**Connect an app to Azure Storage**

**Learning objectives**

In this you will:

* Explore the data types and services supported by an Azure storage account.
* Create a new Azure storage account using the Azure portal.
* Create a new application and connect the app to an Azure Storage account.

**Create a Node.js application**

Node.js is a popular framework for running JavaScript apps. It is most commonly used for web apps, but you can use it to run logic from the command line as well. If you have the tools installed locally, you can run the following steps from a command line. Alternatively, you can use Cloud Shell on the right side of the window to execute the following steps.

1. Sign in to Cloud Shell or open a command-line session, and create a new folder named "PhotoSharingApp".

***mkdir PhotoSharingApp***

1. Change into the new folder, and use npm to initialize a new Node.js app. This will create a **package.json** file containing metadata that describes the app.

***cd PhotoSharingApp***

***npm init -y***

1. Create a new source file, **index.js**, which is where our code will go.

***touch index.js***

1. Open the **index.js** file with an editor. If you are using Cloud Shell, enter code . to open an editor.
2. Paste the following program into the **index.js** file. Press Ctrl+V or right-click to paste.

***#!/usr/bin/env node***

***function main() {***

***console.log('Hello, World!');***

***}***

***main();***

1. Press Ctrl+S to save the file. Alternatively, in the Cloud Shell integrated editor title bar, select the ellipsis, and then select **Save** from the context menu.
2. Run the app to make sure it executes correctly.

***node index.js***

The output "Hello World!" should display in the console.

**Create an Azure storage account**

Now that we have an app, we need an Azure storage account to work with.

**Use the Azure CLI to create an Azure storage account**

We'll use the az storage account create command to create a new storage account. There are several parameters to control the configuration of the storage account.

| **Option** | **Description** |
| --- | --- |
| --name | A **storage account name**. The name will be used to generate the public URL used to access the data in the account. It must be unique across all existing storage account names in Azure. It must be 3 to 24 characters long and can contain only lowercase letters and numbers. |
| --resource-group | Create Resource Group if not already exists. |
| --location | Select a location near you (see below). |
| --sku | This decides the storage account performance and replication model. Options include **Premium\_LRS, Standard\_GRS, Standard\_LRS, Standard\_RAGRS, and Standard\_ZRS**. |

**Select a location**

The free sandbox allows you to create resources in a subset of the Azure global regions. Select a region from this list when you create resources:

* westus2
* southcentralus
* centralus
* eastus
* westeurope
* southeastasia
* japaneast
* brazilsouth
* australiasoutheast
* centralindia

**Create a storage account**

1. Use the following example command to create a storage account. Remember to replace <name> with your unique storage account name.

Az group create –-name <resource Group Name> --location <location for resource group>

az storage account create \

--resource-group <resource group name>\

--location westus \

--sku Standard\_LRS \

--name <name>

**Interact with the Azure Storage APIs**

Azure Storage provides a REST API to work with the containers and data stored in each account. Each type of data you can store has its own independent API. Recall that we have four specific data types:

* **Blobs** for unstructured data such as binary and text files.
* **Queues** for persistent messaging.
* **Tables** for structured storage of key/values.
* **Files** for traditional SMB file shares.

**Use the REST API**

The Storage REST APIs are accessible from anywhere on the Internet by any app that can send an HTTP/HTTPS request and receive an HTTP/HTTPS response.

For example, if you wanted to list all the blobs in a container, you would create a request something like:

GET <https://[url-for-service-account]/?comp=list&include=metadata>

This request returns an XML block with data specific to the account:

*<?xml version="1.0" encoding="utf-8"?>*

*<EnumerationResults AccountName="https://[url-for-service-account]/">*

*<Containers>*

*<Container>*

*<Name>container1</Name>*

*<Url>https://[url-for-service-account]/container1</Url>*

*<Properties>*

*<Last-Modified>Sun, 24 Sep 2018 18:09:03 GMT</Last-Modified>*

*<Etag>0x8CAE7D0C4AF4487</Etag>*

*</Properties>*

*<Metadata>*

*<Color>orange</Color>*

*<ContainerNumber>01</ContainerNumber>*

*<SomeMetadataName>SomeMetadataValue</SomeMetadataName>*

*</Metadata>*

*</Container>*

*<Container>*

*<Name>container2</Name>*

*<Url>https://[url-for-service-account]/container2</Url>*

*<Properties>*

*<Last-Modified>Sun, 24 Sep 2018 17:26:40 GMT</Last-Modified>*

*<Etag>0x8CAE7CAD8C24928</Etag>*

*</Properties>*

*<Metadata>*

*<Color>pink</Color>*

*<ContainerNumber>02</ContainerNumber>*

*<SomeMetadataName>SomeMetadataValue</SomeMetadataName>*

*</Metadata>*

*</Container>*

*<Container>*

*<Name>container3</Name>*

*<Url>https://[url-for-service-account]/container3</Url>*

*<Properties>*

*<Last-Modified>Sun, 24 Sep 2018 17:26:40 GMT</Last-Modified>*

*<Etag>0x8CAE7CAD8EAC0BB</Etag>*

*</Properties>*

*<Metadata>*

*<Color>brown</Color>*

*<ContainerNumber>03</ContainerNumber>*

*<SomeMetadataName>SomeMetadataValue</SomeMetadataName>*

*</Metadata>*

*</Container>*

*</Containers>*

*<NextMarker>container4</NextMarker>*

*</EnumerationResults>*

However, this approach requires a lot of manual parsing and the creation of HTTP packets to work with each API. For this reason, Azure provides pre-built *client libraries* that make working with the service easier for common languages and frameworks.

**Use a client library**

Client libraries can save a significant amount of work for app developers because the API has been tested and often provides nicer wrappers around the data models sent and received by the REST API.

Microsoft has Azure client libraries that support a number of languages and frameworks, including:

* .NET
* Java
* Python
* Node.js
* Go

For example, to retrieve the same list of blobs in **C#,** we could use the following code snippet:

string containerName = "...";

BlobContainerClient container = new BlobContainerClient(connectionString, containerName);

var blobs = container.GetBlobs();

foreach (var blob in blobs)

{

Console.WriteLine($"{blob.Name} --> Created On: {blob.Properties.CreatedOn:YYYY-MM-dd HH:mm:ss} Size: {blob.Properties.ContentLength}");

}

Or in Java Script

const containerName = '...';

const containerClient = blobServiceClient.getContainerClient(containerName);

let blobs = containerClient.listBlobsFlat();

for await (const blob of blobs) {

console.log(`${blob.name} --> Created: ${blob.properties.createdOn} Size: ${blob.properties.contentLength}`);

}

**Add the storage client library to your app**

Let's integrate the **Azure Storage Blob Client Library for JavaScript** into your application. The client library for Node.js is available through the Node Package manager (npm). You'll want to add the **@azure/storage-blob** package to your **packages.json** file.

**Add the Azure Storage package**

1. In Azure Cloud Shell, cd to the **PhotoSharingApp** directory if you aren't already there.
2. Enter the following code to add the **@azure/storage-blob** package to the application. Make sure to supply the --save option so it persists to **packages.json**.

npm install @azure/storage-blob –save

1. You will see some console activity while the client library and all the required dependencies are downloaded. When the download is complete, let's build and run the app again to make sure everything is ready to go. Enter the following command.

node index.js

1. As before, the return output "Hello, World!" appears.

**Connect to your Azure storage account**

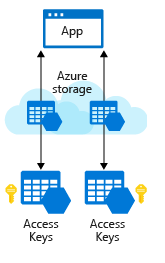
You have added the required client libraries to your application and are ready to connect to your Azure storage account.

To work with data in a storage account, your app will need two pieces of data:

* Access key
* REST API endpoint

**Security access keys**

Each storage account has two unique *access keys* that are used to secure the storage account. If your app needs to connect to multiple storage accounts, your app will require an access key for each storage account.



**REST API endpoint**

In addition to access keys for authentication to storage accounts, your app will need to know the storage service endpoints to issue the REST requests.

The REST endpoint is a combination of your storage account *name*, the data type, and a known domain. For example:

| **Data type** | **Example endpoint** |
| --- | --- |
| Blobs | https://[name].blob.core.windows.net/ |
| Queues | https://[name].queue.core.windows.net/ |
| Table | https://[name].table.core.windows.net/ |
| Files | https://[name].file.core.windows.net/ |

If you have a custom domain tied to Azure, then you can also create a custom domain URL for the endpoint.

**Connection strings**

The simplest way to handle access keys and endpoint URLs within applications is to use **storage account connection strings**. A connection string provides all needed connectivity information in a single text string.

Azure Storage connection strings look similar to the following example, but with the access key and account name of your specific storage account:

*DefaultEndpointsProtocol=https;AccountName={your-storage};*

*AccountKey={your-access-key};*

*EndpointSuffix=core.windows.net*

**Add Azure Storage configuration to your app**

Let's add support to our Node.js application to retrieve a connection string from a configuration file. We'll start by adding the necessary plumbing to manage a configuration from our JavaScript file.

**Create an .env configuration file**

1. In the Azure Cloud Shell session enter the following command to make sure you are in the correct working directory for your project.

cd PhotoSharingApp

1. Enter the following command to create a file named **.env**.

touch .env

1. Open the project in Cloud Shell editor.

code .

1. In the editor, select the **.env** file, and add the following text.

AZURE\_STORAGE\_CONNECTION\_STRING=<value>

1. Save the file using the keyboard shortcut (Ctrl+S) or select the ellipsis icon (...) in the title bar of the editor, and then select Save from the context menu.
2. Now, we need use an Azure command to obtain the actual storage account connection string. In Cloud Shell session, run the following command, replacing <name> with the storage account name that you created in the previous exercise.

az storage account show-connection-string \

--resource-group learn-b86124f7-226b-419d-bfea-8c707e0bcf09 \

--query connectionString \

--name <name>

1. The response is a connection string bounded by quotes, which looks much like the following example:

"DefaultEndpointsProtocol=https;EndpointSuffix=core.windows.net;AccountName=storage1ab;AccountKey=QtSCGB...7AeoW0Hw=="

1. Copy the connection string, and in the **.env** file, replace <value> with this connection string.
2. Press Ctrl+S to save the file.

**Add support to read an environment configuration file**

Node.js apps can include support to read from the **.env** file by adding the **dotenv** package.

1. In Cloud Shell session, run the following command to add a dependency to the **dotenv** package using npm.

npm install dotenv –save

**Add code to read the configuration file**

Now that we have added the required libraries to enable reading the configuration, we need to enable that functionality in our application.

1. In the editor, open the **index.js** file.
2. At the top of the file, is a line of code #!/usr/bin/env node. Underneath that line, add following line of code:

require('dotenv').config();

1. Press Ctrl+S to save the file.

Now that we have the JSON config all wired up, we can start adding code to use our storage account

**Connect your application to your Azure Storage account**

The Azure Storage client library provides an object model that is used to interact with Azure storage accounts. It's used to quickly connect to an Azure storage account and use the Azure Storage service APIs.

**Azure Storage client library object model**

The **Azure Storage Blob client library for JavaScript** contains a number of client objects for interacting with Azure Storage Blobs. At the top of this hierarchy is the BlobServiceClient object. To use this object in your JavaScript code:

1. Open your *index.js* file in the code editor and add the following statement immediately after the require('dotenv').config(); statement.

const { BlobServiceClient } = require("@azure/storage-blob");

1. Now, you need to create a BlobServiceClient object in your code by obtaining the storage account connection string and passing it to the factory method fromConnectionString on the BlobServiceClient object. Add the following lines of code:

const storageAccountConnectionString = process.env.AZURE\_STORAGE\_CONNECTION\_STRING;

const blobServiceClient = BlobServiceClient.fromConnectionString(storageAccountConnectionString);

1. After you have a client object defined in your program, you can use methods on the client object to perform actual work. Methods that make network calls are intentionally asynchronous. The library uses Promises to return asynchronous results. For that reason, you need to mark your main function as async. Replace the main function object with the following code. The line that begins with two forward slashes is a comment

sync function main() {

// Function code here

}

1. Save the changes you have made to *index.js*.

Now let's add some code to execute an operation against our storage account. A storage account is organized into one or more containers which act like folders in your storage account. The blob objects (files) you create in your storage account are stored in one of these containers. You need to create a container in your storage account to store your photos.

The container name must be between 3 and 63 characters long and may only contain lowercase letters and the dash (-) character. For this application, we will simply use the name *photos*.

To create a container in your storage account, you need to obtain a ContainerClient object that represents the container in the storage account. Even though a container does not yet exist in our storage account, the ContainerClient can be used to create the container as well as manage it once it is created.

1. To obtain the ContainerClient object, you call the getContainerClient method on the BlobServiceClient object and provide the name of the container as a parameter. Then, to create the container in your Azure Storage account, use the createIfNotExists method on the **containerClient** object. Replace the // Function Code here comment line with these statements:

// Create a container (folder) if it does not exist

const containerName = 'photos';

const containerClient = blobServiceClient.getContainerClient(containerName);

const createContainerResponse = await containerClient.createIfNotExists();

console.log(`Create container ${containerName} successfully`, createContainerResponse.succeeded);

}

1. Save the changes you made to the *index.js* file.
2. In the Cloud Shell command line enter the following command to build and run your program, which creates the container in your storage account.

node index.js

The first time you run the program, you should see a message that the container was created successfully, with a status of true. The second and subsequent times you run your program, you will see a similar message with a status of false because the container already exists.

1. You can verify you have created the container by running the following Azure CLI command. Remember to replace <name> with the name of your storage account.

az storage container list \

--account-name <name>

Now that our program is connecting to our Azure storage account and we have created a *photos* container, let's have our program upload an image to it.

**Upload an image to your Azure Storage account**

Let's add code to upload an image to our Azure Storage account. In this example, we're going to upload the following image to our Azure Storage container.

*wget https://github.com/MicrosoftDocs/mslearn-connect-app-to-azure-storage/blob/main/images/docs-and-friends-selfie-stick.png?raw=true -O docs-and-friends-selfie-stick.png*

*Or downlod image using*

*https://github.com/MicrosoftDocs/mslearn-connect-app-to-azure-storage/blob/main/images/docs-and-friends-selfie-stick.png*

**Upload an image to blob storage**

To work with blob objects in your Azure Storage container, you use a BlockBlobClient object. The BlockBlobClient object has methods to upload, download, list, and delete blob objects in a container. To get a BlockBlobObject, call the method getBlockBlobClient on the ContainerClient object. Then, you can use the uploadFile method to upload your image to Azure Storage.

1. Add this code to your *index.js* program immediately after the code that creates the container.

const filename = 'docs-and-friends-selfie-stick.png';

const blockBlobClient = containerClient.getBlockBlobClient(filename);

blockBlobClient.uploadFile(filename);

**List objects in an Azure Blob Storage container**

1. To verify that our code is working, we can call the listBlobsFlat method on the ContainerClient object in our program.

let blobs = containerClient.listBlobsFlat();

let blob = await blobs.next();

while (!blob.done) {

console.log(`${blob.value.name} --> Created: ${blob.value.properties.createdOn} Size: ${blob.value.properties.contentLength}`);

blob = await blobs.next();

This code will print all the blobs in our Azure Blob Storage container with the date the blob was created and its size. For our program, this code should print one row representing the single image we have uploaded.

1. The final file should look like this.

#!/usr/bin/env node

require('dotenv').config();

const { BlobServiceClient } = require("@azure/storage-blob");

const storageAccountConnectionString = process.env.AZURE\_STORAGE\_CONNECTION\_STRING;

const blobServiceClient = BlobServiceClient.fromConnectionString(storageAccountConnectionString);

async function main() {

// Create a container (folder) if it does not exist

const containerName = 'photos';

const containerClient = blobServiceClient.getContainerClient(containerName);

if ( !containerClient.exists()) {

const createContainerResponse = await containerClient.createIfNotExists();

console.log(`Create container ${containerName} successfully`, createContainerResponse.succeeded);

}

else {

console.log(`Container ${containerName} already exists`);

}

// Upload the file

const filename = 'docs-and-friends-selfie-stick.png';

const blockBlobClient = containerClient.getBlockBlobClient(filename);

blockBlobClient.uploadFile(filename);

// Get a list of all the blobs in the container

let blobs = containerClient.listBlobsFlat();

let blob = await blobs.next();

while (!blob.done) {

console.log(`${blob.value.name} --> Created: ${blob.value.properties.createdOn} Size: ${blob.value.properties.contentLength}`);

blob = await blobs.next();

}

}

main();

**Run the app**

1. Build and run the application.

node index.js