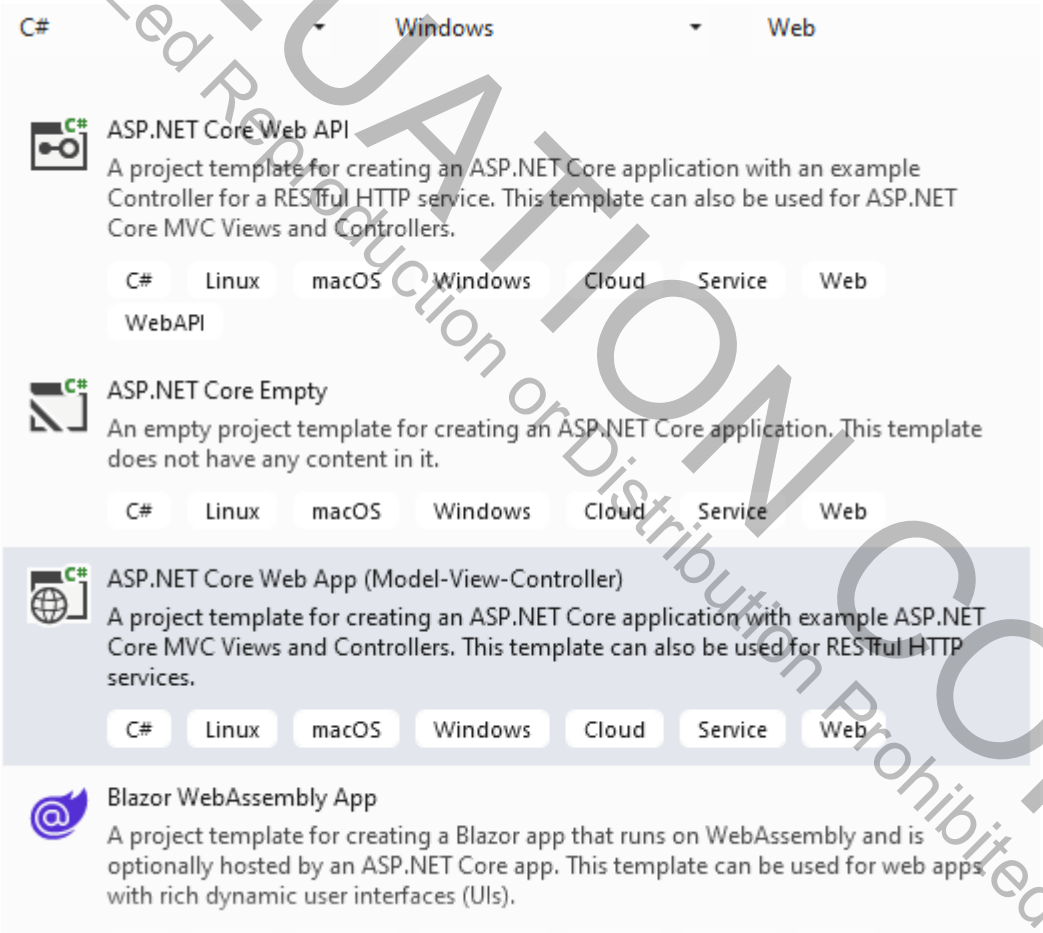
An ASP.NET Core MVC Testbed

- **This courses uses the following software:**
  - Visual Studio 2022. The course was developed using the free Visual Studio Community 2022. **Be sure to include the workload ASP.NET and web development.**
  - .NET 6.0, which is bundled with Visual Studio 2022.
  - SQL Server 2019 LocalDB, which comes bundled with Visual Studio 2022.
- **Required operating system is Windows 10 or higher.**
  - The course was developed and tested on Windows 10.

Visual Studio ASP.NET MVC Demo

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

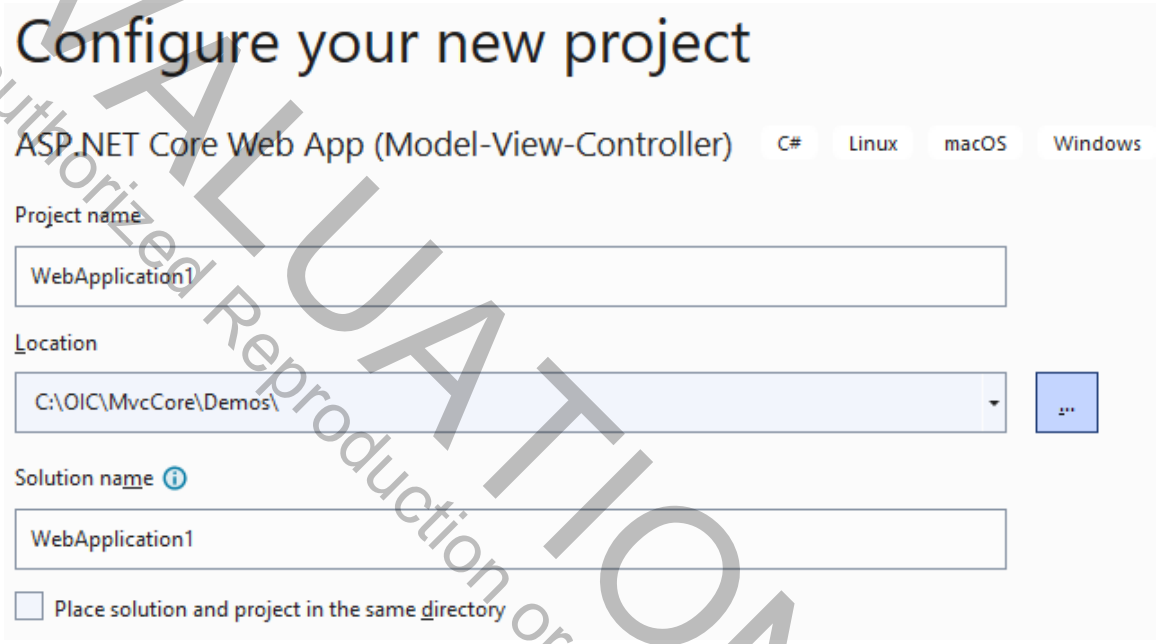
1. From the opening screen choose Create a new project.
2. From among the Web templates choose ASP.NET Core Web App (Model-View-Controller).



3. Click Next.

ASP.NET MVC Demo (Cont'd)

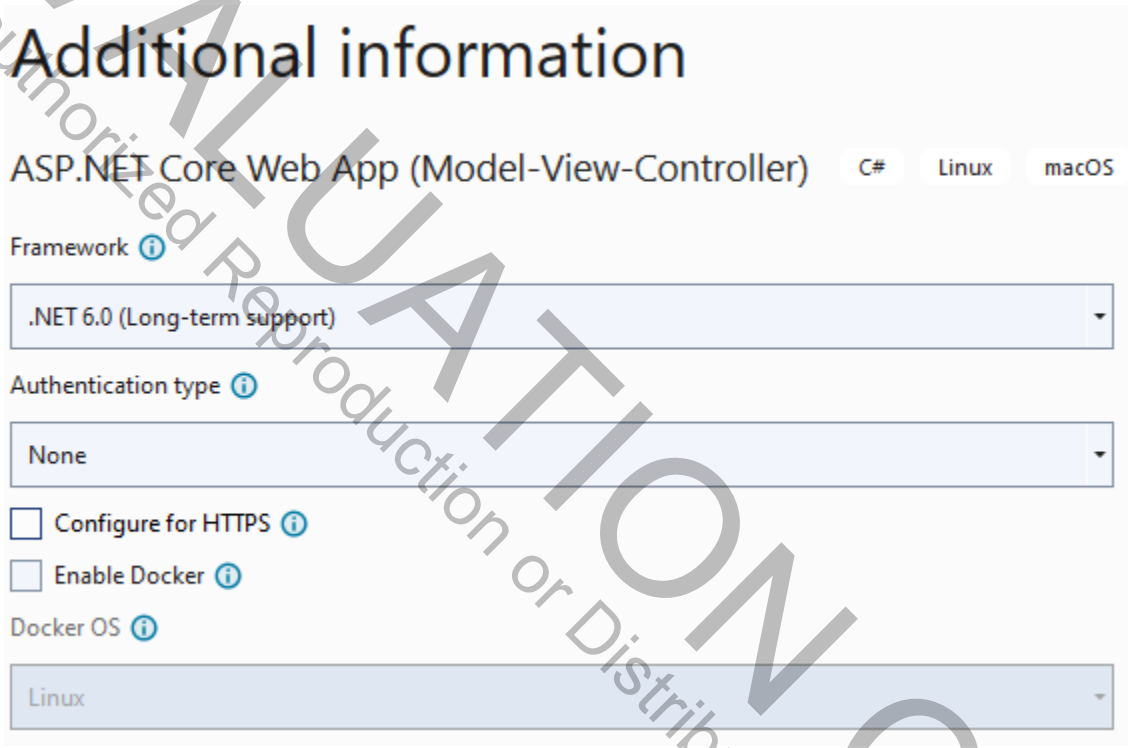
4. Browse to the **C:\OIC\MvcCore\Demos** folder for Location, and leave the Project name as WebApplication1.



5. Click Next.

ASP.NET MVC Demo (Cont'd)

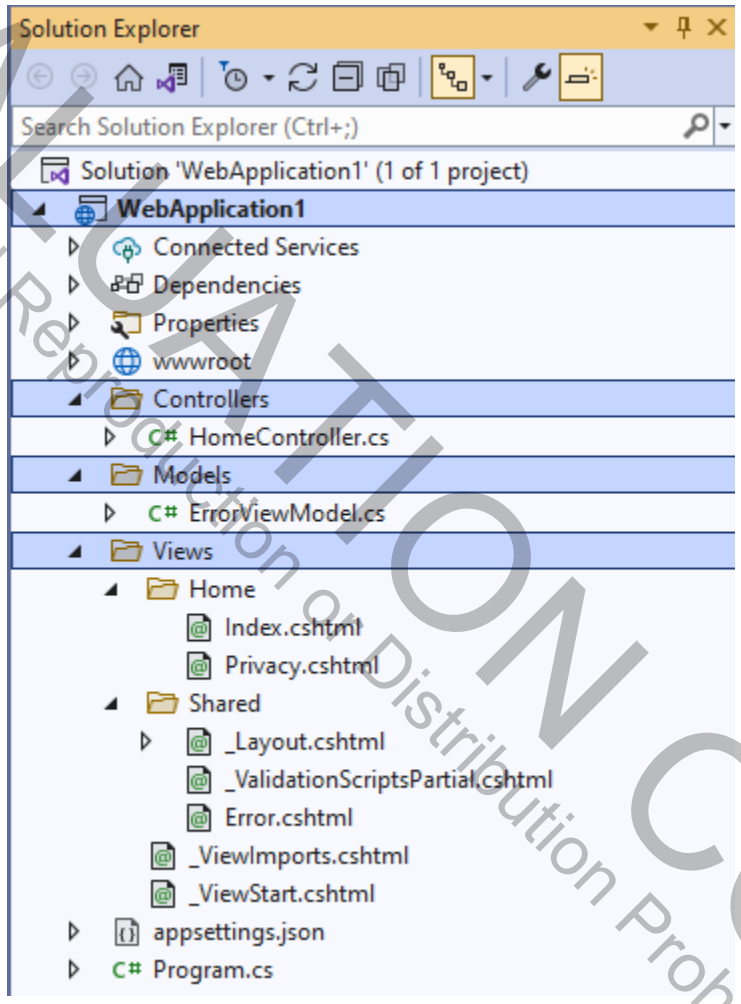
6. Choose .NET 6.0 (Long-term support) as the Target Framework and clear the checkbox Configure for HTTPS



7. Click Create.

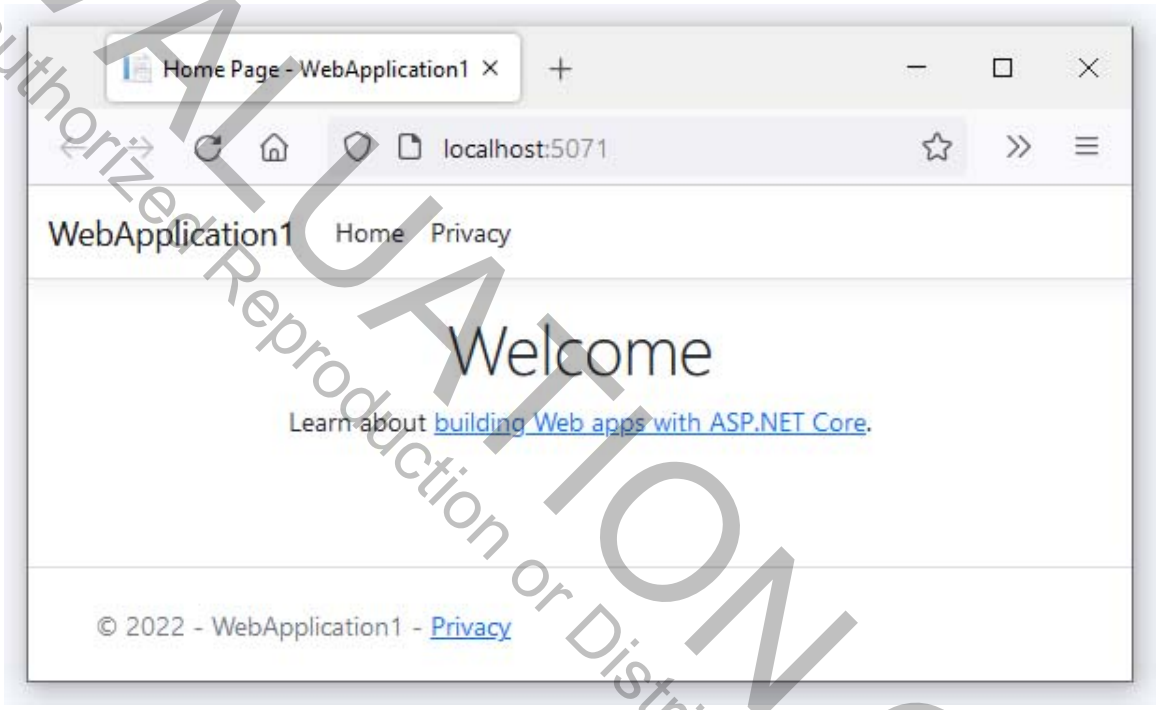
Starter Application

- **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.

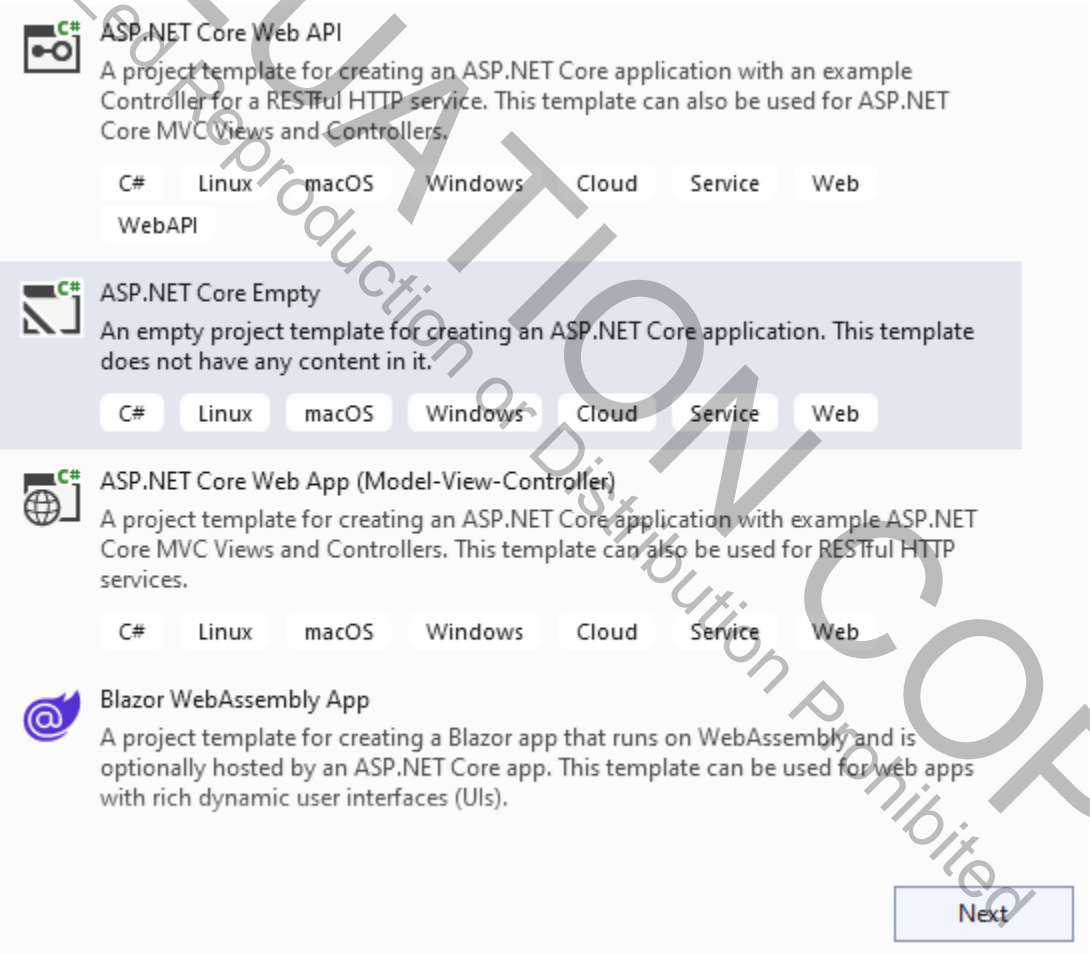


  - This starter application, built out of the box, actually has many features.
  - In particular, through use of the Bootstrap framework, it has responsive design and will display well on mobile devices as well as on the desktop.
  - You can test locally, and you could also deploy to the Azure cloud. 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.**

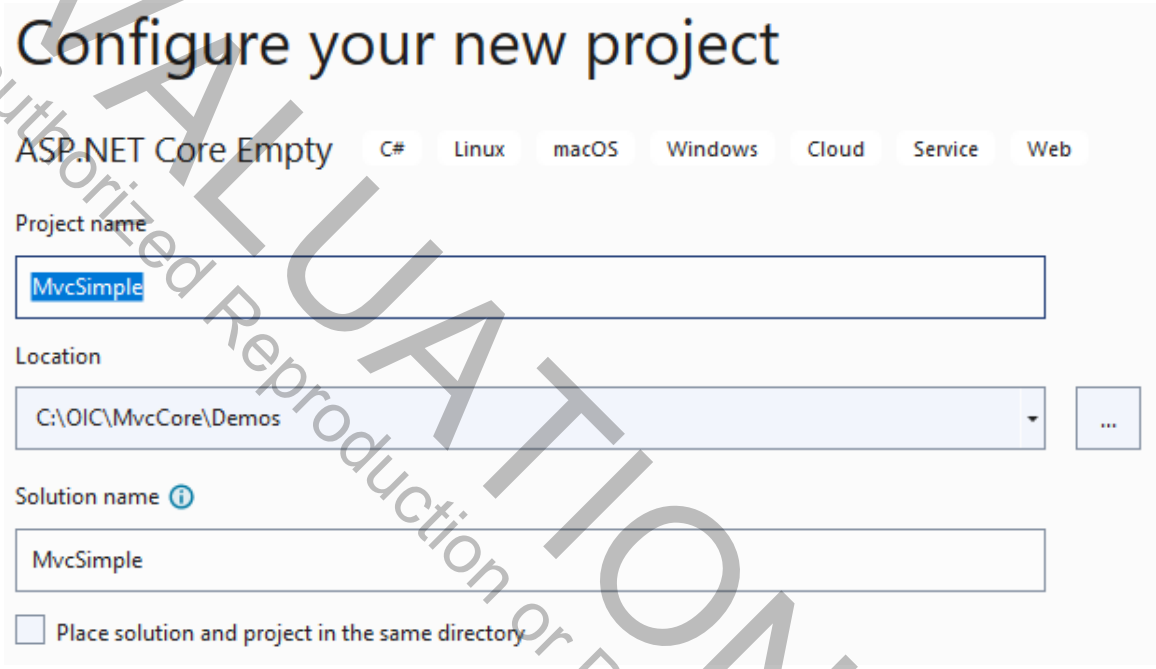
1. Create a new project within Visual Studio from the menu File | New | Project ...
2. This time choose the ASP.NET Core Empty project template.



3. Click Next.

Demo: Controller Only (Cont'd)

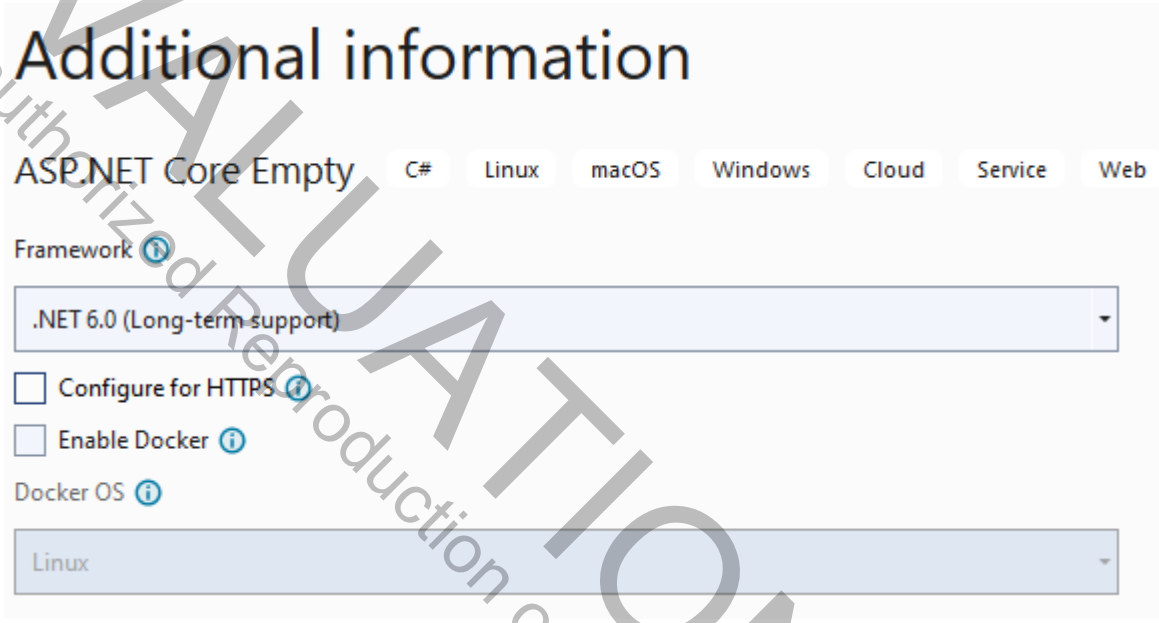
4. Type **MvcSimple** as the project name. The Location should again be the **Demos** folder.



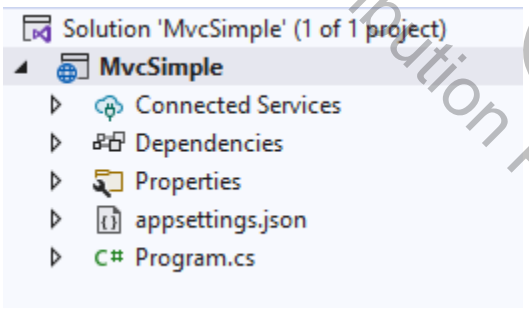
5. Click Next.

Demo: Controller Only (Cont'd)

6. Choose .NET 6.0 (Long-term support) as the Target Framework, and clear the HTTPS checkbox.

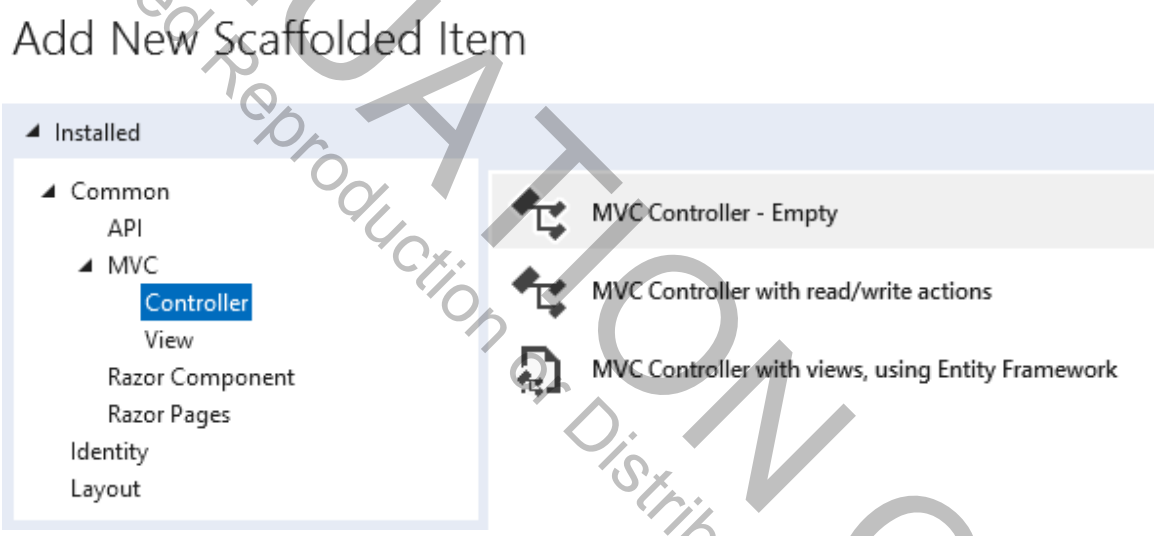


7. Click Create. The generated project is much simpler than before!

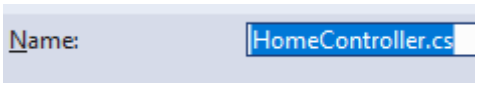


Demo: Controller Only (Cont'd)

8. Add a Controllers folder to the project.
9. Build the project.
10. Right-click over the Controllers folder and choose Add | Controller ... from the context menu.
11. Choose MVC Controller – Empty.



12. Click Add.
13. Specify **HomeController.cs** as the name of the new controller.



14. Click Add.

Demo: Controller Only (Cont'd)

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

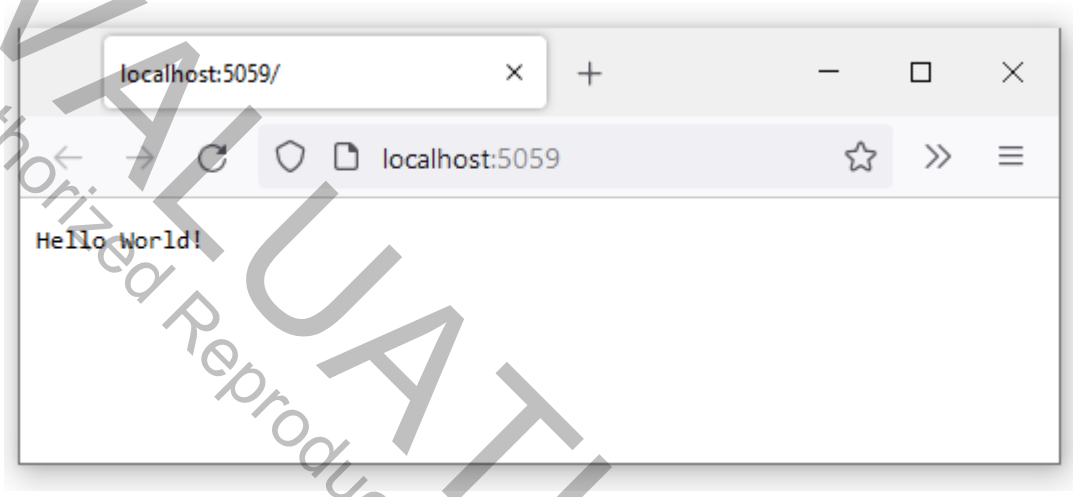
```
using Microsoft.AspNetCore.Mvc;

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

16. 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";
    }
}
```

Demo: Controller Only (Cont'd)

17. Build and run.



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

19. Examine the file **Program.cs**.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Edit Program.cs

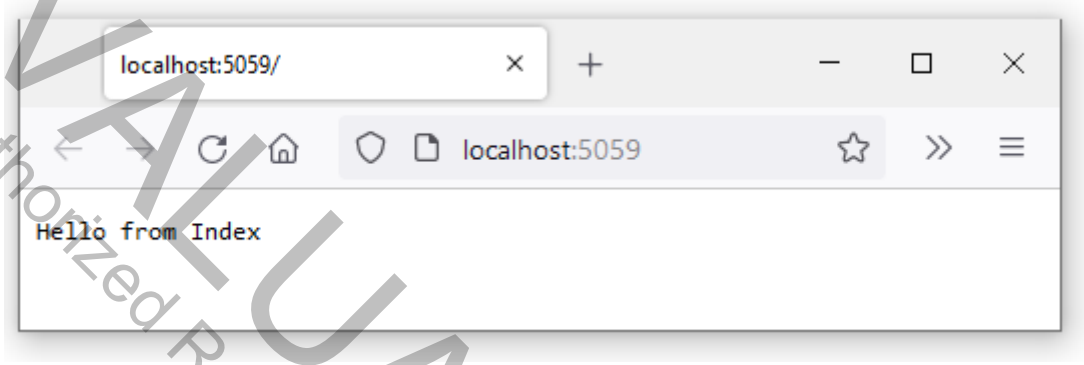
20. Examine the file **Program.cs** in the **WebApplication1** example. This will give us a clue for fixing our new example.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

app.UseRouting();

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

app.Run();
```

- **C# 9 introduced the concept of *top-level statements*.**
  - You don't have to explicitly include a **Main()** method in a **Program** class.
  - The compiler will generates a class and **Main()** method entry point for the application.
- **.NET 6.0 introduces *Implicit Using Directives*.**
  - The compiler automatically adds a set of using directives based on the project type.
- **Thus the code shown above is complete, much more concise than in previous versions of .NET.**

Demo: Controller Only (Cont'd)

21. Build and run. Success!



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

```
http://localhost:5059/
```

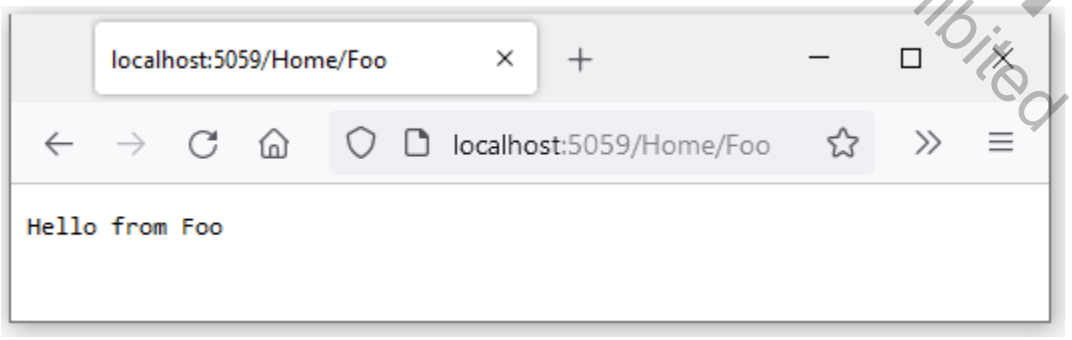
23. Now try using these URLs[1]. You should get the same result.

```
http://localhost:5059/Home/
http://localhost:5059/Home/Index/
```

24. Now try this URL.

```
http://localhost:5059/Home/Foo
```

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



[1] The trailing forward slash in these URLs is optional. Also, the Firefox browser does not http://.

Demo: Controller Only (Cont'd)

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

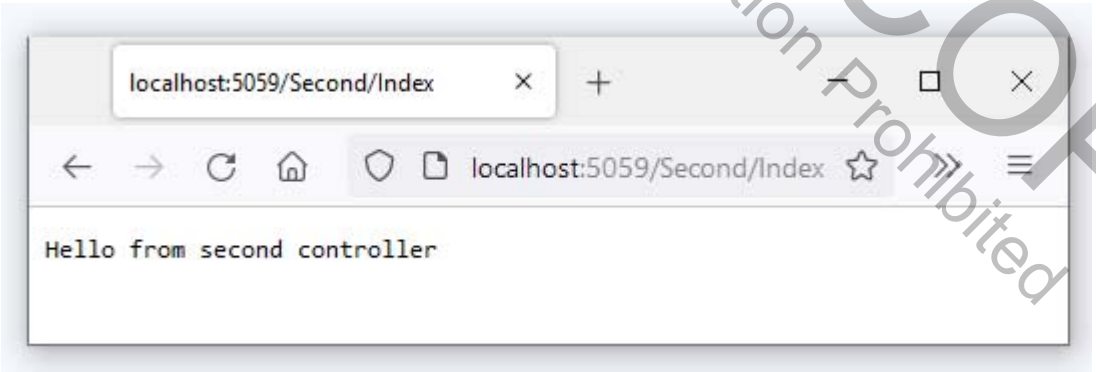
26. Provide the following code for the **Index()** method of the second controller.

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

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

```
http://localhost:5059/Second
http://localhost:5059/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 by a call to the the *MapControllerRoute()* method in the *Program.cs* file.**

```
app.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.**

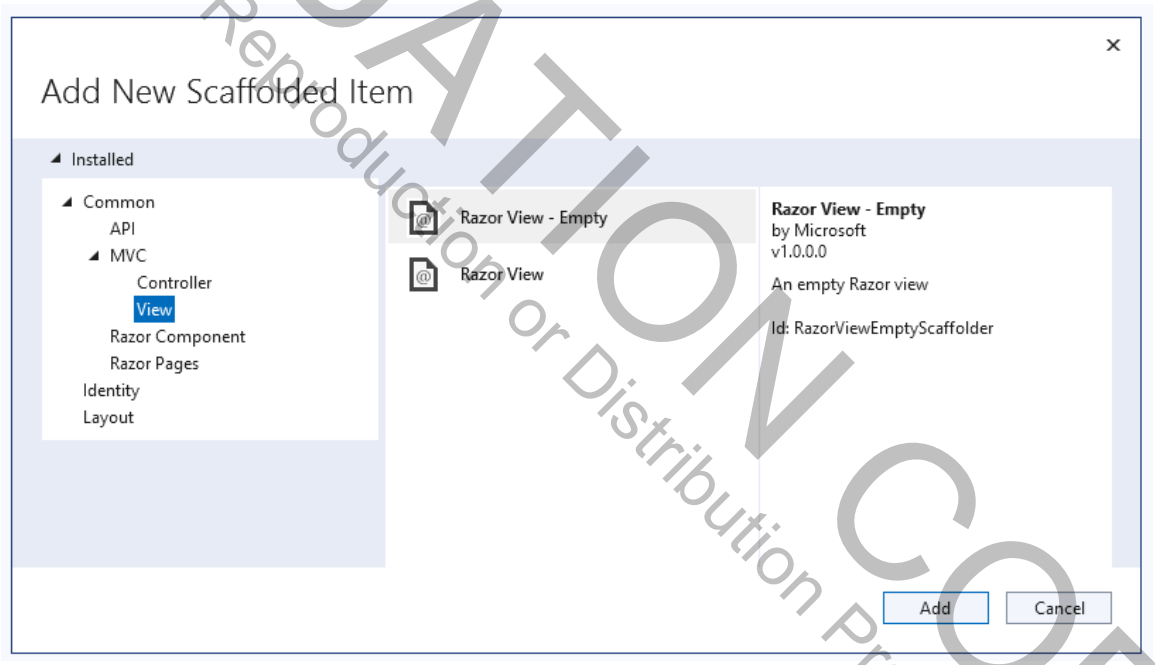
An unhandled exception occurred while processing the request.

InvalidOperationException: The view 'Index' was not found. The following locations were searched:
/Views/Home/Index.cshtml
/Views/Shared/Index.cshtml

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 | View ... from the context menu.



3. Select Razor View – Empty and click Add.

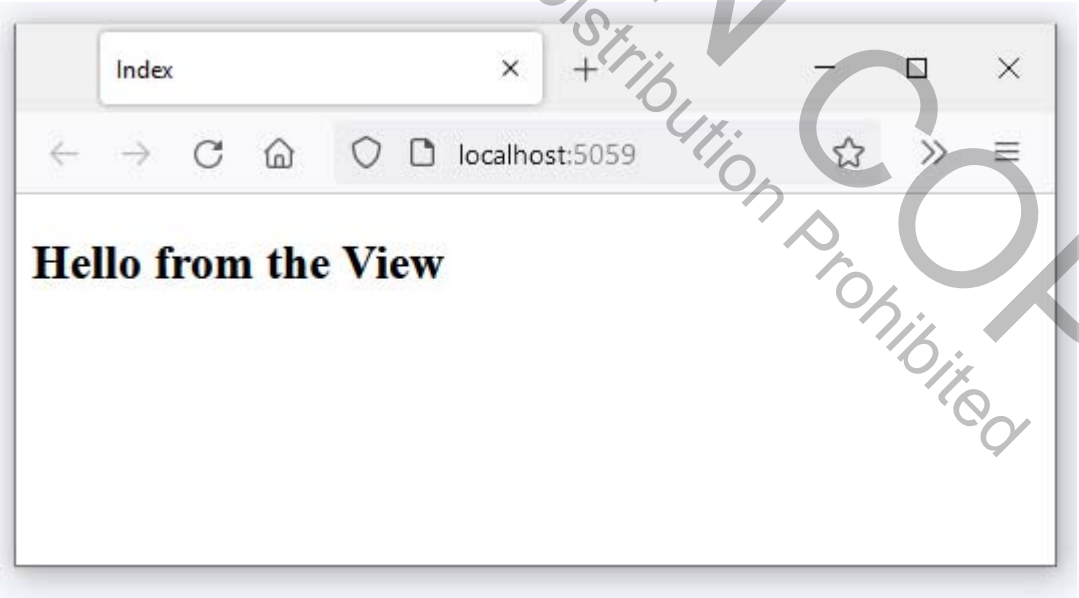
The View Web Page

- **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.**

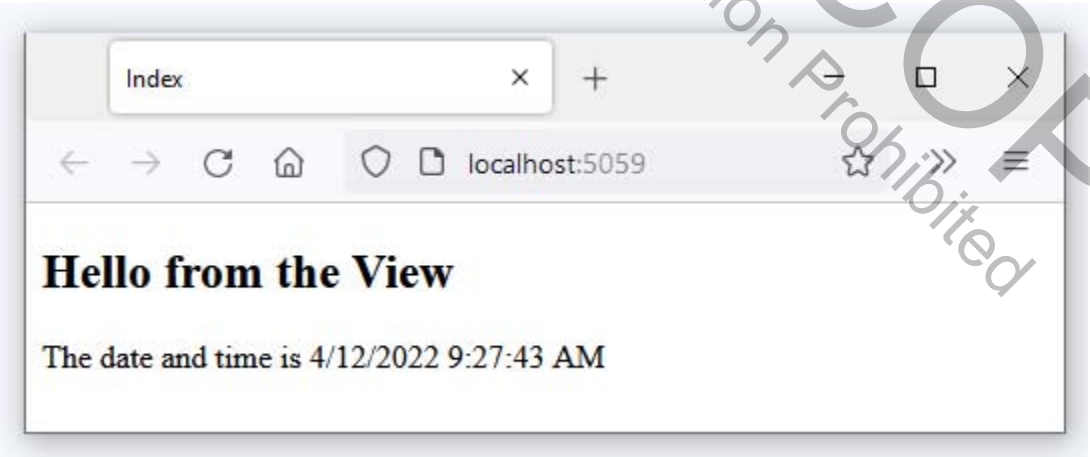
```
public ViewResult Index()
{
    ViewBag.Time = DateTime.Now.ToLocalTime();
    return View();
}
```

  - The **System** namespace is needed and is imported implicitly. See Appendix A.

- **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 6.0, 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 beginning 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 />
  }
}
```

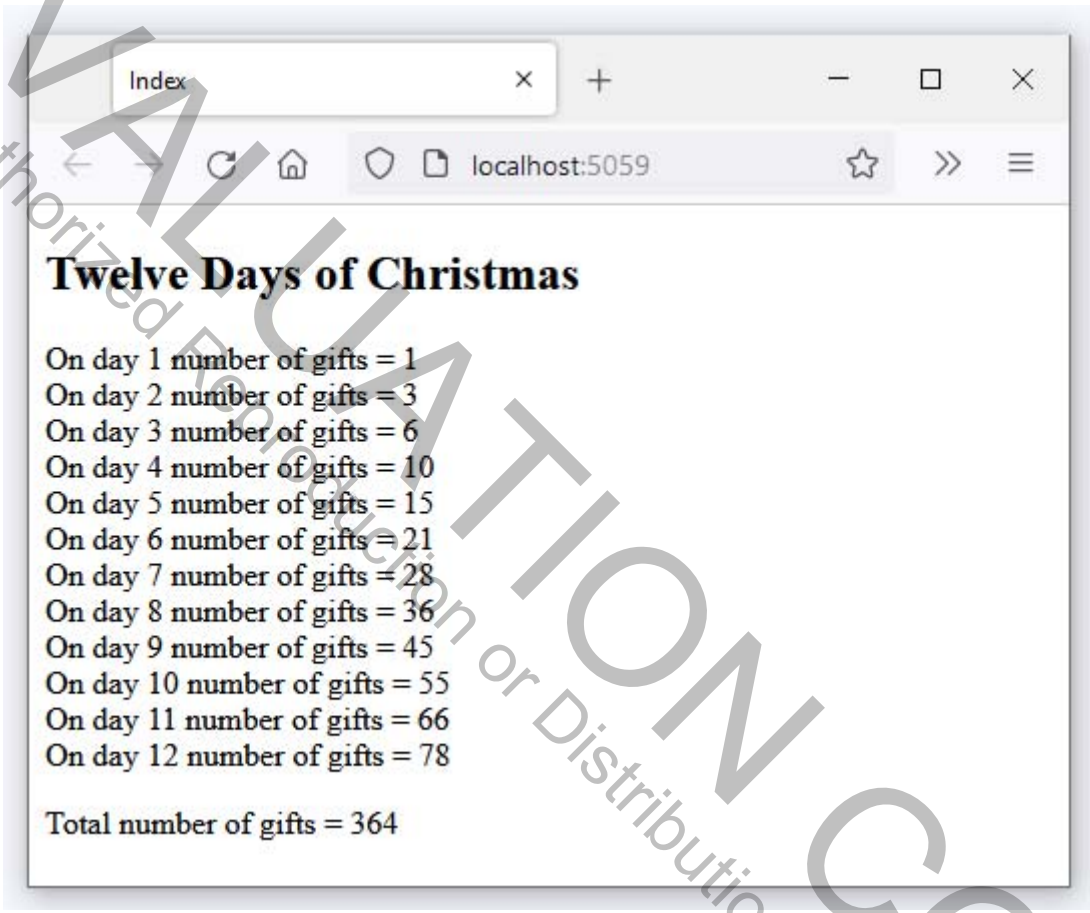
Embedded Script Example

- **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.

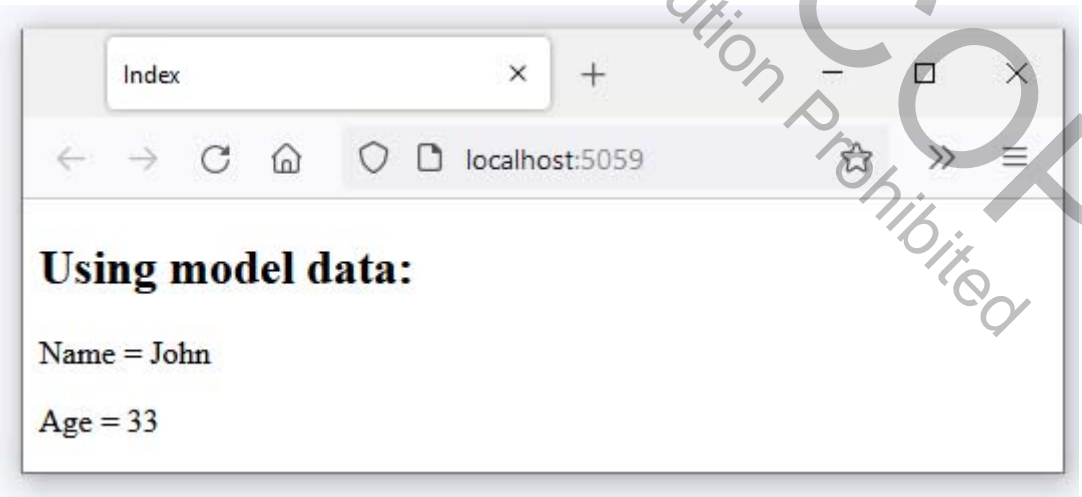
```
using Microsoft.AspNetCore.Mvc;
using MvcSimple.Models;

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:



  - The example is in **MvcSimple\ModelViewBag**.

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\New*.**
  - Pass the name in the query string, for example:

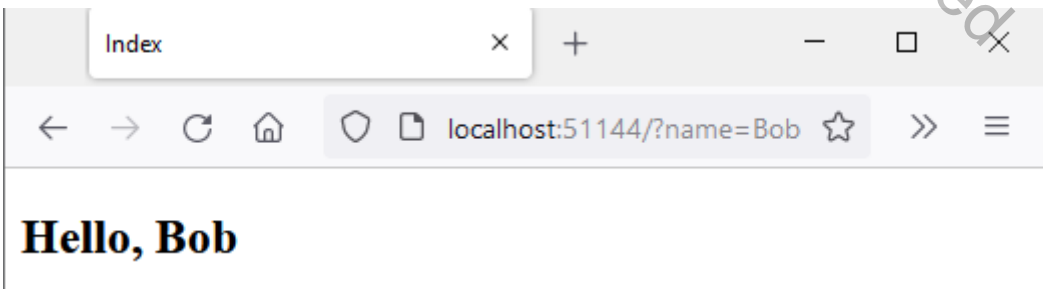
`/Home/Index?name=Bob` or just `/?name=Bob`

  - 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>
```



New and Old Project Templates

- **In creating new ASP.NET Core Web apps, we have been using the new project template.**
  - The generated code uses new features in C# and .NET 6, such as top-level statements and implicit usings, as discussed earlier in the chapter.
- **Many of the examples in this course were created width the old project template.**
  - For example, see **MvcHello\Old** in the chapter folder.
  - Configuration is established in the file **Startup.cs**.

```
public void ConfigureServices(
                      IServiceCollection services)
{
   services.AddControllersWithViews();
}
public void Configure(IApplicationBuilder app,
                      IWebHostEnvironment env)
{
   if (env.IsDevelopment())
   {
       app.UseDeveloperExceptionPage();
   }
   app.UseRouting();
   app.UseEndpoints(endpoints =>
   {
       endpoints.MapControllerRoute(
           name: "default",
           pattern:
"{controller=Home}/{action=Index}/{id?}");
   });
}
```

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 **Program.cs** as done in the **MvcSimple** example.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

app.UseRouting();

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

app.Run();
```

Or, you may simply copy **Program.cs** from the **MvcSimple** example!

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();
    }
}
```

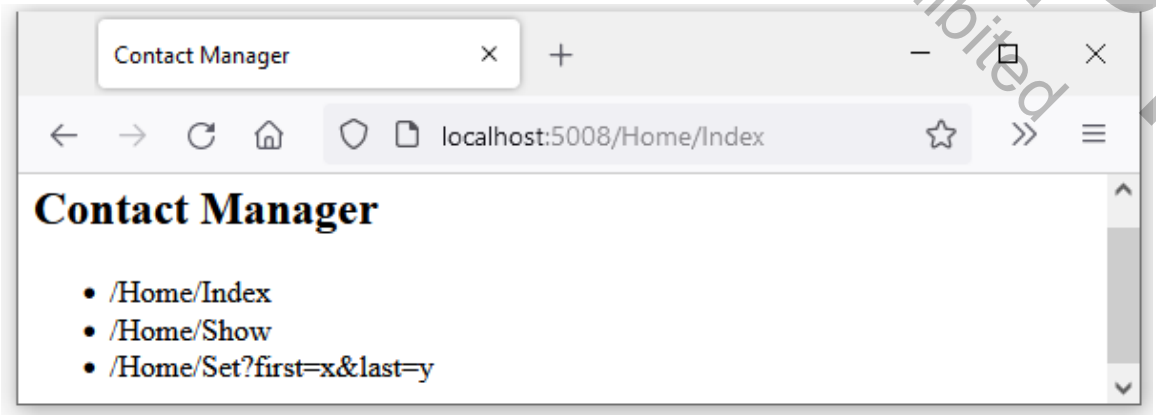
5. Right-click over the **Views\Home** folder and add a new item, a Razor View **Index.cshtml**.

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>
```

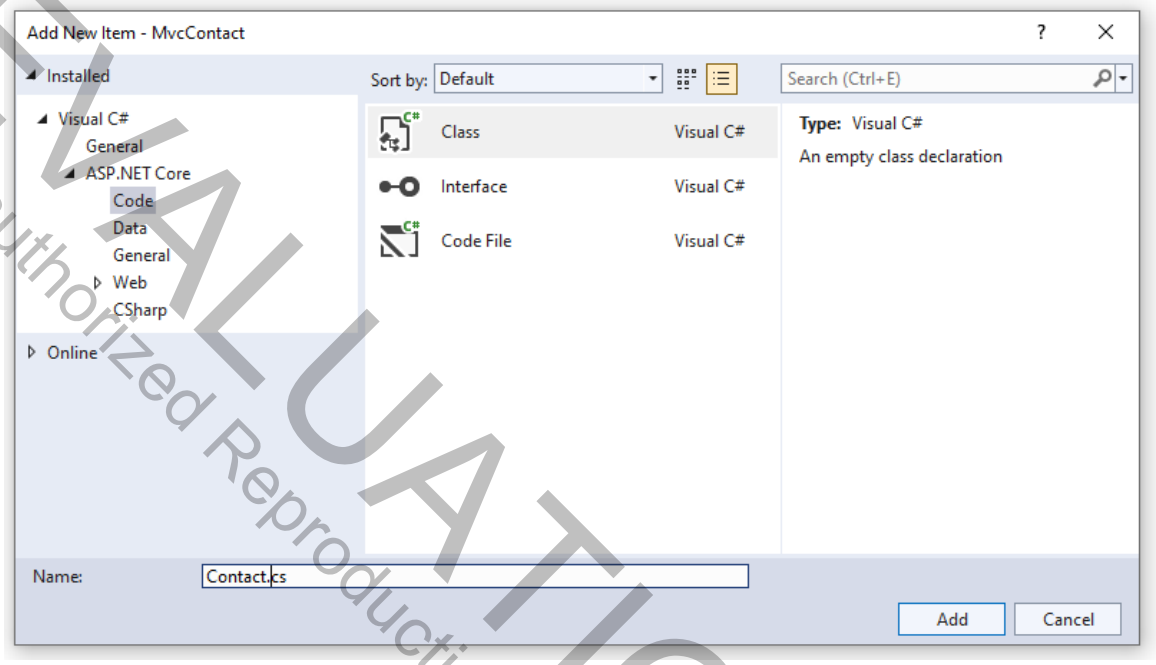
7. Build and run. You should get output like that shown below. For example, try this alternate URL for the index page (your port number will be different).

```
http://localhost:5008/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:5008/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.

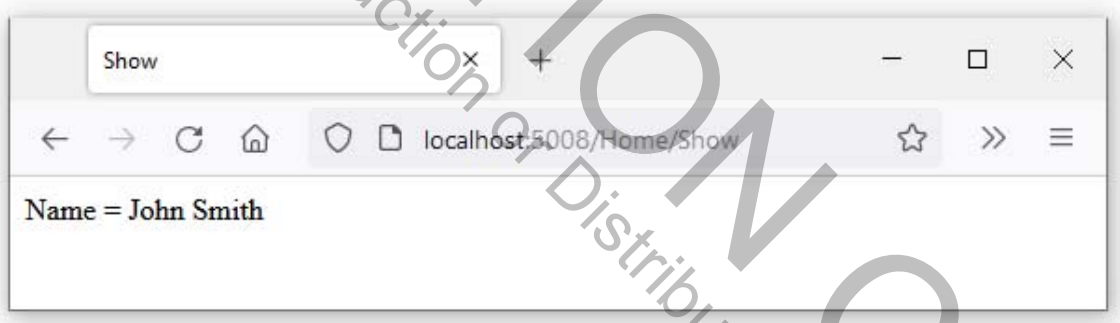
1. Examine the File **Contact.cs** in the **Lab2** folder. It defines an enhanced **Contact** class in which the initial data is read from a file rather than hardcoded. There is also a method.

```
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.

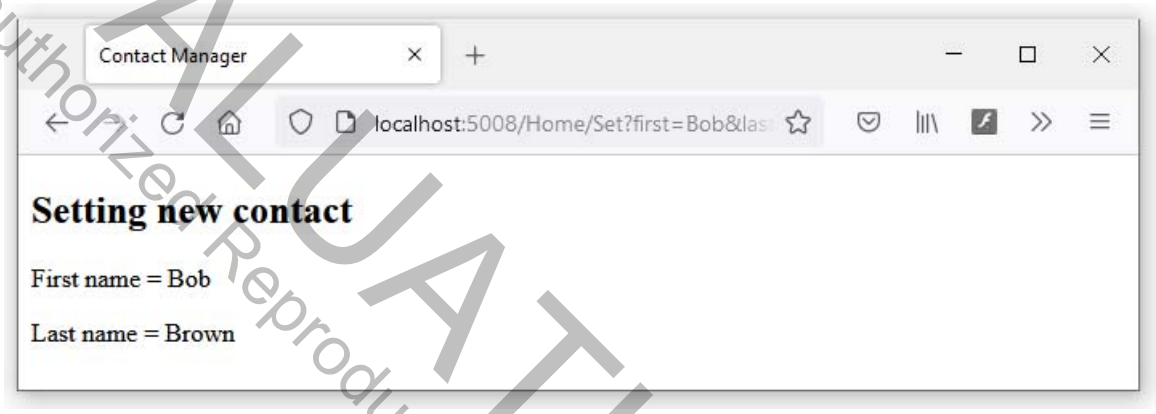
```
// GET: /Home/Set?first=x&last=y
public ActionResult Set(string first, string last)
{
   Contact.WriteContact(first, last);
   ViewBag.First = first;
   ViewBag.Last = last;
   return View();
}
```

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.

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
Ph: 303-302-5280
www.ITCourseware.com