Day 0 Beginners Guide to Terraform

Release 0.1

Jesse Driskill

Dec 14, 2022

CONTENTS:

CHAPTER

ONE

DAY 0 BEGINNERS GUIDE TO TERRAFORM

1.1 Motivation

The most frustrating aspect of Terraform is the "Day One" learning curve. When first starting out most of the examples I found (by which I mean: every single one), seemed to assume that you understand the basic operation of Terraform and focused on the more complex use-cases. The problem with that is that I held certain misconceptions that none of those guides actually addressed. The most significant one was "How do I download the package (provider) to interact with a specific service?". The answer is "You don't", but all of the guides I found seemed to assume that knowledge and didn't address the topic. In hindsight I understand why, but it's been over two years and I still remember the intense frustration of that first day working with Terraform (TF).

This guide is intended to answer those very basic beginner questions, as well as walk you through the creation of your first (extremely) simple Terraform Run so you can see how the various pieces for together.

1.2 Organization

This guide is broken into several small parts intended to let you skip the frustration of discovering them yourself. It definitely does not replace the Terraform Documentation, but as you'll discover if you click the link, this guide will reduce the time you need to "internalize" how to use Terraform.

Sections 12-15 contains simple examples to illustrate how several Terraform concepts fit together. The code for each example is available not only on the example page, but in the corresponding …/example_*/ directory.

1.3 Terrform

As stated, this guide is a simple introduction to Terraform, and we will be going over several of the basic elements of a basic Terraform deployment. Those elements are:

1. *tl;dr Quick Start*
2. Providers
3. Registry
4. Configurations
5. Resources
6. Modules
7. Runs
8. Variables

9. Initialization
10. Execution: Plan, Apply, Destroy
11. Tips and Tricks
12. **Example 1**
    - Variables, local values, null_resource resource, and outputs
13. **Example 2**
    - Variables, local values with embedded values, outputs
14. **Example 3**
    - Module creation and usage, module outputs
15. **Example 4**
    - Module usage with ternary conditional and 'count' meta-argument

1.3.1 tl;dr Quick Start

1. Write your Terraform code
   - I describe this below; however, if you are actually starting with this know that I have experienced your pain and have no sympathy. Helping you avoid that pain is the whole purpose of this guide.
2. `terraform init`
   - Initialize your TF environment, which includes a basic syntax check.
   - NOTE: This will download the required providers and enumerate your modules.
3. `terraform plan`
   - More thorough syntax check
   - Determine order of operations based on dependencies
   - Logic check - checks for unmet dependencies, circular logic, missing variables, etc…
4. `terraform apply [--auto-approve]`
   - Deploy the configuration.
   - Communication/authentication/authorization issues will be caught at this stage.
   - Failures will *not* be rolled back automatically.
5. `terraform destroy [--auto-approve]`
   - Destroy/delete deployed objects
   - Reverses the order of operations determined at the time of deployment

1.4 Examples

1.4.1 Example 1

- Create variable
- Create local value using variable value
- Create *null_resource* to call a bash command
- Create Output blocks to print the values of the variable and local value

1.4.2 Example 2

- Create variable
- Create local values containing several embedded values using the variable to build the names
- Create Output blocks to print the variable and local values

1.4.3 Example 3

- Create and run a module that creates an Azure resource-group
- Create output blocks that print the values of the module

1.4.4 Example 4

- Example usage of the ternary conditional and the count meta-argument

NEXT

CHAPTER

TWO

PROVIDERS

2.1 Overview

A Terraform Provider is a plugin that Terraform calls in order to communicate with the service or resource being configured. For example, in order to configure Azure resources Terraform uses the azurerm provider. Providers are *usually* written by the organization that provides the product being configured, though that is not always the case. For example, the BIG-IP is provided by F5, but the Azure provider is provided directly by Hashicorp (the same company that created Terraform).

The provider(s) required for the Terraform Run are identified by Terraform when you run a `terraform init`, and are *automatically* downloaded. There are a two parts to provider configuration. The first declares that the Provider is required, and the second is the actual Provider configuration.

2.1.1 Provider Declaration

Here is an example of the Provider declaration:

```
terraform {
  required_providers {
    azurerm = { version = "3.14.0" }
  }
}
```

The 'terraform' block is not limited on only one Provider; more can be defined depending on what your Terraform Run requires. Here are a few example Provider configurations:

Azure

NOTE: Along with identifying and downloading the Provider, `terraform init` will also *upgrade* the provider if a newer one is available and the specific version isn't specified in the provider declaration. The example above defines the specific version of the provider to be used, so the release of newer versions will be ignored. That said, a common approach is to use a version statement similar to **'version ~> "3.14.0"**, and that syntax will result in the Provider being automatically upgraded whenever a new version is available. If that newer version includes changes to the options/arguments in any of the resources you are using you may find yourself having to refactor portions of your Terraform configuration to adjust to the new options. The best practice would be to use '=' to restrict the Provider to the specific version you are writing your code for. You can then allow the Provider to be upgraded when it is convenient for you, rather than potentially having to update a bunch of code that is unrelated to whatever you are actually trying to work on.

The *Provider* configuration blocks define changes to the default provider behavior. Even without any changes each provider must still have a configuration block. Here is a default configuration block for the Azure Provider:

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

Provider configuration blocks can be much more complicated. Here is a configuration block for the Azure subscriber that includes a few modifications to the default behavior:

```
provider "azurerm" {
  features {
    virtual_machine {
      # Necessary for license revocation when using BIG-IQ LM.
      graceful_shutdown                       = true
      delete_os_disk_on_deletion              = true
    }
    template_deployment {
      #  Allow the RG to be removed even if resources are still present
      delete_nested_items_during_deletion     = true
    }
    resource_group {
      # Allow the RG to be removed even if resources are still present
      prevent_deletion_if_contains_resources  = false
    }
  }
}
```

AWS

The Terraform AWS Provider configuration block shown here pulls the credentials and profile from local values (variables), and defines the default region for operations.

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

provider "aws" {
  shared_credentials_file   = local.creds_file
  profile                   = local.profile
  region                    = var.region
}
```

Notice that even the terraform configuration block differs from the Azure example. In this case the source of the provider is defined in addition to what version of the provider should be used.

GCP

And finally, the GCP Provider.

```
terraform {
  required_providers {
    google = {
      version = "4.40.0"
    }
  }
}

provider "google" {
  project                   = var.project
  region                    = var.region
  zone                      = var.zone
}
```

NEXT

BACK

HOME

CHAPTER

THREE

REGISTRY

Terraform Providers are retrieved from a Registry. The official Hashicorp Terraform registry is registry.terraform.io, which is used to download Providers for which a more specific source location is not provided. All of the examples shown on the Providers page will use this default registry when downloading the specified provider<s).

3.1 Documentation

In addition to making the providers available for download by Terraform, the registry is also where the official documentation for each provider is located. For example, the documentation for the Azure provider is located at https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs. The link above is for the latest version of the Provider but the documentation for earlier versions can be viewed with the dropdown at the top of the page.

NOTE: When creating or modifying a Terraform plan the documentation in the Registry is best documentation resource without question. It is very rare tht I have to look elsewhere for help with getting a Resource to work.

3.2 Example Configurations

All, or nearly all, Resources supported by a Provider are documented on the Provider documentation page. Further, nearly all of the examples show you not just the usage of the specific resource but also include examples of any pre-requisite resources necessary to make use of the Resource. For instance, the example on the azurerm_linux_virtual_machine page doesn't just show examples of that particular resource, the page includes examples for every resource necessary to use it. The result is that in addition to an example the page contains at least a minimal example of these resources:

- azurerm_resource_group
- azurerm_virtual_network
- azurerm_subnet
- azurerm_network_interface

All of which are pre-requisites to creating a azurerm_linux_virtual_machine resource.

All of the resource configuration options (input) and attributes are shown below the examples.

NOTE: When having a problem working with a Resource it is frequently helpful to take a look at pages that require that resource. The configuration of other Resources that require the one you are working with will include their own examples, and sometimes those examples enhance the examples provided in the primary Documentation page. This situation isn't common, but I have encountered it enough times that this is one of the first things I do when I have a Resource that repeatedly fails to deploy with some cryptic error message.

NEXT

BACK

HOME

CHAPTER

FOUR

CONFIGURATIONS

A Terraform Configuration is a block of code that *declaratively* describes how your infrastructure should be configured. In it's very simplest form a Terraform run can consist of only a single Configuration. A Terraform Run consists Configuration blocks; sometimes many, many configuration blocks. During the initialization process Terraform reads through all of the Configuration blocks and determines the dependency order, which dictates the order in which each configuration block is acted upon. Configuration blocks do not contain executable code, Terraform is not a programming language. That said, it is sometimes helpful to think of Configuration blocks as being the equivalent of commands in a programming language.

Here is a very simple example Configuration block that uses the null_resource resource to call a command in bash:

```
resource "null_resource" "test" {
  provisioner "local-exec" {
    command = "echo \"Hello world\""
  }
}
```

In this examle the Configuration consists of a Resource of the type *null_resource*, but it just as easily could have been a call to a Module, as shown here:

```
module "rg" {
  source                        = "./modules/resource_group"
  rg                            = local.rg
}
```

Or an Output block:

```
output "my_out" {
  value  = "This is the output - user: ${local.data.admin_user}"
}
```

Even a Variable are a type of configuration and defined in the same way:

```
variable "prefix" {
  default = "my_prefix"
}
```

Configuration blocks can become quite complex, as you'll see as soon as you take a look at the configuration for deploying a VM into any cloud. I'm not providing an example of a complex configuration block because that isn't a Day-1 topic, but if you are curious here is the documentation for the azurerm_linux_virtual_machine resource in Azure. This example is just about as simple as it gets for deploying a VM in Azure.

NEXT

BACK

HOME

CHAPTER

FIVE

RESOURCES

5.1 Overview

A Terraform Resource is a block that describes an infrastructure object. For example, you may have a "resource" that describes a virtual-machine in Azure. The resource would describe everything about that VM, like the number of CPU cores, amount of memory, disk size, and number of interfaces. Terraform will send that resource definition to the appropriate Provider so that the described object can be created.

5.2 Example

Here is a very simple example of a Resource block:

```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

5.3 Official Documentation

The official Terraform documentation for Resource blocks can be found here

Resources are easily the most common element of Terraform configuration object. All of the resources supported by a particular provider are described in the provider documentation. For example, here is the documentation for the azurerm_linux_virtual_machine resource.

Note: If you review that documentation you'll notice that the example near the top starts with the creation of the a virtual-network, then a subnet, then an interface, and *finally* an example of the resource we are actually talking about. This is a fairly common theme in Terraform documentation. It is very common for Terraform examples to include every other resource required by the object in question.

NEXT

BACK

HOME

CHAPTER

SIX

MODULES

A Terraform Module is a collection of Terraform Configurations that manage a particular type of infrastructure resource. For example, you may write a module to configure an Azure Resource-Group, or a module to provide configure the virtual-network and subnets that a BIG-IP is deployed to. For example, you may have a module called "virtual_network" that configures an Azure Virtual Network, then call that same module to create a client network, server network, and BIG-IP network. The code to create a virtual-network is only written once, as a module, but then executed multiple times with different input variables.

Modules are to Terraform what library files are to programming languages. They allow you to configure one or more elements, and then can be re-used multiple times in the same Terraform Run to configure those elements multiple times.

Demonstrating Terraform Modules is beyond the scope of this guide, but I may add a simple example if there is demand.

NEXT

BACK

HOME

CHAPTER

SEVEN

RUN

A Terraform Run consists of one or more Resources. Resources are what will actually happen when you enter *terraform apply* to deploy your configuration.

Terraform Resources are blocks of Infrastructure-as-Code (IaC) elements. These define what the deployment *should* be and leave it up to the Terraform logic and the provider to make happen. While several aspects of Terraform seem quite similar to a programming language, what you are actually "coding" is how things should be. When you run *terraform apply*, you are telling Terraform to make them that way.

NEXT

BACK

HOME

CHAPTER

EIGHT

VARIABLES

Variables are just that. They are provided to the various Configurations in lieu of static values to allow for simpler description of what should be deployed. That said, Variables in Terraform are a source of frequent confusion and much gnashing of teeth.

There are two basic types of varibales: Input Variables and Local Values.

8.1 Input Variables

Input variables are provided via configuration files, on the CLI, or pulled from environment variables. These are immutable (i.e. cannot be changed once defined), and they are defined when Terraform is executed. Dynamic values cannot be used, nor can one input variable receive its value from another input variable. These are static, immutable, and defined at run-time.

8.2 Local Values

Local values are also immutable, however these can receive dynamically generated values rather than static values. Unlike Input variables, local values can use other variables as part of the value.

8.3 Variable Usage

Here is a simple example illustrating the defintion of both:

```
variable "prefix" { default = "my_prefix" }
locals { vnet_name = "${var.prefix}-vnet" }
```

The variable "prefix" would contain the value *my_prefix*, and the local value "vnet_name" would contain the value *my_prefix-vnet*. However, you would *not* be able to define input variable that leverage the existing value of the *prefix* variable.

> Tip: All Input variable names are referenced using the 'var.' prefix, and all local values are referenced using the 'local.' prefix.

8.4 Defining Variables

The number of variables in a moderately complex Terraform Configuration can be quite large. For this reason there are several methods of breaking up your variables into different files. The benefit of this approach is that you can isolate the variables that apply to one group of objects in their own file, making it easier to find them if/when you decide to change something.

By default Terraform will source certain files for variable definitions: * vars.tf * variables.tf * terraform.tfvars * terraform.tfvars.json

Additional files can be specified on the CLI using the *-var-file* option as shown here:

```
terraform apply -var-file="my_variables.tfvars"
```

That syntax applies to files with the following extensions: * .tfvars * .tfvars.json

To simplify the import of additional variable files you can insert ".auto" between the filename and the ".tfvars" or ".tfvars.json" extension. Naming them this way removes the need to specify the filenames using the '-var-file' CLI option.

Important Take-away

The names of variables defined in *.tfvars or *.auto.tfvars files must be pre-defined in one of the automatic variable files. For example, if I have a variable called "bigip" in a file called "v_bigip.auto.tfvars", an empty "placeholder" variable defining that name must be located in one of the files that is imported automatically. Here is an example of how I would import a variable file called "v_bigip.auto.tfvars" containing a variable called "bigip":

vars.tf

```
variable "bigip"   {}
```

v_bigip.auto.tfvars

```
bigip = {
  name    = "my_ltm_01"
}
```

Terraform would recognize a variable called "bigip" containing a nested variable called "use_paygo" with a string value of "my_ltm_01". If the placeholder variable is not present the variable(s)`_ within the v_bigip.auto.tfvars file would not be included. You would also get a runtime error when Terraform attempts to import the variables define in the "v_bigip.auto.tfvars" file.

NEXT

BACK

HOME

CHAPTER

NINE

INITIALIZATION: *TERRAFORM INIT*

When you run 'terraform init' Terraform will read all of the configuration files in your current directory and download all of the required *providers* for you. These will be placed in a directory called '.terraform/providers/...'. It will also identify every module used in your configuration and write the list to '.terraform/modules/modules.json'.

When you run 'terraform init' Terraform reads every file that ends with '.tf' in the current directory and looks for a block called 'terraform', which contains a sub-block called 'required_providers'. The 'required_providers' block contains a list of every 3rd party "provider" required by your terraform configuration. It does not need to include default terraform providers - only 3rd party providers that provide additional functionality.

Important Take-away The first thing to understand here is that, other than the Terraform package itself, **nothing is manually downloaded**. Terraform will automatically download the components it requires when it is *initialized*. (I spent a humiliating amount of time figuring this out.) The second thing to understand is that Terraform initialization *is not the first step* in creating a new plan/deployment. Counter-intuitively, terraform initialization takes place after you've written all of the code and right before you actually try to run the plan to do something.

Note: Running *terraform init* does not verify that your code is without flaws or actually does anything at all. It *does* perform a basic syntax check, but that's it.

NEXT

BACK

HOME

CHAPTER

TEN

EXECUTION: PLAN, APPLY, DESTROY

terraform plan is used to perform an exhaustive check of your code, very similar to a 'dry-run' option. It will check the syntax, variable assignments, and attempt to identify circular references. <Circular references are when one resource depends on another, but that other object depends on the first one. Sometimes the dependency chain can include three or four objects, making identification of the circular reference difficult.>`_

Performing a *terraform plan* is always recommended prior to actually applying the configuration, and will most likely become an internalized part of your deployment process.

terraform apply is used to actually apply your configuration. It goes through the same process as terraform plan, but this time the changes are actually implemented. As the 'apply' is executed Terraform will be writing completed actions to the 'terraform.state' file. The 'terraform.state' file contains state information about the current, successfully-deployed configuration. Only resources that are *successfully* deployed are written to the state file. This file is then used for if Terraform is called again to compare the current deployment with a changed configuration to determine what it actually has to do.

For example, if you deployed a BIG-IP and a server in a previous 'terraform apply', then change something about the server configuration, Terraform will use the 'terraform.state' file to determine what it actually needs to do to make the deployed configuration match your IaC configuration. If the change to the servers configuration in Terraform doesn't impact the BIG-IP, then the server would be redeployed with the new configuration while the BIG-IP wouldn't be modified.

terraform destroy is used to destroy/delete resources that have been deployed. The 'destroy' command will use the terraform.state file to remove objects in the opposite order of their deployment. The intent is to remove resources that have dependencies on other objects before attempting to remove those other objects. This process does work quite well, thoough it is not uncommon for an object in a public cloud to be 'marked for deletion' without having actually been deleted when Terraform attempts to delete the object it is dependent on. In these cases the 'destroy' operation will fail and you will need to run 'terraform destroy' again.

The scenario above occurs frequently enough that when dealing with certain public clouds I start out by running 'terraform destroy –auto-approve' three times, separated by semicolons so that the second and third calls will be occur immediately. Calling 'terraform destroy' when nothing is deployed doesn't cause problems because the 'destroy' command updates the 'terraform.state' file as each resource is destroyed, so subsequent calls only act on resources that are still deployed.

```
terraform destroy --auto-approve; terraform destroy --auto-approve; terraform destroy --
↪auto-approve
```

With a Bash alias (see Tips and Tricks) that command can be reduced to:

```
tfda; tfda; tfda
```

NOTE: The 'terraform apply' and 'terraform destroy' commands both require interactive approval before actually making any changes. To bypass the interactive approval use the '–auto-approve' command-line argument as shown here:

```
terraform apply --auto-approve
terraform destroy --auto-approve
```

NEXT

BACK

HOME

CHAPTER

ELEVEN

TIPS AND TRICKS

Terraform has a few execution quirks that can become bothersome with frequent usage. One example is that both 'terraform apply' and 'terraform destroy' require an interactive approval. Bypassing this is possible with the use of the '–auto-approve' argument, but that's a lot to be typing everytime you want to perform or clean-up a terraform run. (And, being exceptionally lazy, I don't want to be typing that much every time I run a terraform command.) So what follows are the Bash aliases I use with terraform, as well as a few other things I do to a) simplify terraform usage and b) diagnose problems with terraform runs.

11.1 Bash aliases

Bash aliases are a way to create Bash commands that call actual programs. Bash aliases are defined as so:

```
alias alias_name="command_to_run <arguments>"
```

Aliases are usually placed in a file that will be sourced by Bash during initialization, such as the ~/.bashrc, ~/.profile, or ~/.bash_aliases. Here is the full contents of my ~/.terraform_aliases.bash file, which is source from ~/.bashrc during Bash initialization:

```
alias tf='terraform'
alias tfaa='terraform apply -auto-approve'
alias tfda='terraform destroy -auto-approve'
alias tfi='terraform init'
alias tfp='terraform plan'
```

This command in my ~/.bashrc sources the ~/.terraform_aliases.bash file, ensuring that these aliases are present every time I open a terminal.

```
source ~/.terraform_aliases.bash
```

NOTE: The aliases could just as easily be located directly in the ~/.bashrc rather than being sourced. I source the file rather than having them in ~/.bashrc because I have (literally) hundreds of aliases, and breaking them into separate files helps keep them organized.

To use an alias you just type the alias name as if it were a command. To use the 'tfaa' alias I would enter the following on the command line:

```
tfaa
```

...which would be the same as typing:

```
terraform apply --auto-approve
```

11.2 Terraform State

Terraform keeps track of the state of configured resources using a file called terraform.state, which is a JSON-formatted text file containing all of the attributes of every resource deployed by a Terraform Run. The 'terraform.state' file *can* be examined by using a tool like jq; however, Terraform provides a set of commands for viewing the resources in the terraform.state file to view the current state of the Run.

You can list the resources in the terraform.state file with:

```
terraform state list
```

Here is the output of 'terraform state list' for what is deployed by deploying Example 4.

```
$ tf state list
module.rg[0].azurerm_resource_group.rg
module.rg[1].azurerm_resource_group.rg
module.rg[2].azurerm_resource_group.rg
```

You can also view details of each object by using 'terraform state show *object_name*, as shown here:

```
$ tf state show module.rg`0 .azurerm_resource_group.rg
# module.rg`0 .azurerm_resource_group.rg:
resource "azurerm_resource_group" "rg" {
    id       = "/subscriptions/0f92c295-b01d-47ab-a709-1868040254df/resourceGroups/my_
↪lab-1-rg"
    location = "westus2"
    name     = "my_lab-1-rg"
}
```

Examining the state of an object in Terraform is particularly useful when you need to use an attribute of an object that isn't well defined in the resource documentation. This doesn't come up often, but when it does being able to examine the object to see what attributes you can access is extremely helpful. Any resource attribute you can see in the terraform.state file is usable within Terraform code.

11.2.1 Azure Terraform State Trick

My favorite aspect of the 'terraform.state' file is that it is the **sole** source of truth for Terraform. This means that if you want to completely reset Terraform's "view" of the current run all you need to do is delete or rename this file. Why is this great? Well, sometimes destroying a complex environment deployed by Terraform can take a really long time. I've been stuck waiting for an Azure lab to be destroyed for 15+ minutes in the past. (This is actually an Azure responsiveness issue rather than an Terraform issue, but knowing that doesn't make the time go by any faster.)

If you organize your lab naming scheme around a single *prefix* value that is incorporated into the name of all objects created by that run, what you can do to save time is just go to the Azure Portal and delete the resource-group(s) created

by your Terraform Run. Then delete the 'terraform.state' file itself. Finally, change the *prefix* you are using with all of your object names. At this point all of the following will be true:

1. Azure will be deleting the Resource-Group and all of the objects it contains. It won't matter if this takes two minutes or an hour because…
2. Terraform will believe nothing is deployed because there is no state file. You can immediately begin testing your most recent changes to the Terraform configuration because…
3. With the new prefix none of the object names Terraform attempts to deploy will collide with existing objects.

- Technically the concern regarding name collisions only applies to the resource-group name itself; however, there are a couple other objects that also require globally (or at least, organizationally) unique names, such as Log Analytics Workbooks and Storage Accounts.

Using this trick will spare you a lot of time if you start to create Terraform Runs with many levels of dependencies.

NOTE: This trick is only really only useful when you are working in an environment that allows a simple, hands-off group deletion option, like deleting an Azure Resource-Group or Kubernetes namesspace. GCP, and especially AWS, have no simple administrative container that can be deleted at-will to destroy all of the grouped objects.

WARNING: The corallary to the note above is that you should avoid deleting your terraform state file in all other cases; especially when working with AWS or GCP. I once had a corrupted deployment to AWS that caused the 'terraform destroy' command to fail due to an AWS error, so I had to track down every oject I had deployed with Terraform and delete them all manually. This was an incredible PITA. Deleting your terraform.state file without first running the 'terraform destroy' command will result in the same thing: to clean up your deployed resources you'll end up having to track all of them down to manually delete them. You have been warned.

11.3 Selective apply / destroy

You can restrict Terraform to deploying or destroying specific objects by using the '–target=<resource_name>' command-line argument. This can be particularly useful if you have a large Run and are trying to debug or test one of the final resources being deployed. (i.e. trying to debug the cloud-init being used with BIG-IP). In those cases all of the time necessary to destroy, then re-deploy, all of the resources that the BIG-IP depends on is effectively wasted time - all you *need* to destroy and re-deploy is the BIG-IP itself. This is not an uncommon scenario, and the answer is the '–target=<name>' argument.

To use –target=name you enter the terraform destroy or plan command like you normally would, but you add the '–target=' argument afterwards. For example, let's say my BIG-IP is deployed in a module called 'bigip'. I can destroy all of the objects related to that object alone by using the following command:

```
terraform destroy --auto-approve --target=module.bigip
```

That command will destroy the resources created in my 'bigip' module and nothing else.

NOTE: If the resource you are trying to destroy in this way is a dependency of a later resource, the command will fail.

To re-deploy I have two options: #. Use the '–target=' argument again when running the 'terraform apply' command #. Run 'terraform apply [–auto-approve]' without the '–target=' argument and jsut let Terraform deploy everything that isn't already deployed (as per the terraform.state file).

NOTE: According to Terraform the '–target=<name>' argument should only be used for debugging/testing.

11.4 Terraform State file manipulation

It is possible to manually remove objects from the state file without destroying them. This only comes up rarely, but if you find yourself in a position where it is important you can do this with the **terraform state rm <resource_name>** command

11.5 Using Terraform with Git

It is extremely common to use Git to provide source control for Terraform configurations. Entire DevOps ecosystems have been created around this relationship, and I would be remiss to not include a section on some best-practices related to the **.gitignore** file.

As you almost certainly know, the `.gitignore` file is used to exclude files from being included by git, and there are some files you really don't want included in your git repository. I've provided a list of these files below and urge you to use the `.gitignore` file to exclude them.

- **.terraform/**
  - Directory containing the dowloaded Providers and files pertaining to the modules defined in your Terraform configuration.
  - Add the following to .gitignore: `.terraform*`
- **.terraform.lock.hcl**
  - File containing a list of the downloaded Providers and the hashes associated with each
  - **Add the following to .gitignore: *.terraform****
    - *.terraform** excludes both the *.terraform.lock.hcl* file and the *.terraform/* directory
- **.terraform.tfstate & .terraform.tfstate.backup**
  - File containing the current state of any resources deployed by Terraform (see above)
  - **Add the following to .gitignore: `terraform.tfstate*`**
    - Excludes both the *terraform.tfstate* and the *terraform.tfstate.backup* files.

More complex Terraform configurations might include an output directory for post-procesing template files, as well as a directory within which those template files might be stored. I names those directories *work_tmp* and *templates*, respectively. The *templates* directory should be included in a Git repository; however, the directory containing the post-processing versions of those templates should not, so I add that directory to my .gitignore.

The complete list of .gitignore additions would be:

```
.terraform*
.terraform.tfstate*
work_tmp/
```

NEXT

BACK

HOME

CHAPTER

TWELVE

EXAMPLE #1 - SIMPLE VARIABLES AND OUTPUT

The following example shows the Terraform code to:

1. Define an input variable called 'prefix'
2. Create a local value called 'rg_name' that incorporates the value of the 'prefix' variable
3. Create a null-resource to call the command-line 'echo' command to print the values of the variable and local value
4. Create Terraform Output blocks that show the values of the variable and the local value

```
variable "prefix" {
  default = "ProjectName"
}

locals {
  rg_name = "${var.prefix}-RG"
}

resource "null_resource" "call_echo" {
  provisioner "local-exec" {
    command = "echo \"The object prefix is ${var.prefix} and the rg_name is ${local.rg_
↪name}\""
  }
}

output "prefix_name" {
  value = var.prefix
}

output "object_name" {
  value = local.rg_name
}
```

NOTE: Terraform Runs that only involve built-in providers like *null-resource* do not require *terraform* or *provider* blocks. This makes it very easy to create trivial runs for the purpose of testing Terraform syntax and behavior. The example above was something I wrote when I first started using Terraform to test the use of a null-resource.

12.1 Lab

Now we're going to run through the one and only lab including with this beginners guide. This is simply to demonstrate the expected output of each of the main terraform commands that you'll use when creating yoru own Runs. To run this example you can either use the code in the ./example_1/ directory, which matches the example above, or create a new directory and copy/paste the example above into a file called *main.tf*.

12.1.1 terraform init

In the directory with the *main.tf*, initialize Terraform:

```
terraform init
```

Once that command completes you'll have a new hidden directory and a new hidden file:

```
$ ls -lA
total 16
drwxr-xr-x@ 3 driskill  1437522721    96 Oct 26 15:21 .terraform
-rw-r--r--@ 1 driskill  1437522721  1152 Oct 26 15:21 .terraform.lock.hcl
-rw-r--r--@ 1 driskill  1437522721   359 Oct 15 11:53 main.tf
```

12.1.2 terraform plan

Now we'll run the 'terrform plan'. Technically this is an optional step as the 'terraform apply' will go through the same process out of necessity; however, finding problems *before* you starting deploying resources is always better. For that reason I recommend running 'terraform plan' prior to every 'terraform apply'.

```
terraform plan
```

The output of the 'terraform plan' command will show everything Terraform will attempt to do when you run the 'apply'. Dynamic values will be replaced with placeholders that say 'known at apply'. Here is the output from the above 'terraform plan' operation:

```
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:

  # null_resource.call_echo will be created
  + resource "null_resource" "call_echo" {
      + id = (known after apply)
    }

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

Changes to Outputs:
  + object_name = "ProjectName-RG"
  + prefix_name = "ProjectName"
```

12.1.3 terraform apply

And finally we will run 'terraform apply'. Optionally you can add the '–auto-approve' argument to avoid the standard manual approval:

```
terraform apply --auto-approve
```

Here is the output of the 'apply' command:

```
$ terraform apply --auto-approve

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:

  # null_resource.call_echo will be created
  + resource "null_resource" "call_echo" {
      + id = (known after apply)
    }

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

Changes to Outputs:
  + object_name = "ProjectName-RG"
  + prefix_name = "ProjectName"
null_resource.call_echo: Creating...
null_resource.call_echo: Provisioning with 'local-exec'...
null_resource.call_echo (local-exec): Executing: ["/bin/sh" "-c" "echo \"The object␣
↪prefix is ProjectName and the rg_name is ProjectName-RG\""]
null_resource.call_echo (local-exec): The object prefix is ProjectName and the rg_name␣
↪is ProjectName-RG
null_resource.call_echo: Creation complete after 0s [id=5577006791947779410]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

object_name = "ProjectName-RG"
prefix_name = "ProjectName"
```

You'll also find a new file in the directory called *terraform.state*. As promised, this is a JSON-formatted text file containing everything Terraform knows about the current deployment. We are only creating a variable, a local value, and a null-resource so this state file is quite small; however, the size grows very quickly once you start working with multi-resource Runs.

At this point you can inspect the various resources Terraform created using the 'terraform state list' and 'terraform state show <name>' commands:

```
$ terraform state list
null_resource.call_echo
$ terraform state show null_resource.call_echo
```

(continues on next page)

(continued from previous page)

```
# null_resource.call_echo:
resource "null_resource" "call_echo" {
    id = "5577006791947779410"
}
```

12.1.4 terraform destroy

Finally, we're going to 'destroy' the Resources created by this run with the 'terraform destroy' command. Like the 'terraform apply' command, the 'terraform –destroy' command supports the '–auto-approve' command-line argument:

```
$ terraform destroy --auto-approve
null_resource.call_echo: Refreshing state... [id=5577006791947779410]

Terraform used the selected providers to generate the following execution plan. Resource␣
↪actions are indicated with the
following symbols:
  - destroy

Terraform will perform the following actions:

  # null_resource.call_echo will be destroyed
  - resource "null_resource" "call_echo" {
      - id = "5577006791947779410" -> null
    }

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

Changes to Outputs:
  - object_name = "ProjectName-RG" -> null
  - prefix_name = "ProjectName" -> null
null_resource.call_echo: Destroying... [id=5577006791947779410]
null_resource.call_echo: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.
```

If you look in the directory now you'll see that the 'terraform.state' file is smaller, which is because we have no more created resources. You'll also notice a new file called 'terraform.state.backup'. The *terraform.state.backup* file is a copy of the 'terraform.state' file created immediately before any changes were made.

NEXT

BACK

HOME

CHAPTER

THIRTEEN

EXAMPLE #2 - OBJECT OUTPUT

The following example illustrates the Terraform code to:

1. Define a variable called 'lab_prefix'
2. Create several local values with embedded values, many of which are dynamic and based on the value of the 'lab_prefix' variable.
3. Print the values of the various local values using output blocks.

```
variable "lab_prefix"       { default = "nginx" }

locals {
  rg = {                    # Resource group
    name                    = format("%s-rg", var.lab_prefix)
    location                = "westus2"
  }
  vnet = {                  # virtual networks
    name                    = format("%s-bigip", var.lab_prefix)
    cidr                    = "10.210.0.0/16"
  }
  nsg = {                   # Network security group
    name                    = format("%s-nsg", var.lab_prefix)
    src_addrs               = ["10.1.1.1","10.53.24.176","10.53.24.178", "10.239.25.19"]
    dst_addrs               = ["172.16.0.0/16", "172.17.0.0/16", "192.168.0.0/24"]
    dst_ports               = ["22","443", "8443"]
  }
  log_analytics = {         # Log Analytics Workspace
    name                    = format("%s-law", var.lab_prefix)
    retention               = "30"
    sku                     = "PerNode"
    ts_region               = "us-west-2"
    ts_type                 = "Azure_Log_Analytics"
    ts_log_group            = "f5telemetry"
    ts_log_stream           = "default"
  }
  lb = {                    # load-balancer
    use_lb                  = 1
    name                    = format("%s-lb", var.lab_prefix)
    pool_name               = format("%s-lb_pool", var.lab_prefix)
    sku                     = "Standard"
    priv_allocation         = "Dynamic"
    priv_version            = "IPv4"
```

(continues on next page)

(continued from previous page)

```
  }
}

output "lab"  { value = var.lab_prefix }
output "rg"   { value = local.rg}
output "vnet" { value = local.vnet}
output "nsg"  { value = local.nsg}
output "law"  { value = local.log_analytics }
output "lb" { value = local.lb }
```

Note: Notice the use of the format function to build a dynamic local value using the variable name.

NEXT

BACK

HOME

CHAPTER

FOURTEEN

EXAMPLE #3 - MODULE CREATION AND USAGE

Official Documentation

NOTE: In order to execute this example you must have access to the Azure CLI and it must be authenticated to work with Azure.

NOTE: This example will create two Azure Resource-Groups called 'my_lab-1-rg' and 'my_lab-2-rg'.

The following example illustrates the Terraform code:

1. Usage of the 'vars.tf', 'outputs.tf', and 'main.tf' as separate files
2. Usage of the Azure provider
3. Creation and usage of a module

Typically Terraform Runs use separate files for input variables, configuration blocks, and output blocks. Modules are defined in a sub-directories beneath the main Terraform directory. The typical directory structure, as well as the separate files, are illustrated here:

```
.
+-- main.tf
+-- vars.tf
+-- outputs.tf
+-- modules
|   +-- resource_group
|      +-- main.tf
|      +-- variables.tf
|      +- -outputs.tf
```

In this example we are creating a single module called *resource_group*. We are calling the *resource_group* module to create an Azure resource-group. Creating a resource-group in Azure reuqires a couple variables, specifically, a name and a location. These variables are passed to the module when it is called using variable assignments specified in *main.tf*. Those variables are also defined in the *variables.tf* file located within the module sub-directory.

The ./modules/resource_group/variables.tf file defines the input variables required by the module. All of the variables defined in the modules *variables.tf* file must be provided when the module is called from main.tf. If a variable is defined in the ./modules/<*module*>/variables.tf file and it is *not* provided when the module is called you will receive an error when running terraform [plan|apply].

Note: Additional variables can be provided to the module when it is called even without being defined in the variables.tf file; however, any additional variables will not be available for use. They simply don't cause a syntax error.

14.1 ./vars.tf

```
variable "prefix"     { default = "my_lab" }
variable "location"   { default = "westus2" }
```

14.2 ./main.tf

```
terraform {
  required_providers {
    azurerm = { version = "~> 3.27.0" }
  }
}

provider "azurerm" {
  features {}
}

module "rg1" {
  source                = "./modules/resource_group"
  prefix                = var.prefix
  location              = var.location
}
module "rg2" {
  source                = "./modules/resource_group"
  prefix                = var.prefix
  location              = var.location
}
```

14.3 ./outputs.tf

```
output "rg1" {
  description = "Resource group details"
  value = {
    rg_name     = module.rg1.out.name
    rg_location = module.rg1.out.location
    rg_id       = module.rg1.out.id
  }
}

output "rg2" {
  description = "Resource group details"
  value = {
    rg_name     = module.rg2.out.name
    rg_location = module.rg2.out.location
    rg_id       = module.rg2.out.id
  }
}
```

14.4 ./modules/resource_group/variables.tf

```
variable prefix     {}
variable location   {}
```

The module variables.tf file defines the variables required by the module. These variables *must* be provided when the module is called. If variables are defined in the module variables.tf file but not provided when the module is called an error will be reported when you run terraform [plan|apply].

14.5 ./modules/resource_group/main.tf

```
resource "azurerm_resource_group" "rg" {
  name      = var.prefix
  location  = var.location
}
```

The module main.tf file defines the actions that will be taken by the module. The syntax is identical to the syntax defined in the primary main.tf; however, the only variables available are those defined in the module variables.tf file.

14.6 ./modules/resource_group/outputs.tf

```
output "out" { value = azurerm_resource_group.rg }
```

The module outputs.tf file sends the outputs back to the main terraform execution. These outputs can then be used as input variables to other configuration blocks, including other modules. They can also be used in output blocks defined in the main directory to print the values after the Terraform Run completes. One very common example of this is printing the IP addresses of virtual-machines instantiated by the Terraform Run.

NEXT

BACK

HOME

CHAPTER

FIFTEEN

EXAMPLE #4 - MODULE USAGE WITH TERNARY CONDITIONAL AND 'COUNT' META-ARGUMENT

NOTE: In order to execute this example you must have access to the Azure CLI and it must be authenticated to work with Azure.

NOTE: This example will create three Azure Resource-Groups called 'my_lab-1-rg', 'my_lab-2-rg', and so on.

The following example illustrates the Terraform code: 1. Usage of the 'vars.tf', 'outputs.tf', and 'main.tf' as separate files 2. Usage of the Azure provider 3. Creation and usage of a module

Typically Terraform Runs use separate files for input variables, configuration blocks, and output blocks. Modules are defined in a sub-directories beneath the main Terraform directory. That directory structure, as well as the separate files, are illustrated here:

The files and directory structure typically follow this pattern:

```
.
| main.tf
| vars.tf
| outputs.tf
|--modules
|
|  |--resource_group
|  |   main.tf
|  |   variables.tf
|  |   outputs.tf
```

In this example we are using the *resource_group* module to create an arbitrary number of resource-groups using the count meta-argument. The *instances* variable controls the number of resource-groups created; however, the 'single' variable controls whether the 'instances' variable is actually used. If the 'single' variable is set to 'true', then the number of resource-groups crated is only one.

Checking the value of the 'single' variable is done by the ternary conditional. The ternary conditional checks the value of a variable. If the condition is true, the first option following the '?' is assigned to the variable. If the condition is false then the option following the ':' is assigned. The basic syntax is as follows:

```
variable = condition ? value_1 : value_2
```

The condition can be a simple comparison, as is shown in the example below, or it can be a compound comparison like so:

```
variable = condition_1 && condition_2 ? value_1 : value_2
```

Notice that 'count' is *not* defined in the module variables.tf file. This is because 'count' is a meta-argument that applies to the module and is not provided to the module itself. Rather, it defines how many times the module is called. Also

notice that the 'prefix' variable sent to the module is updated for each iteration using the 'count.index+1' syntax. This ensures that each resource-group has a unique name, but can be used in other ways.

15.1 vars.tf

```
variable "prefix"     { default = "my_lab" }
variable "location"   { default = "westus2" }
variable "single"     { default = false }
variable "instances   { default = 3 }
```

15.2 main.tf

```
terraform {
  required_providers {
    azurerm = { version = "~> 3.27.0" }
  }
}

provider "azurerm" {
  features {}
}

module "rg" {
  source                = "./modules/resource_group"
  count                 = var.single == false ? var.instances : 1
  prefix                = format("%s-%d-rg", var.prefix, count.index+1)
  location              = var.location
}
```

15.3 outputs.tf

```
output "rg" {
  description = "Resource group details"
  value = {
    rg_name     = module.rg.*.out.name
    rg_location = module.rg.*.out.location
    rg_id       = module.rg.*.out.id
  }
```

15.4 modules/resource_group/variables.tf

```
variable prefix     {}
variable location   {}
```

15.5 modules/resource_group/main.tf

```
resource "azurerm_resource_group" "rg" {
  name      = var.prefix
  location  = var.location
}
```

15.6 modules/resource_group/outputs.tf

```
output "out" { value = azurerm_resource_group.rg }
```

BACK

HOME