Part 1 - Introduction to IaC

In this first part, you will learn about Infrastructure as Code (IaC) and how to work with Bicep or Terraform to create a simple resource group and storage account. You'll also learn about things like:

- running deployments from the command line
- running deployments from GitHub Actions
- building service principals with federated credentials

Prerequisites

To complete this activity, you must have an editor like VSCode, an Azure Subscription with contributor access, and the Azure CLI installed.

Part 1: Introduction to IaC with Terraform

Apart from the common concepts of deployment scopes and resource groups there are Terraform specific features that are key to deploying resources. We will explore those in this section:

Providers

A provider is a plugin that allows Terraform to interact with cloud, SaaS providers and other APIs. Custom terraform providers can be created if needed. Example of providers are:

- Azure, Google and AWS providers.
- MongoDB, Pager Duty.
- Random, arm2tf.

Basic file structure

This is the recommended file structure for a working directory:

- **providers.tf:** Specifies the providers used in the deployment as well as any configuration for each of them.
- **main.tf:** Specified the resources being deployed.
- **variables.tf:** Specifies the variables that will be used to parameterize the deployment.
- **outputs.tf** Specified any values that will be availabe as an output of the deployment.

You can also use the main.tf file to define all the elements mentioned above, however, that will make your deployments harder to read and maintain.

Commands

There are 3 main commands that we will explore in this section:

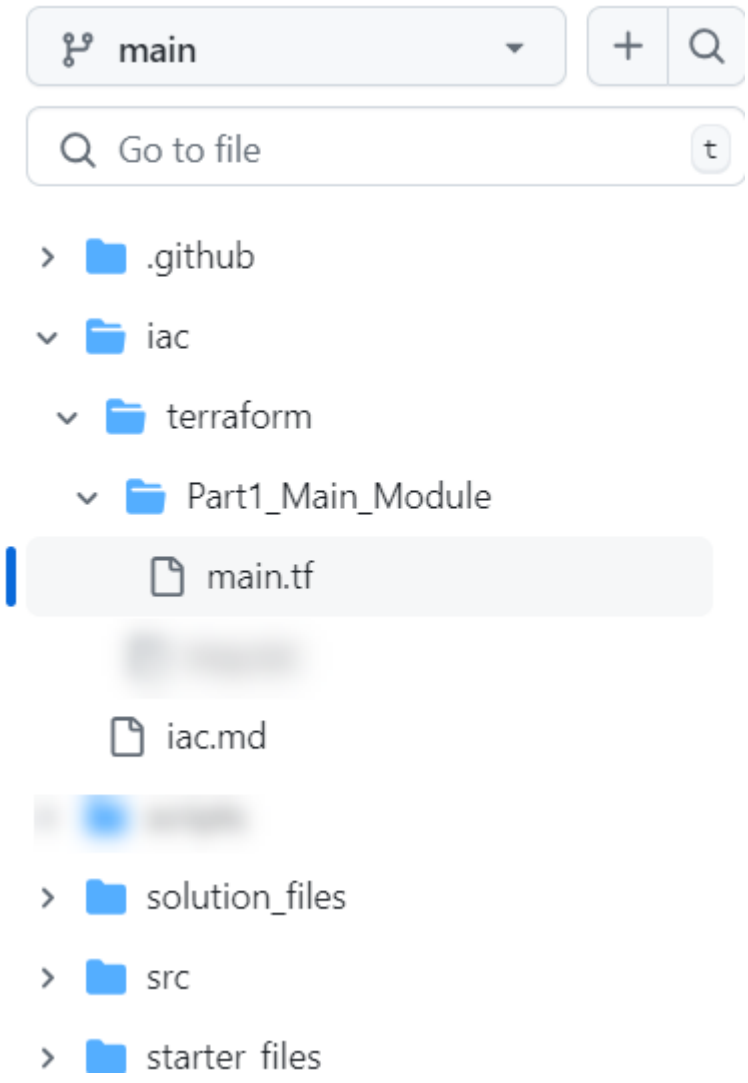
- **terraform init:** Initialized the working directory.
- **terraform plan:** Creates a plan based on the resources specified in your deployment and the resources currently deployed.
- **terraform apply:** Applies the plan generated by the `plan` command.

Task 1 - Create your first Terraform file to deploy a storage account to an existing resource group

To get started, let's create our first Terraform file. The overall goal for this activity is to create the files needed to deploy a storage account. During this activity we will create the recommended file structure mentioned above while learning about using variables and outputs, as well as how to create and use additional files as modules.

> **Note:** for this activity, I'm using VSCode with the Terraform extension. Additionally, I've created a new repository at GitHub which has the starter web application code in it and will be where I'm generating screenshots. For this reason, if you haven't already, you need a GitHub repository where you can store your code and your Terraform files. For simplicity, you can fork this repo: https://github.com/AzureCloudWorkshops/ACW-InfrastructureAsCode_Workshop.

A good way to store this would be similar to the following:



Step 1 - Create your file `main.tf`

Start by creating a main.tf file. This can be done in a bash terminal, in VSCode, or in PowerShell.

1. Create a folder if you don't have one for `iac` and a subfolder `Terraform`. In the terraform subfolder, create a file `main.tf`.

> **Note:** If you forked the repo above, you will already have an `iac` folder in that repo.

Folder:

```
mkdir terraform
cd terraform
mkdir Part1_Main_Module
cd iac\terraform\Part1_Main_Module
```

2. Copy or create the `main.tf` file in the `Part1_Main_Module` folder:

   Create the file `Bash`:

   ```
   touch main.tf
   ```

   or `PowerShell`:

   ```
   "" > "main.tf"
   ```

   or use VSCode:

   ```
   Right-click on the folder and select New File, name it `main.tf`
   ```

   > **Note**: For Bash and PowerShell, make sure you make directories `mkdir` and change directories `cd` to the correct location. For VSCode, you can right-click on the folder and select New File, name it `main.tf`.

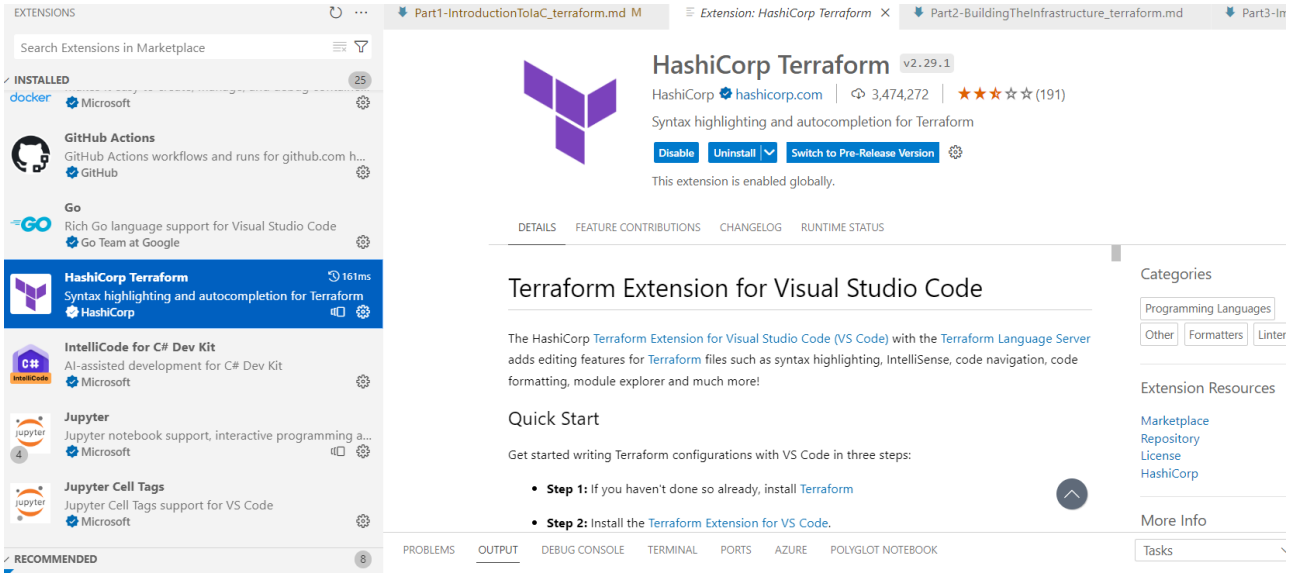
Completion Check

Before moving on, ensure that you have a file called `main.tf` in a folder called `iac\terraform\Part1_Main_Module` at the root of your repository (the _workshop repo has starter files and/or you should have made a folder with the `main.tf` file as shown above).

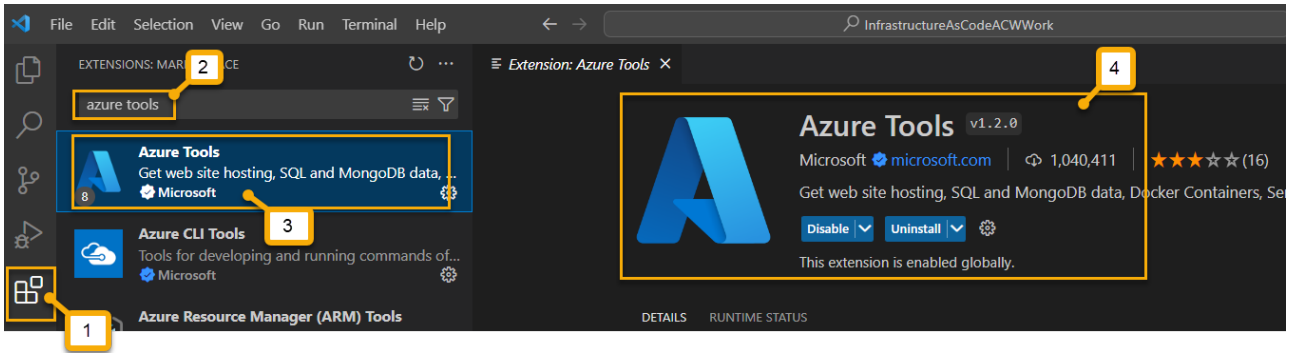
Step 2 - Create the terraform code to create a storage account

For this first activity, you'll be creating a simple storage account. To do this easily, you'll want a couple of extensions for Terraform in place in VSCode:

- Terraform:



- Azure Tools:



> **Note:** We may not need Azure Tools, but it's a good idea to have it in place for other things you will do in the future.

First, we need to specify the providers that we will use in the deployment. Add the following code to your main.tf file:

```
terraform {
  required_version = ">=1.6.6"

  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~>3.0"
    }
  }
}

provider "azurerm" {
  features {
  }
}
```

Next, we need to add a `resource` block to create the storage account:

```
resource "azurerm_storage_account" "cm_stg_acct" {
  name                     = "cmstgacct"
  resource_group_name      = "{YOUR RESOURCE GROUP NAME}"
  location                 = "{YOUR RESOURCE GROUP LOCATION}"
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

> **Note:** The resource group name and location must match the values you provided here.

Task 2 - Run the deployment

As mentioned in part 1, there are 3 commands that make up the basic Terraform workflow:

- terraform init
- terraform plan
- terraform apply

The first command only needs to be executed when creating a new configuration or updating an existing one. However, running the command multiple times should not cause any issues.

Step 1 - Issue commands to run the deployment

Before you execute any commands, make sure that you are in the `terraform\Part1_Main_Module`folder

 "Terraform directory."

Execute the `terraform init` command, after the command completes you should see the following:

```
Initializing the backend...

Initializing provider plugins...
- Reusing previous version of cloud-maker-ai/arm2tf from the dependency lock file
- Reusing previous version of hashicorp/azurerm from the dependency lock file
- Using previously-installed cloud-maker-ai/arm2tf v0.2.2
- Using previously-installed hashicorp/azurerm v3.85.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Next, execute the `terraform plan` command:

```
terraform plan -out main.tfplan
```

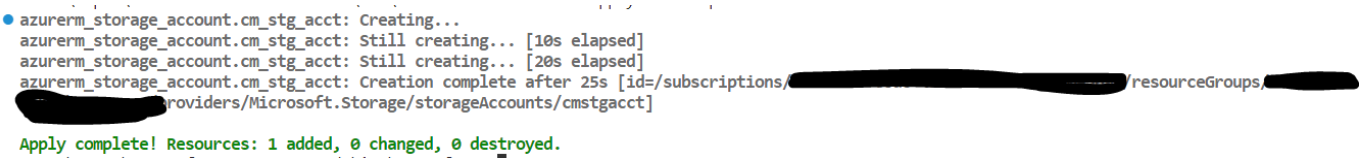
You should see the following output (some details are omitted):

```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following
symbols:
  + create

Terraform will perform the following actions:

  # azurerm_storage_account.cm_stg_acct will be created
  + resource "azurerm_storage_account" "cm_stg_acct" {

Plan: 1 to add, 0 to change, 0 to destroy.

Saved the plan to: main.tfplan

To perform exactly these actions, run the following command to apply:
    terraform apply "main.tfplan"
```

Finally, apply the plan by executing the following command:

```
terraform apply main.tfplan
```

This is the result:



Step 2 - Verify the deployment

Completion Check

You have a storage account in your resource group with a name that matches the value provided in the main.tf file.

Task 4 - Create providers file

As mentioned in part 1, when working with Terraform it is recommended to create separate files to keep everything organized. We will start by creating a separate `providers.tf` file.

Step 1 - Create `providers.tf`file

Step 2 - Move `terraform` and `providers` blocks to providers.tf file

You will now move the `terraform` and `providers` block from the main.tf to the providers.tf file, this should have no impact on your deployment but just to confirm execute the `terraform plan` command, you should see the following messages:

```
No changes. Your infrastructure matches the configuration.
```

```
Terraform has compared your real infrastructure against your configuration and
found no differences, so no changes are needed.
```

Task 5 - Use input variables

In Terraform module parameters are referred to as `input variables` or simply `variables`, in this part of the workshop you'll create input variables for the storage account name and location. You'll also learn how to use the variables in your deployment.

Step 1 - Add input variables to the terraform file

For starters, we will only add input variables for the resource group name, storage account name and location of the storage account.

1. Add the following code to the top of the main.tf file:

```
variable "resourceGroupName" {
    type = string
    nullable = false
    default = "{YOUR RESOURCE GROUP NAME}"
}

variable "storageAccountName" {
    type = string
    nullable = false
    default = "{YOUR STORAGE ACCOUNT NAME}"
}

variable "location" {
    type = string
    nullable = false
    default = "{YOUR RESOURCE GROUP LOCATION}"
}
```

2. Next, use the variables to populate the storage account values, in Terraform input values are referenced using the `var` object. Your storage account resource block should now look like this:

```
resource "azurerm_storage_account" "cm_stg_acct" {
  name                     = var.storageAccountName
  resource_group_name      = var.resourceGroupName
  location                 = var.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

3. Execute the `terraform plan` command again, since there were no infrastructure changes you should see this message again:

```
No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and
found no differences, so no changes are needed.
```

Step 2 - Create a variables file

In the previous step we added the ability to use input variables (or parameters) in the terraform template, in this step we will continue with the best practices mentioned above and we will move those variable definitions to a separate file.

1. Add a variables.tf file to your working directory, please note that this is just a suggested name and that as long as the file is in the same directory Terraform will automatically make those variables available in the main module.

2. Move the variable declarations from the main.tf to the variables.tf file, after the change, the only thing left in the main.tf file should be your resource block.

Step 3 - Deploy via variables file

Execute the `terraform plan` command again, since we are not updating the infrastructure you should again see no changes in the plan.

Step 4 - Deploy without default values

You might have noticed that so far you haven't been asked to provide a value for the variables, that is because of the default values that we have assigned. Lets test a deployment without default values:

1. Remove the default value for the resourceGroupName variable.

2. Execute the `terraform plan` command, you should see this prompt:

```
PS D:\repos\ACW-InfrastructureAsCode\iac\terraform> terraform plan -out main.tfplan
var.resourceGroupName
  Enter a value:
```

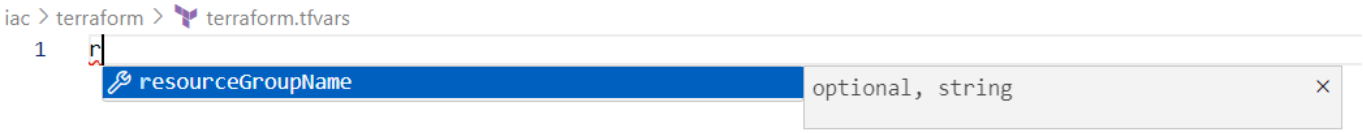
This is obviously not the most efficient deployment strategy and is also error prone, we will look at a better deployment option in the next step.

Step 5 - Use a variable definitions file

In this step, we will use a special file called a variable definitions file to specify the values we want to use in the deployment.

1. Add a file called terraform.tfvars to your working directory, Terraform automatically scans for this specific file when deploying resources. If you want to use a different file name you will need to use the `-var-file` parameter when executing the `plan` command.

2. Type the name of any of the variables declared in the `variables` file, if you are using the Terraform extension for Visual studio code you should see something like this:



Provide a value for all the variables.

3. Execute the `terraform plan` command again, you should not be prompted to provide a value for the resourceGroupName variable and you should see no changes to your configuration.

Completion Check

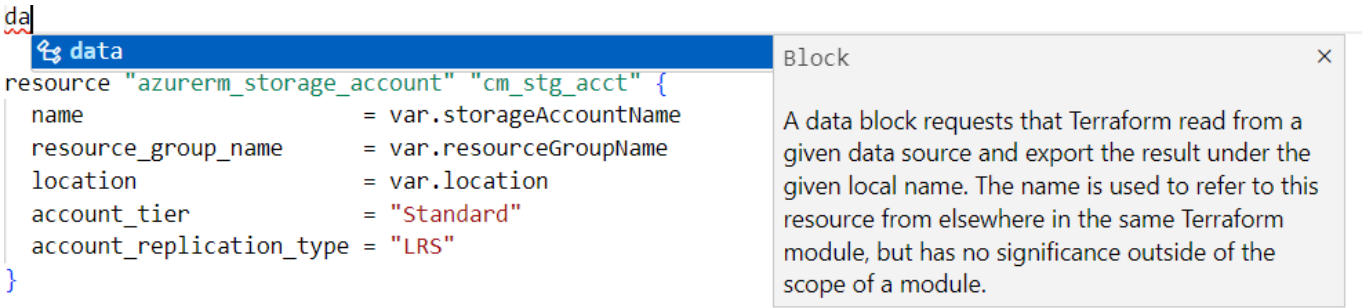
You have a file that you can reuse in multiple resource groups with various storage account names (you would need to change the name in the parameter file at this point to ensure it is unique).

Task 6 - Use data sources

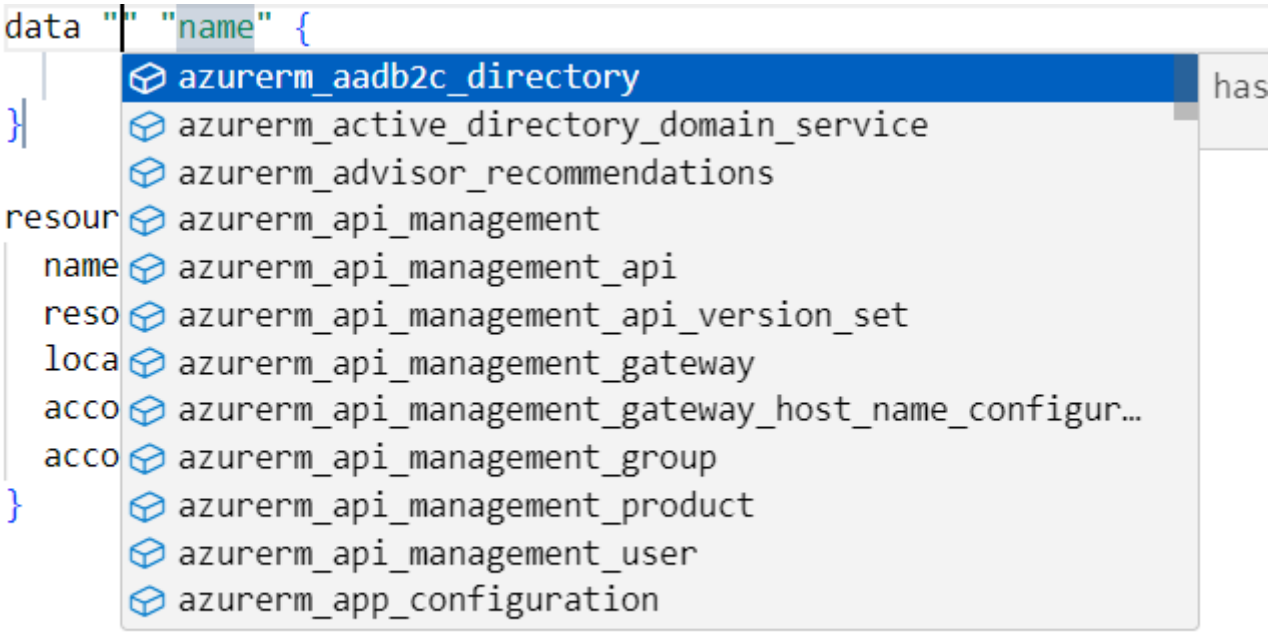
Up until now we have used variables to provide the name and location of the resource group that contains the storage account, however, Terraform has another way to access information defined outside of Terraform or that is part of a different deployment: data sources. In this step, we will modify the files we have to use a data source to access the resource group information instead of providing the values through variables.

Step 1 - Add data source to configuration

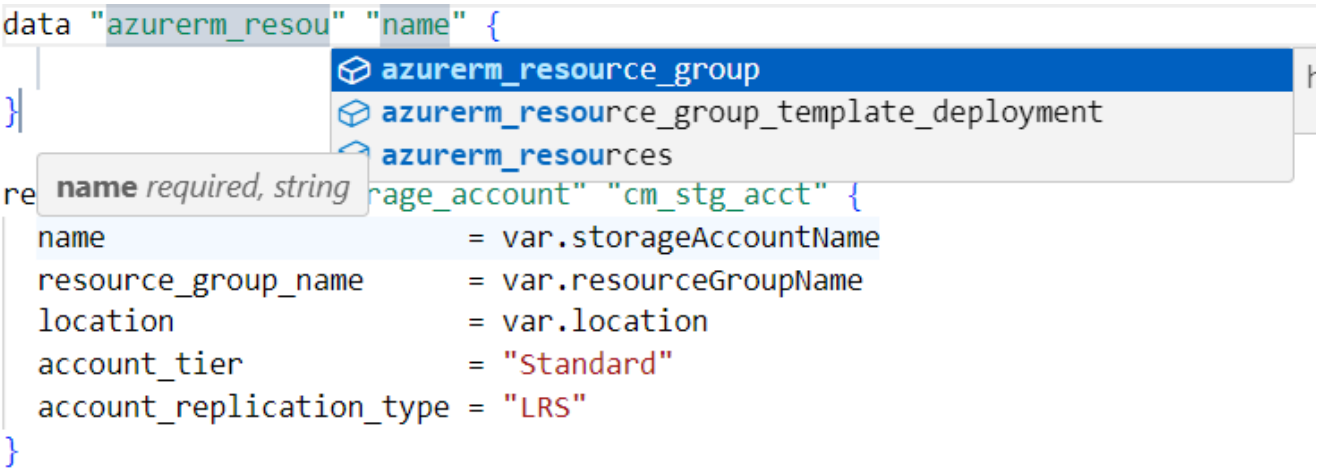
Go to the top of the main.tf file and type `da`, if you are using the Terraform extension for VS code you should see the following:



Hit the tab key, a data block will be created automatically:



Type `azurerm_resource_group`, autocomplete should display the option after you type a few characters:



The resource group data source requires the name of the resource group, you can use the resourceGroupName variable to populate the parameter. The data block should look like this:

```
data "azurerm_resource_group" "data_rg" {
  name = var.resourceGroupName
}
```

Step 2 - Use data source values in storage account configuration

You can now replace the `resource_group_name` and `location` parameters in the storage account block with the values from the data source using the following syntax:

```
data.{RESOURCE_TYPE}.{DATA_SOURCE_NAME}.{DATA_SOURCE_PROPERTY}
```

For example, to access the name of a resource group with a data source:

```
data.azurerm_resource_group.data_rg.name
```

After replacing the name and location of the resource group the storage account block should look something like this:

```
resource "azurerm_storage_account" "cm_stg_acct" {
  name                     = var.storageAccountName
  resource_group_name      = data.azurerm_resource_group.data_rg.name
  location                 = data.azurerm_resource_group.data_rg.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

Step 3 - Remove location variable and execute deployment.

Since we are now getting the resource group information from the data source we can now remove the `location` variable from the `variables.tf` and the value assignment from the `terraform.tfvars` file.

You can now execute the `terraform plan` command again, since we are not adding or removing any resources you should see a message saying that no changes were detected.

Task 7 - Use local variables and functions

In this module you will learn to use local variables and functions to create a unique string name for the storage account name. The term `local variable` in terraform refers to any variable used inside a module.

Step 1 - Add a unique identifier input variable to the storage account

Since the storage account name needs to be unique across all resources in Azure we will now add a unique identifier section to the storage account name to comply with this requirement.

1. Add a `uniqueIdentifier` variable to your variables file.

2. Assign a value in the tfvars file with the following format: YYYYMMDDabc.

Step 2 - Add a local variable for the storage account name

Local variables in Terraform are declared using a `locals` block, add a locals block at the top of the main.tf file and add a variable called `storageAccountNameFull`. Assign a value by concatenating the storageAccountName and uniqueIdentifier variables using interpolation:

```
locals {
  storageAccountNameFull = "${var.storageAccountName}${var.uniqueIdentifier}"
}
```

Add another storage account resource block by copying the existing one and replacing the `name` parameter with the local variable you just created, the following syntax is used to access local variables:

```
local.{YOUR VARIABLE NAME}
```

Your new storage account block should look like this:

```
resource "azurerm_storage_account" "cm_stg_acct_full" {
  name                     = local.storageAccountNameFull
  resource_group_name      = data.azurerm_resource_group.data_rg.name
  location                 = data.azurerm_resource_group.data_rg.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

Execute the `terraform plan` command again, you should see a message saying that 1 resource will be added.

Execute the `terraform apply` command, once it is completed you should see the new storage account in your resource group.

Step 3 - Use a provider to add a unique string to the storage account name

If you completed the Bicep section of this workshop, you will recall that the `uniquestring` function allows you to generate a string that can help make resource names unique. In this step, you will use a provider to access similar functionality in Terraform.

1. Add the following after the azurerm element in the providers.tf file:

```
random = {
  source = "hashicorp/random"
  version = "3.6.2"
}
```

2. Generate a unique id in main.tf file by adding the following block to your main.tf file:

```
resource "random_string" "random" {
  length           = 10
  special          = false
  lower            = true
  upper            = false
}
```

> **Note:** The name for storage accounts does not allow any special characters so we are forcing the generated string to comply to this. You can check the documentation for the random provider for additional configuration options.

3. Add a new local variable called `storageAccountNameUnique` and assign the following value:

```
storageAccountNameUnique =
"${var.storageAccountName}${var.uniqueIdentifier}${random_string.random.result}"
```

4. Add a new storage account resource block and assign the name using the new variable created.

5. Since we added a provider to our deployment, we need to run the `terraform init` command before we can create a new plan, otherwise you'll see the following message:

```
Error: Inconsistent dependency lock file

The following dependency selections recorded in the lock file are inconsistent with the current
configuration:
  - provider registry.terraform.io/cloud-maker-ai/arm2tf: required by this configuration but no version i
s selected

To update the locked dependency selections to match a changed configuration, run:
  terraform init -upgrade
```

6. Deploy the new resource by executing the `plan` and `apply` command, you should see 2 resources being created in the plan: one for the uniqueid and one for the new storage account.

Your deployment should fail due to the length of the new storage account name being over 24 characters, in the next step we will use one of the built-in functions in Terraform to solve the problem.

Step 4 - Use a function to truncate the resource name

Terraform has multiple built-in functions that can be used to transform and combine values, you can find a full list here: https://developer.hashicorp.com/terraform/language/functions. In our case, we will use the `substr` function:

```
substr(string, offset, length)
```

1. Use the `substr` function to ensure the value of the `storageAccountNameFullUnique` variable is only 24 characters.

2. Execute the deployment again, you should see the new storage account in the Azure portal.

Step 5 - Use a validator to ensure the storage account name is unique and long enough

Input variables allow your Terraform files to be dynamic but without any validations there is also the risk that the values provided will result in invalid properties for your resources. Terraform provides different ways to validate your configuration, in this step we will add validations to a couple input variables to prevent any

issues with planning and / or deployment. Variable validations are added using the `validation` property of a variable, for example:

```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    condition     = length(var.image_id) > 4 && substr(var.image_id, 0, 4) ==
"ami-"
    error_message = "The image_id value must be a valid AMI id, starting with
\"ami-\"."
  }
}
```

1. Add the following validation block to the `storageAccountName` variable:

```
validation {
  condition = length(var.storageAccountName) > 3
  error_message = "The storage account name should be at least 3 characters"
}
```

2. Assign the value `c` to the storageAccountName variable in the `terraform.tfvars` file.

3. Execute the `plan` command, you should see the following message:

```
Planning failed. Terraform encountered an error while generating this plan.

Error: Invalid value for variable

  on variables.tf line 6:
   6: variable "storageAccountName" {
     ├────────────────
     │ var.storageAccountName is "c"

The storage account name should be at least 3 characters

This was checked by the validation rule at variables.tf:11,5-15.
```

If time permits, add 2 more validations:

- Validate that the uniqueidentifier variable is 11 characters long.
- Add an `environment` input variable that will allow only 2 values (hint: look at the contains function): `dev` and `prod` and use that to create a new storage account resource.

Completion Check

You can now deploy the same file to different resource groups multiple times and it will create a unique storage account name per group (and per environment if needed) using local variables, built-in functions and validations. If time permits, you can try creating a different resource group and use what we have built so far to deploy these resources there.

Task 8 - Use modules and outputs

So far, we have been working out of our main module. In application deployments like the one we will do in part 2, we want to be able to have our resources distributed in modules. Lets look at the concept of modules using a similar deployment to the one we have so far.

Step 1 - Create a resource group

In our previous deployment we were using information from an existing resource group, however, in a real world scenario we might have to create the entire infrastructure including the resource group. In this step we will create a new template that deploys a new resource group, lets see how much you remember!

1. Create a new folder under the `terraform` folder and add the following files:

- providers.tf
- variables.tf
- main.tf
- terraform.tfvars

2. Add the `azurerm` and `random` providers to the providers.tf file.

3. Add the following variables:

- resourceGroupName: string type, not nullable.
- location: string type, not nullable, validation to only allow `East US` as a value (optional).

4. Assign values to the variables in the terraform.tfvars file.

5. Add a resource group block to the main.tf file, the resource type needed is `azurerm_resource_group`.

6. Deploy the resource group, don't forget to initialize Terraform!

Step 2 - Create a module for the storage account

Now that we have our resource group, lets create a module for the storage account. In Terraform, you can create your own modules locally or you can get modules from the Terraform registry, you can even publish your own modules for other people to use. For this exercise, we will focus on local modules:

1. Add a modules folder in your working directory and add a storageAccount inside of it.

2. Create a main.tf and variables.tf files in the storageAccount folder.

3. Copy any of the storage account resource blocks from the root module created in the previous exercise and replace the values assigned to the `name`, `resource_group_name` and `location` with variables.

4. Add the variables specified in the previous step to the variables.tf file.

5. Add a reference to the storage account module by adding the following block:

```
module "storageAccount" {
  source = "./modules/storageAccount"

  storageAccountNameEnv = local.storageAccountNameEnv
  resourceGroupName     = var.resourceGroupName
  location              = var.location
}
```

The main parameter you need to supply when using modules is the `source`, for local modules it should be populated with the folder location. You also need to pass values for any variables that the module is expecting.

6. Deploy the resources, you should see the storage account created in the resource group.

Step 3 - Create an output for the storage account

There is one more Terraform element that we have not worked with yet: outputs. When creating resources, some of the properties of those resources are exported so you can use them as parameters for other steps in your deployment. In this step, we will export some of the storage account properties as outputs.

1. Create an `outputs.tf` file in the storage account module directory.
2. Add 3 output blocks to the newly created file: account name, id and location. This is an example of the output block for the account name:

```
output "{OUTPUT_VARIABLE_NAME}" {
  value = azurerm_storage_account.{YOUR_STORAGE_ACCOUNT_RESOURCE}.name
}
```

Since we are not updating the infrastructure executing the `plan` and `apply` commands will have no effect, we will do that in the next step.

Step 4 - Leverage an output in another deployment

We will now use the outputs generated by the storage account resource to create a storage container resource, please note that we are creating these resources separately just for purposes of this workshop. In a real world scenario you would most likely deploy all these resources in the same module.

1. Create a new folder inside the `modules` folder called `storageContainer`.
2. Add a `main.tf` and a `variables.tf` file.
3. Add 2 variables: `storageAccountName` and `containerName`.
4. Add an `azurerm_storage_container` resource block and populate its properties with the variables you created.

5. Add a `containerName` variable to the input variables of the root module.

6. Add a `storageContainer` module block to the root module and populate the `storageAccountName` with the output value from the `storageAccount` module:

```
module "storageContainer" {
  source = "./modules/storageContainer"

  containerName      = var.containerName
  storageAccountName = module.storageAccount.stg_acct_name
}
```

Task 8 - Use azurerm backend

As you already know, the state file is key for Terraform to know what changes need to be done to the infrastructure; this file can be saved in different locations. So far, we have been using the local backend but in this step we will add a remote backend which will be needed for automated deployments.

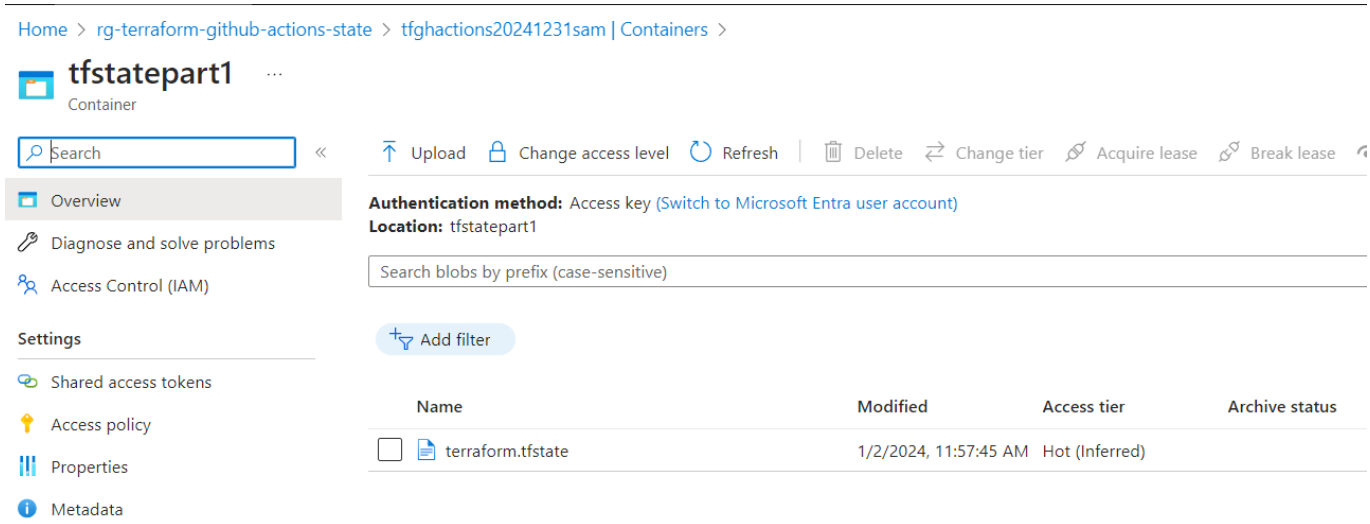
Step 1 - Create infrastructure for state file

Go to the Azure Portal and create the following resources:

- RG: `rg-terraform-github-actions-state`
- Storage Account: `tfghactionsYYYYMMDDxxx`
- container: `tfstatepart1`

Step 2 - Copy state file to storage container

Go to your storage container and upload your terraform.tfstate file:



Step 3 - Add a backend block to the providers.tf file

Open the providers.tf file and add this block after the `required_providers`:

```
backend "azurerm" {
    resource_group_name  = "rg-terraform-github-actions-state"
    storage_account_name = "{YOUR_STORAGE_ACCOUNT_NAME}"
    container_name       = "tfstatepart1"
    key                  = "terraform.tfstate"
    use_oidc             = true
  }
```

Step 4 - Create a deployment plan

Since we updated the backend configuration we need to execute the `terraform init` command and then execute the `terraform plan` command, since the state is the same as the last time you should see the message of no changes needed for the configuration.

Completion check

At this point you have learned most of the basic concepts you will need to work with Terraform for infrastructure deployments. Make sure that all the files were created successfully and that you can re-run your deployments at will. The repetitive and consistent nature of the deployments are some of the main reasons you want to use Infrastructure as Code.

Conclusion

In this first part, you learned how to work with Terraform to create storage account resources in a resource group. Along the way you learned the following concepts:

- creating terraform files
- running deployments from the command line
- using input and local variables
- using data sources
- using functions
- using modules
- using outputs