Lab: Retrieving Azure Storage resources and metadata by using the Azure Storage SDK for .NET
Student lab manual

Lab scenario

You're preparing to host a web application in Microsoft Azure that uses a combination of raster and vector graphics. As a development group, your team has decided to store any multimedia content in Azure Storage and manage it in an automated fashion by using C# code in .NET. Before you begin this significant milestone, you have decided to take some time to learn the newest version of the .NET SDK that's used to access Storage by creating a simple application to manage and enumerate blobs and containers.

Objectives

After you complete this lab, you will be able to:

- Create containers and upload blobs by using the Azure portal.
- Enumerate blobs and containers by using the Microsoft Azure Storage SDK for .NET.
- Pull blob metadata by using the Storage SDK.

Lab setup

- Estimated time: **45 minutes**

Instructions

Exercise 1: Create Azure resources

Task 1: Open the Azure portal

1. Sign in to the Azure portal (https://portal.azure.com).
2. If this is your first time signing in to the Azure portal, you'll notice a dialog box offering a tour of the portal. Select **Get Started** to skip the tour.

Task 2: Create a Storage account

1. Create a new storage account with the following details:
   - New resource group: **StorageMedia**
   - Name: **mediastor*[yourname]***
   - Location: **North Europe**
   - Performance: **Standard**
   - Account kind: **StorageV2 (general purpose v2)**
   - Replication: **Locally-redundant storage (LRS)**

   Note: Wait for Azure to finish creating the storage account before you move forward with the lab. You'll receive a notification when the account is created.
2. Open the **Properties** section of your newly created storage account instance.
3. Record the value in the **Primary Blob Service Endpoint** text box. You'll use this value later in this lab.
4. Open the **Access Keys** section of the storage account instance.

5. Record the value in the **Storage account name** text box and any of the **Key** text boxes. You'll use these values later in this lab.

Review

In this exercise, you created a new Storage account to use throughout the remainder of the lab.

Exercise 2: Upload a blob into a container

Task 1: Create storage account containers

1. Access the **mediastor*[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: **raster-graphics**
   - Public access level: **Private (no anonymous access)**
3. Select the **Containers** link in the **Blob service** section, and then create a new container with the following settings:
   - Name: **compressed-audio**
   - Public access level: **Private (no anonymous access)**
4. Observe the updated list of containers.

Task 2: Upload a storage account blob

1. Access the **mediastor*[yourname]*** storage account that you created earlier in this lab.
2. Select the **Containers** link in the **Blob service** section, and then select the recently created **raster-graphics** container.
3. In the **raster-graphics** container, select **Upload** to upload the **graph.jpg** file in the **Allfiles (F):\Allfiles\Labs\03\Starter\Images** folder on your lab VM.

   Note: We recommend that you enable the **Overwrite if files already exist** option.

Review

In this exercise, you created a couple of placeholder containers in the storage account and populated one of the containers with a blob.

Exercise 3: Access containers by using the .NET SDK

Task 1: Create .NET project

1. Using Visual Studio Code, open the **Allfiles (F):\Allfiles\Labs\03\Starter\BlobManager** folder.
2. Using a terminal, create a new .NET project named **BlobManager** in the current folder:

```
dotnet new console --name BlobManager --output .
```

   Note: The **dotnet new** command will create a new **console** project in a folder with the same name as the project.

3. Using the same terminal, import version 12.0.0 of **Azure.Storage.Blobs** from NuGet:

```
dotnet add package Azure.Storage.Blobs --version 12.0.0
```

Note: The **dotnet add package** command will add the **Azure.Storage.Blobs** package from NuGet. For more information, go to Azure.Storage.Blobs.

4. Using the same terminal, build the .NET web application:

```
dotnet build
```

5. Close the current terminal.

Task 2: Modify the Program class to access Storage

1. Open the **Program.cs** file in Visual Studio Code.
2. Delete all existing code in the **Program.cs** file.
3. Add the following **using** directives for libraries that will be referenced by the application:

```
using Azure.Storage;
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;
using System;
using System.Threading.Tasks;
```

4. Create a new **Program** class with three constant string properties named **blobServiceEndpoint**, **storageAccountName**, and **storageAccountKey**, and then create an asynchronous **Main** entry point method:

```
public class Program
{
    private const string blobServiceEndpoint = "";
    private const string storageAccountName = "";
    private const string storageAccountKey = "";

    public static async Task Main(string[] args)
    {
    }
}
```

5. Update the **blobServiceEndpoint** string constant by setting its value to the **Primary Blob Service Endpoint** of the storage account that you recorded earlier in this lab.
6. Update the **storageAccountName** string constant by setting its value to the **Storage account name** of the Storage account that you recorded earlier in this lab.
7. Update the **storageAccountKey** string constant by setting its value to the **Key** of the Storage account that you recorded earlier in this lab.

Task 3: Connect to the Azure Storage blob service endpoint

1. In the **Main** method, add the following block of code to connect to the storage account and to retrieve account metadata:

```
StorageSharedKeyCredential accountCredentials = new
StorageSharedKeyCredential(storageAccountName, storageAccountKey);

BlobServiceClient serviceClient = new BlobServiceClient(new Uri(blobServiceEndpoint),
accountCredentials);

AccountInfo info = await serviceClient.GetAccountInfoAsync();
```

2. Still in the **Main** method, add the following block of code to print metadata about the storage account:

```
await Console.Out.WriteLineAsync($"Connected to Azure Storage Account");
await Console.Out.WriteLineAsync($"Account name:\t{storageAccountName}");
await Console.Out.WriteLineAsync($"Account kind:\t{info?.AccountKind}");
await Console.Out.WriteLineAsync($"Account sku:\t{info?.SkuName}");
```

3. Save the **Program.cs** file.
4. Using a terminal, run the console application project:

```
dotnet run
```

5. Observe the output from the currently running console application. The output contains metadata for the Storage account that was retrieved from the service.
6. Close the current terminal.

Task 4: Enumerate the existing containers

1. In the **Program** class, create a new **private static** method named **EnumerateContainersAsync** that's asynchronous and has a single parameter of type **BlobServiceClient**:

```
private static async Task EnumerateContainersAsync(BlobServiceClient client)
{
}
```

2. In the **EnumerateContainersAsync** method, create an asynchronous **foreach** loop that iterates over the results of an invocation of the **GetBlobContainersAsync** method of the **BlobServiceClient** class and prints out the name of each container:

```
await foreach (BlobContainerItem container in client.GetBlobContainersAsync())
{
    await Console.Out.WriteLineAsync($"Container:\t{container.Name}");
}
```

3. In the **Main** method, add a new line of code to invoke the **EnumerateContainersAsync** method, passing in the *serviceClient* variable as a parameter:

```
await EnumerateContainersAsync(serviceClient);
```

4. Save the **Program.cs** file.
5. Using a terminal, run the console application project:

```
dotnet run
```

6. Observe the output from the currently running console application. The updated output includes a list of every existing container in the account.
7. Close the current terminal.

Review

In this exercise, you accessed existing containers by using the Storage SDK.

Exercise 4: Retrieve blob Uniform Resource Identifiers (URIs) by using the .NET SDK

Task 1: Enumerate the blobs in an existing container by using the SDK

1. In the **Program** class, create a new **private static** method named **EnumerateBlobsAsync** that's asynchronous and has two types of parameters, **BlobServiceClient** and **string**:

```
private static async Task EnumerateBlobsAsync(BlobServiceClient client, string containerName)
{
}
```

2. In the **EnumerateBlobsAsync** method, get a new instance of the **BlobContainerClient** class by using the **GetBlobContainerClient** method of the **BlobServiceClient** class, passing in the **containerName** parameter:

```
BlobContainerClient container = client.GetBlobContainerClient(containerName);
```

3. In the **EnumerateBlobsAsync** method, render the name of the container that will be enumerated:

```
await Console.Out.WriteLineAsync($"Searching:\t{container.Name}");
```

4. In the **EnumerateBlobsAsync** method, create an asynchronous **foreach** loop that iterates over the results of an invocation of the **GetBlobsAsync** method of the **BlobContainerClient** class and prints out the name of each blob:

```
await foreach (BlobItem blob in container.GetBlobsAsync())
{
    await Console.Out.WriteLineAsync($"Existing Blob:\t{blob.Name}");
}
```

5. In the **Main** method, add a new line of code to create a variable named *existingContainerName* with a value of **raster-graphics**:

```
string existingContainerName = "raster-graphics";
```

6. In the **Main** method, add a new line of code to invoke the **EnumerateBlobsAsync** method, passing in the *serviceClient* and *existingContainerName* variables as parameters:

```
await EnumerateBlobsAsync(serviceClient, existingContainerName);
```

7. Save the **Program.cs** file.
8. Using a terminal, run the console application project:

```
dotnet run
```

Note: If there are any build errors, review the **Program.cs** file in the **Allfiles (F):\Allfiles\Labs\03\Solution\BlobManager** folder.

9. Observe the output from the currently running console application. The updated output includes metadata about the existing container and blobs.
10. Close the current terminal.

Task 2: Create a new container by using the SDK

1. In the **Program** class, create a new **private static** method named **GetContainerAsync** that's asynchronous and has two parameter types, **BlobServiceClient** and **string**:

```
private static async Task<BlobContainerClient> GetContainerAsync(BlobServiceClient client,
string containerName)
{
}
```

2. In the **GetContainerAsync** method, get a new instance of the **BlobContainerClient** class by using the **GetBlobContainerClient** method of the **BlobServiceClient** class, passing in the **containerName** parameter. Invoke the **CreateIfNotExistsAsync** method of the **BlobContainerClient** class:

```
BlobContainerClient container = client.GetBlobContainerClient(containerName);
await container.CreateIfNotExistsAsync(PublicAccessType.Blob);
```

3. In the **GetContainerAsync** method, render the name of the container that was potentially created:

```
await Console.Out.WriteLineAsync($"New Container:\t{container.Name}");
```

4. Return the instance of the **BlobContainerClient** class named **container** as the result of the **GetContainerAsync** method:

```
return container;
```

5. In the **Main** method, add a new line of code to create a variable named *newContainerName* with a value of **vector-graphics**:

```
string newContainerName = "vector-graphics";
```

6. In the **Main** method, add a new line of code to invoke the **GetContainerAsync** method, passing in the *serviceClient* and *newContainerName* variables as parameters:

```
BlobContainerClient containerClient = await GetContainerAsync(serviceClient, newContainerName);
```

7. Save the **Program.cs** file.
8. Using a terminal, run the console application project:

```
dotnet run
```

9. Observe the output from the currently running console application. The updated output includes metadata about the new container and blobs.
10. Close the current terminal.

Task 3: Upload a new blob by using the portal

1. Access the **mediastor*[yourname]*** storage account that you created earlier in this lab.
2. Select the **Containers** link in the **Blob service** section, and then select the newly created **vector-graphics** container.
3. In the **vector-graphics** container, select **Upload** to upload the **graph.svg** file in the **Allfiles (F): \Allfiles\Labs\03\Starter\Images** folder on your lab VM.

   Note: We recommend that you enable the **Overwrite if files already exist** option.

Task 4: Access blob URI by using the SDK

1. In the **Program** class, create a new **private static** method named **GetBlobAsync** that's asynchronous and has two types of parameters, **BlobContainerClient** and **string**:

```
private static async Task<BlobClient> GetBlobAsync(BlobContainerClient client, string blobName)
{
}
```

2. In the **GetBlobAsync** method, get a new instance of the **BlobClient** class by using the **GetBlobClient** method of the **BlobContainerClient** class, passing in the **blobName** parameter:

```
BlobClient blob = client.GetBlobClient(blobName);
```

3. In the **GetBlobAsync** method, render the name of the blob that was referenced:

```
await Console.Out.WriteLineAsync($"Blob Found:\t{blob.Name}");
```

4. Return the instance of the **BlobClient** class named **blob** as the result of the **GetBlobAsync** method:

```
return blob;
```

5. In the **Main** method, add a new line of code to create a variable named *uploadedBlobName* with a value of **vector-graphics**:

```
string uploadedBlobName = "graph.svg";
```

6. In the **Main** method, add a new line of code to invoke the **GetBlobAsync** method, passing in the *containerClient* and *uploadedBlobName* variables as parameters and store the result in a variable named *blobClient* of type **BlobClient**:

```
BlobClient blobClient = await GetBlobAsync(containerClient, uploadedBlobName);
```

7. In the **Main** method, add a new line of code to render the **Uri** property of the *blobClient* variable:

```
await Console.Out.WriteLineAsync($"Blob Url:\t{blobClient.Uri}");
```

8. Save the **Program.cs** file.
9. Using a terminal, run the console application project:

```
dotnet run
```

10. Observe the output from the currently running console application. The updated output includes the final URL to access the blob online. Record the value of this URL to use later in the lab.

    Note: The URL will likely be similar to the following string:
    https://mediastor*[yourname]*.blob.core.windows.net/vector-graphics/graph.svg

11. Close the current terminal.

Task 5: Test the URI by using a browser

1. Using a new browser window or tab, go to the URL for the blob, and then find the blob's contents.
2. You should now notice the Scalable Vector Graphics (SVG) file in your browser window.

Review

In this exercise, you created containers and managed blobs by using the Storage SDK.

Exercise 5: Clean up your subscription

Task 1: Open Azure Cloud Shell and list resource groups

1. In the Azure portal, select the **Cloud Shell** icon to open a new shell instance.
2. If Cloud Shell isn't already configured, configure the shell for Bash by using the default settings.

Task 2: Delete a resource group

1. Enter the following command, and then select Enter to delete the **StorageMedia** resource group:

```
az group delete --name StorageMedia --no-wait --yes
```

2. Close the Cloud Shell pane in the portal.

Task 3: Close the active application

1. the currently running Microsoft Edge application.

Review

In this exercise, you cleaned up your subscription by removing the resource group that was used in this lab.