



Exercise - Access secrets stored in Azure Key Vault

13 minutes

Sandbox activated! Time remaining: 48 min

You have used 1 of 10 sandboxes for today. More sandboxes will be available tomorrow.

Choose your development language

C#

JavaScript

Now that you know how enabling managed identities for Azure resources creates an identity for our app to use for authentication, we'll create an app that uses that identity to access secrets in the vault.

Reading secrets in an ASP.NET Core app

The Azure Key Vault API is a REST API that handles all management and usage of keys and vaults. Each secret in a vault has a unique URL, and secret values are retrieved with HTTP GET requests.

The official Key Vault client for .NET Core is the `KeyVaultClient` class in the `Microsoft.Azure.KeyVault` NuGet package. You don't need to use it directly, though — with ASP.NET Core's `AddAzureKeyVault` method, you can load all the secrets from a vault into the Configuration API at startup. This technique enables you to access all of your secrets by name using the same `IConfiguration` interface you use for the rest of your configuration. Apps that use `AddAzureKeyVault` require both **Get** and **List** permissions to the vault.

Tip

Regardless of the framework or language you use to build your app, you should design it to cache secret values locally or load them into memory at startup unless you have a specific reason not to. Reading them directly from the vault every time you need them is unnecessarily slow and expensive.

`AddAzureKeyVault` only requires the vault name as an input, which we'll get from our local app configuration. It also automatically handles managed identity authentication — when used in an app deployed to Azure App Service with managed identities for Azure resources enabled, it will detect the managed identities token service and use it to authenticate. It's a good fit for most scenarios and implements all best practices, and we'll use it in this unit's exercise.

Handling secrets in an app

Once a secret is loaded into your app, it's up to your app to handle it securely. In the app we build in this module, we write our secret value out to the client response and view it in a web browser to demonstrate that it has been loaded successfully. **Returning a secret value to the client is *not* something you'd normally do!** Usually, you'll use secrets to do things like initialize client libraries for databases or remote APIs.

Important

Always carefully review your code to ensure that your app never writes secrets to any kind of output, including logs, storage, and responses.

Exercise

We'll create a new ASP.NET Core web API and use `AddAzureKeyVault` to load the secret from our vault.

Create the app

In the Azure Cloud Shell terminal, run the following to create a new ASP.NET Core web API application and open it in the editor.

console

 Copy

```
dotnet new webapi -o KeyVaultDemoApp
cd KeyVaultDemoApp
code .
```

After the editor loads, run the following commands in the shell to add the NuGet package containing `AddAzureKeyVault` and restore all of the app's dependencies.

console

 Copy

```
dotnet add package Microsoft.Extensions.Configuration.AzureKeyVault -v 2.1.1
dotnet restore
```

Add code to load and use secrets

To demonstrate good usage of Key Vault, we will modify our app to load secrets from the vault at startup. We'll also add a new controller with an endpoint that gets our **SecretPassword** secret from the vault.

First, the app startup: Open `Program.cs`, delete the contents and replace them with the following code:

C#

 Copy

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;

namespace KeyVaultDemoApp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebApplicationBuilder(args).Build().Run();
        }

        public static IWebApplicationBuilder CreateWebApplicationBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .ConfigureAppConfiguration((context, config) =>
                {
                    // Build the current set of configuration to load values from
                    // JSON files and environment variables, including VaultName.
                    var builtConfig = config.Build();

                    // Use VaultName from the configuration to create the full
                    vault URL.
                    var vaultUrl =
                    $"https://{builtConfig["VaultName"]}.vault.azure.net/";

                    // Load all secrets from the vault into configuration. This
                    will automatically
                    // authenticate to the vault using a managed identity. If a
                    managed identity
                    // is not available, it will check if Visual Studio and/or the
                    Azure CLI are
                    // installed locally and see if they are configured with
                    credentials that can
                    // access the vault.
                    config.AddAzureKeyVault(vaultUrl);
```

```
    })  
    .UseStartup<Startup>();  
}  
}
```

❗ Important

Make sure to save files when you're done editing them. You can do this either through the "..." menu, or the accelerator key (`Ctrl+S` on Windows and Linux, `Cmd+S` on macOS).

The only change from the starter code is the addition of `ConfigureAppConfiguration`. This is where we load the vault name from configuration and call `AddAzureKeyVault` with it.

Next, the controller: Create a new file in the `Controllers` folder called `SecretTestController.cs` and paste the following code into it.

💡 Tip

To create a new file, use the `touch` command in the shell. In this case, use `touch Controllers/SecretTestController.cs`. You'll need to click the refresh button in the Files pane of the editor to see it there.

C#

 Copy

```
using System;  
using Microsoft.AspNetCore.Http;  
using Microsoft.AspNetCore.Mvc;  
using Microsoft.Extensions.Configuration;  
  
namespace KeyVaultDemoApp.Controllers  
{  
    [Route("api/[controller]")]  
    public class SecretTestController : ControllerBase  
    {  
        private readonly IConfiguration _configuration;  
  
        public SecretTestController(IConfiguration configuration)  
        {  
            _configuration = configuration;  
        }  
  
        [HttpGet]  
        public IActionResult Get()  
        {  
            // Get the secret value from configuration. This can be done anywhere  
            // we have access to IConfiguration. This does not call the Key Vault  
            // API, because the secrets were loaded at startup.  
        }  
    }  
}
```


```
var secretName = "SecretPassword";
var secretValue = _configuration[secretName];

if (secretValue == null)
{
    return StatusCode(
        StatusCodes.Status500InternalServerError,
        $"Error: No secret named {secretName} was found...");
}
else {
    return Content($"Secret value: {secretValue}" +
        Environment.NewLine + Environment.NewLine +
        "This is for testing only! Never output a secret " +
        "to a response or anywhere else in a real app!");
}
}
}
```

Run `dotnet build` in the shell to make sure everything compiles. The app is ready to run — now let's get it into Azure!

Next unit: Exercise - Configure, deploy, and run in Azure

[Continue >](#)

 English (United States)

[Previous Version Docs](#) • [Blog](#) • [Contribute](#) • [Privacy & Cookies](#) • [Terms of Use](#) • [Trademarks](#) •

© Microsoft 2020

```
Requesting a Cloud Shell.Succeeded.  
Connecting terminal...  
  
Welcome to Azure Cloud Shell  
  
Type "az" to use Azure CLI  
Type "help" to learn about Cloud Shell
```