az204-handson/Lab2 Access resource secrets.md at master · cloudstateu/az204-handson (github.com)

Lab: Access resource secrets more securely across services | Student lab manual

Time: approximately 45 min.

Objectives

After you complete this lab, you'll be able to:

- Create an Azure Key Vault and store secrets in the key vault.
- Create a system-assigned managed identity for an Azure App Service instance.
- Create a Key Vault access policy for an Azure Active Directory identity or application.
- Use the Azure SDK for .NET to download a blob with an Azure Function.

Instructions

Task 1: Create an Azure Storage account

1. Create a new storage account with the following details:
   - New resource group: **YOUR_RESOURCE_GROUP**
   - Name: **securestor[yourname]**
   - Location: **West Europe**
   - Performance: **Standard**
   - Account kind: **StorageV2 (general purpose v2)**
   - Replication: **Locally-redundant storage (LRS)**
2. Open the **Access Keys** blade of your newly created storage account instance.
3. Record the value in the **Connection string** text box. You'll use this value later in this lab.

Task 2: Create an Azure key vault

1. Create a new key vault with the following details:
    - Existing resource group: **YOUR_RESOURCE_GROUP**
    - Name: **securevault[yourname]**
    - Region: **West Europe**
    - Pricing tier: **Standard**

Task 3: Create an Azure Functions app

1. Create a new function app with the following details:

    - Existing resource group: **YOUR_RESOURCE_GROUP**
    - App name: **securefun[yourname]**
    - Publish: **Code**
    - Runtime Stack: **.NET**
    - Version: **3.1**
    - Region: **West Europe**
    - Operating system: **Linux**
    - Storage account: **securestor[yourname]**
    - Plan: **Consumption (Serverless)**
    - Enable Application Insights: **No**

    Note: Wait for Azure to finish creating the function app before you move forward with the lab. You'll receive a notification when the app is created.

Configure secrets and identities

Task 1: Configure a system-assigned managed service identity

1. Access the **securefun[yourname]** function app that you created earlier in this lab.
2. Browse to the **Identity** option from the **Settings** section.
3. Enable the system-assigned managed identity, and then save your changes.

Task 2: Create a Key Vault secret

1. Access the **securevault[yourname]** key vault that you created earlier in this lab.
2. Select the **Secrets** link in the **Settings** section.
3. Create a new secret with the following settings:

   - Name: **storagecredentials**
   - Value: ***Storage connection string***
   - Enabled: **Yes**

   Note: Use the storage account connection string that you recorded earlier in this lab for the value of this secret.

4. Select through the secret to find the metadata for its latest version.
5. Record the value of the **Secret Identifier** text box because you'll use this later in the lab.

Task 3: Configure a Key Vault access policy

1. Access the **securevault[yourname]** key vault that you created earlier in this lab.
2. Browse to the **Access Policies** link in the **Settings** section.
3. Create a new access policy with the following settings:
   - Principal: **securefunc[yourname]**

     Note: The system-assigned managed identity you created earlier in this lab will have the same name as the Azure Functions resource.

   - Key permissions: **None**
   - Secret permissions: **GET**
   - Certificate permissions: **None**
   - Authorized application: **None**
4. Save your changes to the list of **Access Policies**.

Task 4: Create a Key Vault-derived application setting

1. Access the **securefunc[yourname]** function app that you created earlier in this lab.
2. Browse to the **Configuration** option from the **Settings** section.
3. Create a new application setting by using the following details:
   - Name: **StorageConnectionString**
   - Value: **@Microsoft.KeyVault(SecretUri=*Secret Identifier*)**
   - Deployment slot setting: **Not selected**

   Note: You'll need to build a reference to your ***Secret Identifier*** by using the previous syntax. For example, if your ***Secret Identifier*** is `https://securevaultstudent.vault.azure.net/secrets/storagecredentials/17b41386df3e4191b92f089f5efb4cbf`, your value would be `@Microsoft.KeyVault(SecretUri=https://securevaultstudent.vault.azure.net/secrets/storagecredentials/17b41386df3e4191b92f089f5efb4cbf)`.

4. Save your changes to the application settings.

Review: You created a system-assigned managed service identity for your function app and then gave that identity the appropriate permissions to get the value of a secret in your key vault. Finally, you created a secret that you referenced within your function app's configuration settings.

Build an Azure Functions app

Task 1: Initialize a function project

1. Open the **Windows Terminal** application.

2. Create new empty directory for your new function project. Enter this directory.

   Note: Function created in the next step will be named after this folder.

3. Use the **Azure Functions Core Tools** to create a new local Azure Functions project with the following details:

   ```
   func init --worker-runtime dotnet
   ```

   Note: You can review the documentation to [create a new project][azure-functions-core-tools-new-project] using the **Azure Functions Core Tools**.

4. **Build** the .NET Core 3.1 project:

   ```
   dotnet build
   ```

Task 2: Create an HTTP-triggered function

1. Still in the open command prompt,create a new function with the following details:

   - template: **HTTP trigger**
   - name: **FileParser**

   ```
   func new --template "HTTP trigger" --name "FileParser"
   ```

   Note: You can review the documentation to [create a new function][azure-functions-core-tools-new-function] using the **Azure Functions Core Tools**.

2. Close the currently running **Windows Terminal** application.

Task 3: Configure and read an application setting

1. Open **Visual Studio Code**.

2. Using **Visual Studio Code**, open the solution folder.

3. Open the **local.settings.json** file.

4. Update the value of the **Values** object by adding a new setting named **StorageConnectionString** and setting it to a string value of **[TEST VALUE]**:

   ```
   "Values": {
       "AzureWebJobsStorage": "UseDevelopmentStorage=true",
       "FUNCTIONS_WORKER_RUNTIME": "dotnet",
       "StorageConnectionString": "[TEST VALUE]"
   }
   ```

5. Open the **FileParser.cs** file.

6. In the code editor, delete all the code within the **FileParser.cs** file.

7. Add **using directives** for the **Microsoft.AspNetCore.Mvc**, **Microsoft.Azure.WebJobs**, **Microsoft.AspNetCore.Http**, **System**, and **System.Threading.Tasks** namespaces:

   ```
   using Microsoft.AspNetCore.Mvc;
   using Microsoft.Azure.WebJobs;
   using Microsoft.AspNetCore.Http;
   using System;
   using System.Threading.Tasks;
   ```

8. Create a new **public static** class named **FileParser**:

   ```
   public static class FileParser
   { }
   ```

9. Within the **FileParser** class, create a new **public static** *asynchronous* method named **Run** that returns a variable of type **Task<IActionResult>** and that also takes in a variable of type **HttpRequest** named *request*:

   ```
   public static async Task<IActionResult> Run(
       HttpRequest request)
   { }
   ```

10. Append an attribute to the **Run** method of type **FunctionNameAttribute** that has its **name** parameter set to a value of **FileParser**:

    ```
    [FunctionName("FileParser")]
    public static async Task<IActionResult> Run(
        HttpRequest request)
    { }
    ```

11. Append an attribute to the **request** parameter of type **HttpTriggerAttribute** that has its **methods** parameter array set to a single value of **GET**:

    ```
    [FunctionName("FileParser")]
    public static async Task<IActionResult> Run(
        [HttpTrigger("GET")] HttpRequest request)
    { }
    ```

12. Within the **Run** method, retrieve the value of the **StorageConnectionString** application setting by using the **Environment.GetEnvironmentVariable** method and storing the result in a **string** variable named **connectionString**:

    ```
    string connectionString =
    Environment.GetEnvironmentVariable("StorageConnectionString");
    ```

13. Finally, return the value of the **connectionString** variable as the HTTP response:

    ```
    return new OkObjectResult(connectionString);
    ```

14. **Save** the **FileParser.cs** file.

Task 4: Validate the local function

1. Open the **Windows Terminal** application.

2. Change the current directory to the azure function project directory.

3. Start the function app project:

    ```
    func start --build
    ```

    Note: You can review the documentation to [start the function app project locally][azure-functions-core-tools-start-function] using the **Azure Functions Core Tools**.

4. Open web browser and navigate to`http://localhost:7071/api/FileParser`:

5. Observe the **[TEST VALUE]** value of the **StorageConnectionString** being returned as the result of the HTTP request.

6. Close all currently running instances of the **Windows Terminal** application.

Task 5: Deploy using the Azure Functions Core Tools

1. Open the **Windows Terminal** application.

2. Change the current directory to the Azure function project directory.

3. Log in to the Azure Command-Line Interface (CLI) by using your Azure credentials:

   ```
   az login
   ```

4. Publish the function app project:

   ```
   func azure functionapp publish <function-app-name>
   ```

   Note: For example, if your **Function App name** is **securefuncstudent**, your command would be `func azure functionapp publish securefuncstudent`. You can review the documentation to [publish the local function app project][azure-functions-core-tools-publish-azure] using the **Azure Functions Core Tools**.

5. Wait for the deployment to finalize before you move forward with the lab.

6. Close the currently running **Windows Terminal** application.

Task 6: Test the Key Vault-derived application setting

1. Sign in to the Azure portal (https://portal.azure.com).
2. Access the **securefunc[yourname]** function app that you created earlier in this lab.
3. From the **App Service** blade, locate and open the **Functions** section, and then locate and open the **FileParser** function.
4. In the **Function** blade, select the **Code + Test** option from the **Developer** section.
5. In the function editor, select **Test/Run**.
6. In the pop-up dialog that appears, perform the following actions:
   - In the **HTTP method** list, select **GET**.
7. Select **Run** to test the function.
8. Observe the result of the test run. The result should be your Azure Storage connection string.

Access Azure Blob Storage data

Task 1: Upload a sample Storage blob

1. Access the **securestor[yourname]** storage account that you created earlier in this lab.

2. Select the **Containers** link in the **Blob service** section, and then create a new container with the following settings:
    - Name: **drop**
    - Access level: **Private (no anonymous access)**
3. Browse to the new **drop** container, and then select **Upload** to upload the **records.json** file.

Task 2: Pull and configure the Azure SDK for .NET

1. Open the **Windows Terminal** application.

2. Change the current directory to the Azure function project directory, created earlier in this lab.

3. When you receive the open command prompt, add version **12.6.0** of the **Azure.Storage.Blobs** package from NuGet:

    ```
    dotnet add package Azure.Storage.Blobs --version 12.6.0
    ```

    Note: The Azure.Storage.Blobs NuGet package references the subset of the Azure SDK for .NET required to write code for Azure Blob Storage.

4. Close the currently running **Windows Terminal** application.

5. Using **Visual Studio Code**, open the **FileParser.cs** file.

6. Add a **using directive** for the **Azure.Storage.Blobs** namespace:

    ```
    using Azure.Storage.Blobs;
    ```

Task 3: Write Azure Blob Storage code using the Azure SDK for .NET

1. Within the **Run** method of the **FileParser** class, delete the following line of code:

    ```
    return new OkObjectResult(connectionString);
    ```

2. Still within the **Run** method, create a new instance of the **BlobClient** class by passing in your *connectionString* variable, a `"drop"` string value, and a `"records.json"` string value to the constructor:

    ```
    BlobClient blob = new BlobClient(connectionString, "drop", "records.json");
    ```

3. Still within the **Run** method, use the **BlobClient.DownloadAsync** method to download the contents of the referenced blob asynchronously and store the result in a variable named *response*:

```
var response = await blob.DownloadAsync();
```

4. Still within the **Run** method, return the value of the various content stored in the *content* variable by using the **FileStreamResult** class constructor:

   ```
   return new FileStreamResult(response?.Value?.Content,
   response?.Value?.ContentType);
   ```

5. **Save** the **FileParser.cs** file.

Task 4: Deploy and validate the Azure Functions app

1. Open the **Windows Terminal** application.

2. Change the current directory to the Azure function project directory.

3. Log in to the Azure CLI by using your Azure credentials:

   ```
   az login
   ```

4. Publish the function app project again:

   ```
   func azure functionapp publish <function-app-name>
   ```

   Note: As an example, if your **Function App name** is **securefuncstudent**, your command would be `func azure functionapp publish securefuncstudent`. You can review the documentation to [publish the local function app project][azure-functions-core-tools-publish-azure] using the **Azure Functions Core Tools**.

5. Wait for the deployment to finalize before you move forward with the lab.

6. Close the currently running **Windows Terminal** application.

7. Sign in to the Azure portal (https://portal.azure.com).

8. Access the **securefunc[yourname]** function app that you created earlier in this lab.

9. From the **App Service** blade, locate and open the **Functions** section, then locate and open the **FileParser** function.

10. In the **Function** blade, select the **Code + Test** option from the **Developer** section.

11. In the function editor, select **Test/Run**.

12. In the pop-up dialog that appears, perform the following actions:

- In the **HTTP method** list, select **GET**.

13. Select **Run** to test the function.
14. Observe the results of the test run. The output will contain the content of the **$/drop/records.json** blob stored in your Azure Storage account.