

**TRAINING & REFERENCE**

# **murach's ASP.NET Core MVC 2ND EDITION**

## **(Chapter 2)**

Thanks for downloading this chapter from [\*Murach's ASP.NET Core MVC \(2nd Edition\)\*](#). We hope it will show you how easy it is to learn from any Murach book, with its paired-pages presentation, its “how-to” headings, its practical coding examples, and its clear, concise style.

To view the full table of contents for this book, you can go to our [website](#). From there, you can read more about this book, you can find out about any additional downloads that are available, and you can review our other books on related topics.

Thanks for your interest in our books!



**MIKE MURACH & ASSOCIATES, INC.**

1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963

[murachbooks@murach.com](mailto:murachbooks@murach.com) • [www.murach.com](http://www.murach.com)

*Copyright © 2023 Mike Murach & Associates. All rights reserved.*

## What developers have said about the previous edition

---

“Not only is this a great book on ASP.NET Core MVC, but it’s the best programming book I’ve read on any subject. I love the format with the paired pages. The code is error-free and the numerous example applications are helpful to study. The explanations are clear and avoid some of the overly complicated word salads other books like to give.”

Posted at an online bookseller

“I normally don’t write in or leave reviews on products, but I felt compelled to do so with this book as it’s the first one I went through that was actually fun from start to finish (and I’ve read many, many programming books before).”

Shannon Fairchild, Senior Software Developer,  
Kingston, Ontario, Canada

“I have been programming in ASP.NET Webforms and MVC for over 15 years and this book is hands-down the best book programming book I’ve ever purchased.”

Posted at an online bookseller

“This book is a solid introduction into ASP.NET Core and can really help a beginner get up to speed in this framework. You can get a lot out of doing the exercises.”

Posted at an online bookseller

“Your launch of the ASP.NET Core MVC text is a much-needed contribution to the topic of ASP.NET Core.”

K.S., College Instructor, Illinois

“At last, a dot net MVC book that explains everything clearly!”

Posted at an online bookseller

# How to develop a one-page MVC web app

In the last chapter, you were introduced to the basic concepts of web programming and ASP.NET Core MVC. Now, this chapter shows you how to develop a one-page ASP.NET Core MVC web app that calculates the future value of a series of investments. To do that, this chapter shows how to use Visual Studio because it's the most established tool for working with .NET apps. However, if you prefer to use Visual Studio Code instead, you can refer to chapter 18 to learn how to work with Visual Studio Code.

<b>How to create a Core MVC web app .....</b>	<b>38</b>
How to start a new web app.....	38
How to configure a web app .....	40
How to set up the MVC folders .....	42
How to add a controller .....	44
How to add a Razor view.....	46
How to configure the middleware for an HTTP request pipeline.....	48
<b>How to run a web app and fix errors .....</b>	<b>50</b>
How to run a web app .....	50
How to find and fix errors.....	52
<b>How to work with a model .....</b>	<b>54</b>
How to add a model .....	54
How to add a Razor view imports page.....	56
How to code a strongly-typed view .....	58
How to handle GET and POST requests .....	60
The Future Value app after handling GET and POST requests.....	62
<b>How to organize the files for a view .....</b>	<b>64</b>
How to add a CSS style sheet .....	64
How to add a Razor layout and view start.....	66
The code for a Razor layout and view start.....	68
How to add a Razor view when using a layout with a view start.....	70
<b>How to validate user input.....</b>	<b>72</b>
How to set data validation rules in the model .....	72
The model class with data validation .....	74
How to check the data validation .....	76
How to display validation error messages.....	76
The Future Value app after validating data.....	76
<b>Perspective .....</b>	<b>78</b>

## How to create a Core MVC web app

---

This chapter starts by showing how to create a new ASP.NET Core MVC web app. This includes all the skills you need to set up the folders and files for a simple MVC app that uses a controller to pass some data to a view that displays the data on a web page.

### How to start a new web app

---

To create a web app, you start by creating a new project as shown in figure 2-1. When creating a new project, Visual Studio provides several *templates* that you can use. These templates are displayed by the dialog shown in this figure.

In general, we recommend using the ASP.NET Core Web App (Model-View-Controller) template because it makes it easy to start an MVC web app. However, if you want to manually build your web app from scratch, you can use the ASP.NET Core Empty template. Either template should work fine for the Future Value app presented in this chapter.

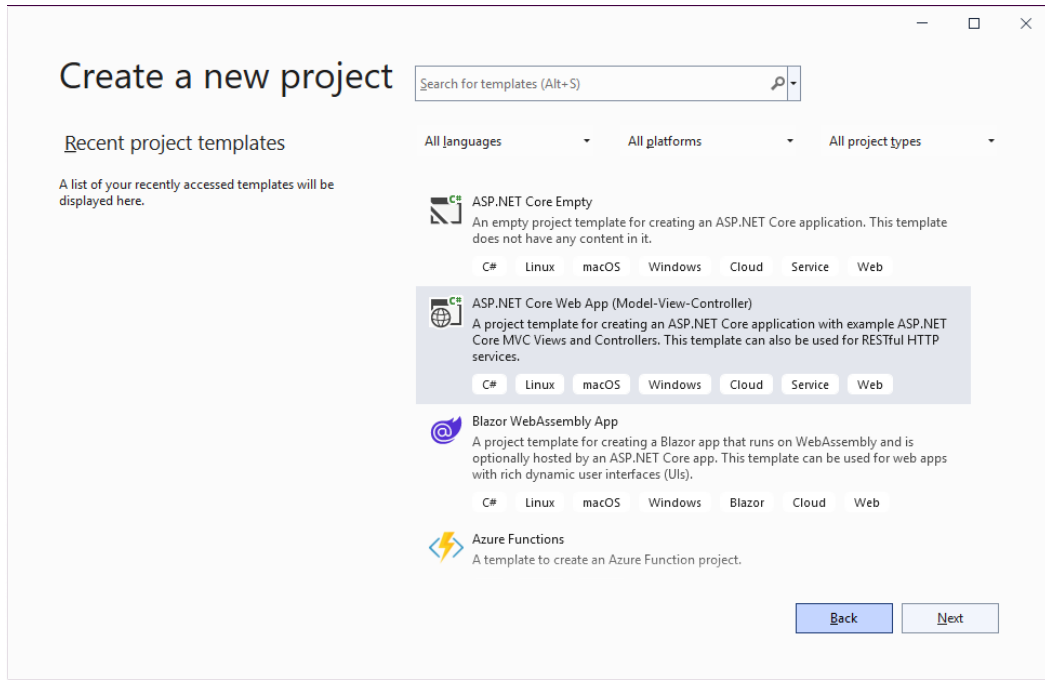
The template you choose determines the folders and files that Visual Studio adds to the project when it creates your web app. In this book, you'll learn how to use the two templates summarized in this figure.

The Web App MVC template sets up the starting folders and files for an ASP.NET Core MVC web app, including a configuration file that configures the default routing for the app. When you use this template, you typically start by deleting files and code that you don't need. Then, as you develop the app, you add the files and code you do need.

The Empty template provides two starting files for an ASP.NET Core app. When you use this template, you must manually add the folders and files for an MVC web app and configure the middleware for the request pipeline. Although we recommend using the MVC template for this chapter, you can also use the Empty template if you prefer to add the folders and files you need instead of deleting the ones that you don't need.

Although you won't learn how to use any of the other templates in this book, you might want to experiment with them. For example, the ASP.NET Core API template sets up the starting folders and files for a RESTful web service. Also, the ASP.NET Core Web App template sets up an ASP.NET Core app that uses Razor Pages.

## The dialog that displays the project templates



## The templates presented in this book

Template	Contains...
Web App MVC	Starting folders and files for an ASP.NET Core MVC web app.
Empty	Two starting files for an ASP.NET Core app.

## How to create a new project

- Select File→New→Project from the menu system, select the template you want to use, and then click the Next button.

## Description

- When creating an ASP.NET Core web app, Visual Studio provides several *templates* that you can use.
- For this chapter, we recommend using the ASP.NET Core Web App (Model-View-Controller) template, also known as the MVC template, because it makes it easy to start an ASP.NET Core MVC web app.
- If you want to manually build your web app from scratch, you can use the ASP.NET Core Empty template.

Figure 2-1 How to start a new web app

## How to configure a web app

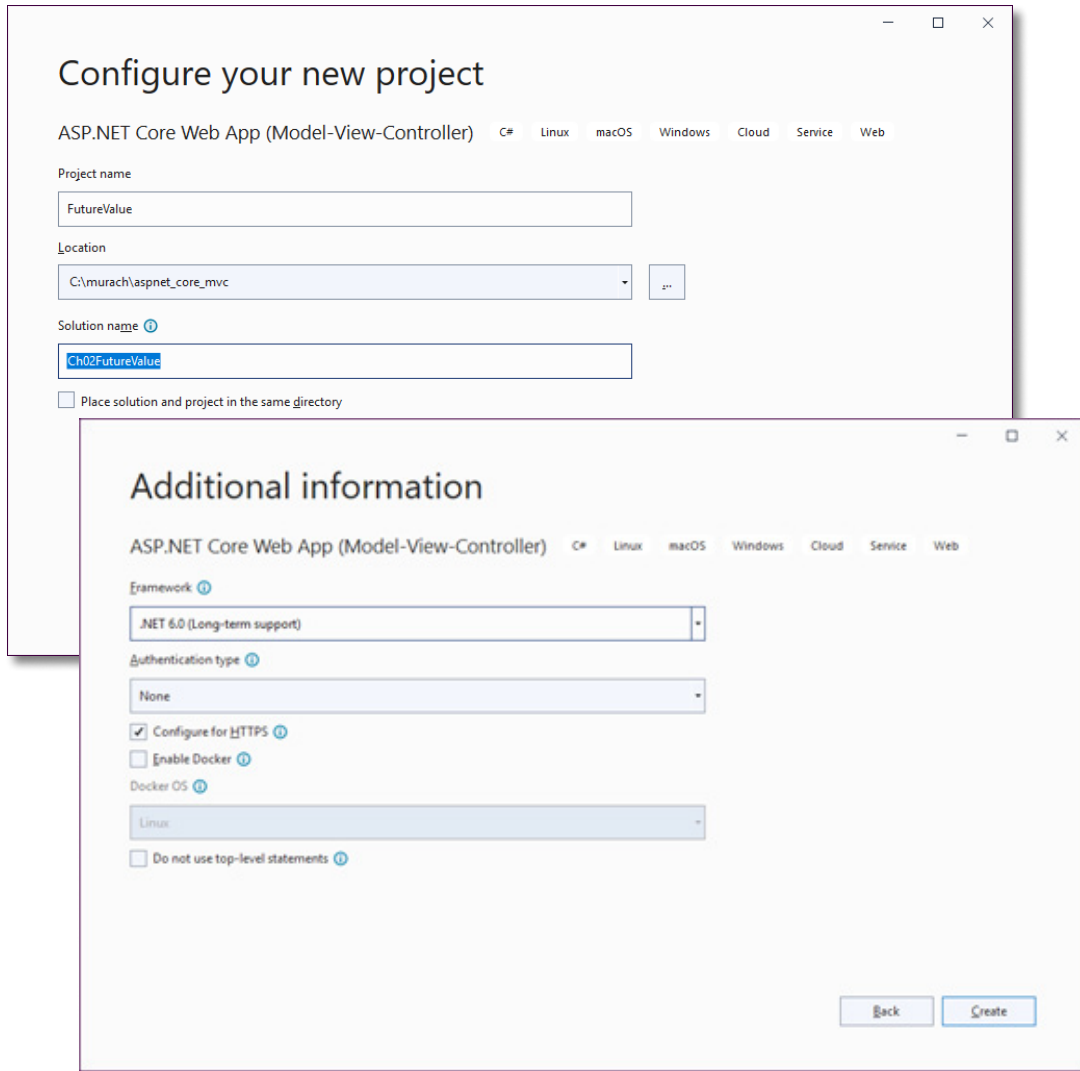
---

When you click the Next button from the dialog in figure 2-1, the first dialog in figure 2-2 is displayed. Then, you can use the dialog to specify the name of the project and the location of the project. In this figure, the name of the project is FutureValue. To specify the location of the project, you typically click the Browse button to select a different folder or use the Location drop-down list to select a location you've used recently. In this figure, the folder is C:\murach\aspnet\_core\_mvc.

If the “Place solution and project in the same directory” box is unchecked, Visual Studio creates a folder for the solution and a subfolder for the project. In this figure, this check box is unchecked and a name of Ch02FutureValue is specified for the solution. As a result, Visual Studio creates a folder for the solution named Ch02FutureValue and a subfolder for the project named FutureValue.

When you click the Next button from this dialog, the second dialog in this figure is displayed. This dialog lets you provide some additional configuration information. In most cases, you'll leave the options in this dialog at their defaults. However, you should notice that the first drop-down list lets you choose the target framework for the app. In most cases, you'll use the most recent framework, which in this case is .NET 6.0. If you need to target an earlier framework, you select it from the drop-down list. You may want to do that, for example, if you add a project to a solution that uses an earlier framework.

## The dialog for configuring a new web app



## How to configure a new ASP.NET Core MVC web app

1. Enter a project name.
2. Specify the location (folder). To do that, you can click the Browse [...] button.
3. Edit the solution name if necessary, and then click the Next button.
4. Use the resulting dialog to provide any additional information for the web app and click Create.

## Description

- If the “Place solution and project in the same directory” box is unchecked, Visual Studio creates a folder for the solution and a subfolder for the project. Otherwise, these files are stored in the same folder.

Figure 2-2 How to configure a new web app

## How to set up the MVC folders

---

The procedures in figure 2-3 show how to set up the folders for an MVC web app. The procedure you use depends on whether you started the web app from the Web App MVC template or the Empty template.

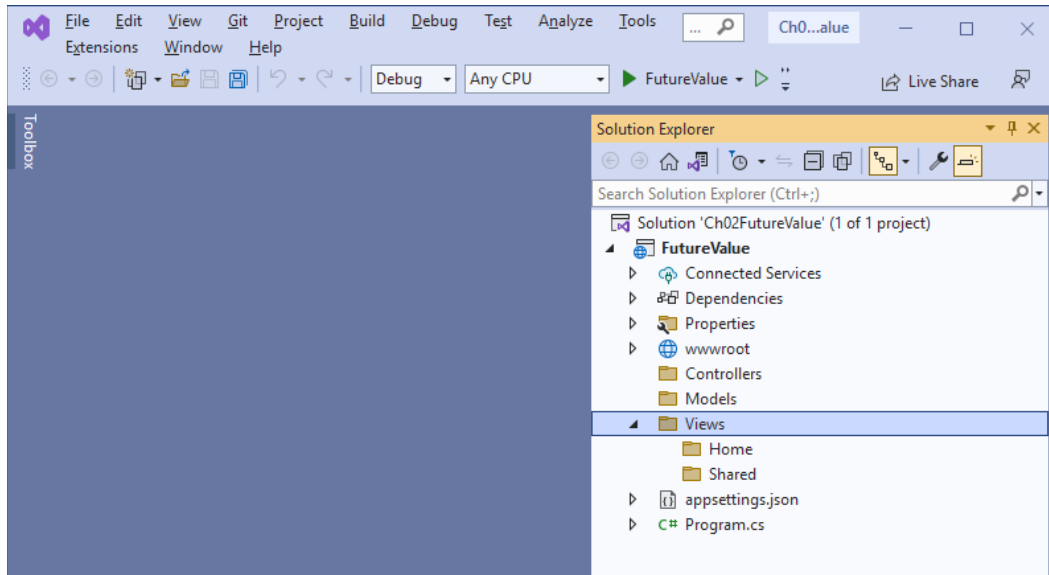
If you started from the Web App MVC template, you can delete all files from the folders named Models, Views, and Controllers. In addition, you can delete all files from the Home and Shared subfolders of the Views folder. This is an excellent approach when you're getting started.

Alternately, you can leave these files and edit them as described later in this chapter. However, this approach leaves extra files and code that can lead to errors, and it doesn't give you practice adding the files described in this chapter. As a result, it's better to use this approach later, after you've learned more about developing MVC web apps.

If you started from the Empty template, you need to add the folders named Models, Views, and Controllers. Then, you need to add the Home and Shared subfolders of the Views folder.



## Visual Studio after the folders have been set up for an MVC web app



### How to delete unnecessary files from the MVC template

1. Expand the Controllers folder and delete all files in that folder.
2. Expand the Models folder and delete all files in that folder.
3. Expand the Views folder and its subfolders and delete all files in those folders, but don't delete the folders.

### How to add folders to the Empty template

1. Add the Controllers, Models, and Views folders.
2. Within the Views folder, add the Home and Shared folders.

### Description

- To add a folder, you can right-click a node and select Add→New Folder.
- To delete a folder or file, you can right-click the folder or file and select Delete.

Figure 2-3 How to set up the MVC folders

## How to add a controller

---

The procedure in figure 2-4 shows how to add a controller file to a web app. In addition, it shows the code for the controller after it has been edited so it's a good starting point for the Future Value app presented in this chapter.

A *controller* is a C# class that inherits from the `Controller` class that's available from the `Microsoft.AspNetCore.Mvc` namespace. When you develop an MVC app, it's common to place controller classes in a namespace that consists of the project name, a dot, and the name of the folder that stores the controllers. In this figure, for example, the `HomeController` class is stored in the `FutureValue.Controllers` namespace. If you follow the procedure in this figure, this is where the `HomeController` class is placed by default.

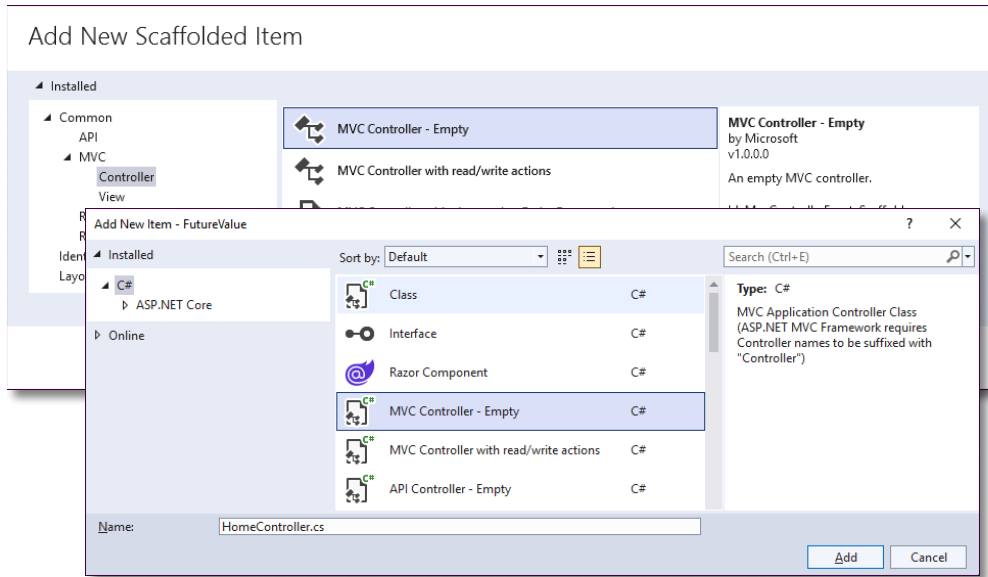
If a method of a controller runs in response to HTTP action verbs such as GET or POST, the method is known as an *action method*, or an *action*. In this figure, for example, the `Index()` method is an action because it runs in response to an HTTP GET or POST request. You'll learn more about how this works later.

In this figure, the `Index()` action begins by setting two properties of the `ViewBag` property that's automatically available to controllers and views. To do that, the first statement sets the `Name` property of the `ViewBag` to a string value of "Mary". Then, the second statement sets the `FV` property to a decimal value of 99999.99. This works because the `ViewBag` property uses dynamic properties to get and set values. As a result, you can dynamically create a property by specifying any property name that you want.

After the `Index()` action has stored some data in the `ViewBag`, it uses the `View()` method to return a `ViewResult` object for the view associated with the action method. For the `Index()` action of the `Home` controller, this returns a `ViewResult` object for the view in the `Views/Home/Index.cshtml` file like the one shown in the next figure. This works because a `ViewResult` object is a type of `ActionResult` object. As a result, the `Index()` method can return a `ViewResult` object.

Because specifying the `ActionResult` interface as the return type for an action method allows you to return any type of action result, it provides a flexible way to code an action method. Then, if you later decide to return a different type of action result, you can do that. However, if you know that you are definitely going to return a `ViewResult` object, you can change the return type of the method to `ViewResult`. Some programmers think this makes your code easier to read.

## The dialogs for adding a controller



### How to add a file for a controller

1. Right-click the Controllers folder and select Add→Controller.
2. In the Add New Scaffolded Item dialog, select “MVC Controller – Empty” and click Add.
3. In the Add New Item dialog, name the controller and click Add.

### The HomeController.cs file

```
using Microsoft.AspNetCore.Mvc;

namespace FutureValue.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            ViewBag.Name = "Mary";
            ViewBag.FV = 99999.99;
            return View();
        }
    }
}
```

### Description

- A method of a *controller* that runs in response to HTTP action verbs such as GET or POST is known as an *action method*, or an *action*.
- The ViewBag property is automatically available to controllers and views. It uses dynamic properties to get and set values.
- The View() method returns a ViewResult object for the view associated with an action method.

Figure 2-4 How to add a controller

## How to add a Razor view

---

The procedure in figure 2-5 shows how to add a Razor view to a web app. In addition, it shows the code for the view after it has been edited so it's a good starting point for the Future Value app presented in this chapter.

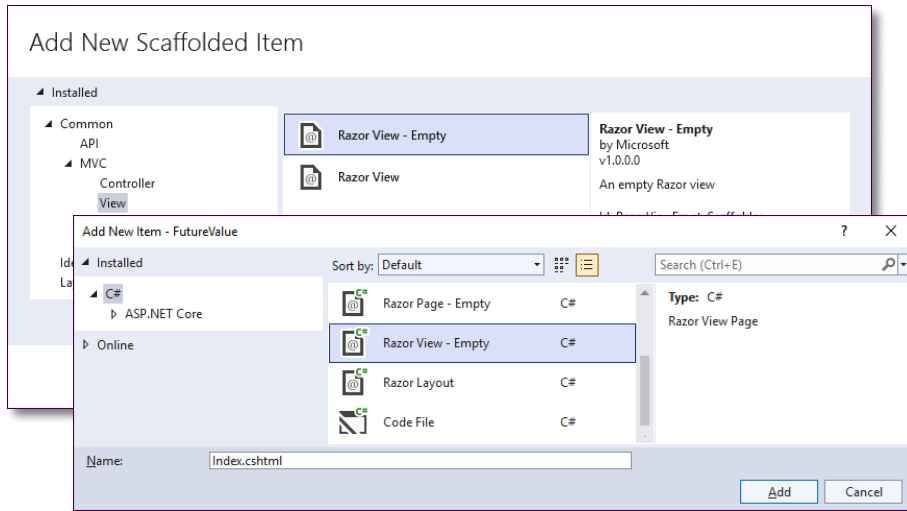
A *Razor view* contains both C# and HTML code. That's why its file extension is .cshtml. In ASP.NET Core MVC, the *Razor view engine* uses server-side code to embed C# code within the HTML. Razor code is preceded by the @ sign.

To execute one or more C# statements, you can declare a *Razor code block* by coding the @ sign followed by a pair of braces ( { } ). In this figure, for example, the Index.cshtml file begins with a code block that contains a single C# statement. This statement sets the Layout property that's available to all views to null. This indicates that the view doesn't have a Razor layout. You'll learn more about Razor layouts later in this chapter.

To evaluate a C# expression and display its result, you can code a *Razor expression* by coding the @ sign followed by the expression. In this figure, for example, the view uses Razor expressions to access the ViewBag property that's available to all views and display the two properties that were set by the controller in the previous figure. Here, the first expression just displays the Name property. However, the second expression gets the FV property and calls the ToString() method to convert the decimal value to a string that uses the currency format with 2 decimal places. To do that, this code supplies a format specifier of "C2".

Besides the Razor code, the rest of the code for this view consists of simple HTML elements. As a result, if you have some experience with HTML, you shouldn't have any trouble understanding this page. If you don't understand the HTML on this page, you need to learn basic HTML skills like the ones presented in the first eight chapters of *Murach's HTML and CSS*.

## The dialogs for adding a Razor view



### How to add a view to the Views/Home folder

1. In the Solution Explorer, right-click the Views/Home folder and select Add→View.
2. In the Add New Scaffolded Item dialog, select Razor View – Empty and click Add.
3. In Add New Item dialog, name the view Index and click Add.

### The Home/Index.cshtml view

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Home Page</title>
</head>
<body>
    <h1>Future Value Calculator</h1>
    <p>Customer Name: @ViewBag.Name</p>
    <p>Future Value: @ViewBag.FV.ToString("C2")</p>
</body>
</html>
```

### Description

- A *Razor view* contains both C# code and HTML. That's why its file extension is .cshtml.
- In ASP.NET Core MVC, the *Razor view engine* uses server-side code to embed C# code within HTML elements.
- To execute one or more C# statements, you can declare a *Razor code block* by coding the @ sign followed by a pair of braces ( { } ).
- To evaluate a C# expression and display its result, you can code a *Razor expression* by coding the @ sign followed by the expression.

Figure 2-5 How to add a Razor view

## How to configure the middleware for an HTTP request pipeline

---

Figure 2-6 shows how to configure a simple MVC web app like the Future Value app presented in this chapter. To do that, you can edit the `Program.cs` file so it configures the middleware for the HTTP request pipeline correctly. If you're starting from the MVC template, you can usually just accept the default configuration that it generates, which is shown in this figure. Or, if you're starting from the Empty template, you can add the statements shown in this figure.

The code for the `Program.cs` file that's generated by the MVC template starts by creating a `WebApplicationBuilder` object named `builder`. Then, it calls the `AddControllersWithViews()` method of the builder object's `Services` property. This enables the services required by the controllers and Razor views of an MVC app. As you move through this book, you'll use the `Services` property of the builder object to add other services as well.

Once the services that the application needs are added, the code calls the `Build()` method of the builder object and assigns the `WebApplication` object it returns to a variable named `app`. Then, it uses this object to configure the pipeline.

To start, the code checks whether the web hosting environment is a development environment. If it isn't, the middleware handles exceptions by displaying a page that the developer customizes for end users. That's typically what you want for a production environment. In addition, this code calls the `UseHsts()` method to configure the middleware to send HTTP Strict Transport Security (HSTS) headers to clients, which is a recommended practice for production apps.

Otherwise, the middleware handles exceptions by displaying a web page that's designed for developers, not end users. That's typically what you want when you're in a development environment as described throughout this book.

Next are several statements that configure the middleware components that are common to development and production environments. Of these statements, it's important to note that the `MapControllerRoute()` method sets the default controller for the app to the `Home` controller, and it sets the default action to the `Index()` action. As a result, when the app starts, it calls the `Index()` action method of the `Home` controller. This displays the `Index` view, which is usually what you want.

## The Program.cs file that's generated by the MVC template

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

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

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this
    // for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

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

app.Run();
```

## Description

- The Program.cs file contains code that configures the middleware for the HTTP request pipeline and then starts the program.
- The code generated by default by the MVC template begins by checking whether the web hosting environment is a development environment. If it isn't, it configures the middleware for a production environment.
- The MapControllerRoute() method in this figure sets the default controller for the app to the Home controller, and it sets the default action to the Index() action. As a result, when the app starts, it calls the Index() action method of the Home controller.

---

Figure 2-6 How to configure the middleware for an HTTP request pipeline

## How to run a web app and fix errors

---

After you write the C# and HTML code shown in the previous figures, you need to test it to be sure it works properly. Then, if you encounter any errors, you need to fix them and test the app again. For now, you can do that with the basic skills presented in the next two figures. Later, in chapter 5, you'll learn more skills for testing and debugging.

### How to run a web app

---

To run a web app, you can use one of the techniques presented in figure 2-7. For example, you can press Ctrl+F5 to run the web app without the debugger. Then, you can stop the app by clicking the close button in the browser's upper right corner.

However, you can also run the web app with the debugger by pressing F5. When you do that, you can use Visual Studio's debugger as described in chapter 5. Then, you can stop the app by clicking the Stop Debugging button in the Debug toolbar.

When you run an app, you need to decide whether to run it on the older Windows-only IIS Express server or the newer cross-platform Kestrel server. Since the Kestrel server runs faster than the IIS Express server, it's excellent for getting started with ASP.NET Core development. As a result, we recommend that you use it with this book.

If you view the drop-down list to the right of the Start button, you'll see that the project name is selected by default. This indicates that the project will use the Kestrel server. If you want to use IIS Express instead, you'll need to select the IIS Express item.

If you run the app with Kestrel, Visual Studio starts the server and uses a console window to display information about the status of each HTTP request. To stop the server, you can close this window.

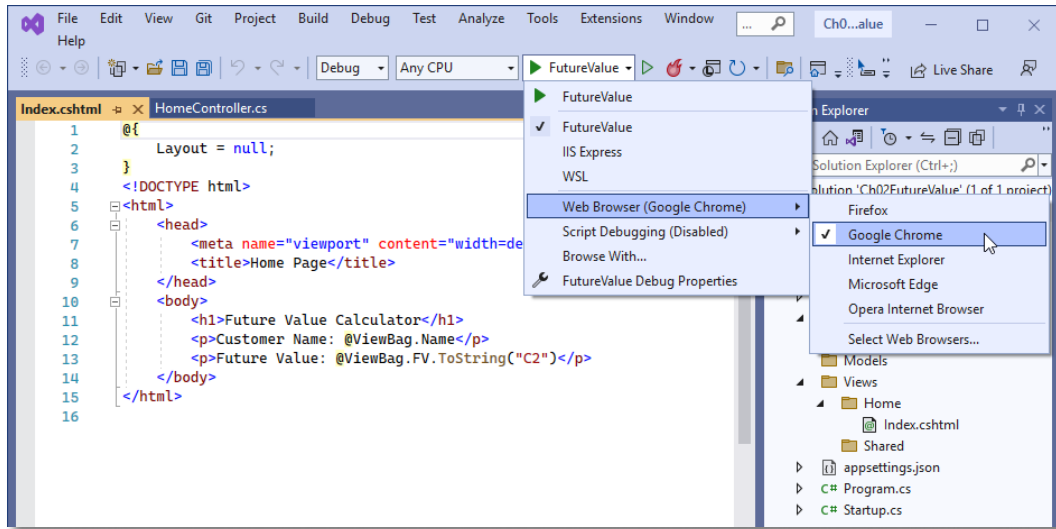
In addition to testing whether a web app runs correctly, you should also check whether it displays correctly in different browsers. Visual Studio makes it easy to change the default browser for this purpose by providing a drop-down browser list. After you use that list to change the default browser, you can run the web app again to test it in that browser.

Before Visual Studio runs an app, it builds the project by compiling the necessary code. Then, if the code compiles without errors, Visual Studio runs the app and displays the starting page in your default browser. At that point, you can test the app to make sure that it works correctly. For now, the app is working correctly if it displays a web page that looks like the one shown in this figure that displays a customer name of "Mary" and a future value of \$99,999.99.

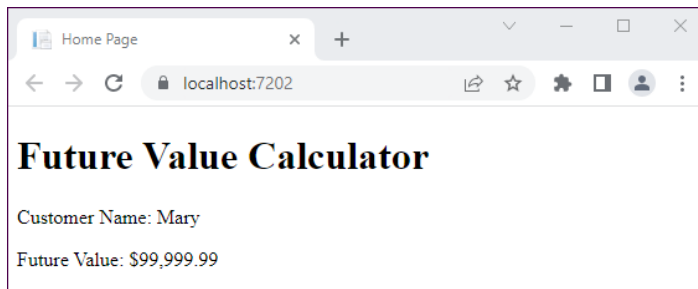
The first time you run a web app, you may get a series of dialogs that display security warnings that indicate that you are about to install two SSL certificates. The IIS Express certificate is used by Visual Studio's Hot Reload feature to create a secure connection to the browser. As you'll learn in chapter 5, this is a



## The Start button drop-down list in Visual Studio



## The Future Value app in the Chrome browser



## Description

- To run an app in the default browser, press Ctrl+F5. This starts the app without debugging.
- To stop an app, click the close button in the browser's upper right corner.
- To change the default browser for the app, display the drop-down list for the Start button, select the Web Browser item, and select the default web browser from the list.
- By default, Visual Studio uses the Kestrel web server, which is usually what you want. To change the web server to the IIS Express server, display the drop-down list for the Start button and select IIS Express.
- When Visual Studio runs the app on the Kestrel server, it uses a console window to display information about the server. To stop the server, you can close the command line window.
- If you press F5 or click the Start button in the toolbar, Visual Studio starts the app with debugging. This is another way to run an app that's especially useful if you need to debug an app as described in chapter 5. Then, to stop the app, you can click the Stop Debugging button in the Debug toolbar.

Figure 2-7 How to run a web app

time-saving feature that you'll want to use while testing your projects. Because of that, you'll want to be sure to install this certificate.

ASP.NET Core needs the second certificate to configure a development environment, so you can install this certificate as well. Then, if a web page is displayed indicating that it may not be safe to proceed, you can click the link or button to proceed.

## How to find and fix errors

---

If any errors are detected as part of the compilation, Visual Studio opens the Error List window and displays the errors as shown in figure 2-8. These errors can consist of *syntax errors* that have to be corrected before the app can be compiled, as well as warning messages. In this figure, just one error message and no warning messages are displayed.

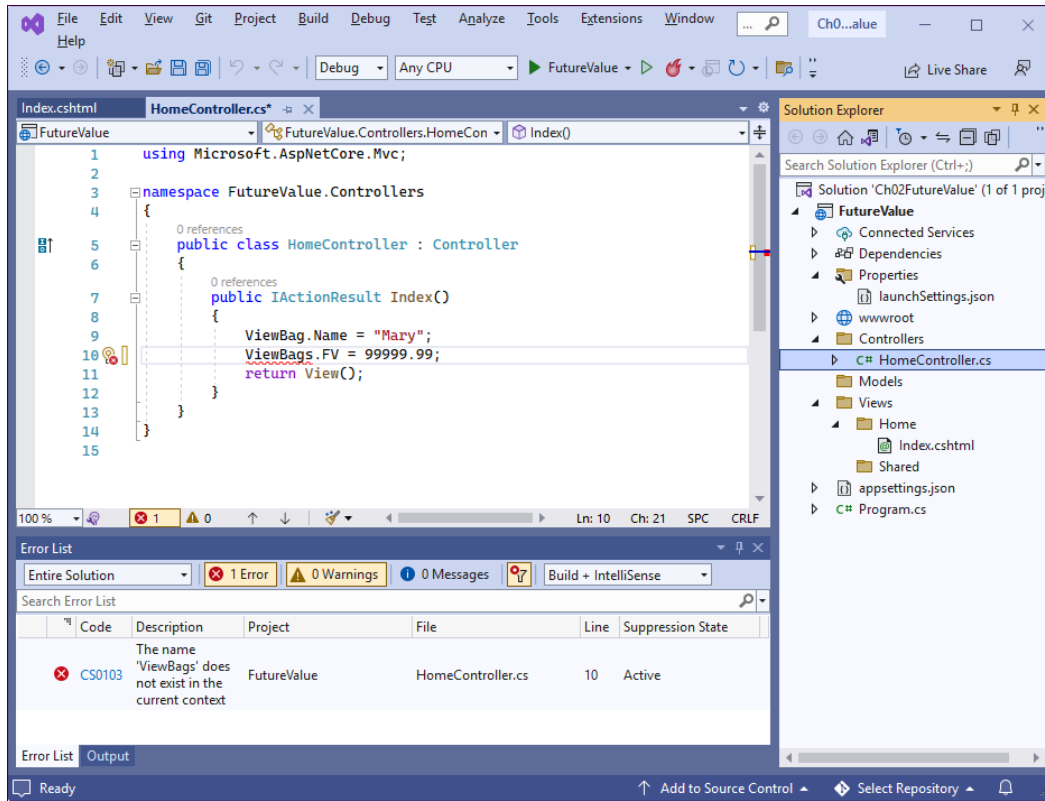
To fix an error, you can double-click it in the Error List window. This moves the cursor into the code editor and to the line of code that caused the error. By moving from the Error List window to the code editor for all of the messages, you should be able to find the coding problems and fix them. In this figure, the error message accurately indicates that the name `ViewBags` doesn't exist. That's because the name of the property that's available to controllers and views is `ViewBag`, not `ViewBags`.

Keep in mind, though, that the error message might not be accurate, and its link might not jump to the line of code that's causing the problem. For example, it's common to need to fix a related statement such as a statement that declares a variable. Still, the error message and the line of code that it links to should help you find the statement that's causing the problem.

After you fix all of the syntax errors and run the app, you may still encounter an *exception*. That happens when ASP.NET Core can't execute one of the compiled C# statements correctly at runtime. Then, if you're running the app without debugging, ASP.NET Core MVC displays a description of the exception in the web browser. At that point, you can stop the app. Then, you can fix the problem and test again.

Alternately, if you're running the app with debugging, ASP.NET Core MVC switches to the code editor and highlights the statement that caused the exception. At that point, you can stop the app by clicking on the Stop Debugging button in the Debug toolbar. Then, you can fix the problem and test again.

## Visual Studio with the Error List window displayed



### Description

- If a *syntax error* is detected as you enter code or when you attempt to build and run an app, a dialog asks whether you want to continue by running the last successful build. If you click No, the app isn't run and an Error List window is displayed.
- The Error List window provides information about the errors in your app.
- To go to the statement that caused a syntax error, double-click the error in the Error List window. This should help you find the cause of the error.
- If a compiled statement can't be executed when you run a web app, an *exception* occurs. Then, you can use the information that's displayed in the browser to attempt to fix this exception, or you can debug the exception as described in chapter 5.

Figure 2-8 How to find and fix errors

## How to work with a model

---

Once you're sure that the controller and view are working correctly, you're ready to add a model to your app. Then, you can modify the controller and view to work with this model. When you're done, the app should get data from the user, pass that data to the model, use the model to perform a calculation, and display the result of the calculation. Along the way, you'll learn a lot about how an ASP.NET Core MVC app works.

## How to add a model

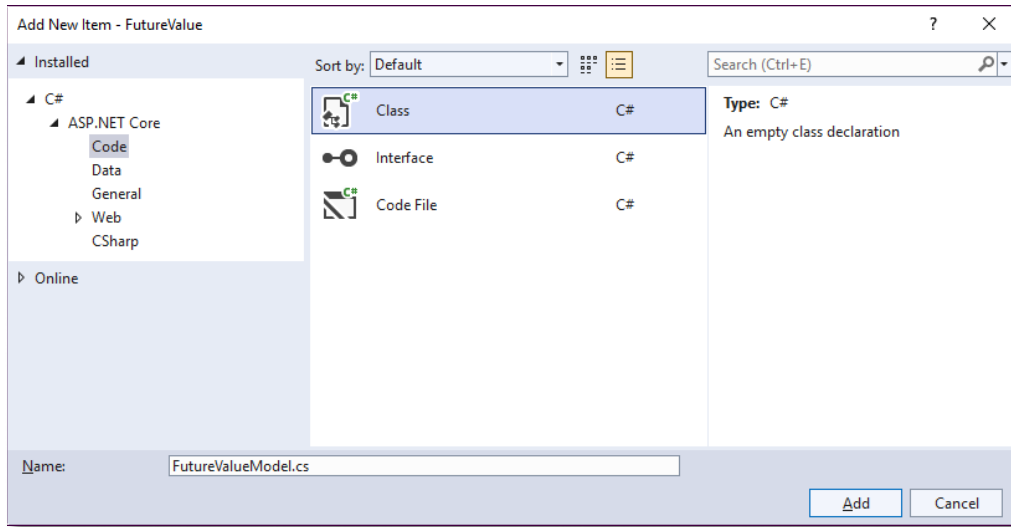
---

A *model* is a regular C# class that defines the data objects for a page and is typically stored in the Models folder. As a result, to add a model to your app, you just need to add a C# class to the Models folder as shown in figure 2-9. In this figure, the model class has a name of FutureValueModel. Here, the “Model” suffix is optional.

To keep the name of the model short, some programmers would prefer to drop the “Model” suffix and give the model a name of FutureValue. However, a model can't have the same name as a namespace, and this particular model is stored in the FutureValue namespace. As a result, this class uses the “Model” suffix to create a name for the model that doesn't conflict with the name of the namespace.

The model shown in this figure is a standard C# class. It provides three properties that can be used to get and set the monthly investment, yearly interest rate, and number of years for a future value. In addition, it provides a method named CalculateFutureValue() that calculates and returns the future value for the specified properties. To do that, this method converts the yearly values to monthly values and uses a loop to calculate the future value.

## The dialog for adding a class



## How to add a file for a model class

1. In the Solution Explorer, right-click the Models folder and select Add→Class.
2. In the resulting dialog, enter the name of the class, and click the Add button.

## The FutureValueModel class with three properties and a method

```
namespace FutureValue.Models
{
    public class FutureValueModel
    {
        public decimal MonthlyInvestment { get; set; }
        public decimal YearlyInterestRate { get; set; }
        public int Years { get; set; }

        public decimal CalculateFutureValue()
        {
            int months = Years * 12;
            decimal monthlyInterestRate = YearlyInterestRate / 12 / 100;
            decimal futureValue = 0;
            for (int i = 0; i < months; i++)
            {
                futureValue = (futureValue + MonthlyInvestment) *
                    (1 + monthlyInterestRate);
            }
            return futureValue;
        }
    }
}
```

## Description

- A *model* is a regular C# class that models the data for the app. The class for a model is typically stored in the Models folder.
- A model can't have the same name as the namespace.

Figure 2-9 How to add a model

## How to add a Razor view imports page

---

A *Razor view imports page* makes it easier to work with models and tag helpers. As a result, most web apps include this page.

The procedure in figure 2-10 shows how to add a Razor view imports page to your web app. This adds a file named `_ViewImports.cshtml` to your app that contains Razor directives that are applied to all views in your app.

To give you an idea of how a Razor view imports page works, this figure shows the code for the Razor view imports page of the Future Value app. Here, the first line imports the namespace for your model classes. That way, you can use classes from that namespace in your views using code like this:

```
@model FutureValueModel
```

The next figure shows how that works.

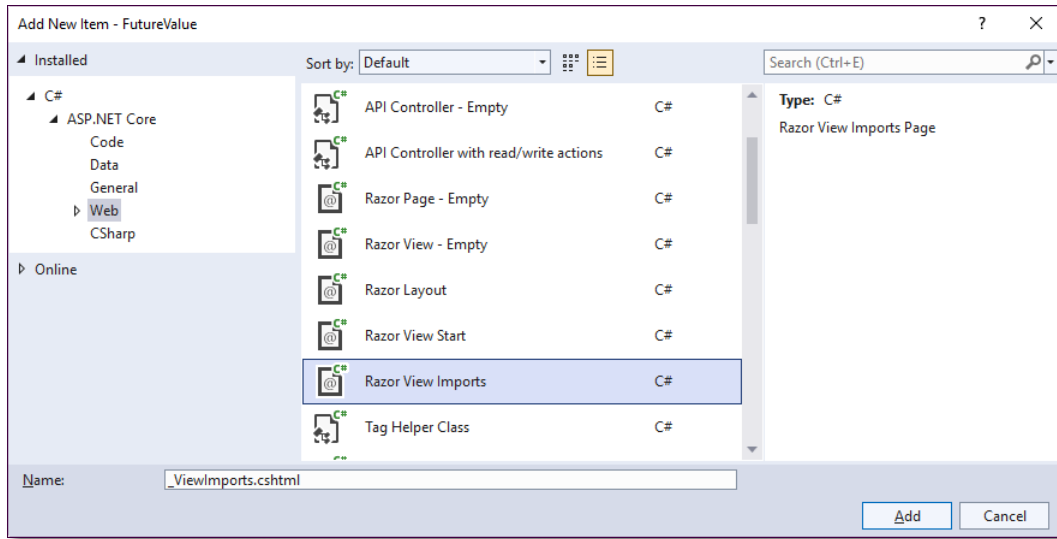
If you don't import the namespace for a model class, you can still use the model in your views. However, you'll need to fully qualify its name like this:

```
@model FutureValue.Models.FutureValueModel
```

As a result, you typically want to include a Razor view imports page that imports the model.

The second line of the Razor view imports page enables all tag helpers that are available from the ASP.NET Core framework. That way, you can use these tag helpers in your views. If you don't enable the tag helpers, you can still use them in your views. However, you need to add a `@addTagHelper` directive to the top of each view that uses tag helpers. As a result, it typically makes sense to include this directive in a Razor view imports page. That way, you don't have to specify this directive for each view.

## The dialog for adding a Razor view imports page



## How to add a Razor view imports page

1. In the Solution Explorer, right-click the Views folder and select Add→New Item.
2. In the resulting dialog, select the Installed→ASP.NET Core→Web category, select the Razor View Imports item, and click the Add button.

## The Views/\_ViewImports.cshtml file for the Future Value app

```
@using FutureValue.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

## A Razor view imports page makes it easier to work with...

- Model classes.
- Tag helpers.

## Description

- Most apps include a *Razor view imports page* that makes it easier to work with your model classes and the tag helpers that are available from ASP.NET Core.

Figure 2-10 How to add a Razor view imports page

## How to code a strongly-typed view

---

You use the `@model` directive to *bind* the model to the view. This kind of view is called a *strongly-typed view*. In figure 2-11, for example, the `@model` directive at the top of the view binds the view to the model class named `FutureValueModel`.

After binding the model to the view, this view uses the `asp-for` tag helper to *bind* HTML elements to the corresponding properties of the object. In particular, this tag helper binds the `MonthlyInvestment`, `YearlyInterestRate`, and `Years` properties to corresponding `<label>` and `<input>` elements in the view. As a result, when the user enters values into the `<input>` elements and clicks the Calculate button, ASP.NET Core MVC automatically updates the model with the values entered by the user. Then, the controller can access those values as shown in the next figure.

The `asp-for` tag helper automatically generates attributes for these HTML elements. For example, it generates the `name` and `id` attributes that MVC needs to be able to access these HTML elements. It also generates a `type` attribute that indicates the type of field to display.

The `asp-action` tag helper generates an `action` attribute for the `<form>` element. Instead of using this tag helper, you could just specify an `action` attribute that refers to the root directory like this:

```
<form action="/" method="post">
```

This maps to the `Index` action of the `Home` controller. However, using the `asp-action` tag helper makes your code more flexible and easier to maintain.

In this figure, the form only uses the `asp-action` tag helper to specify the action. As a result, MVC uses the `Index()` action method of the current controller, which is the `Home` controller. However, if you want to call an action from another controller, you can use the `asp-controller` tag helper to specify the name of that controller. As you progress through this book, you'll see plenty of examples of that.

The code in this figure uses the `asp-for` tag helper to access the properties of the model object. Since this tag helper is designed to bind a model object to HTML elements, you can access the properties of the model object just by specifying their names.

However, if you want to access a property of the model object outside of an `asp-for` tag helper, you must start by coding the `@Model` property (not the `@model` directive) to access the model object. Then, you access any property or method from that object. For example, you can use the `@Model` property to access the `MonthlyInvestment` property of the `FutureValueModel` model object like this:

```
<div>@Model.MonthlyInvestment</div>
```

Before you go on to the next figure, note that this view includes a `<style>` element within its `<head>` element. To save space, this `<style>` element just contains a comment that indicates that it includes all of the same CSS styles shown in figure 2-14. These styles apply some basic formatting to the `<body>`, `<h1>`, `<label>`, and `<div>` elements so this page appears as shown later in this chapter.



## Common tag helpers for forms

Tag helper	HTML tags	Description
<b>asp-for</b>	<b>&lt;label&gt; &lt;input&gt;</b>	Binds the HTML element to the specified model property.
<b>asp-action</b>	<b>&lt;form&gt; &lt;a&gt;</b>	Specifies the action for the URL. If no controller is specified, MVC uses the current controller.
<b>asp-controller</b>	<b>&lt;form&gt; &lt;a&gt;</b>	Specifies the controller for the URL.

## A strongly-typed Index view with tag helpers

```
@model FutureValueModel
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Future Value Calculator</title>
    <style>
        /* all of the CSS styles from figure 2-14 go here */
    </style>
</head>
<body>
    <h1>Future Value Calculator</h1>
    <form asp-action="Index" method="post">
        <div>
            <label asp-for="MonthlyInvestment">Monthly Investment:</label>
            <input asp-for="MonthlyInvestment" />
        </div>
        <div>
            <label asp-for="YearlyInterestRate">Yearly Interest Rate:</label>
            <input asp-for="YearlyInterestRate" />
        </div>
        <div>
            <label asp-for="Years">Number of Years:</label>
            <input asp-for="Years" />
        </div>
        <div>
            <label>Future Value:</label>
            <input value="@ViewBag.FV.ToString("C2")" readonly>
        </div>
        <button type="submit">Calculate</button>
        <a asp-action="Index">Clear</a>
    </form>
</body>
</html>
```

## Description

- You use the `@model` directive to *bind* the model to the view. This kind of view is called a *strongly-typed* view.
- ASP.NET Core MVC *tag helpers* are used to automatically generate *attributes* for some HTML elements. They are also used to *bind* HTML elements to the properties of the object that's the *model* for the view.

Figure 2-11 How to code a strongly-typed view

## How to handle GET and POST requests

---

Figure 2-12 begins by showing how to use the `HttpGet` and `HttpPost` attributes to create one `Index()` method that handles an HTTP GET request and another `Index()` method that handles an HTTP POST request. This is a common pattern in web development.

For example, it's common for a GET request to display a blank input form to the user. That happens by default when an ASP.NET Core MVC app starts, and it happens when a link like the Clear link on the Future Value form is clicked. Then, when the user clicks the submit button, the app sends a POST request to the same URL to process the data entered by the user. If you look back at figure 2-11, you'll see that the method attribute of the `<form>` element determines the type of request that's sent. In this case, it's a POST request.

In MVC, you can use overloaded action methods to handle both GET and POST requests for a page. In this figure, for example, the first `Index()` method doesn't accept any arguments. However, the second `Index()` method accepts a `FutureValueModel` object as an argument. Since each `Index()` method has a unique signature, you can use HTTP attributes to specify the HTTP verb for each method. When you use an attribute like this to provide information about a method, it is often referred to as *decorating* the method.

If you don't provide a unique signature for each version of the action method, you'll get a compiler error. For example, what if both versions of the action method need to specify the model as a parameter? In that case, you can solve the issue by specifying a dummy parameter like this:

```
public IActionResult Index(FutureValueModel model,  
    string dummy)
```

Here, the second argument isn't used by the `Index()` method. However, it provides a unique signature for the method.

When an action method accepts a model object as an argument, MVC uses the data stored in the GET or POST request to set the properties of the model object. In this figure, for example, MVC automatically sets the properties of the model object that's passed to the POST version of the `Index()` method.

As a result, the action method can use the model object to work with the posted data. In this figure, the code just calls the `CalculateFutureValue()` method from the model to get the result of the future value calculation. However, this shows that the other three properties of the model were set automatically, which is what you want.

In addition, the POST version of the `Index()` method can use the `View()` method to pass the model on to the view. In this figure, that's what the second statement does. That way, the strongly-typed view in the previous figure can display the correct values for the properties of the model.

## Two attributes that indicate the HTTP verb an action method handles

Attribute	Description
<b>HttpGet</b>	Specifies that the action method handles a GET request.
<b>HttpPost</b>	Specifies that the action method handles a POST request.

## Two methods you can use to return a view from a controller

Method	Description
<b>View()</b>	Returns the view that corresponds to the current controller and action.
<b>View(model)</b>	Passes the specified model to the view that corresponds to the current controller and action so the view can bind to the model.

## An overloaded Index() action method that handles GET and POST requests

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

public class HomeController : Controller
{
    [HttpGet]
    public IActionResult Index()
    {
        ViewBag.FV = 0;
        return View();
    }

    [HttpPost]
    public IActionResult Index(FutureValueModel model)
    {
        ViewBag.FV = model.CalculateFutureValue();
        return View(model);
    }
}
```

## Description

- A common pattern in web development is for the same URL to handle HTTP GET and POST requests. In particular, it's common to use a GET request for a URL to display a blank input form to the user. Then, a POST request for the same URL can process the data that's submitted when the user fills out the form and submits it.
- In MVC, you can use overloaded action methods to handle both GET and POST requests for a page. When you do, you use HTTP attributes to indicate which action method handles which request.
- When an action method accepts a model object as an argument, MVC uses the data stored in the GET or POST request to set the properties of the model object. Then, the action method can use the model object to work with the posted data, and it can use the View() method to pass the model on to the view.

Figure 2-12 How to handle GET and POST requests

## **The Future Value app after handling GET and POST requests**

---

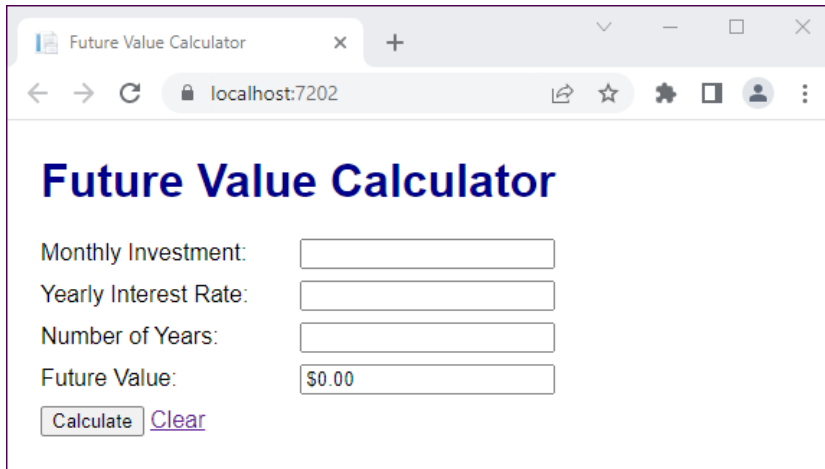
Figure 2-13 shows the Future Value app that has been presented so far in this chapter after it has handled GET and POST requests.

When this app starts, it sends a GET request to the `Index()` action of the Home controller. As a result, the app displays a screen like the first one shown in this figure. This page doesn't contain values for the first three fields, and it displays a value of \$0.00 for the fourth field, which is a read-only field.

When the user enters data in the form and clicks the Calculate button, the app sends a POST request to the `Index()` action of the Home controller. As a result, the app calculates the future value and displays it in the fourth text field as shown by the second screen in this figure.

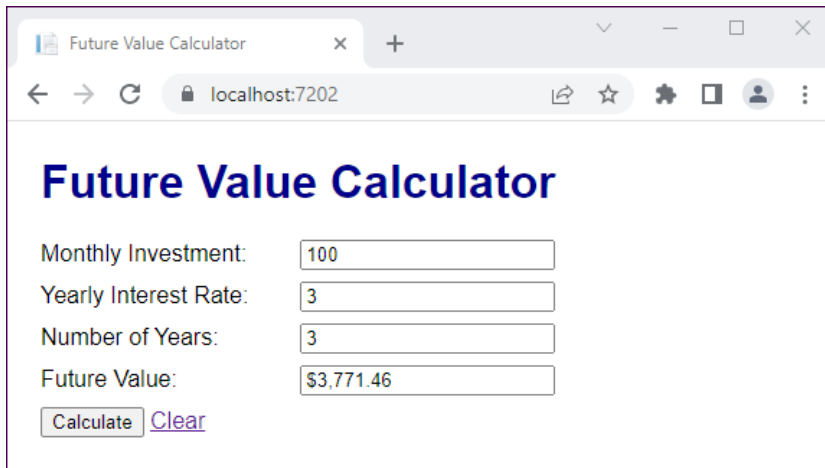
At this point, the user can edit the values and click the Calculate button again to calculate and display a different future value. Or, the user can click the Clear link. Then, the app sends a GET request to the `Index()` action of the Home controller. Since this GET request doesn't include a model, it clears the form as shown by the first screen.

### The Future Value app after a GET request



A screenshot of a web browser window titled "Future Value Calculator" at the URL "localhost:7202". The page displays the title "Future Value Calculator" in a large blue font. Below the title, there are four input fields: "Monthly Investment:", "Yearly Interest Rate:", "Number of Years:", and "Future Value:". The "Future Value:" field is pre-filled with "\$0.00". At the bottom left, there is a "Calculate" button and a "Clear" link.

### The Future Value app after a POST request



A screenshot of the same web browser window after a POST request. The input fields now contain the values: "Monthly Investment:" is 100, "Yearly Interest Rate:" is 3, "Number of Years:" is 3, and "Future Value:" is \$3,771.46. The "Calculate" button and "Clear" link remain at the bottom left.

### Description

- When the Future Value app starts, it sends a GET request to the Index() action of the Home controller.
- When the user clicks the Clear link, the app sends a GET request to the Index() action of the Home controller.
- When the user clicks the Calculate button, the app sends a POST request to the Index() action of the Home controller. If the user has filled out the form correctly, this automatically sets the three properties of the model object.

Figure 2-13 The Future Value app after handling GET and POST requests

## How to organize the files for a view

---

So far, the view for the Future Value app consists of a single view file. That's adequate for a web app that consists of a single page like the Future Value app presented in this chapter. However, most web apps consist of multiple pages. When that's the case, it makes sense to split the view for a web app into multiple files. That way, you can store HTML elements and CSS styles that are common to multiple pages in their own files. Then, you can use the common HTML elements and CSS styles in other pages as shown in the next three figures.

If it's adequate to store the view for a one-page app in a single file, why does this chapter show how to split the view for the Future Value app into multiple files? Well, in the real world, multi-page apps are more common than one-page apps. Even for a simple app like this Future Value app, you might want to add pages such as an About page or a Contact Us page. As a result, it often makes sense to set up your app to support multiple pages, even if it's currently a one-page app.

## How to add a CSS style sheet

---

Figure 2-14 shows how to add a file for a *CSS style sheet*. This provides a way to store the formatting for multiple web pages in a single external file.

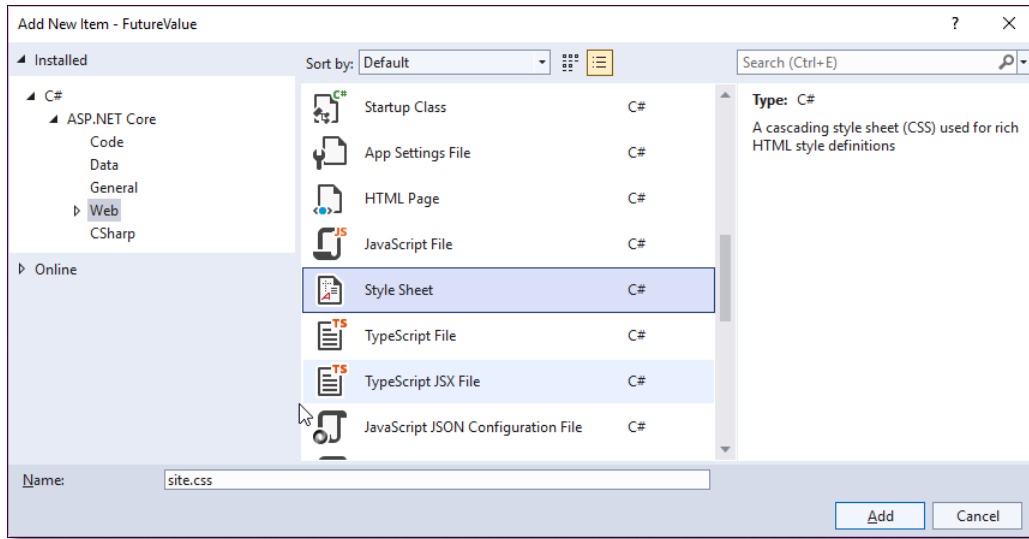
When you add a style sheet to a project, you typically add it to the `css` folder of the `wwwroot` folder. In this figure, for example, a style sheet named `site.css` is added to the `wwwroot/css` folder. If your project is based on the MVC template, this folder and file should already exist. However, if they don't exist, you need to create them.

This figure also shows the styles that are stored in the `site.css` file. These styles format the Future Value app so it looks the way it does in the previous figure. If you're familiar with CSS, you shouldn't have any trouble understanding this code.

The four style rules in this figure select elements by type. These are referred to as type selectors. To code a type selector, you just code the name of the element. As a result, the first style rule in this group selects the `<body>` element, the second selects all `<h1>` elements, the third selects all `<label>` elements, and the fourth selects all `<div>` elements.

Each style rule also includes one or more declarations enclosed in braces that specify the formatting for the selected element. In this figure, the declarations for the `<body>` element set the font family for all elements nested within that element and set the padding for that element. The declarations for the `<h1>` element set the top margin and color for all `<h1>` elements on the page. The declarations for the `<label>` element cause it to be displayed on the same line as the following element and set the width and padding for all `<label>` elements on the page. And the declaration for the `<div>` element sets the bottom margin for all `<div>` elements on the page.

## The dialog for adding a CSS style sheet



## How to add a CSS style sheet

1. If the `wwwroot/css` folder doesn't exist, create it.
2. Right-click the `wwwroot/css` folder and select `Add→New Item`.
3. Select the `ASP.NET Core→Web` category, select the `Style Sheet` item, enter a name for the CSS file, and click the `Add` button.

## The `site.css` file for the Future Value app

```
body {
    padding: 1em;
    font-family: Arial, Helvetica, sans-serif;
}
h1 {
    margin-top: 0;
    color: navy;
}
label {
    display: inline-block;
    width: 10em;
    padding-right: 1em;
}
div {
    margin-bottom: .5em;
}
```

## Description

- A *CSS style sheet* provides a way to store the formatting for multiple web pages in a single external file.
- By default, an ASP.NET Core web app includes a style sheet named `site.css` in the `wwwroot/css` folder. You can modify this style sheet or create one of your own.

Figure 2-14 How to add a CSS style sheet

## How to add a Razor layout and view start

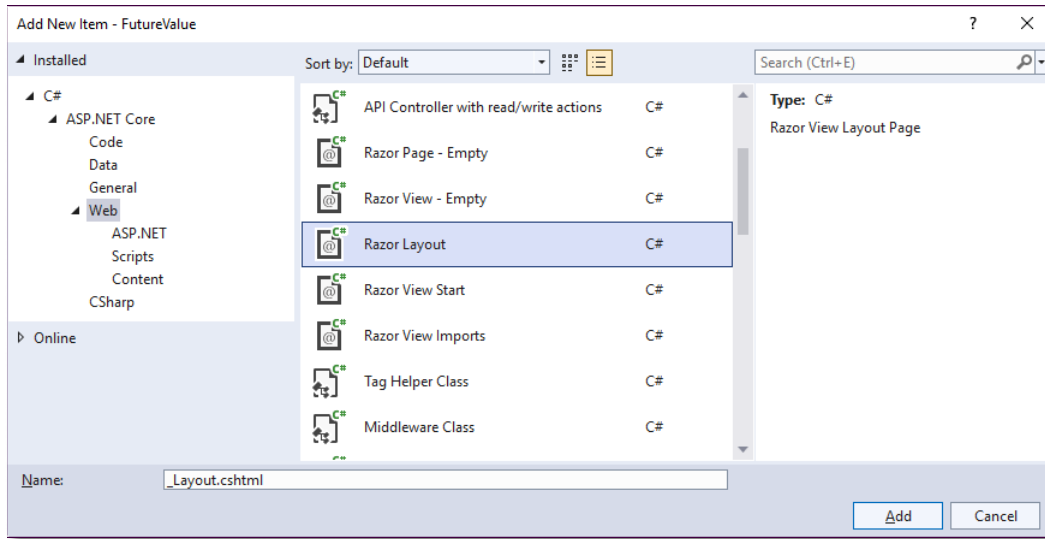
---

When you create a multi-page web app, it's common to have headers, footers, and navigation bars that are displayed on all or most pages of a web app. In other words, it's common to have HTML elements that are common to all pages. In that case, it's a good practice to store the elements that are common to multiple pages in separate files. This allows you to keep a consistent look across all pages, and it makes your app easier to maintain.

To store elements that are common to multiple pages in a separate file, you can add a Razor layout to your web app as described in figure 2-15. A *Razor layout* provides a way to store elements that are common to multiple web pages in a single file. Then, it usually makes sense to add a *Razor view start* to specify the default layout for the views of your web app. Finally, you can add a *Razor view* to provide a way to store elements that are unique to a web page. You'll see how to do that in just a minute.



## The dialog for adding a Razor layout or view start



### How to add a Razor layout

1. Right-click the Views/Shared folder and select Add→New Item.
2. Select the ASP.NET Core→Web category, select the Razor Layout item, and click the Add button.

### How to add a Razor view start

1. Right-click the Views folder (not the Views/Shared folder) and select Add→New Item.
2. Select the ASP.NET Core→Web category, select the Razor View Start item, and click the Add button.

### Description

- A *Razor layout* provides a way to store elements that are common to multiple web pages in a single file.
- A *Razor view start* lets you specify the default Razor layout for the Razor views of a web app.

Figure 2-15 How to add a Razor layout and view start

## The code for a Razor layout and view start

---

Figure 2-16 shows the code for a Razor layout and view start for the Future Value app. If you study this code, you should see how it provides code that can be used by any Razor view in the app. When combined with the view shown in the next figure, you'll see how this code all fits together to form the same strongly-typed view as the one presented earlier in this chapter. The only differences are that it has been split up into the three Razor files and it uses the external CSS file shown earlier in this chapter.

The code for the layout is stored in a Razor file named `_Layout`. This code stores the HTML elements that are common to all pages such as the `<html>`, `<head>`, and `<body>` elements. In addition, it uses Razor code to do some processing. For example, it uses Razor code to get the title for the page from the `Title` property of the `ViewBag` property that's automatically available to all layouts and views. In addition, it uses Razor code to call the `RenderBody()` method that's available to all layouts. This inserts the code from any view file that uses this layout.

The code for the layout also uses a `<link>` element to link to the CSS style sheet shown earlier in this chapter. To do that, it specifies a `rel` attribute of "stylesheet" and an `href` attribute of "`~/css/site.css`". As a result, all of the pages of the web app use the styles from that style sheet.

The code for the view start is stored in a Razor file named `_ViewStart`. This code defines a block of C# statements that are executed before the view is rendered. In this example, the block contains a single statement that sets the `Layout` property to "`_Layout`". In other words, it sets the default layout for all views in the app to the `_Layout` view shown in the first example.

## The Views/Shared/\_Layout.cshtml file

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewBag.Title</title>
  <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
  <div>
    @RenderBody()
  </div>
</body>
</html>
```

## The Views/\_ViewStart.cshtml file

```
@{
    Layout = "_Layout";
}
```

## Description

- You can use the Razor file named \_ViewStart to set the default layout for all the views in your app. However, if necessary, you can use the Layout property of a view to override the default layout.

## How to add a Razor view when using a layout with a view start

---

Figure 2-17 shows how to add a Razor view that works with the Razor layout and view start shown in the previous figure. To do that, you select the Razor View item from the Add New Scaffolded Item dialog instead of the Razor View - Empty item as shown earlier in this chapter. Then, the dialog in this figure is displayed, which lets you name the view and indicate that you want to use a layout page with it.

Here, the code for the view is stored in a Razor file named `Index`. When combined with the `_Layout.cshtml` and `_ViewStart.cshtml` files shown in the previous figure, this code works much like the strongly-typed view presented earlier in this chapter. The main difference is that it uses a Razor code block to set the title for the page. To do that, it sets the `Title` property of the `ViewData` object to “Future Value Calculator”. As you’ll learn in chapter 8, `ViewBag` uses `ViewData` under the hood to store its data. As a result, the layout for the page can use `ViewBag` to get the value of the `Title` property and display it as the title of the page.

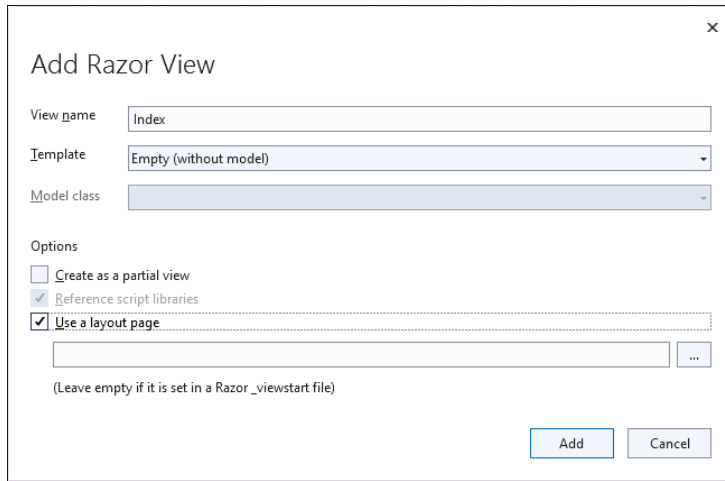
In general, it’s considered a good practice to use a view start to set the default layout for all the views in your app. However, if necessary, you can use the `Layout` property of a view to override the default layout. To do that, you can add a statement below the statement that sets the title for the page like this:

```
@{  
    ViewData["Title"] = "Future Value Calculator";  
    Layout = "_LayoutCalculator";  
}
```

Here, the page is using a hypothetical Razor layout named `_LayoutCalculator` that’s designed especially for all of the calculator pages of the web app.

For now, that’s all you need to know about Razor layouts, view starts, and views. In chapter 7, however, you’ll learn more about working with layouts and views.

## The dialog for adding a Razor view



## How to add a Razor view

1. Right-click the folder for the view (Views/Home, for example) and select Add→View.
2. Use the Add New Scaffolded Item dialog from figure 2-5 to select Razor View, and then click Add.
3. Use the Add Razor View dialog to enter a name for the view, make sure that the “Use a layout page” item is selected, but don’t specify a name for the layout page.

## The Views/Home/Index.cshtml file

```
@model FutureValueModel
@{
    ViewData["Title"] = "Future Value Calculator";
}
<h1>Future Value Calculator</h1>
<form asp-action="Index" method="post">
    <div>
        <label asp-for="MonthlyInvestment">Monthly Investment:</label>
        <input asp-for="MonthlyInvestment" /></div>
    <div>
        <label asp-for="YearlyInterestRate">Yearly Interest Rate:</label>
        <input asp-for="YearlyInterestRate" /></div>
    <div>
        <label asp-for="Years">Number of Years:</label>
        <input asp-for="Years" /></div>
    <div>
        <label>Future Value:</label>
        <label>@ViewBag.FV.ToString("c2")</label></div>
    <button type="submit">Calculate</button>
    <a asp-action="Index">Clear</a>
</form>
```

## Description

- A *Razor view* provides a way to store elements that are unique to a web page.

Figure 2-17 How to add a Razor view when using a layout with a view start

## How to validate user input

---

At this point, the Future Value app works correctly if the user enters valid data. However, if the user enters invalid data and clicks the Calculate button, the app just displays a future value of 0 without displaying any error messages to indicate that the user has entered invalid data. When you code a web app, you typically want to display error messages if the user enters invalid data. Fortunately, ASP.NET Core MVC makes it easy to validate data and display error messages as shown in the next three figures. This is known as *data validation*, and it's an important part of developing most apps.

## How to set data validation rules in the model

---

The first step in validating the data that a user enters is to set data *validation rules* in the model as described in figure 2-18. To start, you can import the `DataAnnotations` namespace. Then, you can use the *validation attributes* from that namespace to set the data validation rules.

The table in this figure describes two of the most common validation attributes. First, you can decorate a model property with the `Required` attribute to indicate that a value is required for that property. Second, you can decorate a property with the `Range` attribute to indicate that the value for that property must be within the specified range of values.

When you code the `Required` attribute, the data type for the property must be nullable for the validation to work properly. To make a non-nullable data type nullable, you can code a question mark (?) after the data type as shown in the first example. Then, if the user doesn't enter a value for this property, the MVC framework generates a default error message.

The second example shows that you can decorate a property with more than one validation attribute. In this example, both the `Required` and `Range` attributes are used to decorate the property. Here, the `Required` attribute works the same as it did in the first example. In addition, the `Range` attribute specifies a minimum value of 1 and a maximum value of 500. As a result, if the user doesn't enter a value that's within the specified range, the MVC framework generates a default error message.

Although the default error messages generated by the MVC framework are adequate in some cases, it's a good practice to specify user-friendly error messages as shown in the third example. To do that, you can pass an argument named `ErrorMessage` as the last argument of the attribute. In this example, the `Required` attribute specifies an error message of "Please enter a monthly investment amount." This is more user-friendly than the default message of "The field `MonthlyInvestment` is required." Similarly, the `Range` attribute specifies an error message of "Monthly investment amount must be between 1 and 500." This is more user-friendly than the default message of "The field `MonthlyInvestment` must be between 1 and 500."

## How to import the DataAnnotations namespace

```
using System.ComponentModel.DataAnnotations;
```

## Two common validation attributes

Attribute	Description
<b>Required</b>	Indicates that a value is required for the property.
<b>Range</b> (min, max)	Indicates that the value for the property must be within a specified range of values.

## A model property with a validation attribute

```
[Required]
public decimal? MonthlyInvestment { get; set; }
```

### The default error message if the property isn't set

The field `MonthlyInvestment` is required.

## A model property with two validation attributes

```
[Required]
[Range(1, 500)]
public decimal? MonthlyInvestment { get; set; }
```

### The default error message if the property isn't in a valid range

The field `MonthlyInvestment` must be between 1 and 500.

## A model property with user-friendly error messages

```
[Required(ErrorMessage = "Please enter a monthly investment amount.")]
[Range(1, 500, ErrorMessage =
    "Monthly investment amount must be between 1 and 500.")]
public decimal? MonthlyInvestment { get; set; }
```

## Description

- The process of checking data to make sure it's valid is known as *data validation*.
- You can use the *validation attributes* of the `DataAnnotations` namespace to add *validation rules* to your model.
- For the `Required` attribute to work properly, the data type for the property must be nullable.
- If you don't specify an error message, the data validation attributes generate a default error message.
- To specify a custom error message, you can pass an argument named `ErrorMessage` as the last argument of the attribute.

## **The model class with data validation**

---

Figure 2-19 shows the entire model class after data validation attributes have been added to its three properties. In addition, since these properties now use nullable data types, the result of any calculation that uses these properties must be assigned to a nullable type, and the return type for the `CalculateFutureValue()` method must also be nullable. Other than that, you shouldn't have much trouble understanding this model class. All three of the properties are decorated with both the `Required` and `Range` attributes. In addition, all three of the properties specify user-friendly error messages.



## The model class with data validation attributes

```
using System.ComponentModel.DataAnnotations;

namespace FutureValue.Models
{
    public class FutureValueModel
    {
        [Required(ErrorMessage = "Please enter a monthly investment.")]
        [Range(1, 500, ErrorMessage = "Monthly investment amount must be between 1 and 500.")]
        public decimal? MonthlyInvestment { get; set; }

        [Required(ErrorMessage = "Please enter a yearly interest rate.")]
        [Range(0.1, 10.0, ErrorMessage = "Yearly interest rate must be between 0.1 and 10.0.")]
        public decimal? YearlyInterestRate { get; set; }

        [Required(ErrorMessage = "Please enter a number of years.")]
        [Range(1, 50, ErrorMessage = "Number of years must be between 1 and 50.")]
        public int? Years { get; set; }

        public decimal? CalculateFutureValue()
        {
            int? months = Years * 12;
            decimal? monthlyInterestRate = YearlyInterestRate / 12 / 100;
            decimal? futureValue = 0;
            for (int i = 0; i < months; i++)
            {
                futureValue = (futureValue + MonthlyInvestment) *
                    (1 + monthlyInterestRate);
            }
            return futureValue;
        }
    }
}
```

## Description

- When you use a property with a nullable type in a calculation, the result must be assigned to a nullable type.
- If a method returns a nullable type, the return type must be defined as nullable in the method declaration.

Figure 2-19 The model class with data validation

## How to check the data validation

---

The first example in figure 2-20 shows the `Index()` action for a POST request in the Home controller. Here, the code has been modified so it uses the `ModelState` property that's available from the controller class to check whether the data in the model is valid. If so, this code calculates the future value and sets the `FV` property of the `ViewBag` to the result of the calculation. Otherwise, this code sets the `FV` property of the `ViewBag` to 0. That way, the view can display the result of the calculation or the error messages depending on the state of the model. Either way, it passes the model object to the view so the values entered by the user are redisplayed.

## How to display validation error messages

---

The second example shows the `<form>` element of the Index view. Here, the view includes code that displays a summary of all data validation errors in the model. In particular, within the `<form>` element, the first `<div>` element includes a tag helper named `asp-validation-summary` that specifies a value of "All". As a result, if the user enters valid data, the MVC framework hides this `<div>` element. However, if the user doesn't enter valid data, the MVC framework displays this `<div>` element and fills it with a list of all validation error messages that apply to the current model.

## The Future Value app after validating data

---

The third example shows that the Future Value app displays validation error messages above the form when a user enters invalid data. Here, the messages indicate that the monthly investment is required, the yearly interest rate is out of range, and the number of years is out of range.

For now, that's all you need to know about validating data in your web apps. Later, in chapter 11, you'll learn more about data validation.

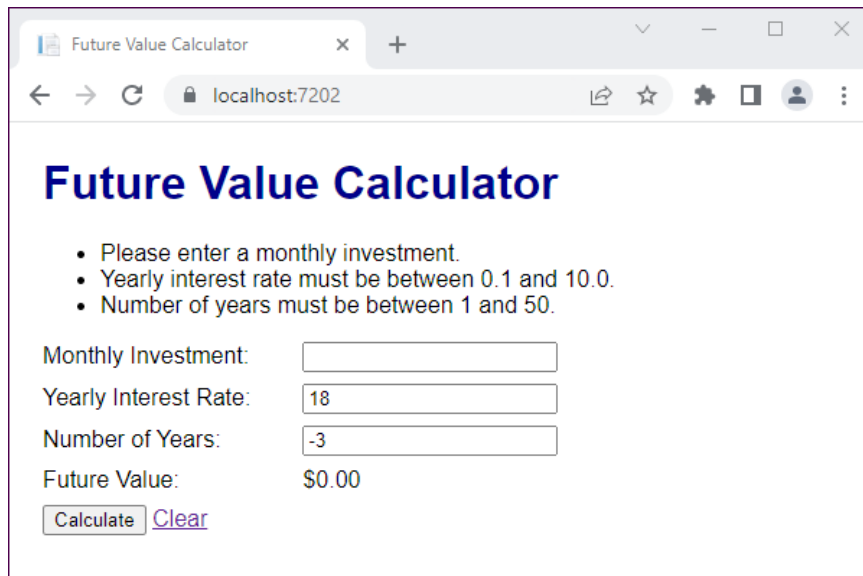
## An action method that checks for invalid data

```
[HttpPost]
public IActionResult Index(FutureValueModel model)
{
    if (ModelState.IsValid)
    {
        ViewBag.FV = model.CalculateFutureValue();
    }
    else
    {
        ViewBag.FV = 0;
    }
    return View(model);
}
```

## A view that displays a summary of validation messages

```
<form asp-action="Index" method="post">
  <div asp-validation-summary="All"></div>
  <div>
    <label asp-for="MonthlyInvestment">Monthly Investment:</label>
    <input asp-for="MonthlyInvestment" />
  </div>
  <!-- rest of input form -->
</form>
```

## The Future Value app with invalid data



The screenshot shows a web browser window titled "Future Value Calculator" at the address "localhost:7202". The page displays the title "Future Value Calculator" in a large blue font. Below the title, there are three bullet points indicating validation errors:

- Please enter a monthly investment.
- Yearly interest rate must be between 0.1 and 10.0.
- Number of years must be between 1 and 50.

Below the messages, there are four input fields with labels:

- Monthly Investment: (empty text box)
- Yearly Interest Rate: (text box containing "18")
- Number of Years: (text box containing "-3")
- Future Value: (text box containing "\$0.00")

At the bottom, there are two buttons: "Calculate" and "Clear".

## Description

- A controller can use the ModelState property that's available from the controller class to check whether the data in the model is valid.
- A view can use the tag helper named asp-validation-summary to display a summary of all data validation errors in the model.

Figure 2-20 How to check data validation and display error messages

## Perspective

---

The purpose of this chapter has been to teach you the basic skills for creating a one-page ASP.NET Core MVC app with Visual Studio. If you've already used Visual Studio and C# to develop other apps, such as Windows Forms apps, and you have basic HTML and CSS skills, you shouldn't have any trouble mastering these skills.

In the next chapter, you'll learn the basics of using Bootstrap. This open-source library provides CSS and JavaScript classes that make it easy to give your pages a professional appearance. In addition, Bootstrap makes it possible to display your web pages on devices of varying sizes.

## Terms

---

Visual Studio template	Razor view imports page
controller	strongly-typed view
action method	decorate a method or property
action	tag helper
Razor view engine	CSS style sheet
Razor view	Razor layout
Razor code block	Razor view start
Razor expression	data validation
syntax error	validation attributes
exception	validation rules
model	

## Summary

---

- You create a web app from a *Visual Studio template* that determines the folders and files for the project.
- A method of a *controller* class that runs in response to HTTP action verbs such as GET or POST is known as an *action method*, or an *action*.
- In ASP.NET Core, the *Razor view engine* uses server-side code to embed C# code within HTML elements.
- A *Razor view* contains both C# and HTML code. That's why its file extension is .cshtml. A Razor view typically stores elements that are unique to a web page.
- To execute one or more C# statements, you can declare a *Razor code block* by coding the @ sign followed by a pair of braces ( { } ). Within the braces, you can code one or more C# statements.
- To evaluate a C# expression and display its result, you can code a *Razor expression* by coding the @ sign followed by the expression.

- When you enter code or attempt to build and run an app, Visual Studio may display *syntax errors* that have to be corrected before the app can be compiled.
- If a compiled statement can't be executed when you run a web app, an *exception* occurs. Then, you can use the information that's displayed in the browser to attempt to fix the exception.
- A *model* is a regular C# class that models the data for the app. The class for a model is typically stored in the Models folder.
- A *Razor view imports page* makes it easier to work with models and tag helpers. As a result, most web apps include a Razor view imports page.
- You use the `@model` directive to *bind* a model to a view. This kind of view is called a *strongly-typed view*.
- You can use the `@Model` property to access the properties and methods of the model object that's specified by the `@model` directive.
- *Tag helpers* automatically generate attributes for some HTML elements. They can also *bind* HTML elements to the properties of the object that's the model for the view.
- A *CSS style sheet* provides a way to store the formatting for multiple web pages in a single external file.
- A *Razor layout* provides a way to store elements that are common to multiple web pages in a single file.
- A *Razor view start* lets you specify the default Razor layout for the Razor views of a web app.
- The process of checking data to make sure it's valid is known as *data validation*.
- You can use the *validation attributes* to add *validation rules* to your model.

## Before you do the exercises for this book...

---

If you haven't already done so, you should install the software that's required for this book, and you should download the source code for this book. Appendixes A (Windows) and B (macOS) show how to do that.

### Exercise 2-1      **Build the Future Value app using the MVC template**

This exercise guides you through the development of the Future Value app that's presented in this chapter. This gives you some hands-on experience using Visual Studio to build a web app.

#### **Create and set up a web app using the MVC template**

1. Start a web app that's based on the ASP.NET Core MVC template as shown in figures 2-1 and 2-2. Use a project name of FutureValue and a solution name of Ch02Ex1FutureValue and store it in this directory:  
`/aspnet_core_mvc/ex_starts`
2. Delete all the files inside the Controllers, Models, and Views folders, including the files inside the Views/Home and Views/Shared folders, but don't delete the Home and Shared folders themselves.
3. Add a controller named HomeController to the Controllers folder and modify it so it contains the code from figure 2-4.
4. Add a new empty Razor view named Index to the Views/Home folder and modify it so it contains the code from figure 2-5.
5. Press Ctrl+F5 to run the app. This should start the default web browser and display the Home/Index view, including the data that the HomeController stored in the ViewBag. Be sure to install any SSL certificates if you're asked to do that.

#### **Add the model, Razor view imports page, and a strongly typed view**

6. Add a class named FutureValueModel to the Models folder and modify it so it contains the code from figure 2-9.
7. Add the Razor view imports page to the Views folder and modify it so it contains the code shown in figure 2-10.
8. Modify the code of the Home/Index view so it contains the code from figure 2-11. Make sure to include all the CSS style rules from figure 2-14 within the <style> element.
9. Modify the HomeController class to handle both GET and POST requests as shown in figure 2-12.
10. Run the app. If you enter valid data, it should calculate and display a future value. However, if you enter invalid data, you may get unexpected results.

**Add the Razor layout and view start, and modify the Razor view**

11. Modify the site.css file in the wwwroot/css folder so it contains the CSS style rules shown in figure 2-14. To do that, you can cut the CSS style rules from the Home/Index file and paste them into the custom.css file.
12. Add a Razor layout named \_Layout.cshtml to the Views/Shared folder and modify it so it contains the code shown in figure 2-16. Make sure to include a <link> element that points to the site.css file.
13. Add a Razor view start named \_ViewStart to the Views folder (not the Views/Shared folder) and modify it so it contains the code shown in figure 2-16.
14. Modify the code in Home/Index view so it contains the code shown in figure 2-17. To do that, you can cut all elements that are already specified by the Razor layout. Note that you do not need to change the code that sets the Title property of the ViewBag so it uses ViewData, since either of these objects will work.
15. Run the app. It should work the same as it did before.

**Add data validation to the Future Value app**

16. Modify the FutureValueModel class so it imports the DataAnnotations namespace. Then, decorate each property with the Required and Range attributes as shown in figure 2-19. To do that, you must use nullable types for the properties and the method.
17. Modify the HomeController class so it checks for invalid data as shown in figure 2-20.
18. Modify the Home/Index view so it displays a summary of validation messages as shown in figure 2-20.
19. Run the app. It should work correctly if you enter valid data, and it should display appropriate messages if you enter invalid data.

**Exercise 2-2 Build the Future Value app using the Empty template**

This exercise guides you through the development of the Future Value app that's presented in this chapter if you start from the Empty template instead of the MVC template.

**Create and set up a web app using the Empty template**

1. Start a web app that's based on the Empty template as shown in figures 2-1 and 2-2. Use a project name of FutureValue and a solution name of Ch02Ex2FutureValue and store it in this directory:  
`/aspnet_core_mvc/ex_starts`
2. Add the Controllers, Models, and Views folders to the project.
3. Add a Home folder and Shared folder within the Views folder that you just created.
4. Follow exercise 2-1 starting at step 3. In step 11, you'll need to add the wwwroot folder and the css subfolder before you can add the site.css file.





# How to build your ASP.NET MVC web programming skills

The easiest way is to let [Murach's ASP.NET Core MVC \(2nd Edition\)](#) be your guide! So if you've enjoyed this chapter, I hope you'll get your own copy of the book today. You can use it to:

- Teach yourself how to develop professional, responsive web apps that follow the MVC pattern and work with databases
- Pick up new skills whenever you want to or need to by focusing on material that's new to you
- Look up coding details or refresh your memory on forgotten details when you're in the middle of developing a web app
- Loan to your colleagues who are always asking you questions about ASP.NET Core MVC programming



Mike Murach, Publisher

To get your copy, you can order online at [www.murach.com](http://www.murach.com) or call us at 1-800-221-5528 (toll-free in the U.S. and Canada). And remember, when you order directly from us, this book comes with my personal guarantee:

## 100% Guarantee

*When you buy directly from us, you must be satisfied. Try our books for 30 days or our eBooks for 14 days. They must outperform any competing book or course you've ever tried, or return your purchase for a prompt refund....no questions asked.*

Thanks for your interest in Murach books!

A handwritten signature in black ink that reads "Mike".