100 XP

Introduction

3 minutes

Azure App Configuration provides a service to centrally manage application settings and feature flags.

After completing this module, you'll be able to:

- Explain the benefits of using Azure App Configuration
- Describe how Azure App Configuration stores information
- Implement feature management
- Securely access your app configuration information

Next unit: Explore the Azure App Configuration service

Continue

How are we doing? ☆ ☆ ☆ ☆ ☆

Explore the Azure App Configuration service

3 minutes

Modern programs, especially programs running in a cloud, generally have many components that are distributed in nature. Spreading configuration settings across these components can lead to hard-to-troubleshoot errors during an application deployment. Use App Configuration to store all the settings for your application and secure their access in one place.

App Configuration offers the following benefits:

- A fully managed service that can be set up in minutes
- Flexible key representations and mappings
- Tagging with labels
- Point-in-time replay of settings
- Dedicated UI for feature flag management
- Comparison of two sets of configurations on custom-defined dimensions
- Enhanced security through Azure-managed identities
- Complete data encryptions, at rest or in transit
- Native integration with popular frameworks

App Configuration complements Azure Key Vault, which is used to store application secrets. App Configuration makes it easier to implement the following scenarios:

- Centralize management and distribution of hierarchical configuration data for different environments and geographies
- Dynamically change application settings without the need to redeploy or restart an application
- Control feature availability in real-time

Use App Configuration

The easiest way to add an App Configuration store to your application is through a client library that Microsoft provides. Based on the programming language and framework, the following best methods are available to you.

Programming language and framework	How to connect
.NET Core and ASP.NET Core	App Configuration provider for .NET Core
.NET Framework and ASP.NET	App Configuration builder for .NET
Java Spring	App Configuration client for Spring Cloud
Others	App Configuration REST API

Next unit: Create paired keys and values

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Create paired keys and values

3 minutes

Azure App Configuration stores configuration data as key-value pairs.

Keys

Keys serve as the name for key-value pairs and are used to store and retrieve corresponding values. It's a common practice to organize keys into a hierarchical namespace by using a character delimiter, such as `/` or `:`. Use a convention that's best suited for your application. App Configuration treats keys as a whole. It doesn't parse keys to figure out how their names are structured or enforce any rule on them.

Keys stored in App Configuration are case-sensitive, unicode-based strings. The keys *app1* and *App1* are distinct in an App Configuration store. Keep this in mind when you use configuration settings within an application because some frameworks handle configuration keys case-insensitively.

You can use any unicode character in key names entered into App Configuration except for `*`, `,`, and `\`. These characters are reserved. If you need to include a reserved character, you must escape it by using `\{Reserved Character}`. There's a combined size limit of 10,000 characters on a key-value pair. This limit includes all characters in the key, its value, and all associated optional attributes. Within this limit, you can have many hierarchical levels for keys.

Design key namespaces

There are two general approaches to naming keys used for configuration data: flat or hierarchical. These methods are similar from an application usage standpoint, but hierarchical naming offers a number of advantages:

- Easier to read. Instead of one long sequence of characters, delimiters in a hierarchical key name function as spaces in a sentence.
- Easier to manage. A key name hierarchy represents logical groups of configuration data.

- Easier to use. It's simpler to write a query that pattern-matches keys in a hierarchical structure and retrieves only a portion of configuration data.

Below are some examples of how you can structure your key names into a hierarchy:

- Based on component services

```
AppName:Service1:ApiEndpoint
AppName:Service2:ApiEndpoint
```

- Based on deployment regions

```
AppName:Region1:DbEndpoint
AppName:Region2:DbEndpoint
```

Label keys

Key values in App Configuration can optionally have a label attribute. Labels are used to differentiate key values with the same key. A key *app1* with labels *A* and *B* forms two separate keys in an App Configuration store. By default, the label for a key value is empty, or `null`.

Label provides a convenient way to create variants of a key. A common use of labels is to specify multiple environments for the same key:

```
Key = AppName:DbEndpoint & Label = Test
Key = AppName:DbEndpoint & Label = Staging
Key = AppName:DbEndpoint & Label = Production
```

Version key values

App Configuration doesn't version key values automatically as they're modified. Use labels as a way to create multiple versions of a key value. For example, you can input an application version number or a Git commit ID in labels to identify key values associated with a particular software build.

Query key values

Each key value is uniquely identified by its key plus a label that can be `null`. You query an App Configuration store for key values by specifying a pattern. The App Configuration store returns all key values that match the pattern and their corresponding values and attributes.

Values

Values assigned to keys are also unicode strings. You can use all unicode characters for values. There's an optional user-defined content type associated with each value. Use this attribute to store information, for example an encoding scheme, about a value that helps your application to process it properly.

Configuration data stored in an App Configuration store, which includes all keys and values, is encrypted at rest and in transit. App Configuration isn't a replacement solution for Azure Key Vault. Don't store application secrets in it.

Next unit: Manage application features

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Manage application features

3 minutes

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
}
```

In this case, if `featureFlag` is set to `True`, the enclosed code block is executed; otherwise, it's skipped. You can set the value of `featureFlag` statically, as in the following code example:

C# Copy

```
bool featureFlag = true;
```

You can also evaluate the flag's state based on certain rules:

C# Copy

```
bool featureFlag = isBetaUser();
```

A slightly more complicated feature flag pattern includes an `else` statement as well:

C# Copy

```
if (featureFlag) {
    // This following code will run if the featureFlag value is true
} else {
    // This following code will run if the featureFlag value is false
}
```

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:

JSON Copy

```
"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.

Next unit: Secure app configuration data

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

Secure app configuration data

3 minutes

In this unit you will learn how to secure your apps configuration data by using:

- Customer-managed keys
- Private endpoints
- Managed identities

Encrypt configuration data by using customer-managed keys

Azure App Configuration encrypts sensitive information at rest using a 256-bit AES encryption key provided by Microsoft. Every App Configuration instance has its own encryption key managed by the service and used to encrypt sensitive information. Sensitive information includes the values found in key-value pairs. When customer-managed key capability is enabled, App Configuration uses a managed identity assigned to the App Configuration instance to authenticate with Azure Active Directory. The managed identity then calls Azure Key Vault and wraps the App Configuration instance's encryption key. The wrapped encryption key is then stored and the unwrapped encryption key is cached within App Configuration for one hour. App Configuration refreshes the unwrapped version of the App Configuration instance's encryption key hourly. This ensures availability under normal operating conditions.

Enable customer-managed key capability

The following components are required to successfully enable the customer-managed key capability for Azure App Configuration:

- Standard tier Azure App Configuration instance
- Azure Key Vault with soft-delete and purge-protection features enabled
- An RSA or RSA-HSM key within the Key Vault: The key must not be expired, it must be enabled, and it must have both wrap and unwrap capabilities enabled

Once these resources are configured, two steps remain to allow Azure App Configuration to use the Key Vault key:

1. Assign a managed identity to the Azure App Configuration instance
2. Grant the identity `GET`, `WRAP`, and `UNWRAP` permissions in the target Key Vault's access policy.

Use private endpoints for Azure App Configuration

You can use private endpoints for Azure App Configuration to allow clients on a virtual network (VNet) to securely access data over a private link. The private endpoint uses an IP address from the VNet address space for your App Configuration store. Network traffic between the clients on the VNet and the App Configuration store traverses over the VNet using a private link on the Microsoft backbone network, eliminating exposure to the public internet.

Using private endpoints for your App Configuration store enables you to:

- Secure your application configuration details by configuring the firewall to block all connections to App Configuration on the public endpoint.
- Increase security for the virtual network (VNet) ensuring data doesn't escape from the VNet.
- Securely connect to the App Configuration store from on-premises networks that connect to the VNet using VPN or ExpressRoutes with private-peering.

Private endpoints for App Configuration

When creating a private endpoint, you must specify the App Configuration store to which it connects. If you have multiple App Configuration stores, you need a separate private endpoint for each store. Azure relies upon DNS resolution to route connections from the VNet to the configuration store over a private link. You can quickly find connections strings in the Azure portal by selecting your App Configuration store, then selecting **Settings** > **Access Keys**.

DNS changes for private endpoints

When you create a private endpoint, the DNS CNAME resource record for the configuration store is updated to an alias in a subdomain with the prefix `privatelink`. Azure also creates a private DNS zone corresponding to the `privatelink` subdomain, with the DNS A resource records for the private endpoints.

When you resolve the endpoint URL from within the VNet hosting the private endpoint, it resolves to the private endpoint of the store. When resolved from outside the VNet, the endpoint

URL resolves to the public endpoint. When you create a private endpoint, the public endpoint is disabled.

Managed identities

A managed identity from Azure Active Directory (AAD) allows Azure App Configuration to easily access other AAD-protected resources, such as Azure Key Vault. The identity is managed by the Azure platform. It does not require you to provision or rotate any secrets.

Your application can be granted two types of identities:

- A **system-assigned identity** is tied to your configuration store. It's deleted if your configuration store is deleted. A configuration store can only have one system-assigned identity.
- A **user-assigned identity** is a standalone Azure resource that can be assigned to your configuration store. A configuration store can have multiple user-assigned identities.

Add a system-assigned identity

To set up a managed identity using the Azure CLI, use the `az appconfig identity assign` command against an existing configuration store. The following Azure CLI example creates a system-assigned identity for an Azure App Configuration store named `myTestAppConfigStore`.

Bash Copy

```
az appconfig identity assign \
    --name myTestAppConfigStore \
    --resource-group myResourceGroup
```

Add a user-assigned identity

Creating an App Configuration store with a user-assigned identity requires that you create the identity and then assign its resource identifier to your store. The following Azure CLI examples create a user-assigned identity called `myUserAssignedIdentity` and assigns it to an Azure App Configuration store named `myTestAppConfigStore`.

Create an identity using the `az identity create` command:

Bash Copy

```
az identity create --resource-group myResourceGroup --name myUserAssignedIdentity
```

Assign the new user-assigned identity to the `myTestAppConfigStore` configuration store:

Bash Copy

```
az appconfig identity assign --name myTestAppConfigStore \
    --resource-group myResourceGroup \
    --identities /subscriptions/[subscription
id]/resourcegroups/myResourceGroup/providers/Microsoft.ManagedIdentity/userAssignedId
entities/myUserAssignedIdentity
```

Next unit: Knowledge check

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

200 XP

Knowledge check

3 minutes

Check your knowledge

1. Which type of encryption does Azure App Configuration use to encrypt data at rest?

- (●) 64-bit AES

  ✗ That's incorrect. Azure App Configuration encrypts sensitive information at rest using a 256-bit AES encryption key provided by Microsoft.

- ( ) 128-bit AES
- ( ) 256-bit AES

  ✓ That's correct. Azure App Configuration encrypts sensitive information at rest using a 256-bit AES encryption key provided by Microsoft.

2. Which of the below evaluates the state of a feature flag?

- ( ) Feature flag
- (●) Feature manager

  ✗ That's incorrect. A feature manager is an application package that handles the lifecycle of all the feature flags in an application.

- ( ) Filter

  ✓ That's correct. 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.

Next unit: Summary

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Summary

3 minutes

In this module, you learned how to:

- Explain the benefits of using Azure App Configuration
- Describe how Azure App Configuration stores information
- Implement feature management
- Securely access your app configuration information

Module complete:

Unlock achievement

How are we doing? ☆ ☆ ☆ ☆ ☆