**Introduction to ASP.NET Core Framework**

**Introduction to ASP.NET Core Framework**

In this article, I am going to give you a brief introduction to **ASP.NET Core Framework**. Nowadays, when it comes to software development, everyone is talking about free, open-source, and cross-platform development. As we all know Microsoft is well known for its Windows-based products. Now we are in the new age of software development. For this, a new revolutionary product came into the market by Microsoft and it is **ASP.NET Core**. As part of this article, we are going to discuss the following pointers in detail.

1. **History of ASP.NET**
2. **What is ASP.NET Core**
3. **Why ASP.NET Core?**
4. **Where can we develop applications using .NET Core?**
5. **What the ASP.NET Core doesn’t have?**
6. **.NET Core vs ASP.NET Core**
7. **Language Support**

**History of ASP.NET**

As we know, ASP.NET is the framework that has been used to develop data-driven web applications for many years. Since then, the ASP.NET Framework went through a steady evolutionary change, and finally, the most decent evolution is ASP.NET Core.

1. The ASP.NET Core is not a continuous part of the ASP.NET 4.x Framework. Instead, it is a completely new framework.
2. This Framework is an actual re-write of the current ASP.NET 4.x Framework, but with much smaller and a lot more modular.
3. Some people think that many things remain the same, but that is not completely true. The ASP.NET Core is actually a big fundamental change to the ASP.NET Framework.

**What is ASP.NET Core?**

ASP.NET Core is the new version of the ASP.NET Web Framework mainly targeted to run on .NET Core Platform.

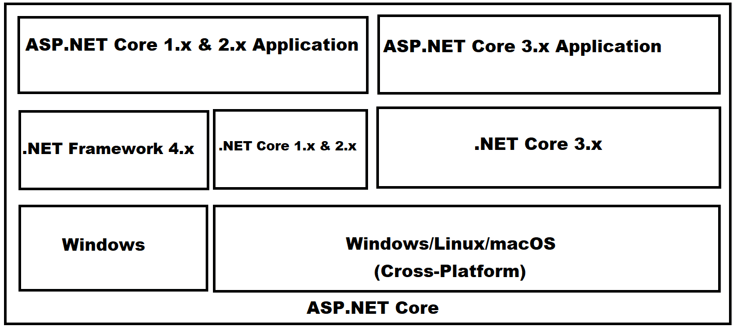
ASP.NET Core is a free, open-source, high-performance, light-weight, and cross-platform framework for building cloud-based applications, such as web apps, IoT apps, and mobile backend. It is designed to run on the cloud as well as on-premises.

Same as .NET Core, it was architected modular with minimum overhead, and then other more advanced features can be added as NuGet Packages as per application requirement. This results in high performance, require less memory, less deployment size, and easy to maintain.

ASP.NET Core Framework is completely rewritten of existing ASP.NET 4.x, with architectural changes that make it more modular, scalable, open-source, high-performance, lightweight, and a cross-platform web framework.

The ASP.NET Core is an open-source framework supported by Microsoft and the community, so you can also contribute or download the source code from the ASP.NET Core Repository on GitHub (**https://github.com/dotnet/aspnetcore**)

ASP.NET Core 3.x application can be targeted to develop and run on top of the .NET Core only whereas ASP.NET Core 1.x and 2.x application can be targeted to develop and run on top of the .NET Core (Cross-platform) as well as .NET Framework (Windows only). For better understanding, please have a look at the following image.



**Note:** ASP.NET Core was initially launched as **ASP.NET 5** but later it was renamed to **ASP.NET Core**.

**Why ASP.NET Core?**

Nowadays, the ASP.NET Core framework becomes more and more popular among developers. There are a number of reasons why modern developers are using it and some of them are listed below:

**Open Source:**

The ASP.NET Core framework is Open Source which is the main reason behind its popularity. The Entire Source Code of this .NET Core Framework is available at **https://github.com/aspnet** and you are free to download the source code and even if you want then you can also modify and compile your own version of it.

.NET Core team is always there to support your effort in the seamless development of the application. It receives bug fixing and improvement updates on a regular basis usually within a short time period. You don’t have to wait longer for updates. The flexibility in framework code is keeping it at the top of the developer’s choice. Depending upon the project’s requirements libraries and framework’s component can be added later.

**Cross-Platform:**

The ASP.NET Core Framework is designed from scratch to keep in mind to be Cross-Platform for both development and deployment. So, we don’t need to build different applications for different platforms using different frameworks. Let us discuss what Cross-Platform is from the ASP.NET Core point of view by comparing it with the earlier versions of the ASP.NET Framework.

The earlier versions of ASP.NET Framework applications can only run on windows platforms whereas the ASP.NET Core applications can develop and run on different platforms such as Windows, Mac, or Linux operating systems. We can host the earlier ASP.NET Framework 4.x applications only on IIS whereas we can host the ASP.NET Core applications on IIS, Nginx, Docker, Apache, or even self-host deployment.

To develop ASP.NET Core applications, you have multiple options like you can use either Visual Studio or Visual Studio Code. If you want, then you can also use any third-party editors as per your choice.

**CLI Support:**Using CLI commands you can develop and run .NET applications as well as you can also publish the application using CLI command.

**Fast:**ASP.NET Core no longer depends on System.Web.dll for browser-server communication. ASP.NET Core allows us to include packages that we need for the application. This reduces the request pipeline and improves performance and scalability.

**IoC Container:**One of the most important used design patterns in the real-time application is the [**Dependency Injection Design Pattern**](https://dotnettutorials.net/lesson/asp-net-core-dependency-injection/). It includes the built-in IoC (Inversion of Control) container for automatic dependency injection which makes it maintainable and testable.

**Unified MVC and Web API Framework:**

The ASP.NET Core provides a unified programming model for developing both Web apps and Web APIs. That means a single controller class can be used to handle both. The Controller we create in ASP.NET Core (either Web APPs or Web APIs) application is going to inherit from the Controller base class and returns the IActionResult interface. The IActionResult interface provides several implementations. The built-in result types such as the JsonResult and the ViewResult and (like this there are so many result types available that we will discuss later) implement the IActionResult interface.

In the ASP.NET Core Web API application, the controller action method is going to return JsonResult. At the same time if it is an ASP.NET Core Web application, then the return type of the controller action method is going to be ViewResult.

**Testing and Maintainability:**

You can easily test and maintain the applications developed using the ASP.NET Core MVC framework. This is possible because it allows you to separate different parts of your application into independent pieces and it also allows you to test them independently. The Testing frameworks such as xUnit and MOQ can be easily integrated into ASP.NET Core MVC application for simulating any scenario.

**Integration with Modern UI Framework:**It allows you to use and manage modern UI frameworks such as AngularJS/Angular, React JS, and Bootstrap, etc…

**Handling Request and Response Pipeline:**

We can handle the request and response in the ASP.NET Core application by using the new [**Middleware Components**](https://dotnettutorials.net/lesson/asp-net-core-middleware-components/). In earlier ASP.NET 4.x we generally use Handlers and Modules to handle the Request and Response pipeline. The ASP.NET Core Framework provides a lot of built-in [**Middleware Components**](https://dotnettutorials.net/lesson/asp-net-core-middleware-components/) and we can use those **Middleware Components** to handle the request and response pipeline. If you want, then also you can create your own middleware components and use them in the request-response pipeline. In a later article, we will discuss the **Middleware Components** in detail. We will also discuss how to create custom Middleware Components and how to use them in ASP.NET Core applications.

**Hosting:**

ASP.NET Core web application can be hosted on multiple platforms with any web server such as IIS, Apache, etc… It is not dependent only on IIS as a standard .NET Framework.

**Code Sharing:**It allows you to build a class library that can be used with .NET Frameworks such as .NET Framework 4.x or Mono. Thus, a single code base can be shared across frameworks.

**Side-by-Side App Versioning:**ASP.NET Core runs in .NET Core, which supports the simultaneous running of multiple versions of applications.

**Extensible Framework:**

The ASP.NET Core MVC Framework is designed to be highly extensible. That means you can create an application today, that can be extended to any levels in the future. Some of the key features provided by this framework that give it the extensible power are as follows.

1. [**View Components**](https://dotnettutorials.net/lesson/views-asp-net-core-mvc/)
2. **Tag Helpers**
3. [**Routing**](https://dotnettutorials.net/lesson/routing-asp-net-core-mvc/)

In our upcoming articles, we will discuss each of these features in detail.

**Smaller Deployment Footprint:**

ASP.NET Core application runs on .NET Core, which is smaller than the full .NET Framework. So, the application which uses only a part of .NET CoreFX will have a smaller deployment size. This reduces the deployment footprint.

**Excellent developer tools:**

ASP.NET Core comes with a lot of preloaded tools. Anybody would love to work with these tools due to their simplicity and ease. Using JetBrains, ReSharper, and other tools it very easy to build tests and other extraordinary applications.

**Versatility:**Apart from modern web application users can also create mobile and desktop applications. They can switch whenever they want.

**Note:** Many people are confused between ASP.NET Core and .NET Core. Please note that ASP.NET Core and .NET Core are not the same. They are different, just like ASP.NET and .NET Framework are different. .NET Core is a fast, lightweight, modular, and open-source framework for creating web applications, services that run on Windows, Linux, and macOS. So, it is a software platform on which ASP.NET Core applications run.

**What the ASP.NET Core doesn’t have?**

If you are coming from ASP.NET 4.x, then you will not find the following things in ASP.NET Core

1. **The Global.asax file**
2. **Web.Config file**
3. **HTTP Handlers and HTTP Modules**
4. **ASP.NET Page Life-Cycle model**

**.NET Core vs ASP.NET Core:**

**.Net Core:**

1. .NET Core is open-source and cross-platform
2. .NET Core is a runtime to execute applications which ate build on it.
3. Install .NET Core Runtime to run applications and install .NET Core SDK to build applications.
4. .NET Core 3.1 – latest version

**ASP.NET Core:**

1. Asp.net core is Open-source and Cross-platform
2. ASP.NET Core is a web framework to build web apps, IoT apps, and mobile backend on top of .NET Core.
3. There is no separate runtime and SDK are available for ASP.NET Core. .NET Core runtime and SDK includes ASP.NET Core libraries.
4. ASP.NET Core 3.1 – latest version
5. There is no separate versioning for ASP.NET Core. It is the same as the .NET Core versions.

**Language Support:**

Dot Net Core Framework supports the following language i.e. using the following languages you can develop .NET Core applications. We are going to use C# as the programming language in this course.

1. [**C#**](https://dotnettutorials.net/course/csharp-dot-net-tutorials/)
2. **F#**
3. **Visual Basic (VB)**

**.NET Core GitHub Repository:**

**.NET Core Runtime: https://github.com/dotnet/runtime**  
**.NET Core SDK: https://github.com/dotnet/sdk**  
**ASP.NET Core GitHub Repository: https://github.com/dotnet/aspnetcore**

The Latest version of .NET Core is 3.1 as of today date.  
The latest version of ASP.NET Core also is 3.1 as of today’s date.  
There is no separate versioning of ASP.NET Core. It is the same as the .NET Core Versions.

.NET Core is named “Core” because it includes core features of .NET Framework. The main objective of .NET Core is to make .NET Framework open-source and cross-platform compatible so that it can be used in resource-constrained environments. It includes minimum features that are required to run a basic .NET Core application and other advanced features that can be as a package from NuGet.

**ASP.NET Core Environment Setup**

**ASP.NET Core Environment Setup**

In this article, I am going to discuss the **ASP.NET Core Environment Setup** required for developing ASP.NET Core application. Please read our previous article before proceeding to this article where we gave a brief introduction to [**ASP.NET Core Framework**](https://dotnettutorials.net/lesson/introduction-to-asp-net-core/). As part of this article, we are going to discuss the following pointers.

1. **How to set up a development machine for dot net core applications development**
2. **Install SDK and Editor**
3. **Verify the installation**

**Tools and Software Requires for the development of .NET Core Applications.**

**Machine:** (Windows, Mac, Linux)  
**Editor:** Recommended Visual Studio, VS Code  
**Dot Net Core SDK:** This is the software development KIT and this KIT is helpful for the development and running of the application in the system.

**How to prepare a development environment for building .NET Core/ASP.NET Core applications?**

The .NET Core can be installed in two ways:

1. **By Installing Visual Studio 2017/2019**
2. **By Installing .NET Core SDK**

The .NET Core SDK installer already contains ASP.NET Core libraries, so there is no separate installer for ASP.NET Core.

**Install Visual Studio 2017/2019:**

Currently, .NET Core 2.1 and .NET Core 3.1 is having Long Term Support (LTS). Visual Studio 2017 supports .NET Core 2.1, whereas Visual Studio 2019 supports both the versions.

You can use your favorite IDE, such as Visual Studio 2017/2019, Visual Studio Code, and Sublime Text, etc. to develop, restore, build and run .NET Core application. In my machine, I have installed Windows operating system. So, I am going to use Visual Studioas my editor, or you can say Integrated Development Environment (IDE) for developing ASP.NET Core application. But this is not mandatory. You can use any editor as per your choice.

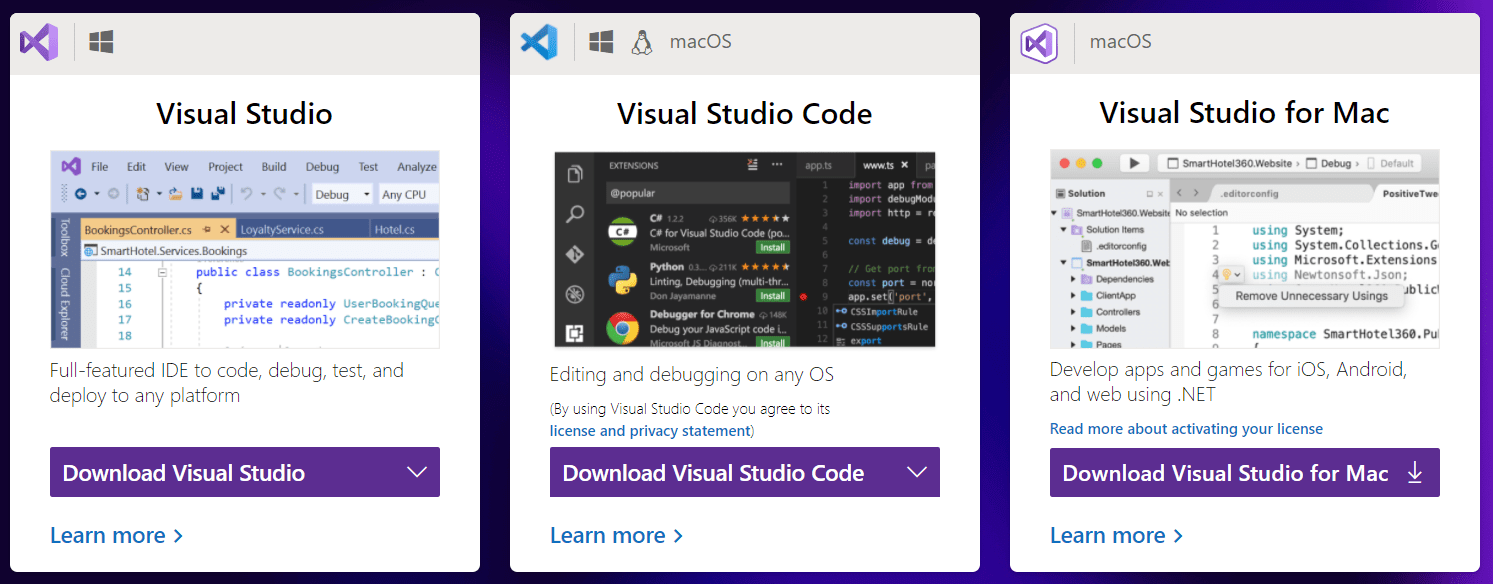
If you don’t have Visual Studio on your development PC, then it is recommended to install the latest Visual Studio 2019. If you already have either Visual Studio 2017 or 2019, then you already have installed .NET Core SDK 2.1.

**Download and install Visual Studio**

Download and install Visual Studio 2019 based on your OS. Select the appropriate edition as per your license. The Visual Studio community edition is free for students, open-source contributors, and individuals. As of this article, the latest version of Visual Studio is **Visual Studio 2019** and it can be downloaded from the following site

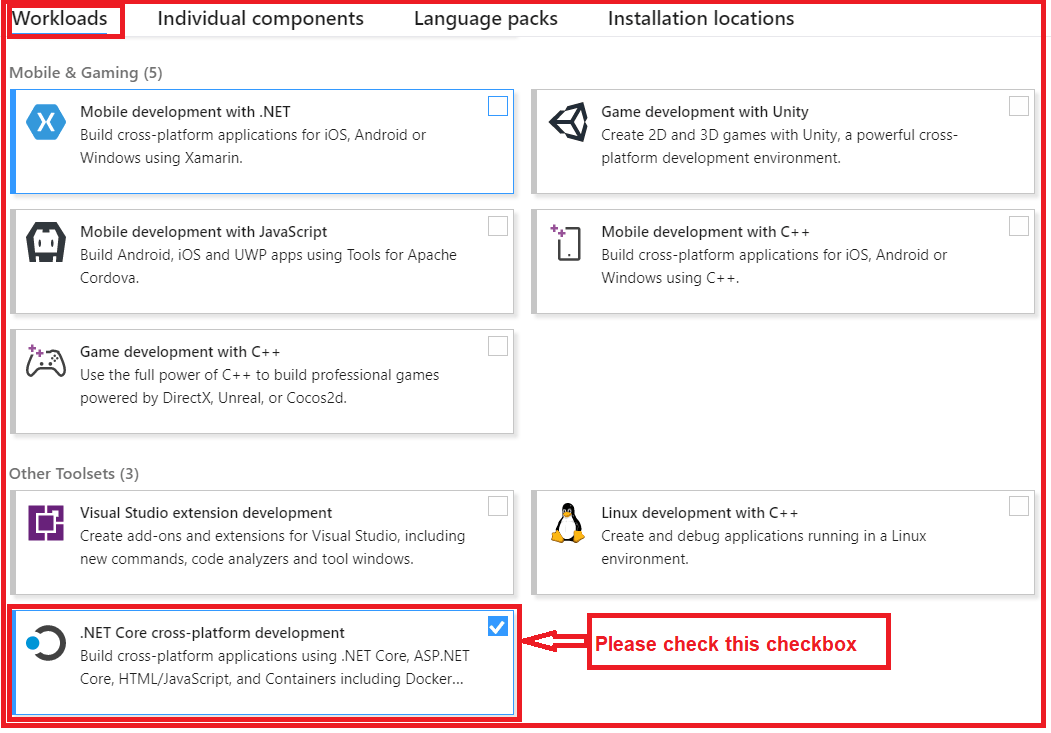
[**https://visualstudio.microsoft.com/**](https://visualstudio.microsoft.com/)

Once you visit the above website, you will see the following.



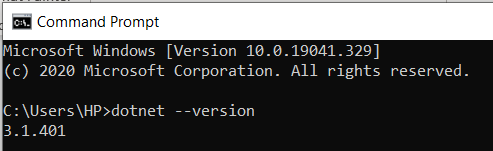
Based on your operating system download the required Visual Studio. In my machine, I have installed the Windows Operating System, so I choose Visual Studio and select the Community Edition and download it.

To develop .NET Core applications in Visual Studio 2019 we need .NET Core cross-platform development. So, while installing the Visual Studio 2019, please make sure to select the .NET Core cross-platform development workload checkbox as shown in the below image. This will install the .NET Core SDK 2.1. However, you need to install .NET Core 3.1 SDK separately if by default not available while installing Visual Studio 2019.



If you already have installed the Visual Studio 2019, then make sure the above .NET Core cross-platform development is installed. If not installed, then you just install this by using Visual Studio Installer.

Once you installed, then you can verify the same using command prompt (Windows OS) and Terminal (Linus OS). As in my Machine, I have installed Windows Operating System, So, I can verify the same using the “**dotnet –version”** command as shown in the below image.



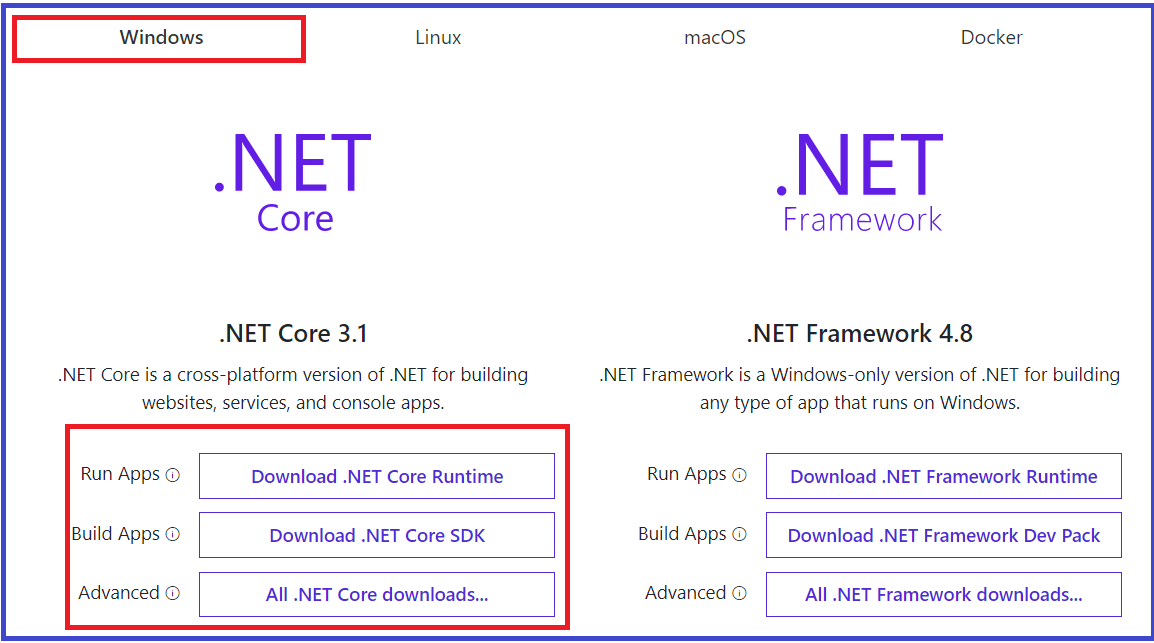
If the .NET Core Framework installed successfully, in your machine, they will see the version number as shown in the above image.

**Download and install .NET Core SDK**

If Visual Studio 2019 installer doesn’t include .NET Core 3.x then you need to install it separately. The ASP.NET Core is a part of .NET Core SDK, so you don’t require to install it separately. To download the later version of .NET Core i.e. 3.1 as of today date, go to the following website and select the platform you are using.

[**https://dotnet.microsoft.com/download**](https://dotnet.microsoft.com/download)

Once you visit the above page, you will find the following. Here you will find two things as shown in the below image



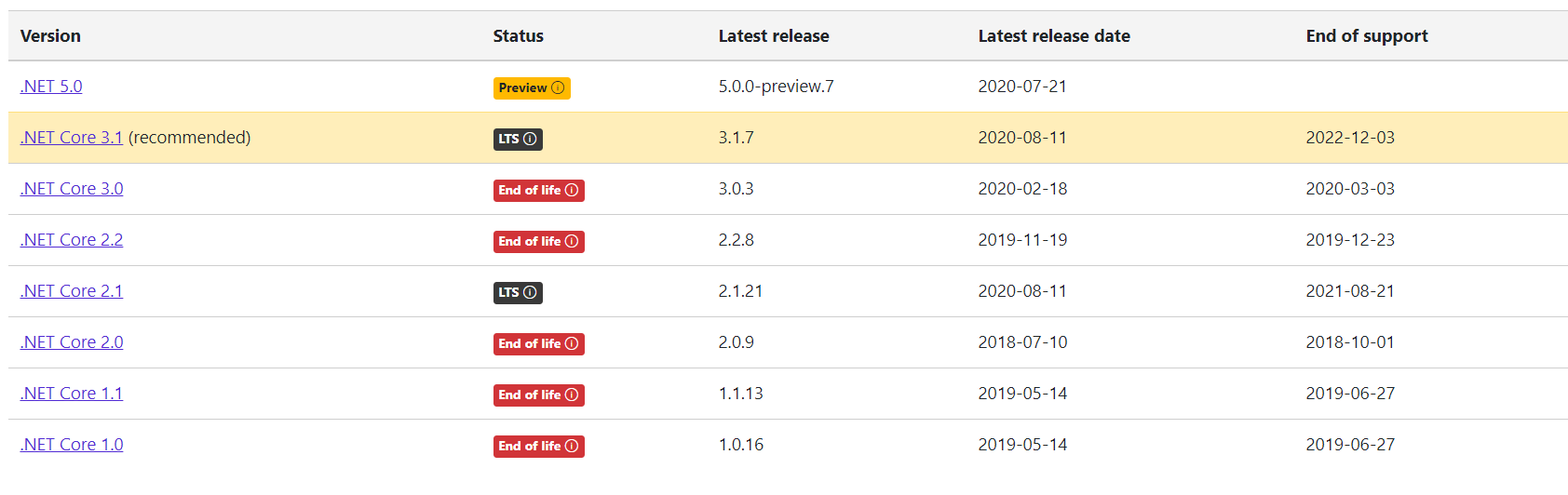
As you can see, when we select the Windows option, then we have two options to develop .NET Core Application i.e. .NET Core 3.1 and .NET Framework 4.8. As you can see in the above image, .NET Core is a cross-platform of .NET for building websites, services, and console apps. On the other hand, .NET Framework is a Windows-only version of .NET for building any type of app that runs on Windows only. Here we are going with .NET Core 3.1 which is the latest version of .NET Core Framework as of this writing.

**Further, in .NET Core there are three options are as follows:**

**.NET Core Runtime:** The .NET Core Runtime is required only to run .NET Core applications. The .NET Core Runtime just contains the resources or libraries which are required to run existing .NET Core applications.

**.NET Core SDK:** If you want to develop and run the .NET Core application, then you need to download the **.**NET Core SDK. The .NET Core SDK contains the .NET Core Runtime. So, if you installed the .NET Core SDK, then there is no need to install .NET Core Runtime separately.

**All .NET Core downloads:** If you want an older version of .NET Core, then you need to click on this link and it will navigate to another page, from where you can download the .NET core Framework version as per your choice as shown in the below image.

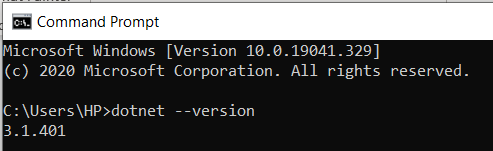


As we are going to set up a development environment for developing and running .NET Core Applications, so, we need to install .NET Core SDK.

Click on the **Download .NET Core SDK** button to download the latest version of .NET Core SDK. It will download the .NET Core 3.1 SDK as of this writing. After downloading the installer, click on it to start the installation.

**Verify the installation**

Once you install.NET Core SDK then again you can verify the same using the command prompt. Open Command Prompt and type **dotnet –version** command and press enter. If the .NET Core Framework installed successfully, in your machine, then you will see the version number as shown in the above image.



**Creating ASP.NET Core Web Application**

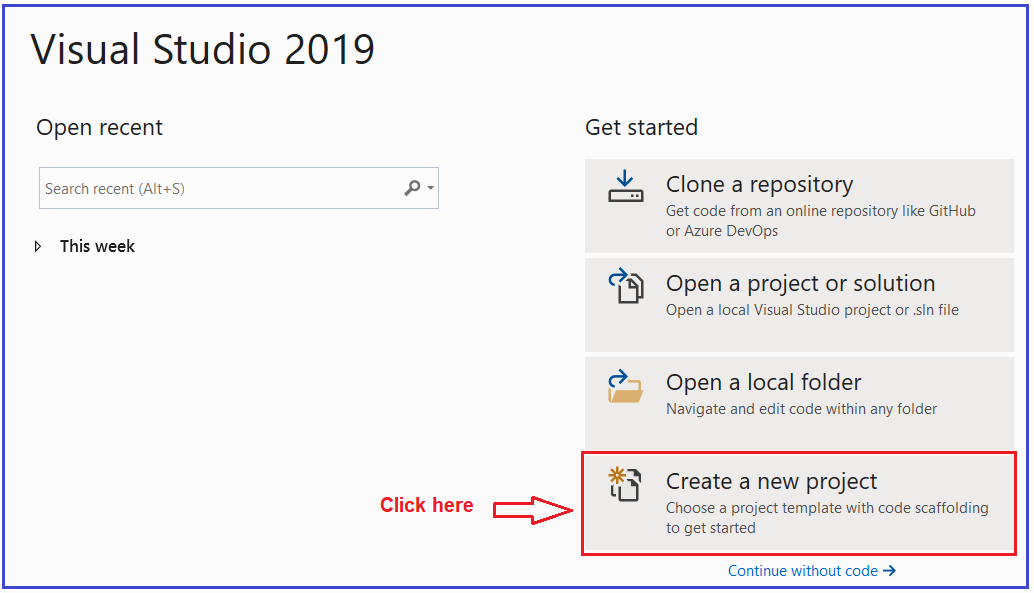
**Creating ASP.NET Core Web Application**

In this article, I am going to discuss how to create **ASP.NET Core Web Application** step by step from scratch. Please, read our previous article before proceeding to this article where we discussed [**what is the software required to develop and run the ASP.NET Core application**](https://dotnettutorials.net/lesson/asp-net-core-environment-setup/) based on your operating system. As part of this article, we are going to discuss the following two things.

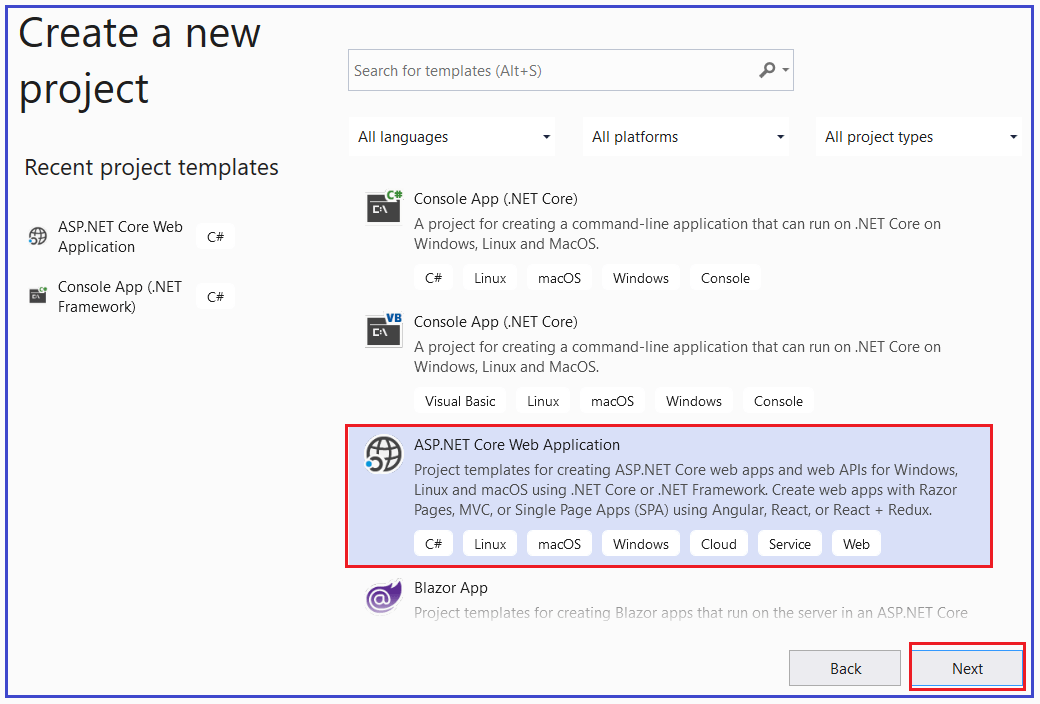
1. **Creating a new ASP.NET Core Web Application step by step using Visual Studio 2019.**
2. **Different project templates that are available as part of .NET Core and what they are going to do.**

**Creating First ASP.NET Core Web Application using Visual Studio** **2019**

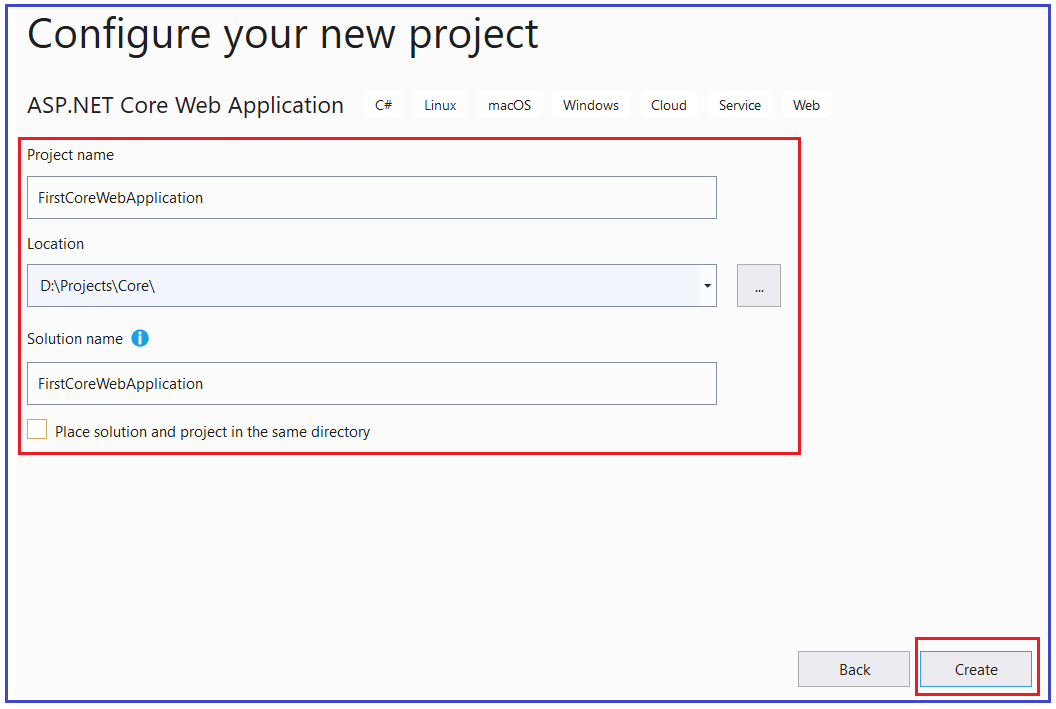
To create a new ASP.NET Core Project, Open Visual Studio 2019. And then click on the Create a new project box as shown in the below image.



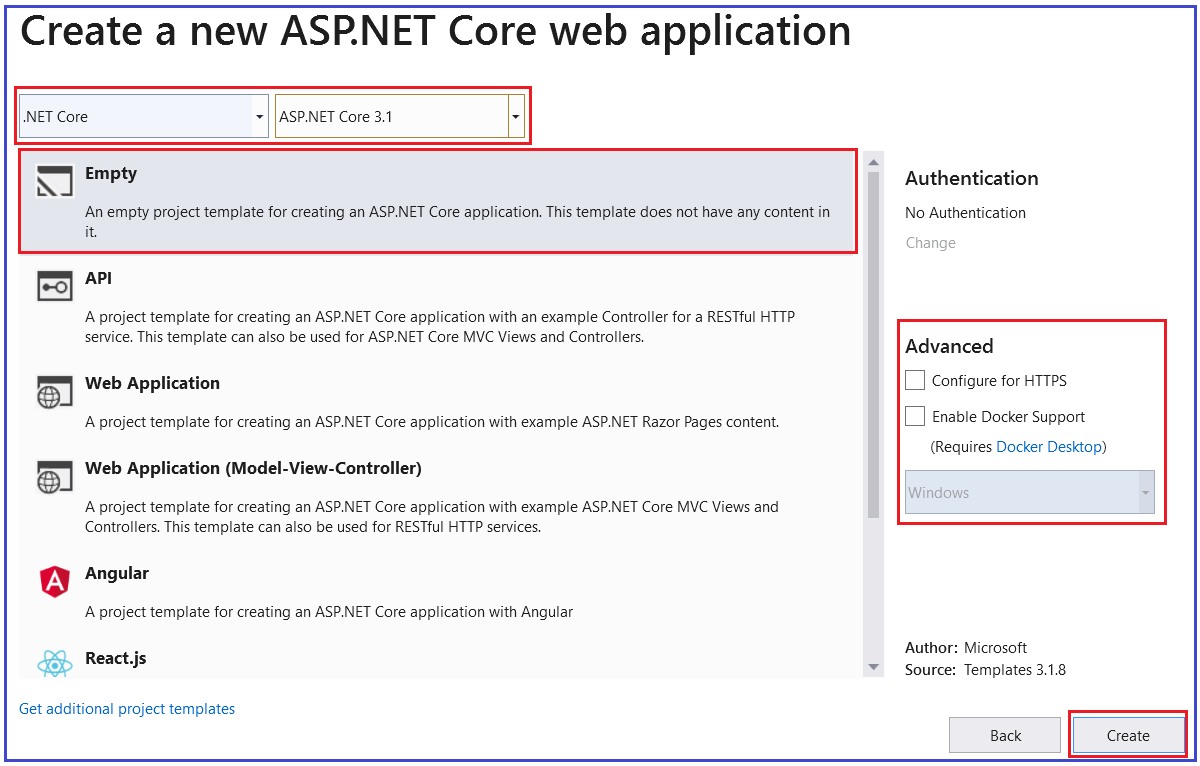
Once you click on the Create a new project box, it will open the “**Create a new project**” window. This window includes different .NET Core 3.1 application templates. Here we will create a simple web application, so select the **ASP.NET Core Web Application** template and click on the Next button as shown in the below image.



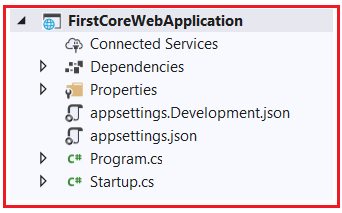
Once you click on the Next button, it will open the following **Configure Your New Project** window. Here, you need to give an appropriate name for your project, set the location where you want to create this project, the solution name for the ASP.NET Core Web application. In this example, we will give the name “**FirstCoreWebApplication**” and click on the Create button as shown in the image below.



Once you click on the create button, it will open the **Create a new ASP.NET Core Web Application** as shown below. Here, you need to select the appropriate ASP.NET Core Web application template such as Empty, API, Web Application, Web Application (MVC), Angular, etc. Here in this demo, we are going to use the Empty template, so that you will understand how the different components fit together to develop an ASP.NET Core application. Also, make sure that you have selected the appropriate .NET Core and ASP.NET Core versions (latest 3.1). Make sure to uncheck all the checkboxes from the Advanced section and finally click on the Create button as shown in the below image.

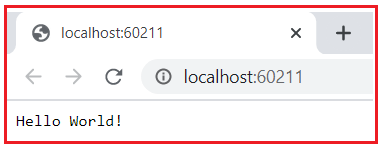


Once you click on the Create button, it will **create a new ASP.NET Core Web Project in Visual Studio 2019**. Wait for some time till Visual Studio restores the packages in the project. The restoring process means Visual Studio will automatically add, update or delete configured dependencies as NuGet packages in the project. The project will be created with the following file and folder structure in Visual Studio 2019.



**Run the ASP.NET Core Application:**

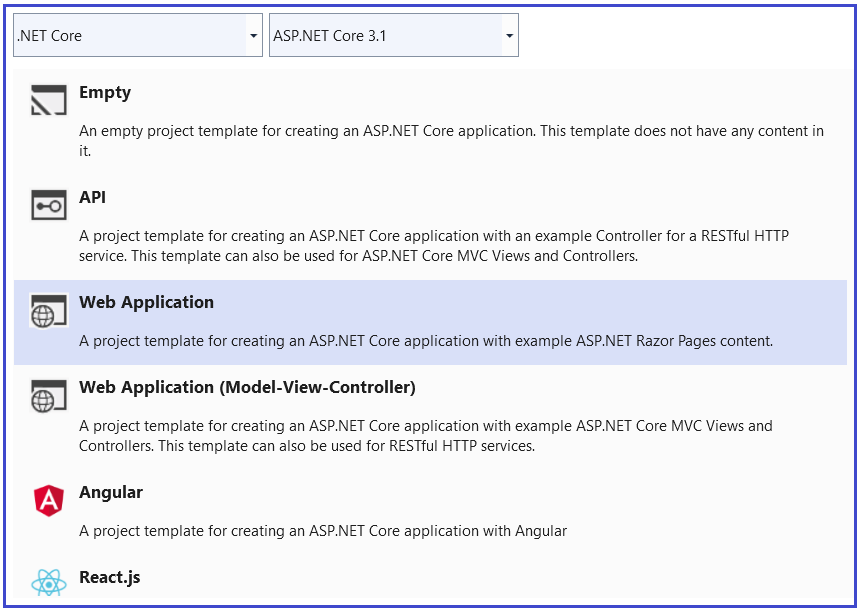
To run this web application, click on IIS Express or press F5 (with Debug) or Ctrl + F5 (without Debug). This will open the browser and display the following output.



Here, the output “**Hello World!**” comes from the **Configure** method of the **Startup** class which is present inside the **Startup.cs** file Open **Startup.cs** file and then change the “**Hello World!**” string to something else and rerun the application and it will change the output accordingly.

**Project templates in ASP.NET Core Application**

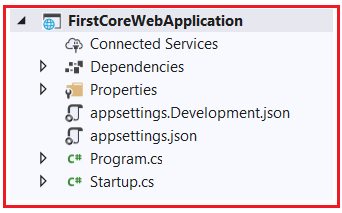
As you can see in the below image, while creating ASP.NET Core Web Application, we have different types of project templates for creating **ASP.NET Core** Web applications.



So, let us discuss a little about all these project templates. In our upcoming articles, we will use and build the ASP.NET Core Web application using all these project templates.

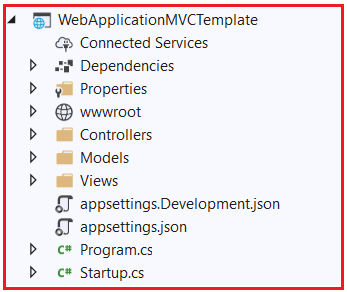
**Empty Project Template:**

As the name says, the Empty Project Template does not have any content by default. If you want to do everything manually from scratch, then you need to select the Empty Template. The following image shows the structure of an Empty template.



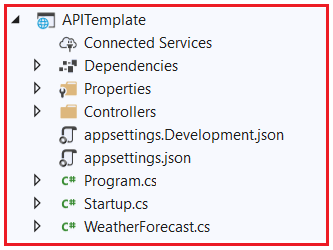
**Web Application (Model-View-Controller) Template:**

The Web Application (Model-View-Controller) template contains everything that is required to create an ASP.NET Core MVC Web Application. The Web Application (Model-View-Controller) template creates Models, Views, and Controllers folders. It also adds web-specific things such as JavaScript, CSS files, Layout files, etc. which are necessary and required in order to develop a web application. The following image shows the structure of a Web Application (Model-View-Controller) template.



**API Template:**

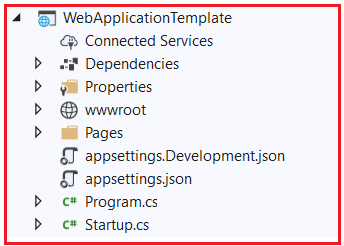
The API template contains everything that is required and necessary to create an ASP.NET Core RESTful HTTP service. The following image shows the structure of an API template.



As you can see from the above image, it contains only the Controllers folder. The website-specific things such as CSS files, JavaScript files, view files, layout files, etc. are not present. This is because an API does not have any user interface hence it does not include such website-specific files. This API template also does not have the Models and Views folder as they are not required for an API.

**Web Application Template**

The Web Application Template uses the new **Razor Pages framework**for building web applications. With new Razor Pages, the coding page-focused scenarios are much easier and more productive. We need to use this template when we want to develop a web application but do not want the full complexity of ASP.NET MVC. The following image shows the structure of the Web Application template.



**Angular, React.js, React.js, and Redux:**

You can also create an asp.net core web application in combination with Angular, React or React and Redux.

**ASP.NET Core Project File**

**ASP.NET Core Project File**

In this article, I am going to discuss the **ASP.NET Core Project File** in detail. Please read our previous article before proceeding to this article, where we discussed how to create [**ASP.NET Core Web Application**](https://dotnettutorials.net/lesson/creating-asp-net-core-web-application/) as well as we also discussed the different project templates that are available as part of ASP.NET Core Web Application.

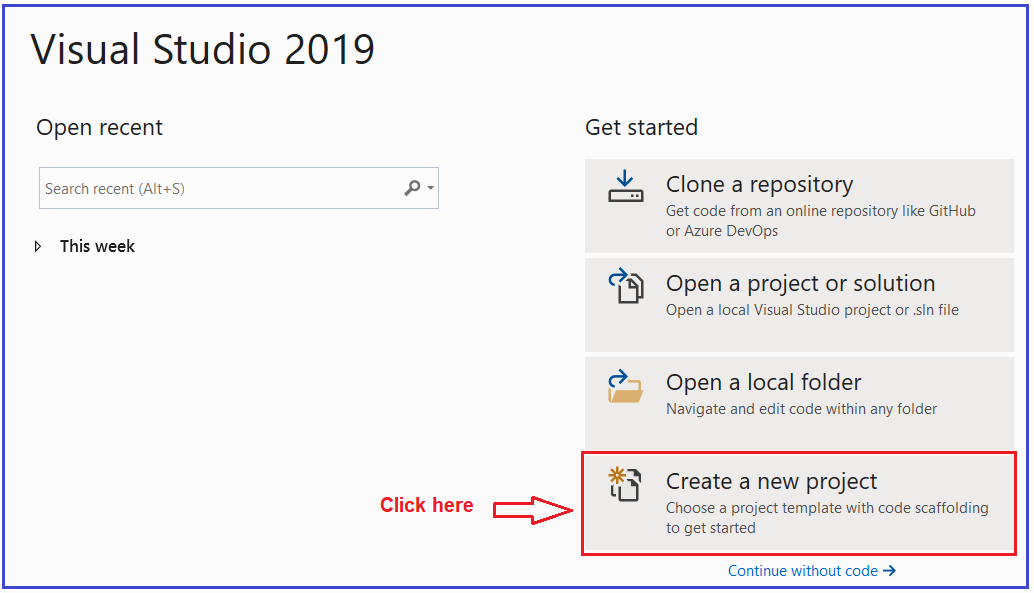
If you have worked with the previous versions of ASP.NET Framework, then you may know, while creating a project in visual studio, it creates a project file for us. If you are using C# as the programming language then it will create the project file with the “.csproj” extension. Similarly, if you are using VB as the programming language, then it will create the project file with the “.vbproj” extension. But with ASP.NET Core, the format and content of the project file have been changed significantly.

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

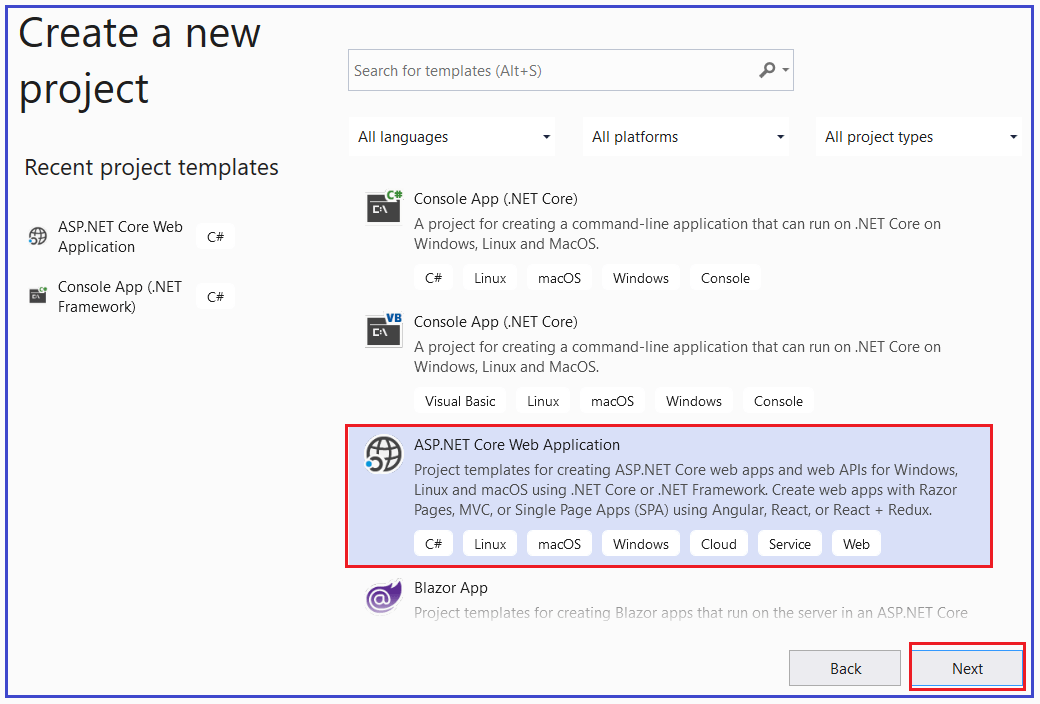
Let us understand what changed are made to the ASP.NET Core Project file with an example. So, let’s create a new ASP.NET Core Project with the Empty template.

**Creating a new ASP.NET Core Project with Empty template:**

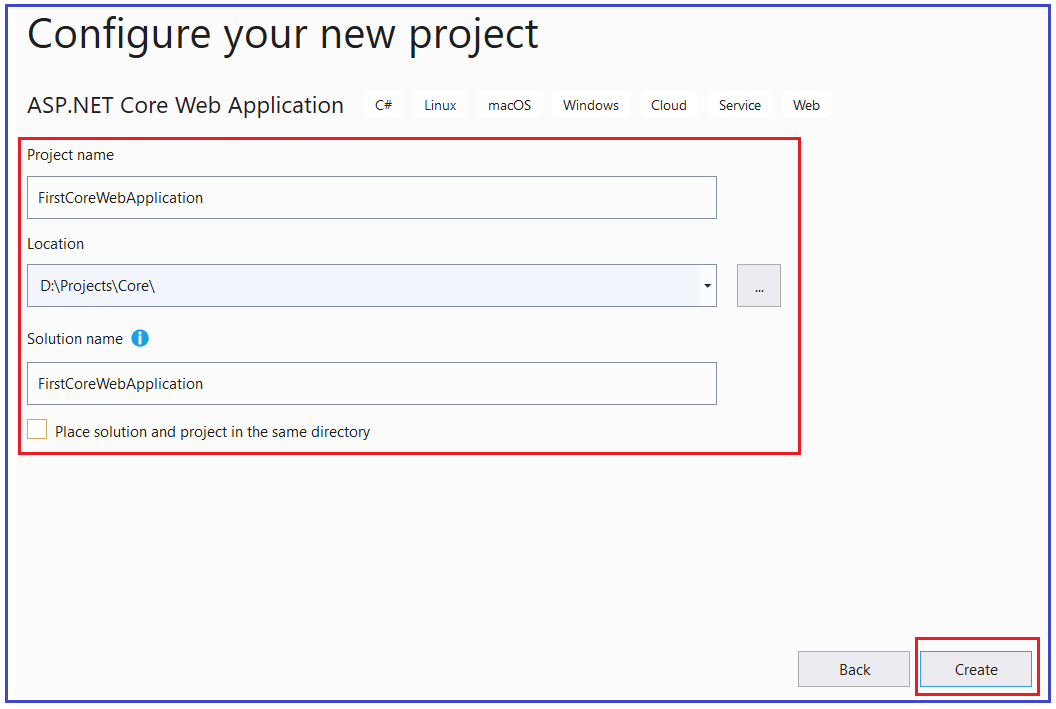
Open Visual Studio 2019. And then click on the Create a new project box as shown in the below image.



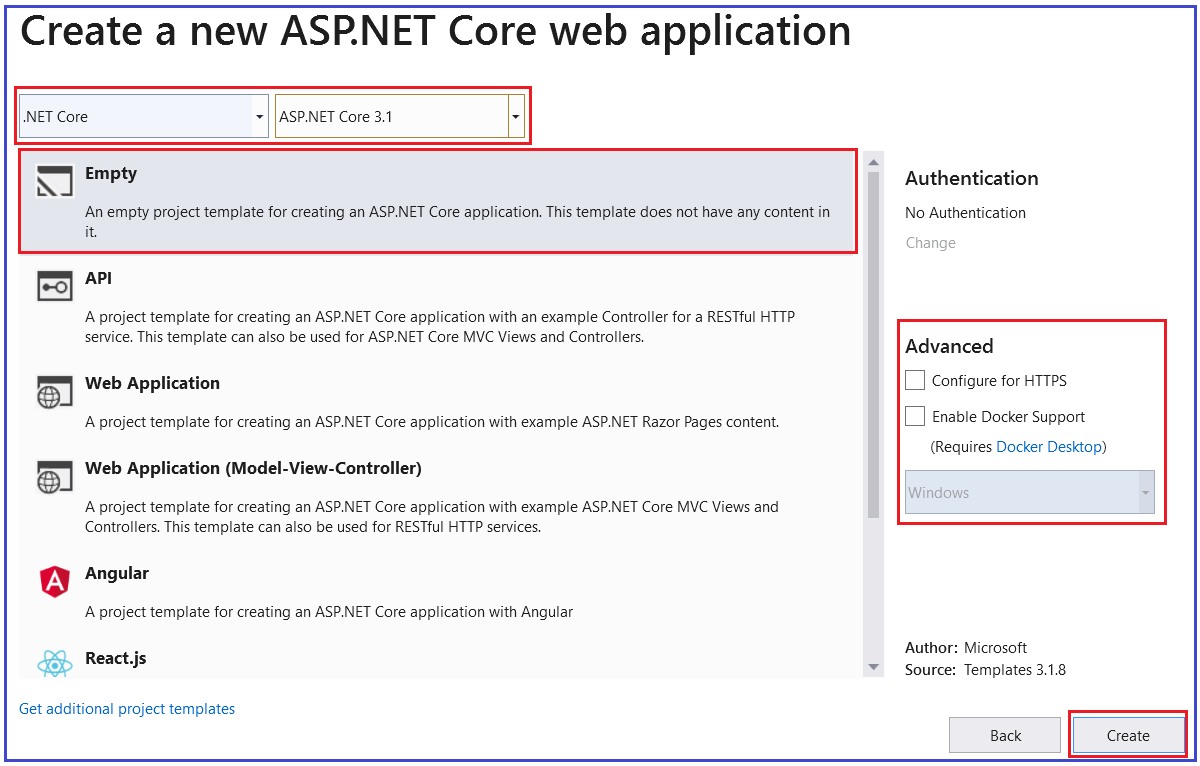
Once you click on the **Create a new project** box, it will open the “**Create a new project**” window. Here, select the **ASP.NET Core Web Application** template and click on the Next button as shown in the below image.



Once you click on the Next button, it will open the **Configure Your New Project** window. Here, you need to give the project name, the location where you want to create this project, and the solution name. In this example, let us give the name as “**FirstCoreWebApplication**” (you can give any name as per your choice) and click on the Create button as shown in the image below.



Once you click on the create button, it will open the “**Create a new ASP.NET Core Web Application**” window. Here, you need to select the Empty Template. Also, make sure that you have selected the.NET Core and ASP.NET Core versions (latest 3.1). Make sure to uncheck all the checkboxes from the Advanced section and finally click on the **Create** button as shown in the below image.



Once you click on the Create button, it will create a new ASP.NET Core Web Project in Visual Studio 2019.

**Changes in the Project file in ASP.NET Core Application:**

The ASP.NET Core Project File does not contain any references for folder or file. In previous versions of ASP.NET Framework, when we add a folder or a file to the project using Solution Explorer, then a reference to that folder or file is added in the project file. But with ASP.NET Core, the Project File does not include any reference for the Files or Folders added the project using Solution Explorer.

Another significant change in ASP.NET Core is, the File System determines what folders and files belong to the project. Generally, the files and folders which are present at the project root folder are part of the project. Those files and folders which are present at the project root folder are only going to display in the Solution Explorer.

So, if you add a file or a folder using the File Explorer, then that file or folder is part of the project. As soon as you add a file or folder using the file explorer, it immediately displayed in the solution explorer. In the same way, if you delete a file or folder from the Project Folder or any of its subfolders, then that deleted file or folder is no longer part of the project and that is reflected immediately in the Solution Explorer. And the vice-versa is also true.

**How we can Edit the Project File in Previous Versions of ASP.NET:**

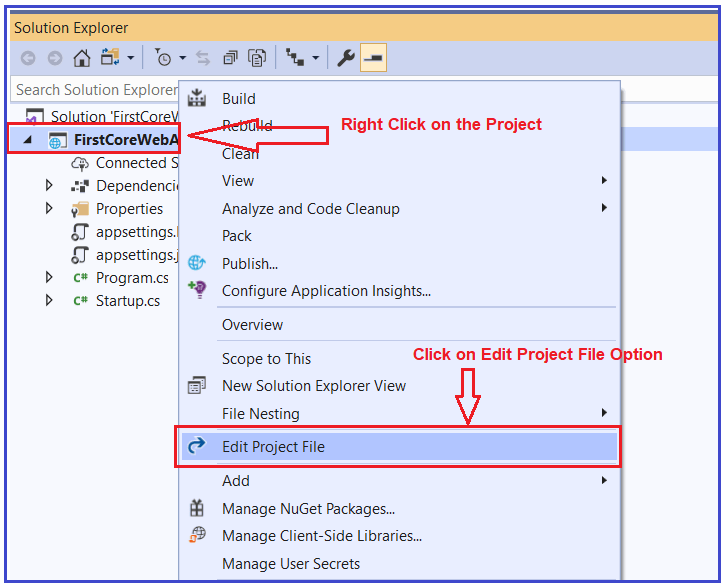
In our previous versions of ASP.NET Framework, in order to edit the project file, we need to follow the below steps

1. First, we need to unload the project
2. Then we need to edit the project file
3. Once we edit the file then we need to save the project file
4. Then reload the project.

But with ASP.NET Core we can edit the project file without unloading the project.

**How we can Edit the ASP.NET Core Project File:**

ASP.NET Core 1.0 does not create a .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 the .csproj file to manage projects. To edit the ASP.NET Core project file, right-click on the project name in the Solution Explorer, and then select “**Edit Project File**” from the context menu as shown in the below image.



Once you click on the “Edit Project File” then the Project file will open in the Visual Studio editor as shown in the below image.



**Understanding the ASP.NET Core Project File:**

Let us understand the ASP.NET Core Project File elements.

**TargetFramework:**

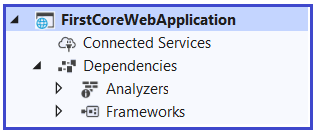
The TargetFramework element in the project file is used to specify the target framework for your application. To specify the target framework in the project file it is using something called Target Framework Moniker (TFM).

In our example, the application targets the framework netcoreapp3.1. The netcoreapp3.1 is the Moniker for .NET Core 3.1. If you remember, we created this project with .NET Core 3.1.

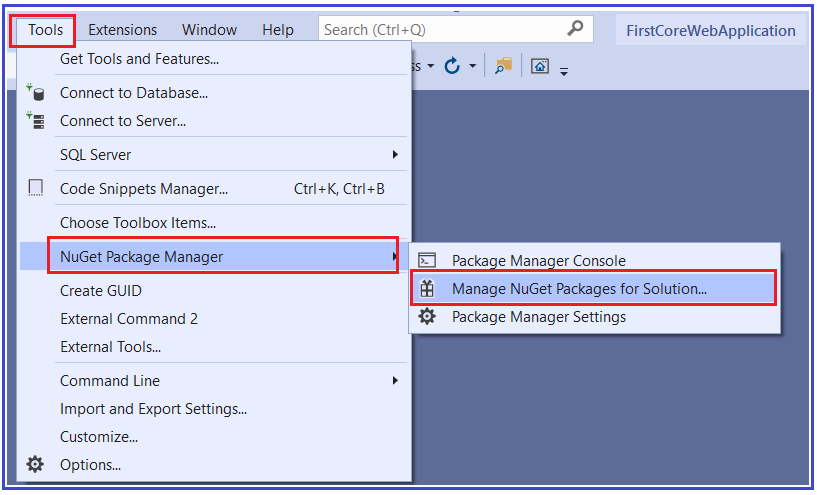
**Adding Packages in ASP.NET Core:**

As we already discussed ASP.NET Core Framework follows modular approaches. It means by default it will not include anything i.e. packages references to the project. Only the necessary packages which are required are added.

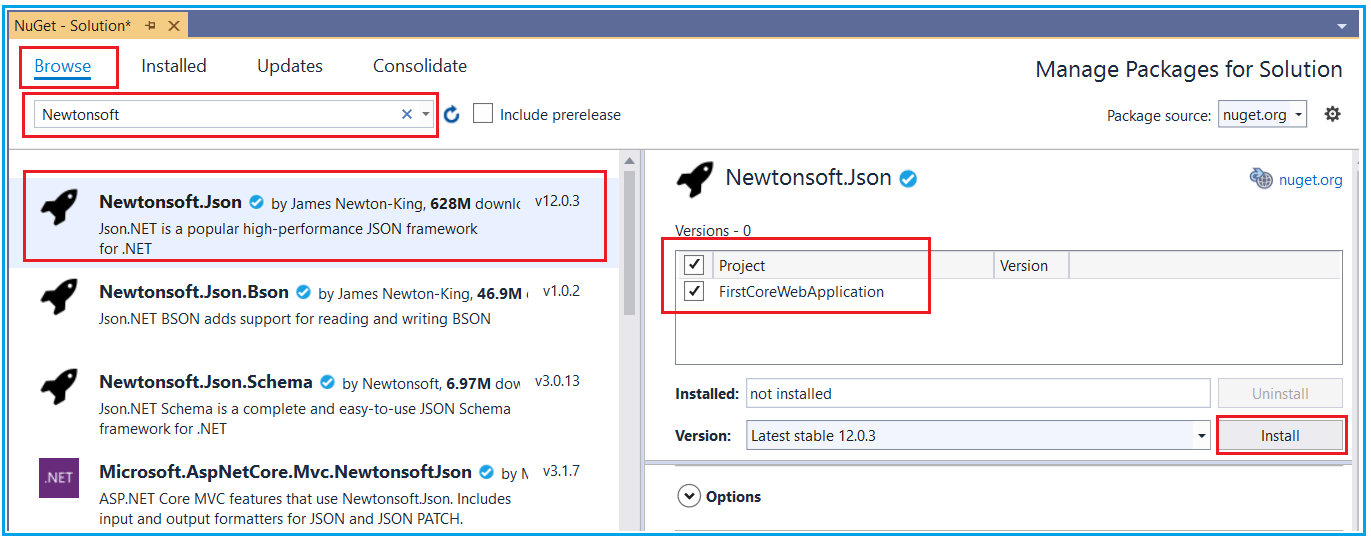
Whenever we add any new packages into our application, then that packages reference is also going to be added into the application project file. Please have a look at the following Dependencies section. Whenever we add any packages, then that package and its dependency packages are going to be stored here.



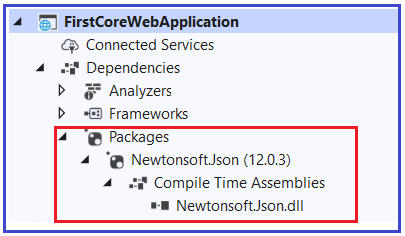
Let us first add a package from the NuGet Package Manager and see what happens. Go to tools => NuGet Package Manager => Manage NuGet Packages for Solution option as shown in the below image.



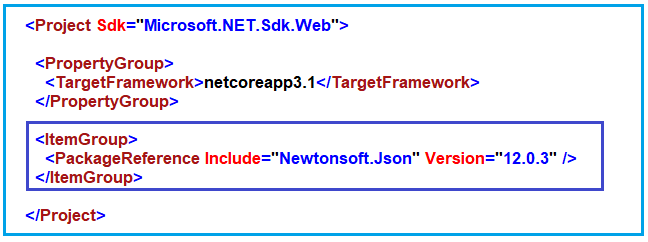
Now let us add the Newtonsoft.json package. So, select the browse tab and then search Newtonsoft and install it as shown in the below image.



Once the package is installed successfully, then you will see, it will add the reference inside the dependencies section as shown in the below image.



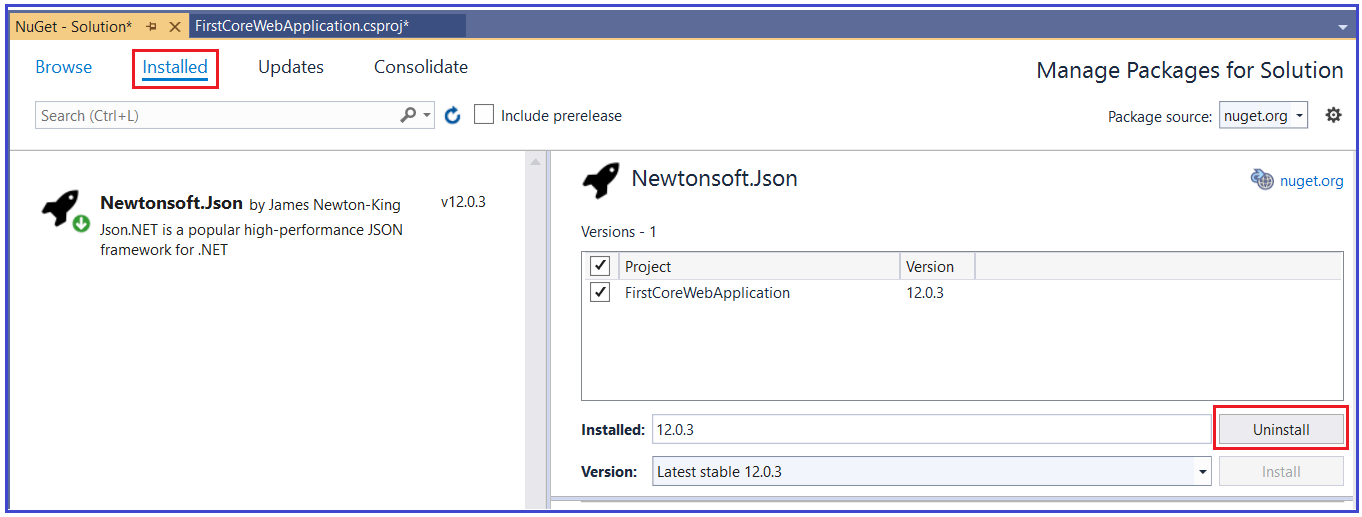
As we have only added one package i.e. Newtonsoft.json, so it added that package references here. Along with the same, it will also add that package reference in the application project file as shown in the below image.



**Note:** Once you delete that package, then it will delete the reference from both the dependencies as well as from the project file.

**Deleting Package in ASP.NET Core:**

Again, go to the Nuget Package Manager and select the installed tab and select the package and uninstall it as shown in the below image.



Once you delete the package, then it should remove the reference from both the dependencies as well as from the project file. The Package Reference element of the project file is used to add the references to all the NuGet packages which are installed for your application.

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

**ASP.NET Core Main Method**

**ASP.NET Core Main Method**

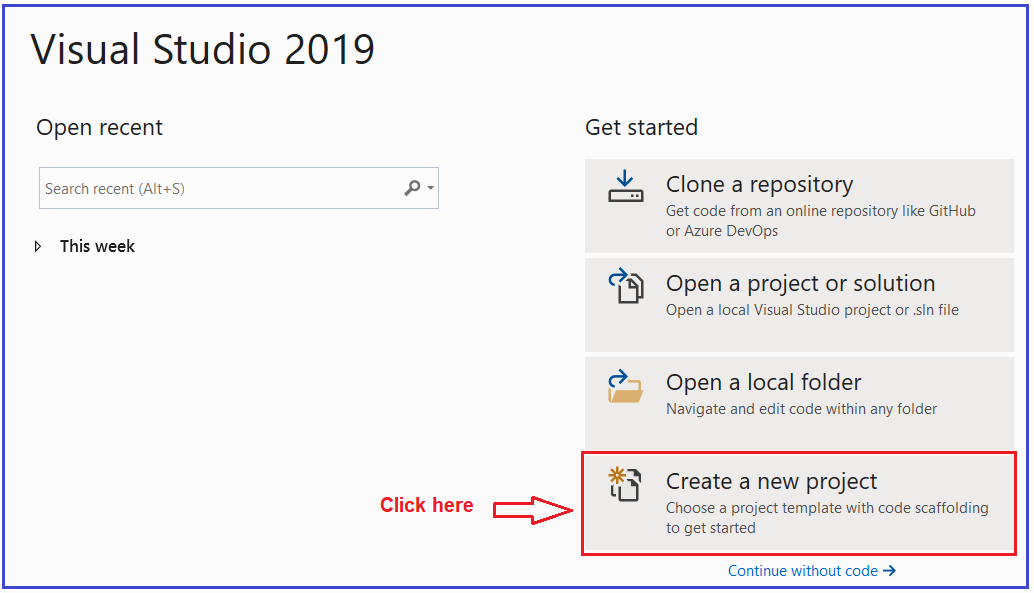
In this article, I am going to discuss the **ASP.NET Core Main Method** in detail. Please read our previous article where we discussed [**ASP.NET Core Project File**](https://dotnettutorials.net/lesson/asp-net-core-project-file/). As part of this article, we are going to discuss the following three important pointers in detail.

1. **The significance of the ASP.NET Core Main Method.**
2. **Why do we have a Main() method in ASP.NET Core?**
3. **What happens behind the scenes when you run a .NET core application?**

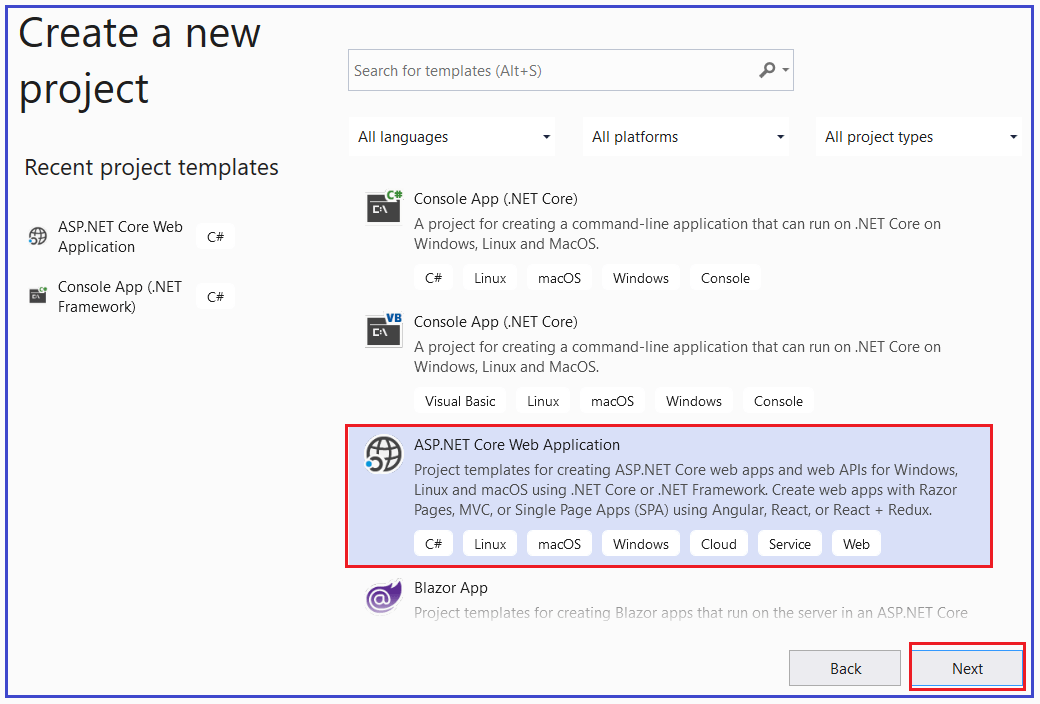
In order to understand the Main method of ASP.NET Core Web application, let’s first create an Empty ASP.NET Core Web application.

**Creating Empty ASP.NET Core Web Application**

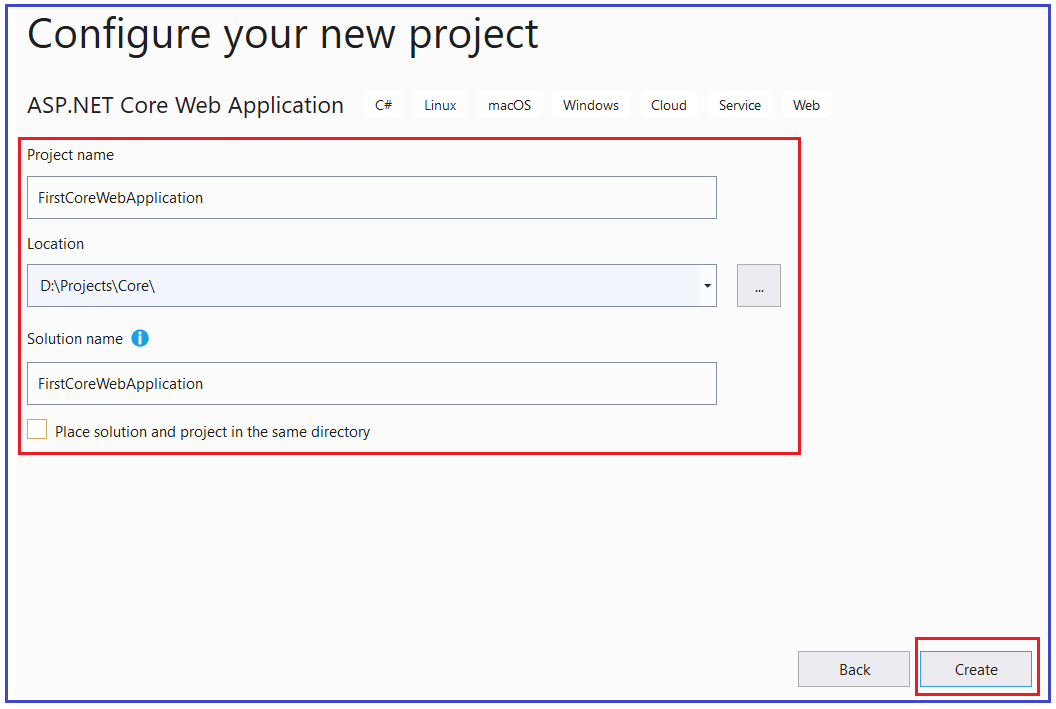
To create a new Empty ASP.NET Core Web Application, Open Visual Studio 2019, and then click on the Create a new project box as shown in the below image.



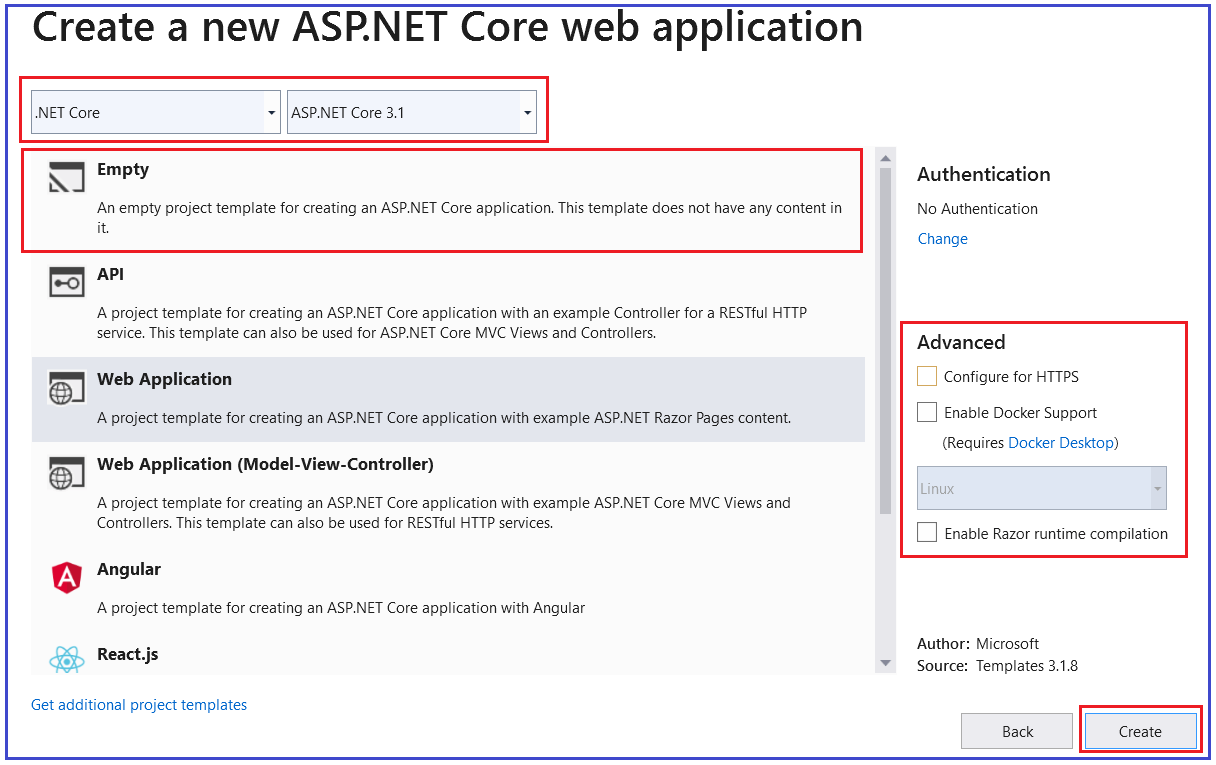
Once you click on the Create a new project box, it will open the Create a new project window. From this window select the **ASP.NET Core Web Application** template and click on the Next button as shown in the below image.



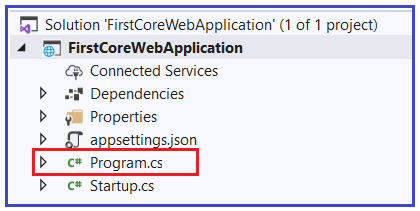
Once you click on the Next button, it will open the following **Configure Your New Project** window. Here, you need to give an appropriate name for your project, set the location where you want to create this project, the solution name for the ASP.NET Core Web application. In this example, we will give the name “**FirstCoreWebApplication**” and click on the Create button as shown in the image below.



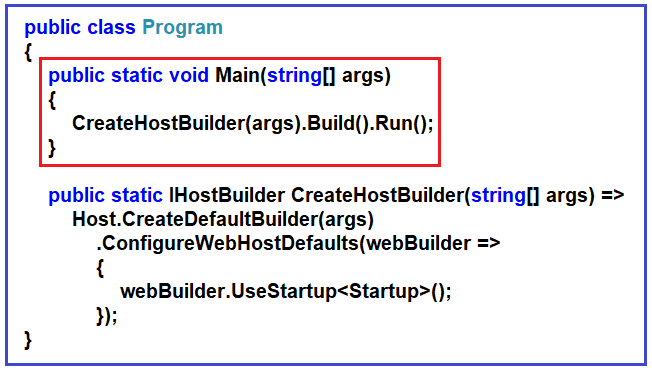
Once you click on the create button, it will open the **Create a new ASP.NET Core Web Application** window. Select the Empty Project template and uncheck all the checkboxes from the Advanced section and finally click on the Create button as shown in the below image.



Once you click on the **Create** button, it will take some time and will create the **Empty ASP.NET Core Web Application** with the following file and folder structure.



As you can see from the above image, we have a class file with the name **Program.cs**. The Program.cs class file of our ASP.NET Core Web Application contains the following code.



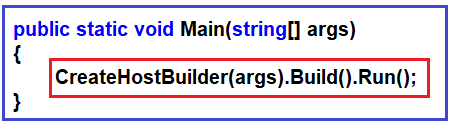
From the above image, you can see that the Program class contains a **public static void Main()**method. As we already know, when we create a console application in .net then by default the .NET Framework creates a class (i.e. Program class) with the Main Method. We also know that the Main method is the entry point for that console application execution.

Now the question is, here we are not creating a console application, here we create an ASP.NET Core Web Application. Then why do we have a Main() method in the ASP.NET Core Web Application?

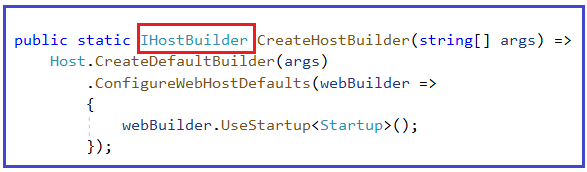
**Why do we have a Main() method in ASP.NET Core?**

The most important point that you need to keep in mind is, the ASP.NET Core Web Application initially starts as a Console Application and the Main() method is the entry point to the application. So, when we execute the ASP.NET Core Web application, first it looks for the Main() methodand this is the method from where the execution starts. The Main() method then configures ASP.NET Coreand starts it. At this point, the application becomes an ASP.NET Core web application.

If you further look at the body of the Main() method, then you will find that it makes a call to the CreateHostBuilder() method by passing the command line arguments args as an argument as shown in the below image.



As shown in the below image, the CreateHostBuilder() method returns an object that implements the IHostBuilder interface. The Host is a static class that can be used for creating an instance of IHostBuilder with pre-configured defaults. The CreateDefaultBuilder() method creates a new instance of the HostBuilder with pre-configured defaults. Internally, it configures Kestrel (Internal Web Server for ASP.NET Core), IISIntegration, and other configurations.

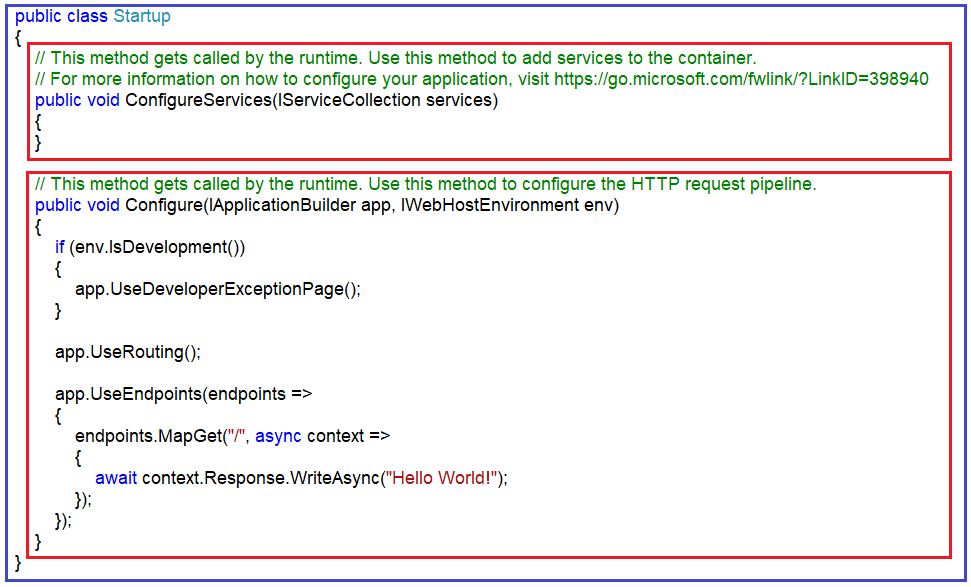


Within the Main() method, on this IHostBuilder object, the Build() method is called which actually builds a web host. Then it hosts our asp.net core web application within that Web Host. Finally, on the web host, it called the Run()method, which will actually run the web application and it starts listening to the incoming HTTP requests.

The CreateHostBuilder() method calls the static CreateDefaultBuilder() method  on the Host class. The CreateDefaultBuilder() method creates a web host with the default configurations. Behind the scene, to create a host, the CreateDefaultBuilder() method does several things. In the next article, we will discuss the [**CreateDefaultBuilder()** **method**](https://dotnettutorials.net/lesson/asp-net-core-inprocess-hosting/) in detail. For now, just understand that the CreateDefaultBuilder() method sets up a web host with default configurations.

**Startup Class**

While setting up the host, the Startup class is also configured using the UseStartup() extension method of the IHostBuilderclass. The Startup class has two methods as shown in the below image.



The ConfigureServices()method of the Startup class configures the services which are required by the application. The Configure() method of the Startup class sets up the pipeline of the application’s request processing. In a later article, we will discuss these two methods in detail.

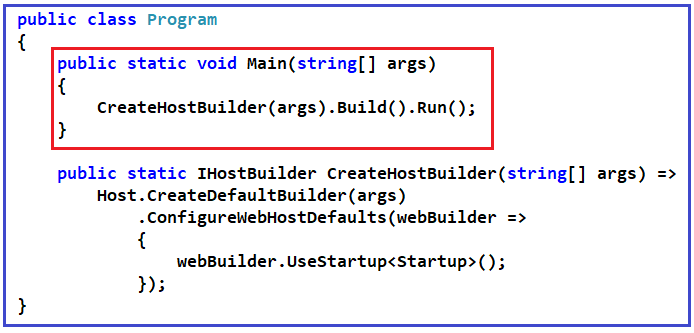
**ASP.NET Core InProcess Hosting**

**ASP.NET Core InProcess Hosting Model**

In this article, I am going to discuss the **ASP.NET Core InProcess Hosting Model** in detail. Please read our previous article before proceeding to this article where we discussed the [**ASP.NET Core Main method**](https://dotnettutorials.net/lesson/asp-net-core-main-method/). As part of this article, we are going to discuss the following pointers in detail.

1. **What are the Tasks performed by CreateDefaultBuilder() method?**
2. **What is InProcess hosting in ASP.NET Core?**
3. **How to Configure InProcess hosting in ASP.NET Core?**
4. **How to host ASP.NET Core Application using InProcess hosting Model?**
5. **What happens when we set the Hosting model as InProcess?**
6. **How does the InProcess hosting work in ASP.NET Core?**
7. **What is IIS Express?**
8. **Why InProcess Hosting Gives Better Performance that the OutOfProcess Hosting Model?**

We are going to work with the same application that we created in our previous article. As we already discussed when we create an ASP.NET Core web application, then the ASP.NET Core framework creates the following Program class with the Main method for us.



When we execute an ASP.NET core application then the .NET Core runtime looks for the Main() method. The Main() method is the entry point for the .net core application to execute.

As you can see from the above image the Main() method of the Program class calls the static CreateHostBuilder() method.  Then the CreateHostBuilder() method calls the static CreateDefaultBuilder()method on the Host class. The CreateDefaultBuilder() method sets the web host which will host our application with default preconfigured configurations.

**What are the Tasks performed by CreateDefaultBuilder() method?**

As part of setting the web host, the CreateDefaultBuilder() method does several things. Some of them are as follows

1. **Setting up the webserver**(will discuss in this article)
2. **Loading the host and application configuration from various configuration sources**(will discuss in our upcoming articles)
3. **Configuring logging**(will discuss in our upcoming articles)

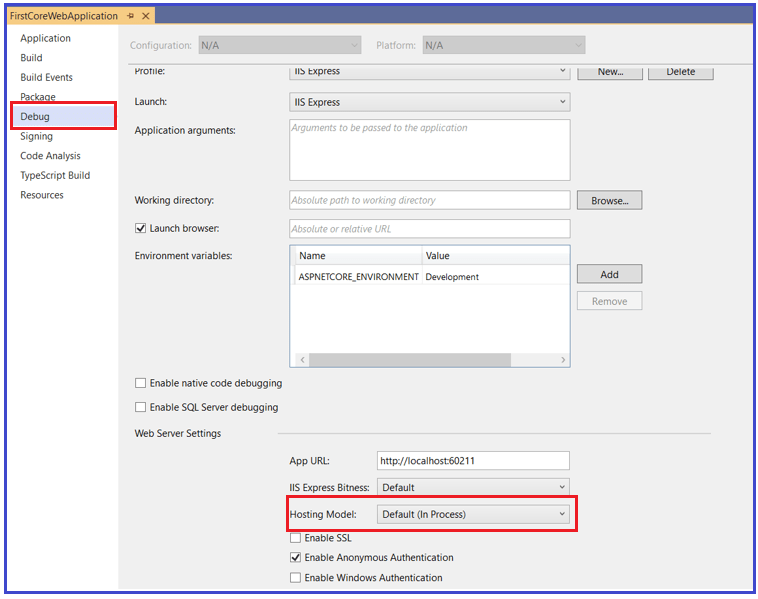
Let us discuss what exactly the CreateDefaultBuilder() method does to configure and set up the webserver. From a hosting point of view, an ASP.NET core Web application can be hosted in two ways i.e. InProcess hosting or OutOfProcess hosting.

Here in this article, we are going to discuss the InProcess hosting model and in a later article, we will discuss the [**OutOfProcess**](https://dotnettutorials.net/lesson/asp-net-core-outofprocess-hosting/)hosting model.

**Note:** When we create a new ASP.NET Core Web application by using any template, by default the project file is created with **InProcess** hosting which is used for hosting the application in IIS or IIS Express scenarios.

**How to verify this?**

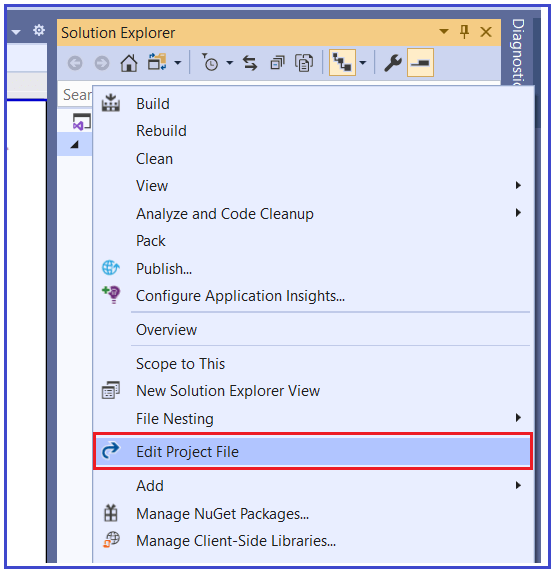
In order to verify the same, open the project properties. Right-click on your project and select the properties option from the context menu. Once you open the properties window, then select the Debug and have a look at the value of the Hosting Model drop-down list as shown in the below image. The drop-down list contains three values i.e. Default (InProcess), InProcess, and Out Of Process. In our upcoming articles, we will discuss the [**Out Of Process hosting model**](https://dotnettutorials.net/lesson/asp-net-core-outofprocess-hosting/).



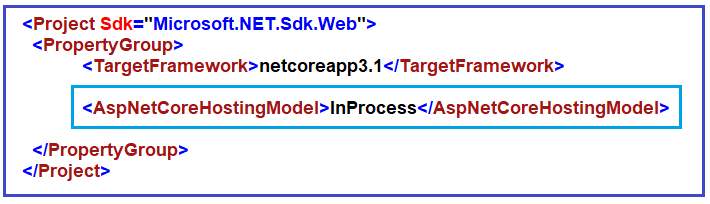
This Confirm that by default it uses In Process hosting model.

**How to Configure InProcess hosting in ASP.NET Core?**

To configure the InProcess hosting for ASP.NET Core Web application there is only one simple setting, just add the **<AspNetCoreHostingModel>** element to the application project file with a value of **InProcess**. To do so, just right-click on your application from the solution explorer and then click on the “**Edit Project File”**option from the context menu as shown in the below code.



Once you open the Application Project file then modify it as shown below. As you can see, here we add **<AspNetCoreHostingModel>** element and set its value to InProcess. The order possible value for this element is OutOfProcess.



**What happens when we set the Hosting model as InProcess?**

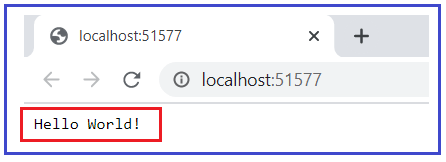
In case of InProcess hosting (i.e. when the CreateDefaultBuilder() sees the value as InProcess for the AspNetCoreHostingModel element in the project file), behind the scene the CreateDefaultBuilder() method internally calls the UseIIS() method. Then host the application inside the IIS worker process (i.e. w3wp.exe for IIS and iisexpress.exe for IISExpress).

From the performance point of view, the InProcess hosting model delivers significantly higher request throughput than the OutOfProcess hosting model. Why we will discuss this at the end of this article.

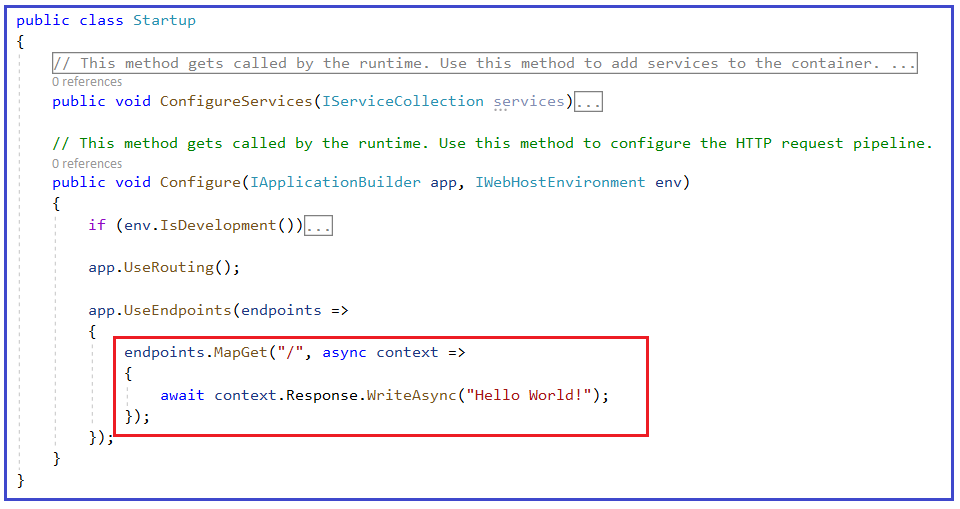
The process name that will be used to execute the application is w3wp in the case of IIS**.**Similarly, if it is IIS Express then the process name will be iisexpress**.**

**Let’s Confirm this:**

Now if you will run the application, then you will see the following hello world message in the browser.



You are getting the above message from the following Startup class



To display the process name in the browser you need to use the **System.Diagnostics.Process.GetCurrentProcess().ProcessName** within the Startup as shown below. In our upcoming article, we will discuss the Startup class in detail. For now, just copy-paste code.

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Startup

**{**

// This method gets called by the runtime. Use this method to add services to the container.

// For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=398940

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

**}**

// 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**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapGet**(**"/", **async** context =**>**

**{**

**await** context.Response.WriteAsync**(**"Worker Process Name : " + System.Diagnostics.Process.GetCurrentProcess**()**.ProcessName**)**;

**})**;

**})**;

**}**

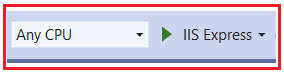
**}**

**}**

Now when we run the application from visual studio, and it should display the message in the browser as shown below.

IIS Express in ASP.NET Core

This is because by default Visual Studio uses IISExpress when we run an application as shown in the below image.



**What is IIS Express?**

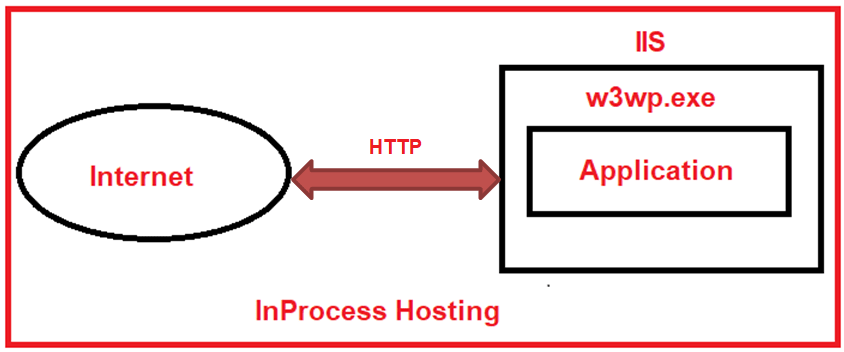
The IIS Express is a lightweight, self-contained version of IIS. It is optimized for web application development. The most important point that you need to remember is we use IIS Express only in development, not on production. In production we generally use IIS. In a later article, we will discuss how to deploy an ASP.NET Core application on IIS.

**Why InProcess Hosting Gives Better Performance that the OutOfProcess Hosting Model?**

In the case of **OutOfProcess** hosting, there are 2 web servers

1. An internal web server and
2. One external web server.

The internal web server is called [**Kestrel**](https://dotnettutorials.net/lesson/kestrel-web-server-asp-net-core/)and the external web server can be **IIS**, **Nginx,** or **Apache**. With the InProcess hosting model, there is only one web serveri.e. IIS. So, in the case of the InProcess hosting model, we do not have the performance penalty for navigating the requests between the internal and external web servers. This is the reason why the InProcess hosting model delivers significantly higher request throughput than the OutOfProcess hosting model.



**Kestrel Web Server in ASP.NET Core**

**Kestrel Web Server in ASP.NET Core Application**

In this article, I am going to discuss the **Kestrel Web Server in ASP.NET Core** Application. Please read our previous article before proceeding to this article where we discussed [**ASP.NET Core InProcess Hosting**](https://dotnettutorials.net/lesson/asp-net-core-inprocess-hosting/) Model. At the end of our previous article, we discussed that with the OutOfProcess hosting model, there are 2 web servers i.e. one internal web server and one external web server. The internal web server is called Kestreland the external web server can be IIS, Apache, or Nginx. As part of this article, we are going to discuss the following two important concepts in detail.

1. **What is a Kestrel Web Server?**
2. **How to Configure Kestrel Web Server?**
3. **How to run a .NET Core application using Kestrel Web Server?**
4. **How to run a .NET Core Application using .NET Core CLI.**

**What is a Kestrel Web Server?**

As we already discussed ASP.NET Core is a cross-platform framework. It means it supports to develop and run applications on different types of operating systems such as Windows, Linux, or Mac.

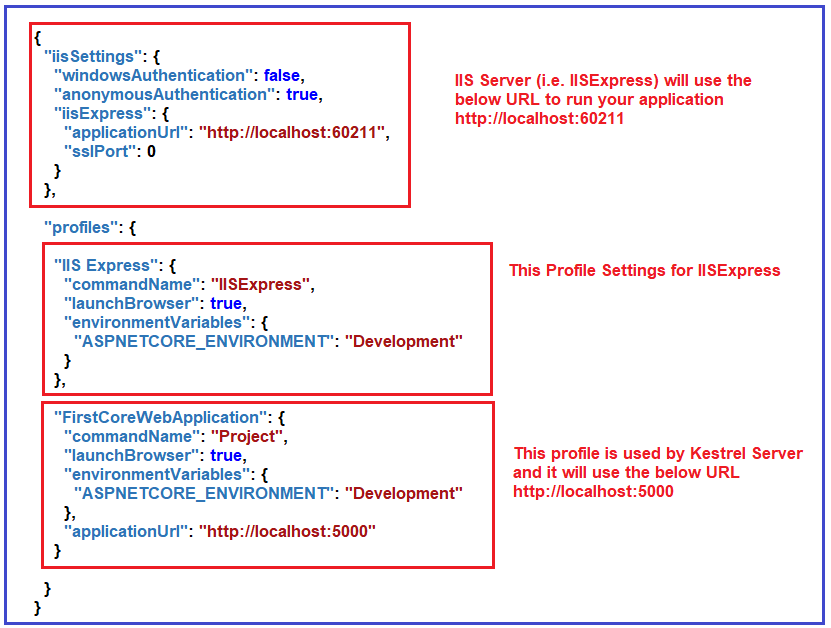
The Kestrel is the cross-platform web server for the ASP.NET Core application. That means this Server supports all the platforms and versions that the ASP.NET Core supports. By default, it is included as the internal web server in the .NET Core application.

The Kestrel Web Server generally used as an edge server i.e. the internet-facing web server which directly processes the incoming HTTP request from the client. In the case of the Kestrel web server, the process name that is used to host and run the ASP.NET Core application is the project name.

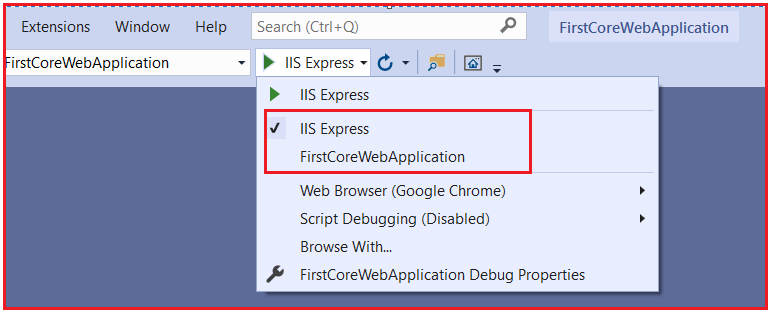
As of now, we are using visual studio to run the ASP.NET Core application. By default, the visual studio uses **IIS Express** to host and run the ASP.NET Core application. So, the process name is **IIS Express** that we already discussed in our previous article.

**How to run the application using Kestrel Web Server?**

Before using the Kestrel server to run our application, let us first open the [**launchSettings.json**](https://dotnettutorials.net/lesson/asp-net-core-launchsettings-json-file/) file which is present inside the Properties folder of your application. Once you open the launchSettings.json file you will find the following code by default.



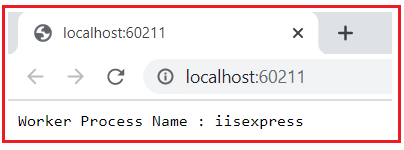
In our upcoming article, we will discuss[**launchSettings.json**](https://dotnettutorials.net/lesson/asp-net-core-launchsettings-json-file/) in detail. But for now, just have a look at the Profiles section. Here, you can see, we have two sections. One is for IIS Express (IIS Server) and the other one is for the Kestrel server. In visual studio, you can find the above two profiles (IIS Express and FirstCoreWebApplication) as shown below.



If you select IIS Express then it will use the IIS server and if you select FirstCoreWebApplication, then it will use Kestrel Server.

**Running the application using IIS Express:**

If you run the application using IIS Express, then it will use the URL and port number mentioned in the iisSettings of your launchSettings.json file. To prove this run the application using IIS Express and see the output as shown below.

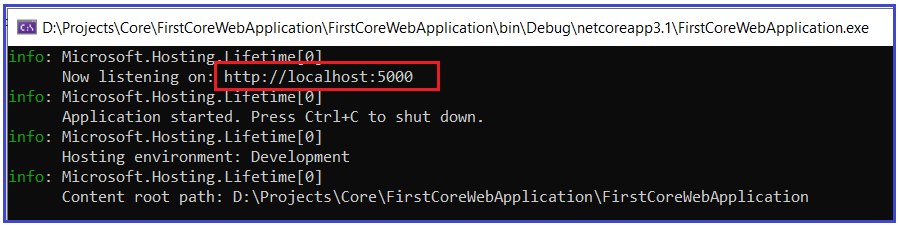


**Running the application using Kestrel Server:**

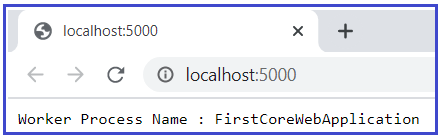
In order to use the Kestrel server to run your application in Visual Studio, first, you need to select the FirstCoreWebApplication profile as shown below.

Running the application using Kestrel Server

Once you select the FirstCoreWebApplication, now run the application. Here, we need to observe two things. First, it will launch the command prompt and host the application using the Kestrel server as shown below. Here, you need to focus on the URL and port number and it should be the URL and port number mentioned in your FirstCoreWebApplication profile of launchSettings.json file.



Secondly, it opens the default browser and listening to that URL and Port Number as shown below.



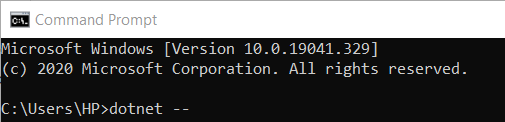
**Note:** In our example, for IIS Express the port number is 60211, and worker process is iisexpress while for Kestrel server the port number is 5000 and the worker process name is FirstCoreWebApplication (It is nothing but your application name).

**How to run .NET Core application using .NET Core CLI?**

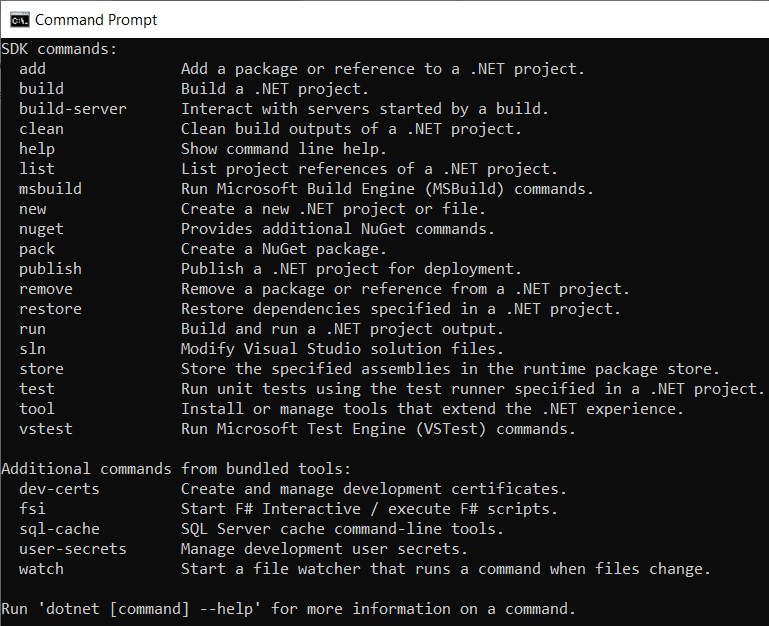
We can also run the ASP.NET Core application from the command line using the .NET Core CLI. The CLI stands for Command Line Interface.

When we run an ASP.NET Core application using the .NET Core CLI, then the .NET Core runtime uses Kestrel as the webserver. We will discuss the .NET Core CLI in detail in our upcoming article. Now, let us see how to run a dot net core application using .NET Core CLI Command.

First. open the command prompt and type “**dotnet —**” and press enter as shown below.



Once you type the “**dotnet —**” and click on the enter button then you will find lots of commands as shown below.



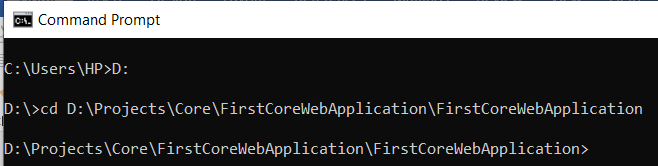
**Using the CLI (above commands)**

You can create a new project using the **new** command, you can also build the project using the **build** command, or you can publish the project using the **publish** command. It is possible to restore the dependencies and tools which are required for a .net core project using the CLI.

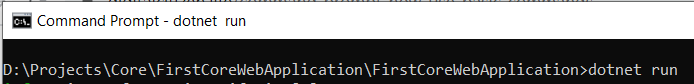
**Running .NET Core application using .NET Core CLI**

Let’s see how to run a .NET Core application using .NET Core CLI command. To do so please follow the below steps

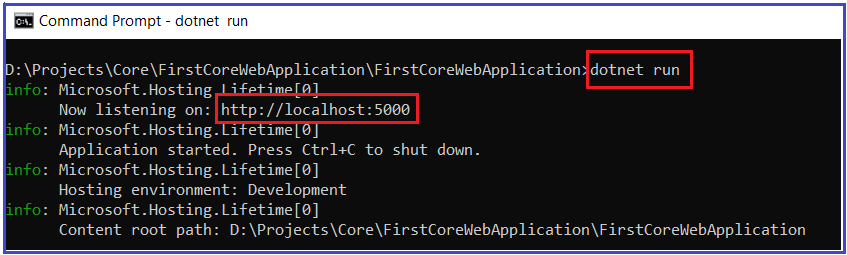
First, open the Windows Command Prompt. To do so, open the run window and then type cmd and click on the enter button which will open the command prompt. Then you need to change the directory to the folder which contains your asp.net core application. My project is present in the “**D:\Projects\Core\FirstCoreWebApplication\FirstCoreWebApplication**” folder so I change the current directory to my project file by using the following command.



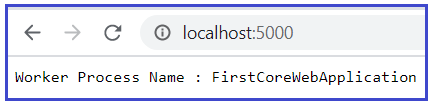
Once you change the directory to your project folder, then execute the “**dotnet run**” command as shown in the below image.



Once you type the **dotnet run** command, press the enter key, then the .NET Core CLI builds and runs the application. It also shows the URL and you can use this URL to access your application as shown in the below image.

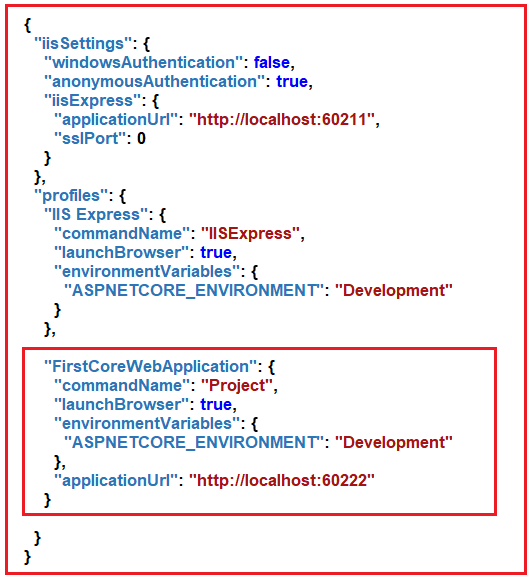


.Here, in my case, the application is available at [**http://localhost:5000**](http://localhost:5000/)**.**If you remember this port is configured in the launchSettings.json file of your application inside the FirstCoreWebApplication profile which is nothing but the profile for the Kestrel server. Now open the browser and navigate to the **http://localhost:5000**URLand It should display the worker process name as **dotnet** as shown below.



**Changing the Port Number:**

If you want then you can also change the Port number for Kestrel Server. To do so open the launchSettings.json file and give any available Port number as shown below. Here, I am changing the Port number to 60222.



Now, save the changes and run the application using Kestrel Server and you should see the changed port number in the URL.

**ASP.NET Core OutOfProcess Hosting**

**ASP.NET Core Out of Process Hosting**

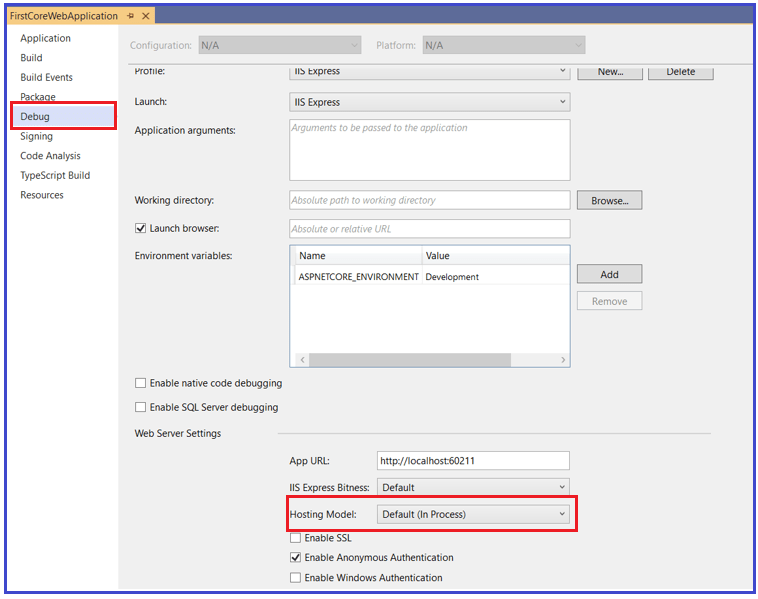
In this article, I am going to discuss **ASP.NET Core OutOfProcess** **Hosting** Model in detail. I strongly recommended you to read [**ASP.NET Core InProcess Hosting**](https://dotnettutorials.net/lesson/asp-net-core-inprocess-hosting/) and [**Kestrel Web Server in ASP.NET Core Application**](https://dotnettutorials.net/lesson/kestrel-web-server-asp-net-core/) articles before proceeding to this article. As part of this article, we are going to discuss the following pointers in detail.

1. **How to Configure OutOfProcess Hosting in ASP.NET Core?**
2. **What is ASP.NET OutOfProcess Hosting?**
3. **How does the OutOfProcess Hosting work in ASP.NET Core?**
4. **Can we run an asp.net core application without using the built-in kestrel web server?**
5. **If Kestrel can be used by itself as a web server which can directly handle and process the incoming HTTP Request, then why do we need a reverse proxy server?**
6. **What happens when we run the ASP.NET Core application using .NET Core CLI?**
7. **Can we run an asp.net core application without using the built-in kestrel web server?**

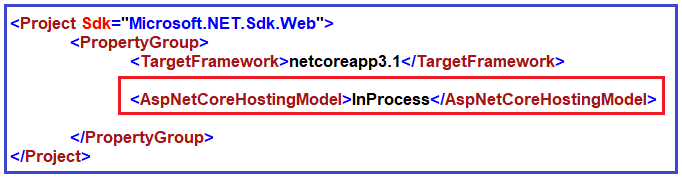
Before understanding the OutOfProcess hosting, let us first have a look at the InProcess hosting model and understand how it works.

**ASP.NET Core InProcess Hosting**

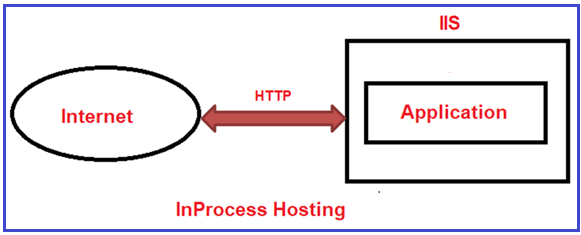
Let first have a look at InProcess Hosting before proceeding to Out of Process Hosting. As we already discussed, by default Visual Studio uses InProcess Hosting. You can verify this using the Debug of your project properties. Right-click on your project in the solution explorer and click on the Properties option from the context menu which will open the Project Properties window. In this window, select the Debug tab and have a look at the Hosting Model value as shown in the below image.



Even though if you want then you can also configure the same (i.e. InProcess hosting) in the Application Project file by setting the value to InProcess of the **<AspNetCoreHostingModel>** element as shown in the below image.



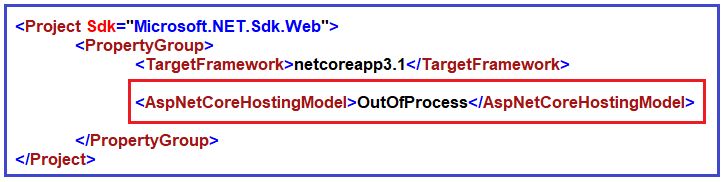
In ASP.NET Core, with InProcess Hosting Model our application is going to be hosted in the IIS worker process. The most important point that you need to remember is we have only one web server i.e. IIS Server in case of InProcess hosting which is going to host our application as shown in the below image.



**How to Configure the OutofProcess Hosting in ASP.NET Core Application?**

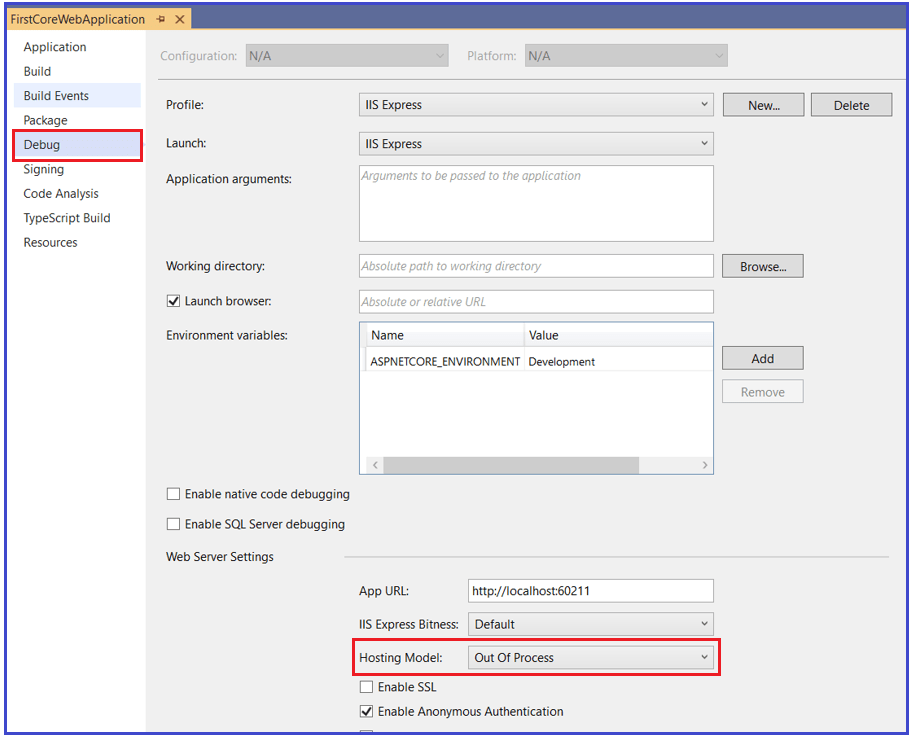
We can configure the Out of Process Hosting in two ways in ASP.NET Core.

**Way1:**In this case, you just need to add the <AspNetCoreHostingModel> element to the applications project file with a value of OutOfProcess as shown below.



**Way2:  Specify the Hosting Model as OutOfProcess in the Project Properties**

To do so, right-click on your project in the solution explorer and click on the Properties option from the context menu which will open the Project Properties window. In this window, select the Debug tab and then select the Hosting Model value as OutOfProcess as shown in the below image.



**What is Out of Process Hosting in ASP.NET Core?**

In the case of the ASP.NET Core OutOfProcess Hosting Model, there are two web servers.

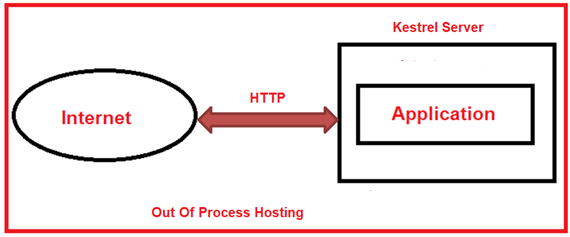
1. An internal webserver which is the Kestrel web Server
2. And an external web server which can be IIS, Apache, and Nginx.

The most important point that you need to keep in mind is depending on how you are running your application with the OutOfProcess hosting model, the external web server may or may not be used.

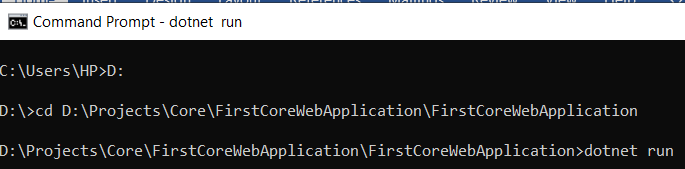
As we already discussed that the Kestrel web server is a cross-platform web server that is already embedded with your ASP.NET Core application. So, if you are using Out of Process Hosting model for your asp.net core application, then the Kestrel web server can be used in one of the following ways.

**Way1:**

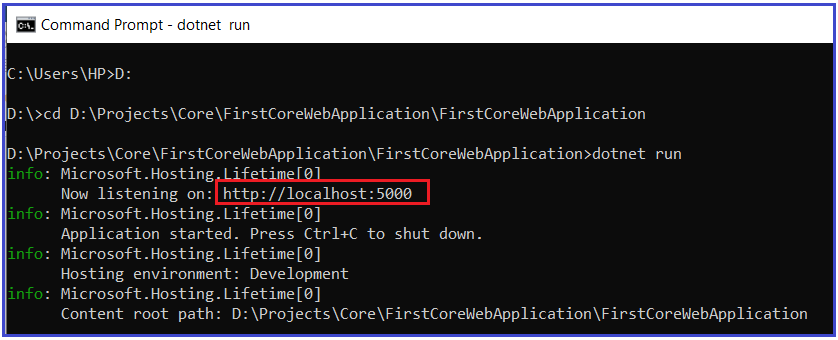
We can use the Kestrel Web Server as the internet-facing web server which will directly process the incoming HTTP requests. In this scenario, only the Kestrel Server is used and the other one i.e. external web server is not going to be used. So, when we run the application using the .NET core CLI then Kestrel is the only web server that is going to be used to handle and process the incoming HTTP request as shown in the below image.



To confirm this, open the command prompt and run the application as shown in the below image. As my First, you need to change the directory to the folder which contains your asp.net core application. My project is present in the “**D:\Projects\Core\FirstCoreWebApplication\FirstCoreWebApplication**” folder so I change the current directory to my project file and then use the **dotnet run** command.



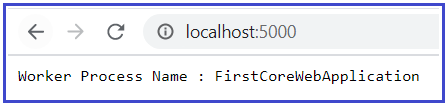
Once you type **dotnet run** command and press the enter key, it will build, host, and run the application as shown in the below image.



Now open the browser window and navigate to the following URL which is shown in the command prompt. In my case, the following is the.

**http://localhost:5000**

And here you will see the worker process name as your application name as shown below



So, in this case, Kestrel is the only server that will handle and process the incoming HTTP Request. The following code of the Startup class displays the worker process name.

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Startup

**{**

// This method gets called by the runtime. Use this method to add services to the container.

// For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=398940

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

**}**

// 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**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapGet**(**"/", **async** context =**>**

**{**

**await** context.Response.WriteAsync**(**"Worker Process Name : " + System.Diagnostics.Process.GetCurrentProcess**()**.ProcessName**)**;

**})**;

**})**;

**}**

**}**

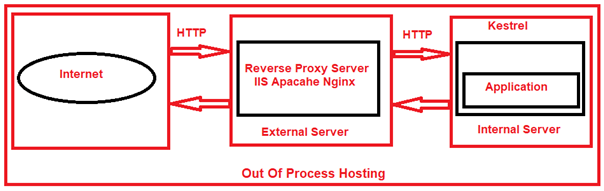
**}**

**Way2:**

The Kestrel Web Server can also be used with the combination of a reverse proxy server such as IIS, Apache, or Nginx. Now the question that should come to your mind is, If Kestrel can be used by itself as a web server which can directly handle and process the incoming HTTP Request, then why do we need a reverse proxy server?

This is because the reverse proxy server provides an additional layer of configuration and security which is not available with the Kestrel Server. It also maintains the load balancing. So, it is a good choice to use Kestrel Server along with a reverse proxy server.

So, when we use Kestrel Server along with a reverse proxy server, then the reverse proxy server will receive the incoming HTTP requests from the client and then forwards that request to the Kestrel server for processing. Once the Kestrel Server process that request, then it sends the response back to the reverse proxy server which then sends the response back to the requested client over the internet as shown in the below image.



In our upcoming articles, we will discuss how to deploy an ASP.NET Core application to IIS and how we can use IIS as a reverse proxy server.

When we run the application directly from Visual Studio, then by default Visual Studio uses IIS Express. Now change the AspNetCoreHostingModel element value as shown below in the application’s project file.

**<AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>**

As we have configured the Out of Process hosting model, now the IIS Express acts as a reverse proxy server and Kestrel acts as the internal webserver.

Now, the IIS Express receives the incoming HTTP request and then forwards it to the Kestrel Web Server for processing. The Kestrel Web Server processes the request and sends the response back to the IIS Express which in turn sends the response back to the client i.e. to the browser.

Now run the application, and you will see the worker process as your project name. So, when you are using Out of Process Hosting model, then the Kestrel Web Server is going to hosts the application and process the request irrespective of whether you are using a reverse proxy server or not.

**What happens when we run the ASP.NET Core application using .NET Core CLI?**

When we are running our application using.NET Core CLI, then by default, it ignores the hosting setting that you specified in the application’s project file i.e. csproj file. So, in that case, the value of the AspNetCoreHostingModel element is going to be ignored.

The .NET Core CLI always uses OutOfProcess Hosting Model and Kestrel is the webserver that will host the ASP.NET Core application and also handles the HTTP requests.

**Can we run an asp.net core application without using the built-in kestrel web server?**

YES. When we use the InProcess Hosting model, then the application is hosted inside the IIS worker process i.e. w3wp.exe in the case of IIS and iisexpress.exe in the case of IIS Express. That means the Kestrel Web Server is not used with the InProcess hosting model.

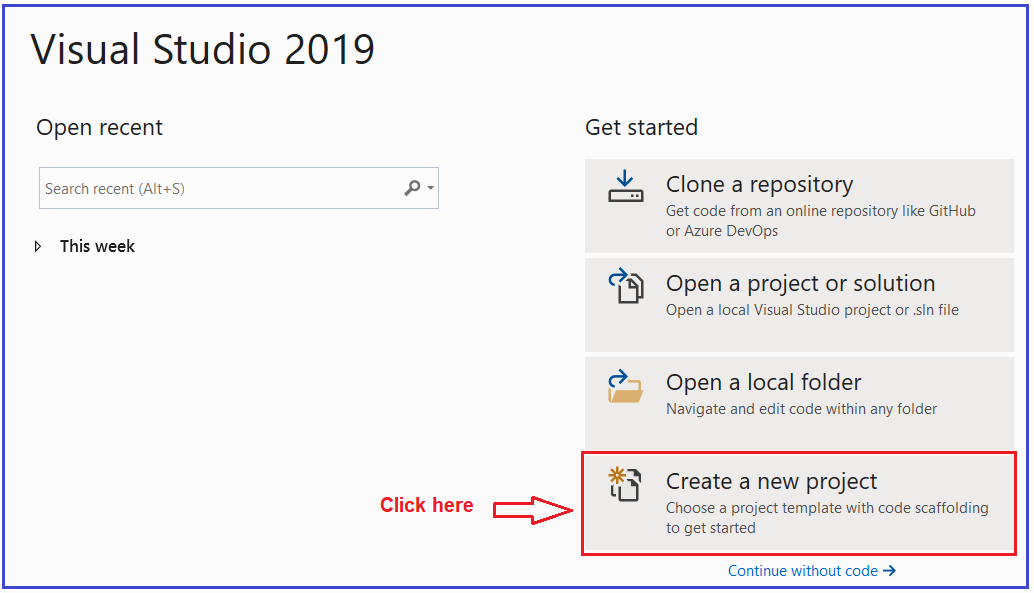
**ASP.NET Core launchSettings.json file**

**ASP.NET Core launchSettings.json file**

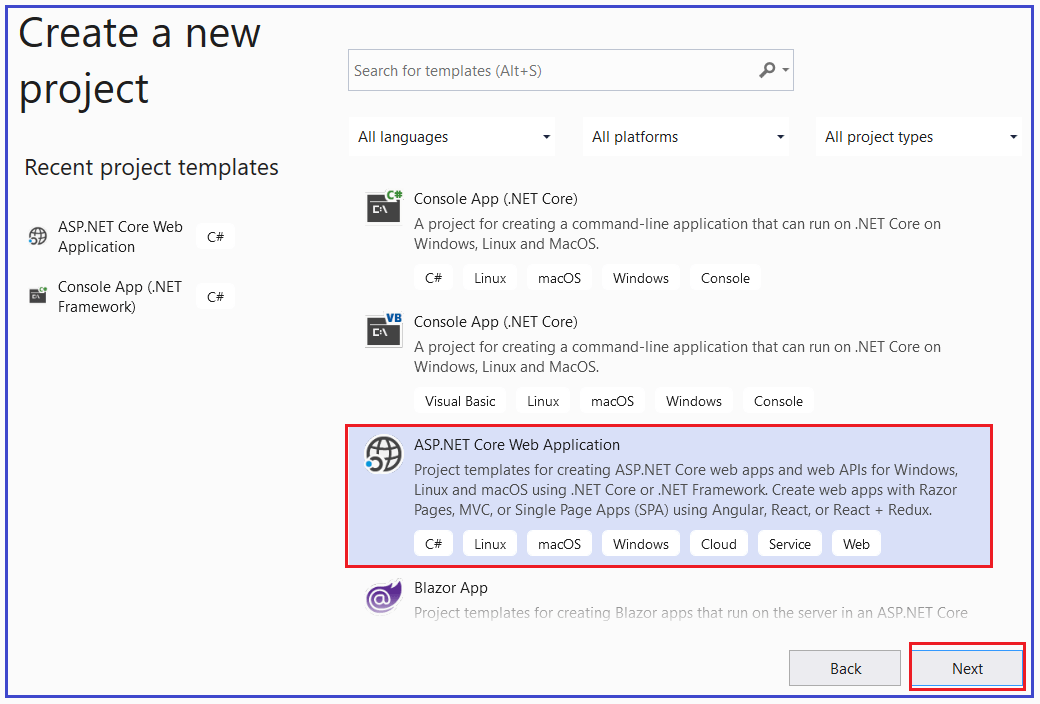
In this article, I am going to discuss the use and importance of the ASP.NET Core launchSettings.json file in detail. Please read our previous article where we discussed the [**OuOfProcess Hosting Model**](https://dotnettutorials.net/lesson/asp-net-core-outofprocess-hosting/) in ASP.NET Core Web Application. In order to understand the ASP.NET Core launchSettings.json file, let us first create a new ASP.NET Core application with an empty template.

**Creating a new Empty ASP.NET Core Web Application**

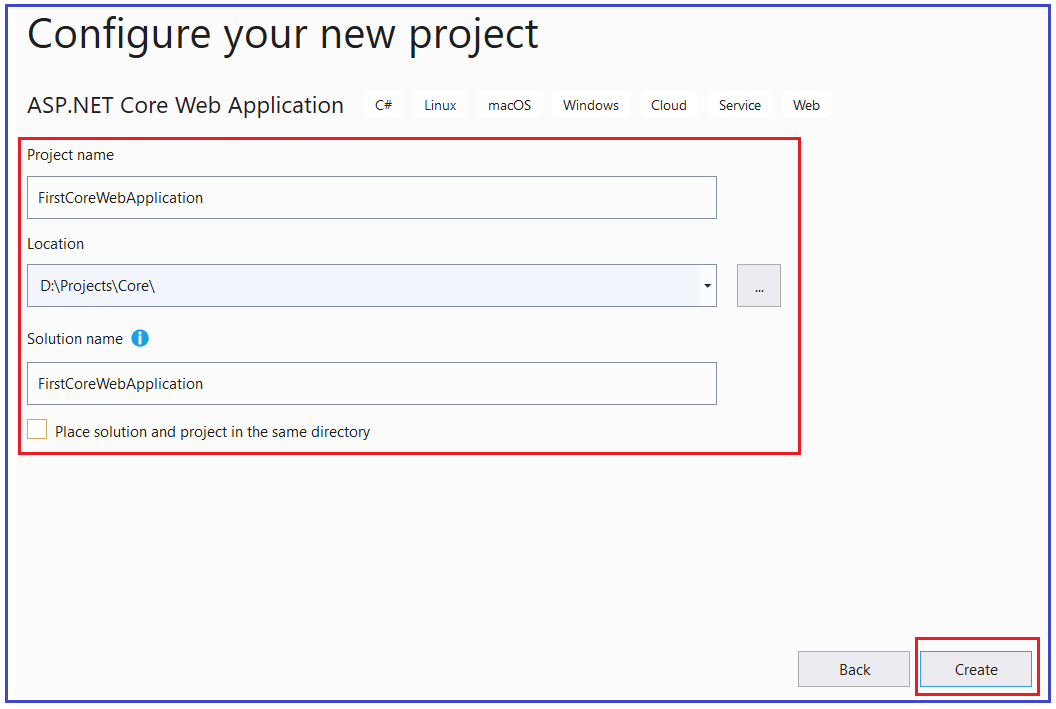
First, open Visual Studio 2019 and then click on the “Create a new project” box as shown in the below image.



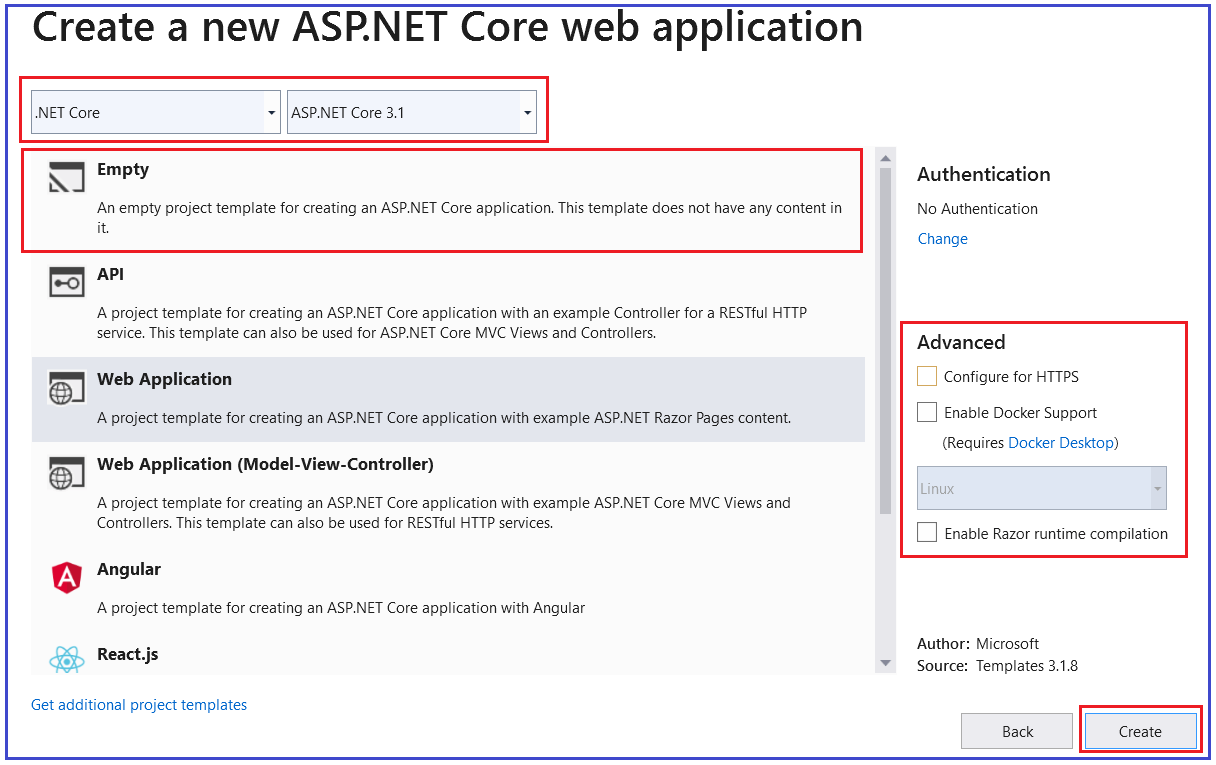
Once you click on the Create a new project box, then it will open the Create a new project window. From this window, you need to select the **ASP.NET Core Web Application** template and then click on the Next button as shown in the below image.



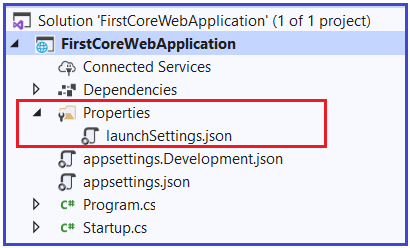
Once you click on the Next button, then it will open the “Configure Your New Project” window. Here, you need to give the name for your project, set the location of your project, give the solution name. In this example, I will give the name “**FirstCoreWebApplication**” and then click on the Create button as shown in the image below.



Once you click on the create button, then it will open the Create a new ASP.NET Core Web Application window. From this window, you need to select the Empty Project template and uncheck all the checkboxes from the Advanced section. Please make sure to select .NET Core and ASP.NET Core 3.1 from the respective dropdown list and finally click on the Create button as shown in the below image.



Once you click on the Create button, it will take some time and will create the Empty ASP.NET Core Web Application with the following file and folder structure.



As you can see from the above image, the ASP.NET Core Project has a file called **launchSettings.json** within the Properties folder. So, let us discuss the need and importance of this **launchSettings.json** file in the ASP.NET Core application.

**Understanding LaunchSettings.json file in ASP.NET Core**

The settings that are present within this file are going to be used when we run the .NET core application either from Visual Studio or by using .NET Core CLI.

The most important point that you need to keep in mind is this launchSettings.json file is only used within the local development machine. That means this file is not required when we publishing our asp.net core application to the production server.

If you have certain settings and you want your application to use such settings when you publish and deploy your asp.net core application to the production server, then you need to store such settings in the appsettings.json file. Generally, in the ASP.NET Core application, the configuration settings are going to be stored in the appsettings.json file. In our upcoming article, we will discuss the [appsettings.json file](https://dotnettutorials.net/lesson/asp-net-core-appsettings-json-file/)in detail.

**ASP.NET Core Application Profile settings in the launchSettings.json file:**

If you open the launchSettings.json file, then by default you will find the following code or you can say settings within that file in ASP.NET Core 3.1 applications.

**{**

"iisSettings": **{**

"windowsAuthentication": **false,**

"anonymousAuthentication": **true,**

"iisExpress": **{**

"applicationUrl": "http://localhost:50409"**,**

"sslPort": 0

**}**

**},**

"profiles": **{**

"IIS Express": **{**

"commandName": "IISExpress"**,**

"launchBrowser": **true,**

"environmentVariables": **{**

"ASPNETCORE\_ENVIRONMENT": "Development"

**}**

**},**

"FirstCoreWebApplication": **{**

"commandName": "Project"**,**

"launchBrowser": **true,**

"applicationUrl": "http://localhost:5000"**,**

"environmentVariables": **{**

"ASPNETCORE\_ENVIRONMENT": "Development"

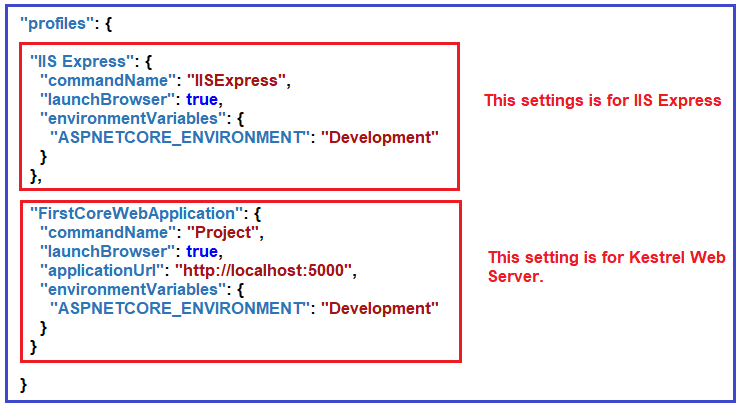
**}**

**}**

**}**

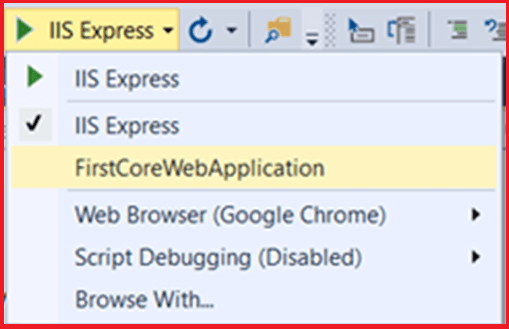
**}**

In the above **launchSettings.json** file, within the profiles, we have two sections i.e. IIS Expressand FirstCoreWebApplication as shown in the below image.



The point that you need to remember is when you run the application from Visual Studio either by pressing **CTRL + F5** or just **F5** then by default the profile with **“commandName”: “IISExpress”** is going to be used. On the other hand, if you run the ASP.NET Core application using .NET Core CLI (i.e. dotnet run command), then the profile with the **“commandName”: “Project”** is going to be used.

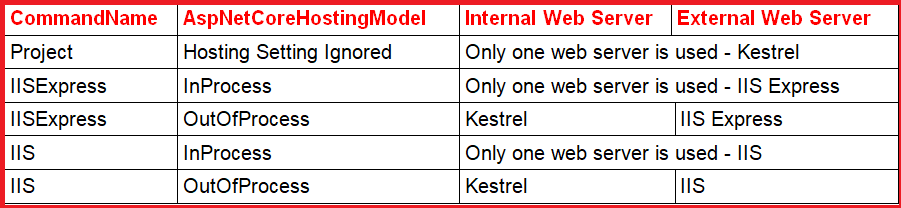
However, if you want then you can choose which profile to use when you run the application by pressing **CTRL + F5** or just **F5,** by clicking on the drop-down list in Visual Studio as shown below



The value of the commandNameproperty of the launchSettings.json file can be any one of the following.

1. **IISExpress**
2. **IIS**
3. **Project**

The CommandName property value of the launchSettings.json file along with the AspNetCoreHostingModelelement value from the application’s project file will determine the internal and external web server (reverse proxy server) that are going to use and handle the incoming HTTP Requests. For better understanding, please have a look at the below table.



**Modifying the Configure method of Startup class**

Modify the Configure method of the Startup class file as shown below to display the Name of the worker process that is going to handle the request in the browser window.

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapGet**(**"/", **async** context =**>**

**{**

**await** context.Response.WriteAsync**(**"Worker Process Name : " +

System.Diagnostics.Process.GetCurrentProcess**()**.ProcessName**)**;

**})**;

**})**;

**}**

**Case1: CommandName as Project**

When we use the CommandName as Project in the launchSettings.json file, then ASP.NET Core is going to ignore the AspNetCoreHostingModel value. The Kestrel is the only server that is going to host the application and handle the incoming request. Let’s prove this. Now, we need to set the launch Profile as **FirstCoreWebApplication** as shown below.



If you look at the launchSettings.json file, then you will see that the **FirstCoreWebApplication** profile uses the **“commandName”: “Project”** value, and as well as please focus on the **application URL** as shown below. In my application the URL is **http://localhost:5000**and the port number might be varying in your example.



Now change the AspNetCoreHostingModel element value to InProcess in the application’s project file as shown below.

**<Project** Sdk="Microsoft.NET.Sdk.Web"**>**

**<PropertyGroup>**

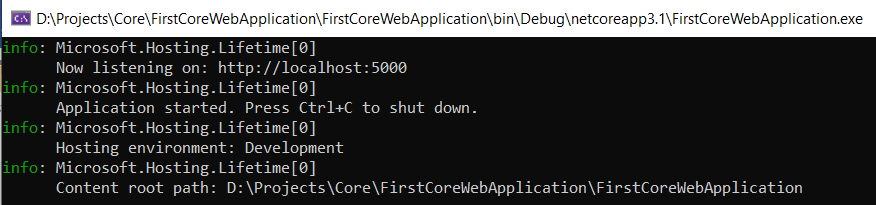
**<TargetFramework>**netcoreapp3.1**</TargetFramework>**

**<AspNetCoreHostingModel>**InProcess**</AspNetCoreHostingModel>**

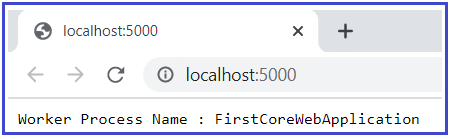
**</PropertyGroup>**

**</Project>**

Now, if you run the project either by pressing CTRL + F5 or just F5, first it will launch the command prompt where it will host the application using the Kestrel server as shown below.



Once it launches the command prompt and hosts the application, then by default it opens your default browser and displays the worker process name as the project name i.e. FirstCoreWebApplication as shown in the below image. This is because when the CommandName value is Project then it ignores the AspNetCoreHostingModel value and Kestrel is the only server that is going to host and process the incoming requests.

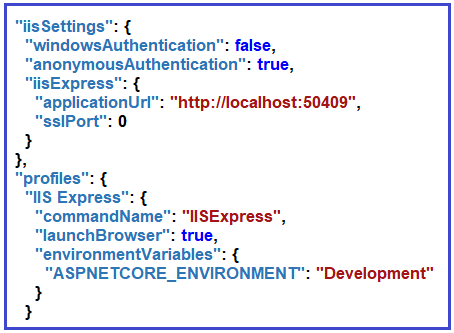


**Case2: CommandName as IISExpress and AspNetCoreHostingModel as InProcess**

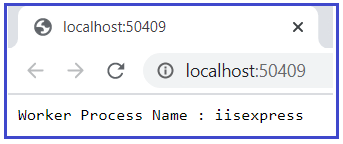
If we use the CommandName as IISExpress profile and if we set the AspNetCoreHostingModel value as InProcess in the applications project file, then IIS Express is the only server that is going to host and handle the incoming HTTP request. Let us prove this. First, use IIS Express as the lunch profile by selecting IIS Express from the drop-down list as shown below.

CommandName as IISExpress and AspNetCoreHostingModel as InProcess

Now, if you look at the **launchSettings.json** file, then you will see that the **IIS Express** profile use the “commandName”: “IISExpress” value and also please keep the focus on the application URL which is coming from the iisSettings section as shown below



We already set the **AspNetCoreHostingModel** element value to **InProcess**. Now, when we run the application either by pressing CTRL + F5 or just F5, then it will display the value as **iisexpress** for the worker process name as shown in the below image and also have a look at the browser URL. This proves that IIS Express is the only web server that is going to host the application as well as handles the incoming HTTP Requests.



**Case3: CommandName as IISExpress and AspNetCoreHostingModel as OutOfProcess**

If we use the CommandName as IISExpress profile and if we set the AspNetCoreHostingModel value as OutOfProcess then ASP.NET Core uses IIS Express as the external web server and Kestrel is the internal webserver. The external web server i.e. IIS Express will receive the incoming HTTP Requests and then forward the request to the internal web server i.e. Kestrel which is going to process the request. Let us prove this.

As we already set the launch profile as IIS Express, we just need to change the AspNetCoreHostingModel element value to OutOfProcess in the application’s project file as shown below.

**<Project** Sdk="Microsoft.NET.Sdk.Web"**>**

**<PropertyGroup>**

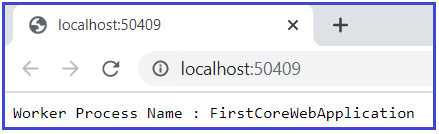
**<TargetFramework>**netcoreapp3.1**</TargetFramework>**

**<AspNetCoreHostingModel>**OutOfProcess**</AspNetCoreHostingModel>**

**</PropertyGroup>**

**</Project>**

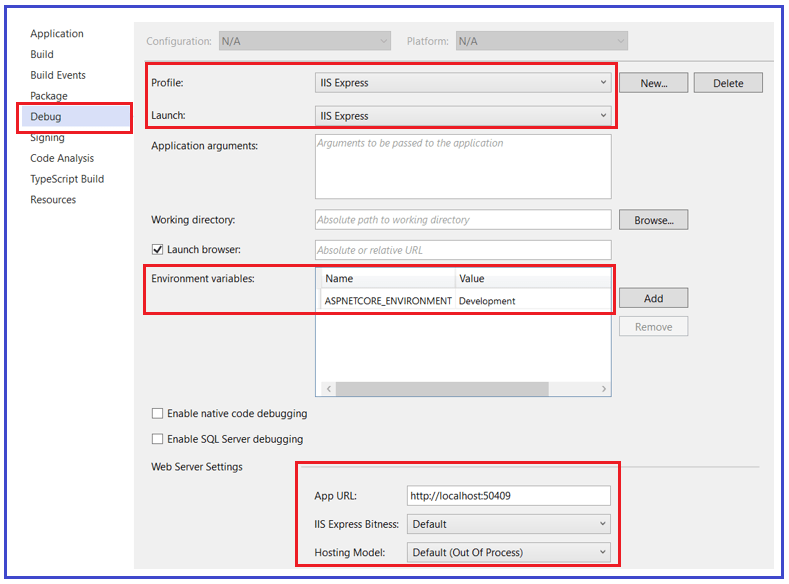
That’s it. Run the application and it should display the project name as the worker process name as shown in the below image as the request is internally handled by Kestrel Web Server.



The rest two cases we will discuss in a later article when we host the application using IIS. If you want then you can also change the settings of the launchSettings.json file using the Graphical User Interface (GUI) provided by Visual Studio.

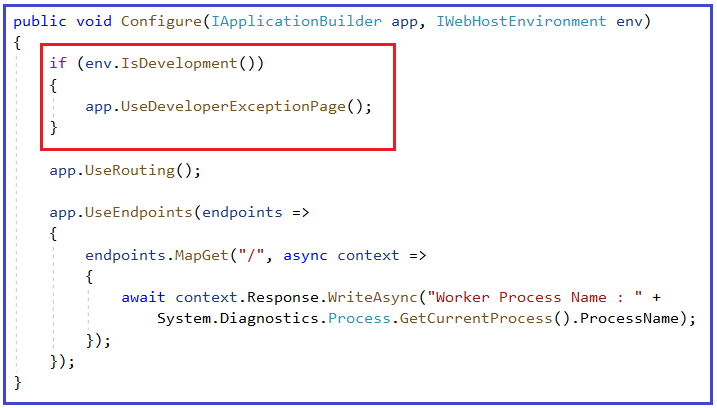
**How to access the Graphical User Interface (GUI) in Visual Studio?**

To do so, right-click on the project name in Solution Explorer and then select the “Properties” option from the context menu. Once you open the project properties window, click on the “Debug” tab on the as shown in the below image.



Using the Graphical User Interface, you can also change the settings of the **launchSettings.json file**. Now here you can see that the Environment Variable **“ASPNETCORE\_ENVIRONMENT”** is set to **“Development”**. You can change this Environment Variable value to **Staging**or **Production**depending on where you are running your application.

If you want, then you can also add new environment Variables. These environment variables are available throughout your application. And if you want then you can also execute some code conditionally depending on the environment variables value. For example, consider the following Configure() method of Startup.cs file.



It checks if the environment is Development, then it is going to display the Developer Exception Page. In our upcoming articles, we are going to discuss more these environment variables.

**ASP.NET Core Startup Class**

**ASP.NET Core Startup Class**

In this article, I am going to discuss the **ASP.NET Core Startup class** in detail. Please read our previous article where we discussed the [**ASP.NET Core launchSettings.json**](https://dotnettutorials.net/lesson/asp-net-core-launchsettings-json-file/) file.

**Startup Class in ASP.NET Core Application:**

The ASP.NET Core application must include a Startup class. It is like the Global.asax file our traditional .NET application. As the name suggests, it is executed first when the application starts. The startup class can be configured using **UseStartup<T>()** extension method at the time of configuring the host in the Main() method of Program class. Please have a look at the below Program class and please focus on the webBuilder.UseStartup<Startup>() method.

**namespace** *FirstCoreWebApplication*

**{**

**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**>()**;

**})**;

**}**

**}**

The name “Startup” is by the ASP.NET Core convention. However, you 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>()**.

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

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Startup

**{**

// This method gets called by the runtime. Use this method to add services to the container.

// For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=398940

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

**}**

// 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**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapGet**(**"/", **async** context =**>**

**{**

**await** context.Response.WriteAsync**(**"Hello World!"**)**;

**})**;

**})**;

**}**

**}**

**}**

As you can see in the code, the Startup class includes two public methods: ConfigureServices and Configure. The Startup class must include the Configure method and can optionally include the ConfigureService method.

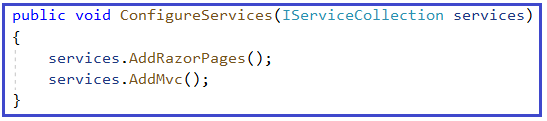
**ConfigureServices() method in ASP.NET Core Startup class**

The Dependency Injection pattern is used heavily in ASP.NET Core architecture. It includes the 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 the 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 to the dependent class as a Service. So, whenever you read “Service” then understand it as a class that is going to be used in some other class.

The ConfigureServices method includes the IServiceCollection parameter to register services to the IoC container. For example, if you want to add RazorPages services or MVC services to your asp.net core application, then you need to add those services to the parameter this method accepts as shown in the below image.

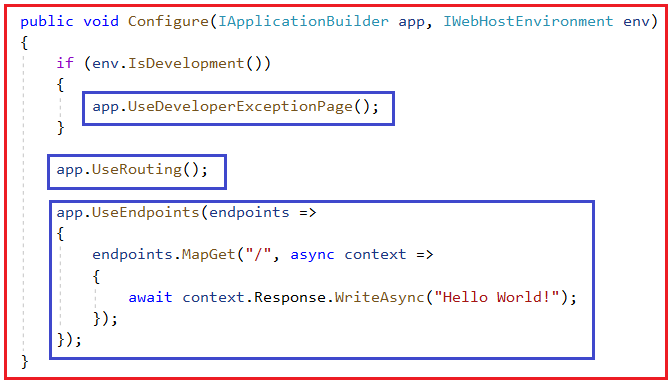


**Configure() method in ASP.NET Core Startup class**

The Configure method is a place where we can configure the application request pipeline for our asp.net core application using the 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 default configure method of ASP.NET Core application with the Empty template includes the following three middlewares as shown in the below image.



As we progress in this course, we will discuss more these two methods.

**ASP.NET Core appsettings.json file**

**ASP.NET Core appsettings.json file**

In this article, I am going to discuss the use and importance of the **ASP.NET Core appsettings.json file** in detail. Please read our previous article before proceeding to this article where we discussed the [**ASP.NET Core Startup class**](https://dotnettutorials.net/lesson/asp-net-core-satrtup-class/). As part of this article, we are going to discuss the following pointers in detail.

1. **What are the different Configuration Sources available in the ASP.NET Core application?**
2. **What is the ASP.NET Core appsettings.json file?**
3. **How to access the configuration information in ASP.NET Core Application?**
4. **What is the Configuration Execution Order in ASP.NET Core Application?**
5. **What is the Default Orders of reading the configuration sources?**
6. **How to Pass Config value from Command Line in ASP.NET Core Application?**

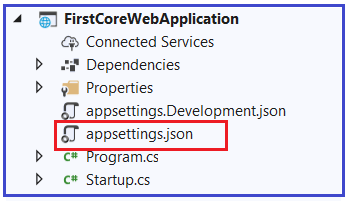
**What are the different Configuration Sources available in the ASP.NET Core application?**

If you have worked with the previous versions of the ASP.NET application, then you make know the importance of the web.config file. In previous versions of ASP.NET application, we generally used to store the application configuration settings such as database connection strings, any application scope global variables, and many more within the web.config file. But in ASP.NET Core, the application configuration settings can come from different configurations sources such as

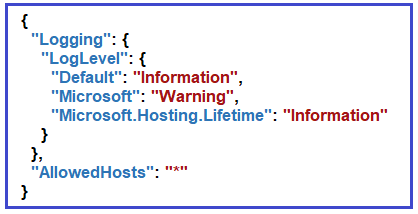
1. **Files** (appsettings.json, appsettings.{Environment}.json, where the {Environment} is the nothing but the applications current hosting environments such as Development, Staging or Production)
2. **User secrets**
3. **Environment variables**
4. **Command-line arguments**

**What is ASP.NET Core appsettings.json File?**

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 as shown in the below image.



The 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 ASP.NET Core appsettings.json file, then you see the following code by default which is created by visual studio.



Now I am going to add a key with the name **MyCustomKey** within this file. To do so, please modify the **appsettings.json** file as shown below. As it is a JSON file, you need to store the value in the form of key-value pair.

**{**

"Logging": **{**

"LogLevel": **{**

"Default": "Information"**,**

"Microsoft": "Warning"**,**

"Microsoft.Hosting.Lifetime": "Information"

**}**

**},**

"AllowedHosts": "\*"**,**

"MyCustomKey": "MyCustomKey Value coming from appsettings.json"

**}**

**How to access the configuration information in the ASP.NET Core application?**

To access the configuration information within the Startup class, you need to use the IConfiguration service which is provided by the ASP.NET Core Framework. So, what you need to do is just inject the IConfiguration service through the constructor of the Startup class. To do so modify the Startup class which is present in the Startup.cs file as shown below.

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.AspNetCore.Http;*

**using** *Microsoft.Extensions.Configuration;*

**using** *Microsoft.Extensions.DependencyInjection;*

**using** *Microsoft.Extensions.Hosting;*

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Startup

**{**

**private** IConfiguration \_config;

// Here we are using Dependency Injection to inject the Configuration object

**public** Startup**(**IConfiguration config**)**

**{**

\_config = config;

**}**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

**}**

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapGet**(**"/", **async** context =**>**

**{**

**await** context.Response.WriteAsync**(**\_config**[**"MyCustomKey"**])**;

**})**;

**})**;

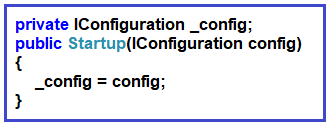
**}**

**}**

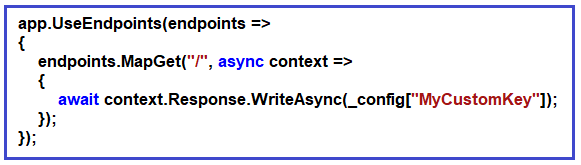
**}**

**Explanation of the above code:**

First, we created a private variable of type IConfiguration \_config (This IConfiguration interface belongs to Microsoft.Extensions.Configuration namespace, so bring this namespace first). Then through constructor dependency injection we inject the IConfiguration object and store it within the private variable \_config. The following code exactly does this.



Then we access the configuration variable i.e. **MyCustomKey** using the **IConfiguration** service instance. The following piece of code exactly does the same thing.



Set the AspNetCoreHostingModel value to InProcess in the application’s project file as shown below.

**<Project** Sdk="Microsoft.NET.Sdk.Web"**>**

**<PropertyGroup>**

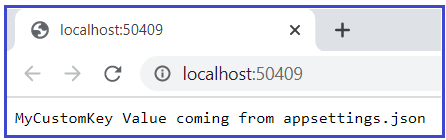
**<TargetFramework>**netcoreapp3.1**</TargetFramework>**

**<AspNetCoreHostingModel>**InProcess**</AspNetCoreHostingModel>**

**</PropertyGroup>**

**</Project>**

Now run the application and you should see the value as expected in the browser as shown in the below image.



**Dependency Injection Design Pattern**

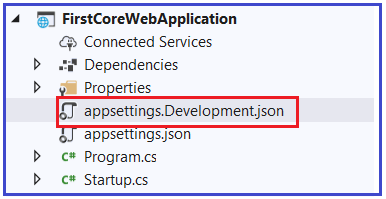
In our previous versions of ASP.NET applications, the [**Dependency Injection design pattern**](https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/) was optional. But if you want to configure it in a traditional .NET application, then you need to use some of the frameworks like Ninject, StructureMap, IUnity container, etc.

But in ASP.NET Core application Dependency Injection is an integral part and the framework provides the inbuilt support for dependency injection. The [**Dependency Injection Design Pattern**](https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/) allows us to develop loosely coupled systems that are extensible and also easy to testable. If this is not clear at the moment don’t worry, we will discuss the Dependency Injection design pattern in great detail in our upcoming articles with asp.net core application. If you want to know how the dependency injection design pattern is used with the previous versions of ASP.NET application, then read the following article.

[**https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/**](https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/)

**What is the Configuration Execution Order in ASP.NET Core Application?**

Before understanding the execution order, let’s have a look at the appsettings.Development.json file. You can find this file within the project as shown in the below image.



**Let us modify the appsettings.Development.json as shown below.**

**{**

"Logging": **{**

"LogLevel": **{**

"Default": "Information"**,**

"Microsoft": "Warning"**,**

"Microsoft.Hosting.Lifetime": "Information"

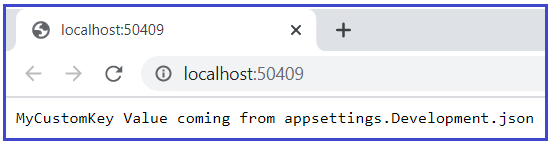
**}**

**},**

"MyCustomKey": "MyCustomKey Value coming from appsettings.Development.json"

**}**

As you can see, here we are using the same key as we use in the appsettings.json file. Now run the application and see the output as shown below.



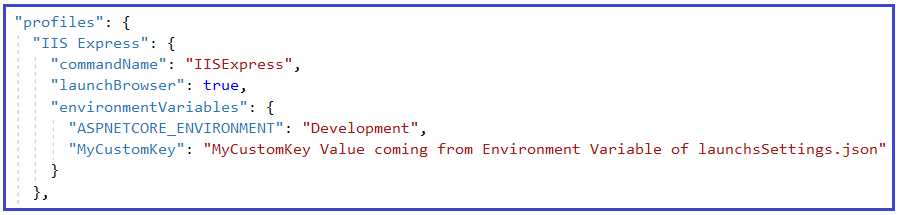
As you see from the above output, it fetches the MyCustomValue from the appsettings.Development.json file. The point that I need to make you clear, if you have a configuration setting in multiple configuration sources with thesame configuration key, then the later configuration sources will override the earlier configuration sources.

**What are the Default Orders of reading the configuration sources?**

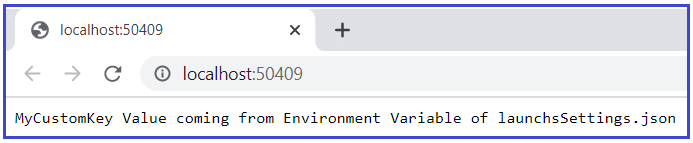
The default orders in which the various configuration sources are read for the same key are as follows

1. **appsettings.json,**
2. **appsettings.{Environment}.json here we use appsettings.development.json**
3. **User secrets**
4. **Environment variables**
5. **Command-line arguments**

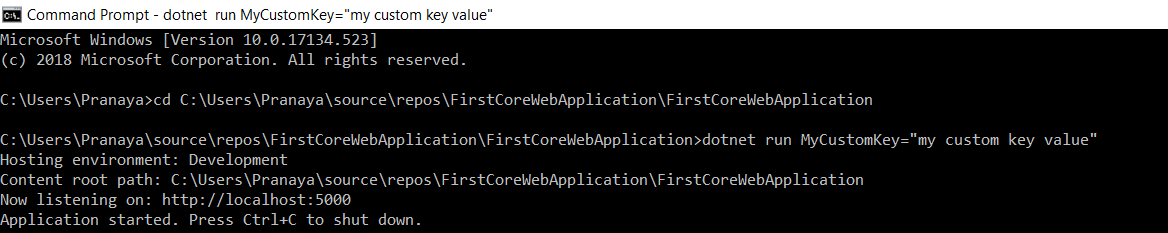
We already have MyCustomKey in two places i.e. appsettings.json and appsettings.development.json. Now add the same key as “MyCustomKey”: “MyCustomKey Value coming from Environment Variable of launchsSettings.json” in the IIS Express profile section of the launchSettings.json file as shown below.



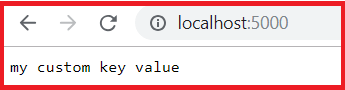
With this change now run the application and it should display the value coming from the environment variable.



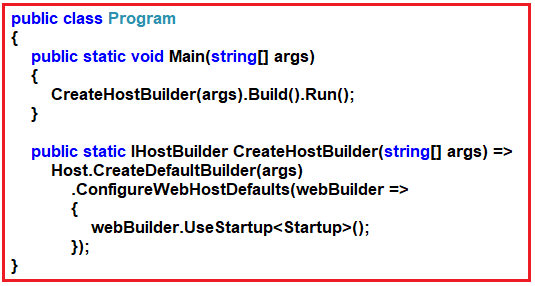
**How to Pass Config value from Command Line in ASP.NET Core Application?**



Now open the browser window and type the following URL



**The following is the auto-generated program class.**



As you can see the Main() method of the Program class calls the CreateHostBuilder() method. Then the CreateHostBuilder() method calls the CreateDefaultBuilder() method on the Host class. This CreateDefaultBuilder() method is the method that sets the default order in which all the configuration sources are read.

If you want then you can also change this default order or even if you want then you can add your own custom configuration sources along with the existing configuration sources. In our upcoming articles, we will discuss setting up a custom configuration source.

**ASP.NET Core Middleware**

**ASP.NET Core Middleware with Examples**

In this article, I am going to discuss the **ASP.NET Core Middleware with Examples**. Please read our previous article before proceeding to this article where we discussed the [**ASP.NET Core appsettings.json**](https://dotnettutorials.net/lesson/asp-net-core-appsettings-json-file/) file. As part of this article, we are going to discuss the following concepts related to the ASP.NET Core Middleware Components**.**

1. **What are the ASP.NET Core Middleware Components?**
2. **Where we use the Middleware Components in the ASP.NET Core application?**
3. **How to Configure Middleware Components in ASP.NET Core application?**
4. **Examples of using Middleware Components?**
5. **What is the Execution Order of Middleware Components in ASP.NET Core?**
6. **What are Request Delegates in ASP.NET Core?**
7. **What is Use, Run, and Map method in ASP.NET Core?**
8. **What is UseDeveloperExceptionPage Middleware Component?**
9. **How to Configure Middleware Components using the Run() and Use() extension methods?**
10. **What is the difference between MapGet and Map method?**

**What are the ASP.NET Core Middleware Components?**

The ASP.NET Core Middleware Components are the software components (technically components are nothing but the C# Classes) that are assembled into the application pipeline to handle the HTTP Requests and Responses. Each middleware component in ASP.NET Core Application performs the following tasks.

1. Chooses whether to pass the HTTP Request to the next component in the pipeline. This can be achieved by calling the next() method within the middleware.
2. Can perform work before and after the next component in the pipeline.

In ASP.NET Core there are so many built-in Middleware components that are already made available that you can use directly. If you want then you can also create your own Middleware components in asp.net core applications. The most important point that you need to keep in mind is, in ASP.NET Core a given Middleware component should only have a specific purpose i.e. single responsibility.

**Where we use Middleware Components in the ASP.NET Core application?**

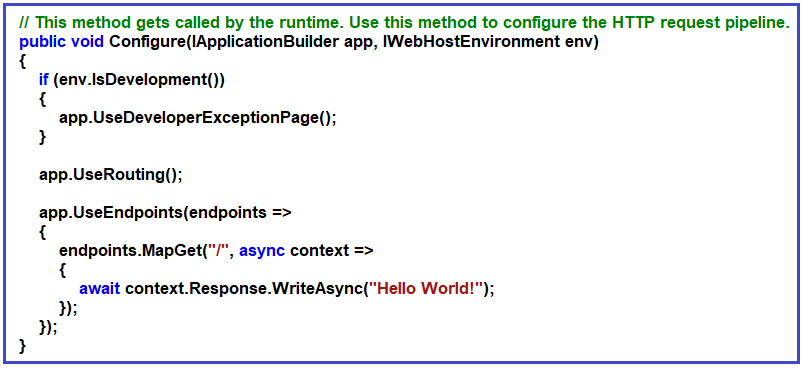
Some of the examples of using Middleware components in the ASP.NET Core application are as follows

1. We may have a Middleware component for authenticating the user
2. Another Middleware component may be used to log the request and response
3. Similarly, we may have a Middleware component that is used to handle the errors
4. We may have a Middleware component that is used to handle the static files such as images, Javascript or CSS files, etc.
5. Another Middleware component may be used to Authorize the users while accessing a specific resource

The Middleware components are the components that we generally use to set up the request processing pipeline in the ASP.NET Core application. If you have worked with previous versions of.NET Framework then you may know, we use HTTP Handlers and HTTP Modules to set up the request processing pipeline. It is this pipeline that will determine how the HTTP request and response are going to be processed.

**How to Configure Middleware Components in ASP.NET Core application?**

In the ASP.NET Core application, we need to configure the Middleware components within the Configure() method of the Startup class which is present inside the Startup.cs file. This is the class that is going to run when the application starts. When we create an ASP.NET Core application with Empty Template, then by default the Startup class is created with the Configure() method as shown in the below image.



So, whenever you want to configure any middleware components in any type of .net core applications, then you need to configure it within the Configure() method of the Startup class by calling the **Use\*** methods on the IApplicationBuilderobject. As you can see in the above image, the Configure() method sets up the request processing pipeline with just three middleware components are as follows.

1. **UseDeveloperExceptionPage() Middleware component**
2. **UseRouting() Middleware component**
3. **UseEndpoints() Middleware component**

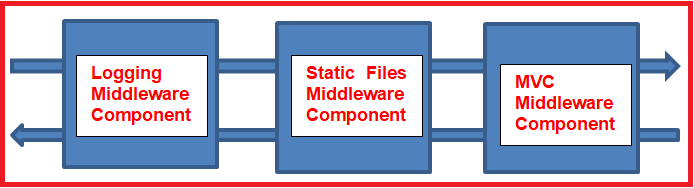
Before understanding the above three built-in Middleware components. Let us first understand what are Middleware components and how exactly these Middleware components work in an ASP.NET Core application.

**Understanding Middleware Components in ASP.NET Core:**

In the ASP.NET Core application, the Middleware component can have access to both the incoming HTTP Request and outgoing HTTP Response. So, a Middleware component in ASP.NET Core can

1. Handle the incoming HTTP request by generating an HTTP response.
2. Process the incoming HTTP request, modify it, and then pass it to the next middleware component
3. Process the outgoing HTTP response, modify it, and then pass it on to either the next middleware component or to the ASP.NET Core web server.

For better understanding, please have a look at the following diagram which shows how the middleware components used in the request processing pipeline of an ASP.NET Core application.



As shown in the above image, we have a logging middleware component. This component simply logs the request time and then passes the request to the next middleware component i.e. Static Files Middleware component in the request pipeline for further processing.

A middleware component in ASP.NET Core may also handle the HTTP Request by generating an HTTP Response. The ASP.NET Core Middleware component may also decide not to call the next middleware component in the request pipeline. This concept is called short-circuiting the request pipeline.

For example, we have a static file middleware component. And if the incoming HTTP request comes for some static files such as images, CSS files, JavaScript, etc. then this Static Files Middleware component can handle the request and then short-circuit the request pipeline by not calling to the next component in the pipeline i.e. the MVC Middleware component.

As we already discussed the ASP.NET Core middleware components can have access to both the HTTP request and response in the pipeline. So, a middleware component can also process the outgoing response. For example, the logging middleware component in our case may log the time when the response is sent back to the client.

**What is the Execution Order of Middleware Components in ASP.NET Core Application?**

It is very important to understand the execution order of Middleware components. The ASP.NET Core middleware components are executed in the same order as they are added to the pipeline. So, we need to take care when adding the middleware components to the request processing pipeline.

As per your application’s business requirements, you may add any number of Middleware components. For example, if you are developing a static web application with some static HTML pages and images, then you may require only “StaticFiles” middleware components in the request processing pipeline.

But, if you are developing a secure dynamic data-driven web application then you may require several middleware components such as Logging Middleware, Authentication middleware, Authorization middleware, MVC middleware, etc.

**What are Request Delegates in ASP.NET Core?**

In ASP.NET Core, Request delegates are used to build the request pipeline i.e. request delegates are used to handle each incoming HTTP request. In ASP.NET Core, you can configure the Request delegates using the **Run, Map, and Use** extension methods. You can specify a request delegate using an in-line anonymous method (called in-line middleware) or you can specify the request delegates using a reusable class. These reusable classes and in-line anonymous methods are called middleware or middleware components. Each middleware component in the request processing pipeline is responsible for invoking the next component in the pipeline or short-circuiting the pipeline by not calling the next middleware component.

**What is the use of the Use and Run method in ASP.NET Core Web Application?**

In ASP.NET Core, you can use the “**Use**” and “**Run**” extension methods to register the Inline Middleware component into the Request processing pipeline. The “Run” extension method allows us to add the terminating middleware (the middleware which will not call the next middleware components in the request processing pipeline). On the other hand, the “Use” extension method allows us to add the middleware components which may call the next middleware component in the request processing pipeline.

If you observe the Configure method, then you will see that it gets an instance of the IApplicationBuilder interface and using that instance along with the extension methods such as Use and Run, it configures the Middleware components.

As you can see, in the configure method, three middleware components are registered in the request processing pipeline using the IApplicationBuilder instance i.e. app. They are as follows

1. **UseDeveloperExceptionPage() Middleware component**
2. **UseRouting()**
3. **UseEndpoints() Middleware component**

**DeveloperExceptionPage Middleware Component:**

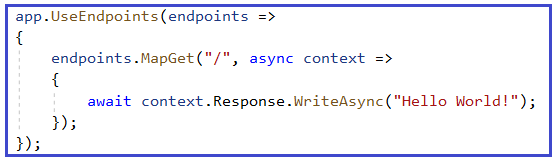
As you can see, within the configure method, the **UseDeveloperExceptionPage()** middleware component is registered into the pipeline and this middleware component will come into the picture only when the hosting environment is set to “development”. This middleware component is going to execute when there is an unhandled exception occurred in the application and since it is in development mode, it is going to show you the culprit line of code. You can consider this as a replacement for the yellow screen of death. In a later article, we will see the use of this middleware component in detail with examples.

**UseRouting() Middleware Component:**

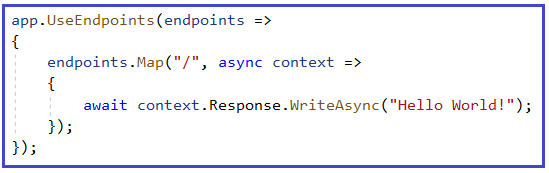
This middleware component is used to add Endpoint Routing Middleware to the request processing pipeline i.e. it will map the URL (or incoming HTTP Request) to a particular resource. We will discuss this in detail in our Routing articles.

**UseEndpoints() Middleware Component:**

In this middleware, the routing decisions are going to be taken using the Map extension method. Following is the default implementation of UseEndpoints middleware components. In the MapGet extension method, we have specified the URL pattern like “/”. This means the domain name only. So, any request with only the domain name is going to be handle by this middleware.



Instead of MapGet, you can also use the Map method as shown below.



Now run the application and you should get the output as expected.

**What is the difference between MapGet and Map method?**

The MapGet method is going to handle the GET HTTP Requests whereas the Map method is going to handle all types of HTTP requests such as GET, POST, PUT, & DELETE, etc.

**How to Configure Middleware Components using the Run() extension method?**

Let us understand how to create and configure custom middleware components using the Run extension method. First of all, comment on all codes which are present inside the Configure method. Once you comment on the existing code, then copy and paste the following code into the Configure method. The following code simply adds a new middleware component to the application’s request pipeline and simply prints a message.

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Getting Response from First Middleware"**)**;

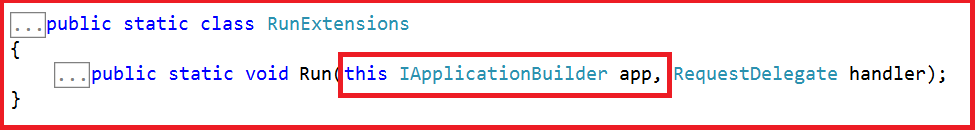
**})**;

**}**

**Output:**



We are invoking the **Run() extension** method on the **IApplicationBuilder** instance (app) to register the middleware component into the request processing pipeline. Following is the definition of the **Run** method.



As you can see from the definition of the Run() method, it is implemented as an extension method of the IApplicationBuilder interface. This is the reason why we are able to call the Run() method using the IApplicationBuilder instance i.e. app. If you are new to the extension method then please read the below article where we discussed the extension methods in detail.

[**https://dotnettutorials.net/lesson/extension-methods-csharp/**](https://dotnettutorials.net/lesson/extension-methods-csharp/)

You can also see from the above image that the Run() method takes an input parameter of type **RequestDelegate**. Following is the definition of RequestDelegate.

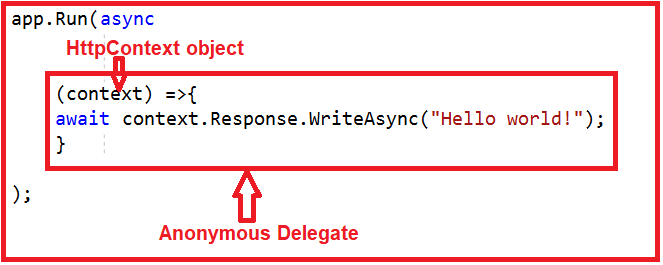


As you can see from the above image, the RequestDelegate is a delegate that takes an input parameter of type HttpContext object. If you are new to delegates then I strongly recommended you read the following article where we discussed the delegates in detail.

[**https://dotnettutorials.net/lesson/delegates-csharp/**](https://dotnettutorials.net/lesson/delegates-csharp/)

As we already discussed the middleware components in the ASP.NET Core application can have access to both HTTP Request and Response and this is because of the above HttpContext object.

In our example, we are passing the request delegate inline as an anonymous method using lambda expression and moreover, we are passing the HTTPContext object as an input parameter to the request delegate. The following diagram shows the above



**Note:** Instead of passing the request delegate inline as an anonymous method, you can also define the request delegate in a separate class and pass it here which we will discuss in a later article.

**Add one more middleware to the application.**

To do so, modify the configure method of the Startup class as shown below.

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Getting Response from First Middleware"**)**;

**})**;

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Getting Response from Second Middleware"**)**;

**})**;

**}**

Now we have two middleware components registered with the Run() extension method. If you run the application then you will get the following output.

**Getting Response from 1st Middleware**

The output is coming from the first middleware component. The reason is, when we registered a middleware component using the Run() extension method then that component becomes a terminal component means it will not call the next middleware component in the request processing pipeline.

**Configuring middleware component using the Use extension method**

Then the question that comes to your mind is how to call the next component in the request processing pipeline, the answer is to register your middleware component using the Use extension method as shown below.

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Getting Response from 1st Middleware"**)**;

**await** next**()**;

**})**;

app.Run**(async** **(**context**)** =**>**

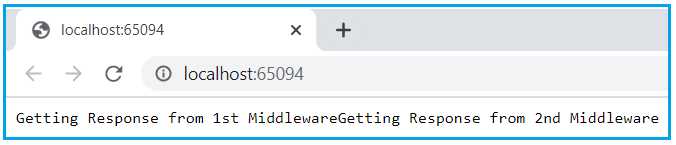
**{**

**await** context.Response.WriteAsync**(**"Getting Response from 2nd Middleware"**)**;

**})**;

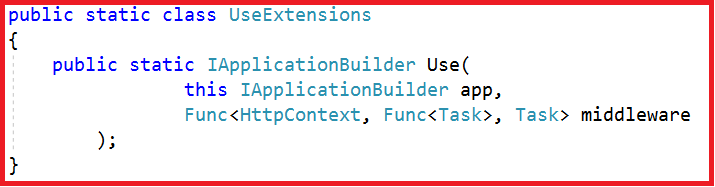
**}**

Now run the application and you will see the output as expected which is coming from both the middleware components.



**Understanding the Use extension method:**

The Use extension method adds a middleware delegate defined in-line to the application’s request pipeline. Following is the definition of the Use extension method:



This method is also implemented as an extension method on the IApplicationBuilder interface. This is the reason why we are able to invoke this method using the IApplicationBuilder instance. As you can see from the above definition, this method takes two input parameters. The first parameter is the HttpContext context object through which it can access both the HTTP request and response. The second parameter is the Func type i.e. it is a generic delegate that can handle the request or call the next middleware component in the request pipeline.

**Note:** If you want to send the request from one middleware to the next middleware then you need to call the next method.

**ASP.NET Core Request Processing Pipeline**

**ASP.NET Core Request Processing Pipeline**

In this article, I am going to discuss the **ASP.NET Core Request Processing Pipeline** with an example. Please read our previous article before proceeding to this article where we discussed [**Middleware Components in ASP.NET Core**](https://dotnettutorials.net/lesson/asp-net-core-middleware-components/) application. As part of this article, we are going to discuss the following pointers in detail.

1. **Understanding the ASP.NET Core Request Processing Pipeline.**
2. **How to create and register multiple middleware components in ASP.NET Core?**
3. **What is the execution order of middleware components in the request processing pipeline?**

**Understanding the ASP.NET Core Request Processing Pipeline:**

In order to understand the Request Processing Pipeline in ASP.NET Core, concept, let us modify the Configure() method of the Startup class as shown below. Here we are registering three middleware components into the request processing pipeline. As you can see the first two components are registered using the **Use()** extension method so that they have the chance to call the next middleware component in the request processing pipeline. The last one is registered using the **Run()** extension method as it is going to be our terminating components i.e. it will not call the next component.

**using** *Microsoft.AspNetCore.Builder;*

**using** *Microsoft.AspNetCore.Hosting;*

**using** *Microsoft.AspNetCore.Http;*

**using** *Microsoft.Extensions.DependencyInjection;*

**using** *Microsoft.Extensions.Hosting;*

**namespace** *FirstCoreWebApplication*

**{**

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

**}**

**public** **void** Configure**(**IApplicationBuilder app**)**

**{**

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Middleware1: Incoming Request\n"**)**;

**await** next**()**;

**await** context.Response.WriteAsync**(**"Middleware1: Outgoing Response\n"**)**;

**})**;

app.Use**(async** **(**context, next**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Middleware2: Incoming Request\n"**)**;

**await** next**()**;

**await** context.Response.WriteAsync**(**"Middleware2: Outgoing Response\n"**)**;

**})**;

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Middleware3: Incoming Request handled and response generated\n"**)**;

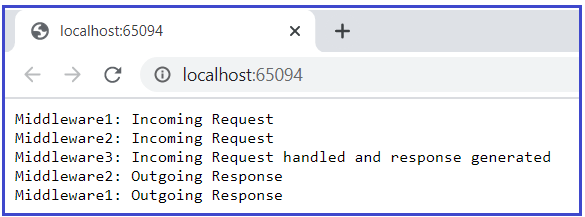
**})**;

**}**

**}**

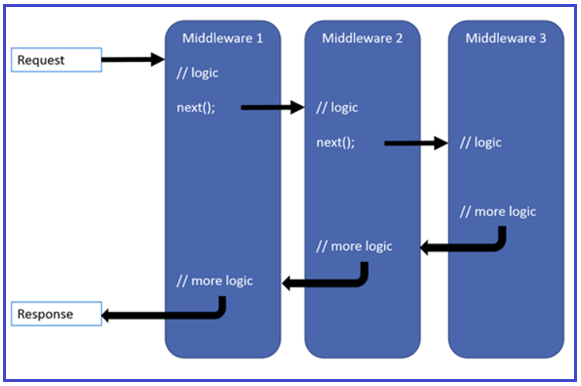
**}**

**Now run the application and see the browser window as shown in the below image.**



**Understanding the ASP.NET Core Request Processing Pipeline Execution Order:**

In order to understand this, let us compare the above output with the following diagram to understand the ASP.NET Core Request Processing Pipeline in an easier way.



When the incoming HTTP request comes, first it receives by the first middleware component i.e. Middleware1 which logs “**Middleware1: Incoming Request**” in the response stream. So as a result, first, we see this message first on the browser.

Once the first middleware logs the information, then it calls the next() method which will invoke the second middleware in the request processing pipeline i.e. Middleware2.

The second middleware logs the information “**Middleware2: Incoming Request**” as a result we see this log information after the first log. Then the second middleware calls the next() which will invoke the third middleware in the request pipeline which is Middleware3.

The third middleware handles the request and then produces the response. So, the third information that we see in the browser is “**Middleware3: Incoming Request handled and response generated**”.

This middleware component is registered using the Run() extension method, so it is a terminal component. So, from this point, the request pipeline starts reversing. That means from this middleware, the control is given back to the second middleware, and the second middleware logs the information as “**Middleware2: Outgoing Response**” and then give the control back to the first middleware component, and the first middleware component logs the information as “**Middleware1: Outgoing Response**” as what we see in the browser.

**Key Points to remember:**

1. The ASP.NET Core request processing pipeline consists of a sequence of middleware components that are going to be called one after the other.
2. Each middleware component can perform some operations before and after invoking the next component using the next method. A middleware component can also decide not to call the next middleware component which is called short-circuiting the request pipeline.
3. The middleware component in asp.net core has access to both the incoming request and the outgoing response.
4. The most important point that you need to keep in mind is the order in which the middleware components are added in the Configure method of the Startup class defines the order in which these middleware components are going to be invoked on requests and the reverse order for the response. So, the order is critical for defining the security, performance, and functionality of the application.

**wwwroot folder in ASP.NET Core**

**wwwroot folder in ASP.NET Core**

In this article, I am going to discuss **wwwroot folder in ASP.NET Core** Application. Please read our previous article where we discussed [**ASP.NET Core Request Processing Pipeline**](https://dotnettutorials.net/lesson/asp-net-core-request-processing-pipeline/). At the end of this article, you will understand what wwwroot folder and its need and how to configure this in ASP.NET Core Application.

**What is wwwroot folder in ASP.NET Core?**

By default, the wwwroot folder in the ASP.NET Core application is treated as the webroot folder and this folder or directory should be present in the root project folder. In ASP.NET Core Application, the Static files can be stored in any folder under the webroot folder and can be accessed with a relative path to that root.

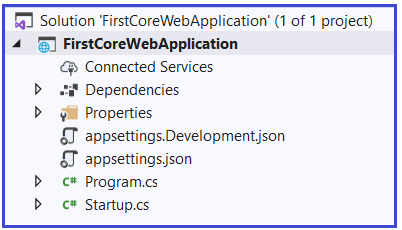
**Adding the wwwroot (webroot) folder in ASP.NET Core Application:**

When you create an ASP.NET Core Web Application with Web and MVC Template, then by default this folder (wwwroot) is created in the root project folder. But if you create a new .NET Core Application with Empty template then by default this folder is not going to be created by Visual Studio.

As we are discussing everything from scratch, let us create an ASP.NET Core Application using the Empty Project template and then understand how to add the wwwroot folder in it.

**Creating an ASP.NET Core Application:**

First, create a new ASP.NET Core Application with the name “FirstCoreWebApplication” with Empty project template. The project should be created with the following structure.



As you can in the above image, there is no such folder called wwwroot in our application.

**Adding wwwroot (webroot) folder in ASP.NET Core:**

In order to add the wwwroot folder, right-click on the project and then select **add => new folder** option and then provide the folder name as **wwwroot**. Once you created the folder then please have a look at the folder symbol as shown below.



**What the wwwroot (webroot) folder is going to contain in ASP.NET Core:**

In the earlier ASP.NET application, the static files can be served either from the project root folder or any other folder under it. But this has been changed in ASP.NET Core. Now, only those files that are present inside in the webroot – wwwroot folder or any subfolder under it can be served over an HTTP request. All other files are blocked and cannot be served by default. But if you want then you can also change this default behavior.

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



Now, you can access static files such as CSS, js, lib with base URL and file name. For example, you can access the above site.js file in the js folder by **https://localhost:<port>/js/site.js**

**Note:** In order to serve the static files, you need to include the [**app.UseStaticFiles()**](https://dotnettutorials.net/lesson/static-files-middleware-asp-net-core/)middleware component in the ‘Configure()’ method of Startup.cs file. If this is not clear at the moment, then don’t worry, we will discuss everything about [**static files middleware**](https://dotnettutorials.net/lesson/static-files-middleware-asp-net-core/) in our next article with examples.

**Can we rename the wwwroot Folder?**

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

For example, let’s rename the wwwroot folder to the “MyWebRoot” folder. Once you rename the wwwroot folder to MyWebRoot, then you need to call the UseWebRoot() method to configure MyWebRoot folder as a webroot 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**(**"MyWebRoot"**)**;

**})**;

**}**

With the above changes in place, now the MyWebRoot folder is going to act as the Webroot folder for your application and can serve the static files HTTP requests.

**Static Files Middleware in ASP.NET Core**

**Static Files Middleware in ASP.NET Core Application**

In this article, I am going to discuss how to serve static files using **Static Files Middleware in ASP.NET Core** Application. Please read our previous article before proceeding to this article where we discussed the[**wwwroot folder in ASP.NET Core**](https://dotnettutorials.net/lesson/wwwroot-folder-asp-net-core/) Application. As part of this article, we are going to discuss the following pointers in details.

1. **Where do we need to store the static files in ASP.NET Core?**
2. **What is wwwroot folder in ASP.NET Core?**
3. **How to Configure Static Files Middleware in ASP.NET Core Web Application?**
4. **How to use your own Webroot folder?**

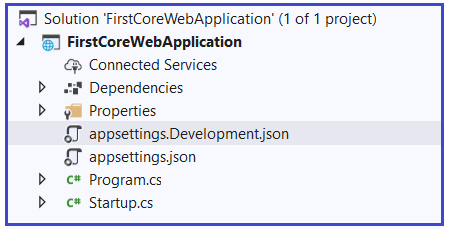
One of the most important features almost all web applications should have the ability to serve the static files directly from the file system. The static files such as HTML, Images, CSS, and JavaScript are the important assets of an application and ASP.NET Core can serve these files directly to the clients. But the important point that you need to keep in mind by default the ASP.NET Core cannot serve these static files. Some configuration is required in order to enable the ASP.NET Core to serve these static files directly.

**Where do we need to store the static files in ASP.NET Core?**

In ASP.NET Core Application, the default directory or location for the static files is **wwwroot** (webroot) folder and moreover, this folder or directory should be present in the project root folder. By default, this is the only place where the ASP.NET Core application can serve the static files directly. But we can change this default behavior by using the **UseWebRoot** method.

**Let us understand everything steps by step:**

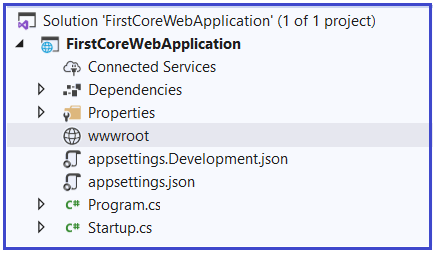
First, create a new ASP.NET Core web application with the Empty template. With the Empty project template by default, you will not found the wwwroot folder. The project structure of ASP.NET Core Web Application with the Empty Project template is shown below.



**Note:** If you create the project using a Web or MVC template then by default the wwwroot folder will be created by visual studio.

**Adding the wwwroot (webroot) folder:**

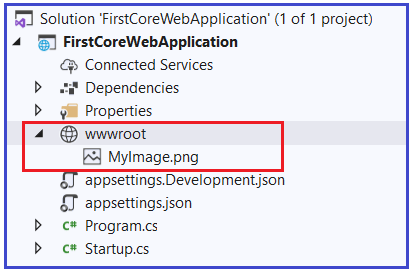
Let us first create the wwwroot project folder. To do so, right-click on the project and then select **add => new folder** option from the context menu and then provide the folder name as wwwroot. Once you created the wwwroot folder, your project structure should be as shown below.



Once you created the **wwwroot** folder, let’s add an Image file within that folder. Please download and paste the following image into the wwwroot folder and modify the image name as MyImage.png.



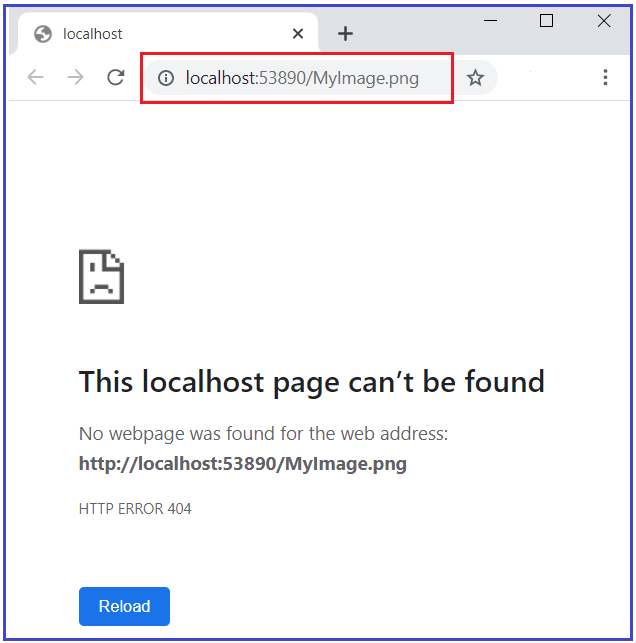
Once you the above image, your **wwwroot** directory looks as shown below.



Now run the application and navigate to the following URL. The point that you need to remember is you need to replace the port number on which your application is running.

**http://localhost:<portnumber>/MyImage.png**

When you navigate to the above URL, you will not get the output as expected, rather than you will get the following output.



The reason why we are not getting the output as expected because we don’t have any middleware which can serve the static files in the request processing pipeline.

**How to Configuring the Static Files Middleware in ASP.NET Core Application?**

In order to handle the static resources, we need to [configure a middleware](https://dotnettutorials.net/lesson/asp-net-core-middleware-components/) called UseStaticFiles() into the application [request processing pipeline of an ASP.NET Core](https://dotnettutorials.net/lesson/asp-net-core-request-processing-pipeline/) Application. The UseStaticFiles() middleware is an inbuilt middleware provided by ASP.NET Core Framework to handle the static files in an ASP.NET Core Application.

Let us Modify the Configure() method of the Startup class as shown below in order to add the UseStaticFiles() [middleware](https://dotnettutorials.net/lesson/asp-net-core-middleware-components/) to the request processing pipeline of the application.

**public** **class** Startup

**{**

**public** **void** ConfigureServices**(**IServiceCollection services**)**

**{**

**}**

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

//Adding Static Files Middleware to serve the static files

app.UseStaticFiles**()**;

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapGet**(**"/", **async** context =**>**

**{**

**await** context.Response.WriteAsync**(**"Hello World!"**)**;

**})**;

**})**;

**}**

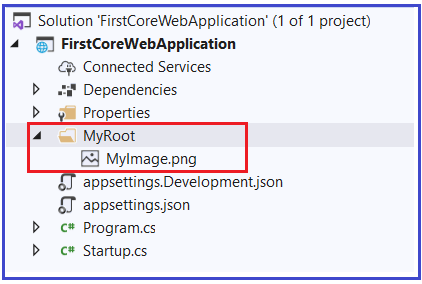
**}**

With the above changes in place, now run the application and navigate to the URL: **http://localhost:<portnumber>/MyImage.png** and you will see the output as expected as shown in the below image.



**How to Create your own Webroot folder in ASP.NET Core?**

Let say we don’t want wwwroot as our webroot folder instead we want MyRoot as the webroot folder for our application. First. Modify the wwwroot folder as MyRoot and once you modify your project structure should be as shown below.



At this point, if you run the application and if you navigate to the below URL, then you will not get the output.

**http://localhost:<portnumber>/MyImage.png**

This is because by default the static files middleware will look for a folder with the name wwwroot and that is not present in our application at the moment. But we don’t want wwwroot, we want the Static files middleware to look MyRoot folder to the server the static files such as CSS, Images, JS, etc. To do so, we need to call the UseWebRoot method by passing MyRoot as a parameter in the CreateHostBuilder method which available in the 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**(**"MyRoot"**)**;

**})**;

**}**

With the above changes in place, now run the application and you should get the output as expected.

**Configuring Default Page in ASP.NET Core**

**Configuring Default Page in ASP.NET Core**

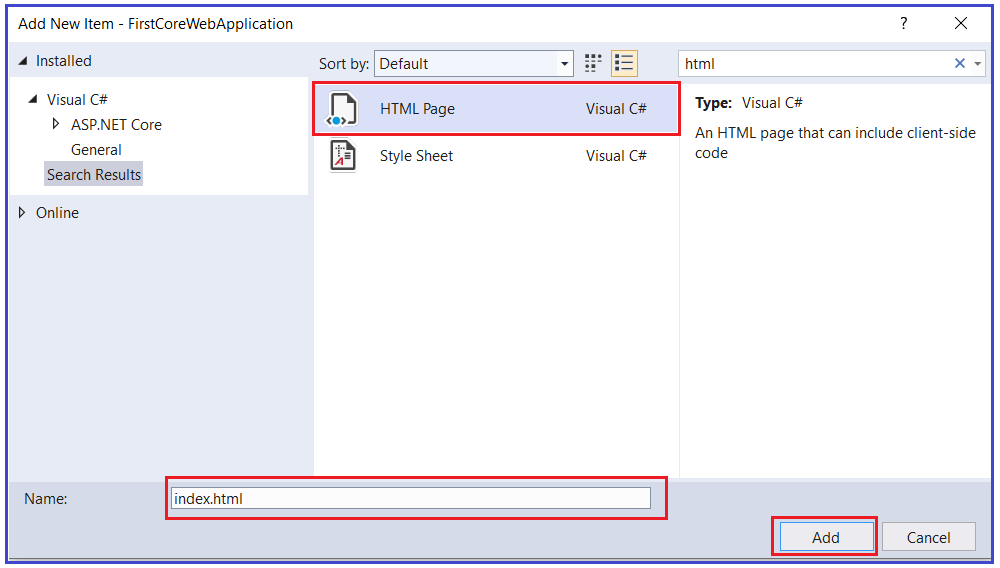
In this article, I am going to discuss **Configuring Default Page in ASP.NET Core** Application. Please read our previous article where we discussed [**How to Configure Static Files Middleware in ASP.NET Core**](https://dotnettutorials.net/lesson/static-files-middleware-asp-net-core/) application to serve the static files such as Image, CSS, JavaScript, etc.

**How to Configure Default Page in ASP.NET Core:**

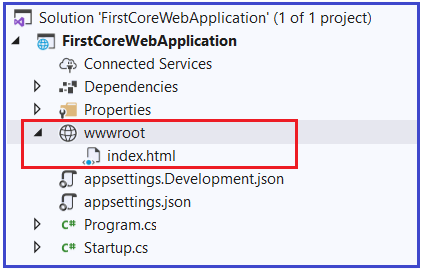
As we are going to discuss everything from scratch, let us create a new ASP.NET Core Application with an Empty template. With the empty template by default, the [**webroot folder i.e. wwwroot folder**](https://dotnettutorials.net/lesson/wwwroot-folder-asp-net-core/) will not be available. So, let us add the wwwroot folder to the project root directory. To do so, right-click on the project and then select add => new folder option from the context menu and then provide the folder name as wwwroot**.**

**Adding one HTML Page:**

Let us add one HTMP page with the name **index**within the wwwroot folder. To do so, right-click on the **wwwroot** folder and then select **add => new item** which will open add new item window. From the new item window, select the **HTML template**, provide the name as “**index.html**” and then click on the **Add** button as shown in the below image.



Once you add the HTML Page within the wwwroot folder, your project folder structure should be shown like below.



Now, open the **index.html**file which is present inside the wwwroot folder and then copy and paste the following code in it.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** charset="utf-8" **/>**

**<title></title>**

**</head>**

**<body>**

**<h1>** This is Index HTML File**</h1>**

**</body>**

**</html>**

**Modifying the Configure Method of the Startup class:**

In order to server the static files in asp.net core applications, we need to use the UseStaticFiles [**middleware in the request processing pipeline**](https://dotnettutorials.net/lesson/asp-net-core-middleware-components/). Let us modify the Configure method as shown below to use the UseStaticFiles middleware.

**public** **class** Startup

**{**

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

//Adding Static Files Middleware to serve the static files

app.UseStaticFiles**()**;

app.Run**(async** **(**context**)** =**>**

**{**

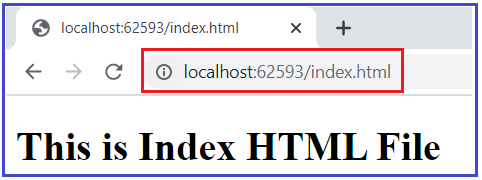
**await** context.Response.WriteAsync**(**"Request handled and response generated"**)**;

**})**;

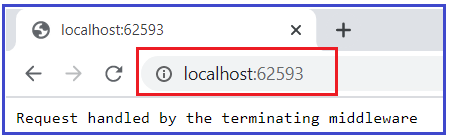
**}**

**}**

Now run the application and navigate to the URL: **http://localhost:<portnumber>/index.html** and you will see the output as expected that is coming from the static Index.HTML file as shown below.



Now, if you remove index.html from the URL or navigate to the base URL, then the request is going to be handled by the terminating middleware which is registered using the Run method. That the default request is going to be server by the Run method as shown below.



But what we want is, when we navigate to the base URL as shown above, we want our index.html page to serve the request. That is, we need to set the index.html page as our default page.

**Setting the Default Page in ASP.NET Core Application:**

Most of the web applications have a default page such as index.htm(l) or default.htm(l) as their startup page as it is easy to remember. This is the web page that is going to be displayed when a user visits the root URL of that application. For example, if you have a page with the name index.html and you want that page to be your default page so that whenever any user visits your root URL, then that page is going to be displayed.

In order to serve the index.html page which is present inside the wwwroot folder as the default page of your application, you need to add another middleware i.e. **UseDefaultFiles()** middleware into the request processing pipeline.

So, modify the **Configure()** method of the **Startup** class as shown below to use the **UseDefaultFiles()** middleware which will set the default page for your application.

**public** **class** Startup

**{**

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

//Setting the Default Files

app.UseDefaultFiles**()**;

//Adding Static Files Middleware to serve the static files

app.UseStaticFiles**()**;

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Request handled by the terminating middleware"**)**;

**})**;

**}**

**}**

With the above changes in place, now run the application and you should the output as expected as shown below. That index.html page serves as your default page.



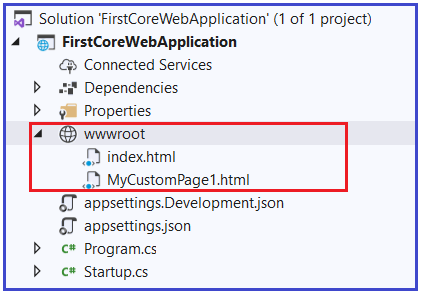
**Note:** You need to add the **UseDefaultFiles()** middleware before the **UseStaticFiles()** middleware in order to serve the default file. The point that you need to remember is the UseDefaultFiles() middleware is just a URL rewriter and it never serves the static files. The job of this middleware is to simply rewrite the incoming URL to the default file which will then be served by the Static Files Middleware.

**How to set Custom HTML Page as the Default Page?**

The **UseDefaultFiles()** middleware will search the **wwwroot** folder for the following files.

1. **index.htm**
2. **index.html**
3. **default.htm**
4. **default.html**

This is the default behavior. But if you want then you can also change this default behavior. For Example, let us add another HTML page into the project wwwroot folder with the name MyCustomPage1.html. Once you add the MyCustomPage1.html file then the wwwroot folder contains two HTML files as shown in the below image.



Now, open the MyCustomPage1.html file and then copy and paste the following code in it.

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** charset="utf-8" **/>**

**<title></title>**

**</head>**

**<body>**

**<h1>** This is MyCustomPage1 HTML File**</h1>**

**</body>**

**</html>**

**Setting MyCustomPage1.html as Default Page:**

Now, we want the MyCustomPage1.html page to be our default page instead of the index.html page. To do, you need to modify the Configure() method of the Startup class as follows. Here, we are creating an instance of DefaultFilesOptions class and adding the default file name as MyCustomPage1.html and then passing this object to the UseDefaultFiles middleware.

**public** **class** Startup

**{**

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

//Specify the MyCustomPage1.html as the default page

DefaultFilesOptions defaultFilesOptions = new DefaultFilesOptions**()**;

defaultFilesOptions.DefaultFileNames.Clear**()**;

defaultFilesOptions.DefaultFileNames.Add**(**"MyCustomPage1.html"**)**;

//Setting the Default Files

app.UseDefaultFiles**(**defaultFilesOptions**)**;

//Adding Static Files Middleware to serve the static files

app.UseStaticFiles**()**;

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Request handled by the terminating middleware"**)**;

**})**;

**}**

**}**

Now run the application and you will see the output as expected that is coming from the MyCustomPage1.html file as shown below. If you still see the output from the index.html page then it may be due to cache so just try to reload the page. If still, you are not getting the data from the MyCustomPage1.html file then just restart the visual studio.



**What is the use of the UseFileServer() Middleware component?**

The UseFileServer() middleware components combines the functionality of UseStaticFiles, UseDefaultFiles and UseDirectoryBrowser middleware. We already discussed the UseStaticFiles and UseDefaultFiles middleware. The DirectoryBrowser middleware as the name says enables the directory browsing which allows the users to see the files which are stored in a specific directory. In our example, we can replace the UseStaticFiles() and UseDefaultFiles() middlewares with the UseFileServer() Middleware as shown below.

**public** **class** Startup

**{**

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

// Use UseFileServer instead of UseDefaultFiles & UseStaticFiles

FileServerOptions fileServerOptions = new FileServerOptions**()**;

fileServerOptions.DefaultFilesOptions.DefaultFileNames.Clear**()**;

fileServerOptions.DefaultFilesOptions.DefaultFileNames.Add**(**"MyCustomPage1.html"**)**;

app.UseFileServer**(**fileServerOptions**)**;

app.Run**(async** **(**context**)** =**>**

**{**

**await** context.Response.WriteAsync**(**"Request handled by the terminating middleware"**)**;

**})**;

**}**

**}**

Now run the application and you will see the output as expected.

**Developer Exception Page Middleware in ASP.NET Core**

**Developer Exception Page Middleware in ASP.NET Core Application**

In this article, I am going to discuss how to handle an unhandled exception using **Developer Exception Page Middleware in ASP.NET Core** application. Please read our previous article where we discussed [**How to configure the Default Page in ASP.NET Core**](https://dotnettutorials.net/lesson/configuring-default-page-asp-net-core/) Application. The exception handling is one of the key features of any application. We can handle the exception in many different ways. But in this article, we are going to discuss how we can use the Developer Exception Page Middleware to handle the unhandled exception. As part of this article, we are going to discuss the following concepts.

1. **What is Developer Exception Page Middleware?**
2. **How to use Developer Exception Page Middleware in ASP.NET Core Application?**
3. **How to Customize the UseDeveloperExceptionPage Middleware in ASP.NET Core?**
4. **Where do we need to configure the UseDeveloperExceptionPage Middleware?**

**Understanding Developer Exception Page Middleware in ASP.NET Core:**

First, create an ASP.NET Core Application with the Empty Project template. By default, ASP.NET Core application simply returns a status code for an exception that is not handled by the application. Let us understand this with an example. Please modify the Configure() method of the startup class as shown below where we throw an exception.

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapGet**(**"/", **async** context =**>**

**{**

**throw** new Exception**(**"Error Occurred while processing your request"**)**;

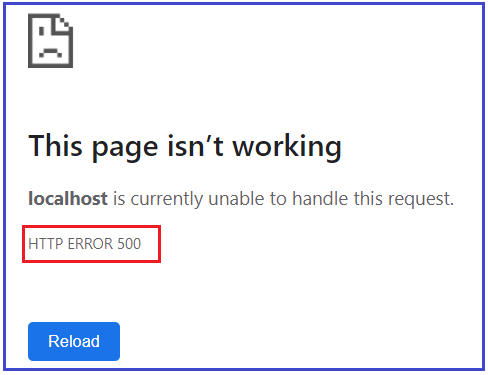
**await** context.Response.WriteAsync**(**"Hello World!"**)**;

**})**;

**})**;

**}**

When you run the application, you will get the following output.



As you can see in the above image it gives you the status code as 500 which means Internal Server Error. But as a developer when you are developing the application, you should know the detailed information about the exception on the page so that you can take necessary actions to fix the error. And this where DeveloperExceptionPage Middleware comes into the picture.

**How to use DeveloperExceptionPage Middleware in ASP.NET Core Application?**

If you want your application to display a page that shows the detailed information about the unhandled exception, then you need to configure the Developer Exception Page middleware in the request processing pipeline. To do so, modify the Configure() method of the Startup class as shown below to add the Developer Exception Page middleware which will handle the unhandled exception that occurred in your application.

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapGet**(**"/", **async** context =**>**

**{**

**throw** new Exception**(**"Error Occurred while processing your request"**)**;

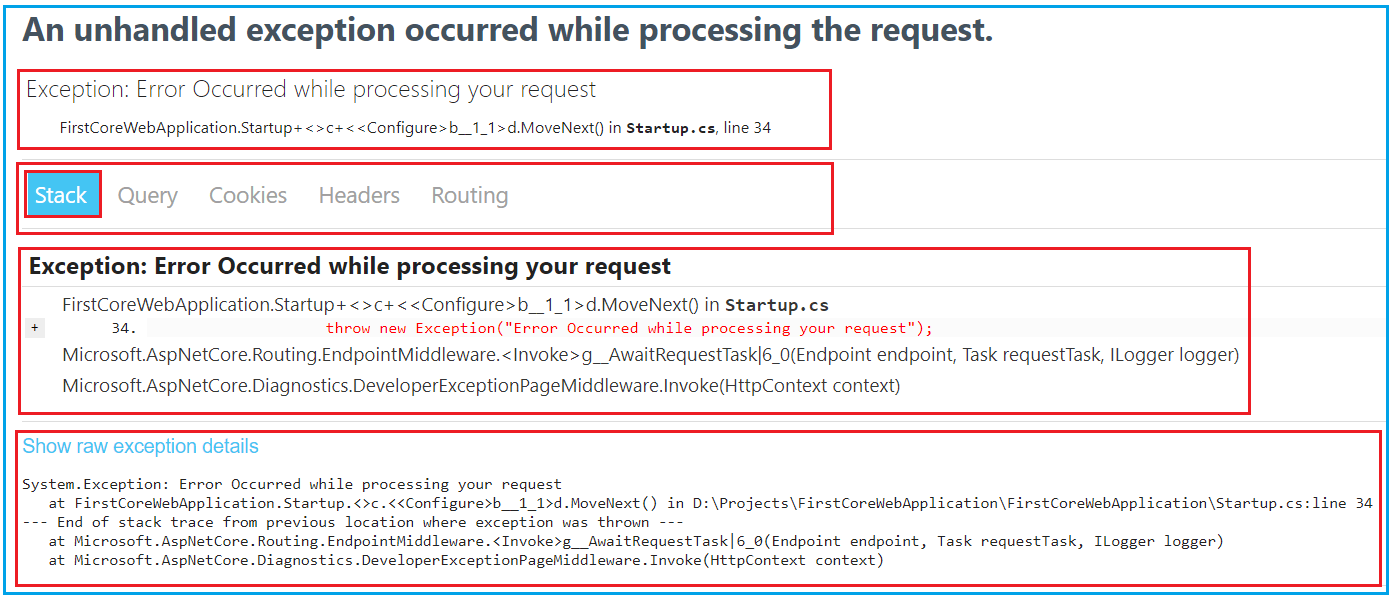
**await** context.Response.WriteAsync**(**"Hello World!"**)**;

**})**;

**})**;

**}**

With the above change in place, now run the application and it should display the following page with the detailed information about the unhandled exception.



As you can see in the above image, the Developer Exception Page contains five tabs such as Stack, Queue, Cookies, Headers, and Routing.

1. **Stack**: The Stack tab gives the information of stack trace which indicated where exactly the exception occurred, the file name, and the line number that causes the exception.
2. **Query**: The Query tab gives information about the query strings.
3. **Cookies**: The Cookies tab displays the information about the cookies set by the request.
4. **Header**: The Header tab gives information about the headers which is sent by the client when makes the request.
5. **Route**: The Route tab gives information about the Route Pattern and Route HTTP Verb type of the method, etc.

Now if you verify the Query tab and Cookies tab, then you will not see any information as you are not passing any query string value in the URL or you are not setting the cookies in the request. In our upcoming articles, we will discuss the Query string and Cookies in detail.

**Note:**Please Enable the Developer Exception Page Middleware only when the application is running in the Development environment. You don’t want to share detailed exception information when the application is running in the production environment.

**How to Customize the UseDeveloperExceptionPage Middleware in ASP.NET Core?**

If you want then you can also customize the UseDeveloperExceptionPage middleware. The point that you need to remember is whenever you want to customize a middleware component in ASP.NET Core then you need to use the respective Options object. For example

1. **UseDeveloperExceptionPage =>** to customize this middleware use DeveloperExceptionPageOptions object
2. **UseDefaultFiles =>** to customize this middleware use DefaultFilesOptions object
3. **UseStaticFiles =>** to customize this [**middleware use StaticFileOptions**](https://dotnettutorials.net/lesson/static-files-middleware-asp-net-core/) object
4. **UseFileServer =>** to customize this middleware use FileServerOptions object

As we are going to customize the UseDeveloperExceptionPage() [**middleware componen**](https://dotnettutorials.net/lesson/asp-net-core-middleware-components/)t, so we need to use the DeveloperExceptionPageOptions object. So, modify the Configure method of the Startup class as shown below.

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

**if** **(**env.IsDevelopment**())**

**{**

DeveloperExceptionPageOptions developerExceptionPageOptions = new DeveloperExceptionPageOptions

**{**

SourceCodeLineCount = 5

**}**;

app.UseDeveloperExceptionPage**(**developerExceptionPageOptions**)**;

**}**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapGet**(**"/", **async** context =**>**

**{**

**throw** new Exception**(**"Error Occurred while processing your request"**)**;

**await** context.Response.WriteAsync**(**"Hello World!"**)**;

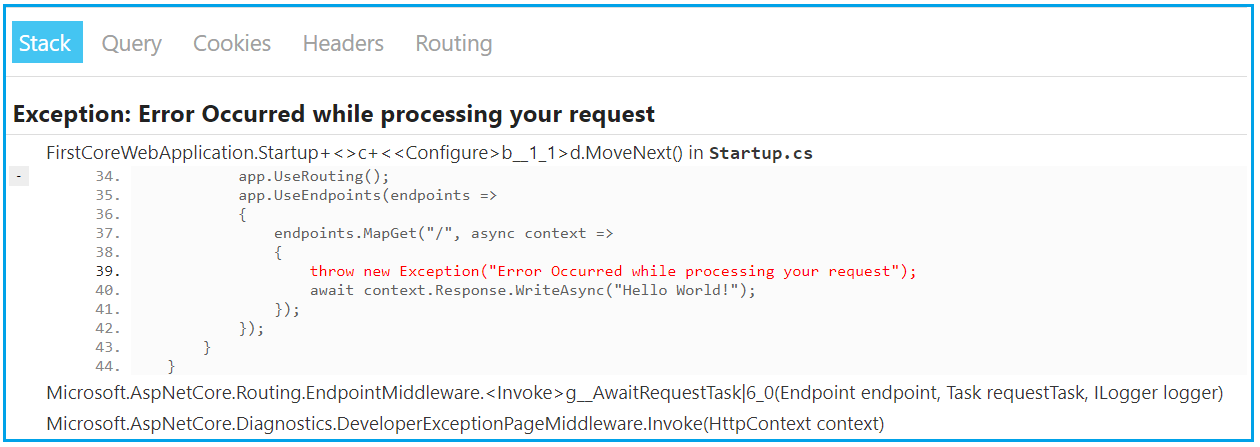
**})**;

**})**;

**}**

As you can see in the above code, we are using one property called **SourceCodeLineCount.** The **SourceCodeLineCount**property of the **DeveloperExceptionPageOptions**class specifies the number of lines of code to include before and after the line of code that caused the exception.

Now if you run the application with the above changes in place, then you will get the following error. Please have a look at the line number of the error i.e. 39. And also please look at the numbers of lines before and after the error line.



**Where do we need to configure the UseDeveloperExceptionPage Middleware?**

We need to configure the UseDeveloperExceptionPage() Middleware as early as possible in the application’s request processing pipeline so that it can handle the unhandled exception and then display the Developer Exception Page with the detailed information about the exception.

Let us see what happened when we configure the UseDeveloperExceptionPage() middleware after the middleware which is causing the exception. Please modify the Configure() method as shown below.

**public** **void** Configure**(**IApplicationBuilder app, IWebHostEnvironment env**)**

**{**

app.UseRouting**()**;

app.UseEndpoints**(**endpoints =**>**

**{**

endpoints.MapGet**(**"/", **async** context =**>**

**{**

**throw** new Exception**(**"Error Occurred while processing your request"**)**;

**await** context.Response.WriteAsync**(**"Hello World!"**)**;

**})**;

**})**;

**if** **(**env.IsDevelopment**())**

**{**

DeveloperExceptionPageOptions developerExceptionPageOptions = new DeveloperExceptionPageOptions

**{**

SourceCodeLineCount = 5

**}**;

app.UseDeveloperExceptionPage**(**developerExceptionPageOptions**)**;

**}**

**}**

With the above changes in place, when we run the application, it will not display the developer’s exception page instead of simply returns the default error status code. This is the reason why we need to configure the **UseDeveloperExceptionPage()** middleware as early as possible to handle the unhandled exception of the application in the [**request processing pipeline**](https://dotnettutorials.net/lesson/asp-net-core-request-processing-pipeline/).

**ASP.NET Core Command Line Interface**

**ASP.NET Core Command Line Interface (.NET Core CLI)**

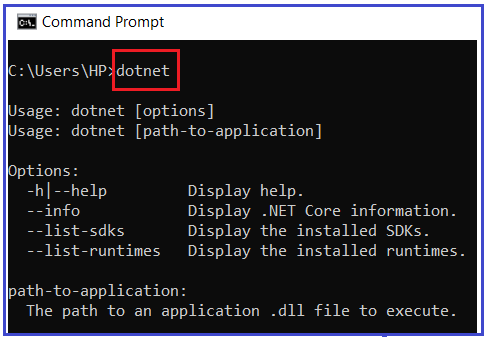
In this article, I am going to discuss how to create, build, and run ASP.NET Core Application using **.NET Core CLI (Command Line Interface)** Commands. Please read our previous article where we discussed [**Developer Exception Page Middleware in ASP.NET Core**](https://dotnettutorials.net/lesson/developer-exception-page-middleware-asp-net-core/)Application.

**ASP.NET Core Command Line Interface:**

The .NET Core CLI (Command Line Interface) is a new cross-platform tool that is used for creating, restoring packages, building, running, and publishing ASP.NET Core Applications. The .NET Core CLI command for any kind of web application uses Out of Process hosting i.e. it uses the Kestrel server to run the application.

As of now, all the applications we created are using Visual Studio. Visual Studio internally uses this .NET CLI commands to restore, build, and publish the applications. Other higher-level IDEs, editors, and tools, for example, Visual Studio Code uses these CLI Commands to support to create, restore, publish, and run .NET Core applications.

When we installed .NET Core SDK, then by default the .NET Core CLI is also installed. So, we don’t require installing it separately on the development environment i.e. on our local machine. We can verify the same i.e. whether the .NET CLI is installed or not using the command prompt. To verify the same, open command prompt (Windows), terminal (Linux) and type “**dotnet**” and press enter as shown in the below. If it displays usage and helps options as you can see in the below image means the .NET Core CLI is installed properly.



**.NET Core CLI Command Structure:**

The .NET Core CLI command structure is nothing but how we write the .NET Core CLI command. The following is the command structure of .NET Core CLI Command:

**dotnet <command> <argument> <option>**

**Note:** All the .NET Core CLI commands start with the driver named **dotnet**. The driver i.e. dotnet starts the execution of the specified command. After dotnet, we need to specify the **command** (also known as the verb) to perform a specific action. Each command can be followed by **arguments** and **options**.

**How to get the all the .NET Core Commands:**

Open the command prompt and type **dotnet help** and press enter which will display all the .NET Core CLI commands. Some of the command and their use is given below.

1. **add**: Add a package or reference to a .NET project.
2. **build**: Build a .NET project.
3. **build-server**: Interact with servers started by a build.
4. **clean**: Clean build outputs of a .NET project.
5. **help**: Show command-line help.
6. **list:** List project references for a .NET project.
7. **msbuild:** Run Microsoft Build Engine (MSBuild) commands.
8. **new**: Create a new .NET project or file.
9. **nuget**: Provides additional NuGet commands.
10. **pack**: Create a NuGet package.
11. **publish**: Publish a .NET project for deployment.
12. **remove**: Remove a package or reference from a .NET project.
13. **restore**: Restore dependencies specified in a .NET project.
14. **run**: Build and run a .NET project output.
15. **sln**: Modify Visual Studio solution files.
16. **store**: Store the specified assemblies in the runtime package store.
17. **test**: Run unit tests using the test runner specified in a .NET project.
18. **tool**: Install or manage tools that extend the .NET experience.
19. **vstest**: Run Microsoft Test Engine (VSTest) commands.

**Project Modification Commands:**

1. **add package:** Adds a package reference to a project.
2. **add reference:** Adds project-to-project (P2P) references.
3. **remove package:** Removes package reference from the project.
4. **remove reference:** Removes project reference.
5. **list reference:** Lists all project-to-project references.

**Advanced Command:**

1. **nuget delete:** Deletes or un-lists a package from the server.
2. **nuget locals:** Clear or lists NuGet resources.
3. **nuget push:** Pushes a package to the server and publishes it.
4. **msbuild:** Builds a project and all of its dependencies.
5. dotnet install script: Script used to install .NET Core CLI tools and the shared runtime.

**Create a New Project using the .NET Core CLI Command:**

Let’s create, restore, build, and run .NET Core console application using the command-line interface without using Visual Studio. To create a new .NET Core project, we have to use the ‘new’ command followed by the template name argument. We can create the console, class library, web, webapp, mvc, webapi, razor, angular, react, etc. project using CLI.

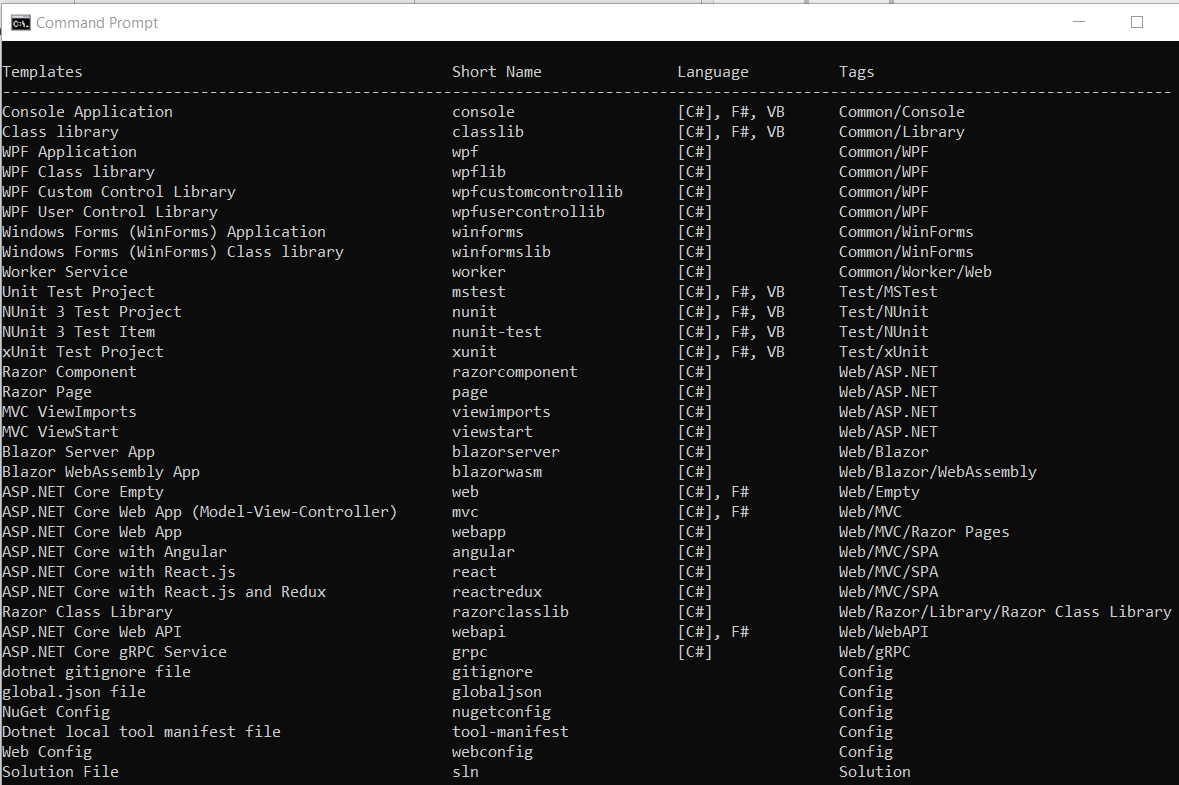
The following command creates a new dotnet core project using the TEMPLATE:

**dotnet new <TEMPLATE>**

You can find out the list of templates using:

**dotnet new -l**

Once you type **dotnet new -l**and press enter, it will show you the list of available templates based on .NET Core Version installed on your machine as shown in the below image:

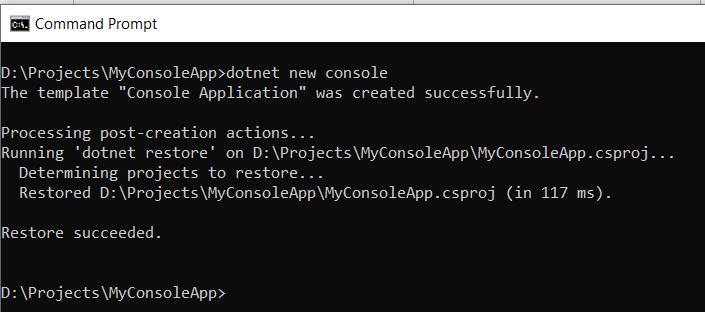


**Example: Creating a Console Application using .NET Core CLI**

The following command creates a new console project in the current directory with the same name as the current directory.

**D:\Projects\MyConsoleApp>dotnet new console**

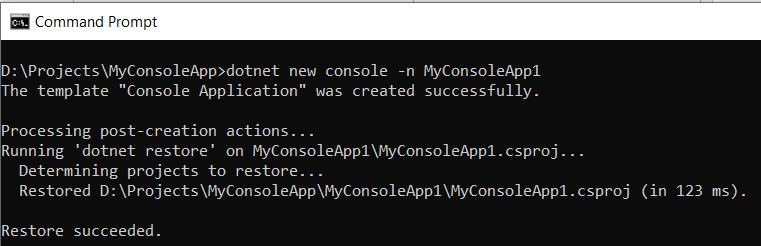
Once you execute the above command, it will create a console application and you will get the following output.



The following command creates a new console project named **‘MyConsoleApp1**“. The **-n or –name** option specifies the name of the project.

**D:\DotNetCoreApps>dotnet new console -n MyConsoleApp1**

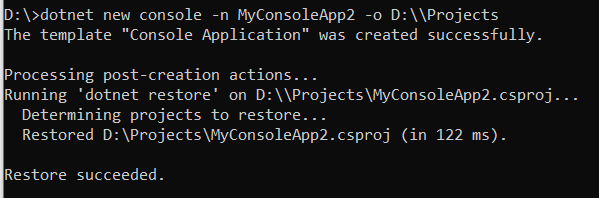
Once you execute the above command, it will create a new console application with the name MyConsoleApp1 and you will get the following output.



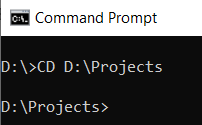
The following command creates a new console application named MyConsoleApp2 to \Projects directory. The **-o or –output** option is used to specify an output directory where the project should be generated.

**dotnet new console -n MyConsoleApp2 -o D:\\Projects**

Once you execute the above command, it will create a new console application with the name MyConsoleApp2 with the D=>Projects directory and you will get the following output.



After creating a project, navigate to the project directory (folder) in the command prompt to apply project-specific commands. As we created the project in **D=>Projects** folder, then go to the **D => Projects** directory in the command prompt as shown below.

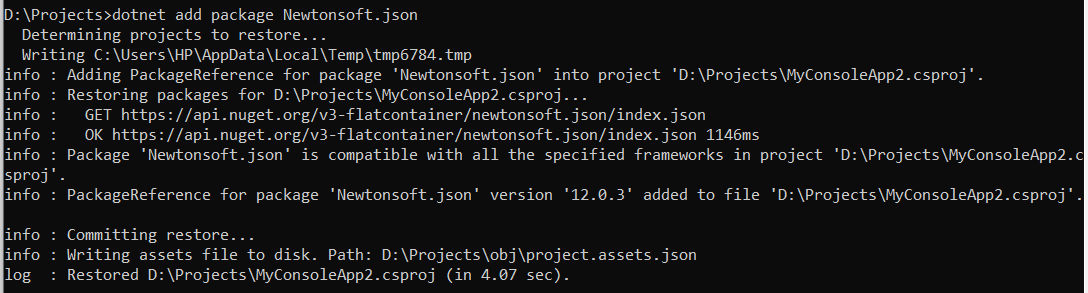


**Add Package Reference using .NET Core CLI Command:**

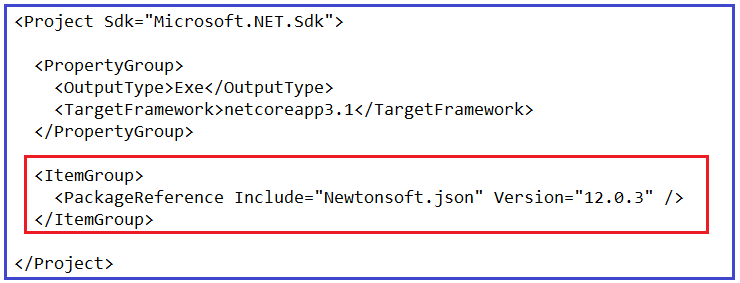
We often need to add the NuGet package references for different purposes. For example, apply the following command to add **Newtonsoft.json** package to your console project.

**dotnet add package Newtonsoft.json**

Once you type the above command and press enter then you should get the following output.



This will add the Newtonsoft.json package to your project. You can verify the same in the project file. So, open .csproj file and you should get the following.



**Remove Package Reference using .NET Core CLI Command:**

The “**dotnet remove package**” command provides a convenient option to remove a NuGet package reference from a project. If you want to remove the NuGet Package that we just installed Newtonsoft.json then you need to execute the below command,

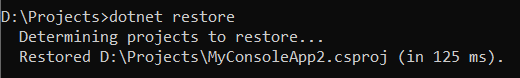
**dotnet remove package Newtonsoft.json**

So, in the command prompt and type “**dotnet remove package Newtonsoft.json**” and press enter as shown in the below image which will remove the Newtonsoft.json package from your project and you verify the same in the project file.

Remove Package Reference using .NET Core CLI Command

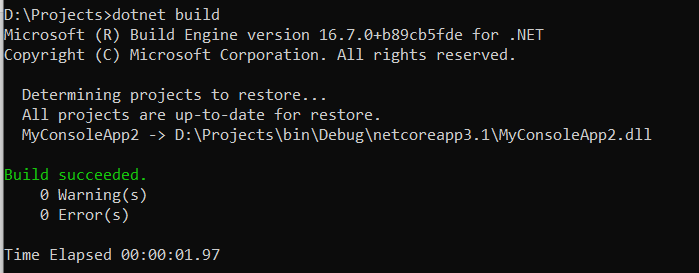
**Restore Packages using .NET Core CLI Command:**

To restore packages or to update existing packages in your project, you can use the “**dotnet restore**” command as below:



**Build Project using .NET Core CLI Command:**

In order to build a new or existing project, we need to use the “**dotnet build**” command as below which will build your .NET Core Project:



**Run .NET Core Project using .NET Core CLI Command:**

To run the .NET Core project, we need to use “**dotnet run**” command as shown below: Here, you can see it display the output Hello World!

Run .NET Core Project using .NET Core CLI Command