Why use App Service?

Azure App Service is a fully managed platform as a service (PaaS) offering for developers. Here are some key features of App Service:

- **Multiple languages and frameworks** - App Service has first-class support for ASP.NET, ASP.NET Core, Java, Node.js, PHP, and Python. You can also run PowerShell and other scripts or executables as background services.

- **Managed production environment** - App Service automatically patches and maintains the OS and language frameworks for you. Spend time writing great apps and let Azure worry about the platform.

- **Containerization and Docker** - Dockerize your app and host a custom Windows or Linux container in App Service. Run sidecar containers of your choice. Migrate your Docker skills directly to App Service.

- **DevOps optimization** - Set up continuous integration and deployment with Azure DevOps, GitHub, BitBucket, Docker Hub, or Azure Container Registry. Promote updates through test and staging environments. Manage your apps in App Service by using Azure PowerShell or the cross-platform command-line interface (CLI).

- **Global scale with high availability** - Scale up or out manually or automatically. Host your apps anywhere in the global Microsoft datacenter infrastructure, and the App Service SLA promises high availability.

- **Connections to SaaS platforms and on-premises data** - Choose from many hundreds of connectors for enterprise systems (such as SAP), SaaS services (such as Salesforce), and internet services (such as Facebook). Access on-premises data using Hybrid Connections and Azure Virtual Network.

- **Security and compliance** - App Service is ISO, SOC, and PCI compliant. Create IP address restrictions and managed service identities. Protect against subdomain takeovers.

- **Authentication** - Authenticate users using the built-in authentication component. Authenticate users with Microsoft Entra ID, Google, Facebook, X, or Microsoft accounts.

- **Application templates** - Choose from an extensive list of application templates in the Azure Marketplace, such as WordPress, Joomla, and Drupal.

- **Visual Studio and Visual Studio Code integration** - Dedicated tools in Visual Studio and Visual Studio Code streamline the work of creating, deploying, and debugging.

- **Java tools integration** - Develop and deploy to Azure without leaving your favorite development tools, such as Maven, Gradle, Visual Studio Code, IntelliJ, and Eclipse.

- **API and mobile features** - App Service provides turn-key CORS support for RESTful API scenarios and simplifies mobile app scenarios by enabling authentication, offline data sync, push notifications, and more.

- **Serverless code** - Run a code snippet or script on-demand without having to explicitly provision or manage infrastructure, and pay only for the compute time your code actually uses. (See Azure Functions.)

Limitations

- App Service on Linux isn't supported on the Shared pricing tier.
- The Azure portal shows only features that currently work for Linux apps. As features are enabled, they're activated on the portal.
- When deployed to built-in images, your code ad content are allocated a storage volume for web content, backed by Azure Storage. The disk latency of this volume is higher and more variable than the latency of the container filesystem. Apps that require heavy read-only access to content files might benefit from the custom container option, which places files in the container filesystem instead of on the content volume.

1.1 Azure App Service TLS overview

Transport Layer Security (TLS) is a widely adopted security protocol designed to secure connections and communications between servers and clients.

App Service allows customers to use TLS/SSL certificates to secure incoming requests to their web apps. App Service currently supports different set of TLS features for customers to secure their web apps.

For incoming requests to your web app, App Service supports TLS versions 1.0, 1.1, 1.2, and 1.3.

Set Minimum TLS Version

Follow these steps to change the Minimum TLS version of your App Service resource:

1. Browse to your app in the Azure portal
2. In the left menu, select configuration and then select the General settings tab.
3. On Minimum Inbound TLS Version, using the dropdown, select your desired version.
4. Select Save to save the changes.

TLS 1.0 and 1.1

TLS 1.0 and 1.1 are considered legacy protocols and are no longer considered secure. It's generally recommended for customers to use TLS 1.2 or above as the minimum TLS version.

When creating a web app, the default minimum TLS version is TLS 1.2.

To ensure backward compatibility for TLS 1.0 and TLS 1.1, App Service will continue to support TLS 1.0 and 1.1 for incoming requests to your web app. However, since the default minimum TLS version is set to TLS

1.2, you need to update the minimum TLS version configurations on your web app to either TLS 1.0 or 1.1 so the requests won't be rejected.
The **minimum TLS cipher suite** includes a fixed list of cipher suites with an optimal priority order that you cannot change.
Reordering or reprioritizing the cipher suites is not recommended as it could expose your web apps to weaker encryption.

A cipher suite is a set of instructions that contains algorithms and protocols to help secure network connections between clients and servers.

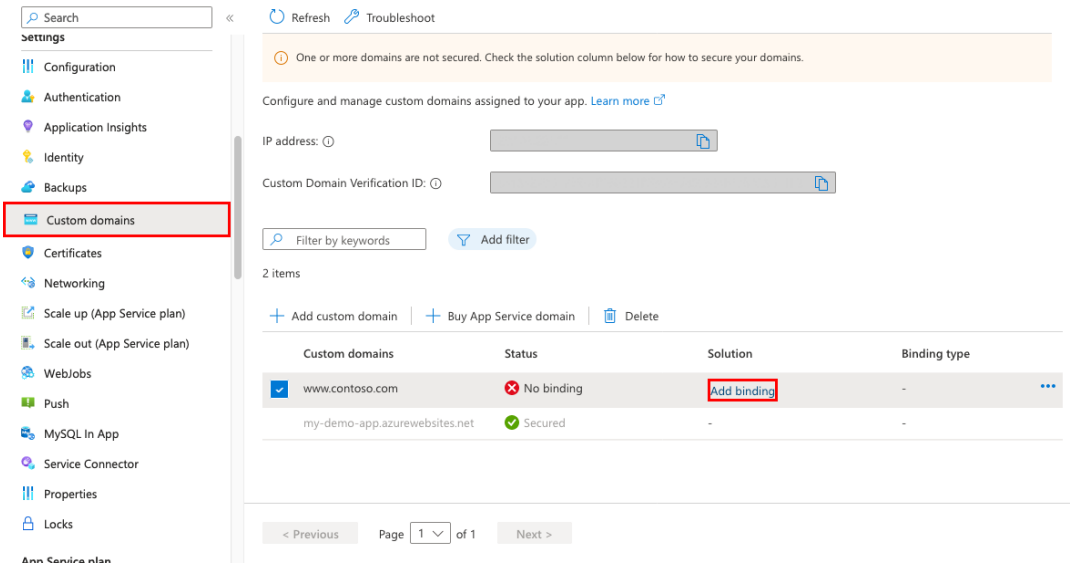
For **App Service Environments** with **FrontEndSSLCipherSuiteOrder** cluster setting, you need to update your settings to include two TLS 1.3 cipher suites (TLS_AES_256_GCM_SHA384 and TLS_AES_128_GCM_SHA256).

End-to-end (E2E) TLS encryption is available in Standard App Service plans and higher.

Front-end intra-cluster traffic between App Service front ends and the workers running application workloads can now be encrypted.

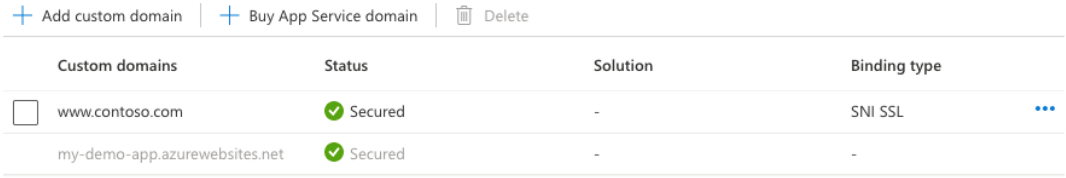
1.2 Provide security for a custom DNS name with a TLS/SSL binding in App Service

1. From the left menu, select **App Services** > **<app-name>**.
2. From the left navigation of your app, select **Custom domains**.(No adding custom domains)



3. Next to the custom domain, select **Add binding**.
4. If your app already has a certificate for the selected custom domain, you can select it in **Certificate**. If not, you must add a certificate using one of the selections in **Source**.

5. In **TLS/SSL type**, select either **SNI SSL** or **IP based SSL**.
6. When adding a new certificate, validate the new certificate by selecting **Validate**.
7. Select **Add**.



Can I disable the forced redirect from HTTP to HTTPS?

By default, App Service forces a redirect from HTTP requests to HTTPS.

In App Service, **TLS termination** happens at the network load balancers, so all HTTPS requests reach your app as unencrypted HTTP requests. If your app logic needs to check if the user requests are encrypted, inspect the X-Forwarded-Proto header.

How do I make sure that the app's IP address doesn't change when I make changes to the certificate binding?

Your inbound IP address can change when you delete a binding, even if that binding is IP SSL. This is especially important when you renew a certificate that's already in an IP SSL binding. To avoid a change in your app's IP address, follow these steps, in order:

1. Upload the new certificate.
2. Bind the new certificate to the custom domain you want without deleting the old one. This action replaces the binding instead of removing the old one.
3. Delete the old certificate.

1.3 Add and manage TLS/SSL certificates in Azure App Service

You can add digital security certificates to use in your application code or to help secure custom DNS names in Azure App Service, which provides a highly scalable, self-patching web hosting service.

Currently called Transport Layer Security (TLS) certificates, also previously known as Secure Socket Layer (SSL) certificates, these private or public certificates help you secure internet connections by encrypting data sent between your browser, websites that you visit, and the website server.

The app's App Service plan must be in the Basic, Standard, Premium, or Isolated tier.

The following table lists the options for you to add certificates in App Service:

Option	Description
Create a free App Service managed certificate	A private certificate that's free of charge and easy to use if you just need to improve security for your custom domain in App Service.
Import an App Service certificate	A private certificate that's managed by Azure. It combines the simplicity of automated certificate management and the flexibility of renewal and export options.
Import a certificate from Key Vault	Useful if you use Azure Key Vault to manage your PKCS12 certificates. See Private certificate requirements.
Upload a private certificate	If you already have a private certificate from a third-party provider, you can upload it. See Private certificate requirements.
Upload a public certificate	Public certificates aren't used to secure custom domains, but you can load them into your code if you need them to access remote resources.

1.4 Use a TLS/SSL certificate in your code in Azure App Service

In your application code, you can access the public or private certificates you add to App Service.

Your app code may act as a client and access an external service that requires certificate authentication, or it may need to perform cryptographic tasks.

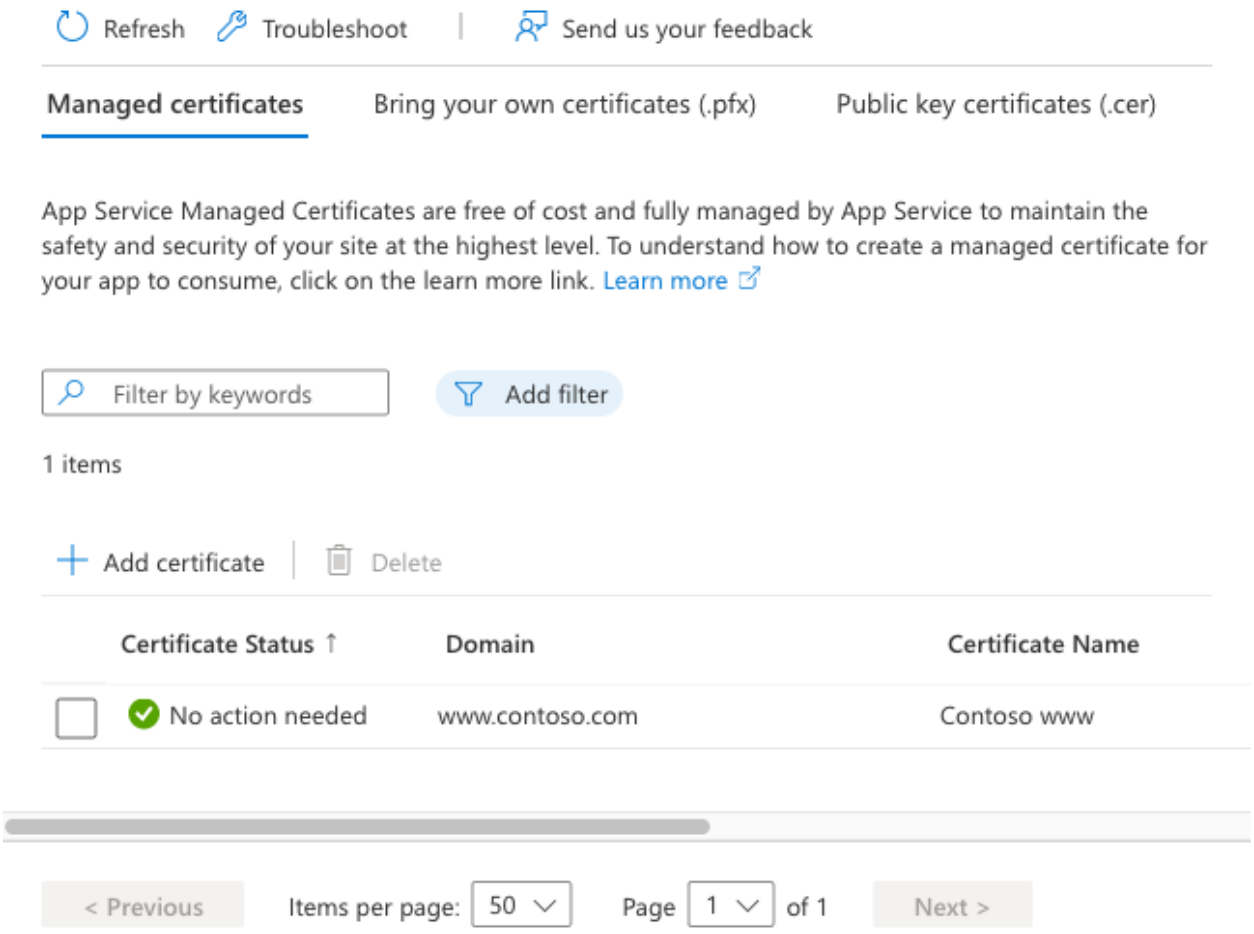
This approach to using certificates in your code makes use of the TLS functionality in App Service, which requires your app to be in Basic tier or higher.

If your app is in *Free or Shared tier*, you can *include the certificate file in your app repository*.

Find the thumbprint

In the Azure portal, from the left menu, select **App Services** > **<app-name>**.

From the left navigation of your app, select **Certificates**, then select **Bring your own certificates (.pfx)** or **Public key certificates (.cer)**.
Find the certificate you want to use and copy the thumbprint.



Make the certificate accessible

To *access a certificate* in your app code, add its thumbprint to the **WEBSITE_LOAD_CERTIFICATES** app setting, by running the following command in the Cloud Shell:

```
az webapp config appsettings set --name <app-name> --resource-group <resource-group-name> --settings WEBSITE_LOAD_CERTIFICATES=<comma-separated-certificate-thumbprints>
```

When WEBSITE_LOAD_CERTIFICATES is set *, all previously added certificates are accessible to application code. If you add a certificate to your app later, restart the app to make the new certificate accessible to your app.

Load certificate in Windows apps

The WEBSITE_LOAD_CERTIFICATES app setting makes the specified certificates accessible to your Windows hosted app in the Windows certificate store, in Current User\My.

```
using System;
using System.Linq;
using System.Security.Cryptography.X509Certificates;

string certThumbprint = "E661583E8FABEF4C0BEF694CBC41C28FB81CD870";
bool validOnly = false;

using (X509Store certStore = new X509Store(StoreName.My, StoreLocation.CurrentUser))
{
  certStore.Open(OpenFlags.ReadOnly);

  X509Certificate2Collection certCollection = certStore.Certificates.Find(
                        X509FindType.FindByThumbprint,
                        // Replace below with your certificate's thumbprint
                        certThumbprint,
                        validOnly);
  // Get the first cert with the thumbprint
  X509Certificate2 cert = certCollection.OfType<X509Certificate2>().FirstOrDefault();

  if (cert is null)
     throw new Exception($"Certificate with thumbprint {certThumbprint} was not found");

  // Use certificate
  Console.WriteLine(cert.FriendlyName);

  // Consider to call Dispose() on the certificate after it's being used, available in .NET 4.6 and later
}
```

Load certificate in Linux/Windows containers

The WEBSITE_LOAD_CERTIFICATES app setting makes the specified certificates accessible to your Windows or Linux custom containers (including built-in Linux containers) as files. The files are found under the following directories:

Container platform	Public certificates	Private certificates
Windows container	C:\appservice\certificates\public	C:\appservice\certificates\private
Linux container	/var/ssl/certs	/var/ssl/private

When updating (renewing) a certificate

When you renew a certificate and add it to your app, it gets a new thumbprint, which also needs to be made accessible. How it works depends on your certificate type.

If you manually upload the public or private certificate:

- If you list thumbprints explicitly in WEBSITE_LOAD_CERTIFICATES, add the new thumbprint to the app setting.

- If WEBSITE_LOAD_CERTIFICATES is set to *, restart the app to make the new certificate accessible.

If you renew a certificate in Key Vault, such as with an App Service certificate, the daily sync from Key Vault makes the necessary update automatically when synchronizing your app with the renewed certificate.

- If WEBSITE_LOAD_CERTIFICATES contains the old thumbprint of your renewed certificate, the daily sync updates the old thumbprint to the new thumbprint automatically.

- If WEBSITE_LOAD_CERTIFICATES is set to *, the daily sync makes the new certificate accessible automatically.

Azure App Service on Linux FAQ

With the release of App Service on Linux, we're working on adding features and making improvements to our platform. This article provides answers to questions that our customers have been asking us recently. If you have a question, comment on this article.

Built-in images
What are the expected values for the Startup File section when I configure the runtime stack?

Stack	Expected Value
Java SE	the command to start your JAR app (for example, java -jar /home/site/wwwroot/app.jar --server.port=80)
Tomcat	the location of a script to perform any necessary configurations (for example, /home/site/deployments/tools/startup_script.sh)
Node.js	the PM2 configuration file or your script file
.NET Core	the compiled DLL name as dotnet <myapp>.dll
PHP	optional custom startup
Python	optional startup script
Ruby	the Ruby script that you want to initialize your app with

Management

What happens when I press the restart button in the Azure portal?
This action is the same as a Docker restart.

Can I use Secure Shell (SSH) to connect to the app container virtual machine (VM)?
Yes, you can do that through the source control management (SCM) site.

Note
You can also connect to the app container directly from your local development machine using SSH, SFTP, or Visual Studio Code (for live debugging Node.js apps). For more information, see **Remote debugging and SSH in App Service on Linux**.

How can I create a Linux App Service plan through an SDK or an Azure Resource Manager template?
Set the **reserved** field of the app service to *true*.

Continuous integration and deployment

My web app still uses an old Docker container image after I've updated the image on Docker Hub. Do you support continuous integration and deployment of custom containers?

Yes, to set up continuous integration/deployment for Azure Container Registry or DockerHub, by following Continuous Deployment with Web App for Containers. For private registries, you can refresh the container by stopping and then starting your web app. Or you can change or add a dummy application setting to force a refresh of your container.

Do you support staging environments?
Yes.

Can I use 'WebDeploy/MSDeploy' to deploy my web app?
Yes, you need to set an app setting called WEBSITE_WEBDEPLOY_USE_SCM to *false*.

Git deployment of my application fails when using Linux web app. How can I work around the issue?
If Git deployment fails to your Linux web app, choose one of the following options to deploy your application code:

- Use the Continuous Delivery (Preview) feature: You can store your app's source code in an Azure DevOps Git repo or GitHub repo to use Azure Continuous Delivery. For more information, see How to configure Continuous Delivery for Linux web app.
- Use the ZIP deploy API: To use this API, SSH into your web app and go to the folder where you want to deploy your code. Run the following code:

BashCopy
curl -X POST -u <user> --data-binary @<zipfile> https://{your-sitename}.scm.azurewebsites.net/api/zipdeploy
If you get an error that the curl command is not found, make sure you install curl by using apt-get install curl before you run the previous curl command.

Language support
I want to use web sockets in my Node.js application, any special settings, or configurations to set?
Yes, disable perMessageDeflate in your server-side Node.js code. For example, if you are using socket.io, use the following code:
Node.jsCopy
```
const io = require('socket.io')(server,{
  perMessageDeflate :false
});
```

Do you support uncompiled .NET Core apps?
Yes.

Do you support Composer as a dependency manager for PHP apps?

Yes, during a Git deployment, Kudu should detect that you're deploying a PHP application (thanks to the presence of a composer.lock file), and Kudu will then trigger a composer install.

Custom containers

Can I use Managed Identities with App Service when pulling images form ACR?
Yes, this functionality is available from the Azure CLI. You can use system-assigned or user-assigned identities. This functionality isn't currently supported in the Azure portal.

I'm using my own custom container. I want the platform to mount an SMB share to the `/home/` directory.
If WEBSITES_ENABLE_APP_SERVICE_STORAGE setting is **unspecified** or set to *false*, the /home/ directory **will not be shared** across scale instances, and files written **will not persist** across restarts. Explicitly setting WEBSITES_ENABLE_APP_SERVICE_STORAGE to *true* will enable the mount. Once this is set to true, if you wish to disable the mount, you need to explicitly set WEBSITES_ENABLE_APP_SERVICE_STORAGE to *false*.

My container fails to start with "no space left on device". What does this error mean?
App Service on Linux uses two different types of storage:

- File system storage: The file system storage is included in the App Service plan quota. It's used when files are saved to the persistent storage that's rooted in the /home directory.
- Host disk space: The host disk space is used to store container images. It's managed by the platform through the docker storage driver.

The host disk space is separate from the file system storage quota. It's not expandable and there is a 15 GB limit for each instance. It's used to store any custom images on the worker. You might be able to use larger than 15 GBs depending on the exact availability of host disk space, but this isn't guaranteed.
If the container's writable layer saves data outside of the /home directory or a mounted azure storage path, the host disk space will also be consumed.
The platform routinely cleans the host disk space to remove unused containers. If the container writes a large quantity of data outside of the /home directory or Bring Your Own Storage (BYOS), it will result in startup failures or runtime exceptions once the host disk space limit is exceeded.
We recommend that you keep your container images as small as possible and write data to the persistent storage or BYOS when running on Linux App Service. If not possible, you have to split the App Service plan because the host disk space is fixed and shared between all containers in the App Service Plan.

My custom container takes a long time to start, and the platform restarts the container before it finishes starting up.
You can configure the amount of time the platform will wait before it restarts your container. To do so, set the WEBSITES_CONTAINER_START_TIME_LIMIT app setting to the value you want. The default value is 230 seconds, and the maximum value is 1800 seconds.

What is the format for the private registry server URL?
Provide the full registry URL, including http:// or https://.

What is the format for the image name in the private registry option?

Add the full image name, including the private registry URL (for example, myacr.azurecr.io/dotnet:latest). Image names that use a custom port cannot be entered through the portal. To set docker-custom-image-name, use the az command-line tool.

Can I expose more than one port on my custom container image?
We don't support exposing more than one port.

Can I bring my own storage?
Yes, bring your own storage is in preview.

Why can't I browse my custom container's file system or running processes from the SCM site?
The SCM site runs in a separate container. You can't check the file system or running processes of the app container.

Do I need to implement HTTPS in my custom container?
No, the platform handles HTTPS termination at the shared front ends.

Do I need to use WEBSITES_PORT for custom containers?
Yes, this is required for custom containers. To manually configure a custom port, use the EXPOSE instruction in the Dockerfile and the app setting, WEBSITES_PORT, with a port value to bind on the container.

Can I use ASPNETCORE_URLS in the Docker image?
Yes, overwrite the environmental variable before .NET core app starts. E.g. In the init.sh script: export ASPNETCORE_URLS={Your value}

Multi-container with Docker Compose

How do I configure Azure Container Registry (ACR) to use with multi-container?
In order to use ACR with multi-container, **all container images** need to be hosted on the same ACR registry server. Once they are on the same registry server, you will need to create application settings and then update the Docker Compose configuration file to include the ACR image name.
Create the following application settings:

- DOCKER_REGISTRY_SERVER_USERNAME
- DOCKER_REGISTRY_SERVER_URL (full URL, ex: https://<server-name>.azurecr.io)
- DOCKER_REGISTRY_SERVER_PASSWORD (enable admin access in ACR settings)

Within the configuration file, reference your ACR image like the following example:
YAMLCopy
image: <server-name>.azurecr.io/<image-name>:<tag>

How do I know which container is internet accessible?

- Only one container can be open for access
- Only port 80 and 8080 is accessible (exposed ports)

Here are the rules for determining which container is accessible - in the order of precedence:

- Application setting WEBSITES_WEB_CONTAINER_NAME set to the container name
- The first container to define port 80 or 8080

- If neither of the above is true, the first container defined in the file will be accessible (exposed)

How do I use depends_on?
The depends_on option is *unsupported* on App Service and it'll be ignored. Just as the control startup and shutdown recommendation from Docker, App Service Multi-container apps should check dependencies through application code - both at startup and disconnection. The example code below shows a Python app checking to see if a Redis container is running.

```
PythonCopy
import time
import redis
from flask import Flask
app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)
def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)
@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello from Azure App Service team! I have been seen {} times.\n'.format(count)
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

Web Sockets

Web Sockets are supported on Linux apps. The webSocketsEnabled ARM setting does not apply to Linux apps since Web Sockets are always enabled for Linux.

Important
Web Sockets are now supported for Linux apps on Free App Service plans. We support up to five web socket connections on Free App Service plans. Exceeding this limit results in an HTTP 429 (Too Many Requests) response.

Pricing and SLA

What is the pricing, now that the service is generally available?
Pricing varies by SKU and region but you can see more details at our pricing page: App Service Pricing.
Other questions

How does the container warmup request work?

When Azure App Services starts your container, the warmup request sends an HTTP request to the /robots933456.txt endpoint of your application. This is simply a dummy endpoint, but your application needs to reply with any non-5XX status code. If your application logic doesn't reply with any HTTP status code to nonexistent endpoints, the warmup request can't receive a response and it perpetually restarts your container. The warmup request also might fail due to port misconfiguration.
To ensure that the port is correctly configured on Azure App Services, see the question *How do I specify port in my Linux container?*

Is it possible to increase the container warmup request timeout?
The warmup request by default fails after waiting 240 seconds for a reply from the container. You may increase the container warmup request timeout by adding the application setting WEBSITES_CONTAINER_START_TIME_LIMIT with a value between 240 and 1800 seconds.

How do I specify port in my Linux container?

Container type	Description	How to set/use port
Built-in containers	If you select a language/framework version for a Linux app, a predefined container is selected for you.	To point your app code to the right port, use the PORT environment variable.
Custom containers	You have full control over the container.	App Service has no control about which port your container listens on. What it does need is to know which port to forward requests to. If your container listens to port 80 or 8080, App Service is able to automatically detect it. If it listens to any other port, you need to set the WEBSITES_PORT app setting to the port number, and App Service forwards requests to that port in the container. The WEBSITES_PORT app setting does not have any effect within the container, and you can't access it as an environment variable within the container.

Can I use a file based database (like SQLite) with my Linux Webapp?
The file system of your application is a mounted network share. This enables scale out scenarios where your code needs to be executed across multiple hosts. Unfortunately this blocks the use of file-based database providers like SQLite since it's not possible to acquire exclusive locks on the database file. We recommend a managed database service: Azure SQL, Azure Database for MySQL or Azure Database for PostgreSQL

What are the supported characters in application settings names?
You can use only letters (A-Z, a-z), numbers (0-9), and the underscore character (_) for application settings.

Where can I request new features?
You can submit your idea at the Web Apps feedback forum. Add "[Linux]" to the title of your idea.

2.1 Configure a custom container for Azure App Service

Change the Docker image of a custom container

To change an existing custom container from the current Docker image to a new image, use the following command:

az webapp config container set --name <app-name> --resource-group <group-name> --docker-custom-image-name <docker-hub-repo>/<image>

Use an image from a private registry

To use an image from a private registry, such as Azure Container Registry, run the following command:

az webapp config container set --name <app-name> --resource-group <group-name> --docker-custom-image-name <image-name> --docker-registry-server-url <private-repo-url> --docker-registry-server-user <username> --docker-registry-server-password <password>

For *<username>* and *<password>*, supply the sign-in credentials for your private registry account.

Use an image from a network protected registry

To connect and pull from a registry inside a virtual network or on-premises, your app must integrate with a virtual network (VNET). VNET integration is also needed for Azure Container Registry with private endpoint. When your network and DNS resolution is configured, you enable the routing of the image pull through the virtual network by configuring the vnetImagePullEnabled site setting:

az resource update –resource-group <group-name> --name <app-name> --resource-type "Microsoft.Web/sites" –set properties.vnetImagePullEnabled [true|false]

Configure port number

By default, App Service assumes your custom container is listening on port 80. If your container listens to a different port, set the WEBSITES_PORT app setting in your App Service app. You can set it via the Cloud Shell. In Bash:

az webapp config appsettings set --resource-group <group-name> --name <app-name> --settings WEBSITES_PORT=8000

In PowerShell:

Set-AzWebApp -ResourceGroupName <group-name> -Name <app-name> -AppSettings @{"WEBSITES_PORT"="8000"}
App Service currently allows your container to expose only one port for HTTP requests.

Configure environment variables

Your custom container might use environment variables that need to be supplied externally. You can pass them in via the Cloud Shell. In Bash:
az webapp config appsettings set --resource-group <group-name> --name <app-name> --settings DB_HOST="myownserver.mysql.database.azure.com"

If your app uses images from a private registry or from Docker Hub, credentials for accessing the repository are saved in environment
variables: DOCKER_REGISTRY_SERVER_URL, DOCKER_REGISTRY_SERVER_USERNAME and DOCKER_REGISTRY_SERVER_PASSWORD. Because of security risks, none of these reserved variable names are exposed to the application.

This method works both for single-container apps or multi-container apps, where the environment variables are specified in the *docker-compose.yml* file.

Use persistent shared storage

You can use the */home* directory in your custom container file system to persist files across restarts and share them across instances.

az webapp config appsettings set --resource-group <group-name> --name <app-name> --settings WEBSITES_ENABLE_APP_SERVICE_STORAGE=true

Access diagnostic logs

You can access the console logs generated from inside the container.

az webapp log config --name <app-name> --resource-group <resource-group-name> --docker-container-logging filesystem

Once container logging is turned on, run the following command to see the log stream:

az webapp log tail --name <app-name> --resource-group <resource-group-name>
If you don't see console logs immediately, check again in 30 seconds.

Multi-container is currently in preview. The following App Service platform features aren't supported:

- Authentication / Authorization
- Managed Identities
- CORS
- Virtual network integration isn't supported for Docker Compose scenarios
- Docker Compose on Azure App Services currently has a limit of 4,000 characters at this time.

2.2 Migrate custom software to Azure App Service using a custom container

- Push a custom Docker image to Azure Container Registry.
- Deploy the custom image to App Service.
- Configure environment variables.
- Pull the image into App Service by using a managed identity.
- Access diagnostic logs.
- Enable CI/CD from Azure Container Registry to App Service.
- Connect to the container by using SSH.

I. Create a user-assigned managed identity
az group create --name msdocs-custom-container-tutorial --location westeurope

az identity create --name myID --resource-group msdocs-custom-container-tutorial

II. Create a container registry
az acr create --name <registry-name> --resource-group msdocs-custom-container-tutorial --sku Basic --admin-enabled true

az acr credential show --resource-group msdocs-custom-container-tutorial --name <registry-name>

III. Push the sample image to Azure Container Registry

docker login <registry-name>.azurecr.io --username <registry-username>

docker tag appsvc-tutorial-custom-image <registry-name>.azurecr.io/appsvc-tutorial-custom-image:latest

docker push <registry-name>.azurecr.io/appsvc-tutorial-custom-image:latest

IV. Authorize the managed identity for your registry

V. Create the web app

az appservice plan create --name myAppServicePlan --resource-group msdocs-custom-container-tutorial --is-linux

az webapp create --resource-group msdocs-custom-container-tutorial --plan myAppServicePlan --name <app-name> --deployment-container-image-name <registry-name>.azurecr.io/appsvc-tutorial-custom-image:latest

VI. Configure the web app

az webapp config appsettings set --resource-group msdocs-custom-container-tutorial --name <app-name> --settings WEBSITES_PORT=8000

3.1 Run your app in Azure App Service directly from a ZIP package

In Azure App Service, you can run your apps directly from a deployment ZIP package file.

All other deployment methods in App Service have something in common: your files are deployed to *D:\home\site\wwwroot* in your app (or */home/site/wwwroot* for Linux apps). Since the same directory is used by your app at runtime, it's possible for deployment to fail because of file lock conflicts, and for the app to behave unpredictably because some of the files are not yet updated.

In contrast, when you run directly from a package, the files in the package are not copied to the *wwwroot* directory. Instead, the ZIP package itself gets mounted directly as the read-only *wwwroot* directory. There are several benefits to running directly from a package:

- Eliminates file lock conflicts between deployment and runtime.
- Ensures only full-deployed apps are running at any time.
- Can be deployed to a production app (with restart).
- Improves the performance of Azure Resource Manager deployments.
- May reduce cold-start times, particularly for JavaScript functions with large npm package trees.

This directory should contain the entry file to your web app, such as *index.html*, *index.php*, and *app.js*.

It can also contain package management files like *project.json*, *composer.json*, *package.json*, *bower.json*, and *requirements.txt*.

```
# Bash
zip -r <file-name>.zip .

# PowerShell
Compress-Archive -Path * -DestinationPath <file-name>.zip
```

Enable running from package

The WEBSITE_RUN_FROM_PACKAGE app setting enables running from a package. To set it, run the following command with Azure CLI.

```
az webapp config appsettings set --resource-group <group-name> --name <app-name> --settings WEBSITE_RUN_FROM_PACKAGE="1"
```

WEBSITE_RUN_FROM_PACKAGE="1" lets you run your app from a package local to your app. You can also run from a remote package.

Run from external URL instead

You can also run a package from an external URL, such as Azure Blob Storage. You can use the Azure Storage Explorer to upload package files to your Blob storage account.

You should use a private storage container with a Shared Access Signature (SAS) or use a managed identity to enable the App Service runtime to access the package securely.

Note
Currently, an existing App Service resource that runs a local package cannot be migrated to run from a remote package. You will have to create a new App Service resource configured to run from an external URL.
Once you upload your file to Blob storage and have an SAS URL for the file, set the WEBSITE_RUN_FROM_PACKAGE app setting to the URL. The following example does it by using Azure

az webapp config appsettings set --name <app-name> --resource-group <resource-group-name> --settings WEBSITE_RUN_FROM_PACKAGE=https://myblobstorage.blob.core.windows.net/content/SampleCoreMVCApp.zip?st=2018-02-13T09%3A48%3A00Z&se=2044-06-14T09%3A48%3A00Z&sp=rl&sv=2017-04-17&sr=b&sig=bNrVrEFzRHQB17GFJ7boEanetyJ9DGwBSV8OM3Mdh%2FM%3D

If you publish an updated package with the same name to Blob storage, you need to restart your app so that the updated package is loaded into App Service.

Access a package in Azure Blob Storage using a managed identity

You can configure Azure Blob Storage to authorize requests with Microsoft Entra ID. This configuration means that instead of generating a SAS key with an expiration, you can instead rely on the application's managed identity. By default, the app's system-assigned identity is used.

If you wish to specify a user-assigned identity, you can set the WEBSITE_RUN_FROM_PACKAGE_BLOB_MI_RESOURCE_ID app setting to the resource ID of that identity. The setting can also accept SystemAssigned as a value, which is equivalent to omitting the setting.

To enable the package to be fetched using the identity:

1. Ensure that the blob is configured for private access.
2. Grant the identity the Storage Blob Data Reader role with scope over the package blob. See Assign an Azure role for access to blob data for details on creating the role assignment.
3. Set the WEBSITE_RUN_FROM_PACKAGE application setting to the blob URL of the package. This URL is usually of the form https://{storage-account-name}.blob.core.windows.net/{container-name}/{path-to-package} or similar.

Deploy WebJob files when running from package
There are two ways to deploy WebJob files when you enable running an app from package:

- Deploy in the same ZIP package as your app: include them as you normally would in <project-root>\app_data\jobs\... (which maps to the deployment path \site\wwwroot\app_data\jobs\... as specified in the WebJobs quickstart).

- Deploy separately from the ZIP package of your app: Since the usual deployment path \site\wwwroot\app_data\jobs\... is now read-only, you can't deploy WebJob files there.

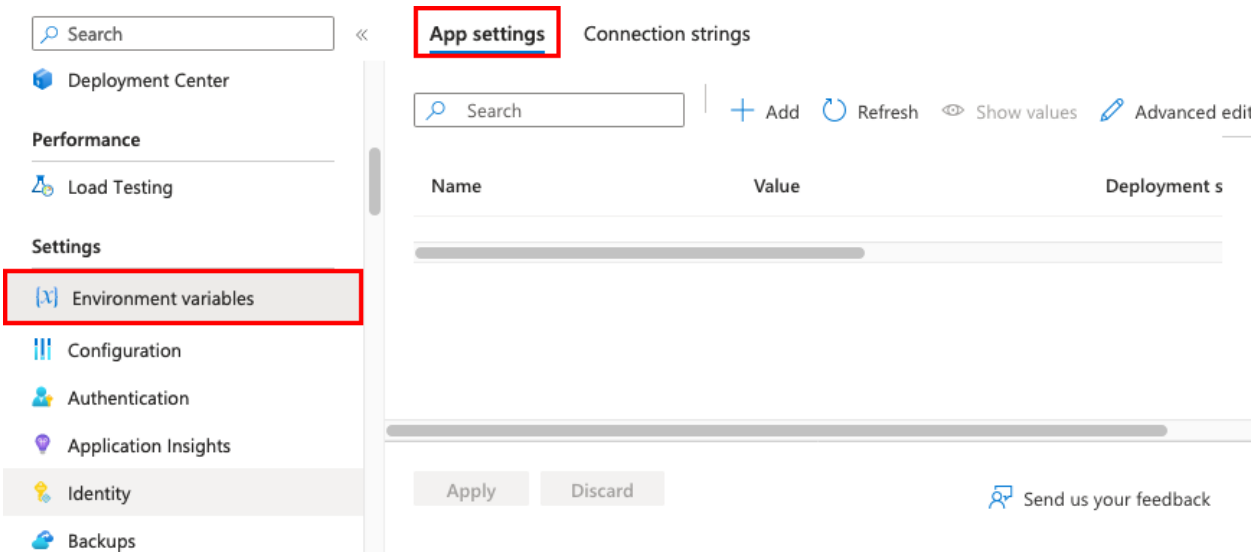
Instead, deploy WebJob files to \site\jobs\..., which is not read only. WebJobs deployed to \site\wwwroot\app_data\jobs\... and \site\jobs\... both run.

4.1 Configure an App Service app for secrets

In App Service, app settings are variables passed as environment variables to the application code.

For Linux apps and custom containers, App Service passes app settings to the container using the --env flag to set the environment variable in the container.

In either case, they're injected into your app environment at app startup. When you add, remove, or edit app settings, App Service triggers an app restart.



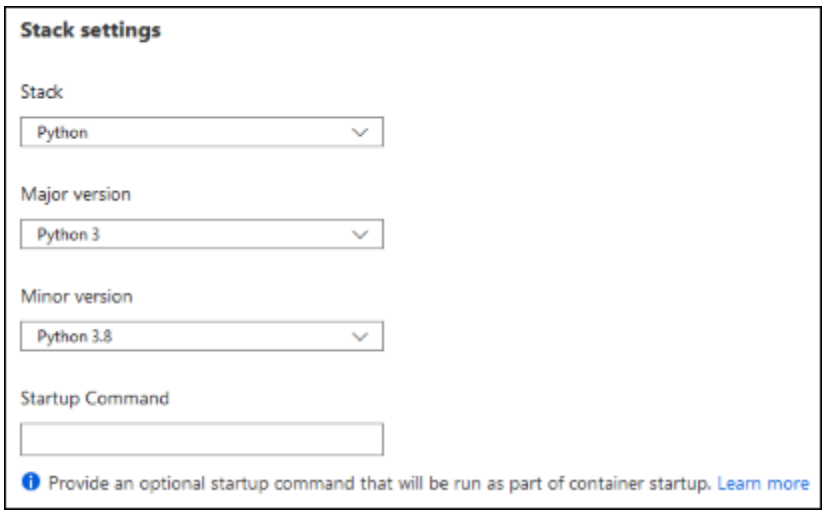
Configure connection strings

Consider more secure connectivity options that don't require connection secrets at all.

For ASP.NET and ASP.NET Core developers, setting connection strings in App Service are like setting them in <connectionStrings> in *Web.config*, but the values you set in App Service override the ones in *Web.config*. You can keep development settings (for example, a database file) in *Web.config* and production secrets (for example, SQL Database credentials) safely in App Service. The same code uses your development settings when you debug locally, and it uses your production secrets when deployed to Azure.
For other language stacks, it's better to use app settings instead, because connection strings require special formatting in the variable keys in order to access the values.

- Stack settings: The software stack to run the app, including the language and SDK versions.

For Linux apps, you can select the language runtime version and set an optional Startup command or a startup command file.



- Platform settings: Lets you configure settings for the hosting platform, including:
    - Platform bitness: 32-bit or 64-bit. For Windows apps only.
    - FTP state: Allow only FTPS or disable FTP altogether.
    - HTTP version: Set to 2.0 to enable support for HTTPS/2 protocol.

Note
Most modern browsers support HTTP/2 protocol over TLS only, while non-encrypted traffic continues to use HTTP/1.1. To ensure that client browsers connect to your app with HTTP/2, secure your custom DNS name. For more information, see Secure a custom DNS name with a TLS/SSL binding in Azure App Service.

- Web sockets: For ASP.NET SignalR or socket.io, for example.

- **Always On**: Keeps the app loaded even when there's no traffic. When Always On isn't turned on (default), the app is unloaded after 20 minutes without any incoming requests. The unloaded app can cause high latency for new requests because of its warm-up time. When Always On is turned on, the front-end load balancer sends a GET request to the application root every five minutes. It's important to ensure this request receives a 200 OK response to ensure any re-imaging operations are performed correctly. The continuous ping prevents the app from being unloaded.

Always On is required for continuous WebJobs or for WebJobs that are triggered using a CRON expression.

- Session affinity: In a multi-instance deployment, ensure that the client is routed to the same instance for the life of the session. You can set this option to Off for stateless applications.

- Session affinity proxy: Session affinity proxy can be turned on if your app is behind a reverse proxy (like Azure Application Gateway or Azure Front Door) and you are using the default host name. The domain for the session affinity cookie will align with the forwarded host name from the reverse proxy.

- HTTPS Only: When enabled, all HTTP traffic is redirected to HTTPS.
- Minimum TLS version: Select the minimum TLS encryption version required by your app.

- Debugging: Enable remote debugging for ASP.NET, ASP.NET Core, or Node.js apps. This option turns off automatically after 48 hours.
- Incoming client certificates: require client certificates in mutual authentication.

4.2 Configure default documents

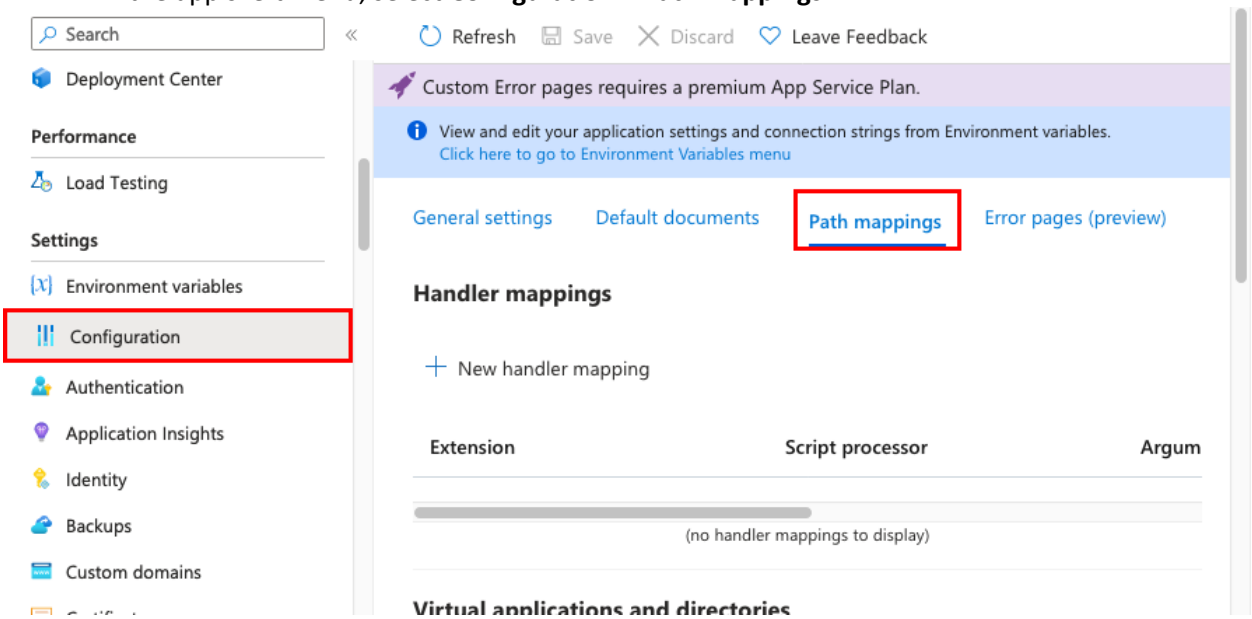
This setting is only for Windows apps.

The default document is the web page that's displayed at the root URL of an App Service app. The first matching file in the list is used. If the app uses modules that route based on URL instead of serving static content, there's no need for default documents.

4.3 Configure handler mappings

For Windows apps, you can customize the IIS handler mappings and virtual applications and directories. Handler mappings let you add custom script processors to handle requests for specific file extensions.
To add a custom handler:

1. In the Azure portal, search for and select **App Services**, and then select your app.
2. In the app's left menu, select **Configuration** > **Path mappings**.



3. Select **New handler mapping**. Configure the handler as follows:
    - **Extension**. The file extension you want to handle, such as **.php* or *handler.fcgi*.
    - **Script processor**. The absolute path of the script processor to you. Requests to files that match the file extension are processed by the script processor. Use the path D:\home\site\wwwroot to refer to your app's root directory.
    - **Arguments**. Optional command-line arguments for the script processor.
4. Select **OK**. Don't forget to select **Save** in the **Configuration** page.

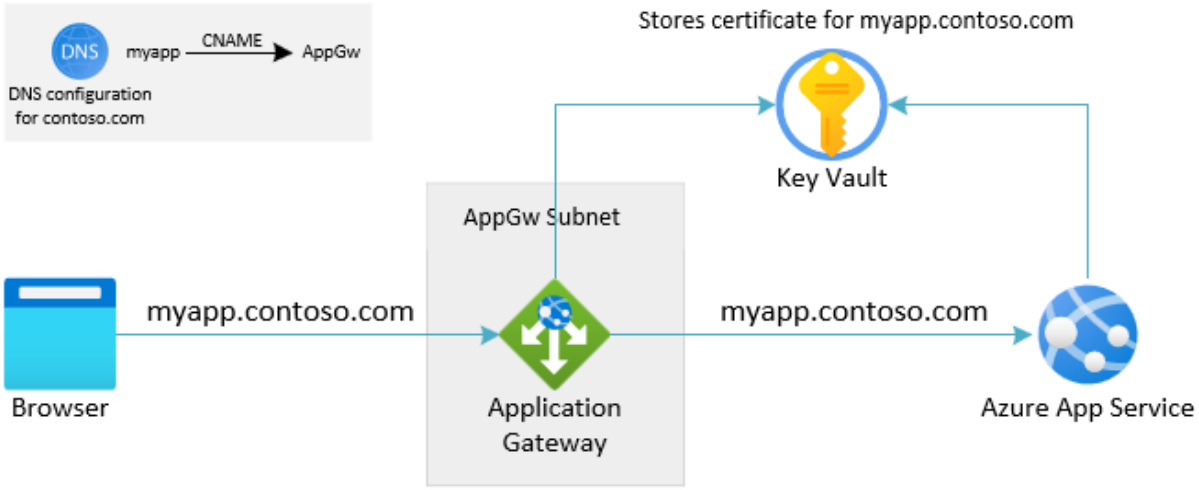
Type	Platform	Log storage location	Description
Application logging	Windows, Linux	App Service file system and/or Azure Storage blobs	Logs messages generated by your application code. The messages can be generated by the web framework you choose, or from your application code directly using the standard logging pattern of your language. Each message is assigned one of the following categories: **Critical**, **Error**, **Warning**, **Info**, **Debug**, and **Trace**. You can select how verbose you want the logging to be by setting the severity level when you enable application logging.
Web server logging	Windows	App Service file system or Azure Storage blobs	Raw HTTP request data in the W3C extended log file format. Each log message includes data such as the HTTP method, resource URI, client IP, client port, user agent, response code, and so on.
Detailed Error Messages	Windows	App Service file system	Copies of the *.htm* error pages that would have been sent to the client browser. For security reasons, detailed error pages shouldn't be sent to clients in production, but App Service can save the error page each time an application error occurs that has HTTP code 400 or higher. The page may contain information that can help determine why the server returns the error code.
Failed request tracing	Windows	App Service file system	Detailed tracing information on failed requests, including a trace of the IIS components used to process the request and the time taken in each component. This information is useful if you want to improve site performance or isolate a specific HTTP error. One folder is generated for each failed request. The folder contains the XML log file and the XSL stylesheet to view the log file with.
Deployment logging	Windows, Linux	App Service file system	Logs for when you publish content to an app. Deployment logging happens automatically and there are no configurable settings for deployment logging. It helps you determine why a deployment failed. For example, if you use a custom deployment script, you might use deployment logging to determine why the script is failing.

When stored in the App Service file system, logs are subject to the available storage for your pricing tier (see App Service limits).

5.1 Configure App Service with Application Gateway

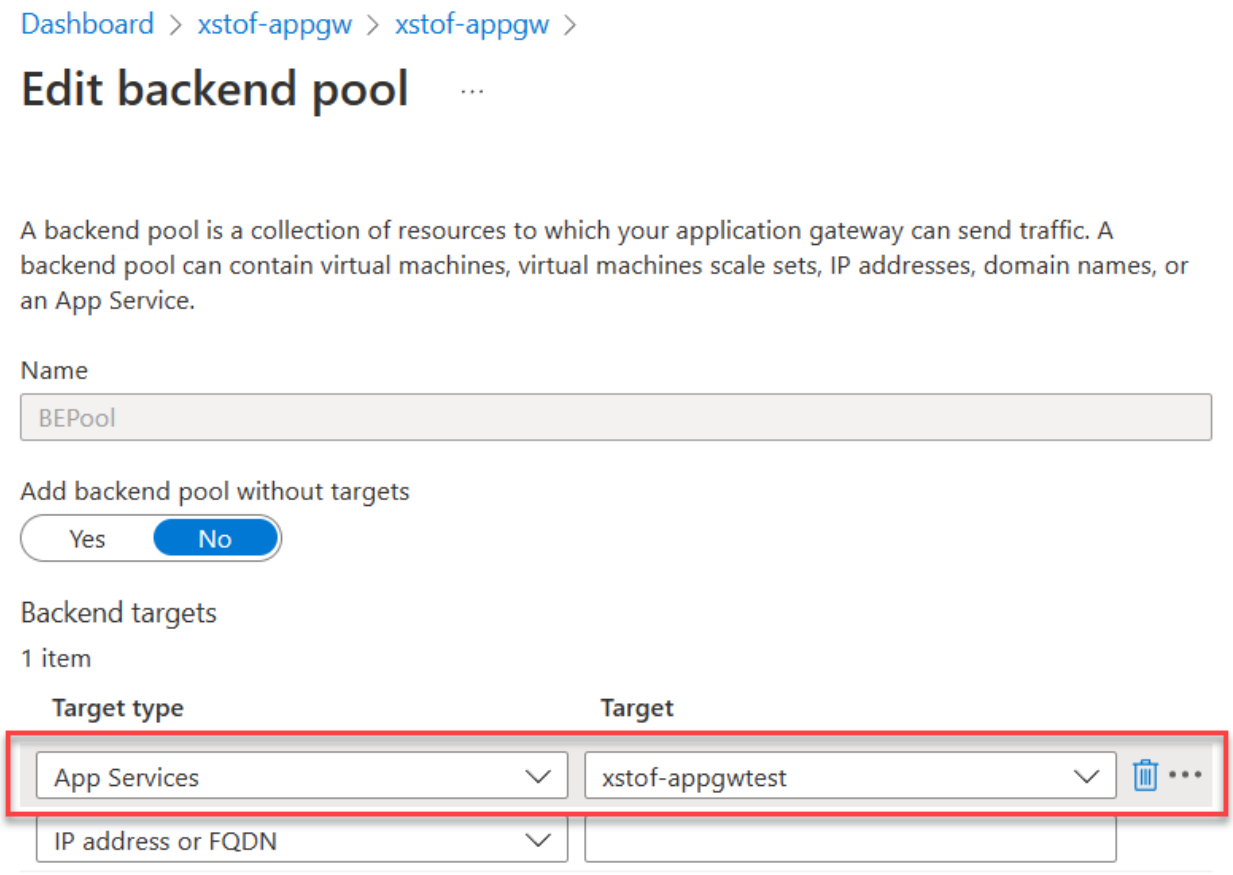
Application gateway allows you to have an App Service app or other multi-tenant service as a backend pool member. In this article, you learn to configure an App Service app with Application Gateway. The configuration for Application Gateway will differ depending on how App Service will be accessed:

- The first option makes use of a **custom domain** on both Application Gateway and the App Service in the backend.
- The second option is to have Application Gateway access App Service using its **default domain**, suffixed as ".azurewebsites.net".



- Configure DNS
  The DNS name, which the user or client is using towards Application Gateway and what is shown in a browser
  The DNS name, which Application Gateway is internally using to access the App Service in the backend

- Add App Service as backend pool to the Application Gateway



- Configure HTTP Settings for the connection to App Service

Add HTTP setting

HTTP settings name

http-setting-to-app-service

Backend protocol

HTTP HTTPS

Backend port *

443

Trusted root certificate

For end-to-end SSL encryption, the backends must be in the allowlist of the application gateway. Upload the public certificate of the backend servers to this HTTP setting.

Use well known CA certificate

Yes No

Additional settings

Cookie-based affinity

Enable Disable

Connection draining

Enable Disable

Request time-out (seconds) *

20

Override backend path

Host name

By default, Application Gateway does not change the incoming HTTP host header from the client and sends the header unaltered to the backend. Multi-tenant services like App service or API management rely on a specific host header or SNI extension to resolve to the correct endpoint. Change these settings to overwrite the incoming HTTP host header.

Override with new host name

Yes No

Host name override

Pick host name from backend target

Override with specific domain name

e.g. contoso.com

Use custom probe

Yes No

Custom probe *

probe-for-appvs-backend-https

- Configure an HTTP Listener

Add listener

xstof-appgw

Listener name * ⓘ

public-https-listener

Frontend IP * ⓘ

Public

Port * ⓘ

443

Protocol ⓘ

○ HTTP ◉ HTTPS

Choose a certificate

◉ Create new ○ Select existing

Https Settings

Choose a certificate

○ Upload a certificate ◉ Choose a certificate from Key Vault

Cert name *

my-domain-wildcard-cert

Managed identity * ⓘ

xstof-user-assigned-mi

Key vault * ⓘ

xstof-domain-kv

Certificate *

wildcard-xstof-net-with-intermediaries

☐ Enable SSL Profile ⓘ

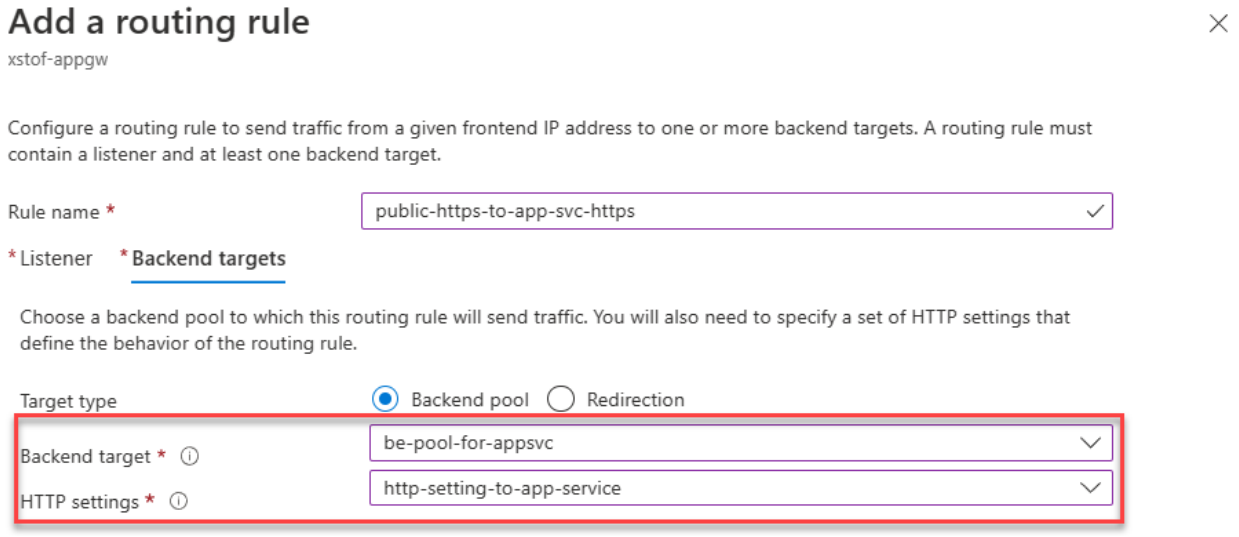
Additional settings

Listener type ⓘ

◉ Basic ○ Multi site

Add Cancel

- Configure a Request Routing Rule



5.2 Azure App Service and Azure Functions

5.3 Work with user identities in Azure App Service authentication

Access user claims in app code

For all language frameworks, App Service makes the claims in the incoming token (whether from an authenticated end user or a client application) available to your code by injecting them into the request headers. External requests aren't allowed to set these headers, so they're present only if set by App Service. Some example headers include:

Header	Description
X-MS-CLIENT-PRINCIPAL	A Base64 encoded JSON *representation of available claims*.
X-MS-CLIENT-PRINCIPAL-ID	An identifier for the caller *set by the identity provider*.
X-MS-CLIENT-PRINCIPAL-NAME	A human-readable name for the caller *set by the identity provider*, such as email address or user principal name.
X-MS-CLIENT-PRINCIPAL-IDP	The *name of the identity provider* used by App Service Authentication.

Provider tokens are also exposed through similar headers. For example, Microsoft Entra also sets X-MS-TOKEN-AAD-ACCESS-TOKEN and X-MS-TOKEN-AAD-ID-TOKEN as appropriate.

X-MS-CLIENT-PRINCIPAL contains the full set of available claims as Base64 encoded JSON. These claims go through a default claims-mapping process, so some might have different names than you would see if processing the token directly.

Access user claims using the API

If the token store is enabled for your app, you can also obtain other details on the authenticated user by calling /.auth/me.

6.1 Authentication and authorization in Azure App Service and Azure Functions

Considerations for using built-in authentication

Enabling this feature will cause all requests to your application to be automatically redirected to HTTPS, regardless of the App Service configuration setting to enforce HTTPS. You can disable this with the requireHttps setting in the V2 configuration. However, we do recommend sticking with HTTPS, and you should ensure no security tokens ever get transmitted over non-secure HTTP connections.

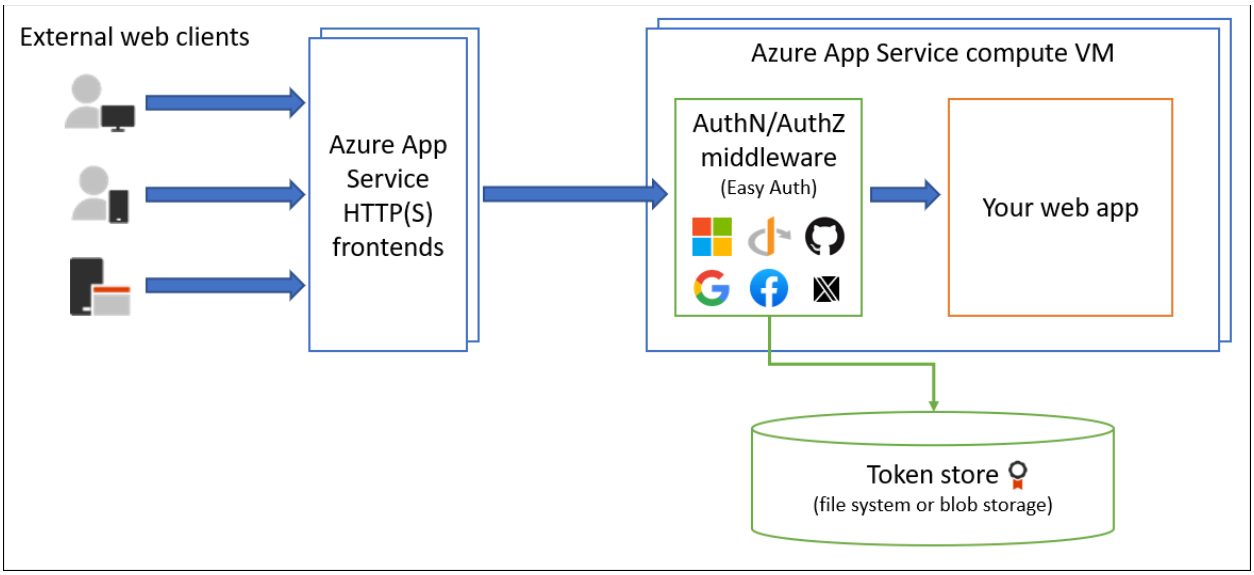
App Service can be used for authentication with or without restricting access to your site content and APIs.

Access restrictions can be set in the Authentication > Authentication settings section of your web app.

To restrict app access only to authenticated users, set Action to take when request is not authenticated to log in with one of the configured identity providers.
To authenticate but not restrict access, set Action to take when request is not authenticated to "Allow anonymous requests (no action)."

Feature architecture

The authentication and authorization middleware component is a feature of the platform that runs on the same VM as your application. When it's enabled, every incoming HTTP request passes through it before being handled by your application.



The platform middleware handles several things for your app:

- Authenticates users and clients with the specified identity provider(s)
- Validates, stores, and refreshes OAuth tokens issued by the configured identity provider(s)
- Manages the authenticated session
- Injects identity information into HTTP request headers

Authentication flow

The authentication flow is the same for all providers, but differs depending on whether you want to sign in with the provider's SDK:

- Without provider SDK:
  The application delegates federated sign-in to App Service. This is typically the case with browser apps, which can present the provider's login page to the user.

  The server code manages the sign-in process, so it is also called *server-directed flow* or *server flow*. This case applies to browser apps and mobile apps that use an embedded browser for authentication.

- With provider SDK:
  The application signs users in to the provider manually and then submits the authentication token to App Service for validation. This is typically the case with browser-less apps, which can't present the provider's sign-in page to the user.

  The application code manages the sign-in process, so it is also called *client-directed flow* or *client flow*. This case applies to REST APIs, Azure Functions, and JavaScript browser clients, as well as browser apps that need more flexibility in the sign-in process. It also applies to native mobile apps that sign users in using the provider's SDK.

Calls from a trusted browser app in App Service to another REST API in App Service or Azure Functions can be authenticated using the server-directed flow.

Set up staging environments in Azure App Service

Deploying your application to a nonproduction slot has the following benefits:

- You can validate app changes in a staging deployment slot before swapping it with the production slot.
- Deploying an app to a slot first and swapping it into production makes sure that all instances of the slot are warmed up before being swapped into production. This eliminates downtime when you deploy your app. The traffic redirection is seamless, and no requests are dropped because of swap operations. You can automate this entire workflow by configuring auto swap when pre-swap validation isn't needed.
- After a swap, the slot with previously staged app now has the previous production app. If the changes swapped into the production slot aren't as you expect, you can perform the same swap immediately to get your "last known good site" back.

Swap operation steps

When you swap two slots (usually from a staging slot *as the source* into the production slot *as the target*), App Service does the following to ensure that the target slot doesn't experience downtime:

1. Apply the following settings from the target slot (for example, the production slot) to all instances of the source slot:
    - Slot-specific app settings and connection strings, if applicable.
    - Continuous deployment settings, if enabled.
    - App Service authentication settings, if enabled.

When any of the settings is applied to the source slot, the change triggers all instances in the source slot to restart. During swap with preview, this marks the end of the first phase.

The swap operation is paused, and you can validate that the source slot works correctly with the target slot's settings.

2. Wait for every instance in the source slot to complete its restart. If any instance fails to restart, the swap operation reverts all changes to the source slot and stops the operation.

3. If local cache is enabled, trigger local cache initialization by making an HTTP request to the application root ("/") on each instance of the source slot. Wait until each instance returns any HTTP response. Local cache initialization causes another restart on each instance.

4. If auto swap is enabled with custom warm-up, trigger the custom Application Initiation on each instance of the source slot.

If applicationInitialization isn't specified, trigger an HTTP request to the application root of the source slot on each instance.
If an instance returns any HTTP response, it's considered to be warmed up.

5. If all instances on the source slot are warmed up successfully, swap the two slots by switching the routing rules for the two slots. After this step, the target slot (for example, the production slot) has the app that's previously warmed up in the source slot.

6. Now that the source slot has the pre-swap app previously in the target slot, perform the same operation by applying all settings and restarting the instances.

Specify custom warm-up
Some apps might require custom warm-up actions before the swap. The applicationInitialization configuration element in web.config lets you specify custom initialization actions.

The swap operation waits for this custom warm-up to finish before swapping with the target slot. Here's a sample web.config fragment.

```
<system.webServer>
  <applicationInitialization>
    <add initializationPage="/" hostName="[app hostname]" />
    <add initializationPage="/Home/About" hostName="[app hostname]" />
  </applicationInitialization>
</system.webServer>
```

For more information on customizing the applicationInitialization element, see Most common deployment slot swap failures and how to fix them.

You can also customize the warm-up behavior with one or both of the following app settings:

- WEBSITE_SWAP_WARMUP_PING_PATH: The path to ping over HTTP to warm up your site. Add this app setting by specifying a custom path that begins with a slash as the value. An example is /statuscheck. The default value is /.

- WEBSITE_SWAP_WARMUP_PING_STATUSES: Valid HTTP response codes for the warm-up operation. Add this app setting with a comma-separated list of HTTP codes. An example is 200,202 . If the returned status code isn't in the list, the warmup and swap operations are stopped. By default, all response codes are valid.

- WEBSITE_WARMUP_PATH: A relative path on the site that should be pinged whenever the site restarts (not only during slot swaps). Example values include /statuscheck or the root path, /.

Enable diagnostics logging for apps in Azure App Service

Azure provides built-in diagnostics to assist with debugging an App Service app. In this article, you learn how to enable diagnostic logging and add instrumentation to your application, as well as how to access the information logged by Azure.

Type	Platform	Log storage location	Description
Application logging	Windows, Linux	App Service file system and/or Azure Storage blobs	Logs messages generated by your application code. The messages can be generated by the web framework you choose, or from your application code directly using the standard logging pattern of your language. Each message is assigned one of the following categories: **Critical**, **Error**, **Warning**, **Info**, **Debug**, and **Trace**. You can select how verbose you want the logging to be by setting the severity level when you enable application logging.
Web server logging	Windows	App Service file system or Azure Storage blobs	Raw HTTP request data in the W3C extended log file format. Each log message includes data such as the HTTP method, resource URI, client IP, client port, user agent, response code, and so on.
Detailed Error Messages	Windows	App Service file system	Copies of the *.htm* error pages that would have been sent to the client browser. For security reasons, detailed error pages shouldn't be sent to clients in production, but App Service can save the error page each time an application error occurs that has HTTP code 400 or higher. The page may contain information that can help determine why the server returns the error code.
Failed request tracing	Windows	App Service file system	Detailed tracing information on failed requests, including a trace of the IIS components used to process the request and the time taken in each component. This information is useful if you want to improve site performance or isolate a specific HTTP error. One folder is generated for each failed request. The folder contains the XML log file and the XSL stylesheet to view the log file with.
Deployment logging	Windows, Linux	App Service file system	Logs for when you publish content to an app. Deployment logging happens automatically and there are no configurable settings for deployment logging. It helps you determine why a deployment failed. For example, if you use a custom deployment script, you might use deployment logging to determine why the script is failing.

Enable application logging (Windows)

Select the **Level**, or the level of details to log. The following table shows the log categories included in each level:

Level	Included categories
Disabled	None
Error	Error, Critical
Warning	Warning, Error, Critical
Information	Info, Warning, Error, Critical
Verbose	Trace, Debug, Info, Warning, Error, Critical (all categories)

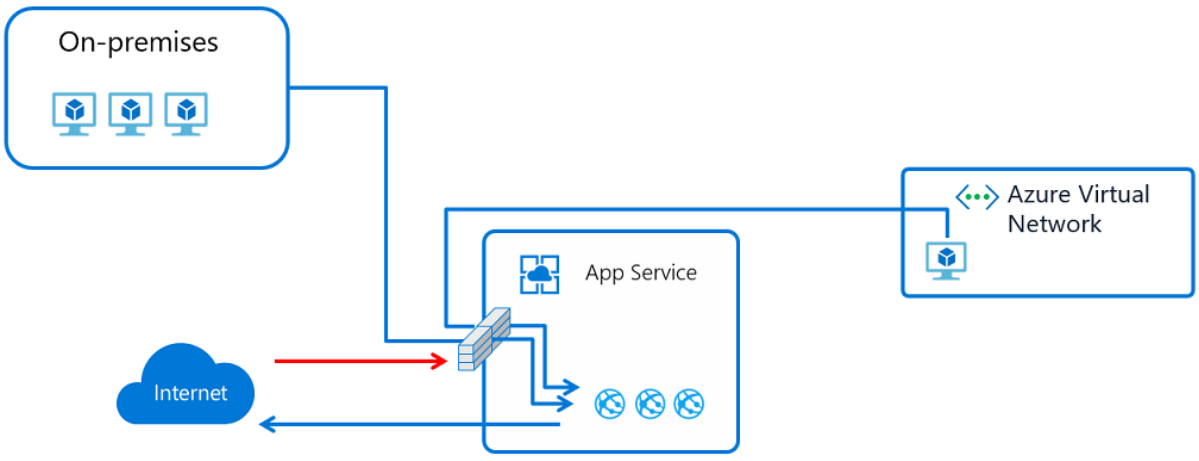
Enable application logging (Linux/Container)
In **Application logging**, select **File System**.
In **Quota (MB)**, specify the disk quota for the application logs. In **Retention Period (Days)**, set the number of days the logs should be retained.

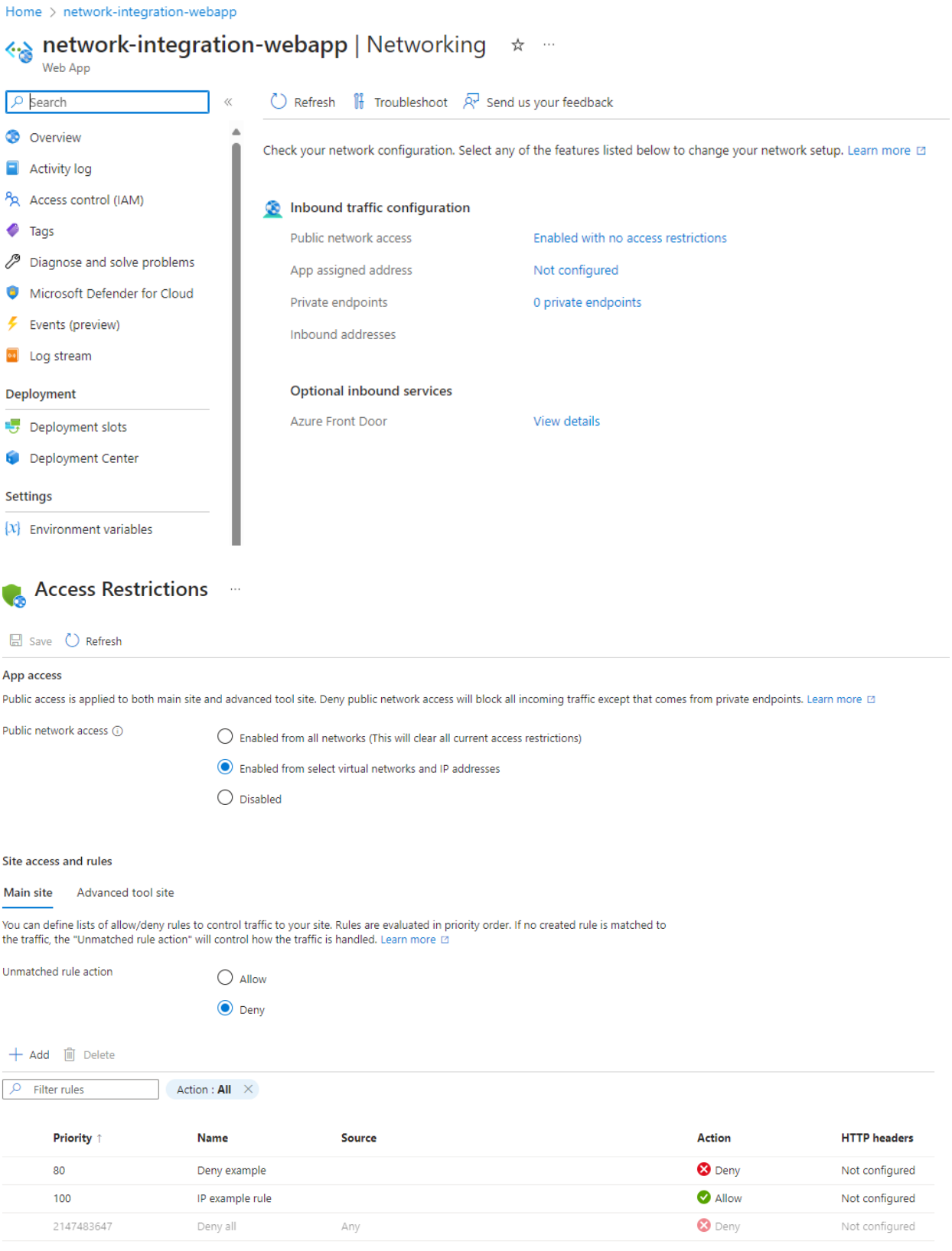
Log detailed errors
The Failed Request Tracing feature by default captures a log of requests that failed with HTTP status codes between 400 and 600. To specify custom rules, you can override the `<traceFailedRequests>` section in the *web.config* file.

Stream logs
Before you stream logs in real time, enable the log type that you want. Any information written to the console output or files ending in .txt, .log, or .htm that are stored in the */home/LogFiles* directory (D:\home\LogFiles) is streamed by App Service.

Set up Azure App Service access restrictions

By setting up access restrictions, you can define a priority-ordered allow/deny list that controls network access to your app




Home > network-integration-webapp
network-integration-webapp | Networking
Web App
Search
Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Microsoft Defender for Cloud
Events (preview)
Log stream
Deployment
Deployment slots
Deployment Center
Settings
Environment variables
Refresh
Troubleshoot
Send us your feedback
Check your network configuration. Select any of the features listed below to change your network setup. Learn more
Inbound traffic configuration
Public network access
Enabled with no access restrictions
App assigned address
Not configured
Private endpoints
0 private endpoints
Inbound addresses
Optional inbound services
Azure Front Door
View details
Access Restrictions
Save
Refresh
App access
Public access is applied to both main site and advanced tool site. Deny public network access will block all incoming traffic except that comes from private endpoints. Learn more
Public network access
Enabled from all networks (This will clear all current access restrictions)
Enabled from select virtual networks and IP addresses
Disabled
Site access and rules
Main site
Advanced tool site
You can define lists of allow/deny rules to control traffic to your site. Rules are evaluated in priority order. If no created rule is matched to the traffic, the "Unmatched rule action" will control how the traffic is handled. Learn more
Unmatched rule action
Allow
Deny
Add
Delete
Filter rules
Action : All
Priority
Name
Source
Action
HTTP headers
80
Deny example
Deny
Not configured
100
IP example rule
Allow
Not configured
2147483647
Deny all
Any
Deny
Not configured

Add an access restriction rule

To add an access restriction rule to your app, on the **Access Restrictions** page, select **Add**. The rule is only effective after saving.

Rules are enforced in priority order, starting from the lowest number in the **Priority** column. If you don't configure unmatched rule, an implicit *deny all* is in effect after you add even a single rule.

On the **Add Access Restriction** pane, when you create a rule, do the following:

1. Under **Action**, select either **Allow** or **Deny**.

2. Optionally, enter a name and description of the rule.
3. In the **Priority** box, enter a priority value.
4. In the **Type** drop-down list, select the type of rule. The different types of rules are described in the following sections.
5. Select **Add rule** after typing in the rule specific input to add the rule to the list.

IP-based access restriction rules only handle virtual network address ranges when your app is in an App Service Environment. If your app is in the multi-tenant service, you need to use **service endpoints** to restrict traffic to select subnets in your virtual network.

Add Access Restriction

Name

Enter name for the IpAddress rule

Action

Allow | Deny

Priority *

Ex. 300

Description

Type

Virtual Network

Subscription *

Purple Demo Subscription

Virtual Network *

networking-demos-vnet

Subnet *

nat-gw-subnet

Selected subnet 'networking-demos-vnet/nat-gw-subnet' does not have service endpoint enabled for Microsoft.Web. Enabling access may take up to 15 minutes to complete.

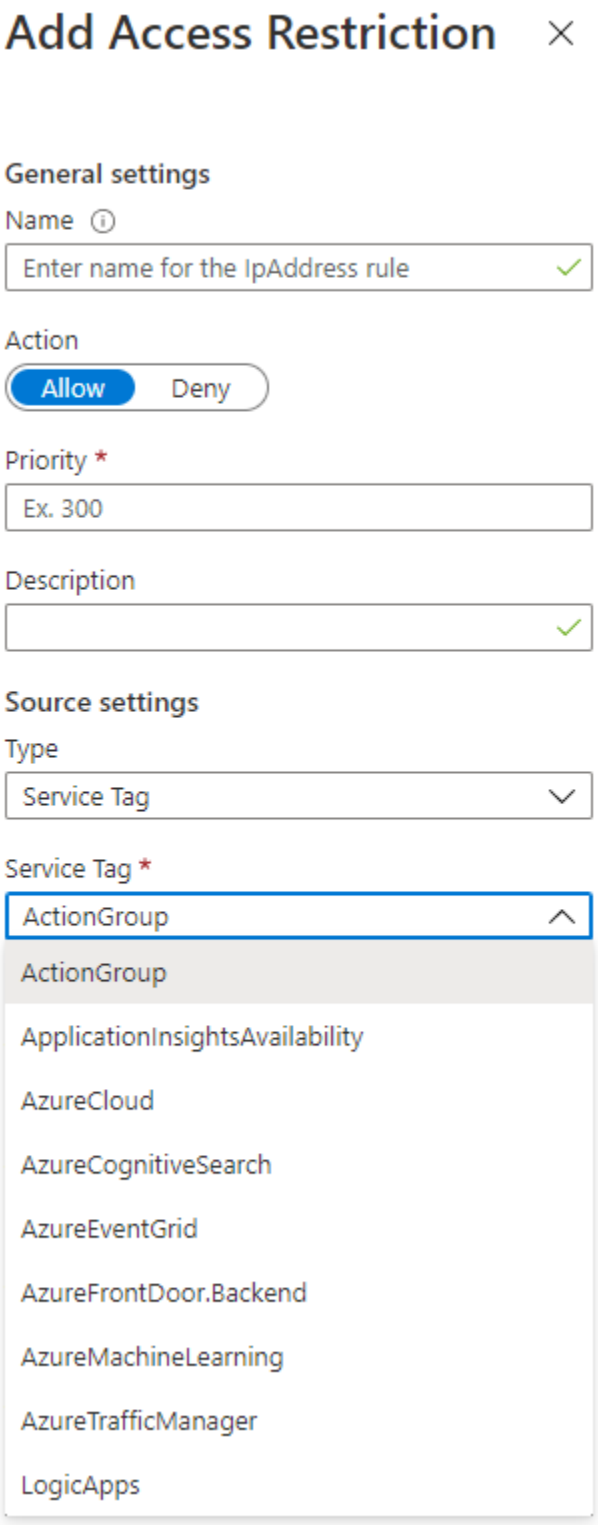
Ignore missing Microsoft.Web service endpoints

Add rule

By using service endpoints, you can restrict access to selected Azure virtual network subnets.

Set a service tag-based rule

- For step 4, in the **Type** drop-down list, select **Service Tag**.



All publicly available service tags are supported in access restriction rules. Each service tag represents a list of IP ranges from Azure services.
Use Azure Resource Manager templates or scripting to configure more advanced rules like regional scoped rules.

Filter by http header
As part of any rule, you can add http header filters. The following http header names are supported:

- X-Forwarded-For
- X-Forwarded-Host
- X-Azure-FDID
- X-FD-HealthProbe

For each header name, you can add up to eight values separated by comma. The http header filters are evaluated after the rule itself and both conditions must be true for the rule to apply.

Tutorial: Authenticate and authorize users end-to-end in Azure App Service

1. Clone the sample application

1. In the Azure Cloud Shell, run the following command to clone the sample repository.

Azure CLICopy
Open Cloud Shell
git clone https://github.com/Azure-Samples/js-e2e-web-app-easy-auth-app-to-app

2. Create and deploy apps
Create the resource group, web app plan, the web app and deploy in a single step.

1. Change into the frontend web app directory.

Azure CLICopy
Open Cloud Shell
cd frontend

2. Create and deploy the frontend web app with az webapp up. Because web app name has to be globally unique, replace <front-end-app-name> with a unique set of initials or numbers.

Azure CLICopy
Open Cloud Shell
az webapp up --resource-group myAuthResourceGroup --name <front-end-app-name> --plan myPlan --sku FREE --location "West Europe" --os-type Linux --runtime "NODE:16-lts"

3. Change into the backend web app directory.

Azure CLICopy
Open Cloud Shell
cd ../backend

4. Deploy the backend web app to same resource group and app plan. Because web app name has to be globally unique, replace <back-end-app-name> with a unique set of initials or numbers.

Azure CLICopy
Open Cloud Shell
az webapp up --resource-group myAuthResourceGroup --name <back-end-app-name> --plan myPlan --sku FREE --location "West Europe" --runtime "NODE:16-lts"

3. Configure app setting

The frontend application needs to the know the URL of the backend application for API requests. Use the following Azure CLI command to configure the app setting. The URL should be in the format of https://<back-end-app-name>.azurewebsites.net.

Azure CLICopy

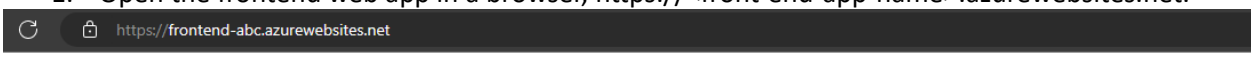
Open Cloud Shell

az webapp config appsettings set --resource-group myAuthResourceGroup --name <front-end-app-name> --settings BACKEND_URL="https://<back-end-app-name>.azurewebsites.net"

4. Frontend calls the backend

Browse to the frontend app and return the *fake* profile from the backend. This action validates that the frontend is successfully requesting the profile from the backend, and the backend is returning the profile.

1. Open the frontend web app in a browser, https://<front-end-app-name>.azurewebsites.net.



Easy auth - Get Profile from API server - 1

Get user's profile from server API

2. Select the Get user's profile link.
3. View the *fake* profile returned from the backend web app.

Success


```
{
    "displayName": "John Doe",
    "withAuthentication": false
}
```


The withAuthentication value of **false** indicates the authentication *isn't* set up yet.

5. Configure authentication

In this step, you enable authentication and authorization for the two web apps. This tutorial uses Microsoft Entra ID as the identity provider.

You also configure the frontend app to:

- Grant the frontend app access to the backend app
- Configure App Service to return a usable token
- Use the token in your code.

For more information, see Configure Microsoft Entra authentication for your App Services application.

Enable authentication and authorization for backend app

1. In the Azure portal menu, select **Resource groups** or search for and select *Resource groups* from any page.
2. In **Resource groups**, find and select your resource group. In **Overview**, select your backend app.
3. In your backend app's left menu, select **Authentication**, and then select **Add identity provider**.
4. In the **Add an identity provider** page, select **Microsoft** as the **Identity provider** to sign in Microsoft and Microsoft Entra identities.
5. Accept the default settings and select **Add**.

Add an identity provider

Basics | Permissions

Identity provider *: Microsoft

Choose a tenant for your application and its users

A tenant contains applications and a directory for user accounts. Choose a tenant based on whether it's configured for workforce users (employees and business guests) or external users. Learn more

- (•) Workforce configuration (current tenant)
  Manage employees and business guests
- ( ) External configuration
  Manage external users

App registration

An app registration associates your identity provider with your app. Enter the app registration information here, or go to your provider to create a new one. Learn more

App registration type *
- (•) Create new app registration
- ( ) Pick an existing app registration in this directory
- ( ) Provide the details of an existing app registration

Name * : dotnet-core-back-end-auth

Supported account types *
- (•) Current tenant - Single tenant
- ( ) Any Microsoft Entra directory - Multi-tenant
- ( ) Any Microsoft Entra directory & personal Microsoft accounts
- ( ) Personal Microsoft accounts only

Help me choose...

Additional checks

You can configure additional checks that will further control access, but your app may still need to make additional authorization decisions in code. Learn more

Client application requirement *
- (•) Allow requests only from this application itself
- ( ) Allow requests from specific client applications
- ( ) Allow requests from any application (Not recommended)

Identity requirement *
- (•) Allow requests from any identity
- ( ) Allow requests from specific identities

Tenant requirement *
- (•) Allow requests only from the issuer tenant
- ( ) Allow requests from specific tenants
- ( ) Use default restrictions based on issuer

App Service authentication settings

Requiring authentication ensures that requests to your app include information about the caller, but your app may still need to make additional authorization decisions to control access. If unauthenticated requests are allowed, any client can call the app and your code will need to handle both authentication and authorization. Learn more

Restrict access *
- (•) Require authentication
- ( ) Allow unauthenticated access

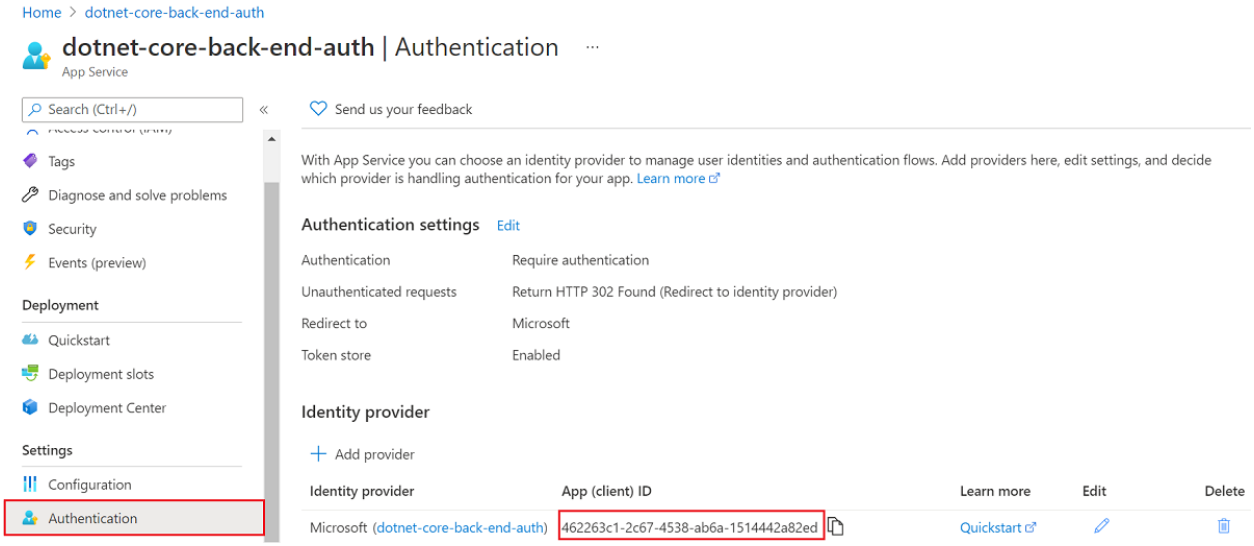
Unauthenticated requests *
- (•) HTTP 302 Found redirect: recommended for websites
- ( ) HTTP 401 Unauthorized: recommended for APIs
- ( ) HTTP 403 Forbidden
- ( ) HTTP 404 Not found

Redirect to: Microsoft

Token store: [✓]

Add | < Previous | Next: Permissions >

6. The **Authentication** page opens. Copy the **Client ID** of the Microsoft Entra application to a notepad. You need this value later.



If you stop here, you have a self-contained app that's already secured by the App Service authentication and authorization. The remaining sections show you how to secure a multi-app solution by "flowing" the authenticated user from the frontend to the backend.

Enable authentication and authorization for frontend app

1. In the Azure portal menu, select **Resource groups** or search for and select *Resource groups* from any page.
2. In **Resource groups**, find and select your resource group. In **Overview**, select your frontend app's management page.
3. In your frontend app's left menu, select **Authentication**, and then select **Add identity provider**.
4. In the **Add an identity provider** page, select **Microsoft** as the **Identity provider** to sign in Microsoft and Microsoft Entra identities.
5. Accept the default settings and select **Add**.
6. The **Authentication** page opens. Copy the **Client ID** of the Microsoft Entra application to a notepad. You need this value later.

Grant frontend app access to backend

Now that you've enabled authentication and authorization to both of your apps, each of them is backed by an AD application. To complete the authentication, you need to do three things:

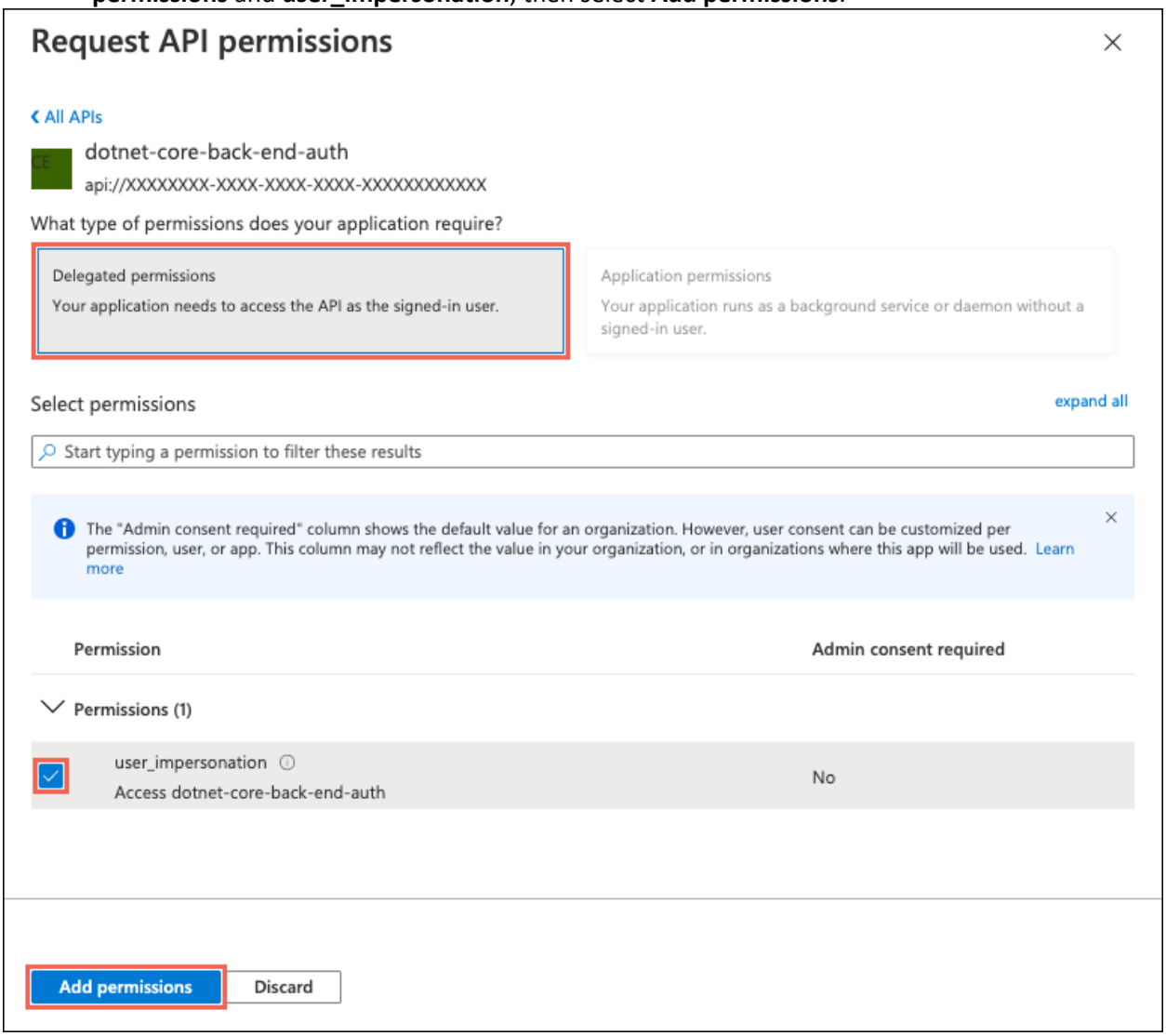
- Grant the frontend app access to the backend app
- Configure App Service to return a usable token
- Use the token in your code.

Tip

If you run into errors and reconfigure your app's authentication/authorization settings, the tokens in the token store may not be regenerated from the new settings. To make sure your tokens are regenerated, you need to sign out and sign back in to your app. An easy way to do it is to use your browser in private mode, and close and reopen the browser in private mode after changing the settings in your apps.

In this step, you **grant the frontend app access to the backend app** on the user's behalf. (Technically, you give the frontend's *AD application* the permissions to access the backend's *AD application* on the user's behalf.)

1. In the **Authentication** page for the frontend app, select your frontend app name under **Identity provider**. This app registration was automatically generated for you. Select **API permissions** in the left menu.
2. Select **Add a permission**, then select **My APIs** > **<back-end-app-name>**.
3. In the **Request API permissions** page for the backend app, select **Delegated permissions** and **user_impersonation**, then select **Add permissions**.



Configure App Service to return a usable access token

The frontend app now has the required permissions to access the backend app as the signed-in user. In this step, you configure App Service authentication and authorization to give you a usable access token for accessing the backend. For this step, you need the backend's client ID, which you copied from Enable authentication and authorization for backend app.

In the Cloud Shell, run the following commands on the frontend app to add the scope parameter to the authentication setting identityProviders.azureActiveDirectory.login.loginParameters. Replace *<front-end-app-name>* and *<back-end-client-id>*.

Azure CLICopy

Open Cloud Shell

az extension add --name authV2

```
authSettings=$(az webapp auth show -g myAuthResourceGroup -n <front-end-app-name>)
authSettings=$(echo "$authSettings" | jq '.properties' | jq '.identityProviders.azureActiveDirectory.login += {"loginParameters":["scope=openid offline_access api://<back-end-client-id>/user_impersonation"]}')
az webapp auth set --resource-group myAuthResourceGroup --name <front-end-app-name> --body "$authSettings"
```

The commands effectively add a loginParameters property with additional custom scopes. Here's an explanation of the requested scopes:

- openid is requested by App Service by default already. For information, see OpenID Connect Scopes.
- offline_access is included here for convenience (in case you want to refresh tokens).
- api://<back-end-client-id>/user_impersonation is an exposed API in your backend app registration. It's the scope that gives you a JWT token that includes the backend app as a token audience.

Tip

- To view the api://<back-end-client-id>/user_impersonation scope in the Azure portal, go to the **Authentication** page for the backend app, click the link under **Identity provider**, then click **Expose an API** in the left menu.
- To configure the required scopes using a web interface instead, see the Microsoft steps at **Refresh auth tokens**.
- Some scopes require admin or user consent. This requirement causes the consent request page to be displayed when a user signs into the frontend app in the browser. To avoid this consent page, add the frontend's app registration as an authorized client application in the **Expose an API** page by clicking **Add a client application** and supplying the client ID of the frontend's app registration.

Your apps are now configured. The frontend is now ready to access the backend with a proper access token.

For information on how to configure the access token for other providers, see Refresh identity provider tokens.

6. Configure backend App Service to accept a token only from the frontend App Service

You should also configure the backend App Service to only accept a token from the frontend App Service. Not doing this may result in a "403: Forbidden error" when you pass the token from the frontend to the backend.

You can set this via the same Azure CLI process you used in the previous step.

1. Get the appId of the frontend App Service (you can get this on the "Authentication" blade of the frontend App Service).
2. Run the following Azure CLI, substituting the <back-end-app-name> and <front-end-app-id>.

Azure CLICopy

Open Cloud Shell

```
authSettings=$(az webapp auth show -g myAuthResourceGroup -n <back-end-app-name>)
authSettings=$(echo "$authSettings" | jq '.properties' | jq '.identityProviders.azureActiveDirectory.validation.defaultAuthorizationPolicy.allowedApplications += ["<front-end-app-id>"]')
az webapp auth set --resource-group myAuthResourceGroup --name <back-end-app-name> --body "$authSettings"

authSettings=$(az webapp auth show -g myAuthResourceGroup  -n <back-end-app-name>)
authSettings=$(echo "$authSettings" | jq '.properties' | jq '.identityProviders.azureActiveDirectory.validation.jwtClaimChecks += { "allowedClientApplications": ["<front-end-app-id>"]}')
```

az webapp auth set --resource-group myAuthResourceGroup --name <back-end-app-name> --body "$authSettings"

7. Frontend calls the authenticated backend

The frontend app needs to pass the user's authentication with the correct user_impersonation scope to the backend. The following steps review the code provided in the sample for this functionality.

View the frontend app's source code:

1. Use the frontend App Service injected x-ms-token-aad-access-token header to programmatically get the user's accessToken.

JavaScriptCopy

```
// ./src/server.js
const accessToken = req.headers['x-ms-token-aad-access-token'];
```

2. Use the accessToken in the Authentication header as the bearerToken value.

JavaScriptCopy

```
// ./src/remoteProfile.js
// Get profile from backend
const response = await fetch(remoteUrl, {
   cache: "no-store", // no caching -- for demo purposes only
   method: 'GET',
   headers: {
      'Authorization': `Bearer ${accessToken}`
   }
});
if (response.ok) {
   const { profile } = await response.json();
   console.log(`profile: ${profile}`);
} else {
   // error handling
}
```

This tutorial returns a *fake* profile to simplify the scenario. The next tutorial in this series demonstrates how to exchange the backend bearerToken for a new token with the scope of a downstream Azure service, such as Microsoft Graph.

7. Backend returns profile to frontend

If the request from the frontend isn't authorized, the backend App service rejects the request with a 401 HTTP error code *before* the request reaches your application code. When the backend code is reached (because it including an authorized token), extract the bearerToken to get the accessToken.

View the backend app's source code:

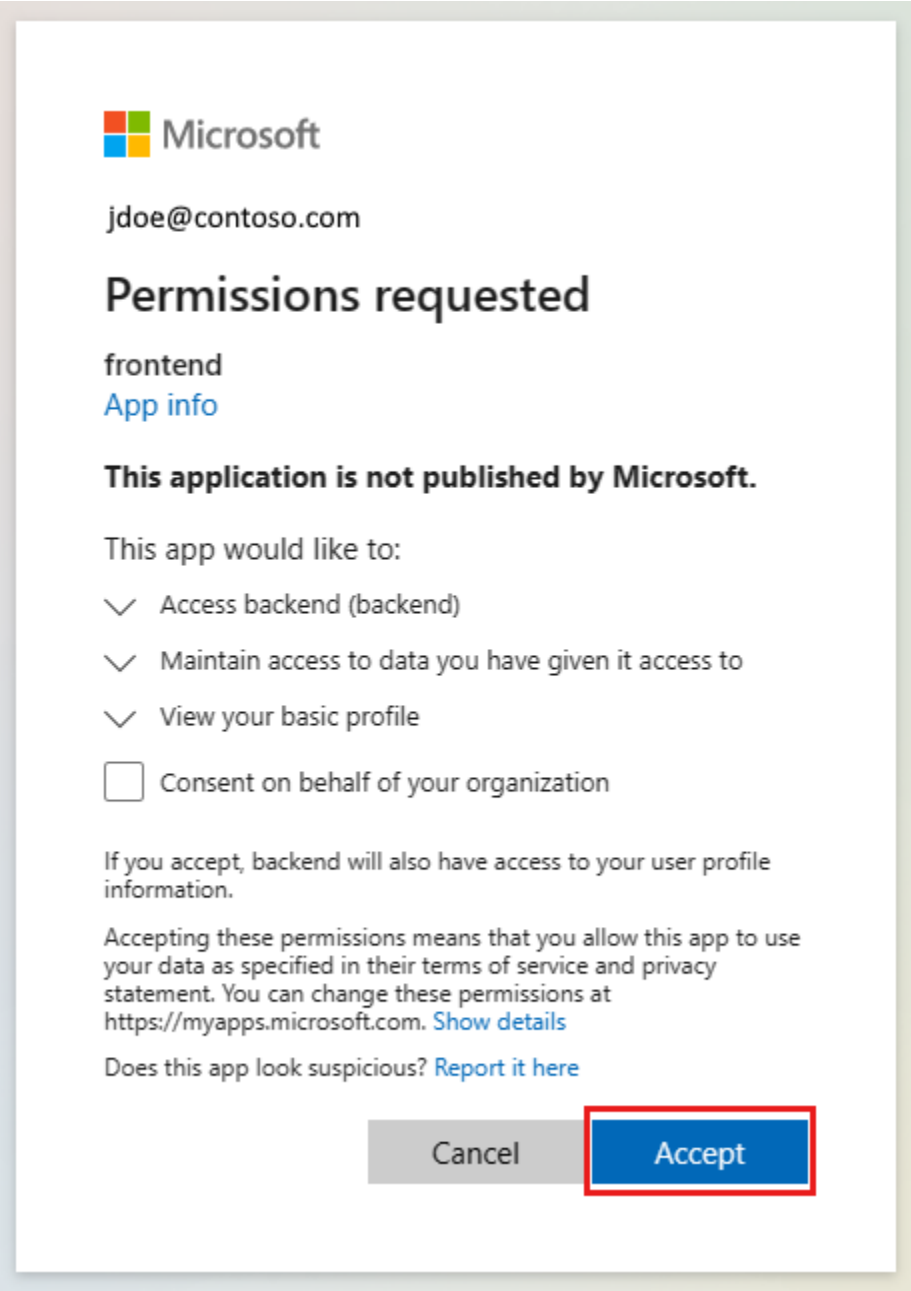
JavaScriptCopy

```
// ./src/server.js
const bearerToken = req.headers['Authorization'] || req.headers['authorization'];

if (bearerToken) {
   const accessToken = bearerToken.split(' ')[1];
   console.log(`backend server.js accessToken: ${!!accessToken ? 'found' : 'not found'}`);

   // TODO: get profile from Graph API
   // provided in next article in this series
   // return await getProfileFromMicrosoftGraph(accessToken)
```

```
  // return fake profile for this tutorial
  return {
    "displayName": "John Doe",
    "withAuthentication": !!accessToken ? true : false
  }
}
```

8. Browse to the apps

1. Use the frontend web site in a browser. The URL is in the format of https://<front-end-app-name>.azurewebsites.net/.
2. The browser requests your authentication to the web app. Complete the authentication.

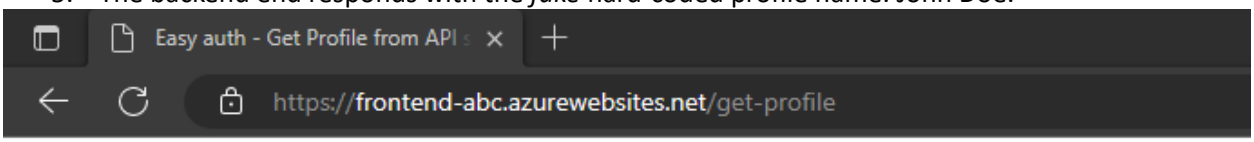


3. After authentication completes, the frontend application returns the home page of the app.

Easy auth - Get Profile from API server - 1

Get user's profile from server API

4. Select Get user's profile. This passes your authentication in the bearer token to the backend.
5. The backend end responds with the *fake* hard-coded profile name: John Doe.



API server - Profile

Remote url: https://backend-abc.azurewebsites.net/get-profile

Success


```
{
    "displayName": "John Doe",
    "withAuthentication": true
}
```


The withAuthentication value of **true** indicates the authentication *is* set up yet.