Развој на серверски WEB апликации

Razor pages

What Razor Pages are and the benefit they provide

- **Razor Pages** is a server-side, **page-centric programming model** for building rich web UI with ASP.NET Core.
  - Each page represents a separate unit of functionality and is responsible for handling its own requests and responses
- Razor Pages makes it easy to get started building dynamic web applications when all you need is to define UI logic using a combination of HTML, CSS, and C#.
- Razor Pages provides a productivity advantage over the more complex **M**odel-**V**iew-**C**ontroller (MVC) application model.
- Razor Pages encourages organization of files by feature (rather than organizing files by the type -> MVC), therefore easing maintenance of your application.
- Razor Pages can be broadly described as an HTML file where you can work with markup, but you also have the advantage of adding server-side C# code by using Razor syntax.
- Razor pages have the extension *.cshtml*.

What Razor Pages are and the benefit they provide

- Razor syntax is a combination of HTML and C# where the C# code defines the dynamic rendering logic for the page.
- In a webpage that uses the Razor syntax, there can be two kinds of content: client content and server code.
- **Client content**: Contains HTML markup (elements), style information such as CSS, maybe some client script such as JavaScript, and plain text.
- **Server code**: Razor syntax lets you add server code to your client content.
  - If there is server code in the page, the server runs that code first, before it sends the page to the browser.
  - By running on the server, the code can perform more complex tasks than using client content alone, like securely accessing server-based databases.
  - Most importantly, server code can dynamically create client content — it can generate HTML markup or other content and send it to the browser along with any static HTML that the page might contain.
  - From the browser's perspective, client content that's generated by your server code is no different than any other client content.

Separation of concerns in the *PageModel*

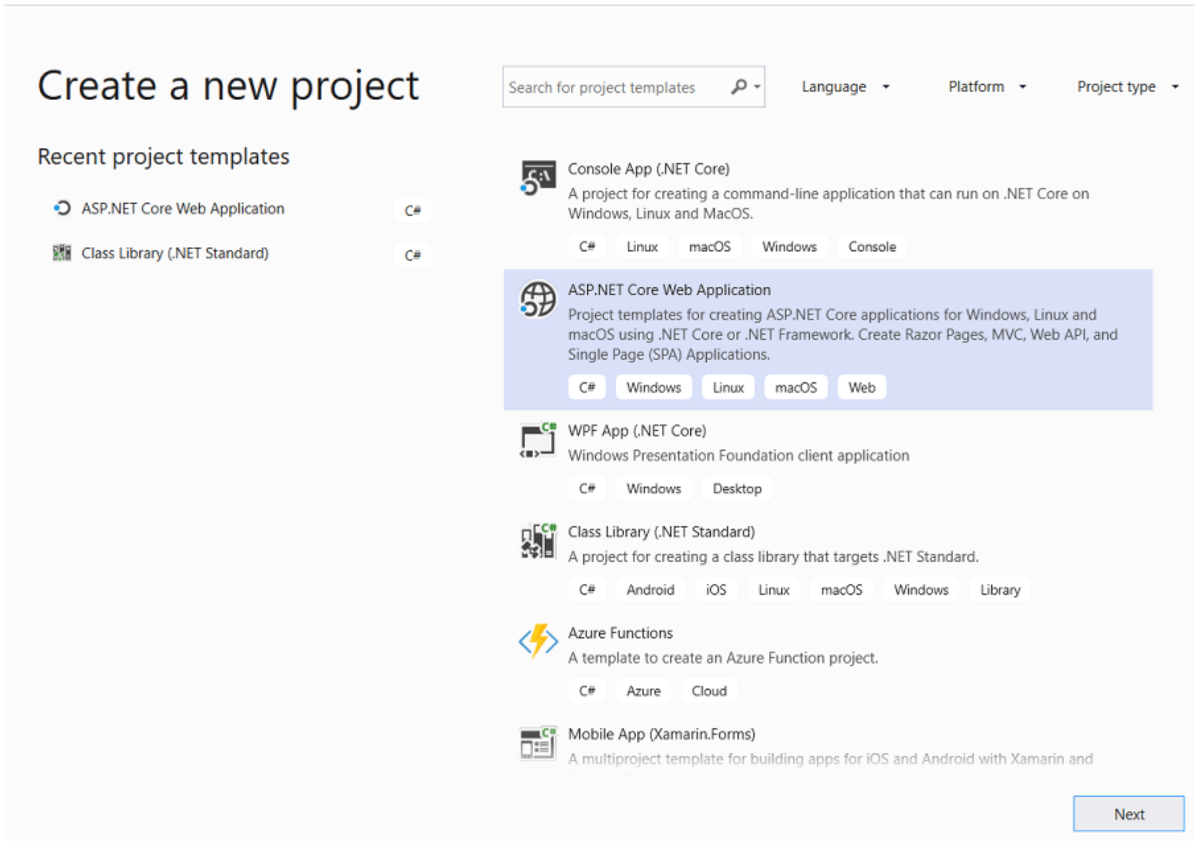
- Razor Pages enforces separation of concerns for page-related data properties and logic operations in a C# *PageModel* class file.
    - A model object typically defines data properties and encapsulates any logic or operations related to those data properties.
- A Razor Pages *PageModel* more specifically encapsulates the data properties and logic operations scoped just to its Razor page.
- It defines page handlers for requests sent to the page and for the data used to render the page.

When to use Razor Pages?

- Use Razor Pages in your ASP.NET Core application when:
  - You want to generate dynamic UI for a browser from your ASP.NET Core application.
  - You prefer a **page-focused approach** to developing web applications, where the page markup and *PageModel* are in close proximity.
  - You want your page-focused ASP.NET Core application to use shared common HTML elements across your site and reusable partial views.
- Razor Pages allow you to keep your ASP.NET Core pages organized in a simpler way:
  - All view (page) specific logic and page properties defined in the Razor page can be kept together in their own namespace and directory.
  - Groups of related pages can be kept in their own namespace and directory.

Create a Razor Pages web app

- From the Visual Studio **File** menu, select **New** > **Project**.
- Create a new ASP.NET Core Web Application and select **Next**.


Create a new project
Search for project templates
Language
Platform
Project type
Recent project templates
ASP.NET Core Web Application C#
Class Library (.NET Standard) C#
Next

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
C# Linux macOS Windows Console

ASP.NET Core Web Application
Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.
C# Windows Linux macOS Web

WPF App (.NET Core)
Windows Presentation Foundation client application
C# Windows Desktop

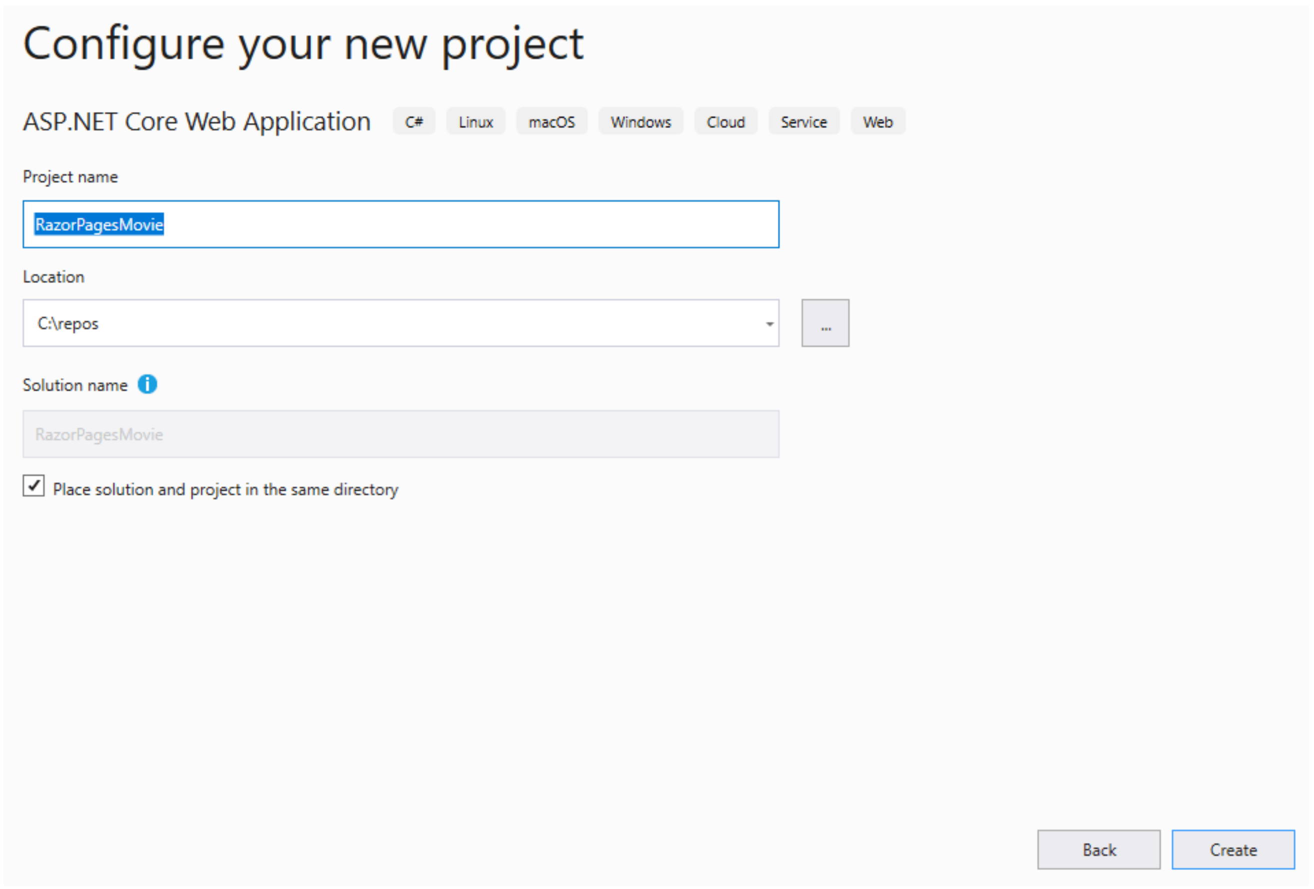
Class Library (.NET Standard)
A project for creating a class library that targets .NET Standard.
C# Android iOS Linux macOS Windows Library

Azure Functions
A template to create an Azure Function project.
C# Azure Cloud

Mobile App (Xamarin.Forms)
A multiproject template for building apps for iOS and Android with Xamarin and

Create a Razor Pages web app

- Name the project
- Set location
- Check or uncheck the option "Place solution and project in the same directory"



Configure your new project

ASP.NET Core Web Application

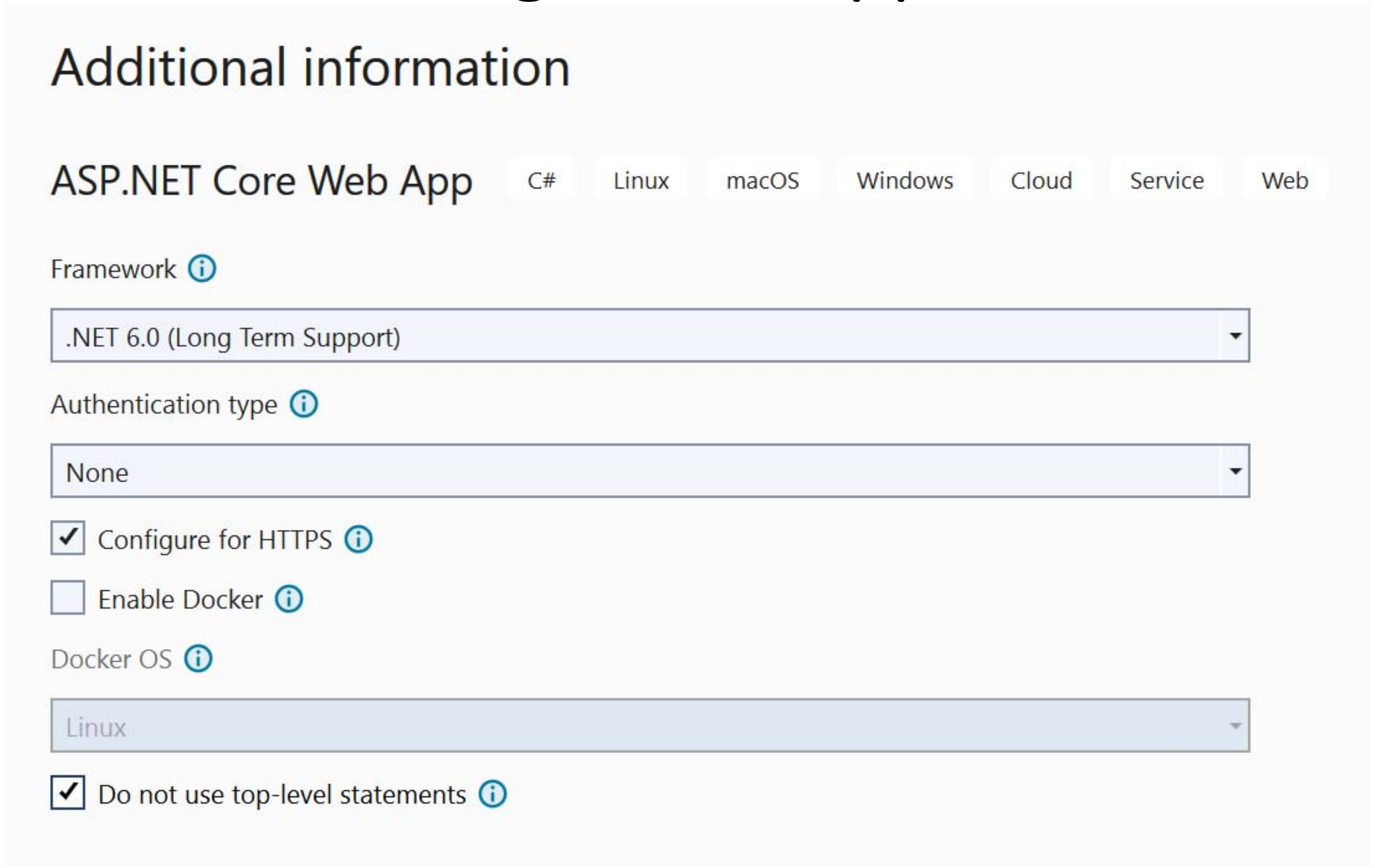
Project name: RazorPagesMovie

Location: C:\repos

Solution name: RazorPagesMovie

[✓] Place solution and project in the same directory

Create a Razor Pages web app



Additional information

ASP.NET Core Web App

Framework: .NET 6.0 (Long Term Support)

Authentication type: None

[✓] Configure for HTTPS

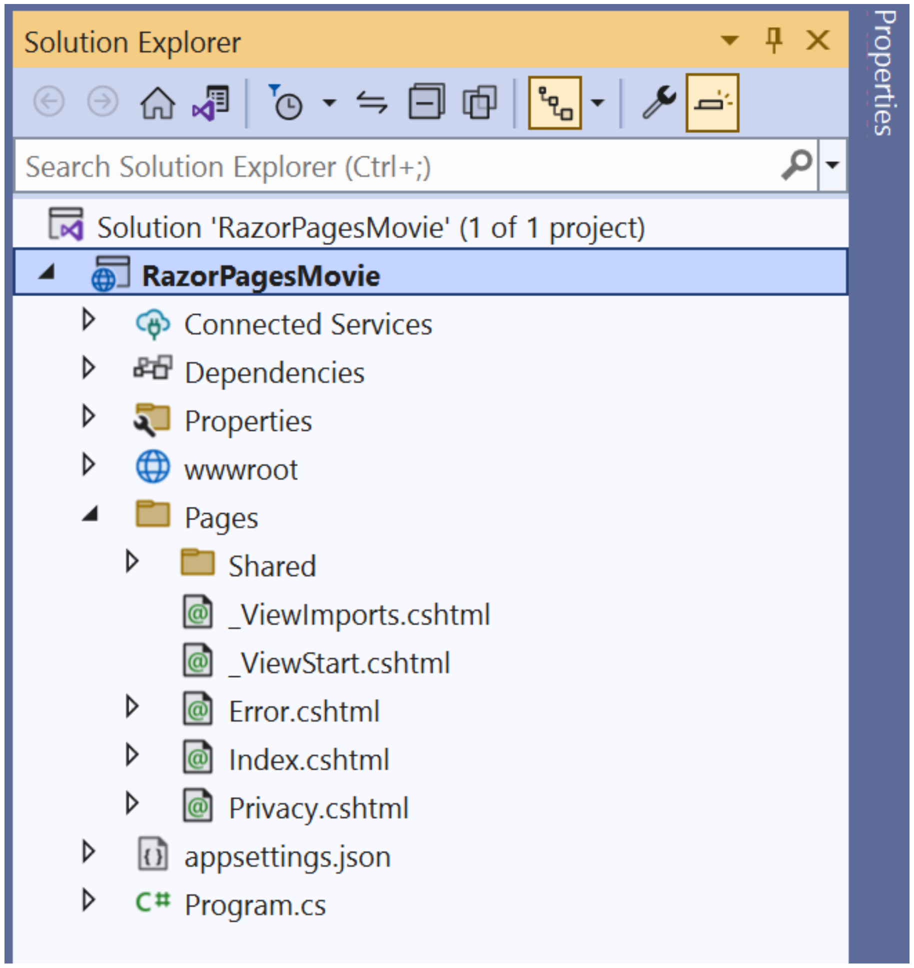
[ ] Enable Docker

Docker OS: Linux

[✓] Do not use top-level statements

Create a Razor Pages web app

- The following starter project is created:



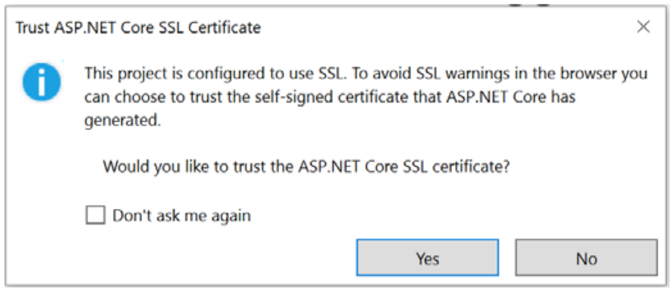
Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'RazorPagesMovie' (1 of 1 project)
◢ RazorPagesMovie
    ▷ Connected Services
    ▷ Dependencies
    ▷ Properties
    ▷ wwwroot
    ◢ Pages
        ▷ Shared
          _ViewImports.cshtml
          _ViewStart.cshtml
        ▷ Error.cshtml
        ▷ Index.cshtml
        ▷ Privacy.cshtml
    ▷ appsettings.json
    ▷ Program.cs
```

Run the app

- Press Ctrl+F5 to run without the debugger.
- Visual Studio displays the following dialog:



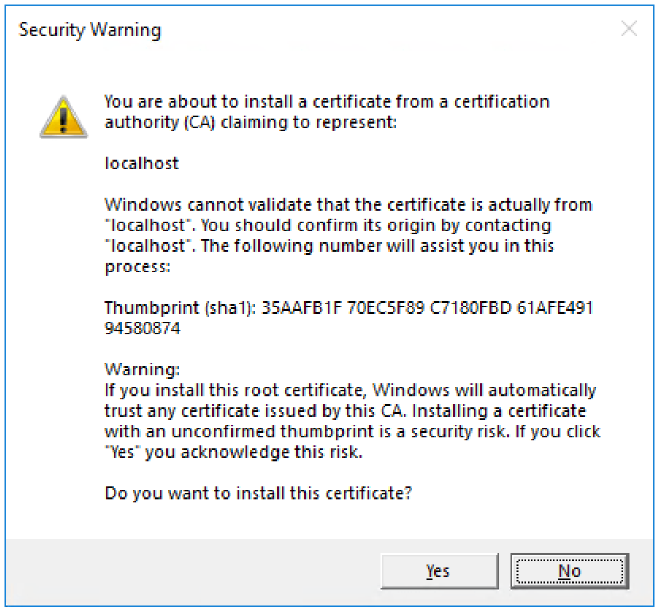
Trust ASP.NET Core SSL Certificate

This project is configured to use SSL. To avoid SSL warnings in the browser you can choose to trust the self-signed certificate that ASP.NET Core has generated.

Would you like to trust the ASP.NET Core SSL certificate?

☐ Don't ask me again

- Select **Yes** if you trust the SSL certificate.
- The following dialog is displayed:



Security Warning

You are about to install a certificate from a certification authority (CA) claiming to represent:

localhost

Windows cannot validate that the certificate is actually from "localhost". You should confirm its origin by contacting "localhost". The following number will assist you in this process:

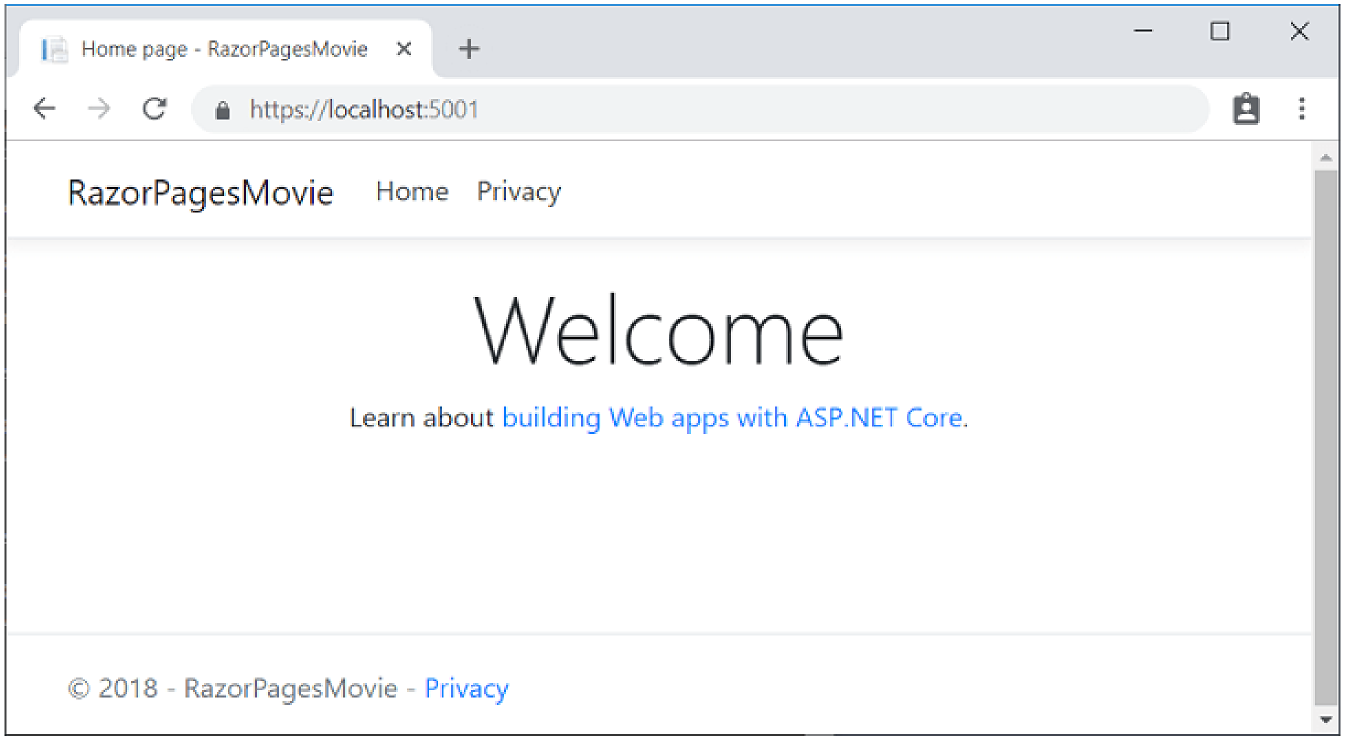
Thumbprint (sha1): 35AAFB1F 70EC5F89 C7180FBD 61AFE491 94580874

Warning:
If you install this root certificate, Windows will automatically trust any certificate issued by this CA. Installing a certificate with an unconfirmed thumbprint is a security risk. If you click "Yes" you acknowledge this risk.

Do you want to install this certificate?

- Select Yes if you agree to trust the development certificate.
- Visual Studio runs the app. The address bar shows localhost:port#
    - localhost is the standard hostname for the local computer.
    - Localhost only serves web requests from the local computer. When Visual Studio creates a web project, a random port is used for the web server.

The result



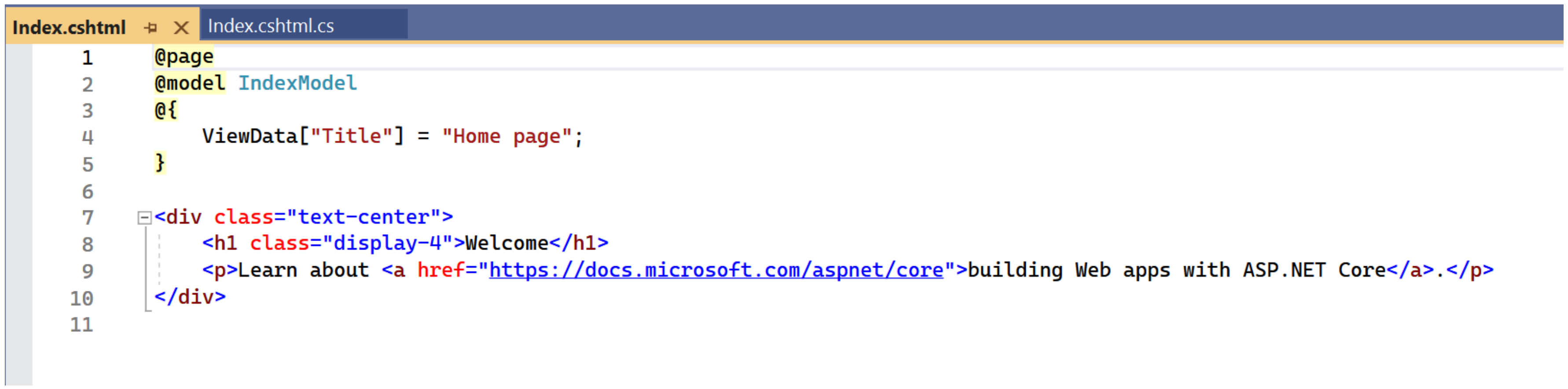
Welcome

Learn about building Web apps with ASP.NET Core.

© 2018 - RazorPagesMovie - Privacy

Examine the project files

- **Pages folder**
- Contains Razor pages and supporting files.
- **Each Razor page is a pair of files**:
  - A *.cshtml* file that contains HTML markup with C# code using Razor syntax.
  - A *.cshtml.cs* file that contains C# code that handles page events.
- Index.cshtml: What makes it different is the **@page directive** - which means that it handles requests directly, without going through a controller. @page must be the first Razor directive on a page.



```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>
</div>
```

Page model

- By convention, the PageModel class file has the same name as the Razor Page file with .cs appended.

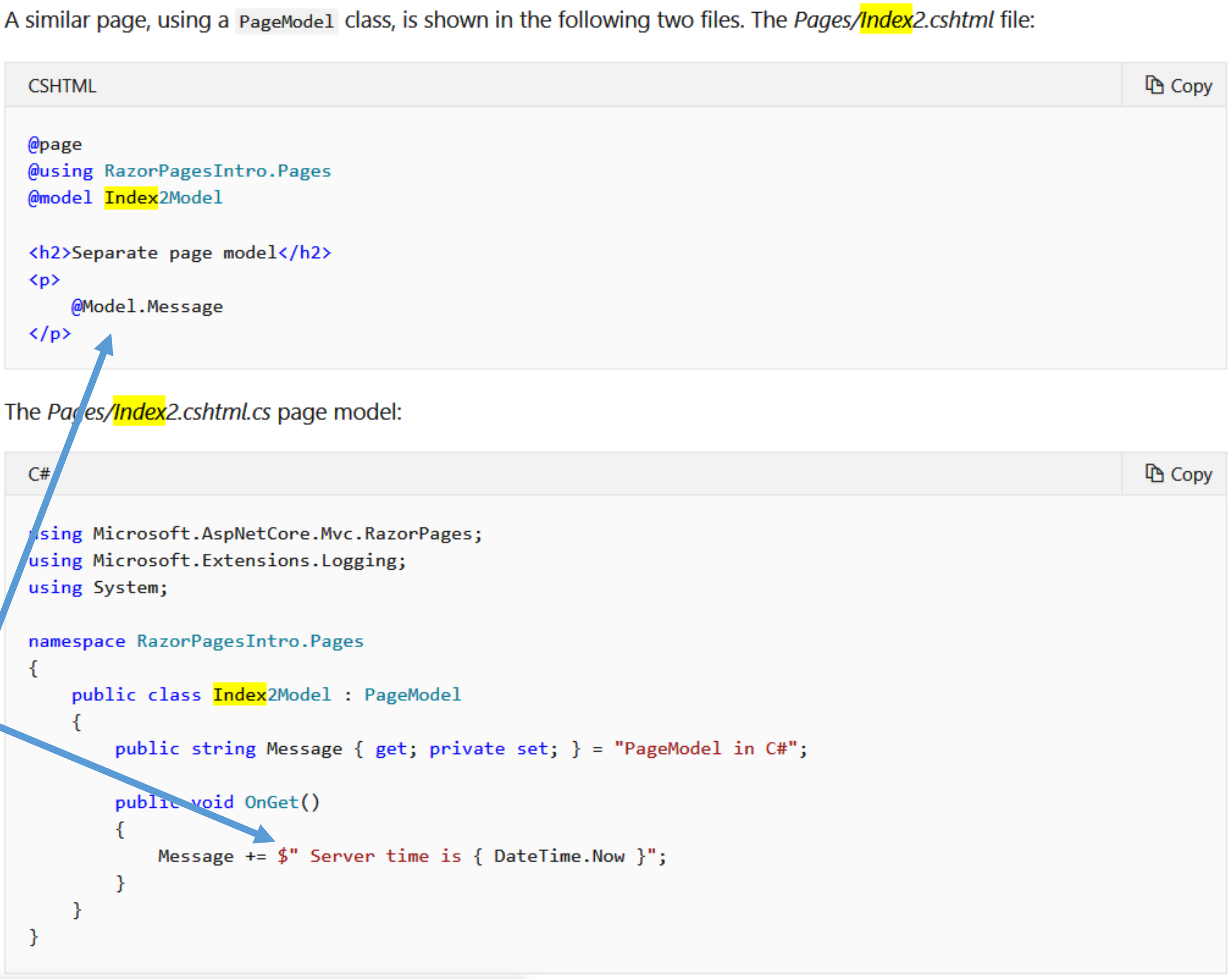
- @model IndexModel directive tells Razor to expect an instance of the IndexModel class to be passed to the view when it is rendered.
  - By using the @model directive, you can access the properties and methods of the IndexModel class within your Razor view to generate the content that will be displayed on the home page.

- You can use the @Model keyword to reference properties and methods of the model and use them to generate HTML content.

Page model

- For example, the Razor Page is Pages/Index2.cshtml
- The file containing the PageModel class is named Pages/Index2.cshtml.cs

- An interpolated string
  - contains placeholders for expressions that will be evaluated at runtime and inserted into the string.



A similar page, using a `PageModel` class, is shown in the following two files. The *Pages/Index2.cshtml* file:

CSHTML

```
@page
@using RazorPagesIntro.Pages
@model Index2Model

<h2>Separate page model</h2>
<p>
    @Model.Message
</p>
```

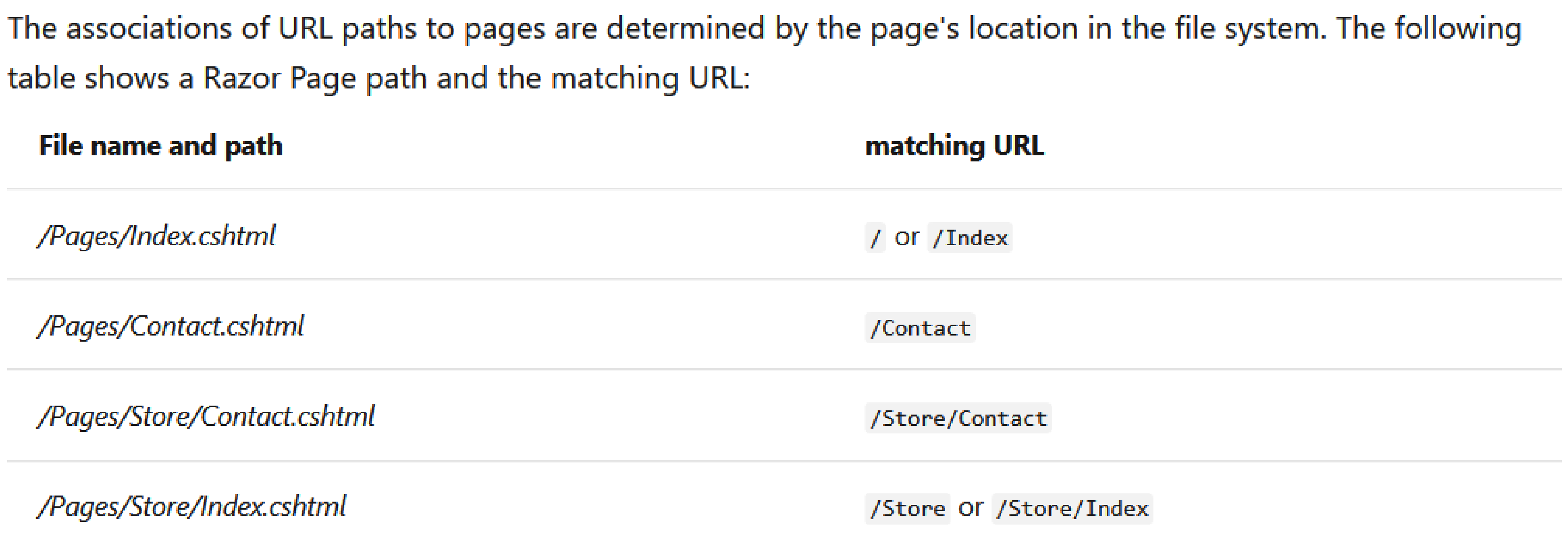
The *Pages/Index2.cshtml.cs* page model:

C#

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;
using System;

namespace RazorPagesIntro.Pages
{
    public class Index2Model : PageModel
    {
        public string Message { get; private set; } = "PageModel in C#";

        public void OnGet()
        {
            Message += $" Server time is { DateTime.Now }";
        }
    }
}
```

URL paths

- The runtime looks for Razor Pages files in the Pages folder by default.
- Index is the default page when a URL doesn't include a page.



The associations of URL paths to pages are determined by the page's location in the file system. The following table shows a Razor Page path and the matching URL:

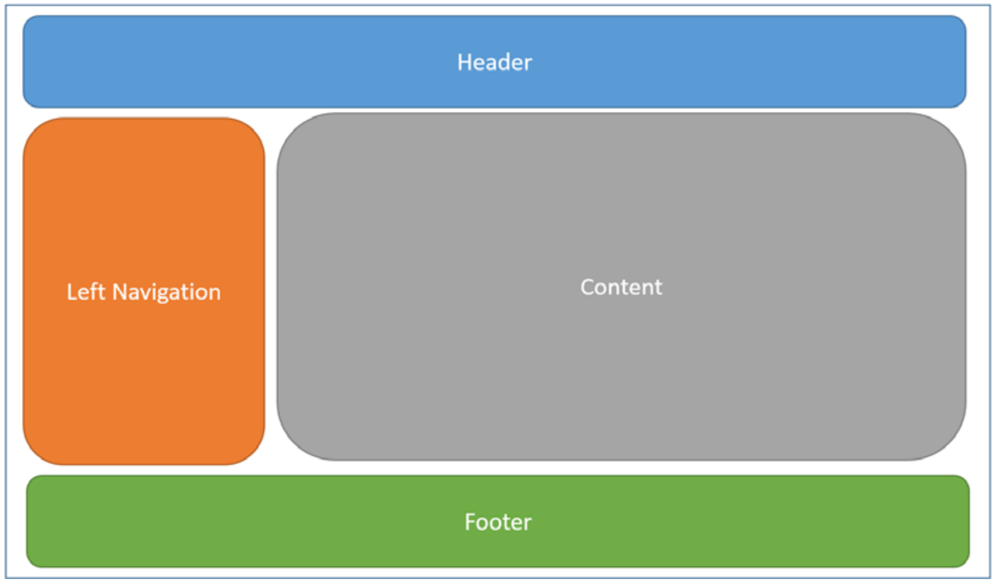
File name and path	matching URL
/Pages/Index.cshtml	`/` or `/Index`
/Pages/Contact.cshtml	`/Contact`
/Pages/Store/Contact.cshtml	`/Store/Contact`
/Pages/Store/Index.cshtml	`/Store` or `/Store/Index`

Examine the project files

- **Pages folder**
- Contains Razor pages and supporting files.
- Supporting files have names that begin with an underscore.
    - For example, the *_Layout.cshtml* file configures UI elements common to all pages (Shared).
    - This file sets up the navigation menu at the top of the page and the copyright notice at the bottom of the page.

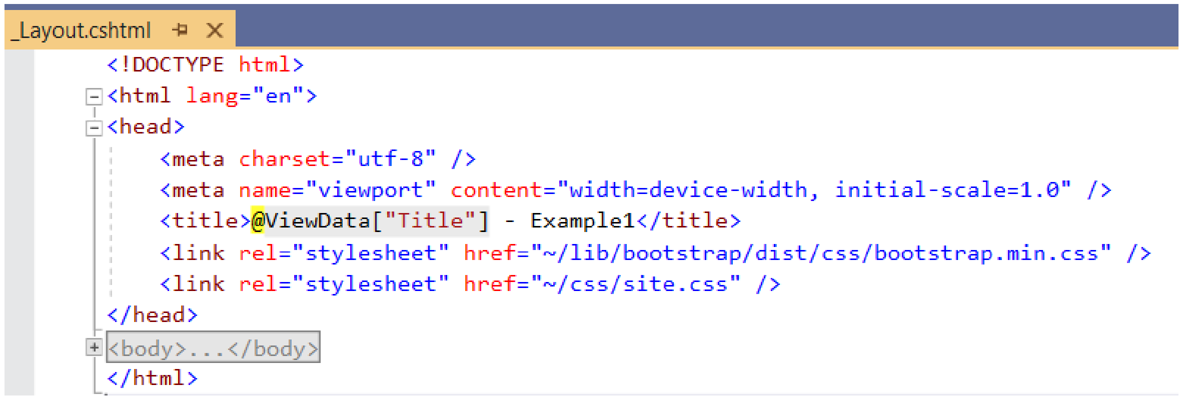
What is a Layout?

- Most web apps have a common layout that provides the user with a consistent experience as they navigate from page to page.
- The layout typically includes common user interface elements such as the app header, navigation or menu elements, and footer.
- All of these shared elements may be defined in a layout file, which can then be referenced by any view used within the app. **Layouts reduce duplicate code in views.**
- Apps don't require a layout. Apps can define more than one layout, with different views specifying different layouts.

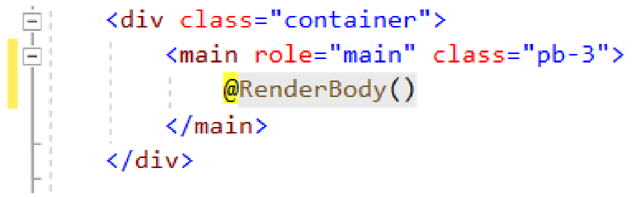


Specifying a Layout

- The layout specified can use a full path (for example, /Pages/Shared/_Layout.cshtml) or a partial name (example: _Layout).
- When a partial name is provided, the Razor view engine searches for the layout file using its standard discovery process.
    - The folder where the handler method (or controller) exists is searched first, followed by the Shared folder.
- By default, every layout must call RenderBody() method



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Example1</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>...</body>
</html>
```

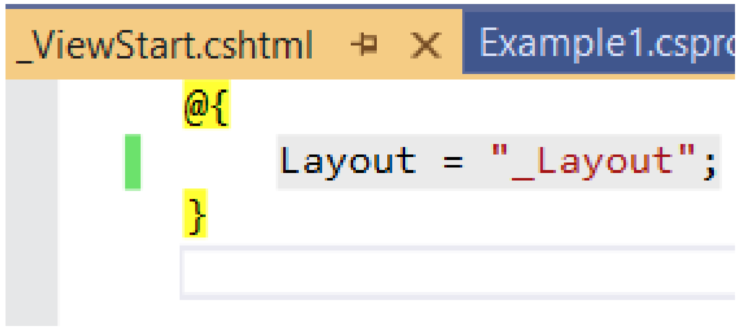


```
<div class="container">
    <main role="main" class="pb-3">
        @RenderBody()
    </main>
</div>
```

The position of the RenderBody method in the layout page determines where the information from the content page will be included.

Running Code Before Each View

- Code that needs to run before each view or page should be placed in the *_ViewStart.cshtml* file.
- By convention, the *_ViewStart.cshtml* file is located in the *Pages* (or *Views*) folder.
- The statements listed in *_ViewStart.cshtml* are run before every full view.
- The example specifies that all views will use the _Layout.cshtml layout.



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

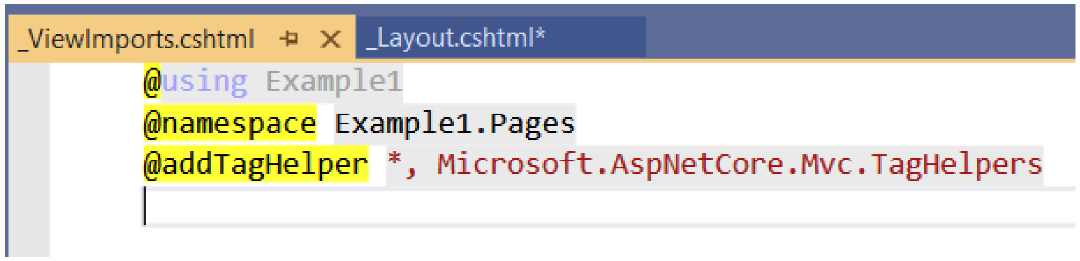
Importing Shared Directives

- Views and pages can use Razor directives to import namespaces and use dependency injection.
- Directives shared by many views may be specified in a common _ViewImports.cshtml file.
- The _*ViewImports.cshtml* file for an ASP.NET Core MVC app is typically placed in the *Pages* (or *Views*) folder.
- Examples:

  @using

  @model

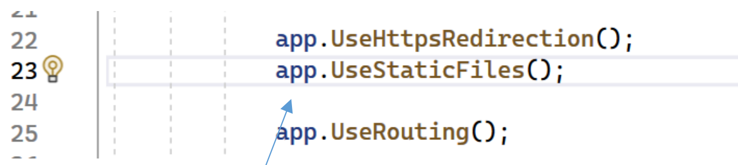
  @namespace



```
@using Example1
@namespace Example1.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Examine the project files
wwwroot folder

- **Contains static files**, such as HTML files, JavaScript files, and CSS files.
- Static files are assets an ASP.NET Core app serves directly to clients.
- Static files are stored within the project's web root directory. The default directory is *{content root}/wwwroot*
- Static files are accessible via a path relative to the web root. For example, the **Web Application** project template contains several folders within the *wwwroot* folder:
- **wwwroot**
    - **css**
    - **images**
    - **js**

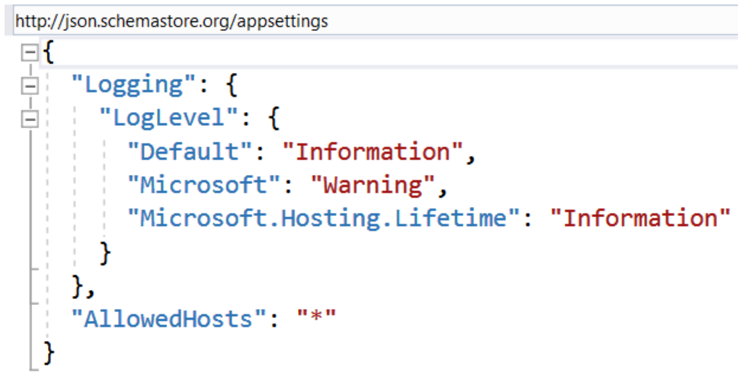


```
app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
```

- The URI format to access a file in the *images* subfolder is *http://<server_address>/images/<image_file_name>*. For example, *http://localhost:9189/images/banner3.svg*.
- Invoke the app.UseStaticFiles() method within Program.cs

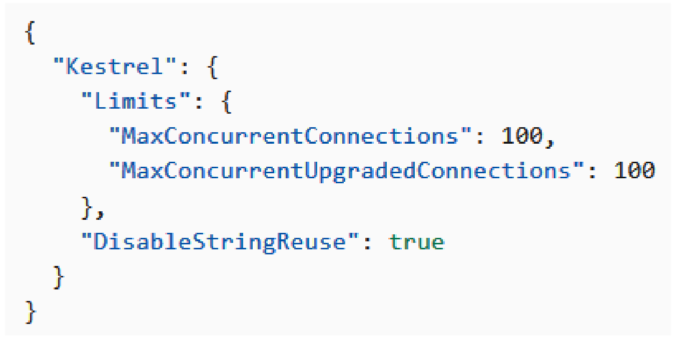
Examine the project files
appSettings.json

- Contains **configuration data**.
- App configuration in ASP.NET Core is based on key-value pairs
- Examples:
  - Logging configuration provided by the Logging section of app settings files
  - Kestrel configuration



http://json.schemastore.org/appsettings

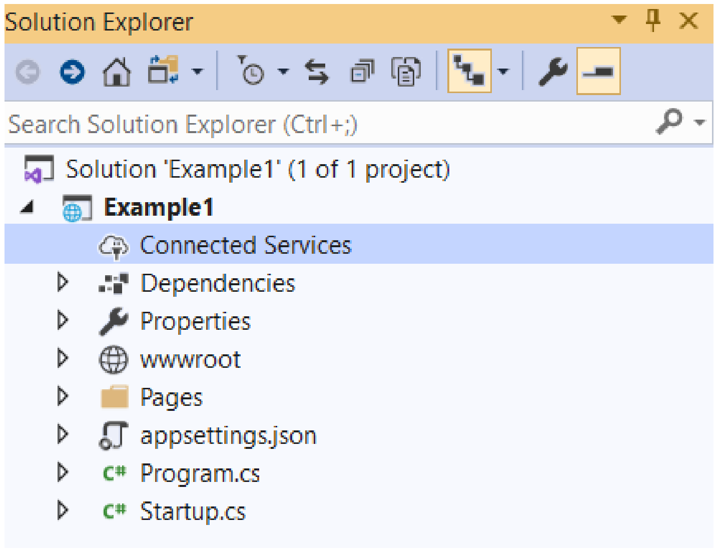
```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```



```
{
  "Kestrel": {
    "Limits": {
      "MaxConcurrentConnections": 100,
      "MaxConcurrentUpgradedConnections": 100
    },
    "DisableStringReuse": true
  }
}
```

Special nodes

- Although the wwwroot and Properties folders exist on disk, you can see that Solution Explorer shows them as special nodes, out of alphabetical order, near the top of your project.
- There are two more special nodes in the project, Dependencies and Connected Services, but they don't have a corresponding folder on disk.
  - Instead, they show a collection of all the dependencies, such as NuGet packages, client-side dependencies, and remote services that the project relies on.
- The Properties folder contains a single file, launchSettings.json, which controls how Visual Studio will run and debug the application.



Program.cs

- A new approach is designed to make it easier and more intuitive to build simple web applications with minimal ceremony (no more Startup.cs)
- *Program.cs* contains the **code that configures app behavior**. It configures services and the app's request handling pipeline.
  - *Service registration*—Any classes that your application depends on for providing functionality—both those used by the framework and those specific to your application—must be registered so that they can be correctly instantiated at runtime.
    - A service is a reusable component that provides app functionality. Services are registered in and consumed across the app via **dependency injection (DI) container**
  - *Middleware* —How your application handles and responds to requests.

Adding and configuring services

- ASP.NET Core uses small, modular components for each distinct feature.
    - This allows individual features to evolve separately, with only a loose coupling to others, and is generally considered good design practice.
    - The downside to this approach is that it places the burden on the consumer of a feature to correctly instantiate it.
- Within your application, these modular components are exposed as one or more *services* that are used by the application.
- DEFINITION: Within the context of ASP.Net Core, ***service*** refers to any class that provides functionality to an application and could be classes exposed by a library or code you've written for your application.

Adding and configuring services

- For example, in an e-commerce app, you might have a TaxCalculator that calculates the tax due on a particular product, taking into account the user’s location in the world.
- Or you might have a ShippingCostService that calculates the cost of shipping to a user’s location.
- A third service, OrderTotalCalculatorService, might use both of these services to work out the total price the user must pay for an order.

Adding and configuring services

- Each service provides a small piece of independent functionality, but you can combine them to create a complete application.
- This is known as the ***single responsibility principle***.
- DEFINITION: The *single responsibility principle* (SRP) states that every class should be responsible for only a single piece of functionality—it should only need to change if that required functionality changes.

Adding and configuring services

- When writing a service, you can declare your **dependencies** and let another class fill those dependencies for you.
- Your service can then focus on the functionality for which it was designed, instead of trying to work out how to build its dependencies.
- This technique is called **dependency injection** or the **inversion of control (IoC)** principle and is a well-recognized *design pattern* that is used extensively.
- DEFINITION: *Design patterns* are solutions to common software design problems.

Adding and configuring services

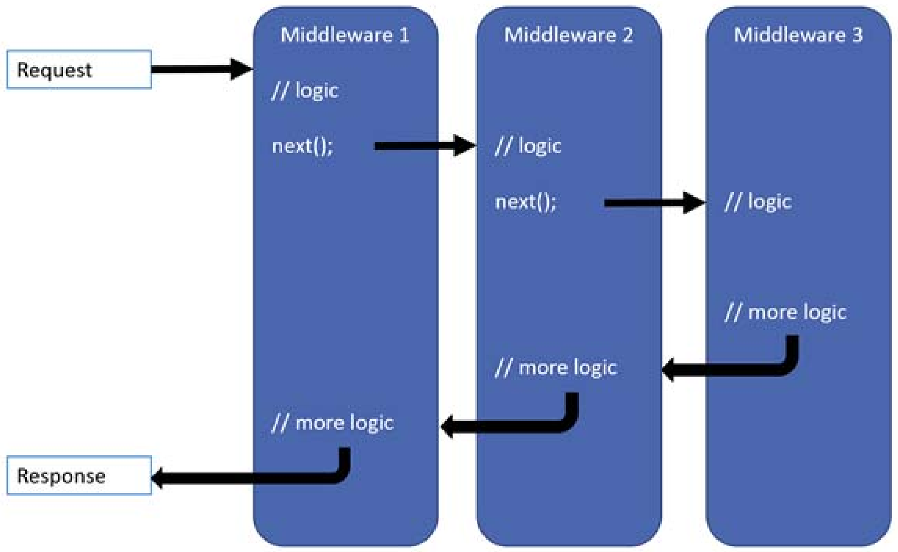
- A complete MVC application only includes a single call to add the necessary services, but the AddMvc() method is an extension method that encapsulates all the code required to set up the MVC services.
  - Behind the scenes, it adds various Razor services for rendering HTML, formatter services, routing services, and many more!
- As well as registering framework-related services, Program.cs is where you'd register any custom services you have in your application, such as the example TaxCalculator discussed previously.
- Services are added to the DI container with WebApplicationBuilder.Services
- Services are typically resolved from DI using constructor injection. The DI framework provides an instance of this service at runtime.

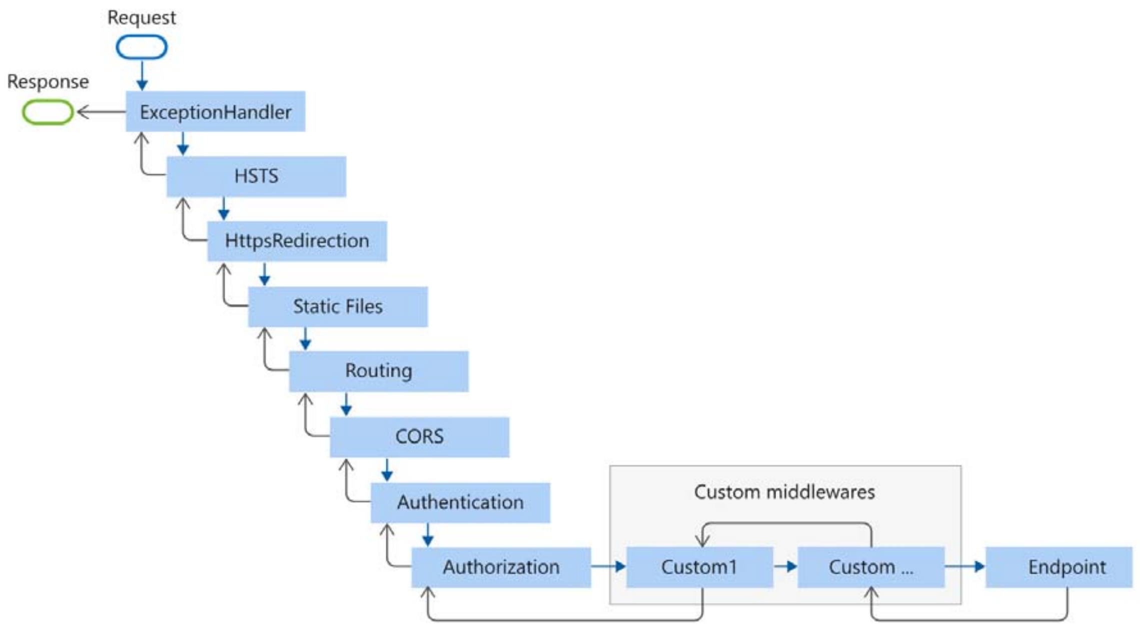
Defining how requests are handled with middleware

- The middleware consists of small components that execute in sequence when the application receives an HTTP request.
- Each component performs operations on an HttpContext and either invokes the next middleware in the pipeline or terminates the request.
- They can perform a whole host of functions, such as logging, identifying the current user for a request, serving static files, and handling errors.
- The order of the calls in this method is important, as the order they're added to the builder in is the order they'll execute in the final pipeline.
- Middleware can only use objects created by previous middleware in the pipeline—it can't access objects created by later middleware.
- If you're performing authorization in middleware to restrict the users that may access your application, you must ensure it comes *after* the authentication middleware that identifies the current user.

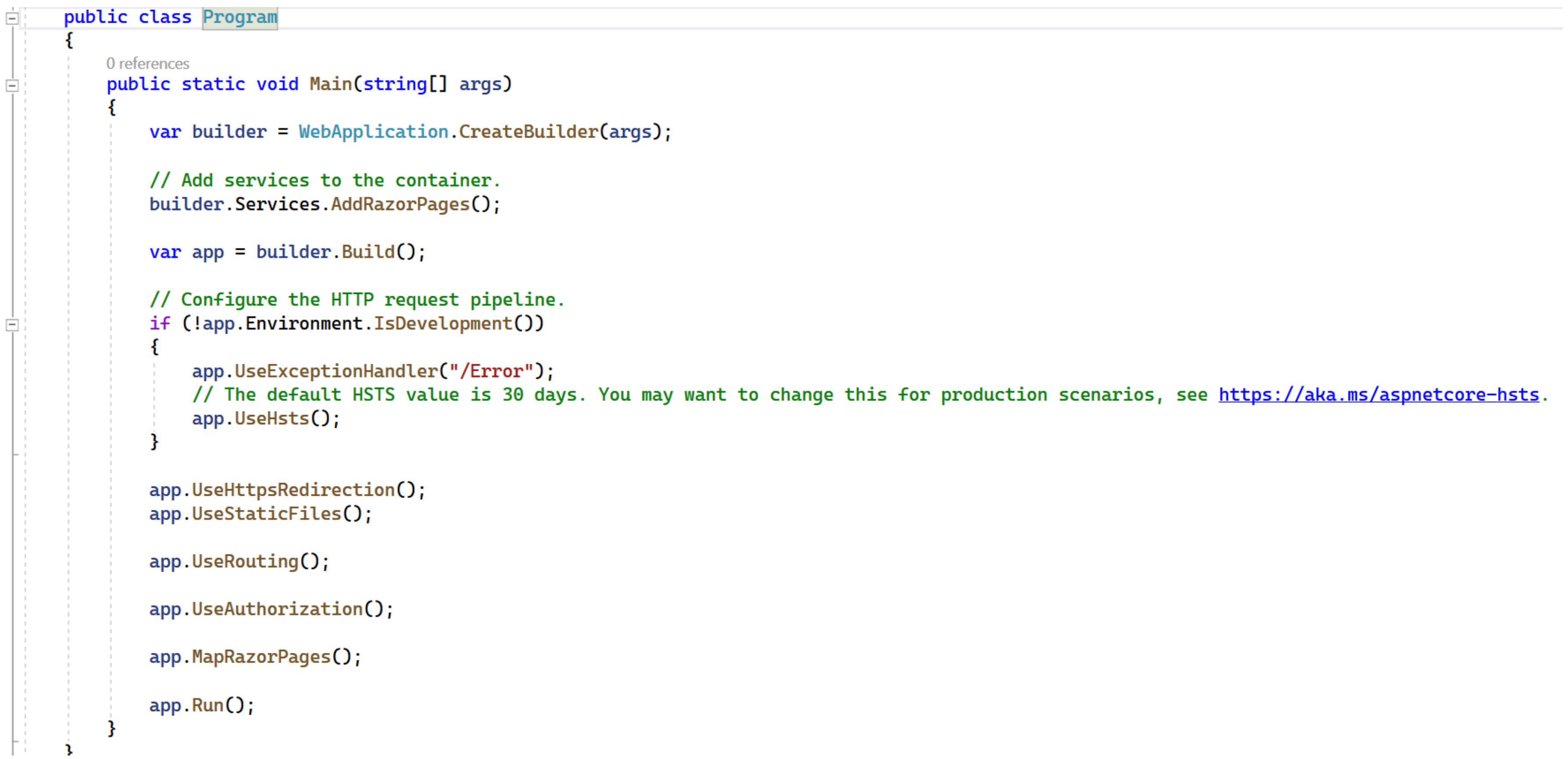
Defining how requests are handled with middleware

- By convention, a middleware component is added to the pipeline using Run, Map, and Use extension methods.





Program.cs for RazorPages



```
public class Program
{
    0 references
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

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

        var app = builder.Build();

        // Configure the HTTP request pipeline.
        if (!app.Environment.IsDevelopment())
        {
            app.UseExceptionHandler("/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.MapRazorPages();

        app.Run();
    }
}
```

Program.cs

- Program.cs still contains the entry point for the program.
  - This file contains a **static void Main function**, which is a standard characteristic of console apps.
- In ASP.NET Core 6, the WebApplication class is a new lightweight alternative to the WebHost class, which was used in earlier versions of ASP.NET Core.
  - The WebApplication class is designed to be more lightweight and performant than the WebHost class, while still providing the same functionality for building and running web applications.
- `WebApplication.CreateBuilder(args)` This method creates a new WebApplicationBuilder instance that can be used to build and configure the web host.

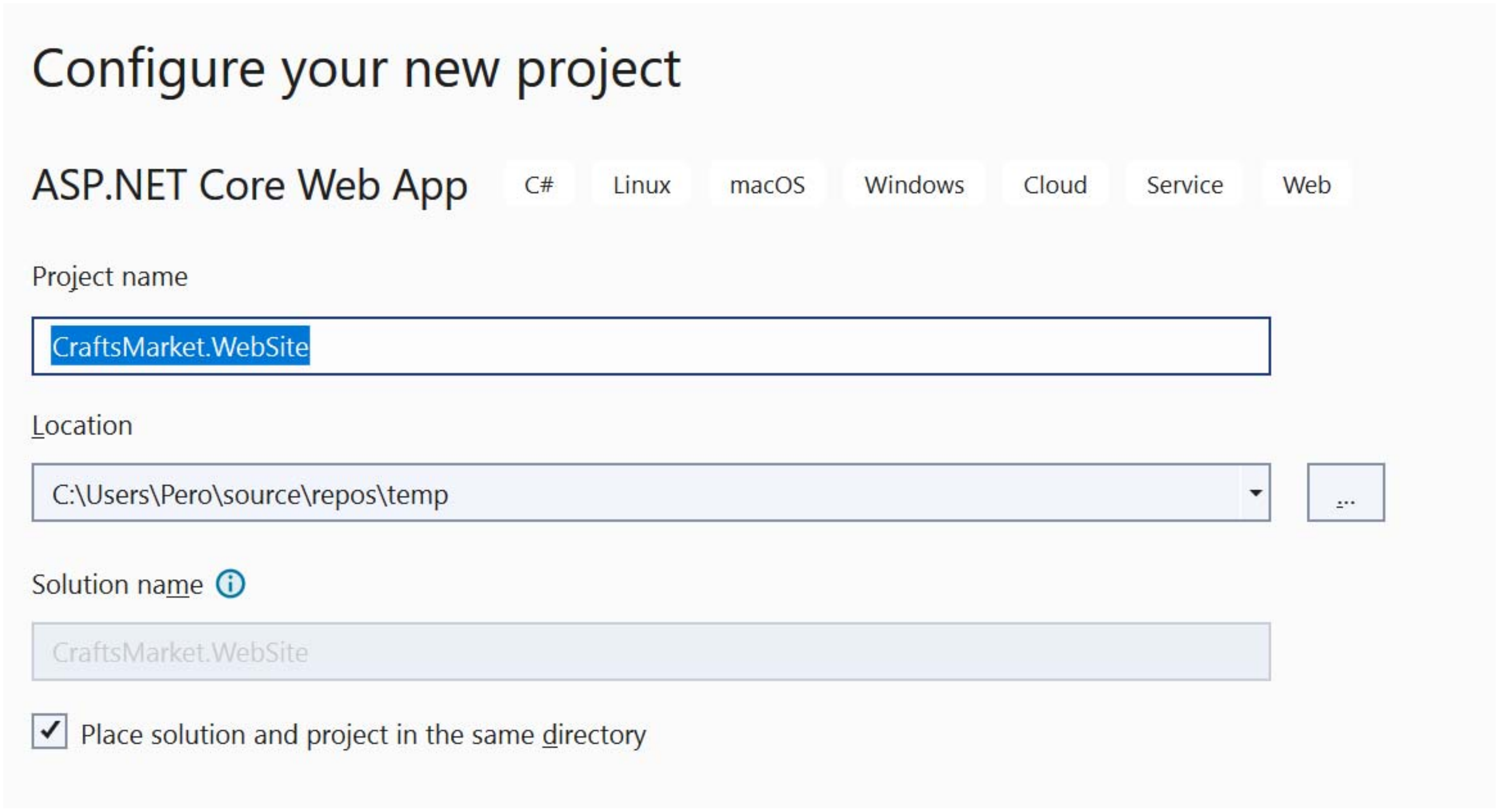
```
// Add services to the DI container.
```

- use the `builder.Services.AddRazorPages()` method to register the Razor Pages services with the dependency injection container. Later use the `app.MapRazorPages()` method to map Razor Pages to incoming requests (adds endpoints for Razor Pages to the IEndpointRouteBuilder.)
- `var app = builder.Build();` This statement builds the web host and returns an instance of WebApplication.

Program.cs

- How to add middleware components to the pipeline, map Razor Pages to incoming requests, and run the application
- `// Configure the HTTP request pipeline.`
- `if (!app.Environment.IsDevelopment())` This statement is typically used to configure middleware that should only be enabled in production environments.
  - `app.UseExceptionHandler("/Error")` is used to configure a middleware that catches any unhandled exceptions that occur during request processing and handles them gracefully.
  - you might want to use the app.UseHsts() middleware to enable HTTP Strict Transport Security (HSTS) in production, but not in development or testing environments.
  - HSTS is a security feature that instructs web browsers to only access your site over HTTPS, and can help protect your users against certain types of attacks.
- `app.UseHttpsRedirection()` method: This method adds middleware to redirect HTTP requests to HTTPS if the app is running on HTTPS.
- `app.Run();` This method runs the application.

Пример (online market for crafts)



Configure your new project

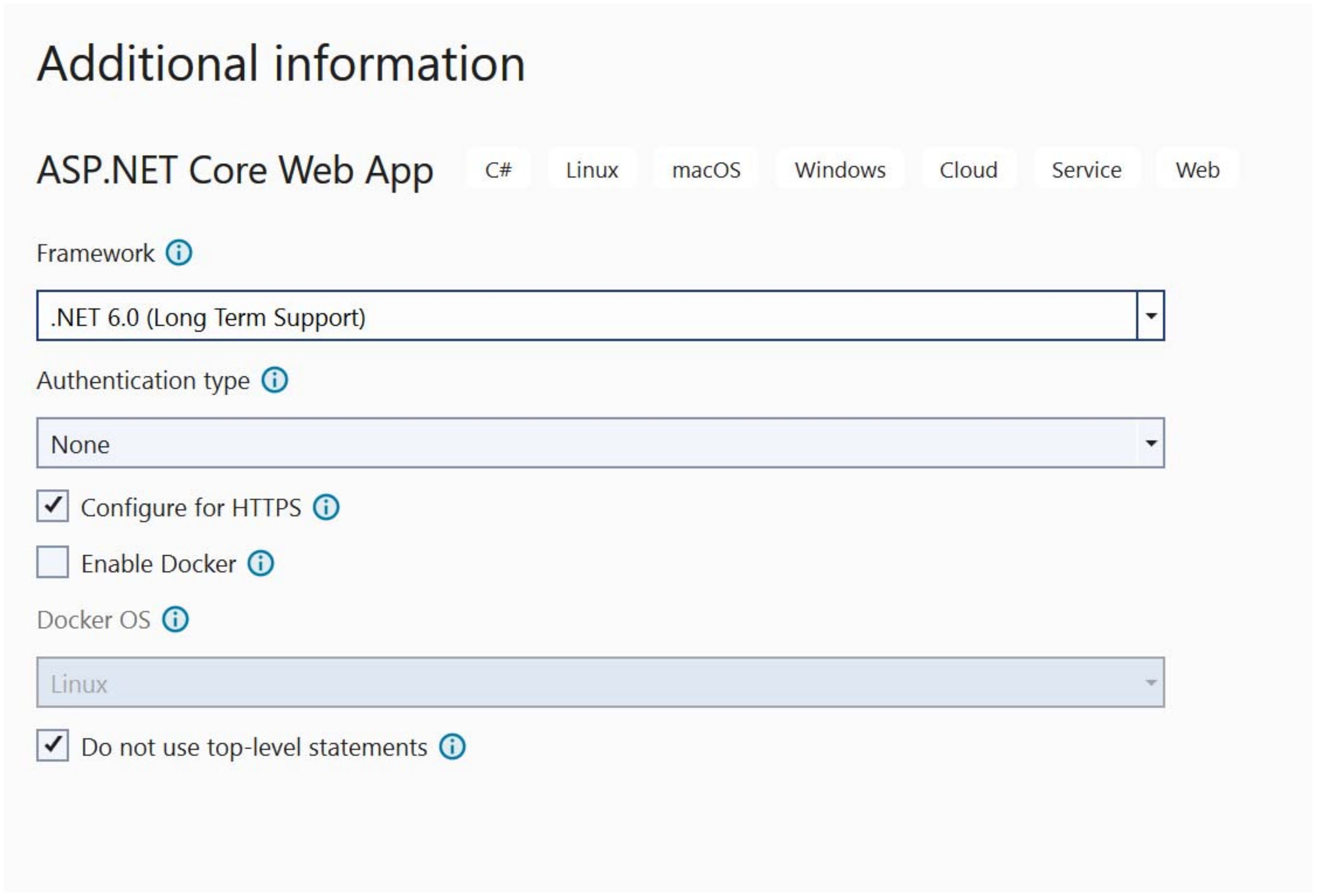
ASP.NET Core Web App

Project name: CraftsMarket.WebSite

Location: C:\Users\Pero\source\repos\temp

Solution name: CraftsMarket.WebSite

[✓] Place solution and project in the same directory



Additional information

ASP.NET Core Web App

Framework: .NET 6.0 (Long Term Support)

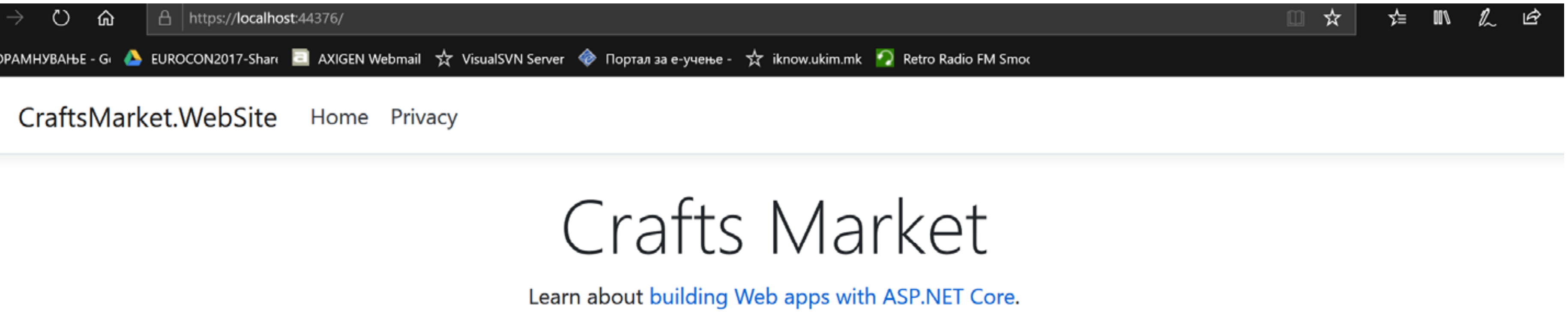
Authentication type: None

[✓] Configure for HTTPS

[ ] Enable Docker

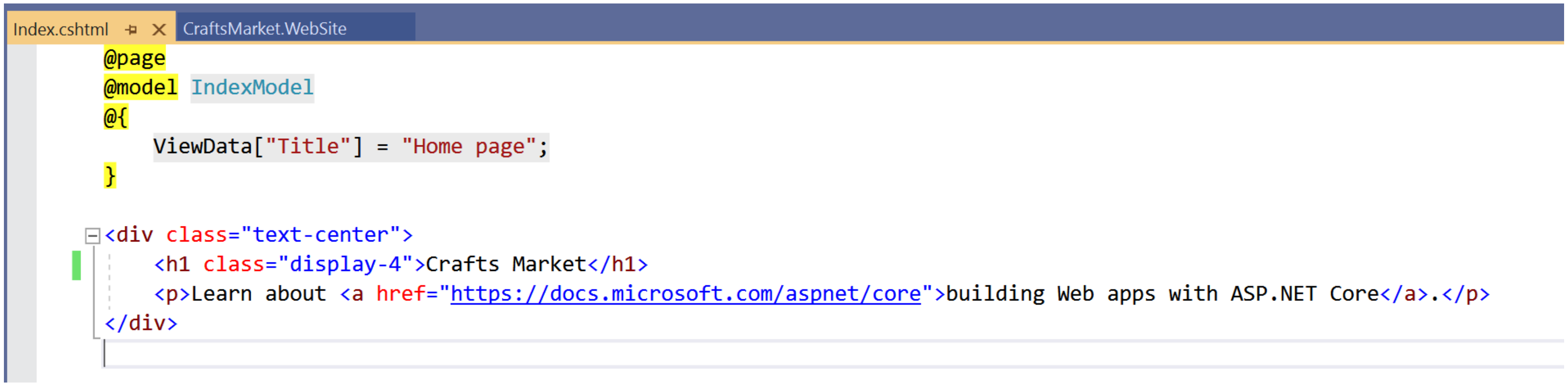
Docker OS: Linux

[✓] Do not use top-level statements



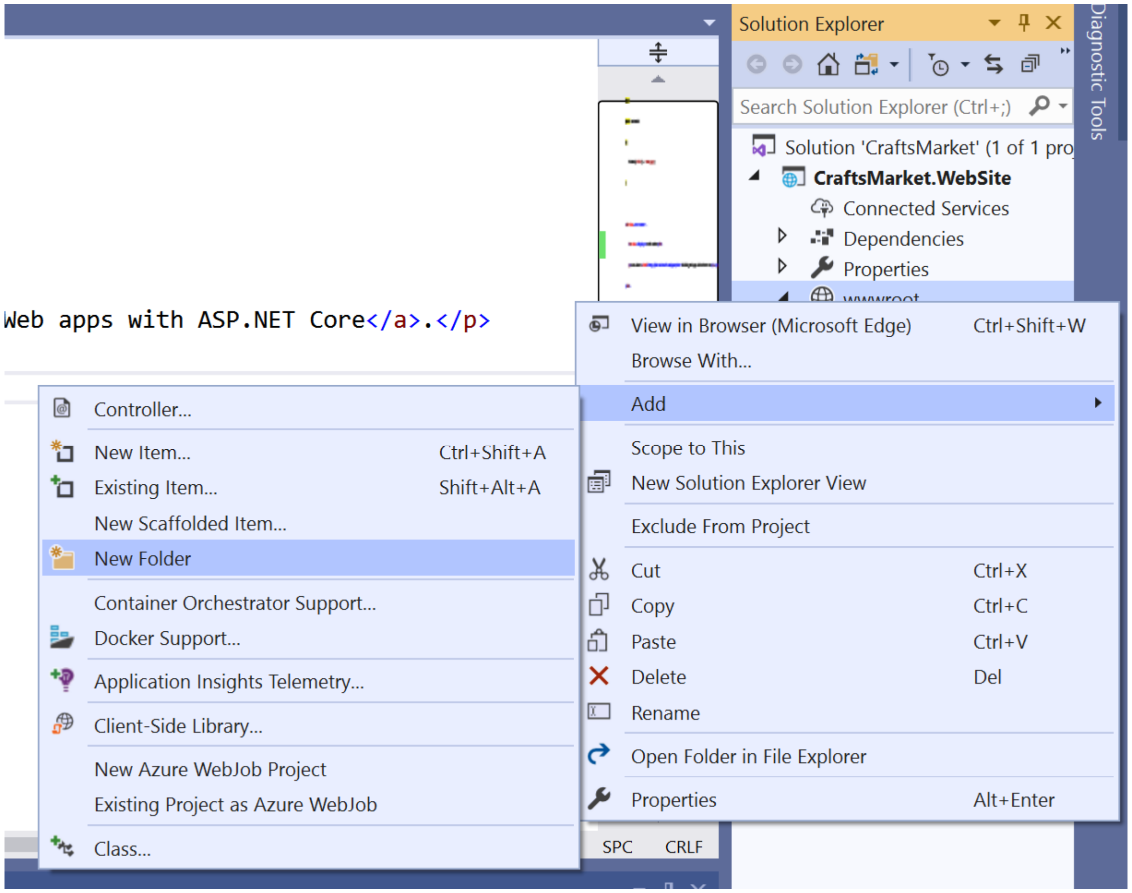
Crafts Market

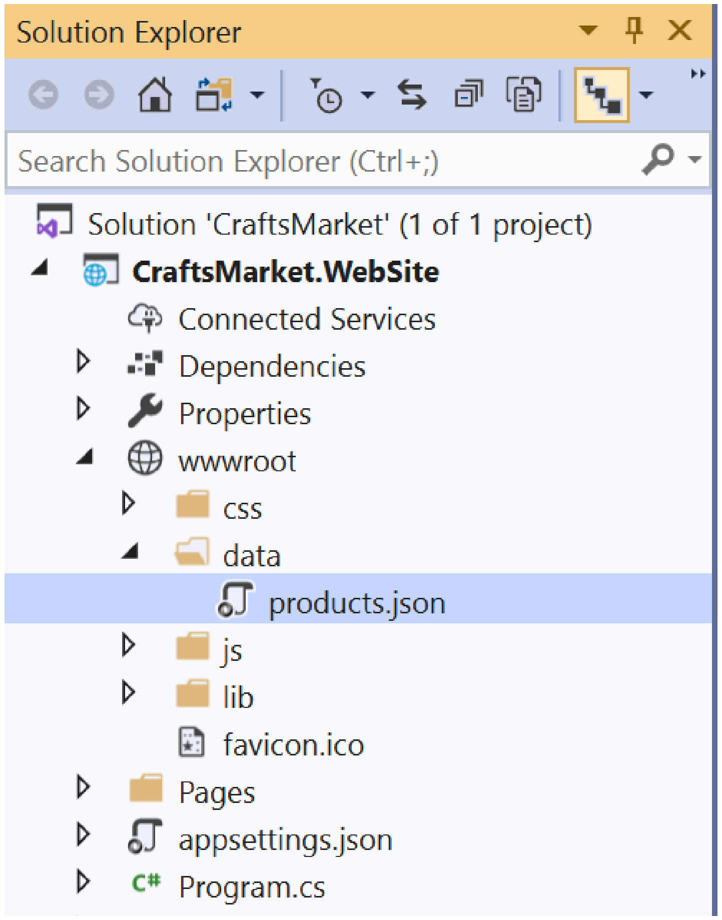
Learn about building Web apps with ASP.NET Core.

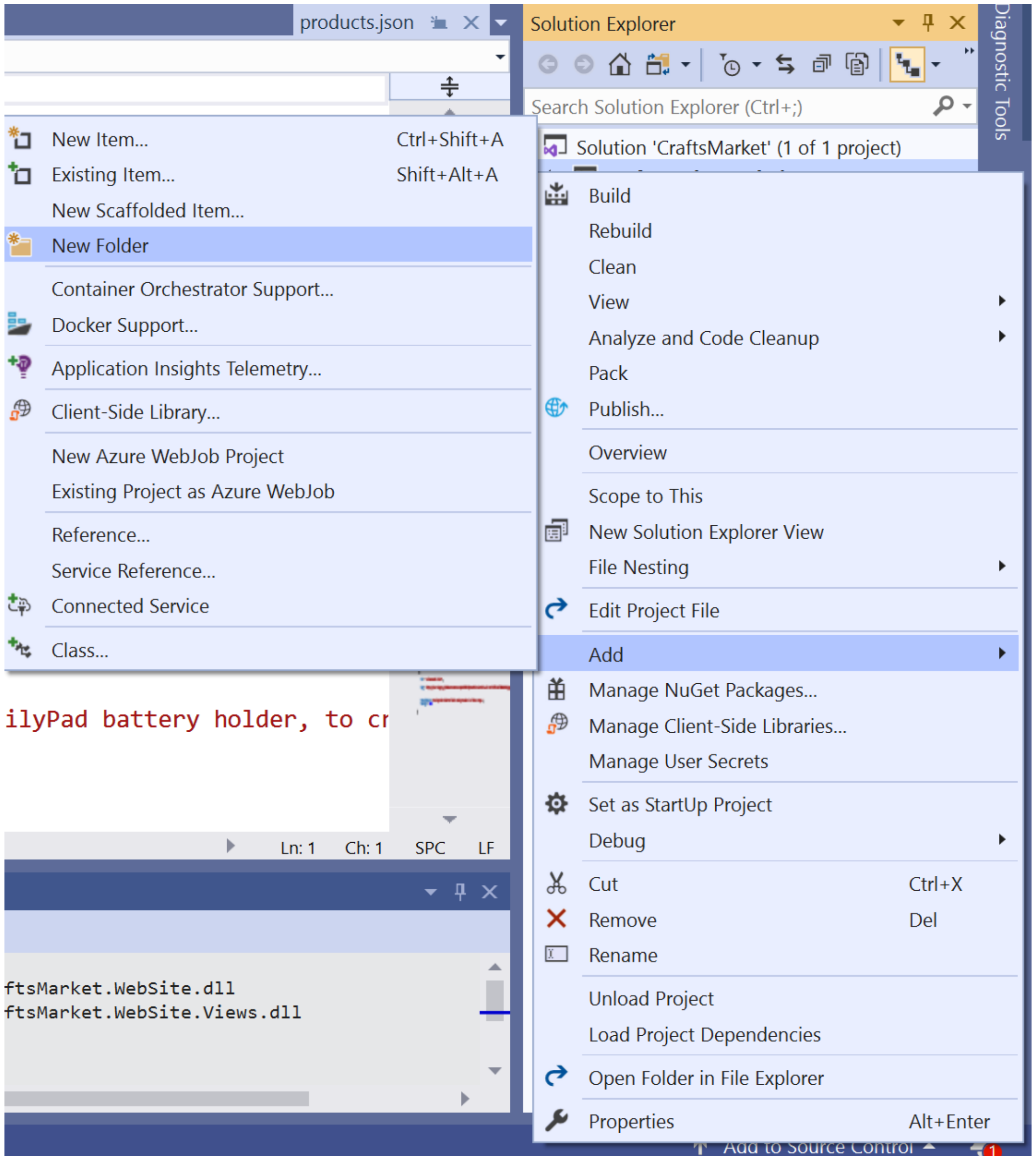


```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Crafts Market</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>
</div>
```

Adding data to ASP.NET Core

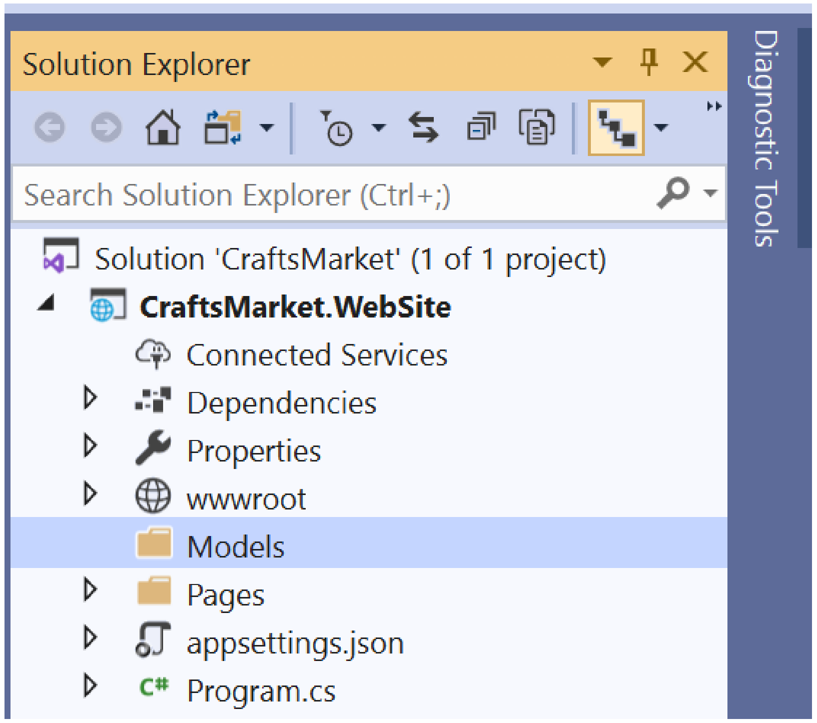


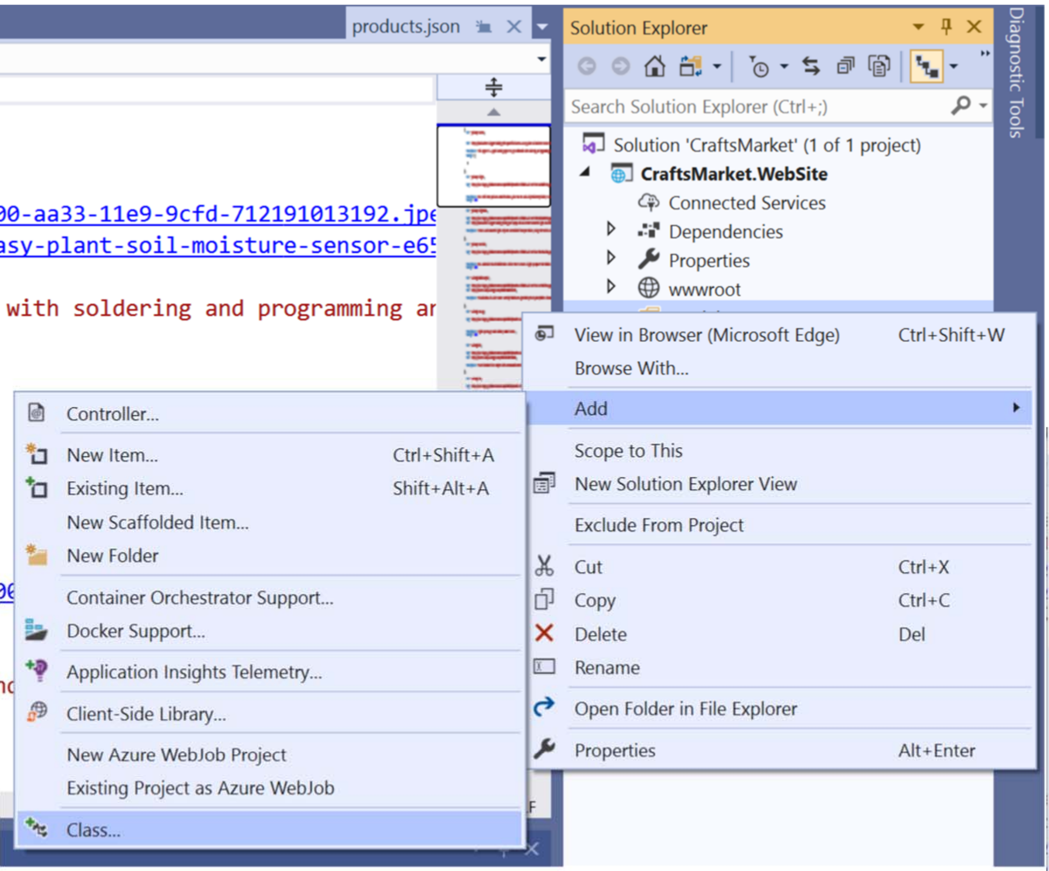




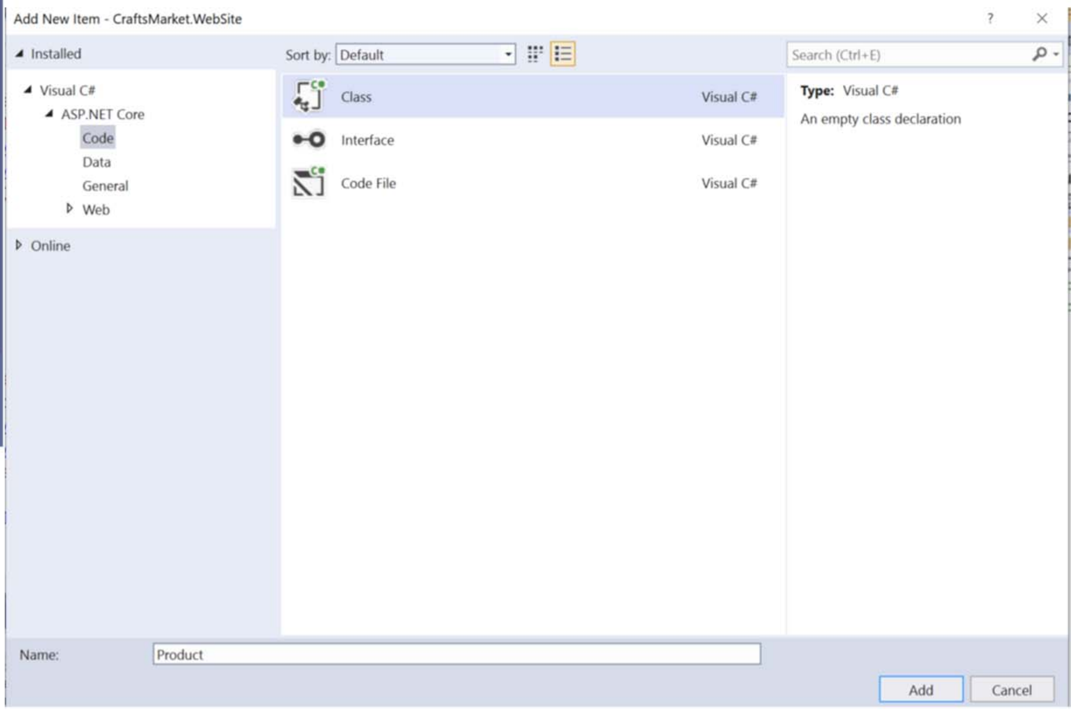
```
ilyPad battery holder, to cr
```

```
ftsMarket.WebSite.dll
ftsMarket.WebSite.Views.dll
```





```
00-aa33-11e9-9cfd-712191013192.jpe
asy-plant-soil-moisture-sensor-e65
 with soldering and programming ar
```



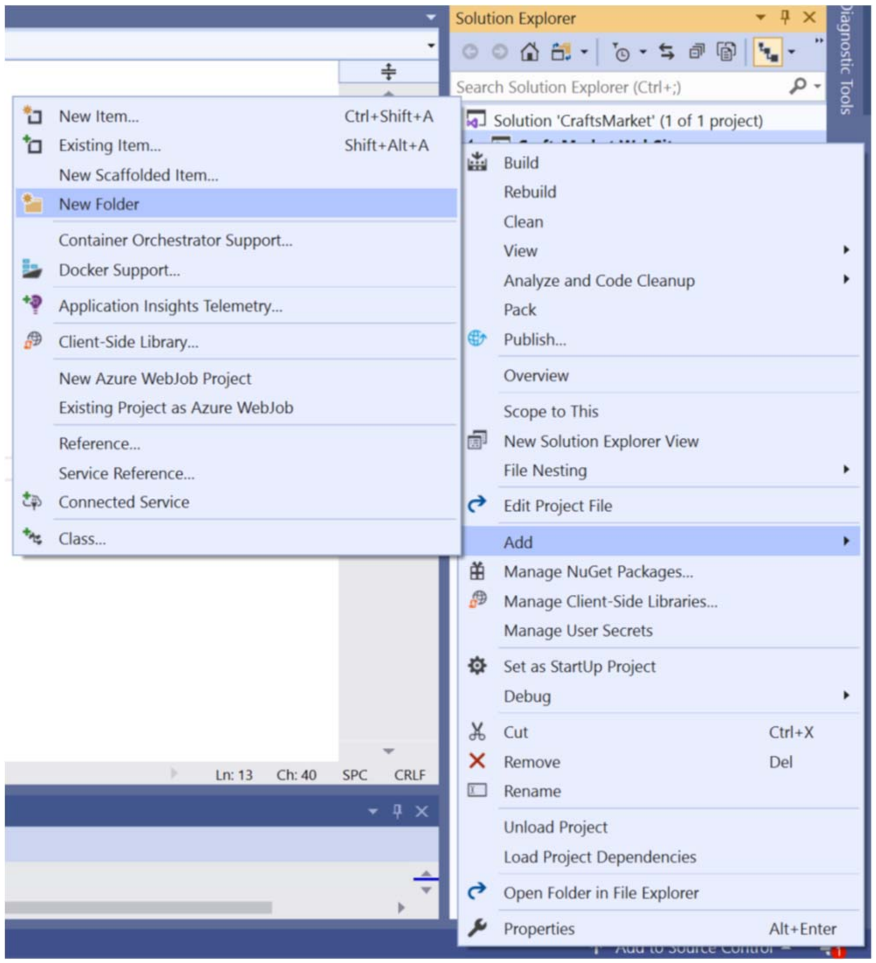
Name: Product

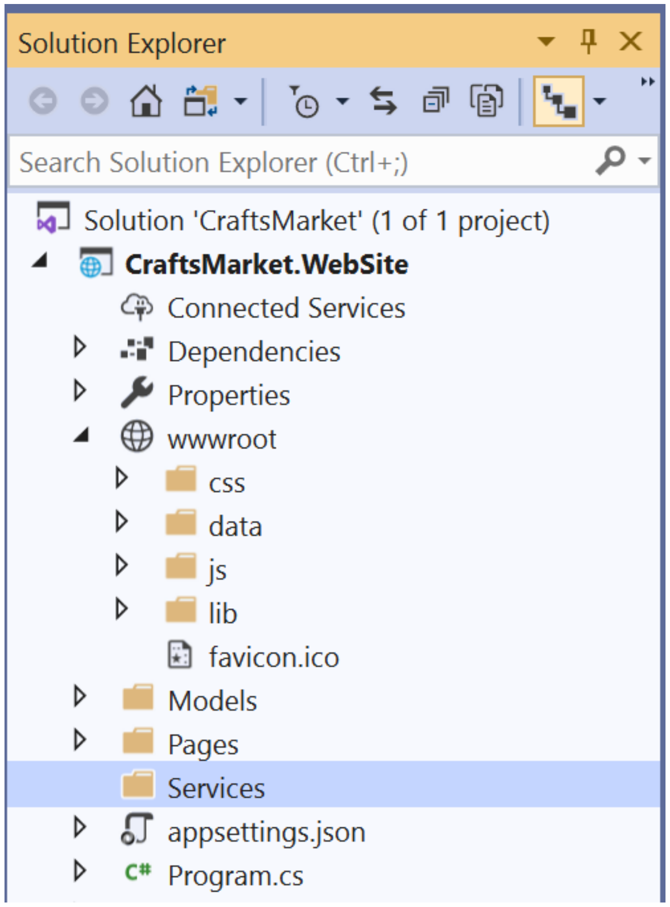
Product.cs

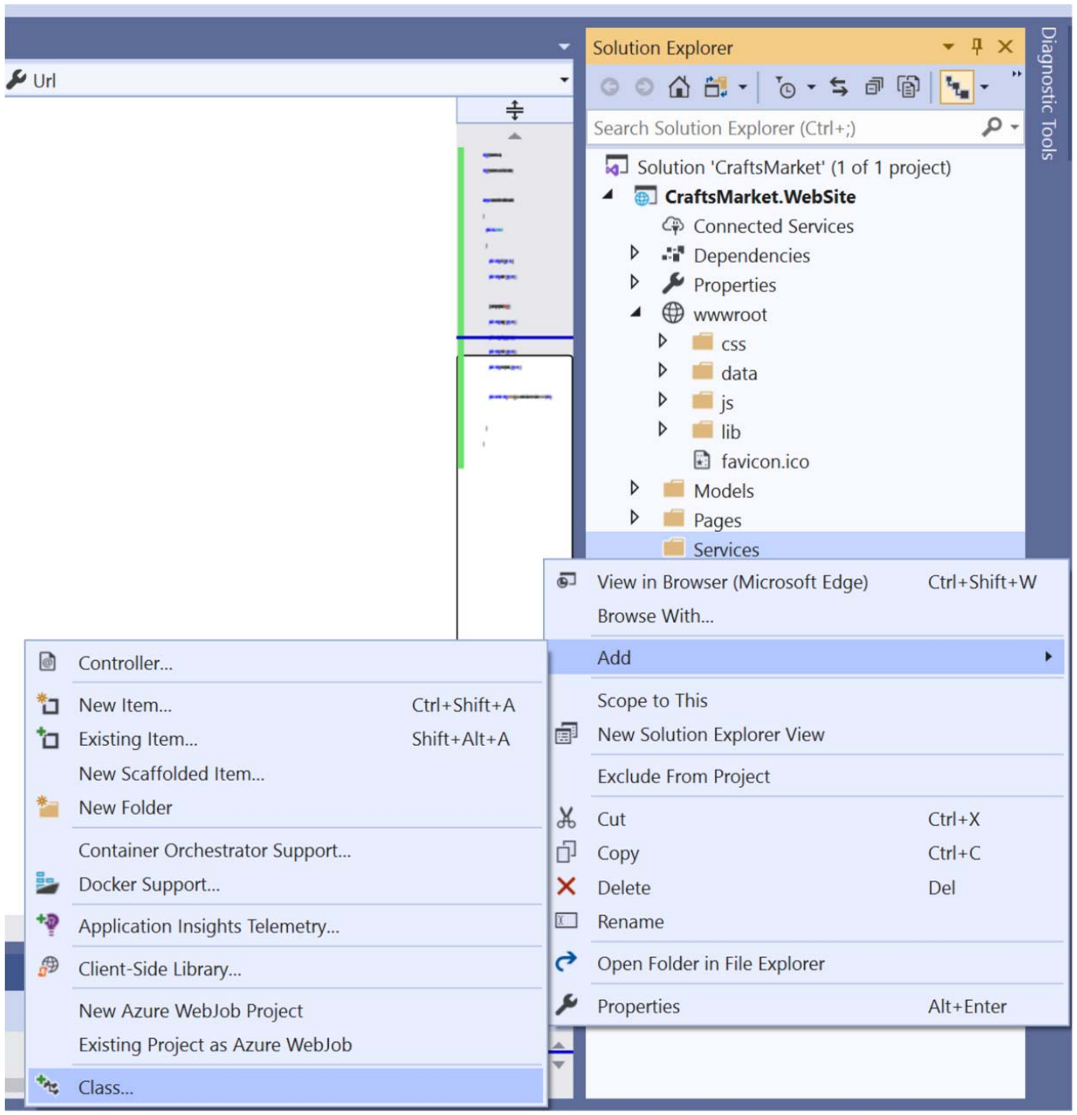
```
using System.Text.Json.Serialization;

namespace CraftsMarket.WebSite.Models
{
    public class Product
    {
        public string Id { get; set; }
        public string Maker { get; set; }
        [JsonPropertyName("img")]
        public string Image { get; set; }
        public string Url { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public int[] Ratings { get; set; }

    }
}
```

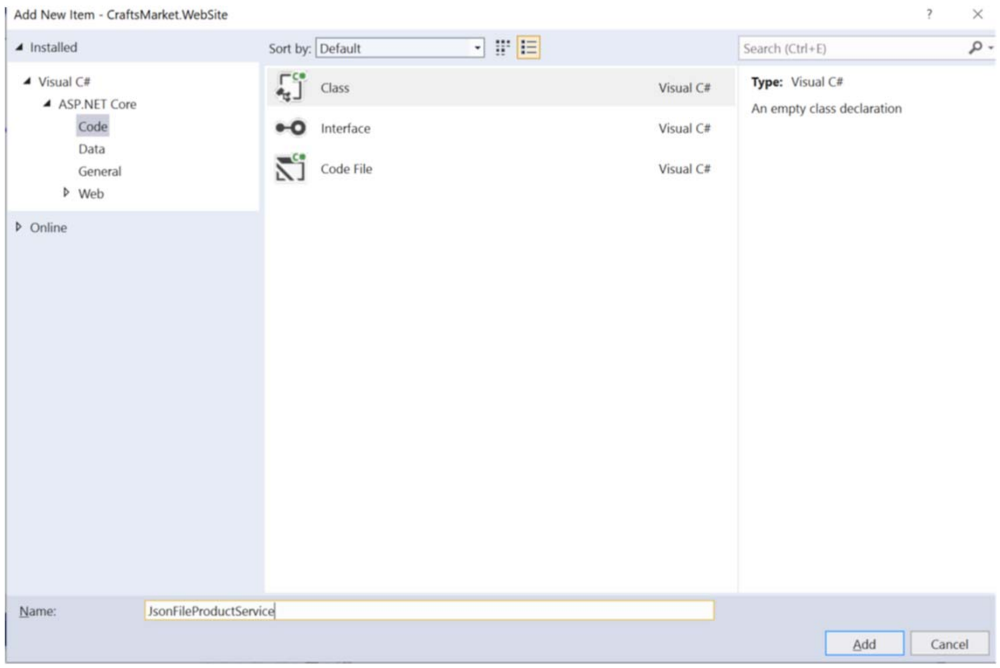
an attribute in the System.Text.Json.Serialization namespace used when you want to map a property name in your .NET class to a different property name in the JSON object.

Adding a service to ASP.NET Core









Name: JsonFileProductService

JsonFileProductService.sc

```
using CraftsMarket.WebSite.Models;
using Microsoft.AspNetCore.Hosting;
using System.Text.Json;

namespace CraftsMarket.WebSite.Services
{
    public class JsonFileProductService
    {
        public JsonFileProductService(IWebHostEnvironment webHostEnvironment)
        {
            WebHostEnvironment = webHostEnvironment;
        }
        public IWebHostEnvironment WebHostEnvironment { get; }

        private string JsonFileName
        {
            get { return Path.Combine(WebHostEnvironment.WebRootPath, "data", "products.json"); }
        }

        public Product[] GetProducts()
        {
            var jsonFileReader = File.OpenText(JsonFileName);
            return JsonSerializer.Deserialize<Product[]>(jsonFileReader.ReadToEnd());
        }

    }
}
```

Program.sc

```
// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddTransient<JsonFileProductService>();
```

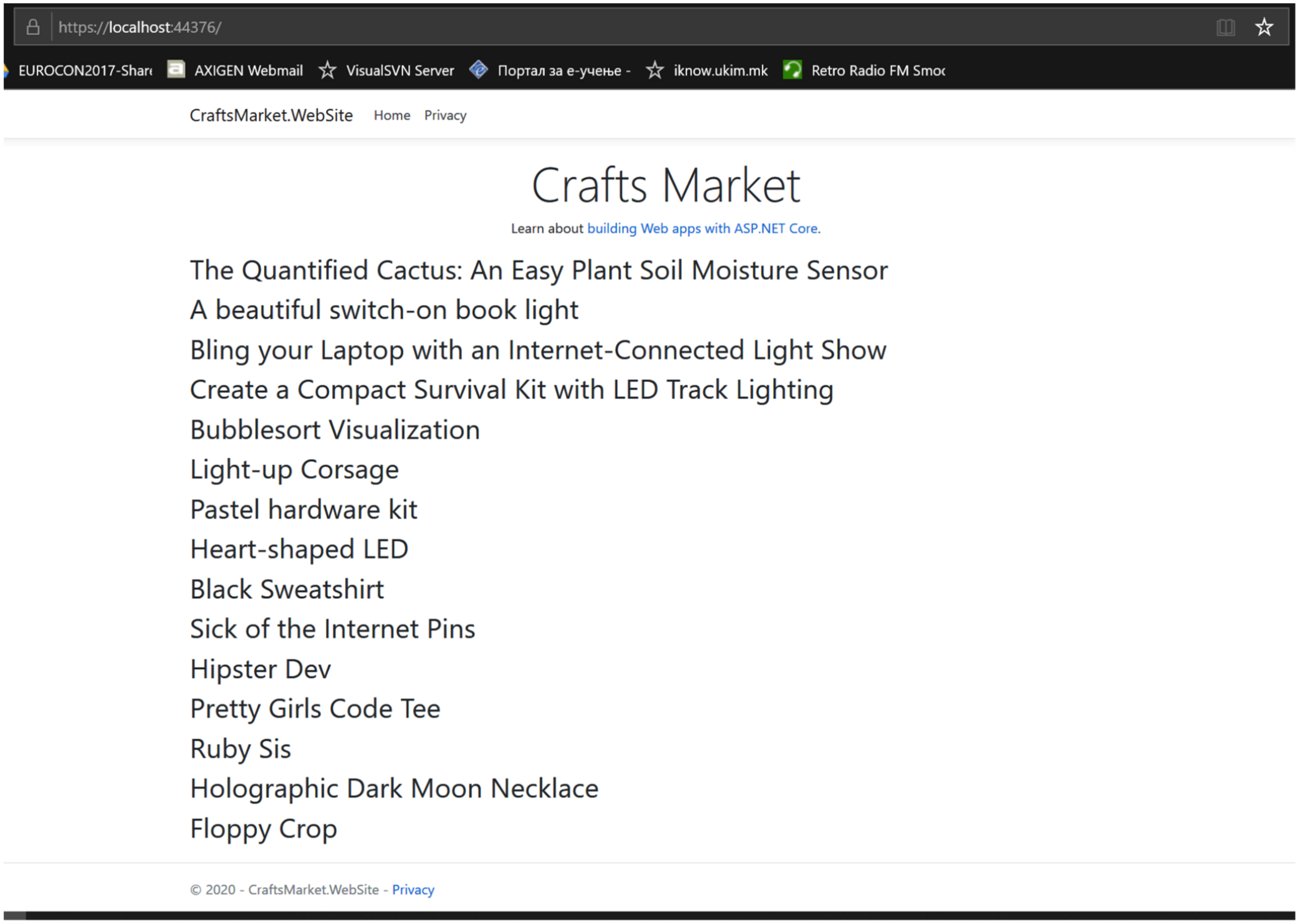
When a service is registered as transient, a new instance of the service will be created every time it is requested from the dependency injection container. This can be useful in scenarios where you want a fresh instance of the service each time it is requested.

Data in a Razor Page (PageModel)

```
using CraftsMarket.WebSite.Models;
using CraftsMarket.WebSite.Services;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace CraftsMarket.WebSite.Pages
{
    public class IndexModel : PageModel
    {
        private readonly ILogger<IndexModel> _logger;
        public JsonFileProductService ProductService { get; }
        public Product[] Products { get; private set; }


        public IndexModel(
            ILogger<IndexModel> logger,
            JsonFileProductService productService)
        {
            _logger = logger;
            ProductService = productService;
        }


        public void OnGet()
        {
            Products = ProductService.GetProducts();

        }
    }
}
```

Data in the Razor Page

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Crafts Market</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps 
with ASP.NET Core</a>.</p>
</div>

@foreach (var product in Model.Products)
{
    <h2>@product.Title</h2>
}
```



Crafts Market

Learn about building Web apps with ASP.NET Core.

The Quantified Cactus: An Easy Plant Soil Moisture Sensor
A beautiful switch-on book light
Bling your Laptop with an Internet-Connected Light Show
Create a Compact Survival Kit with LED Track Lighting
Bubblesort Visualization
Light-up Corsage
Pastel hardware kit
Heart-shaped LED
Black Sweatshirt
Sick of the Internet Pins
Hipster Dev
Pretty Girls Code Tee
Ruby Sis
Holographic Dark Moon Necklace
Floppy Crop

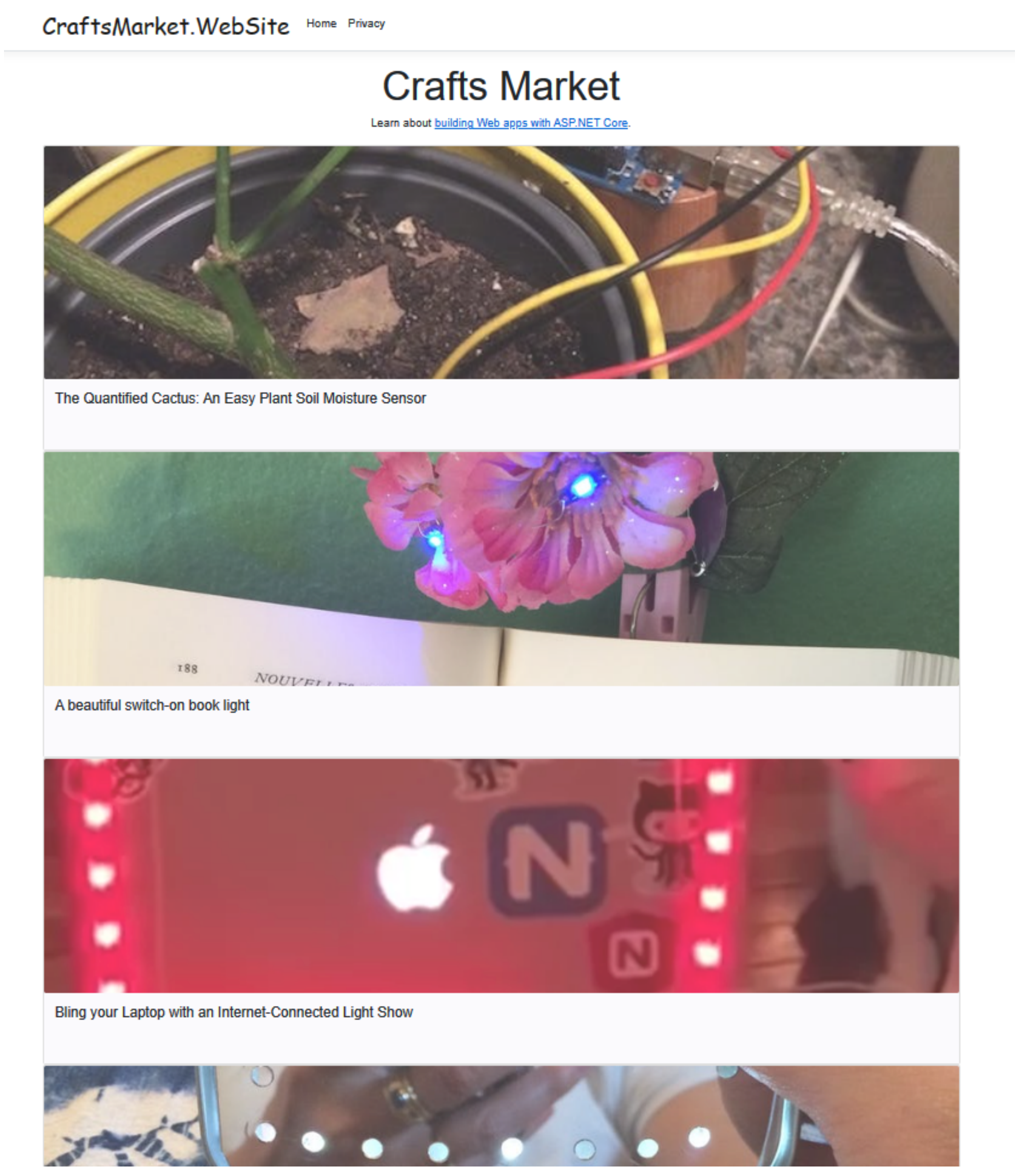
© 2020 - CraftsMarket.WebSite - Privacy

Styling the Razor Page

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Crafts Market</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>
</div>

<div class="card-columns">
    @foreach (var product in Model.Products)
    {
        <div class="card">
            <div class="card-img" style="background-image:url('@product.Image')"></div>
            <div class="card-body">
                <h5>@product.Title</h5>
            </div>
        </div>
    }
</div>
```

CSS

```css
.checked {
    color: orange;
}

.fa-star {
    cursor: pointer;
}

/* Please see documentation at https://docs.microsoft.com/aspnet/core/client-side/bundling-and-minification
for details on configuring this project to bundle and minify static web assets. */
.card-columns .card:hover .card-img {
    opacity: 1;
    -webkit-transform-style: preserve-3d;
}

.card-columns .card-body {
    height: 100px;
    font-family: 'Nunito', sans-serif;
    background: #fbfafd;
}

.card-columns .card-img {
    height: 330px;
    vertical-align: bottom;
    background-position: center; /* Center the image */
    background-repeat: no-repeat; /* Do not repeat the image */
    background-size: cover; /* Resize the background image to cover the entire container */
    opacity: .8;
}

.modal .card-img {
    height: 500px;
    vertical-align: bottom;
    background-position: center; /* Center the image */
    background-repeat: no-repeat; /* Do not repeat the image */
    background-size: cover; /* Resize the background image to cover the entire container */
}

.card-columns .card:hover {
    transform: scale(1.05);
    box-shadow: 0 10px 20px rgba(37,33,82,.12), 0 4px 8px rgba(37,33,82,.06);
}

.card-footer {
    background: #fbfafd;
}


a.navbar-brand {
    white-space: normal;
    font-family: 'Yellowtail', cursive;
    font-size: xx-large;
    text-align: center;
    word-break: break-all;
}

/* Provide sufficient contrast against white background */
a {
    color: #0366d6;
}

.btn-primary {
    color: #fff;
    background-color: #252152;
    border-color: #252152;
}

    .btn-primary:hover {
        color: #fff;
        background-color: #e9a8a6;
        border-color: #e9a8a6;
    }

.nav-pills .nav-link.active, .nav-pills .show > .nav-link {
    color: #fff;
    background-color: #1b6ec2;
    border-color: #1861ac;
}

/* Sticky footer styles
-------------------------------------------------- */
html {
    font-size: 14px;
}

@media (min-width: 768px) {
    html {
        font-size: 16px;
    }
}

.border-top {
    border-top: 1px solid #e5e5e5;
}

.border-bottom {
    border-bottom: 1px solid #e5e5e5;
}

.box-shadow {
    box-shadow: 0 .25rem .75rem rgba(0, 0, 0, .05);
}

button.accept-policy {
    font-size: 1rem;
    line-height: inherit;
}

/* Sticky footer styles
-------------------------------------------------- */
html {
    position: relative;
    min-height: 100%;
}

body {
    /* Margin bottom by footer height */
    margin-bottom: 60px;
}

p {
    font-family: 'Nunito', sans-serif;
}

h1 {
    font-family: 'Nunito', sans-serif;
}

ul {
    font-family: 'Nunito', sans-serif;
}

.footer {
    position: absolute;
    bottom: 0;
    width: 100%;
    white-space: nowrap;
    line-height: 60px; /* Vertically center the text there */
    font-family: 'Nunito', sans-serif;
}
/*Backgrounds*/
.bg-navbar {
    background: -webkit-linear-gradient(110deg, #e9a8a6 60%, #252152 60%);
    background: -o-linear-gradient(110deg, #e9a8a6 60%, #252152 60%);
    background: -moz-linear-gradient(110deg, #e9a8a6 60%, #252152 60%);
    background: linear-gradient(110deg, #e9a8a6 60%, #252152 60%);
}

.bg-footer {
    background: -webkit-linear-gradient(110deg, #252152 60%, #e9a8a6 60%);
    background: -o-linear-gradient(110deg, #252152 60%, #e9a8a6 60%);
    background: -moz-linear-gradient(110deg, #252152 60%, #e9a8a6 60%);
    background: linear-gradient(110deg, #252152 60%, #e9a8a6 60%);
}
```



CraftsMarket.WebSite Home Privacy

Crafts Market

Learn about [building Web apps with ASP.NET Core](#).

The Quantified Cactus: An Easy Plant Soil Moisture Sensor

A beautiful switch-on book light

Bling your Laptop with an Internet-Connected Light Show