

How to Deploy ASP.NET Core to IIS & How ASP.NET Core Hosting Works

JAMES MICHAELIS JANUARY 14, 2022

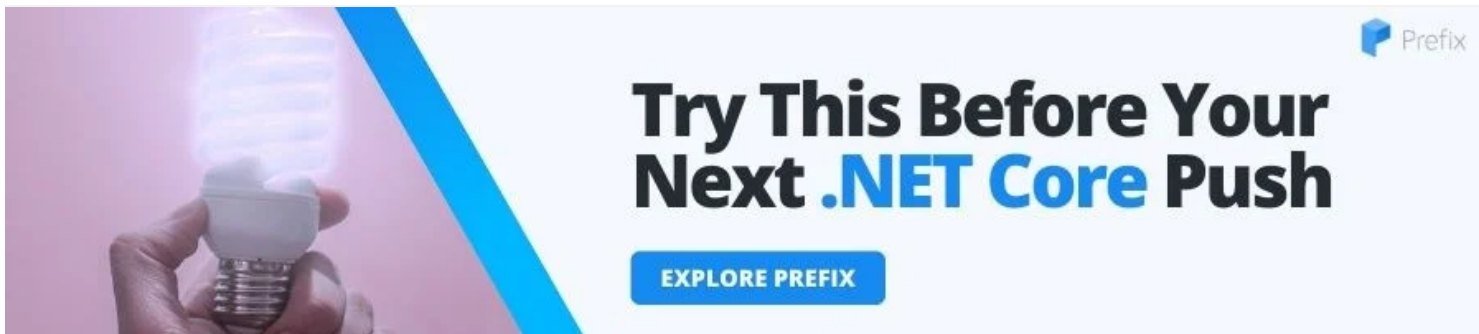
[DEVELOPER TIPS, TRICKS & RESOURCES](#), [LIVE QUEUE](#)

Previously, we discussed the differences between [Kestrel vs IIS](#). In this article, we will review how to deploy an ASP.NET Core application to IIS.

Deploying an ASP.NET Core app to IIS isn't complicated. However, ASP.NET Core hosting is different compared to hosting with ASP.NET, because ASP.NET Core uses different configurations. You may read more about ASP.NET Core in this [entry](#).

On the other hand, IIS is a web server that runs on the ASP.NET platform within the Windows OS. The purpose of IIS, in this context, is to host applications built on ASP.NET Core. There's more information on IIS and ASP.NET in our previous blog, "[What is IIS?](#)"

In this entry, we'll explore how to make both ASP.NET Core and IIS work together. Without further ado, let's explore the steps on how we can deploy ASP.NET Core to IIS.



How to Configure Your ASP.NET Core App For IIS

The first thing you will notice when creating a new ASP.NET Core project is that it's a console application. Your project now contains a Program.cs file, just like a console app, plus the following code:

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

        host.Run();
    }
}
```

What is the WebHostBuilder?

All ASP.NET Core applications require a **WebHost** object that essentially serves as the application and web server. In this case, a **WebHostBuilder** is used to configure and create the WebHost. You will normally see `UseKestrel()` and `UseIISIntegration()` in the WebHostBuilder setup code.

What do these do?

- **UseKestrel()** – Registers the `IServer` interface for Kestrel as the server that will be used to host your application. In the future, there could be other options, including [WebListener](#) which will be Windows only.
- **UseIISIntegration()** – Tells ASP.NET that IIS will be working as a reverse proxy in front of Kestrel and specifies some settings around which port Kestrel should listen on, forward headers and other details.

If you are planning to deploy your application to IIS, `UseIISIntegration()` is required

What is AspNetCoreModule?

You may have noticed that ASP.NET Core projects create a web.config file. This is only used when deploying your application to IIS and registers the **AspNetCoreModule** as an HTTP handler.

Default web.config for ASP.NET Core:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified"/>
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%" arguments="%LAUNCHER_ARGS%" stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout" forwardWindowsAuthToken="false"/>
  </system.webServer>
</configuration>
```

AspNetCoreModule handles all incoming traffic to IIS, then acts as the reverse proxy that knows how to hand traffic off to your ASP.NET Core application. You can view the source code of it [on GitHub](#). AspNetCoreModule also ensures that your web application is running and is responsible for starting your process up.

Install .NET Core Windows Server Hosting Bundle

Before you deploy your application, you need to install the .NET Core hosting bundle for IIS – .NET Core runtime, libraries and the ASP.NET Core module for IIS.

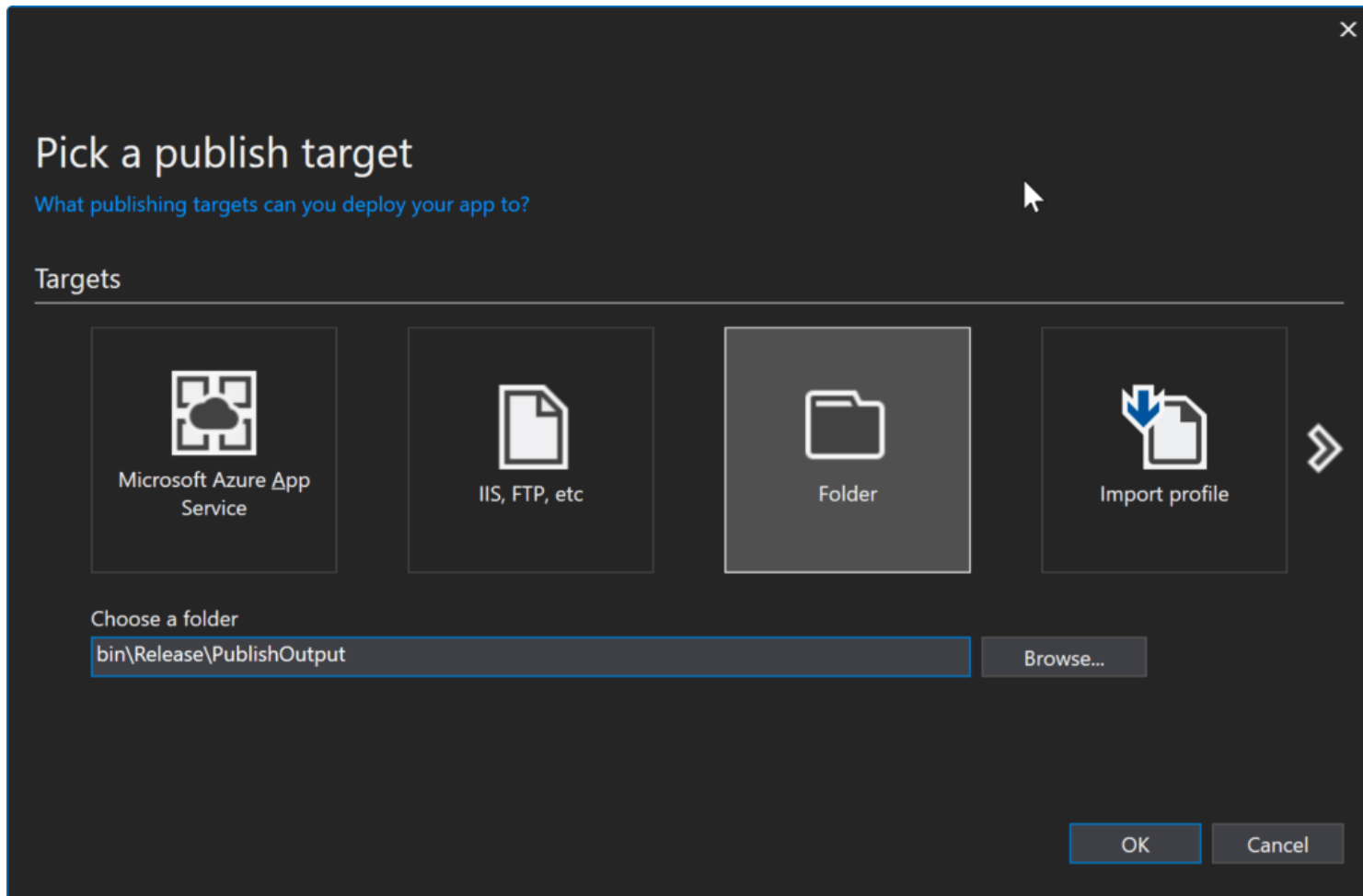
After installation, you may need to do a “net stop was /y” and “net start w3svc” to ensure all the changes are picked up for IIS.

Download: [.NET Core Windows Server Hosting](#) <- Make sure you pick “Windows Server Hosting”

Steps to Deploy ASP.NET Core to IIS

Before you deploy, you need to make sure that WebHostBuilder is configured properly for Kestrel and IIS. Your web.config file should also exist and look similar to our example above.

Step 1: Publish to a File Folder





















Step 2: Copy Files to Preferred IIS Location

Now you need to copy your publish output to where you want the files to live. If you are deploying to a remote server, you may want to zip up the files and move to the server. If you are deploying to a local dev box, you can copy them locally.

For our example, I am copying the files to C:\inetpub\wwwroot\AspNetCore46

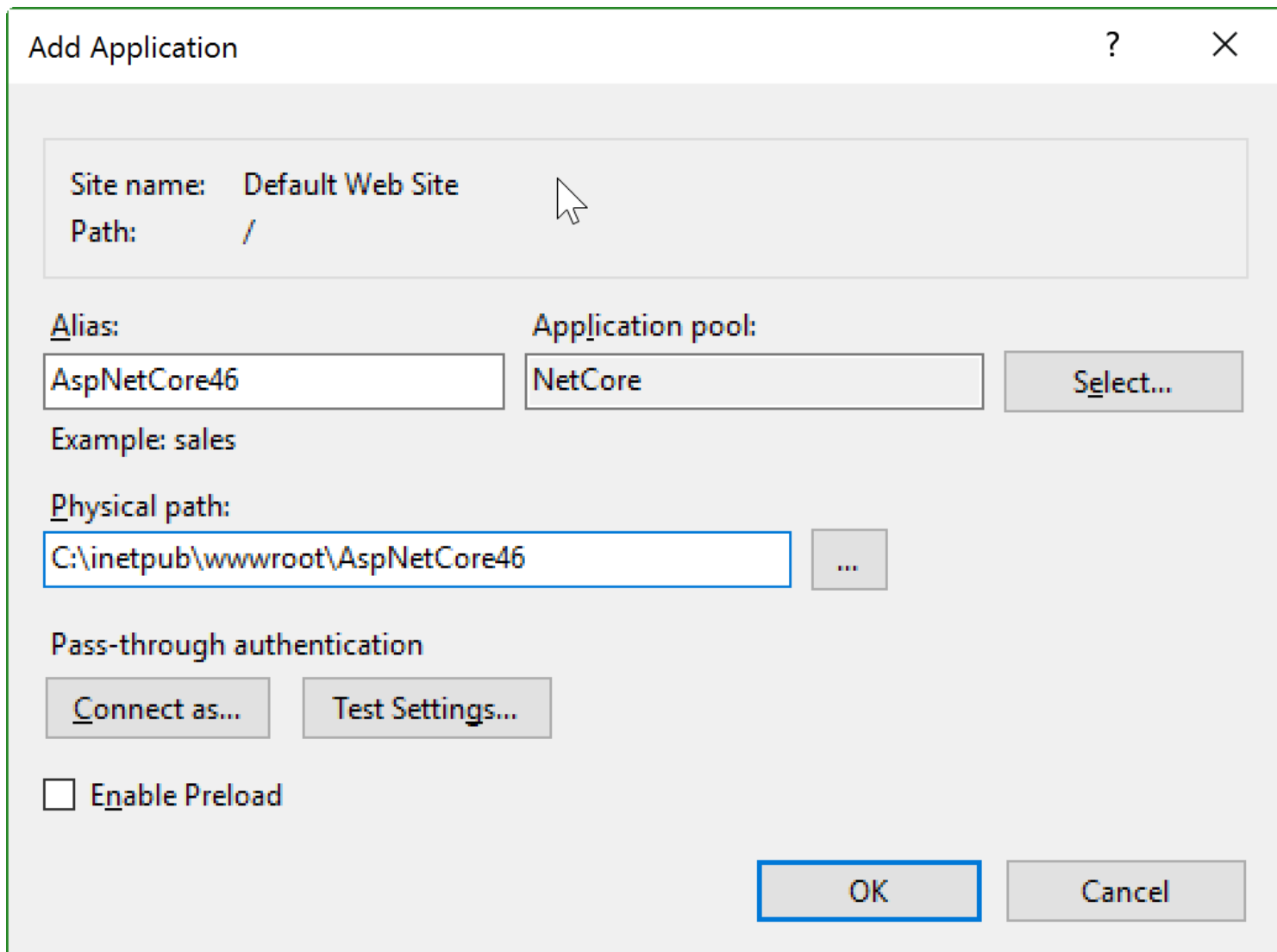
You will notice that with ASP.NET Core, there is no bin folder and it potentially copies over a ton of different .NET DLLs. Your application may also be an EXE file if you are targeting the full .NET Framework. This little sample project had over 100 DLLs in the output.

<input type="checkbox"/> Name	Date modified	Type	Size
<input type="checkbox"/>  refs	4/12/2017 12:54 PM	File folder	
 Views	4/12/2017 12:54 PM	File folder	
 wwwroot	4/12/2017 12:54 PM	File folder	
 appsettings.json	9/14/2016 2:13 PM	JSON File	1 KB
 ASPNetCore46.deps.json	4/12/2017 12:54 PM	JSON File	160 KB
<input checked="" type="checkbox"/>  ASPNetCore46.exe	4/12/2017 12:54 PM	Application	14 KB
 ASPNetCore46.exe.config	4/12/2017 12:54 PM	XML Configuration Fi...	6 KB
 ASPNetCore46.pdb	4/12/2017 12:54 PM	Program Debug Data...	3 KB
 ASPNetCore46.runtimeconfig.json	4/12/2017 12:54 PM	JSON File	1 KB
 bower.json	9/14/2016 2:13 PM	JSON File	1 KB
 bundleconfig.json	9/14/2016 2:13 PM	JSON File	1 KB
 CoreSharedLib.dll	4/12/2017 12:53 PM	Application extension	35 KB
 CoreSharedLib.dll.config	12/21/2016 9:34 AM	XML Configuration Fi...	1 KB
 CoreSharedLib.pdb	4/12/2017 12:53 PM	Program Debug Data...	8 KB
 libuv.dll	6/13/2016 2:17 PM	Application extension	228 KB
 Microsoft.AspNetCore.Antiforgery.dll	2/17/2017 1:40 PM	Application extension	49 KB
 Microsoft.AspNetCore.Authorization.dll	2/17/2017 1:40 PM	Application extension	37 KB
 Microsoft.AspNetCore.Cors.dll	2/17/2017 1:40 PM	Application extension	31 KB

Step 3: Create Application in IIS

While creating your application in IIS is listed as a single “Step,” you will take multiple actions. First, create a new IIS Application Pool under the .NET CLR version of “**No Managed Code**”. Since IIS only works as a reverse proxy, it isn’t actually executing any .NET code.

Second, you can create your application under an existing or a new IIS Site. Either way, you will want to pick your new IIS Application Pool and point it to the folder you copied your ASP.NET publish output files to.



The screenshot shows the 'Add Application' dialog box in IIS Manager. The fields are as follows:

- Site name: Default Web Site
- Path: /
- Alias: AspNetCore46
- Application pool: NetCore
- Example: sales
- Physical path: C:\inetpub\wwwroot\AspNetCore46
- Buttons: Connect as..., Test Settings..., Enable Preload (checkbox), OK, Cancel

Step 4: Load Your App!

At this point, your application should load just fine. If it does not, check the output logging from it. Within your web.config file you define how IIS starts up your ASP.NET Core process. Enable output logging by setting **stdoutLogEnabled=true**. You may also want to change the log output location as configured in **stdoutLogFile**. Check out the example web.config before to see where they are set.

Advantages of Using IIS with ASP.NET Core Hosting

Microsoft recommends using IIS with any public facing site for ASP.NET Core hosting. IIS provides additional levels of configurability, management, security and logging, among many other things.

Check out our blog post about [Kestrel vs IIS](#) to see a whole matrix of feature differences. The post goes into more depth about what Kestrel is and why you need both Kestrel and IIS.

One of the big advantages to using IIS is the **process management**. IIS will automatically **start your app** and potentially restart it if a crash were to occur. If you were running your [ASP.NET Core app as a Windows Service](#) or console app, you would not have that safety net there to start and monitor the process for you.