Develop solutions that use Blob storage

Explore Azure Blob storage

Introduction

This module will delve into the world of Azure Blob storage, Microsoft's cloud-based solution for storing large amounts of unstructured data. This service is versatile and can be used for a variety of purposes, from serving images or documents to a browser, to storing files for distributed access, and even streaming video and audio.

Explore Azure Blob storage

Azure Blob storage is Microsoft's object storage solution for the cloud. Blob storage is optimized for storing massive amounts of unstructured data. Unstructured data is data that doesn't adhere to a particular data model or definition, such as text or binary data.

Blob storage is designed for:

- Serving images or documents directly to a browser.
- Storing files for distributed access.
- Streaming video and audio.
- Writing to log files.
- Storing data for backup and restore, disaster recovery, and archiving.
- Storing data for analysis by an on-premises or Azure-hosted service.

Users or client applications can access objects in Blob storage via HTTP/HTTPS, from anywhere in the world. Objects in Blob storage are accessible via the Azure Storage REST API, Azure PowerShell, Azure CLI, or an Azure Storage client library.

An Azure Storage account is the top-level container for all of your Azure Blob storage. The storage account provides a unique namespace for your Azure Storage data that is accessible from anywhere in the world over HTTP or HTTPS.

Types of storage accounts

Azure Storage offers two performance levels of storage accounts, standard and premium. Each performance level supports different features and has its own pricing model.

- **Standard**: This is the standard general-purpose v2 account and is recommended for most scenarios using Azure Storage.
- **Premium**: Premium accounts offer higher performance by using solid-state drives. If you create a premium account you can choose between three account types, block blobs, page blobs, or file shares.

The following table describes the types of storage accounts recommended by Microsoft for most scenarios using Blob storage.

Type of storage account	Supported storage services	Redundancy options	Usage
Standard general-purpose v2	Blob Storage (including Data Lake Storage), Queue Storage, Table Storage, and Azure Files	Locally redundant storage (LRS) / geo-redundant storage (GRS) / read-access geo-redundant storage (RA-GRS) Zone-redundant storage (ZRS) / geo-zone-redundant storage (GZRS) / read-access geo-zone-redundant storage (RA-GZRS)	Standard storage account type for blobs, file shares, queues, and tables. Recommended for most scenarios using Azure Storage. If you want support for network file system (NFS) in Azure Files, use the premium file shares account type.
Premium block blobs	Blob Storage (including Data Lake Storage)	LRS and ZRS	Premium storage account type for block blobs and append blobs. Recommended for scenarios with high transaction rates or that use smaller objects or require consistently low storage latency.
Premium file shares	Azure Files	LRS and ZRS	Premium storage account type for file shares only. Recommended for enterprise or high-performance scale applications.
Premium page blobs	Page blobs only	LRS and ZRS	Premium storage account type for page blobs only.

Access tiers for block blob data

Azure Storage provides different options for accessing block blob data based on usage patterns. Each access tier in Azure Storage is optimized for a particular pattern of data usage. By selecting the right access tier for your needs, you can store your block blob data in the most cost-effective manner.

The available access tiers are:

- The **Hot** access tier, which is optimized for frequent access of objects in the storage account. The Hot tier has the highest storage costs, but the lowest access costs. New storage accounts are created in the hot tier by default.
- The **Cool** access tier, which is optimized for storing large amounts of data that is infrequently accessed and stored for a minimum of 30 days. The Cool tier has lower storage costs and higher access costs compared to the Hot tier.

- The **Cold** access tier, which is optimized for storing data that is infrequently accessed and stored for a minimum of 90 days. The cold tier has lower storage costs and higher access costs compared to the cool tier.
- The **Archive** tier, which is available only for individual block blobs. The archive tier is optimized for data that can tolerate several hours of retrieval latency and remains in the Archive tier for a minimum 180 days. The archive tier is the most cost-effective option for storing data, but accessing that data is more expensive than accessing data in the hot or cool tiers.

If there's a change in the usage pattern of your data, you can switch between these access tiers at any time.

Discover Azure Blob storage resource types

Blob storage offers three types of resources:

- The **storage account**.
- A **container** in the storage account
- A **blob** in a container

Storage accounts

A storage account provides a unique namespace in Azure for your data. Every object that you store in Azure Storage has an address that includes your unique account name. The combination of the account name and the Azure Storage blob endpoint forms the base address for the objects in your storage account.

For example, if your storage account is named *mystorageaccount*, then the default endpoint for Blob storage is:

```
http://mystorageaccount.blob.core.windows.net
```

Containers

A container organizes a set of blobs, similar to a directory in a file system. A storage account can include an unlimited number of containers, and a container can store an unlimited number of blobs.

A container name must be a valid DNS name, as it forms part of the unique URI (Uniform resource identifier) used to address the container or its blobs. Follow these rules when naming a container:

- Container names can be between 3 and 63 characters long.
- Container names must start with a letter or number, and can contain only lowercase letters, numbers, and the dash (-) character.
- Two or more consecutive dash characters aren't permitted in container names.

The URI for a container is similar to:

```
https://myaccount.blob.core.windows.net/mycontainer
```

Blobs

Azure Storage supports three types of blobs:

- **Block blobs** store text and binary data. Block blobs are made up of blocks of data that can be managed individually. Block blobs can store up to about 190.7 TiB.
- **Append blobs** are made up of blocks like block blobs, but are optimized for append operations. Append blobs are ideal for scenarios such as logging data from virtual machines.
- **Page blobs** store random access files up to 8 TB in size. Page blobs store virtual hard drive (VHD) files and serve as disks for Azure virtual machines.

The URI for a blob is similar to:

```
https://myaccount.blob.core.windows.net/mycontainer/myblob
```

or

```
https://myaccount.blob.core.windows.net/mycontainer/myvirtualdirectory/myb
lob
```

Explore Azure Storage security features

Azure Storage uses service-side encryption (SSE) to automatically encrypt your data when it's persisted to the cloud. Azure Storage encryption protects your data and to help you to meet your organizational security and compliance commitments.

Microsoft recommends using service-side encryption to protect your data for most scenarios. However, the Azure Storage client libraries for Blob Storage and Queue Storage also provide client-side encryption for customers who need to encrypt data on the client.

Azure Storage encryption for data at rest

Azure Storage automatically encrypts your data when persisting it to the cloud. Encryption protects your data and helps you meet your organizational security and compliance commitments. Data in Azure Storage is encrypted and decrypted transparently using 256-bit Advanced Encryption Standard (AES) encryption, one of the strongest block ciphers available, and is Federal Information Processing Standards (FIPS) 140-2 compliant. Azure Storage encryption is similar to BitLocker encryption on Windows.

Azure Storage encryption is enabled for all storage accounts and can't be disabled. Because your data is secured by default, you don't need to modify your code or applications to take advantage of Azure Storage encryption.

Data in a storage account is encrypted regardless of performance tier, access tier, or deployment model. All new and existing block blobs, append blobs, and page blobs are encrypted, including blobs in the archive tier. All Azure Storage redundancy options support encryption, and all data in both the primary and

secondary regions is encrypted when geo-replication is enabled. All Azure Storage resources are encrypted, including blobs, disks, files, queues, and tables. All object metadata is also encrypted.

There's no extra cost for Azure Storage encryption.

Encryption key management

Data in a new storage account is encrypted with Microsoft-managed keys by default. You can continue to rely on Microsoft-managed keys for the encryption of your data, or you can manage encryption with your own keys. If you choose to manage encryption with your own keys, you have two options. You can use either type of key management, or both:

- You can specify a *customer-managed* key to use for encrypting and decrypting data in Blob Storage and in Azure Files.Customer-managed keys must be stored in Azure Key Vault or Azure Key Vault Managed Hardware Security Model (HSM).
- You can specify a *customer-provided* key on Blob Storage operations. A client can include an encryption key on a read/write request for granular control over how blob data is encrypted and decrypted.

The following table compares key management options for Azure Storage encryption.

Key management parameter	Microsoft-managed keys	Customer-managed keys	Customer-provided keys
Encryption/decryption operations	Azure	Azure	Azure
Azure Storage services supported	All	Blob Storage, Azure Files	Blob Storage
Key storage	Microsoft key store	Azure Key Vault or Key Vault HSM	Customer's own key store
Key rotation responsibility	Microsoft	Customer	Customer
Key control	Microsoft	Customer	Customer
Key scope	Account (default), container, or blob	Account (default), container, or blob	N/A

Client-side encryption

The Azure Blob Storage client libraries for .NET, Java, and Python support encrypting data within client applications before uploading to Azure Storage, and decrypting data while downloading to the client. The Queue Storage client libraries for .NET and Python also support client-side encryption.

The Blob Storage and Queue Storage client libraries uses AES in order to encrypt user data. There are two versions of client-side encryption available in the client libraries:

- Version 2 uses Galois/Counter Mode (GCM) mode with AES. The Blob Storage and Queue Storage SDKs support client-side encryption with v2.
- Version 1 uses Cipher Block Chaining (CBC) mode with AES. The Blob Storage, Queue Storage, and Table Storage SDKs support client-side encryption with v1.

Manage the Azure Blob storage lifecycle

Introduction

Data sets have unique lifecycles. Early in the lifecycle, people access some data often. But the need for access drops drastically as the data ages. Some data stays idle in the cloud and is rarely accessed once stored.

Explore the Azure Blob storage lifecycle

Data sets have unique lifecycles. Early in the lifecycle, people access some data often. But the need for access drops drastically as the data ages. Some data stays idle in the cloud and is rarely accessed once stored. Some data expires days or months after creation, while other data sets are actively read and modified throughout their lifetimes.

Access tiers

Azure storage offers different access tiers, allowing you to store blob object data in the most cost-effective manner. Available access tiers include:

- **Hot** - An online tier optimized for storing data that is accessed frequently.
- **Cool** - An online tier optimized for storing data that is infrequently accessed and stored for a minimum of 30 days.
- **Cold tier** - An online tier optimized for storing data that is infrequently accessed and stored for a minimum of 90 days. The cold tier has lower storage costs and higher access costs compared to the cool tier.
- **Archive** - An offline tier optimized for storing data that is rarely accessed and stored for at least 180 days with flexible latency requirements, on the order of hours.

Data storage limits are set at the account level and not per access tier. You can choose to use all of your limit in one tier or across all three tiers.

Manage the data lifecycle

Azure Blob Storage lifecycle management offers a rule-based policy that you can use to transition blob data to the appropriate access tiers or to expire data at the end of the data lifecycle.

With the lifecycle management policy, you can:

- Transition blobs from cool to hot immediately when accessed, to optimize for performance.
- Transition current versions of a blob, previous versions of a blob, or blob snapshots to a cooler storage tier if these objects aren't accessed or modified for a period of time, to optimize for cost.
- Delete current versions of a blob, previous versions of a blob, or blob snapshots at the end of their lifecycles.

- Apply rules to an entire storage account, to select containers, or to a subset of blobs using name prefixes or blob index tags as filters.

Consider a scenario where data is frequently accessed during the early stages of the lifecycle, but only occasionally after two weeks. Beyond the first month, the data set is rarely accessed. In this scenario, hot storage is best during the early stages. Cool storage is most appropriate for occasional access. Archive storage is the best tier option after the data ages over a month. By moving data to the appropriate storage tier based on its age with lifecycle management policy rules, you can design the least expensive solution for your needs.

Discover Blob storage lifecycle policies

A lifecycle management policy is a collection of rules in a JSON document. Each rule definition within a policy includes a filter set and an action set. The filter set limits rule actions to a certain set of objects within a container or objects names. The action set applies the tier or delete actions to the filtered set of objects:

```
{
  "rules": [
    {
      "name": "rule1",
      "enabled": true,
      "type": "Lifecycle",
      "definition": {...}
    },
    {
      "name": "rule2",
      "type": "Lifecycle",
      "definition": {...}
    }
  ]
}
```

A policy is a collection of rules:

Parameter name	Parameter type	Notes
`rules`	An array of rule objects	At least one rule is required in a policy. You can define up to 100 rules in a policy.

Each rule within the policy has several parameters:

Parameter name	Parameter type	Notes	Required
name	String	A rule name can include up to 256 alphanumeric characters. Rule name is case-sensitive. It must be unique within a policy.	True

Parameter name	Parameter type	Notes	Required
enabled	Boolean	An optional boolean to allow a rule to be temporarily disabled. Default value is true.	False
type	An enum value	The current valid type is Lifecycle.	True
definition	An object that defines the lifecycle rule	Each definition is made up of a filter set and an action set.	True

Rules

Each rule definition includes a filter set and an action set. The filter set limits rule actions to a certain set of objects within a container or objects names. The action set applies the tier or delete actions to the filtered set of objects.

The following sample rule filters the account to run the actions on objects that exist inside `sample-container` and start with `blob1`.

- Tier blob to cool tier 30 days after last modification
- Tier blob to archive tier 90 days after last modification
- Delete blob 2,555 days (seven years) after last modification
- Delete blob snapshots 90 days after snapshot creation

```
{
  "rules": [
    {
      "enabled": true,
      "name": "sample-rule",
      "type": "Lifecycle",
      "definition": {
        "actions": {
          "baseBlob": {
            "tierToCool": {
              "daysAfterModificationGreaterThan": 30
            },
            "tierToArchive": {
              "daysAfterModificationGreaterThan": 90,
              "daysAfterLastTierChangeGreaterThan": 7
            },
            "delete": {
              "daysAfterModificationGreaterThan": 2555
            }
          },
          "snapshot": {
            "delete": {
              "daysAfterCreationGreaterThan": 90
            }
          }
```

```
          },
          "filters": {
            "blobTypes": [
              "blockBlob"
            ],
            "prefixMatch": [
              "sample-container/blob1"
            ]
          }
        }
      }
    ]
  }
```

Rule filters

Filters limit rule actions to a subset of blobs within the storage account. If more than one filter is defined, a logical AND runs on all filters. Filters include:

Filter name	Type	Is Required
blobTypes	An array of predefined enum values.	Yes
prefixMatch	An array of strings for prefixes to be match. Each rule can define up to 10 prefixes. A prefix string must start with a container name.	No
blobIndexMatch	An array of dictionary values consisting of blob index tag key and value conditions to be matched. Each rule can define up to 10 blob index tag condition.	No

Rule actions

Actions are applied to the filtered blobs when the run condition is met.

Lifecycle management supports tiering and deletion of blobs and deletion of blob snapshots. Define at least one action for each rule on blobs or blob snapshots.

Action	Current Version	Snapshot	Previous Versions
tierToCool	Supported for `blockBlob`	Supported	Supported
tierToCold	Supported for `blockBlob`	Supported	Supported
enableAutoTierToHotFromCool	Supported for `blockBlob`	Not supported	Not supported
tierToArchive	Supported for `blockBlob`	Supported	Supported
delete	Supported for `blockBlob` and `appendBlob`	Supported	Supported

> **Note**: If you define more than one action on the same blob, lifecycle management applies the least expensive action to the blob. For example, action `delete` is cheaper than action `tierToArchive`. Action `tierToArchive` is cheaper than action `tierToCool`.

The run conditions are based on age. Base blobs use the last modified time to track age, and blob snapshots use the snapshot creation time to track age.

Action run condition	Condition value	Description
daysAfterModificationGreaterThan	Integer value indicating the age in days	The condition for base blob actions
daysAfterCreationGreaterThan	Integer value indicating the age in days	The condition for blob snapshot actions
daysAfterLastAccessTimeGreaterThan	Integer value indicating the age in days	The condition for a current version of a blob when access tracking is enabled
daysAfterLastTierChangeGreaterThan	Integer value indicating the age in days after last blob tier change time	The minimum duration in days that a rehydrated blob is kept in hot, cool, or cold tiers before being returned to the archive tier. This condition applies only to `tierToArchive` actions.

Implement Blob storage lifecycle policies

You can add, edit, or remove a policy by using any of the following methods:

- Azure portal
- Azure PowerShell
- Azure CLI
- REST APIs

The following are the steps and some examples for the Portal and Azure CLI.

Azure portal

There are two ways to add a policy through the Azure portal: Azure portal List view, and Azure portal Code view. Following is an example of how to add a policy in the Azure portal Code view.

Azure portal Code view

1. In the Azure portal, navigate to your storage account.
2. Under **Data management**, select **Lifecycle Management** to view or change lifecycle management policies.

3. Select the **Code View** tab. On this tab, you can define a lifecycle management policy in JSON.

The following JSON is an example of a policy that moves a block blob whose name begins with *log* to the cool tier if it has been more than 30 days since the blob was modified.

```
{
  "rules": [
    {
      "enabled": true,
      "name": "move-to-cool",
      "type": "Lifecycle",
      "definition": {
        "actions": {
          "baseBlob": {
            "tierToCool": {
              "daysAfterModificationGreaterThan": 30
            }
          }
        },
        "filters": {
          "blobTypes": [
            "blockBlob"
          ],
          "prefixMatch": [
            "sample-container/log"
          ]
        }
      }
    }
  ]
}
```

Azure CLI

To add a lifecycle management policy with Azure CLI, write the policy to a JSON file, then call the `az storage account management-policy create` command to create the policy.

```
az storage account management-policy create \
    --account-name <storage-account> \
    --policy @policy.json \
    --resource-group <resource-group>
```

A lifecycle management policy must be read or written in full. Partial updates aren't supported.

Rehydrate blob data from the archive tier

While a blob is in the archive access tier, it's considered to be offline and can't be read or modified. In order to read or modify data in an archived blob, you must first rehydrate the blob to an online tier, either the hot

or cool tier. There are two options for rehydrating a blob that is stored in the archive tier:

- **Copy an archived blob to an online tier**: You can rehydrate an archived blob by copying it to a new blob in the hot or cool tier with the Copy Blob or Copy Blob from URL operation. Microsoft recommends this option for most scenarios.
- **Change a blob's access tier to an online tier**: You can rehydrate an archived blob to hot or cool by changing its tier using the Set Blob Tier operation.

Rehydrating a blob from the archive tier can take several hours to complete. Microsoft recommends rehydrating larger blobs for optimal performance. Rehydrating several small blobs concurrently might require extra time.

Rehydration priority

When you rehydrate a blob, you can set the priority for the rehydration operation via the optional `x-ms-rehydrate-priority` header on a Set Blob Tier or **Copy Blob/Copy Blob From URL** operation. Rehydration priority options include:

- **Standard priority**: The rehydration request is processed in the order it was received and might take up to 15 hours.
- **High priority**: The rehydration request is prioritized over standard priority requests and might complete in under one hour for objects under 10 GB in size.

To check the rehydration priority while the rehydration operation is underway, call Get Blob Properties to return the value of the `x-ms-rehydrate-priority` header. The rehydration priority property returns either *Standard* or *High*.

Copy an archived blob to an online tier

The first option for moving a blob from the archive tier to an online tier is to copy the archived blob to a new destination blob that is in either the hot or cool tier. You can use the Copy Blob operation to copy the blob. When you copy an archived blob to a new blob in an online tier, the source blob remains unmodified in the archive tier. You must copy the archived blob to a new blob with a different name or to a different container. You can't overwrite the source blob by copying to the same blob.

Rehydrating an archived blob by copying it to an online destination tier is supported within the same storage account only for service versions earlier than 2021-02-12. Beginning with service version 2021-02-12, you can rehydrate an archived blob by copying it to a different storage account, as long as the destination account is in the same region as the source account.

Change a blob's access tier to an online tier

The second option for rehydrating a blob from the archive tier to an online tier is to change the blob's tier by calling **Set Blob Tier**. With this operation, you can change the tier of the archived blob to either hot or cool.

Once a **Set Blob Tier** request is initiated, it can't be canceled. During the rehydration operation, the blob's access tier setting continues to show as archived until the rehydration process is complete.

To learn how to rehydrate a blob by changing its tier to an online tier, see Rehydrate a blob by changing its tier.

> **Caution**: Changing a blob's tier doesn't affect its last modified time. If there is a lifecycle management policy in effect for the storage account, then rehydrating a blob with **Set Blob Tier** can result in a scenario where the lifecycle policy moves the blob back to the archive tier after rehydration because the last modified time is beyond the threshold set for the policy.

Work with Azure Blob storage

Introduction

The Azure Storage client libraries for .NET offer a convenient interface for making calls to Azure Storage.

Explore Azure Blob storage client library

The Azure Storage client libraries for Python provide a convenient interface for accessing Azure Storage. The latest version of the Azure Storage client library for Python is 12.x, and Microsoft recommends using version 12.x for new Python applications.

The following table lists the key Python classes and a brief description:

Class	Description
`BlobClient`	The `BlobClient` class lets you manipulate Azure Storage blobs.
`BlobClientOptions`	Provides client configuration options for connecting to Azure Blob Storage (exposed as keyword arguments in client constructors).
`ContainerClient`	The `ContainerClient` class allows you to manipulate Azure Storage containers and their blobs.
`BlobServiceClient`	The `BlobServiceClient` class enables you to manipulate Azure Storage service resources and blob containers. The storage account acts as the top-level namespace for the Blob service.
`BlobUriBuilder`	The `BlobUriBuilder` type provides a convenient way to build or modify a URI for an Azure Storage resource (available as `BlobClient.from_blob_url` and similar helpers in Python).

The following Python packages provide the necessary classes for working with Blob Storage:

- `azure.storage.blob`: Contains the primary classes (client objects) needed to operate on the service, containers, and blobs.
- `azure.storage.blob.specialized`: Includes classes for operations specific to a blob type, such as block blobs (as in `BlockBlobClient`, `AppendBlobClient`, etc.).
- `azure.storage.blob._models`: Houses utility classes, structures, and enumerations for blob operations (exposed under the main package as models).

Create Blob storage resources with the Python client library

In this exercise, you create an Azure Storage account and build a Python application using the Azure Storage Blob client library to create containers, upload files to blob storage, list blobs, and download files. You learn how to authenticate with Azure, perform blob storage operations programmatically, and verify results in the Azure portal.

Create an Azure Storage account

In this section of the exercise you create the needed resources in Azure with the Azure CLI.

1. In your browser navigate to the Azure portal https://portal.azure.com; signing in with your Azure credentials if prompted.

2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new cloud shell in the Azure portal, selecting a ***Bash*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal. If you are prompted to select a storage account to persist your files, select **No storage account required**, your subscription, and then select **Apply**.

   > **Note**: If you have previously created a cloud shell that uses a *PowerShell* environment, switch it to ***Bash***.

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

4. Create a resource group for the resources needed for this exercise. Replace **myResourceGroup** with a name you want to use for the resource group. You can replace **eastus2** with a region near you if needed. If you already have a resource group you want to use, proceed to the next step.

   ```
   az group create --location eastus2 --name myResourceGroup
   ```

5. Many of the commands require unique names and use the same parameters. Creating some variables will reduce the changes needed to the commands that create resources. Run the following commands to create the needed variables. Replace **myResourceGroup** with the name you're using for this exercise.

   ```
   resourceGroup=myResourceGroup
   location=eastus
   accountName=storageacct$RANDOM
   ```

6. Run the following commands to create the Azure Storage account, each account name must be unique. The first command creates a variable with a unique name for your storage account. Record the name of your account from the output of the **echo** command.

   ```
   az storage account create --name $accountName \
       --resource-group $resourceGroup \
       --location $location \
       --sku Standard_LRS
   ```

```
echo $accountName
```

Assign a role to your Microsoft Entra user name

To allow your app to create resources and items, assign your Microsoft Entra user to the **Storage Blob Data Owner** role. Perform the following steps in the cloud shell.

> **Tip:** Resize the cloud shell to display more information, and code, by dragging the top border. You can also use the minimize and maximize buttons to switch between the cloud shell and the main portal interface.

1. Run the following command to retrieve the **userPrincipalName** from your account. This represents who the role will be assigned to.

```
userPrincipal=$(az rest --method GET --url
https://graph.microsoft.com/v1.0/me \
    --headers 'Content-Type=application/json' \
    --query userPrincipalName --output tsv)
```

2. Run the following command to retrieve the resource ID of the storage account. The resource ID sets the scope for the role assignment to a specific namespace.

```
resourceID=$(az storage account show --name $accountName \
    --resource-group $resourceGroup \
    --query id --output tsv)
```

3. Run the following command to create and assign the **Storage Blob Data Owner** role. This role gives you the permissions to manage containers and items.

```
az role assignment create --assignee $userPrincipal \
    --role "Storage Blob Data Owner" \
    --scope $resourceID
```

Create a Python app to create containers and blobs

Now that the needed resources are deployed to Azure, the next step is to set up your Python project. The following steps are performed in your terminal or cloud shell.

1. Create a directory for your project and switch to it

```
mkdir azstor
cd azstor
```

2. (Optional but recommended) Create and activate a virtual environment

```
python -m venv .venv
source .venv/bin/activate  # on macOS/Linux
.venv\Scripts\activate     # on Windows
```

3. Install the required packages

```
pip install azure-storage-blob azure-identity
```

4. Create a data folder in your project

```
mkdir data
```

Now it's time to add the code for the project.

Add the starter code for the project

1. Run the following command in the cloud shell to begin editing the application.

```
code myscript.py
```

2. Replace any existing contents with the following code. Be sure to review the comments in the code.

```
import os
import uuid
from azure.identity import DefaultAzureCredential
from azure.storage.blob import BlobServiceClient, BlobClient

print("Azure Blob Storage exercise\n")

# Create a DefaultAzureCredential with options to exclude specific
cred types
credential = DefaultAzureCredential(
    exclude_environment_credential=True,
    exclude_managed_identity_credential=True
)

def main():
    # CREATE A BLOB STORAGE CLIENT
```

```
    # CREATE A CONTAINER


    # CREATE A LOCAL FILE FOR UPLOAD TO BLOB STORAGE


    # UPLOAD THE FILE TO BLOB STORAGE


    # LIST BLOBS IN THE CONTAINER


    # DOWNLOAD THE BLOB TO A LOCAL FILE


if __name__ == "__main__":
    main()
    print("\nPress enter to exit the sample application.")
    input()
```

Add code to complete the project

Throughout the rest of the exercise you add code in specified areas to create the full application.

1. Locate the `# CREATE A BLOB STORAGE CLIENT` comment, then add the following code directly beneath the comment.

```
    # Create a credential using DefaultAzureCredential with configured
options
    account_name = "YOUR_ACCOUNT_NAME"  # Replace with your storage
account name

    # Create the BlobServiceClient using the endpoint and
DefaultAzureCredential
    blob_service_endpoint =
f"https://{account_name}.blob.core.windows.net"
    blob_service_client = BlobServiceClient(blob_service_endpoint,
credential=credential)
```

2. Locate the `# CREATE A CONTAINER` comment, then add the following code directly beneath the comment.

```
    # Create a unique name for the container
    container_name = "wtblob" + str(uuid.uuid4())
```

```
    # Create the container and get a container client object
    print("Creating container:", container_name)
    try:
        container_client =
blob_service_client.create_container(container_name)
        print("Container created successfully, press 'Enter' to
continue.")
        input()
    except Exception as e:
        print("Failed to create the container, exiting program.",
str(e))
        return
```

3. Find the `# CREATE A LOCAL FILE FOR UPLOAD TO BLOB STORAGE` comment, then add the following code directly beneath the comment.

```
    # Create a local file in the ./data/ directory for uploading and
downloading
    print("Creating a local file for upload to Blob storage...")
    local_path = "./data/"
    os.makedirs(local_path, exist_ok=True)
    file_name = "wtfile" + str(uuid.uuid4()) + ".txt"
    local_file_path = os.path.join(local_path, file_name)

    with open(local_file_path, "w") as f:
        f.write("Hello, World!")
    print("Local file created, press 'Enter' to continue.")
    input()
```

4. Locate the `# UPLOAD THE FILE TO BLOB STORAGE` comment, then add the following code directly beneath the comment.

```
    # Get a reference to the blob and upload the file
    blob_client = container_client.get_blob_client(file_name)

    print(f"Uploading to Blob storage as blob:\n\t {blob_client.url}")

    with open(local_file_path, "rb") as
        blob_client.upload_blob(data)

    # Verify if the file was uploaded successfully
    try:
        blob_client.get_blob_properties()
        print("File uploaded successfully, press 'Enter' to
continue.")
        input()
    except Exception:
```

```
            print("File upload failed, exiting program..")
            return
```

5. Locate the `# LIST BLOBS IN THE CONTAINER` comment, then add the following code directly beneath the comment.

```
    print("Listing blobs in container...")
    for blob_item in container_client.list_blobs():
        print("\t" + blob_item.name)
    print("Press 'Enter' to continue.")
    input()
```

6. Locate the `# DOWNLOAD THE BLOB TO A LOCAL FILE` comment, then add the following code directly beneath the comment.

```
    # Adds the string "DOWNLOADED" before the .txt extension so it
doesn't overwrite the original file
    download_file_path = local_file_path.replace(".txt",
"DOWNLOADED.txt")
    print(f"Downloading blob to: {download_file_path}")

    with open(download_file_path, "wb") as download_file:
        download_stream = blob_client.download_blob()
        download_file.write(download_stream.readall())

    print(f"Blob downloaded successfully to: {download_file_path}")
```

Sign into Azure and run the app

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

```
az login
```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the --*tenant* parameter. See Sign into Azure interactively using Azure CLI for details.

2. Run the following command to start the console app. The app will pause many times during execution waiting for you to press any key to continue. This gives you an opportunity to view the messages in the Azure portal.

```
python run
```

3. In the Azure portal, navigate to the Azure Storage account you created.
4. Expand **> Data storage** in the left navigation and select **Containers**.
5. Select the container the application created and you can view the blob that was uploaded.
6. Run the two commands below to change into the **data** directory and list the files that were uploaded and downloaded.

```
cd data
ls
```

Clean up resources

Now that you finished the exercise, you should delete the cloud resources you created to avoid unnecessary resource usage.

1. In your browser navigate to the Azure portal https://portal.azure.com; signing in with your Azure credentials if prompted.
2. Navigate to the resource group you created and view the contents of the resources used in this exercise.
3. On the toolbar, select **Delete resource group**.
4. Enter the resource group name and confirm that you want to delete it.

> **CAUTION:** Deleting a resource group deletes all resources contained within it. If you chose an existing resource group for this exercise, any existing resources outside the scope of this exercise will also be deleted.

Manage container properties and metadata by using Python

Blob containers support system properties and user-defined metadata, in addition to the data they contain.

- **System properties:** System properties exist on each Blob storage resource. Some of them can be read or set, while others are read-only. Under the covers, some system properties correspond to standard HTTP headers, and the Azure Storage client library for Python manages these for you.
- **User-defined meta** Metadata is a set of name-value pairs that you specify. These can store values for your own purposes, do not affect behavior, and should follow HTTP header conventions—use only ASCII, treat as case-insensitive, and encode non-ASCII values.

Retrieve container properties

To retrieve container properties, call the following methods of the `ContainerClient` class (Python equivalent of `BlobContainerClient`):

- `get_container_properties()`

The following code example fetches a container's system properties and writes some property values to the console:

```
from azure.storage.blob import ContainerClient
from azure.core.exceptions import ResourceNotFoundError, AzureError

def read_container_properties(container_client: ContainerClient):
    try:
        # Fetch some container properties and write out their values.
        properties = container_client.get_container_properties()
        print(f"Properties for container {container_client.url}")
        print(f"Public access level: {properties['public_access']}")
        print(f"Last modified time in UTC: {properties['last_modified']}")
    except AzureError as e:
        print(f"HTTP error code {getattr(e, 'status_code', '')}:
{getattr(e, 'error_code', '')}")
        print(str(e))
        input()
```

Set and retrieve metadata

You can specify metadata as one or more name-value pairs on a blob or container resource. To set metadata, add name-value pairs to a dictionary object, and then call the following methods of the ContainerClient class:

- `set_container_metadata()`

The following code example sets metadata on a container:

```
def add_container_metadata(container_client: ContainerClient):
    try:
        metadata = {}
        # Add some metadata to the container.
        metadata["docType"] = "textDocuments"
        metadata["category"] = "guidance"

        # Set the container's metadata.
        container_client.set_container_metadata(metadata=metadata)
    except AzureError as e:
        print(f"HTTP error code {getattr(e, 'status_code', '')}:
{getattr(e, 'error_code', '')}")
        print(str(e))
        input()
```

You use `get_container_properties()` to retrieve metadata in addition to other properties.

The following code example retrieves metadata from a container:

```
def read_container_metadata(container_client: ContainerClient):
    try:
        properties = container_client.get_container_properties()

        # Enumerate the container's metadata.
        print("Container meta")
        metadata = properties.get('metadata', {})
        for key, value in metadata.items():
            print(f"\tKey: {key}")
            print(f"\tValue: {value}")
    except AzureError as e:
        print(f"HTTP error code {getattr(e, 'status_code', '')}:
{getattr(e, 'error_code', '')}")
        print(str(e))
        input()
```

Set and retrieve properties and metadata for blob resources by using REST

Containers and blobs support custom metadata, represented as HTTP headers. Metadata headers can be set on a request that creates a new container or blob resource, or on a request that explicitly creates a property on an existing resource.

Metadata header format

Metadata headers are name/value pairs. The format for the header is:

```
x-ms-meta-name:string-value
```

Beginning with version 2009-09-19, metadata names must adhere to the naming rules for C# identifiers.

Names are case-insensitive. Metadata names preserve the case with which they were created, but are case-insensitive when set or read. If two or more metadata headers with the same name are submitted for a resource, the Blob service returns status code `400 (Bad Request)`.

The metadata consists of name/value pairs. The total size of all metadata pairs can be up to 8 KB in size.

Metadata name/value pairs are valid HTTP headers, and so they adhere to all restrictions governing HTTP headers.

Operations on metadata

Metadata on a blob or container resource can be retrieved or set directly, without returning or altering the content of the resource.

Metadata values can only be read or written in full; partial updates aren't supported. Setting metadata on a resource overwrites any existing metadata values for that resource.

Retrieving properties and metadata

The GET/HEAD operation retrieves metadata headers for the specified container or blob. These operations return headers only; they don't return a response body. The URI syntax for retrieving metadata headers on a container is as follows:

```
GET/HEAD https://myaccount.blob.core.windows.net/mycontainer?
restype=metadata
```

The URI syntax for retrieving metadata headers on a blob is as follows:

```
GET/HEAD https://myaccount.blob.core.windows.net/mycontainer/myblob?
comp=metadata
```

Setting Metadata Headers

The PUT operation sets metadata headers on the specified container or blob, overwriting any existing metadata on the resource. Calling PUT without any headers on the request clears all existing metadata on the resource.

The URI syntax for setting metadata headers on a container is as follows:

```
PUT https://myaccount.blob.core.windows.net/mycontainer?
comp=metadata&restype=container
```

The URI syntax for setting metadata headers on a blob is as follows:

```
PUT https://myaccount.blob.core.windows.net/mycontainer/myblob?
comp=metadata
```

Standard HTTP properties for containers and blobs

Containers and blobs also support certain standard HTTP properties. Properties and metadata are both represented as standard HTTP headers; the difference between them is in the naming of the headers. Metadata headers are named with the header prefix `x-ms-meta-` and a custom name. Property headers use standard HTTP header names, as specified in the Header Field Definitions section 14 of the HTTP/1.1 protocol specification.

The standard HTTP headers supported on containers include:

- `ETag`
- `Last-Modified`

The standard HTTP headers supported on blobs include:

- `ETag`
- `Last-Modified`
- `Content-Length`
- `Content-Type`
- `Content-MD5`
- `Content-Encoding`
- `Content-Language`
- `Cache-Control`
- `Origin`
- `Range