What is Terraform?

HashiCorp Terraform is an Infrastructure as Code (IaC) tool to help you more reliably deploy and manage infrastructure deployments across one or more clouds, or even on-premise. As with all IaC tools, [Terraform](https://www.terraform.io/) allows you to write code that is used to build, change, and version your infrastructure deployments in a reliable and efficient manner.

There are two main types of Infrastructure as Code (IaC); declarative and imperative. Terraform belongs to the declarative type of IaC where you write a template file, or in the case of Terraform it’s one or more .tf Terraform Configuration Language files. These files declare what the end result of the infrastructure deployment is supposed to be, then when you run the deployment with the Terraform tool, it will make the necessary changes or updates to your infrastructure to meet the desired definition that’s been declared. In comparison, an imperative approach to IaC would be using traditional command-line (CLI) scripts that perform all the necessary steps in order as written. Declarative IaC is more efficient and reliable, since you do not need to know what state the infrastructure is currently in when you write the IaC code, and the tooling will figure out what changes / updates to make when deployed.

The Terraform code written can be used to declare (or define) all the infrastructure components needed for a given workload. This workload could be a single application, or even a larger scale system such that comprised of multiple microservices, databases, virtual machines, and other resources. The infrastructure components of the workload will be low-level infrastructure like networking, storage, and compute resources; while it may also include high-level infrastructure like Platform as a Service (PaaS) compute and database services, or a number of other components.

Why use Terraform?

All the different cloud platforms offer their own IaC tooling, like Microsoft Azure ARM (Azure Resource Manager) Templates. If you are needing to automate the deployment and configuration of infrastructure within multiple environments or may want to move to a different environment later, then using platform specific IaC tooling would require you to write completely different code for each platform. Terraform offers the ability to use a single tool that can integrate with multiple platforms in order to consolidate all your IaC code into a single tool and format.

There are a few more benefits that Terraform brings to your Infrastructure as Code (IaC) code development:

* **Deploy with ease** – Ability to deploy / redeploy infrastructure more easily using a declarative, high-level language to define your infrastructure configuration.
* **Platform agnostic** – The only IaC tool that is platform agnostic and allows you to use a single IaC code base to target infrastructure deployments across any cloud provider, or even multiple cloud providers at once.
* **Source control & versioning** – With IaC code written in source code files, the infrastructure deployment code can be committed to a Git or other source control system and versioned similarly to any other application code.
* **Change plan validation** – Terraform enables you to create a change plan (via terraform plan command) that allows you to verify the infrastructure changes that will be made, before running the deployment and applying those changes. This can help increase the reliability of your infrastructure deployments.

Terraform Terms

When getting started with Terraform, there are a few terms that are important to know. These will help you when navigating Terraform files, the documentation, and other resources when working with your Terraform projects.

Here are a few helpful terms that are important to understand for all DevOps Engineers, or Site Reliability Engineers (SREs):

* **Terraform File**– A Terraform file (.tf) that contains Terraform configuration language code.
* **Terraform Project**– A directory / folder that contains one or more Terraform files that are all a part of the same infrastructure deployment.
* **Resources** – The primary construct in Terraform. These are the things that will be provisioned, like Storage Accounts, Virtual Machines, Virtual Networks, etc.
* **Providers** – The “plugins” that enable Terraform to know about and work with resources within Microsoft Azure or another cloud provider.
* **Variables** – A name given to a value so it can be referenced in one or more locations within a Terraform file. Similar to a Variable in any other programming language.
* **Input Variables** – Values that are passed into your Terraform when running a deployment. These can be passed in using the Terraform CLI, or as Environment Variables.
* **Module** – Essentially a directory / folder of Terraform files that are re-used across multiple Terraform deployment project.

There are obviously a few other terms in Terraform. However, these are the primary terms that you will need to know when getting started with Terraform automation, as well as terms you will be using for a long time to come.

**FYI, this article covers that very basics of using Terraform to manage Microsoft Azure resources. Features like variables, input variables, modules, and other are outside the scope of this introductory article. Once you learn the basics of using Terraform you’ll be able to build on this foundation to explore the other more advanced features and capabilities of Terraform.**

Terraform Files and Directory

When writing Terraform automation for deploying resources, the code is written using the HashiCorp Configuration Language (HCL) and saved in files with a .tf file extension. You can add all your Terraform code to a single .tf file, or even break it out into multiple files that all reside within the same directory / folder.

**In short, a Terraform project consists of a directory / folder with one or more**.tf**files within it. The tooling will combine these files when generating a deployment plan or applying an infrastructure deployment.**

When Terraform commands are run, the tooling will look at the contents of all the .tf files within the folder. This enables you to separate out and organize the Terraform code for your infrastructure deployment in a manner that makes sense for you, your team, and your organization.

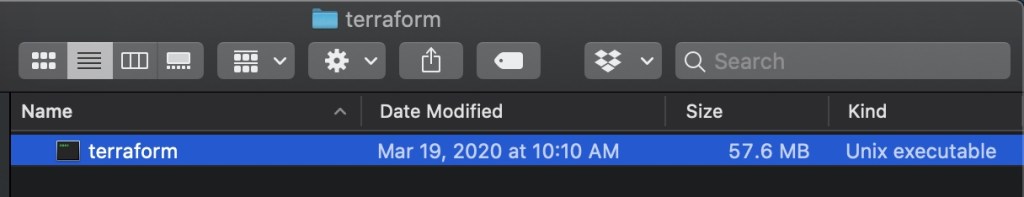
When getting started with Terraform, it’s great to start by writing all the Terraform code in a single Terraform (.tf) file. A general standard is to name this file main.tf. This is something that works great for learning, testing, and for smaller infrastructure deployments.

When an infrastructure deployment starts growing in size, and becoming more complex, it’s a good best practice to break out the Terraform code into multiple files. The following file names are a common pattern used across the community that uses Terraform:

* main.tf – Define locals (aka variables used within the deployment), call modules, and data sources to create all resources. Generally, this contains these primary components of the Terraform deployment.
* providers.tf – Add the provider definitions for the Terraform Providers your deployment is using (such as azurerm). Often times, this file may be combined within the main.tf file instead of having a separate file since often only one or a couple providers will be in use.
* variables.tf – Define the variables used by the Terraform deployment. These are the “input parameters” that will be passed into the Terraform code at deployment time.
* outputs.tf – Define variables to output from the Terraform project or module.
* {customname}.tf – Your own custom Terraform files. You can optionally separate your Terraform project code into any number of Terraform files.

Basic Terraform CLI Commands

Terraform commands are called using the Terraform CLI utility that can be [downloaded](https://www.terraform.io/downloads.html) locally. Terraform is distributed as a single binary, you simply unzip the downloaded executable (**for Windows, macOS, or Linux**) and run it from your local file system. This Terraform executable (terraform.exe on Windows) is the CLI (command-line interface) tool that you will use to run / perform all the various Terraform commands that are exposed.

macOS Finder showing ‘terraform’ executable

Obviously, Terraform includes a number of commands for initializing, planning, deploying, and performing other actions with your deployments and terraform files.

To get started, there are really only a few basic Terraform CLI command that you will need to know:

* terraform init – initialize the current directory of Terraform files
* terraform plan – process the Terraform files and create an execution plan for what infrastructure needs to be created or updated according to the Infrastructure as Code contained within the Terraform files.
* terraform apply – take the execution plan an apply it to perform all the creation and updating of infrastructure resources, and save the new state of infrastructure in the Terraform state file.
* terraform destroy – destroy or delete all the infrastructure that was previously deployed (created or updated) for the Terraform project. Basically, tear it all down!
* terraform -h – when is doubt on Terraform CLI commands, get some help.

Let’s take a little deeper look into these commands:

### **Initialize Terraform (terraform init)**

Before you can plan or apply your infrastructure deployment using Terraform you must Initialize your working directory of Terraform configuration files. This will always be the first command you run after writing new Terraform configuration files, or even cloning existing files from a source control repository. This command is completely safe to run multiple times as it doesn’t change any infrastructure when run.

The command is used in the following format:

terraform init

This command will look at your Terraform configuration (.tf) files, and initialize the Terraform providers that are declared in the Terraform files. The plugins for the Terraform providers defined within the Terraform configuration files (such as the azurerm Terraform provider for Azure Resource Management) will then be downloaded into the .terraform/plugins folder that sits at the root of the main folder containing your Terraform configuration files.

chris@Azure:~$ terraform init

Initializing the backend...

Initializing provider plugins...

- Checking for available provider plugins...

- Downloading plugin for provider "azurerm" (hashicorp/azurerm) 2.19.0...

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.

Once the terraform init command has been run successfully, then you will be able to move on to using other Terraform commands for managing your infrastructure as defined in the Terraform configuration files.

### **Generate Terraform Plan (terraform plan)**

Once the terraform init command has been run to initialize your directory of Terraform configuration (.tf) files, you will then be able to run the terraform plancommand to generate an execution plan for your Terraform infrastructure as code.

terraform plan

This command will look at your Terraform configuration files (.tf), and your already provisioned resources based on previous deployments using terraform apply command and it will figure out what changes need to be made. This plan of Terraform changes to make the deployed resources match what’s configured in the Terraform configuration files will be output to the terminal so you can inspect it.

When the Terraform plan is output to the terminal, it will use the +, -, ~ symbols to indicate what type of change to the existing infrastructure Terraform will be performing when terraform apply is run on your environment. This lets you see what the plan will be modifying from creating new resources, destroying / deleting resources, or making certain updates to existing resources. This isn’t just shown on the resource level, but will also show the particular property configuration changes for each of the resources managed by the Terraform code as well.

+ create

- destroy

~ update

The Terraform plan is what you will use to determine if the Terraform code will be performing the desired changes to your infrastructure that are intended. You can use this to inspect the changes that will be applied before making the changes so you can be sure it will not inadvertently break your environment. Essentially, this lets you double check there will not be any unintentional infrastructure changes made, and allows you to fix your Terraform code accordingly before actually applying, or deploying, the infrastructure changes.

Here’s a partial example of the Terraform Plan output to the console by running the terraform plan command. This example uses the Terraform code examples found below in this article.

chris@Azure:~$ terraform plan

Refreshing Terraform state in-memory prior to plan...

The refreshed state will be used to calculate this plan, but will not be

persisted to local or remote state storage.

------------------------------------------------------------------------

An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

# azurerm\_resource\_group.b59 will be created

+ resource "azurerm\_resource\_group" "b59" {

+ id = (known after apply)

+ location = "northcentralus"

+ name = "b59-rg"

}

# azurerm\_storage\_account.example will be created

+ resource "azurerm\_storage\_account" "example" {

+ access\_tier = (known after apply)

+ account\_kind = "StorageV2"

+ account\_replication\_type = "GRS"

The Terraform Plan can also be output to a file using the -out flag on the terraform plan command. This allows you to save the Terraform plan to a tfplan file that can be used later on to run a terraform apply for to deploy the infrastructure changes.

terraform plan -out=deploy.tfplan

Saving the Terraform Plan file (tfplan) is useful especially when you are integrating Terraform into some kind of automated build and release pipeline (like CI/CD). This allows you to have one pipeline build the Terraform Plan, and a second pipeline perform the Terraform Apply to deploy out the changes. This is a common tactic employed by DevOps Engineers and Site Reliability Engineers (SREs) saving the Terraform configuration files in version control like Git and then implementing an automated CI/CD deployment pipeline using tools like Azure DevOps, GitHub Actions, Jenkins, or other tools.

### **Apply Terraform Resources Changes (terraform apply)**

The terraform apply command is used to go through the Terraform configuration files in your current directory, and apply the necessary changes to deploy the declared infrastructure state to your environment. This is the command that deploys your infrastructure and makes changes to your environment.

terraform apply

Here’s an example terminal output from the terraform apply command showing the status of applying the infrastructure changes, including the eventual “Apply complete!” success message. This will look at the Terraform configuration files (.tf) within the current directory and apply the defined infrastructure by making the necessary changes to your environment.

chris@Azure:~$ terraform apply

azurerm\_resource\_group.b59: Creating...

azurerm\_resource\_group.b59: Creation complete after 1s [id=/subscriptions/<azure-subscription-id>/resourceGroups/b59-rg]

azurerm\_storage\_account.example: Creating...

azurerm\_storage\_account.example: Still creating... [10s elapsed]

azurerm\_storage\_account.example: Still creating... [20s elapsed]

azurerm\_storage\_account.example: Creation complete after 21s [id=/subscriptions/<azure-subscription-id>/resourceGroups/b59-rg/providers/Microsoft.Storage/storageAccounts/b59storage]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

The state of your infrastructure has been saved to the path

below. This state is required to modify and destroy your

infrastructure, so keep it safe. To inspect the complete state

use the `terraform show` command.

State path: terraform.tfstate

You can also, optionally, pass in a pre-generated Terraform plan file that was created using the terraform plan -out=tfplan command. This allows you to take the previously generated Terraform plan and use that plan to perform the necessary changes to your infrastructure deployment.

# Format

terraform apply [tfplan filename]

# Apply using existing tfplan file

terraform apply deploy.tfplan

After the terraform apply command is finished executing it will save a .tfstate file. This file contains the Terraform state information for the Terraform-managed infrastructure. This state information is largely how Terraform knows what’s already been deployed and being managed by Terraform. It is also used as an input to both the plan and apply commands for Terraform to be aware of existing infrastructure as it prepares the plan of what necessary changes to make to your environment.

### **Remove Terraform Resources (terraform destroy)**

While the terraform apply command is used to apply infrastructure configurations to your environment, the terraform destroy command is used to destroy (or delete) Terraform-managed infrastructure. This command will take into account the .tfstate file for your environment and go through the process of deleting the resources that were previously deployed.

terraform destroy

**The**terraform destroy**command used to delete your Terraform-managed resources from your environment. Be especially careful using this command in Production environments as it can be easy to delete critical resources your organization may need; such as databases or virtual machines.**

With the most basic usage, terraform destroy will destroy all the Terraform-managed resources in your environment. This will essentially delete your entire environment. However, if you wish to only delete specific resources from your environment, then you can use the -target flag with the command to explicitly tell Terraform what to destroy. The use of this flag is outside the scope of this article, but definitely something great to be aware of.

### **Terraform Help (terraform -h)**

You can run terraform by itself to get an output of all the supported commands. Also, you can run terraform with the -h flag to get help for any specific command.

terraform

# or

terraform -h

Here’s an example of the terraform -h output of help information showing the list of common commands that are supported by Terraform:

chris@Azure:~$ terraform

Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.

The most common, useful commands are shown first, followed by

less common or more advanced commands. If you're just getting

started with Terraform, stick with the common commands. For the

other commands, please read the help and docs before usage.

Common commands:

apply Builds or changes infrastructure

console Interactive console for Terraform interpolations

destroy Destroy Terraform-managed infrastructure

env Workspace management

fmt Rewrites config files to canonical format

get Download and install modules for the configuration

graph Create a visual graph of Terraform resources

import Import existing infrastructure into Terraform

init Initialize a Terraform working directory

login Obtain and save credentials for a remote host

logout Remove locally-stored credentials for a remote host

output Read an output from a state file

plan Generate and show an execution plan

providers Prints a tree of the providers used in the configuration

refresh Update local state file against real resources

show Inspect Terraform state or plan

taint Manually mark a resource for recreation

untaint Manually unmark a resource as tainted

validate Validates the Terraform files

version Prints the Terraform version

workspace Workspace management

All other commands:

0.12upgrade Rewrites pre-0.12 module source code for v0.12

debug Debug output management (experimental)

force-unlock Manually unlock the terraform state

push Obsolete command for Terraform Enterprise legacy (v1)

state Advanced state management

### **Terraform Command Help (terraform [command] -h)**

Help can be retrieved for specific Terraform commands using the following format:

terraform [command] -h

Simply replace the [command]placeholder with the Terraform command you wish to get help for. This help information for the command will then be output to the terminal for you.

Here’s an example of the output for getting help with the terraform apply command:

chris@Azure:~$ terraform apply -h

Usage: terraform apply [options] [DIR-OR-PLAN]

Builds or changes infrastructure according to Terraform configuration

files in DIR.

By default, apply scans the current directory for the configuration

and applies the changes appropriately. However, a path to another

configuration or an execution plan can be provided. Execution plans can be

used to only execute a pre-determined set of actions.

Options:

-auto-approve Skip interactive approval of plan before applying.

-backup=path Path to backup the existing state file before

modifying. Defaults to the "-state-out" path with

".backup" extension. Set to "-" to disable backup.

-compact-warnings If Terraform produces any warnings that are not

accompanied by errors, show them in a more compact

form that includes only the summary messages.

-lock=true Lock the state file when locking is supported.

-lock-timeout=0s Duration to retry a state lock.

-input=true Ask for input for variables if not directly set.

-no-color If specified, output won't contain any color.

-parallelism=n Limit the number of parallel resource operations.

Defaults to 10.

-refresh=true Update state prior to checking for differences. This

has no effect if a plan file is given to apply.

-state=path Path to read and save state (unless state-out

is specified). Defaults to "terraform.tfstate".

-state-out=path Path to write state to that is different than

"-state". This can be used to preserve the old

state.

-target=resource Resource to target. Operation will be limited to this

resource and its dependencies. This flag can be used

multiple times.

-var 'foo=bar' Set a variable in the Terraform configuration. This

flag can be set multiple times.

-var-file=foo Set variables in the Terraform configuration from

a file. If "terraform.tfvars" or any ".auto.tfvars"

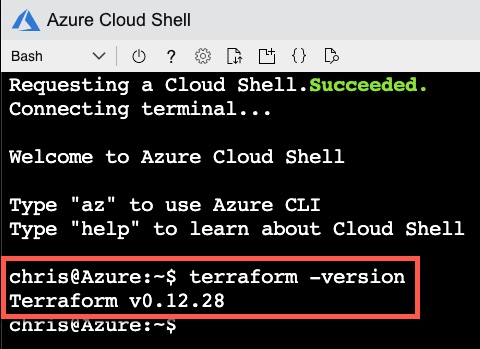
files are present, they will be automatically loaded.

Alternatively, you can always look up Terraform commands using the [Terraform official documentation](https://www.terraform.io/docs/commands/index.html) as well.

## Azure Cloud Shell + Terraform

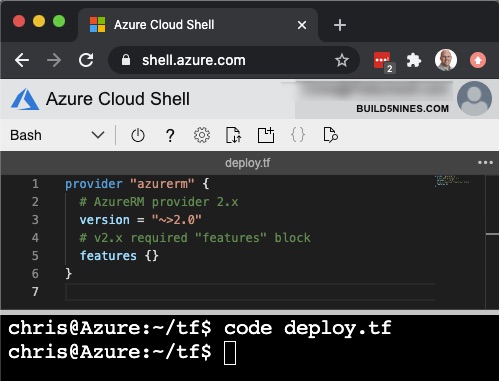
The [Azure Cloud Shell](https://build5nines.com/azure-cloud-shell-tips-and-tricks/) provides an excellent environment for working with Terraform code. The Cloud Shell includes built-in support the terraform, in addition to a built-in code file editor (via code command). Plus, when you open up the Azure Cloud Shell, it automatically authenticates the built-in support for the Azure CLI (az) against your Azure Subscription based on your Azure Cloud Shell logged in session. All these features provide a really great environment to work with Terraform code from any machine without any tool installation necessary.

To check the version of Terraform installed and available for you in the Azure Cloud Shell, you can run the terraform -version command. This is shown in the below screenshot.

Screenshot: Azure Cloud Shell checking Terraform version

Editing files using the built-in code editor inside the Azure Cloud Shell is really easy using the code command. Plus it includes syntax highlighting for Terraform .tf files too! To open a file (.tf or any other) simply type the code command followed by the file name. You can use this to both edit existing files, or even create new files by opening the code editor to the desired file, then saving its contents.

Here’s a screenshot that shows the Azure Cloud Shell editor editing a file by using the code deploy.tf command to edit/create the file:

Screenshot: Edit Terraform .tf file using Azure Cloud Shell

**Tips & Tricks: We recommend you read the “**[**Azure Cloud Shell Tips and Tricks**](https://build5nines.com/azure-cloud-shell-tips-and-tricks/)**” article written by Chris Pietschmann to get many more amazing tips to help you be much more productive using the Azure Cloud Shell with Terraform!**

## Azure Provider for Terraform

Terraform uses a “plugin” style model for adding providers for handling the communication with various infrastructure APIs. This enables the ability for your own Terraform code to pull in the specific Terraform Providers you need to work with your infrastructure deployments. For working with Microsoft Azure infrastructure from Terraform, the azurerm provider is used to code against the Microsoft Azure Resource Manager (ARM) REST APIs.

The following block is added to the Terraform code within your .tf files to tell Terraform that you are going to use the Azure Resource Manager azurerm provider to work with / manage Azure resources:

provider "azurerm" {

# AzureRM provider 2.x

version = "~>2.0"

# v2.x required "features" block

features {}

}

Whether you separate your Terraform code into multiple .tf files, or include it all within a single file, you will only include the provider declaration once. For easy readibility of a single .tf file, you could place it at the top of the file. Or, in a multiple .tf file project, the standard is to place this in a file named providers.tf.

### **Authenticate Terraform with Azure**

When using the Azure Provider for Terraform, at the most basic level it utilizes the Azure CLI login context to connect to and authenticate against your Azure Subscription. This keeps it simple by not adding a complicated Terraform to Azure authentication method. It also prevents you from saving any username/password combination in a configuration file that would be less than secure. As a result, this method will require you to have the Azure CLI installed on your machine you’re running Terraform deployments from.

To authenticate with Azure and set the login context to your desired Azure Subscription, use the following Azure CLI commands:

# Login to Azure

az login

# List all Azure Subscriptions your account has access to

az account list

# Set Azure CLI context to a specified Azure Subscription

# Replace placeholder with the Azure Subscription ID

az account set -s "azure-subscription-id"

# Display the selected Azure Subscription to verify your

# context is set to the desired Azure Subscription

az account show

Once you have used the Azure CLI to login and set the context to your desired Azure Subscription, you can then start running Terraform deployments against that Azure Subscription.

When integrating CI/CD pipelines to run your Terraform deployments, there are a couple additional options that can be utilized for authenticating against Azure in a headless (non-interactive) manner:

* **Authenticate to Azure using Managed Identity** – This method requires you to setup a Managed Identity within Azure that will be used to authenticate so an automated process running Terraform has its own identity and permissions.
* **Authenticate to Azure using a Service Principal and Client Certificate or Secret** – This method requires you to setup a Service Principal identity within Azure AD for the automated process running Terraform to authenticate with. You can setup your process to use either a Client Secret or Client Certificate as means of authenticating with the Azure AD Service Principal identity.

For the duration of this article, and when using Terraform for learning and testing purposes, it is recommended to use the Azure CLI authentication method. This is the simplest authentication method to use on your local machine, and can also be utilized through the Azure Cloud Shell in your browser as well.

## Azure Terraform Example – Resource Group and Storage Account

Next, let’s take a look at some sample Terraform code using the Azure Resource Manager (azurerm) Terraform Provider to create an Azure Resource Group, and then an Azure Storage Account within that Resource Group. All infrastructure deployments in Microsoft Azure will use Resource Groups, and most will also use Azure Storage Accounts. The following examples and introduction will give you a better understanding of the basics of using Terraform to manage Microsoft Azure resources.

### **Create Azure Resource Group**

Here’s a simple example of Terraform code that uses the azurerm\_resource\_group Azure RM resource type to specify the IaC for deploying an Azure Resource Group.

# Create a resource group

resource "azurerm\_resource\_group" "b59" {

name = "b59-rg"

location = "North Central US"

}

When running a Terraform deployment using this code, the Azure Resource Group will be created.

Notice that within the Terraform code for the azurerm\_resource\_group resource, the Terraform code is setting the name of the “Terraform resource” to b59. This name can be used to reference the resource in later Terraform code (such as the Azure Storage Account creation below) to be able to access properties / values of the resource for configuring other resources that are to be deployed.

For example, in other Terraform code, you can reference this Resource Group using the following format:

# Format

[Terraform Resource Type].[Terraform Resource Name]

# Specify the AzureRM Resource Group resource type

azurerm\_resource\_group.[Terraform Resource Name]

# Specify the specific Resource Group named "b59"

azurerm\_resource\_group.b59

Notice that to reference the Resource Group named b59, the Resource Group resource type (azurerm\_resource\_group) needs to be given this name. When declaring Terraform resources, the first value in quotes (") is the Terraform resource type, and the second value in quotes (") is the Terraform resource name.

Here’s an example of breaking this out:

# Format

resource "[Terraform Resource Type]" "[Terraform Resource Name]" {

...

}

# Specify the AzureRM Resource Group resource type

resource "azurerm\_resource\_group" "[Terraform Resource Name]" {

...

}

# Specify the Resource Group resource type to be named "b59"

resource "azurerm\_resource\_group" "b59" {

...

}

All resources defined within Terraform configuration files will be given a name that must be unique within you Terraform configuration (.tf) files. This uniqueness must be met across all the .tf files for your deployment whether you use a single .tf file or multiple. Also, the name is unique per the Terraform resource type being deployed. This Terraform Resource Type + Resouce Name are used to define each unique resource the Terraform configuration will be used to deploy and manage. It can also be used to setup resource dependencies between Terraform resources, so you can reference one resource when configuring another; as is seen below.

### **Create Azure Storage Account**

Here’s an example of Terraform code to create an Azure Storage Account using the azurerm\_storage\_account resource type. In this example the Terraform resource name for the Storage Account is set to b59storage, and the resource\_group\_name to organize the resource within Azure is referencing the Azure Resource Group created be the above example.

# Create an Azure Storage Account

resource "azurerm\_storage\_account" "b59storage" {

name = "b59storage"

# Place this in the Resource Group that's managed by this same

# Terraform code

resource\_group\_name = azurerm\_resource\_group.b59.name

# Provision this in the same Azure Region as the Resource Group's default location

location = azurerm\_resource\_group.b59.location

account\_tier = "Standard"

account\_replication\_type = "GRS"

tags = {

environment = "dev"

}

}

When running a Terraform deployment using this code, the Azure Storage Account will be created with the specified configurations; such as being placed within the Azure Resource Group that was previously created with the same deployment. Setting the azurerm\_storage\_account.resource\_group\_name property to the value of azurerm\_resource\_group.b59.name it is referencing the Azure Resource Group that was previously created and using it’s .name to tell the azurerm provider what Resource Group to place the Azure Storage Account within.

### **Full Terraform Code + CLI Commands**

Here’s the above examples all pulled together into single Terraform .tf file that uses the Terraform Azure Provider (azurerm) to create a new Resource Group, and provision an Azure Storage Account inside the resource group.

provider "azurerm" {

# AzureRM provider 2.x

version = "~>2.0"

# v2.x required "features" block

features {}

}

# Create a resource group

resource "azurerm\_resource\_group" "b59" {

name = "b59-rg"

location = "North Central US"

}

# Create an Azure Storage Account

resource "azurerm\_storage\_account" "b59storage" {

name = "b59storage"

resource\_group\_name = azurerm\_resource\_group.b59.name

location = azurerm\_resource\_group.b59.location

account\_tier = "Standard"

account\_replication\_type = "GRS"

tags = {

environment = "dev"

}

}

To run the Terraform deployment of Infrastructure as Code (IaC), you’ll need to save the above example in a .tf file, such as deploy.tf if you’re using a single file. Then, you can run the following commands to initialize, plan, and apply / deploy the infrastructure defined in your Terraform configuration to your Azure Subscription:

# Step 1: Login to the Azure CLI

az login

# NOTE: If running in the Azure Cloud Shell,

# you can skip this step.

# Step 2: Terraform Initialization

terraform init

# Step 3: Generate the Terraform Plan

terraform plan

# OPTIONAL: This step isn't required, but is useful

# depending on your workflow process

# Step 4: Apply the Terraform configuration, and

# make the necessary changes to deploy the infrastructure

# defined out to your Azure Subscription

terraform apply

**Related: If you need to manage multiple environment deployments with Terraform, then Feature Flags and Environment Toggles help make everything easier to manage. For an introduction into implementing these, we encourage you go to read the “**[**Terraform Feature Flags and Environment Toggle Design Patterns**](https://build5nines.com/terraform-feature-flags-environment-toggle-design-patterns/)**” article written by Chris Pietschmann.**

## Wrap Up

This article walked through all the main features and concepts of using Terraform to build declarative Infrastructure as Code (IaC) to create / manage Microsoft Azure resources. These concepts will provide you with a great starting point for working with Azure infrastructure using Terraform; whether you do it manually form your local machine or Azure Cloud Shell, or even if you take this further to implement Terraform infrastructure deployments from your CI/CD workflows using Azure DevOps Pipelines or GitHub Actions.