# Azure web app configuration

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.

Azure portal -> Your App -> Left Menu -> under settings -> Configuration

## Application settings:

-> Application Settings

For ASP.NET and ASP.NET Core, the values in App Service override the ones in Web.config or appsettings.json.

You can keep development settings (for example, local MySQL password) in Web.config or appsettings.json, but production secrets (for example, Azure MySQL database password) safe in App Service.

The same code uses your development settings when you debug locally, and it uses your production secrets when deployed to Azure.

App settings are always encrypted when stored (encrypted-at-rest).

## Configure connection strings

-> Application Settings -> Scroll down to see “Connection Strings”

For ASP.NET and ASP.NET Core developers the values you set in App Service override the ones in Web.config.

Connection strings are always encrypted when stored (encrypted-at-rest).

## General settings

-> General settings

### Stack settings:

The software stack to run the app, including the language and SDK versions. For Linux apps and custom container apps, you can also set an optional start-up command or file.

### Platform settings:

Lets you configure settings for the hosting platform, including:

* **Bitness**: 32-bit or 64-bit.
* **WebSocket protocol**: For ASP.NET SignalR or socket.io, for example.
* **Always On**: Keep the app loaded even when there's no traffic. By default, **Always On** is not enabled and the app is unloaded after 20 minutes without any incoming requests. It's required for continuous WebJobs or for WebJobs that are triggered using a CRON expression.
* **Managed pipeline version**: The IIS pipeline mode. Set it to **Classic** if you have a legacy app that requires an older version of IIS.
* **HTTP version**: Set to 2.0 to enable support for HTTPS/2 protocol.
* **ARR 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.

### 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. TLS mutual authentication is used to restrict access to your app by enabling different types of authentications for it.

## Configure path mappings

-> Path mappings

Path mappings section you can configure handler mappings, and virtual application and directory mappings.

The Path mappings page will display different options based on the OS type.

**Windows apps (uncontainerized):**

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, select **New handler**. 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. 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.

Each app has the default root path (/) mapped to D:\home\site\wwwroot, where your code is deployed by default. If your app root is in a different folder, or if your repository has more than one application, you can edit or add virtual applications and directories.

You can configure virtual applications and directories by specifying each virtual directory and its corresponding physical path relative to the website root (D:\home). To mark a virtual directory as a web application, clear the **Directory** check box.

**Linux and containerized apps:**

You can add custom storage for your containerized app. Containerized apps include all Linux apps and also the Windows and Linux custom containers running on App Service. Click **New Azure Storage Mount** and configure your custom storage as follows:

* **Name**: The display name.
* **Configuration options**: Basic or Advanced.
* **Storage accounts**: The storage account with the container you want.
* **Storage type**: **Azure Blobs** or **Azure Files**. Windows container apps only support Azure Files.
* **Storage container**: For basic configuration, the container you want.
* **Share name**: For advanced configuration, the file share name.
* **Access key**: For advanced configuration, the access key.
* **Mount path**: The absolute path in your container to mount the custom storage.

## Enable diagnostic logging

There are built-in diagnostics to assist with debugging an App Service app.

The table below shows the types of logging, the platforms supported, and where the logs can be stored and located for accessing the information.

| **Type** | **Platform** | **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**. |
| 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 like the HTTP method, resource URI, client IP, client port, user agent, response code, and so on. |
| Detailed error logging | Windows | App Service file system | Copies of the *.html* 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 greater. |
| 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. One folder is generated for each failed request, which contains the XML log file, and the XSL stylesheet to view the log file with. |
| Deployment logging | Windows, Linux | App Service file system | Helps determine why a deployment failed. Deployment logging happens automatically and there are no configurable settings for deployment logging. |

### Enable application logging (Windows)

Your app -> App Service logs

-> Select On for either Application Logging (Filesystem) or Application Logging (Blob), or both.

The Filesystem option is for temporary debugging purposes, and turns itself off in 12 hours. The Blob option is for long-term logging, and needs a blob storage container to write logs to.

You can also set the Level of details included in the log as shown in the table below.

| **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)

1. In **App Service logs** set the **Application logging** option to **File System**.
2. 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.
3. When finished, select **Save**.

### Enable web server logging

1. For **Web server logging**, select **Storage** to store logs on blob storage, or **File System** to store logs on the App Service file system.
2. In **Retention Period (Days)**, set the number of days the logs should be retained.
3. When finished, select **Save**.

### Add log messages in code

ASP.NET applications can use the **System.Diagnostics.Trace** class

**System.Diagnostics.Trace.TraceError("If you're seeing this, something bad happened");**

**By default, ASP.NET Core uses the Microsoft.Extensions.Logging.AzureAppServices logging provider.**

### Stream logs

* **Azure portal -** To stream logs in the Azure portal, navigate to your app and select **Log stream**.
* **Azure CLI** - To stream logs live in Cloud Shell, use the following command:

**az webapp log tail --name appname --resource-group myResourceGroup**

* Local console - To stream logs in the local console, install Azure CLI and sign into your account. Once signed in, follow the instructions for Azure CLI above.

### Access log files

* Linux/container apps: https://<app-name>.scm.azurewebsites.net/api/logs/docker/zip
* Windows apps: https://<app-name>.scm.azurewebsites.net/api/dump

## Configure security certificates

| **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 secure your custom domain in App Service. |
| Purchase 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 certificates. |
| Upload a private certificate | If you already have a private certificate from a third-party provider, you can upload it. |
| Upload a public certificate | Public certificates are not used to secure custom domains, but you can load them into your code if you need them to access remote resources. |

### Private certificate requirements

The free **App Service managed certificate** and the **App Service certificate** already satisfy the requirements of App Service. If you want to use a private certificate in App Service, your certificate must meet the following requirements:

* Exported as a password-protected PFX file, encrypted using triple DES.
* Contains private key at least 2048 bits long
* Contains all intermediate certificates in the certificate chain

To secure a custom domain in a TLS binding, the certificate has additional requirements:

* Contains an Extended Key Usage for server authentication (OID = 1.3.6.1.5.5.7.3.1)
* Signed by a trusted certificate authority

### Creating a free managed certificate

To create custom TLS/SSL bindings or enable client certificates for your App Service app, your App Service plan must be in the **Basic**, **Standard**, **Premium**, or **Isolated** tier. Custom SSL is not supported in the **F1** or **D1** tier.

The free App Service managed certificate is a turn-key solution for securing your custom DNS name in App Service. It's a TLS/SSL server certificate that's fully managed by App Service and renewed continuously and automatically in six-month increments, 45 days before expiration. You create the certificate and bind it to a custom domain, and let App Service do the rest.

The free certificate comes with the following limitations:

* Does not support wildcard certificates.
* Does not support usage as a client certificate by certificate thumbprint.
* Is not exportable.
* Is not supported on App Service Environment (ASE).
* Is not supported with root domains that are integrated with Traffic Manager.
* If a certificate is for a CNAME-mapped domain, the CNAME must be mapped directly to <app-name>.azurewebsites.net.

### Import an App Service Certificate

If you purchase an App Service Certificate from Azure, Azure manages the following tasks:

* Takes care of the purchase process from GoDaddy.
* Performs domain verification of the certificate.
* Maintains the certificate in Azure Key Vault.
* Manages certificate renewal.
* Synchronize the certificate automatically with the imported copies in App Service apps.

If you already have a working App Service certificate, you can:

* Import the certificate into App Service.
* Manage the certificate, such as renew, rekey, and export it.

### Upload a private certificate

If your certificate authority gives you multiple certificates in the certificate chain, you need to merge the certificates in order. Then you can Export your merged TLS/SSL certificate with the private key that your certificate request was generated with.

If you generated your certificate request using OpenSSL, then you have created a private key file. To export your certificate to PFX, run the following command. Replace the placeholders <private-key-file> and <merged-certificate-file> with the paths to your private key and your merged certificate file.

**openssl pkcs12 -export -out myserver.pfx -inkey <private-key-file> -in <merged-certificate-file>**

When prompted, define an export password. You'll use this password when uploading your TLS/SSL certificate to App Service.

### Enforce HTTPS

By default, anyone can still access your app using HTTP. You can redirect all HTTP requests to the HTTPS port.

Your app -> Left menu -> **TLS/SSL settings** -> **HTTPS Only** -> **On**.

## Manage app features

Feature management is a modern software-development practice that decouples feature release from code deployment and enables quick changes to feature availability on demand.

It uses a technique called feature flags (also known as feature toggles, feature switches, and so on) to dynamically administer a feature's lifecycle.

### Basic concepts

Here are several new terms related to feature management:

* **Feature flag**: A feature flag is a variable with a binary state of *on* or *off*. The feature flag also has an associated code block. The state of the feature flag triggers whether the code block runs or not.
* **Feature manager**: A feature manager is an application package that handles the lifecycle of all the feature flags in an application. The feature manager typically provides additional functionality, such as caching feature flags and updating their states.
* **Filter**: A filter is a rule for evaluating the state of a feature flag. A user group, a device or browser type, a geographic location, and a time window are all examples of what a filter can represent.

An effective implementation of feature management consists of at least two components working in concert:

* An application that makes use of feature flags.
* A separate repository that stores the feature flags and their current states.

How these components interact is illustrated in the following examples.

**Feature flag usage in code**

The basic pattern for implementing feature flags in an application is simple. You can think of a feature flag as a Boolean state variable used with an if conditional statement in your code:

**C#Copy**

if (featureFlag) {

// Run the following code

}

### Feature flag declaration

Each feature flag has two parts: a name and a list of one or more filters that are used to evaluate if a feature's state is *on* (that is, when its value is True). A filter defines a use case for when a feature should be turned on.

When a feature flag has multiple filters, the filter list is traversed in order until one of the filters determines the feature should be enabled. At that point, the feature flag is *on*, and any remaining filter results are skipped. If no filter indicates the feature should be enabled, the feature flag is *off*.

The feature manager supports *appsettings.json* as a configuration source for feature flags. The following example shows how to set up feature flags in a JSON file:

"FeatureManagement": {

"FeatureA": true, // Feature flag set to on

"FeatureB": false, // Feature flag set to off

"FeatureC": {

"EnabledFor": [

{

"Name": "Percentage",

"Parameters": {

"Value": 50

}

}

]

}

}

### Feature flag repository

To use feature flags effectively, you need to externalize all the feature flags used in an application. This approach allows you to change feature flag states without modifying and redeploying the application itself.

Azure App Configuration is designed to be a centralized repository for feature flags. You can use it to define different kinds of feature flags and manipulate their states quickly and confidently. You can then use the App Configuration libraries for various programming language frameworks to easily access these feature flags from your application.