



## 3.0 Introduction

Almost every solution needs to persist data in a data store, and your cloud solutions are no exception. Your data store should be protected by multiple security measures, to safeguard the data from hackers and unauthorized users. Data redundancy should be in place in case one, or more data centers go offline due to expected, or unexpected events. Scalability is also a key requirement for modern applications. The data store should accommodate any traffic volume, and expand to store more data as they appear. Microsoft offers *Azure Storage Accounts* to meet your data storage needs at a reasonably low price point. 

Azure Storage Accounts remain Microsoft Azure's main general purpose data storage service. It offers native integration with key Azure service such as *Azure Active Directory*, *Azure Key Vault*, *Managed Identities*, *Azure Functions*, *Logic Apps*, *Virtual Machine Disks*, *Azure Cognitive Search*, *Event Grids*, and *Azure Synapse Analytics*. The infinite scaling capability makes it a great choice for storing big data, telemetry, log files, images, media files, and unstructured, and NoSQL data.

You have multiple options to migrate your on-premises data to Azure Storage. Use *Azure Data Box* (<https://azure.microsoft.com/en-us/services/databox/>) service to move large amounts of local data to your Azure Storage Account. You also have the option to directly upload files using the portal, CLI, PowerShell, or command line tools such as AzCopy (<https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azcopy-v10>). Use Azure Storage to: 

- Automatically invoke an Azure Function upon a new file arrival at your Azure Storage

- Index your blob files and build content search APIs for them using Azure Cognitive Search
- Store big data to be analyzed by *Azure Databricks* or *Azure Synapse Analytics*
- Use *Azure Files* to implement traditional *SMB file shares* (<https://docs.microsoft.com/en-us/windows/win32/fileio/microsoft-smb-protocol-and-cifs-protocol-overview>) for your *Windows* or *Linux* workstations

Azure Storage Accounts offer the following services: *blobs*, *files*, *queues*, *tables*, and *disks*. VM Disks are stored as blob object.

In this chapter you will use the key Azure Storage Account features to configure a secure, reliable, and affordable data store in your Azure subscription.



Microsoft is continuously updating Azure services to offer richer features. Visit the *Azure Updates* portal (<https://azure.microsoft.com/en-ca/updates/>) for new updates, and take advantage of the new offerings as they become generally available.

## Workstation Configuration

You will need to prepare your workstation before starting on the book recipes.

### Setting Up CLI

Follow the “General workstation setup steps for CLI recipes” on page ??? to setup your machine to run Azure CLI commands. Then, you can clone the following code repository:

```
git clone https://github.com/AzureCookbook/Storage
```

## 3.1 Using Azure Key Vault Keys to Configure Azure Storage Account Encryption at Rest

### Problem

You need to use CMK (customer-managed keys) to encrypt your Azure Storage Account data at rest.

### Solution

Create a new encryption key in Azure Key Vault and configure Azure Storage Account to use it instead of the default Microsoft-managed keys. (See Figure 3-1)

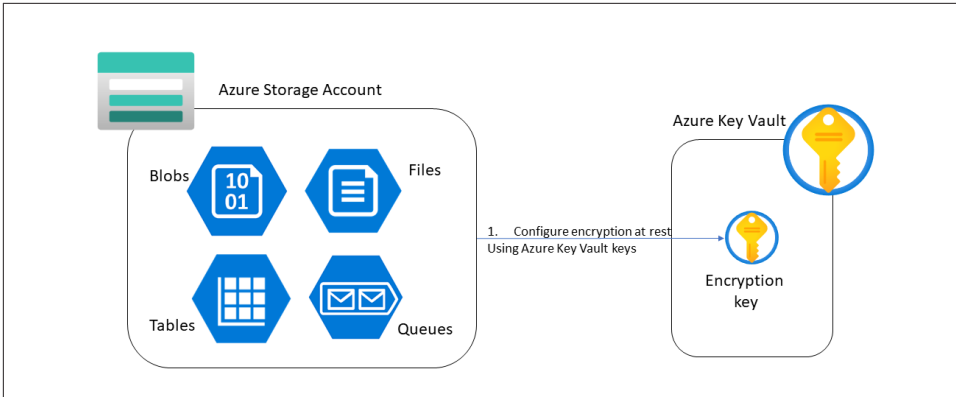


Figure 3-1. Use Azure Key Vault Encryption Keys to Configure Azure Storage Encryption at Rest

## Steps

1. Login to your Azure Subscription using a user with the *Owner* role.
2. Create a new Azure Key Vault. You can also use an existing Key Vault. Replace the *{resource-group-id}* and *{kv-name}* placeholders with the desired resource group, and Key Vault names.

```
RG_NAME="{resource-group-id}"
KV_NAME="{kv-name}"
```

```
az keyvault create --name $KV_NAME \
  --resource-group $RG_NAME \
  --location eastus
  --enable-purge-protection
```

3. You need to use an encryption key for configure Azure Storage encryption. Let's create a new key in our Key Vault resource:

```
az keyvault key create \
  --name storage-cmk-key
  --vault-name $KV_NAME
  --key RSA
  --size 4096
```



An encryption key has several properties which you can change based on your scenario. For example you can create hardware-protected keys if you are in the premium Key Vault tier. The key size, curve, and type can also be configured. See Key Vault documentation (<https://docs.microsoft.com/en-us/cli/azure/keyvault/key?view=azure-cli-latest#az-keyvault-key-create>) for details.

4. Now create an Azure Storage Account to test your encryption key using the following command:

```
STORAGE_NAME="{storage_account_name}"

az storage account create --name $STORAGE_NAME \
  --resource-group $RG_NAME --location eastus \
  --sku Standard_LRS
```

5. So far, you already created your encryption key. It is time to configure your Storage Account to use this key for encryption at rest (<https://docs.microsoft.com/en-us/azure/storage/common/customer-managed-keys-overview?toc=%2Fazure%2Fstorage%2Fblobs%2Ftoc.json>). In order to access the encryption key from Key Vault, your Storage Account need to have *wrapkey*, *unwrapkey*, and *get* access policy set on the Key Vault. First you need to assign an identity to your Storage Account. The access policy can then be set to this identity. Store the identity id in a variable as follows:

```
az storage account update \
  --name $STORAGE_NAME \
  --resource-group $RG_NAME \
  --assign-identity

STORAGE_IDENTITY_OBJECTID=$(az storage account show \
  --name $STORAGE_NAME \
  --query "identity.principalId" \
  --output tsv)
```

6. Now let's configure mentioned access policy (<https://docs.microsoft.com/en-us/azure/key-vault/general/security-features#key-vault-authentication-options>) in the Key Vault:

```
az keyvault set-policy --name $KV_NAME \
  --object-id $STORAGE_IDENTITY_OBJECTID \
  --key-permissions get unwrapKey wrapKey
```



You are logged in as the subscription's *owner* account. That's why you are able to set a Key Vault access policy.

7. Now the stage is set for you to configure your Storage Account to use your encryption key for data at rest:

```
KEYVAULT_URI=$(az keyvault show \
  --name $KV_NAME \
  --resource-group $RG_NAME \
  --query properties.vaultUri \
  --output tsv)

az storage account update \
```

```
--name $STORAGE_NAME \  
--resource-group $RG_NAME \  
--encryption-key-source Microsoft.Keyvault \  
--encryption-services blob \  
--encryption-key-vault $KEYVAULT_URI \  
--encryption-key-name storage-cmk-key
```

Wait for the command to succeed. You configured your Azure Storage Account to use your own managed key instead of the default Microsoft managed key. See Azure documentation (<https://docs.microsoft.com/en-us/azure/storage/common/customer-managed-keys-configure-existing-account?tabs=azure-cli>) for details.

## Discussion

Encryption for data at rest is automatically enabled for all the Azure Storage Account instances (<https://docs.microsoft.com/en-us/azure/storage/common/storage-service-encryption>). By default, the encryption key is managed by Microsoft.

However, due to several security standards, commitments, or for compliance reasons, your organization might want to be in charge of the Storage Account encryption keys by creating their own keys (CMK keys). You can use this recipe to create your own Key Vault encryption key. This key is stored in the Azure Key Vault and you are responsible to manage and maintain its health and availability. If you don't have the above commitments, it is perfectly fine to leave the default Storage Account encryption behavior to use Microsoft-managed keys. This approach has less administrative overhead for your team.

Keep in mind that Azure Storage queues and tables are not automatically protected by a customer-managed key when customer-managed keys are enabled for your Storage Account. If needed, you can configure these services to be included in the CMK encryption at the time you create your Storage Account. See Azure documentation (<https://docs.microsoft.com/en-us/azure/storage/common/customer-managed-keys-overview#customer-managed-keys-for-queues-and-tables>) for details.



Loosing your CMK key will result in loosing your Azure Storage data. Make sure the soft-delete and purge-protection (<https://docs.microsoft.com/en-us/azure/key-vault/general/soft-delete-overview>) are enabled on your Key Vault and the key expiry date and auto-rotation settings (<https://docs.microsoft.com/en-us/azure/key-vault/keys/how-to-configure-key-rotation>) are configured as expected.

## 3.2 Controlling Azure Storage Account Network Access

### Problem

You need to limit Azure Storage Account network access to a single or multiple Azure VNet(s)

### Solution

Make sure Public Network Access setting is enabled for your Storage Account. Then define a network access rule to whitelist traffic from one or more Azure VNets. (See Figure 3-2)

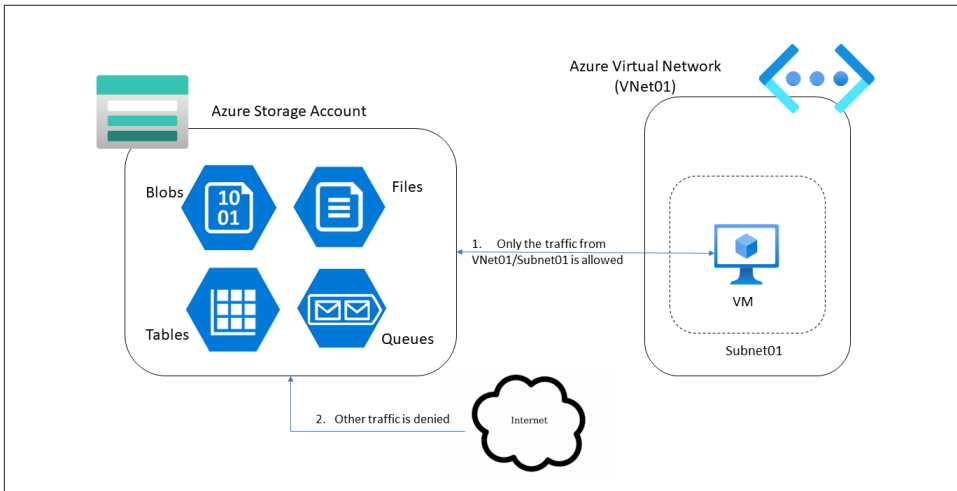


Figure 3-2. Only Allow Traffic from an Azure VNet to reach Azure Storage Account

### Steps

1. Login to your Azure Subscription using a user with the *Owner* role.
2. Create a new Azure Storage Account and make sure the *--public-network-access* setting is set to *enabled*. This ensures that traffic from public networks (including Azure VNet traffic) is allowed. You also need to set the *--default-action* parameter *Deny*. This is going to deny any traffic unless it is allowed in the *virtual network rules*. Use the following CLI command to provision your Storage Account:

```
RG_NAME="{resource-group-id}"  
STORAGE_NAME="{storage_account_name}"
```

```
az storage account create \  
  --name $STORAGE_NAME \  
  --resource-group $RG_NAME \  
  --public-network-access enabled \  
  --default-action Deny
```

```
--location eastus \
--sku Standard_LRS \
--default-action Deny \
--public-network-access Enabled
```

3. Wait for the command to succeed. Now let's create a new Azure Virtual Network (VNet). We will allow traffic from any resource in this VNet's *Subnet01* (for example VMs). You can also use an existing VNet. Use the following command to create your net VNet with a single child subnet:

```
az network vnet create --resource-group $RG_NAME \
--name VNet01 --address-prefix 10.0.0.0/16 \
--subnet-name Subnet01 --subnet-prefix 10.0.0.0/26
```

4. You need to allow the egress (outgoing) traffic from your subnet to Azure Storage. You can achieve this by adding a *Microsoft.Storage* service endpoint (<https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-service-endpoints-overview>) to your subnet:

```
az network vnet subnet update \
--resource-group $RG_NAME \
--vnet-name VNet01 \
--name Subnet01 \
--service-endpoints "Microsoft.Storage"
```

5. Let's grab the subnet id and store it in a variable:

```
subnetid=$(az network vnet subnet show \
--resource-group $RG_NAME \
--vnet-name VNet01 \
--name Subnet01 \
--query id \
--output tsv)
```

6. Now let's add a storage network rule to allow ingress (incoming) traffic from *VNet01/subnet01* to your Storage Account:

```
az storage account network-rule add \
--resource-group $RG_NAME \
--account-name $STORAGE_NAME \
--vnet-name VNet01 \
--subnet Subnet01 \
--action Allow
```

7. You can see all storage network rules for your Azure Storage Account using the following command:

```
az storage account network-rule list \
--account-name $STORAGE_NAME \
--resource-group $RG_NAME
```

You configured your Azure Storage Account to accept traffic only from one Azure Virtual Network subnet. Any other request from the internet, or other networks will fail reaching the Storage Account.

## Discussion

There are three different network access scenarios for an Azure Storage Account.

1. Allow traffic from all public and private networks (including the internet) to reach your Storage Account.
2. Allow traffic only from specific Azure VNets
3. Allow traffic only from Private Endpoints with no public network access

The first option is the default behavior when you create a new Storage Account, unless you explicitly configure other network access settings.

In this recipe we covered the second option which allows traffic from one or more VNets. Keep in mind that in this case the traffic still goes through the public internet.

If you need your traffic to only go through Azure's private backbone network, you need to choose option three and configure Private Endpoints (<https://docs.microsoft.com/en-us/azure/private-link/private-endpoint-overview>) for Azure Storage Account. Check Azure Storage Account documentation for details (<https://docs.microsoft.com/en-us/azure/storage/common/storage-network-security>).

## 3.3 Granting Limited Access to Azure Storage Accounts Using SAS Tokens

### Problem

You need a way to grant granular access to Azure Storage Account services

### Solution

Instead of Azure Storage Keys which grant full access, generate Shared Access Signature (SAS) tokens (<https://docs.microsoft.com/en-us/azure/storage/common/storage-sas-overview>) with the desired access and share that with the client applications/users. (See Figure 3-3)



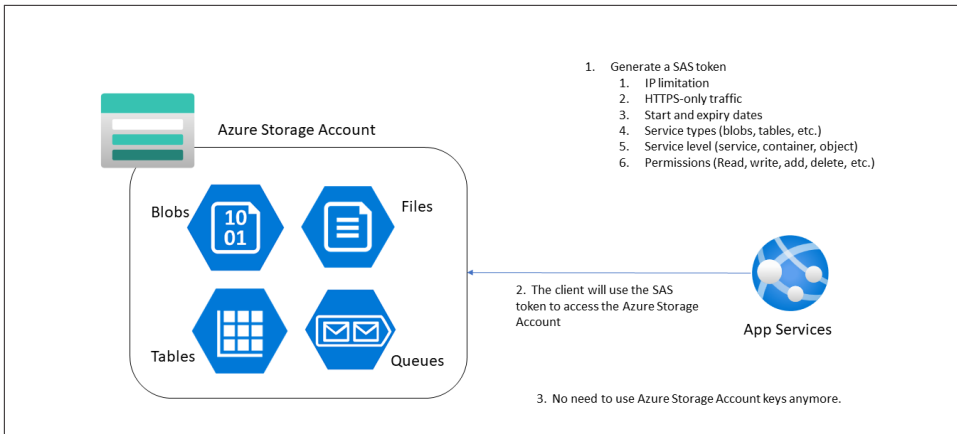


Figure 3-3. Grant Limited Access to Azure Storage Account Using SAS Tokens

## Steps

1. Login to your Azure Subscription using a user with the *Owner* role.
2. Use the following CLI command to provision a new Azure Storage Account:

```
RG_NAME="{resource-group-id}"
STORAGE_NAME="{storage_account_name}"
```

```
az storage account create \
  --name $STORAGE_NAME \
  --resource-group $RG_NAME \
  --location eastus \
  --sku Standard_LRS
```



3. Wait for the command to succeed. Imagine an Azure App Service needs access to this Storage Account. For example, grant *read* access to the *blob* and *table* services, and reject any unsecured HTTP (none HTTPS) traffic. You need this token to expire in 15 minutes. First let's calculate the expiry date for the SAS token using this command:

```
expiryDate=`date -u -d "15 minutes" '+%Y-%m-%dT%H:%MZ'`
```

4. The Storage Account key is also needed to authenticate the the SAS generation command. We only need one of the Storage Account keys. Use the following command to store key01 in a variable:

```
storageKey=$(az storage account keys list \
  --resource-group $RG_NAME \
  --account-name $STORAGE_NAME \
  --query [0].value \
  --output tsv)
```

5. Now you can use the following command to create the SAS token:

```

sasToken=$(az storage account generate-sas \
  --account-name $STORAGE_NAME \
  --account-key $storageKey \
  --expiry $expiryDate \
  --permissions r \
  --resource-types co \
  --services bq \
  --https-only \
  --output tsv)

```



For `--permissions` you can combine the following values (*a*)dd, (*c*)reate, (*d*)elete, (*f*)ilter\_by\_tags, (*i*)set\_immutability\_policy, (*l*)ist, (*p*)rocess, (*r*)ead, (*t*)ag, (*u*)pdate, (*w*)rite, (*x*)delete\_previous\_version, (*y*)permanent\_delete. For instance *rw* means read and write access. For `--services` you can combine the following values (*b*)lob, (*f*)ile, (*q*)ueue, (*t*)able. See Azure CLI documentation (<https://docs.microsoft.com/en-us/cli/azure/storage/account?view=azure-cli-latest#az-storage-account-generate-sas>) for details.

6. You will not be able to recover this SAS token in future so let's store it in a variable (or copy it from the command output). You can grab it using the following command:

```
echo $sasToken
```

You successfully generated a SAS token with the desired limited access. Provide the Storage Account clients (users, Function Apps, App Service web apps, etc.) with this token so they can use the Storage Account within the defined access.

## Discussion



The *Principle of least privilege* ([https://en.wikipedia.org/wiki/Principle\\_of\\_least\\_privilege](https://en.wikipedia.org/wiki/Principle_of_least_privilege)) requires every service/user having only the minimum permissions it needs. If your application is meant to only read data, it should not have write permissions. SAS tokens are the perfect tool to implement the principle of least privilege for Azure Storage Accounts.

SAS tokens give you great flexibility in granting Azure Storage Account access. Use your client's access requirements to design a SAS token and use the token instead of Azure Storage Account keys which have full access to every Storage Account service.

SAS tokens allow the following granular access to be set:

- Start and expiry dates
- Require HTTPS-only traffic

- Choose which storage services to access (blobs, tables, queues, files)
- Choose which resource level to access (service, container, object)
- Allowed client IP address

Refer to the Azure Storage Account documentation for details on generating SAS tokens using CLI (<https://docs.microsoft.com/en-us/cli/azure/storage/account?view=azure-cli-latest#az-storage-account-generate-sas>).

This is a sample of a generated SAS token:

```
se=2022-08-30T19%3A05Z&sp=r&spr=https&
sv=2021-06-08&ss=qb&srt=oc&
sig=E5/XvKCnUd30cRwXNS9cD9Lb4P9g5/W7NVYhGMolEvw%3D
```




SAS tokens are signed by one of the Storage Account keys. Rotating/resetting said key will result in all related SAS tokens to invalidate.

## 3.4 Granting Azure Function Apps Access to Azure Storage Account Using Managed Identity and RBAC

### Problem

You need to grant Storage Account read and write access to an Azure Function App without using the Storage Account Key, or SAS tokens

### Solution

Enable managed identity (<https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/overview>) for the Azure Function App and assign the **Storage Blob Data Contributor** built-in RBAC role to it for the Storage Account scope. 

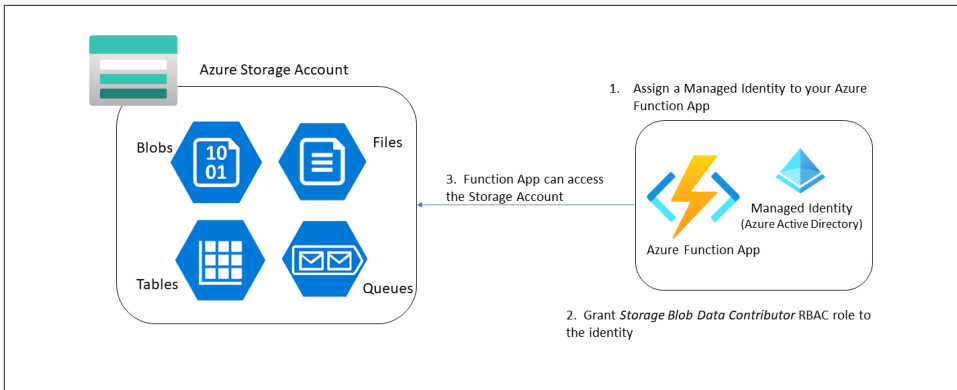


Figure 3-4. Grant Azure Storage Account Read and Write Data Access to Azure Function App Using RBAC and Managed Identities

## Steps

1. Login to your Azure Subscription using a user with the *Owner* role.
2. Use the following CLI command to provision a new Azure Storage Account:
 

```
RG_NAME="{resource-group-id}"
STORAGE_NAME="{storage_account_name}"

az storage account create \
  --name $STORAGE_NAME \
  --resource-group $RG_NAME \
  --location eastus \
  --sku Standard_LRS
```
3. Wait for the command to succeed. Now, let's create another Storage Account. This Storage Account will be used for the new Azure Function App's internal operations. Replace the *{func\_storage\_account\_name}* placeholder with a globally unique name:
 

```
FUNC_STORAGE_NAME="{func_storage_account_name}"

az storage account create \
  --name $FUNC_STORAGE_NAME \
  --resource-group $RG_NAME \
  --location eastus \
  --sku Standard_LRS
```
4. Now create your new Azure Function App and assign a Managed Identity to it. You can either choose System-assigned, or User-assigned identities (<https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/overview>). Choose the System-assigned identity by passing the *--assign-identity [system]* parameter. Use the following command to create the Function App:

```

APP_SERVICE_PLAN_NAME="{plan_name}"
FUNCTION_APP_NAME="{function_name}"

az appservice plan create --resource-group $RG_NAME \
    --name $APP_SERVICE_PLAN_NAME \
    --sku S1 \
    --location eastus

az functionapp create \
    --resource-group $RG_NAME \
    --name $FUNCTION_APP_NAME \
    --storage-account $FUNC_STORAGE_NAME \
    --assign-identity [system] \
    --functions-version 3 \
    --plan $APP_SERVICE_PLAN_NAME

```



You need to create an *App Service Plan* before creating an Azure Function App. We will discuss App Service Plans in Chapter 7. See Azure documentation for details (<https://docs.microsoft.com/en-us/azure/app-service/overview-hosting-plans>).

5. So far you have a Function App which needs to access data on the Storage Account created in step 1. Use the following command to obtain the Function App Managed Identity id (GUID), and Storage Account resource id. You will need these information later to assign a RBAC role to the Function App:

```

STORAGE_RESOURCE_ID=$(az storage account show \
    --name $STORAGE_NAME \
    --resource-group $RG_NAME \
    --query id \
    --output tsv)

FUNC_OBJECT_ID=$(az functionapp show \
    --name $FUNCTION_APP_NAME \
    --resource-group $RG_NAME \
    --query identity.principalId \
    --output tsv)

```

6. Use this command to store the *Storage Blob Data Contributor* built-in RBAC role definition id in a variable:

```

ROLE_DEF_ID=$(az role definition list \
    --name "Storage Blob Data Contributor" \
    --query [].id --output tsv)

```

7. Now you have all the information you need to assign the *Storage Blob Data Contributor* built-in RBAC role to your Function App:

```

MSYS_NO_PATHCONV=1 az role assignment create \
    --assignee $FUNC_OBJECT_ID \

```

```
--role $ROLE_DEF_ID \  
--scope $STORAGE_RESOURCE_ID
```



*Git Bash* will automatically translate resource ids to Windows paths which will break your command. Simply add `MSYS_NO_PATHCONV=1` in front of your command to temporarily disable this behavior. See the Bash documentation for details([https://github.com/Azure/azure-cli/blob/dev/doc/use\\_cli\\_with\\_git\\_bash.md#auto-translation-of-resource-ids](https://github.com/Azure/azure-cli/blob/dev/doc/use_cli_with_git_bash.md#auto-translation-of-resource-ids)).

You successfully generated Azure Storage Data access to a Function App without using any Storage Account keys, or SAS token.

## Discussion

Azure Storage Accounts offer multiple authentications options (<https://docs.microsoft.com/en-us/azure/data-explorer/kusto/api/connection-strings/storage-authentication-methods>). You learnt about Azure Storage Keys and SAS tokens in the previous recipe (Recipe 3.3). The issue with these options is their security. There is always a chance that a SAS token or Storage Key is accidentally hard-coded and checked into a source control repository. Developers might send these keys over insecure communication lines such as instant chat, or emails. You need to make sure the keys and SAS tokens are well protected.

There is a better way to authenticate clients to Azure Storage Accounts, and any other resource type supporting Azure Active Directory authentication (<https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/services-azure-active-directory-support>). You can create a Managed Identity for your client resources, Function App in your recipe, and configure Azure Storage Account to allow access to this identity using RBAC roles. Both built-in and custom roles can be used.

Refer to Azure documentation (<https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/managed-identities-status>) for the list of Azure services that can use managed identities to access other services.

## 3.5 Enabling Point in Time Storage Blob Restore Using Snapshots

### Problem

You need to create snapshots of a Storage Account blob file so you have a history of the blob updates.

## Solution

Create Blob Snapshots (<https://docs.microsoft.com/en-us/azure/storage/blobs/snapshots-overview>) for your Azure blob object and store them in the Azure Storage Account. You can access these snapshots in the future using the datetime it was created. (Figure 3-5)

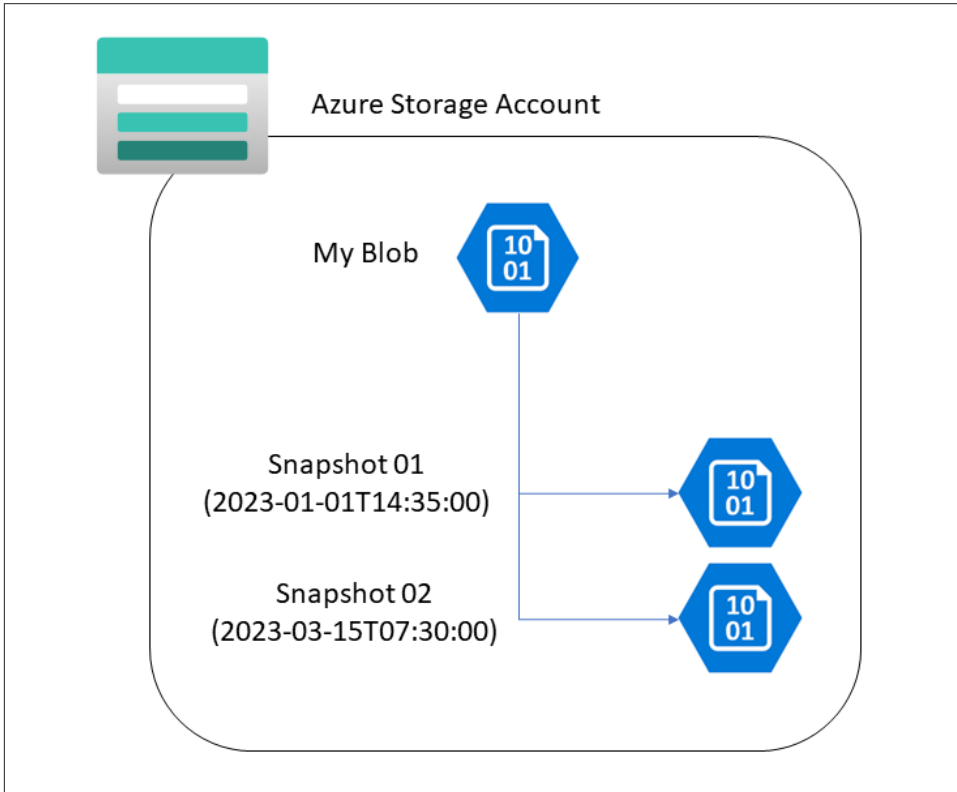


Figure 3-5. Creating Azure Blob Storage Snapshots

### Steps

1. Login to your Azure Subscription using a user with the *Owner* role.
2. Use the following CLI command to provision a new Azure Storage Account:

```
RG_NAME="{resource-group-id}"  
STORAGE_NAME="{storage_account_name}"
```

```
az storage account create \  
  --name $STORAGE_NAME \  
  --resource-group $RG_NAME \  
  --location eastus \  
  --sku Standard_LRS
```

```
--sku Standard_LRS \
--default-action Allow
```

3. If your account has required RBAC permissions over the Storage Account, you can choose *login* for the *--auth-mode* parameter. Alternatively, store the Storage Account key in a variable and use it to work with the Storage Account in the next steps as the following command demonstrates:

```
STORAGE_KEY_1=$(az storage account keys list \
--resource-group $RG_NAME \
--account-name $STORAGE_NAME \
--query [0].value \
--output tsv)
```

4. Create a container in the new Storage Account using this command:

```
CONTAINER_NAME="mycontainer"
```

```
az storage container create \
--name $CONTAINER_NAME \
--account-name $STORAGE_NAME \
--account-key $STORAGE_KEY_1
```

5. Wait for the command to succeed. Now let's upload a file to your new Storage Account using the following command. Replace the *{local\_path}* placeholder with the linux path to a small text file you created on your machine, for example */path/to/Myblob01.txt*.

```
BLOB_NAME="Myblob01.txt"
```

```
az storage blob upload \
--account-key $STORAGE_KEY_1 \
--file {local_path} \
--account-name $STORAGE_NAME \
--container-name $CONTAINER_NAME \
--name $BLOB_NAME
```

6. So far you created a blob file in a brand new Storage Account. Let's create a new snapshot of this blob using the following command:

```
az storage blob snapshot \
--account-key $STORAGE_KEY_1 \
--account-name $STORAGE_NAME \
--container-name $CONTAINER_NAME \
--name $BLOB_NAME
```



You can use the *az snapshot create* command to create Azure Disks snapshots. See Azure documentation (<https://docs.microsoft.com/en-us/azure/virtual-machines/disks-incremental-snapshots?tabs=azure-cli>), and the CLI documentation (<https://docs.microsoft.com/en-us/cli/azure/snapshot?view=azure-cli-latest>) for details.



7. Use the following command to get the snapshot details:

```
az snapshot show \  
  --name $SNAPSHOT_NAME \  
  --resource-group $RG_NAME
```

You successfully created a snapshot of your Azure Storage blob file. This is a read-only copy of your blob in the given time.

## Discussion

Azure Blob snapshots allow you to keep track of blob file changes over time. Think of Azure blob snapshots as database backup, allowing you to perform a point-in-time data restore. You can create snapshots of your blob object in given time intervals and store them as backups. This allows you to easily recover the previous versions on each blob as long as the snapshot for the given time exists.

Keep in mind that blob snapshots are taking space and your Azure subscription will be charged for the extra storage.

Blob versioning (<https://docs.microsoft.com/en-us/azure/storage/blobs/versioning-overview>) offers another method to keep track of your blob object changes. Blob versions are automatically created, as long as you enabled blob versioning for your Storage Account. We will cover Blob Versioning in the next recipe.



## 3.6 Working With Azure Storage Blob Versioning

### Problem

You need to keep track of all Azure Storage blob changes in order to recover previous versions if needed.

### Solution

Enable blob Versioning for your Azure Storage Account (<https://docs.microsoft.com/en-us/azure/storage/blobs/versioning-overview>). Each blob update will create a new blob file version. You can access these versions by their *version id* when needed. (Figure 3-6)

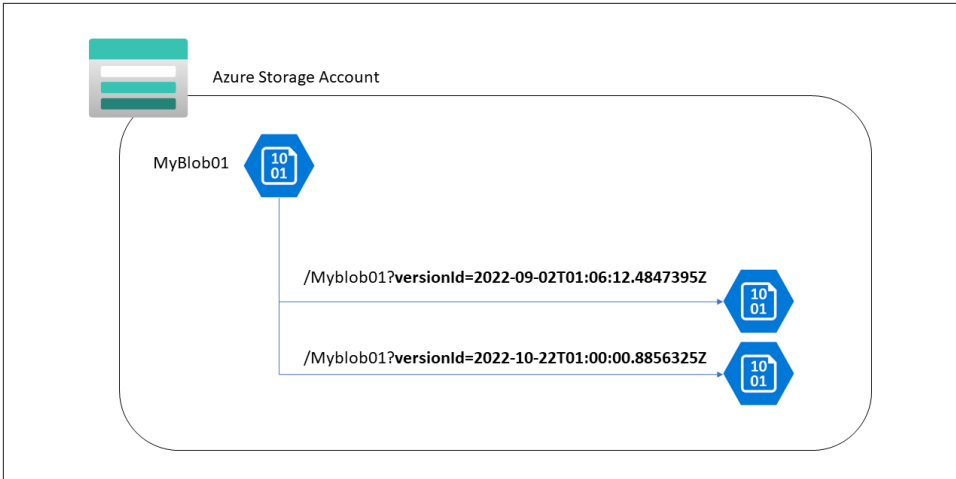


Figure 3-6. Accessing Azure Blob Storage Versions Using VersionId

## Steps

1. Follow steps 1 through 5 in the previous recipe (Recipe 3.5) to create a new Azure Storage Account and upload a text blob to it. Make sure you run all the commands so all the variables such as `$STORAGE_KEY_1` are properly initialized.
2. So far you created a blob file in a brand new Storage Account. By default, versioning is not enabled on Azure Storage Accounts. Run the following command to enable blob versioning for your Storage Account:

```
az storage account blob-service-properties update \
  --account-name $STORAGE_NAME \
  --enable-versioning true
```



Check `az storage account blob-service-properties update` command documentation for details (<https://docs.microsoft.com/en-US/cli/azure/storage/account/blob-service-properties?view=azure-cli-latest#az-storage-account-blob-service-properties-update>).

3. Now that you have blob versioning enabled, update your local `Myblob01.txt` and upload it again using this command. `--overwrite true` makes sure any file with the same name will be overwritten.

```
az storage blob upload \
  --account-key $STORAGE_KEY_1 \
  --file {local_path} \
  --account-name $STORAGE_NAME \
  --container-name $CONTAINER_NAME \
  --overwrite true
```

```
--name $BLOB_NAME
--overwrite true
```

4. By uploading *Myblob01.txt* over the existing blob, you automatically created a new blob version. Use this command to get the latest version of your blob file. Store this *versionId* in a variable.

```
LATEST_VERSION=$(az storage blob show \
  --account-name $STORAGE_NAME \
  --account-key $STORAGE_KEY_1 \
  --container-name $CONTAINER_NAME \
  --name Myblob01.txt \
  --query versionId \
  --output tsv)
```

5. You can download any blob version that you wish using this command. Replace the *{downloaded\_local\_path}* placeholder with the local Linux path you wish to save the file:

```
az storage blob download
  --account-name $STORAGE_NAME \
  --account-key $STORAGE_KEY_1 \
  --container-name $CONTAINER_NAME \
  --name Myblob01.txt \
  --file {downloaded_local_path}
  --version-id $LATEST_VERSION
```

You successfully created multiple versions of a blob file and downloaded the desired version using Azure CLI. You can also download older versions of *Myblob01.txt* by passing the appropriate *versionId*.

## Discussion

Azure Blob versions is a convenient solution to keep track of your Azure Storage blob changes. This is a superior method comparing to Azure blob snapshots because you need to explicitly create snapshots. Keep in mind that similar to blob snapshots, blob versions are also taking space and your Azure subscription will be charged for that extra storage.

Check Azure blob documentation (<https://docs.microsoft.com/en-us/azure/storage/blobs/versioning-overview>) for more details on blob versioning.




Microsoft recommends to stop creating blob snapshots after you enable blob versioning. With versioning enabled, taking snapshots does not offer any additional protections for your block blob data. See blob versioning documentation (<https://docs.microsoft.com/en-us/azure/storage/blobs/versioning-overview#blob-versioning-and-blob-snapshots>) for details.

## 3.7 Using a Lifecycle Management Policy to Save Storage Account Costs

### Problem

You need to move blob objects to less expensive access tiers to save on Storage Account costs.

### Solution

Configure one or more Azure Storage lifecycle management policies (<https://docs.microsoft.com/en-us/azure/storage/blobs/lifecycle-management-policy-configure>) to move blob objects from *Hot* to *Cool* or *Cool* to *Archive* access tiers based on blob access pattern 

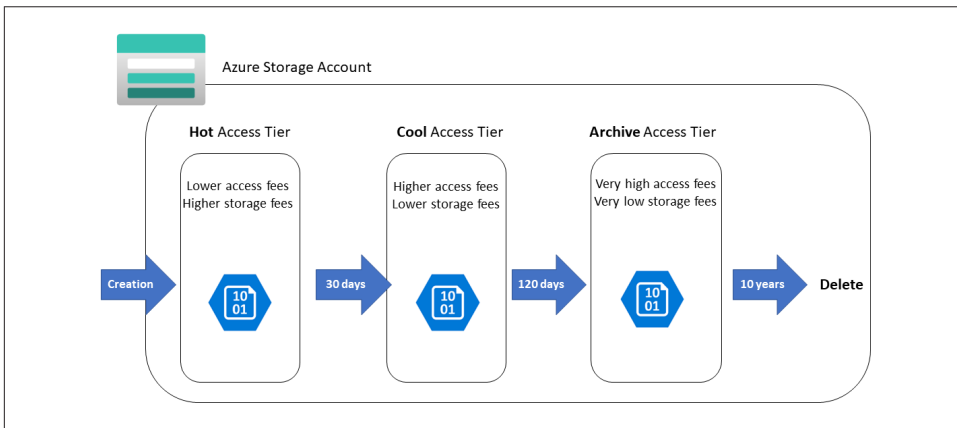


Figure 3-7. Using Lifecycle Management Policies to Optimize Storage Account Costs

### Steps

1. Login to your Azure Subscription using a user with the *Owner* role.
2. Use the `--access-tier` parameter to set the access tier when creating the Storage Account. The default value is *Hot* but added the parameter for clarity. Accepted values for this parameter are *Cool*, *Hot*, and *Premium*. Use the following CLI command to provision a new Azure Storage Account:

```
RG_NAME="{resource-group-id}"
STORAGE_NAME="{storage_account_name}"
```

```
az storage account create \
  --name $STORAGE_NAME \
  --access-tier Hot \
  --resource-group $RG_NAME \
```

```
--location eastus \  
--sku Standard_LRS \  
--default-action Allow
```

3. So far you created a new Storage Account in the *Hot* access tier. This means by default all the new blobs will be saved in the *Hot* tier, which is optimized for frequent access. Lets define a new lifecycle management policy which moves blobs from the *Hot* access tier to the *Cool* if they are unchanged for 60 days, and from the *Cool* access tier to the *Archive* after 120 days of being unchanged. You will finally delete them if unchanged for over 8 years (2,920 days). Save this following JSON content as ACCESS\_POLICY.JSON. You can also find it in this chapter's Github repository (<https://github.com/AzureCookbook/Storage>).

```
{  
  "rules": [  
    {  
      "enabled": true,  
      "name": "sample-rule",  
      "type": "Lifecycle",  
      "definition": {  
        "actions": {  
          "baseBlob": {  
            "tierToCool": {  
              "daysAfterModificationGreaterThan": 60  
            },  
            "tierToArchive": {  
              "daysAfterModificationGreaterThan": 120,  
              "daysAfterLastTierChangeGreaterThan": 60  
            },  
            "delete": {  
              "daysAfterModificationGreaterThan": 2920  
            }  
          }  
        }  
      },  
      "filters": {  
        "blobTypes": [  
          "blockBlob"  
        ]  
      }  
    }  
  ]  
}
```



You can also filter which blob types (block, page, etc.) and blob files to include in the access policy, or include *blob versions* in the policy if required by your scenario. See the Storage Account documentation (<https://docs.microsoft.com/en-us/azure/storage/blobs/lifecycle-management-overview>) for the policy JSON schema details.

4. Use the following command to create the policy for your Storage Account. Replace the `{path_to_json/ACCESS_POLICY.JSON}` with the Linux-style path to your local policy file created in the previous step:

```
az storage account management-policy create \  
  --resource-group $RG_NAME \  
  --account-name $STORAGE_NAME \  
  --policy {path_to_json/ACCESS_POLICY.JSON}
```

You successfully created a lifecycle management policy for your Azure Storage Account. Data will be moved from *Hot* to *Cool* access tier after 60 days of being unchanged, and to the *Archive* after 120 days of being unchanged. Finally the blobs will be deleted if not updated for 8 years (2,920 days).

## Discussion

Microsoft Azure Storage accounts offer three access tiers for blob objects (<https://docs.microsoft.com/en-us/azure/storage/blobs/access-tiers-overview>):

- *Hot tier*: Optimized for frequent access. You will pay less for accessing a blob object but more for storing it. Use this access tier if you need to access your blob objects frequently.
- *Cool tier*: Using this access tier, you will pay less for storing blobs comparing to the *Hot tier* but will pay more to access them. Use this tier for blob objects which you access less frequently but still need immediate access when needed. The blob should be at least 45 days old before being moved to this tier.
- *Archive tier*: An offline tier optimized for storing rarely accessed data. Use this tier if you don't need to immediately access the blob objects. You need hours to rehydrate the data before it being available to you. Data should be at least 180 days old before being moved to the archive tier.

Imagine you use Azure Storage blobs to store system access logs. The auditing application needs to frequently access the logs in the first month. The logs should be still accessible for the next year and should be kept for 10 years due to compliance requirements. Storing all logs in the *Hot* tier works, but you are overpaying for the expensive storage cost associated with the *Hot* tier. Simply optimize cost by creating a lifecycle management policy which:

- Moves the logs from the *Hot* tier to *Cool* after 31 days
- Moves data from *Cool* to *Archive* after 365 days (12 months)
- Finally deletes the logs from the *Archive* tier after 10 years (3,650 days)

Lifecycle management rules can also move blob data to *warmer* tiers if needed. For example you can create a lifecycle management policy to move a blob from the *Cool* tier to *Hot* as soon as it is accessed.



Do not move your blobs to the *Archive* tier if they need to be immediately readable! Blobs in the archive tier cannot be read unless they are first *rehydrated*, a process which may be time-consuming and expensive. For more information, see the archive tier documentation for details. (<https://docs.microsoft.com/en-us/azure/storage/blobs/archive-rehydrate-overview>).

## 3.8 Using AZCopy to Upload Multiple Files to Azure Storage Blobs

### Problem

You need to upload all the files within a local directory to Azure Storage Account blobs.

### Solution

Install the AZCopy tool (<https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azcopy-v10>) and use it to bulk-upload files to Azure Storage Account. (Figure 3.8)

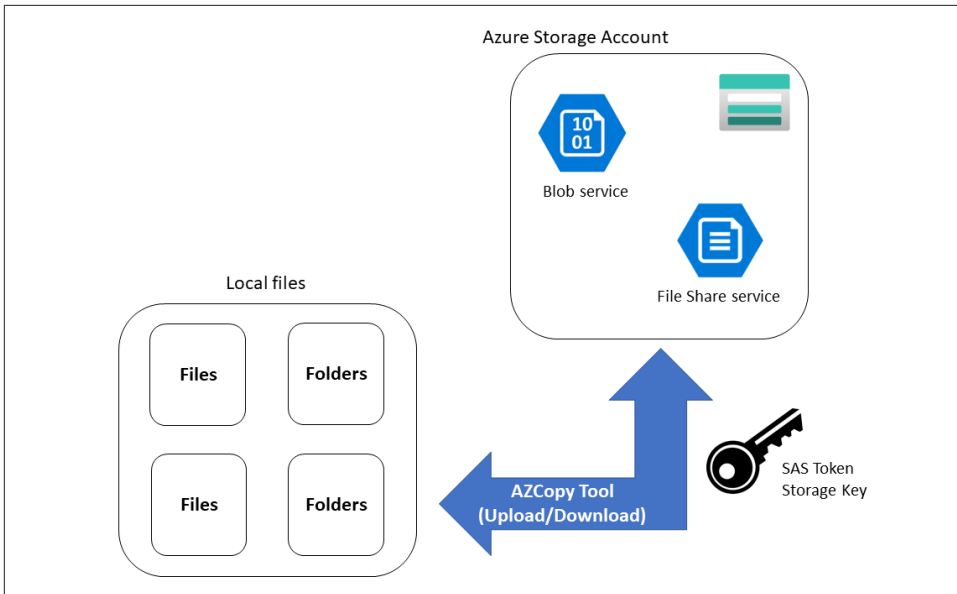


Figure 3-8. Using AZCopy to Bulk-upload Files to Azure Storage Account Blobs or File Shares

## Steps

1. Login to your Azure Subscription using a user with the *Owner* role.
2. Use the following CLI command to provision a new Azure Storage Account and create a new container in it:

```
RG_NAME="{resource-group-id}"
STORAGE_NAME="{storage_account_name}"
```

```
az storage account create \
  --name $STORAGE_NAME \
  --resource-group $RG_NAME \
  --location eastus \
  --sku Standard_LRS \
  --default-action Allow
```

```
STORAGE_KEY_1=$(az storage account keys list \
  --resource-group $RG_NAME \
  --account-name $STORAGE_NAME \
  --query [0].value \
  --output tsv)
```

```
az storage container create \
  --name "mycontainer" \
```



```
--account-name $STORAGE_NAME \
--account-key $STORAGE_KEY_1
```

3. Download and install the AZCopy tool for your OS. Check the AZCopy installation documentation for details. (<https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azcopy-v10>)

4. Create a local folder on your machine and copy/create several non-sensitive text files to it. Keep note of the local folder path and save it in the below variable (for example *C:\files*):

```
LOCAL_PATH="{local_folder_path}"
```

5. As you probably remember from Recipe 3.3, you can use an Azure Storage Key, or SAS token to grant access to a Storage Account. Take a look at Recipe 3.3 for detailed explanation. Run the following commands to create a new SAS token with read and write accesses (rw) over the Blob service (b).

```
expiryDate=`date -u -d "60 minutes" '+%Y-%m-%dT%H:%MZ'`
```

```
sasToken=$(az storage account generate-sas \
--account-name $STORAGE_NAME \
--account-key $STORAGE_KEY_1 \
--expiry $expiryDate \
--permissions rw \
--resource-types co \
--services b \
--https-only \
--output tsv)
```

6. Now use AZCopy to upload all the files within the *LOCAL\_PATH* folder to Azure Storage Account:

```
STORAGE_CONTAINER_URL=$(az storage account show \
--resource-group $RG_NAME \
--name $STORAGE_NAME \
--query primaryEndpoints.blob
--output tsv)"mycontainer/?"$sasToken
```

```
azcopy copy $LOCAL_PATH/"*" $STORAGE_CONTAINER_URL
```

7. To confirm your files are uploaded, list the blobs within your container using the following command:

```
az storage blob list \
--account-name $STORAGE_NAME \
--account-key $STORAGE_KEY_1 \
--container-name "mycontainer" \
--query [].name
```

You successfully uploaded multiple files to the Azure Storage Blob service using the AZCopy utility. The same tool can be used to upload files to the Storage Account File Shares as well. We will try this in the next recipe. See the AZCopy documen-

tation for details (<https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azcopy-files>).

## Discussion

AzCopy is a command-line tool that you can use to copy blobs or files to or from a storage account. Both Azure Blobs and Azure File Shares are supported. Use this tool to migrate your small to medium size local datasets to Azure Storage Account. For gigabyte-sized datasets, you might want to check other services such as *Azure Data Box* (<https://docs.microsoft.com/en-us/azure/databox/data-box-overview>).

You can use Storage Account Keys or SAS tokens to authenticate the AZCopy tool into your Storage Account. A SAS token is recommended because you can grant scoped access which expires after your migration is completed.

## 3.9 Using AZCopy to Upload Multiple Files to Azure Storage File Shares

### Problem

You need to upload all the files within a local directory to Azure Storage Account File Shares.

### Solution

Install the AZCopy tool (<https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azcopy-v10>) and use it to bulk-upload files to Azure Storage Account.

### Steps

1. Login to your Azure Subscription using a user with the *Owner* role.
2. Use the following CLI command to provision a new Azure Storage Account and create a new container in it:

```
RG_NAME="{resource-group-id}"
STORAGE_NAME="{storage_account_name}"

az storage account create \
  --name $STORAGE_NAME \
  --resource-group $RG_NAME \
  --location eastus \
  --sku Standard_LRS \
  --default-action Allow

STORAGE_KEY_1=$(az storage account keys list \
  --resource-group $RG_NAME \
```

```
--account-name $STORAGE_NAME \
--query [0].value \
--output tsv)
```

```
az storage share create \
--name "myfileshare" \
--account-name $STORAGE_NAME \
--account-key $STORAGE_KEY_1
```

3. Follow Step 3 in the previous scenario to download the AZCopy utility.
4. Create a local folder on your machine and copy/create several non-sensitive text files to it. Keep note of the local folder path and save it in the below variable (for example *C:\files*):

```
LOCAL_PATH="{local_folder_path}"
```

5. Run the following commands to create a new SAS token with read and write accesses (rw) over the File Share service (f).

```
expiryDate='date -u -d "60 minutes" '+%Y-%m-%dT%H:%MZ''
```

```
sasToken=$(az storage account generate-sas \
--account-name $STORAGE_NAME \
--account-key $STORAGE_KEY_1 \
--expiry $expiryDate \
--permissions rw \
--resource-types co \
--services f \
--https-only \
--output tsv)
```

6. Now use AZCopy to upload all the files within the *LOCAL\_PATH* folder to Azure Storage Account:

```
STORAGE_FILESHARE_URL=$(az storage account show \
--resource-group $RG_NAME \
--name $STORAGE_NAME \
--query primaryEndpoints.file
--output tsv)"myfileshare/?"$sasToken
```

```
azcopy copy $LOCAL_PATH $STORAGE_FILESHARE_URL \
--recursive=true
```



The AZCopy tool creates a log file which can be consulted in case of issues. Simply look into the AZCopy command output for the log file path.

7. To confirm your files are uploaded, list the blobs within your container using the following command:

```
az storage file list \
  --account-name $STORAGE_NAME \
  --account-key $STORAGE_KEY_1 \
  --share-name "myfileshare" \
  --query [].name
```

You successfully uploaded multiple files to the Azure Storage File Share service using the AZCopy utility. If the `--recursive=true` parameter is passed, all the sub-folders within the parent directory will be copied to the File Share as well.

## Discussion

Many companies are migrating their on-premises network files shares to Microsoft Azure to take advantage of availability, security, and manageability of Azure Storage Accounts. Use the AZCopy utility to conveniently upload your local file shares to Azure Storage Account File Shares. For gigabyte-sized shares, *Azure Data Box* (<https://docs.microsoft.com/en-us/azure/databox/data-box-overview>) remains the most efficient service.

## 3.10 Protecting Azure Storage Blobs From Accidental Deletion by Configuring the Soft-Delete Option

### Problem

You want to protect Azure storage blob files from accidental deletion.

### Solution

Enable the soft-delete option (<https://docs.microsoft.com/en-us/azure/storage/blobs/soft-delete-blob-overview>) for your Azure Storage Account blobs and containers. This enables you to recover deleted blobs within a retention period that you set.

### Steps

1. Follow steps 1 through 5 in Recipe 3.5 to create a new Azure Storage Account and upload a text blob to it. Make sure you run all the commands so all the variables such as `$STORAGE_KEY_1`, `$CONTAINER_NAME` and `$BLOB_NAME` are initialized.
2. So far you created a blob file in a brand new Storage Account. By default, soft-delete is not enabled on Azure Storage Accounts. Run the following command to enable soft-delete for your Storage Account blobs. The retention period is set to 14 days:

```
az storage account blob-service-properties update \
  --account-name $STORAGE_NAME \
```

```
--enable-delete-retention true \
--delete-retention-days 14
```



Check `az storage account blob-service-properties update` command documentation for details (<https://docs.microsoft.com/en-US/cli/azure/storage/account/blob-service-properties?view=azure-cli-latest#az-storage-account-blob-service-properties-update>).

3. Now any deleted blob can be recovered within 14 days of the deletion time. Let's try this by deleting the blob object we uploaded earlier:

```
az storage blob delete \
  --account-name $STORAGE_NAME \
  --account-key $STORAGE_KEY_1 \
  --container-name $CONTAINER_NAME \
  --name MyBlob $BLOB_NAME
```

4. To confirm your file is deleted, list the blobs within your container using the following command. Make sure you get any empty list [].

```
az storage blob list \
  --account-name $STORAGE_NAME \
  --account-key $STORAGE_KEY_1 \
  --container-name $CONTAINER_NAME \
  --query [].name
```

5. Use the following command to **undeleted** your blob object. The *Undelete* command will be successful only if it is used within the delete retention policy. Also attempting to undeleted a blob or snapshot that is deleted without enabling soft-delete will succeed without any changes! This command succeed because we are within the 14 day delete retention period:

```
az storage blob undelete \
  --account-name $STORAGE_NAME \
  --account-key $STORAGE_KEY_1 \
  --container-name $CONTAINER_NAME \
  --name MyBlob $BLOB_NAME
```

6. To confirm your file is recovered, list the blobs within your container using the following command. Make sure your blob file is back to the list:

```
az storage blob list \
  --account-name $STORAGE_NAME \
  --account-key $STORAGE_KEY_1 \
  --container-name $CONTAINER_NAME \
  --query [].name
```

In this recipe you added soft-delete protection for Azure Storage blobs, so any accidentally deleted blobs can be recovered within the set retention period.

## Discussion

Enable the soft-delete option for your Azure Storage Accounts containing sensitive or hard-to-replace blob objects. This feature protects your blobs and their snapshots/versions against accidental deletion.

Microsoft recommends the following steps to protect your blob data.

- Enable *soft-delete for blobs, snapshots, and/or blob versions*. You achieved this in the current recipe,
- Enable *soft-delete for containers* to protect containers against accidental deletion.
- Enable *Azure Storage blob versioning* for your Storage Account

Check Azure Storage blobs documentation (<https://docs.microsoft.com/en-us/azure/storage/blobs/soft-delete-blob-overview>) and soft-delete documentation for details. (<https://docs.microsoft.com/en-us/azure/storage/blobs/soft-delete-blob-overview#recommended-data-protection-configuration>)



Enabling *soft delete for blobs* does not protect parent containers. For complete protection also enable *soft-delete for containers* as recommended in the above list.

## 3.11 Protecting Azure Storage Account From Deletion Using Azure Locks

### Problem

You want to protect Azure Storage Account from deletion by users or applications.

### Solution

Create a delete lock (<https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/lock-resources?tabs=json>) for your Azure Storage Account. 

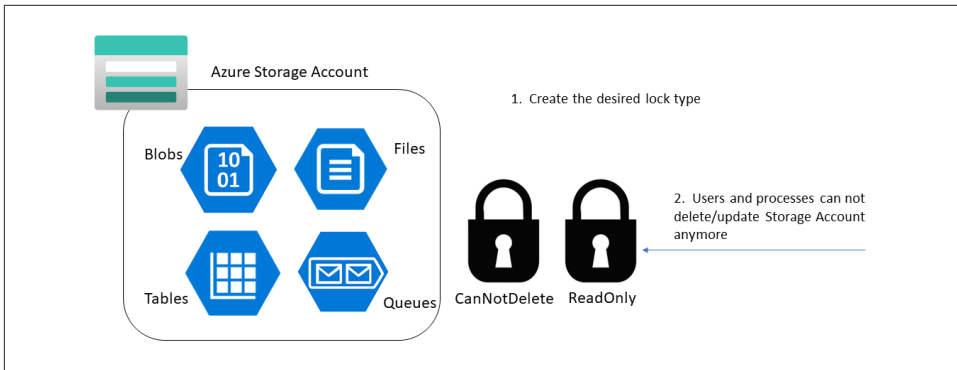


Figure 3-9. Protecting Storage Accounts Using Azure Locks

## Steps

1. Login to your Azure Subscription using a user with the *Owner* role.
2. Use the following CLI command to provision a new Azure Storage Account and create a new container in it:

```
RG_NAME="{resource-group-id}"
STORAGE_NAME="{storage_account_name}"
```

```
az storage account create \
  --name $STORAGE_NAME \
  --resource-group $RG_NAME \
  --location eastus \
  --sku Standard_LRS
```

3. Create a new Azure *CanNotDelete* Lock for your Storage Account using the following CLI command:

```
az lock create \
  --name MyStorageDeleteLock \
  --lock-type CanNotDelete \
  --resource-group $RG_NAME \
  --resource $STORAGE_NAME \
  --resource-type "Microsoft.Storage/storageAccounts"
```

4. You successfully created a *CanNotDelete* lock for your Azure Storage Account. The Storage Account can not be deleted by any user until this lock is deleted. You can use the following command to delete this lock:

```
az lock delete \
  --name MyStorageDeleteLock \
  --resource-group $RG_NAME \
  --resource $STORAGE_NAME \
  --resource-type "Microsoft.Storage/storageAccounts"
```

In this recipe you successfully created a *CanNotDelete Lock*. You can also create *\_ReadOnly Locks\_* using the same CLI command.

## Discussion

Use *Azure Locks* to protect your Azure subscriptions, resource groups, or resources against deletion and updates by other users or processes. Azure offers two lock types:

- *CanNotDelete lock*: which prevents the resource to be deleted by users/processes with any permissions level
- *ReadOnly lock*: as the name implies, this lock prevents users and processes to update the resource. They can still read the resource.

See *Azure Locks* documentation (<https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/lock-resources>) to read more about locks.



To create or delete locks, you need to have *Microsoft.Authorization/\** or *Microsoft.Authorization/locks/\** permissions/actions. Only the *Owner* and the *User Access Administrator* built-in roles have such access levels. As you saw in Recipe 1.2, a custom role with such permissions can also be created. See Locks documentation for details (<https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/lock-resources?tabs=json#who-can-create-or-delete-locks>).