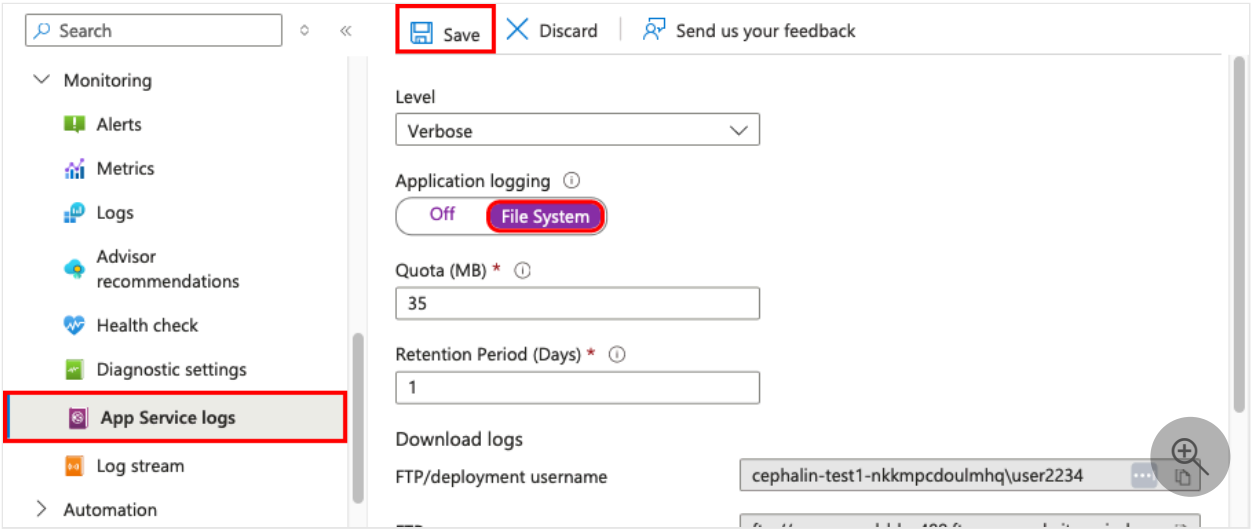
Processing time in the webpage shows a much faster time because it's loading the data from the cache instead of the database.

7. Stream diagnostic logs

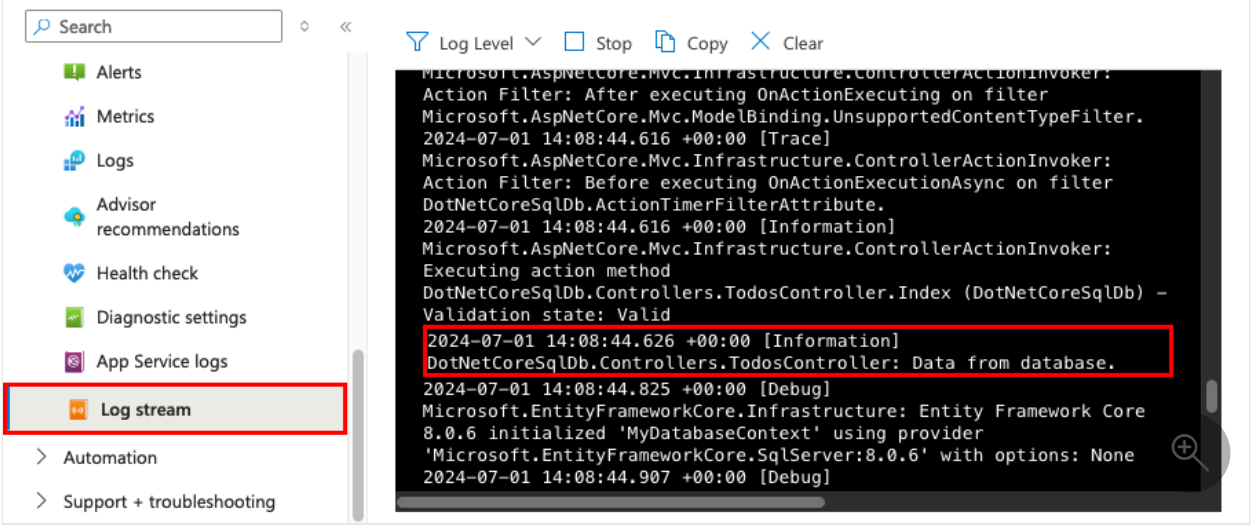
Azure App Service captures all messages logged to the console to assist you in diagnosing issues with your application. The sample app outputs console log messages in each of its endpoints to demonstrate this capability.

Step 1: In the App Service page:

1. From the left menu, select **Monitoring** > **App Service logs**.
2. Under **Application logging**, select **File System**, then select **Save**.



Step 2: From the left menu, select **Log stream**. You see the logs for your app, including platform logs and logs from inside the container.

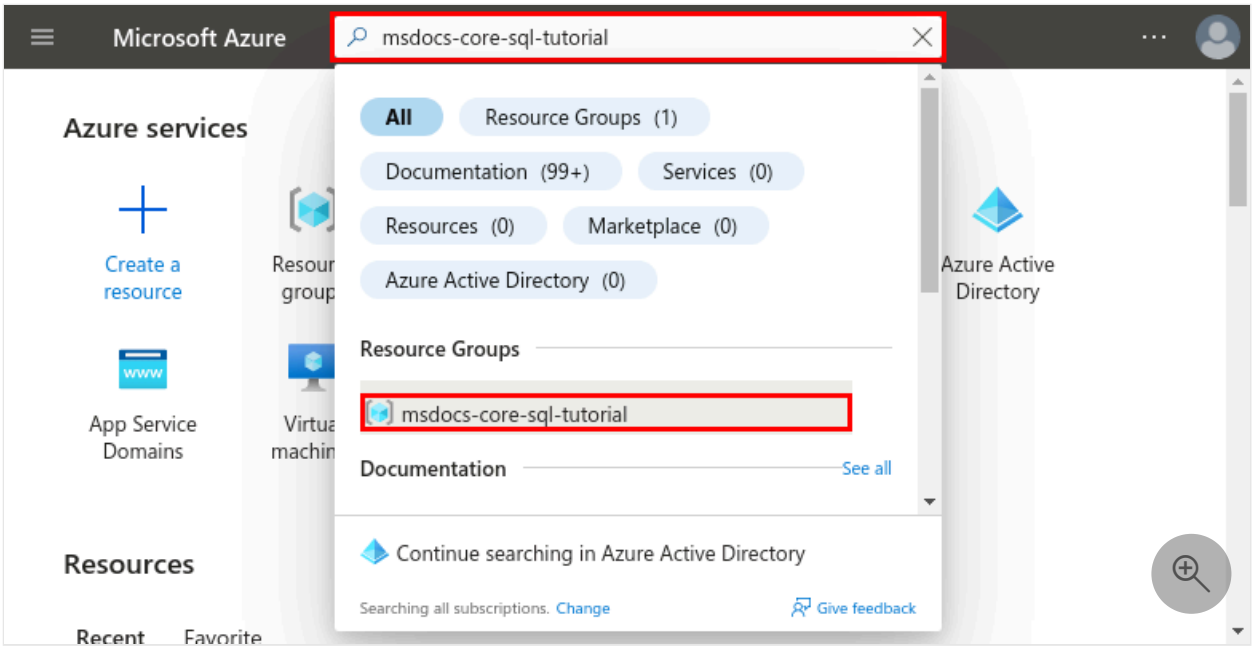


8. Clean up resources

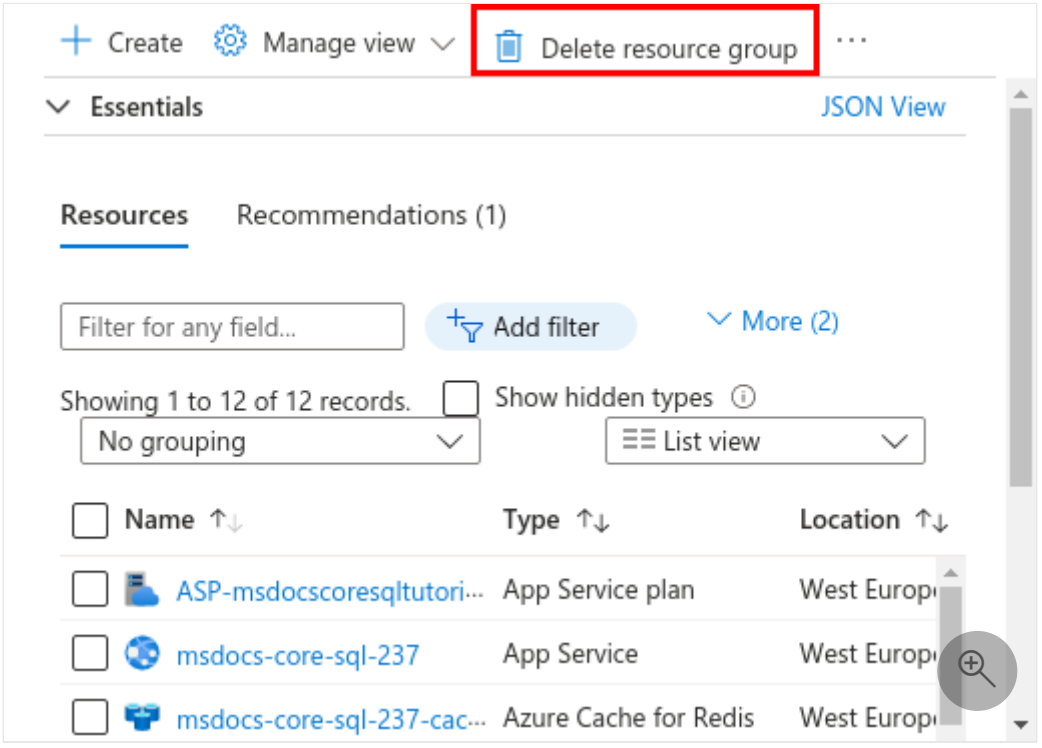
When you're finished, you can delete all of the resources from your Azure subscription by deleting the resource group.

Step 1: In the search bar at the top of the Azure portal:

1. Enter the resource group name.
2. Select the resource group.

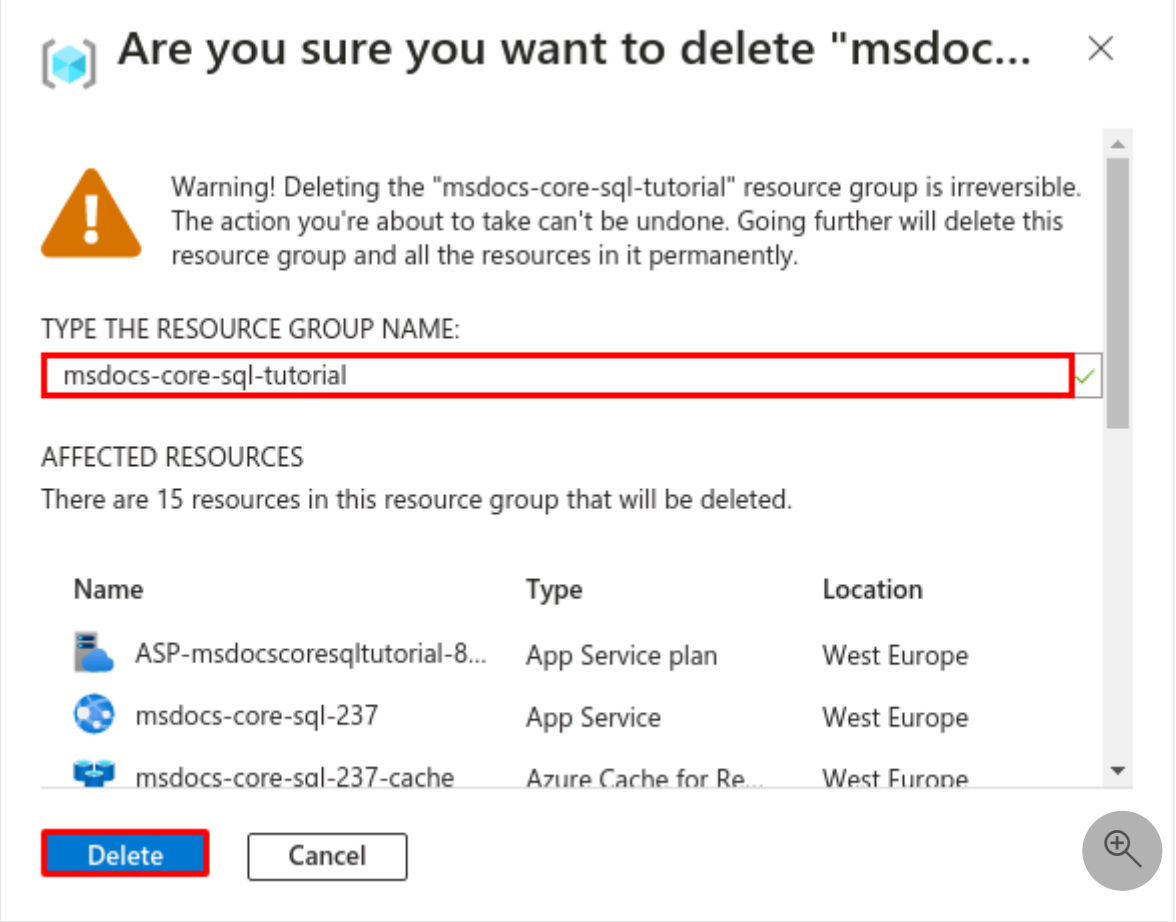


Step 2: In the resource group page, select **Delete resource group**.



Step 3:

1. Enter the resource group name to confirm your deletion.
2. Select **Delete**.



:

Troubleshooting

- The portal deployment view for Azure SQL Database shows a Conflict status
- In the Azure portal, the log stream UI for the web app shows network errors
- The SSH session in the browser shows SSH CONN CLOSED
- The portal log stream page shows Connected! but no logs

The portal deployment view for Azure SQL Database shows a Conflict status

Depending on your subscription and the region you select, you might see the deployment status for Azure SQL Database to be `Conflict`, with the following message in Operation details:

```
Location '<region>' is not accepting creation of new Windows Azure SQL Database servers at this time.
```

This error is most likely caused by a limit on your subscription for the region you select. Try choosing a different region for your deployment.

In the Azure portal, the log stream UI for the web app shows network errors

You might see this error:

```
Unable to open a connection to your app. This may be due to any network
security groups or IP restriction rules that you have placed on your app. To
use log streaming, please make sure you are able to access your app directly
from your current network.
```

This is usually a transient error when the app is first started. Wait a few minutes and check again.

The SSH session in the browser shows `SSH CONN CLOSED`

It takes a few minutes for the Linux container to start up. Wait a few minutes and check again.

The portal log stream page shows `Connected!` but no logs

After you configure diagnostic logs, the app is restarted. You might need to refresh the page for the changes to take effect in the browser.

Frequently asked questions

How much does this setup cost?

Pricing for the created resources is as follows:

- The App Service plan is created in **Basic** tier and can be scaled up or down. See App Service pricing.
- The Azure SQL Database is created in general-purpose, serverless tier on Standard-series hardware with the minimum cores. There's a small cost and can be distributed to other regions. You can minimize cost even more by reducing its maximum size, or you can scale it up by adjusting the serving tier, compute tier, hardware configuration, number of cores, database size, and zone redundancy. See Azure SQL Database pricing.
- The Azure Cache for Redis is created in **Basic** tier with the minimum cache size. There's a small cost associated with this tier. You can scale it up to higher performance tiers for higher availability, clustering, and other features. See Azure Cache for Redis pricing.
- The virtual network doesn't incur a charge unless you configure extra functionality, such as peering. See Azure Virtual Network pricing.
- The private DNS zone incurs a small charge. See Azure DNS pricing.

How do I connect to the Azure SQL Database server that's secured behind the virtual network with other tools?

- For basic access from a command-line tool, you can run `sqlcmd` from the app's SSH terminal. The app's container doesn't come with `sqlcmd`, so you must install it manually. Remember that the installed client doesn't persist across app restarts.
- To connect from a SQL Server Management Studio client or from Visual Studio, your machine must be within the virtual network. For example, it could be an Azure VM that's connected to one of the subnets, or a machine in an on-premises network that has a site-to-site VPN connection with the Azure virtual network.

How does local app development work with GitHub Actions?

Take the autogenerated workflow file from App Service as an example, each `git push` kicks off a new build and deployment run. From a local clone of the GitHub repository, you make the desired updates push it to GitHub. For example:

terminal

```
git add .
git commit -m "<some-message>"
git push origin main
```

How do I debug errors during the GitHub Actions deployment?

If a step fails in the autogenerated GitHub workflow file, try modifying the failed command to generate more verbose output. For example, you can get more output from any of the `dotnet` commands by adding the `-v` option. Commit and push your changes to trigger another deployment to App Service.

I don't have permissions to create a user-assigned identity

See Set up GitHub Actions deployment from the Deployment Center.

How do I change the SQL Database connection to use a managed identity instead?

The default connection string to the SQL database is managed by Service Connector, with the name *defaultConnector*, and it uses SQL authentication. To replace it with a connection that uses a managed identity, run the following commands in the cloud shell after replacing the placeholders:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
az sql server update --enable-public-network true
az webapp connection delete sql --connection defaultConnector --resource-
group <group-name> --name <app-name>
az webapp connection create sql --connection defaultConnector --resource-
group <group-name> --name <app-name> --target-resource-group <group-name> --
server <database-server-name> --database <database-name> --client-type
dotnet --system-identity --config-connstr true
az sql server update --enable-public-network false
```

By default, the command `az webapp connection create sql --client-type dotnet --system-identity --config-connstr` does the following:

- Sets your user as the Microsoft Entra ID administrator of the SQL database server.
- Create a system-assigned managed identity and grants it access to the database.
- Generates a passwordless connection string called `AZURE_SQL_CONNECTIONGSTRING`, which your app is already using at the end of the tutorial.

Your app should now have connectivity to the SQL database. For more information, see Tutorial: Connect to Azure databases from App Service without secrets using a managed

identity.

> **Tip**
>
> **Don't want to enable public network connection?** You can skip `az sql server update --enable-public-network true` by running the commands from an **Azure cloud shell that's integrated with your virtual network** if you have the **Owner** role assignment on your subscription.
>
> To grant the identity the required access to the database that's secured by the virtual network, `az webapp connection create sql` needs direct connectivity with Entra ID authentication to the database server. By default, the Azure cloud shell doesn't have this access to the network-secured database.

What can I do with GitHub Copilot in my codespace?

You might have noticed that the GitHub Copilot chat view was already there for you when you created the codespace. For your convenience, we include the GitHub Copilot chat extension in the container definition (see *.devcontainer/devcontainer.json*). However, you need a GitHub Copilot account (30-day free trial available).

A few tips for you when you talk to GitHub Copilot:

- In a single chat session, the questions and answers build on each other and you can adjust your questions to fine-tune the answer you get.
- By default, GitHub Copilot doesn't have access to any file in your repository. To ask questions about a file, open the file in the editor first.
- To let GitHub Copilot have access to all of the files in the repository when preparing its answers, begin your question with `@workspace`. For more information, see Use the @workspace agent.
- In the chat session, GitHub Copilot can suggest changes and (with `@workspace`) even where to make the changes, but it's not allowed to make the changes for you. It's up to you to add the suggested changes and test it.

Here are some other things you can say to fine-tune the answer you get:

- I want this code to run only in production mode.
- I want this code to run only in Azure App Service and not locally.
- The --output-path parameter seems to be unsupported.

Related content

Advance to the next tutorial to learn how to secure your app with a custom domain and certificate.

Secure with custom domain and certificate

Or, check out other resources:

Tutorial: Connect to SQL Database from App Service without secrets using a managed identity

Configure ASP.NET Core app

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Create your first pipeline

Article • 04/03/2024

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

This is a step-by-step guide to using Azure Pipelines to build a sample application from a Git repository. This guide uses YAML pipelines configured with the YAML pipeline editor. If you'd like to use Classic pipelines instead, see Define your Classic pipeline. For guidance on using TFVC, see Build TFVC repositories.

Prerequisites - Azure DevOps

Make sure you have the following items:

- A GitHub account where you can create a repository. Create one for free.
- An Azure DevOps organization. Create one for free. If your team already has one, then make sure you're an administrator of the Azure DevOps project that you want to use.
- An ability to run pipelines on Microsoft-hosted agents. To use Microsoft-hosted agents, your Azure DevOps organization must have access to Microsoft-hosted parallel jobs. You can either purchase a parallel job or you can request a free grant.

Create your first pipeline

Java

Get the Java sample code

To get started, fork the following repository into your GitHub account.

```
https://github.com/MicrosoftDocs/pipelines-java
```

Create your first Java pipeline

1. Sign in to your Azure DevOps organization and go to your project.

2. Go to **Pipelines**, and then select **New pipeline** or **Create pipeline** if creating your first pipeline.

3. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.

4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.

5. When you see the list of repositories, select your repository.

6. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.

7. Azure Pipelines will analyze your repository and recommend the **Maven** pipeline template.

8. When your new pipeline appears, take a look at the YAML to see what it does. When you're ready, select **Save and run**.

9. You're prompted to commit a new `azure-pipelines.yml` file to your repository. After you're happy with the message, select **Save and run** again.

   If you want to watch your pipeline in action, select the build job.

   You just created and ran a pipeline that we automatically created for you, because your code appeared to be a good match for the Maven template.

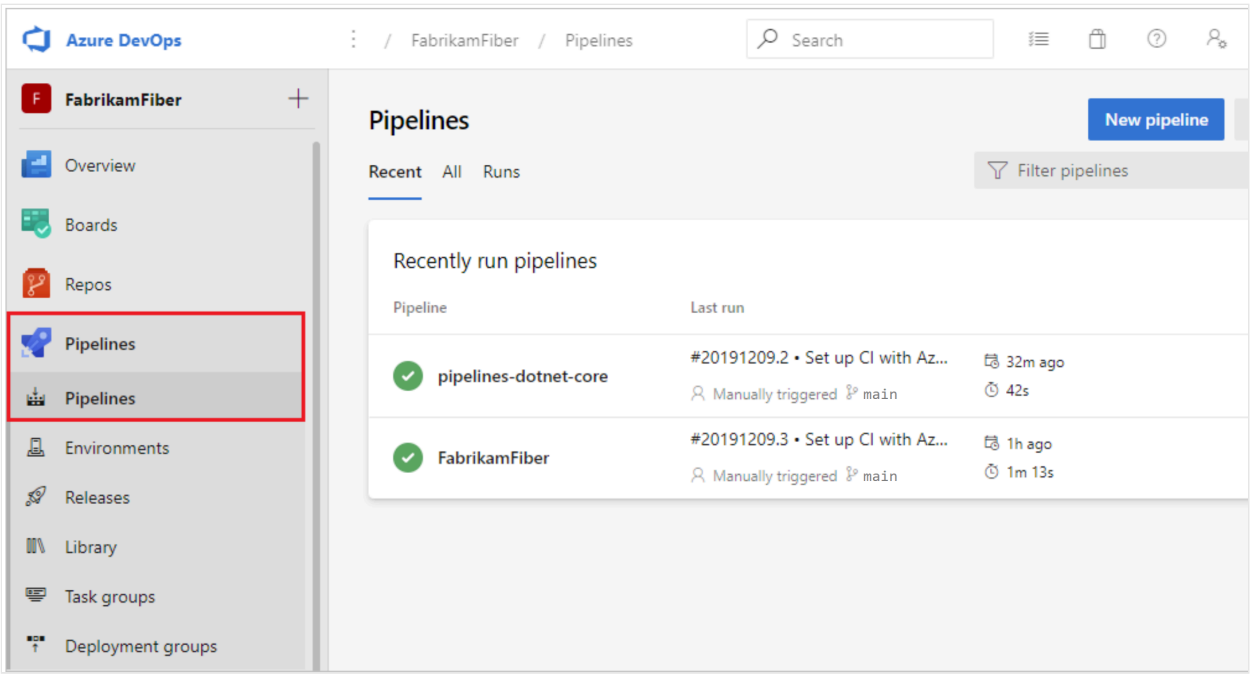
   You now have a working YAML pipeline (`azure-pipelines.yml`) in your repository that's ready for you to customize!

10. When you're ready to make changes to your pipeline, select it in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file.

Learn more about working with Java in your pipeline.

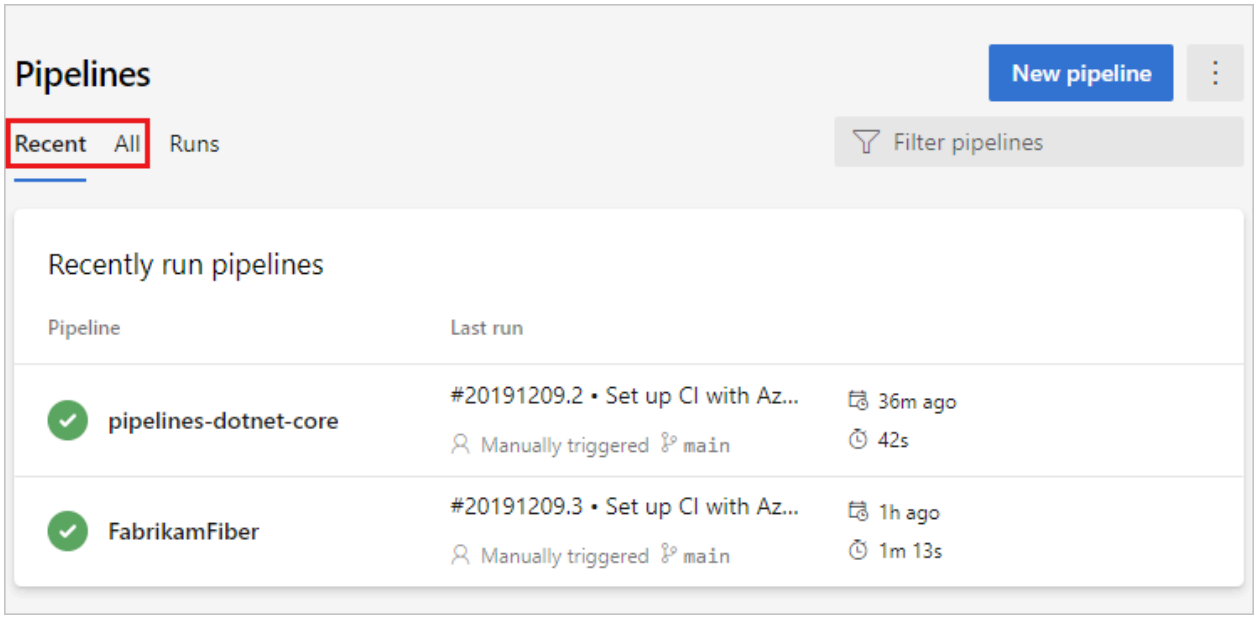
View and manage your pipelines

You can view and manage your pipelines by choosing **Pipelines** from the left-hand menu to go to the pipelines landing page.

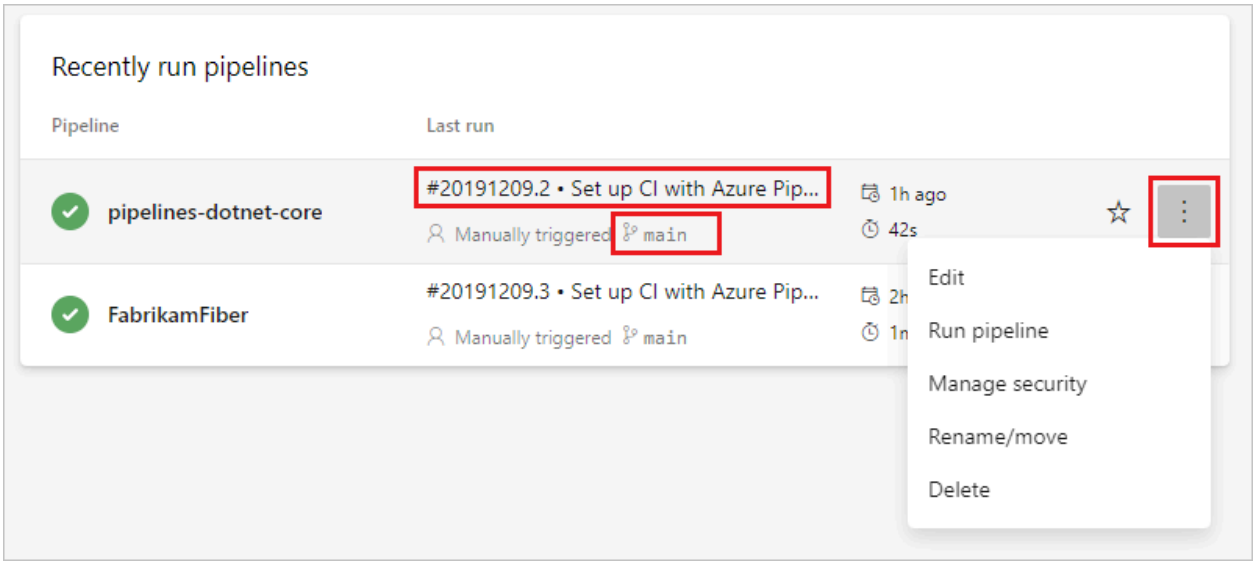


From the pipelines landing page you can view pipelines and pipeline runs, create and import pipelines, manage security, and drill down into pipeline and run details.

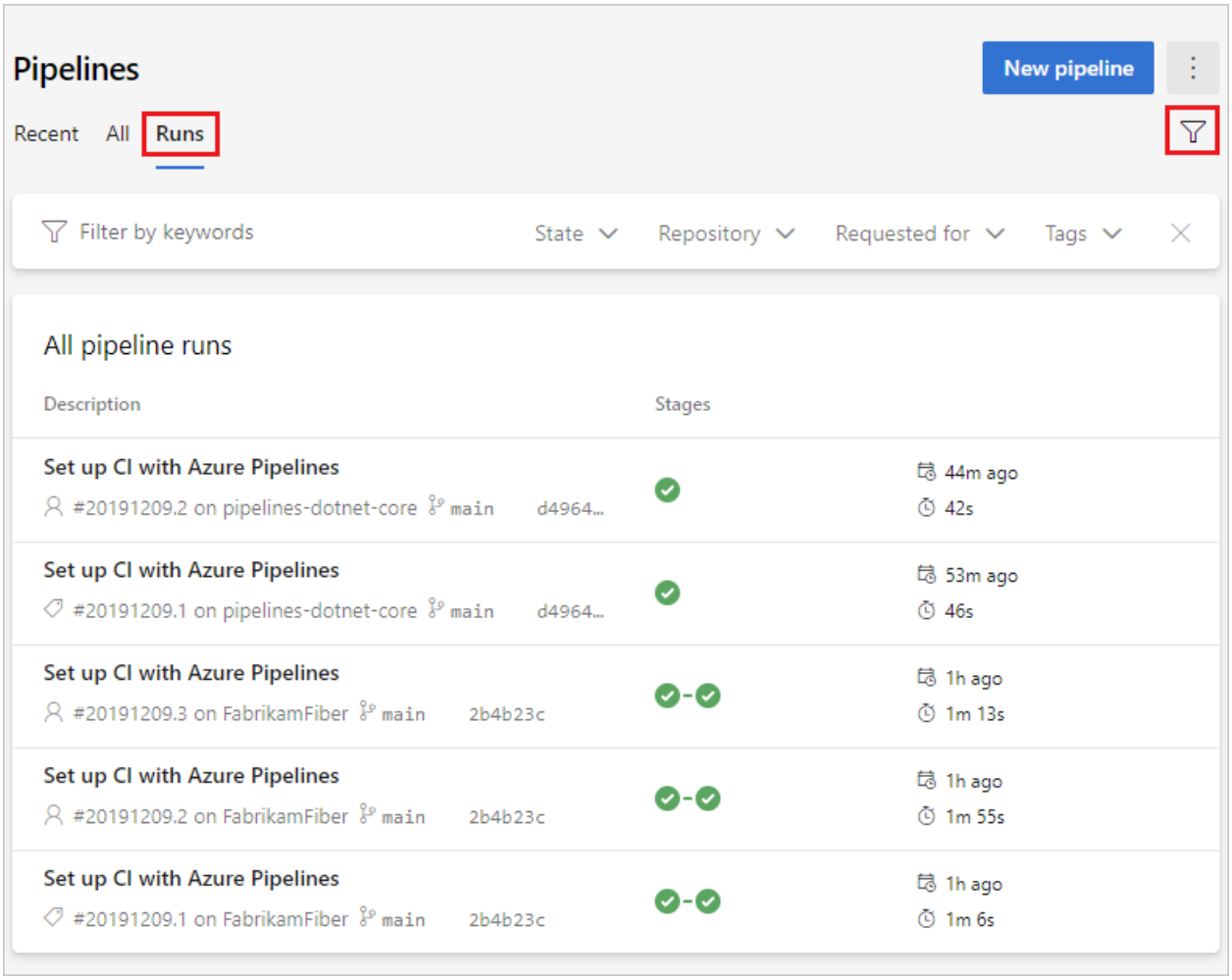
Choose **Recent** to view recently run pipelines (the default view), or choose **All** to view all pipelines.



Select a pipeline to manage that pipeline and view the runs. Select the build number for the last run to view the results of that build, select the branch name to view the branch for that run, or select the context menu to run the pipeline and perform other management actions.

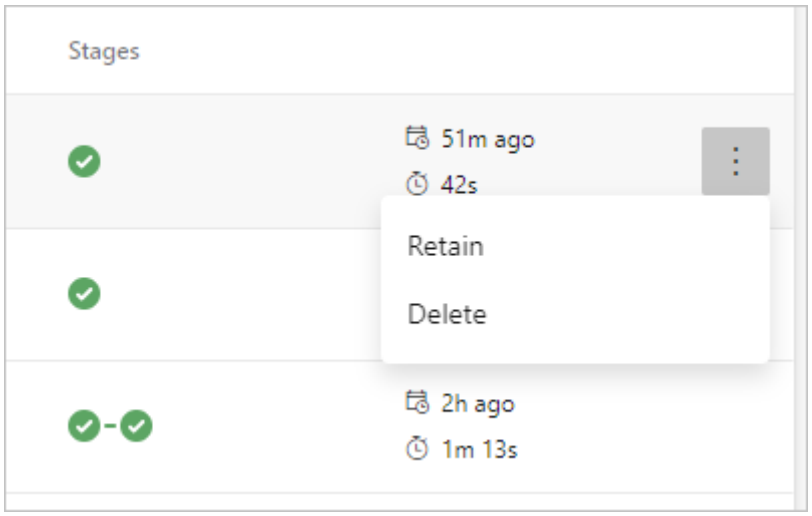


Select **Runs** to view all pipeline runs. You can optionally filter the displayed runs.



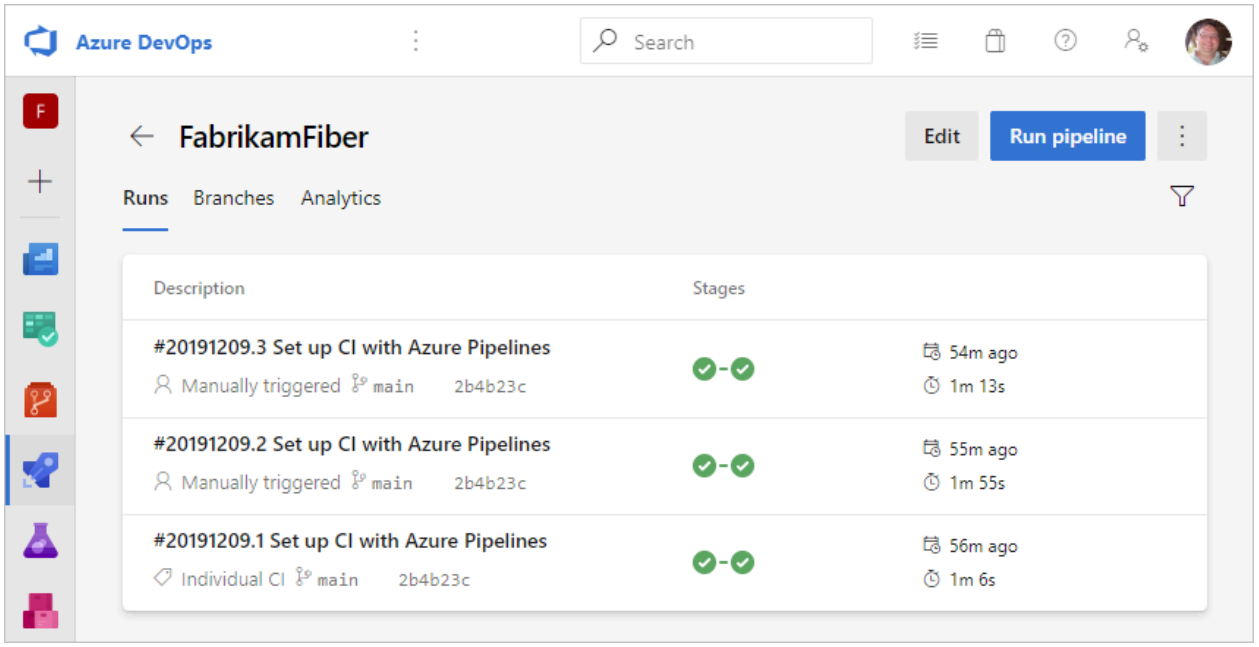
Select a pipeline run to view information about that run.

You can choose to **Retain** or **Delete** a run from the context menu. For more information on run retention, see Build and release retention policies.



View pipeline details

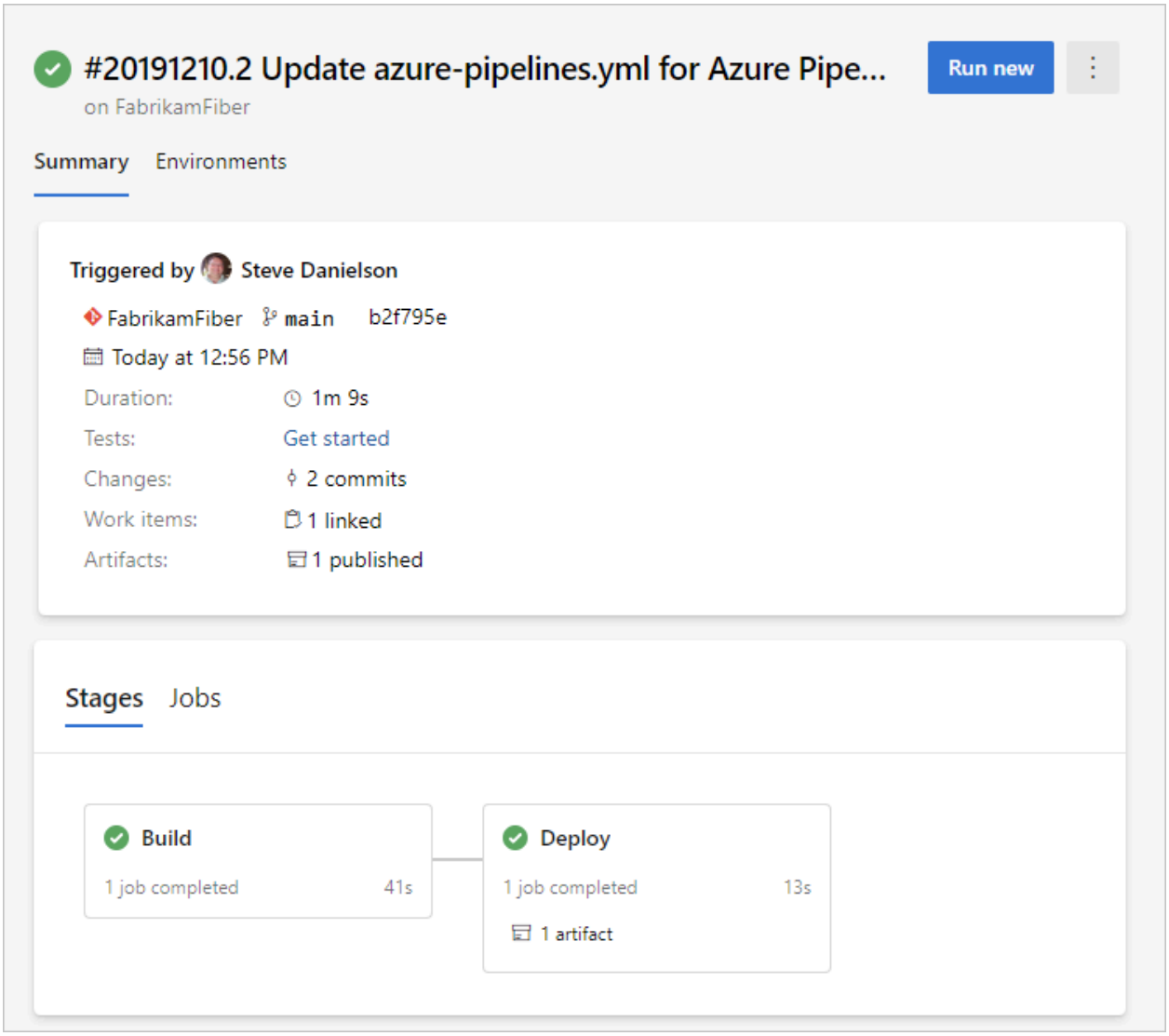
The details page for a pipeline allows you to view and manage that pipeline.



Choose **Edit** to edit your pipeline. For more information, see YAML pipeline editor. You can also edit your pipeline by modifying the **azure-pipelines.yml** file directly in the repository that hosts the pipeline.

View pipeline run details

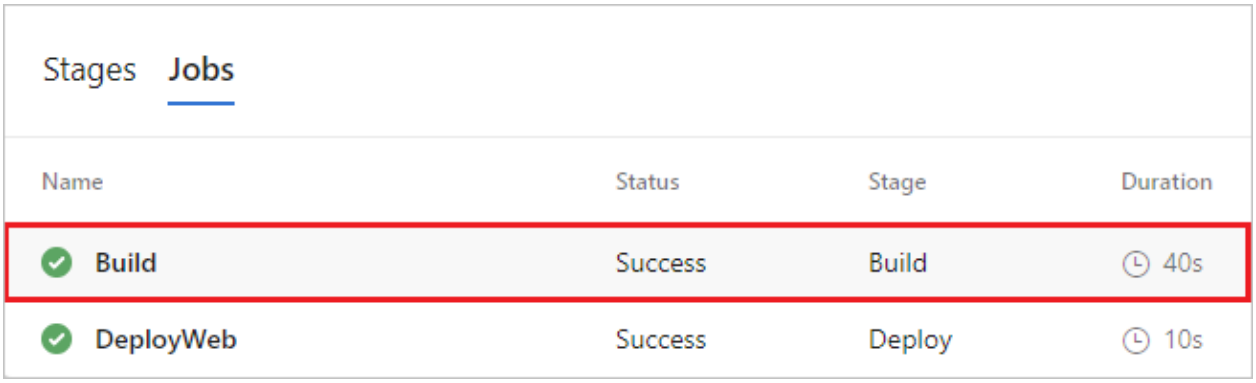
From the pipeline run summary you can view the status of your run, both while it is running and when it is complete.



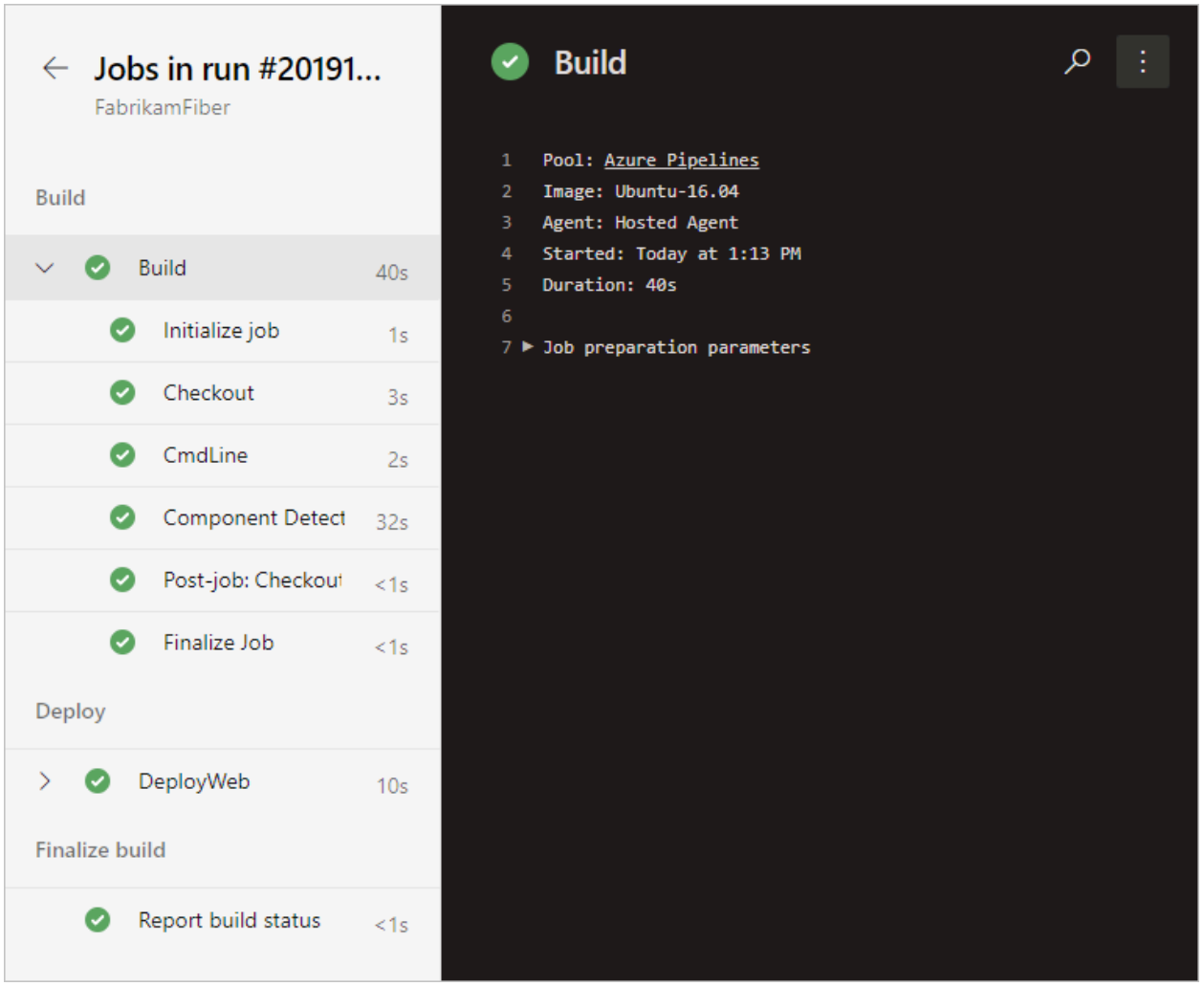
From the summary pane you can view job and stage details, download artifacts, and navigate to linked commits, test results, and work items.

Jobs and stages

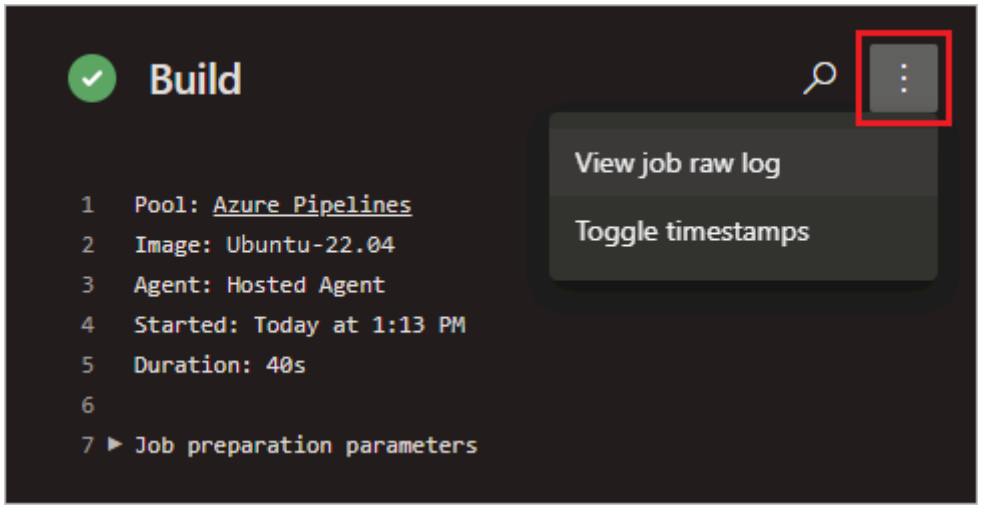
The jobs pane displays an overview of the status of your stages and jobs. This pane may have multiple tabs depending on whether your pipeline has stages and jobs, or just jobs. In this example, the pipeline has two stages named **Build** and **Deploy**. You can drill down into the pipeline steps by choosing the job from either the **Stages** or **Jobs** pane.



Choose a job to see the steps for that job.

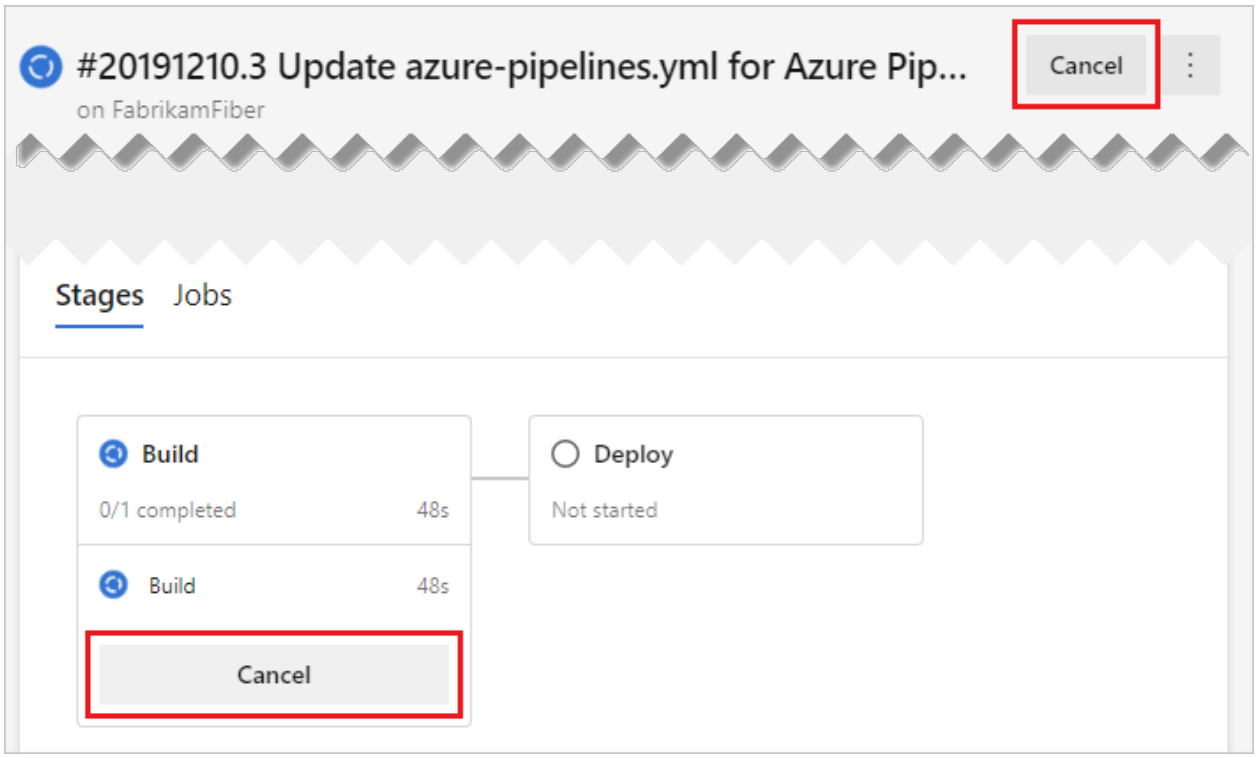


From the steps view, you can review the status and details of each step. From the **More actions** ⋮ you can toggle timestamps or view a raw log of all steps in the pipeline.



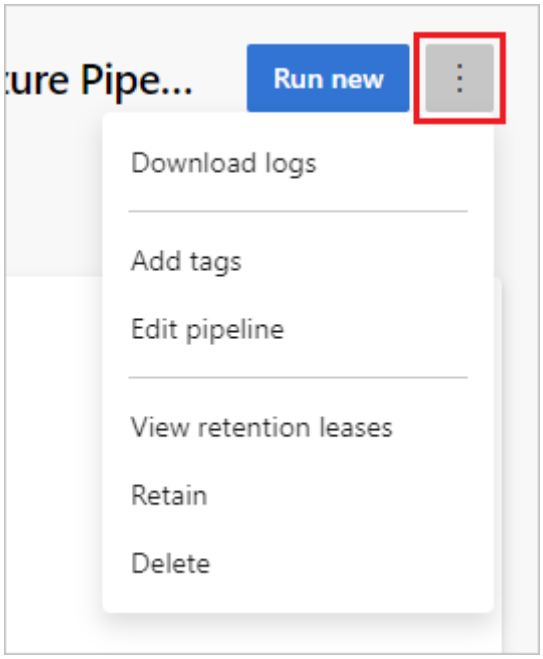
Cancel and re-run a pipeline

If the pipeline is running, you can cancel it by choosing **Cancel**. If the run has completed, you can re-run the pipeline by choosing **Run new**.



Pipeline run more actions menu

From the **More actions** ⋮ menu you can download logs, add tags, edit the pipeline, delete the run, and configure retention for the run.



> ⓘ Note
>
> You can't delete a run if the run is retained. If you don't see **Delete**, choose **Stop retaining run**, and then delete the run. If you see both **Delete** and **View retention releases**, one or more configured retention policies still apply to your run. Choose

View retention releases, delete the policies (only the policies for the selected run are removed), and then delete the run.

Add a status badge to your repository

Many developers like to show that they're keeping their code quality high by displaying a status badge in their repo.

To copy the status badge to your clipboard:

1. In Azure Pipelines, go to the **Pipelines** page to view the list of pipelines. Select the pipeline you created in the previous section.
2. Select ⋮, and then select **Status badge**.
3. Select **Status badge**.
4. Copy the sample Markdown from the Sample markdown section.

Now with the badge Markdown in your clipboard, take the following steps in GitHub:

1. Go to the list of files and select `Readme.md`. Select the pencil icon to edit.
2. Paste the status badge Markdown at the beginning of the file.
3. Commit the change to the `main` branch.
4. Notice that the status badge appears in the description of your repository.

To configure anonymous access to badges for private projects:

1. Navigate to **Project Settings** in the bottom left corner of the page
2. Open the **Settings** tab under **Pipelines**
3. Toggle the **Disable anonymous access to badges** slider under **General**

> ⓘ **Note**
>
> Even in a private project, anonymous badge access is enabled by default. With anonymous badge access enabled, users outside your organization might be able

to query information such as project names, branch names, job names, and build status through the badge status API.

Because you just changed the `Readme.md` file in this repository, Azure Pipelines automatically builds your code, according to the configuration in the `azure-pipelines.yml` file at the root of your repository. Back in Azure Pipelines, observe that a new run appears. Each time you make an edit, Azure Pipelines starts a new run.

Next steps

You learned how to create your first pipeline in Azure. Now, Learn more about configuring pipelines in the language of your choice:

- .NET Core
- Go
- Java
- Node.js
- Python
- Containers

Or, you can proceed to customize the pipeline you created.

To run your pipeline in a container, see Container jobs.

For details about building GitHub repositories, see Build GitHub repositories.

To learn how to publish your Pipeline Artifacts, see Publish Pipeline Artifacts.

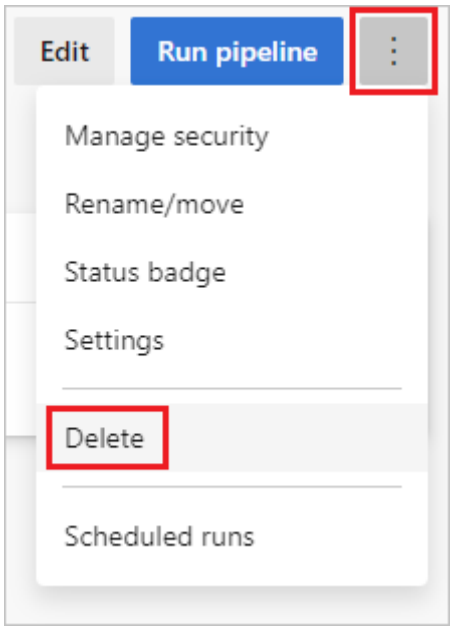
To find out what else you can do in YAML pipelines, see YAML schema reference.

Clean up

If you created any test pipelines, they're easy to delete when you finish with them.

Azure Pipelines UI

To delete a pipeline, navigate to the summary page for that pipeline, and choose **Delete** from the ... menu at the top-right of the page. Type the name of the pipeline to confirm, and choose **Delete**.



FAQ

Where can I read articles about DevOps and CI/CD?

What is Continuous Integration?

What is Continuous Delivery?

What is DevOps?

What version control system can I use?

When you're ready to get going with CI/CD for your app, you can use the version control system of your choice:

- Clients
  - Visual Studio Code for Windows, macOS, and Linux
  - Visual Studio with Git for Windows or Visual Studio for Mac
  - Eclipse
  - Xcode
  - IntelliJ
  - Command line
- Services
  - Azure Pipelines
  - Git service providers such as Azure Repos Git, GitHub, and Bitbucket Cloud
  - Subversion

How can I delete a pipeline?

To delete a pipeline, navigate to the summary page for that pipeline, and choose **Delete** from the **...** menu in the top-right of the page. Type the name of the pipeline to confirm, and choose **Delete**.

What else can I do when I queue a build?

You can queue builds automatically or manually.

When you manually queue a build, you can, for a single run of the build:

- Specify the pool into which the build goes.
- Add and modify some variables.
- Add demands.
- In a Git repository
  - Build a branch or a tag.
  - Build a commit.

Where can I learn more about pipeline settings?

To learn more about pipeline settings, see:

- Getting sources
- Tasks
- Variables
- Triggers
- Retention
- History

How do I programmatically create a build pipeline?

REST API Reference: Create a build pipeline

> **Note**
>
> You can also manage builds and build pipelines from the command line or scripts using the **Azure Pipelines CLI**.

Feedback

Was this page helpful? Yes No

Provide product feedback

ASP.NET Core Module (ANCM) for IIS

Article • 09/27/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

The ASP.NET Core Module (ANCM) is a native IIS module that plugs into the IIS pipeline, allowing ASP.NET Core applications to work with IIS. Run ASP.NET Core apps with IIS by either:

- Hosting an ASP.NET Core app inside of the IIS worker process (`w3wp.exe`), called the in-process hosting model.
- Forwarding web requests to a backend ASP.NET Core app running the Kestrel server, called the out-of-process hosting model.

There are trade-offs between each of the hosting models. By default, the in-process hosting model is used due to better performance and diagnostics.

For more information and configuration guidance, see the following topics:

- Web server implementations in ASP.NET Core

Install ASP.NET Core Module (ANCM)

The ASP.NET Core Module (ANCM) is installed with the .NET Core Runtime from the .NET Core Hosting Bundle. The ASP.NET Core Module is forward and backward compatible with in-support releases of .NET.

Breaking changes and security advisories are reported on the Announcements repo. Announcements can be limited to a specific version by selecting a **Label** filter.

Download the installer using the following link:

Current .NET Core Hosting Bundle installer (direct download)

For more information, including installing an earlier version of the module, see Hosting Bundle.

For a tutorial experience on publishing an ASP.NET Core app to an IIS server, see Publish an ASP.NET Core app to IIS.

Additional resources

- Host ASP.NET Core on Windows with IIS
- Deploy ASP.NET Core apps to Azure App Service
- ASP.NET Core Module reference source [default branch (main)] : Use the **Branch** drop down list to select a specific release (for example, `release/3.1`).
- IIS modules with ASP.NET Core

Troubleshoot ASP.NET Core on Azure App Service and IIS

Article • 09/27/2024

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, the ASP.NET Core project default server is Kestrel. Visual studio can be configured to use IIS Express. A *502.5 - Process Failure* or a *500.30 - Start Failure* that occurs when debugging locally with IIS Express can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:

    - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
    - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.
    - When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.

3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module (ANCM) for IIS.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

This error also may occur when the .NET Core Hosting Bundle isn't installed or is corrupted. Installing or repairing the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express) may fix the problem.

500.0 In-Process Handler Load Failure

The worker process fails. The app doesn't start.

An unknown error occurred loading ASP.NET Core Module components. Take one of the following actions:

- Contact Microsoft Support (select **Developer Tools** then **ASP.NET Core**).
- Ask a question on Stack Overflow.
- File an issue on our GitHub repository.

500.30 In-Process Startup Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core CLR in-process, but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

Common failure conditions:

- The app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine.
- Using Azure Key Vault, lack of permissions to the Key Vault. Check the access policies in the targeted Key Vault to ensure that the correct permissions are granted.

500.31 ANCM Failed to Find Native Dependencies

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core runtime in-process, but it fails to start. The most common cause of this startup failure is when the `Microsoft.NETCore.App` or `Microsoft.AspNetCore.App` runtime isn't installed. If the app is deployed to target ASP.NET Core 3.0 and that version doesn't exist on the machine, this error occurs. An example error message follows:

```
```

```
The specified framework 'Microsoft.NETCore.App', version '3.0.0' was not
found.
  - The following frameworks were found:
      2.2.1 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview5-27626-15 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27713-13 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27714-15 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27723-08 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
```

The error message lists all the installed .NET Core versions and the version requested by the app. To fix this error, either:

- Install the appropriate version of .NET Core on the machine.
- Change the app to target a version of .NET Core that's present on the machine.
- Publish the app as a self-contained deployment.

When running in development (the `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development`), the specific error is written to the HTTP response. The cause of a process startup failure is also found in the Application Event Log.

500.32 ANCM Failed to Load dll

The worker process fails. The app doesn't start.

The most common cause for this error is that the app is published for an incompatible processor architecture. If the worker process is running as a 32-bit app and the app was published to target 64-bit, this error occurs.

To fix this error, either:

- Republish the app for the same processor architecture as the worker process.
- Publish the app as a framework-dependent deployment.

500.33 ANCM Request Handler Load Failure

The worker process fails. The app doesn't start.

The app didn't reference the `Microsoft.AspNetCore.App` framework. Only apps targeting the `Microsoft.AspNetCore.App` framework can be hosted by the ASP.NET Core Module.

To fix this error, confirm that the app is targeting the `Microsoft.AspNetCore.App` framework. Check the `.runtimeconfig.json` to verify the framework targeted by the app.

500.34 ANCM Mixed Hosting Models Not Supported

The worker process can't run both an in-process app and an out-of-process app in the same process.

To fix this error, run apps in separate IIS application pools.

500.35 ANCM Multiple In-Process Applications in same Process

The worker process can't run multiple in-process apps in the same process.

To fix this error, run apps in separate IIS application pools.

500.36 ANCM Out-Of-Process Handler Load Failure

The out-of-process request handler, *aspnetcorev2_outofprocess.dll*, isn't next to the *aspnetcorev2.dll* file. This indicates a corrupted installation of the ASP.NET Core Module.

To fix this error, repair the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express).

500.37 ANCM Failed to Start Within Startup Time Limit

ANCM failed to start within the provided startup time limit. By default, the timeout is 120 seconds.

This error can occur when starting a large number of apps on the same machine. Check for CPU/Memory usage spikes on the server during startup. You may need to stagger the startup process of multiple apps.

500.38 ANCM Application DLL Not Found

ANCM failed to locate the application DLL, which should be next to the executable.

This error occurs when hosting an app packaged as a single-file executable using the in-process hosting model. The in-process model requires that the ANCM load the .NET Core app into the existing IIS process. This scenario isn't supported by the single-file

deployment model. Use **one** of the following approaches in the app's project file to fix this error:

1. Disable single-file publishing by setting the `PublishSingleFile` MSBuild property to `false`.
2. Switch to the out-of-process hosting model by setting the `AspNetCoreHostingModel` MSBuild property to `OutOfProcess`.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.

2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:

    - If deploying a 32-bit (x86) app, set the value to `True`.
    - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Failed to start application (ErrorCode '0x800701b1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x800701b1'.
```

The app failed to start because a Windows Service failed to load.

One common service that needs to be enabled is the "null" service. The following command enables the `null` Windows Service:

Windows Command Prompt

```
sc.exe start null
```

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

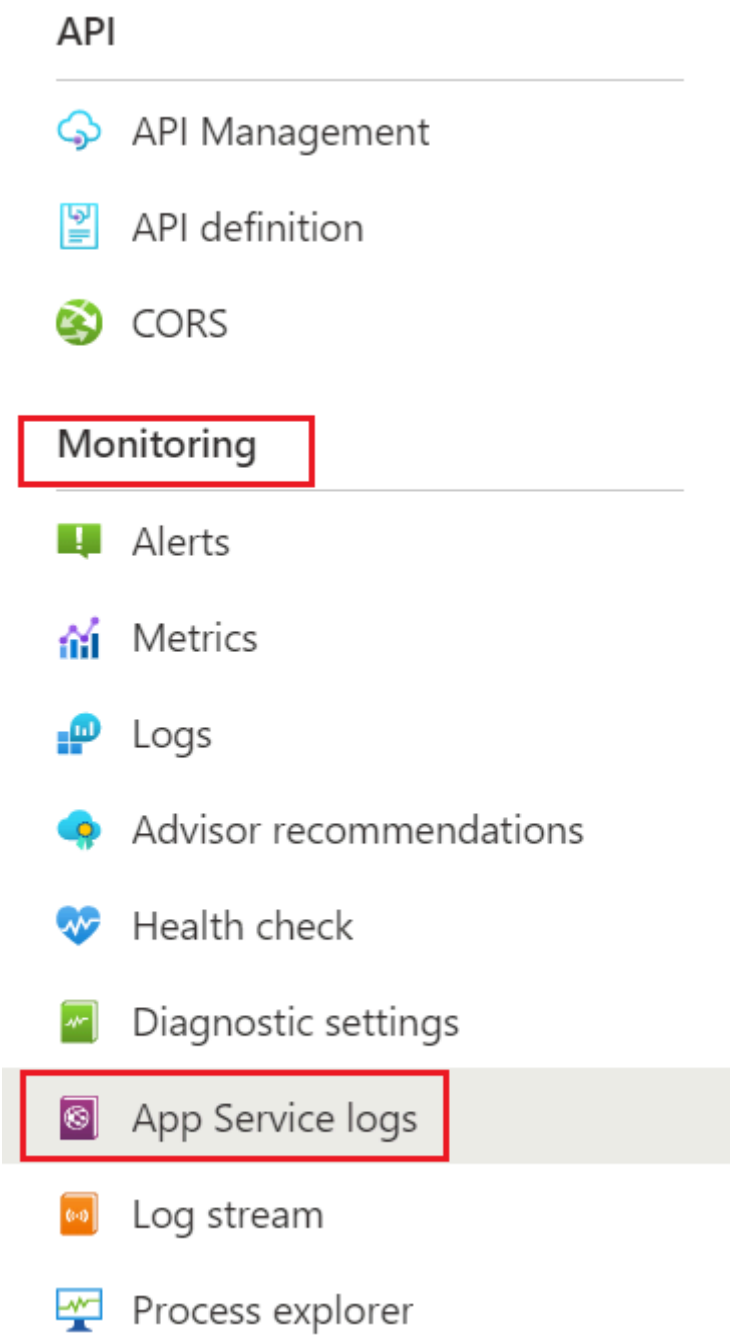
> **Important**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Azure App Services Log stream

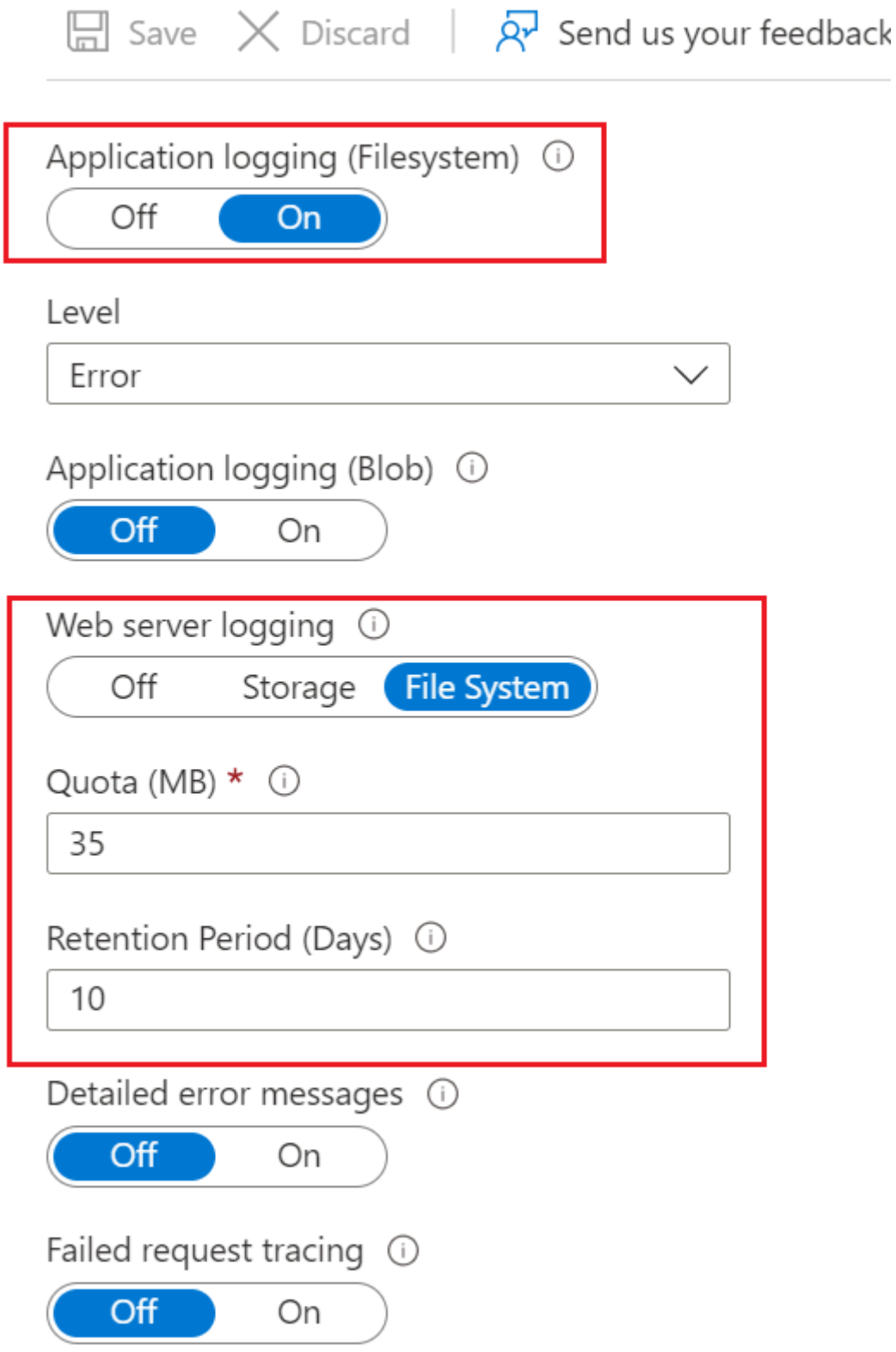
The Azure App Services Log streams logging information as it occurs. To view streaming logs:

1. In the Azure portal, open the app in **App Services**.

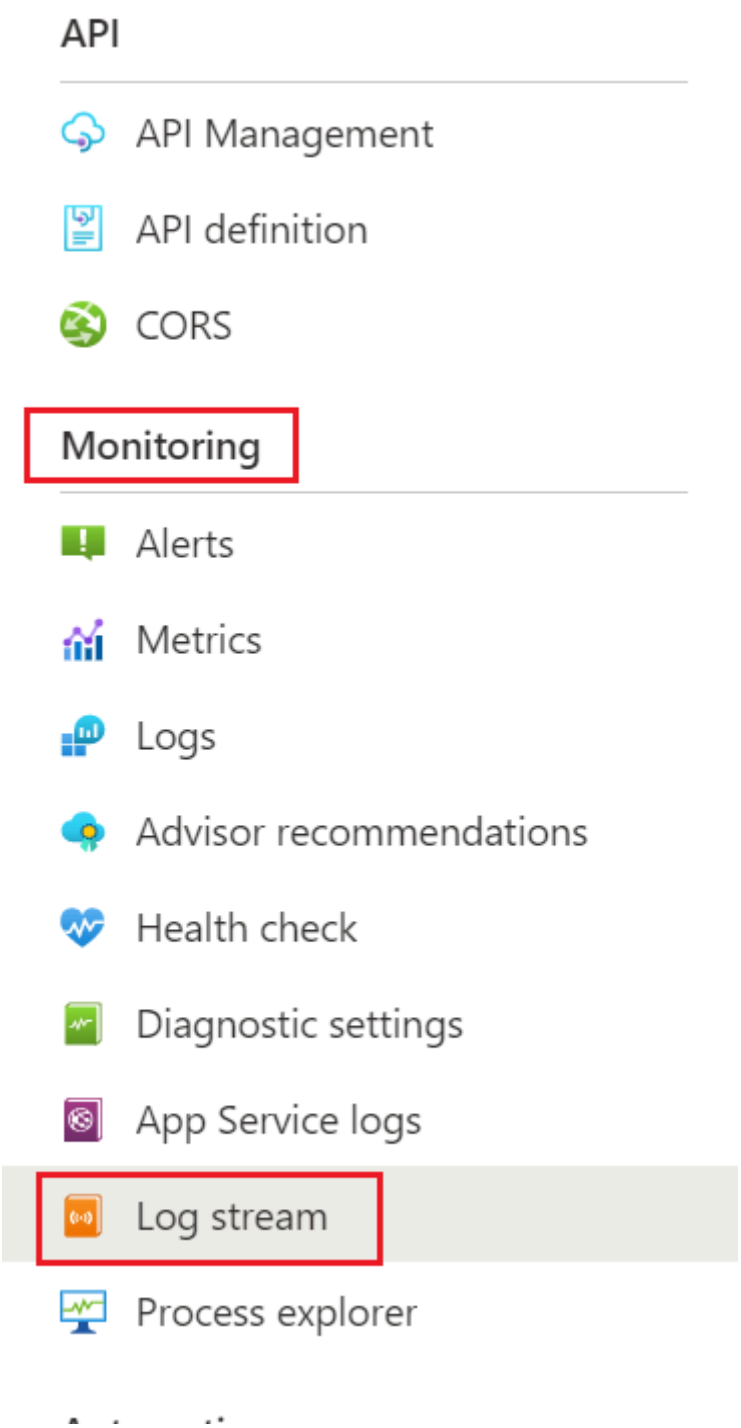
2. In the left pane, navigate to **Monitoring** > **App Service Logs**.



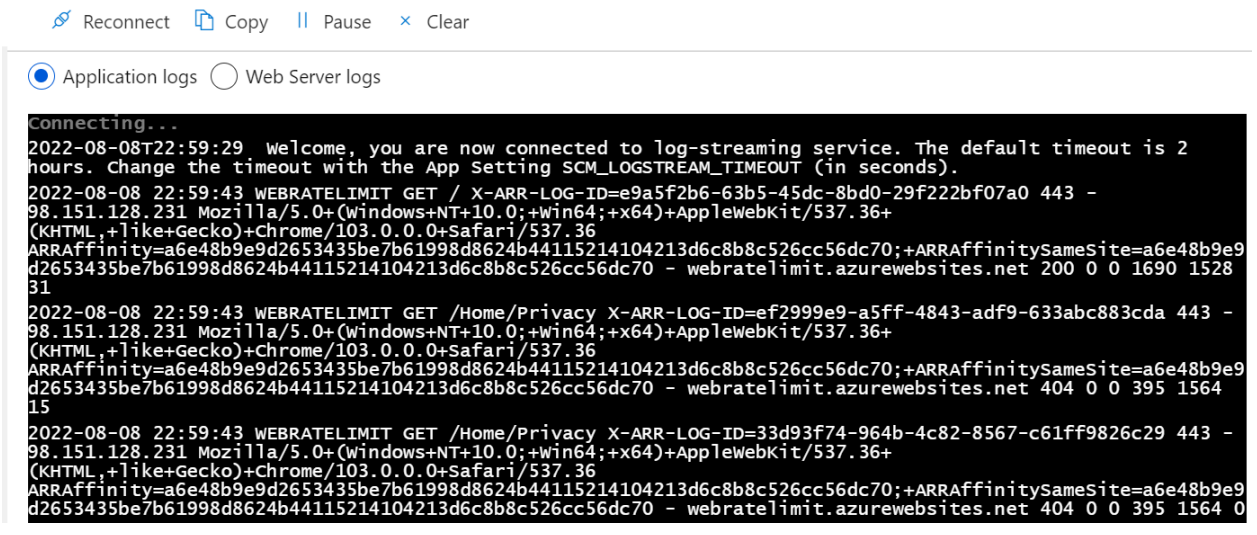
3. Select **File System** for **Web Server Logging**. Optionally enable **Application logging**.



4. In the left pane, navigate to **Monitoring** > **Log stream**, and then select **Application logs** or **Web Server Logs**.



The following images shows the application logs output:



Streaming logs have some latency and might not display immediately.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go** → button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the `eventlog.xml` file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go** → button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:

- If the app is a framework-dependent deployment:

  .NET CLI

  ```
  dotnet .\{ASSEMBLY NAME}.dll
  ```

- If the app is a self-contained deployment:

  Console

  ```
  {ASSEMBLY NAME}.exe
  ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:

  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

- If the app is a self-contained deployment:

  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. In the Azure Portal, navigate to the web app.
2. In the **App Service** blade, enter **kudu** in the search box.
3. Select **Advanced Tools** > **Go**.
4. Select **Debug console** > **CMD**.
5. Navigate to *site/wwwroot*
6. Select the pencil icon to edit the *web.config* file.
7. In the `<aspNetCore />` element, set `stdoutLogEnabled="true"` and select **Save**.

Disable stdout logging when troubleshooting is complete by setting `stdoutLogEnabled="false"`.

For more information, see ASP.NET Core Module (ANCM) for IIS.

ASP.NET Core Module debug log (Azure App Service)

The ASP.NET Core Module debug log provides additional, deeper logging from the ASP.NET Core Module. To enable and view stdout logs:

1. To enable the enhanced diagnostic log, perform either of the following:

- Follow the instructions in Enhanced diagnostic logs to configure the app for an enhanced diagnostic logging. Redeploy the app.
- Add the `<handlerSettings>` shown in Enhanced diagnostic logs to the live app's *web.config* file using the Kudu console:
  a. Open **Advanced Tools** in the **Development Tools** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
  b. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
  c. Open the folders to the path **site** > **wwwroot**. Edit the *web.config* file by selecting the pencil button. Add the `<handlerSettings>` section as shown in Enhanced diagnostic logs. Select the **Save** button.

2. Open **Advanced Tools** in the **Development Tools** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
3. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
4. Open the folders to the path **site** > **wwwroot**. If you didn't supply a path for the *aspnetcore-debug.log* file, the file appears in the list. If you supplied a path, navigate to the location of the log file.
5. Open the log file with the pencil button next to the file name.

Disable debug logging when troubleshooting is complete:

To disable the enhanced debug log, perform either of the following:

- Remove the `<handlerSettings>` from the *web.config* file locally and redeploy the app.
- Use the Kudu console to edit the *web.config* file and remove the `<handlerSettings>` section. Save the file.

For more information, see Log creation and redirection with the ASP.NET Core Module.

> ⚠ **Warning**
>
> Failure to disable the debug log can lead to app or server failure. There's no limit on log file size. Only use debug logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or nonresponsive app (Azure App Service)

When an app responds slowly or doesn't respond to a request, see Troubleshoot slow web app performance issues in Azure App Service.

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.

5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`).

`stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.

4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module (ANCM) for IIS.

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (IIS)

Add the following handler settings to the app's *web.config* file to enable ASP.NET Core Module debug log:

XML

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

Confirm that the path specified for the log exists and that the app pool's identity has write permissions to the location.

For more information, see Log creation and redirection with the ASP.NET Core Module.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

XML

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development"
/>
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or non-responding app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.

2. Run the EnableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     Console

     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     Console

     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.

4. After the crash has occurred, run the DisableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     Console

     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     Console

     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> ⚠ **Warning**
>
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App doesn't respond, fails during startup, or runs normally

When an app stops responding but doesn't crash, fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Debug .NET and ASP.NET Core source code with Visual Studio
- Troubleshoot and debug ASP.NET Core projects
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module (ANCM) for IIS

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

Common error troubleshooting for Azure App Service and IIS with ASP.NET Core

Article • 07/31/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This topic describes the most common errors and provides troubleshooting advice when hosting ASP.NET Core apps on Azure Apps Service and IIS.

See Troubleshoot ASP.NET Core on Azure App Service and IIS information on common app startup errors and instructions on how to diagnose errors.

Collect the following information:

- Browser behavior such as status code and error message.
- Application Event Log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS

    1. Select **Start** on the **Windows** menu, type *Event Viewer*, and press **Enter**.
    2. After the **Event Viewer** opens, expand **Windows Logs** > **Application** in the sidebar.

- ASP.NET Core Module stdout and debug log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS: Follow the instructions in the Log creation and redirection and Enhanced diagnostic logs sections of the ASP.NET Core Module topic.

Compare error information to the following common errors. If a match is found, follow the troubleshooting advice.

The list of errors in this topic isn't exhaustive. If you encounter an error not listed here, open a new issue using the **Content feedback** button at the bottom of this topic with detailed instructions on how to reproduce the error.

> ⓘ **Important**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any app pool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Hosting bundle. Select **Repair** when the installer is run.

Missing site extension, 32-bit (x86) and 64-bit (x64) site extensions installed, or wrong process bitness set

Applies to apps hosted by Azure App Services.

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found. Failed to start application '/LM/W3SVC/1416782824/ROOT', ErrorCode '0x8000ffff'.

- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff. Could not find inprocess request handler. It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.

Troubleshooting:

- If running the app on a preview runtime, install either the 32-bit (x86) **or** 64-bit (x64) site extension that matches the bitness of the app and the app's runtime version. **Don't install both extensions or multiple runtime versions of the extension.**
  - ASP.NET Core {RUNTIME VERSION} (x86) Runtime
  - ASP.NET Core {RUNTIME VERSION} (x64) Runtime

  Restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and both the 32-bit (x86) and 64-bit (x64) site extensions are installed, uninstall the site extension that doesn't match the bitness of the app. After removing the site extension, restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and the site extension's bitness matches that of the app, confirm that the preview site extension's *runtime version* matches the app's runtime version.
- Confirm that the app's **Platform** in **Application Settings** matches the bitness of the app.

For more information, see Deploy ASP.NET Core apps to Azure App Service.

An x86 app is deployed but the app pool isn't enabled for 32-bit apps

- **Browser:** HTTP Error 500.30 - ANCM In-Process Start Failure

- **Application Log:** Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' hit unexpected managed exception, exception code = '0xe0434352'. Please check the stderr logs for more information. Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' failed to load clr and managed application. CLR worker thread exited prematurely

- **ASP.NET Core Module stdout Log:** The log file is created but empty.

- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8007023e

This scenario is trapped by the SDK when publishing a self-contained app. The SDK produces an error if the RID doesn't match the platform target (for example, `win10-x64` RID with `<PlatformTarget>x86</PlatformTarget>` in the project file).

Troubleshooting:

For an x86 framework-dependent deployment (`<PlatformTarget>x86</PlatformTarget>`), enable the IIS app pool for 32-bit apps. In IIS Manager, open the app pool's **Advanced Settings** and set **Enable 32-Bit Applications** to **True**.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure

- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ', ErrorCode = '0x80004005 : ff.

- **ASP.NET Core Module stdout Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{ASSEMBLY}.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED **--OR--** Unable to connect
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm the correct URI endpoint for the app is in use. Check the bindings.
- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that the proper role and features are enabled. See IIS Configuration.

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied **--OR--** 403.14 Forbidden - The Web server is configured to not list the contents of this directory.
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS website **Physical path**.

Incorrect role, ASP.NET Core Module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid. --**OR**-- This page can't be displayed
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm that the proper role is enabled. See IIS Configuration.
- Open **Programs & Features** or **Apps & features** and confirm that **Windows Server Hosting** is installed. If **Windows Server Hosting** isn't present in the list of installed programs, download and install the .NET Core Hosting Bundle.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.
- Make sure that the **Application Pool** > **Process Model** > **Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.
- If you uninstalled the ASP.NET Core Hosting Bundle and installed an earlier version of the hosting bundle, the *applicationHost.config* file doesn't include a section for the ASP.NET Core Module. Open *applicationHost.config* at *%windir%/System32/inetsrv/config* and find the `<configuration><configSections>` `<sectionGroup name="system.webServer">` section group. If the section for the ASP.NET Core Module is missing from the section group, add the section element:

  ```
  XML
  <section name="aspNetCore" overrideModeDefault="Allow" />
  ```

  Alternatively, install the latest version of the ASP.NET Core Hosting Bundle. The latest version is backwards-compatible with supported ASP.NET Core apps.

Incorrect processPath, missing PATH variable, Hosting Bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"{...}" ', ErrorCode = '0x80070002 : 0. Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed to start application '/LM/W3SVC/2/ROOT', ErrorCode '0x8007023e'.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** Event Log: 'Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed HRESULT returned: 0x8007023e

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's `dotnet` for a framework-dependent deployment (FDD) or `.\{ASSEMBLY}.exe` for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the app pool. Confirm that the app pool user identity has access to the *C:\Program Files\dotnet* directory. Confirm that there are no deny rules configured for the app pool user identity on the *C:\Program Files\dotnet* and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Hosting Bundle installer** on the system.

Current .NET Core Hosting Bundle installer (direct download)

For more information, see Install the .NET Core Hosting Bundle.

If a specific runtime is required, download the runtime from the .NET Downloads page and install it on the system. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed HRESULT returned: 0x8000ffff

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

- Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's either (a) `.\{ASSEMBLY}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="{ARGUMENT_1}, {ARGUMENT_2}, ... {ARGUMENT_X}"`) for a self-contained deployment (SCD).

Missing .NET Core shared framework

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.

Failed to start application '/LM/W3SVC/5/ROOT', ErrorCode '0x8000ffff'.

- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.
- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8000ffff

Troubleshooting:

For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Confirm that the Application Pool isn't in the *Stopped* state.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.
- **ASP.NET Core Module Debug Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

Troubleshooting:

Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section or that the sub-app doesn't inherit the parent app's handlers.

The parent app's `<system.webServer>` section of *web.config* is placed inside of a `<location>` element. The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the parent app. For more information, see ASP.NET Core Module (ANCM) for IIS.

stdout log path incorrect

- **Browser:** The app responds normally.
- **Application Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module debug Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core

Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.

Troubleshooting:

- The `stdoutLogFile` path specified in the `<aspNetCore>` element of *web.config* doesn't exist. For more information, see ASP.NET Core Module: Log creation and redirection.
- The app pool user doesn't have write access to the stdout log path.

Application configuration general issue

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure **--OR--** HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Variable
- **ASP.NET Core Module stdout Log:** The log file is created but empty or created with normal entries until the point of the app failing.
- **ASP.NET Core Module Debug Log:** Variable

Troubleshooting:

The process failed to start, most likely due to an app configuration or programming issue.

For more information, see the following topics:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Troubleshoot and debug ASP.NET Core projects

Publish an ASP.NET Core app to IIS

Article • 07/01/2024

This tutorial shows how to host an ASP.NET Core app on an IIS server.

This tutorial covers the following subjects:

- ✓ Install the .NET Core Hosting Bundle on Windows Server.
- ✓ Create an IIS site in IIS Manager.
- ✓ Deploy an ASP.NET Core app.

Prerequisites

- .NET Core SDK installed on the development machine.
- Windows Server configured with the **Web Server (IIS)** server role. If your server isn't configured to host websites with IIS, follow the guidance in the *IIS configuration* section of the Host ASP.NET Core on Windows with IIS article and then return to this tutorial.

> ⚠ **Warning**
>
> **IIS configuration and website security involve concepts that aren't covered by this tutorial.** Consult the IIS guidance in the Microsoft IIS documentation and the ASP.NET Core article on hosting with IIS before hosting production apps on IIS.
>
> Important scenarios for IIS hosting not covered by this tutorial include:
>
> - Creation of a registry hive for ASP.NET Core Data Protection
> - Configuration of the app pool's Access Control List (ACL)
> - To focus on IIS deployment concepts, this tutorial deploys an app without HTTPS security configured in IIS. For more information on hosting an app enabled for HTTPS protocol, see the security topics in the Additional resources section of this article. Further guidance for hosting ASP.NET Core apps is provided in the Host ASP.NET Core on Windows with IIS article.

Install the .NET Core Hosting Bundle

Install the *.NET Core Hosting Bundle* on the IIS server. 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.

Download the installer using the following link:

Current .NET Core Hosting Bundle installer (direct download)

1. Run the installer on the IIS server.
2. Restart the server or execute `net stop was /y` followed by `net start w3svc` in a command shell.

Create the IIS site

1. On the IIS server, 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 that you created. Provide the **Binding** configuration and create the website by selecting **OK**.

   > ⚠ **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 RFC 9110: HTTP Semantics (Section 7.2. Host and :authority) for more information.

4. 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.

Create an ASP.NET Core Razor Pages app

Follow the Get started with ASP.NET Core tutorial to create a Razor Pages app.

Publish and deploy the app

Publish an app means to produce a compiled app that can be hosted by a server. *Deploy an app* means to move the published app to a hosting system. The publish step is handled by the .NET Core SDK, while the deployment step can be handled by a variety of approaches. This tutorial adopts the *folder* deployment approach, where:

- The app is published to a folder.
- The folder's contents are moved to the IIS site's folder (the **Physical path** to the site in IIS Manager).

Visual Studio

1. Right-click on the project in **Solution Explorer** and select **Publish**.
2. In the **Pick a publish target** dialog, select the **Folder** publish option.
3. Set the **Folder or File Share** path.

    - If you created a folder for the IIS site that's available on the development machine as a network share, provide the path to the share. The current user must have write access to publish to the share.
    - If you're unable to deploy directly to the IIS site folder on the IIS server, publish to a folder on removable media and physically move the published app to the IIS site folder on the server, which is the site's **Physical path** in IIS Manager. Move the contents of the `bin/Release/{TARGET FRAMEWORK}/publish` folder to the IIS site folder on the server, which is the site's **Physical path** in IIS Manager.

4. Select the **Publish** button.

Browse the website

The app is accessible in a browser after it receives the first request. Make a request to the app at the endpoint binding that you established in IIS Manager for the site.

Next steps

In this tutorial, you learned how to:

- ✓ Install the .NET Core Hosting Bundle on Windows Server.
- ✓ Create an IIS site in IIS Manager.
- ✓ Deploy an ASP.NET Core app.

To learn more about hosting ASP.NET Core apps on IIS, see the IIS Overview article:

Host ASP.NET Core on Windows with IIS

Additional resources

Articles in the ASP.NET Core documentation set

- ASP.NET Core Module (ANCM) for IIS
- ASP.NET Core directory structure
- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Enforce HTTPS in ASP.NET Core
- WebSockets on IIS

Articles pertaining to ASP.NET Core app deployment

- Publish an ASP.NET Core app to Azure with Visual Studio
- Publish an ASP.NET Core app to Azure with Visual Studio Code
- Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment

Articles on IIS HTTPS configuration

- Configuring SSL in IIS Manager
- How to Set Up SSL on IIS

Articles on IIS and Windows Server

- The Official Microsoft IIS Site
- Windows Server technical content library

Deployment resources for IIS administrators

- IIS documentation
- Getting Started with the IIS Manager in IIS
- .NET Core application deployment
- ASP.NET Core Module (ANCM) for IIS
- ASP.NET Core directory structure
- IIS modules with ASP.NET Core
- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
- Sticky sessions with Application Request Routing

Host ASP.NET Core in a Windows Service

Article • 09/27/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

An ASP.NET Core app can be hosted on Windows as a Windows Service without using IIS. When hosted as a Windows Service, the app automatically starts after server reboots.

Prerequisites

- ASP.NET Core SDK 7.0 or later
- PowerShell 6.2 or later

Worker Service template

The ASP.NET Core Worker Service template provides a starting point for writing long running service apps. To use the template as a basis for a Windows Service app:

1. Create a Worker Service app from the .NET Core template.
2. Install the NuGet package Microsoft.Extensions.Hosting.WindowsServices.
3. Follow the guidance in the App configuration section to update the Worker Service app so that it can run as a Windows Service.

Visual Studio

1. Create a new project.
2. Select **Worker Service**. Select **Next**.
3. Provide a project name in the **Project name** field or accept the default project name. Select **Create**.
4. In the **Create a new Worker service** dialog, select **Create**.

App configuration

Update Program.cs to call AddWindowsService. When the app is running as a Windows Service, `AddWindowsService`:

- Sets the host lifetime to `WindowsServiceLifetime`.
- Sets the content root to AppContext.BaseDirectory. For more information, see the Current directory and content root section.
- Enables logging to the event log:
  - The application name is used as the default source name.
  - The default log level is *Warning* or higher for an app based on an ASP.NET Core template that calls `CreateDefaultBuilder` to build the host.
  - Override the default log level with the `Logging:EventLog:LogLevel:Default` key in `appsettings.json`/`appsettings.{Environment}.json` or other configuration provider.
  - Only administrators can create new event sources. When an event source can't be created using the application name, a warning is logged to the *Application* source and event logs are disabled.

Consider the following `ServiceA` class:

C#

```
namespace SampleApp.Services;

public class ServiceA : BackgroundService
{
    public ServiceA(ILoggerFactory loggerFactory)
    {
        Logger = loggerFactory.CreateLogger<ServiceA>();
    }

    public ILogger Logger { get; }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        Logger.LogInformation("ServiceA is starting.");

        stoppingToken.Register(() => Logger.LogInformation("ServiceA is
stopping."));

        while (!stoppingToken.IsCancellationRequested)
        {
            Logger.LogInformation("ServiceA is doing background work.");

            await Task.Delay(TimeSpan.FromSeconds(5), stoppingToken);
        }
```

```
        Logger.LogInformation("ServiceA has stopped.");
    }
}
```

The following `Program.cs` calls AddHostedService to register `ServiceA`:

C#

```
using SampleApp.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddWindowsService();
builder.Services.AddHostedService<ServiceA>();

var app = builder.Build();

app.MapRazorPages();

app.Run();
```

The following sample apps accompany this topic:

- Background Worker Service Sample: A non-web app sample based on the Worker Service template that uses hosted services for background tasks.
- Web App Service Sample: A Razor Pages web app sample that runs as a Windows Service with hosted services for background tasks.

For MVC guidance, see the articles under Overview of ASP.NET Core MVC and Migrate from ASP.NET Core 2.2 to 3.0.

Deployment type

For information and advice on deployment scenarios, see .NET Core application deployment.

SDK

For a web app-based service that uses the Razor Pages or MVC frameworks, specify the Web SDK in the project file:

XML

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

If the service only executes background tasks (for example, hosted services), specify the Worker SDK in the project file:

XML

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

Framework-dependent deployment (FDD)

Framework-dependent deployment (FDD) relies on the presence of a shared system-wide version of .NET Core on the target system. When the FDD scenario is adopted following the guidance in this article, the SDK produces an executable (*.exe*), called a *framework-dependent executable*.

If using the Web SDK, a *web.config* file, which is normally produced when publishing an ASP.NET Core app, is unnecessary for a Windows Services app. To disable the creation of the *web.config* file, add the `<IsTransformWebConfigDisabled>` property set to `true`.

XML

```
<PropertyGroup>
  <TargetFramework>net7.0</TargetFramework>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

Self-contained deployment (SCD)

Self-contained deployment (SCD) doesn't rely on the presence of a shared framework on the host system. The runtime and the app's dependencies are deployed with the app.

A Windows Runtime Identifier (RID) is included in the `<PropertyGroup>` that contains the target framework:

XML

```
<RuntimeIdentifier>win-x64</RuntimeIdentifier>
```

To publish for multiple RIDs:

- Provide the RIDs in a semicolon-delimited list.

- Use the property name <RuntimeIdentifiers> (plural).

For more information, see .NET Core RID Catalog.

Service user account

To create a user account for a service, use the New-LocalUser cmdlet from an administrative PowerShell 6 command shell.

On Windows 10 October 2018 Update (version 1809/build 10.0.17763) or later:

PowerShell

```
New-LocalUser -Name {SERVICE NAME}
```

On Windows OS earlier than the Windows 10 October 2018 Update (version 1809/build 10.0.17763):

Console

```
powershell -Command "New-LocalUser -Name {SERVICE NAME}"
```

Provide a strong password when prompted.

Unless the `-AccountExpires` parameter is supplied to the New-LocalUser cmdlet with an expiration DateTime, the account doesn't expire.

For more information, see Microsoft.PowerShell.LocalAccounts and Service User Accounts.

An alternative approach to managing users when using Active Directory is to use Managed Service Accounts. For more information, see Group Managed Service Accounts Overview.

Log on as a service rights

To establish *Log on as a service* rights for a service user account:

1. Open the Local Security Policy editor by running *secpol.msc*.
2. Expand the **Local Policies** node and select **User Rights Assignment**.
3. Open the **Log on as a service** policy.
4. Select **Add User or Group**.
5. Provide the object name (user account) using either of the following approaches:

a. Type the user account (`{DOMAIN OR COMPUTER NAME\USER}`) in the object name field and select **OK** to add the user to the policy.
b. Select **Advanced**. Select **Find Now**. Select the user account from the list. Select **OK**. Select **OK** again to add the user to the policy.

6. Select **OK** or **Apply** to accept the changes.

Create and manage the Windows Service

Create a service

Use PowerShell commands to register a service. From an administrative PowerShell 6 command shell, execute the following commands:

PowerShell

```
$acl = Get-Acl "{EXE PATH}"
$aclRuleArgs = "{DOMAIN OR COMPUTER NAME\USER}",
"Read,Write,ReadAndExecute", "ContainerInherit,ObjectInherit", "None",
"Allow"
$accessRule = New-Object
System.Security.AccessControl.FileSystemAccessRule($aclRuleArgs)
$acl.SetAccessRule($accessRule)
$acl | Set-Acl "{EXE PATH}"

New-Service -Name {SERVICE NAME} -BinaryPathName "{EXE FILE PATH} --
contentRoot {EXE FOLDER PATH}" -Credential "{DOMAIN OR COMPUTER NAME\USER}"
-Description "{DESCRIPTION}" -DisplayName "{DISPLAY NAME}" -StartupType
Automatic
```

- `{EXE PATH}`: Path of the app's executable on the host (for example, `d:\myservice`). Don't include the app's executable file name in the path. A trailing slash isn't required.
- `{DOMAIN OR COMPUTER NAME\USER}`: Service user account (for example, `Contoso\ServiceUser`).
- `{SERVICE NAME}`: Service name (for example, `MyService`).
- `{EXE FILE PATH}`: The app's full executable path (for example, `d:\myservice\myservice.exe`). Include the executable's file name with extension.
- `{EXE FOLDER PATH}`: The app's full executable folder path (for example `d:\myservice`).
- `{DESCRIPTION}`: Service description (for example, `My sample service`).
- `{DISPLAY NAME}`: Service display name (for example, `My Service`).

Start a service

Start a service with the following PowerShell 6 command:

```PowerShell
Start-Service -Name {SERVICE NAME}
```

The command takes a few seconds to start the service.

Determine a service's status

To check the status of a service, use the following PowerShell 6 command:

```PowerShell
Get-Service -Name {SERVICE NAME}
```

The status is reported as one of the following values:

- `Starting`
- `Running`
- `Stopping`
- `Stopped`

Stop a service

Stop a service with the following PowerShell 6 command:

```PowerShell
Stop-Service -Name {SERVICE NAME}
```

Remove a service

After a short delay to stop a service, remove a service with the following PowerShell 6 command:

```PowerShell
Remove-Service -Name {SERVICE NAME}
```

Proxy server and load balancer scenarios

Services that interact with requests from the Internet or a corporate network and are behind a proxy or load balancer might require additional configuration. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Configure endpoints

By default, ASP.NET Core binds to `http://localhost:5000`. Configure the URL and port by setting the `ASPNETCORE_URLS` environment variable.

For additional URL and port configuration approaches, see the relevant server article:

- Configure endpoints for the ASP.NET Core Kestrel web server
- HTTP.sys web server implementation in ASP.NET Core

The preceding guidance covers support for HTTPS endpoints. For example, configure the app for HTTPS when authentication is used with a Windows Service.

> **Note**
>
> Use of the ASP.NET Core HTTPS development certificate to secure a service endpoint isn't supported.

Current directory and content root

The current working directory returned by calling GetCurrentDirectory for a Windows Service is the *C:\WINDOWS\system32* folder. The *system32* folder isn't a suitable location to store a service's files (for example, settings files). Use one of the following approaches to maintain and access a service's assets and settings files.

Use ContentRootPath or ContentRootFileProvider

Use IHostEnvironment.ContentRootPath or ContentRootFileProvider to locate an app's resources.

When the app runs as a service, UseWindowsService sets the ContentRootPath to AppContext.BaseDirectory.

The app's default settings files, `appsettings.json` and `appsettings.{Environment}.json`, are loaded from the app's content root by calling CreateDefaultBuilder during host

construction.

For other settings files loaded by developer code in ConfigureAppConfiguration, there's no need to call SetBasePath. In the following example, the `custom_settings.json` file exists in the app's content root and is loaded without explicitly setting a base path:

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .UseWindowsService()
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddJsonFile("custom_settings.json");
            })
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHostedService<Worker>();
            });
}
```

Don't attempt to use GetCurrentDirectory to obtain a resource path because a Windows Service app returns the *C:\WINDOWS\system32* folder as its current directory.

Store a service's files in a suitable location on disk

Specify an absolute path with SetBasePath when using an IConfigurationBuilder to the folder containing the files.

Troubleshoot

To troubleshoot a Windows Service app, see Troubleshoot and debug ASP.NET Core projects.

Common errors

- An old or pre-release version of PowerShell is in use.

- The registered service doesn't use the app's **published** output from the dotnet publish command. Output of the dotnet build command isn't supported for app deployment. Published assets are found in either of the following folders depending on the deployment type:
  - *bin/Release/{TARGET FRAMEWORK}/publish* (FDD)
  - *bin/Release/{TARGET FRAMEWORK}/{RUNTIME IDENTIFIER}/publish* (SCD)
- The service isn't in the RUNNING state.
- The paths to resources that the app uses (for example, certificates) are incorrect. The base path of a Windows Service is *c:\Windows\System32*.
- The user doesn't have *Log on as a service* rights.
- The user's password is expired or incorrectly passed when executing the `New-Service` PowerShell command.
- The app requires ASP.NET Core authentication but isn't configured for secure connections (HTTPS).
- The request URL port is incorrect or not configured correctly in the app.

System and Application Event Logs

Access the System and Application Event Logs:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **System** to open the System Event Log. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app.

Run the app at a command prompt

Many startup errors don't produce useful information in the event logs. You can find the cause of some errors by running the app at a command prompt on the hosting system. To log additional detail from the app, lower the log level or run the app in the Development environment.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.

2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.

3. Restore and rebuild the project.

4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Slow or unresponsive app

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`.

2. Run the EnableDumps PowerShell script with the application executable name:

   PowerShell

   ```
   .\EnableDumps {APPLICATION EXE} c:\dumps
   ```

3. Run the app under the conditions that cause the crash to occur.

4. After the crash has occurred, run the DisableDumps PowerShell script:

   PowerShell

   ```
   .\DisableDumps {APPLICATION EXE}
   ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> ⚠ **Warning**

Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App is unresponsive, fails during startup, or runs normally

When an app stops responding but doesn't crash, fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Additional resources

- View or download sample code (how to download)
- Kestrel endpoint configuration (includes HTTPS configuration and SNI support)
- .NET Generic Host in ASP.NET Core
- Troubleshoot and debug ASP.NET Core projects

Host ASP.NET Core on Linux with Nginx

Article • 03/14/2024

By Sourabh Shirhatti

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

This guide explains setting up a production-ready ASP.NET Core environment for Ubuntu, Red Hat Enterprise (RHEL), and SUSE Linux Enterprise Server.

For information on other Linux distributions supported by ASP.NET Core, see Prerequisites for .NET Core on Linux.

This guide:

- Places an existing ASP.NET Core app behind a reverse proxy server.
- Sets up the reverse proxy server to forward requests to the Kestrel web server.
- Ensures the web app runs on startup as a daemon.
- Configures a process management tool to help restart the web app.

Prerequisites

Ubuntu

- Access to Ubuntu 20.04 with a standard user account with sudo privilege.
- The latest stable .NET runtime installed on the server.
- An existing ASP.NET Core app.

At any point in the future after upgrading the shared framework, restart the ASP.NET Core apps hosted by the server.

Publish and copy over the app

Configure the app for a framework-dependent deployment.

If the app is run locally in the Development environment and isn't configured by the server to make secure HTTPS connections, adopt either of the following approaches:

- Configure the app to handle secure local connections. For more information, see the HTTPS configuration section.
- Configure the app to run at the insecure endpoint:
  - Deactivate HTTPS Redirection Middleware in the Development environment (`Program.cs`):

    C#

    ```
    if (!app.Environment.IsDevelopment())
    {
        app.UseHttpsRedirection();
    }
    ```

    For more information, see Use multiple environments in ASP.NET Core.
  - Remove `https://localhost:5001` (if present) from the `applicationUrl` property in the `Properties/launchSettings.json` file.

For more information on configuration by environment, see Use multiple environments in ASP.NET Core.

Run dotnet publish from the development environment to package an app into a directory (for example, `bin/Release/{TARGET FRAMEWORK MONIKER}/publish`, where the `{TARGET FRAMEWORK MONIKER}` placeholder is the Target Framework Moniker (TFM)) that can run on the server:

.NET CLI

```
dotnet publish --configuration Release
```

The app can also be published as a self-contained deployment if you prefer not to maintain the .NET Core runtime on the server.

Copy the ASP.NET Core app to the server using a tool that integrates into the organization's workflow (for example, `SCP`, `SFTP`). It's common to locate web apps under the `var` directory (for example, `var/www/helloapp`).

> **Note**
>
> Under a production deployment scenario, a continuous integration workflow does the work of publishing the app and copying the assets to the server.

Test the app:

1. From the command line, run the app: `dotnet <app_assembly>.dll`.
2. In a browser, navigate to `http://<serveraddress>:<port>` to verify the app works on Linux locally.

Configure a reverse proxy server

A reverse proxy is a common setup for serving dynamic web apps. A reverse proxy terminates the HTTP request and forwards it to the ASP.NET Core app.

Use a reverse proxy server

Kestrel is great for serving dynamic content from ASP.NET Core. However, the web serving capabilities aren't as feature rich as servers such as IIS, Apache, or Nginx. A reverse proxy server can offload work such as serving static content, caching requests, compressing requests, and HTTPS termination from the HTTP server. A reverse proxy server may reside on a dedicated machine or may be deployed alongside an HTTP server.

For the purposes of this guide, a single instance of Nginx is used. It runs on the same server, alongside the HTTP server. Based on requirements, a different setup may be chosen.

Because requests are forwarded by reverse proxy, use the Forwarded Headers Middleware from the Microsoft.AspNetCore.HttpOverrides package, which is automatically included in ASP.NET Core apps via the shared framework's Microsoft.AspNetCore.App metapackage. The middleware updates the `Request.Scheme`, using the `X-Forwarded-Proto` header, so that redirect URIs and other security policies work correctly.

Forwarded Headers Middleware should run before other middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing. To run Forwarded Headers Middleware after diagnostics and error handling middleware, see Forwarded Headers Middleware order.

Invoke the UseForwardedHeaders method before calling other middleware. Configure the middleware to forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers:

C#

```
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication();

var app = builder.Build();

app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor |
ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();

app.MapGet("/", () => "Hello ForwardedHeadersOptions!");

app.Run();
```

If no ForwardedHeadersOptions are specified to the middleware, the default headers to forward are `None`.

Proxies running on loopback addresses (`127.0.0.0/8`, `[::1]`), including the standard localhost address (`127.0.0.1`), are trusted by default. If other trusted proxies or networks within the organization handle requests between the internet and the web server, add them to the list of KnownProxies or KnownNetworks with ForwardedHeadersOptions. The following example adds a trusted proxy server at IP address `10.0.0.100` to the Forwarded Headers Middleware `KnownProxies`:

C#

```
using Microsoft.AspNetCore.HttpOverrides;
using System.Net;

var builder = WebApplication.CreateBuilder(args);

// Configure forwarded headers
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});

builder.Services.AddAuthentication();
```

```
var app = builder.Build();

app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor |
ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();

app.MapGet("/", () => "10.0.0.100");

app.Run();
```

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Install Nginx

Ubuntu

Use `apt-get` to install Nginx. The installer creates a systemd init script that runs Nginx as daemon on system startup. Follow the installation instructions for Ubuntu at Nginx: Official Debian/Ubuntu packages.

> **Note**
>
> If optional Nginx modules are required, building Nginx from source might be required.

Since Nginx was installed for the first time, explicitly start it by running:

Bash

```
sudo service nginx start
```

Verify a browser displays the default landing page for Nginx. The landing page is reachable at `http://<server_IP_address>/index.nginx-debian.html`.

Configure Nginx

Ubuntu

To configure Nginx as a reverse proxy to forward HTTP requests to the ASP.NET Core app, modify `/etc/nginx/sites-available/default` and recreate the symlink. After creating the `/etc/nginx/sites-available/default` file, use the following command to create the symlink:

Bash

```
sudo ln -s /etc/nginx/sites-available/default /etc/nginx/sites-
enabled/default
```

Open `/etc/nginx/sites-available/default` in a text editor, and replace the contents with the following snippet:

text

```
http {
  map $http_connection $connection_upgrade {
    "~*Upgrade" $http_connection;
    default keep-alive;
  }

  server {
    listen        80;
    server_name   example.com *.example.com;
    location / {
        proxy_pass         http://127.0.0.1:5000/;
        proxy_http_version 1.1;
        proxy_set_header   Upgrade $http_upgrade;
        proxy_set_header   Connection $connection_upgrade;
        proxy_set_header   Host $host;
        proxy_cache_bypass $http_upgrade;
        proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto $scheme;
    }
  }
}
```

If the app is a SignalR or Blazor Server app, see ASP.NET Core SignalR production hosting and scaling and Host and deploy ASP.NET Core server-side Blazor apps respectively for more information.

When no `server_name` matches, Nginx uses the default server. If no default server is defined, the first server in the configuration file is the default server. As a best practice,

add a specific default server that returns a status code of 444 in your configuration file. A default server configuration example is:

text

```
server {
    listen   80 default_server;
    # listen [::]:80 default_server deferred;
    return   444;
}
```

With the preceding configuration file and default server, Nginx accepts public traffic on port 80 with host header `example.com` or `*.example.com`. Requests not matching these hosts won't get forwarded to Kestrel. Nginx forwards the matching requests to Kestrel at `http://127.0.0.1:5000/`. For more information, see How nginx processes a request. To change Kestrel's IP/port, see Kestrel: Endpoint configuration.

> **⚠ Warning**
>
> Failure to specify a proper server_name directive exposes your app to security vulnerabilities. Subdomain wildcard binding (for example, `*.example.com`) doesn't pose this security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). For more information, see RFC 9110: HTTP Semantics (Section 7.2: Host and :authority).

Once the Nginx configuration is established, run `sudo nginx -t` to verify the syntax of the configuration files. If the configuration file test is successful, force Nginx to pick up the changes by running `sudo nginx -s reload`.

To directly run the app on the server:

1. Navigate to the app's directory.
2. Run the app: `dotnet <app_assembly.dll>`, where `app_assembly.dll` is the assembly file name of the app.

Ubuntu

If the app runs on the server but fails to respond over the internet, check the server's firewall and confirm port 80 is open. If using an Azure Ubuntu VM, add a Network Security Group (NSG) rule that enables inbound port 80 traffic. There's no

need to enable an outbound port 80 rule, as the outbound traffic is automatically granted when the inbound rule is enabled.

When done testing the app, shut down the app with Ctrl+C (Windows) or ⌘+C (macOS) at the command prompt.

Increase keepalive_requests

keepalive_requests can be increased for higher performance, For more information, see this GitHub issue.

Monitor the app

The server is set up to forward requests made to `http://<serveraddress>:80` on to the ASP.NET Core app running on Kestrel at `http://127.0.0.1:5000`. However, Nginx isn't set up to manage the Kestrel process. systemd can be used to create a service file to start and monitor the underlying web app. `systemd` is an init system that provides many powerful features for starting, stopping, and managing processes.

Create the service file

Create the service definition file:

Bash

```
sudo nano /etc/systemd/system/kestrel-helloapp.service
```

The following example is an `.ini` service file for the app:

text

```
[Unit]
Description=Example .NET Web API App running on Linux

[Service]
WorkingDirectory=/var/www/helloapp
ExecStart=/usr/bin/dotnet /var/www/helloapp/helloapp.dll
Restart=always
# Restart service after 10 seconds if the dotnet service crashes:
RestartSec=10
KillSignal=SIGINT
SyslogIdentifier=dotnet-example
User=www-data
Environment=ASPNETCORE_ENVIRONMENT=Production
```

```
Environment=DOTNET_NOLOGO=true

[Install]
WantedBy=multi-user.target
```

In the preceding example, the user that manages the service is specified by the `User` option. The user (`www-data`) must exist and have proper ownership of the app's files.

Use `TimeoutStopSec` to configure the duration of time to wait for the app to shut down after it receives the initial interrupt signal. If the app doesn't shut down in this period, SIGKILL is issued to terminate the app. Provide the value as unitless seconds (for example, `150`), a time span value (for example, `2min 30s`), or `infinity` to disable the timeout. `TimeoutStopSec` defaults to the value of `DefaultTimeoutStopSec` in the manager configuration file (`systemd-system.conf`, `system.conf.d`, `systemd-user.conf`, `user.conf.d`). The default timeout for most distributions is 90 seconds.

text

```
# The default value is 90 seconds for most distributions.
TimeoutStopSec=90
```

Linux has a case-sensitive file system. Setting `ASPNETCORE_ENVIRONMENT` to `Production` results in searching for the configuration file `appsettings.Production.json`, not `appsettings.production.json`.

Some values (for example, SQL connection strings) must be escaped for the configuration providers to read the environment variables. Use the following command to generate a properly escaped value for use in the configuration file:

Console

```
systemd-escape "<value-to-escape>"
```

Colon (`:`) separators aren't supported in environment variable names. Use a double underscore (`__`) in place of a colon. The Environment Variables configuration provider converts double-underscores into colons when environment variables are read into configuration. In the following example, the connection string key `ConnectionStrings:DefaultConnection` is set into the service definition file as `ConnectionStrings__DefaultConnection`:

Console

```
Environment=ConnectionStrings__DefaultConnection={Connection String}
```

Save the file and enable the service.

Bash

```
sudo systemctl enable kestrel-helloapp.service
```

Start the service and verify that it's running.

Bash

```
sudo systemctl start kestrel-helloapp.service
sudo systemctl status kestrel-helloapp.service

● kestrel-helloapp.service - Example .NET Web API App running on Linux
    Loaded: loaded (/etc/systemd/system/kestrel-helloapp.service; enabled)
    Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s ago
Main PID: 9021 (dotnet)
    CGroup: /system.slice/kestrel-helloapp.service
            └─9021 /usr/local/bin/dotnet /var/www/helloapp/helloapp.dll
```

With the reverse proxy configured and Kestrel managed through `systemd`, the web app is fully configured and can be accessed from a browser on the local machine at `http://localhost`. It's also accessible from a remote machine, barring any firewall that might be blocking. Inspecting the response headers, the `Server` header shows the ASP.NET Core app being served by Kestrel.

text

```
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Transfer-Encoding: chunked
```

View logs

Since the web app using Kestrel is managed using systemd, all events and processes are logged to a centralized journal. However, this journal includes all entries for all services and processes managed by `systemd`. To view the `kestrel-helloapp.service`-specific items, use the following command:

Bash

```
sudo journalctl -fu kestrel-helloapp.service
```

For further filtering, time options such as `--since today`, `--until 1 hour ago`, or a combination of these can reduce the number of entries returned.

Bash

```
sudo journalctl -fu kestrel-helloapp.service --since "2016-10-18" --until
"2016-10-18 04:00"
```

Data protection

The ASP.NET Core Data Protection stack is used by several ASP.NET Core middlewares, including authentication middleware (for example, cookie middleware) and cross-site request forgery (CSRF) protections. Even if Data Protection APIs aren't called by user code, data protection should be configured 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 to persist and encrypt the key ring, see:

- Key storage providers in ASP.NET Core
- Key encryption at rest in Windows and Azure using ASP.NET Core

Long request header fields

Proxy server default settings typically limit request header fields to 4 K or 8 K depending on the platform. An app may require fields longer than the default (for example, apps that use Microsoft Entra ID). If longer fields are required, the proxy server's default settings require adjustment. The values to apply depend on the scenario. For more information, see your server's documentation.

- proxy_buffer_size

- proxy_buffers
- proxy_busy_buffers_size
- large_client_header_buffers

> ⚠ **Warning**
>
> Don't increase the default values of proxy buffers unless necessary. Increasing these values increases the risk of buffer overrun (overflow) and Denial of Service (DoS) attacks by malicious users.

Secure the app

Enable AppArmor

Linux Security Modules (LSM) is a framework that's part of the Linux kernel since Linux 2.6. LSM supports different implementations of security modules. AppArmor is an LSM that implements a Mandatory Access Control system, which allows confining the program to a limited set of resources. Ensure AppArmor is enabled and properly configured.

Configure the firewall

Close off all external ports that aren't in use. Uncomplicated firewall (ufw) provides a front end for `iptables` by providing a CLI for configuring the firewall.

Ubuntu

> ⚠ **Warning**
>
> A firewall prevents access to the whole system if not configured correctly. Failure to specify the correct SSH port effectively locks you out of the system if you are using SSH to connect to it. The default port is 22. For more information, see the introduction to ufw and the manual.

Install `ufw` and configure it to allow traffic on any ports needed.

Bash

```
sudo apt-get install ufw

sudo ufw allow 22/tcp
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp

sudo ufw enable
```

Secure Nginx

Change the Nginx response name

Edit `src/http/ngx_http_header_filter_module.c`:

C

```
static char ngx_http_server_string[] = "Server: Web Server" CRLF;
static char ngx_http_server_full_string[] = "Server: Web Server" CRLF;
```

Configure options

Configure the server with additional required modules. Consider using a web app firewall, such as ModSecurity, to harden the app.

HTTPS configuration

Configure the app for secure (HTTPS) local connections

The dotnet run command uses the app's `Properties/launchSettings.json` file, which configures the app to listen on the URLs provided by the `applicationUrl` property. For example, `https://localhost:5001;http://localhost:5000`.

Configure the app to use a certificate in development for the `dotnet run` command or development environment (F5 or Ctrl+F5 in Visual Studio Code) using one of the following approaches:

- Replace the default certificate from configuration (*Recommended*)
- KestrelServerOptions.ConfigureHttpsDefaults

Configure the reverse proxy for secure (HTTPS) client connections

> ⚠ **Warning**
>
> The security configuration in this section is a general configuration to be used as a starting point for further customization. We're unable to provide support for third-party tooling, servers, and operating systems. *Use the configuration in this section at your own risk.* For more information, access the following resources:
>
> - Configuring HTTPS servers (Nginx documentation)
> - mozilla.org SSL Configuration Generator

- Configure the server to listen to HTTPS traffic on port 443 by specifying a valid certificate issued by a trusted Certificate Authority (CA).
- Harden the security by employing some of the practices depicted in the following */etc/nginx/nginx.conf* file.
- The following example doesn't configure the server to redirect insecure requests. We recommend using HTTPS Redirection Middleware. For more information, see Enforce HTTPS in ASP.NET Core.

  > ⓘ **Note**
  >
  > For development environments where the server configuration handles secure redirection instead of HTTPS Redirection Middleware, we recommend using temporary redirects (302) rather than permanent redirects (301). Link caching can cause unstable behavior in development environments.

- Adding a `Strict-Transport-Security` (HSTS) header ensures all subsequent requests made by the client are over HTTPS. For guidance on setting the `Strict-Transport-Security` header, see Enforce HTTPS in ASP.NET Core.
- If HTTPS will be disabled in the future, use one of the following approaches:
  - Don't add the HSTS header.
  - Choose a short `max-age` value.

Add the */etc/nginx/proxy.conf* configuration file:

nginx

```
proxy_redirect          off;
proxy_set_header        Host $host;
proxy_set_header        X-Real-IP $remote_addr;
proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
```

```
proxy_set_header        X-Forwarded-Proto $scheme;
client_max_body_size    10m;
client_body_buffer_size 128k;
proxy_connect_timeout   90;
proxy_send_timeout      90;
proxy_read_timeout      90;
proxy_buffers           32 4k;
```

Ubuntu

Replace the contents of the */etc/nginx/nginx.conf* configuration file with the following file. The example contains both `http` and `server` sections in one configuration file.

nginx

```
http {
    include        /etc/nginx/proxy.conf;
    limit_req_zone $binary_remote_addr zone=one:10m rate=5r/s;
    server_tokens  off;

    sendfile on;
    # Adjust keepalive_timeout to the lowest possible value that makes sense
    # for your use case.
    keepalive_timeout   29;
    client_body_timeout 10; client_header_timeout 10; send_timeout 10;

    upstream helloapp{
        server 127.0.0.1:5000;
    }

    server {
        listen                  443 ssl http2;
        listen                  [::]:443 ssl http2;
        server_name             example.com *.example.com;
        ssl_certificate         /etc/ssl/certs/testCert.crt;
        ssl_certificate_key     /etc/ssl/certs/testCert.key;
        ssl_session_timeout     1d;
        ssl_protocols           TLSv1.2 TLSv1.3;
        ssl_prefer_server_ciphers off;
        ssl_ciphers             ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-
SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-
AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384;
        ssl_session_cache       shared:SSL:10m;
        ssl_session_tickets     off;
        ssl_stapling            off;

        add_header X-Frame-Options DENY;
        add_header X-Content-Type-Options nosniff;
```

```
        #Redirects all traffic
        location / {
            proxy_pass http://helloapp;
            limit_req  zone=one burst=10 nodelay;
        }
    }
}
```

> **Note**
>
> Blazor WebAssembly apps require a larger `burst` parameter value to accommodate the larger number of requests made by an app. For more information, see Host and deploy ASP.NET Core Blazor WebAssembly.

> **Note**
>
> The preceding example disables Online Certificate Status Protocol (OCSP) Stapling. If enabled, confirm that the certificate supports the feature. For more information and guidance on enabling OCSP, see the following properties in the Module ngx_http_ssl_module (Nginx documentation) article:
>
> - `ssl_stapling`
> - `ssl_stapling_file`
> - `ssl_stapling_responder`
> - `ssl_stapling_verify`

Secure Nginx from clickjacking

Clickjacking, also known as a *UI redress attack*, is a malicious attack where a website visitor is tricked into clicking a link or button on a different page than they're currently visiting. Use `X-FRAME-OPTIONS` to secure the site.

To mitigate clickjacking attacks:

1. Edit the *nginx.conf* file:

```
Bash

sudo nano /etc/nginx/nginx.conf
```

Within the `http{}` code block, add the line: `add_header X-Frame-Options "SAMEORIGIN";`

2. Save the file.

3. Restart Nginx.

MIME-type sniffing

This header prevents most browsers from MIME-sniffing a response away from the declared content type, as the header instructs the browser not to override the response content type. With the `nosniff` option, if the server says the content is `text/html`, the browser renders it as `text/html`.

1. Edit the *nginx.conf* file:

```
Bash

sudo nano /etc/nginx/nginx.conf
```

Within the `http{}` code block, add the line: `add_header X-Content-Type-Options "nosniff";`

2. Save the file.

3. Restart Nginx.

Additional Nginx suggestions

After upgrading the shared framework on the server, restart the ASP.NET Core apps hosted by the server.

Additional resources

- Prerequisites for .NET Core on Linux
- Nginx: Binary Releases: Official Debian/Ubuntu packages
- Troubleshoot and debug ASP.NET Core projects
- Configure ASP.NET Core to work with proxy servers and load balancers
- NGINX: Using the Forwarded header

Host ASP.NET Core in Docker containers

Article • 07/31/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

The following articles are available for learning about hosting ASP.NET Core apps in Docker:

Introduction to Containers and Docker

See how containerization is an approach to software development in which an application or service, its dependencies, and its configuration are packaged together as a container image. The image can be tested and then deployed to a host.

What is Docker

Discover how Docker is an open-source project for automating the deployment of apps as portable, self-sufficient containers that can run on the cloud or on-premises.

Docker Terminology

Learn terms and definitions for Docker technology.

Docker containers, images, and registries

Find out how Docker container images are stored in an image registry for consistent deployment across environments.

Run an ASP.NET Core app in Docker containers Learn how to build and dockerize an ASP.NET Core app. Explore Docker images maintained by Microsoft and examine use cases.

.NET Docker samples Samples and guidance that demonstrate how to use .NET and Docker for development, testing and production.

Visual Studio Container Tools

Discover how Visual Studio supports building, debugging, and running ASP.NET Core apps targeting either .NET Framework or .NET Core on Docker for Windows. Both Windows and Linux containers are supported.

Publish to Azure Container Registry
Find out how to use the Visual Studio Container Tools extension to deploy an ASP.NET Core app to a Docker host on Azure using PowerShell.

Configure ASP.NET Core to work with proxy servers and load balancers
Additional configuration might be required for apps hosted behind proxy servers and load balancers. Passing requests through a proxy often obscures information about the original request, such as the scheme and client IP. It might be necessary to forward some information about the request manually to the app.

GC using Docker and small containers Discusses GC selection with small containers.

System.IO.IOException: The configured user limit (128) on the number of inotify instances has been reached

Disabling `reloadOnChange` can significantly reduce the number of opened files. To disable reloading configuration files, set the environment variable `DOTNET_HOSTBUILDER__RELOADCONFIGONCHANGE=false`

For alternative approaches or to leave feedback on this problem, see this GitHub issue.

Run an ASP.NET Core app in Docker containers

Article • 07/31/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

This article shows how to run an ASP.NET Core app in Docker containers.

Windows Home Edition doesn't support Hyper-V, and Hyper-V is needed for Docker.

See Containerize a .NET app with dotnet publish for information on containerized a .NET app with `dotnet publish`.

ASP.NET Core Docker images

For this tutorial, you download an ASP.NET Core sample app and run it in Docker containers. The sample works with both Linux and Windows containers.

The sample Dockerfile uses the Docker multi-stage build feature to build and run in different containers. The build and run containers are created from images that are provided in Docker Hub by Microsoft:

- `dotnet/sdk`

  The sample uses this image for building the app. The image contains the .NET SDK, which includes the Command Line Tools (CLI). The image is optimized for local development, debugging, and unit testing. The tools installed for development and compilation make the image relatively large.

- `dotnet/aspnet`

  The sample uses this image for running the app. The image contains the ASP.NET Core runtime and libraries and is optimized for running apps in production. Designed for speed of deployment and app startup, the image is relatively small, so network performance from Docker Registry to Docker host is optimized. Only

the binaries and content needed to run an app are copied to the container. The contents are ready to run, enabling the fastest time from `docker run` to app startup. Dynamic code compilation isn't needed in the Docker model.

Prerequisites

- .NET SDK 8.0
- Docker client 18.03 or later
  - Linux distributions
    - Debian
    - Fedora
    - Ubuntu
  - macOS
  - Windows
- Git

Download the sample app

- Download the sample by cloning the .NET Docker repository:

```
Console
git clone https://github.com/dotnet/dotnet-docker
```

Run the app locally

- Navigate to the project folder at *dotnet-docker/samples/aspnetapp/aspnetapp*.
- Run the following command to build and run the app locally:

```
.NET CLI
dotnet run
```

- Go to `http://localhost:<port>` in a browser to test the app.
- Press Ctrl+C at the command prompt to stop the app.

Run in a Linux container or Windows container

- To run in a Linux container, right-click the System Tray's Docker client icon and select switch to Linux containers.
- To run in a Windows container, right-click the System Tray's Docker client icon and select switch to Windows containers.
- Navigate to the Dockerfile folder at *dotnet-docker/samples/aspnetapp*.
- Run the following commands to build and run the sample in Docker:

  Console

  ```
  docker build -t aspnetapp .
  docker run -it --rm -p <port>:8080 --name aspnetcore_sample aspnetapp
  ```

  The `build` command arguments:
  - Name the image aspnetapp.
  - Look for the Dockerfile in the current folder (the period at the end).

  The run command arguments:
  - Allocate a pseudo-TTY and keep it open even if not attached. (Same effect as `--interactive --tty`.)
  - Automatically remove the container when it exits.
  - Map `<port>` on the local machine to port 8080 in the container.
  - Name the container aspnetcore_sample.
  - Specify the aspnetapp image.
- Go to `http://localhost:<port>` in a browser to test the app.

Build and deploy manually

In some scenarios, you might want to deploy an app to a container by copying its assets that are needed at run time. This section shows how to deploy manually.

- Navigate to the project folder at *dotnet-docker/samples/aspnetapp/aspnetapp*.
- Run the dotnet publish command:

  .NET CLI

  ```
  dotnet publish -c Release -o published
  ```

  The command arguments:

    - Build the app in release mode (the default is debug mode).
    - Create the assets in the *published* folder.

- Run the app.

    - Windows:

```
.NET CLI

dotnet published\aspnetapp.dll
```

    - Linux:

```
.NET CLI

dotnet published/aspnetapp.dll
```

- Browse to `http://localhost:<port>` to see the home page.

To use the manually published app within a Docker container, create a new *Dockerfile* and use the `docker build .` command to build an image.

```
Dockerfile

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime
WORKDIR /app
COPY published/ ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

To see the new image use the `docker images` command.

The Dockerfile

Here's the *Dockerfile* used by the `docker build` command you ran earlier. It uses `dotnet publish` the same way you did in this section to build and deploy.

```
Dockerfile

# https://hub.docker.com/_/microsoft-dotnet
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /source

# copy csproj and restore as distinct layers
COPY *.sln .
COPY aspnetapp/*.csproj ./aspnetapp/
RUN dotnet restore
```

```
# copy everything else and build app
COPY aspnetapp/. ./aspnetapp/
WORKDIR /source/aspnetapp
RUN dotnet publish -c release -o /app --no-restore

# final stage/image
FROM mcr.microsoft.com/dotnet/aspnet:8.0
WORKDIR /app
COPY --from=build /app ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

In the preceding *Dockerfile*, the `*.csproj` files are copied and restored as distinct *layers*. When the `docker build` command builds an image, it uses a built-in cache. If the `*.csproj` files haven't changed since the `docker build` command last ran, the `dotnet restore` command doesn't need to run again. Instead, the built-in cache for the corresponding `dotnet restore` layer is reused. For more information, see Best practices for writing Dockerfiles.

Additional resources

- Containerize a .NET app with dotnet publish
- Docker build command
- Docker run command
- ASP.NET Core Docker sample (The one used in this tutorial.)
- Configure ASP.NET Core to work with proxy servers and load balancers
- Working with Visual Studio Docker Tools
- Debugging with Visual Studio Code
- GC using Docker and small containers
- System.IO.IOException: The configured user limit (128) on the number of inotify instances has been reached
- Updates to Docker images

Next steps

The Git repository that contains the sample app also includes documentation. For an overview of the resources available in the repository, see the README file. In particular, learn how to implement HTTPS:

Developing ASP.NET Core Applications with Docker over HTTPS

Visual Studio Container Tools with ASP.NET Core

Article • 07/31/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

Visual Studio 2017 and later versions support building, debugging, and running containerized ASP.NET Core apps targeting .NET Core. Both Windows and Linux containers are supported.

View or download sample code (how to download)

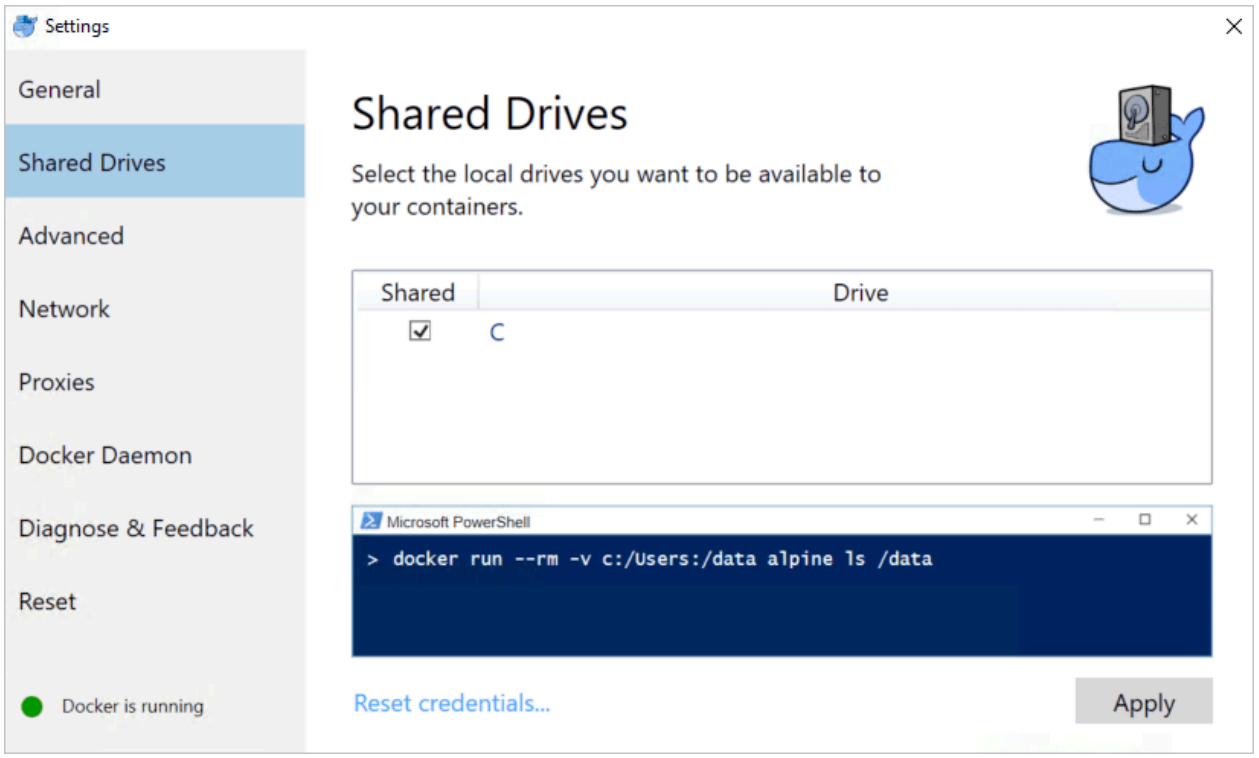
Prerequisites

- Docker for Windows
- Visual Studio 2019 with the **.NET Core cross-platform development** workload

Installation and setup

For Docker installation, first review the information at Docker for Windows: What to know before you install. Next, install Docker For Windows.

Shared Drives in Docker for Windows must be configured to support volume mapping and debugging. Right-click the System Tray's Docker icon, select **Settings**, and select **Shared Drives**. Select the drive where Docker stores files. Click **Apply**.



> **Tip**
>
> Visual Studio 2017 versions 15.6 and later prompt when **Shared Drives** aren't configured.

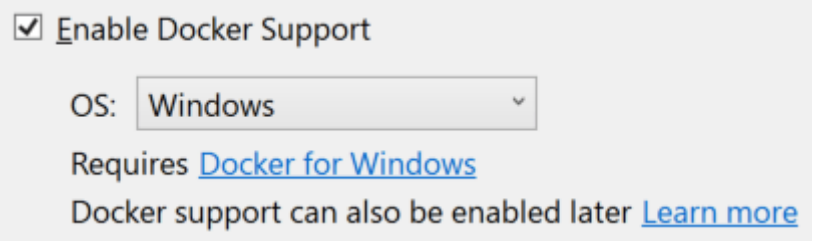
Add a project to a Docker container

To containerize an ASP.NET Core project, the project must target .NET Core. Both Linux and Windows containers are supported.

When adding Docker support to a project, choose either a Windows or a Linux container. The Docker host must be running the same container type. To change the container type in the running Docker instance, right-click the System Tray's Docker icon and choose **Switch to Windows containers...** or **Switch to Linux containers...**.

New app

When creating a new app with the **ASP.NET Core Web Application** project templates, select the **Enable Docker Support** checkbox:



If the target framework is .NET Core, the **OS** drop-down allows for the selection of a container type.

Existing app

For ASP.NET Core projects targeting .NET Core, there are two options for adding Docker support via the tooling. Open the project in Visual Studio, and choose one of the following options:

- Select **Docker Support** from the **Project** menu.
- Right-click the project in **Solution Explorer** and select **Add** > **Docker Support**.

The Visual Studio Container Tools don't support adding Docker to an existing ASP.NET Core project targeting .NET Framework.

Dockerfile overview

A *Dockerfile*, the recipe for creating a final Docker image, is added to the project root. Refer to Dockerfile reference for an understanding of the commands within it. This particular *Dockerfile* uses a multi-stage build with four distinct, named build stages:

Dockerfile

```
FROM mcr.microsoft.com/dotnet/core/aspnet:2.1 AS base
WORKDIR /app
EXPOSE 59518
EXPOSE 44364

FROM mcr.microsoft.com/dotnet/core/sdk:2.1 AS build
WORKDIR /src
COPY HelloDockerTools/HelloDockerTools.csproj HelloDockerTools/
RUN dotnet restore HelloDockerTools/HelloDockerTools.csproj
COPY . .
WORKDIR /src/HelloDockerTools
RUN dotnet build HelloDockerTools.csproj -c Release -o /app

FROM build AS publish
RUN dotnet publish HelloDockerTools.csproj -c Release -o /app

FROM base AS final
```

```
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]
```

The preceding *Dockerfile* image includes the ASP.NET Core runtime and NuGet packages. The packages are just-in-time (JIT) compiled to improve startup performance.

When the new project dialog's **Configure for HTTPS** checkbox is checked, the *Dockerfile* exposes two ports. One port is used for HTTP traffic; the other port is used for HTTPS. If the checkbox isn't checked, a single port (80) is exposed for HTTP traffic.

Add container orchestrator support to an app

Visual Studio 2017 versions 15.7 or earlier support Docker Compose as the sole container orchestration solution. The Docker Compose artifacts are added via **Add** > **Docker Support**.

Visual Studio 2017 versions 15.8 or later add an orchestration solution only when instructed. Right-click the project in **Solution Explorer** and select **Add** > **Container Orchestrator Support**. The following choices are available:

- Docker Compose
- Service Fabric
- Kubernetes/Helm

Docker Compose

The Visual Studio Container Tools add a *docker-compose* project to the solution with the following files:

- *docker-compose.dcproj*: The file representing the project. Includes a `<DockerTargetOS>` element specifying the OS to be used.
- *.dockerignore*: Lists the file and directory patterns to exclude when generating a build context.
- *docker-compose.yml*: The base Docker Compose file used to define the collection of images built and run with `docker-compose build` and `docker-compose run`, respectively.
- *docker-compose.override.yml*: An optional file, read by Docker Compose, with configuration overrides for services. Visual Studio executes `docker-compose -f "docker-compose.yml" -f "docker-compose.override.yml"` to merge these files.

The *docker-compose.yml* file references the name of the image that's created when the project runs:

```yaml
version: '3.4'

services:
  hellodockertools:
    image: ${DOCKER_REGISTRY}hellodockertools
    build:
      context: .
      dockerfile: HelloDockerTools/Dockerfile
```

In the preceding example, `image: hellodockertools` generates the image `hellodockertools:dev` when the app runs in **Debug** mode. The `hellodockertools:latest` image is generated when the app runs in **Release** mode.

Prefix the image name with the Docker Hub username (for example, `dockerhubusername/hellodockertools`) if the image is pushed to the registry. Alternatively, change the image name to include the private registry URL (for example, `privateregistry.domain.com/hellodockertools`) depending on the configuration.

If you want different behavior based on the build configuration (for example, Debug or Release), add configuration-specific *docker-compose* files. The files should be named according to the build configuration (for example, *docker-compose.vs.debug.yml* and *docker-compose.vs.release.yml*) and placed in the same location as the *docker-compose-override.yml* file.

Using the configuration-specific override files, you can specify different configuration settings (such as environment variables or entry points) for Debug and Release build configurations.

For Docker Compose to display an option to run in Visual Studio, the docker project must be the startup project.

Service Fabric

In addition to the base Prerequisites, the Service Fabric orchestration solution demands the following prerequisites:

- Microsoft Azure Service Fabric SDK version 2.6 or later
- Visual Studio's **Azure Development** workload

Service Fabric doesn't support running Linux containers in the local development cluster on Windows. If the project is already using a Linux container, Visual Studio prompts to switch to Windows containers.

The Visual Studio Container Tools do the following tasks:

- Adds a *<project_name>Application* **Service Fabric Application** project to the solution.
- Adds a *Dockerfile* and a *.dockerignore* file to the ASP.NET Core project. If a *Dockerfile* already exists in the ASP.NET Core project, it's renamed to *Dockerfile.original*. A new *Dockerfile*, similar to the following, is created:

  Dockerfile

  ```
  # See https://aka.ms/containerimagehelp for information on how to use
  Windows Server 1709 containers with Service Fabric.
  # FROM microsoft/aspnetcore:2.0-nanoserver-1709
  FROM microsoft/aspnetcore:2.0-nanoserver-sac2016
  ARG source
  WORKDIR /app
  COPY ${source:-obj/Docker/publish} .
  ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]
  ```

- Adds an `<IsServiceFabricServiceProject>` element to the ASP.NET Core project's `.csproj` file:

  XML

  ```
  <IsServiceFabricServiceProject>True</IsServiceFabricServiceProject>
  ```

- Adds a *PackageRoot* folder to the ASP.NET Core project. The folder includes the service manifest and settings for the new service.

For more information, see Deploy a .NET app in a Windows container to Azure Service Fabric.

Debug

Select **Docker** from the debug drop-down in the toolbar, and start debugging the app. The **Docker** view of the **Output** window shows the following actions taking place:

- The *2.1-aspnetcore-runtime* tag of the *microsoft/dotnet* runtime image is acquired (if not already in the cache). The image installs the ASP.NET Core and .NET Core

runtimes and associated libraries. It's optimized for running ASP.NET Core apps in production.
- The `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development` within the container.
- Two dynamically assigned ports are exposed: one for HTTP and one for HTTPS. The port assigned to localhost can be queried with the `docker ps` command.
- The app is copied to the container.
- The default browser is launched with the debugger attached to the container using the dynamically assigned port.

The resulting Docker image of the app is tagged as *dev*. The image is based on the *2.1-aspnetcore-runtime* tag of the *microsoft/dotnet* base image. Run the `docker images` command in the **Package Manager Console** (PMC) window. The images on the machine are displayed:

Console

```
REPOSITORY        TAG                   IMAGE ID      CREATED         SIZE
hellodockertools  dev                   d72ce0f1dfe7  30 seconds ago
255MB
microsoft/dotnet  2.1-aspnetcore-runtime  fcc3887985bb  6 days ago
255MB
```

> **Note**
>
> The *dev* image lacks the app contents, as **Debug** configurations use volume mounting to provide the iterative experience. To push an image, use the **Release** configuration.

Run the `docker ps` command in PMC. Notice the app is running using the container:

Console

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED
STATUS              PORTS                   NAMES
baf9a678c88d        hellodockertools:dev   "C:\\remote_debugge..."   21
seconds ago      Up 19 seconds       0.0.0.0:37630->80/tcp
dockercompose4642749010770307127_hellodockertools_1
```

Edit and continue

Changes to static files and Razor views are automatically updated without the need for a compilation step. Make the change, save, and refresh the browser to view the update.

Code file modifications require compilation and a restart of Kestrel within the container. After making the change, use `CTRL+F5` to perform the process and start the app within the container. The Docker container isn't rebuilt or stopped. Run the `docker ps` command in PMC. Notice the original container is still running as of 10 minutes ago:

Console

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED
STATUS              PORTS                   NAMES
baf9a678c88d        hellodockertools:dev   "C:\\remote_debugge..."   10
minutes ago      Up 10 minutes       0.0.0.0:37630->80/tcp
dockercompose4642749010770307127_hellodockertools_1
```

Publish Docker images

Once the develop and debug cycle of the app is completed, the Visual Studio Container Tools assist in creating the production image of the app. Change the configuration drop-down to **Release** and build the app. The tooling acquires the compile/publish image from Docker Hub (if not already in the cache). An image is produced with the *latest* tag, which can be pushed to the private registry or Docker Hub.

Run the `docker images` command in PMC to see the list of images. Output similar to the following is displayed:

Console

```
REPOSITORY        TAG                    IMAGE ID      CREATED
SIZE
hellodockertools  latest                 e3984a64230c  About a minute ago
258MB
hellodockertools  dev                    d72ce0f1dfe7  4 minutes ago
255MB
microsoft/dotnet  2.1-sdk                9e243db15f91  6 days ago
1.7GB
microsoft/dotnet  2.1-aspnetcore-runtime  fcc3887985bb  6 days ago
255MB
```

> ⓘ **Note**
>
> The `docker images` command returns intermediary images with repository names and tags identified as <*none*> (not listed above). These unnamed images are

produced by the **multi-stage build** *Dockerfile*. They improve the efficiency of building the final image—only the necessary layers are rebuilt when changes occur. When the intermediary images are no longer needed, delete them using the **docker rmi** command.

There may be an expectation for the production or release image to be smaller in size by comparison to the *dev* image. Because of the volume mapping, the debugger and app were running from the local machine and not within the container. The *latest* image has packaged the necessary app code to run the app on a host machine. Therefore, the delta is the size of the app code.

Additional resources

- Container development with Visual Studio
- Azure Service Fabric: Prepare your development environment
- Deploy a .NET app in a Windows container to Azure Service Fabric
- Troubleshoot Visual Studio development with Docker
- Visual Studio Container Tools GitHub repository
- GC using Docker and small containers
- System.IO.IOException: The configured user limit (128) on the number of inotify instances has been reached
- Updates to Docker images

Deploy an ASP.NET container to a container registry using Visual Studio

Article • 08/22/2024

Docker is a lightweight container engine, similar in some ways to a virtual machine, which you can use to host applications and services. This tutorial walks you through using Visual Studio to publish your containerized application to an Azure Container Registry.

If you don't have an Azure subscription, create a free account before you begin.

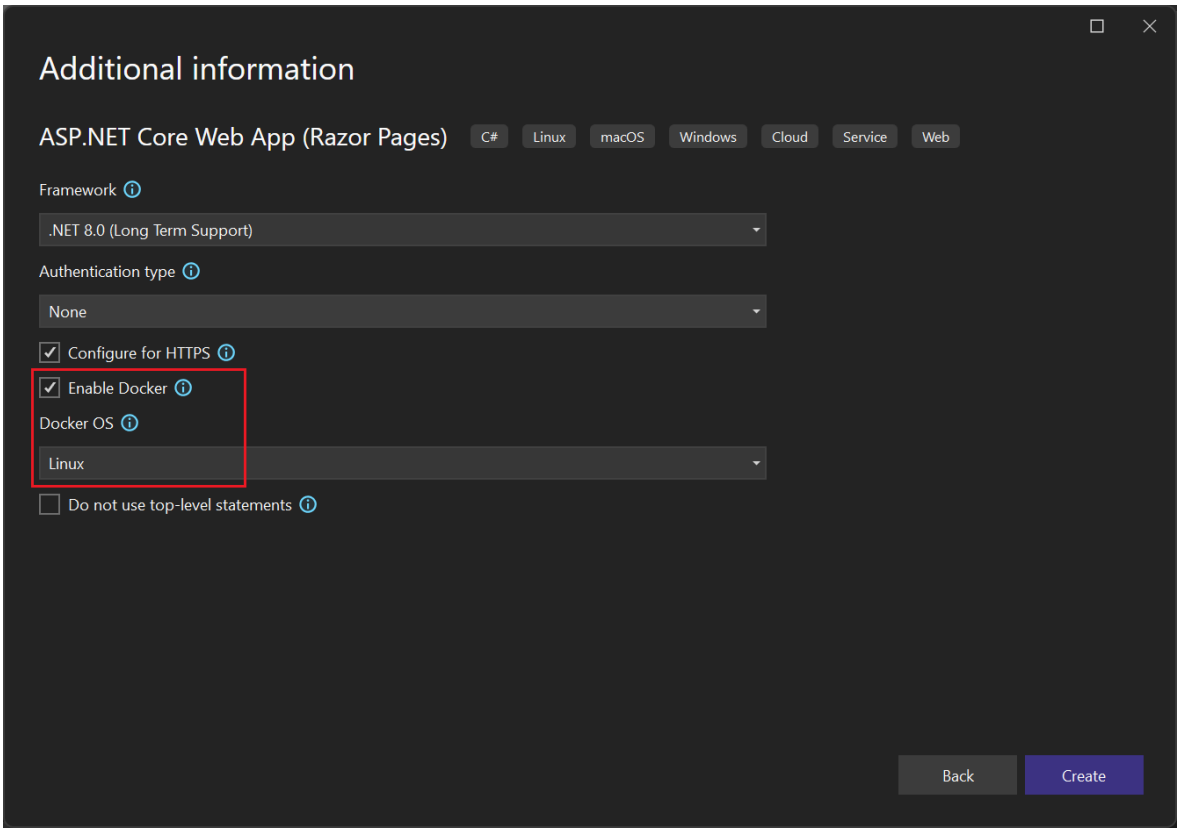
Prerequisites

- Install the latest version of Visual Studio 2022 with the "ASP.NET and web development" workload.
- Install Docker Desktop for Windows.

Create an ASP.NET Core web app

The following steps guide you through creating a basic ASP.NET Core app that you use in this tutorial. If you already have a project, you can skip this section.

1. In the Visual Studio start window, select **Create a new project**.
2. Select **ASP.NET Core Web App**, and then select **Next**.
3. Enter a name for your new application (or use the default name), specify the location on disk, and then select **Next**.
4. Select the .NET version you want to target. If you're not sure, choose the LTS (long-term support) release.



5. Choose whether you want SSL support by selecting or clearing the **Configure for HTTPS** checkbox.

6. Select the **Enable Docker** checkbox.

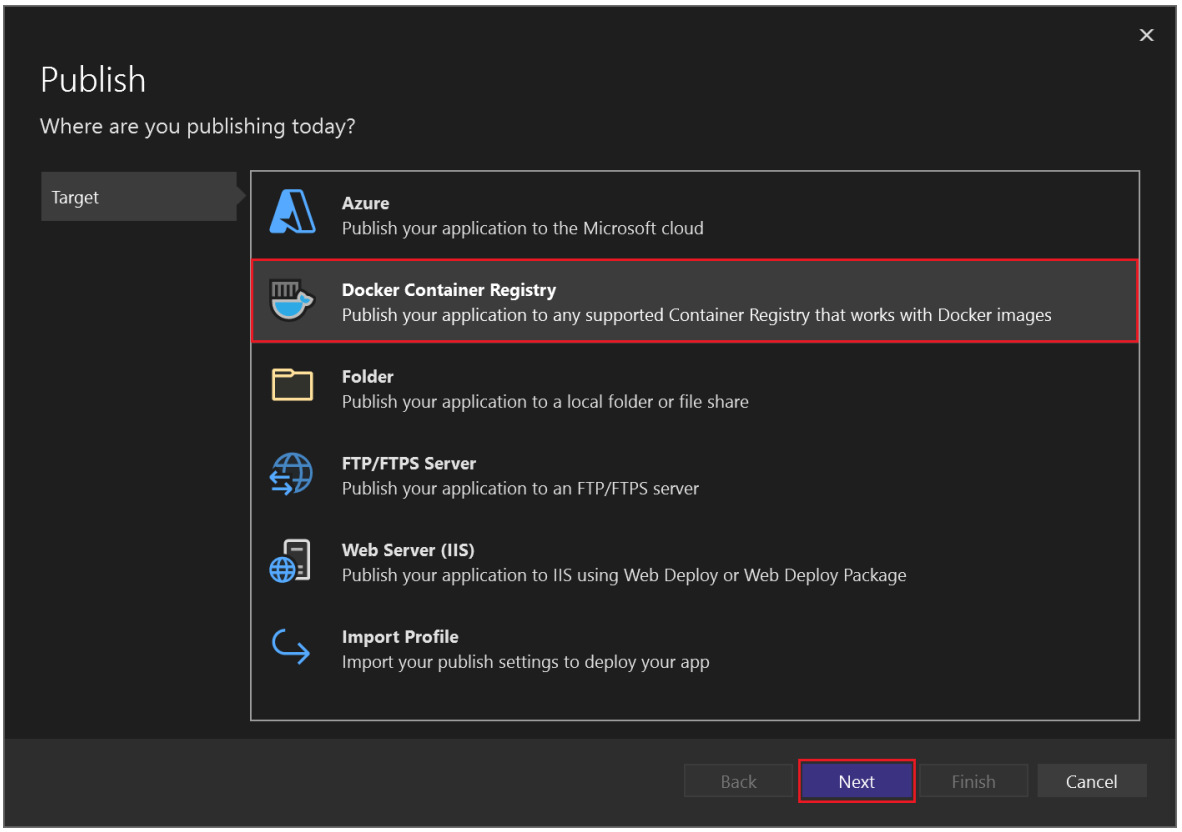
7. Use the **Docker OS** dropdown list to select the type of container you want: **Windows** or **Linux**.

8. Select **Create** to complete the process.

Publish your container to Azure Container Registry

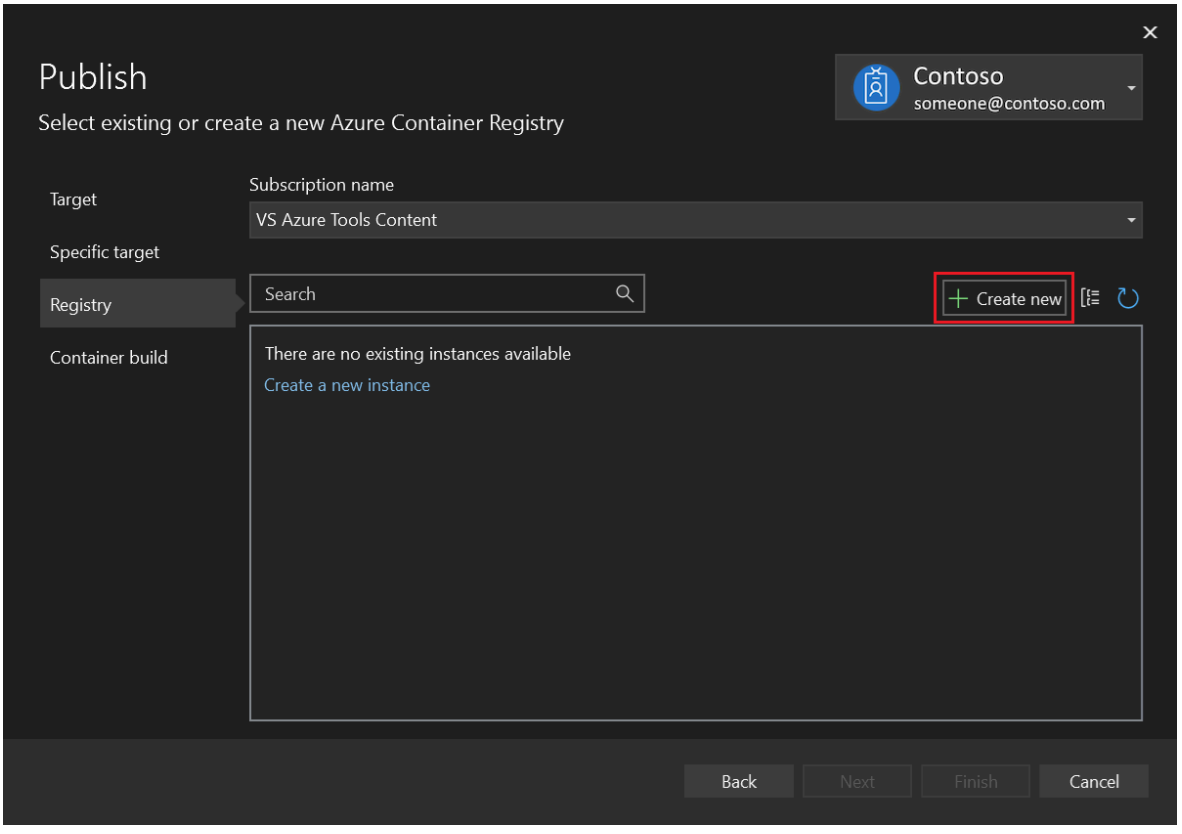
1. Right-click your project in **Solution Explorer** and choose **Publish**. The **Publish** dialog opens.

2. On the **Target** tab, select **Docker Container Registry**, and then select **Next**.



3. On the **Specific target** tab, select **Azure Container Registry**, and then select **Next**.

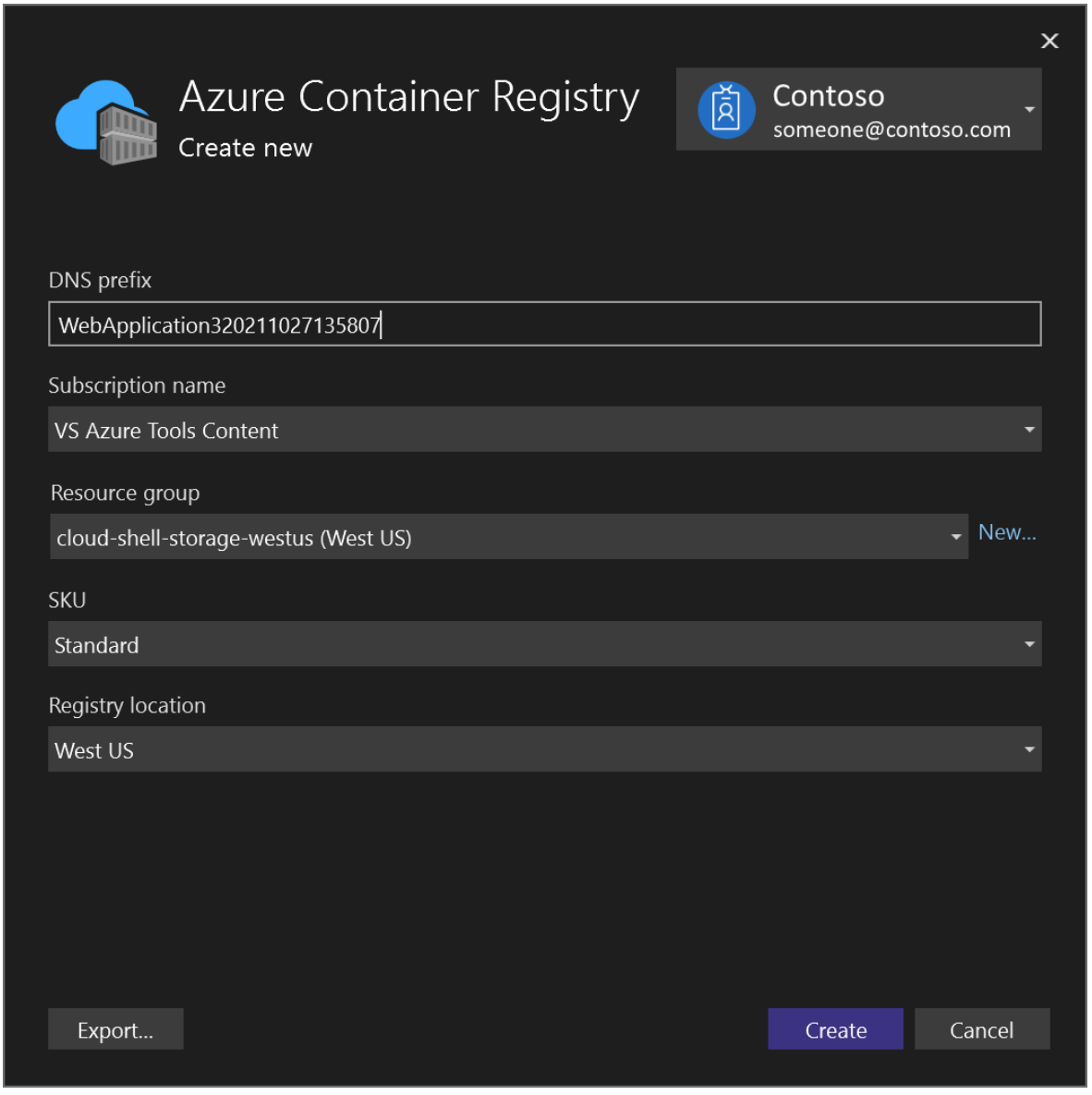
4. On the **Registry** tab, select the **Create new** (+) option at the right:



5. Fill in your desired values in the **Azure Container Registry** screen.

Expand table

Setting	Suggested value	Description
DNS Prefix	Globally unique name	Name that uniquely identifies your container registry.
Subscription	Your subscription	The Azure subscription to use.
Resource Group	Your resource group	Name of the resource group in which to create your container registry. Select **New** to create a new resource group.
SKU	"Standard"	Select the service tier of the container registry.
Registry Location	A nearby location	Choose a location in a region close to you or close to other services that you expect to use the container registry.



6. After you enter the resource values, select **Create**.

Visual Studio validates the property values and creates the new container resource. When the process completes, Visual Studio returns to the **Publish** dialog and selects the new container in the list.

7. Select **Finish** to publish the new container.

You can now pull the container from the registry to any host capable of running Docker images, such as Azure Container Instances.

Related content

- Quickstart: Deploy a container instance in Azure using the Azure CLI

Feedback

Configure ASP.NET Core to work with proxy servers and load balancers

Article • 09/27/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Chris Ross

In the recommended configuration for ASP.NET Core, the app is hosted using ASP.NET Core Module (ANCM) for IIS, Nginx, or Apache. Proxy servers, load balancers, and other network appliances often obscure information about the request before it reaches the app:

- When HTTPS requests are proxied over HTTP, the original scheme (HTTPS) is lost and must be forwarded in a header.
- Because an app receives a request from the proxy and not its true source on the Internet or corporate network, the originating client IP address must also be forwarded in a header.

This information may be important in request processing, for example in redirects, authentication, link generation, policy evaluation, and client geolocation.

Apps intended to run on web farm should read Host ASP.NET Core in a web farm.

Forwarded headers

By convention, proxies forward information in HTTP headers.

Expand table

Header	Description
X-Forwarded-For (XFF)	Holds information about the client that initiated the request and subsequent proxies in a chain of proxies. This parameter may contain IP addresses and, optionally, port numbers. In a chain of proxy servers, the first parameter indicates

Header	Description
	the client where the request was first made. Subsequent proxy identifiers follow. The last proxy in the chain isn't in the list of parameters. The last proxy's IP address, and optionally a port number, are available as the remote IP address at the transport layer.
X-Forwarded-Proto (XFP)	The value of the originating scheme, HTTP or HTTPS. The value may also be a list of schemes if the request has traversed multiple proxies.
X-Forwarded-Host (XFH)	The original value of the Host header field. Usually, proxies don't modify the Host header. See Microsoft Security Advisory CVE-2018-0787 for information on an elevation-of-privileges vulnerability that affects systems where the proxy doesn't validate or restrict Host headers to known good values.
`X-Forwarded-Prefix`	The original base path requested by the client. This header can be useful for applications to correctly generate URLs, redirects, or links back to the client.

The Forwarded Headers Middleware, ForwardedHeadersMiddleware, reads these headers and fills in the associated fields on HttpContext.

The middleware updates:

- HttpContext.Connection.RemoteIpAddress: Set using the `X-Forwarded-For` header value. Additional settings influence how the middleware sets `RemoteIpAddress`. For details, see the Forwarded Headers Middleware options. The consumed values are removed from `X-Forwarded-For`, and the old value of HttpContext.Connection.RemoteIpAddress is persisted in `X-Original-For`. Note: This process may be repeated several times if there are multiple values in `X-Forwarded-For/Proto/Host/Prefix`, resulting in several values moved to `X-Original-*`, including the original `RemoteIpAddress/Host/Scheme/PathBase`.
- HttpContext.Request.Scheme: Set using the X-Forwarded-Proto header value. The consumed value is removed from `X-Forwarded-Proto`, and the old value of HttpContext.Request.Scheme is persisted in `X-Original-Proto`.
- HttpContext.Request.Host: Set using the `X-Forwarded-Host` header value. The consumed value is removed from `X-Forwarded-Host`, and the old value of HttpContext.Request.Host is persisted in `X-Original-Host`.
- HttpContext.Request.PathBase: Set using the `X-Forwarded-Prefix` header value. The consumed value is removed from `X-Forwarded-Prefix`, and the old value of HttpContext.Request.PathBase is persisted in `X-Original-Prefix`.

For more information on the preceding, see this GitHub issue.

Forwarded Headers Middleware default settings can be configured. For the default settings:

- There is only ***one proxy*** between the app and the source of the requests.
- Only loopback addresses are configured for known proxies and known networks.
- The forwarded headers are named X-Forwarded-For, X-Forwarded-Proto, X-Forwarded-Host and `X-Forwarded-Prefix`.
- The `ForwardedHeaders` value is `ForwardedHeaders.None`, the desired forwarders must be set here to enable the middleware.

Not all network appliances add the `X-Forwarded-For` and `X-Forwarded-Proto` headers without additional configuration. Consult your appliance manufacturer's guidance if proxied requests don't contain these headers when they reach the app. If the appliance uses different header names than `X-Forwarded-For` and `X-Forwarded-Proto`, set the ForwardedForHeaderName and ForwardedProtoHeaderName options to match the header names used by the appliance. For more information, see Forwarded Headers Middleware options and Configuration for a proxy that uses different header names.

IIS/IIS Express and ASP.NET Core Module

Forwarded Headers Middleware is enabled by default by IIS Integration Middleware when the app is hosted out-of-process behind IIS and the ASP.NET Core Module (ANCM) for IIS. Forwarded Headers Middleware is activated to run first in the middleware pipeline with a restricted configuration specific to the ASP.NET Core Module. The restricted configuration is due to trust concerns with forwarded headers, for example, IP spoofing. The middleware is configured to forward the X-Forwarded-For and X-Forwarded-Proto headers and is restricted to a single localhost proxy. If additional configuration is required, see the Forwarded Headers Middleware options.

Other proxy server and load balancer scenarios

Outside of using IIS Integration when hosting out-of-process, Forwarded Headers Middleware isn't enabled by default. Forwarded Headers Middleware must be enabled for an app to process forwarded headers with UseForwardedHeaders. After enabling the middleware if no ForwardedHeadersOptions are specified to the middleware, the default ForwardedHeadersOptions.ForwardedHeaders are ForwardedHeaders.None.

Configure the middleware with ForwardedHeadersOptions to forward the X-Forwarded-For and X-Forwarded-Proto headers.

Forwarded Headers Middleware order

Forwarded Headers Middleware should run before other middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing. Forwarded Headers Middleware can run after diagnostics and error handling, but it must be run before calling UseHsts:

C#

```
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseForwardedHeaders();
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
    app.UseForwardedHeaders();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Alternatively, call `UseForwardedHeaders` before diagnostics:

C#

```
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
});

var app = builder.Build();

app.UseForwardedHeaders();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

> **Note**
>
> If no ForwardedHeadersOptions are specified or applied directly to the extension method with UseForwardedHeaders, the default headers to forward are ForwardedHeaders.None. The ForwardedHeaders property must be configured with the headers to forward.

Nginx configuration

To forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers, see Host ASP.NET Core on Linux with Nginx. For more information, see NGINX: Using the Forwarded header.

Apache configuration

`X-Forwarded-For` is added automatically. For more information, see Apache Module mod_proxy: Reverse Proxy Request Headers.

Forwarded Headers Middleware options

ForwardedHeadersOptions control the behavior of the Forwarded Headers Middleware. The following example changes the default values:

- Limits the number of entries in the forwarded headers to `2`.
- Adds a known proxy address of `127.0.10.1`.
- Changes the forwarded header name from the default `X-Forwarded-For` to `X-Forwarded-For-My-Custom-Header-Name`.

C#

```
using System.Net;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardLimit = 2;
    options.KnownProxies.Add(IPAddress.Parse("127.0.10.1"));
    options.ForwardedForHeaderName = "X-Forwarded-For-My-Custom-Header-
Name";
});

var app = builder.Build();

app.UseForwardedHeaders();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Expand table

Option	Description
AllowedHosts	Restricts hosts by the `X-Forwarded-Host` header to the values provided. • Values are compared using ordinal-ignore-case.

Option	Description
	• Port numbers must be excluded. • If the list is empty, all hosts are allowed. • A top-level wildcard `*` allows all non-empty hosts. • Subdomain wildcards are permitted but don't match the root domain. For example, `*.contoso.com` matches the subdomain `foo.contoso.com` but not the root domain `contoso.com`. • Unicode host names are allowed but are converted to Punycode for matching. • IPv6 addresses must include bounding brackets and be in conventional form (for example, `[ABCD:EF01:2345:6789:ABCD:EF01:2345:6789]`). IPv6 addresses aren't special-cased to check for logical equality between different formats, and no canonicalization is performed. • Failure to restrict the allowed hosts may allow a cyberattacker to spoof links generated by the service. The default value is an empty `IList<string>`.
ForwardedForHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XForwardedForHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-For` header but uses some other header to forward the information. The default is `X-Forwarded-For`.
ForwardedHeaders	Identifies which forwarders should be processed. See the ForwardedHeaders Enum for the list of fields that apply. Typical values assigned to this property are `ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto`. The default value is ForwardedHeaders.None.
ForwardedHostHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XForwardedHostHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-Host` header but uses some other header to forward the information. The default is `X-Forwarded-Host`.
ForwardedProtoHeaderName	Use the header specified by this property instead of the one specified by

Option	Description
	ForwardedHeadersDefaults.XForwardedProtoHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-Proto` header but uses some other header to forward the information. The default is `X-Forwarded-Proto`.
ForwardLimit	Limits the number of entries in the headers that are processed. Set to `null` to disable the limit, but this should only be done if `KnownProxies` or `KnownNetworks` are configured. Setting a non-`null` value is a precaution (but not a guarantee) to protect against misconfigured proxies and malicious requests arriving from side-channels on the network. Forwarded Headers Middleware processes headers in reverse order from right to left. If the default value (`1`) is used, only the rightmost value from the headers is processed unless the value of `ForwardLimit` is increased. The default is `1`.
KnownNetworks	Address ranges of known networks to accept forwarded headers from. Provide IP ranges using Classless Interdomain Routing (CIDR) notation. If the server is using dual-mode sockets, IPv4 addresses are supplied in an IPv6 format (for example, `10.0.0.1` in IPv4 represented in IPv6 as `::ffff:10.0.0.1`). See IPAddress.MapToIPv6. Determine if this format is required by looking at the HttpContext.Connection.RemoteIpAddress. The default is an `IList`<IPNetwork> containing a single entry for `new IPNetwork(IPAddress.Loopback, 8)`.
KnownProxies	Addresses of known proxies to accept forwarded headers from. Use `KnownProxies` to specify exact IP address matches. If the server is using dual-mode sockets, IPv4 addresses are supplied in an IPv6 format (for example, `10.0.0.1` in IPv4 represented in IPv6 as `::ffff:10.0.0.1`). See IPAddress.MapToIPv6. Determine if this format is required by looking at the HttpContext.Connection.RemoteIpAddress. The default is an `IList`<IPAddress> containing a single entry for `IPAddress.IPv6Loopback`.
OriginalForHeaderName	Use the header specified by this property instead of the one specified by

Option	Description
	ForwardedHeadersDefaults.XOriginalForHeaderName. The default is `X-Original-For`.
OriginalHostHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XOriginalHostHeaderName. The default is `X-Original-Host`.
OriginalProtoHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XOriginalProtoHeaderName. The default is `X-Original-Proto`.
RequireHeaderSymmetry	Require the number of header values to be in sync between the ForwardedHeadersOptions.ForwardedHeaders being processed. The default in ASP.NET Core 1.x is `true`. The default in ASP.NET Core 2.0 or later is `false`.

Scenarios and use cases

When it isn't possible to add forwarded headers and all requests are secure

In some cases, it might not be possible to add forwarded headers to the requests proxied to the app. If the proxy is enforcing that all public external requests are HTTPS, the scheme can be manually set before using any type of middleware:

C#

```
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
});

var app = builder.Build();
```

```
app.Use((context, next) =>
{
    context.Request.Scheme = "https";
    return next(context);
});

app.UseForwardedHeaders();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

This code can be disabled with an environment variable or other configuration setting in a development or staging environment:

C#

```
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
});

var app = builder.Build();

if (!app.Environment.IsProduction())
{
    app.Use((context, next) =>
    {
        context.Request.Scheme = "https";
        return next(context);
    });
}

app.UseForwardedHeaders();

if (!app.Environment.IsDevelopment())
```

```
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Work with path base and proxies that change the request path

Some proxies pass the path intact but with an app base path that should be removed so that routing works properly. UsePathBaseExtensions.UsePathBase middleware splits the path into HttpRequest.Path and the app base path into HttpRequest.PathBase.

If `/foo` is the app base path for a proxy path passed as `/foo/api/1`, the middleware sets `Request.PathBase` to `/foo` and `Request.Path` to `/api/1` with the following command:

C#

```
app.UsePathBase("/foo");
// ...
app.UseRouting();
```

> **Note**
>
> When using WebApplication (see Migrate from ASP.NET Core 5.0 to 6.0), app.UseRouting must be called after `UsePathBase` so that the routing middleware can observe the modified path before matching routes. Otherwise, routes are matched before the path is rewritten by `UsePathBase` as described in the Middleware Ordering and Routing articles.

The original path and path base are reapplied when the middleware is called again in reverse. For more information on middleware order processing, see ASP.NET Core Middleware.

If the proxy trims the path (for example, forwarding `/foo/api/1` to `/api/1`), fix redirects and links by setting the request's PathBase property:

C#

```
app.Use((context, next) =>
{
    context.Request.PathBase = new PathString("/foo");
    return next(context);
});
```

If the proxy is adding path data, discard part of the path to fix redirects and links by using StartsWithSegments and assigning to the Path property:

C#

```
app.Use((context, next) =>
{
    if (context.Request.Path.StartsWithSegments("/foo", out var remainder))
    {
        context.Request.Path = remainder;
    }

    return next(context);
});
```

Configuration for a proxy that uses different header names

If the proxy doesn't use headers named `X-Forwarded-For` and `X-Forwarded-Proto` to forward the proxy address/port and originating scheme information, set the ForwardedForHeaderName and ForwardedProtoHeaderName options to match the header names used by the proxy:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedForHeaderName = "HeaderNamUsedByProxy_X-Forwarded-
For_Header";
    options.ForwardedProtoHeaderName = "HeaderNamUsedByProxy_X-Forwarded-
Proto_Header";
});

var app = builder.Build();

app.UseForwardedHeaders();
```

```
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Forward the scheme for Linux and non-IIS reverse proxies

Apps that call UseHttpsRedirection and UseHsts put a site into an infinite loop if deployed to an Azure Linux App Service, Azure Linux virtual machine (VM), or behind any other reverse proxy besides IIS. TLS is terminated by the reverse proxy, and Kestrel isn't made aware of the correct request scheme. OAuth and OIDC also fail in this configuration because they generate incorrect redirects. UseIISIntegration adds and configures Forwarded Headers Middleware when running behind IIS, but there's no matching automatic configuration for Linux (Apache or Nginx integration).

To forward the scheme from the proxy in non-IIS scenarios, enable the Forwarded Headers Middleware by setting `ASPNETCORE_FORWARDEDHEADERS_ENABLED` to `true`. Warning: This flag uses settings designed for cloud environments and doesn't enable features such as the KnownProxies option to restrict which IPs forwarders are accepted from.

Certificate forwarding

Azure

To configure Azure App Service for certificate forwarding, see Configure TLS mutual authentication for Azure App Service. The following guidance pertains to configuring the ASP.NET Core app.

- Configure Certificate Forwarding Middleware to specify the header name that Azure uses. Add the following code to configure the header from which the middleware builds a certificate.
- Call UseCertificateForwarding before the call to UseAuthentication.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddCertificateForwarding(options =>
    options.CertificateHeader = "X-ARR-ClientCert");

var app = builder.Build();

app.UseCertificateForwarding();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();
app.UseAuthentication();

app.MapRazorPages();

app.Run();
```

Other web proxies

If a proxy is used that isn't IIS or Azure App Service's Application Request Routing (ARR), configure the proxy to forward the certificate that it received in an HTTP header.

- Configure Certificate Forwarding Middleware to specify the header name. Add the following code to configure the header from which the middleware builds a certificate.
- Call UseCertificateForwarding before the call to UseAuthentication.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddCertificateForwarding(options =>
    options.CertificateHeader = "YOUR_CERTIFICATE_HEADER_NAME");

var app = builder.Build();

app.UseCertificateForwarding();
```

```
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();
app.UseAuthentication();

app.MapRazorPages();

app.Run();
```

If the proxy isn't base64-encoding the certificate, as is the case with Nginx, set the `HeaderConverter` option. Consider the following example:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddCertificateForwarding(options =>
{
    options.CertificateHeader = "YOUR_CUSTOM_HEADER_NAME";
    options.HeaderConverter = (headerValue) =>
    {
        // Conversion logic to create an X509Certificate2.
        var clientCertificate = ConversionLogic.CreateAnX509Certificate2();
        return clientCertificate;
    };
});

var app = builder.Build();

app.UseCertificateForwarding();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();
app.UseAuthentication();

app.MapRazorPages();
```

```
app.Run();
```

Troubleshoot

When headers aren't forwarded as expected, enable `debug` level logging and HTTP request logging. UseHttpLogging must be called after UseForwardedHeaders:

C#

```
using Microsoft.AspNetCore.HttpLogging;
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddHttpLogging(options =>
{
    options.LoggingFields = HttpLoggingFields.RequestPropertiesAndHeaders;
});

builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
});

var app = builder.Build();

app.UseForwardedHeaders();
app.UseHttpLogging();

app.Use(async (context, next) =>
{
    // Connection: RemoteIp
    app.Logger.LogInformation("Request RemoteIp: {RemoteIpAddress}",
        context.Connection.RemoteIpAddress);

    await next(context);
});

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
```

```
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

If there are multiple values in a given header, Forwarded Headers Middleware processes headers in reverse order from right to left. The default `ForwardLimit` is `1` (one), so only the rightmost value from the headers is processed unless the value of `ForwardLimit` is increased.

The request's original remote IP must match an entry in the KnownProxies or KnownNetworks lists before forwarded headers are processed. This limits header spoofing by not accepting forwarders from untrusted proxies. When an unknown proxy is detected, logging indicates the address of the proxy:

Console

```
September 20th 2018, 15:49:44.168 Unknown proxy: 10.0.0.100:54321
```

In the preceding example, 10.0.0.100 is a proxy server. If the server is a trusted proxy, add the server's IP address to `KnownProxies`, or add a trusted network to `KnownNetworks`. For more information, see the Forwarded Headers Middleware options section.

C#

```
using Microsoft.AspNetCore.HttpOverrides;
using System.Net;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseForwardedHeaders();
    app.UseHsts();
}
else
```

```
{
    app.UseDeveloperExceptionPage();
    app.UseForwardedHeaders();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

To display the logs, add `"Microsoft.AspNetCore.HttpLogging": "Information"` to the `appsettings.Development.json` file:

JSON

```
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.HttpLogging": "Information"
    }
  }
}
```

ⓘ Important

Only allow trusted proxies and networks to forward headers. Otherwise, IP spoofing attacks are possible.

Additional resources

- Host ASP.NET Core in a web farm
- Microsoft Security Advisory CVE-2018-0787: ASP.NET Core Elevation Of Privilege Vulnerability
- YARP: Yet Another Reverse Proxy

Deploying and scaling an ASP.NET Core app on Azure Container Apps

Article • 09/18/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

Apps deployed to Azure that experience intermittent high demand benefit from scalability to meet demand. Scalable apps can scale out to ensure capacity during workload peaks and then scale down automatically when the peak drops, which can lower costs. Horizontal scaling (scaling out) adds new instances of a resource, such as VMs or database replicas. This article demonstrates how to deploy a horizontally scalable ASP.NET Core app to Azure container apps by completing the following tasks:

1. Set up the sample project
2. Deploy the app to Azure Container Apps
3. Scale and troubleshoot the app
4. Create the Azure Services
5. Connect the Azure services
6. Configure and redeploy the app

This article uses Razor Pages, but most of it applies to other ASP.NET Core apps.

In some cases, basic ASP.NET Core apps are able to scale without special considerations. However, apps that utilize certain framework features or architectural patterns require extra configurations, including the following:

- **Secure form submissions**: Razor Pages, MVC and Web API apps often rely on form submissions. By default these apps use cross site forgery tokens and internal data protection services to secure requests. When deployed to the cloud, these apps must be configured to manage data protection service concerns in a secure, centralized location.
- **SignalR circuits**: Blazor Server apps require the use of a centralized Azure SignalR service in order to securely scale. These services also utilize the data protection

services mentioned previously.

- **Centralized caching or state management services**: Scalable apps may use Azure Cache for Redis to provide distributed caching. Azure storage may be needed to store state for frameworks such as Microsoft Orleans, which can help write apps that manage state across many different app instances.

The steps in this article demonstrate how to properly address the preceding concerns by deploying a scalable app to Azure Container Apps. Most of the concepts in this tutorial also apply when scaling Azure App Service instances.

Set up the sample project

Use the GitHub Explorer sample app to follow along with this tutorial. Clone the app from GitHub using the following command:

.NET CLI

```
git clone "https://github.com/dotnet/AspNetCore.Docs.Samples.git"
```

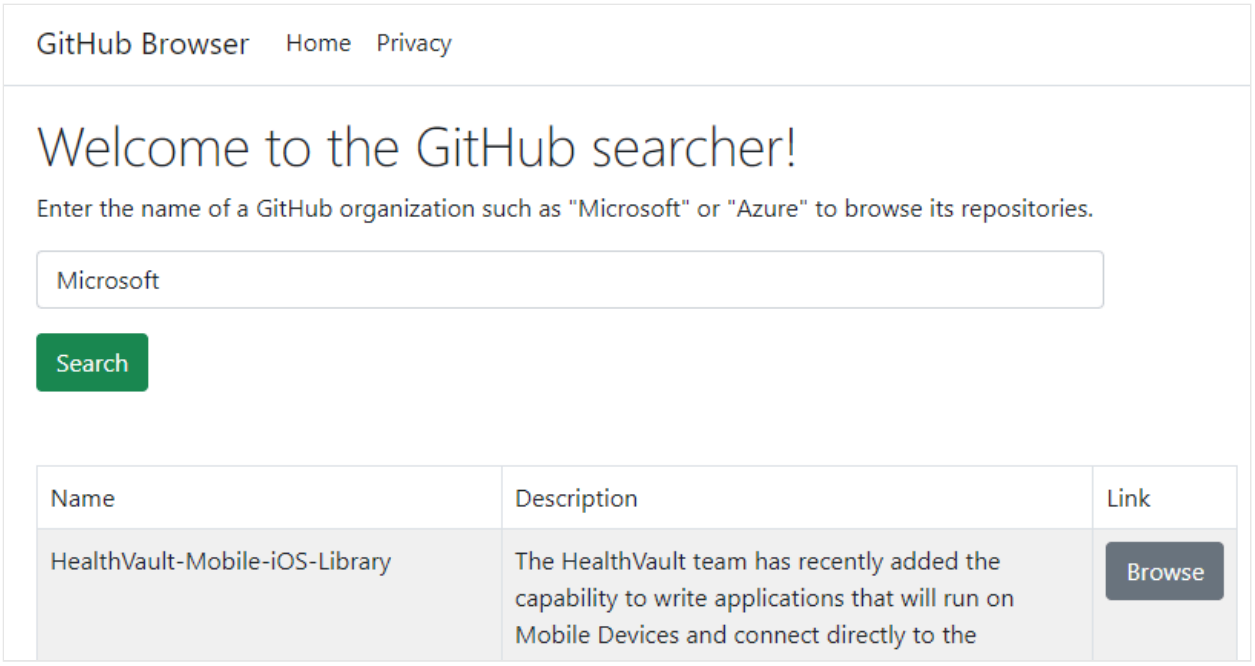
Navigate to the `/tutorials/scalable-razor-apps/start` folder and open the `ScalableRazor.csproj`.

The sample app uses a search form to browse GitHub repositories by name. The form relies on the built-in ASP.NET Core data protection services to handle antiforgery concerns. By default, when the app scales horizontally on Container Apps, the data protection service throws an exception.

Test the app

1. Launch the app in Visual Studio. The project includes a Docker file, which means that the arrow next to the run button can be selected to start the app using either a Docker Desktop setup or the standard ASP.NET Core local web server.

Use the search form to browse for GitHub repositories by name.

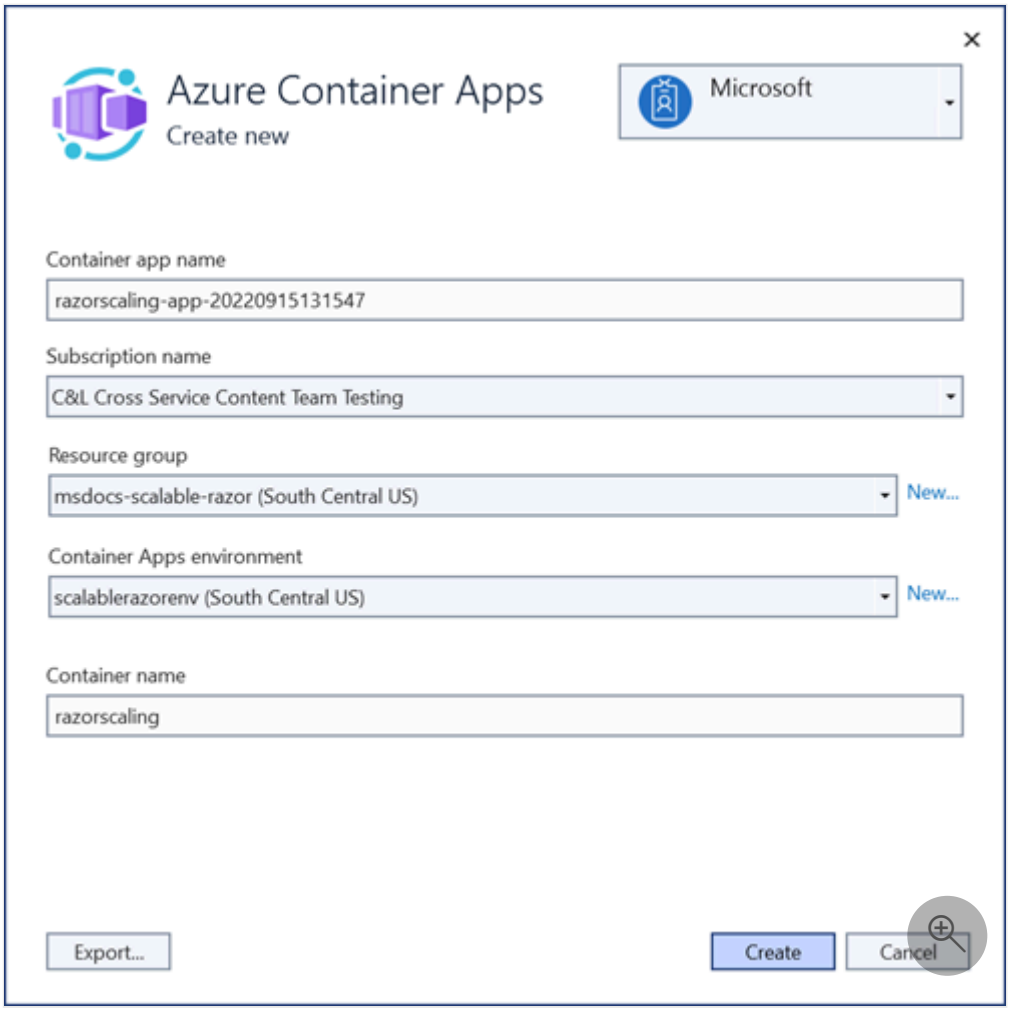


Deploy the app to Azure Container Apps

Visual Studio is used to deploy the app to Azure Container Apps. Container apps provide a managed service designed to simplify hosting containerized apps and microservices.

> ⓘ **Note**
>
> Many of the resources created for the app require a location. For this app, location isn't important. A real app should select a location closest to the clients. You may want to select a location near you.

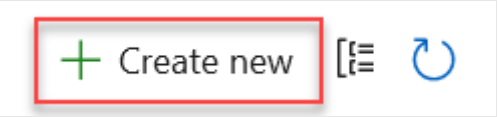
1. In Visual Studio solution explorer, right click on the top level project node and select **Publish**.
2. In the publishing dialog, select **Azure** as the deployment target, and then select **Next**.
3. For the specific target, select **Azure Container Apps (Linux)**, and then select **Next**.
4. Create a new container app to deploy to. Select the green + icon to open a new dialog and enter the following values:



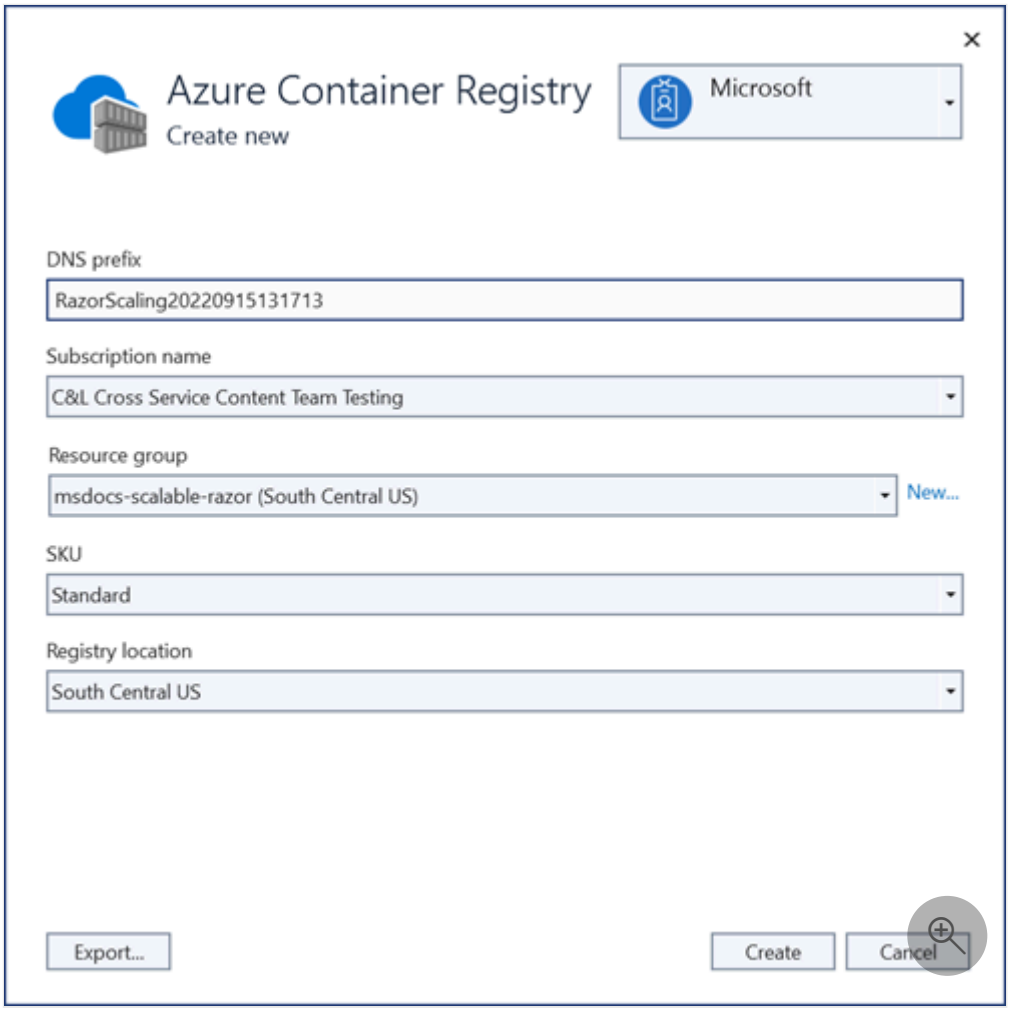
- **Container app name**: Leave the default value or enter a name.
- **Subscription name**: Select the subscription to deploy to.
- **Resource group**: Select **New** and create a new resource group called *msdocs-scalable-razor*.
- **Container apps environment**: Select **New** to open the container apps environment dialog and enter the following values:
  - **Environment name**: Keep the default value.
  - **Location**: Select a location near you.
  - **Azure Log Analytics Workspace**: Select **New** to open the log analytics workspace dialog.
    - **Name**: Leave the default value.
    - **Location**: Select a location near you and then select **Ok** to close the dialog.
  - Select **Ok** to close the container apps environment dialog.
- Select **Create** to close the original container apps dialog. Visual Studio creates the container app resource in Azure.

5. Once the resource is created, make sure it's selected in the list of container apps, and then select **Next**.

6. You'll need to create an Azure Container Registry to store the published image artifact for your app. Select the green + icon on the container registry screen.



7. Leave the default values, and then select **Create**.



8. After the container registry is created, make sure it's selected, and then select finish to close the dialog workflow and display a summary of the publishing profile.

   If Visual Studio prompts you to enable the Admin user to access the published docker container, select **Yes**.

9. Select **Publish** in the upper right of the publishing profile summary to deploy the app to Azure.

When the deployment finishes, Visual Studio launches the browser to display the hosted app. Search for `Microsoft` in the form field, and a list of repositories is displayed.

Scale and troubleshoot the app

The app is currently working without any issues, but we'd like to scale the app across more instances in anticipation of high traffic volumes.

1. In the Azure portal, search for the `razorscaling-app-****` container app in the top level search bar and select it from the results.
2. On the overview page, select **Scale** from the left navigation, and then select **+ Edit and deploy**.
3. On the revisions page, switch to the **Scale** tab.
4. Set both the min and max instances to **4** and then select **Create**. This configuration change guarantees your app is scaled horizontally across four instances.

Navigate back to the app. When the page loads, at first it appears everything is working correctly. However, when a search term is entered and submitted, an error may occur. If an error is not displayed, submit the form several more times.

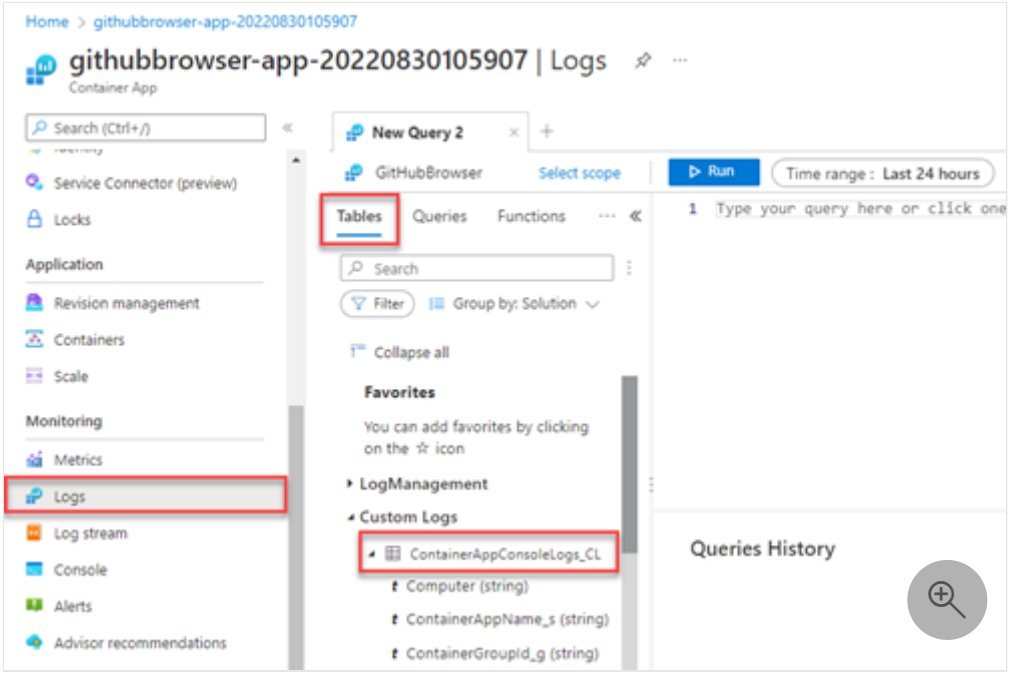
Troubleshooting the error

It's not immediately apparent why the search requests are failing. The browser tools indicate a 400 Bad Request response was sent back. However, you can use the logging features of container apps to diagnose errors occurring in your environment.

1. On the overview page of the container app, select **Logs** from the left navigation.

2. On the **Logs** page, close the pop-up that opens and navigate to the **Tables** tab.

3. Expand the **Custom Logs** item to reveal the **ContainerAppConsoleLogs_CL** node. This table holds various logs for the container app that can be queried to troubleshoot problems.

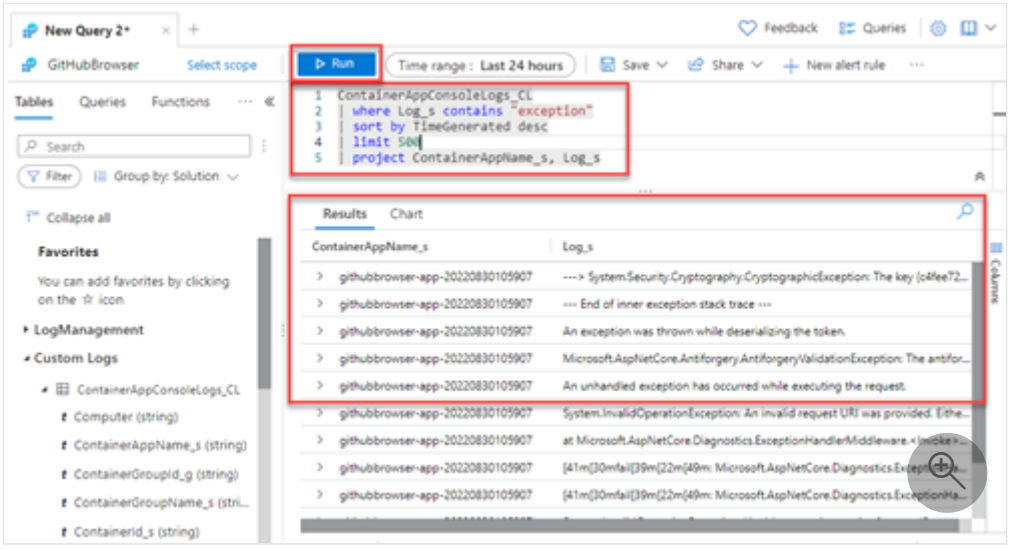


4. In the query editor, compose a basic query to search the **ContainerAppConsoleLogs_CL Logs** table for recent exceptions, such as the following script:

```KQL
ContainerAppConsoleLogs_CL
| where Log_s contains "exception"
| sort by TimeGenerated desc
| limit 500
| project ContainerAppName_s, Log_s
```

   The preceding query searches the **ContainerAppConsoleLogs_CL** table for any rows that contain the word exception. The results are ordered by the time generated, limited to 500 results, and only include the **ContainerAppName_s** and **Log_s** columns to make the results easier to read.

5. Select **Run**, a list of results is displayed. Read through the logs and note that most of them are related to antiforgery tokens and cryptography.



> **ⓘ Important**
>
> The errors in the app are caused by the .NET data protection services. When multiple instances of the app are running, there is no guarantee that the HTTP POST request to submit the form is routed to the same container that initially loaded the page from the HTTP GET request. If the requests are handled by different instances, the antiforgery tokens aren't handled correctly and an exception occurs.

   In the steps ahead, this issue is resolved by centralizing the data protection keys in an Azure storage service and protecting them with Key Vault.

Create the Azure Services

To resolve the preceding errors, the following services are created and connected to the app:

- Azure Storage Account: Handles storing data for the Data Protection Services. Provides a centralized location to store key data as the app scales. Storage accounts can also be used to hold documents, queue data, file shares, and almost any type of blob data.
- Azure KeyVault: This service stores secrets for an app, and is used to help manage encryption concerns for the Data Protection Services.

Create the storage account service

1. In the Azure portal search bar, enter `Storage accounts` and select the matching result.
2. On the storage accounts listing page, select **+ Create**.
3. On the **Basics** tab, enter the following values:

    - **Subscription**: Select the same subscription that you chose for the container app.
    - **Resource Group**: Select the *msdocs-scalable-razor* resource group you created previously.
    - **Storage account name**: Name the account *scalablerazorstorageXXXX* where the X's are random numbers of your choosing. This name must be unique across all of Azure.
    - **Region**: Select the same region you previously selected.

4. Leave the rest of the values at their default and select **Review**. After Azure validates the inputs, select **Create**.

Azure provisions the new storage account. When the task completes, choose **Go to resource** to view the new service.

Create the storage container

Create a Container to store the app's data protection keys.

1. On the overview page for the new storage account, select **Storage browser** on the left navigation.
2. Select **Blob containers**.
3. Select **+ Add container** to open the **New container** flyout menu.

4. Enter a name of *scalablerazorkeys*, leave the rest of the settings at their defaults, and then select **Create**.

The new containers appear on the page list.

Create the key vault service

Create a key vault to hold the keys that protect the data in the blob storage container.

1. In the Azure portal search bar, enter `Key Vault` and select the matching result.
2. On the key vault listing page, select **+ Create**.
3. On the **Basics** tab, enter the following values:

    - **Subscription**: Select the same subscription that was previously selected.
    - **Resource Group**: Select the *msdocs-scalable-razor* resource group previously created.
    - **Key Vault name**: Enter the name *scalablerazorvaultXXXX*.
    - **Region**: Select a region near your location.

4. Leave the rest of the settings at their default, and then select **Review + create**. Wait for Azure to validate your settings, and then select **Create**.

Azure provisions the new key vault. When the task completes, select **Go to resource** to view the new service.

Create the key

Create a secret key to protect the data in the blob storage account.

1. On the main overview page of the key vault, select **Keys** from the left navigation.
2. On the **Create a key** page, select **+ Generate/Import** to open the **Create a key** flyout menu.
3. Enter *razorkey* in the **Name** field. Leave the rest of the settings at their default values and then select **Create**. A new key appears on the key list page.

Connect the Azure Services

The Container App requires a secure connection to the storage account and the key vault services in order to resolve the data protection errors and scale correctly. The new services are connected together using the following steps:

> **ⓘ Important**

> Security role assignments through Service Connector and other tools typically take a minute or two to propagate, and in some rare cases can take up to eight minutes.

Connect the storage account

1. In the Azure portal, navigate to the Container App overview page.
2. On the left navigation, select **Service connector**
3. On the Service Connector page, choose **+ Create** to open the **Creation Connection** flyout panel and enter the following values:

    - **Container**: Select the Container App created previously.
    - **Service type**: Choose **Storage - blob**.
    - **Subscription**: Select the subscription previously used.
    - **Connection name**: Leave the default value.
    - **Storage account**: Select the storage account created previously.
    - **Client type**: Select **.NET**.

4. Select **Next: Authentication** to progress to the next step.
5. Select **System assigned managed identity** and choose **Next: Networking**.
6. Leave the default networking options selected, and then select **Review + Create**.
7. After Azure validates the settings, select **Create**.

The service connector enables a system-assigned managed identity on the container app. It also assigns a role of **Storage Blob Data Contributor** to the identity so it can perform data operations on the storage containers.

Connect the key vault

1. In the Azure portal, navigate to your Container App overview page.
2. On the left navigation, select **Service connector**.
3. On the Service Connector page, choose **+ Create** to open the **Creation Connection** flyout panel and enter the following values:

    - **Container**: Select the container app created previously.
    - **Service type**: Choose **Key Vault**.
    - **Subscription**: Select the subscription previously used.
    - **Connection name**: Leave the default value.
    - **Key vault**: Select the key vault created previously.
    - **Client type**: Select **.NET**.

4. Select **Next: Authentication** to progress to the next step.
5. Select **System assigned managed identity** and choose **Next: Networking**.

6. Leave the default networking options selected, and then select **Review + Create**.
7. After Azure validates the settings, select **Create**.

The service connector assigns a role to the identity so it can perform data operations on the key vault keys.

Configure and redeploy the app

The necessary Azure resources have been created. In this section the app code is configured to use the new resources.

1. Install the following NuGet packages:

    - **Azure.Identity**: Provides classes to work with the Azure identity and access management services.
    - **Microsoft.Extensions.Azure**: Provides helpful extension methods to perform core Azure configurations.
    - **Azure.Extensions.AspNetCore.DataProtection.Blobs**: Allows storing ASP.NET Core DataProtection keys in Azure Blob Storage so that keys can be shared across several instances of a web app.
    - **Azure.Extensions.AspNetCore.DataProtection.Keys**: Enables protecting keys at rest using the Azure Key Vault Key Encryption/Wrapping feature.

    .NET CLI

    ```
    dotnet add package Azure.Identity
    dotnet add package Microsoft.Extensions.Azure
    dotnet add package Azure.Extensions.AspNetCore.DataProtection.Blobs
    dotnet add package Azure.Extensions.AspNetCore.DataProtection.Keys
    ```

2. Update `Program.cs` with the following highlighted code:

    C#

    ```
    using Azure.Identity;
    using Microsoft.AspNetCore.DataProtection;
    using Microsoft.Extensions.Azure;
    var builder = WebApplication.CreateBuilder(args);
    var BlobStorageUri = builder.Configuration["AzureURIs:BlobStorage"];
    var KeyVaultURI = builder.Configuration["AzureURIs:KeyVault"];

    builder.Services.AddRazorPages();
    builder.Services.AddHttpClient();
    builder.Services.AddServerSideBlazor();
    ```

```
builder.Services.AddAzureClientsCore();

builder.Services.AddDataProtection()
                .PersistKeysToAzureBlobStorage(new Uri(BlobStorageUri),
                                               new
DefaultAzureCredential())
                .ProtectKeysWithAzureKeyVault(new Uri(KeyVaultURI),
                                               new
DefaultAzureCredential());
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding changes allow the app to manage data protection using a centralized, scalable architecture. `DefaultAzureCredential` discovers the managed identity configurations enabled earlier when the app is redeployed.

Update the placeholders in `AzureURIs` section of the `appsettings.json` file to include the following:

1. Replace the `<storage-account-name>` placeholder with the name of the `scalablerazorstorageXXXX` storage account.
2. Replace the `<container-name>` placeholder with the name of the `scalablerazorkeys` storage container.
3. Replace the `<key-vault-name>` placeholder with the name of the `scalablerazorvaultXXXX` key vault.
4. Replace the `<key-name>` placeholder in the key vault URI with the `razorkey` name created previously.

```
C#
```

```
{
  "GitHubURL": "https://api.github.com",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "AzureURIs": {
    "BlobStorage": "https://<storage-account-
name>.blob.core.windows.net/<container-name>/keys.xml",
    "KeyVault": "https://<key-vault-name>.vault.azure.net/keys/<key-
name>/"
  }
}
```

Redeploy the app

The app is now configured correctly to use the Azure services created previously. Redeploy the app for the code changes to be applied.

1. Right click on the project node in the solution explorer and select **Publish**.
2. On the publishing profile summary view, select the **Publish** button in the upper right corner.

Visual Studio redeploys the app to the container apps environment created previously. When the processes finished, the browser launches to the app home page.

Test the app again by searching for *Microsoft* in the search field. The page should now reload with the correct results every time you submit.

Configure roles for local development

The existing code and configuration of the app can also work while running locally during development. The `DefaultAzureCredential` class configured previously is able to pick up local environment credentials to authenticate to Azure Services. You'll need to assign the same roles to your own account that were assigned to your app's managed identity in order for the authentication to work. This should be the same account you use to log into Visual Studio or the Azure CLI.

Sign-in to your local development environment

You'll need to be signed in to the Azure CLI, Visual Studio, or Azure PowerShell for your credentials to be picked up by `DefaultAzureCredential`.

Azure CLI

Azure CLI

```
az login
```

Assign roles to your developer account

1. In the Azure portal, navigate to the `scalablerazor****` storage account created previously.
2. Select **Access Control (IAM)** from the left navigation.
3. Choose **+ Add** and then **Add role assignment** from the drop-down menu.
4. On the **Add role assignment** page, search for `Storage blob data contributor`, select the matching result, and then select **Next**.
5. Make sure **User, group, or service principal** is selected, and then select **+ Select members**.
6. In the **Select members** flyout, search for your own *user@domain* account and select it from the results.
7. Choose **Next** and then select **Review + assign**. After Azure validates the settings, select **Review + assign** again.

As previously stated, role assignment permissions might take a minute or two to propagate, or in rare cases up to eight minutes.

Repeat the previous steps to assign a role to your account so that it can access the key vault service and secret.

1. In the Azure portal, navigate to the `razorscalingkeys` key vault created previously.
2. Select **Access Control (IAM)** from the left navigation.
3. Choose **+ Add** and then **Add role assignment** from the drop-down menu.
4. On the **Add role assignment** page, search for `Key Vault Crypto Service Encryption User`, select the matching result, and then select **Next**.
5. Make sure **User, group, or service principal** is selected, and then select **+ Select members**.
6. In the **Select members** flyout, search for your own *user@domain* account and select it from the results.

7. Choose **Next** and then select **Review + assign**. After Azure validates your settings, select **Review + assign** again.

You may need to wait again for this role assignment to propagate.

You can then return to Visual Studio and run the app locally. The code should continue to function as expected. `DefaultAzureCredential` uses your existing credentials from Visual Studio or the Azure CLI.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Host ASP.NET Core in a web farm

Article • 09/18/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Chris Ross

A *web farm* is a group of two or more web servers (or *nodes*) that host multiple instances of an app. When requests from users arrive to a web farm, a *load balancer* distributes the requests to the web farm's nodes. Web farms improve:

- **Reliability/availability**: When one or more nodes fail, the load balancer can route requests to other functioning nodes to continue processing requests.
- **Capacity/performance**: Multiple nodes can process more requests than a single server. The load balancer balances the workload by distributing requests to the nodes.
- **Scalability**: When more or less capacity is required, the number of active nodes can be increased or decreased to match the workload. Web farm platform technologies, such as Azure App Service, can automatically add or remove nodes at the request of the system administrator or automatically without human intervention.
- **Maintainability**: Nodes of a web farm can rely on a set of shared services, which results in easier system management. For example, the nodes of a web farm can rely upon a single database server and a common network location for static resources, such as images and downloadable files.

This topic describes configuration and dependencies for ASP.NET core apps hosted in a web farm that rely upon shared resources.

General configuration

Host and deploy ASP.NET Core

Learn how to set up hosting environments and deploy ASP.NET Core apps. Configure a process manager on each node of the web farm to automate app starts and restarts.

Each node requires the ASP.NET Core runtime. For more information, see the topics in the Host and deploy area of the documentation.

Configure ASP.NET Core to work with proxy servers and load balancers
Learn about configuration for apps hosted behind proxy servers and load balancers, which often obscure important request information.

Deploy ASP.NET Core apps to Azure App Service
Azure App Service is a Microsoft cloud computing platform service for hosting web apps, including ASP.NET Core. App Service is a fully managed platform that provides automatic scaling, load balancing, patching, and continuous deployment.

App data

When an app is scaled to multiple instances, there might be app state that requires sharing across nodes. If the state is transient, consider sharing an IDistributedCache. If the shared state requires persistence, consider storing the shared state in a database.

Required configuration

Data Protection and Caching require configuration for apps deployed to a web farm.

Data Protection

The ASP.NET Core Data Protection system is used by apps to protect data. Data Protection relies upon a set of cryptographic keys stored in a *key ring*. When the Data Protection system is initialized, it applies default settings that store the key ring locally. Under the default configuration, a unique key ring is stored on each node of the web farm. Consequently, each web farm node can't decrypt data that's encrypted by an app on any other node. The default configuration isn't generally appropriate for hosting apps in a web farm. An alternative to implementing a shared key ring is to always route user requests to the same node. For more information on Data Protection system configuration for web farm deployments, see Configure ASP.NET Core Data Protection.

Caching

In a web farm environment, the caching mechanism must share cached items across the web farm's nodes. Caching must either rely upon a common Redis cache, a shared SQL Server database, or a custom caching implementation that shares cached items across the web farm. For more information, see Distributed caching in ASP.NET Core.

Dependent components

The following scenarios don't require additional configuration, but they depend on technologies that require configuration for web farms.

Expand table

Scenario	Depends on ...
Authentication	Data Protection (see Configure ASP.NET Core Data Protection). For more information, see Use cookie authentication without ASP.NET Core Identity and Share authentication cookies among ASP.NET apps.
Identity	Authentication and database configuration. For more information, see Introduction to Identity on ASP.NET Core.
Session	Data Protection (encrypted cookies) (see Configure ASP.NET Core Data Protection) and Caching (see Distributed caching in ASP.NET Core). For more information, see Session and state management: Session state.
TempData	Data Protection (encrypted cookies) (see Configure ASP.NET Core Data Protection) or Session (see Session and state management: Session state). For more information, see Session and state management: TempData.
Antiforgery	Data Protection (see Configure ASP.NET Core Data Protection). For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Troubleshoot

Data Protection and caching

When Data Protection or caching isn't configured for a web farm environment, intermittent errors occur when requests are processed. This occurs because nodes don't share the same resources and user requests aren't always routed back to the same node.

Consider a user who signs into the app using cookie authentication. The user signs into the app on one web farm node. If their next request arrives at the same node where they signed in, the app is able to decrypt the authentication cookie and allows access to the app's resource. If their next request arrives at a different node, the app can't decrypt

the authentication cookie from the node where the user signed in, and authorization for the requested resource fails.

When any of the following symptoms occur **intermittently**, the problem is usually traced to improper Data Protection or caching configuration for a web farm environment:

- Authentication breaks: The authentication cookie is misconfigured or can't be decrypted. OAuth (Facebook, Microsoft, Twitter) or OpenIdConnect logins fail with the error "Correlation failed."
- Authorization breaks: Identity is lost.
- Session state loses data.
- Cached items disappear.
- TempData fails.
- POSTs fail: The antiforgery check fails.

For more information on Data Protection configuration for web farm deployments, see Configure ASP.NET Core Data Protection. For more information on caching configuration for web farm deployments, see Distributed caching in ASP.NET Core.

Obtain data from apps

If the web farm apps are capable of responding to requests, obtain request, connection, and additional data from the apps using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Additional resources

- Custom Script Extension for Windows: Downloads and executes scripts on Azure virtual machines, which is useful for post-deployment configuration and software installation.
- Configure ASP.NET Core to work with proxy servers and load balancers

Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment

Article • 11/26/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Sayed Ibrahim Hashimi and Rick Anderson

This document focuses on using Visual Studio 2022 or later to create and use publish profiles. The publish profiles created with Visual Studio can be used with MSBuild and Visual Studio. For instructions on publishing to Azure, see Publish an ASP.NET Core app to Azure with Visual Studio.

For the most current and detailed information on:

- Publishing with Visual studio, see Overview of Visual Studio Publish
- MSBuild, see MSBuild
- Publishing with MSBuild, see Microsoft.NET.Sdk.Publish

The `dotnet new mvc` command produces a project file containing the following root-level <Project> element:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">
    <!-- omitted for brevity -->
</Project>
```

The preceding `<Project>` element's `Sdk` attribute imports the MSBuild properties and targets from *$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.props* and *$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.targets*, respectively. The default location for `$(MSBuildSDKsPath)` (with Visual Studio 2022) is the *%programfiles%\Microsoft Visual Studio\2022\Preview\MSBuild\Sdks* folder.

`Microsoft.NET.Sdk.Web` (Web SDK) depends on other SDKs, including `Microsoft.NET.Sdk` (.NET Core SDK) and `Microsoft.NET.Sdk.Razor` (Razor SDK). The MSBuild properties and targets associated with each dependent SDK are imported. Publish targets import the appropriate set of targets based on the publish method used.

When MSBuild or Visual Studio loads a project, the following high-level actions occur:

- Build project
- Compute files to publish
- Publish files to destination

Compute project items

When the project is loaded, the MSBuild project items (files) are computed. The item type determines how the file is processed. By default, `.cs` files are included in the `Compile` item list. Files in the `Compile` item list are compiled.

The `Content` item list contains files that are published in addition to the build outputs. By default, files matching the patterns `wwwroot\**`, `**\*.config`, and `**\*.json` are included in the `Content` item list. For example, the `wwwroot\**` globbing pattern matches all files in the *wwwroot* folder and its subfolders.

The Web SDK imports the Razor SDK. As a result, files matching the patterns `**\*.cshtml` and `**\*.razor` are also included in the `Content` item list.

To explicitly add a file to the publish list, add the file directly in the `.csproj` file as shown in the Include Files section.

When selecting the **Publish** button in Visual Studio or when publishing from the command line:

- The properties/items are computed (the files that are needed to build).
- **Visual Studio only**: NuGet packages are restored. (Restore needs to be explicit by the user on the CLI.)
- The project builds.
- The publish items are computed (the files that are needed to publish).
- The project is published (the computed files are copied to the publish destination).

When an ASP.NET Core project references `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.

Basic command-line publishing

Command-line publishing works on all .NET Core-supported platforms and doesn't require Visual Studio. In the following examples, the .NET CLI's dotnet publish command is run from the project directory (which contains the `.csproj` file). If the project folder isn't the current working directory, explicitly pass in the project file path. For example:

.NET CLI

```
dotnet publish C:\Webs\Web1
```

Run the following commands to create and publish a web app:

.NET CLI

```
dotnet new mvc
dotnet publish
```

The `dotnet publish` command produces a variation of the following output:

Console

```
C:\Webs\Web1>dotnet publish
Microsoft (R) Build Engine version {VERSION} for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 36.81 ms for C:\Webs\Web1\Web1.csproj.
  Web1 -> C:\Webs\Web1\bin\Debug\{TARGET FRAMEWORK MONIKER}\Web1.dll
  Web1 -> C:\Webs\Web1\bin\Debug\{TARGET FRAMEWORK MONIKER}\Web1.Views.dll
  Web1 -> C:\Webs\Web1\bin\Debug\{TARGET FRAMEWORK MONIKER}\publish\
```

The default publish folder format is `bin\Debug\{TARGET FRAMEWORK MONIKER}`. For example, `bin\Release\net9.0\`

The following command specifies a `Release` build and the publishing directory:

.NET CLI

```
dotnet publish -c Release -o C:\MyWebs\test
```

The `dotnet publish` command calls MSBuild, which invokes the `Publish` target. Any parameters passed to `dotnet publish` are passed to MSBuild. The `-c` and `-o` parameters map to MSBuild's `Configuration` and `OutputPath` properties, respectively.

MSBuild properties can be passed using either of the following formats:

- `-p:<NAME>=<VALUE>`
- `/p:<NAME>=<VALUE>`

For example, the following command publishes a `Release` build to a network share. The network share is specified with forward slashes (*//r8/*) and works on all .NET Core supported platforms.

.NET CLI

```
dotnet publish -c Release /p:PublishDir=//r8/release/AdminWeb
```

Confirm that the published app for deployment isn't running. Files in the *publish* folder are locked when the app is running. Deployment can't occur because locked files can't be copied.

See the Microsoft.NET.Sdk.Publish readme file for more information.

Publish profiles

This section uses Visual Studio 2022 or later to create a publishing profile. Once the profile is created, publishing from Visual Studio or the command line is available. Publish profiles can simplify the publishing process, and any number of profiles can exist.

Create a publish profile in Visual Studio by choosing one of the following paths:

- Right-click the project in **Solution Explorer** and select **Publish**.
- Select **Publish {PROJECT NAME}** from the **Build** menu.

The **Publish** tab of the app capabilities page is displayed. Several publish targets are available, including:

- Azure
- Docker Container Registry
- Folder
- IIS, FTP, Web Deploy (for any web server)
- Import Profile

To determine the most appropriate publish target, see What publishing options are right for me.

When the **Folder** publish target is selected, specify a folder path to store the published assets. The default folder path is *bin\{PROJECT CONFIGURATION}\{TARGET FRAMEWORK MONIKER}\publish*. For example, *bin\Release\netcoreapp2.2\publish*. Select the **Create Profile** button to finish.

Once a publish profile is created, the **Publish** tab's content changes. The newly created profile appears in a drop-down list. Below the drop-down list, select **Create new profile** to create another new profile.

Visual Studio's publish tool produces a `Properties/PublishProfiles/{PROFILE NAME}.pubxml` MSBuild file describing the publish profile. The *.pubxml* file:

- Contains publish configuration settings and is consumed by the publishing process.
- Can be modified to customize the build and publish process.

When publishing to an Azure target, the *.pubxml* file:

- Contains the Azure subscription identifier.
- Should not be checked into source control because the subscription identifier is sensitive information.

Sensitive information, for example, the publish password, is encrypted on a per user/machine level. The `Properties/PublishProfiles/{PROFILE NAME}.pubxml.user` file contains the information needed by MSBuild to retrieve the user name and password.

For an overview of how to publish an ASP.NET Core web app, see Host and deploy ASP.NET Core. The MSBuild tasks and targets necessary to publish an ASP.NET Core web app are open-source in the dotnet/websdk repository.

`dotnet publish` and `dotnet build`:

- Can use folder, MSDeploy, and Kudu publish profiles. Because MSDeploy lacks cross-platform support, MSDeploy options are supported only on Windows.
- Support Kudu APIs to publish to Azure from any platform. Visual Studio publish supports the Kudu APIs, but it's supported by WebSDK for cross-platform publish to Azure.

Don't pass `DeployOnBuild` to the `dotnet publish` command.

For more information, see Microsoft.NET.Sdk.Publish.

Folder publish example

When publishing with a profile named `FolderProfile`, use any of the following commands:

.NET CLI

```
dotnet publish /p:Configuration=Release /p:PublishProfile=FolderProfile
```

.NET CLI

```
dotnet build /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

Bash

```
msbuild /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

The .NET CLI's dotnet build command calls `msbuild` to run the build and publish process. The `dotnet build` and `msbuild` commands are equivalent when passing in a folder profile. When calling `msbuild` directly on Windows, the .NET Framework version of MSBuild is used. Calling `dotnet build` on a non-folder profile:

- Invokes `msbuild`, which uses MSDeploy.
- Results in a failure (even when running on Windows). To publish with a non-folder profile, call `msbuild` directly.

The following folder publish profile was created with Visual Studio and publishes to a network share:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<!--
https://go.microsoft.com/fwlink/?LinkID=208121.
-->
<Project>
  <PropertyGroup>
    <DeleteExistingFiles>false</DeleteExistingFiles>
    <ExcludeApp_Data>false</ExcludeApp_Data>
    <LaunchSiteAfterPublish>true</LaunchSiteAfterPublish>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <PublishProvider>FileSystem</PublishProvider>
    <PublishUrl>\\r8\Release\AdminWeb</PublishUrl>
    <WebPublishMethod>FileSystem</WebPublishMethod>
```

```
    <_TargetId>Folder</_TargetId>
  </PropertyGroup>
</Project>
```

In the preceding example:

- The `<ExcludeApp_Data>` property is present to satisfy an XML schema requirement. The `<ExcludeApp_Data>` property has no effect on the publish process, even if there's an *App_Data* folder in the project root. The *App_Data* folder doesn't receive special treatment as it does in ASP.NET 4.x projects.
- The `<LastUsedBuildConfiguration>` property is set to `Release`. When publishing from Visual Studio, the value of `<LastUsedBuildConfiguration>` is set using the value when the publish process is started. `<LastUsedBuildConfiguration>` is special and shouldn't be overridden in an imported MSBuild file. This property can, however, be overridden from the command line using one of the following approaches.
  - Using the .NET CLI:

    .NET CLI

    ```
    dotnet publish /p:Configuration=Release
    /p:PublishProfile=FolderProfile
    ```

    .NET CLI

    ```
    dotnet build -c Release /p:DeployOnBuild=true
    /p:PublishProfile=FolderProfile
    ```

  - Using MSBuild:

    Bash

    ```
    msbuild /p:Configuration=Release /p:DeployOnBuild=true
    /p:PublishProfile=FolderProfile
    ```

Publish to an MSDeploy endpoint from the command line

See Microsoft.NET.Sdk.Publish .

Set the environment

Include the `<EnvironmentName>` property in the publish profile (*.pubxml*) or project file to set the app's environment:

XML

```xml
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

If you require *web.config* transformations (for example, setting environment variables based on the configuration, profile, or environment), see Transform web.config.

Exclude files

When publishing ASP.NET Core web apps, the following assets are included:

- Build artifacts
- Folders and files matching the following globbing patterns:
  - `**\*.config` (for example, *web.config*)
  - `**\*.json` (for example, `appsettings.json`)
  - `wwwroot\**`

MSBuild supports globbing patterns. For example, the following `<Content>` element suppresses the copying of text (*.txt*) files in the *wwwroot\content* folder and its subfolders:

XML

```xml
<ItemGroup>
  <Content Update="wwwroot/content/**/*.txt" CopyToPublishDirectory="Never"
/>
</ItemGroup>
```

The preceding markup can be added to a publish profile or the `.csproj` file. When added to the `.csproj` file, the rule is added to all publish profiles in the project.

The following `<MsDeploySkipRules>` element excludes all files from the *wwwroot\content* folder:

XML

```
<ItemGroup>
  <MsDeploySkipRules Include="CustomSkipFolder">
    <ObjectName>dirPath</ObjectName>
    <AbsolutePath>wwwroot\\content</AbsolutePath>
  </MsDeploySkipRules>
</ItemGroup>
```

`<MsDeploySkipRules>` won't delete the *skip* targets from the deployment site. `<Content>` targeted files and folders are deleted from the deployment site. For example, suppose a deployed web app had the following files:

- `Views/Home/About1.cshtml`
- `Views/Home/About2.cshtml`
- `Views/Home/About3.cshtml`

If the following `<MsDeploySkipRules>` elements are added, those files wouldn't be deleted on the deployment site.

XML

```
<ItemGroup>
  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About1.cshtml</AbsolutePath>
  </MsDeploySkipRules>

  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About2.cshtml</AbsolutePath>
  </MsDeploySkipRules>

  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About3.cshtml</AbsolutePath>
  </MsDeploySkipRules>
</ItemGroup>
```

The preceding `<MsDeploySkipRules>` elements prevent the *skipped* files from being deployed. It won't delete those files once they're deployed.

The following `<Content>` element deletes the targeted files at the deployment site:

XML

```
<ItemGroup>
  <Content Update="Views/Home/About?.cshtml" CopyToPublishDirectory="Never"
```

```
/>
</ItemGroup>
```

Using command-line deployment with the preceding `<Content>` element yields a variation of the following output:

Console

```
MSDeployPublish:
  Starting Web deployment task from source:
manifest(C:\Webs\Web1\obj\Release\{TARGET FRAMEWORK
MONIKER}\PubTmp\Web1.SourceManifest.
  xml) to Destination: auto().
  Deleting file (Web11112\Views\Home\About1.cshtml).
  Deleting file (Web11112\Views\Home\About2.cshtml).
  Deleting file (Web11112\Views\Home\About3.cshtml).
  Updating file (Web11112\web.config).
  Updating file (Web11112\Web1.deps.json).
  Updating file (Web11112\Web1.dll).
  Updating file (Web11112\Web1.pdb).
  Updating file (Web11112\Web1.runtimeconfig.json).
  Successfully executed Web deployment task.
  Publish Succeeded.
Done Building Project "C:\Webs\Web1\Web1.csproj" (default targets).
```

Include files

The following sections outline different approaches for file inclusion at publish time. The General file inclusion section uses the `DotNetPublishFiles` item, which is provided by a publish targets file in the Web SDK. The Selective file inclusion section uses the `ResolvedFileToPublish` item, which is provided by a publish targets file in the .NET Core SDK. Because the Web SDK depends on the .NET Core SDK, either item can be used in an ASP.NET Core project.

General file inclusion

The following example's `<ItemGroup>` element demonstrates copying a folder located outside of the project directory to a folder of the published site. Any files added to the following markup's `<ItemGroup>` are included by default.

XML

```
<ItemGroup>
  <_CustomFiles Include="$(MSBuildProjectDirectory)/../images/**/*" />
  <DotNetPublishFiles Include="@(_CustomFiles)">
```

```
    <DestinationRelativePath>wwwroot/images/%(RecursiveDir)%(Filename)%
(Extension)</DestinationRelativePath>
  </DotNetPublishFiles>
</ItemGroup>
```

The preceding markup:

- Can be added to the `.csproj` file or the publish profile. If it's added to the `.csproj` file, it's included in each publish profile in the project.
- Declares a `_CustomFiles` item to store files matching the `Include` attribute's globbing pattern. The *images* folder referenced in the pattern is located outside of the project directory. A reserved property, named `$(MSBuildProjectDirectory)`, resolves to the project file's absolute path.
- Provides a list of files to the `DotNetPublishFiles` item. By default, the item's `<DestinationRelativePath>` element is empty. The default value is overridden in the markup and uses well-known item metadata such as `%(RecursiveDir)`. The inner text represents the *wwwroot/images* folder of the published site.

Selective file inclusion

The highlighted markup in the following example demonstrates:

- Copying a file located outside of the project into the published site's *wwwroot* folder. The file name of *ReadMe2.md* is maintained.
- Excluding the *wwwroot\Content* folder.
- Excluding *Views\Home\About2.cshtml*.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>FileSystem</WebPublishMethod>
    <PublishProvider>FileSystem</PublishProvider>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish />
    <LaunchSiteAfterPublish>True</LaunchSiteAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <PublishFramework />
    <ProjectGuid>
      <GUID Here=""></GUID>
    </ProjectGuid>
    <publishUrl>bin\Release\PublishOutput</publishUrl>
    <DeleteExistingFiles>False</DeleteExistingFiles>
  </PropertyGroup>
```

```
  <ItemGroup>
    <ResolvedFileToPublish Include="..\ReadMe2.md">
      <RelativePath>wwwroot\ReadMe2.md</RelativePath>
    </ResolvedFileToPublish>

    <Content Update="wwwroot\Content\**\*" CopyToPublishDirectory="Never" />
    <Content Update="Views\Home\About2.cshtml"
CopyToPublishDirectory="Never" />
  </ItemGroup>
</Project>
```

The preceding example uses the `ResolvedFileToPublish` item, whose default behavior is to always copy the files provided in the `Include` attribute to the published site. Override the default behavior by including a `<CopyToPublishDirectory>` child element with inner text of either `Never` or `PreserveNewest`. For example:

XML

```
<ResolvedFileToPublish Include="..\ReadMe2.md">
  <RelativePath>wwwroot\ReadMe2.md</RelativePath>
  <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
</ResolvedFileToPublish>
```

For more deployment samples, see the Web SDK README file.

Run a target before or after publishing

The built-in `BeforePublish` and `AfterPublish` targets execute a target before or after the publish target. Add the following elements to the publish profile to log console messages both before and after publishing:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This file is used by the publish/package process of your Web project. You
can customize the behavior of this process
by editing this MSBuild file. In order to learn more about this please visit
https://go.microsoft.com/fwlink/?LinkID=208121.
-->
<Project>
  <PropertyGroup>
    <WebPublishMethod>MSDeploy</WebPublishMethod>

<ResourceId>/subscriptions/SomeGuid/resourcegroups/TP_RG/providers/Microsoft
.Web/sites/TP22</ResourceId>
    <ResourceGroup>TP_RG</ResourceGroup>
    <PublishProvider>AzureWebSite</PublishProvider>
```

```
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>

<SiteUrlToLaunchAfterPublish>https://tp22.azurewebsites.net</SiteUrlToLaunch
AfterPublish>
    <LaunchSiteAfterPublish>true</LaunchSiteAfterPublish>
    <ExcludeApp_Data>false</ExcludeApp_Data>
    <ProjectGuid>GuidHere</ProjectGuid>

<MSDeployServiceURL>something.scm.azurewebsites.net:443</MSDeployServiceURL>
    <DeployIisAppPath>myDeploysIISpath</DeployIisAppPath>
    <RemoteSitePhysicalPath />
    <SkipExtraFilesOnServer>true</SkipExtraFilesOnServer>
    <MSDeployPublishMethod>WMSVC</MSDeployPublishMethod>
    <EnableMSDeployBackup>true</EnableMSDeployBackup>
    <EnableMsDeployAppOffline>true</EnableMsDeployAppOffline>
    <UserName />
    <_SavePWD>false</_SavePWD>
    <_DestinationType>AzureWebSite</_DestinationType>
    <InstallAspNetCoreSiteExtension>false</InstallAspNetCoreSiteExtension>
  </PropertyGroup>
    <Target Name="CustomActionsBeforePublish" BeforeTargets="BeforePublish">
        <Message Text="Inside BeforePublish" Importance="high" />
    </Target>
    <Target Name="CustomActionsAfterPublish" AfterTargets="AfterPublish">
        <Message Text="Inside AfterPublish" Importance="high" />
    </Target>
</Project>
```

Publish to a server using an untrusted certificate

Add the `<AllowUntrustedCertificate>` property with a value of `True` to the publish profile:

XML

```
<PropertyGroup>
  <AllowUntrustedCertificate>True</AllowUntrustedCertificate>
</PropertyGroup>
```

The Kudu service

To view the files in an Azure App Service web app deployment, use the Kudu service. Append the `scm` token to the web app name. For example:

Expand table

URL	Result
`http://mysite.azurewebsites.net/`	Web App
`http://mysite.scm.azurewebsites.net/`	Kudu service

Select the Debug Console menu item to view, edit, delete, or add files.

Additional resources

- Web SDK README file
- Web SDK GitHub repository: File issues and request features for deployment.
- Web Deploy (MSDeploy) simplifies deployment of web apps and websites to IIS servers.
- Transform web.config

ASP.NET Core directory structure

Article • 07/31/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

The *publish* directory contains the app's deployable assets produced by the dotnet publish command. The directory contains:

- Application files
- Configuration files
- Static assets
- Packages
- A runtime (self-contained deployment only)

Expand table

App Type	Directory Structure
Framework-dependent Executable (FDE)	• publish† ◦ Views† MVC apps; if views aren't precompiled ◦ Pages† MVC or Razor Pages apps, if pages aren't precompiled ◦ wwwroot† ◦ *.dll files ◦ {ASSEMBLY NAME}.deps.json ◦ {ASSEMBLY NAME}.dll ◦ {ASSEMBLY NAME}{.EXTENSION}.exe extension on Windows, no extension on macOS or Linux ◦ {ASSEMBLY NAME}.pdb ◦ {ASSEMBLY NAME}.runtimeconfig.json ◦ web.config (IIS deployments) ◦ createdump (Linux createdump utility) ◦ *.so (Linux shared object library) ◦ *.a (macOS archive) ◦ *.dylib (macOS dynamic library)

App Type	Directory Structure
Self-contained Deployment (SCD)	• publish† ◦ Views† MVC apps, if views aren't precompiled ◦ Pages† MVC or Razor Pages apps, if pages aren't precompiled ◦ wwwroot† ◦ *.dll files ◦ {ASSEMBLY NAME}.deps.json ◦ {ASSEMBLY NAME}.dll ◦ {ASSEMBLY NAME}{.EXTENSION} .exe extension on Windows, no extension on macOS or Linux ◦ {ASSEMBLY NAME}.pdb ◦ {ASSEMBLY NAME}.runtimeconfig.json ◦ web.config (IIS deployments)

†Indicates a directory

The *publish* directory represents the *content root path*, also called the *application base path*, of the deployment. Whatever name is given to the *publish* directory of the deployed app on the server, its location serves as the server's physical path to the hosted app.

The *wwwroot* directory, if present, only contains static assets.

Additional resources

- dotnet publish
- .NET Core application deployment
- Target frameworks
- .NET Core RID Catalog

ASP.NET Core security topics

Article • 11/11/2024

ASP.NET Core enables developers to configure and manage security. The following list provides links to security topics:

- Authentication
- Authorization
- Data protection
- HTTPS enforcement
- Safe storage of app secrets in development
- XSRF/CSRF prevention
- Cross Origin Resource Sharing (CORS)
- Cross-Site Scripting (XSS) attacks

These security features allow you to build robust and secure ASP.NET Core apps.

For Blazor security coverage, which adds to or supersedes the guidance in this node, see ASP.NET Core Blazor authentication and authorization and the other articles in Blazor's *Security and Identity* node.

ASP.NET Core security features

ASP.NET Core provides many tools and libraries to secure ASP.NET Core apps such as built-in identity providers and third-party identity services such as Facebook, Twitter, and LinkedIn. ASP.NET Core provides several approaches to store app secrets.

Authentication vs. Authorization

Authentication is a process in which a user provides credentials that are then compared to those stored in an operating system, database, app or resource. If they match, users authenticate successfully, and can then perform actions that they're authorized for, during an authorization process. The authorization refers to the process that determines what a user is allowed to do.

Another way to think of authentication is to consider it as a way to enter a space, such as a server, database, app or resource, while authorization is which actions the user can perform to which objects inside that space (server, database, or app).

Common Vulnerabilities in software

ASP.NET Core and EF contain features that help you secure your apps and prevent security breaches. The following list of links takes you to documentation detailing techniques to avoid the most common security vulnerabilities in web apps:

- Cross-Site Scripting (XSS) attacks
- SQL injection attacks
- Cross-Site Request Forgery (XSRF/CSRF) attacks
- Open redirect attacks

There are more vulnerabilities that you should be aware of. For more information, see the other articles in the **Security and Identity** section of the table of contents.

Secure authentication flows

We recommend using the most secure secure authentication option. For Azure services, the most secure authentication is managed identities.

Avoid Resource Owner Password Credentials Grant because it:

- Exposes the user's password to the client.
- Is a significant security risk.
- Should only be used when other authentication flows are not possible.

Managed identities are a secure way to authenticate to services without needing to store credentials in code, environment variables, or configuration files. Managed identities are available for Azure services, and can be used with Azure SQL, Azure Storage, and other Azure services:

- Managed identities in Microsoft Entra for Azure SQL
- Managed identities for App Service and Azure Functions
- Secure authentication flows

When the app is deployed to a test server, an environment variable can be used to set the connection string to a test database server. For more information, see Configuration. Environment variables are generally stored in plain, unencrypted text. If the machine or process is compromised, environment variables can be accessed by untrusted parties. We recommend against using environment variables to store a production connection string as it's not the most secure approach.

Configuration data guidelines:

- Never store passwords or other sensitive data in configuration provider code or in plain text configuration files. The Secret Manager tool can be used to store secrets in development.

- Don't use production secrets in development or test environments.
- Specify secrets outside of the project so that they can't be accidentally committed to a source code repository.

For more information, see:

- Managed identity best practice recommendations
- Connecting from your application to resources without handling credentials in your code
- Azure services that can use managed identities to access other services
- IETF OAuth 2.0 Security Best Current Practice

For information on other cloud providers, see:

- AWS (Amazon Web Services): AWS Key Management Service (KMS)
- Google Cloud Key Management Service overview

Additional resources

- Introduction to Identity on ASP.NET Core
- Enable QR code generation for TOTP authenticator apps in ASP.NET Core
- Facebook and Google authentication in ASP.NET Core
- Identity management solutions for .NET web apps

Overview of ASP.NET Core authentication

Article • 02/14/2024

By Mike Rousos

Authentication is the process of determining a user's identity. Authorization is the process of determining whether a user has access to a resource. In ASP.NET Core, authentication is handled by the authentication service, IAuthenticationService, which is used by authentication middleware. The authentication service uses registered authentication handlers to complete authentication-related actions. Examples of authentication-related actions include:

- Authenticating a user.
- Responding when an unauthenticated user tries to access a restricted resource.

The registered authentication handlers and their configuration options are called "schemes".

Authentication schemes are specified by registering authentication services in `Program.cs`:

- By calling a scheme-specific extension method after a call to AddAuthentication, such as AddJwtBearer or AddCookie. These extension methods use AuthenticationBuilder.AddScheme to register schemes with appropriate settings.
- Less commonly, by calling `AuthenticationBuilder.AddScheme` directly.

For example, the following code registers authentication services and handlers for cookie and JWT bearer authentication schemes:

C#

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme,
        options => builder.Configuration.Bind("JwtSettings", options))
    .AddCookie(CookieAuthenticationDefaults.AuthenticationScheme,
        options => builder.Configuration.Bind("CookieSettings", options));
```

The `AddAuthentication` parameter JwtBearerDefaults.AuthenticationScheme is the name of the scheme to use by default when a specific scheme isn't requested.

If multiple schemes are used, authorization policies (or authorization attributes) can specify the authentication scheme (or schemes) they depend on to authenticate the

user. In the example above, the cookie authentication scheme could be used by specifying its name (CookieAuthenticationDefaults.AuthenticationScheme by default, though a different name could be provided when calling `AddCookie`).

In some cases, the call to `AddAuthentication` is automatically made by other extension methods. For example, when using ASP.NET Core Identity, `AddAuthentication` is called internally.

The Authentication middleware is added in `Program.cs` by calling UseAuthentication. Calling `UseAuthentication` registers the middleware that uses the previously registered authentication schemes. Call `UseAuthentication` before any middleware that depends on users being authenticated.

Authentication concepts

Authentication is responsible for providing the ClaimsPrincipal for authorization to make permission decisions against. There are multiple authentication scheme approaches to select which authentication handler is responsible for generating the correct set of claims:

- Authentication scheme
- The default authentication scheme, discussed in the next two sections.
- Directly set HttpContext.User.

When there is only a single authentication scheme registered, it becomes the default scheme. If multiple schemes are registered and the default scheme isn't specified, a scheme must be specified in the authorize attribute, otherwise, the following error is thrown:

> InvalidOperationException: No authenticationScheme was specified, and there was no DefaultAuthenticateScheme found. The default schemes can be set using either AddAuthentication(string defaultScheme) or AddAuthentication(Action<AuthenticationOptions> configureOptions).

`DefaultScheme`

When there is only a single authentication scheme registered, the single authentication scheme:

- Is automatically used as the DefaultScheme.
- Eliminates the need to specify the `DefaultScheme` in AddAuthentication(IServiceCollection) or

AddAuthenticationCore(IServiceCollection).

To disable automatically using the single authentication scheme as the `DefaultScheme`, call `AppContext.SetSwitch("Microsoft.AspNetCore.Authentication.SuppressAutoDefaultScheme")`.

Authentication scheme

The authentication scheme can select which authentication handler is responsible for generating the correct set of claims. For more information, see Authorize with a specific scheme.

An authentication scheme is a name that corresponds to:

- An authentication handler.
- Options for configuring that specific instance of the handler.

Schemes are useful as a mechanism for referring to the authentication, challenge, and forbid behaviors of the associated handler. For example, an authorization policy can use scheme names to specify which authentication scheme (or schemes) should be used to authenticate the user. When configuring authentication, it's common to specify the default authentication scheme. The default scheme is used unless a resource requests a specific scheme. It's also possible to:

- Specify different default schemes to use for authenticate, challenge, and forbid actions.
- Combine multiple schemes into one using policy schemes.

Authentication handler

An authentication handler:

- Is a type that implements the behavior of a scheme.
- Is derived from IAuthenticationHandler or AuthenticationHandler<TOptions>.
- Has the primary responsibility to authenticate users.

Based on the authentication scheme's configuration and the incoming request context, authentication handlers:

- Construct AuthenticationTicket objects representing the user's identity if authentication is successful.
- Return 'no result' or 'failure' if authentication is unsuccessful.

- Have methods for challenge and forbid actions for when users attempt to access resources:
  - They're unauthorized to access (forbid).
  - When they're unauthenticated (challenge).

`RemoteAuthenticationHandler<TOptions>` vs `AuthenticationHandler<TOptions>`

RemoteAuthenticationHandler<TOptions> is the class for authentication that requires a remote authentication step. When the remote authentication step is finished, the handler calls back to the `CallbackPath` set by the handler. The handler finishes the authentication step using the information passed to the HandleRemoteAuthenticateAsync callback path. OAuth 2.0 and OIDC both use this pattern. JWT and cookies don't since they can directly use the bearer header and cookie to authenticate. The remotely hosted provider in this case:

- Is the authentication provider.
- Examples include Facebook, Twitter, Google, Microsoft, and any other OIDC provider that handles authenticating users using the handlers mechanism.

Authenticate

An authentication scheme's authenticate action is responsible for constructing the user's identity based on request context. It returns an AuthenticateResult indicating whether authentication was successful and, if so, the user's identity in an authentication ticket. See AuthenticateAsync. Authenticate examples include:

- A cookie authentication scheme constructing the user's identity from cookies.
- A JWT bearer scheme deserializing and validating a JWT bearer token to construct the user's identity.

Challenge

An authentication challenge is invoked by Authorization when an unauthenticated user requests an endpoint that requires authentication. An authentication challenge is issued, for example, when an anonymous user requests a restricted resource or follows a login link. Authorization invokes a challenge using the specified authentication scheme(s), or the default if none is specified. See ChallengeAsync. Authentication challenge examples include:

- A cookie authentication scheme redirecting the user to a login page.

- A JWT bearer scheme returning a 401 result with a `www-authenticate: bearer` header.

A challenge action should let the user know what authentication mechanism to use to access the requested resource.

Forbid

An authentication scheme's forbid action is called by Authorization when an authenticated user attempts to access a resource they're not permitted to access. See ForbidAsync. Authentication forbid examples include:

- A cookie authentication scheme redirecting the user to a page indicating access was forbidden.
- A JWT bearer scheme returning a 403 result.
- A custom authentication scheme redirecting to a page where the user can request access to the resource.

A forbid action can let the user know:

- They're authenticated.
- They're not permitted to access the requested resource.

See the following links for differences between challenge and forbid:

- Challenge and forbid with an operational resource handler.
- Differences between challenge and forbid.

Authentication providers per tenant

ASP.NET Core doesn't have a built-in solution for multi-tenant authentication. While it's possible for customers to write one using the built-in features, we recommend customers consider Orchard Core, ABP Framework, or Finbuckle.MultiTenant for multi-tenant authentication.

Orchard Core is:

- An open-source, modular, and multi-tenant app framework built with ASP.NET Core.
- A content management system (CMS) built on top of that app framework.

See the Orchard Core source for an example of authentication providers per tenant.

ABP Framework supports various architectural patterns including modularity, microservices, domain driven design, and multi-tenancy. See ABP Framework source on GitHub.

Finbuckle.MultiTenant:

- Open source
- Provides tenant resolution
- Lightweight
- Provides data isolation
- Configure app behavior uniquely for each tenant

Additional resources

- Authorize with a specific scheme in ASP.NET Core
- Policy schemes in ASP.NET Core
- Create an ASP.NET Core app with user data protected by authorization
- Globally require authenticated users
- GitHub issue on using multiple authentication schemes

Choose an identity management solution

Article • 08/30/2023

Most web apps support authentication to ensure that users are who they claim to be. A *user* might be a person or another app. Management of access ensures users are only able to see and modify the information they're authorized to see and modify. For example, an end user shouldn't have access to the administrative section of a website. Identity management solutions are built to handle the requirements of authentication and authorization-related tasks. To learn more about identity management, see What is identity and access management?. Many identity management solutions for .NET web apps are available, each with different capabilities and requirements to use or install. This article provides guidance on how to choose the right solution.

Basic identity management with ASP.NET Core Identity

ASP.NET Core ships with a built-in authentication provider: ASP.NET Core Identity. The provider includes the APIs, UI, and backend database configuration to support managing user identities, storing user credentials, and granting or revoking permissions. Other features it supports include:

- External logins
- Two-factor authentication (2FA)
- Password management
- Account lockout and reactivation
- Authenticator apps

For most scenarios, this may be the only provider needed.

To learn more:

- Read the Introduction to Identity on ASP.NET Core
- Follow a tutorial to build your own secure .NET web app: Secure a .NET web app with the ASP.NET Core Identity framework.

In other scenarios, a server or service that manages authentication and identity may be beneficial.

Determine if an OIDC server is needed

Web apps require a way to *remember* past actions because the web, by default, is stateless. Otherwise, users would be forced to enter their credentials every time they navigated to a new page. The common solution for remembering state is *cookies*, a browser-based mechanism for storing data. The web server sends the initial cookie, then the browser stores it and sends it back with each request. This is done automatically without the need for the developer to write any code. Cookies are easy to use and built into the browser but are designed for use within a single website or web domain. The default solution that is built into ASP.NET Core uses cookie-based authentication.

Tokens are containers with metadata that are explicitly passed through the headers or body of HTTP requests. The main advantage of tokens over cookies is that they are not tied to a specific app or domain. Instead, tokens are usually *signed* with asymmetric cryptography. For example, OIDC servers issue tokens with information about identity using the JSON Web Token (JWT) format which includes signing. Asymmetric cryptography uses a combination of a private key known only to the signer, and a public key which everyone can know. Tokens may also be encrypted.

The signed token can't be tampered with due to the private key. The public key:

- Makes it possible to validate the token to ensure it hasn't been changed.
- Guarantees that it was generated by the entity holding the private key.

The main disadvantage to using tokens is that they require a service (typically an OIDC server) to both issue and provide validation for tokens. The service must be installed, configured, and maintained.

A common reason an OIDC server is required is for applications that expose web-based APIs that are consumed by other apps. For exposed web-based APIs, client UIs such as Single Page Applications (SPA), mobile clients, and desktop clients are considered to be part of the same app. SPA examples include Angular, React, and Blazor WebAssembly. If apps other than your web app or any client UIs must make a secure API call to your app, you'll likely want to use tokens. If you only have client UIs, ASP.NET Core Identity provides the option to acquire a token during authentication. The authentication token issued by ASP.NET Core Identity:

- Can be used by mobile and desktop clients. Cookies are preferred over tokens for both security and simplicity.
- Isn't suitable for managing access from third-party apps.

Another reason an OIDC server is required is for sharing sign-ins with other apps. Commonly referred to as *single sign on*, this feature enables users to:

- Sign in once with a web app's form.
- Use the resulting credentials to authenticate with other apps without having to sign-in again or choose a different password.

An OIDC server is typically preferred to provide a secure and scalable solution for single sign on.

For apps that don't share logins with other apps, the simplest way to quickly secure an app is to use the built-in ASP.NET Core Identity provider. Otherwise, an OIDC server provided by a third-party identity management solution is needed. OIDC servers are available as:

- Products you install on your server, called *self-host*.
- Containers run in a host like Docker.
- Web-based services you integrate with to manage identity.

Some solutions are free and open source, while others are commercially licensed. See identity management solutions for a list of available options. It's possible that your organization already uses an identity provider. In that case, it may make sense to use the existing provider instead of going with a different solution. All of the major solutions provide documentation for configuring ASP.NET Core to use their product or service.

Disconnected scenarios

Many solutions, such as Microsoft Entra ID, are cloud-based and require an Internet connection to work. If your environment doesn't allow Internet connectivity, you won't be able to use the service.

ASP.NET Core Identity works perfectly well in disconnected scenarios, such as:

- The app can't access the Internet.
- The app must still function on the local network even if the Internet is disconnected.

If you require a full OIDC server for a disconnected scenario, choose one of the following options:

- A solution that allows you to install and run the service on your own machines.
- Run the authentication service locally as a container.

Decide where user data such as sign-ins are stored

Another important factor to consider is where the user sign-in data is stored. Many developers choose external, cloud-based services like Microsoft Entra ID to manage identity. A cloud-based service provider:

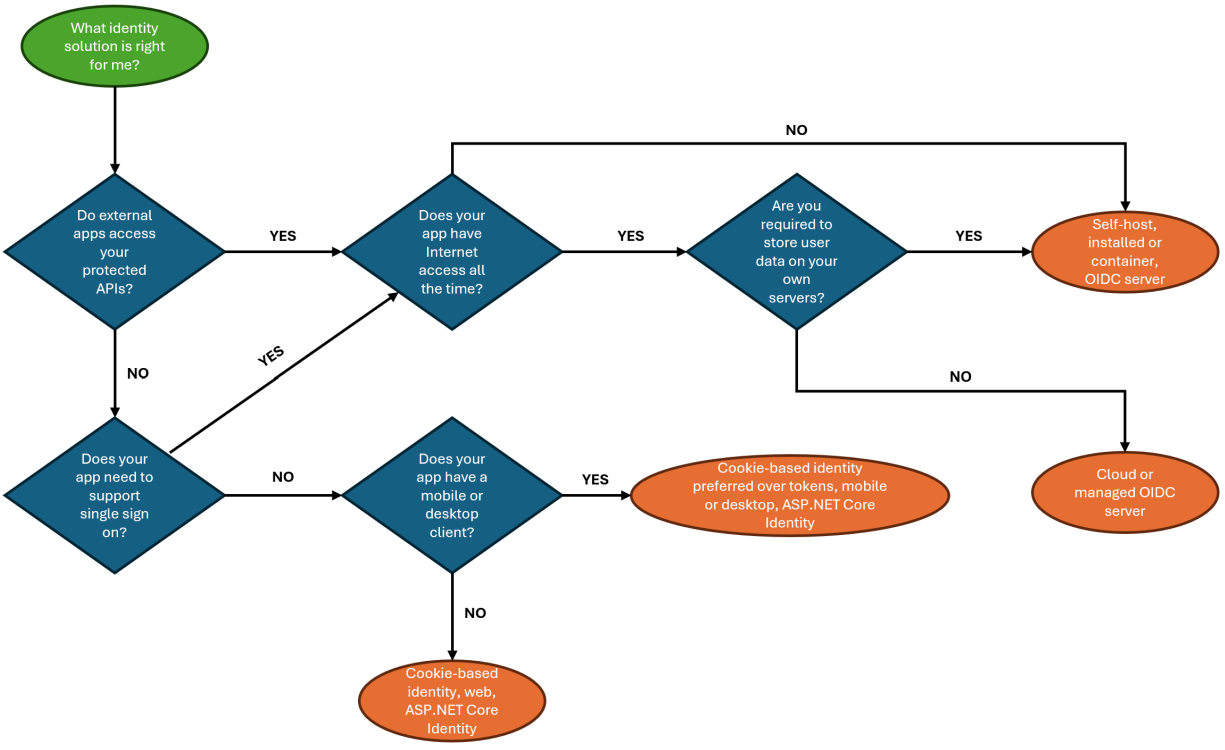
- Takes on the responsibilities of securely storing data.
- keeps the software up to date with the latest security patches and releases.
- Complies with privacy regulations.

Others prefer to store data on their own servers due to regulations, compliance, policy, or other reasons.

If the data is stored on your servers, you'll most likely need to choose an installable or container-based solution.

Identity vs OIDC server

Use the following diagram to help you decide whether to use the ASP.NET Core Identity system or an OIDC server for authentication and authorization:



The following table lists some of the things to consider when choosing your identity management solution.

Expand table

Feature	Self-host (infrastructure or container)	Cloud
App integration	Local solutions that are implemented as libraries or frameworks can often be integrated directly in your own app. Container-based solutions require a hand-off to occur between your web app and the container-based service.	Cloud-based solutions typically integrate at specific points in your sign-in flow and provide configuration to update the UI to match your theme, but the level of customization available is limited.
Configuration	Self host solutions require configuring the software for the environment in addition to setting up how you want to manage identities. Container-based solutions typically provide a web-based UI for configuration.	Cloud-based solutions typically provide a web-based UI for configuration.
Customization	Self-host solutions are usually highly customizable, including code-based changes. Although containerized solutions provide extensibility options, they are often more limited.	Cloud-based services allow customization, but it's typically limited to configuration-based changes.
Maintenance	Installed products require a dedicated resource to ensure all security patches are applied in a timely fashion and to manage upgrades. The upgrade and patch process for containers is usually lower-friction and involves simply installing the provided container image.	The service provider maintains their cloud-based solution, including applying needed patches and handling upgrades.
User credentials storage	You are responsible for data governance and handling breaches.	Managing the risks associated with handling user credentials, and complying with regulations. is delegated to the service provider.

For more information about available options, see Identity management solutions for ASP.NET Core.

Next steps

- Learn about JSON Web Tokens
- Browse sample apps with authentication/authorization and identity for ASP.NET Core.
- Follow a tutorial to secure a .NET web app using built-in ASP.NET Core Identity.
- Learn more about how to protect web APIs.

Introduction to Identity on ASP.NET Core

Article • 08/30/2024

By Rick Anderson

ASP.NET Core Identity:

- Is an API that supports user interface (UI) login functionality.
- Manages users, passwords, profile data, roles, claims, tokens, email confirmation, and more.

Users can create an account with the login information stored in Identity or they can use an external login provider. Supported external login providers include Facebook, Google, Microsoft Account, and Twitter.

For information on how to globally require all users to be authenticated, see Require authenticated users.

The Identity source code is available on GitHub. Scaffold Identity and view the generated files to review the template interaction with Identity.

Identity is typically configured using a SQL Server database to store user names, passwords, and profile data. Alternatively, another persistent store can be used, for example, Azure Table Storage.

In this topic, you learn how to use Identity to register, log in, and log out a user. Note: the templates treat username and email as the same for users. For more detailed instructions about creating apps that use Identity, see Next Steps.

ASP.NET Core Identity isn't related to the Microsoft identity platform. Microsoft identity platform is:

- An evolution of the Azure Active Directory (Azure AD) developer platform.
- An alternative identity solution for authentication and authorization in ASP.NET Core apps.

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Microsoft Entra ID
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Duende Identity Server enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

> **Important**
>
> Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see Migrate from ASP.NET Core 5.0 to 6.0.

For more information, see the Duende Identity Server documentation (Duende Software website).

View or download the sample code (how to download).

Create a Web app with authentication

Create an ASP.NET Core Web Application project with Individual User Accounts.

Visual Studio

- Select the **ASP.NET Core Web App** template. Name the project **WebApp1** to have the same namespace as the project download. Click **OK**.
- In the **Authentication type** input, select **Individual User Accounts**.

The generated project provides ASP.NET Core Identity as a Razor class library. The Identity Razor class library exposes endpoints with the `Identity` area. For example:

- /Identity/Account/Login
- /Identity/Account/Logout
- /Identity/Account/Manage

Apply migrations

Apply the migrations to initialize the database.

Visual Studio

Run the following command in the Package Manager Console (PMC):

```
Update-Database
```

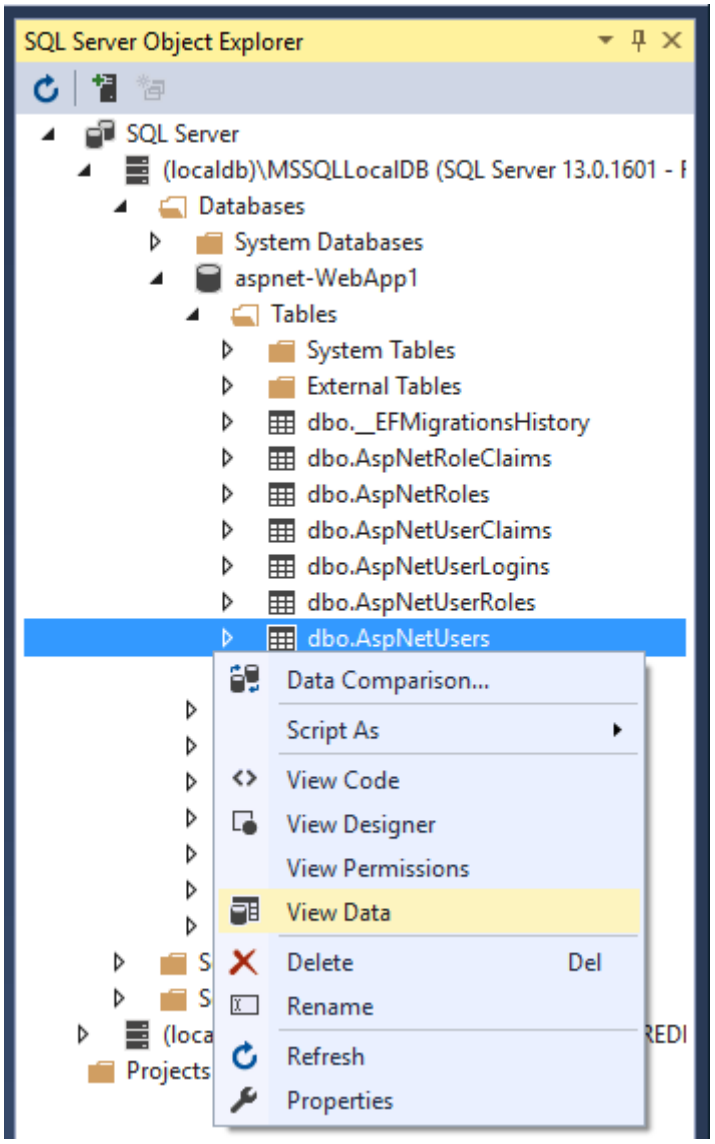
Test Register and Login

Run the app and register a user. Depending on your screen size, you might need to select the navigation toggle button to see the **Register** and **Login** links.

View the Identity database

Visual Studio

- From the **View** menu, select **SQL Server Object Explorer** (SSOX).
- Navigate to **(localdb)MSSQLLocalDB(SQL Server 13)**. Right-click on **dbo.AspNetUsers** > **View Data**:



Configure Identity services

Services are added in `Program.cs`. The typical pattern is to call methods in the following order:

1. `Add{Service}`
2. `builder.Services.Configure{Service}`

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebApp1.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
```

```
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

builder.Services.Configure<IdentityOptions>(options =>
{
    // Password settings.
    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;

    // Lockout settings.
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;

    // User settings.
    options.User.AllowedUserNameCharacters =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = false;
});

builder.Services.ConfigureApplicationCookie(options =>
{
    // Cookie settings
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(5);

    options.LoginPath = "/Identity/Account/Login";
    options.AccessDeniedPath = "/Identity/Account/AccessDenied";
    options.SlidingExpiration = true;
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
```

```
app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding code configures Identity with default option values. Services are made available to the app through dependency injection.

Identity is enabled by calling UseAuthentication. `UseAuthentication` adds authentication middleware to the request pipeline.

The template-generated app doesn't use authorization. `app.UseAuthorization` is included to ensure it's added in the correct order should the app add authorization. `UseRouting`, `UseAuthentication`, and `UseAuthorization` must be called in the order shown in the preceding code.

For more information on `IdentityOptions`, see IdentityOptions and Application Startup.

Scaffold Register, Login, LogOut, and RegisterConfirmation

Visual Studio

Add the `Register`, `Login`, `LogOut`, and `RegisterConfirmation` files. Follow the Scaffold identity into a Razor project with authorization instructions to generate the code shown in this section.

Examine Register

When a user clicks the **Register** button on the `Register` page, the `RegisterModel.OnPostAsync` action is invoked. The user is created by CreateAsync(TUser) on the `_userManager` object:

C#

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    ExternalLogins = (await
```

```
_signInManager.GetExternalAuthenticationSchemesAsync())
                                            .ToList();
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = Input.Email, Email =
Input.Email };
        var result = await _userManager.CreateAsync(user, Input.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with
password.");

            var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
            code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
            var callbackUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { area = "Identity", userId = user.Id, code =
code },
                protocol: Request.Scheme);

            await _emailSender.SendEmailAsync(Input.Email, "Confirm your
email",
                $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

            if (_userManager.Options.SignIn.RequireConfirmedAccount)
            {
                return RedirectToPage("RegisterConfirmation",
                                      new { email = Input.Email });
            }
            else
            {
                await _signInManager.SignInAsync(user, isPersistent: false);
                return LocalRedirect(returnUrl);
            }
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

Disable default account verification

With the default templates, the user is redirected to the `Account.RegisterConfirmation` where they can select a link to have the account confirmed. The default `Account.RegisterConfirmation` is used ***only*** for testing, automatic account verification should be disabled in a production app.

To require a confirmed account and prevent immediate login at registration, set `DisplayConfirmAccountLink = false` in `/Areas/Identity/Pages/Account/RegisterConfirmation.cshtml.cs`:

C#

```
[AllowAnonymous]
public class RegisterConfirmationModel : PageModel
{
    private readonly UserManager<IdentityUser> _userManager;
    private readonly IEmailSender _sender;

    public RegisterConfirmationModel(UserManager<IdentityUser> userManager,
IEmailSender sender)
    {
        _userManager = userManager;
        _sender = sender;
    }

    public string Email { get; set; }

    public bool DisplayConfirmAccountLink { get; set; }

    public string EmailConfirmationUrl { get; set; }

    public async Task<IActionResult> OnGetAsync(string email, string
returnUrl = null)
    {
        if (email == null)
        {
            return RedirectToPage("/Index");
        }

        var user = await _userManager.FindByEmailAsync(email);
        if (user == null)
        {
            return NotFound($"Unable to load user with email '{email}'.");
        }

        Email = email;
        // Once you add a real email sender, you should remove this code
that lets you confirm the account
        DisplayConfirmAccountLink = false;
        if (DisplayConfirmAccountLink)
        {
            var userId = await _userManager.GetUserIdAsync(user);
            var code = await
```

```
_userManager.GenerateEmailConfirmationTokenAsync(user);
            code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
            EmailConfirmationUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { area = "Identity", userId = userId, code =
code, returnUrl = returnUrl },
                protocol: Request.Scheme);
        }

        return Page();
    }
}
```

Log in

The Login form is displayed when:

- The **Log in** link is selected.
- A user attempts to access a restricted page that they aren't authorized to access **or** when they haven't been authenticated by the system.

When the form on the Login page is submitted, the `OnPostAsync` action is called. `PasswordSignInAsync` is called on the `_signInManager` object.

C#

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout,
        // set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(Input.Email,
                            Input.Password, Input.RememberMe,
lockoutOnFailure: true);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new
            {
                ReturnUrl = returnUrl,
```

```
                RememberMe = Input.RememberMe
            });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login
attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

For information on how to make authorization decisions, see Introduction to authorization in ASP.NET Core.

Log out

The **Log out** link invokes the `LogoutModel.OnPost` action.

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

namespace WebApp1.Areas.Identity.Pages.Account
{
    [AllowAnonymous]
    public class LogoutModel : PageModel
    {
        private readonly SignInManager<IdentityUser> _signInManager;
        private readonly ILogger<LogoutModel> _logger;

        public LogoutModel(SignInManager<IdentityUser> signInManager,
ILogger<LogoutModel> logger)
        {
            _signInManager = signInManager;
            _logger = logger;
        }
```

```
        public void OnGet()
        {
        }

        public async Task<IActionResult> OnPost(string returnUrl = null)
        {
            await _signInManager.SignOutAsync();
            _logger.LogInformation("User logged out.");
            if (returnUrl != null)
            {
                return LocalRedirect(returnUrl);
            }
            else
            {
                return RedirectToPage();
            }
        }
    }
}
```

In the preceding code, the code `return RedirectToPage();` needs to be a redirect so that the browser performs a new request and the identity for the user gets updated.

SignOutAsync clears the user's claims stored in a cookie.

Post is specified in the `Pages/Shared/_LoginPartial.cshtml`:

CSHTML

```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
@if (SignInManager.IsSignedIn(User))
{
    <li class="nav-item">
        <a  class="nav-link text-dark" asp-area="Identity" asp-
page="/Account/Manage/Index"
                                                title="Manage">Hello
@User.Identity.Name!</a>
    </li>
    <li class="nav-item">
        <form class="form-inline" asp-area="Identity" asp-
page="/Account/Logout"
                                    asp-route-returnUrl="@Url.Page("/", new {
area = "" })"
                                    method="post" >
            <button  type="submit" class="nav-link btn btn-link text-
dark">Logout</button>
        </form>
    </li>
```

```
}
else
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity" asp-
page="/Account/Register">Register</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity" asp-
page="/Account/Login">Login</a>
    </li>
}
</ul>
```

Test Identity

The default web project templates allow anonymous access to the home pages. To test Identity, add [Authorize]:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;

namespace WebApp1.Pages
{
    [Authorize]
    public class PrivacyModel : PageModel
    {
        private readonly ILogger<PrivacyModel> _logger;

        public PrivacyModel(ILogger<PrivacyModel> logger)
        {
            _logger = logger;
        }

        public void OnGet()
        {
        }
    }
}
```

If you are signed in, sign out. Run the app and select the **Privacy** link. You are redirected to the login page.

Explore Identity

To explore Identity in more detail:

- Create full identity UI source
- Examine the source of each page and step through the debugger.

Identity Components

All the Identity-dependent NuGet packages are included in the ASP.NET Core shared framework.

The primary package for Identity is Microsoft.AspNetCore.Identity. This package contains the core set of interfaces for ASP.NET Core Identity, and is included by `Microsoft.AspNetCore.Identity.EntityFrameworkCore`.

Migrating to ASP.NET Core Identity

For more information and guidance on migrating your existing Identity store, see Migrate Authentication and Identity.

Setting password strength

See Configuration for a sample that sets the minimum password requirements.

AddDefaultIdentity and AddIdentity

AddDefaultIdentity was introduced in ASP.NET Core 2.1. Calling `AddDefaultIdentity` is similar to calling the following:

- AddIdentity
- AddDefaultUI
- AddDefaultTokenProviders

See AddDefaultIdentity source for more information.

Prevent publish of static Identity assets

To prevent publishing static Identity assets (stylesheets and JavaScript files for Identity UI) to the web root, add the following `ResolveStaticWebAssetsInputsDependsOn` property and `RemoveIdentityAssets` target to the app's project file:

XML

```xml
<PropertyGroup>

<ResolveStaticWebAssetsInputsDependsOn>RemoveIdentityAssets</ResolveStaticWebAssetsInputsDependsOn>
</PropertyGroup>

<Target Name="RemoveIdentityAssets">
  <ItemGroup>
    <StaticWebAsset Remove="@(StaticWebAsset)" Condition="%(SourceId) == 'Microsoft.AspNetCore.Identity.UI'" />
  </ItemGroup>
</Target>
```

Next Steps

- ASP.NET Core Identity source code
- How to work with Roles in ASP.NET Core Identity

- See this GitHub issue for information on configuring Identity using SQLite.
- Configure Identity
- Create an ASP.NET Core app with user data protected by authorization
- Add, download, and delete user data to Identity in an ASP.NET Core project
- Enable QR code generation for TOTP authenticator apps in ASP.NET Core
- Migrate Authentication and Identity to ASP.NET Core
- Account confirmation and password recovery in ASP.NET Core
- Two-factor authentication with SMS in ASP.NET Core
- Host ASP.NET Core in a web farm

How to use Identity to secure a Web API backend for SPAs

Article • 09/10/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

ASP.NET Core Identity provides APIs that handle authentication, authorization, and identity management. The APIs make it possible to secure endpoints of a Web API backend with cookie-based authentication. A token-based option is available for clients that can't use cookies, but in using this you are responsible for ensuring the tokens are kept secure. We recommend using cookies for browser-based applications, because, by default, the browser automatically handles them without exposing them to JavaScript.

This article shows how to use Identity to secure a Web API backend for SPAs such as Angular, React, and Vue apps. The same backend APIs can be used to secure Blazor WebAssembly apps.

Prerequisites

The steps shown in this article add authentication and authorization to an ASP.NET Core Web API app that:

- Isn't already configured for authentication.
- Targets `net8.0` or later.
- Can be either minimal API or controller-based API.

Some of the testing instructions in this article use the Swagger UI that's included with the project template. The Swagger UI isn't required to use Identity with a Web API backend.

Install NuGet packages

Install the following NuGet packages:

- Microsoft.AspNetCore.Identity.EntityFrameworkCore - Enables Identity to work with Entity Framework Core (EF Core).
- One that enables EF Core to work with a database, such as one of the following packages:
  - Microsoft.EntityFrameworkCore.SqlServer or
  - Microsoft.EntityFrameworkCore.Sqlite or
  - Microsoft.EntityFrameworkCore.InMemory.

For the quickest way to get started, use the in-memory database.

Change the database later to SQLite or SQL Server to save user data between sessions when testing or for production use. That introduces some complexity compared to in-memory, as it requires the database to be created through migrations, as shown in the EF Core getting started tutorial.

Install these packages by using the NuGet package manager in Visual Studio or the dotnet add package CLI command.

Create an `IdentityDbContext`

Add a class named `ApplicationDbContext` that inherits from IdentityDbContext<TUser>:

C#

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : IdentityDbContext<IdentityUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options) :
        base(options)
    { }
}
```

The code shown provides a special constructor that makes it possible to configure the database for different environments.

Add one or more of the following `using` directives as needed when adding the code shown in these steps.

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
```

Configure the EF Core context

As noted earlier, the simplest way to get started is to use the in-memory database. With in-memory each run starts with a fresh database, and there's no need to use migrations. After the call to `WebApplication.CreateBuilder(args)`, add the following code to configure Identity to use an in-memory database:

C#

```
builder.Services.AddDbContext<ApplicationDbContext>(
    options => options.UseInMemoryDatabase("AppDb"));
```

To save user data between sessions when testing or for production use, change the database later to SQLite or SQL Server.

Add Identity services to the container

After the call to `WebApplication.CreateBuilder(args)`, call AddAuthorization to add services to the dependency injection (DI) container:

C#

```
builder.Services.AddAuthorization();
```

Activate Identity APIs

After the call to `WebApplication.CreateBuilder(args)`, call AddIdentityApiEndpoints<TUser>(IServiceCollection) and AddEntityFrameworkStores<TContext>(IdentityBuilder).

C#

```
builder.Services.AddIdentityApiEndpoints<IdentityUser>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

By default, both cookies and proprietary tokens are activated. Cookies and tokens are issued at login if the `useCookies` query string parameter in the login endpoint is `true`.

Map Identity routes

After the call to `builder.Build()`, call MapIdentityApi<TUser>(IEndpointRouteBuilder) to map the Identity endpoints:

```
C#

app.MapIdentityApi<IdentityUser>();
```

Secure selected endpoints

To secure an endpoint, use the RequireAuthorization extension method on the `Map{Method}` call that defines the route. For example:

```
C#

app.MapGet("/weatherforecast", (HttpContext httpContext) =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = summaries[Random.Shared.Next(summaries.Length)]
        })
        .ToArray();
    return forecast;
})
.WithName("GetWeatherForecast")
.WithOpenApi()
.RequireAuthorization();
```

The `RequireAuthorization` method can also be used to:

- Secure Swagger UI endpoints, as shown in the following example:

  ```
  C#

  app.MapSwagger().RequireAuthorization();
  ```

- Secure with a specific claim or permission, as shown in the following example:

```C#
.RequireAuthorization("Admin");
```

In a controller-based web API project, secure endpoints by applying the [Authorize] attribute to a controller or action.

Test the API

A quick way to test authentication is to use the in-memory database and the Swagger UI that's included with the project template. The following steps show how to test the API with the Swagger UI. Make sure that the Swagger UI endpoints aren't secured.

Attempt to access a secured endpoint

- Run the app and navigate to the Swagger UI.
- Expand a secured endpoint, such as `/weatherforecast` in a project created by the web API template.
- Select **Try it out**.
- Select **Execute**. The response is `401 - not authorized`.

Test registration

- Expand `/register` and select **Try it out**.
- In the **Parameters** section of the UI, a sample request body is shown:

```JSON
{
  "email": "string",
  "password": "string"
}
```

- Replace "string" with a valid email address and password, and then select **Execute**.

  To comply with the default password validation rules, the password must be at least six characters long and contain at least one of each of the following characters:
  - Uppercase letter
  - Lowercase letter
  - Numeric digit

- Nonalphanumeric character

If you enter an invalid email address or a bad password, the result includes the validation errors. Here's an example of a response body with validation errors:

JSON

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "PasswordTooShort": [
      "Passwords must be at least 6 characters."
    ],
    "PasswordRequiresNonAlphanumeric": [
      "Passwords must have at least one non alphanumeric character."
    ],
    "PasswordRequiresDigit": [
      "Passwords must have at least one digit ('0'-'9')."
    ],
    "PasswordRequiresLower": [
      "Passwords must have at least one lowercase ('a'-'z')."
    ]
  }
}
```

The errors are returned in the ProblemDetails format so the client can parse them and display validation errors as needed.

A successful registration results in a `200 - OK` response.

Test login

- Expand `/login` and select **Try it out**. The example request body shows two additional parameters:

JSON

```
{
  "email": "string",
  "password": "string",
  "twoFactorCode": "string",
  "twoFactorRecoveryCode": "string"
}
```

The extra JSON properties aren't needed for this example and can be deleted. Set `useCookies` to `true`.

- Replace "string" with the email address and password that you used to register, and then select **Execute**.

  A successful login results in a `200 - OK` response with a cookie in the response header.

Retest the secured endpoint

After a successful login, rerun the secured endpoint. The authentication cookie is automatically sent with the request, and the endpoint is authorized. Cookie-based authentication is securely built-in to the browser and "just works."

Testing with nonbrowser clients

Some web clients might not include cookies in the header by default:

- If you're using a tool for testing APIs, you might need to enable cookies in the settings.
- The JavaScript `fetch` API doesn't include cookies by default. Enable them by setting `credentials` to the value `include` in the options.
- An `HttpClient` running in a Blazor WebAssembly app needs the `HttpRequestMessage` to include credentials, like the following example:

  ```
  C#

  request.SetBrowserRequestCredential(BrowserRequestCredentials.Include);
  ```

Use token-based authentication

We recommend using cookies in browser-based applications, because, by default, the browser automatically handles them without exposing them to JavaScript.

A custom token (one that is proprietary to the ASP.NET Core identity platform) is issued that can be used to authenticate subsequent requests. The token is passed in the `Authorization` header as a bearer token. A refresh token is also provided. This token allows the application to request a new token when the old one expires without forcing the user to log in again.

The tokens aren't standard JSON Web Tokens (JWTs). The use of custom tokens is intentional, as the built-in Identity API is meant primarily for simple scenarios. The token