**Terraform user guide**

Terraform is a tool for building, changing and versioning infrastructure safely and efficiently. It is used to create virtual environments using cloud based services such as Amazon Web Service and Google Cloud as well as managing storage and networking. It is also capable of managing DNS entries, Saas features, and other high-level components.

**Installing Terraform**

To install Terraform simply download the zip archive and unzip it. This directory will contain a binary file called *terraform.* You will then need to update the path on your machine to include the path to this directory. To do this for Linux/Mac use the following into the terminal

PATH=/usr/local/terraform/bin:/home/your-user-name/terraform:$PATH

On Windows, simply update the environment variable PATH to include the path to the Terraform directory.

To verify that Terraform has been installed, open a new terminal and write the command terraform. If it is properly installed then there should be a list of possible commands to follow it, such as

$ 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

destroy Destroy Terraform-managed infrastructure

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 Initializes Terraform configuration from a module

output Read an output from a state file

plan Generate and show an execution plan

push Upload this Terraform module to Atlas to run

refresh Update local state file against real resources

remote Configure remote state storage

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

All other commands:

state Advanced state management

If you get an error, saying that terraform could not be found, then your PATH variable is probably not set correctly.

**Creating an instance with Terraform**

As stated before, terraform can be used to create instances from cloud based services. So for this example we will create an AWS EC2 instance.

Terraform configuration files, where the infrastructure is described, have their own format but can accept the JSON format. If they follow the Terraform configuration format then the file will be a .tf file, else if it follows the JSON format it should end in .tf.json

The configuration file for the AWS instance is as follows

provider “aws” {

access\_key = “ACCESS\_KEY\_HERE”

secret\_key = “SECRET\_KEY\_HERE”

region = “us-east-1”

}

resource “aws\_instance” “example”{

ami = “ami-0d729a60”

instance\_type = “t2.micro”

}

Before running the new configuration file, make sure there are no other .tf files in the same directory as Terraform will attempt to run them all at once.

The *provider* block is used to specify the provider, i.e. the service that will create and manage the instance. It is quite common to have multiple provider blocks when using Terraform, as there are quite often multiple providers when using Terraform on a large scale The ACCESS\_KEY\_HERE and SECRET\_KEY\_HERE are to be replaced by the AWS key and access key which you can find after logging in to the AWS webpage.

The *resource* block is used to define the resources that exist in the infrastructure. This could be anything from a virtual machine or a physical server to a DNS record or database provider. The two strings before the block is opened are the resource type and the resource name respectively. The prefix of the type maps to the provider, i.e. in this instance “aws”. The ami in the resource block refers to the build image, so this instance will be building from an Ubuntu 14.04 image.

You can then run the terraform plan command in the same directory as your .tf file to see what changes will be made when applying the configuration files. The example above would provide the following output.

$ terraform plan

...

+ aws\_instance.example

ami: "ami-0d729a60"

availability\_zone: "<computed>"

ebs\_block\_device.#: "<computed>"

ephemeral\_block\_device.#: "<computed>"

instance\_state: "<computed>"

instance\_type: "t2.micro"

key\_name: "<computed>"

placement\_group: "<computed>"

private\_dns: "<computed>"

private\_ip: "<computed>"

public\_dns: "<computed>"

public\_ip: "<computed>"

root\_block\_device.#: "<computed>"

security\_groups.#: "<computed>"

source\_dest\_check: "true"

subnet\_id: "<computed>"

tenancy: "<computed>"

vpc\_security\_group\_ids.#: "<computed>”

If there is an error message, you can then make the appropriate changes before applying the configurations.

If there are no errors, use the terraform applycommand in the same directory. This will then create the instance, and you can use the terraform show command to show details about the instance such as the DNS and IP address, which you can then use to SSH into the instance.

We can also create an elastic IP attached to this AWS instance so that we keep the same IP address each time we shut down and start up again. To do this we can create another resource block

resource "aws\_eip" "ip" {

instance = "${aws\_instance.example.id}"

depends\_on = [“aws\_instance.example”]

}

Once again the “aws” prefix maps to the provider “aws” and the eip refers to the elastic IP being placed on the instance. The second line is not necessarily needed, it is simply a line that invokes an explicit relationship between the two resources, instance and IP. However, the instance line also invokes the relationship, so the inclusion of the depends\_on line is down to personal preference.

**Changing an Instance**

If you wish to make any changes to the core details of the instance, you can simply change the settings in the .tf file. Once this is done use the terraform plan and terraform apply commands as you would when building the instance.

For example changing the build image for the example above, simply change the resource block to

resource "aws\_instance" "example" {

ami = "ami-13be557e"

instance\_type = "t2.micro"

}

This is now the ami that refers to the Ubuntu 16.04 image. Then run the terraform plan command again to see which changes will be made once applied.

$ terraform plan

...

-/+ aws\_instance.example

ami: "ami-0d729a60" => "ami-13be557e" (forces new resource)

availability\_zone: "us-east-1a" => "<computed>"

ebs\_block\_device.#: "0" => "<computed>"

ephemeral\_block\_device.#: "0" => "<computed>"

instance\_state: "running" => "<computed>"

instance\_type: "t2.micro" => "t2.micro"

private\_dns: "ip-172-31-17-94.ec2.internal" => "<computed>"

private\_ip: "172.31.17.94" => "<computed>"

public\_dns: "ec2-54-82-183-4.compute-1.amazonaws.com" => "<computed>"

public\_ip: "54.82.183.4" => "<computed>"

subnet\_id: "subnet-1497024d" => "<computed>"

vpc\_security\_group\_ids.#: "1" => "<computed>”

Again if there are no errors then run the terraform apply command. If there are then correct them and check again.

**Destroying an Instance**

Using the destroy command will destroy all parts of the infrastructure named in the current directory. To confirm what we will be deleting, we can again use the plan command with the –destroy flag. E.g.

$ terraform plan –destroy

...

- aws\_instance.example

If this is what we wish to destroy, then simply use the command terraform destroy.