MCSE 541:Web Computing and Mining

ASP.NET Framework to ASP.NET Core

Prof. Dr. Shamim Akhter
Professor, CSE, Stamford University Bangladesh

OWIN and Katana @.Net Framework

- System.Web is something that has existed ever since ASP (non .NET version) and internally contains many things that you might not even need (such as Web Forms or URL Authorization), which by default all run on every request, thus consuming resources and making ASP.NET applications in general lot slower than its counterparts such as Node.js for example.
- Open Web Interface for .NET (OWIN) is a specification on how web servers and web applications should be built in order to decouple them and allow movement of ASP.NET applications to environments which were not supported before.
- Katana on the other hand, is a fully developed framework made to make a bridge between current ASP.NET frameworks and OWIN specification. At the moment, Katana has successfully adapted the following ASP.NET frameworks to OWIN:
  - Web API
  - Signal R
- ASP.NET MVC and Web Forms are still running exclusively via System.Web, and in the long run there is a plan to decouple those as well.

Katana and ASP.NET 5

- Katana is slowly getting retired. Version 3.0 was the last major release of Katana as a standalone framework.

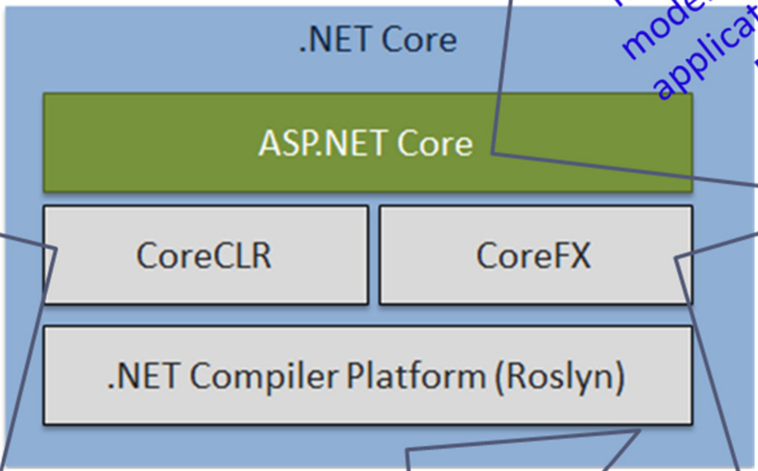
- ASP.NET 5 is the successor to Katana.
  - Katana was the beginning of the break away from System.Web and to more modular components for the web stack.
  - vNext/5 as a continuation of that work but going much further (new CLR, new Project System, new http abstractions).
  - ASP.NET 5 was built on top of .NET Core. This .NET Core was lightweight factored version of .NET Framework.

- After ASP.NET 5 ASP.NET Core 1.0 and .NET Core 1.0 were introduced.

Emergence of ASP.NET Core and .NET Core

- They re-wrote ASP.NET from the ground up and created a new cross-platform .NET runtime that later came to be .NET Core.
  - To make web development on .NET possible outside of Visual Studio and on other platforms.

.NET Core is a cross-platform and opensource implementation of the .NET Standard Library, and it is made of a few components:



Cross-platform .NET framework for building modern cloud-based web applications on Windows, Mac, or Linux.

CLR: Common Language Runtime Engine

Runtime for .NET Core. It includes the garbage collector, JIT compiler, primitive data types and low-level classes.

Foundational class libraries for .NET Core. It includes types for collections, file systems, console, JSON, XML, async and many others.

Provides C# and Visual Basic languages with rich code analysis

ASP.NET Core

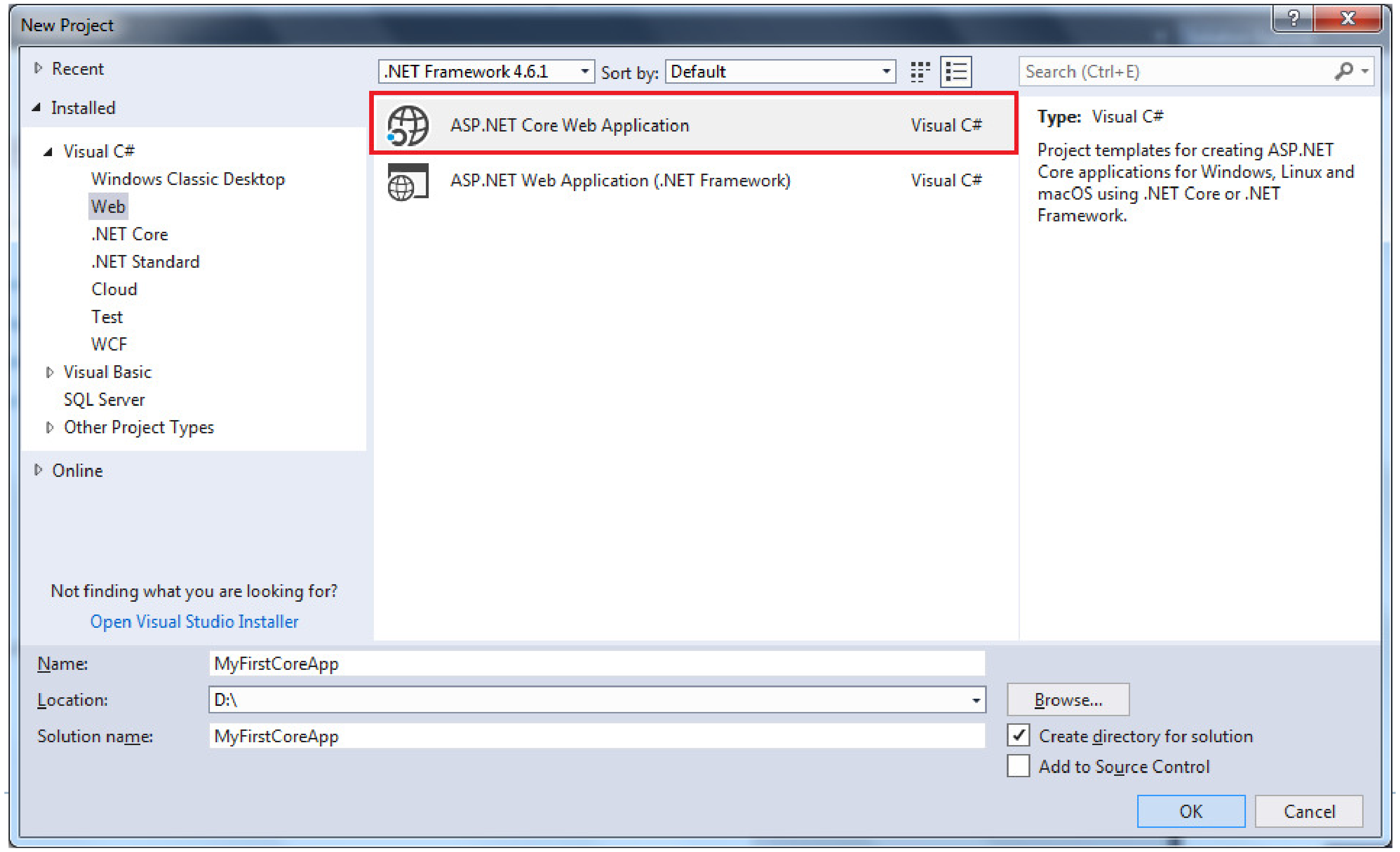
- ASP.NET Core is a newer version of ASP.NET 4.x Framework
- High Performance, Cross Platform and open-source web framework
  - run able on Windows, Mac, Linux, Android, or iOS.
  - hosted on Internet Information Services (IIS), Apache, Docker –PaaS
  - Unified programming model for MVC controller and Web API
    - Inherits from same controller classes
    - Returns IActionResult(View Result, JSON Result)

  - Service available via dependency injection
    - configure and register a service, any controller can use it.
  - Modular framework distributed as NuGet packages
    - allows us to include packages that are required in our application.

https://www.infoworld.com/article/3232636/how-to-use-dependency-injection-in-aspnet-core.html

First ASP.NET Core 2.0 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.


New Project

Recent
Installed
Visual C#
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
SQL Server
Other Project Types
Online

Not finding what you are looking for?
Open Visual Studio Installer

.NET Framework 4.6.1 | Sort by: Default | Search (Ctrl+E)

ASP.NET Core Web Application	Visual C#
ASP.NET Web Application (.NET Framework)	Visual C#

Type: Visual C#

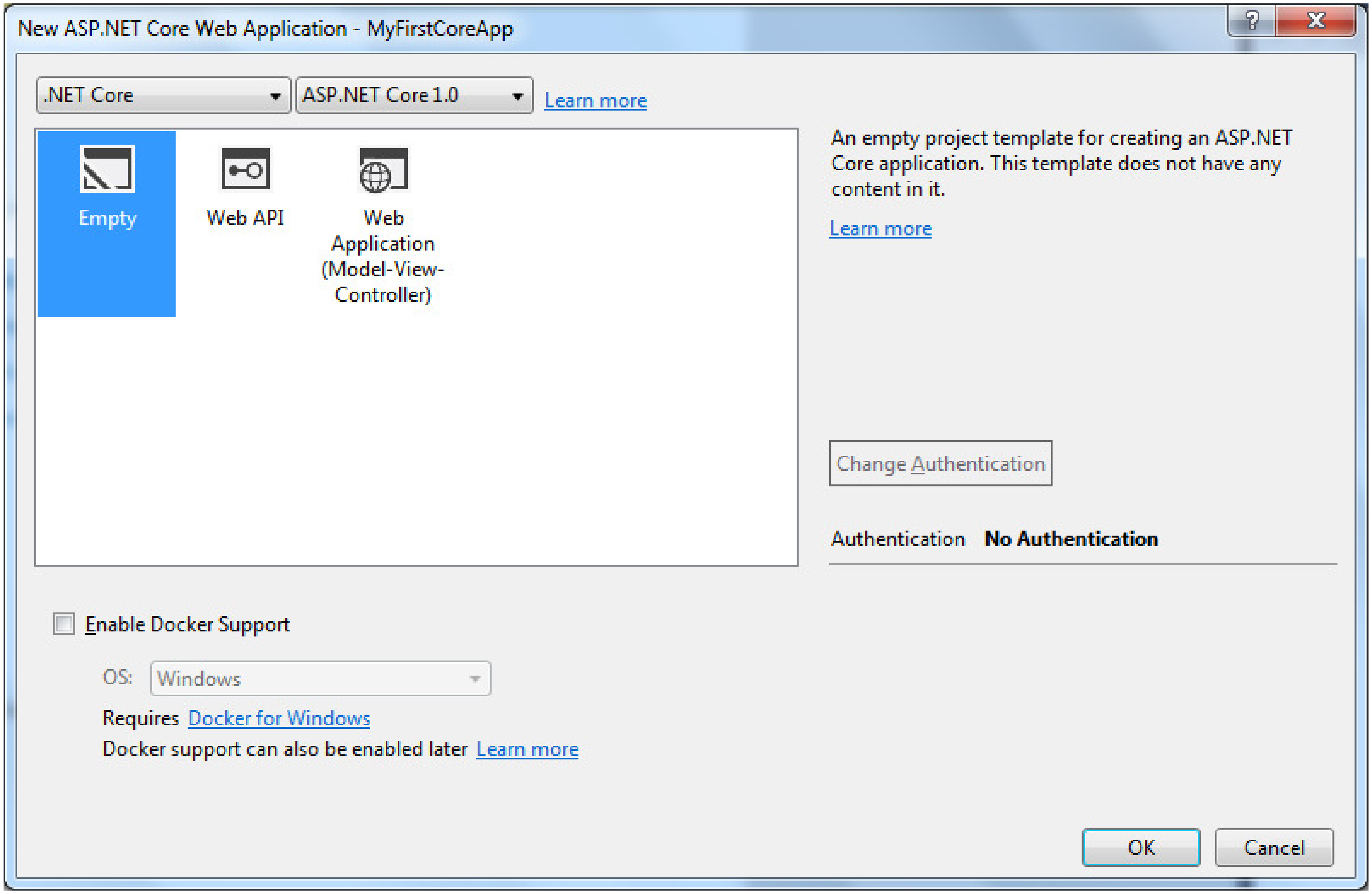
Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework.

Name: MyFirstCoreApp
Location: D:\
Solution name: MyFirstCoreApp

Browse...
[✓] Create directory for solution
[ ] Add to Source Control

OK | Cancel

First ASP.NET Core 2.0 Application



New ASP.NET Core Web Application - MyFirstCoreApp

.NET Core | ASP.NET Core 1.0 | Learn more

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

Learn more

Authentication **No Authentication**

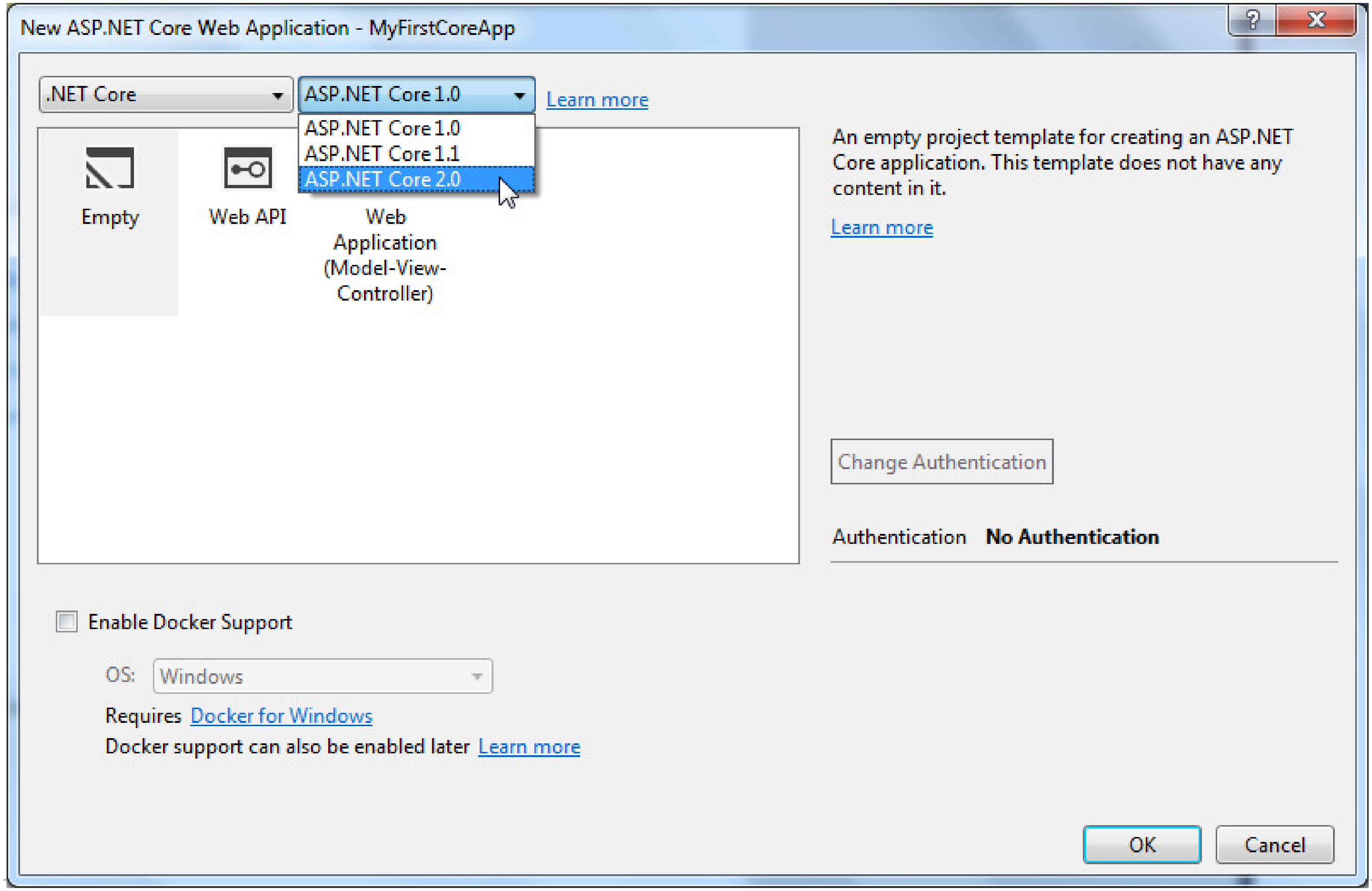
[ ] Enable Docker Support

OS: Windows

Requires Docker for Windows

Docker support can also be enabled later Learn more

First ASP.NET Core 2.0 Application



New ASP.NET Core Web Application - MyFirstCoreApp

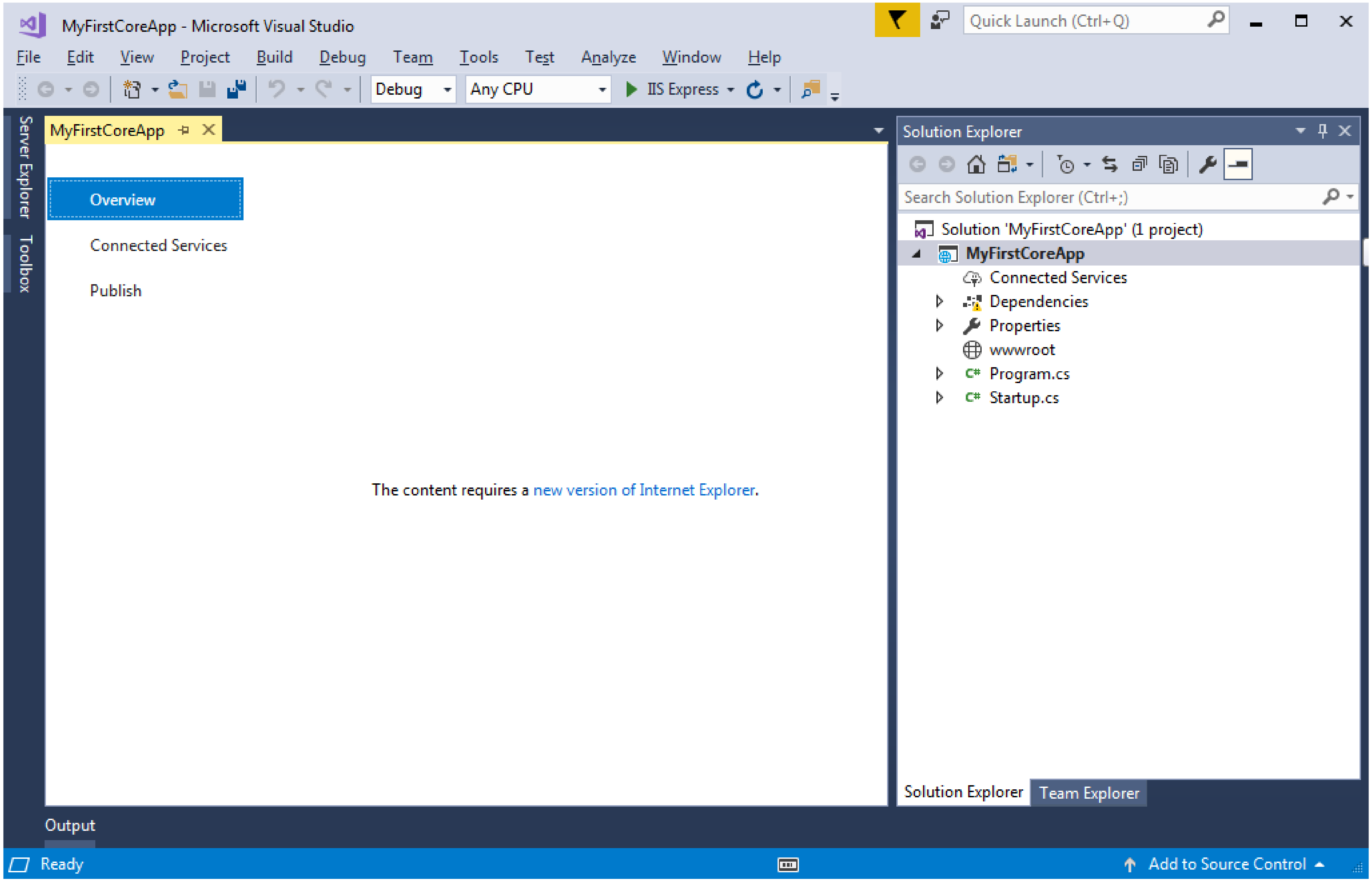
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

Authentication **No Authentication**

Requires Docker for Windows

Docker support can also be enabled later Learn more

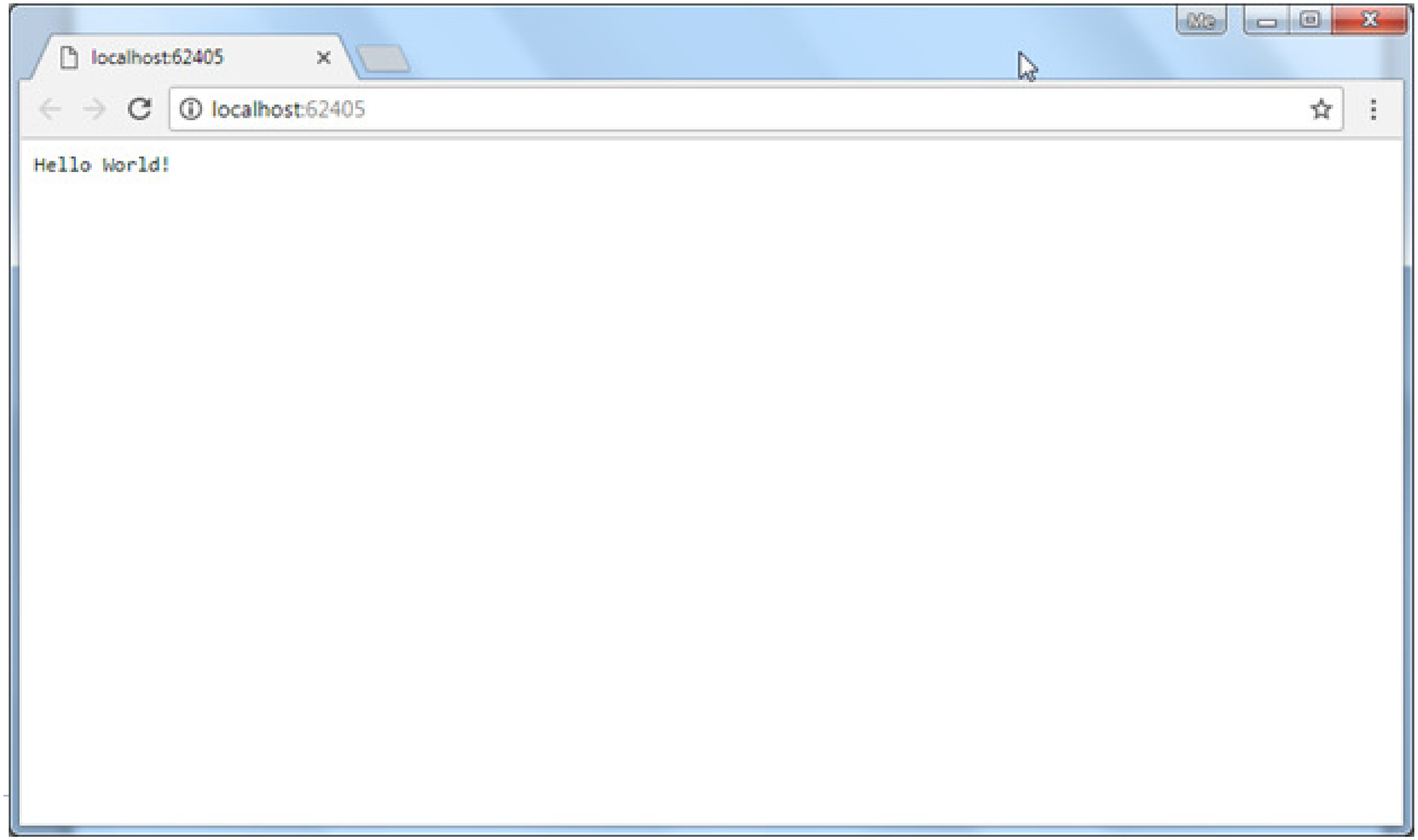
First ASP.NET Core 2.0 Application



The content requires a new version of Internet Explorer.

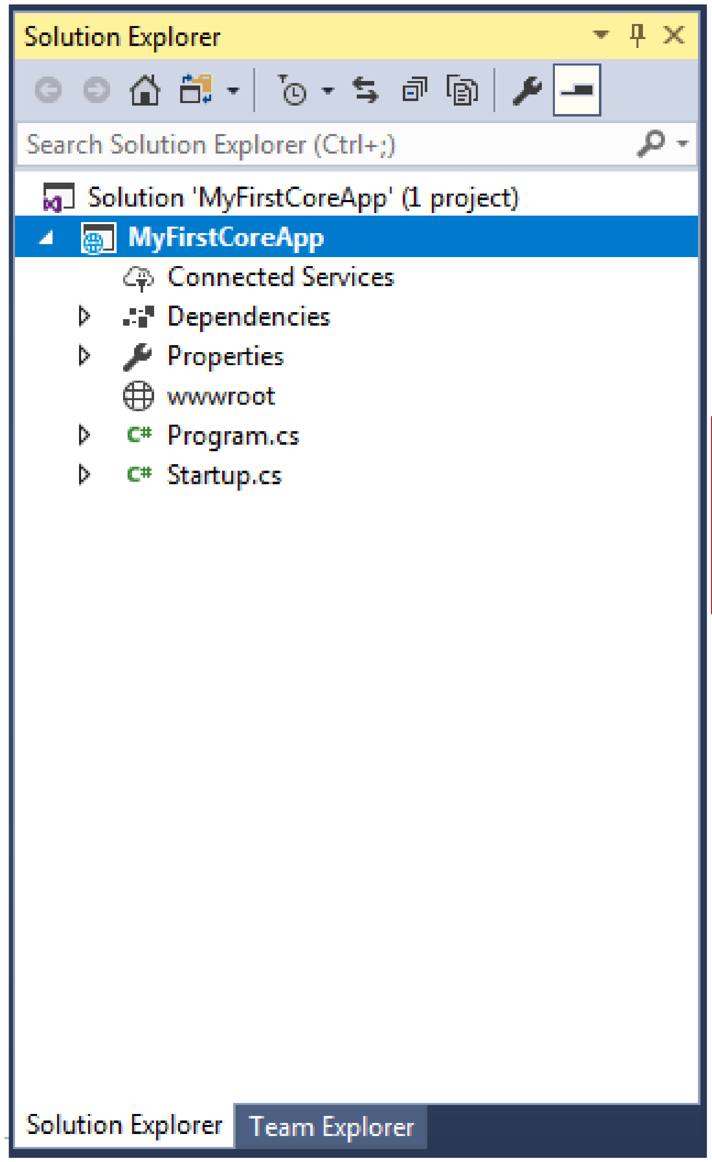
First ASP.NET Core 2.0 Application

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.



```
Hello World!
```

A Default Project Structure

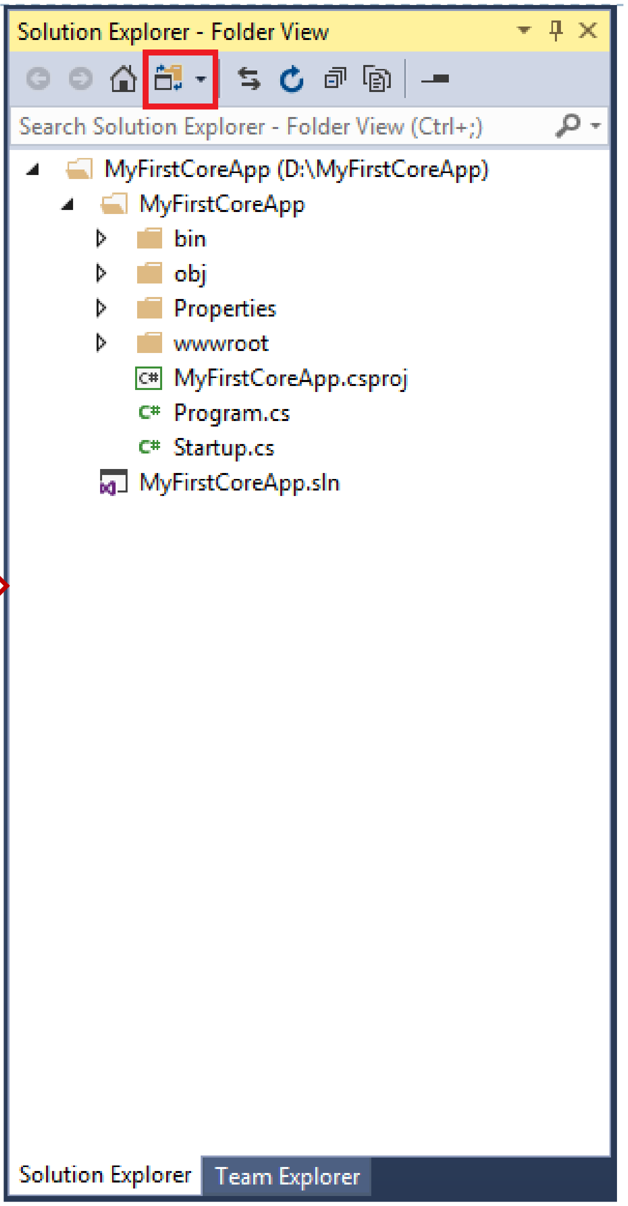


Solution Explorer

Search Solution Explorer (Ctrl+;)

Solution 'MyFirstCoreApp' (1 project)
- MyFirstCoreApp
  - Connected Services
  - Dependencies
  - Properties
  - wwwroot
  - Program.cs
  - Startup.cs

Folder View option



Solution Explorer - Folder View

Search Solution Explorer - Folder View (Ctrl+;)

- MyFirstCoreApp (D:\MyFirstCoreApp)
  - MyFirstCoreApp
    - bin
    - obj
    - Properties
    - wwwroot
    - MyFirstCoreApp.csproj
    - Program.cs
    - Startup.cs
  - MyFirstCoreApp.sln

Solution View

Starting from the top, the first new element is the Connected Services node, which contains the list of extensions that connect to a third party remote service.

The next element is a node called Dependencies. This contains all the dependencies the application has, which can be .NET packages (via NuGet), Bower, as shown in Figure 1-4, or NPM if you application needs it.

Bower is a "package manager for the web." Bower lets you install and restore client-side packages, including JavaScript and CSS libraries.

A reference to Bower appears also later in the tree with the file `bower.json`, which contains the actual configuration of all the dependencies. These dependencies, once downloaded, will be stored in the `lib` folder inside the new `wwwroot` folder.

The next element is the `wwwroot` folder, which is even represented with a different "globe" icon. This is where all the static files of the application, CSS styles, images and JavaScript files, will be.

These files in the root of the project are also new additions:

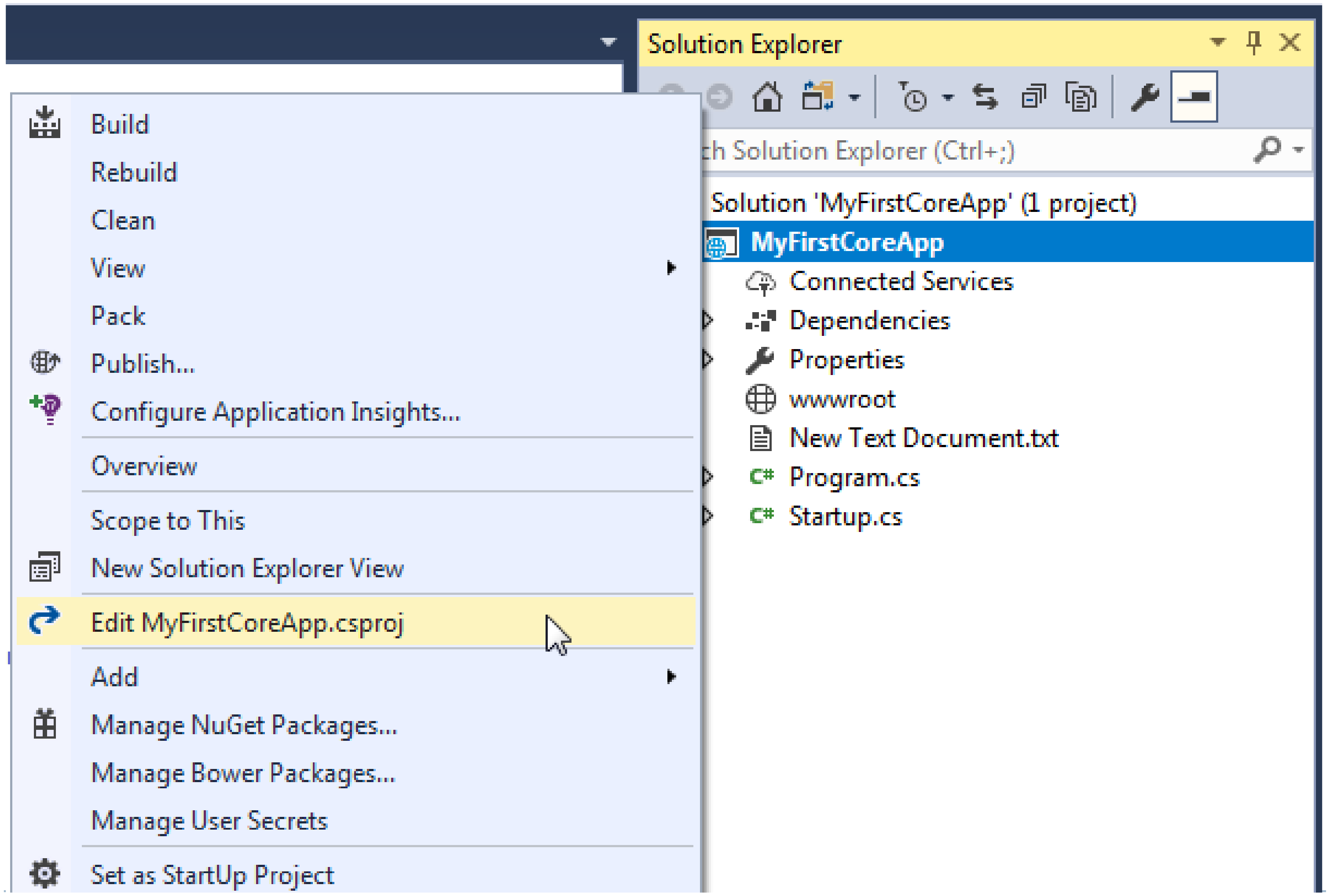
- `appsettings.json` is the new location for storing application settings instead of storing them in the `appsetting` element in the `web.config`.
- `bower.json` is the configuration file for Bower dependencies.
- `bundleconfig.json` defines the configuration for bundling and minifying JavaScript and CSS files.

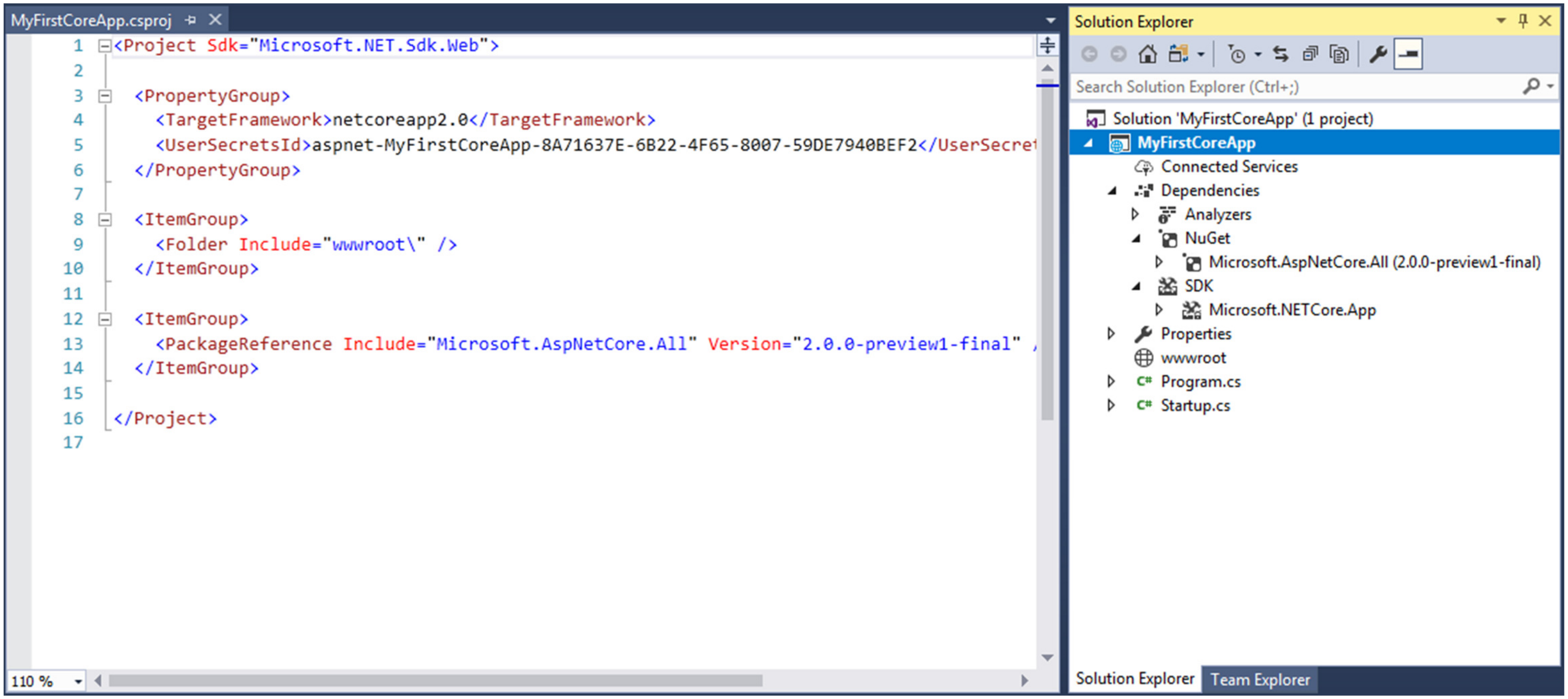
- `Program.cs` is where the web application starts. As mentioned earlier, the .NET Core app host can only start console applications, so web projects also need an instance of `Program.cs`.
- `Startup.cs` is the main entry point for ASP.NET Core web applications. It is used to configure how the application behaves. Thus the `Global.asax` file, which was used for this purpose before, has disappeared.
- `web.config` disappeared as it's not needed any more.

Folder View

.csproj

- ASP.NET Core 1.0 uses .xproj and project.json files to manage the project.
- ASP.NET Core 2.0.& Core 3.1 use .csproj





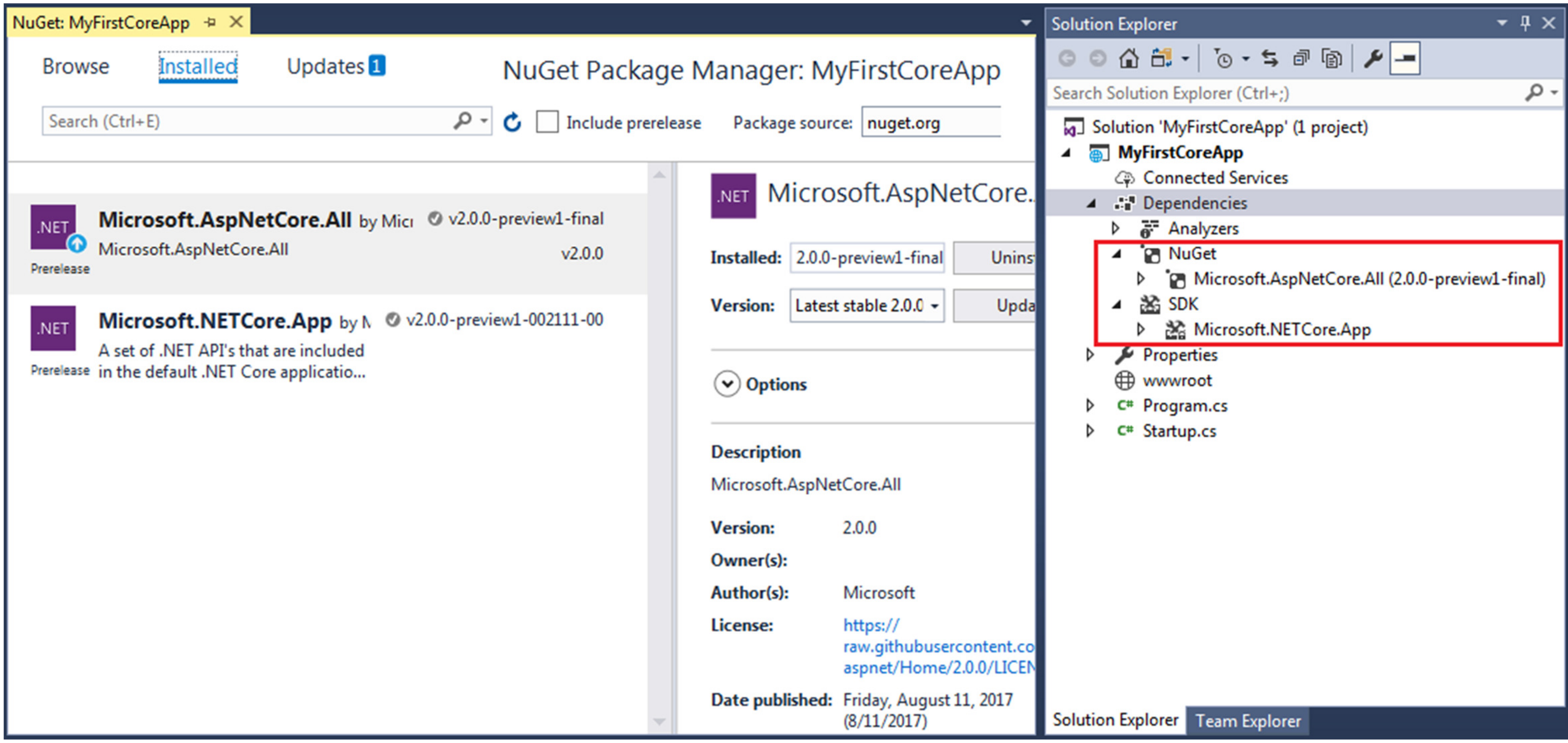
```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <UserSecretsId>aspnet-MyFirstCoreApp-8A71637E-6B22-4F65-8007-59DE7940BEF2</UserSecre
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0-preview1-final" /
  </ItemGroup>

</Project>
```

There is a SecretManager tool, provided by the .NET Core SDK (Microsoft.Extensions.SecretManager.Tools), which you can access with the dotnet CLI.

- %APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json

Dependencies

- Dependencies in the ASP.NET Core 2.0 project contains 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.



Microsoft.AspNetCore.All by Micı v2.0.0-preview1-final
Microsoft.AspNetCore.All v2.0.0
Prerelease

Microsoft.NETCore.App by M v2.0.0-preview1-002111-00
A set of .NET API's that are included in the default .NET Core applicatio...
Prerelease

Microsoft.AspNetCore.

Installed: 2.0.0-preview1-final
Version: Latest stable 2.0.0

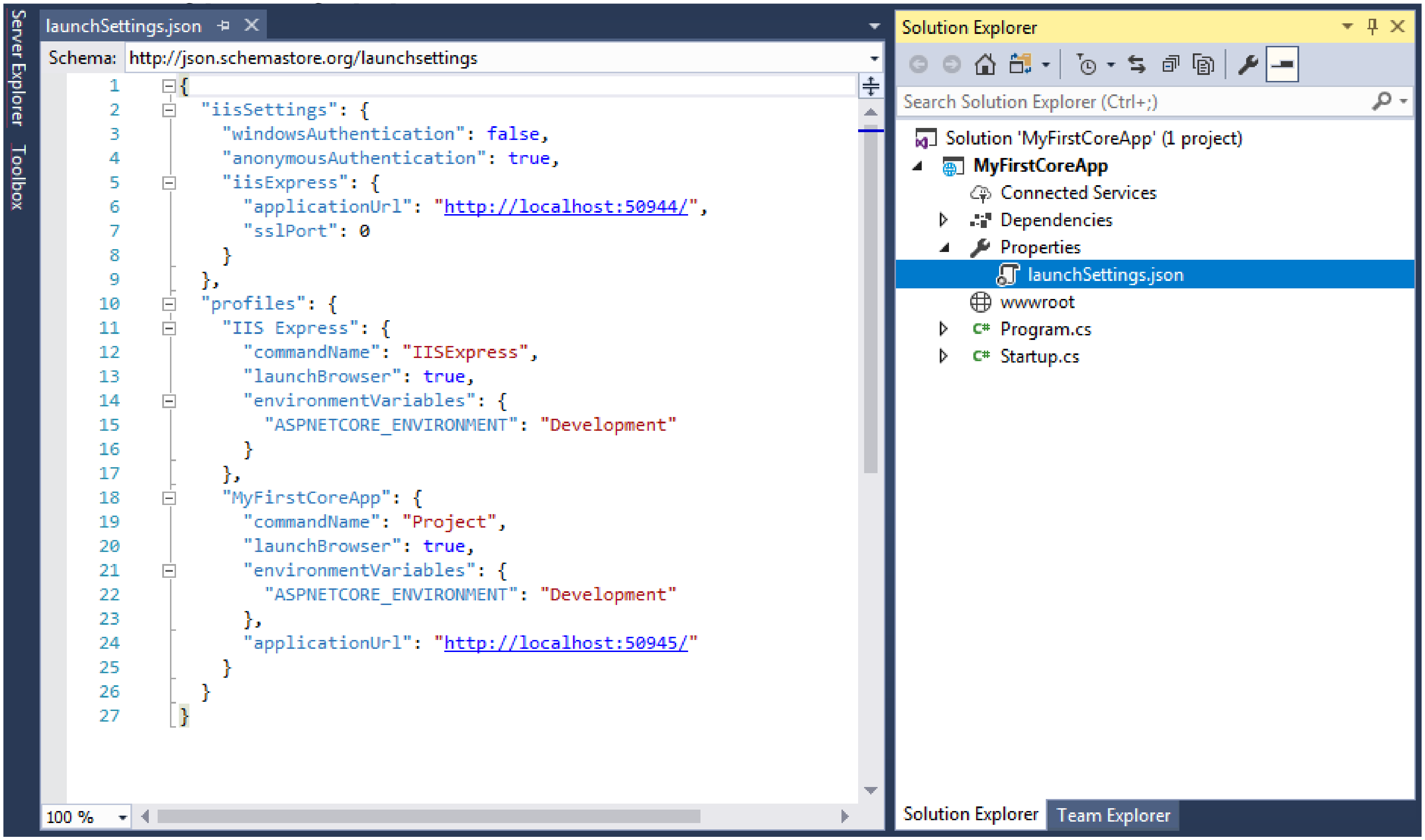
Description
Microsoft.AspNetCore.All

Version: 2.0.0
Owner(s):
Author(s): Microsoft
License: https://raw.githubusercontent.co aspnet/Home/2.0.0/LICEN
Date published: Friday, August 11, 2017 (8/11/2017)

Solution 'MyFirstCoreApp' (1 project)
- MyFirstCoreApp
  - Connected Services
  - Dependencies
    - Analyzers
    - NuGet
      - Microsoft.AspNetCore.All (2.0.0-preview1-final)
    - SDK
      - Microsoft.NETCore.App
  - Properties
  - wwwroot
  - Program.cs
  - Startup.cs

Properties

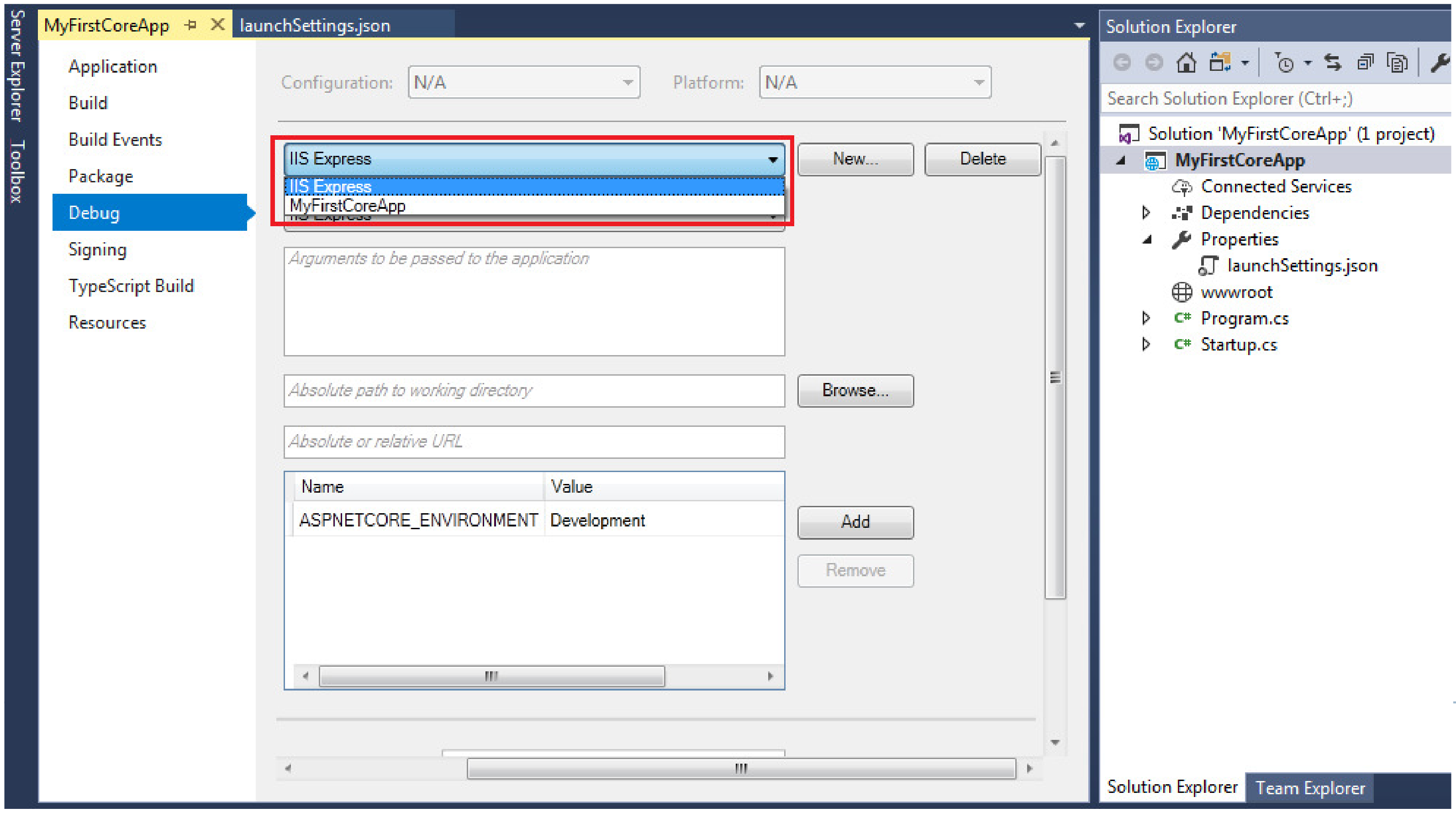
- launchSettings.json file which includes Visual Studio



```json
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:50944/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "MyFirstCoreApp": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:50945/"
    }
  }
}
```

Properties

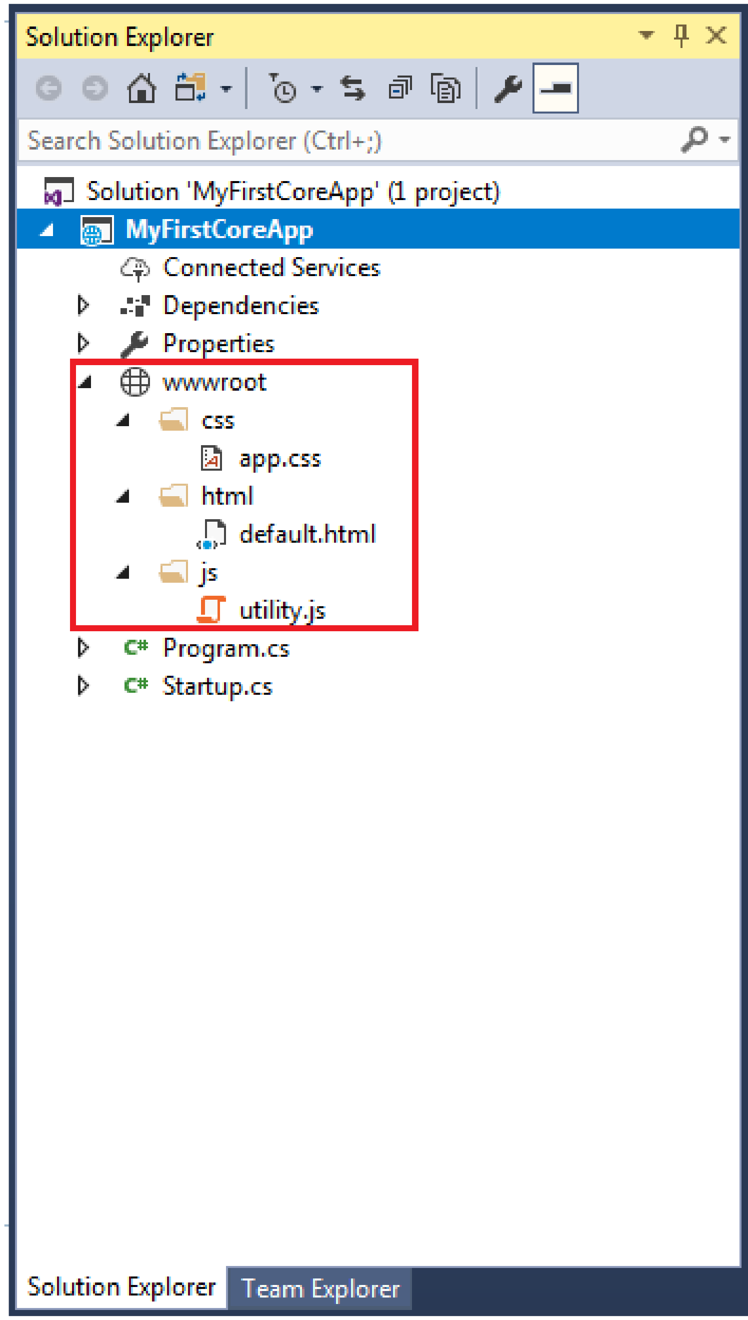
- Editing Project Properties
  - Right click on the project -> select Properties -> click Debug tab.
  - In the debug tab, select a profile which you want to edit as shown above. You may change environment variables, url etc.



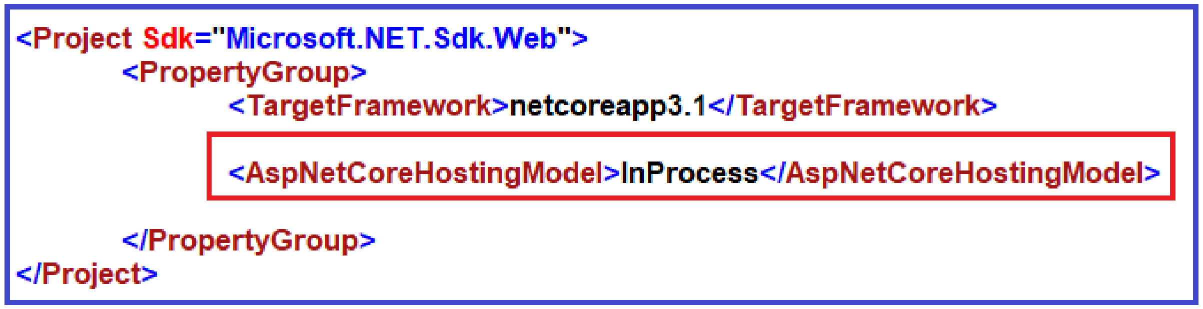
Name	Value
ASPNETCORE_ENVIRONMENT	Development

ASP.NET Core - wwwroot Folder

- Treated as a web root folder.
- Static files can be stored in any folder under the web root
- ASP.NET Core supports 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.
- For example, we can access above site.css file in the css folder by *http://localhost:<port>/css/app.css*.
- You will need to include a middleware for serving static files in the Configure method of Startup.cs.
- 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.

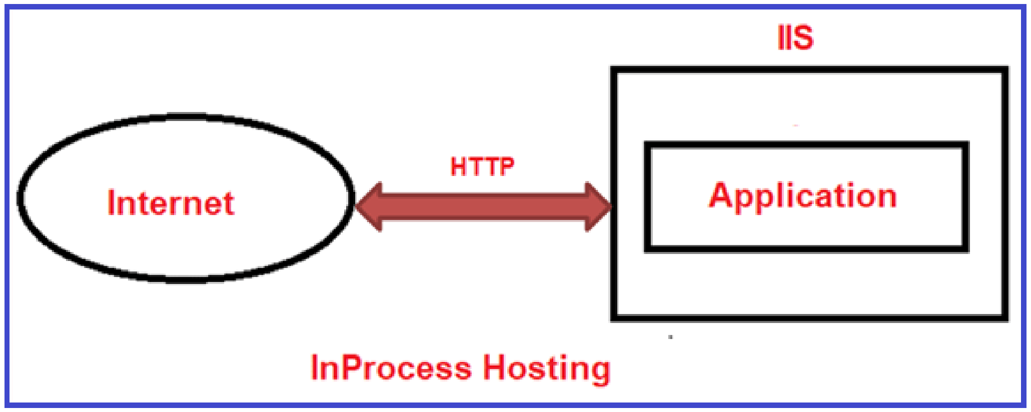


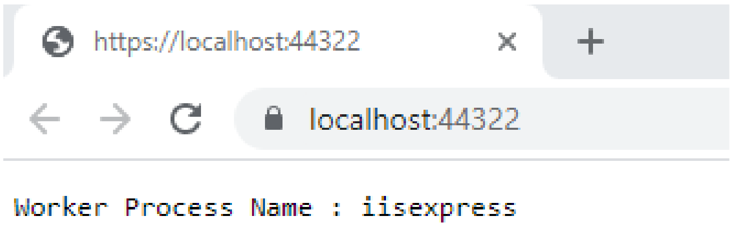
Hosting Model: InProcess



```xml
<Project Sdk="Microsoft.NET.Sdk.Web">
        <PropertyGroup>
                <TargetFramework>netcoreapp3.1</TargetFramework>

                <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>

        </PropertyGroup>
</Project>
```

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.





```
Worker Process Name : iisexpress
```

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

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

IIS settings modify launchsettings

```
"IIS": {
      "commandName": "IIS",
      "launchBrowser": true,
      "launchUrl": "http://localhost/WebApplication8",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "ancmHostingModel": "OutOfProcess"
    }
```

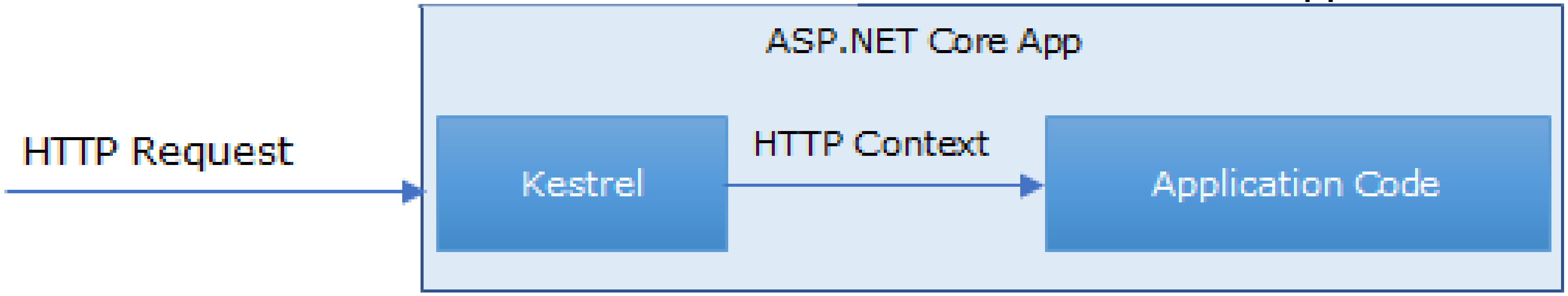
```
"IIS": {
      "commandName": "IIS",
      "launchBrowser": true,
      "launchUrl": "http://localhost/WebApplication8",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "ancmHostingModel": "InProcess"
    }
```

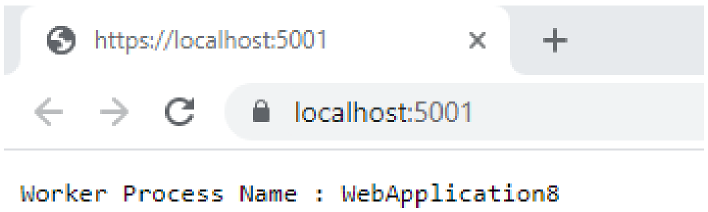
Hosting Model: Out of Process

Kestrel as an Edge Server

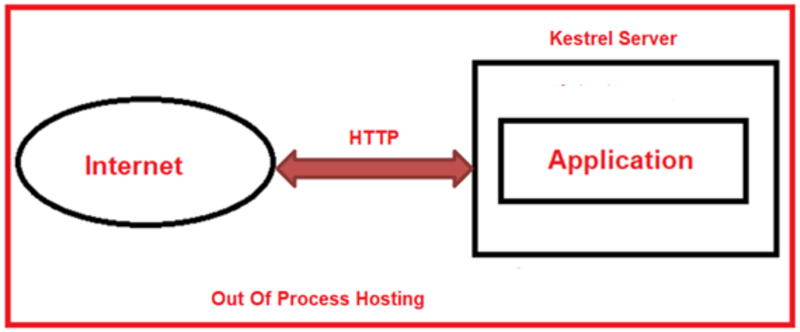
ASP.NET Core is a cross-platform framework.

Kestrel is the cross-platform web server for the ASP.NET Core application.





```
Worker Process Name : WebApplication8
```

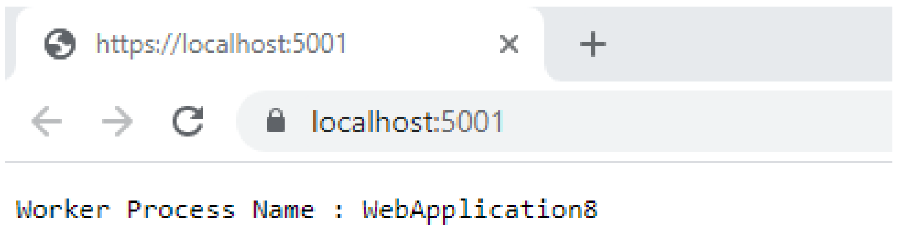


IIS/IIS Express is there But not in use

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

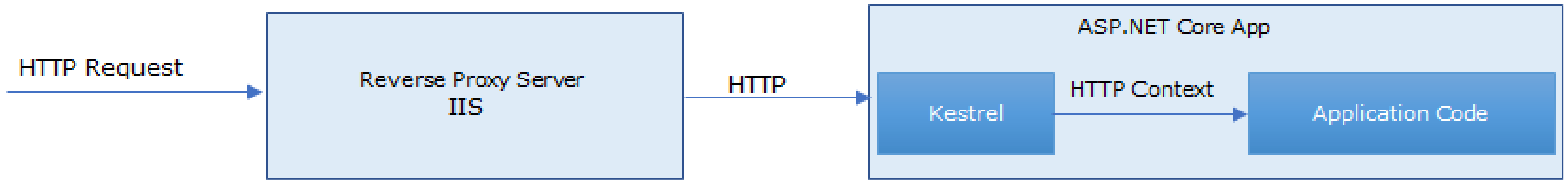
- An internal webserver which is the Kestrel web Server
- And an external web server which can be IIS, Apache, and Nginx.

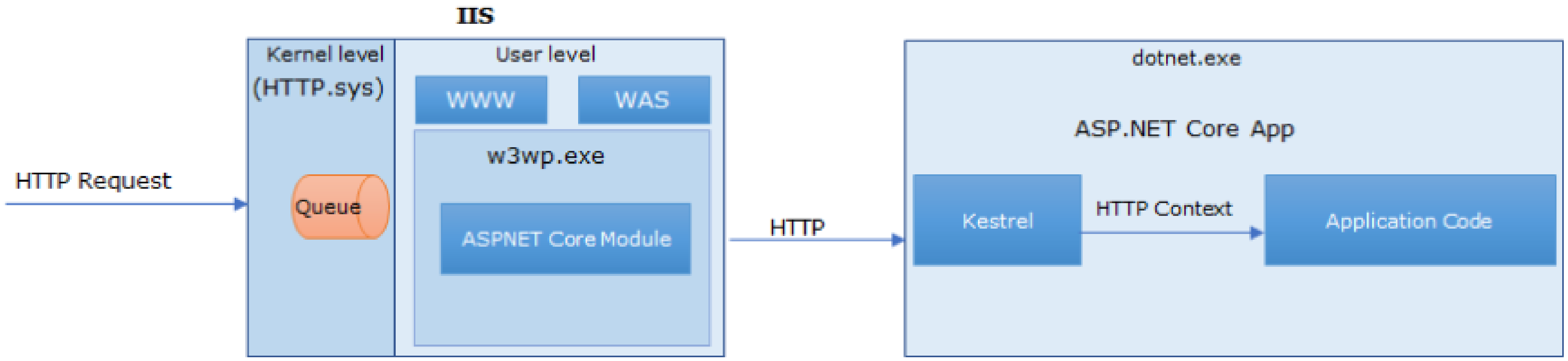
- We can use the Kestrel Web Server as the internet-facing web server which will directly process the incoming HTTP requests.
- In this scenario, only the Kestrel Server is used and the other one i.e. external web server is not going to be used.
- So, when we run the application using the **.NET core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

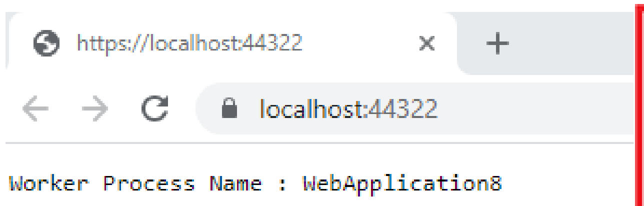


```
Worker Process Name : WebApplication8
```

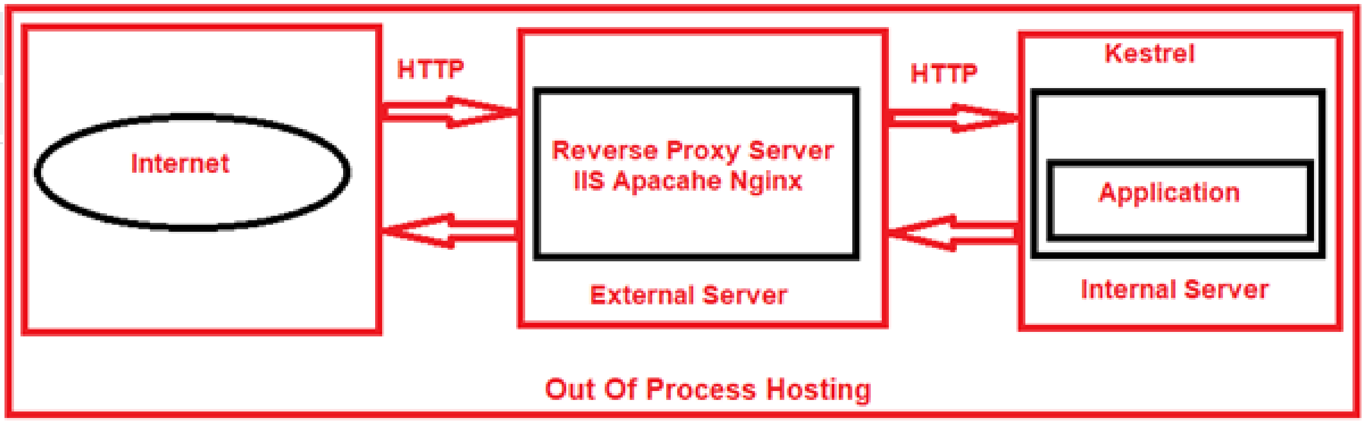
Using IIS as a Reverse Proxy:



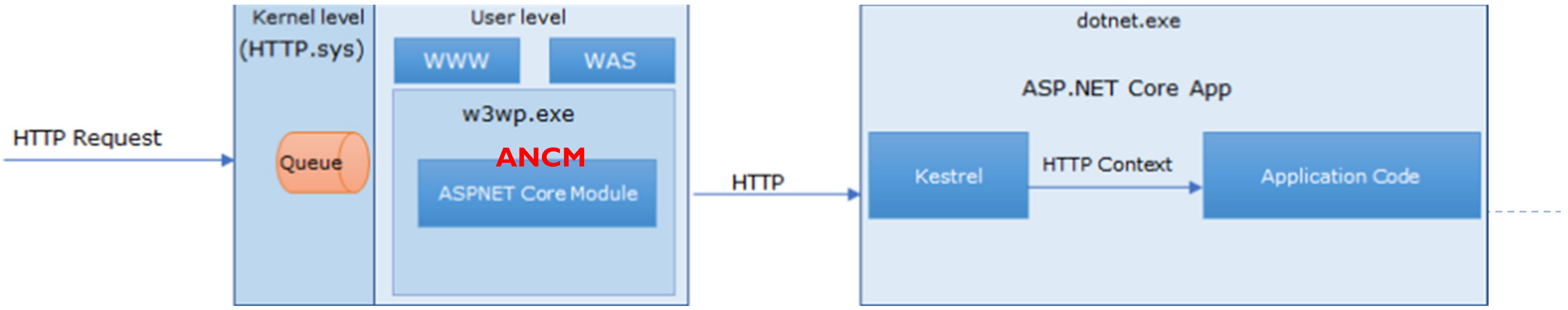




```
Worker Process Name : WebApplication8
```

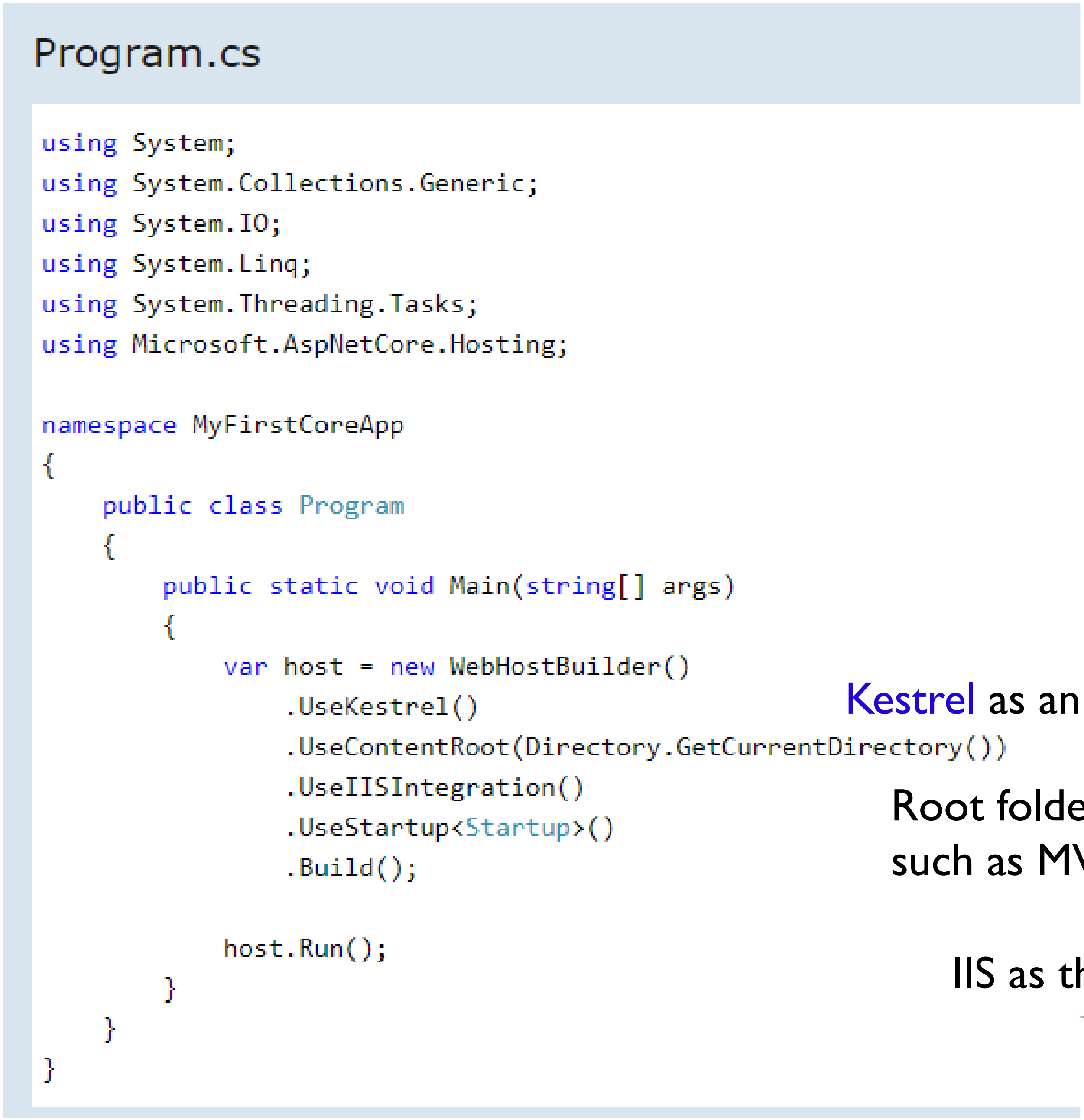


https://codewala.net/2019/01/29/hosting-asp-net-core-applications-in-iis/



1. The request is received by the *HTTP.sys* from the network.
2. If response is cached at *HTTP.sys* then it is sent back from there else gets a place the corresponding Application Pool's queue.
3. When a thread is available in the thread pool, it picks up the request and start processing it.
4. The request goes through IIS processing pipeline. As mentioned earlier the request goes through few native IIS modules and once it reaches to *ANCM*, it forwards the request to *Kestrel* (under dotnet.exe).
5. ANCM has a responsibility to manage the process as well. If (re)starts the process (if not running or crashed) and IIS integration middleware configure the server to listen the request on port defined in environment variable. It only accepts the requests which originates from ANCM.
   Note -Please do note that in ASP.NET Webforms/MVC the application is hosted under the worker process w3wp.exe which is managed by Windows Activation Service (WAS) which was part of IIS.
6. Once the request is received by Kestrel, it creates the *HTTPContext* object and request is handed over to ASP.NET Core middleware pipeline.
7. The request is passed to routing middleware which invokes the right controller and action method (model binding, various filters almost similar way as earlier versions).
8. Finally, the response is returned from the action and passed to kestrel via Middlewares and later sent back to client via IIS.

ASP.NET Core - Program.cs 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();
        }
    }
}
```

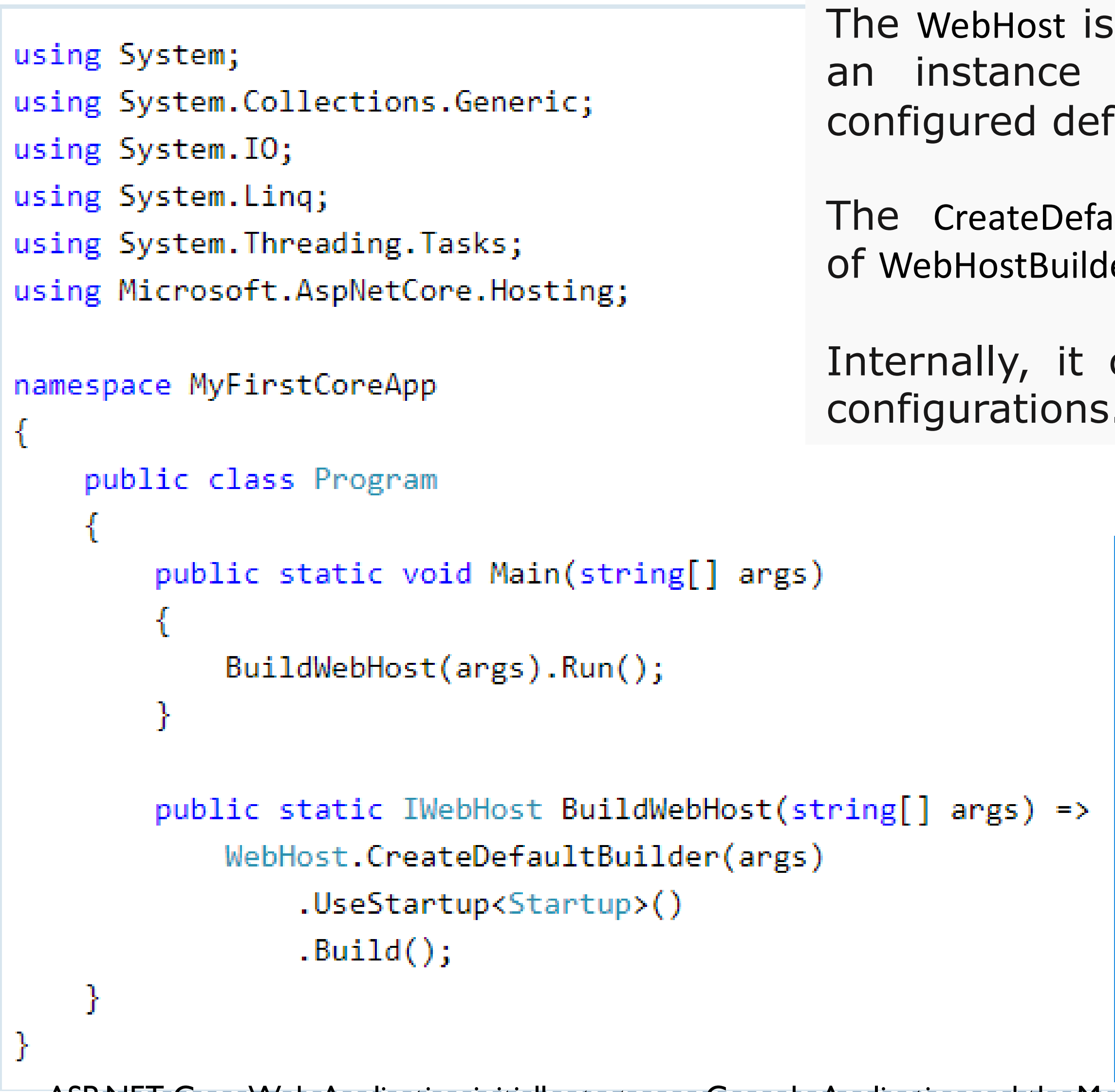
Kestrel as an internal web server.

Root folder where the content files are located such as MVC view files, CSS, images etc.

IIS as the external web server

ASP.NET Core - Program.cs 2.0

Main method is the entry point for that console application execution.

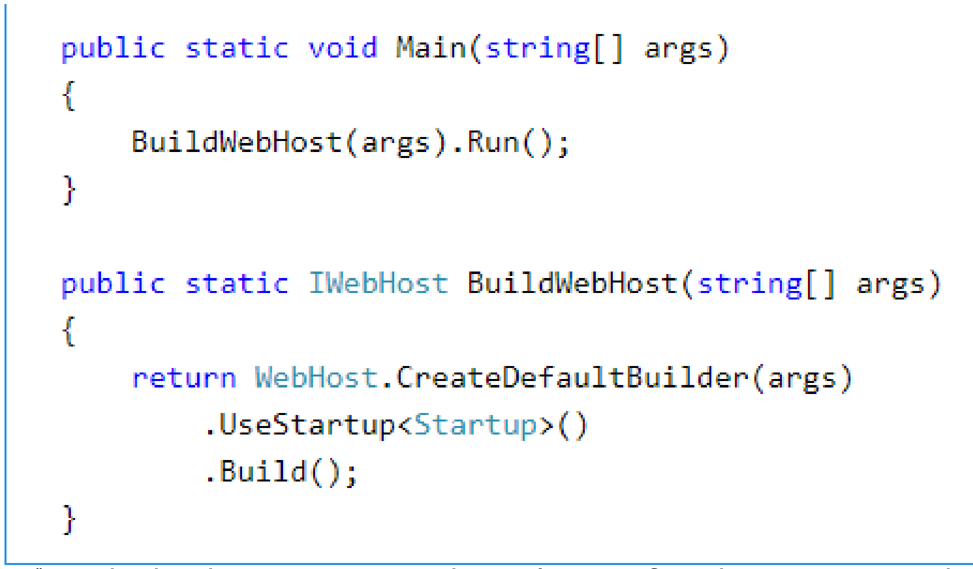


```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;

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();
    }
}
```

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.

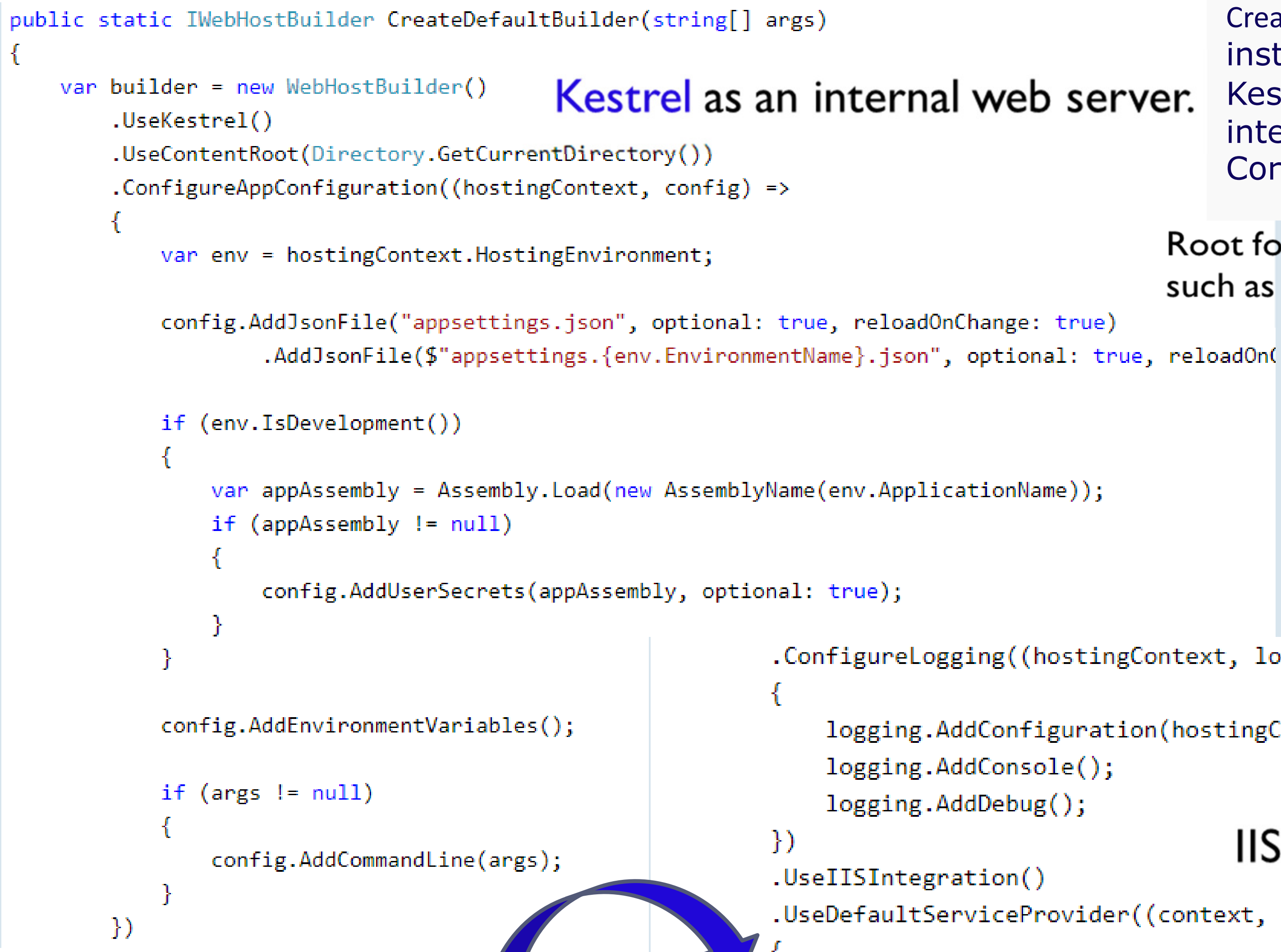
Internally, it configures Kestrel, IISIntegration and other configurations.



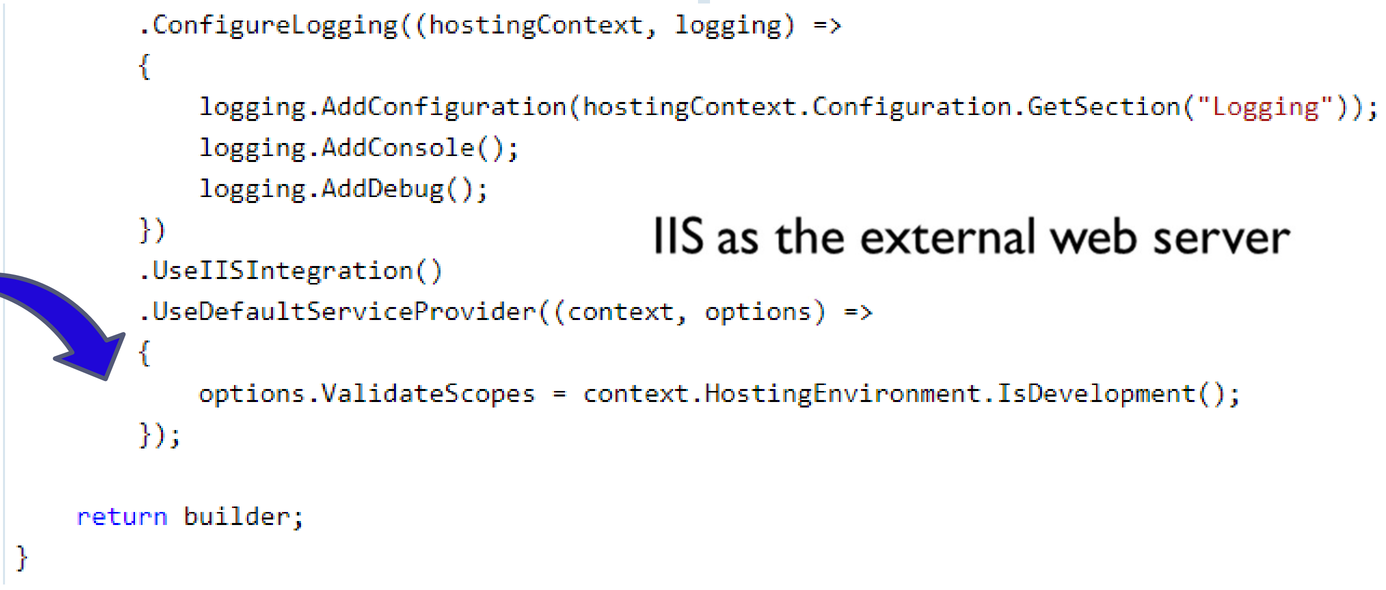
```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args)
{
    return WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
}
```

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() method and this is the method from where the execution starts. The Main() method then configures ASP.NET Core and starts it. At this point, the application becomes an ASP.NET Core web application.

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, reloadOn

            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;
}
```

Kestrel as an internal web server.

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.

Root folder where the content files are located such as MVC view files, CSS, images etc.

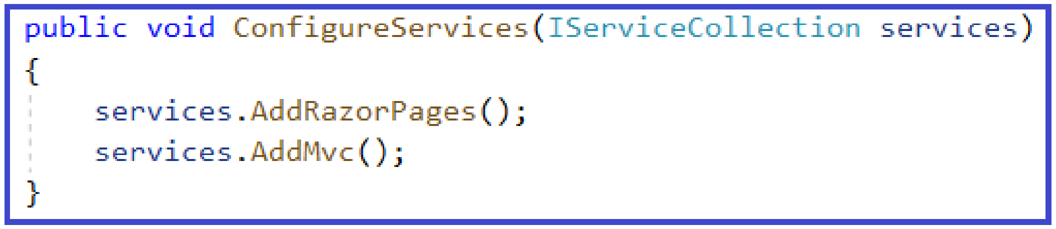
It also calls ConfigureAppConfiguration() to load configurations from appsettings.json files, environment variables and user secrets.

The ConfigureLogging() method setup logging to console and debug window.

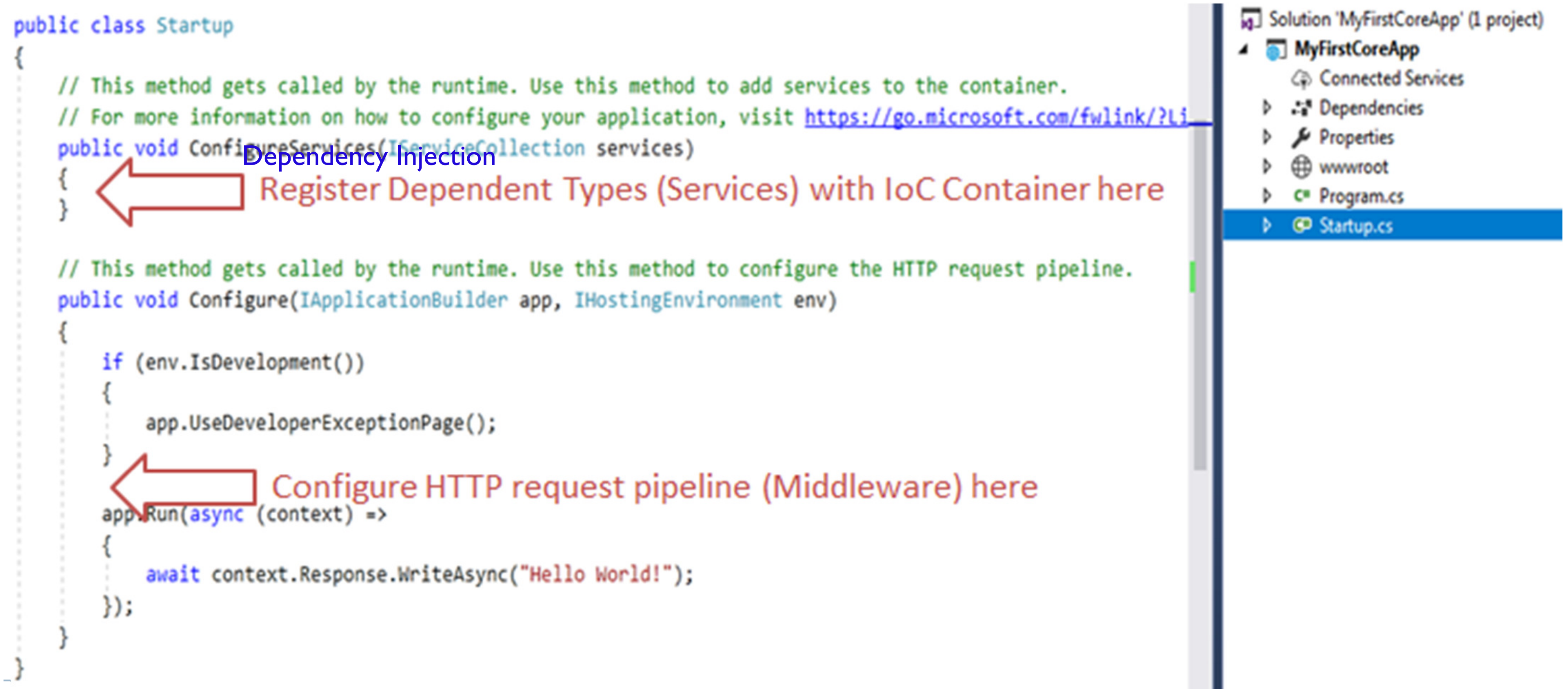
IIS as the external web server

ASP.NET Core - Startup Class

- Must include a Configure method and can optionally include ConfigureService method.



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddMvc();
}
```



```
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/?Li
    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, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    }
}
```

Dependency Injection
Register Dependent Types (Services) with IoC Container here
Configure HTTP request pipeline (Middleware) here
Solution 'MyFirstCoreApp' (1 project)
MyFirstCoreApp
Connected Services
Dependencies
Properties
wwwroot
Program.cs
Startup.cs

ConfigureServices()

- The ConfigureServices method is a place where we can register our dependent classes with the built-in IoC container.
  - Inversion of control (IoC) is a programming principle. IoC inverts the flow of control as compared to traditional control flow.
- 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.

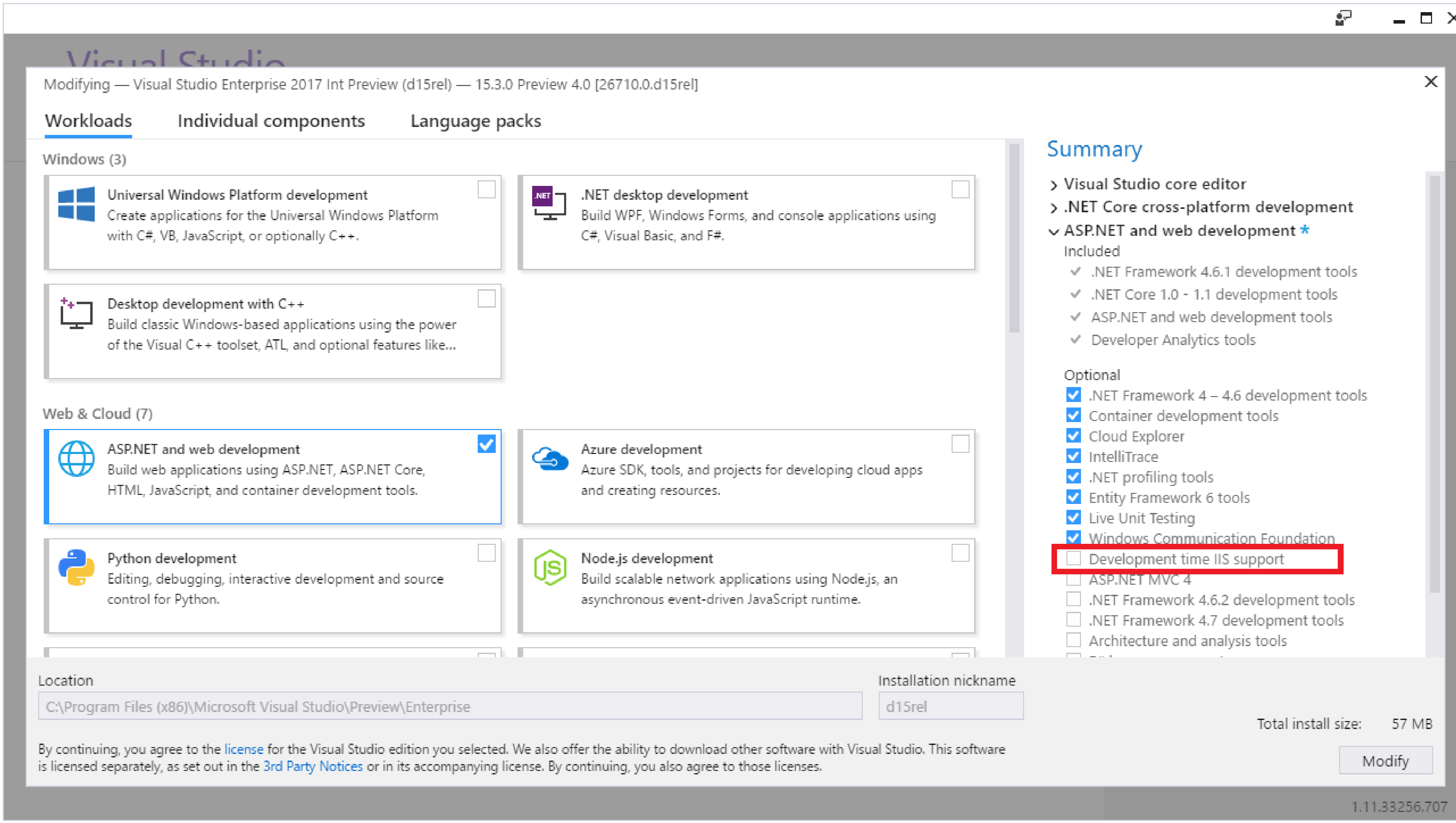
At run time, the ConfigureServices method is called before the Configure method. This is so that you can register your custom service with the IoC container which you may use in the Configure method.

Configure( )

- You can configure application request pipeline 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.

https://www.tutorialsteacher.com/core/aspnet-core-startup

- **Development time IIS support**
- Once you’ve installed IIS, you can launch the Visual Studio installer to modify your existing Visual Studio installation. In the installer select the *Development time IIS support* component which is listed as optional component under the *ASP.NET and web development* workload. This will install the ASP.NET Core Module which is a native IIS module required to run ASP.NET Core applications on IIS.


Modifying — Visual Studio Enterprise 2017 Int Preview (d15rel) — 15.3.0 Preview 4.0 [26710.0.d15rel]

Windows (3)

- [ ] Universal Windows Platform development — Create applications for the Universal Windows Platform with C#, VB, JavaScript, or optionally C++.
- [ ] .NET desktop development — Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F#.
- [ ] Desktop development with C++ — Build classic Windows-based applications using the power of the Visual C++ toolset, ATL, and optional features like...

Web & Cloud (7)

- [✓] ASP.NET and web development — Build web applications using ASP.NET, ASP.NET Core, HTML, JavaScript, and container development tools.
- [ ] Azure development — Azure SDK, tools, and projects for developing cloud apps and creating resources.
- [ ] Python development — Editing, debugging, interactive development and source control for Python.
- [ ] Node.js development — Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Summary

- Visual Studio core editor
- .NET Core cross-platform development
- ASP.NET and web development *

Included
- ✓ .NET Framework 4.6.1 development tools
- ✓ .NET Core 1.0 - 1.1 development tools
- ✓ ASP.NET and web development tools
- ✓ Developer Analytics tools

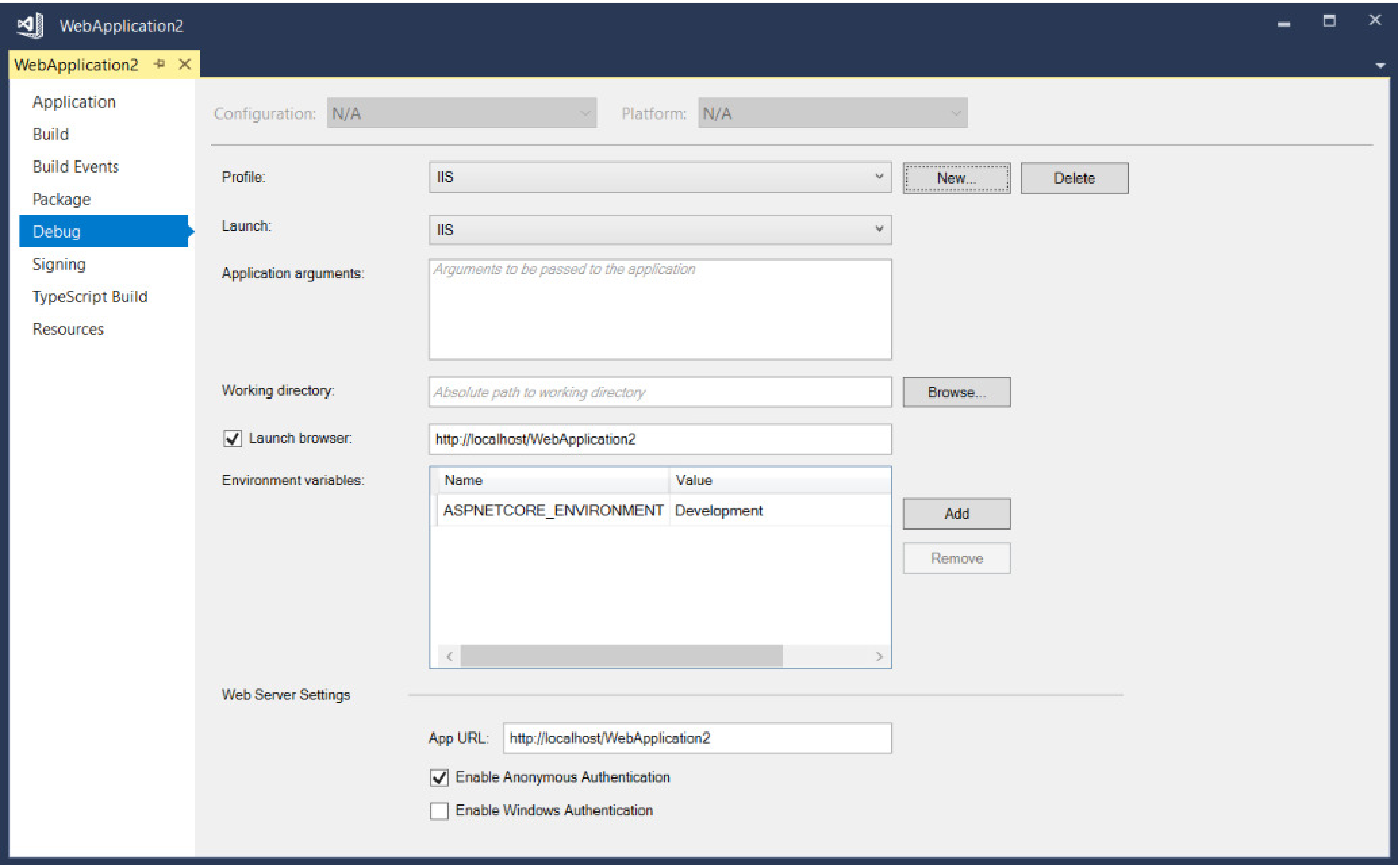
Optional
- [✓] .NET Framework 4 – 4.6 development tools
- [✓] Container development tools
- [✓] Cloud Explorer
- [✓] IntelliTrace
- [✓] .NET profiling tools
- [✓] Entity Framework 6 tools
- [✓] Live Unit Testing
- [✓] Windows Communication Foundation
- [ ] Development time IIS support
- [ ] ASP.NET MVC 4
- [ ] .NET Framework 4.6.2 development tools
- [ ] .NET Framework 4.7 development tools
- [ ] Architecture and analysis tools

Location: C:\Program Files (x86)\Microsoft Visual Studio\Preview\Enterprise

Installation nickname: d15rel

Total install size: 57 MB

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.



Configuration: N/A Platform: N/A

Profile: IIS

Launch: IIS

Application arguments: *Arguments to be passed to the application*

Working directory: *Absolute path to working directory*

[✓] Launch browser: http://localhost/WebApplication2

Environment variables:

Name	Value
ASPNETCORE_ENVIRONMENT	Development

Web Server Settings

App URL: http://localhost/WebApplication2

[✓] Enable Anonymous Authentication

[ ] Enable Windows Authentication

EndPoint Routing(new in 3.1)

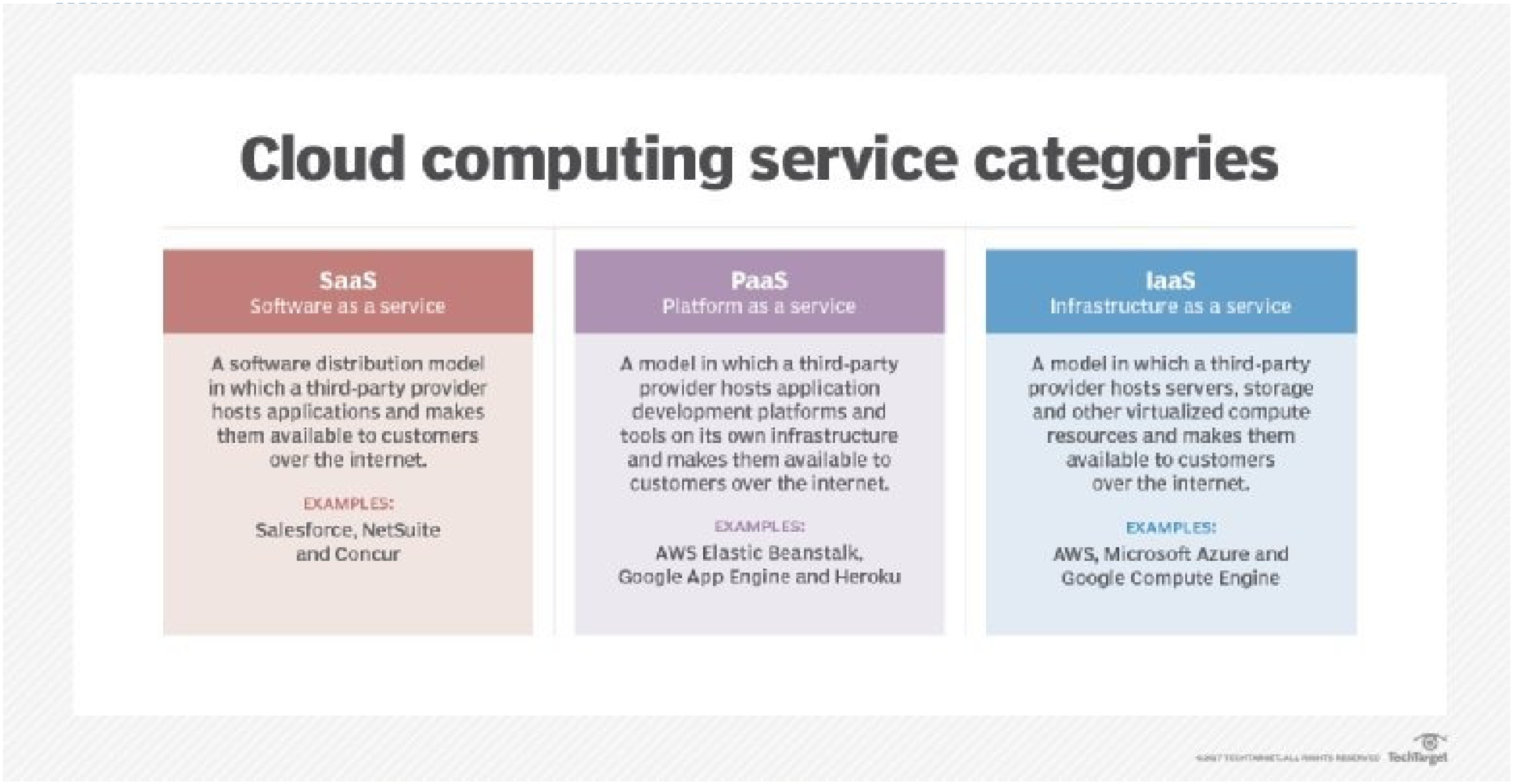
- EndPoint Routing is the new way to implement the Routing in ASP.NET Core.
- It splits up the old routing middleware into two separate middleware's and also decouples the MVC from the Routing Middleware.
- Learn what is Endpoint is and how to register these routing middleware's using the UseRouting & UseEndpoints methods in the Configure method of the startup class.

What is Endpoint Routing in ASP.NET Core

- An Endpoint is an object that contains everything that you need to execute the incoming Request. The Endpoint object contains the following information

- Metadata of the request.
- The delegate (Request handler) that ASP.NET core uses to process the request.
- We define the Endpoint at the application startup using the UseEndpoints method.

https://www.tektutorialshub.com/asp-net-core/asp-net-core-endpoint-routing/



Cloud computing service categories

SaaS Software as a service	PaaS Platform as a service	IaaS Infrastructure as a service
A software distribution model in which a third-party provider hosts applications and makes them available to customers over the internet.	A model in which a third-party provider hosts application development platforms and tools on its own infrastructure and makes them available to customers over the internet.	A model in which a third-party provider hosts servers, storage and other virtualized compute resources and makes them available to customers over the internet.
EXAMPLES: Salesforce, NetSuite and Concur	EXAMPLES: AWS Elastic Beanstalk, Google App Engine and Heroku	EXAMPLES: AWS, Microsoft Azure and Google Compute Engine