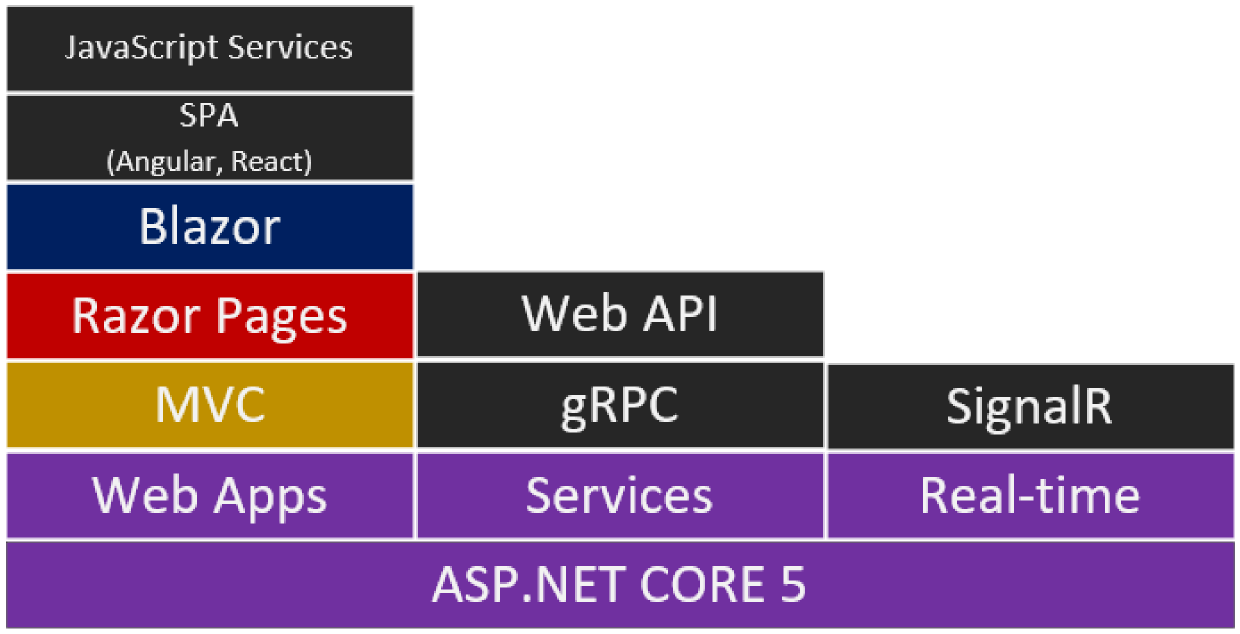
Razor View Engine

Building dynamic and data-driven web applications is pretty easy; however, things can sometimes be confusing, especially if you are new to the technology. As a beginner, you might find yourself having a hard time understanding how the stateless nature of the web works. The main reason for this is either you have never been exposed to how to apply the framework or simply because you are completely new to web development and you have no idea where to begin.

Even though there are many tutorials that you can use as a reference to learn, you may still find it hard to connect pieces, which could result in you losing interest. The good news is that ASP.NET Core makes things easier for you to learn how to carry out web development. As long as you understand C#, basic HTML, and CSS, you should be able to learn web development in no time. If you are new, confused, and have no idea how to start building an ASP.NET Core application, then this module is for you.

This module is mainly targeted at beginner to intermediate .NET developers who want to jump into ASP.NET Core 5, get a feel of the different web frameworks, and get their hands dirty with coding examples.

As you may know, there are lots of technologies that you can choose to integrate certain capabilities with ASP.NET Core, as shown in figure below:



In the preceding diagram, you can see that ASP.NET Core provides most of the common capabilities that you can integrate with your application. This gives you the flexibility to choose whatever framework and services you want to use when building your application. In fact, you can even combine any of these frameworks to produce powerful applications. Bear in mind though that we won't be covering all the technologies shown in the preceding diagram in this module.

In this module, we will mainly be focusing on the Web Apps stack by looking at a couple of web framework flavors that you can choose for building web applications in ASP.NET Core. We'll cover the basics of MVC and Razor Pages by doing some hands-on coding exercises so that you get a feel of how each of them works and understand their differences.

Here is a list of the main topics that we'll go through in this module:

- Understanding the Razor view engine
- Learning the basics of Razor syntax
- Building a to-do application with MVC
- Building a to-do application with Razor Pages
- Differences between MVC and Razor Pages

By the end of this module, you should understand the fundamentals of the Razor view engine and its syntax and know how to build a basic, interactive, data-driven web application using two of the popular web frameworks that ship with ASP.NET Core. You should then be able to weigh in on their pros and cons and decide which web framework is best suited for you. Finally, you'll understand when to use each web framework when building real-world ASP.NET Core applications.

Technical requirements

This module uses Visual Studio 2019 to demonstrate various examples, but the process should be the same if you're using Visual Studio Code.

Before diving into this module, make sure that you have a basic understanding of ASP.NET Core and C# in general and how each of them works separately, as well as together. Though it's not required, having a basic knowledge of HTML and CSS is helpful for you to easily understand how the pages are constructed.

Understanding the Razor view engine

Before we deep dive into the Razor view engine in the context of ASP.NET Core, let's talk a bit about the history of the various view engines in ASP.NET.

The previous versions of the ASP.NET frameworks had their own view/markup engines for rendering dynamic web pages. Back in the old days, Active Server Pages (Classic ASP) used a .ASP file extension. ASP.NET Web Forms, which is commonly known as the Web Forms view engine, used a .ASPX file extension. These file types were markup engines that contained server-side code, such as VBScript, VB.NET, or C#, which were processed by the web server (IIS) to output HTML in the browser. A few years later, after ASP.NET Web Forms became popular, Microsoft introduced ASP.NET MVC 1.0 as a new, alternative web framework for building dynamic web applications in the full .NET Framework. Bringing MVC into .NET opened it up to a wider audience of developers, because it values the clean separation of concerns and friendly URL routings, allows deeper extensibility, and follows real web development experience.

While the early versions of MVC addressed most of the Web Forms downsides, they still used the .ASPX-based markup engine to serve up pages. Many were not glad about the integration of the .ASPX markup engine in MVC, because as it was too complex to work with the UI. It could potentially affect the overall performance of the application due to its processing overhead. When Microsoft released ASP.NET MVC 3.0 in early January 2011, the Razor view engine came to life as a new view engine addition to power ASP.NET MVC views. The Razor view engine in the ASP.NET full .NET Framework supports both VB.NET (.vbhtml) and C# (.cshtml) as the server-side language.

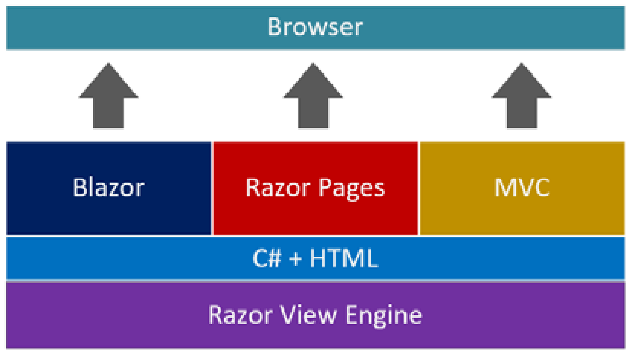
When ASP.NET Core was introduced, a lot of things were changed for the better. Since the framework was redesigned to be modular, unified, and cross-platform, many features and capabilities from the full .NET framework were discontinued, such as Web Forms and VB.NET support. Because of these changes, the Razor view engine also dropped support for the .vbhtml file extension, leaving it to only support C# code.

Now that you have a little bit of background about the various view engines in the different ASP.NET web frameworks, let's move on to the next section. There, you will better understand why the ASP.NET team came to the decision to use the Razor view engine as the default markup engine to power all ASP.NET Core web frameworks.

Reviewing the Razor view engine

As the ASP.NET Core framework has evolved, the ASP.NET Core team has been working hard to provide a better view engine that offers a lot of benefits and productivity. The new Razor view engine is the default view engine for all ASP.NET Core web frameworks, and it has been optimized to provide us with faster HTML generation using a code-focused templating approach.

The Razor view engine, often referred to as Razor, is a C#-based template markup syntax for generating HTML with dynamic content. It's the view engine that powers not just the ASP.NET Core MVC, but all other ASP.NET Core web frameworks for generating dynamic pages (as shown in figure below).



In the preceding diagram, we can see that the Blazor, Razor Pages, and MVC web frameworks rely on Razor view engines to generate content pages and components. Blazor differs a bit from MVC and Razor Pages because it a single-page application (SPA) web framework that uses a component-based approach. Blazor components are files that use the .razor extension, which still uses the Razor engine under the hood. Content pages, often referred to as the UI, are simply Razor files with the .cshtml extension. Razor files are mainly composed of the HTML and Razor syntax, which enables you to embed C# code in the content itself. So, if you request a page, the C# code gets executed on the server. It then processes whatever logic it requires, takes data from somewhere, and then returns the generated data, along with the HTML that makes up the page, to the browser.

Having the ability to use the same templating syntax for building up your UI enables you to easily transition from one web framework to another without much of a learning curve. In fact, you can combine any of the web frameworks for building web applications.

However, it's not recommended to do so, as things can get messy and it may cause your application code to be difficult to maintain. One exception, though is if you are migrating your whole application from one web framework to another, and you want to start replacing portions of your application to use other web frameworks; then, it makes a lot of sense to combine them.

Razor offers a lot of benefits, including the following:

- Easy to learn: As long as you know basic HTML and a little bit of C#, then learning Razor is quite easy and fun. Razor was designed to enable C# developers to take advantage of their skills and boost productivity when building UIs for their ASP.NET Core applications.
- Clean and fluid: Razor was designed to be compact and simple and does not require you to write a lot of code. Unlike other view templating engines, where you need to specify certain areas within your HTML to denote a server-side code block, the Razor engine is smart enough to detect server code in your HTML, which enables you to write clean and more manageable code.
- Editor-agnostic: Razor isn't tied to a specific editor like Visual Studio. This enables you to write code in whatever text editor you prefer to improve productivity.
- IntelliSense support: While you can write Razor-based code in any text editor, using Visual Studio can boost your productivity even more because of the statement completion support built into it.
- Ease of unit testing: Razor-based pages/views support unit tests.

Understanding how the Razor view engine works is very important when building dynamic and interactive pages in ASP.NET Core. In the next section, we'll discuss some of the basic syntaxes of Razor.

Learning the basics of Razor syntax

The beauty of Razor, compared to other templating view engines, is that it minimizes the code required when constructing your views or content pages. This enables a clean, fast, and fluid coding workflow to boost your productivity when composing UIs.

To embed C# code into your Razor files (.cshtml), you need to tell the engine that you are injecting a server-side code block by using the @ symbol. Typically, your C# code block must appear within the @{…} expression. This means that as soon as you type @, the engine is smart enough to know that you are starting to write C# code. Everything that follows after the opening { symbol is assumed to be server-side code, until it reaches the matching closing block } symbol.

Let's take a look at some examples for you to better understand the Razor syntax basics.

Rendering simple data

In a typical ASP.NET Core MVC web application generated from the default template, you'll see the following code within the Index.cshtml file for the home page:

```
@{
  ViewData["Title"] = "Home Page";
}
```

The preceding code is referred to as a Razor code block. Razor code blocks normally start with the @ symbol and are enclosed by curly braces {}. In the preceding example, you'll see that the line starts with the @ symbol, which tells the Razor engine that you are about to embed some server code. The code within the open and close curly braces are assumed to be C# code. The code within the block will be evaluated and executed from the server, allowing you to access the value and reference it in your view. This example is the same as setting a variable in the Controller class.

Here's another example of creating a new ViewData variable and assigning a value to it in the Index() method of the HomeController class, as shown in the following code block:

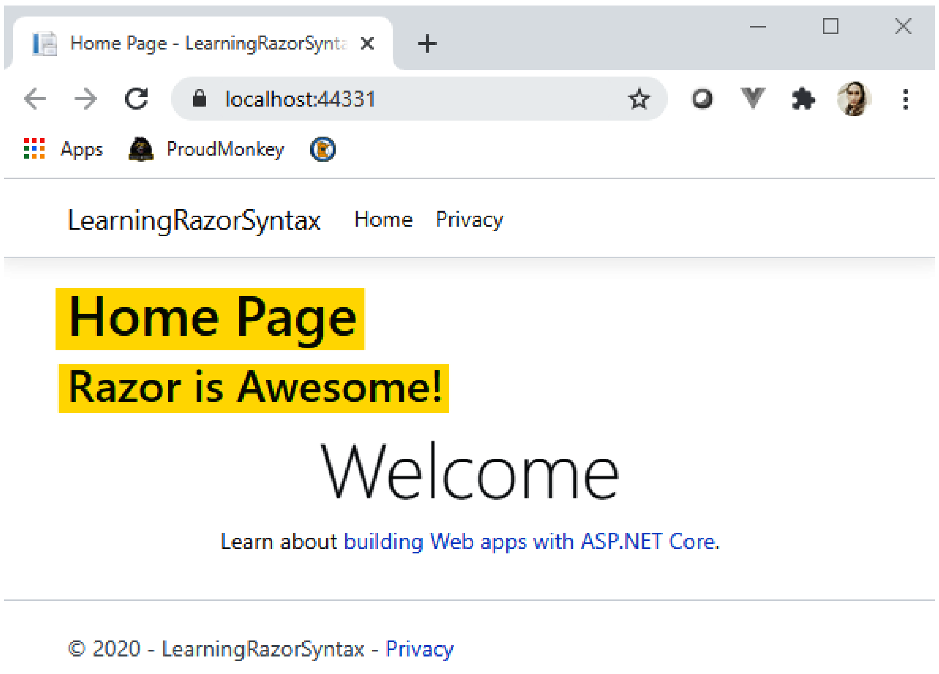
```
public IActionResult Index() {
  ViewData["Message"] = "Razor is Awesome!";
  return View();
}
```

In the preceding example, we've set the ViewData["Message"] value to "Razor is Awesome!". ViewData is nothing but a dictionary of objects, and it is accessible by using string as the key. Now, let's try to display the values of each ViewData object by adding the following code:

```
<h1>@ViewData["Title"]</h1>
<h2>@ViewData["Message"]</h2>
```

The preceding code is an example of implicit Razor expressions. These expressions normally start with the @ symbol and are then followed by C# code. Unlike Razor codeblocks, Razor expression code is rendered into the browser.

In the preceding code, we've referenced the values of both ViewData["Title"] and ViewData["Message"], then contained them within the <h1> and <h2> HTML tags. The value of any variable is rendered along with the HTML. Following figure shows you the sample output of what we just did.


Home Page - LearningRazorSynta

localhost:44331

Apps ProudMonkey

LearningRazorSyntax Home Privacy

Home Page

Razor is Awesome!

Welcome

Learn about building Web apps with ASP.NET Core.

© 2020 - LearningRazorSyntax - Privacy

In the preceding screenshot, we can see that each value from ViewData is printed on the page. This is what Razor is all about; it enables you to mix HTML with server-side code using a simplified syntax.

The Razor implicit expressions described in the previous example typically should not contain spaces, with the exception of using the C# await keyword:

```
<p>@await SomeService.GetSomethingAsync()</p>
```

The await keyword in the preceding code denotes an asynchronous call to the server by invoking the GetSomethingAsync() method of the SomeService class. Razor allows you to inject a server-side method into your content page using view injection. For more information about dependency injection, you can review module Dependency Injection.

Implicit expressions also do not allow you to use C# generics, as in the following code:

```
<p>@SomeGenericMethod<T>()</p>
```

The reason why the preceding code won't work and will throw an error is that data type T within the <> brackets is parsed as an HTML tag. To use generics in Razor, you would need to use a **Razor code block** or **explicit expression**s, just like in the following code:

```
<p>@(SomeGenericMethod<T>())</p>
```

Razor explicit expressions start with the @ symbol with balanced matching parentheses. Here's an example of an explicit expression that displays the date from yesterday:

```
<p>
  @((DateTime.Now - TimeSpan.FromDays(1)).ToShortDateString())
</p>
```

The preceding code gets yesterday's date and uses the ToShortDateString() extension method to convert the value into a short date format. Razor will process the code within the @() expression and render the result to the page.

Razor will ignore any content containing the @ symbol in between text. For example, the following line remains untouched by Razor parsing:

```
<a href="mailto:user@email.com">user@email.com</a>
```

Explicit expressions are useful for string concatenation as well. For example, if you want to combine static text with dynamic data and render it, you can do something like this:

```
<p>Time@(DateTime.Now.Hour) AM</p>
```

The preceding code will render something like <p>Time@10 AM</p>. Without using the explicit @() expression, the code will render as <p>Time@DateTime.Now.Hour AM</p> instead. Razor will evaluate it as plain text like an email address.

If you want to display static content that includes an @ symbol before the text, then you can simply append another @ symbol to escape it. For example, if we want to display the text @vmsdurano on the page, then you can simply do something such as the following:

```
<p>@@vmsdurano</p>
```

Now that you've learned how the basic syntax of Razor works, let's move on to the next section and take a look at some advanced examples.

Rendering data from a view model

In most cases, you would typically be dealing with real data to present dynamic content on a page when working with a real application. This data would normally come from ViewModel, which holds some information related to the content that you are interested in.

In this section, we'll see how we can present data that comes from the server on your page using the Razor syntax. Let's start off by creating the following class in the Models folder of your MVC application:

```
public class BeerModel {
  public int Id { get; set; }
  public string Name { get; set; }
  public string Type { get; set; }
}
```

The preceding code is just a plain class that represents ViewModel. In this case, ViewModel is called BeerModel, which houses some properties that the view expects.

Next, we'll create a new class that will populate the view model. The new class would look something like this:

```
public class Beer {
  public List<BeerModel> GetAllBeer() {
    return new List<BeerModel>{
      new BeerModel { Id =1, Name="Redhorse",Type="Lager" },
      new BeerModel { Id =2, Name="Furious", Type="IPA"},
      new BeerModel { Id =3, Name="Guinness",Type="Stout"},
      new BeerModel { Id =4, Name="Sierra", Type="Ale" },
      new BeerModel { Id =5, Name="Stella", Type="Pilsner"},
    };
  }
}
```

The preceding code is nothing but a plain class that represents the model. This class contains a GetAllBeer() method, which is responsible for returning all items from the list. In this case, we are returning a List<BeerModel> type. The implementation could vary depending on your datastore and what data access framework you're using. You could be pulling the data from a database or via an API call. However, for this example, we will just return a static list of data for simplicity's sake.

You can think of ViewModel as a placeholder to hold properties that are only required for your views. Model, on the other hand, is a class that implements the domain logic for your application. Often, these classes are retrieved and store data in databases. We'll talk more about these concepts later in this module.

Now that we already modeled some sample data, let's modify our Index() method of the HomeController class so that it looks something like this:

```
public IActionResult Index() {
  var beer = new Beer();
  var listOfBeers = beer.GetAllBeer();
  return View(listOfBeers);
}
```

The preceding code initializes an instance of the Beer class and then invokes the GetAllBeer() method. We then set the result to a variable called listOfBeers and then pass it to the view as an argument to return the response.

Now, let's see how we can display the result on the page. Go ahead and switch back to the Index.cshtml file that is located in the Views/Home folder.

The first thing that we need to do for us to access the data from the view model is to declare a class reference using the @model directive:

```
@model IEnumerable<Module_04_LearningRazorSyntax.Models.BeerModel>
```

The preceding code declares a reference to the view model as a type of IEnumerable<BeerMode>, which makes the view a strongly typed view. The @model directive is one of the Razor reserved keywords. This particular directive enables you to specify the type of class to be passed in the view or page. Razor directives are also expressed as implicit expressions by using the @ symbol, followed by the directive name or Razor reserved keywords.

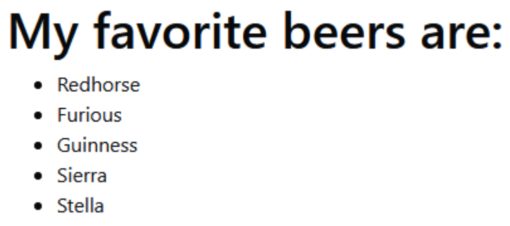
At this point, we now have access to the view model that we created earlier. Since we are declaring the view model as enumerable, you can easily iterate to each item in the collection and present the data however you want. Here's an example of displaying just the Name property of the BeerModel class:

```
<h1>My favorite beers are:</h1>
<ul>
  @foreach (var item in Model) {
    <li>@item.Name</li>
  }
</ul>
```

In the preceding code, we've used the <ul> HTML tag to present the data in a bulleted list format. Within the <ul> tag, you should notice that we've used the @ symbol to start manipulating the data in C# code. The foreach keyword is one of the C# reserved keywords, which are used for iterating data in a collection. Within the foreach block, we have constructed the items to be displayed in the <li> tag. In this case, the Name property is rendered using implicit expressions.

Notice how fluid and easy it is to embed C# logic into the HTML. The way it works is that Razor will look for any HTML tags within the expression. If it sees one, it jumps out of the C# code and will only jump back in when it sees a matching closing tag.

Here’s the output when rendered in the browser:



My favorite beers are:

- Redhorse
- Furious
- Guinness
- Sierra
- Stella

The preceding is just an example of how we can easily display a formatted list of data on a page. If you want to filter the list based on some condition, you can do something like this:

```
<ul>
  @foreach (var item in Model) {
    if (item.Id == 2) {
      <li>@item.Name</li>
    }
  }
</ul>
```

In the preceding code, we’ve used the C# if-statement within the foreach loop to filter only the item that we need. In this case, we checked to see whether the Id property is equal to 2 and then constructed an <li> element to display the value when the condition is met.

There are many ways to present information on the page depending on your requirements.In most cases, you may be required to present a complex UI to display information. In such cases, that’s where HTML and tag helpers can be useful.

Introduction to HTML helpers and tag helpers

Before tag helpers were introduced, HTML helpers were used to render dynamic HTML content in Razor files. Typically, you will find code that looks similar to this in the view of MVC applications:

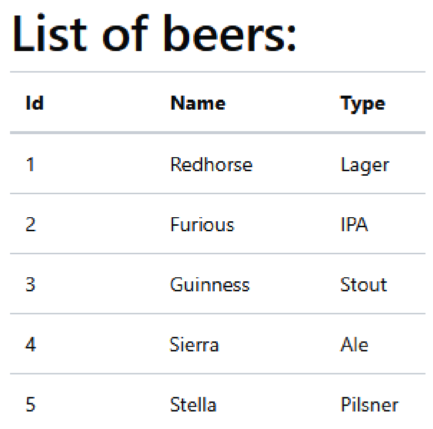
```
<h1>List of beers:</h1>
<table class=”table”>
  <thead>
    <tr>
      <th>
        @Html.DisplayNameFor(model => model.Id)
      </th>
      @* Removed other headers for brevity *@
    </tr>
  </thead>
  <tbody>
    @foreach (var item in Model) {
      <tr>
        <td>
          @Html.DisplayFor(modelItem => item.Id)
        </td>
        @* Removed other rows for brevity *@
      </tr>
    }
  </tbody>
</table>
```

The preceding code uses a <table> tag to present data in a tabular form. In the <thead> section, we've used the DisplayNameFor HTML helper to display each property name from the view model. We then iterated to each item within the <tbody> section using the C# foreach iterator. This is pretty much the same as what we did in our previous example. The difference now is we've constructed the data to be presented in tabular format.

The <tr> element represents the rows and the <td> element represents the columns. In each column, we've used the DisplayFor HTML helper to display the actual data in the browser. Keep in mind though that the DisplayFor helper doesn't generate any HTML tags when rendered; instead, it will just display the value in plain text. So, use DisplayFor only when there's a reason for you to use it. Ideally, the foreach block from the preceding code can be replaced with this code:

```
<tbody>
  @foreach (var item in Model) {
    <tr>
      <td>@item.Id</td>
      @* Removed other rows for brevity *@
    </tr>
  }
</tbody>
```

The preceding code is much cleaner and will render much faster, compared to using the DisplayFor HTML helper. Running the code should result an output like figure below:



List of beers:

Id	Name	Type
1	Redhorse	Lager
2	Furious	IPA
3	Guinness	Stout
4	Sierra	Ale
5	Stella	Pilsner

While other HTML helpers are useful when dealing with collections, complex objects, templates, and other situations, there are certain cases where things can become cumbersome, especially when dealing with UI customization. For example, if we want to apply some CSS style to elements that were generated by HTML helpers, then we will have to use the overload method to do that without any IntelliSense help. Here's a quick example:

```
<h1>My most favorite beer:</h1>
@{ var first = Model.FirstOrDefault(); }
@* Removed other line for brevity *@
@Html.LabelFor(model => first.Name, new { @class = “font - weight - bold” })
: @first.Name
@* Removed other line for brevity *@
```

The preceding code uses the LabelFor HTML helper to display information. In this example, we were only displaying the first item set from the ViewModel collection using the LINQ FirstOrDefault extension method. The second argument in the LabelFor method represents the htmlAttributes parameter, where we are forced to pass an anonymous object just to set the CSS class. In this case, we applied the CSS class attribute to font-weight-bold for the label element. The reason for this is that the class keyword is a reserved keyword in C#, thus we need to tell Razor to evaluate @class=expression as an element attribute by using the @ symbol before it. This kind of situation makes it a little bit harder to maintain and not quite friendly to read as your page gets bigger, especially to frontend developers who are not familiar with C#. To address this, we can use **tag helpers**.

ASP.NET Core offers a bunch of built-in tag helpers that you can use to help improve your productivity when creating and rendering HTML elements in the Razor markup. Unlike **HTML helpers**, which are invoked as C# methods, tag helpers are attached directly to HTML elements. This makes tag helpers much more friendly and fun to use for frontend developers because they can have full control over HTML.

While tag helpers is a huge topic to cover, we'll try to look at a common example for you to understand their purpose and benefits.

Going back to our previous example, we can rewrite the code using tag helpers with the following code:

```
<h1>My most favorite beer:</h1>
@* Removed other line for brevity *@
<label asp-for="@first.Name" class="font-weight-bold"></label>
: @first.Name
@* Removed other line for brevity *@
```

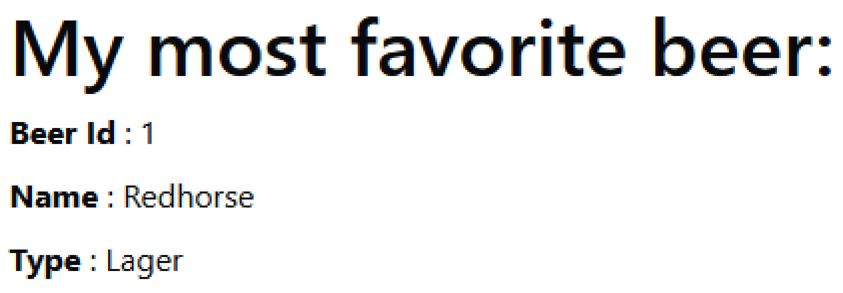
In the preceding code, notice that we have now used a standard <label> HTML tag and used the asp-for tag helper to display the Name property from ViewModel. Note that the closing tag is required. If you use a self-closing tag, such as `<label asp-for="@first.Id" />`, the value will not be rendered.

In cases where you want to change the property name to be rendered in the HTML, you can use the [Display] attribute. For example, if we want to display the value Beer Id for the property ID, we can simply do something like the following code:

```
Display(Name = "Beer Id")]
public int Id { get; set; }
```

What we did in the preceding code is called data annotation. This enables you to define certain metadata that you want to apply for properties in the model/view model, such as conditions, validations, custom formatting, and so on. For more information about data annotation, see https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations.

Figure below displays the sample output when running the code:


My most favorite beer:

Beer Id : 1

Name : Redhorse

Type : Lager

There are many things that you can do with tag helpers. ASP.NET Core provides most of the tag helpers that are common for building up your pages, such as form actions, input controls, routings, validations, components, scripts, and many others. In fact, you can even create your own or extend tag helpers to customize your needs.

Tag helpers give you a lot of flexibility when generating HTML elements, provide rich IntelliSense support, and provide an HTML-friendly development experience, which helps you save some development time when building UIs.

For more information about tag helpers in ASP.NET Core, see https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers.

Learning the basic fundamentals of the Razor view engine and understanding how the syntax works are crucial for building any ASP.NET Core web applications. In the following sections, we will do some hands-on exercises by building a to-do application in various web frameworks. This is to give you a better understanding of how each web framework works and help you decide which approach to choose when building real-world web applications.

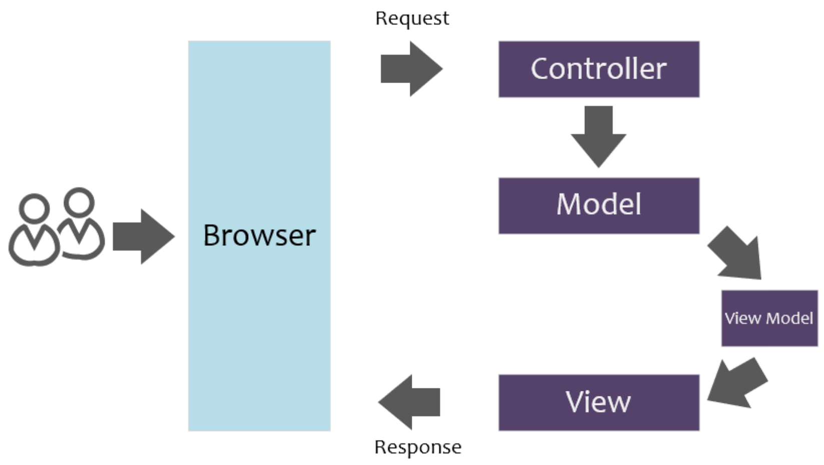
Building a to-do application with MVC

A to-do application is a great example to demonstrate how to perform adding and modifying information on a web page. Understanding how this works in the stateless nature of the web is of great value when building real-world, data-driven web applications.

Before we get started, let's take a quick refresher on MVC first so that you have a better understanding of what it is.

Understanding the MVC pattern

To better understand the MVC pattern approach, Figure 4.7 illustrates an attempt that describes the high-level process in a graphical way:



The preceding diagram is pretty much self-explanatory by just looking at the request flow. But to verify your understanding, it might be helpful to give a brief explanation of the process. The term MVC represents the three components that make up the application:

M for **Models**, **V** for **Views**, and **C** for **Controller**s. In the preceding diagram, you can see that the controller is the very first entry that is invoked when a user requests a page in the browser. Controllers are responsible for handling any user interactions and requests, processing any logic that is required to fulfill the request, and ultimately, returning a response to the user. In other words, controllers orchestrate the flow of logic.

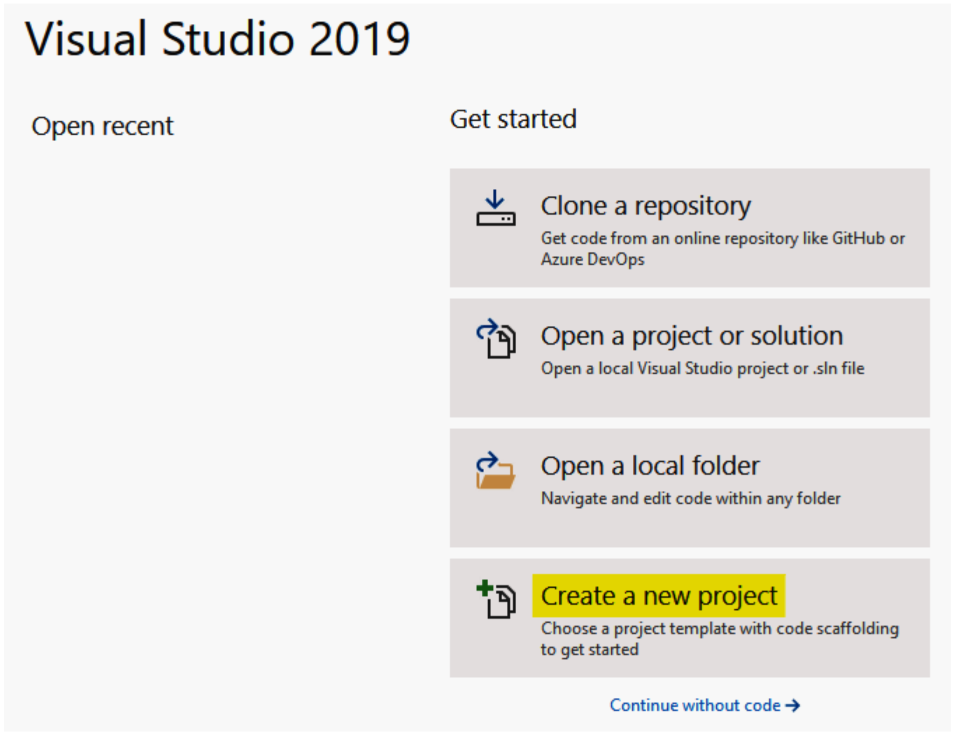
Models are components that actually implement domain-specific logic. Often, models contain entity objects, your business logic, and data access code that retrieves and stores data. Bear in mind though that in real applications, you should consider separating your business logic and data access layer to value the separation of concerns and single responsibility. ViewModel is simply a class that houses some properties that are only needed for the view. ViewModel is optional because you can technically return a model to a view directly. In fact, it is not part of the MVC term. However, it's worth including it in the flow because it is very useful and recommended when building real applications.

Adding this extra layer enables you to expose only the data that you need instead of returning all data from your entity object via models. Finally, views are components that make up your UI or page. Typically, views are just Razor files (.cshtml) that contain HTML, CSS, JavaScript, and C#-embedded code.

Now that you have an idea of how MVC works, let's start building a web application from scratch to apply these concepts and get a feel of the framework.

Creating an MVC application

Let's go ahead and fire up Visual Studio 2019, and then select the Create a new project box, as shown figure below:



Visual Studio 2019

Open recent

Get started

Clone a repository
Get code from an online repository like GitHub or Azure DevOps

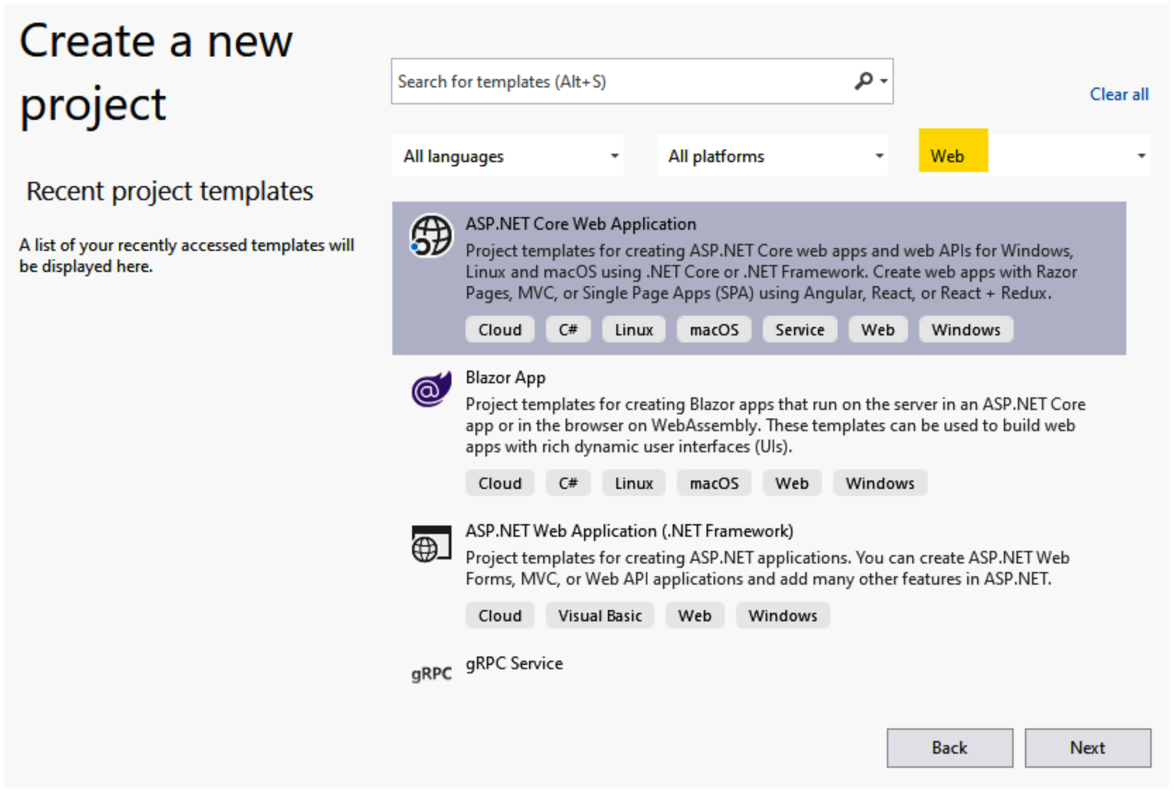
Open a project or solution
Open a local Visual Studio project or .sln file

Open a local folder
Navigate and edit code within any folder

Create a new project
Choose a project template with code scaffolding to get started

Continue without code →

The **Create a new project** dialog should show up. In the dialog, select **Web** as the project type, and then find the **ASP.NET Core Web** Application project template, as shown in in figure below:



Create a new project

Recent project templates

A list of your recently accessed templates will be displayed here.

Search for templates (Alt+S)

Clear all

All languages | All platforms | Web

ASP.NET Core Web Application
Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Angular, React, or React + Redux.
Cloud | C# | Linux | macOS | Service | Web | Windows

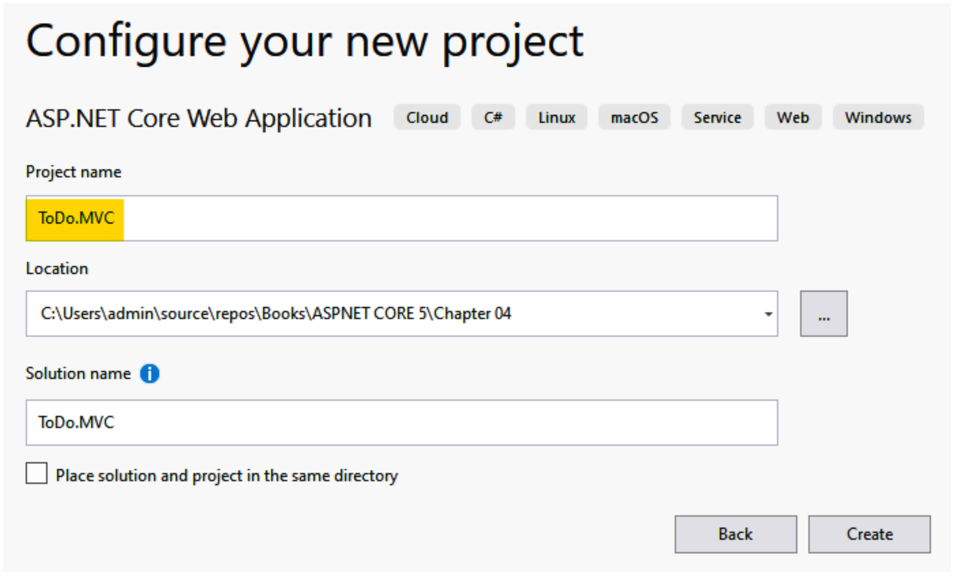
Blazor App
Project templates for creating Blazor apps that run on the server in an ASP.NET Core app or in the browser on WebAssembly. These templates can be used to build web apps with rich dynamic user interfaces (UIs).
Cloud | C# | Linux | macOS | Web | Windows

ASP.NET Web Application (.NET Framework)
Project templates for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.
Cloud | Visual Basic | Web | Windows

gRPC Service

Back | Next

To continue, double-click the ASP.NET Core Web Application template or simply click the Next button. The Configure your new project dialog should show up, as shown in figure below.



Configure your new project

ASP.NET Core Web Application

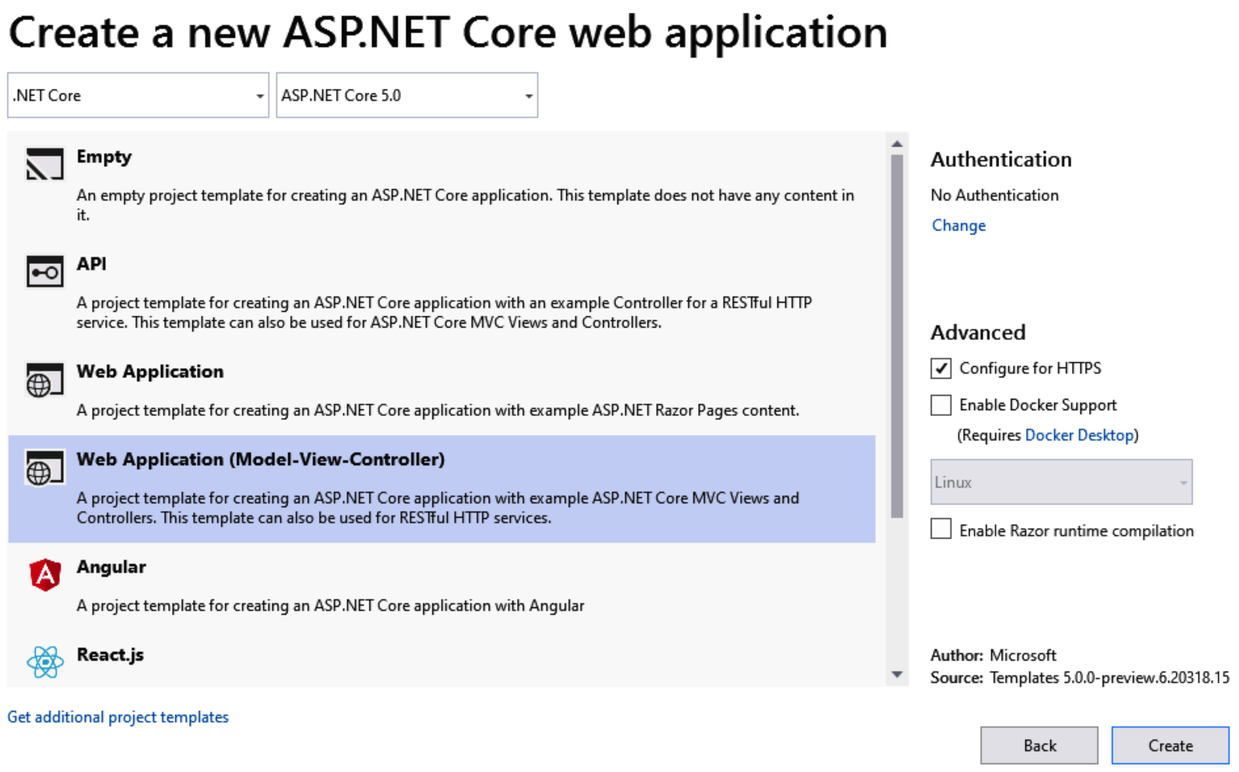
Project name: ToDo.MVC

Location: C:\Users\admin\source\repos\Books\ASPNET CORE 5\Chapter 04

Solution name: ToDo.MVC

☐ Place solution and project in the same directory

The preceding dialog allows you to configure your project name and the location path to where you would want the project to be created. In a real application, you should consider giving a meaningful name to your project that clearly suggests what the project is all about. In this example, we'll just name the project as ToDo.MVC. Now, click Create and it should bring up the dialog shown in figure below:



Create a new ASP.NET Core web application

.NET Core | ASP.NET Core 5.0

Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Web Application
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

Web Application (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Angular
A project template for creating an ASP.NET Core application with Angular

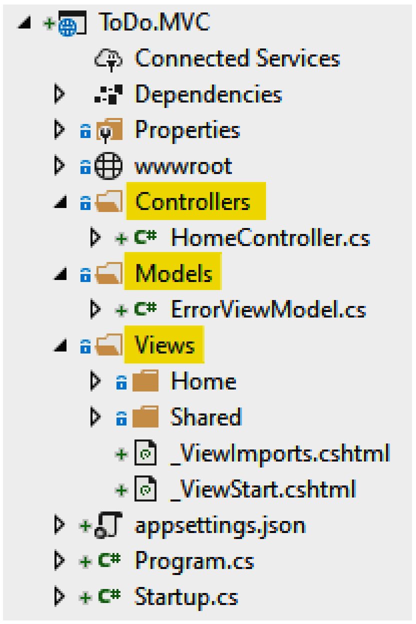
React.js

Authentication
No Authentication
Change

Advanced
☑ Configure for HTTPS
☐ Enable Docker Support
(Requires Docker Desktop)
Linux
☐ Enable Razor runtime compilation

Author: Microsoft
Source: Templates 5.0.0-preview.6.20318.15

The preceding dialog allows you to choose what type of web framework you want to create. For this example, just select **Web Application (Model-View-Controller)** and then click **Create** to let Visual Studio generate the necessary files for you. The default files generated should look something like figure below:



```
ToDo.MVC
    Connected Services
    Dependencies
    Properties
    wwwroot
    Controllers
        HomeController.cs
    Models
        ErrorViewModel.cs
    Views
        Home
        Shared
        _ViewImports.cshtml
        _ViewStart.cshtml
    appsettings.json
    Program.cs
    Startup.cs
```

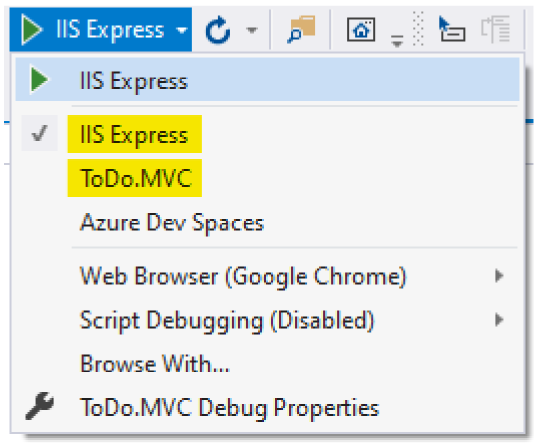
The preceding screenshot shows the default structure of the MVC application. You will notice that the template automatically generates the Models, Views, and Controllers folders. The names of each folder don't really matter in order for the application to function, but it's recommended and good practice to name the folders that way to conform with the MVC pattern. In MVC applications, functionalities are grouped into functions. This means that each folder that represents MVC will contain its own dedicated logical functions. Models contains data and validation; Views contains UI-related elements for displaying data, and Controllers contains actions that handle any user interactions.

If you already know the significant changes of the ASP.NET Core project structure, then you can skip this part, but if you are new to ASP.NET Core, then it's worth covering a few of the core files generated so that you have a better understanding of their purpose. Here's the anatomy of the core files aside from the MVC folders:

- **Connected Services:** Allows you to connect to services such as Application Insights, Azure Storage, mobile, and other ASP.NET Core services that your application depends on, without you having to manually configure their connection and configurations.
- **Dependencies:** This is where project dependencies are located, such as NuGet packages, external assemblies, the SDK, and framework dependencies needed for the application.
- **Properties:** This folder contains the launchSettings.json file, where you can define application variables and profiles for running the app.
- **wwwroot:** This folder contains all your static files, which will be served directly to the clients, including HTML, CSS, images, and JavaScript files.
- **appsettings.json:** This is where you configure application-specific settings. Keep in mind though that sensitive data should not be added to this file. You should consider storing secrets and sensitive information in a vault or secrets manager.
- **Program.cs:** This file is the main entry point for the application. This is where you build the host for your application. By default, the ASP.NET Core app builds a generic host that encapsulates all framework services needed to run the application.
- **Startup.cs:** This file is the heart of any .NET application. This is where you configure the services and dependencies required for your application.

Running the app for the first time

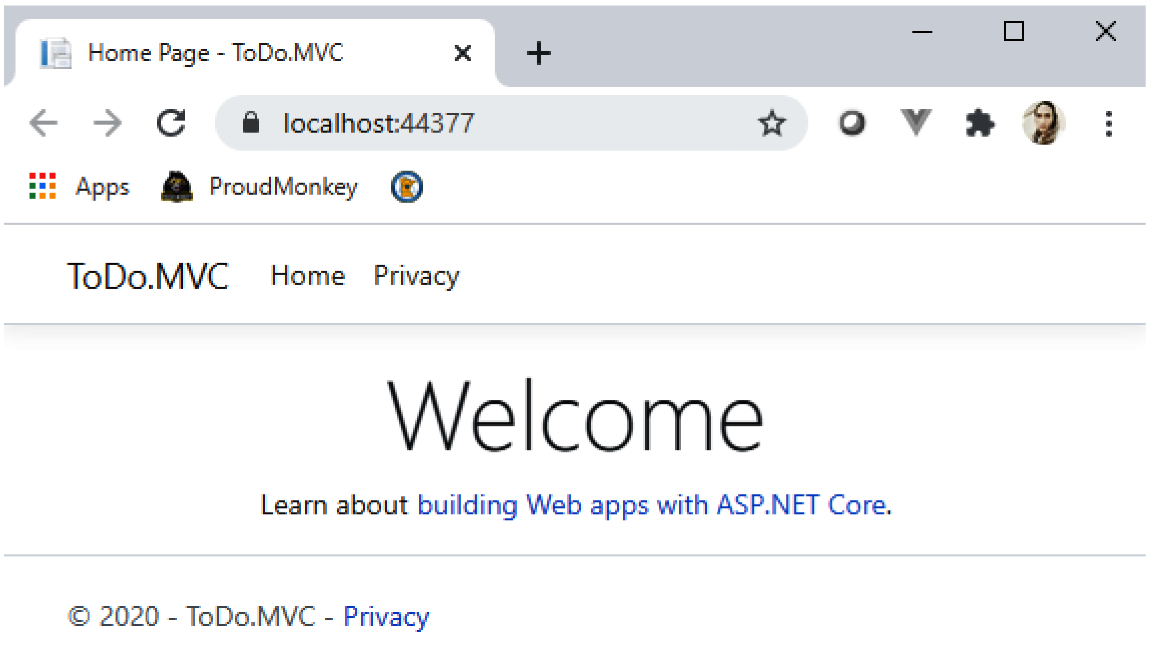
Let’s try to build and run the default generated template to ensure that everything is working. Go ahead and press the **Ctrl + F5** keyboard keys or simply click the play button located on the Visual Studio menu toolbar, as shown in figure below:


IIS Express
IIS Express
IIS Express
ToDo.MVC
Azure Dev Spaces
Web Browser (Google Chrome)
Script Debugging (Disabled)
Browse With...
ToDo.MVC Debug Properties

In the preceding screenshot, you will see that the default template automatically configures two web server profiles for running the app in localhost from inside Visual Studio:

IIS Express and ToDo.MVC. The default profile used is IIS Express and the ToDo.MVC profile runs on the Kestrel web server. You can see how this was configured by looking at the launchSettings.json file. For more information about configuring ASP.NET Core environments, see https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments.

Visual Studio will compile, build, and automatically apply whatever configuration you’ve set up for each profile in the application. If everything builds successfully, then you should be presented with the output shown in figure below:



Welcome

Learn about building Web apps with ASP.NET Core.

© 2020 - ToDo.MVC - Privacy

Configuring in-memory databases

One of the great features of ASP.NET Core is that it allows you to create a database in memory. This enables you to easily create a data-driven app without the need to spin up a real server for storing your data. With that said, we are going to take advantage of this feature in concert with Entity Framework (EF) Core so that we can play around with the data and dispose of it when no longer needed.

Working with a real database will be covered in module APIs and Data Access, as it mainly focuses on APIs and data access. For now, let's just use an in-memory working database for the sole purpose of this demo application.

Installing EF Core

The first thing we need to do is to add the Microsoft.EntityFrameworkCore and Microsoft.EntityFrameworkCore.InMemory NuGet packages as project references, so that we will be able to use EF as our data access mechanism to query data against the in-memory datastore. To do this, navigate to the Visual Studio menu, then go to Tools | NuGet Package Manager | Package Manager Console. In the console window, install each package by running the following commands:

```
Install-Package Microsoft.EntityFrameworkCore -Version 5.0.0
Install-Package Microsoft.EntityFrameworkCore.InMemory -Version 5.0.0
```

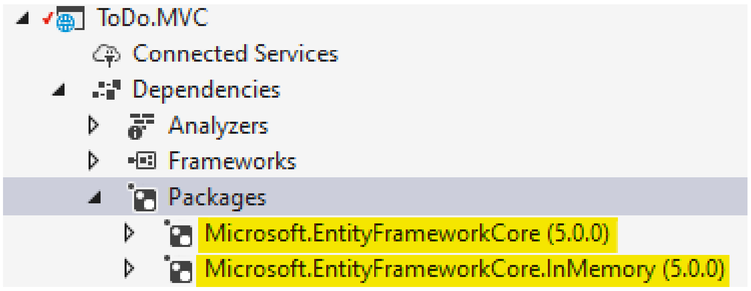
Each command in the preceding code will pull all the required dependencies needed for the application.

Note

The latest official version of Microsoft.EntityFrameworkCore as of the time of writing is 5.0.0. Future versions may change and could impact the sample code demonstrated in this module. So, make sure to always check for any breaking changes when deciding to upgrade to newer versions.

Another way to install NuGet dependencies in your project is using the **Manage NuGet Packages for Solution...** option, or by simply right-clicking on the Dependencies folder of the project and then selecting the Manage **NuGet Packages...** option. Both options provide a UI where you can easily search for and manage your project dependencies.

After successfully installing both packages, make sure to check your project **Dependencies** folder and verify whether they were added, just like in figure below.


ToDo.MVC
Connected Services
Dependencies
Analyzers
Frameworks
Packages
Microsoft.EntityFrameworkCore (5.0.0)
Microsoft.EntityFrameworkCore.InMemory (5.0.0)

Now that we have EF Core in place.

Creating a view model

Next, we need to create a model that will contain some properties needed for our to-do page. Let's go ahead and create a new class called Todo in the Models folder and then copy the following code:

```
namespace ToDo.MVC.Models {
  public class Todo {
    public int Id { get; set; }
    public string TaskName { get; set; }
    public bool IsComplete { get; set; }
  }
}
```

The preceding code is nothing more than a plain class that houses some properties.

Defining DbContext

EF Core requires DbContext for us to query the datastore. This is typically done by creating a class that inherits from the DbContext class. Now, let's add another class to the Models folder. Name the class TodoDbContext and then copy the following code:

```
using Microsoft.EntityFrameworkCore;
namespace ToDo.MVC.Models {
  public class TodoDbContext : DbContext {
    public TodoDbContext(DbContextOptions<TodoDbContext>
    options)
    : base(options) { }
    public DbSet<Todo> Todos { get; set; }
  }
}
```

The preceding code defines DbContext and a single entity that exposes Model as DbSet. DbContext requires an instance of DbContextOptions. We can then override the OnConfiguring() method to implement our own code, or just pass DbContextOptions to the DbContext base constructor, as we've done in the preceding code.

Seeding test data in memory

Now, since we don't have an actual database for us to pull some data, we need to create a helper function that will initialize some data when the application starts. Let's go ahead and create a new class called TodoDbSeeder in the Models folder, and then copy the following code:

```
public class TodoDbSeeder {
  public static void Seed(IServiceProvider serviceProvider) {
    using var context = new TodoDbContext(
      serviceProvider.GetRequiredService<DbContextOptions<TodoDbContext>>());
    // Look for any todos.
    if (context.Todos.Any()) {
      //if we get here then the data already seeded
      return;
    }
    context.Todos.AddRange(
    new Todo {
      Id = 1,
      TaskName = "Work on book module",
      IsComplete = false
    },
    new Todo {
      Id = 2,
      TaskName = "Create video content",
      IsComplete = false
    }
    );
    context.SaveChanges();
  }
}
```

The preceding code looked for the TodoDbContext service from IServiceCollection and created an instance of it. The method is responsible for generating a couple of test Todo items on application startup. This is done by adding the data to the Todos entity of TodoDbContext.

At this point, we now have DbContext that enables us to access our Todo items and a helper class that will generate some data. What we need to do next is to wire them into the Startup.cs and Program.cs files to get our data populated.

Modifying the Startup class

Let's update the `ConfigureServices()` method of the `Startup` class to the following code:

```
public void ConfigureServices(IServiceCollection services) {
  services.AddDbContext<TodoDbContext>(options => options.
  UseInMemoryDatabase("Todos"));
  services.AddControllersWithViews();
}
```

The preceding code registers TodoDbContext into IServiceCollection and defines an in-memory database called Todos. We need to do this so that we can reference an instance of DbContext in the Controller class or anywhere in our code within the application via dependency injection.

Now, let's move on to the next step by invoking the seeder helper function to generate the test data.

Modifying the Program class

Update the Main() method of the Program.cs file so that it looks similar to the following code:

```
public static void Main(string[] args) {
  var host = CreateHostBuilder(args).Build();
  using (var scope = host.Services.CreateScope()) {
    var services = scope.ServiceProvider;
    TodoDbSeeder.Seed(services);
  }
  host.Run();
}
```

The preceding code creates a scope within the Host lifetime and looks for a service provider that is available from Host. Finally, we invoke the Seed() method of the TodoDbSeeder class and pass the service provider as an argument to the method.

At this point, our test data should be loaded into our memory "database" when the application starts and is ready for use in our application.

Creating the to-do controller

Now, let's create a new Controller class for our Todo page. Go ahead and navigate to the Controllers folder and create a new MVC Controller-Empty class called TodoController. Replace the default-generated code so that it looks similar to the following code:

```
public class TodoController : Controller {
  private readonly TodoDbContext _dbContext;
  public TodoController(TodoDbContext dbContext) {
    _dbContext = dbContext;
  }
  [HttpGet]
  public IActionResult Index() {
    var todos = _dbContext.Todos.ToList();
    return View(todos);
  }
}
```

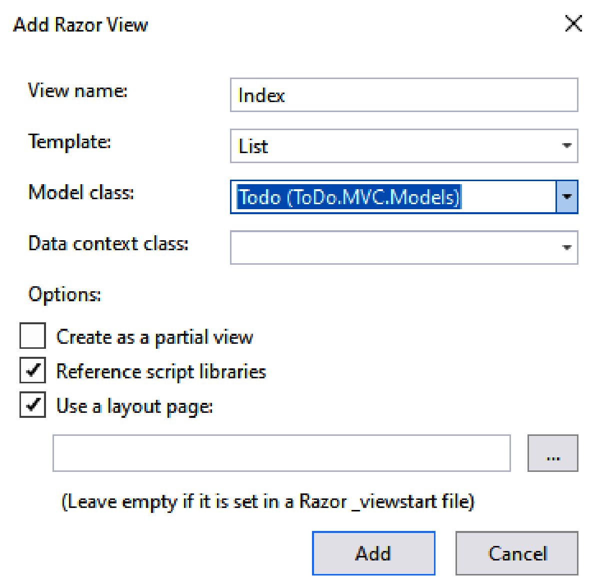
The preceding code first defines a private and read-only field of TodoDbContext. The next line of code defines the constructor class and uses the constructor injection approach to initialize the dependency object. In this case, any methods within the TodoController class will be able to access the instance of TodoDbContext and can invoke all its available methods and properties. For more information about dependency injection, review Module 3, Dependency Injection.

The Index() method is responsible for returning all Todo items from our in-memory datastore to the view. You can see that the method has been decorated with the [HttpGet] attribute, which signifies that the method can only be invoked in an HTTP GET request.

Now, that we have TodoController configured, let's move on to the next step and create the view for displaying all items on the page.

Creating a view

Before creating a view, make sure to build your application first to verify any compilation errors. After a successful build, right-click on the Index() method and then select **Add View…**. In the window dialog, select **Razor View** and it should bring up the dialog shown in figure below.


Add Razor View

View name: Index

Template: List

Model class: Todo (ToDo.MVC.Models)

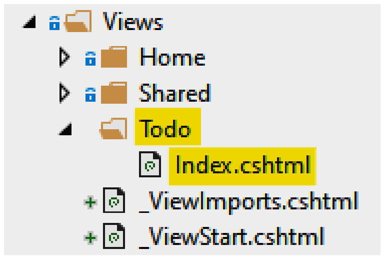
Data context class:

Options:

- [ ] Create as a partial view
- [x] Reference script libraries
- [x] Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

In the preceding dialog, select **List** for Template and select **Todo (ToDo.MVC.Models)** for **Model class**. Finally, click **Add** to generate the views (as shown in Figure below).



In the preceding screenshot, notice that the scaffolding engine automatically creates the views in a way that conforms to the MVC pattern. In this case, the Index.cshtml file was created under the Todo folder.

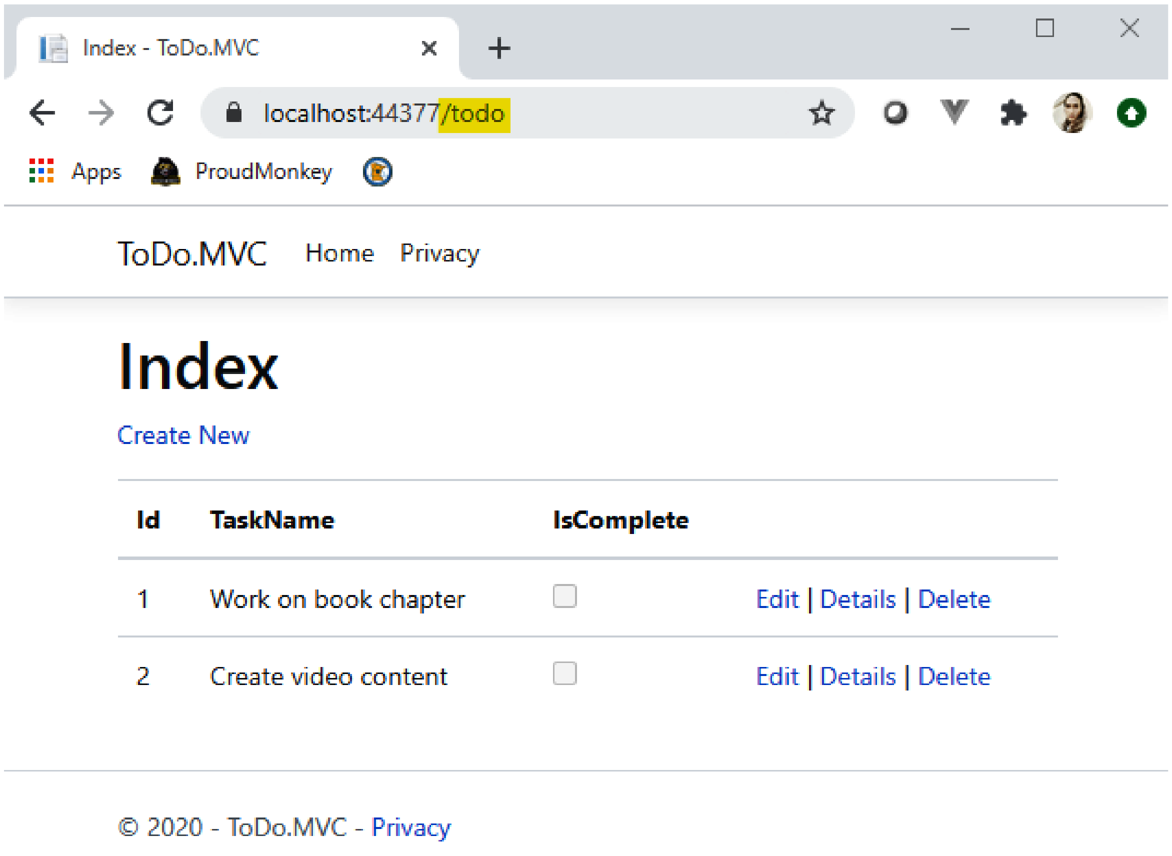
Note

You are free to manually add view files if you'd like. However, using the scaffolding template is much more convenient to generate simple views that match your controller action methods.

Now that we have wired our model, controllers, and views together, let's run the application to see the result.

Running the to-do app

Press the **Ctrl + F5** keys to launch the application in the browser, and then append /todo to the URL. You should be redirected to the to-do page and be presented with the output shown in figure below:



Index

Create New

Id	TaskName	IsComplete	
1	Work on book chapter	☐	Edit | Details | Delete
2	Create video content	☐	Edit | Details | Delete

© 2020 - ToDo.MVC - Privacy

Notice in the preceding screenshot that the test data that we configured earlier has been displayed and the scaffolding template automatically constructs the HTML markup based on ViewModel. This is very convenient and definitely saves you some development time when creating simple pages in your application.

To know how the MVC routing works and how it was configured, just navigate to the Startup class. You should find the following code within the Configure() method:

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

The preceding code configures a default routing pattern for your application using the UseEndpoints() middleware. The default pattern sets a value of Home as the default controller, Index as the default Action value, and id as the optional parameter holder for any routes. In other words, the /home/index path is the default route when the application starts. The MVC pattern follows this routing convention to route URL paths into Controller actions. So, if you want to configure custom routing rules for your application, then this is the middleware that you should look at. For more information about ASP.NET Core routing, see https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing.

At this point, we can confirm that our to-do page is up and running with test data. Now, let's take a look at how to extend the application by implementing some basic functionalities, such as adding, editing, and deleting items.

Implementing add item functionality

Let's modify our TodoController class and add the following code snippet for the add new item functionality:

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

[HttpPost]
public IActionResult Create(Todo todo) {
  var todoId = _dbContext.Todos.Select(x => x.Id).Max() + 1;
  todo.Id = todoId;
  _dbContext.Todos.Add(todo);
  _dbContext.SaveChanges();
  return RedirectToAction(“Index”);
}
```

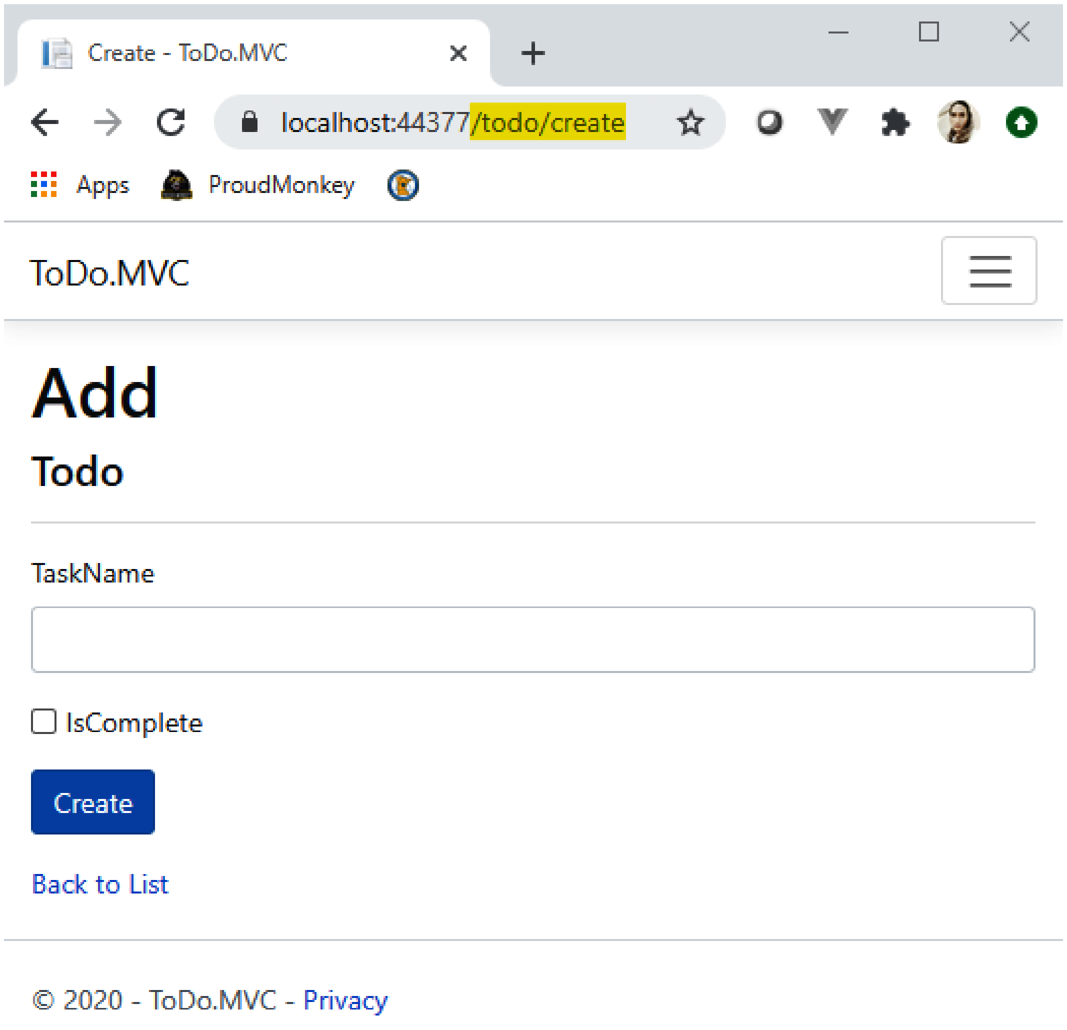
As you will notice in the preceding code, there are two methods with the same name. The first Create() method is responsible for returning the view when a user requests the page. We will create this view in the next step. The second Create() method is an overload method that accepts a Todo view model as an argument, which is responsible for creating a new entry in our in-memory database. You can see that this method has been decorated with the [HttpPost] attribute, which signifies that the method can be invoked only for POST requests. Keep in mind that we are generating an ID manually by incrementing the existing maximum ID from our datastore. In real applications where you use a real database, you may not need to do this as you can let the database auto-generate the ID for you.

Now, let's create the corresponding view of the Create() method. To create a new view, just follow the same steps as we did for the Index() method, but this time select Create as the scaffolding template. This process should generate a Razor file called Create .cshtml in the View/Todo folder.

If you look at the generated view, the Id property of the Todo view model has been generated as well. This is normal as the scaffolding template will generate a Razor view based on the view model/model provided. We don't want the Id property to be included in the view as we are generating it in the code. So, remove the following HTML markup from the view:

```
<div class="form-group">
  <label asp-for="Id" class="control-label"></label>
  <input asp-for="Id" class="form-control" />
  <span asp-validation-for=”Id” class="text-danger"></span>
</div>
```

Now, run the application again and navigate to /todo/create, and you should be presented with a page that looks similar to figure below:



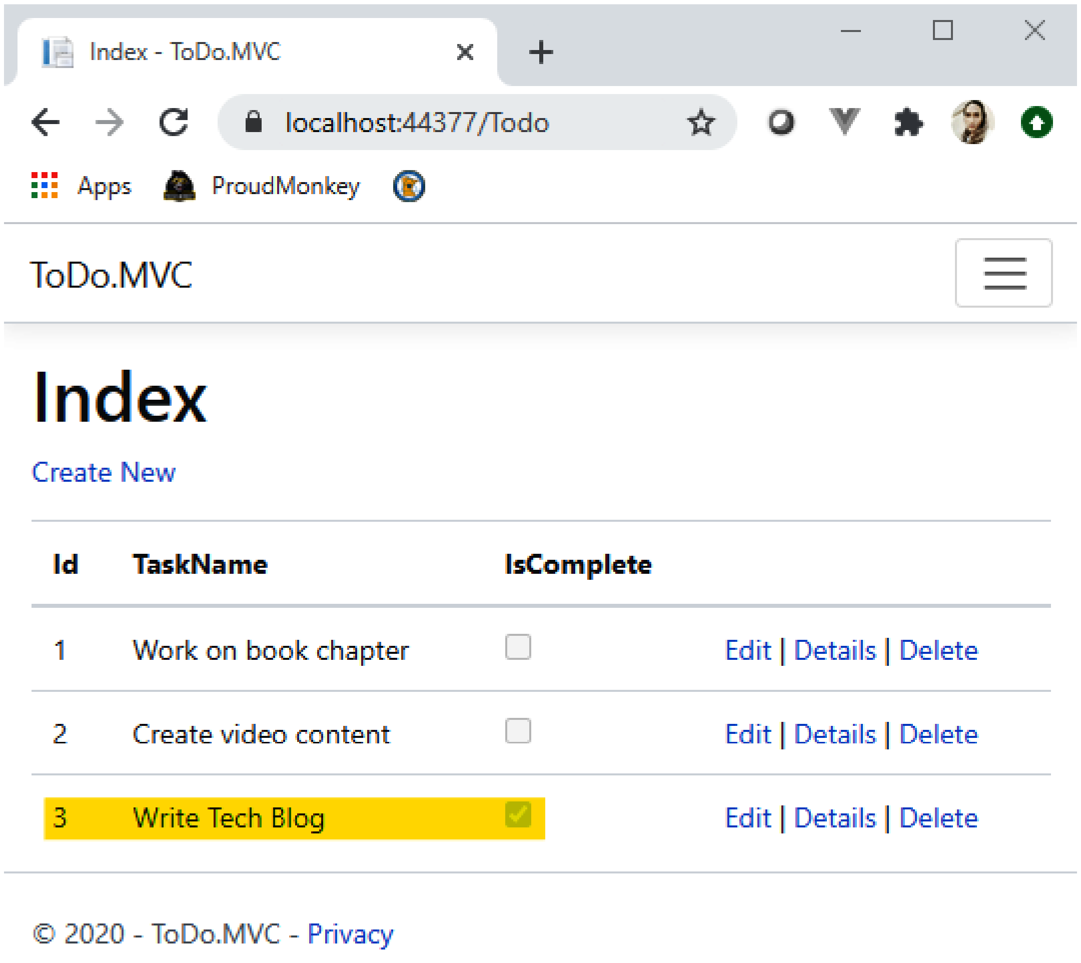
Add

Todo

TaskName

IsComplete

Now, type the value Write Tech Blog in the TaskName textbox and tick the IsComplete checkbox. Clicking the Create button should add a new entry to our in-memory database and redirect you to the Index page, as shown in figure below:



Index

Create New

Id	TaskName	IsComplete	
1	Work on book chapter	☐	Edit | Details | Delete
2	Create video content	☐	Edit | Details | Delete
3	Write Tech Blog	✓	Edit | Details | Delete

© 2020 - ToDo.MVC - Privacy

Sweet! To add more items, you can click the Create New link at the top of the list and you should be redirected back to the create view. Keep in mind though that we are not implementing any input validation here for simplicity's sake. In real applications, you should consider implementing model validations using either data annotation or FluentValidation. You can read more about these by referring to the following links:

- https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation
- https://docs.fluentvalidation.net/en/latest/aspnet.html

Now, let's move on to the next step.

Implementing edit functionality

Switch back to the TodoController class and add the following code snippet for the edit functionality:

```
[HttpGet]
public IActionResult Edit(int id) {
  var todo = _dbContext.Todos.Find(id);
  return View(todo);
}

[HttpPost]
public IActionResult Edit(Todo todo) {
  _dbContext.Todos.Update(todo);
  _dbContext.SaveChanges();
  return RedirectToAction("Index");
}
```

The preceding code also has just two action methods. The first Edit() method is responsible for populating the fields in the view based on the ID being passed to the route. The second Edit() method will be invoked during an HTTP POST request, which handles the actual update to the datastore.

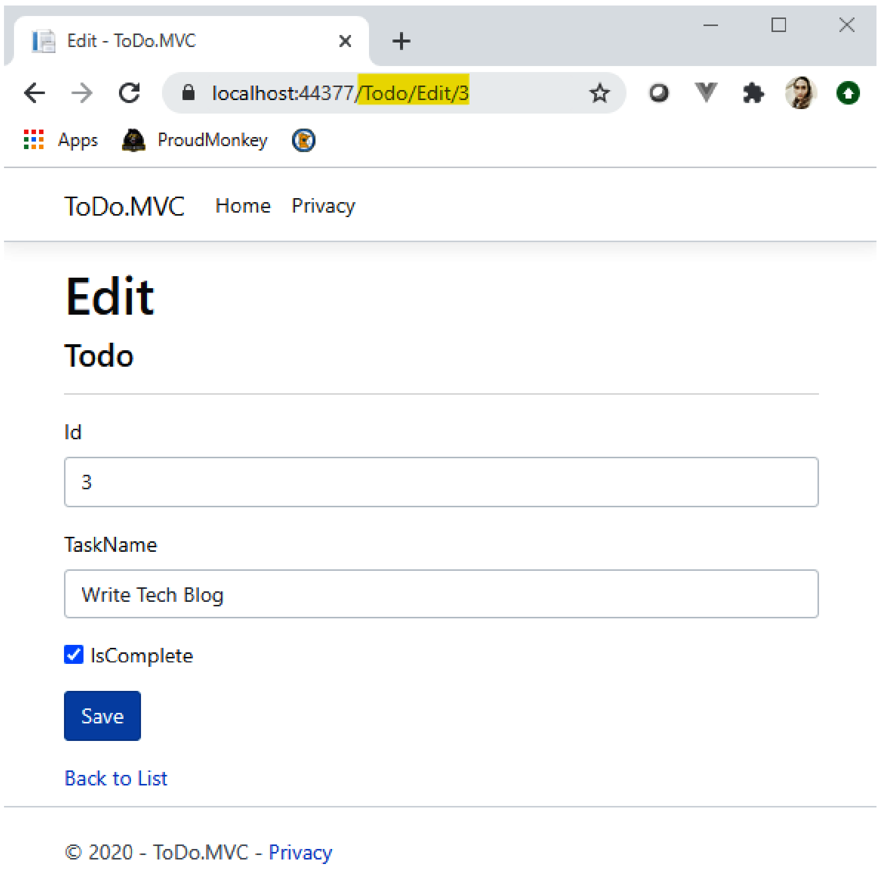
To create the corresponding view for the Edit action, just follow the same steps as we did in the previous functionality, but this time, select Edit as the scaffolding template. This process should generate a Razor file called Edit.cshtml in the View/Todo folder.

The next step is to update our Index view to map the routes for edit and delete actions. Go ahead and update the Action link to the following:

```
@Html.ActionLink("Edit", "Edit", new { id = item.Id }) |
@Html.ActionLink("Delete", "Delete", new { id = item.Id })
```

The preceding code defines a couple of ActionLink HTML helpers for navigating between views with parameters. The changes we made in the preceding code are passing the ID as the parameter to each route and removing the details link, as we won't be covering that here. Anyway, implementing the details page should be pretty straightforward. You can also view the GitHub code repository of this module to see how it was implemented.

Now, when you run the application, you should be able to navigate from the to-do Index page to the **Edit** page by clicking the **Edit** link. Figure below shows you a sample screenshot of the Edit page.



Edit

Todo

Id

3

TaskName

Write Tech Blog

☑ IsComplete

© 2020 - ToDo.MVC - Privacy

In the preceding screenshot, notice that the ID is now included in the route and the page is automatically being populated with the corresponding data. Now, let's move on to the next step.

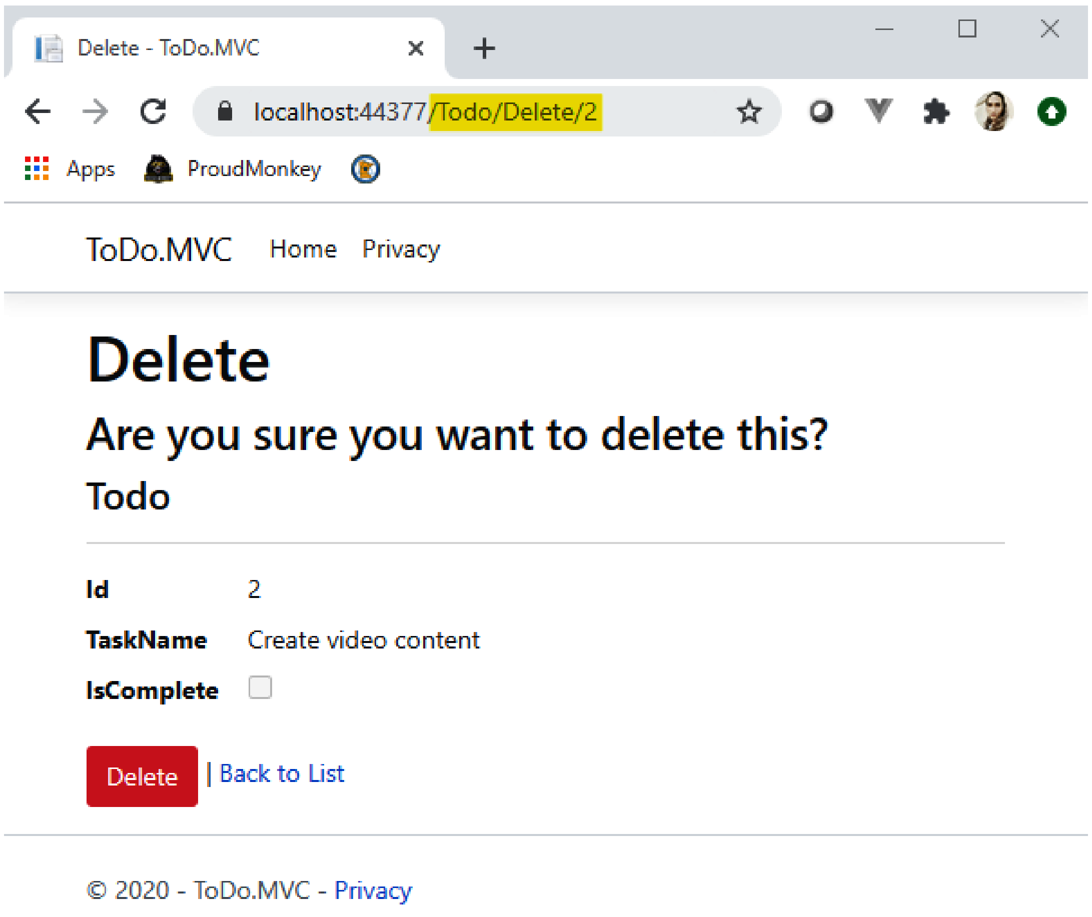
Implementing the delete functionality

Switch back to the TodoController class and add the following code snippet for the delete functionality:

```
public IActionResult Delete(int? id) {
  var todo = _dbContext.Todos.Find(id);
  if (todo == null) {
    return NotFound();
  }
  return View(todo);
}

[HttpPost]
public IActionResult Delete(int id) {
  var todo = _dbContext.Todos.Find(id);
  _dbContext.Todos.Remove(todo);
  _dbContext.SaveChanges();
  return RedirectToAction("Index");
}
```

The first Delete() method in the preceding code is responsible for populating the page with the corresponding data based on the ID. If the ID does not exist in our in-memory datastore, then we simply return a NotFound() result. The second Delete() method will be triggered when clicking the Delete button. This method executes the deletion of the item from the datastore. Figure 4.22 shows you a sample screenshot of the Delete page.


Delete - ToDo.MVC

localhost:44377/Todo/Delete/2

ToDo.MVC Home Privacy

Delete

Are you sure you want to delete this?

Todo

Id	2
TaskName	Create video content
IsComplete	☐

Delete | Back to List

© 2020 - ToDo.MVC - Privacy

At this point, you should have a better understanding of how MVC works and how we can easily implement **Create**, **Read**, **Update**, and **Delete** (CRUD) operations on a page. There are a lot of things that we can do to improve the application, so take a moment to add the missing features as an extra exercise. You could try integrating model validations, logging, or any features that you want to see in the application. You can also refer to the following project template to help you get up to speed on using MVC in concert with other technologies to build web applications:

https://github.com/proudmonkey/MvcBoilerPlate