


Host ASP.NET Core on Windows with IIS

05/07/2020 76 minutes to read  +8

In this article

Supported operating systems

Supported platforms

Application configuration

IIS configuration

Install the .NET Core Hosting Bundle

Install Web Deploy when publishing with Visual Studio

Create the IIS site

Deploy the app

Browse the website

Locked deployment files

Data protection

Virtual Directories

Sub-applications

Configuration of IIS with web.config

Configuration sections of web.config

Application Pools

Application Pool Identity

HTTP/2 support

CORS preflight requests

Deployment resources for IIS administrators

Additional resources

For a tutorial experience on publishing an ASP.NET Core app to an IIS server, see [Publish an ASP.NET Core app to IIS](#).

[Install the .NET Core Hosting Bundle](#)

Supported operating systems

The following operating systems are supported:

- Windows 7 or later
- Windows Server 2008 R2 or later

[HTTP.sys server](#) (formerly called WebListener) doesn't work in a reverse proxy configuration with IIS. Use the [Kestrel server](#).

For information on hosting in Azure, see [Deploy ASP.NET Core apps to Azure App Service](#).

For troubleshooting guidance, see [Troubleshoot and debug ASP.NET Core projects](#).

Supported platforms

Apps published for 32-bit (x86) or 64-bit (x64) deployment are supported. Deploy a 32-bit app with a 32-bit (x86) .NET Core SDK unless the app:

- Requires the larger virtual memory address space available to a 64-bit app.
- Requires the larger IIS stack size.
- Has 64-bit native dependencies.

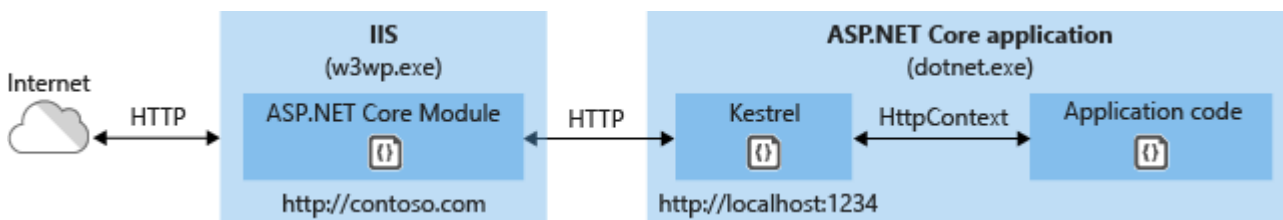
Use a 64-bit (x64) .NET Core SDK to publish a 64-bit app. A 64-bit runtime must be present on the host system.

ASP.NET Core ships with [Kestrel server](#), a default, cross-platform HTTP server.

When using [IIS](#) or [IIS Express](#), the app runs in a process separate from the IIS worker process (*out-of-process*) with the [Kestrel server](#).

Because ASP.NET Core apps run in a process separate from the IIS worker process, the module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the [Windows Process Activation Service \(WAS\)](#).

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted out-of-process:



Requests arrive from the web to the kernel-mode HTTP.sys driver. The driver routes the requests to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS). The module forwards the requests to Kestrel on a random port for the app, which isn't port 80 or 443.

The module specifies the port via an environment variable at startup, and the [IIS Integration Middleware](#) configures the server to listen on `http://localhost:{port}`. Additional checks are performed, and requests that don't originate from the module are rejected. The module doesn't support HTTPS forwarding, so requests are forwarded over HTTP even if received by IIS over HTTPS.

After Kestrel picks up the request from the module, the request is pushed into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic. Middleware added by IIS Integration updates the scheme,

remote IP, and pathbase to account for forwarding the request to Kestrel. The app's response is passed back to IIS, which pushes it back out to the HTTP client that initiated the request.

`CreateDefaultBuilder` configures [Kestrel](#) server as the web server and enables IIS Integration by configuring the base path and port for the [ASP.NET Core Module](#).

The ASP.NET Core Module generates a dynamic port to assign to the backend process. `CreateDefaultBuilder` calls the [UseIISIntegration](#) method. `UseIISIntegration` configures Kestrel to listen on the dynamic port at the localhost IP address (127.0.0.1). If the dynamic port is 1234, Kestrel listens at 127.0.0.1:1234. This configuration replaces other URL configurations provided by:

- `UseUrls`
- Kestrel's `Listen` API
- Configuration (or command-line `--urls` option)

Calls to `UseUrls` or Kestrel's `Listen` API aren't required when using the module. If `UseUrls` or `Listen` is called, Kestrel listens on the port specified only when running the app without IIS.

For ASP.NET Core Module configuration guidance, see [ASP.NET Core Module](#).

For more information on hosting, see [Host in ASP.NET Core](#).

Application configuration

Enable the `IISIntegration` components

When building a host in `CreateWebApplicationBuilder` (*Program.cs*), call [CreateDefaultBuilder](#) to enable IIS integration:

C#

Copy

```
public static IWebHostBuilder CreateWebApplicationBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
    ...
```

For more information on `CreateDefaultBuilder`, see [ASP.NET Core Web Host](#).

IIS options

| Option | Default | Setting |
|--------|---------|---------|
|--------|---------|---------|

| Option | Default | Setting |
|---------------------------|---------|--|
| AutomaticAuthentication | true | If <code>true</code> , IIS Server sets the <code>HttpContext.User</code> authenticated by Windows Authentication. If <code>false</code> , the server only provides an identity for <code>HttpContext.User</code> and responds to challenges when explicitly requested by the <code>AuthenticationScheme</code> . Windows Authentication must be enabled in IIS for <code>AutomaticAuthentication</code> to function. For more information, see Windows Authentication. |
| AuthenticationDisplayName | null | Sets the display name shown to users on login pages. |

To configure IIS options, include a service configuration for [IIsoptions](#) in [ConfigureServices](#). The following example prevents the app from populating `HttpContext.Connection.ClientCertificate`:

C#

Copy

```
services.Configure<IIsoptions>(options =>
{
    options.ForwardClientCertificate = false;
});
```

| Option | Default | Setting |
|---------------------------|---------|--|
| AutomaticAuthentication | true | If <code>true</code> , IIS Integration Middleware sets the <code>HttpContext.User</code> authenticated by Windows Authentication. If <code>false</code> , the middleware only provides an identity for <code>HttpContext.User</code> and responds to challenges when explicitly requested by the <code>AuthenticationScheme</code> . Windows Authentication must be enabled in IIS for <code>AutomaticAuthentication</code> to function. For more information, see the Windows Authentication topic. |
| AuthenticationDisplayName | null | Sets the display name shown to users on login pages. |
| ForwardClientCertificate | true | If <code>true</code> and the <code>MS-ASPNETCORE-CLIENTCERT</code> request header is present, the <code>HttpContext.Connection.ClientCertificate</code> is populated. |

Proxy server and load balancer scenarios

The [IIS Integration Middleware](#), which configures Forwarded Headers Middleware, and the ASP.NET Core Module are configured to forward the scheme (HTTP/HTTPS) and the remote IP address where the request originated. Additional configuration might be required for apps hosted behind additional proxy servers and load balancers. For more information, see [Configure ASP.NET Core to work with proxy servers and load balancers](#).

web.config file

The *web.config* file configures the [ASP.NET Core Module](#). Creating, transforming, and publishing the *web.config* file is handled by an MSBuild target (`_TransformWebConfig`) when the project is published. This target is present in the Web SDK targets (`Microsoft.NET.Sdk.Web`). The SDK is set at the top of the project file:

XML

Copy

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

If a *web.config* file isn't present in the project, the file is created with the correct *processPath* and *arguments* to configure the ASP.NET Core Module and moved to [published output](#).

If a *web.config* file is present in the project, the file is transformed with the correct *processPath* and *arguments* to configure the ASP.NET Core Module and moved to published output. The transformation doesn't modify IIS configuration settings in the file.

The *web.config* file may provide additional IIS configuration settings that control active IIS modules. For information on IIS modules that are capable of processing requests with ASP.NET Core apps, see the [IIS modules](#) topic.

To prevent the Web SDK from transforming the *web.config* file, use the **<IsTransformWebConfigDisabled>** property in the project file:

XML

Copy

```
<PropertyGroup>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

When disabling the Web SDK from transforming the file, the *processPath* and *arguments* should be manually set by the developer. For more information, see [ASP.NET Core Module](#).

web.config file location

In order to set up the [ASP.NET Core Module](#) correctly, the *web.config* file must be present at the [content root](#) path (typically the app base path) of the deployed app. This is the same location as the website physical path provided to IIS. The *web.config* file is required at the root of the app to enable the publishing of multiple apps using Web Deploy.

Sensitive files exist on the app's physical path, such as *<assembly>.runtimeconfig.json*, *<assembly>.xml* (XML Documentation comments), and *<assembly>.deps.json*. When the *web.config* file is present and the site starts normally, IIS doesn't serve these sensitive files if they're requested. If the *web.config* file is missing, incorrectly named, or unable to configure the site for normal startup, IIS may serve sensitive files publicly.

The *web.config* file must be present in the deployment at all times, correctly named, and able to configure the site for normal start up. Never remove the *web.config* file from a production deployment.

Transform web.config

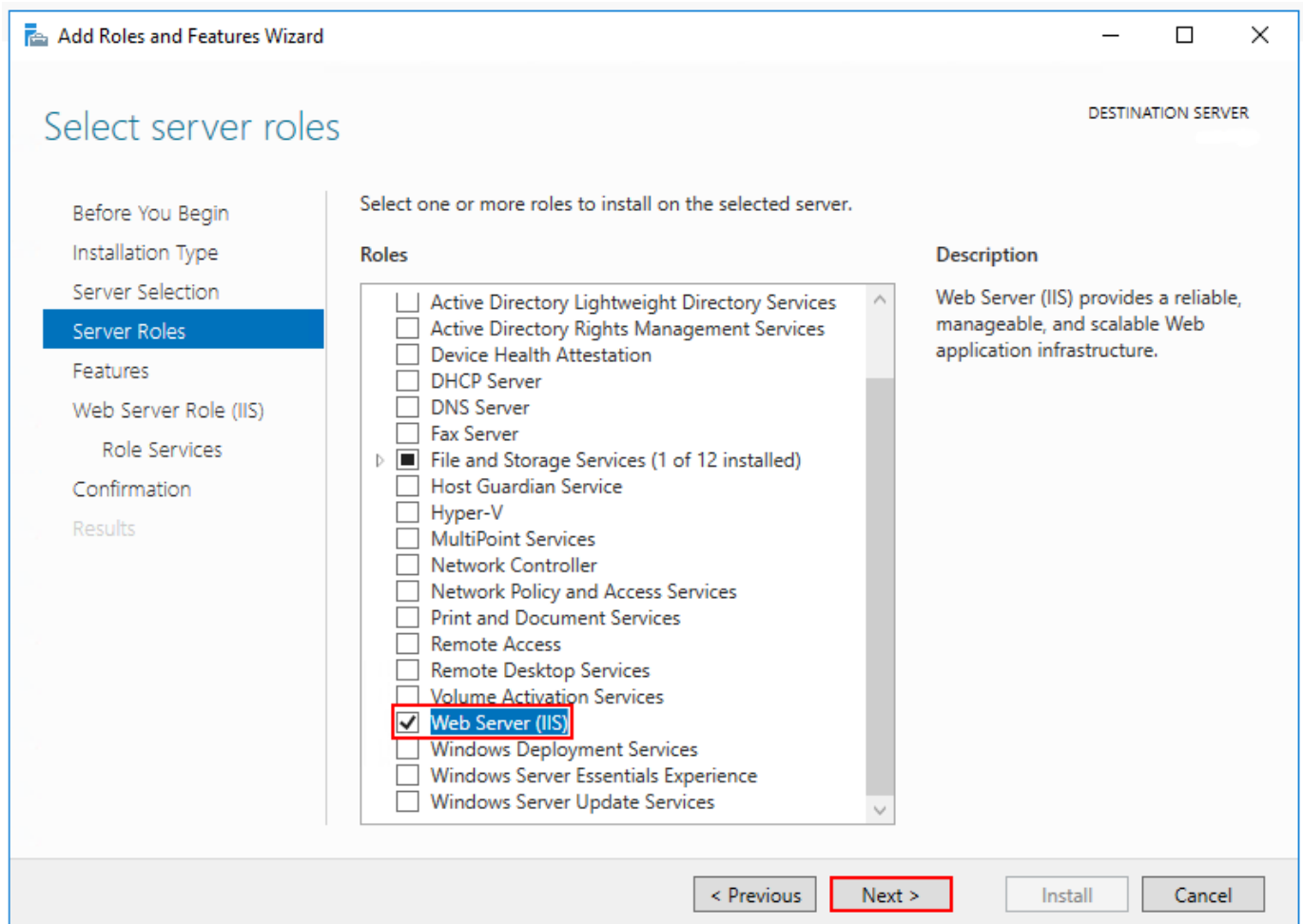
If you need to transform *web.config* on publish (for example, set environment variables based on the configuration, profile, or environment), see [Transform web.config](#).

IIS configuration

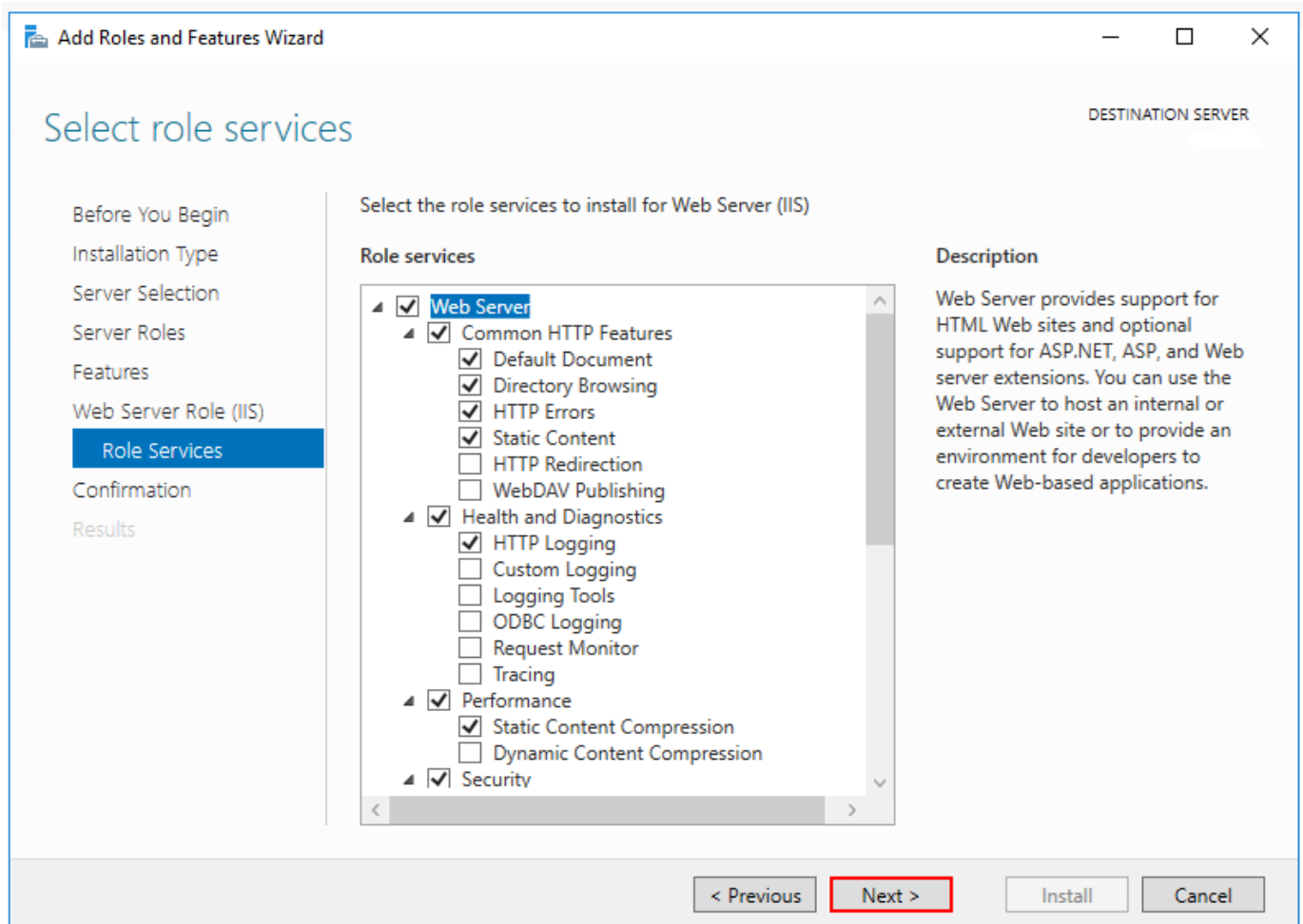
Windows Server operating systems

Enable the **Web Server (IIS)** server role and establish role services.

1. Use the **Add Roles and Features** wizard from the **Manage** menu or the link in **Server Manager**. On the **Server Roles** step, check the box for **Web Server (IIS)**.



- After the **Features** step, the **Role services** step loads for Web Server (IIS). Select the IIS role services desired or accept the default role services provided.



Windows Authentication (Optional)

To enable Windows Authentication, expand the following nodes: **Web Server** > **Security**. Select the **Windows Authentication** feature. For more information, see [Windows Authentication <windowsAuthentication>](#) and [Configure Windows authentication](#).

WebSockets (Optional)

WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **Web Server** > **Application Development**. Select the **WebSocket Protocol** feature. For more information, see [WebSockets](#).

3. Proceed through the **Confirmation** step to install the web server role and services. A server/IIS restart isn't required after installing the **Web Server (IIS)** role.

Windows desktop operating systems

Enable the **IIS Management Console** and **World Wide Web Services**.

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open the **Internet Information Services** node. Open the **Web Management Tools** node.
3. Check the box for **IIS Management Console**.

4. Check the box for **World Wide Web Services**.
5. Accept the default features for **World Wide Web Services** or customize the IIS features.

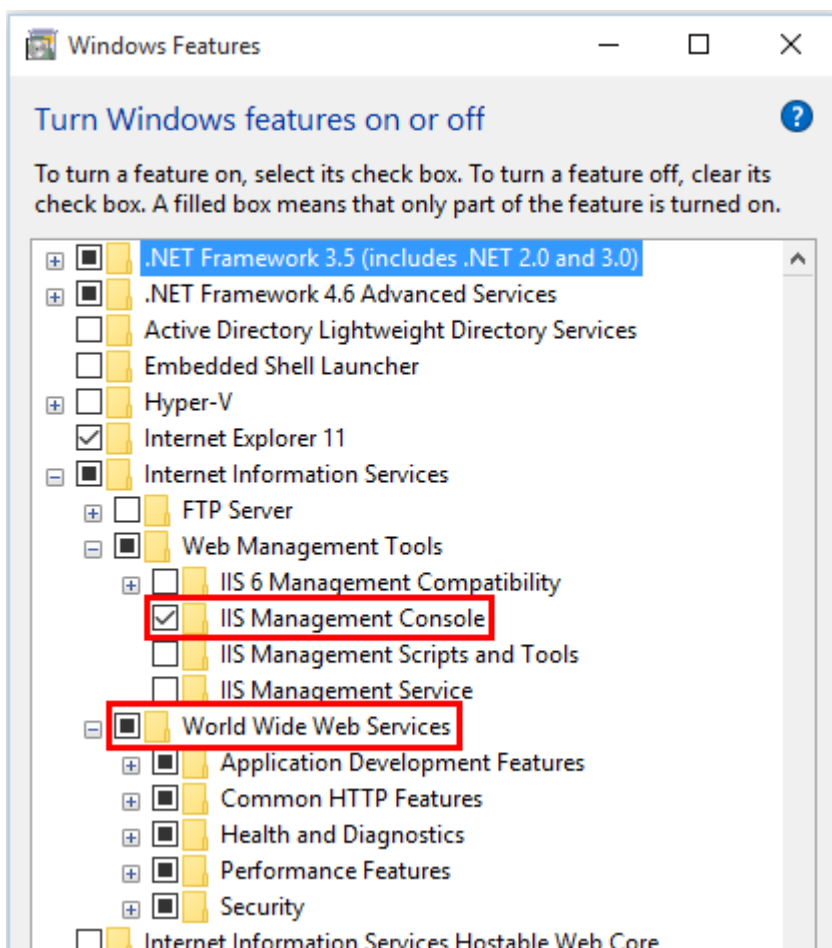
Windows Authentication (Optional)

To enable Windows Authentication, expand the following nodes: **World Wide Web Services** > **Security**. Select the **Windows Authentication** feature. For more information, see [Windows Authentication <windowsAuthentication>](#) and [Configure Windows authentication](#).

WebSockets (Optional)

WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **World Wide Web Services** > **Application Development Features**. Select the **WebSocket Protocol** feature. For more information, see [WebSockets](#).

6. If the IIS installation requires a restart, restart the system.



Install the .NET Core Hosting Bundle

Install the *.NET Core Hosting Bundle* on the hosting system. The bundle installs the .NET Core Runtime, .NET Core Library, and the [ASP.NET Core Module](#). The module allows ASP.NET Core apps to run behind IIS.

Important

If the Hosting Bundle is installed before IIS, the bundle installation must be repaired. Run the Hosting Bundle installer again after installing IIS.

If the Hosting Bundle is installed after installing the 64-bit (x64) version of .NET Core, SDKs might appear to be missing (**No .NET Core SDKs were detected**). To resolve the problem, see [Troubleshoot and debug ASP.NET Core projects](#).

Download

1. Navigate to the Download .NET Core page.
2. Select the desired .NET Core version.
3. In the **Run apps - Runtime** column, find the row of the .NET Core runtime version desired.
4. Download the installer using the **Hosting Bundle** link.

Warning

Some installers contain release versions that have reached their end of life (EOL) and are no longer supported by Microsoft. For more information, see the [support policy](#).

Install the Hosting Bundle

1. Run the installer on the server. The following parameters are available when running the installer from an administrator command shell:

- `OPT_NO_ANCM=1`: Skip installing the ASP.NET Core Module.
- `OPT_NO_RUNTIME=1`: Skip installing the .NET Core runtime. Used when the server only hosts self-contained deployments (SCD).
- `OPT_NO_SHAREDFX=1`: Skip installing the ASP.NET Shared Framework (ASP.NET runtime). Used when the server only hosts self-contained deployments (SCD).
- `OPT_NO_X86=1`: Skip installing x86 runtimes. Use this parameter when you know that you won't be hosting 32-bit apps. If there's any chance that you will host both 32-bit and 64-bit apps in the future, don't use this parameter and install both runtimes.
- `OPT_NO_SHARED_CONFIG_CHECK=1`: Disable the check for using an IIS Shared Configuration when the shared configuration (*applicationHost.config*) is on the same machine as the IIS installation. *Only available for ASP.NET Core 2.2 or later Hosting Bundler installers.* For more information, see ASP.NET Core Module.

2. Restart the system or execute the following commands in a command shell:

Console

```
net stop was /y
net start w3svc
```

Copy

Restarting IIS picks up a change to the system PATH, which is an environment variable, made by the installer.

It isn't necessary to manually stop individual sites in IIS when installing the Hosting Bundle. Hosted apps (IIS sites) restart when IIS restarts. Apps start up again when they receive their first request, including from the [Application Initialization Module](#).

ASP.NET Core adopts roll-forward behavior for patch releases of shared framework packages. When apps hosted by IIS restart with IIS, the apps load with the latest patch releases of their referenced packages when they receive their first request. If IIS isn't restarted, apps restart and exhibit roll-forward behavior when their worker processes are recycled and they receive their first request.

Note

For information on IIS Shared Configuration, see [ASP.NET Core Module with IIS Shared Configuration](#).

Install Web Deploy when publishing with Visual Studio

When deploying apps to servers with [Web Deploy](#), install the latest version of Web Deploy on the server. To install Web Deploy, use the [Web Platform Installer \(WebPI\)](#) or obtain an installer directly from the [Microsoft Download Center](#). The preferred method is to use WebPI. WebPI offers a standalone setup and a configuration for hosting providers.

Create the IIS site

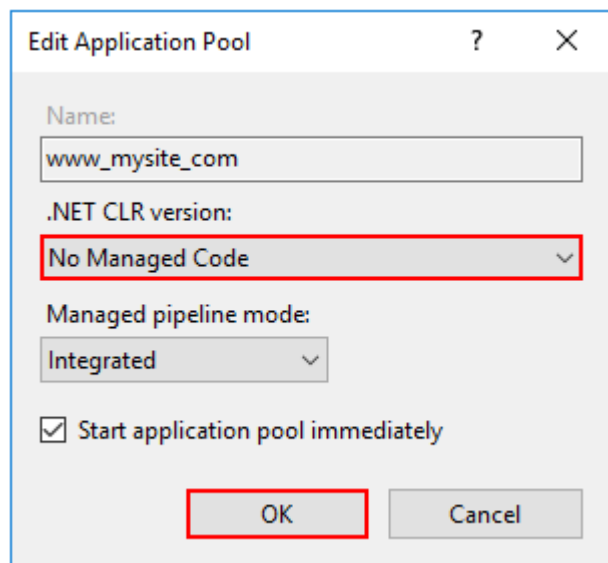
1. On the hosting system, create a folder to contain the app's published folders and files. In a following step, the folder's path is provided to IIS as the physical path to the app. For more information on an app's deployment folder and file layout, see [ASP.NET Core directory structure](#).
2. In IIS Manager, open the server's node in the **Connections** panel. Right-click the **Sites** folder. Select **Add Website** from the contextual menu.
3. Provide a **Site name** and set the **Physical path** to the app's deployment folder. Provide the **Binding** configuration and create the website by selecting **OK**:

The screenshot shows the 'Add Website' dialog box in IIS Manager. The 'Site name' field is set to 'www_mysite_com'. The 'Application pool' is set to 'www_mysite_com'. The 'Physical path' is set to 'F:\www_mysite_com'. The 'Type' is set to 'http', 'IP address' is 'All Unassigned', and 'Port' is '80'. The 'Host name' is set to 'www.mysite.com'. The 'Start Website immediately' checkbox is checked. The 'OK' button is highlighted.

Warning

Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings can open up your app to security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) doesn't have this security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). See [rfc7230 section-5.4](#) for more information.

4. Under the server's node, select **Application Pools**.
5. Right-click the site's app pool and select **Basic Settings** from the contextual menu.
6. In the **Edit Application Pool** window, set the **.NET CLR version** to **No Managed Code**:



ASP.NET Core runs in a separate process and manages the runtime. ASP.NET Core doesn't rely on loading the desktop CLR (.NET CLR)—the Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process. Setting the **.NET CLR version** to **No Managed Code** is optional but recommended.

7. *ASP.NET Core 2.2 or later:* For a 64-bit (x64) self-contained deployment that uses the in-process hosting model, disable the app pool for 32-bit (x86) processes.

In the **Actions** sidebar of IIS Manager > **Application Pools**, select **Set Application Pool Defaults** or **Advanced Settings**. Locate **Enable 32-Bit Applications** and set the value to `False`. This setting doesn't affect apps deployed for out-of-process hosting.

8. Confirm the process model identity has the proper permissions.

If the default identity of the app pool (**Process Model** > **Identity**) is changed from **ApplicationPoolIdentity** to another identity, verify that the new identity has the required permissions to access the app's folder, database, and other required resources. For example, the app pool requires read and write access to folders where the app reads and writes files.

Windows Authentication configuration (Optional)

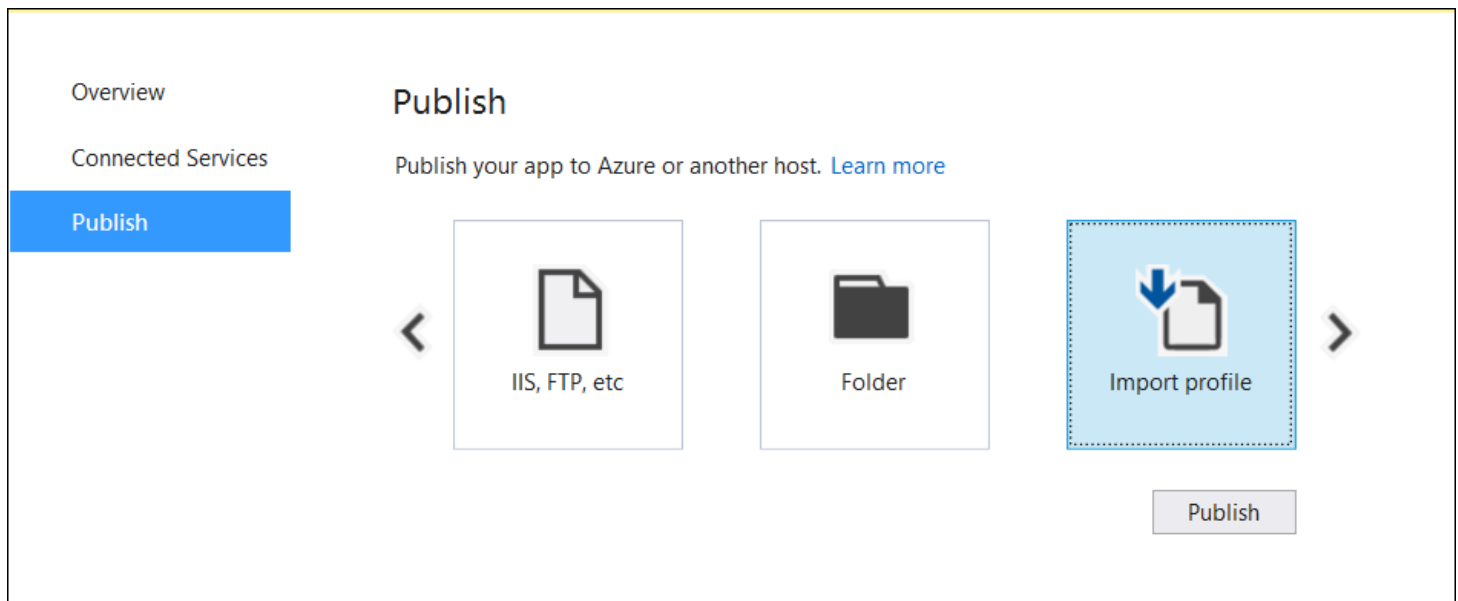
For more information, see [Configure Windows authentication](#).

Deploy the app

Deploy the app to the IIS **Physical path** folder that was established in the [Create the IIS site](#) section. [Web Deploy](#) is the recommended mechanism for deployment, but several options exist for moving the app from the project's *publish* folder to the hosting system's deployment folder.

Web Deploy with Visual Studio

See the [Visual Studio publish profiles for ASP.NET Core app deployment](#) topic to learn how to create a publish profile for use with Web Deploy. If the hosting provider provides a Publish Profile or support for creating one, download their profile and import it using the Visual Studio **Publish** dialog:



Web Deploy outside of Visual Studio

[Web Deploy](#) can also be used outside of Visual Studio from the command line. For more information, see [Web Deployment Tool](#).

Alternatives to Web Deploy

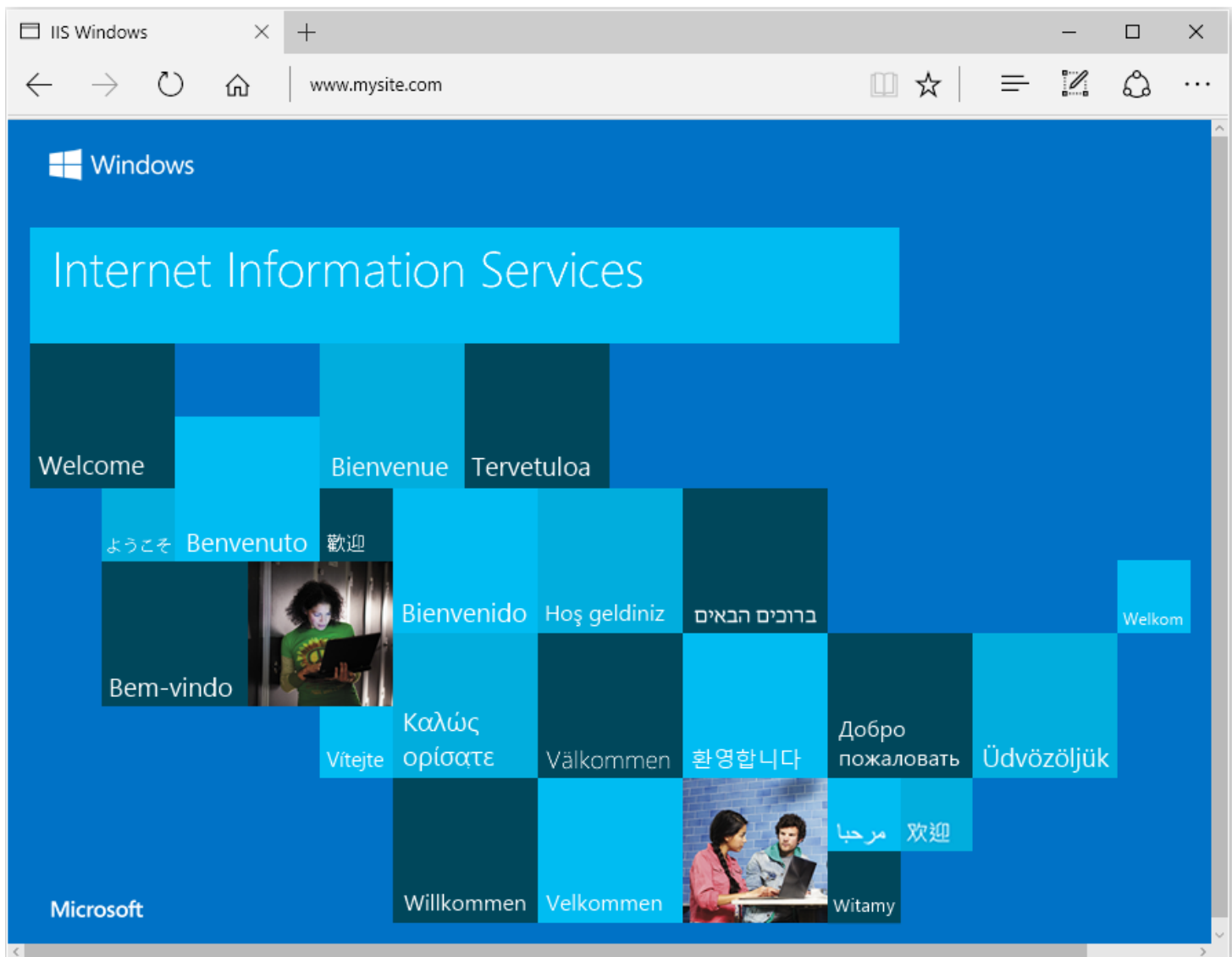
Use any of several methods to move the app to the hosting system, such as manual copy, [Xcopy](#), [Robocopy](#), or [PowerShell](#).

For more information on ASP.NET Core deployment to IIS, see the [Deployment resources for IIS administrators](#) section.

Browse the website

After the app is deployed to the hosting system, make a request to one of the app's public endpoints.

In the following example, the site is bound to an IIS **Host name** of `www.mysite.com` on **Port** 80. A request is made to `http://www.mysite.com`:



Locked deployment files

Files in the deployment folder are locked when the app is running. Locked files can't be overwritten during deployment. To release locked files in a deployment, stop the app pool using **one** of the following approaches:

- Use Web Deploy and reference `Microsoft.NET.Sdk.Web` in the project file. An `app_offline.htm` file is placed at the root of the web app directory. When the file is present, the ASP.NET Core Module gracefully shuts down the app and serves the `app_offline.htm` file during the deployment. For more information, see the [ASP.NET Core Module configuration reference](#).
- Manually stop the app pool in the IIS Manager on the server.
- Use PowerShell to drop `app_offline.htm` (requires PowerShell 5 or later):

PowerShell

Copy

```
$pathToApp = 'PATH_TO_APP'  
  
# Stop the AppPool  
New-Item -Path $pathToApp app_offline.htm  
  
# Provide script commands here to deploy the app  
  
# Restart the AppPool  
Remove-Item -Path $pathToApp app_offline.htm
```

Data protection

The [ASP.NET Core Data Protection stack](#) is used by several ASP.NET Core [middlewares](#), including middleware used in authentication. Even if Data Protection APIs aren't called by user code, data protection should be configured with a deployment script or in user code to create a persistent cryptographic [key store](#). If data protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the key ring is stored in memory when the app restarts:

- All cookie-based authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the key ring can no longer be decrypted. This may include CSRF tokens and ASP.NET Core MVC TempData cookies.

To configure data protection under IIS to persist the key ring, use **one** of the following approaches:

- **Create Data Protection Registry Keys**

Data protection keys used by ASP.NET Core apps are stored in the registry external to the apps. To persist the keys for a given app, create registry keys for the app pool.

For standalone, non-webfarm IIS installations, the [Data Protection Provision-AutoGenKeys.ps1 PowerShell script](#) can be used for each app pool used with an ASP.NET Core app. This script creates a registry key in the HKLM registry that's accessible only to the worker process account of the app's app pool. Keys are encrypted at rest using DPAPI with a machine-wide key.

In web farm scenarios, an app can be configured to use a UNC path to store its data protection key ring. By default, the data protection keys aren't encrypted. Ensure that the file permissions for the network share are limited to the Windows account the app runs under. An X509 certificate can be used to protect keys at rest. Consider a mechanism to allow users to upload certificates: Place certificates into the user's trusted certificate store and ensure they're available on all machines where the user's app runs. See [Configure ASP.NET Core Data Protection](#) for details.

- **Configure the IIS Application Pool to load the user profile**

This setting is in the **Process Model** section under the **Advanced Settings** for the app pool. Set **Load User Profile** to `True`. When set to `True`, keys are stored in the user profile directory and protected using DPAPI with a key specific to the user account. Keys are persisted to the `%LOCALAPPDATA%/ASP.NET/DataProtection-Keys` folder.

The app pool's `setProfileEnvironment` attribute must also be enabled. The default value of `setProfileEnvironment` is `true`. In some scenarios (for example, Windows OS), `setProfileEnvironment` is set to `false`. If keys aren't stored in the user profile directory as expected:

1. Navigate to the `%windir%/system32/inetsrv/config` folder.
2. Open the `applicationHost.config` file.
3. Locate the `<system.applicationHost><applicationPools><applicationPoolDefaults><processModel>` element.
4. Confirm that the `setProfileEnvironment` attribute isn't present, which defaults the value to `true`, or explicitly set the attribute's value to `true`.

- **Use the file system as a key ring store**

Adjust the app code to [use the file system as a key ring store](#). Use an X509 certificate to protect the key ring and ensure the certificate is a trusted certificate. If the certificate is self-signed, place the certificate in the Trusted Root store.

When using IIS in a web farm:

- Use a file share that all machines can access.
- Deploy an X509 certificate to each machine. Configure data protection in code.

- **Set a machine-wide policy for data protection**

The data protection system has limited support for setting a default [machine-wide policy](#) for all apps that consume the Data Protection APIs. For more information, see [ASP.NET Core Data Protection](#).

Virtual Directories

[IIS Virtual Directories](#) aren't supported with ASP.NET Core apps. An app can be hosted as a [sub-application](#).

Sub-applications

An ASP.NET Core app can be hosted as an [IIS sub-application](#) (sub-app). The sub-app's path becomes part of the root app's URL.

A sub-app shouldn't include the ASP.NET Core Module as a handler. If the module is added as a handler in a sub-app's `web.config` file, a `500.19 Internal Server Error` referencing the faulty config file is received when attempting to browse the sub-app.

The following example shows a published *web.config* file for an ASP.NET Core sub-app:

XML

Copy

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <aspNetCore processPath="dotnet"
      arguments=".\\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

When hosting a non-ASP.NET Core sub-app underneath an ASP.NET Core app, explicitly remove the inherited handler in the sub-app's *web.config* file:

XML

Copy

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <remove name="aspNetCore" />
    </handlers>
    <aspNetCore processPath="dotnet"
      arguments=".\\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

Static asset links within the sub-app should use tilde-slash (~/) notation. Tilde-slash notation triggers a [Tag Helper](#) to prepend the sub-app's pathbase to the rendered relative link. For a sub-app at /subapp_path, an image linked with `src="~/image.png"` is rendered as `src="/subapp_path/image.png"`. The root app's Static File Middleware doesn't process the static file request. The request is processed by the sub-app's Static File Middleware.

If a static asset's `src` attribute is set to an absolute path (for example, `src="/image.png"`), the link is rendered without the sub-app's pathbase. The root app's Static File Middleware attempts to serve the asset from the root app's [web root](#), which results in a *404 - Not Found* response unless the static asset is available from the root app.

To host an ASP.NET Core app as a sub-app under another ASP.NET Core app:

1. Establish an app pool for the sub-app. Set the **.NET CLR Version** to **No Managed Code** because the Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process, not the desktop CLR (.NET CLR).

2. Add the root site in IIS Manager with the sub-app in a folder under the root site.
3. Right-click the sub-app folder in IIS Manager and select **Convert to Application**.
4. In the **Add Application** dialog, use the **Select** button for the **Application Pool** to assign the app pool that you created for the sub-app. Select **OK**.

The assignment of a separate app pool to the sub-app is a requirement when using the in-process hosting model.

For more information on the in-process hosting model and configuring the ASP.NET Core Module, see [ASP.NET Core Module](#).

Configuration of IIS with web.config

IIS configuration is influenced by the `<system.webServer>` section of *web.config* for IIS scenarios that are functional for ASP.NET Core apps with the ASP.NET Core Module. For example, IIS configuration is functional for dynamic compression. If IIS is configured at the server level to use dynamic compression, the `<urlCompression>` element in the app's *web.config* file can disable it for an ASP.NET Core app.

For more information, see the following topics:

- Configuration reference for `<system.webServer>`
- ASP.NET Core Module
- IIS modules with ASP.NET Core

To set environment variables for individual apps running in isolated app pools (supported for IIS 10.0 or later), see the *AppCmd.exe command* section of the [Environment Variables <environmentVariables>](#) topic in the IIS reference documentation.

Configuration sections of web.config

Configuration sections of ASP.NET 4.x apps in *web.config* aren't used by ASP.NET Core apps for configuration:

- `<system.web>`
- `<appSettings>`
- `<connectionStrings>`
- `<location>`

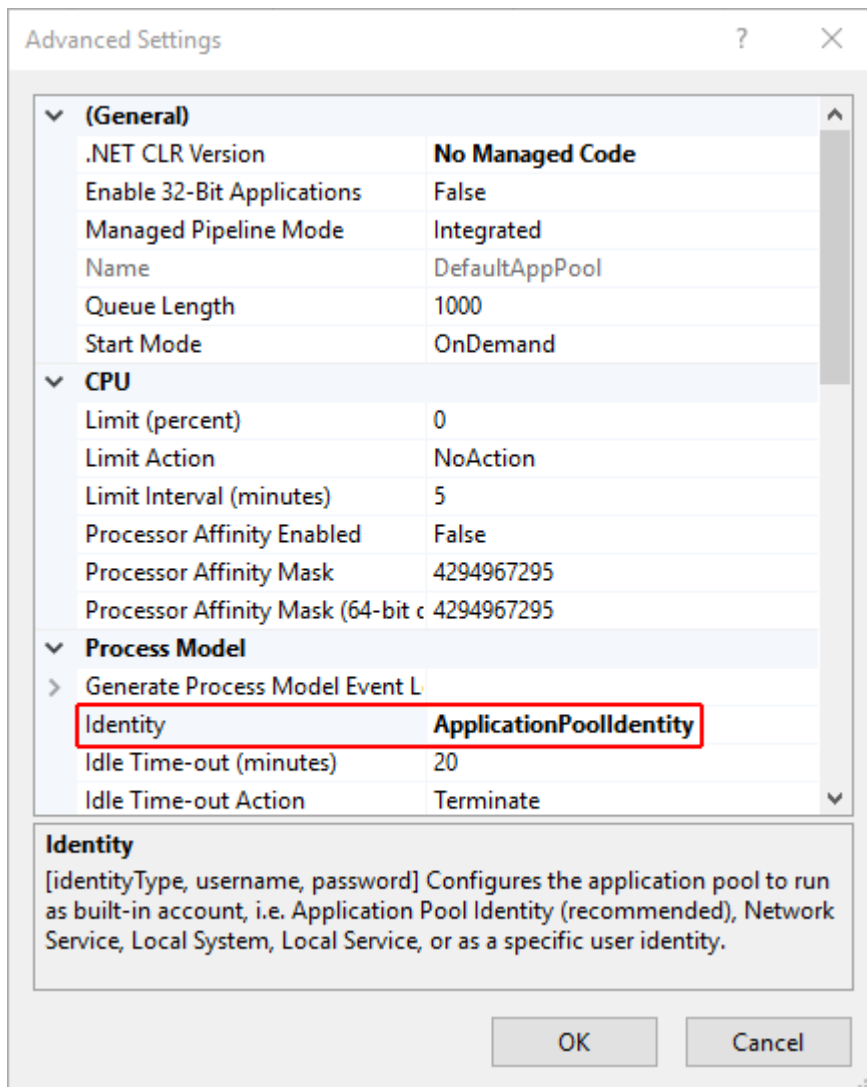
ASP.NET Core apps are configured using other configuration providers. For more information, see [Configuration](#).

Application Pools

When hosting multiple websites on a server, we recommend isolating the apps from each other by running each app in its own app pool. The IIS **Add Website** dialog defaults to this configuration. When a **Site name** is provided, the text is automatically transferred to the **Application pool** textbox. A new app pool is created using the site name when the site is added.

Application Pool Identity

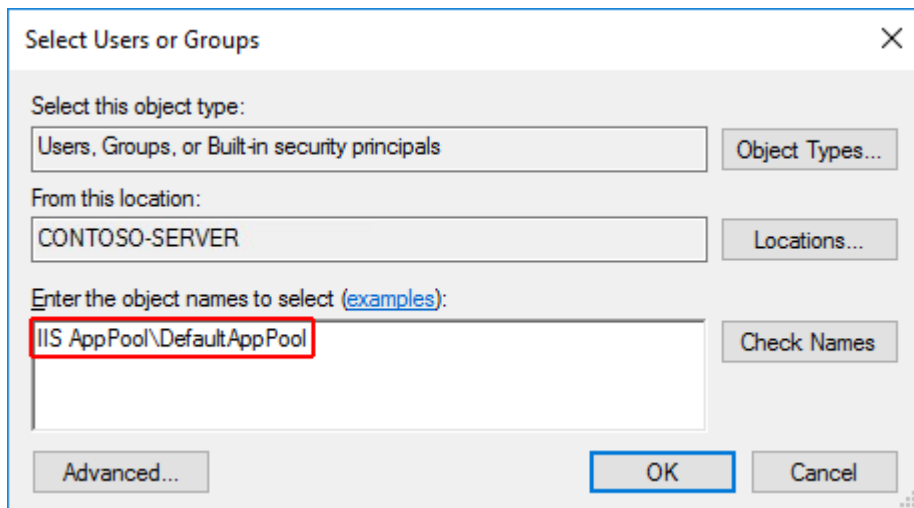
An app pool identity account allows an app to run under a unique account without having to create and manage domains or local accounts. On IIS 8.0 or later, the IIS Admin Worker Process (WAS) creates a virtual account with the name of the new app pool and runs the app pool's worker processes under this account by default. In the IIS Management Console under **Advanced Settings** for the app pool, ensure that the **Identity** is set to use **ApplicationPoolIdentity**:



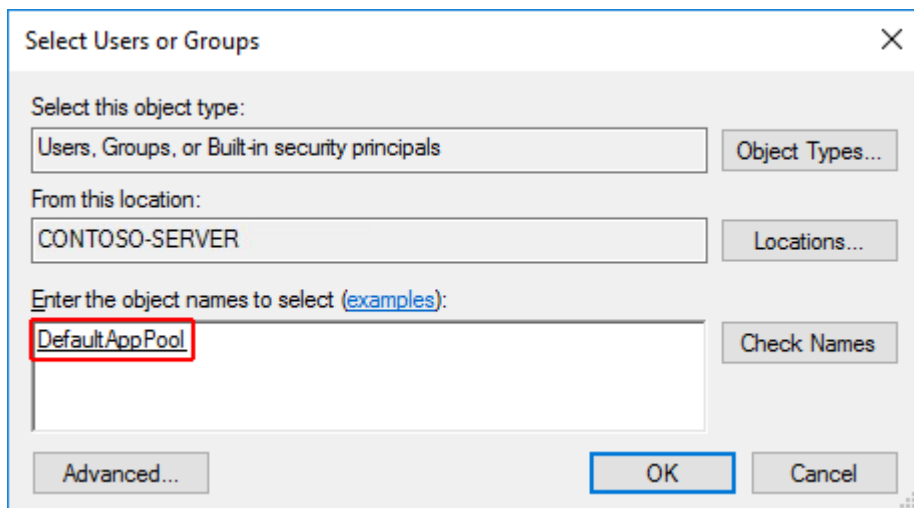
The IIS management process creates a secure identifier with the name of the app pool in the Windows Security System. Resources can be secured using this identity. However, this identity isn't a real user account and doesn't show up in the Windows User Management Console.

If the IIS worker process requires elevated access to the app, modify the Access Control List (ACL) for the directory containing the app:

1. Open Windows Explorer and navigate to the directory.
2. Right-click on the directory and select **Properties**.
3. Under the **Security** tab, select the **Edit** button and then the **Add** button.
4. Select the **Locations** button and make sure the system is selected.
5. Enter **IIS AppPool\<app_pool_name>** in **Enter the object names to select** area. Select the **Check Names** button. For the *DefaultAppPool* check the names using **IIS AppPool\DefaultAppPool**. When the **Check Names** button is selected, a value of **DefaultAppPool** is indicated in the object names area. It isn't possible to enter the app pool name directly into the object names area. Use the **IIS AppPool\<app_pool_name>** format when checking for the object name.



6. Select **OK**.



7. Read & execute permissions should be granted by default. Provide additional permissions as needed.

Access can also be granted at a command prompt using the **ICACLS** tool. Using the *DefaultAppPool* as an example, the following command is used:

```
ICACLS C:\sites\MyWebApp /grant "IIS AppPool\DefaultAppPool":F
```

For more information, see the [icaccls](#) topic.

HTTP/2 support

[HTTP/2](#) is supported for out-of-process deployments that meet the following base requirements:

- Windows Server 2016/Windows 10 or later; IIS 10 or later
- Public-facing edge server connections use HTTP/2, but the reverse proxy connection to the Kestrel server uses HTTP/1.1.
- Target framework: Not applicable to out-of-process deployments, since the HTTP/2 connection is handled entirely by IIS.
- TLS 1.2 or later connection

If an HTTP/2 connection is established, [HttpRequest.Protocol](#) reports HTTP/1.1.

HTTP/2 is enabled by default. Connections fall back to HTTP/1.1 if an HTTP/2 connection isn't established. For more information on HTTP/2 configuration with IIS deployments, see [HTTP/2 on IIS](#).

CORS preflight requests

This section only applies to ASP.NET Core apps that target the .NET Framework.

For an ASP.NET Core app that targets the .NET Framework, OPTIONS requests aren't passed to the app by default in IIS. To learn how to configure the app's IIS handlers in *web.config* to pass OPTIONS requests, see [Enable cross-origin requests in ASP.NET Web API 2: How CORS Works](#).

Deployment resources for IIS administrators

- IIS documentation
- Getting Started with the IIS Manager in IIS
- .NET Core application deployment
- ASP.NET Core Module
- ASP.NET Core directory structure
- IIS modules with ASP.NET Core
- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common errors reference for Azure App Service and IIS with ASP.NET Core

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- [Introduction to ASP.NET Core](#)
- The Official Microsoft IIS Site
- Windows Server technical content library
- HTTP/2 on IIS
- Transform web.config

Is this page helpful?

Yes No