Implement secure Azure solutions

Implement Azure Key Vault

Introduction

Azure Key Vault is a cloud service for securely storing and accessing secrets. A secret is anything that you want to tightly control access to, such as API keys, passwords, certificates, or cryptographic keys.

Explore Azure Key Vault

The Azure Key Vault service supports two types of containers: vaults and managed hardware security module(HSM) pools. Vaults support storing software and HSM-backed keys, secrets, and certificates. Managed HSM pools only support HSM-backed keys.

Azure Key Vault helps solve the following problems:

- **Secrets Management**: Azure Key Vault can be used to Securely store and tightly control access to tokens, passwords, certificates, API keys, and other secrets
- **Key Management**: Azure Key Vault can also be used as a Key Management solution. Azure Key Vault makes it easy to create and control the encryption keys used to encrypt your data.
- **Certificate Management**: Azure Key Vault is also a service that lets you easily provision, manage, and deploy public and private Secure Sockets Layer/Transport Layer Security (SSL/TLS) certificates for use with Azure and your internal connected resources.

Azure Key Vault has two service tiers: Standard, which encrypts with a software key, and a Premium tier, which includes hardware security module(HSM)-protected keys. To see a comparison between the Standard and Premium tiers, see the Azure Key Vault pricing page.

Key benefits of using Azure Key Vault

- **Centralized application secrets**: Centralizing storage of application secrets in Azure Key Vault allows you to control their distribution. For example, instead of storing the connection string in the app's code you can store it securely in Key Vault. Your applications can securely access the information they need by using URIs. These URIs allow the applications to retrieve specific versions of a secret.
- **Securely store secrets and keys**: Access to a key vault requires proper authentication and authorization before a caller (user or application) can get access. Authentication is done via Microsoft Entra ID. Authorization may be done via Azure role-based access control (Azure RBAC) or Key Vault access policy. Azure RBAC can be used for both management of the vaults and to access data stored in a vault, while key vault access policy can only be used when attempting to access data stored in a vault. Azure Key Vaults may be either software-protected or, with the Azure Key Vault Premium tier, hardware-protected by hardware security modules (HSMs).
- **Monitor access and use**: You can monitor activity by enabling logging for your vaults. You have control over your logs and you may secure them by restricting access and you may also delete logs that you no

longer need. Azure Key Vault can be configured to:

  - Archive to a storage account.
  - Stream to an event hub.
  - Send the logs to Azure Monitor logs.

- **Simplified administration of application secrets**: Security information must be secured, it must follow a life cycle, and it must be highly available. Azure Key Vault simplifies the process of meeting these requirements by:

  - Removing the need for in-house knowledge of Hardware Security Modules
  - Scaling up on short notice to meet your organization's usage spikes.
  - Replicating the contents of your Key Vault within a region and to a secondary region. Data replication ensures high availability and takes away the need of any action from the administrator to trigger the failover.
  - Providing standard Azure administration options via the portal, Azure CLI and PowerShell.
  - Automating certain tasks on certificates that you purchase from Public CAs, such as enrollment and renewal.

Discover Azure Key Vault best practices

Azure Key Vault is a tool for securely storing and accessing secrets. A secret is anything that you want to tightly control access to, such as API keys, passwords, or certificates. A vault is logical group of secrets.

Authentication

To do any operations with Key Vault, you first need to authenticate to it. There are three ways to authenticate to Key Vault:

- **Managed identities for Azure resources**: When you deploy an app on a virtual machine in Azure, you can assign an identity to your virtual machine that has access to Key Vault. You can also assign identities to other Azure resources. The benefit of this approach is that the app or service isn't managing the rotation of the first secret. Azure automatically rotates the service principal client secret associated with the identity. We recommend this approach as a best practice.

- **Service principal and certificate**: You can use a service principal and an associated certificate that has access to Key Vault. We don't recommend this approach because the application owner or developer must rotate the certificate.

- **Service principal and secret**: Although you can use a service principal and a secret to authenticate to Key Vault, we don't recommend it. It's hard to automatically rotate the bootstrap secret that's used to authenticate to Key Vault.

Encryption of data in transit

Azure Key Vault enforces Transport Layer Security (TLS) protocol to protect data when it's traveling between Azure Key Vault and clients. Clients negotiate a TLS connection with Azure Key Vault. TLS provides strong authentication, message privacy, and integrity (enabling detection of message tampering, interception, and forgery), interoperability, algorithm flexibility, and ease of deployment and use.

Perfect Forward Secrecy (PFS) protects connections between customers' client systems and Microsoft cloud services by unique keys. Connections also use RSA-based 2,048-bit encryption key lengths. This combination makes it difficult for someone to intercept and access data that is in transit.

Azure Key Vault best practices

- **Use separate key vaults**: Recommended using a vault per application per environment (Development, Pre-Production and Production). This pattern helps you not share secrets across environments and also reduces the threat if there is a breach.
- **Control access to your vault**: Key Vault data is sensitive and business critical, you need to secure access to your key vaults by allowing only authorized applications and users.
- **Backup**: Create regular back ups of your vault on update/delete/create of objects within a Vault.
- **Logging**: Be sure to turn on logging and alerts.
- **Recovery options**: Turn on soft-delete and purge protection if you want to guard against force deletion of the secret.

Authenticate to Azure Key Vault

Authentication with Key Vault works with Microsoft Entra ID, which is responsible for authenticating the identity of any given security principal.

For applications, there are two ways to obtain a service principal:

- Enable a system-assigned **managed identity** for the application. With managed identity, Azure internally manages the application's service principal and automatically authenticates the application with other Azure services. Managed identity is available for applications deployed to various services.
- If you can't use managed identity, you instead register the application with your Microsoft Entra tenant. Registration also creates a second application object that identifies the app across all tenants.

> **Note**: It is recommended to use a system-assigned managed identity.

The following is information on authenticating to Key Vault without using a managed identity.

Authentication to Key Vault in application code

Key Vault SDK is using Azure Identity client library, which allows seamless authentication to Key Vault across environments with same code. The table below provides information on the Azure Identity client libraries:

.NET	Python	Java	JavaScript
Azure Identity SDK .NET	Azure Identity SDK Python	Azure Identity SDK Java	Azure Identity SDK JavaScript

Authentication to Key Vault with REST

Access tokens must be sent to the service using the HTTP Authorization header:

```
PUT /keys/MYKEY?api-version=<api_version>  HTTP/1.1
Authorization: Bearer <access_token>
```

When an access token isn't supplied, or when a token isn't accepted by the service, an `HTTP 401` error is returned to the client and will include the `WWW-Authenticate` header, for example:

```
401 Not Authorized
WWW-Authenticate: Bearer authorization="…", resource="…"
```

The parameters on the `WWW-Authenticate` header are:

- authorization: The address of the OAuth2 authorization service that may be used to obtain an access token for the request.
- resource: The name of the resource (`https://vault.azure.net`) to use in the authorization request.

Other resources

- Azure Key Vault developer's guide
- Azure Key Vault availability and redundancy

Exercise: Set and retrieve a secret from Azure Key Vault by using Azure CLI

Create a Key Vault

1. Let's set some variables for the CLI commands to use to reduce the amount of retyping. Replace the `<myLocation>` variable string with a region that makes sense for you. The Key Vault name needs to be a globally unique name, and the following script generates a random string.

   ```
   myKeyVault=az204vault-$RANDOM
   myLocation=<myLocation>
   ```

2. Create a resource group.

   ```
   az group create --name az204-vault-rg --location $myLocation
   ```

3. Create a Key Vault by using the `az keyvault create` command.

   ```
   az keyvault create --name $myKeyVault --resource-group az204-vault-rg --
   location $myLocation
   ```

   > **Note**: This can take a few minutes to run.

Add and retrieve a secret

To add a secret to the vault, you just need to take a couple of extra steps.

1. Create a secret. Let's add a password that could be used by an app. The password is called **ExamplePassword** and will store the value of **hVFkk965BuUv** in it.

   ```
   az keyvault secret set --vault-name $myKeyVault --name "ExamplePassword" --
   value "hVFkk965BuUv"
   ```

2. Use the `az keyvault secret show` command to retrieve the secret.

   ```
   az keyvault secret show --name "ExamplePassword" --vault-name $myKeyVault
   ```

   This command returns some JSON. The last line contains the password in plain text.

   ```
   "value": "hVFkk965BuUv"
   ```

You've created a Key Vault, stored a secret, and retrieved it.

Clean up resources

If you no longer need the resources in this exercise use the following command to delete the resource group and associated Key Vault.

```
az group delete --name az204-vault-rg --no-wait
```

Implement managed identities

Introduction

A common challenge for developers is the management of secrets and credentials used to secure communication between different components making up a solution. Managed identities eliminate the need for developers to manage credentials.

Explore managed identities

A common challenge for developers is the management of secrets, credentials, certificates, and keys used to secure communication between services. Managed identities eliminate the need for developers to manage these credentials.

While developers can securely store the secrets in Azure Key Vault, services need a way to access Azure Key Vault. Managed identities provide an automatically managed identity in Microsoft Entra ID for applications to use when connecting to resources that support Microsoft Entra authentication. Applications can use managed identities to obtain Microsoft Entra tokens without having to manage any credentials.

Types of managed identities

There are two types of managed identities:

- A **system-assigned managed identity** is enabled directly on an Azure service instance. When the identity is enabled, Azure creates an identity for the instance in the Microsoft Entra tenant trusted by the subscription of the instance. After the identity is created, the credentials are provisioned onto the instance. The lifecycle of a system-assigned identity is directly tied to the Azure service instance that it's enabled on. If the instance is deleted, Azure automatically cleans up the credentials and the identity in Microsoft Entra ID.
- A **user-assigned managed identity** is created as a standalone Azure resource. Through a create process, Azure creates an identity in the Microsoft Entra tenant that's trusted by the subscription in use. After the identity is created, the identity can be assigned to one or more Azure service instances. The lifecycle of a user-assigned identity is managed separately from the lifecycle of the Azure service instances to which it's assigned.

Internally, managed identities are service principals of a special type, which are locked to only be used with Azure resources. When the managed identity is deleted, the corresponding service principal is automatically removed.

Characteristics of managed identities

The following table highlights some of the key differences between the two types of managed identities.

Property	System-assigned managed identity	User-assigned managed identity
Creation	Created as part of an Azure resource (for example, an Azure virtual machine or Azure App Service)	Created as a stand-alone Azure resource
Lifecycle	Shared lifecycle with the Azure resource that the managed identity is created with. When the parent resource is deleted, the managed identity is deleted as well.	Independent life-cycle. Must be explicitly deleted.
Sharing across Azure resources	Can't be shared, it can only be associated with a single Azure resource.	Can be shared. The same user-assigned managed identity can be associated with more than one Azure resource.

Following are common use cases for managed identities:

- System-assigned managed identity
  - Workloads contained within a single Azure resource.
  - Workloads needing independent identities.
  - For example, an application that runs on a single virtual machine.

- User-assigned managed identity
  - Workloads that run on multiple resources and can share a single identity.
  - Workloads needing preauthorization to a secure resource, as part of a provisioning flow.
  - Workloads where resources are recycled frequently, but permissions should stay consistent.
  - For example, a workload where multiple virtual machines need to access the same resource.

What Azure services support managed identities?

Managed identities for Azure resources can be used to authenticate to services that support Microsoft Entra authentication. For a list of Azure services that support the managed identities for Azure resources feature, visit Services that support managed identities for Azure resources.

The rest of this module uses Azure virtual machines in the examples, but the same concepts and similar actions can be applied to any resource in Azure that supports Microsoft Entra authentication.

Discover the managed identities authentication flow

In this unit, you learn how managed identities work with Azure virtual machines. Below are the flows detailing how the two types of managed identities work with an Azure virtual machine.

How a system-assigned managed identity works with an Azure virtual machine

1. Azure Resource Manager receives a request to enable the system-assigned managed identity on a virtual machine.

2. Azure Resource Manager creates a service principal in Microsoft Entra ID for the identity of the virtual machine. The service principal is created in the Microsoft Entra tenant that's trusted by the subscription.

3. Azure Resource Manager configures the identity on the virtual machine by updating the Azure Instance Metadata Service identity endpoint with the service principal client ID and certificate.

4. After the virtual machine has an identity, use the service principal information to grant the virtual machine access to Azure resources. To call Azure Resource Manager, use role-based access control in Microsoft Entra ID to assign the appropriate role to the virtual machine service principal. To call Key Vault, grant your code access to the specific secret or key in Key Vault.

5. Your code that's running on the virtual machine can request a token from the Azure Instance Metadata service endpoint, accessible only from within the virtual machine: `http://169.254.169.254/metadata/identity/oauth2/token`

6. A call is made to Microsoft Entra ID to request an access token (as specified in step 5) by using the client ID and certificate configured in step 3. Microsoft Entra ID returns a JSON Web Token (JWT) access token.

7. Your code sends the access token on a call to a service that supports Microsoft Entra authentication.

How a user-assigned managed identity works with an Azure virtual machine

1. Azure Resource Manager receives a request to create a user-assigned managed identity.

2. Azure Resource Manager creates a service principal in Microsoft Entra ID for the user-assigned managed identity. The service principal is created in the Microsoft Entra tenant that's trusted by the subscription.
3. Azure Resource Manager receives a request to configure the user-assigned managed identity on a virtual machine and updates the Azure Instance Metadata Service identity endpoint with the user-assigned managed identity service principal client ID and certificate.
4. After the user-assigned managed identity is created, use the service principal information to grant the identity access to Azure resources. To call Azure Resource Manager, use role-based access control in Microsoft Entra ID to assign the appropriate role to the service principal of the user-assigned identity. To call Key Vault, grant your code access to the specific secret or key in Key Vault.

   > **Note**: You can also do this step before step 3.

5. Your code that's running on the virtual machine can request a token from the Azure Instance Metadata Service identity endpoint, accessible only from within the virtual machine: `http://169.254.169.254/metadata/identity/oauth2/token`
6. A call is made to Microsoft Entra ID to request an access token (as specified in step 5) by using the client ID and certificate configured in step 3. Microsoft Entra ID returns a JSON Web Token (JWT) access token.
7. Your code sends the access token on a call to a service that supports Microsoft Entra authentication.

Configure managed identities

You can configure an Azure virtual machine with a managed identity during, or after, the creation of the virtual machine. CLI examples showing the commands for both system- and user-assigned identities are used in this unit.

System-assigned managed identity

To create, or enable, an Azure virtual machine with the system-assigned managed identity your account needs the **Virtual Machine Contributor** role assignment. No other Microsoft Entra directory role assignments are required.

Enable system-assigned managed identity during creation of an Azure virtual machine

The following example creates a virtual machine named *myVM* with a system-assigned managed identity, as requested by the `--assign-identity` parameter, with the specified `--role` and `--scope`. The `--admin-username` and `--admin-password` parameters specify the administrative user name and password account for virtual machine sign-in. Update these values as appropriate for your environment:

```
az vm create --resource-group myResourceGroup \
    --name myVM --image win2016datacenter \
    --generate-ssh-keys \
    --assign-identity \
    --role contributor \
    --scope mySubscription \
    --admin-username azureuser \
    --admin-password myPassword12
```

Enable system-assigned managed identity on an existing Azure virtual machine

Use the `az vm identity assign` command to assign the system-assigned identity to an existing virtual machine:

```
az vm identity assign -g myResourceGroup -n myVm
```

User-assigned managed identity

To assign a user-assigned identity to a virtual machine during its creation, your account needs the **Virtual Machine Contributor** and **Managed Identity Operator** role assignments. No other Microsoft Entra directory role assignments are required.

Enabling user-assigned managed identities is a two-step process:

1. Create the user-assigned identity
2. Assign the identity to a virtual machine

Create a user-assigned identity

Create a user-assigned managed identity using `az identity create`. The `-g` parameter specifies the resource group where the user-assigned managed identity is created, and the `-n` parameter specifies its name.

```
az identity create -g myResourceGroup -n myUserAssignedIdentity
```

Assign a user-assigned managed identity during the creation of an Azure virtual machine

The following example creates a virtual machine associated with the new user-assigned identity, as specified by the `--assign-identity` parameter, with the given `--role` and `--scope`.

```
az vm create \
--resource-group <RESOURCE GROUP> \
--name <VM NAME> \
--image Ubuntu2204 \
--admin-username <USER NAME> \
--admin-password <PASSWORD> \
--assign-identity <USER ASSIGNED IDENTITY NAME> \
--role <ROLE> \
--scope <SUBSCRIPTION>
```

Assign a user-assigned managed identity to an existing Azure virtual machine

Assign the user-assigned identity to your virtual machine using `az vm identity assign`.

```
az vm identity assign \
    -g <RESOURCE GROUP> \
    -n <VM NAME> \
    --identities <USER ASSIGNED IDENTITY>
```

Azure SDKs with managed identities for Azure resources support

Azure supports multiple programming platforms through a series of Azure SDKs. Several of them have been updated to support managed identities for Azure resources, and provide corresponding samples to demonstrate usage.

SDK	Sample
.NET	Manage resource from a virtual machine enabled with managed identities for Azure resources enabled
Java	Manage storage from a virtual machine enabled with managed identities for Azure resources
Node.js	Create a virtual machine with system-assigned managed identity enabled
Python	Create a virtual machine with system-assigned managed identity enabled
Ruby	Create Azure virtual machine with an system-assigned identity enabled

Acquire an access token

A client application can request managed identities for Azure resources app-only access token for accessing a given resource. The token is based on the managed identities for Azure resources service principal. The recommended method is to use the `DefaultAzureCredential`.

The Azure Identity library supports a `DefaultAzureCredential` type. `DefaultAzureCredential` automatically attempts to authenticate via multiple mechanisms, including environment variables or an interactive sign-in. The credential type can be used in your development environment using your own credentials. It can also be used in your production Azure environment using a managed identity. No code changes are required when you deploy your application.

> **Note**: `DefaultAzureCredential` is intended to simplify getting started with the SDK by handling common scenarios with reasonable default behaviors. Developers who want more control or whose scenario isn't served by the default settings should use other credential types.

The `DefaultAzureCredential` attempts to authenticate via the following mechanisms, in this order, stopping when one succeeds:

1. **Environment** - The `DefaultAzureCredential` reads account information specified via environment variables and use it to authenticate.

2. **Managed Identity** - If the application is deployed to an Azure host with Managed Identity enabled, the `DefaultAzureCredential` authenticates with that account.
3. **Visual Studio** - If the developer authenticated via Visual Studio, the `DefaultAzureCredential` authenticates with that account.
4. **Azure CLI** - If the developer authenticated an account via the Azure CLI `az login` command, the `DefaultAzureCredential` authenticates with that account. Visual Studio Code users can authenticate their development environment using the Azure CLI.
5. **Azure PowerShell** - If the developer authenticated an account via the Azure PowerShell `Connect-AzAccount` command, the `DefaultAzureCredential` authenticates with that account.
6. **Interactive browser** - If enabled, the `DefaultAzureCredential` interactively authenticates the developer via the current system's default browser. By default, this credential type is disabled.

Examples

The following examples use the Azure Identity SDK that can be added to a project with this command:

```
pip install azure-identity
```

Authenticate with `DefaultAzureCredential`

This example demonstrates authenticating the `SecretClient` from the `azure.keyvault.secrets` client library using the `DefaultAzureCredential`.

```python
from azure.identity import DefaultAzureCredential
from azure.keyvault.secrets import SecretClient

# Create a secret client using the DefaultAzureCredential
vault_url = "https://myvault.vault.azure.net/"
secret_client = SecretClient(vault_url, DefaultAzureCredential())
```

Specify a user-assigned managed identity with `DefaultAzureCredential`

This example demonstrates configuring the `DefaultAzureCredential` to authenticate a user-assigned identity when deployed to an Azure host. It then authenticates a `BlobClient` from the `azure.storage.blob` client library with the credential.

```python
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobClient

# When deployed to an azure host, the default azure credential will authenticate
the specified user assigned managed identity.
user_assigned_client_id = "<your managed identity client Id>"
credential =
DefaultAzureCredential(managed_identity_client_id=user_assigned_client_id)
```

```
blob_url = "https://myaccount.blob.core.windows.net/mycontainer/myblob"
blob_client = BlobClient(blob_url, credential)
```

Define a custom authentication flow with ChainedTokenCredential

While the `DefaultAzureCredential` is generally the quickest way to get started developing applications for Azure, more advanced users may want to customize the credentials considered when authenticating. The `ChainedTokenCredential` enables users to combine multiple credential instances to define a customized chain of credentials. This example demonstrates creating a `ChainedTokenCredential` which attempts to authenticate using managed identity, and fall back to authenticating via the Azure CLI if managed identity is unavailable in the current environment. The credential is then used to authenticate an `EventHubProducerClient` from the `azure.messaging.eventhubs` client library.

```
from azure.identity import ManagedIdentityCredential, AzureCliCredential,
ChainedTokenCredential
from azure.messaging.eventhubs import EventHubProducerClient

# Authenticate using managed identity if it is available; otherwise use the Azure
CLI to authenticate.
credential = ChainedTokenCredential(ManagedIdentityCredential(),
AzureCliCredential())

eventhub_namespace = "myeventhub.eventhubs.windows.net"
eventhub_name = "myhubpath"
eventhub_client = EventHubProducerClient(eventhub_namespace, eventhub_name,
credential)
```

Implement Azure App Configuration

Introduction

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

Explore the Azure App Configuration service

Azure App Configuration provides a service to centrally manage application settings and feature flags. 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 accesses 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
- Encryption of sensitive information at rest and 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	App Configuration provider for .NET
ASP.NET Core	App Configuration provider for .NET
.NET Framework and ASP.NET	App Configuration builder for .NET
Java Spring	App Configuration provider for Spring Cloud
JavaScript/Node.js	App Configuration provider for JavaScript
Python	App Configuration provider for Python
Others	App Configuration REST API

Create paired keys and values

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.

Here's an example of key names structured into a hierarchy based on component services:

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

The use of configuration data within application frameworks might dictate specific naming schemes for key-values. For example, Java's Spring Cloud framework defines `Environment` resources that supply settings to a Spring application. These resources are parameterized by variables that include *application name* and *profile*. Keys for Spring Cloud-related configuration data typically start with these two elements separated by a delimiter.

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 many 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.

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, a key-value has no label. To explicitly reference a key-value without a label, use `\0` (URL encoded as `%00`).

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 `\0`. 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 including 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.

Manage application 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 extra 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:

```
if feature_flag:
    # Run the following code
```

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

```
feature_flag = True
```

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

```
feature_flag = is_beta_user()
```

You can extend the conditional to set application behavior for either state:

```
if feature_flag:
    # This following code will run if the feature_flag value is true
else:
    # This following code will run if the feature_flag 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:

```
"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.

Secure app configuration data

In this unit you 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 Microsoft Entra ID. 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 to securely access data over a private link. The private endpoint uses an IP address from the virtual network address space for your App Configuration store. Network traffic between the clients on the virtual network and the App Configuration store traverses over the virtual network 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 ensuring data doesn't escape.
- Securely connect to the App Configuration store from on-premises networks that connect to the virtual network using VPN or ExpressRoutes with private-peering.

Managed identities

A managed identity from Microsoft Entra ID allows Azure App Configuration to easily access other Microsoft Entra ID-protected resources, such as Azure Key Vault. The identity is managed by the Azure platform. It doesn't 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`.

```
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 assign it to an Azure App Configuration store named `myTestAppConfigStore`.

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

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

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

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