

# Microsoft.Net Solution

Develop Your **Microsoft** Skills and Knowledge through great Training



**Hyderabad's Best Institute for .NET**

# ASP.NET Core Razor Pages

**Rakesh Singh**

**RAKESHNET TECHNOLOGIES Hyderabad**

 <http://www.rakeshsoftnet.com>

 <https://www.facebook.com/RakeshSoftNet>

 <https://www.facebook.com/RakeshSoftNetTech>

 <https://twitter.com/RakeshSoftNet>

 **+91 89191 36822**

## ASP.NET Core

ASP.NET Core is a free, open-source and cloud optimized framework which can run on Windows, Linux or macOS. You can say that it is the new version of ASP.NET. The framework is a complete rewrite from scratch in order to make it open source, modular and cross-platform.

### ASP.NET Core Web Application with Razor Pages:

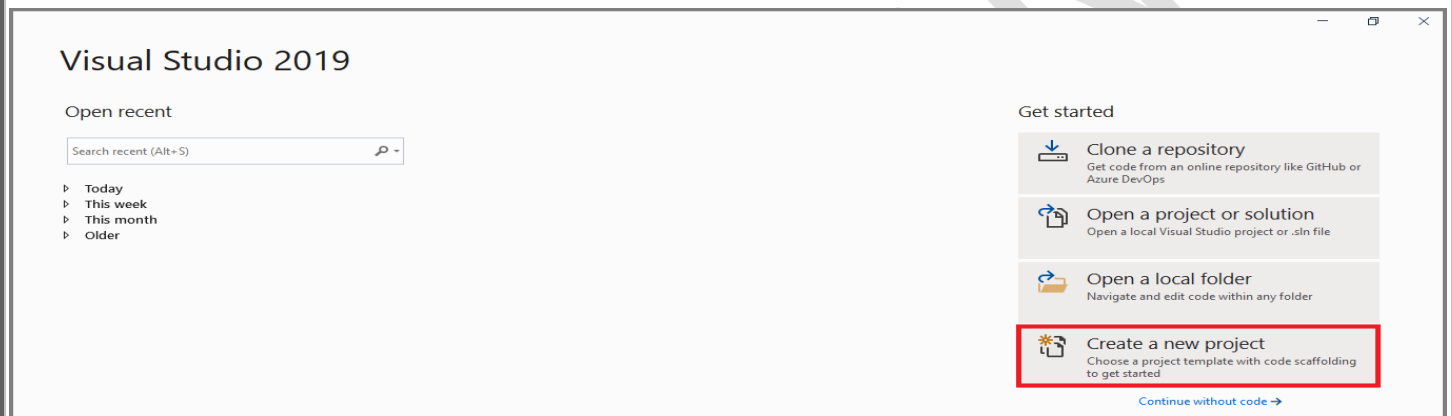
#### Introduction:

This document is dedicated to helping developers who want to use the ASP.NET Core Razor Pages web development framework to build web applications.

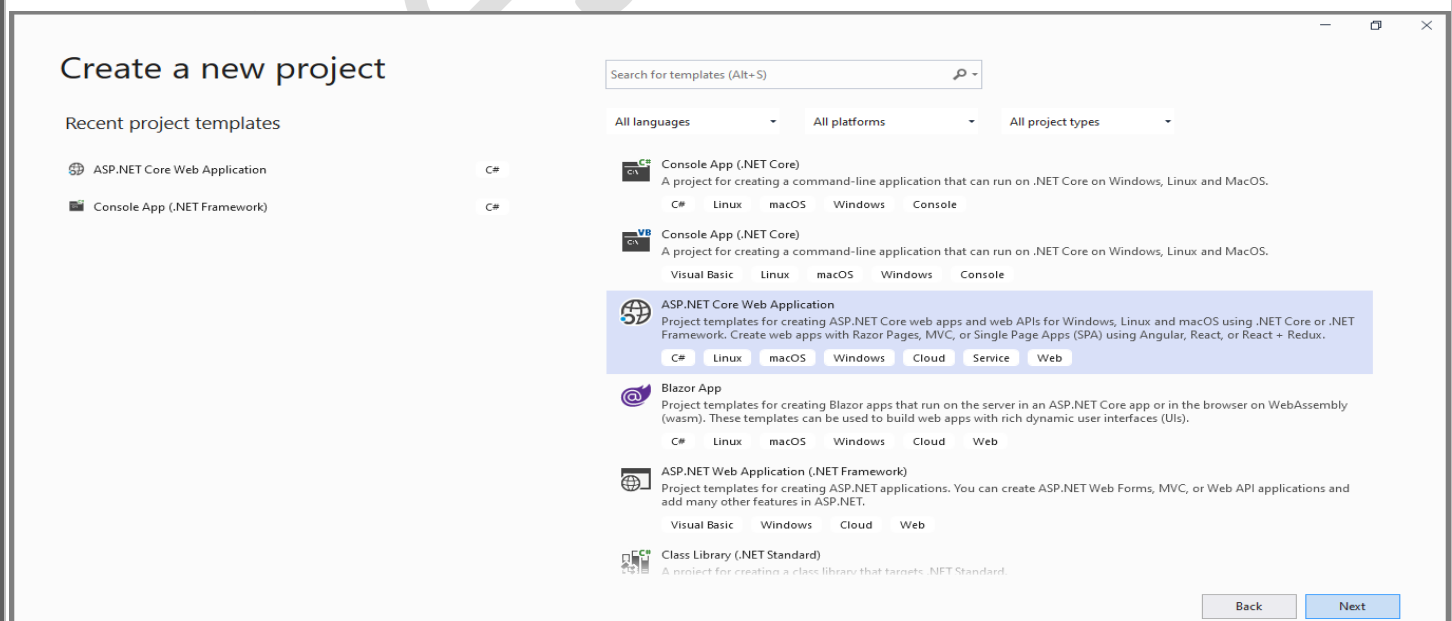
#### Create ASP.NET Core Razor Pages Web Application:

Here, we will learn how to create our first ASP.NET Core Razor Pages web application in Visual Studio 2019.

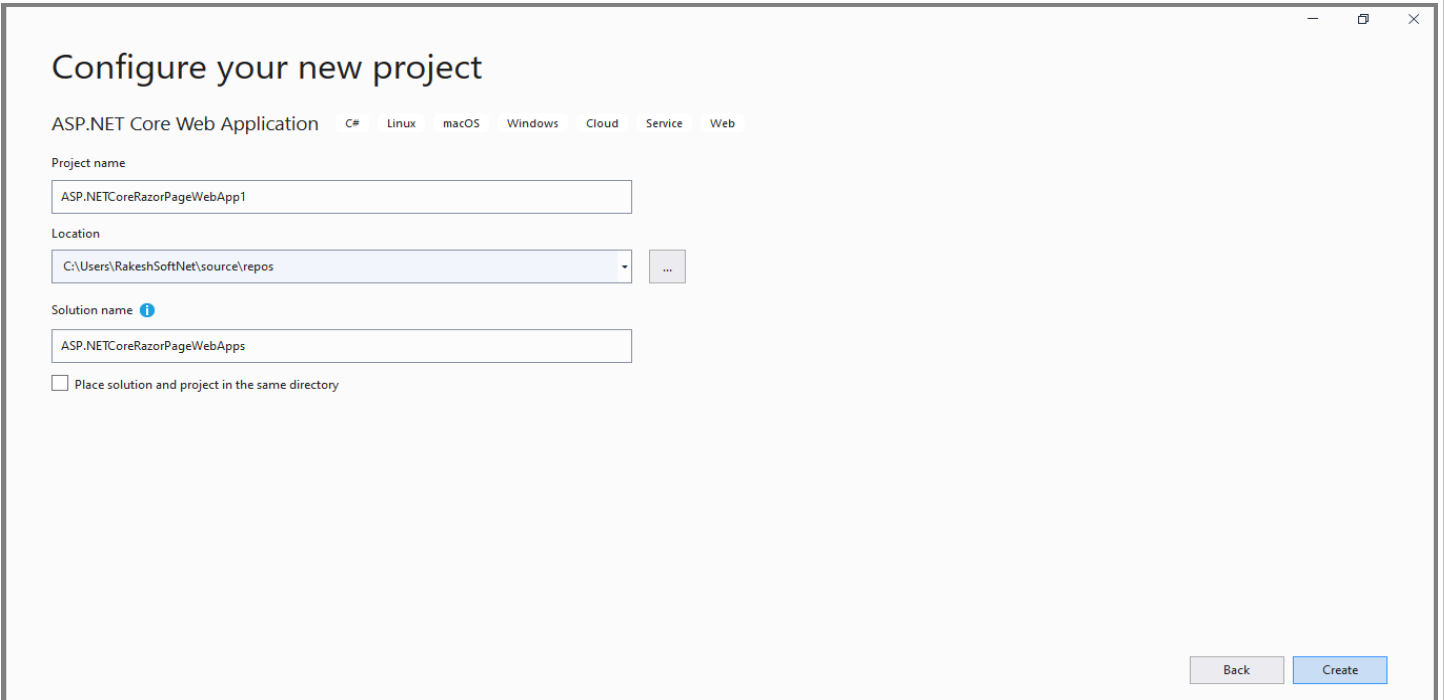
Open Visual Studio 2019 and click on **Create a new project**, as shown below:



The "Create a new project" dialog box includes different .NET Core application templates. Each will create predefined project files and folders depends on the application type. Here we will create a simple web application, so select **ASP.NET Core Web Application** template and click 'Next' button, as shown below:



Next, give the appropriate name, location, and the solution name for the ASP.NET Core application. In this example, we will give the project name "**ASP.NETCoreRazorPageWebApp1**", use default location and solution name "**ASP.NETCoreRazorPageWebApps**" and then click on the **Create** button, as shown below:



Configure your new project

ASP.NET Core Web Application C# Linux macOS Windows Cloud Service Web

Project name  
ASP.NETCoreRazorPageWebApp1

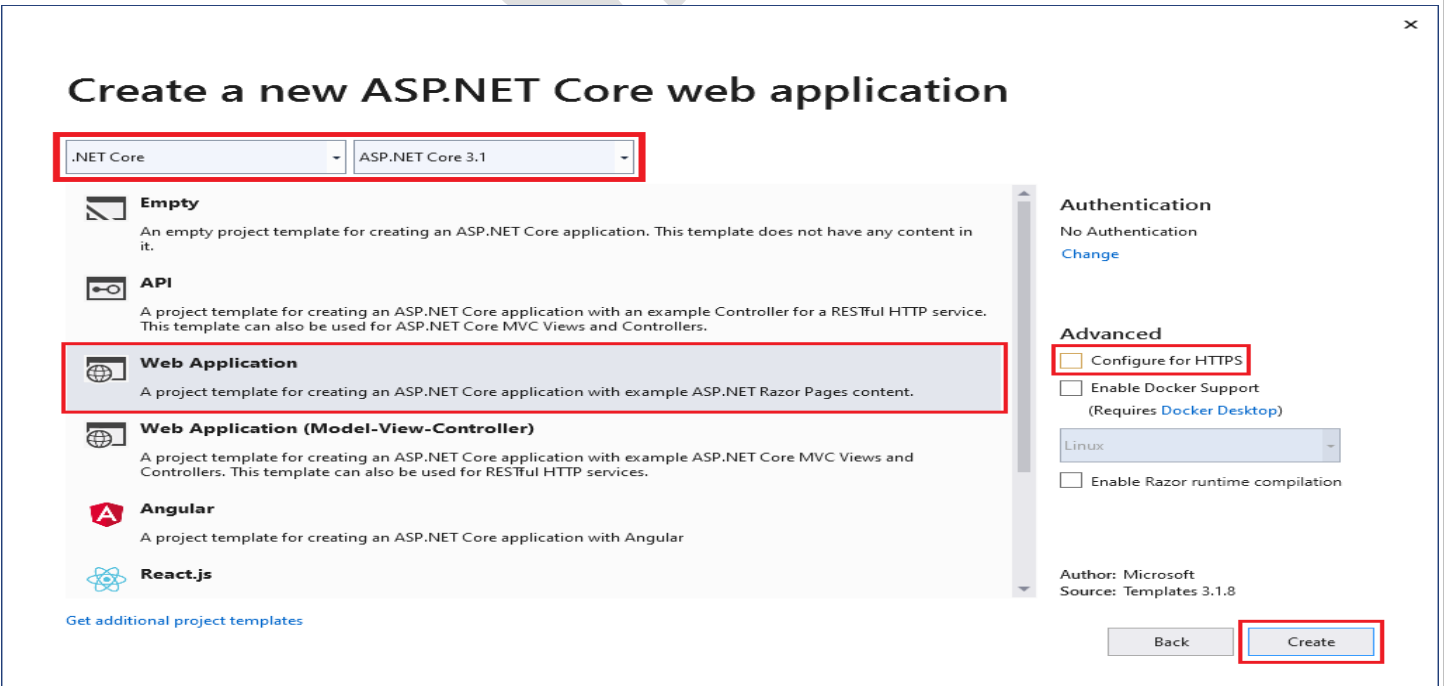
Location  
C:\Users\RakeshSoftNet\source\repos

Solution name ⓘ  
ASP.NETCoreRazorPageWebApps

☐ Place solution and project in the same directory

Back Create

Next, select appropriate ASP.NET Core Web application template such as Empty, API, Web Application, MVC, etc. Here, we want to create a web application, so select the **Web Application** template. We don't want HTTPS at this point, so uncheck **Configure for HTTPS** checkbox, as shown below. Also, make sure you have selected the appropriate .NET Core and ASP.NET Core versions. Click on the **Create** button to create a project.



Create a new ASP.NET Core web application

.NET Core ASP.NET Core 3.1

**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**

Get additional project templates

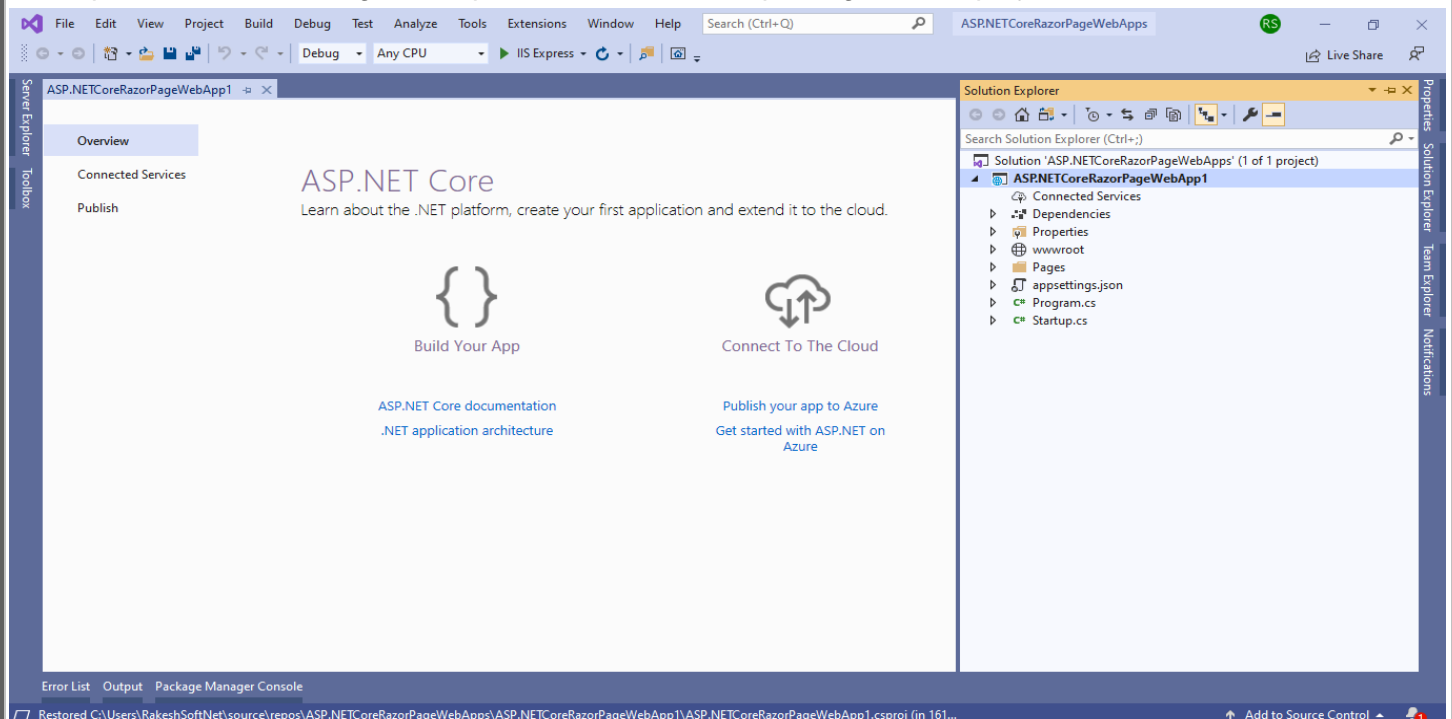
**Authentication**  
No Authentication  
[Change](#)

**Advanced**  
☐ **Configure for HTTPS**  
☐ Enable Docker Support (Requires [Docker Desktop](#))  
Linux  
☐ Enable Razor runtime compilation

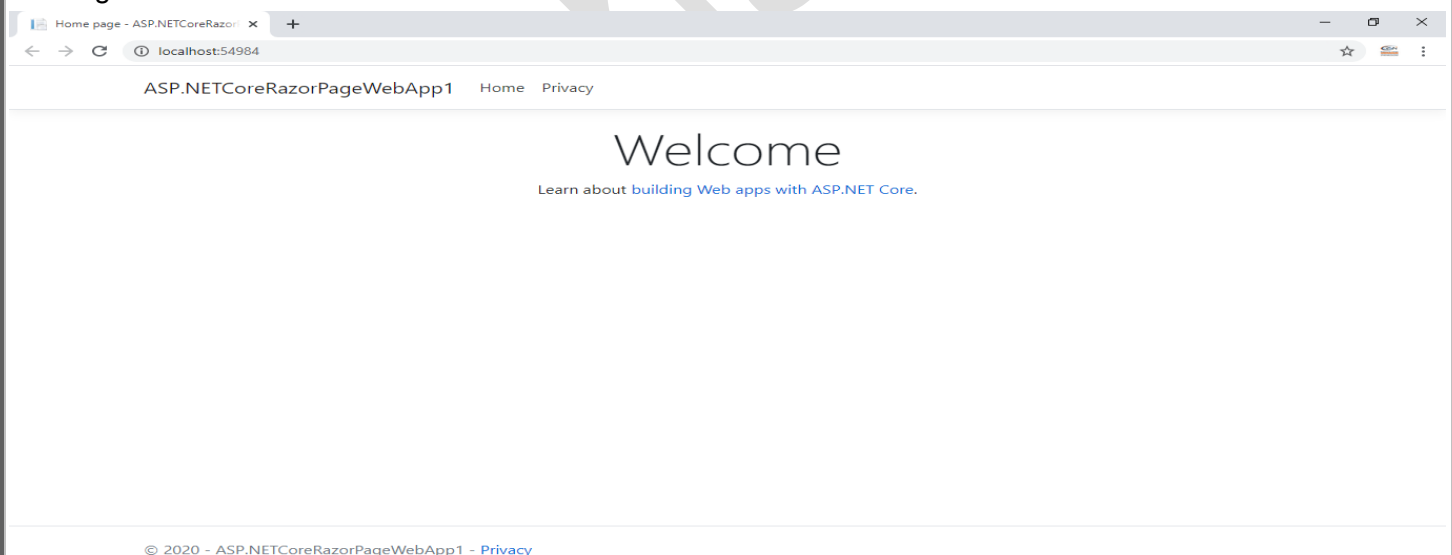
Author: Microsoft  
Source: Templates 3.1.8

Back Create

This will create a new ASP.NET Core web project in Visual Studio 2019, as shown below. Wait for some time till Visual Studio restores the packages in the project. Restoring process means Visual Studio will automatically add, update or delete configured dependencies as NuGet packages in the project.



To run this web application, click on **IIS Express** or press **F5** (with Debug) or **Ctrl + F5** (without Debug). This will open the browser and display the Welcome screen page output comes from the Index.cshhtml page under the Pages folder as shown below:



You can also see the IIS express icon on the system tray. Right click on it. You can see the ASP.NET sites currently running in your development machine.



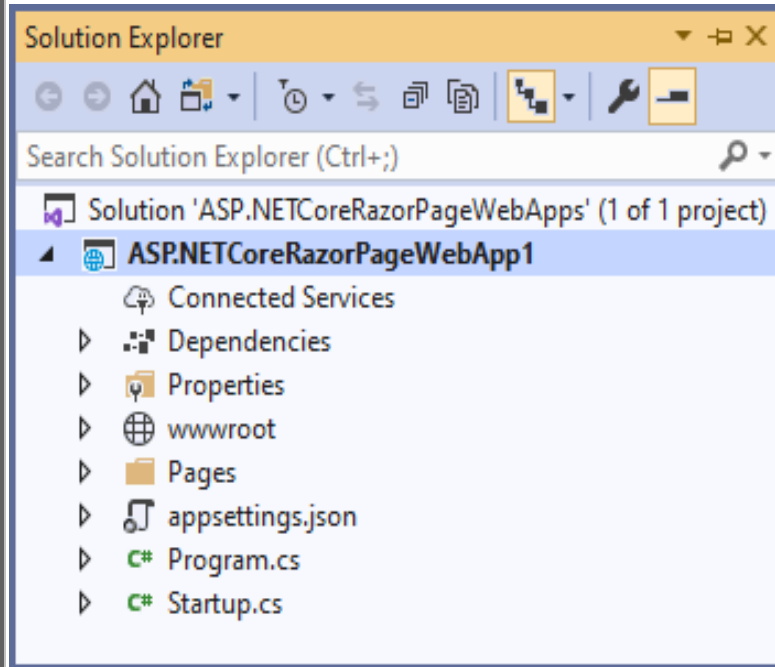
Thus, we can create a new cross-platform ASP.NET Core 3.1 application that runs on .NET Core.

Copyright © 2020 - 2021 <https://www.facebook.com/groups/RakeshSoftNetMVC/> All Rights Reserved.

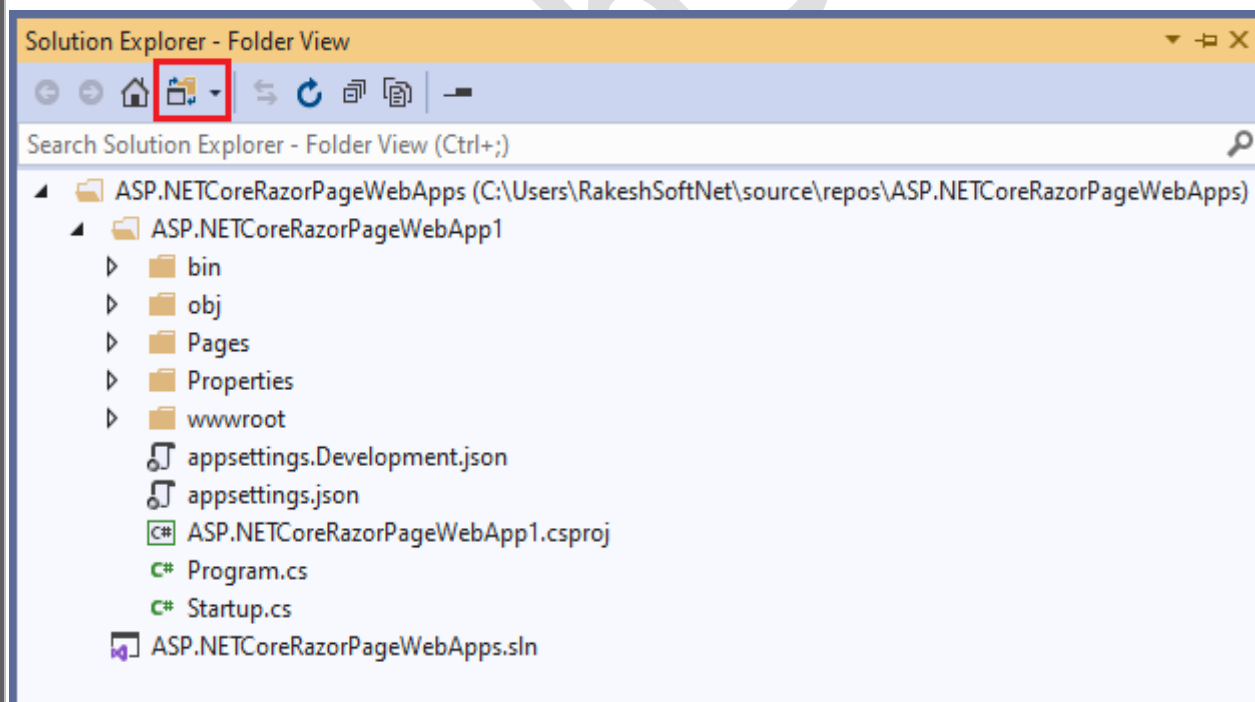


## ASP.NET Core Web Application - Project Structure:

The following is a default project structure when you create an ASP.NET Core web application in Visual Studio.



The above solution explorer displays project solution. We can change it to folder view by clicking Solution and Folders icon and selecting Folder View option. This displays the solution explorer with all project folders and files as shown below:



**Note:** ASP.NET Core Project files and folders are synchronized with physical files and folders. If you add a new file in project folder then it will directly reflect in the solution explorer. You don't need to add it in the project explicitly by right clicking on the project.

Copyright © 2020 - 2021 <https://www.facebook.com/groups/RakeshSoftNetMVC/> All Rights Reserved.

## .csproj:

ASP.NET Core 1.0 does not create .csproj file, instead, it uses .xproj and project.json files to manage the project. This has changed in ASP.NET Core 2.0. Visual Studio now uses .csproj file to manage projects. We can edit the .csproj settings by right clicking on the project and selecting "Edit Project File".

The .csproj for our project looks like as shown below:

```

ASP.NETCoreRazorPageWebApp1.csproj
1 <Project Sdk="Microsoft.NET.Sdk.Web">
2
3   <PropertyGroup>
4     <TargetFramework>netcoreapp3.1</TargetFramework>
5   </PropertyGroup>
6
7 </Project>
    
```

The csproj file includes settings related to targeted .NET Core Framework, project folders, NuGet package references etc...

## Dependencies:

The Dependencies in the ASP.NET Core project contain all the installed server side NuGet Packages. You can install all other required server side dependencies as NuGet packages from Manage NuGet Packages window or using Package Manager Console.

## Properties:

The Properties node includes launchSettings.json file which includes Visual Studio profiles of debug settings. The following is a default launchSettings.json file:

```

launchSettings.json
Schema: https://json.schemastore.org/launchsettings
1 {
2   "iisSettings": {
3     "windowsAuthentication": false,
4     "anonymousAuthentication": true,
5     "iisExpress": {
6       "applicationUrl": "http://localhost:54984",
7       "sslPort": 0
8     }
9   },
10  "profiles": {
11    "IIS Express": {
12      "commandName": "IISExpress",
13      "launchBrowser": true,
14      "environmentVariables": {
15        "ASPNETCORE_ENVIRONMENT": "Development"
16      }
17    },
18    "ASP.NETCoreRazorPageWebApp1": {
19      "commandName": "Project",
20      "launchBrowser": true,
21      "applicationUrl": "http://localhost:5000",
22      "environmentVariables": {
23        "ASPNETCORE_ENVIRONMENT": "Development"
24      }
25    }
26  }
27 }
    
```

We can also edit settings from debug tab of the project properties. Right click on the project -> select properties -> click Debug tab.

In the debug tab, select a profile which you want to edit. You may change environment variables, URL etc.

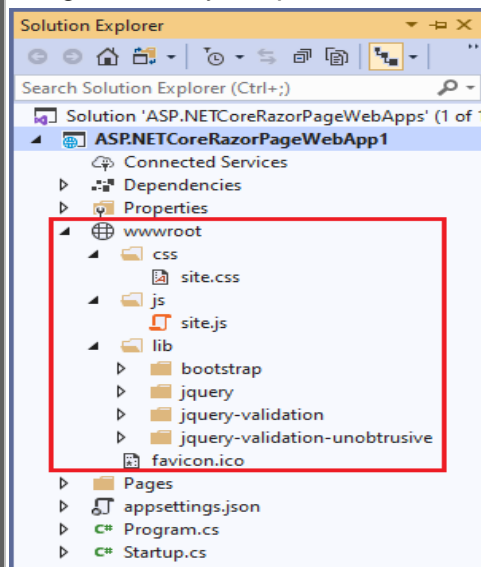
Copyright © 2020 - 2021 <https://www.facebook.com/groups/RakeshSoftNetMVC/> All Rights Reserved.

## wwwroot:

By default, the **wwwroot** folder in the ASP.NET Core project is treated as a web root folder. Static files can be stored in any folder under the web root folder and accessed with a relative path to that root.

In the standard ASP.NET application, static files can be served from the root folder of an application or any other folder under it. This has been changed in ASP.NET Core. Now, only those files that are in the web root - **wwwroot** folder can be served over an http request. All other files are blocked and cannot be served by default.

Generally, there should be separate folders for the different types of static files such as JavaScript, CSS, Images, Library scripts etc... In the **wwwroot** folder as shown below:



You can access static files with base URL and file name. For example, we can access above **site.css** file in the **css** folder by **<https://localhost:<port>/css/site.css>**

Remember you need to include a middleware for serving static files in the '**Configure()**' method of **Startup.cs** file.

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

## Rename wwwroot Folder:

You can rename **wwwroot** folder to any other name as per your choice and set it as a web root while preparing hosting environment in the **Program.cs** file.

For Example, let's rename **wwwroot** folder to "**Content**" folder. Now, call **UseWebRoot()** method to configure **Content** folder as a web root folder in the **Main()** method of **Program** class as shown below:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

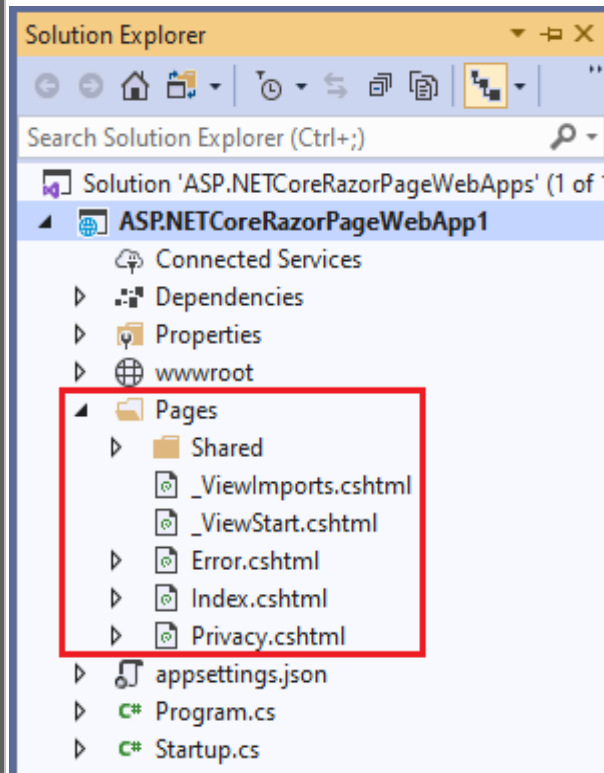
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>()
                    .UseWebRoot("Content");
            })
    }
}
```

Thus, you can rename the default web root folder **wwwroot** as per your choice.

Copyright © 2020 - 2021 <https://www.facebook.com/groups/RakeshSoftNetMVC/> All Rights Reserved.

## Pages: (Folder)

There are two ways you can create web pages in ASP.NET Core. One is the MVC approach. The other one is using the Razor pages. This folder (Pages) is created, if you choose the web application with the Razor pages template. All the razor pages will go into this folder as shown below:



## appsettings.json:

When we create an ASP.NET Core Web application with an Empty project template or Razor Pages or MVC Template or Web API Template, then the visual studio automatically creates the appsettings.json file for us.

appsettings.json file is an application configuration file used to store configuration settings such as database connections strings, any application scope global variables etc...

If you open the appsettings.json file, then you see the following default code:

```

appsettings.json*
Schema: https://json.schemastore.org/appsettings
1  {
2    "Logging": {
3      "LogLevel": {
4        "Default": "Information",
5        "Microsoft": "Warning",
6        "Microsoft.Hosting.Lifetime": "Information"
7      }
8    },
9    "AllowedHosts": "*"
10 }
    
```



## Program.cs:

ASP.NET Core web application is actually a console project which starts executing from the entry point "**public static void Main()**" in Program class where we can create a host for the web application.

The following is the Program class in ASP.NET Core 3.1:

```

Program.cs
ASP.NETCoreRazorPageWebApp1
ASP.NETCoreRazorPageWebApp1.Program
Main(string[] args)

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Threading.Tasks;
5 using Microsoft.AspNetCore.Hosting;
6 using Microsoft.Extensions.Configuration;
7 using Microsoft.Extensions.Hosting;
8 using Microsoft.Extensions.Logging;
9
10 namespace ASP.NETCoreRazorPageWebApp1
11 {
12     public class Program
13     {
14         public static void Main(string[] args)
15         {
16             CreateHostBuilder(args).Build().Run();
17         }
18
19         public static IHostBuilder CreateHostBuilder(string[] args) =>
20         {
21             Host.CreateDefaultBuilder(args)
22                 .ConfigureWebHostDefaults(webBuilder =>
23                 {
24                     webBuilder.UseStartup<Startup>();
25                 });
26         }
27     }
28 }
    
```

As you can see above, the Main() method calls method expression CreateHostBuilder() to build host with pre-configured defaults. The CreateHostBuilder expression can also be written as a method that returns IHostBuilder as below:

```

public static IHostBuilder CreateHostBuilder(string[] args)
{
    return Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
}
    
```

The Host is a static class which can be used for creating an instance of IHostBuilder with pre-configured defaults. The CreateDefaultBuilder() method creates a new instance of HostBuilder with pre-configured defaults. Internally, it configures Kestrel (Internal Web Server for ASP.NET Core), IISIntegration and other configurations.

## Setup Host in ASP.NET Core 1.x:

```

public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
        .Build();

    host.Run();
}
    
```

### Setup Host in ASP.NET Core 2.x:

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

--references
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
```

### Setup Host in ASP.NET Core 3.x:

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}

--references
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

### Startup.cs:

ASP.NET Core application must include Startup class. It is like Global.asax in the traditional .NET application. As the name suggests, it is executed first when the application starts.

The Startup class can be configured using **UseStartup<T>()** method at the time of configuring the host in the **Main()** method of **Program** class as shown below:

```
public class Program
{
    --references
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    --references
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The name "**Startup**" is by ASP.NET Core convention. However, we can give any name to the Startup class, just specify it as the generic parameter in the UseStartup<T>() method. For example, to name the Startup class as MyStartup, specify it as .UseStartup<MyStartup>().

Copyright © 2020 - 2021 <https://www.facebook.com/groups/RakeshSoftNetMVC/> All Rights Reserved.

Open Startup class in Visual Studio by clicking on the **Startup.cs** in the solution explorer. The following is a default Startup class in ASP.NET Core 3.x.

```

Startup.cs
ASP.NETCoreRazorPageWebApp1
ASP.NETCoreRazorPageWebApp1.Startup
Startup(IConfiguration configuration)

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Threading.Tasks;
5 using Microsoft.AspNetCore.Builder;
6 using Microsoft.AspNetCore.Hosting;
7 using Microsoft.Extensions.Configuration;
8 using Microsoft.Extensions.DependencyInjection;
9 using Microsoft.Extensions.Hosting;
10
11 namespace ASP.NETCoreRazorPageWebApp1
12 {
13     public class Startup
14     {
15         public Startup(IConfiguration configuration)
16         {
17             Configuration = configuration;
18         }
19
20         public IConfiguration Configuration { get; }
21
22         // This method gets called by the runtime. Use this method to add services to the container.
23         public void ConfigureServices(IServiceCollection services)
24         {
25             services.AddRazorPages();
26         }
27
28         // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
29         public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
30         {
31             if (env.IsDevelopment())
32             {
33                 app.UseDeveloperExceptionPage();
34             }
35             else
36             {
37                 app.UseExceptionHandler("/Error");
38             }
39
40             app.UseStaticFiles();
41
42             app.UseRouting();
43
44             app.UseAuthorization();
45
46             app.UseEndpoints(endpoints =>
47             {
48                 endpoints.MapRazorPages();
49             });
50         }
51     }
52 }

```

As you can see, Startup class includes two public methods: **ConfigureServices** and **Configure**.

The Startup class must include a **Configure** method and can optionally include **ConfigureService** method.

### **ConfigureServices():**

The Dependency Injection pattern is used heavily in ASP.NET Core architecture. It includes built-in IoC container to provide dependent objects using constructors.

The ConfigureServices method is a place where you can register your dependent classes with the built-in IoC container. After registering dependent class, it can be used anywhere in the application. You just need to include it in the parameter of the constructor of a class where you want to use it. The IoC container will inject it automatically.

ASP.NET Core refers dependent class as a Service. So, whenever you read "Service" then understand it as a class which is going to be used in some other class.

ConfigureServices method includes **IServiceCollection** parameter to register services to the IoC container.

```

// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
}

```

## Configure():

The Configure method is a place where you can configure application request pipeline for your application using **IApplicationBuilder** instance that is provided by the built-in IoC container.

ASP.NET Core introduced the middleware components to define a request pipeline, which will be executed on every request. You include only those middleware components which are required by your application and thus increase the performance of your application.

The following is a default **Configure** method:

*// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.*

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

As you can see, the **Configure** method includes parameters like **IApplicationBuilder**, and **IHostingEnvironment** by default. These services are framework services injected by built-in IoC container.

At run time, the **ConfigureServices** method is called before the **Configure** method. This is so that you can register your custom service with the IoC container which you may use in the Configure method.

## Bin folder:

The bin folder is the default output location for binary files generated when the application is compiled. Typically, this folder contains two sub-folders, Debug and Release. The first is for the binaries that result from compiling the application in Debug mode, and the second is where binaries generated from a compilation in Release mode are placed. Both of these folders contain a subfolder named netcoreapp[version number], where [version number] represents the version of .NET Core used to create the application. If you are using .NET Core 3.1, for example, the folder name will be netcoreapp3.1.

## Obj folder:

The *Obj* folder is used to store temporary object files and other files that are used to create the final binary during the compilation process.

Copyright © 2020 - 2021 <https://www.facebook.com/groups/RakeshSoftNetMVC/> All Rights Reserved.



**ASP.NET Core** provides a web development framework based on the **Model-View-Controller (MVC) pattern**. On top of that sits the **Razor Pages framework** for developers who are more familiar with or prefer a **page-centric development approach** to building web applications. ASP.NET Core also includes a framework for developing **REST-based web services (Web API)**. Work is also being done to include a **Web Sockets-based framework (SignalR)** which will enable real-time updating of page content initiated by the server.

### **Razor Pages in ASP.NET Core:**

#### **What Is Razor Pages?**

ASP.NET Core Razor Pages is a page-focused framework for building dynamic, data-driven web sites with clean separation of concerns. Based on the latest version of ASP.NET from Microsoft - ASP.NET Core (ASP.NET Core is the latest version of Microsoft's framework for building web-based applications. It sits on top of .NET Core, which is an open source development platform, consisting of a set of framework libraries, a software development kit (SDK) and a runtime.), Razor Pages supports cross platform development and can be deployed to Windows, UNIX and Mac operating systems.

The Razor Pages framework is lightweight and very flexible. It provides the developer with full control over rendered HTML. The framework is built on top of ASP.NET Core MVC, and is enabled by default when MVC is enabled in a .NET Core application. Razor Pages is the recommended framework for cross-platform server-side HTML generation on .NET Core. You do not need to have any knowledge or understanding of MVC to work with Razor Pages.

Razor Pages makes use of the popular C# programming language for server-side programming, and the easy-to-learn Razor templating syntax for embedding C# in HTML markup to generate content for browsers dynamically.

#### **Who should use Razor Pages?**

Razor Pages is suitable for all kinds of developers from beginners to enterprise level. It is based on a page-centric development model, offering a familiarity to web developers with experience of other page-centric frameworks such as PHP, Classic ASP, Java Server Pages, ASP.NET Web Pages and ASP.NET Web Forms. It is also relatively easy for the beginner to learn, and it includes all of the advanced features of ASP.NET Core making it just as suitable for large, scalable, team-based projects.

#### **How to get Razor Pages?**

Razor Pages is included within .NET Core from version 2.0 onwards, which is available as a free download as either an SDK (Software Development Kit) or a Runtime. The SDK includes the runtime and command line tools for creating .NET Core applications. The SDK is installed for you when you install Visual Studio 2017 Update 3 or later. The runtime is used to run .NET Core applications. The Runtime-only installation is intended for use on machines where no development takes place.

#### **Why should you use Razor Pages?**

If you want a dynamic web site, that is one where the content is regularly being added to, you have a number of options available to you. You can use a Content Management System (CMS), of which there are many to choose from including WordPress, Joomla, Drupal, and Orchard CMS and so on. Or you can hire someone to build a suitable site for you. Or you can build your own if you have an interest in, and an aptitude for programming.

If you choose to build your own, you can choose from a wide range of programming languages and frameworks. If you are a beginner, you will probably want to start with a framework and language that is easy to learn, well supported and robust. If you are considering making a career as a programmer, you probably want to know that the skills you acquire while learning your new framework will enhance your value to potential employers. In both cases, learning C# as a language and ASP.NET Core as a framework will tick those boxes. If you are a seasoned developer, the Razor Pages framework is likely to add to your skillset with the minimum amount of effort.

These are following major characteristic of Razor Pages in ASP.Net Core:

- ❖ ASP.NET Core Razor Pages is a page-centric framework for building dynamic, data-driven web application.
- ❖ Razor Pages can support cross platform development and easily can be deployed on different operating systems like Windows, UNIX and Mac.
- ❖ The Razor Pages is very lightweight and flexible. It provides full control over rendering HTML as per need, easy to work with.
- ❖ The Razor Pages framework is built on top of ASP.NET Core MVC, but you don't need to have any knowledge or understanding of MVC to work with Razor Pages.
- ❖ By default every page has a Model, actually pages are inherited from PageModel.

### Different types of Razor files:

All Razor files end with .cshtml. Most Razor files are intended to be browsable and contain a mixture of client-side and server-side code, which, when processed, results in HTML being sent to the browser. These pages are usually referred to as "**Razor Page**" (also known as "**Content Page**").

Other Razor files have a leading underscore (\_) in their file name. These files are not intended to be browsable. The leading underscore is often used for naming **Partial Page**, but three files named in this way have a particular function within a Razor Pages application.

#### \_Layout.cshtml:

The \_Layout.cshtml file acts a template for all content pages that reference it. Consistent part of a site's design are declared in the \_Layout.cshtml file. These can include the header, footer, site navigation and so on. Typically, the \_Layout.cshtml file also includes the <head> section of the page, so they also reference the common CSS style sheet files and JavaScript files. If you want to make changes to the overall design of the site, you often only need to make adjustments to the content of the \_Layout.cshtml file.

#### \_ViewStart.cshtml:

The \_ViewStart.cshtml file contains code that executes after the code in any content page in the same folder or any child folders. It provides a convenient location to specify the layout file for all content pages that are affected by it, and that is typically what you see in the \_ViewStart.cshtml file that comes with any Razor Pages (or MVC) template.

#### \_ViewImports.cshtml:

The purpose of the \_ViewImports.cshtml file is to provide a mechanism to make directives available to Razor pages globally so that you don't have to add them to pages individually.

The default Razor Pages template includes a \_ViewImports.cshtml file in the Pages folder - the root folder for Razor pages. All Razor pages in the folder hierarchy will be affected by the directives set in the \_ViewImports.cshtml file.

### Razor Page:

All Razor Page files end with .cshtml that are intended to be browsable and contain a mixture of client-side and server-side code, which, when processed, results in HTML being sent to the browser. **Razor Page** is also known as **Content Page**.

### Content Page:

For a file to act as a Razor content page, it must have three characteristics:

- It cannot have a leading underscore in its file name
- The file extension is .cshtml
- The first line in the file is @page

Placing the @page directive as the first line of code is critical. If this is not done, the file will not be seen as a Razor page, and will not be found if you try to browse to it. There can be empty space before the @page directive, but there cannot be any other characters, even an empty code block. The only other content permitted on the same line as the @page directive is a route template.

Content pages can have a layout file specified, but this is not mandatory. They can optionally include code blocks, HTML, JavaScript and inline Razor code.

Content pages are largely comprised of HTML, but they also include Razor syntax which enables the inclusion of executable C# code within the content. The C# code is executed on the server, and typically results in dynamic content being included within the response sent to the browser.

### Understanding Razor Pages

Razor pages are in the **Pages** folder in the root web application folder.

For Example: Index.cshtml

```

1  @page
2  @model IndexModel
3  @{
4      ViewData["Title"] = "Home page";
5  }
6
7  <div class="text-center">
8      <h1 class="display-4">Welcome</h1>
9      <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>
10 </div>

```

**Index.cshtml** is the display template and has the extension **.cshtml**.

It is much like a razor view file in MVC.

**@page** directive specifies it's a razor page.

**@model** directive specifies the model. The model is the corresponding PageModel class which is shown below.

**Index.cshtml.cs:**

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Mvc;
6  using Microsoft.AspNetCore.Mvc.RazorPages;
7  using Microsoft.Extensions.Logging;
8
9  namespace ASP.NETCoreRazorPageWebApp1.Pages
10 {
11     -- references
12     public class IndexModel : PageModel
13     {
14         private readonly ILogger<IndexModel> _logger;
15
16         -- references
17         public IndexModel(ILogger<IndexModel> logger)
18         {
19             _logger = logger;
20         }
21
22         -- references
23         public void OnGet()
24         {
25
26         }
27     }
28 }

```



**Index.cshtml.cs** is the corresponding **PageModel** class.

It has the same name as the display template and ends with **.cs** extension. CS because the programming language is C#.

The class in this file is the model for the display template. It derives from the **PageModel** class.

Just like MVC, razor pages also support dependency injection.

The built-in **ILogger** service is injected using the constructor.

This **ILogger** service enables us to log to several different logging destinations.

In addition to Dependency Injection and Logging, other asp.net core features like configuration sources, model binding, model validation etc. are also supported by razor pages.

### Razor Pages 'Hello World' Example:

The **PageModel** class (**Index.cshtml.cs**):

```
public class IndexModel : PageModel
{
    - references
    public string Message { get; set; }

    - references
    public void OnGet()
    {
        Message = "Hello World of ASP.NET Core 3.1";
    }
}
```

Razor pages use **public properties** to expose data to the display templates.

The public property **Message** is available in the display template.

In addition to these public properties which carry data to the display template, the **PageModel** class also includes methods like **OnGet()** and **OnPost()**.

These are the methods that respond to HTTP **GET** and **POST** requests respectively.

The display template (**Index.cshtml**):

```
1 @page
2 @model IndexModel
3 @{
4     ViewData["Title"] = "Home page";
5 }
6
7 <div class="text-center">
8     <h1 class="display-4">@Model.Message</h1>
9 </div>
```

Notice, the public property **Message** is available in the display template through **@Model.Message**

When we run this project and navigate to <http://localhost:54984/Index> we see the Index page in the browser. Similarly if you have a Privacy.chnl razor page and when you navigate to <https://localhost:54984/Privacy>, you will see the Privacy page in the browser. The extension .cshtml is not required in the URL.



## ASP.NET Webforms vs Razor Pages:

ASP.NET Core Razor Pages framework is a new technology to build page-focused web applications quicker and more efficiently with clean separation of concerns. Razor pages are introduced in .NET Core 2.0. It is lightweight, flexible and provides the developer the full control over the rendered HTML.



In some respects, **razor pages** are similar to the classic **ASP.NET Webforms** framework. In ASP.NET Webforms, we have an ASPX page and a code-behind class. The ASPX page contains the HTML and controls the visual part. The code-behind class contains the server-side C# or Visual Basic code that handles the page events. For example, if you have a Web Form with name **WebForm1**. It's actually a pair of files - **WebForm1.aspx** (the display template) and **WebForm1.aspx.cs** (the code-behind class).

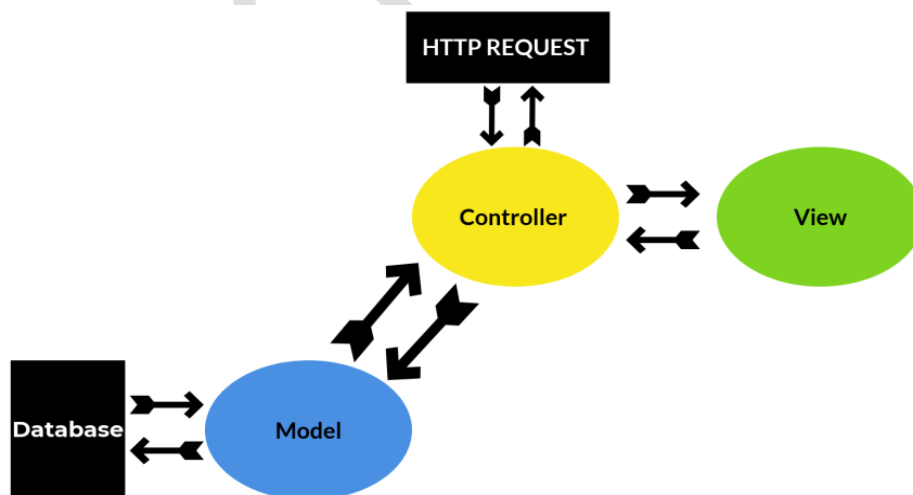
**Similarly, each razor page is also a pair of files - .cshtml and .cshtml.cs**

.cshtml - Is the display template. So it contains the HTML and razor syntax.

.cshtml.cs - Contains the server side C# code that handles the page events and provides the data the template needs.

## ASP.NET Core MVC vs Razor Pages:

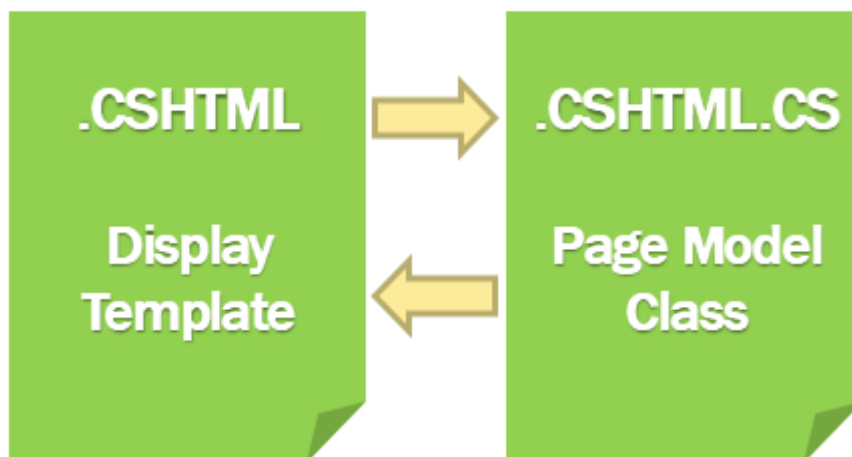
With the MVC design pattern we have the Model, View and Controller. It is the Controller in the MVC design pattern that receives the request and handles it. The controller creates the model. The model has the classes that describe the data. In addition to the data, Model also contains the logic to retrieve data from the underlying data source such as a database. In addition to creating the Model, the controller also selects a View and passes the Model object to the View. The view contains the presentation logic to display the Model data provided to it by the Controller.



In MVC, in addition to Model, View and Controller, we also have Actions and ViewModels. If we are building a fairly complex portal, chances are we may end up with controllers that work with many different dependencies and view models and return many different views. In short we may end up with large controllers with many actions that are not related to each other in any way. This not only results in unnecessary complexity but also violate the fundamental principles of programming like **Single Responsibility Principle** and **Open/Closed Principle**.

Copyright © 2020 - 2021 <https://www.facebook.com/groups/RakeshSoftNetMVC/> All Rights Reserved.

On the other hand a razor page is just a pair of files - a display template and the corresponding PageModel class. As the name implies the display template contains the HTML. The PageModel class contains the server side code and combines the responsibilities of a Controller and a ViewModel. Everything we put in the PageModel class is related to the Page. So, unlike controllers in MVC, bloating the PageModel class with unrelated methods is almost impossible. Since the PageModel class and the display template are at one place and closely related to each other, building individual pages with razor pages is fairly straightforward while still using all the architectural features of ASP.NET Core MVC like dependency injection, middleware components, configuration system, model binding, validation etc.



So the recommendation from Microsoft is to use razor pages if we are building a Web UI (Web Pages) and ASP.NET Core MVC if we are building a Web API.

Whether you use ASP.NET Core MVC or Razor Pages for building a web application, there's no difference from performance standpoint.

It's also possible to combine both the patterns (i.e. ASP.NET Core MVC and Razor Pages) in a given single ASP.NET Core Web Application.

### Create Razor Page:

ASP.NET Core Razor Pages framework provides two ways for structuring the code of our Razor Content Pages.

- The first way utilizes the **Single File Approach (Code-Inline Model)**. This technique should be familiar to Classic ASP developers because all the code is contained within a single .cshtml file.
- The second way uses ASP.NET Core's **PageModel File (Code-Behind Model)**, which allows for code separation of the page's business logic from its presentation logic. In this model, the presentation logic for the page is stored in .cshtml file, whereas the business logic piece is stored in a separate class file: .cshtml.cs. Using the code-behind model is considered the best practice because it provides a clean model in separation of pure UI elements from code that manipulates these elements. It is also seen as a better means in maintaining code.

### Single File Approach (Code-Inline Model):

**Code-Inline** refers to the code that is written inside a Razor Content Page that has an extension of .cshtml. It allows the code to be written along with the HTML source code using a **Razor code block** i.e. '@{ }' and **@functions** block.

Although this approach is not recommended, it is possible to develop Razor Page applications that rely solely on content pages.

The following example features an approach that is most like that which is familiar to developers with a scripting background, like PHP or Classic ASP:

In **Visual Studio 2019** updated version, Right click on **Pages** folder => **Add** => **New Item** => **Razor View – Empty** and name it **Example.cshtml**

Remove the default template of created page and specify it as following to make a razor content page:

```
@page
@{
}
```

**Creating Razor Page using the .NET Core command-line interface (CLI):**

**Command:** dotnet new page

**Options:**

-n|--name: The name of the page.

-o|--output: Location to place the generated page.

**A Razor page with or without a page model:**

**Options:**

-na|--namespace: namespace for the generated code  
string - Optional  
Default: MyApp.Namespace

-np|--no-pagemodel: create page without a PageModel  
bool - Optional  
Default: false / (\*) true

**Creating a Razor Content Page named 'Example' in 'Pages' folder without a page model file called Single File Approach (Code-Inline Model):**

➤ dotnet new page --name Example --no-pagemodel --output Pages

It creates a razor content page named **Example.cshtml** (Single File Approach) in **Pages** folder of an application with the following template by default:

```
@page
@{
}
```

**Note:** @page directive specifies it's a razor page which is must to be defined in any razor page.

**Now below is an example of a razor content page (Single File Approach) without attaching to a Layout page:**

```
Example.cshtml* -P X
1  @page
2  @{
3      Layout = null;
4  }
5  <!DOCTYPE html>
6  <html>
7  <head>
8      <meta name="viewport" content="width=device-width, initial-scale=1.0">
9      <title>Example</title>
10 </head>
11 <body>
12     @{
13         var name = string.Empty;
14         if (Request.HasFormContentType)
15         {
16             name = Request.Form["name"];
17         }
18     }
19     <div style="margin-top:10px;">
20         <form method="post">
21             <b>Name: </b><input name="name" value="@name" />
22             <input type="submit" />
23         </form>
24     </div>
25     <div>
26         <@if (!string.IsNullOrEmpty(name))>
27         {
28             <p>Hello @name!</p>
29         }
30     </div>
31 </body>
32 </html>
```

Copyright © 2020 - 2021 <https://www.facebook.com/groups/RakeshSoftNetMVC/> All Rights Reserved.



The Razor content page requires the **@page** directive at the top of the file. The **HasFormContentType** property is used to determine whether a form has been posted and the **Request.Form** collection is referenced within a Razor code block with the relevant value within it assigned to the **name** variable.

The Razor code block is denoted by and opening **@{** and is terminated with a closing **}**. The content within the block is standard C# code.

Single control structures do not need a code block. You can simply prefix them with the **@** sign. This is illustrated by **if** block in the preceding example.

To render the value of a C# variable or expression, you prefix it with **@** sign as shown with the **name** variable within **if** block.

### Functions Blocks:

The next example results in the same functionality as the previous example, but it uses an **@functions** block to declare a public property which is decorated with the **BindProperty** attribute, ensuring that the property takes part in **model binding**, removing the need to manually assign form values to variables.

```
Example.cshtml* X
1  @page
2  @{
3      Layout = null;
4  }
5  <!DOCTYPE html>
6  <html>
7  <head>
8      <meta name="viewport" content="width=device-width, initial-scale=1.0">
9      <title>Example</title>
10 </head>
11 <body>
12     @functions {
13         [BindProperty]
14         public string Name { get; set; }
15     }
16     <div style="margin-top:10px;">
17         <form method="post">
18             <b>Name: </b><input name="name" value="@Name" />
19             <input type="submit" />
20         </form>
21     </div>
22     <div>
23         @if (!string.IsNullOrEmpty(Name))
24         {
25             <p>Hello @Name!</p>
26         }
27     </div>
28 </body>
29 </html>
```

This approach is an improvement on the previous in that it makes use of strong typing.

Copyright © 2020 - 2021 <https://www.facebook.com/groups/RakeshSoftNetMVC/> All Rights Reserved.



You can also use the **@functions** block to declare local methods that include HTML to act as display helpers for the current page. You might do this if a page has multiple blocks of code that included HTML and require similar formatting to be applied. This is only possible in ASP.NET Core 3.x

In the following examples, a method is declared that formats **DateTime** values:

```
@functions {
    void DisplayDate(DateTime dt)
    {
        <span>@dt.ToString("dddd, dd MMMM yyyy")</span>
    }
}

@{
    DisplayDate(DateTime.Now);
}
```

This is useful if you ever need to modify the output, since you only need to make changes in one place. It is possible to do something similar in **Razor Pages 2.x**, but the method must be declared in a **code block** as follows:

```
@using Microsoft.AspNetCore.Html

@{
    Func<DateTime, IHtmlContent> DisplayDate = @<span>@item.ToString("dddd, dd MMMM yyyy")</span>;
}

@DisplayDate(DateTime.Now)
```

The **item** variable is a special one that refers to the first parameter passed in to the method. Usage is the same, but you will need to add a **using** directive for **Microsoft.AspNetCore.Html**.

### Razor Content Page with Page Model Class along with their Properties & Handler Methods in Single File Approach (Code-Inline Model):

Example.cshtml:

```
Example.cshtml* -> X
1  @page
2  @model ExampleModel
3  @using Microsoft.AspNetCore.Mvc.RazorPages
4  @{
5      Layout = null;
6  }
7  <!DOCTYPE html>
8  <html>
9  <head>
10     <meta name="viewport" content="width=device-width, initial-scale=1.0">
11     <title>Example</title>
12 </head>
13 <body>
14     @functions{
15         public class ExampleModel : PageModel
16         {
17             public string Message { get; set; }
18
19             public void OnGet()
20             {
21                 Message = "Welcome to ASP.NET Core Examples !!!";
22             }
23         }
24     }
25     <div style="margin:10px;text-align:center">
26         <h1>@Model.Message</h1>
27     </div>
28 </body>
29 </html>
```

**Note:** We can see that all the business logic is encapsulated in the `@functions{ }` block and **razor code block** `@{ }`. The nice feature of the code-inline model is that the business logic and the presentation logic are contained within the same file. Some developers find that having everything in a single viewable instance makes working with the ASP.NET Core razor content page easier. Another great thing is that it supports IntelliSense and error highlighting too when working with the inline coding model which makes easy for developer to work with this approach.

### Page Model File Approach (Code-Behind Model):

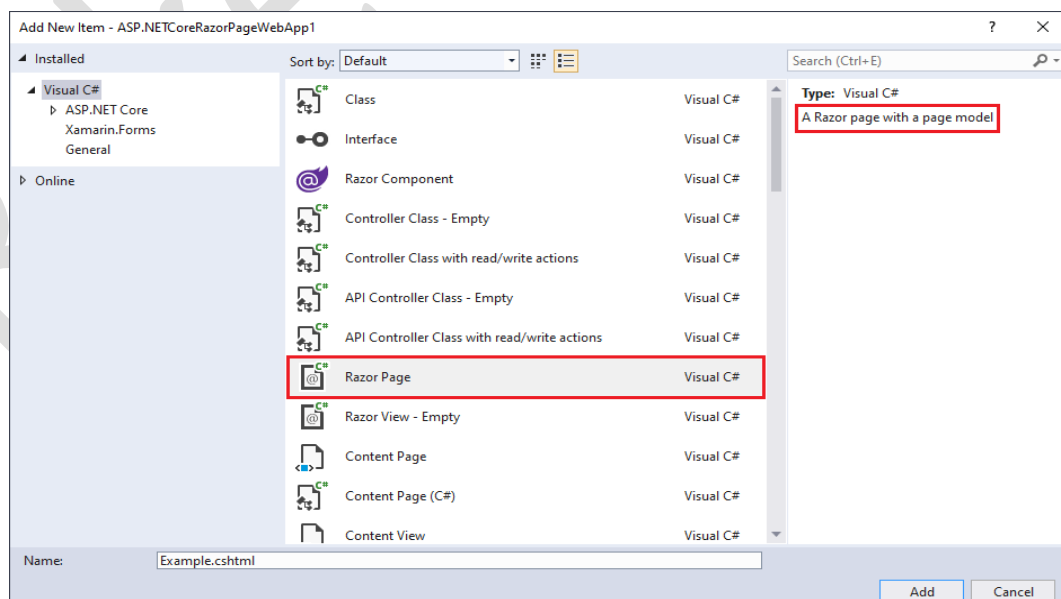
Code Behind refers to the code for an ASP.NET Core razor content page that is written in a separate class file (Page Model Class File) that can have the extension of `.cshtml.cs`. So, you can write the code in a separate `.cs` code file for each `.cshtml` razor content page. This allows a clean separation of our UI from our Code.

This is recommended way to develop Razor Pages applications which minimise the amount of server-side code in the razor content page. Any code relating to the processing of user input or data should be placed in Page Model file, which share a one-to-one mapping with their associated content page. They even share the same file name, though with an additional `.cs` on the end to denote the fact that they are actually C# class files.

The main purpose of the Razor Pages Page Model class is to provide clear separation between the UI layers (the `.cshtml` view file) and processing logic for the page. There are a number of reasons why this separation is beneficial:

- It reduces the complexity of the UI layer making it easier to maintain.
- It facilitates automated unit testing.
- It enables greater flexibility for teams in that one member can work on the view while another can work on the processing logic.
- It encourages smaller, reusable units of code for specific purposes, which aids maintenance and scalability (i.e. the ease with which the application's code base can be added to in order to cater for additional future requirements).

To create a new razor content page in our project that uses the code-behind model, choose the '**Razor Page**' (A Razor page with a page model) option when adding a new item in the '**Pages**' folder of the project.



The Page Model class is declared in a separate class file - a file with a .cs extension. Page Model classes are placed in the same namespace as the page, which by default follows the pattern **<default namespace (project name)>.<folder name>** and are named after the page file, with "Model" as a suffix. A Page Model class for **Example.cshtml** will be named **ExampleModel** and will be generated in a file named **Example.cshtml.cs**.

In terms of its features and functionality, the Page Model class is a combination of a **Controller** and a **ViewModel**.

### Controllers:

Controllers feature in a number of design and architectural patterns concerned with the presentation layer of an application. They are found in the Model-View-Controller (MVC) pattern, Front Controller, Application Controller and Page Controller patterns. A Razor Page is an implementation of the Page Controller pattern.

The Page Controller pattern is characterised by the fact that there is a one-to-one mapping between pages and their controllers. The role of the controller in the Page Controller pattern is to accept input from the page request, to ensure that any requested operations on the model (data) are applied, and then to determine the correct view to use for the resulting page.

### ViewModels:

A ViewModel is an implementation of the Presentation Model design pattern. It is a self-contained class that represents the data and behaviour of a specific "view" or page. The ViewModel pattern is used extensively in MVC application development, where it mainly represents data, but typically little behaviour. In Razor Pages, the Page Model is also the ViewModel. For this reason, Razor Pages is often described as implementing the **MVVM** (Model, View ViewModel) pattern.

### Default Template:

The following code shows the content that is generated for each file when you use the Razor Page (**A Razor page with a page model**) option to add a new page to a Razor Pages application:

### Example.cshtml:

```

Example.cshtml*  X
1  @page
2  @model ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel
3  @{
4  }
    
```

### Example.cshtml.cs:

```

Example.cshtml.cs*  X
ASP.NETCoreRazorPageWebApp1  ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel  OnGet()
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Mvc;
6  using Microsoft.AspNetCore.Mvc.RazorPages;
7
8  namespace ASP.NETCoreRazorPageWebApp1.Pages
9  {
10     public class ExampleModel : PageModel
11     {
12         public void OnGet()
13         {
14         }
15     }
16 }
    
```



The Page Model class is made available to the view file via the **@model** directive. The generated Page Model class inherits from **Microsoft.AspNetCore.Mvc.RazorPages.PageModel**, which has a number of properties that enable you to work with various items associated with the HTTP request such as the HttpContext, Request, Response, ViewData, ModelState and TempData. It also includes a range of methods that enable you to specify the type of the resulting response, including another Razor Page, a file, some JSON, a string or a redirection to another resource.

### Request Processing:

Request processing in a Page Model is performed within handler methods which are similar to Action methods on an ASP.NET MVC controller. By convention, handler method selection is based on matching the HTTP verb that was used for the request with the name of the handler method using the pattern On<verb> with Async appended optionally to denote that the method is intended to run asynchronously.

The OnGet or OnGetAsync method is selected for GET requests and the OnPost or OnPostAsync method is selected for POST requests. If you want to create a fully RESTful APIs application, all other HTTP verbs (PUT, DELETE etc.) are also supported.

The matching algorithm only looks at the name of the method. Neither the return type nor any parameters are taken into consideration. The only other requirement for a handler method is that it must be public.

Named handler methods allow you to specify a number of alternative methods for a specific verb. You might want to use these if your page contains multiple forms, each requiring a different process to be executed.

### Properties and Methods:

The properties and methods that you apply to the PageModel class are available on the **Model** property in the Razor Page. The properties can be simple ones like string, int, DateTime etc., or they can be complex classes, or a combination. If your page is designed for adding new products to a database, you might have the following range of properties:

```
public string Name { get; set; }
public SelectList Categories { get; set; }
public int CategoryId { get; set; }
```

You can also add properties or methods to the Page Model that take care of formatting values for display, to minimise the amount of code you add to the Razor Page. The following example shows how you can use a property to format the result of a calculation:

```
public List<OrderItems> Orders { get; set; }
public string TotalRevenue => Orders.Sum(o => o.NetPrice).ToString("f");
```

Then your Razor Page will only need **@Model.TotalRevenue** to display the total of all sales to two decimal places, negating the need for LINQ calculations in the HTML.

The properties that you add to the Page Model also enable you to develop a form in a strongly typed manner, which reduces the potential for runtime errors. They are available to the 'for' attribute of the **Label** and **Input** tag helpers, for example.



The following code shows the **Example.cshtml** file adapted to work with a Page Model:

**Example.cshtml** (View File):

```

Example.cshtml - X
1  @page
2  @model ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel
3  @{
4      Layout = null;
5  }
6  <!DOCTYPE html>
7  <html>
8  <head>
9      <meta charset="utf-8" />
10     <title>Example Page</title>
11 </head>
12 <body>
13     <div style="margin-top:10px;">
14         <form method="post">
15             <div>Name: <input asp-for="Name" /></div>
16             <br />
17             <div><input type="submit" /></div>
18         </form>
19         @if (!string.IsNullOrEmpty(Model.Name))
20         {
21             <p>Hello @Model.Name!</p>
22         }
23     </div>
24 </body>
25 </html>
    
```

**Example.cshtml.cs** (Page Model Class File):

```

Example.cshtml.cs - X
ASP.NETCoreRazorPageWebApp1
ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel
OnGet()

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Mvc;
6  using Microsoft.AspNetCore.Mvc.RazorPages;
7  using Microsoft.AspNetCore.Mvc.Rendering;
8
9  namespace ASP.NETCoreRazorPageWebApp1.Pages
10 {
11     public class ExampleModel : PageModel
12     {
13         [BindProperty]
14         public string Name { get; set; }
15
16         public void OnGet()
17         {
18         }
19     }
20 }
    
```

The Page Model class has a single property defined as in the earlier example, and it is decorated with the **BindProperty** attribute. The content page no longer has the **@functions** block, but it now includes a **@model** directive, specifying that the **ExampleModel** is the model for the page. This also enables Tag helpers in the page, further taking advantage of compile-time type checking.

The default projects generate content pages paired with Page Model files. This is the recommended approach. However, it is also useful to know how to work with content pages without a Page Model for circumstances where they are not needed.

**Output:**



## Razor Page Methods in ASP.NET Core (Handler Methods in ASP.NET Core Razor Pages):

Handler methods in Razor Pages are methods that are automatically executed as a result of a request. The Razor Pages framework uses a naming convention to select the appropriate handler method to execute. The default convention works by matching the HTTP verb used for the request to the name of the method, which is prefixed with "On": **OnGet()**, **OnPost()** etc.

```
0 references
public void OnGet()
{
    //.....
}

0 references
public void OnPost()
{
    //.....
}
```

Handler methods also have optional asynchronous equivalents: **OnGetAsync()**, **OnPostAsync()** etc. You do not need to add the **Async** suffix. The option is provided for developers who prefer to use the **Async** suffix on methods that contain asynchronous code.

As far as the Razor Pages framework is concerned, **OnGet** and **OnGetAsync** are the same handler. You cannot have both in the same page. If you do, the framework will raise an exception:

```
0 references
public void OnGet()
{
    //.....
}

0 references
public void OnGetAsync()
{
    //.....
}
```

### An unhandled exception occurred while processing the request.

InvalidOperationException: Multiple handlers matched. The following handlers matched route data and had all constraints satisfied:

Void OnGet(), Void OnGetAsync()

Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.DefaultPageHandlerMethodSelector.Select(PageContext context)

Parameters play no part in disambiguating between handlers based on the same verb, despite the fact that the compiler will allow it. Therefore the same exception will be raised even if the **OnGet** method takes parameters and the **OnGetAsync** method doesn't.

Handler methods must be **public** and can have any return type, although typically, they are most likely to have a return type of **void** (or **Task** if asynchronous) or an **action result**.

The following example illustrates basic usage in a Page Model file:

```
Example.cshtml.cs - ASP.NETCoreRazorPageWebApp1 - ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel - OnPost()
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Threading.Tasks;
5 using Microsoft.AspNetCore.Mvc;
6 using Microsoft.AspNetCore.Mvc.RazorPages;
7
8 namespace ASP.NETCoreRazorPageWebApp1.Pages
9 {
10     public class ExampleModel : PageModel
11     {
12         public string Message { get; set; }
13         public void OnGet()
14         {
15             Message = "Get Method Used !!!";
16         }
17         public void OnPost()
18         {
19             Message = "Post Method Used !!!";
20         }
21     }
22 }
```

The HTML part of the page includes a form that uses the **POST** method and a hyperlink, which initiates a **GET** request:

```
Example.cshtml
1 @page
2 @model ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel
3 @{
4     Layout = null;
5 }
6 <!DOCTYPE html>
7 <html>
8 <head>
9     <meta charset="utf-8" />
10    <title>Example Page</title>
11 </head>
12 <body>
13     <div style="margin-top:10px;">
14         <h3>@Model.Message</h3>
15         <form method="post">
16             <button type="submit">Click to Post</button>
17         </form>
18         <p><a asp-page="/Example">Click to Get</a></p>
19     </div>
20 </body>
21 </html>
```

When the page is first navigated to, the **"Get Method Used!!!"** message is displayed because the HTTP **GET** verb was used for the request, firing the **OnGet()** handler.

When the **"Click to Post"** button is pressed, the form is posted and the **OnPost()** handler fires, resulting in the **"Post Method Used!!!"** message being displayed.

Clicking the hyperlink results in the **"Get Method Used!!!"** message being displayed once more.

### Named Handler Methods:

Razor Pages includes a feature called **"named handler methods"**. This feature enables you to specify multiple methods that can be executed for a single verb. You might want to do this if your page features multiple forms, each one responsible for a different outcome, for example.

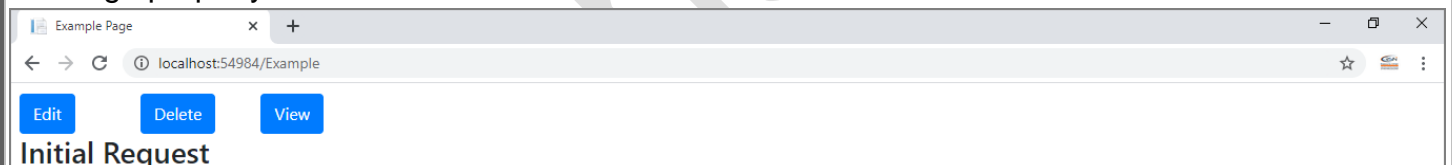
The following code shows a collection of named handler methods declared in the Page Model class of a Razor page (although they can also be placed in a **@functions** block at the top of a Razor page if you are using single file approach [code-inline model]):

```
Example.cshtml.cs
1 using Microsoft.AspNetCore.Mvc.RazorPages;
2
3 namespace ASP.NETCoreRazorPageWebApp1.Pages
4 {
5     public class ExampleModel : PageModel
6     {
7         public string Message { get; set; }
8         public void OnGet()
9         {
10             Message = "Initial Request";
11         }
12         public void OnPost()
13         {
14             Message = "Form Posted";
15         }
16         public void OnPostEdit()
17         {
18             Message = "Edit handler fired";
19         }
20         public void OnPostDelete()
21         {
22             Message = "Delete handler fired";
23         }
24         public void OnPostView()
25         {
26             Message = "View handler fired";
27         }
28     }
29 }
```

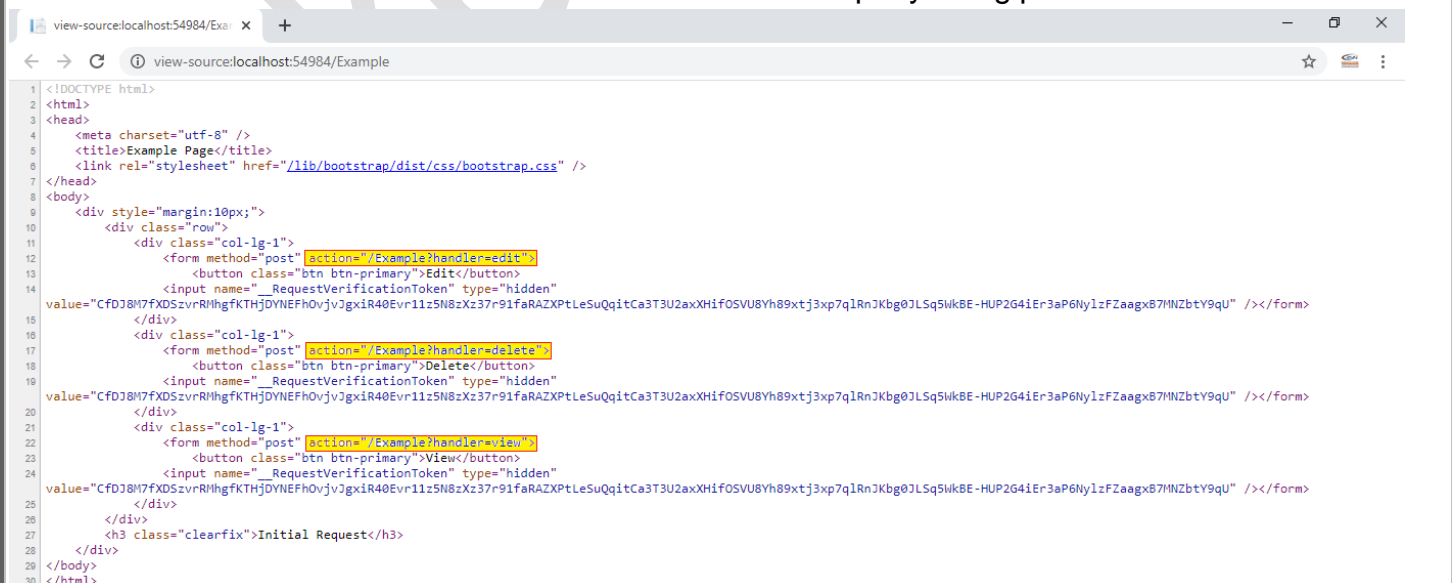
The name of the method is appended to "OnPost" or "OnGet", depending on whether the handler should be called as a result of a **POST** or **GET** request. The next step is to associate a specific form action with a named handler. This is achieved by setting the **asp-page-handler** attribute value for a form tag helper:

```
Example.cshtml
1 @page
2 @model ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel
3 @{
4     Layout = null;
5 }
6 <!DOCTYPE html>
7 <html>
8 <head>
9     <meta charset="utf-8" />
10    <title>Example Page</title>
11    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
12 </head>
13 <body>
14     <div style="margin:10px;">
15         <div class="row">
16             <div class="col-lg-1">
17                 <form asp-page-handler="edit" method="post">
18                     <button class="btn btn-primary">Edit</button>
19                 </form>
20             </div>
21             <div class="col-lg-1">
22                 <form asp-page-handler="delete" method="post">
23                     <button class="btn btn-primary">Delete</button>
24                 </form>
25             </div>
26             <div class="col-lg-1">
27                 <form asp-page-handler="view" method="post">
28                     <button class="btn btn-primary">View</button>
29                 </form>
30             </div>
31         </div>
32         <h3 class="clearfix">@Model.Message</h3>
33     </div>
34 </body>
35 </html>
```

The code above renders as three buttons, each in their own form along with the default value for the Message property:

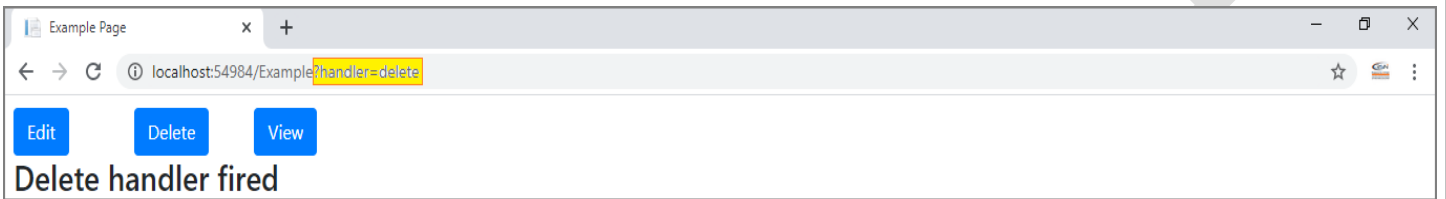
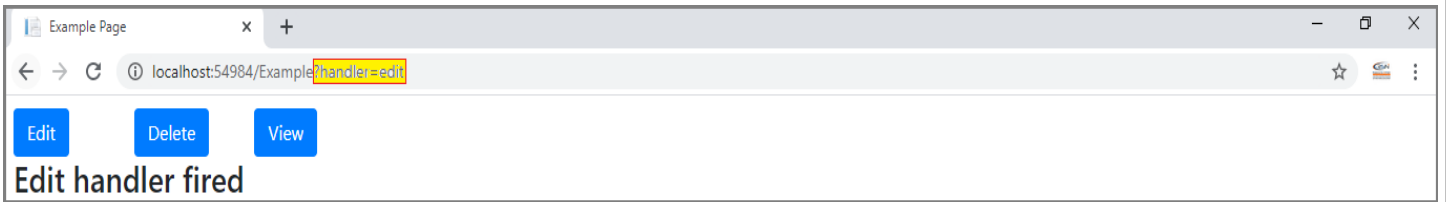


The name of the handler is added to the form's action as a query string parameter:





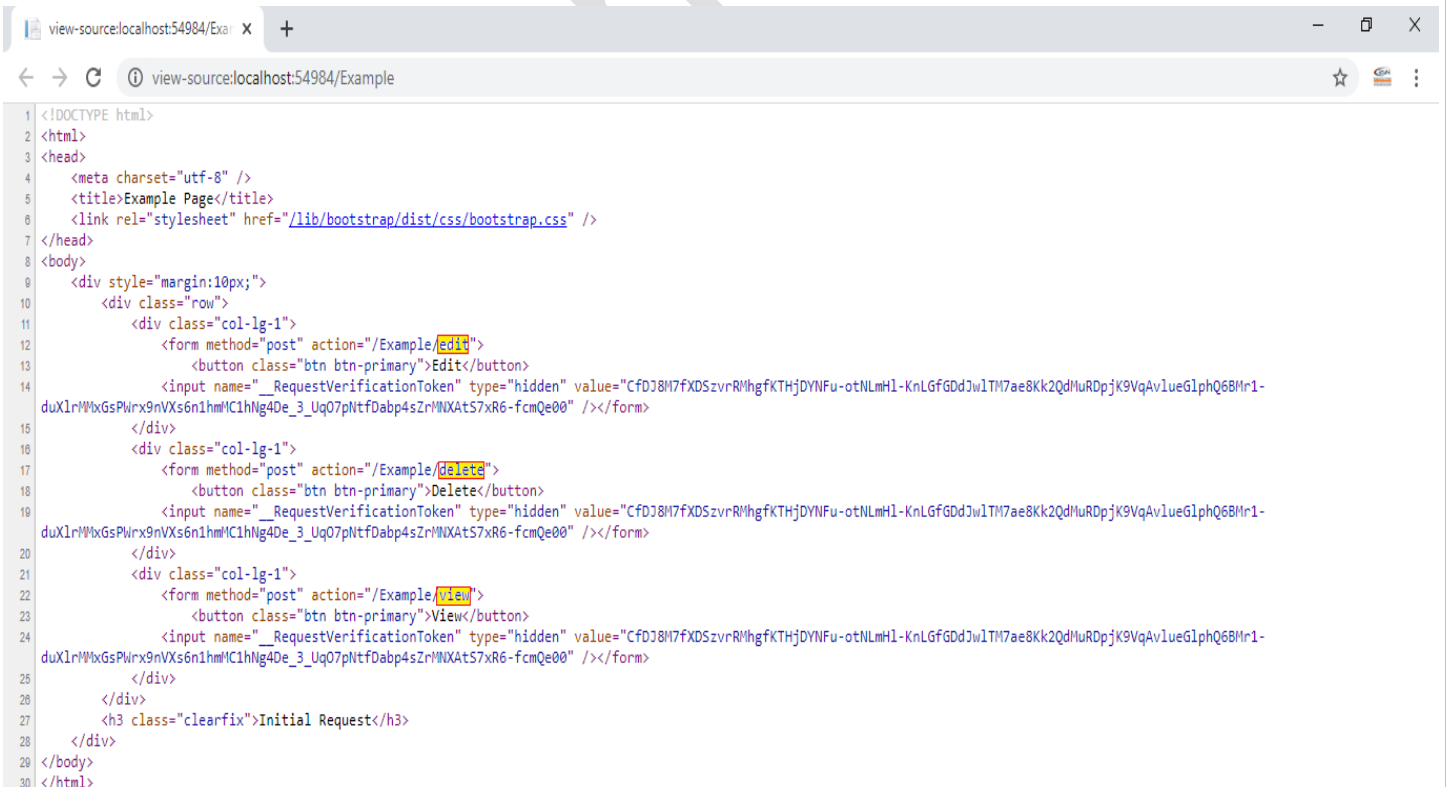
As you click each button, the code in the handler associated with the query string value is executed, changing the message each time.



If you prefer not to have query string values representing the handler's name in the URL, you can use **routing** to add an optional route value for "handler" as part of the route template in the **@page** directive:

```
Example.cshtml 1 | @page "{handler?}"
```

The name of the handler is then appended to the URL:



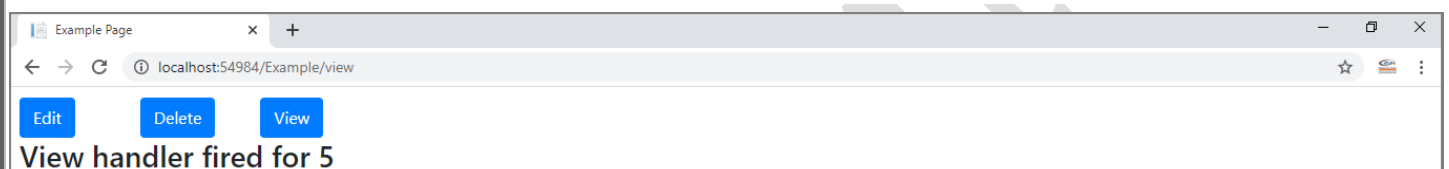
## Parameters in Handler Methods:

Handler methods can be designed to accept parameters:

```
public void OnPostView(int id)
{
    Message = $"View handler fired for {id}";
}
```

The parameter name must match a form field name for it to be automatically bound to the value:

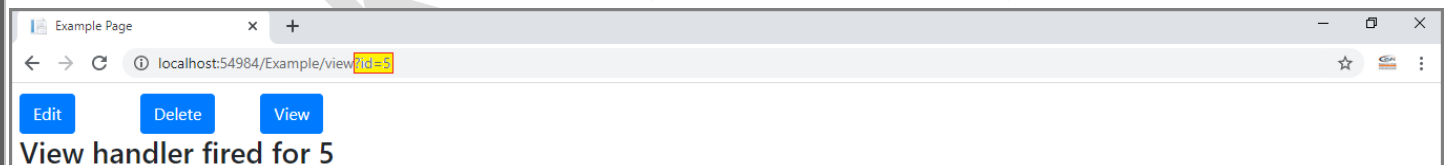
```
<div class="col-lg-1">
    <form asp-page-handler="view" method="post">
        <button class="btn btn-primary">View</button>
        <input type="hidden" name="id" value="5" />
    </form>
</div>
```



Alternatively, you can use the **form tag helper's asp-route** attribute to pass parameter values as part of the URL, either as a query string value or as route data:

```
<div class="col-lg-1">
    <form asp-page-handler="view" asp-route-id="5" method="post">
        <button class="btn btn-primary">View</button>
    </form>
</div>
```

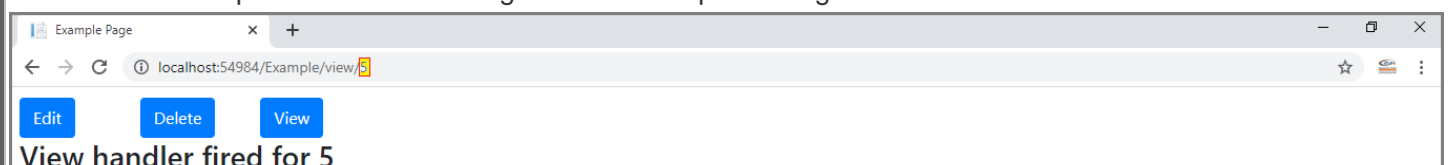
You append the name of the parameter to the **asp-route** attribute (in this case "id") and then provide a value. This will result in the parameter being passed as a query string value:



Or you can extend the route definition for the page to account for an optional parameter:

```
Example.cshtml 1 @page "{handler?}/{id?}"
```

This results in the parameter value being added as a separate segment in the URL:



## Handling Multiple Actions for the Same Form:

Some forms need to be designed to supply for more than one possible action. Where this is the case, you can either write some conditional code to determine which action should be taken, or you can write separate named handler methods and then use the **form action tag helper** to specify the handler method to execute on submission of the form:

```
Example.cshtml.cs -> X
ASP.NETCoreRazorPageWebApp1 -> ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel

1 using Microsoft.AspNetCore.Mvc.RazorPages;
2
3 namespace ASP.NETCoreRazorPageWebApp1.Pages
4 {
5     public class ExampleModel : PageModel
6     {
7         public string Message { get; set; }
8         public void OnGet()
9         {
10             Message = "Initial Request";
11         }
12         public void OnPost()
13         {
14             Message = "Form Posted";
15         }
16         public void OnPostEdit()
17         {
18             Message = "Edit handler fired";
19         }
20         public void OnPostDelete()
21         {
22             Message = "Delete handler fired";
23         }
24         public void OnPostView()
25         {
26             Message = "View handler fired";
27         }
28     }
29 }
```

```
Example.cshtml -> X
1 @page
2 @model ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel
3 @{
4     Layout = null;
5 }
6 <!DOCTYPE html>
7 <html>
8 <head>
9     <meta charset="utf-8" />
10    <title>Example Page</title>
11    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
12 </head>
13 <body>
14     <div style="margin:10px;">
15         <form method="post">
16             <button class="btn btn-primary" asp-page-handler="edit">Edit</button>
17             <button class="btn btn-primary" asp-page-handler="delete">Delete</button>
18             <button class="btn btn-primary" asp-page-handler="view">View</button>
19         </form>
20         <h3>@Model.Message</h3>
21     </div>
22 </body>
23 </html>
```

**Note:** The value passed to the **page-handler** attribute is the name of the handler method without the **OnPost** prefix or **Async** suffix.

## The NonHandler Attribute:

There may be occasions where you don't want a public method on a page to be considered as a handler method, despite its name matching the conventions for handler method discovery. You can use the **NonHandler** attribute to specify that the decorated method is not a page handler method:

```
[NonHandler]
public void OnGetInfo()
{
    //.....
}
```

## Working with HTTP **GET** verb:

Whenever you make a request to access a razor content page through the browser or by clicking on a given link to navigate a particular page, it uses always HTTP Get request which mapped to **OnGet()** or **OnGetAsync()** handler method of a razor page to process any given logic. Even HTTP Get verb can be also used to post the form data but it is not recommended option because of following two major problems:

- Form posted data will be appended to the URL in the form of query string (a series of name/value pairs. After the URL web address has ended, include a question mark (?) followed by the name/value pairs, each one separated by an ampersand (&)) which is clearly visible and leads to security issues if any sensitive data.
- As all posted data of a form is appended to the URL in the form of query string which leads into increasing the length of a URL and almost every browsers limit the lengths of URLs, In addition, many servers limit the lengths of URLs they accept. So if the length of URLs is exceeded than the limit set by a browser or server then there is a possibility of losing the data.

**Note:** To submit the form data, it is recommended to always use HTTP Post verb instead of HTTP Get verb because above both given problems get resolved. However, the HTTP Post verb is not limited by the size (length) of the URL for submitting the data (name/value pairs). These data are appended to the **headers** of the HTTP request and not in the URL.

## Example1: HTTP **GET** verb

```

Example.cshtml.cs
ASP.NETCoreRazorPageWebApp1 - ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel - OnGet(string Command)
1 using Microsoft.AspNetCore.Mvc.RazorPages;
2
3 namespace ASP.NETCoreRazorPageWebApp1.Pages
4 {
5     public class ExampleModel : PageModel
6     {
7         public void OnGet(string Command)
8         {
9             if(Command == "Submit")
10             {
11                 ViewData["Info"] = "Page is Post Back using Get Method.";
12             }
13             else
14             {
15                 ViewData["Info"] = "Page is First Time Requested.";
16             }
17         }
18     }
19 }
    
```

The HTML part of the page includes a **form** that uses the **GET** method with a submit button, which submits the form using a **GET** request:

```

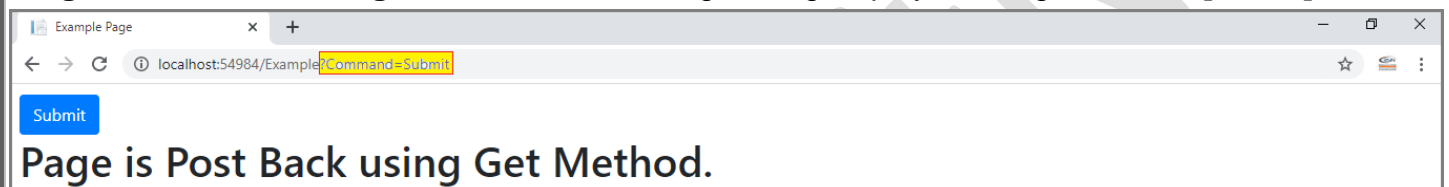
Example.cshtml
1 @page
2 @model ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel
3 @{
4     Layout = null;
5 }
6 <!DOCTYPE html>
7 <html>
8 <head>
9     <meta charset="utf-8" />
10    <title>Example Page</title>
11    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
12 </head>
13 <body>
14     <div style="margin:10px;">
15         <form method="get">
16             <input type="submit" class="btn btn-primary" name="Command" value="Submit" />
17         </form>
18         <h1>@ViewData["Info"]</h1>
19     </div>
20 </body>
21 </html>
    
```



When the page is first navigated/requested to, the “**Page is First Time Requested.**” message is displayed using **ViewData[“Info”]** because the HTTP **GET** verb was used for the request, firing the **OnGet()** handler without any ‘**Command**’ parameter.



When the “**Submit**” button is pressed, the form is being posted using **GET** verb as it is mentioned in **<form>** tag along with form data i.e. button name & value by appending to the URL in the form of query string. The **OnGet()** handler fires once again and if handler parameter name is matched with the query string name then automatically that query string copied to the handler parameter, so in our case query string name is ‘**Command**’ and their value is ‘**Submit**’ which matches the **if** condition, resulting in the “**Page is Post Back using Get Method.**” message being displayed using **ViewData[“Info”]**.



The above example code can also be done without mapping of query string to handler parameter, so in this case we need to access the form posted data which are coming in the form of query string using **Request.Query[“KeyName”]** as following:

```

Example.cshtml.cs
ASP.NETCoreRazorPageWebApp1 - ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel - OnGet()
1 using Microsoft.AspNetCore.Mvc.RazorPages;
2
3 namespace ASP.NETCoreRazorPageWebApp1.Pages
4 {
5     public class ExampleModel : PageModel
6     {
7         public void OnGet()
8         {
9             if(Request.Query["Command"] == "Submit")
10             {
11                 ViewData["Info"] = "Page is Post Back using Get Method.";
12             }
13             else
14             {
15                 ViewData["Info"] = "Page is First Time Requested.";
16             }
17         }
18     }
19 }
    
```

**Working with different handler for the first time page request and posted form:**

```

Example.cshtml.cs
ASP.NETCoreRazorPageWebApp1 - ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel - OnGetSubmit()
1 using Microsoft.AspNetCore.Mvc.RazorPages;
2
3 namespace ASP.NETCoreRazorPageWebApp1.Pages
4 {
5     public class ExampleModel : PageModel
6     {
7         public void OnGet()
8         {
9             ViewData["Info"] = "Page is First Time Requested.";
10         }
11
12         public void OnGetSubmit()
13         {
14             ViewData["Info"] = "Page is Post Back using Get Method.";
15         }
16     }
17 }
    
```

In the above example code, we have two handler for GET verb request i.e. OnGet() and OnGetSubmit().

OnGet() handler is default convention method which is automatically executed on first time page request.

OnGetSubmit() handler is custom method which is executed if a page request with handler in the form of query string e.g. /Example?handler=Submit

The HTML part of the page includes a **form tag** that uses the **GET** method with a hidden field and a submit button, which submits the form using a **GET** request. Here we are using a hidden field just for specifying the handler information with name i.e. 'handler' and value i.e. 'Submit' in our case so when we submit the form by pressing the button, hidden field data will be appended to the URL in the form query string like /Example?handler=Submit then Razor Pages framework automatically looks for custom handler named 'Submit' to execute it and then returns the response.

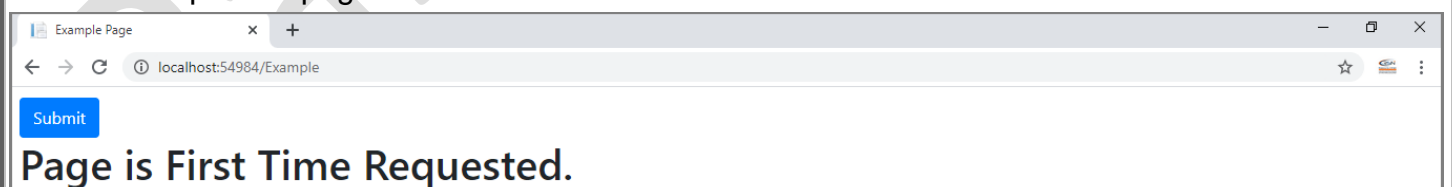
**Note:** In case of form posting using a **GET** request, the **asp-page-handler** attribute value for a form tag helper do not support to specify a named handler as it supports in case of **POST** request only.

```

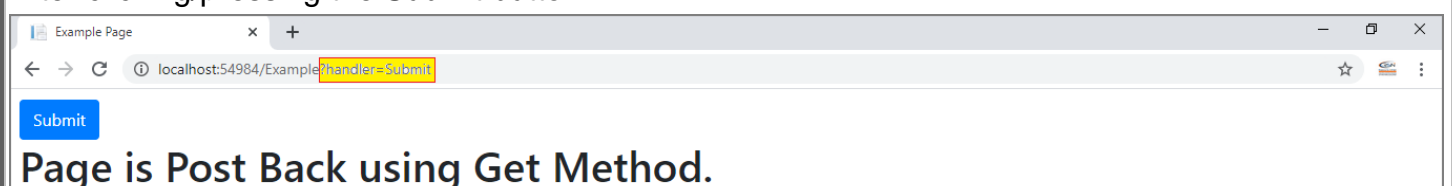
Example.cshtml
1  @page
2  @model ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel
3  @{
4      Layout = null;
5  }
6  <!DOCTYPE html>
7  <html>
8  <head>
9      <meta charset="utf-8" />
10     <title>Example Page</title>
11     <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
12 </head>
13 <body>
14     <div style="margin:10px;">
15         <form method="get">
16             <input type="hidden" name="handler" value="Submit" />
17             <input type="submit" class="btn btn-primary" value="Submit" />
18         </form>
19         <h1>@ViewData["Info"]</h1>
20     </div>
21 </body>
22 </html>
    
```

### Output:

First time request to page:



After clicking/pressing the Submit button:



## Example2: HTTP GET verb

```

Example.cshtml*  X
1  @page
2  @model ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel
3  @{
4      Layout = null;
5  }
6  <!DOCTYPE html>
7  <html>
8  <head>
9      <meta charset="utf-8" />
10     <title>Example Page</title>
11     <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
12 </head>
13 <body>
14     <div style="margin:10px;">
15         <form method="get">
16             <b>Value1: </b><input type="text" name="Value1" value="@Request.Query["Value1"]" />
17             <br /><br />
18             <b>Value2: </b><input type="text" name="Value2" value="@Request.Query["Value2"]" />
19             <br /><br />
20             <input type="submit" class="btn btn-primary" name="Command" value="Add" />
21         </form>
22         <h1 class="clearfix">@ViewData["Result"]</h1>
23     </div>
24 </body>
25 </html>
  
```

```

Example.cshtml.cs  X
ASP.NETCoreRazorPageWebApp1  ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel
1  using Microsoft.AspNetCore.Mvc.RazorPages;
2
3  namespace ASP.NETCoreRazorPageWebApp1.Pages
4  {
5      public class ExampleModel : PageModel
6      {
7          public void OnGet(string Value1, string Value2, string Command)
8          {
9              if(Command == "Add")
10              {
11                  float value1 = float.Parse(Value1);
12                  float value2 = float.Parse(Value2);
13
14                  float result = value1 + value2;
15
16                  ViewData["Result"] = $"Result of Addition: {result}";
17              }
18              else
19              {
20                  ViewData["Result"] = "Page is First Time Requested.";
21              }
22          }
23      }
24  }
  
```

## Output:

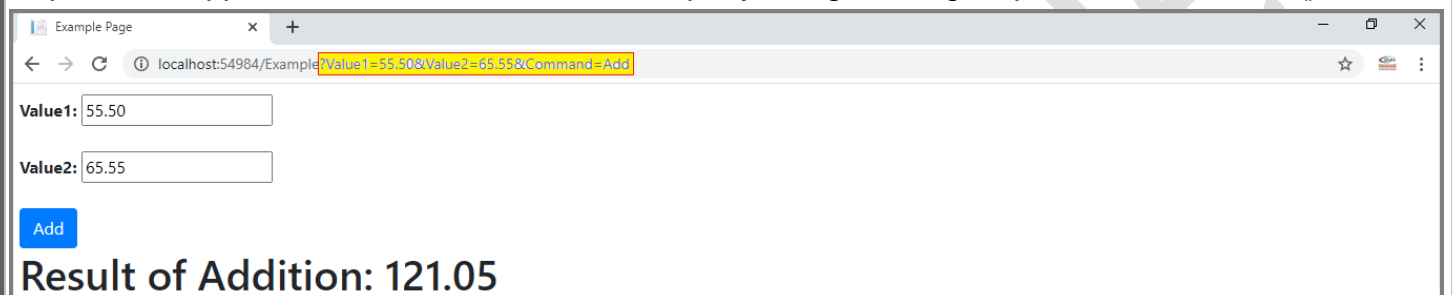


Value1:

Value2:

**Page is First Time Requested.**

Now enter the Value1 and Value2, then click Add button to submit the entered form data using a GET request that appends to the URL in the form of query string which gets processed in OnGet() handler.



Value1:

Value2:

**Result of Addition: 121.05**

In the above example, OnGet() handler uses parameters to map query strings values. Even though it can also be done without mapping query strings values to parameters as following:

```

Example.cshtml.cs
ASP.NETCoreRazorPageWebApp1
ASP.NETCoreRazorPageWebApp1.Pages.ExampleModel

1 using Microsoft.AspNetCore.Mvc.RazorPages;
2
3 namespace ASP.NETCoreRazorPageWebApp1.Pages
4 {
5     public class ExampleModel : PageModel
6     {
7         public void OnGet()
8         {
9             if(Request.Query["Command"] == "Add")
10            {
11                float value1 = float.Parse(Request.Query["Value1"]);
12                float value2 = float.Parse(Request.Query["Value2"]);
13
14                float result = value1 + value2;
15
16                ViewData["Result"] = $"Result of Addition: {result}";
17            }
18            else
19            {
20                ViewData["Result"] = "Page is First Time Requested.";
21            }
22        }
23    }
24 }
    
```