terraform {  
 required\_providers {  
 azurerm = "~> 2.64"  
 }  
}provider "azuread" {  
}resource "azurerm\_resource\_group" "example" {  
 name = "example-resources"  
 location = "West Europe"  
}  
  
resource "azurerm\_storage\_account" "example" {  
 name = "examplestoracc"  
 resource\_group\_name = azurerm\_resource\_group.example.name  
 location = azurerm\_resource\_group.example.location  
 account\_tier = "Standard"  
 account\_replication\_type = "LRS"  
}  
  
resource "azurerm\_storage\_container" "example" {  
 name = "content"  
 storage\_account\_name = azurerm\_storage\_account.example.name  
 container\_access\_type = "private"  
}

When we tell Terraform to do a deployment, it will do a sequence of steps:

1. It will parse our HCL configuration/code files.
2. Using the information in our HCL, Terraform will build up a graph of all the resources we want to provision ***(desired state)*** and figure out any dependencies between them to try and decide a logical order they need to be created in.
3. Terraform will next inspect its State to better understand what it has and hasn’t deployed (if it is our first deployment, the State will be empty). This is known as ***perceived state.***It is perceived state because there is a disconnect between what Terraform “thinks” exists and what “actually” exists.
4. Terraform next performs a logical delta between our **desired state**, and what it knows to be our ***perceived state.***It then decides which CRUD actions it needs to perform, and the order to perform them in, in order to bring our ***perceived state***in-line with our ***desired state***.
5. Terraform next performs all necessary operations to achieve the ***desired state***. The result of this operation will be that resources will likely start to appear in our Azure subscription and this then becomes known as ***actual state.***
6. Terraform updates the state to reflect what it has done.

**initialisation**

*$ terraform init*

The first command we need to run is init . This command is used to initialise a working directory containing Terraform configuration files and instructs Terraform to interrogate the HCL files, determine the Providers needed, download them, and initialise a State if it doesn’t already exist. It is perfectly safe to run init multiple times, and in fact you will likely need to do so if you add new providers or change any settings within your Terraform block.

For more detailed info on the init command [you can check all the docs here](https://www.terraform.io/docs/cli/commands/init.html).

**Planning**

*$ terraform plan*

The next command we run is plan . This command instructs Terraform to parse our HCL files, build its graph of our resources, check its state and attempt to come up with an execution plan to perform. It is perfectly valid for our init to succeed but our plan to fail. This is because init doesn’t really concern itself with trying to determine if any of our resources are valid or if they exist. Only when Terraform starts to interpret our “desired state” will we start to find syntax errors in our files.

The output of the plan command will be a complex list of operations that Terraform has decided to perform. It is highly recommended that you review these changes to ensure they match with those you expected to see as destructive operations can be costly if they are done in error. Think of this as a “dry run” of all your Terraform.

It is also worth noting that the output of the plan can be saved into a file and used for the next step apply as this ensures that there can be no confusion/discrepancies between what was planned and that which was applied.

For more detailed info on the plan command the [documentation can be found here for your convenience.](https://www.terraform.io/docs/cli/commands/plan.html)

**Applying**

*$ terraform apply*

Simply put this operation tells Terraform to execute all its planned operations. This will usually cause Terraform to redo its plan (unless a plan file is provided to the command) and then present you with an “Are you sure?” prompt. This is your point of no return. If you say yes then Terraform is going to start really provisioning things for you.

This is probably the simplest of all the commands, but it does have some interesting abilities, such as “auto approval” and the ability to increase or decrease the parallelisation of the execution.

In Summary (TL;DR)

Terraform is a state driven engine that allows us to provision cloud infrastructure easily and consistently.

Terraform utilises code known as HCL (Hashicorp Configuration Language)

HCL uses the keyword resource to define “resources” we wish to have provisioned in our Cloud.

HCL allows us to use the configuration/output from one resource as the input for configuration/attributes of another resource.

Terraform interfaces with different Cloud technologies using Providers.

When terraform runs it will parse the HCL files and build a graph of resources we want — known as Desired State

By “chaining” resources together, it enables Terraform to make explicit dependencies between objects in its graph.

Terraform stores knowledge about all the resources it has provisioned previously in a file known as its State file.

The contents of the State file are known as Perceived State — the state Terraform left the environment the last time it was run using our HCL files.

Terraform uses Backends to determine how State should be persisted.

There are many different Backends that can be used depending on the Cloud Service Provider we use and how we want Terraform to persist State.

When Terraform wants to provision a resource, and that resource exists in its Perceived State, it will interrogate the Cloud Provider to determine the Actual State.

If there are discrepancies between desired, perceived and actual states, Terraform will determine the corrective action required to bring actual inline with desired.

Terraform has 3 distinct lifecycle stages: init, plan and apply

If you rename a resource identifier, Terraform will act as if its a brand new resource its never seen before, and purge everything to do with resources relating to the old identifier.

<https://itnext.io/terraform-for-beginners-dd8701c1ebdd>

A data source in Terraform is a lot like a resource, but it only implements the “read” operation. So what does that actually mean? Let’s take a walk.

It’s like a query

Well, that was a short walk. A data source is a query, a means of getting data from the outside world and making it available to your Terraform configuration, perhaps to use in an input attribute of a resource or a submodule.

Take for example a Terraform configuration like the following (borrowed from the doco):

data "azurerm\_role\_definition" "example" {

name = "Contributor"

}

output "contributor\_role\_definition\_id" {

value = data.azurerm\_role\_definition.example.id

}

This can be thought of as equivalent to the following SQL query:

SELECT id AS contributor\_role\_definition\_id

FROM azurerm\_role\_definitions

WHERE name = "Contributor";

In this example, we’re sending the data from the data source directly to an output, but you could also send it (for example) into a role assignment resource:

resource "azurerm\_role\_assignment" "example" {

scope = data.azurerm\_subscription.primary.id

role\_definition\_id = data.azurerm\_role\_definition.example.id

principal\_id = data.azurerm\_client\_config.example.object\_id

}

/\* Other data sources omitted... borrowing them from the doco is left as an exercise for the reader... \*/

**OUTPUT.TF**

In Terraform, **outputs.tf** is a file used to define output values that you want to expose from your infrastructure code. Output values allow you to extract information or data from your deployed infrastructure for reference, reporting, or for use in other parts of your system.

***Basically the values which would be created only after creation (like IP address) and to use them further.***

Here's how to use **outputs.tf**:

1. **Create an outputs.tf File**: You typically create a file named **outputs.tf** in the same directory as your Terraform configuration files (**.tf** files). This file is where you define the output values you want to expose.
2. **Define Output Blocks**: Within **outputs.tf**, you define output blocks that specify the values you want to expose. An output block has a name and an expression to compute the value. For example:

output "instance\_ip"

{

value = aws\_instance.example.public\_ip

}

In this example, the output block defines an output named "instance\_ip," and its value is set to the public IP address of an AWS EC2 instance created in the Terraform configuration.

1. **Retrieve Output Values**: After applying your Terraform configuration, you can use the **terraform output** command to retrieve the values you defined in **outputs.tf**. For example:

terraform output instance\_ip

**MODULES**

Module is nothing but a function which can be called in another resource creation. Like we want to create a function app with a storage account, so for that storage account you can call the module. If we want to create an AKS, Data lake or anything for which we require a storage account, we can create a module of a storage account and call it inside the main.tf

In many ways, Terraform modules are similar to the concepts of libraries, packages, or modules found in most programming languages

Module “sqldb” {

Source = ‘/SDO/sql’

Name = ‘mywerk-emea-prod-001’

} -----🡪 This is how you call module

Module declaration is normal like main.tf and variable.tf

The only thing that matters is that you correctly mention source repo name

**COUNT VS FOREACH (.tf file is in repo)**

If your instances are almost identical, count is appropriate. If some of their arguments need distinct values that can't be directly derived from an integer, it's safer to use for\_each.

Because in for\_each we have key as well as value