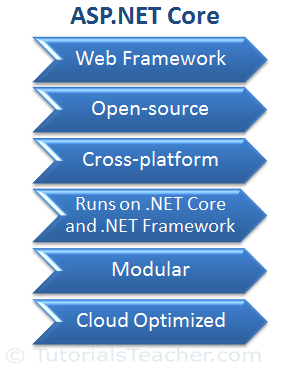
**ASP.NET Core Notes**

Prerequisites

Basic knowledge of C#, HTML, Visual Studio, and Object Oriented Programming is required.

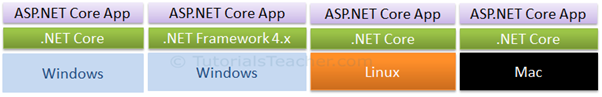
ASP.NET Core is a new version of ASP.NET by Microsoft. It is an open-source web framework which can be run on Windows, Mac, or Linux.

You can say that it is the new version of ASP.NET. The framework is a complete rewrite from scratch in order to make it open source, modular and cross-platform. It was initially launched as ASP.NET 5 but then it was renamed to ASP.NET Core.



ASP.NET Core is a modular framework distributed as NuGet packages. This allows us to include packages that are required in our application.

ASP.NET Core applications run on both, .NET Core and traditional .NET framework (.NET Framework 4.x).



ASP.NET Core is designed to be deployed on cloud as well as on-premises. Developers can now build cloud-based web applications, IoT (Internet of Thing) and mobile backend applications using ASP.NET Core framework which can run on Windows, Linux, and Mac operating systems.

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 respective repositories on Github.

## ASP.NET Core Version History

| Version | Release Date |
| --- | --- |
| ASP.NET Core 2.0 | August 2017 |
| ASP.NET Core 1.1 | November 2016 |
| ASP.NET Core 1.0 | June 2016 |

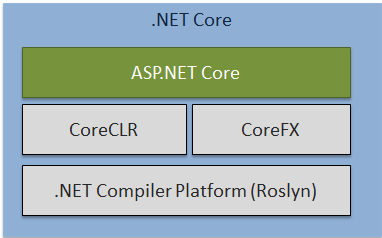
## .NET Core

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 and services that run on Windows, Linux and Mac. So, it is a platform on which ASP.NET Core application runs.

.NET Core is named "Core" because it includes core features of the .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 app and other advanced features that can be included as a package from NuGet.

**The following figure illustrates the components of .NET Core.**



.NET Core Components

As you can see above, .NET Core includes .NET Compiler platform Roslyn, .NET Core runtime CoreCLR, .NET Core framework CoreFX and ASP.NET Core . ASP.NET Core is a part of .NET Core SDK so you don't need to install ASP.NET Core separately. ASP.NET Core and .NET Core is a part of .NET Foundation.

.NET Core comes under MIT or Apache 2 licenses. Visit .NET Core repository on Github to contribute or download the source code.

## Why ASP.NET Core?

* **Supports Multiple Platforms:** ASP.NET Core applications can run on Windows, Linux, and Mac. So you don't need to build different apps for different platforms using different frameworks.
* **Fast:** ASP.NET Core no longer depends on System.Web.dll for browser-server communication. ASP.NET Core allows us to include packages which we need for our application. This reduces the request pipeline and improves the performance and scalability.
* **IoC Container:** It includes built-in IoC container for automatic dependency injection which makes it maintainable and testable.
* **Integration with Modern UI Frameworks:** It allows you to use and manage modern UI frameworks such as AngularJS, ReactJS, Umber, Bootstrap etc. using Bower (a package manager for the web).
* **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 which can be used with other .NET frameworks such as .NET Framework 4.x or Mono. Thus a single code based can be shared across frameworks.
* **Side-by-Side App Versioning:** ASP.NET Core runs on .NET Core which supports simultaneous running of multiple versions of applications.
* **Smaller Deployment Footprint:** ASP.NET Core application runs on .NET Core which is smaller than full .NET Framework. So, the application which uses only a part of .NET CoreFX will have smaller deployment size. This reduces the deployment foot print.

# ASP.NET Core - Development Environment Setup

To develop ASP.NET Core application, the following must be installed in your system:

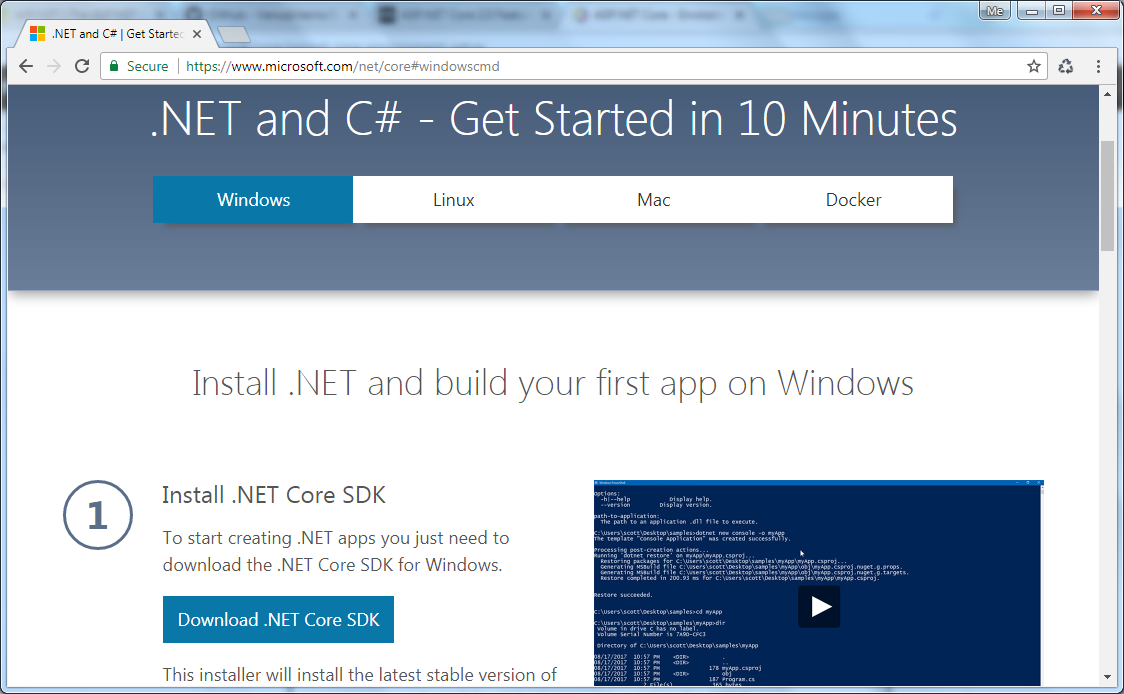
1. .NET Core SDK
2. Integrated Development Environment (IDE)

ASP.NET Core is a part of .NET Core SDK, so you don't need to install it separately. As of this writing, the current release is .NET Core 1.1. Read .NET Core Release Notes to know more about all the releases.

## Install .NET Core SDK

.NET Core SDK can be installed on the platform you are using such as Windows, Linux or Mac.

Go to https://www.microsoft.com/net/core and select the platform you are using. Here, we use Windows so select Windows as shown below.



Install .NET Core SDK for Windows

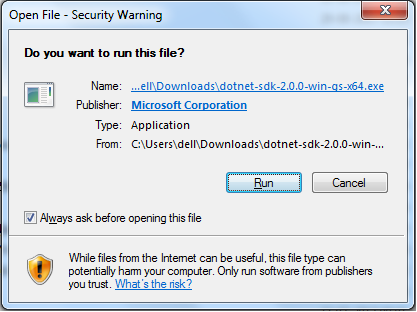
As you can see above, click on the **Download .NET Core SDK** button to download the latest version of .NET Core SDK installer. It will download .NET Core 2.0 SDK as of this writing.

**Alternatively,**

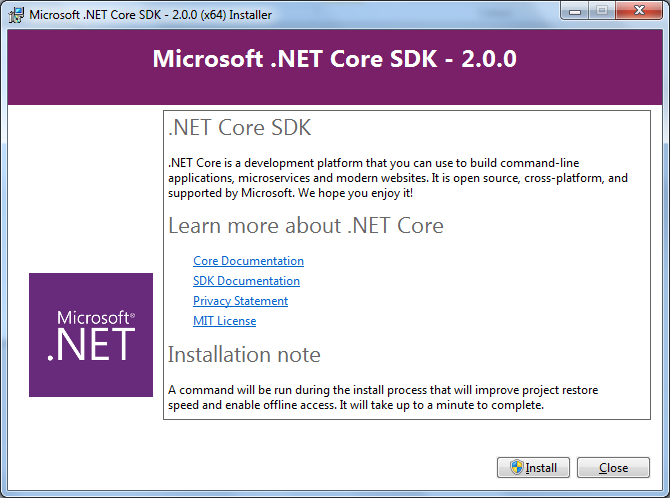
**Download .NET Core SDK:** Download .NET Core SDK for different platform from here.

**Download .NET Core Runtime:** Download .NET Core Runtime for different platform from here. Remember, .NET Core runtime is used only to run .NET Core application but not for the development.

After downloading installer, click on it to start the installation of .NET Core 2.0 SDK.



Click **Run** to go to the next step as shown below.



Click on **Install** button to install .NET Core 2.0 SDK.

## IDE

You can develop, restore, build and run .NET Core application either with Visual Studio or with command line interface for .NET Core. Here, we will use Visual Studio 2017 to develop .NET Core 2.0 applications.

### Visual Studio 2017:

You can download Visual Studio 2017 installer from the same page https://www.microsoft.com/net/core. Click on the **Download Visual Studio** button to download Visual Studio 2017 Community edition. Alternatively, you can go to https://www.visualstudio.com/downloads/ and download installer for the specific Visual Studio edition.

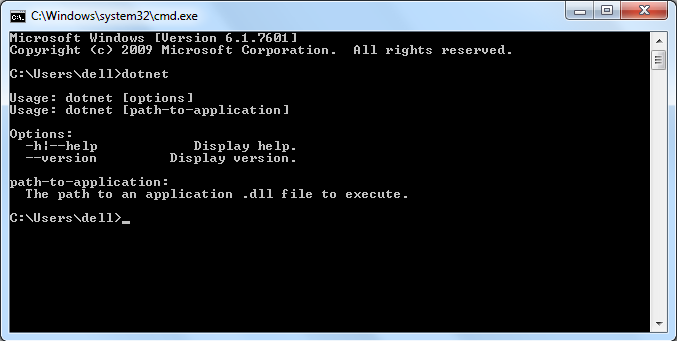
After installation, you can start to create .NET Core applications.

### Command-line Interface (CLI):

If you do not use Visual Studio for .NET core application development for some reason and want to use different IDE then you can use command-line interface to create, compile, build, restore and run .NET Core application.

.NET Core SDK installation also installs command-line interface for the selected platform. It installs the latest stable version of the tools and put them on your PATH so you can run dotnet from the Console.

Once installed, you can verify it by opening command prompt and type **dotnet**and press **Enter**. This will display installed version and usage information as shown below.

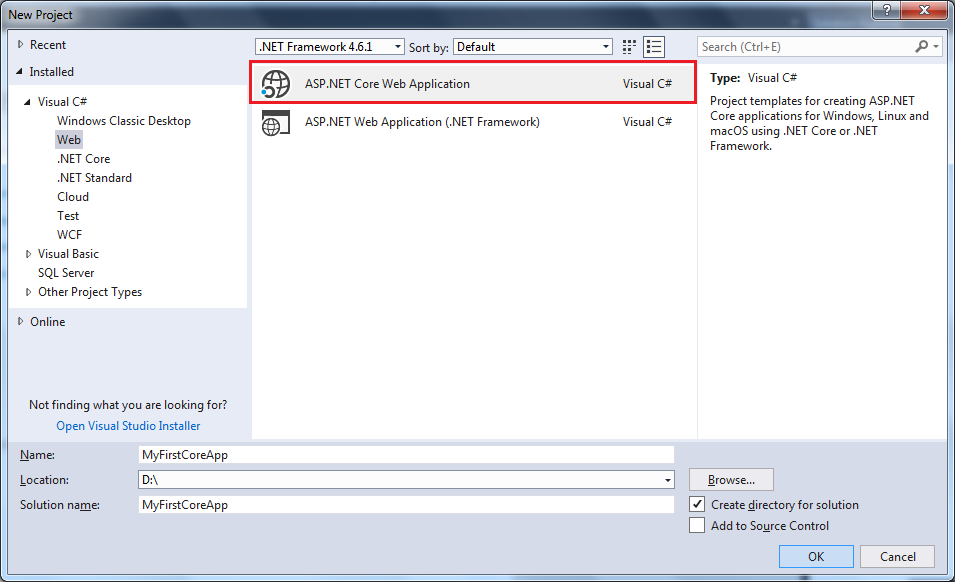


# First ASP.NET Core Application

Here, we will learn how to create our first .NET core 2.0 application. We will use Visual Studio 2017 to create ASP.NET Core 2.0 web application.

The first step is to open Visual Studio. Click on File->New, and click on Projects.

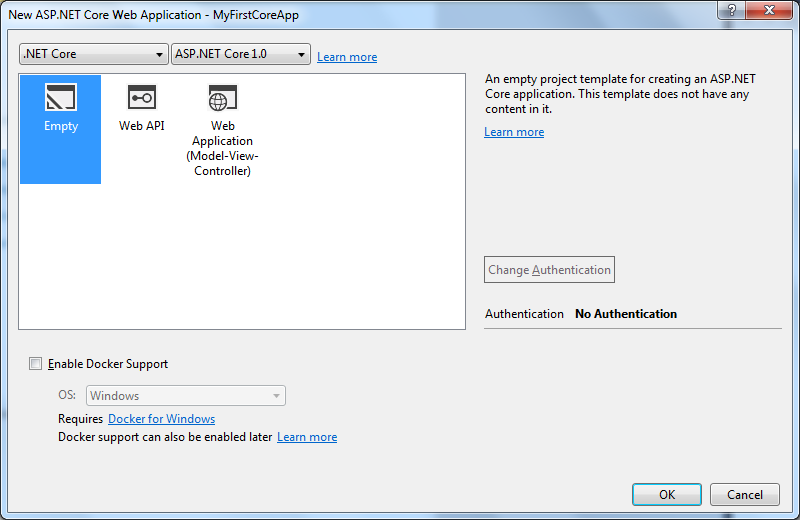
In the New Project dialog box, click on the Templates node. Expand the Templates node, then expand Visual C#, and click on the Web template.

ASP.NET Project Templates

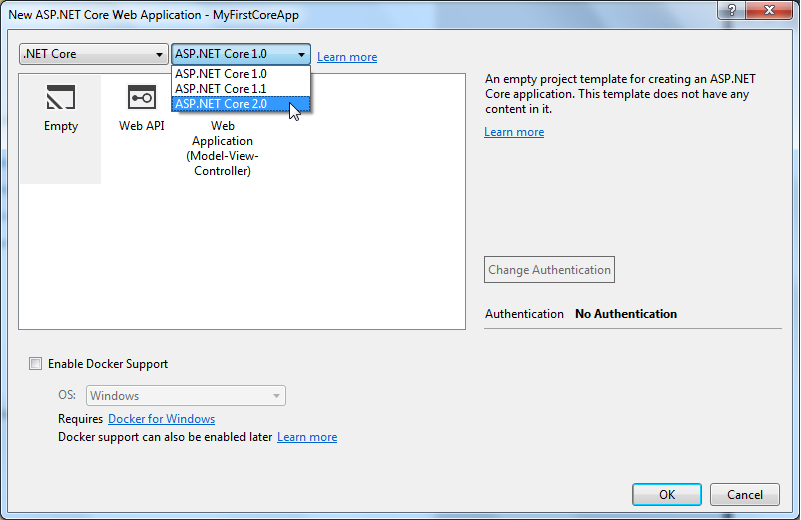
As shown above, the middle pane on the New Project dialog box includes the following two templates for ASP.NET Web projects:

* **ASP.NET Core Web Application** - Select this template to create a new crossplatform compatible ASP.NET Core web application project that runs on the .NET Core framework.
* **ASP.NET Web Application (.NET Framework)** - Select this template to create a new ASP.NET web application project that runs on standard .NET Framework.

Here, we want to create a cross-platform ASP.NET Core web application. So, select **ASP.NET Core Web Application** template. Give the appropriate name, location, and the solution name for the ASP.NET Core application. In this example, we will give the name MyFirstCoreApp, and click on the **OK**button. This will open another popup as shown below.

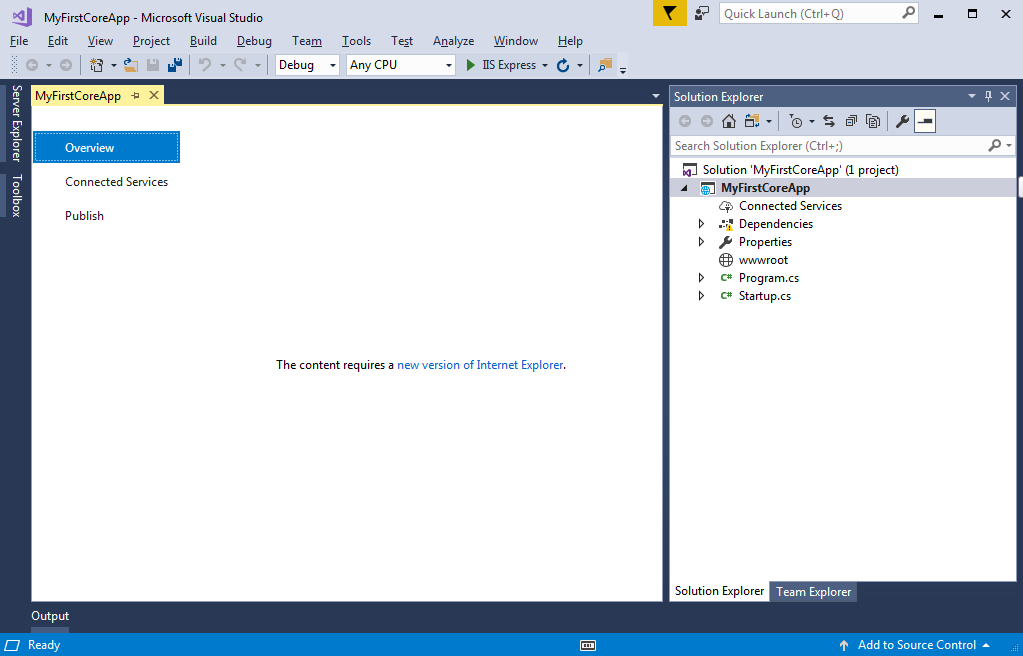
ASP.NET Core Templates

As you can see, we can select the version of the framework we want to use in our application. We are going to use .NET Core 2.0 framework here. So select ASP.NET Core 2.0 in the dropdown as shown below.



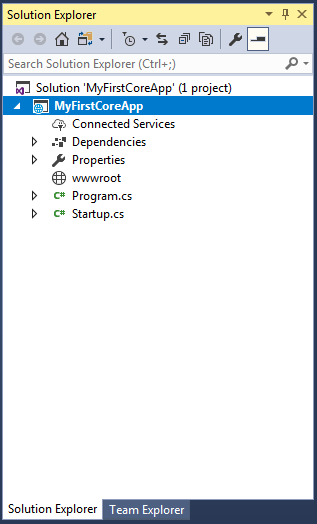
ASP.NET Core Version

Now, select an **Empty** ASP.NET Core template in order to understand the basics of ASP.NET core web application. We will not use Docker support or authentication here, so click on **OK** to create an ASP.NET core web application as shown below.

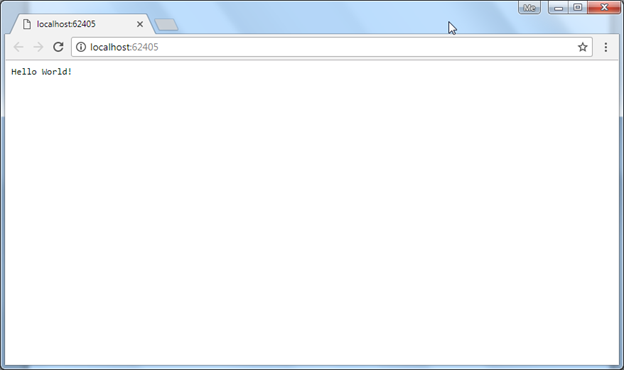


ASP.NET Core Web Project in Visual Studio

Wait for some time till Visual Studio restores the packages in the project. Restoring process means Visual Studio will automatically add, update or delete configured dependencies as NuGet packages in the project. The entire project structure of the created project will look like below.

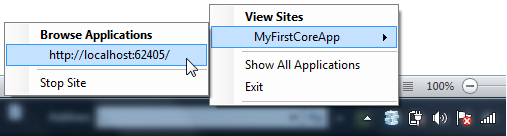
ASP.NET Core Project Structure

We will understand the project structure in the next chapter. To run this web application, go to Debug menu and click on Start without Debugging, or press Ctrl + F5. This will open the browser and display the following result.



The above output "Hello World!" comes from the Configure method of Startup class in the Startup.cs file in the project. Open Startup.cs file and see Configure method. Change "Hello World!" string to something else and it will change the output accordingly. Learn about Startup class in the Startup.cs chapter.

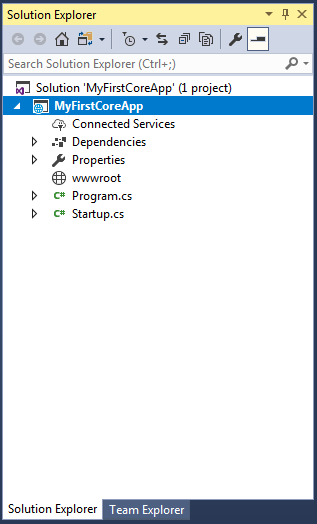
You can also see the IIS express icon on the system tray. Right click on it. You can see the ASP.NET sites currently running in your development machine.

ASP.NET Core app in System tray

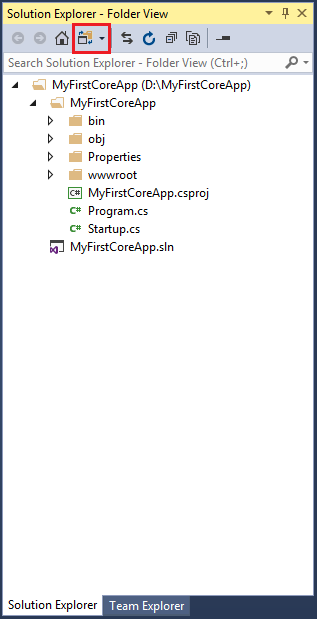
# ASP.NET Core - Project Structure

In the previous chapter, we created our first ASP.NET Core 2.0 web application. Here, you will learn about the project structure and significance of each file created by ASP.NET Core application template in Visual Studio 2017.

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

ASP.NET Core Project Structure

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

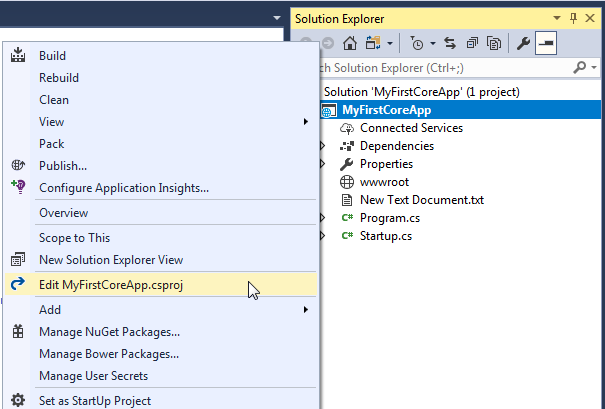
Solution Explorer - Folder View

 Note:

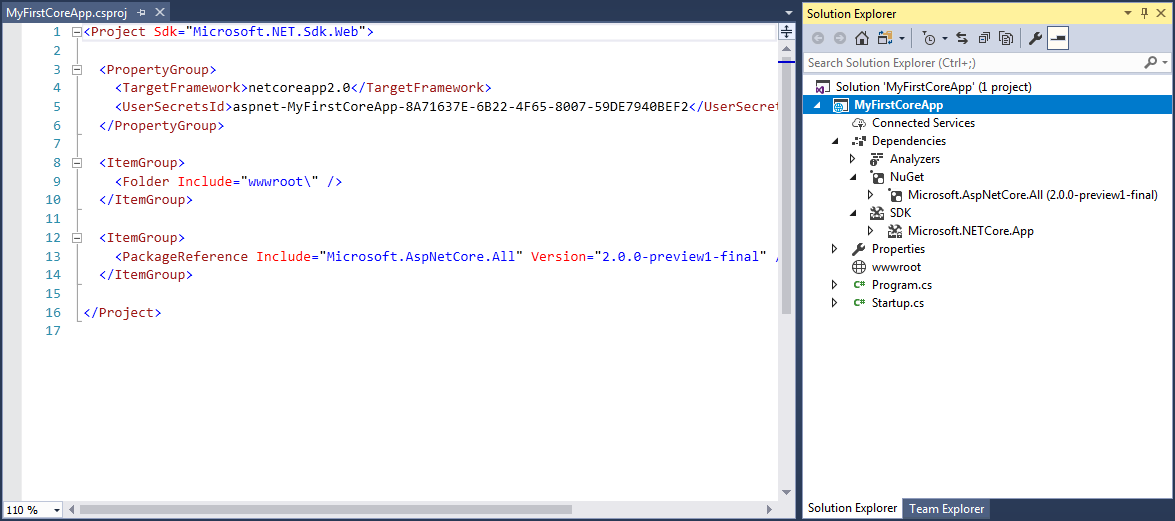
ASP.NET Core project files and folders are synchronized with physical files and folders. If you add a new file or folder in project folder then it will directly reflect in the solution explorer. You don't need to add it in the project explicitly by right clicking on the project.

## .csproj

ASP.NET Core 1.0 does not create .csproj file, instead, it uses .xproj and project.json files to manage the project. This has changed in ASP.NET Core 2.0. Visual Studio now uses .csproj file to manage projects. We can edit the .csproj settings by right clicking on the project and selecting **Edit <project-name>.csproj** as shown below.

Edit .csproj

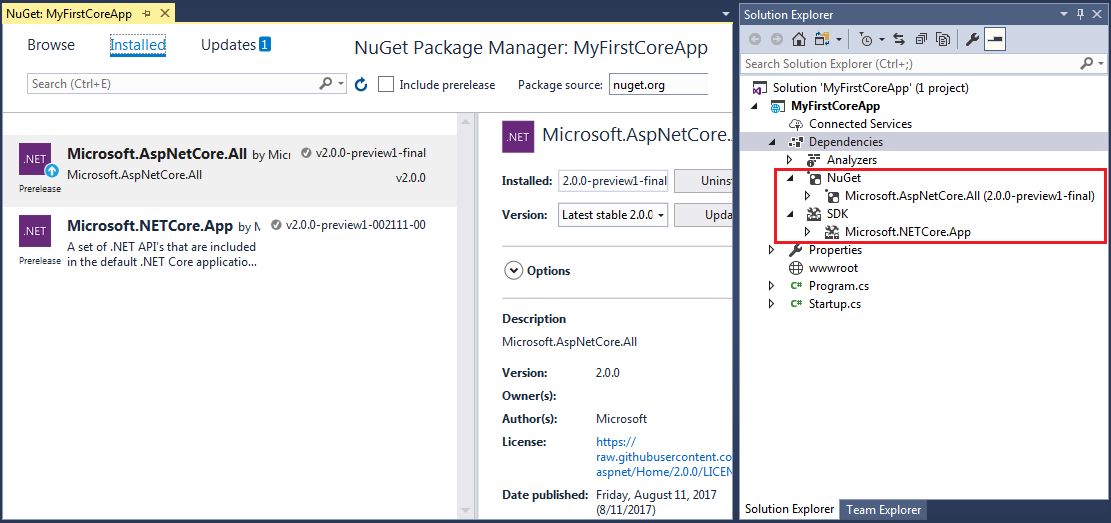
The .csproj for the above project looks like below.

Edit .csproj

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

## Dependencies

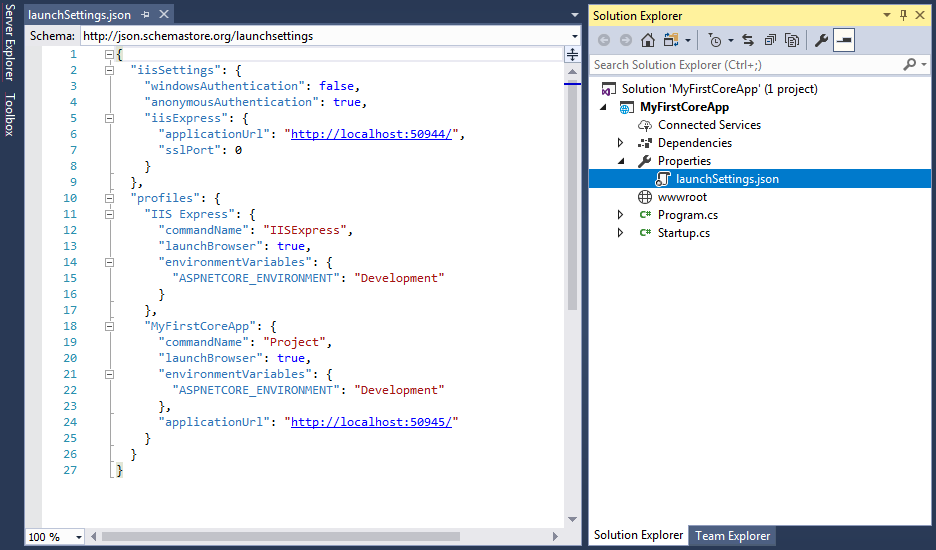
The Dependencies in the ASP.NET Core 2.0 project contain all the installed server-side NuGet packages as well as client-side frameworks such as jQuery, AngularJS, Bootstrap etc. These client-side dependencies are managed using Bower in Visual Studio.

Dependencies

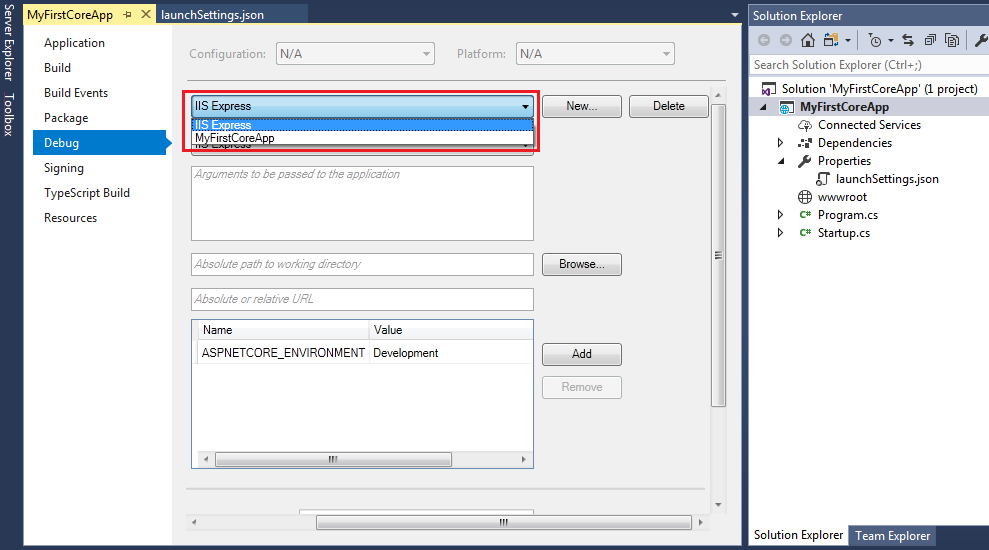
As you can see above, **dependencies** node in solution explorer displays installed NuGet packages. This also includes *bower* folder which has all the client-side frameworks library installed it using Bower.

## Properties

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

launchSettings.json

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

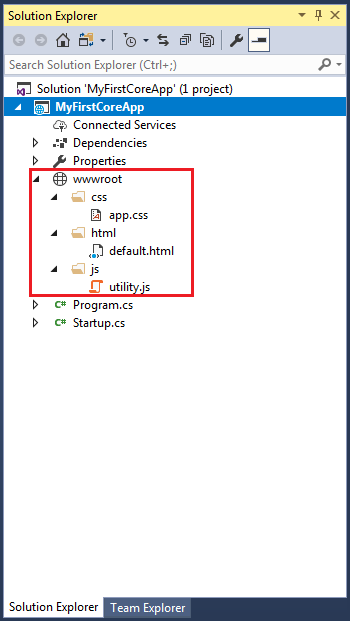
Project Properties

# ASP.NET Core - wwwroot Folder

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

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

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



wwwroot

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

Remember, you need to include a middleware for serving static files in the Configure method of Startup.cs. Learn more about it in Serving Static File section.

## Rename wwwroot Folder

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

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

public class Program

{

public static void Main(string[] args)

{

var host = new WebHostBuilder()

.UseKestrel()

.UseContentRoot(Directory.GetCurrentDirectory())

.UseWebRoot("Content")

.UseIISIntegration()

.UseStartup<MyStartup>()

.Build();

host.Run();

}

}

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

# ASP.NET Core - Program.cs

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

The steps for creating a host in ASP.NET Core 1.x is slightly different in ASP.NET Core 2.x. Let's understand Program class in ASP.NET Core 1.x application so that it will be easy to understand it in ASP.NET Core 2.x.

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

The following is a typical Program.cs in ASP.NET Core 1.x.

Program.cs

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Hosting;

namespace MyFirstCoreApp

{

public class Program

{

public static void Main(string[] args)

{

var host = new WebHostBuilder()

.UseKestrel()

.UseContentRoot(Directory.GetCurrentDirectory())

.UseIISIntegration()

.UseStartup<Startup>()

.Build();

host.Run();

}

}

}

Every ASP.NET Core web application requires a host to be executed. In the above Main() method, we configure a web hosting environment for the ASP.NET Core 1.x web application. A host must implement IWebHost interface. Let's understand the above code step by step.

First, var host = new WebHostBuilder()

**.UseKestrel()**

The UseKestrel() method is an extension method which specifies Kestrel as an internal web server. The Kestrel is an open-source, cross-platform web server for ASP.NET Core. It is designed to be used behind proxy because it has not yet matured to be exposed as a full-fledge web server.

ASP.NET Core application can be a cross-platform application so it can be used with any web server, and not only IIS. Hence, there will be an external web server such as IIS, Apache, Nginx etc. which will dispatch http requests to the internal web server Kestrel. Learn more about web servers in ASP.NET Core here.

**.UseContentRoot(Directory.GetCurrentDirectory())**

The UseContentRoot() method specifies the current directory as a root directory which will be **src** folder in the default ASP.NET Core project. The content root directory determines where the content files are located such as MVC view files, CSS, images etc.

**.UseIISIntegration()**

The UseIISIntegration() method specifies the IIS as the external web server or the reverse proxy server.

**.UseStartup<Startup>()**

The UseStartup<startup>() method specifies the Startup class to be used by the web host. Visual Studio creates Startup.cs by default with every new ASP.NET Core application. This Startup class is like Global.asax of .NET framework where you can configure request pipeline (middleware). We may give any other name to the Startup class instead of Startup. We just need to specify it as a generic parameter in UseStartup<T>() method. You will learn about it in the next chapter.

And lastly, the Build() method returns an instance of IWebHost using the configuration specified above.

So now, we have built our hosting environment and it's time to start the web application.

**host.Run();**

The Run() method starts the web application and blocks the calling thread till the host is shutdown. The command line application will become web application from this point onwards.

Thus, ASP.NET Core application starts from the Main() method of the Program class where you can build the hosting environment and start the web application.

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

The following is the Program class in ASP.NET Core 2.x:

Program.cs

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Hosting;

namespace MyFirstCoreApp

{

public class Program

{

public static void Main(string[] args)

{

BuildWebHost(args).Run();

}

public static IWebHost BuildWebHost(string[] args) =>

WebHost.CreateDefaultBuilder(args)

.UseStartup<Startup>()

.Build();

}

}

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

public static void Main(string[] args)

{

BuildWebHost(args).Run();

}

public static IWebHost BuildWebHost(string[] args)

{

return WebHost.CreateDefaultBuilder(args)

.UseStartup<Startup>()

.Build();

}

Let's understand hosting steps.

The WebHost is a static class which can be used for creating an instance of IWebHost and IWebHostBuilder with pre-configured defaults. The CreateDefaultBuilder() method creates a new instance of WebHostBuilder with pre-configured defaults. Internally, it configures Kestrel, IISIntegration and other configurations. The following is CreateDefaultBuilder() method from the source code on GitHub.

CreateDefaultBuilder()

public static IWebHostBuilder CreateDefaultBuilder(string[] args)

{

var builder = new WebHostBuilder()

.UseKestrel()

.UseContentRoot(Directory.GetCurrentDirectory())

.ConfigureAppConfiguration((hostingContext, config) =>

{

var env = hostingContext.HostingEnvironment;

config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)

.AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange: true);

if (env.IsDevelopment())

{

var appAssembly = Assembly.Load(new AssemblyName(env.ApplicationName));

if (appAssembly != null)

{

config.AddUserSecrets(appAssembly, optional: true);

}

}

config.AddEnvironmentVariables();

if (args != null)

{

config.AddCommandLine(args);

}

})

.ConfigureLogging((hostingContext, logging) =>

{

logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));

logging.AddConsole();

logging.AddDebug();

})

.UseIISIntegration()

.UseDefaultServiceProvider((context, options) =>

{

options.ValidateScopes = context.HostingEnvironment.IsDevelopment();

});

return builder;

}

As you can see above, the CreateDefaultBuilder method creates an instance of WebHostBuilder and sets up Kestrel, content root directory, IIS integration which is same as ASP.NET Core 1.x Main() method.

It also calls ConfigureAppConfiguration() to load configurations from appsettings.json files, environment variables and user secrets. The ConfigureLogging() method setup logging to console and debug window.

Thus, Program.cs in ASP.NET Core 2.x makes it easy for us to setup a web host.

# ASP.NET Core - Startup Class

Here, we will have an overview of Startup class contained in Startup.cs in the root folder of the project.

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

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

public class Program

{

public static void Main(string[] args)

{

BuildWebHost(args).Run();

}

public static IWebHost BuildWebHost(string[] args)

{

WebHost.CreateDefaultBuilder(args)

**.UseStartup<Startup>()**

.Build();

}

}

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

Open Startup class in Visual Studio by clicking on the Startup.cs in the solution explorer. The following is a default Startup class in ASP.NET Core 2.x.

startup.cs

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

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

## ConfigureServices()

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

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

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

ConfigureServices method includes IServiceCollection parameter to register services to the IoC container. Learn more about it in the next chapter.

## Configure()

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

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

The following is a default Configure method.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.Run(async (context) =>

{

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

});

}

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

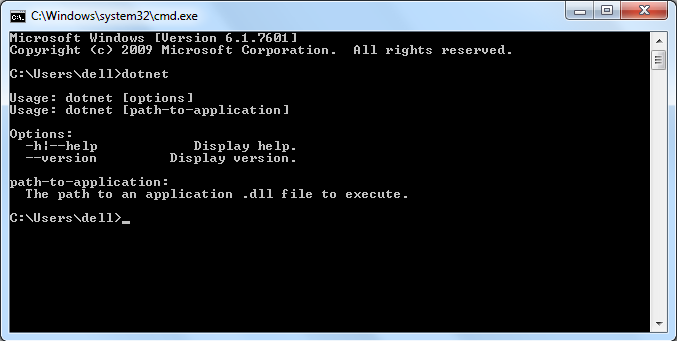
At run time, the ConfigureServices method is called before the Configure method. This is so that you can register your custom service with the IoC container which you may use in the Configure method.

# .NET Core Command-Line Interface

The .NET Core command-line interface (CLI) is a new cross-platform tool for creating, restoring packages, building, running and publishing .NET applications.

We created our first ASP.NET Core application using Visual Studio in the previous chapter. Visual Studio internally uses this CLI to restore, build and publish an application. Other higher level IDEs, editors and tools can use CLI to support .NET Core applications.

The .NET Core CLI is installed with .NET Core SDK for selected platforms. So we don't need to install it separately on the development machine. We can verify whether the CLI is installed properly by opening command prompt in Windows and writing dotnet and pressing Enter. If it displays usage and help as shown below then it means it is installed properly.

.NET Core Command-line Interface

## Command Structure

The following is a command structure.

dotnet <command> <argument> <option>

All the commands start with driver named dotnet. The driver starts the execution of the specified command. After dotnet, we can supply command (also known as verb) to perform a specific action. Each command can be followed by arguments and options. The following are .NET Core 2.x CLI commands.

| Basic Commands | Description |
| --- | --- |
| new | Creates a new project, configuration file, or solution based on the specified template. |
| restore | Restores the dependencies and tools of a project. |
| build | Builds a project and all of its dependencies. |
| Run | Runs source code without any explicit compile or launch commands. |
| publish | Packs the application and its dependencies into a folder for deployment to a hosting system. |
| test | Executes unit tests. |
| vtest | Runs tests from the specified files. |
| pack | Packs the code into a NuGet package. |
| clean | Cleans the output of a project. |
| sln | Modifies a .NET Core solution file. |
| help | Display help on the specified command |
| store | Stores the specified assemblies in the runtime package store. |

| Project Modification Commands | Description |
| --- | --- |
| add package | Adds a package reference to a project. |
| add reference | Adds project-to-project (P2P) references. |
| remove package | Removes package reference from the project. |
| remove reference | Removes project reference |
| list reference | Lists all project-to-project references |

| Advanced Commands | Description |
| --- | --- |
| nuget delete | Deletes or unlists a package from the server. |
| nuget locals | Clears or lists local NuGet resources. |
| nuget push | Pushes a package to the server and publishes it. |
| msbuild | Builds a project and all of its dependencies. |
| dotnet install script | Script used to install the .NET Core CLI tools and the shared runtime. |

Let's create, restore, build, and run .NET Core console application using command-line interface without using Visual Studio.

## Create a New Project

To create a new .NET Core project, we have to use new command followed by template name argument. We can create console, class library, web, mvc, webapi, razor, angular, react etc. projects using CLI. Use console template to create a new .NET Core console application.

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

dotnet new console

The following command creates a new console project named MyConsoleApp. The -n or --name option species the name of a project.

dotnet new console -n MyConsoleApp

The following command creates a new console application named MyConsoleApp to MyProjects directory. The -o or --output option is used to specify an output directory where the project should be generated.

dotnet new console -n MyConsoleApp -o C:\MyProjects

After creating a project, navigate to the project directories in command prompt to apply project specific commands which is C:\MyConsoleApp in our case.

## Add Package Reference

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

C:\MyConsoleApp>dotnet add package Newtonsoft.json

This will add Newtonsoft.json package to our project. We can verify it by opening .csproj file.

## Restore Packages

To restore packages or to update existing packages, we can use restore command as below.

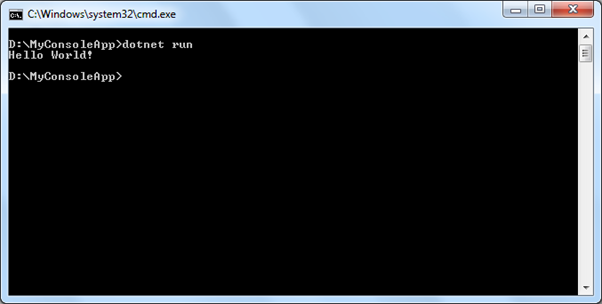
C:\MyConsoleApp>dotnet restore

## Build Project

To build a new or existing project, apply C:\MyConsoleApp>dotnet build command.

## Run project

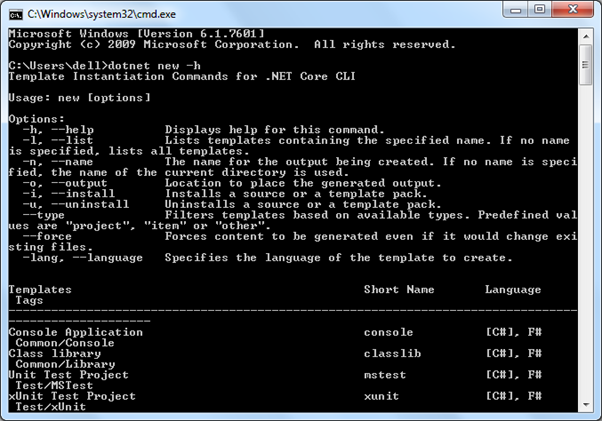
To run our console project, apply dotnet run command as shown below.



As you can see above, it displays an output "Hello World!".

## Getting Help

We can get help on any .NET Core CLI commands by typing -h or -help at the end of the command we want to get help on. For example, dotnet new -h will display help on the new command, arguments and options we can use with it, as shown below.



Thus, we can use .NET Core command-line interface to create, restore packages, build, run, and publish different types of .NET Core applications.