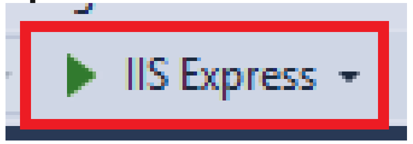
Unit 4

Creating ASP.NET Core MVC Applications

Creating a Web App & Run

- From the Visual Studio, select Create a new project.
- Select ASP.NET Core Web Application and then select Next.
- Name the project as you like or WebApplicationCoreS1
- Choose the location path to save your project.
- Click Create
- Select Web Application(Model-View-Controller), and then select Create.
- Now, To run the App,
  - Select **Ctrl-F5** to run the app in non-debug mode, Or
  - Select IIS Express Button.



Setting up the Environment

ASP.NET Core wwroot Folder

- By default, the wwwroot folder in the ASP.NET Core project is treated as a web root folder. Static files can be stored in any folder under the web root and accessed with a relative path to that root.
- In ASP.NET Core, only those files that are in the web root - wwwroot folder can be served over an http request. All other files are blocked and cannot be served by default.
- Generally, we find static files such as JavaScript, CSS, Images, library scripts etc. in the wwwroot folder
- You can access static files with base URL and file name. For example, for css folder, we can access via http://localhost:<port>/css/app.css

Setting up the Environment

ASP.NET Core – Program.cs Class

- ASP.NET Core web application project starts executing from the entry point - public static void Main() in Program class.

ASP.NET Core – Startup.cs Class

- It is like Global.asax in the traditional .NET application. As the name suggests, it is executed first when the application starts.
- The startup class can be configured at the time of configuring the host in the Main() method of Program class.

Add a controller

- In Solution Explorer, right-click Controllers > Add > Controller
- In the Add Scaffold dialog box, select Controller Class – Empty
- In the Add Empty MVC Controller dialog, enter HelloWorldController and select Add.
- Replace the contents of Controllers/HelloWorldController.cs

```
public string Index() {
        return "This is my default action...";
}
public string Welcome() {
        return "This is the Welcome action method...";
}
```

Add a controller

- MVC invokes controller classes (and the action methods within them) depending on the incoming URL.
- The default URL routing logic used by MVC uses a format like this:

  /[Controller]/[ActionName]/[Parameters]
- The routing format is set in the Configure method in Startup.cs file.

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

- Run your app & Check with these url in your browser
  - https://localhost:{PORT}/HelloWorld
  - https://localhost:{PORT}/HelloWorld/Index
  - https://localhost:{PORT}/HelloWorld/Welcome
- Make changes for Welcome Method like this:

```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

- Check on your browser with these:
  - https://localhost:{PORT}/HelloWorld/Welcome?name=AAA&numtimes=4

- Make changes again for Welcome Method with following code

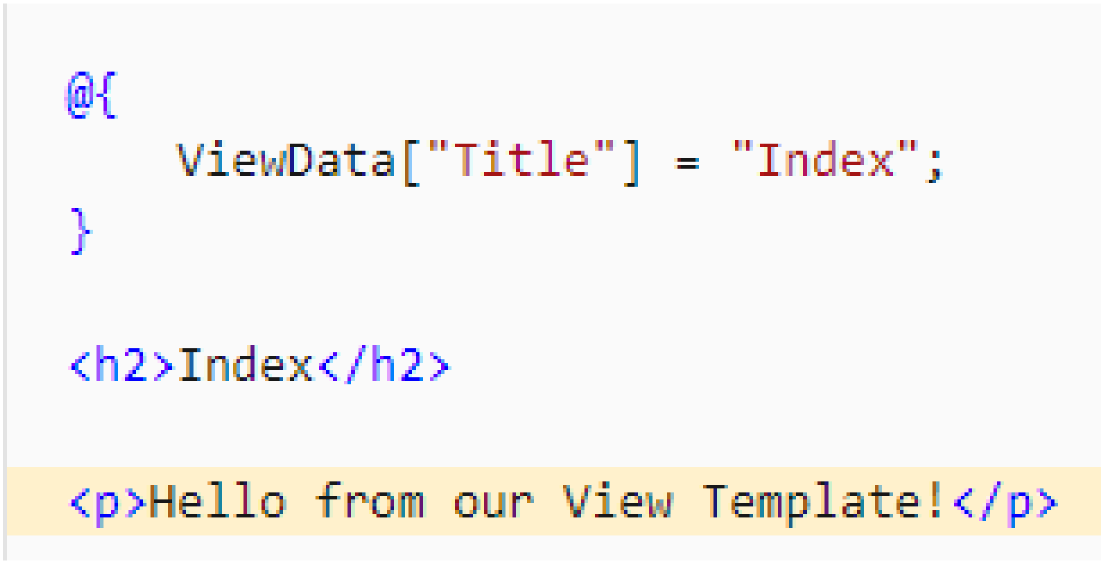
```
public string Welcome(string name, int ID = 1) {
    return HtmlEncoder.Default.Encode($"Hello {name},  ID is: {ID}");
}
```

- Check on your browser with these:
  - http://localhost:{PORT}/HelloWorld/Welcome/3?name=AAA
- Here, the third URL segment matched the route parameter id. The Welcome method contains a parameter id that matched the URL template in the MapControllerRoute method in Startup.cs file. The trailing ? (in id?) indicates the id parameter is optional.

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

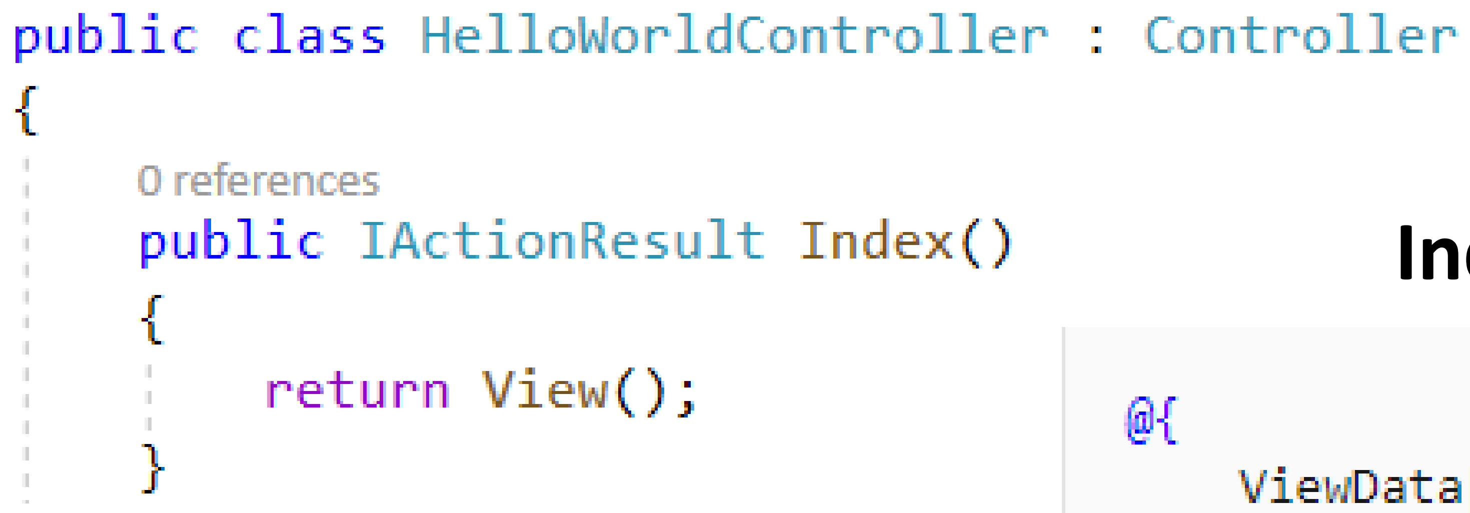
Add a view

- In your Project, Right click on the *Views* folder, and then **Add > New Folder** and name the folder *HelloWorld*.
- Right click on the *Views/HelloWorld* folder, and then **Add > New Item**.
- In the **Add New Item** dialog
    - In the search box in the upper-right, enter *view*
    - Select **Razor View**
    - Keep the **Name** box value, *Index.cshtml*.
    - Select **Add**
- Replace the contents of the Views/HelloWorld/Index.cshtml Razor view file with the following



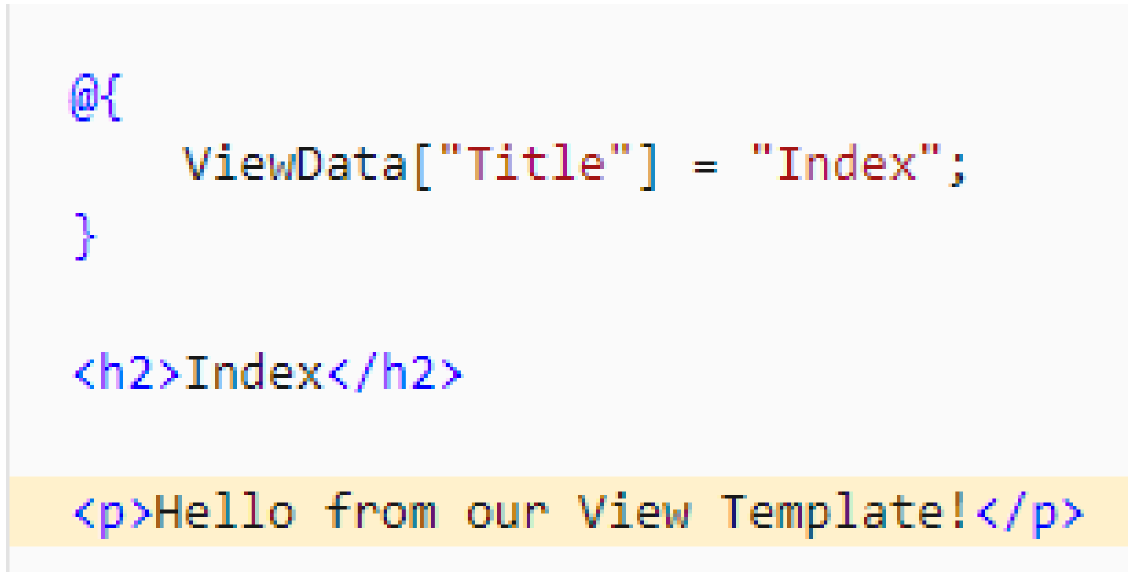
```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Your Controller and View



```
public class HelloWorldController : Controller
{
    0 references
    public IActionResult Index()
    {
        return View();
    }
```

Index.cshtml



```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

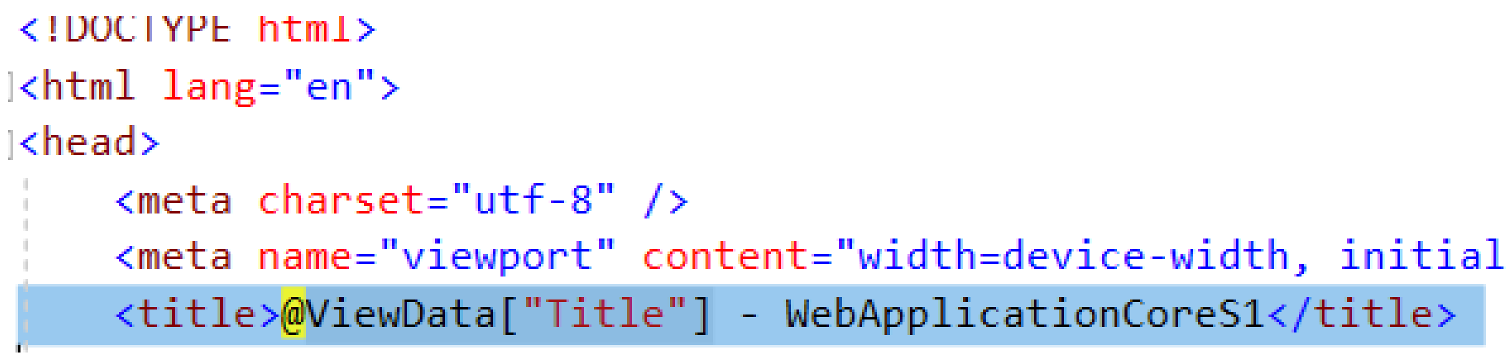
- Navigate to https://localhost:{PORT}/HelloWorld

Change views and layout pages

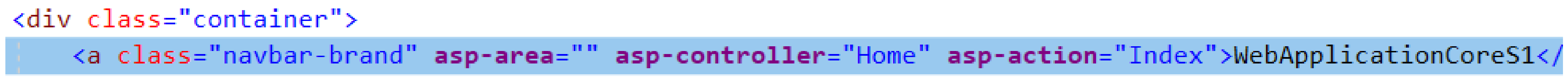
- In page, Select the menu links (WebApplicationCoreS1, Home, Privacy)
- Each page shows the same menu layout.
- The menu layout is implemented in the Views/Shared/_Layout.cshtml file.
- Open the Views/Shared/_Layout.cshtml file.
- Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site.
- Find the @RenderBody() line. RenderBody is a placeholder where all the view-specific pages you create show up, wrapped in the layout page.
- For example, if you select the Privacy link, the Views/Home/Privacy.cshtml view is rendered inside the RenderBody method.

Change the title, footer, and menu link in the layout file

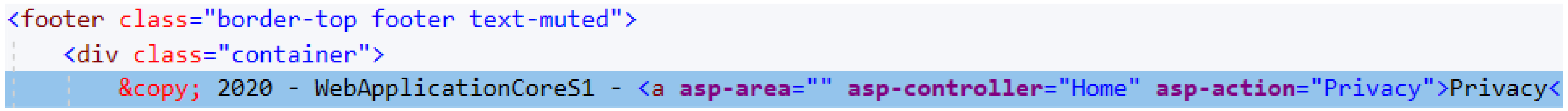
- Replace the content of the Views/Shared/_Layout.cshtml file with the following markup. The changes are highlighted:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebApplicationCoreS1</title>
```



```
<div class="container">
    <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">WebApplicationCoreS1</
```



```
<footer class="border-top footer text-muted">
    <div class="container">
        &copy; 2020 - WebApplicationCoreS1 - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy<
```

Passing Data from the Controller to the View

- Controllers are responsible for providing the data required in order for a view template to render a response.
- In HelloWorldController.cs, change the Welcome method to add a Message and NumTimes value to the ViewData dictionary.
- The ViewData dictionary is a dynamic object, which means any type can be used; the ViewData object has no defined properties until you put something inside it. The MVC model binding system automatically maps the named parameters (name and numTimes) from the query string in the address bar to parameters in your method.
- Ex looks like :

Passing Data from the Controller to the View

```
public class HelloWorldController : Controller
{
    public IActionResult Index()
    {
        return View();
    }

    public IActionResult Welcome(string name, int numTimes = 1)
    {
        ViewData["Message"] = "Hello " + name;
        ViewData["NumTimes"] = numTimes;

        return View();
    }
}
```

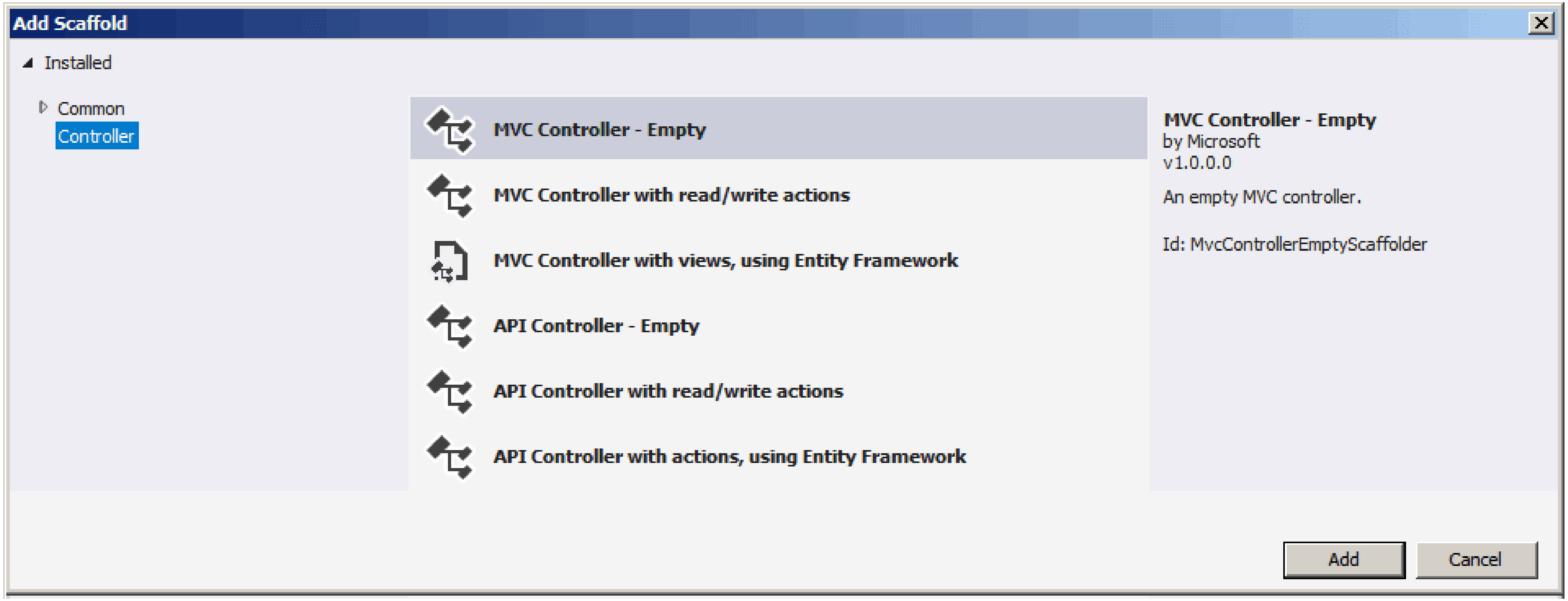
Passing Data from the Controller to the View

```
@{
    ViewData["Title"] = "Welcome";
}

<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Controllers Responsibities

- Controllers are usually placed in a folder called "Controllers", directly in the root of your MVC project.
- They are usually named based on their purpose, with the word "Controller" as a suffix.
- The Controller has three major responsibilities
  - Handle the Request from the User
  - Build a Model – Controller Action method executes the application logic and builds a model
  - Send the Response – it returns the Result in HTML/File/JSON/XML or other format as requested by user.

Controller Scaffolding Option



Add Scaffold

Installed

- Common
- Controller

- MVC Controller - Empty
- MVC Controller with read/write actions
- MVC Controller with views, using Entity Framework
- API Controller - Empty
- API Controller with read/write actions
- API Controller with actions, using Entity Framework

MVC Controller - Empty
by Microsoft
v1.0.0.0

An empty MVC controller.

Id: MvcControllerEmptyScaffolder

Controller Scaffolding Option

- Both the MVC and API Controller inherits from the same Controller class and there is not much difference between them, except that API Controller is expected to return the data in serialized format to the client.
- Further, we have three options under both types of controllers.
  - Empty
  - With Read/Write Actions
  - With Views, using entity framework

Actions

- Controller is just a regular .NET class, it can have fields, properties and methods.
- Methods of a Controller class is referred to as **actions** - a method usually corresponds to an action in your application, which then returns something to the browser/user.
- All public methods on a Controller class is considered an Action.
- For instance, the browser might request a URL like /Products/Details/1 and then you want your ProductsController to handle this request with a method/action called Details.

When creating Action Method, points to remember

- Action methods Must be a public method
- The Action method cannot be a Static method or an Extension method.
- The Constructor, getter or setter cannot be used.
- Inherited methods cannot be used as the Action method.
- Action methods cannot contain ref or out parameters.
- Action Methods cannot contain the attribute [NonAction].
- Action methods cannot be overloaded

Actions Verbs

- To gain more control of how your actions are called, you can decorate them with the so-called Action Verbs.
- an action can be accessed using all possible HTTP methods (the most common ones are GET and POST)
- Edit action can be accessed with a GET request.

```
[HttpGet]
public IActionResult Edit()
{
  return View();
}
```

Actions Verbs

```
[HttpGet]
public IActionResult Edit()
{
  return Content("Edit");
}
[HttpPost]
public IActionResult Edit(Product product)
{
  product.Save();
  return Content("Product Updated!");
}
```

Actions Result Types

- When the Action (method) finishes it work, it will usually return something, as IActionResult interface
- Some list of Action Result are:
  - Content() - returns specified string as plain text
  - View() - returns a View to the client
  - PartialView() - returns a Partial View to the client
  - File() - returns the content of a specified file to the client
  - Json() - returns a JSON response to the client
  - Redirect() and RedirectPermanent() - returns a redirect response to the browser (temporary or permanent), redirecting the user to another URL
  - StatusCode() - returns a custom status code to the client

Actions Result - Ex

- A common use case for this is to return either a View or a piece of content if the requested content is found, or a 404 (Page not Found) error if its not found. It could look like this:

```
public IActionResult Details(int id)
{
    Product product = GetProduct(id);
    if (product != null)
        return View(product);
    return NotFound();
}
```

Rendering HTML with Views

- In MVC pattern, the view handles the app's data presentation and user interaction.
- A view is an HTML template with embedded Razor markup. Razor markup is code that interacts with HTML markup to produce a webpage that's sent to the client
- In ASP.NET Core MVC, views are .cshtml files that use the C# programming language in Razor markup. Usually, view files are grouped into folders named for each of the app's controllers. The folders are stored in a Views folder at the root of the app

Rendering HTML with Views

- The Home controller is represented by a Home folder inside the Views folder. The Home folder contains the views for the About, Contact, and Index (homepage) webpages. When a user requests one of these three webpages, controller actions in the Home controller determine which of the three views is used to build and return a webpage to the user.
- Use layouts to provide consistent webpage sections and reduce code repetition. Layouts often contain the header, navigation and menu elements, and the footer. The header and footer usually contain boilerplate markup for many metadata elements and links to script and style assets. Layouts help you avoid this boilerplate markup in your views.

Creating a View

- To create a view, add a new file and give it the same name as its associated controller action with the .cshtml file extension.
- For Ex - For About action in the Home controller, create an About.cshtml file in the Views/Home folder:

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>
<p>Use this area to provide additional information.</p>
```

Creating a View

- Razor markup starts with the @ symbol.
- Your can write C# code within Razor code blocks set off by curly braces ({ ... }).
- You can display values within HTML by simply referencing the value with the @ symbol. See the contents of the <h2> and <h3> elements above.

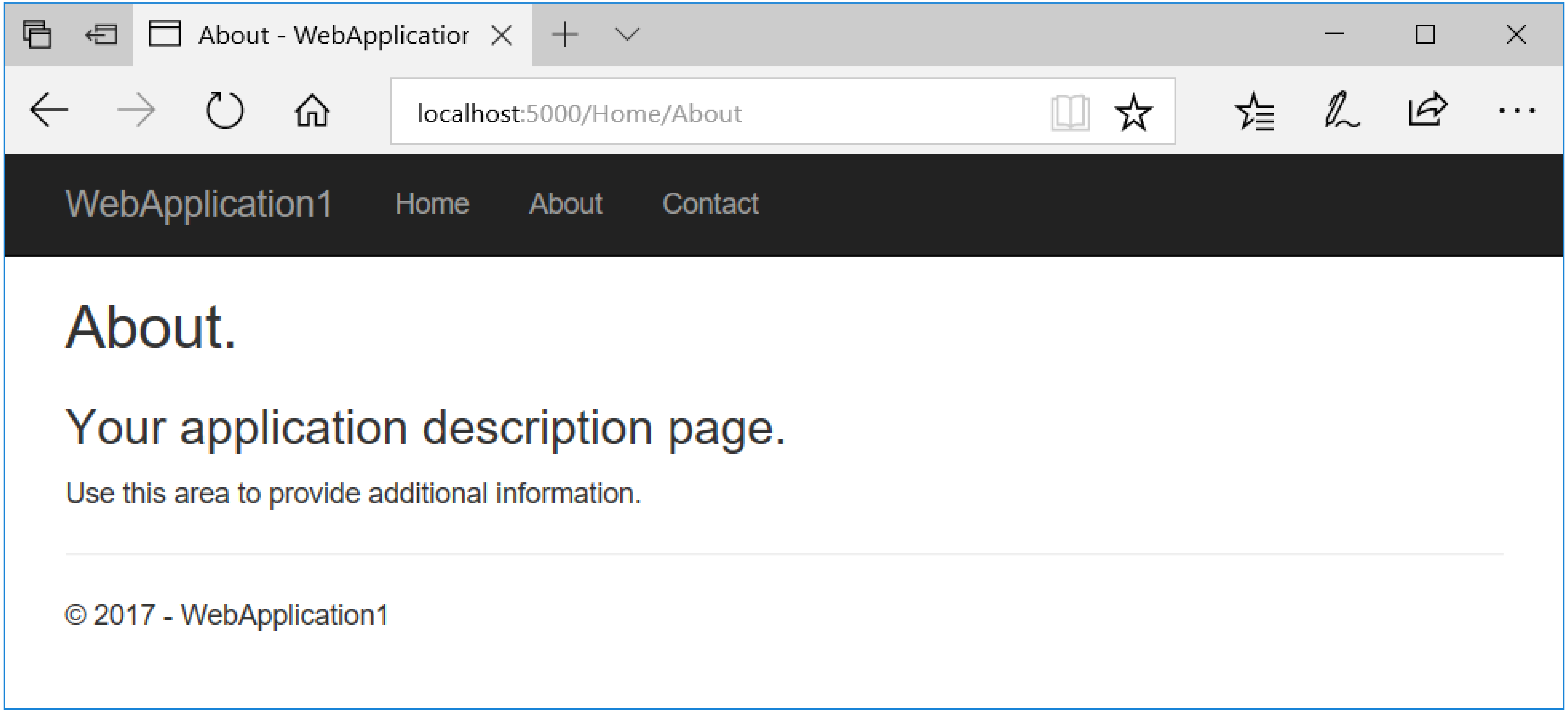
How Controllers Specify Views

- Views are typically returned from actions as a ViewResult, which is a type of ActionResult.

HomeController.cs

```
public IActionResult About()
{
        ViewData["Message"] = "Your application description page.";
        return View();
}
```

How Controllers Specify Views



About.

Your application description page.

Use this area to provide additional information.

© 2017 - WebApplication1

Passing Data to Views: ViewData & ViewBag

- Views have access to a weakly typed(loosely typed) collection of data. You can use these for passing small amounts of data in and out of controllers and views
- The ViewData property is a dictionary of weakly typed objects.
- The ViewBag property is a wrapper around ViewData that provides dynamic properties for the underlying ViewData collection.
- ViewData and ViewBag are dynamically resolved at runtime.
- ViewData is a ViewDataDictionary object accessed through string keys

Ex - ViewData

```
// HomeController.cs
public class HomeController : Controller {
    public IActionResult About() {
        ViewData["Message"] = "Your application description page.";
        return View();
    }
}
// About.cshtml
   @{
       ViewData["Title"] = "About";
   }
   <h3>@ViewData["Message"]</h3>
```

Ex - ViewBag

```
// HomeController.cs
public class HomeController : Controller {
    public IActionResult SomeAction() {
        ViewBag.Greeting = “Hello";
        return View();
    }
}
//SomeAction.cshtml
   @{
       ViewData["Title"] = “My Title";
   }
   <h3>@ViewBag.Greeting</h3>
```

Razor Syntax

- The biggest advantage of the Razor is the fact that you can mix client-side markup (HTML) with server-side code (e.g C# or VB.NET), without having to explicitly jump in and out of the two syntax types.
- In Razor, you can reference server-side variables etc. by simply prefixing it with an at-character (@).

```
<p>Hello, the current date is: @DateTime.Now.ToString()</p>
```

Ex - Razor & HTML Encoding

```
@{
    var helloWorld = "<b>Hello, world!</b>";
}
<p>@helloWorld</p>
<p>@Html.Raw(helloWorld)</p>
```

Ex – Razor Explicit Expressionsc

```
@{
    var name = "John Doe";
}
Hello, @(name.Substring(0,4)).
Your age is: <b>@(37 + 5).</b>
```

Ex – Multi-statement Razor blocks

```
@{
   var sum = 32 + 10;
   var greeting = "Hello, world!";
   var text = "";
   for(int i = 0; i <3; i++)
   {
      text += greeting + " The result is: " + sum + "\n";
   }
}
<h2>CodeBlocks</h2>
Text: @text
```

Razor Server-side Comments

- Sometimes you may want to leave comments in your code, or comment out lines of code temporarily to test things.

```
@*
    Here's a Razor server-side comment
    It won't be rendered to the browser
*@
```

Razor Server-side Comments

- If you're inside a Razor code block, you can even use the regular C# style comments:

```
@{
    @*
        Here's a Razor server-side comment
    *@
    // C# style single-line comment
    /*
    C# style multiline comment
    It can span multiple lines
    */
}
```

Razor Syntax – Variables and Expressions

```
@{
    string helloWorldMsg = "Good day";
    if(DateTime.Now.Hour >17){
        helloWorldMsg = "Good evening";
    }
    helloWorldMsg += ", world!";
    helloWorldMsg = helloWorldMsg.ToUpper();
}
<div> @helloWorldMsg </div>
```

Razor Syntax – The if-else statement

```
@if(DateTime.Now.Year >= 2042)
{
    <span>The year 2042 has finally arrived!</span>
}
else
{
    <span>We're still waiting for the year of 2042...</span>
}
```

Razor Syntax – Loop

```
@{
    List<string> names = new List<string>() {
    "VB.NET", "C#", "Java"
    };
}
<ul>
   @for (int i = 0; i < names.Count; i++)
   {
       <li>@names[i]</li>
   }
</ul>
<ul>
@foreach (string name in names)
{
    <li>@name</li>
}
</ul>
```

Understanding Tag Helpers

- Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. Tag helpers are a new feature and similar to HTML helpers, which help us render HTML.