QATIP Intermediate
Azure Lab07
Managing Azure Storage using Terraform

Contents

Lab Objectives

1. In this lab, you will use terraform to create the following resources...
   - Resource Group

- Storage Account
- Container
- Storage Management Policy
- Shared Access Signature (SAS) token

Teaching Points

This lab focusses on using terraform to create and manage Azure storage accounts, including creating containers, uploading data, storage management policies and SAS tokens.

Before you begin

1. Ensure you have completed Lab0 before attempting this lab

2. In the IDE terminal pane, enter the following command…

   cd c:\azure-tf-int\labs\08

3. This shifts your current working directory to c:\azure-tf-int\labs\08. Ensure all commands are executed in this directory

4. Close any open files and use the Explorer pane to navigate to and open the bonuslab

Solution

The solution to this lab is in folder c:\azure-tf-int\labs\solutions\08 Try to use this only as a last resort if you are struggling to complete the step-by-step processes.

Task 1. Create a Resource Group

[Blob Storage](#)

Try it yourself

Using the Registry resource url above for guidance....

1. Populate the **Task1** section of **main.tf** with an Azure provider block and a resource block creating a resource group called **"RG1"** in **"East US"**

2. Save, plan, and apply the deployment.

3. Switch to the Azure portal to verify the deployment

Step-by-Step

Using the Registry resource url above for guidance....

1. Click on "**USE PROVIDER**" and copy the terraform and provider blocks into the Task1 section of main.tf

2. Copy the **azurerm_resource_group** example block and paste below the provider block.

3. Change the resource block identifier to **"RG_1"**

4. Change the resource name to **"RG1"**

5. Change the location to **"East US"**

6. Save changes

7. Run **terraform init**

8. Run **terraform plan**

9. An error is returned because the azurerm provider block is missing a features section

10. The features section allows you to configure the way terraform behaves when configuring certain resources. See [Features Block](#) for more details.

11. To accept the default behaviour, enter a blank block; **features {}** within the provider block..

```
provider "azurerm" {
 features{}
 # Configuration options
}
```

12. Save the file changes

13. Run **terraform plan**

14. Another error is returned regarding the requirement to provide a **subscription_id** provider property.

15. Beneath the features section; add the following, replacing <your subscription id here> with your subscription id...

```
subscription_id = "<your subscription id here>"
```

16. Save the file changes

17. Run **terraform plan**

18. The planning stage should complete successfully.

19. Run **terraform apply** followed by **yes**

20. Switch to the Azure portal and verify the new Resource group **RG1** exists.

Task 2. Add a Blob Storage Account and Container

[Blob Storage](#)

Try it yourself

Using the Registry resource above for guidance, update the **Task2** section of **main.tf** with the following...

1. An **azurerm_storage_account** resource block with the following parameter/values...

       identifier: "storage_acct_1"

       name: "storageacct<add a unique-identifier>"

       resource_group_name: reference your resource group name

       location: reference your resource group location

       account_tier: "Standard"

       account_replication_type: "LRS"

2. An **azurerm_storage_container** resource block with the following parameter/values...

       identifier: "storage_cont_1"

       name: "example-container"

       storage_account_id: reference the id of your azurerm_storage_account

       container_access_type: "private"

3. Save, plan, and apply the deployment.

4. Switch to the Azure portal to verify the deployment of the storage account and container.

Step-by-Step

1. Open the Registry resource above for guidance.

2. Copy the **azurerm_storage_account** resource block example code into the **Task2** section of **main.tf**

3. Update the code as follows..

       identifier: "storage_acct_1"

       name: "storageacct<add a unique-identifier>"

       resource_group_name: azurerm_resource_group.RG_1.name

       location: azurerm_resource_group.RG_1.location

       account_tier: "Standard"

       account_replication_type: "LRS"

4. Copy the **azurerm_storage_container** resource block example code into the Task2 section of main.tf

5. Update the code as follows

 identifier: "storage_cont_1"

 name: "example-container"

 change **storage_account_name** to **storage_account_id**
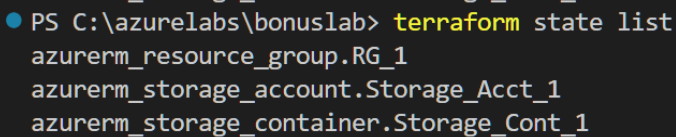
 storage_account_id: azurerm_storage_account.Storage_Acct_1.id

 container_access_type: "private"

6. Save, plan, and apply the deployment.

7. Switch to the Azure portal to verify the deployment of the storage account and container.

8. Run **terraform state list** to verify the following resources have been deployed..

```
PS C:\azurelabs\bonuslab> terraform state list
azurerm_resource_group.RG_1
azurerm_storage_account.Storage_Acct_1
azurerm_storage_container.Storage_Cont_1
```

Task 3. Examine the Files Upload code

https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/storage_blob

1. This is a guided task due to its complexity.

2. The Registry resource above gives a simple example of copying a local file into a storage container...

 DO NOT USE THIS CODE

```
resource "azurerm_storage_blob" "example" {
  name = "my-awesome-content.zip"
  storage_account_name = azurerm_storage_account.example.name
```

```
    storage_container_name = azurerm_storage_container.example.name
    type = "Block"
    source = "some-local-file.zip"
  }
```

3. Whilst this code would work as shown, it has very limited use. A more realistic example would be to upload several files and identify the mime type of the files to ensure its metadata is correctly stored.

4. To achieve this requires code of more complexity. Your **main.tf** is pre-populated with a Task3 section that contains an example of such code, but all lines are currently commented out.

5. Uncomment all code lines in the Task3 section of your main.tf

Code breakdown.

Mime types

```
locals {
  mime_types = jsondecode(file("${path.module}/mime.json"))
}
```

mime_types: This variable is a map (key-value pairs) of MIME types, loaded from the **mime.json** file in the root directory

This map will be used to nominate the content-type of a file based upon its file extension....

```
".txt": "text/plain",
".csv": "text/csv",
".jpeg": "image/jpeg",
".jpg": "image/jpeg",
".png": "image/png"
```

for_each in the Resource Block..

```
resource "azurerm_storage_blob" "upload_files" {
  for_each = fileset("${path.module}/static_files", "**/*")
```

for_each: This lets you create multiple instances of the resource, one for each file.

fileset(): Scans the folder **static_files** for all files that match the pattern **/* (all files and subdirectories recursively).

Blob Name

```
resource "azurerm_storage_blob" "upload_files" {
  for_each = fileset("${path.module}/static_files", "**/*")
  name                    = each.key
```

each.key: Refers to the relative path of the current file found by fileset().

Example Output: "Teide.jpeg"

Blob Source

```
  type                  = "Block"
  source                = "${path.module}/static_files/${each.key}"
  content_md5           = filemd5("${path.module}/static_files/${each.key}"
```

source: Specifies the full local path to the file being uploaded.

${path.module}: Refers to the current working directory of the Terraform module.

Example Source: "c:\azure-tf-int\labs\08\static_files\Teide.jpeg"

File Integrity

```
  source                = "${path.module}/static_files/${each.key}"
  content_md5           = filemd5("${path.module}/static_files/${each.key}")
```

filemd5(): Computes the MD5 checksum of the file. This ensures file integrity during the upload process.

Azure uses the checksum to verify that the file has not been corrupted during upload.

MIME Type

```
content_type = lookup(
  local.mime_types,
  regex("\\.[^.]+$", each.key),
  "application/octet-stream"
```

Purpose: Assigns a content type (like image/jpeg or text/plain) based on the file extension.

Extract File Extension:

regex("\\.[^.]+$", each.key):

Finds the file extension (e.g., .jpeg, .png).

Example Input: "Teide.jpeg" → Output: ".jpeg".

The regex **\\.[^.]+$** is used to extract the file extension from a file name by matching a dot (**.**) followed by one or more characters that are not dots, located at the end of the string. This helps identify the file type (e.g., **.txt**, **.jpeg**) for mapping to its corresponding MIME type in the **local.mime_types** map. If no match is found, it defaults to "**application/octet-stream**".

Code Summary

This Terraform code defines a process for uploading files as blobs to our Azure Storage container with appropriate metadata:

Local MIME Type Mapping: The locals block reads a JSON file (**mime.json**) from the module path. This file contains mappings of file extensions to their respective MIME types, and the **jsondecode()** function converts it into a usable map in Terraform.

Iterating Over Files: The **azurerm_storage_blob** resource uses the **fileset()** function to iterate over all files in the **static_files** directory (including subdirectories). Each file is identified by its relative path.

Blob Name Assignment: The name attribute of each blob is set.

Storage Container Details: The storage account and container names are set using existing resources, **azurerm_storage_account.storage_acct_1** and **azurerm_storage_container.storage_cont_1**.

Blob Type: The type of blob is specified as Block, which is commonly used for uploading large files.

File as Blob Source: The source attribute specifies the absolute file path from which the blob content will be uploaded.

MD5 Checksum for Verification: The content_md5 attribute computes the MD5 hash of the file to ensure data integrity during the upload process.

MIME Type Assignment: The content_type attribute determines the appropriate MIME type for the file. It uses regex("\\.[^.]+$", each.key) to extract the file extension from the file name, then looks up the

corresponding MIME type in the local.mime_types map. If no match is found, it defaults to "application/octet-stream".

So, this configuration automates the process of iterating through local files, mapping them to their MIME types, and uploading them as blobs into the Azure Storage container while preserving metadata and ensuring integrity.

6. Save the file changes

7. Run **terraform plan**

8. The planning stage should complete successfully.

9. Run **terraform apply** followed by **yes**

10. Switch to the Azure portal and verify the uploading of the 3 files to the storage container that the correct content-type has been attributed...
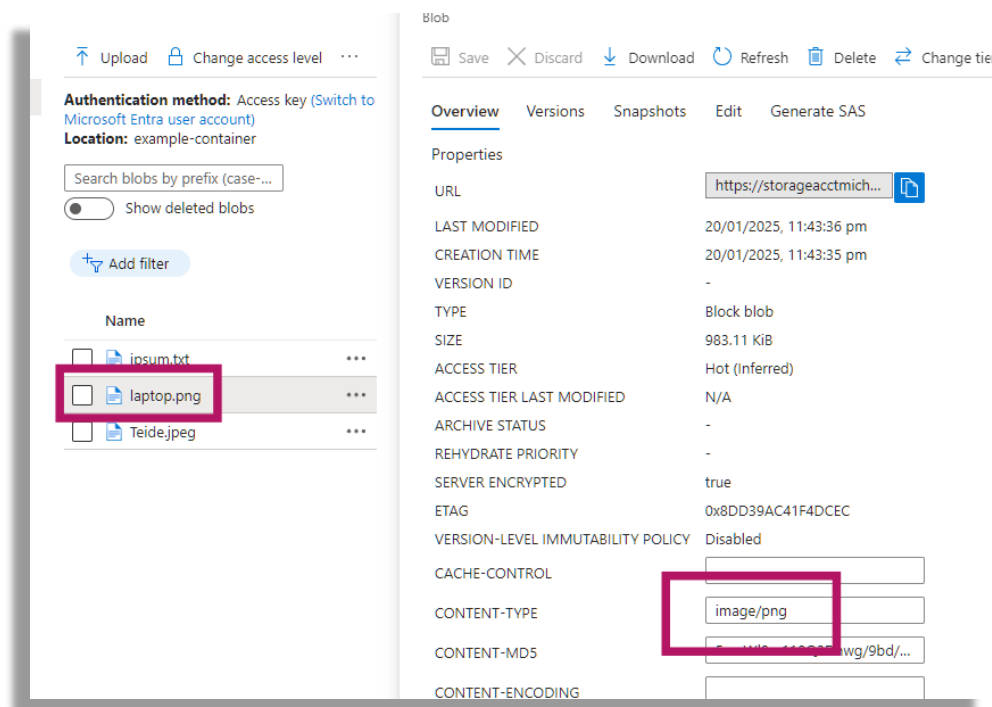
Task 4. Create an Azure Storage management policy

Storage Management Policy

Try it yourself

1. Review the sample code using the above link to familiarize yourself with storage management policy.

2. Using the example code for reference, in the Task 4 section of main.tf, create a policy that has a single rule focussing on the **base_blob** storage tier for **all** blob changes over time...

> **Identifier: storage_policy_1**
>
> **Rule name: blob-rule-1**
>
> **Actions:**
>
>> tier_to_cool_after_days_since_modification_greater_than = **11**
>> tier_to_archive_after_days_since_modification_greater_than = **51**
>> delete_after_days_since_modification_greater_than = **101**

3. Associate the policy with your storage account.

4. Save, plan, and apply the deployment.

5. Switch to the portal and verify the policy creation.

Step-by-Step

1. Open Terraform Registry Resources link above.

2. Copy the sample code for an **azurerm_storage_management_policy** resource block into the Task4 section of main.tf

3. Change the resource block identifier to **"storage_policy_1"**

4. Associate the rule with your storage account by changing the **storage_account_id** value to **azurerm_storage_account.storage_acct_1.id**

5. The example code contains 2 rules. **Remove all of rule2**.

6. Rename the remaining rule to **"blob-rule-1"**

7. For filtering, we want the policy to apply to all blobs, so remove the **prefix_match** and **match_blob_index_tag** portions of the filters block.

8. We are focussing on the blobs themselves here so remove the **snapshot** block

9. Configure changing to cool tier after **11** days

10. Configure changing to archive tier after **51** days

11. Configure blob deletion after **101** days since last modification

12. Save, plan, and apply the deployment.

13. Switch to the portal and verify the policy creation. (your storage account name will differ)..
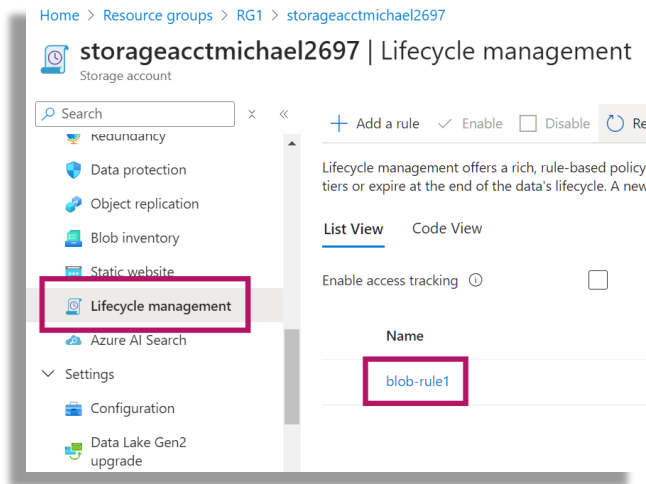
14. Click into the rule to view the parameters..
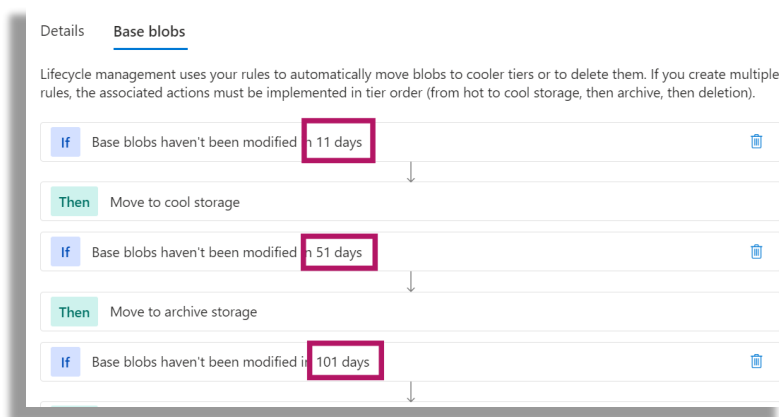
Task 5. Create an Azure Storage Access Signature token

SAS Tokens

Try it yourself

1. Review the sample code using the above link to familiarize yourself with Storage Access Signature token creation.

2. Create a data resource block to generate a SAS token with the following parameters:

> Identifier: sas_1
> Permissions to apply to **container** resource types and **blob** services only
> SAS should be valid for all of year 2025
> SAS should grant read/write/list permissions

3. Link the SAS to your storage account

4. Mark the output as sensitive

5. Save, plan, and apply your changes

Step-by-Step

1. Open Terraform Registry Resources link above.

2. Copy the azurerm_storage_account_sas example data block into the Task5 section of main.tf

3. Change to block identifier to **"sas_1"**

4. Change the connection_string value to link this SAS with our storage account..

   azurerm_storage_account.storage_acct_1.primary_connection_string

5. Remove the **signed_version_attribute**

6. Set the resource types to **containers** only

7. Leave **blob** as the only service

8. Change the date portion of the **start** date to **2025-01-01**

9. Change the date portion of the **expiry** date to **2026-01-01**

10. Set **read/write/list** permissions to **true**

11. Change the value attribute of the output block from data.azurerm_storage_account_sas.example.sas to value = **data.azurerm_storage_account_sas.sas_1.sas**

12. Save your file changes and then run **terraform plan**

13. An error should be displayed regarding the output code block...

```
To reduce the risk of accidentally exporting sensitive data that was intended to be only internal,
Terraform requires that any root module output containing sensitive data be explicitly marked as
sensitive, to confirm your intent.

If you do intend to export this data, annotate the output value as sensitive by adding the following
argument:
    sensitive = true
```

14. The SAS token is security sensitive and terraform will not display it directly as part of an output block. To fix this issue we cand mark this value as being 'sensitive' which means it will not be displayed but can be explicitly requested. To mark a value as sensitive, enter a line under the value..

    output "sas_url_query_string" {

      value = data.azurerm_storage_account_sas.sas_1.sas

      sensitive = true

    }

15. Save your file changes and then run **terraform plan**

16. Planning should complete

17. Run **terraform apply** followed by **yes**

18. **Note.** The Shared Access Signature (SAS) token generated via Terraform using the azurerm_storage_account_sas data block is not stored or visible directly in the Azure Portal. This is because SAS tokens created through APIs, SDKs, or Terraform are generated on the client-side and not stored by Azure for security reasons.

Task 6. Using the SAS token

1. Output the SAS token using **terraform output sas_url_query_string**

2. Reconstruct the following URL, substituting values as appropriate to your resources (ensure there are no spaces)..

    https://**<storage account name>**.blob.core.windows.net/**<container name>**?restype=container&comp=list&**<your sas token (minus leading ?)>**

3. Paste the re-constructed URL into a web browser to view the container object details using the SAS token..
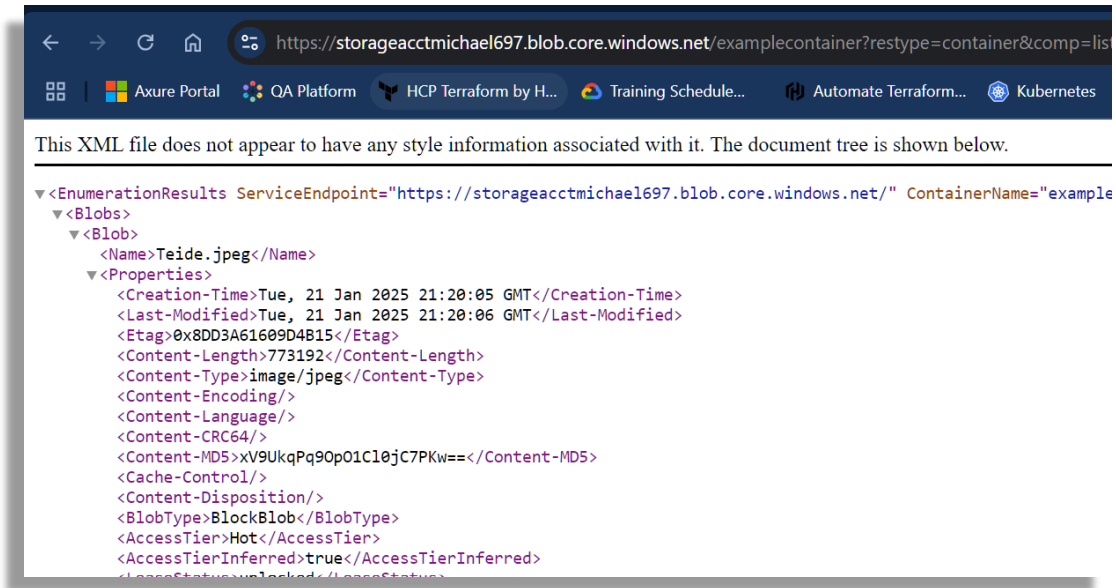
Task 7. Lab Clean-Up

1. Run **terraform destroy** and confirm with **yes** when prompted.

**** Congratulations, you have completed this lab ****