# III.2 Implement secure cloud solutions

jeudi 16 septembre 2021 14:30

## Secure app configuration data by using App Configuration and Azure Key Vault

**Azure key vault**

- **Secrets** management : tokens, passwords, certificates (base64), API keys
- **Key** management
- **Certificate** management **:** public & private SSL/TLS certificates

**Why key vault ?**

- Centralized application secrets
- Securely store secrets and keys
- Monitor access and use
- Simplified administration of application secrets.

**Best practices**

- Authentication:
  - MI for Azure resources : Assign it to VM (or other resources) that has access to KV. No secret rotation management.
  - Service principal and certificate : Not recommended because the application owner must rotate the certificate.
  - Service principal and secret : Not recommended because the application owner must rotate the certificate.
- Use separate key vault per environment
- Control access to vault
- Regular backup
- Logging
- Turn on soft delete then purge protection

**Key vault creation**

Create a resource group.

Azure CLI

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

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

Azure CLI

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

**Azure App Configuration**

**Def :** Store all the settings for your app and secure their access in one place. Complements to Azure Key vault.

**Why to use :**

- 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
- 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

**How to use :**

With .NET Core : Use the App Configuration provider for .NET Core.

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Configuration.AzureAppConfiguration;

static void Main(string[] args)
{
    var builder = new ConfigurationBuilder();
    builder.AddAzureAppConfiguration(Environment.GetEnvironmentVariable("ConnectionString"));

    var config = builder.Build();
    Console.WriteLine(config["TestApp:Settings:Message"] ?? "Hello world!");
```

```
}
```

Key-value pairs based.

**Keys :**
- Hierarchical : by using / or :
- Case-sensitive
- Unicode based
- Can have a label attribute (typically used to create multiple version of a key value)
- Can be queried by specifying a pattern

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

**Values :**
- Unicode strings

-> Key /values are encrypted at rest & in transit in App Configuration store
-> Don't store application secrets in App Configuration store, use Key vault instead

**Feature flag**

Boolean + associated code block triggered if the boolean is true : if(featureFlag) { do; }

Azure App Configuration centralizes all the feature flags for the application.

Use a **feature manager** (= application package) to handle the lifecycle of all the feature flags and add caching and update states. The feature manager supports appsettings.json as a configuration source for feature flags.

**Declaration** : (key + true/false or filter)

```
"FeatureManagement": {
    "FeatureA": true, // Feature flag set to on
    "FeatureB": false, // Feature flag set to off
    "FeatureC": {
        "EnabledFor": [
            {
                "Name": "Percentage",
                "Parameters": {
                    "Value": 50
                }
            }
        ]
    }
}
```

**Secure app configuration data**

-> Enable customer-managed key capability to encrypt configuration data
- Assign a **MI** to the Azure App configuration instance (must be Standard tier)
- Grant the identity **GET, WRAP & UNWRAP** permissions in the target Key Vault's access policy. In the key vault, a RSA key is required. The key vault must have soft delete and purge protection enabled.

-> Use private endpoint to allow client on a VPN to securely access App Configuration instance over a private link. When private endpoint is enabled, it also changes the DNS of the App Configuration.

-> Use managed identity (system or user-assigned)

## Develop code that uses keys, secrets, and certificates stored in Azure Key Vault

**Authentication**

- Using DefaultAzureCredential

```
// Create a secret client using the DefaultAzureCredential
var client = new SecretClient(new Uri("https://myvault.vault.azure.net/"), new DefaultAzureCredential());
```

- Interactive authentication

```
// the includeInteractiveCredentials constructor parameter can be used to enable interactive authentication
var credential = new DefaultAzureCredential(includeInteractiveCredentials: true);

var eventHubClient = new EventHubProducerClient("myeventhub.eventhubs.windows.net", "myhubpath", credential);
```

- User managed identity

```
// When deployed to an azure host, the default azure credential will authenticate the specified user assigned managed identity.

string userAssignedClientId = "<your managed identity client Id>";
var credential = new DefaultAzureCredential(new DefaultAzureCredentialOptions { ManagedIdentityClientId = userAssignedClientId });

var blobClient = new BlobClient(new Uri("https://myaccount.blob.core.windows.net/mycontainer/myblob"), credential);
```

- Custom

```
// Authenticate using managed identity if it is available; otherwise use the Azure CLI to authenticate.

var credential = new ChainedTokenCredential(new ManagedIdentityCredential(), new AzureCliCredential());

var eventHubProducerClient = new EventHubProducerClient("myeventhub.eventhubs.windows.net", "myhubpath", credential);
```

**Manage keys**

```
using System;
using System.Threading.Tasks;
using Azure.Identity;
using Azure.Security.KeyVault.Keys;

namespace key_vault_console_app
{
    class Program
    {
        static async Task Main(string[] args)
        {
            const string keyName = "myKey";
            var keyVaultName = Environment.GetEnvironmentVariable("KEY_VAULT_NAME");
            var kvUri = $"https://{keyVaultName}.vault.azure.net";

            var client = new KeyClient(new Uri(kvUri), new DefaultAzureCredential());

            Console.Write($"Creating a key in {keyVaultName} called '{keyName}' ...");
            var createdKey = await client.CreateKeyAsync(keyName, KeyType.Rsa);
            Console.WriteLine("done.");

            Console.WriteLine($"Retrieving your key from {keyVaultName}.");
            var key = await client.GetKeyAsync(keyName);
            Console.WriteLine($"Your key version is '{key.Value.Properties.Version}'.");

            Console.Write($"Deleting your key from {keyVaultName} ...");
            var deleteOperation = await client.StartDeleteKeyAsync(keyName);
            // You only need to wait for completion if you want to purge or recover the key.
            await deleteOperation.WaitForCompletionAsync();
            Console.WriteLine("done.");

            Console.Write($"Purging your key from {keyVaultName} ...");
            await client.PurgeDeletedKeyAsync(keyName);
            Console.WriteLine(" done.");
        }
    }
}
```

**Add & retrieve a secret**

Azure CLI

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

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

Azure CLI

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

**Manage Certificates**

```
using Azure.Identity;
using Azure.Security.KeyVault.Certificates;

namespace key_vault_console_app
{
    class Program
    {
        static async Task Main(string[] args)
        {
            const string certificateName = "myCertificate";
            var keyVaultName = Environment.GetEnvironmentVariable("KEY_VAULT_NAME");
            var kvUri = $"https://{keyVaultName}.vault.azure.net";

            var client = new CertificateClient(new Uri(kvUri), new DefaultAzureCredential());
```

```
            Console.Write($"Creating a certificate in {keyVaultName} called '{certificateName}' ...");
            CertificateOperation operation = await client.StartCreateCertificateAsync(certificateName, CertificatePolicy.Default);
            await operation.WaitForCompletionAsync();
            Console.WriteLine(" done.");

            Console.WriteLine($"Retrieving your certificate from {keyVaultName}.");
            var certificate = await client.GetCertificateAsync(certificateName);
            Console.WriteLine($"Your certificate version is '{certificate.Value.Properties.Version}'.");

            Console.Write($"Deleting your certificate from {keyVaultName} ...");
            DeleteCertificateOperation deleteOperation = await client.StartDeleteCertificateAsync(certificateName);
            // You only need to wait for completion if you want to purge or recover the certificate.
            await deleteOperation.WaitForCompletionAsync();
            Console.WriteLine(" done.");

            Console.Write($"Purging your certificate from {keyVaultName} ...");
            await client.PurgeDeletedCertificateAsync(certificateName);
            Console.WriteLine(" done.");
        }
    }
}
```

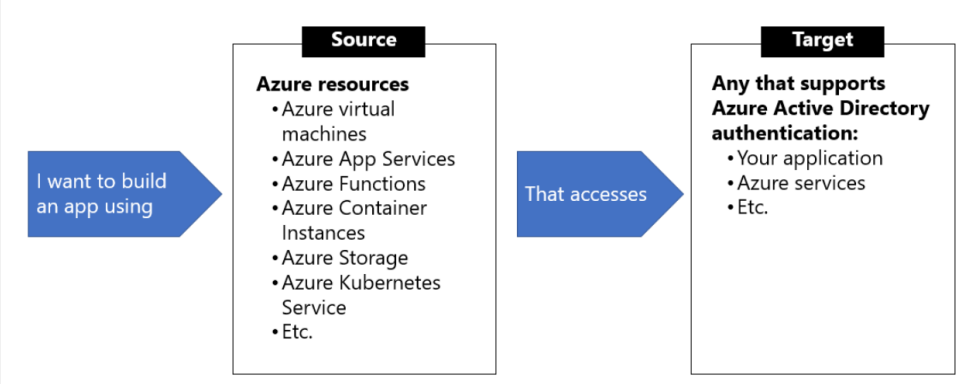
## Implement Managed Identities for Azure resources

**Def :** Provide an identity for applications to use when connecting to resources that support AAD authentication.

**Types of MI :**

| Characteristic | 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 | Cannot 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. |

**When to use :**




**Configuration**

System-assigned

- During creation of VM :

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

- Assignation to existing VM : **az vm identity assign** -g myResourceGroup -n myVm

User-assigned

- Create : **az identity create** -g myResourceGroup -n myUserAssignedIdentity
- Assign during creation of VM

```
az vm create \
--resource-group <RESOURCE GROUP> \
--name <VM NAME> \
--image UbuntuLTS \
--admin-username <USER NAME> \
--admin-password <PASSWORD> \
```

```
--admin-password <PASSWORD> \
--assign-identity <USER ASSIGNED IDENTITY NAME>
```

- Assign to existing VM

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

**Get token**

GET 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https://management.azure.com/' HTTP/1.1 Metadata: true

## Implement solutions that interact with Microsoft Graph

**Def :** RESTful web API that enables to interact with Microsoft cloud services resources. After you register your app and get authentication tokens for a user or service, you can make requests to the Microsoft Graph API.

**REST :**

HTTP Copy

```
{HTTP method} https://graph.microsoft.com/{version}/{resource}?{query-parameters}
```

The components of a request include:

- `{HTTP method}` - The HTTP method used on the request to Microsoft Graph.
- `{version}` - The version of the Microsoft Graph API your application is using.
- `{resource}` - The resource in Microsoft Graph that you're referencing.
- `{query-parameters}` - Optional OData query options or REST method parameters that customize the response.

After you make a request, a response is returned that includes:

- Status code - An HTTP status code that indicates success or failure.
- Response message - The data that you requested or the result of the operation. The response message can be empty for some operations.
- `nextLink` - If your request returns a lot of data, you need to page through it by using the URL returned in `@odata.nextLink`.

**Permission constraints**

- **All** grants permission for the app to perform the operations on all of the resources of the specified type in a directory. For example, *User.Read.All* potentially grants the app privileges to read the profiles of all of the users in a directory.
- **Shared** grants permission for the app to perform the operations on resources that other users have shared with the signed-in user. This constraint is mainly used with Outlook resources like mail, calendars, and contacts. For example, *Mail.Read.Shared*, grants privileges to read mail in the mailbox of the signed-in user as well as mail in mailboxes that other users in the organization have shared with the signed-in user.
- **AppFolder** grants permission for the app to read and write files in a dedicated folder in OneDrive. This constraint is only exposed on Files permissions and is only valid for Microsoft accounts.
- If **no constraint** is specified the app is limited to performing the operations on the resources owned by the signed-in user. For example, *User.Read* grants privileges to read the profile of the signed-in user only, and *Mail.Read* grants permission to read only mail in the mailbox of the signed-in user.

**Using .NET SDK**

Library : Microsoft.Graph

Create client

```
// Build a client application.
IPublicClientApplication publicClientApplication = PublicClientApplicationBuilder
            .Create("INSERT-CLIENT-APP-ID")
            .Build();
// Create an authentication provider by passing in a client application and graph scopes.
DeviceCodeProvider authProvider = new DeviceCodeProvider(publicClientApplication, graphScopes);
// Create a new instance of GraphServiceClient with the authentication provider.
GraphServiceClient graphClient = new GraphServiceClient(authProvider);
```

Read info

```
// GET https://graph.microsoft.com/v1.0/me

var user = await graphClient.Me
    .Request()
```

```
    .Request()
    .GetAsync();
```

Retrieve a list of entities

```
var messages = await graphClient.Me.Messages
    .Request()
    .Select(m => new {
        m.Subject,
        m.Sender
    })
    .Filter("<filter condition>")
    .OrderBy("receivedDateTime")
    .GetAsync();
```

Create entity

```
// POST https://graph.microsoft.com/v1.0/me/calendars

var calendar = new Calendar
{
    Name = "Volunteer"
};

var newCalendar = await graphClient.Me.Calendars
    .Request()
    .AddAsync(calendar);
```

Delete entity

```
// DELETE https://graph.microsoft.com/v1.0/me/messages/{message-id}

string messageId = "AQMkAGUy...";
var message = await graphClient.Me.Messages[messageId]
    .Request()
    .DeleteAsync();
```

**Best practices**

- **Authentication** : Use Oauth 2.0 access token (in Authorization header)
- **Authorization** : least privilege, right permission and consent type
- **Responses** : Use pagination and cache