**Terraform: (Infrastructure as a Code with Terraform:**

Is used to **create**, **manage** and ***update infrastructure resources*** such as **physical machines**, **VMs** **network switches**, **containers** and more.

Almost **any infrastructure type** can be **represented** as a **resource** in **Terraform**.

A provider is **responsible** for **understanding API interactions** and **exposing resources**. Providers generally are Iaas. ( eg. Alibaba Clould, AWS, Azure, OpenStack(, Pass(eg. Heroku), or Saas (eg. Terraform cloud, DNSimple, CloudFlare).

**Concept and Background:**

Once upon a time in data- centres, system admin’s job was to purchase hardware, setup servers with network and software and also maintain/modify them manually.

All these repetitive processes are time-consuming, inefficient and take a lot of energy, thus making it frustrating, while not adding any learning value since the system admin is already well-versed with the process.

Also, in the manual process, there is the fear of human errors resulting in misconfiguration, downtime caused by hardware issues or the sysadmin being unavailable at a critical time.

With changing times, this has also changed. The ***cloud era*** has shown that ***everything can be configured using console or APIs which enables us to interact with infrastructure through code*** thus ***making the process less prone to human errors and extensible and maintainable.***

* ***Infrastructure as a code (IAC) is the process of managing and provisioning infrastructure*** *through the* ***code instead of physical hardware configuration or interactive configuration tools***.
* ***With IAC, infrastructure can be provisioned in seconds and scale can be achieved with capacity planning.***

Advantages of using **infrastructure as code**:

* **Reusable**: ***Same code*** can be used for ***multiple environments, different setups***.
* **Maintainable:** ***Code*** can be ***pushed to version control tools***.
* **Extensible:** ***More code*** can be ***added*** to ***existing code*** to ***extend existing infrastructure***.
* **Testable:** ***Code*** can be ***tested*** against ***expected behaviour*** for ***reliability***.
* **Repeatable:** We can hit the same, again and again, to get the same results, making all the repeatable jobs easy. ***Write code once*** and ***run it multiple times*** to get exactly the same results.
* **Documentation:** ***Code*** can be ***documented*** for better understanding.
* **Sharable:** ***Code*** can be ***shared among developers*** to ***perform*** the ***same task across the organisation***.
* **Auditable:** ***All changes*** are done as ***code*** which is managed in the ***version control system to track changes***.

[If something goes wrong then it is easy to check the history, revert, and rebuild if we are using code].

***Code -> Version Control -> Code Review -> Validate -> Create Infrastructure***

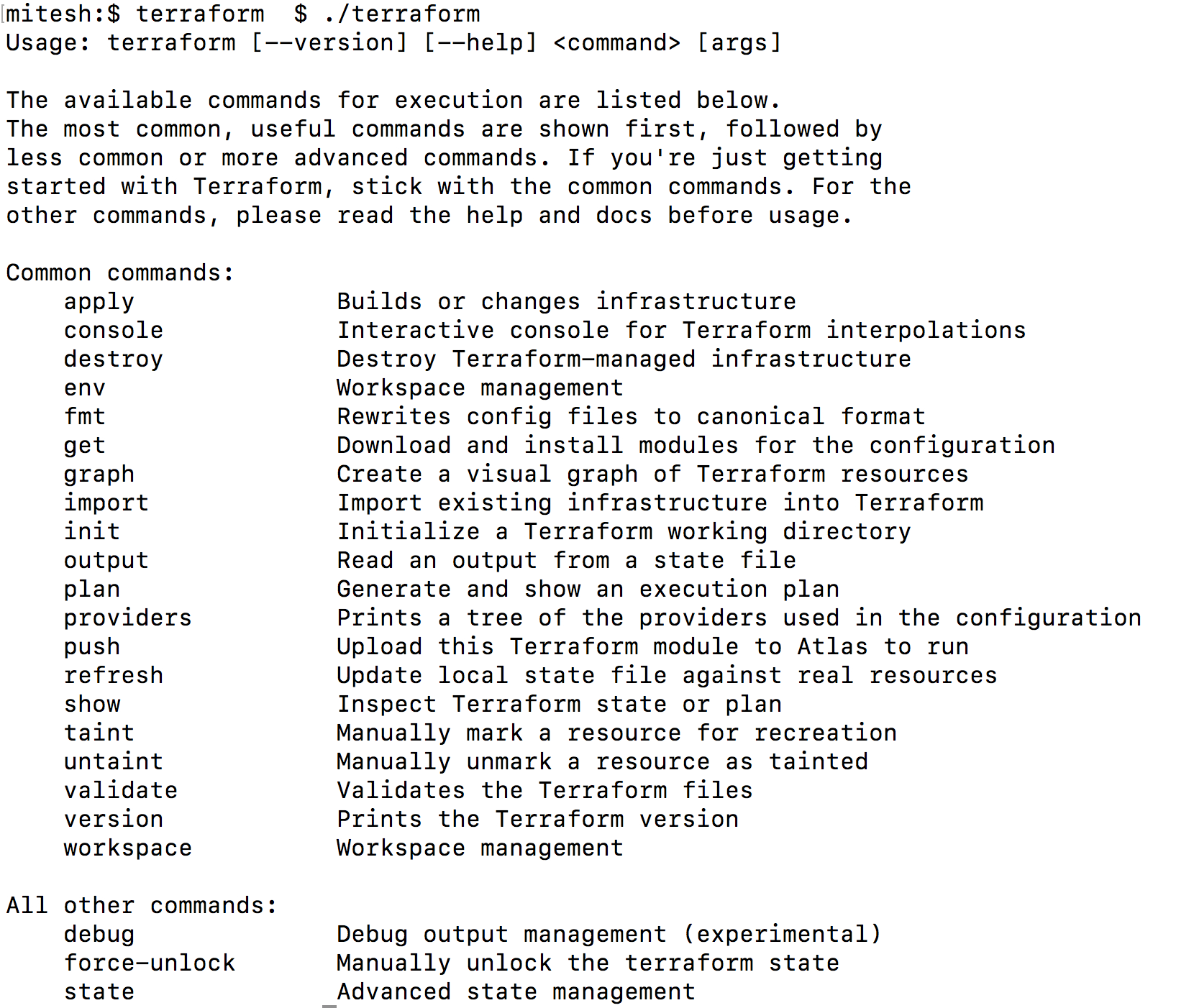
***Application development*** and ***infrastructure provisioning*** works the ***same way*** to ***achieve desired results***, both have their **own code** which can be ***interpreted*** by ***either interpreter in case of application code*** or ***API provider in case of infra code***.

***Application: SourceCode -> Interpreter -> Desired Application State  
Infrastructure: Infra Code -> API Request to Cloud Provider -> Desired Infrastructure State***

**Terraform** is one of the tools which can be used for building, changing, versioning our infrastructure safely and efficiently; which supports many service providers as well as custom in-house solutions.

**Infrastructure** is ***defined*** using **high-level configuration languages**: **HCL** and **JSON**. We are going to use HCL as it is easy to start, is human-readable and editable and is also recommended for terraform.

The first step towards running terraform code is to download zip archive from [here](https://www.terraform.io/downloads.html). Once it is downloaded, unzip. It will have a single binary named terraform which will run our terraform code. Verify installation by opening a terminal with the same folder in which we have our binary then run this binary.



There are many [providers](https://www.terraform.io/docs/providers/index.html) available for terraform but we are going to start with AWS since it is popular. If you don’t have an aws account, please create one for free from [here](https://aws.amazon.com/free/).

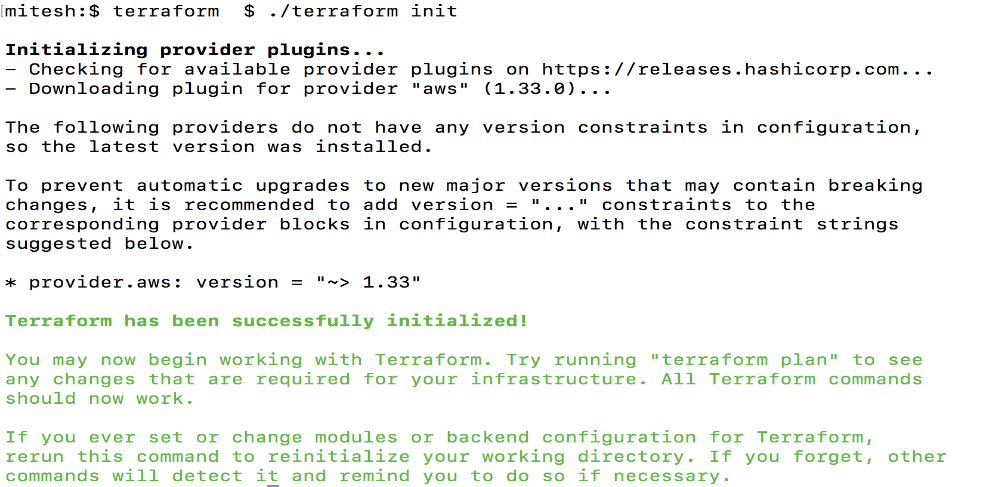
Let’s write some terraform configuration file with extension .tf. We first need to define our provider and provide access and secret key with the right set of permissions along with region where we want to create our infrastructure.

provider “aws” {  
 access\_key = “ACCESS\_KEY\_HERE”  
 secret\_key = “SECRET\_KEY\_HERE”  
 region = “us-east-2”  
}

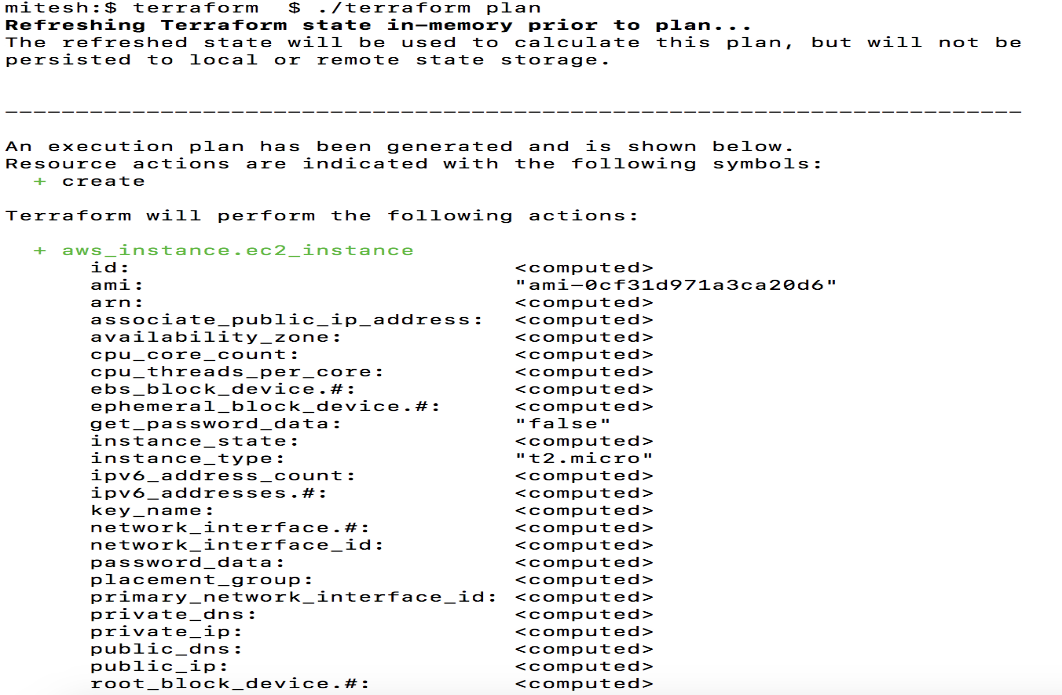
Once our provider is set, let’s create an EC2 instance of instance type t2 micro with ami: ami-0cf31d971a3ca20d6.

resource “aws\_instance” “ec2\_instance” {  
 ami = “ami-0cf31d971a3ca20d6”  
 instance\_type = “t2.micro”  
}

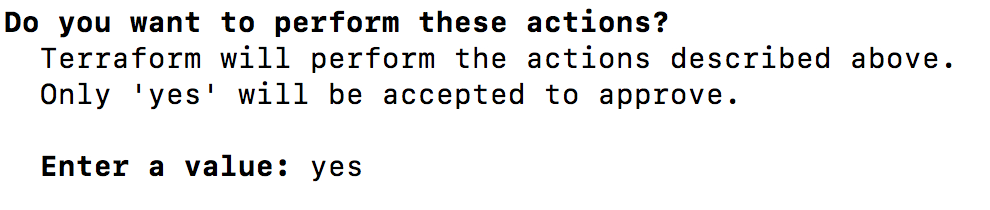
As we are ready with code, try to run this using terraform binary. The first command to run for a new configuration is **terraform init** which initialises various settings and data, download provider plugin and install in a subdirectory of the current working directory.



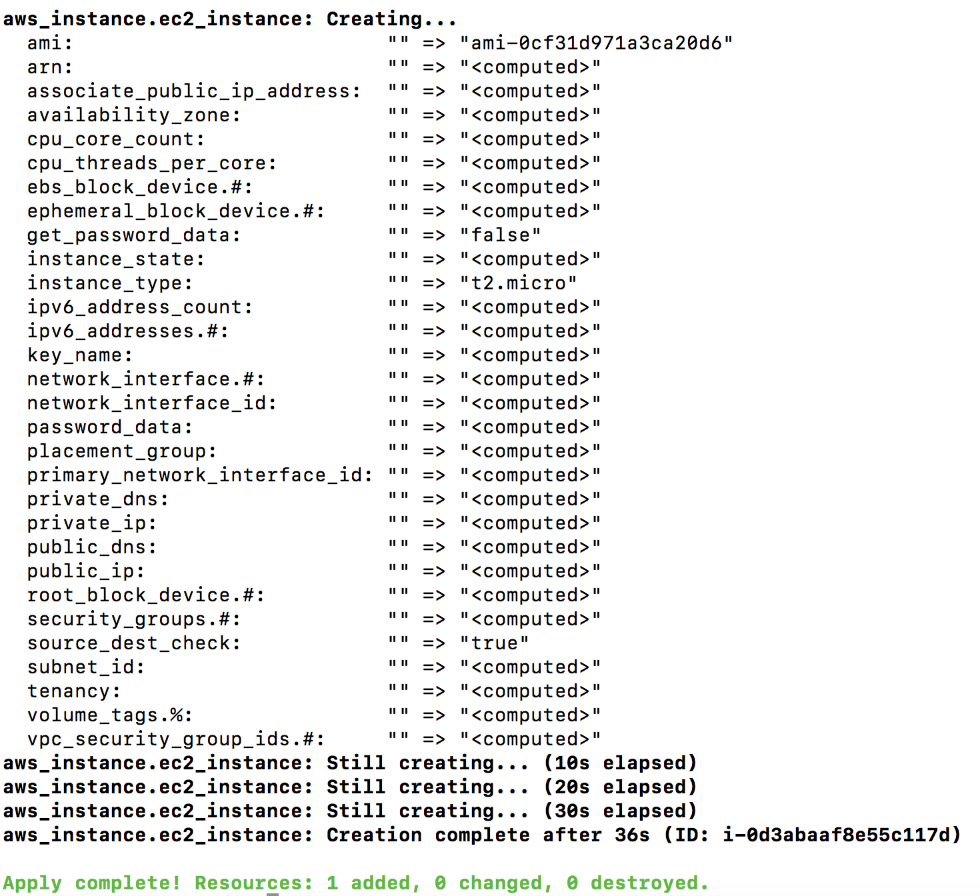
Once we init, we want to know what exactly is going to happen on aws infrastructure, this can be done by running command **terraform plan**. Terraform plan is used to create an execution plan which is a way to check whether the set of changes matches our expectation without making any real infrastructure changes.



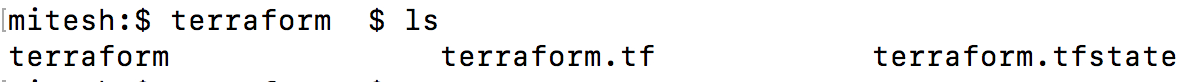
This is the time to actually create our resources on aws, **terraform apply** command is used to apply the changes required to reach the desired state. Terraform apply scans current directory for the configuration and apply the changes on aws. When we run apply it shows us plan and ask for confirmation before executing the plan.



Once we confirm then all operations are applied on aws.



Now you will notice a terraform.tfstate file is generated. This file records the state of everything this terraform configuration has done. It is used to map real-world resources to the configuration, keep track of metadata. Terraform uses this local state to create a plan and make changes to infrastructure.



We might want to change ami or instance type based on requirement, input variables can act as a way to do this. Variables should not be part of our resource code and could be injected from CLI arguments and environment variables.

**There are multiple ways to assign variables:**

1. Command-line flags: Variables can be set directly on the command-line with the -var flag.

./terraform apply -var instance\_type**=**“t2.micro”

2. From a file: Create a file named terraform.tfvars and assign variables within this file.

instance\_type**=**“t2.micro”

3. From environment variables: Variables can be read environment params in the form of

TF\_VAR\_name to find the value.

TF\_VAR\_instance\_type=t2.micro

Let’s see some code with variables:

variable “instance\_type” {  
 default = “t2.micro”  
}resource “aws\_instance” “ec2\_instance” {  
 ami = “ami-0cf31d971a3ca20d6”  
 instance\_type = “${var.instance\_type}”  
}

Once this instance is created, we want to get its IP address for further use as the output variable. Output variables act as a way to organize data to be easily queried and shown back to the Terraform user.

output “ip” {  
 value = “${aws\_instance.ec2\_instance.public\_ip}”  
}

Once we are done with the changes we can run **terraform destroy** to destroy terraform managed infrastructure.

