Btool is a command line tool that can help us troubleshoot configurations file issue or see what values are being used by splunk

★) TERRAFORM

→ Tool used to build and manage & version Infrastructure as code

HCL → Hashicorp configuration Language

Json is also supported

Single configuration ⟹ can support different Service Providers

→ Terraform supports

*] Heterogenous cloud deployment → Some of your Services are deployed in AWS and some services in the Azure.

1] Reduce Opex → operation cost

2] better testing & quality

3] Reduce overall cost.

4] fast working

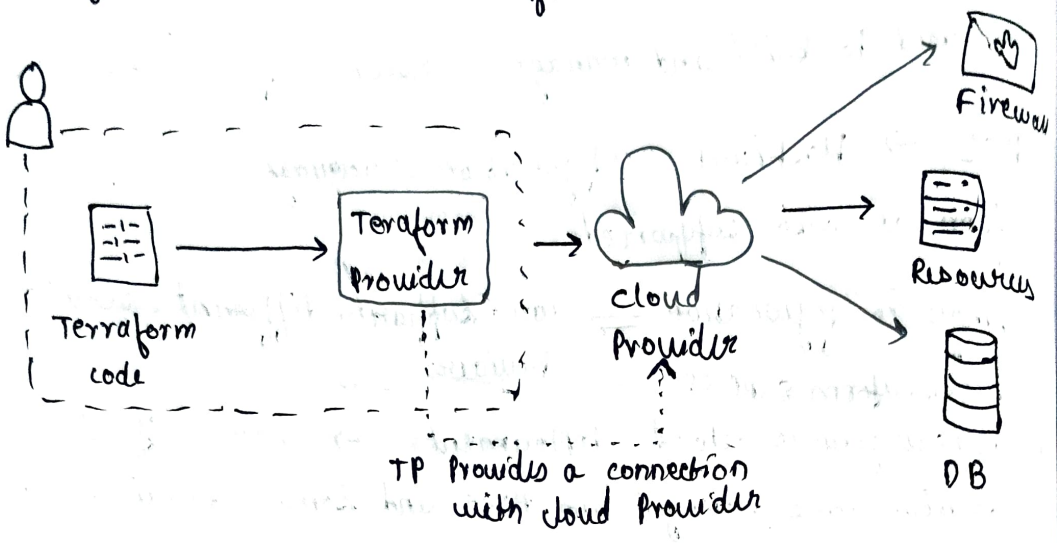
1] Terraform' configuration → what we are writing in our code ⟹ contains groups of Resources written in Hashicorp HCL or JSON. This is code the tells what infrastructure we want.

2] Terraform Provider → A Terraform plugin
→ It is a cloud Service Provider we will be working with eg AWS, Azure, GCP

3] Terraform Plan → Teraform generates a build execution Plan containing all the steps it will take to converge the current stage with the desired State.

4] Terraform State file → file contains the current state & desired state. (Should not be corrupted.)

5] Provisioner → Script that can be used to bootstrap Resource upon creation.
eg → cloud-init scripts, chef, Puppet etc.



Step 1. describe the Infrastructure
→ describe infrastructure using Terraform HCL

```
Resource "aws_instance" "myexample" {
    ami = "ami-6374d5a5"
    instance_type = "t2.micro"
}
```

Step 2. Build an execution plan.
→ By Running 'terraform plan'.
+ = Resource will be created
- = Resource will be destroyed.
-/+ = Resource will be destroyed then Re-created
↳ this could happen while changing the machine size.

Step 3: Execute a Plan

Run 'terraform apply'

→ Terraform file ends with `.tf`

→ Terraform CLI loads all files ending .tf by default.

```
# Hello  - Single line comment

/*
module "Learn tf - VPC"  - Multiline comment
*/
```

Key - Value Pair

```
Variable "aws_region" {
        default = "us-east-1"
}
```

STRINS ARE ALWAYS IN DOUBLE QUOTES (NO single quotes)

1] PROVIDER SYNTAX

→ used to configure cloud service provider

```
Provider "Provider-type" {
    attr1 = "Value A"
    attr2 = "Value B"
}
```

```
Provider "aws" {
aws_access_key = "ACCESS_KEY"
Secret_key = "SECRET_KEY"
Region = "us-east-1"
}
```

2] RESOURCE

→ used to define infrastructure Resources.

```
resource TYPE RESOURCE_NAME {
    attr = "value1"
    attrb = "value2"
}
```

EC2 Resource example.

```
resource "aws_instance" "my_web_server" {
    ami = "ami-b376c2ab"
    instance_type = "t2.micro"
}
```

*) `terraform destroy` → destroy every instance that we created & is in teraform state file

```
variable NAME {
    type = "string, list, map, boolean"
    default = "default value"
    description = "used for documentation"
}

variable "region" {}

variable "web_server_ami" {
    default = "ami-5567342"
}
```

1] Strings

```
variable "hc" {
        type = "string"
        default = <<END
          Hello world,
          Rohit
          END }
```

2] Booleans

```
variable "He" {
        default = "true"
        }
```

3] List

```
# implicit definition
  variable "mylist" {
# assign          default = [] }
# assign values to a list
  mylist = [ "value1", "value2", "value3" ]
```

```
# explicit definition.
  variable "sizes" {
        type = "list"
        default = [ "small", "medium", "large" ]
```

4] Map

→ map is a collection of key value pairs

```
variable "HE" {
    type = "map"
    default = {
        "us-west-1" = "ami-6213"
        "us-east-1" = "ami-342"
    }
```

*) Refrencing a variable

→ variable are Refrence using interpolation

${var.variable-name}

eg

```
resource "aws_instance" "my_web_server" {
    instance_type = "t2.micro"
    ami = "${var.web_server}"
}
```

*) Assign Value to Variable

- CLI `$ terraform apply -var "region=us-east-1"`
- file `region = "us-east-1"`
- Environment Variable
  - ↳ export TF_VAR_region = us-east-1

```
# web_server_amis.tf

variable "web_server_amis" {
    type = "map" }
```

Assign key values in terraform.tfvars

```
web_server_amis = {
        "us-east-1" = "ami-b21364"
        "us-west-1" = "ami-2036"
        }
```

★) OUTPUT VARIABLES

→ output variables are displayed when terraform apply is called
→ can be used by other scripts to extract

```
output "web_server_public_ip" {
    value = "${aws_instance.myWebServer.public_ip}"
    }
```

(aws_instance: Resource type; myWebServer: Resource name; public_ip: attribute)

```
$ terraform apply

outputs:
    web_server_Public_ip = 10.0.0.12
```

★) Interpolations In-Depth

→ Reference variables, resource attributes, data sources
→ calls built-in functions
→ use conditionals

1] conditionals:

```
CONDITION ? TRUE_VALUE : FALSE_VALUE

count = "${var.clustered == "true" ? 2 : 1}"
```

(condition; TRUE; false)

*) functions

function_name (arg1, arg2)

eg "${lookup (map, key)}"

*) Templates

→ manage interpolations in long string or text files.

→ templates are difined as data source

*) Math functions

"${3*4+3+3/3}" = 16

*) MANAGING MULTIPLE INSTANCES WITH COUNT

→ count is used to create multiple instances eg the Resources

```
resource "aws_instance" "webserver" {
    count = "2"
}
```

*) Resource Dependency

→ Some Resources may have one more dependencies on another Resources.

eg:- load Balancer with a floating IP address
- DB with Block storage.

→ TF automatically handles implicit & explicit dependencies

→ TF configuration can contain multiple Resources

*) IMPLICIT DEPENDENCY (~~can be defined in different file~~)

```
resource aws_eip "web_server_eip" {
        instance = "${aws_instance.my_web_server.id}"
}
```

Resource Reference Syntax : ${TYPE.NAME.ATTR}

TYPE = aws_instance
NAME = my_web_server
ATTR = id

*) EXPLICIT DEPENDENCY

```
resource "aws_instance" "my_web_server" {
        instance_type = "t2.micro"
        ami = "${lookup(var.web_server_amis, var.Region)}"
        depends_on = ["aws_S3_bucket.static_web_content"]
}
```

*] PROVISIONERS

→ used to bootstrap or initialize a compute instance
1] upload file
2] Run Scripts
3] Install additional Software (chef, Puppet)

* - splat -

i] Local - exec

→ lets you run script on local systems

```
resource "aws_instance" "abc" {
    Provisioner "local-exec" {
    command = "echo ${aws_instance.abc.Private-IP}
                                >> Hello.txt"
    }
}
```

ii] Remote- exec

→ Run Script on Remote System

```
Provisioner "Remote-exec"
  inline = [
        "Sudo yum -y update"
  ]
}
```

iii] File

→ copy from terraform client to the Resource.

```
resource "aws_instance" "my-web server" {
    Provisioner "file" {
        Source = "conf/httpd.conf"
        distination = "etc/apache2/conf/httpd.conf"
    }
}
```

```
ssh-keygen -t rsa -b 4096 -f aws-rsa
```

-t rsa → type of algo.

-b 4096 → size

aws-rsa → file to store keys

↓

linux command to generate Public & Private keys

ingress → allow ssh traffic in (incoming)

egress → allow any traffic out (outgoing)

★) Terraform Data Sources

→ Show the list of EC2 Resources created out of your terraform.

→ Data Sources allow you to fetch and use the data from external Sources that are not managed by terraform itself.

→ Data Sources are Read-Only.

"${ data.TYPE.NAME.ATTR }"

★) Terraform Modules

→ modules are containers for multiple Resources that are used together. They allow you to organize and Reuse your code efficiently

1] Root module → consist of Resources

2] Child module → Root module can call other modules child module can be reused across different configuration

3] Published module →

```
module "my_module" {

    source = "location of module"
    version = "0.1.0"
    CONFIG
}
```

- Terraform Registry (TF Repo)
  - Private Registry
    - Github
    - Generic Git
    - S3 Bucket
    - Local files

→ Modules are simply folders with terraform files

→ root module → current working directory

→ we can give any name to the module

"${module.mod_name.mod_type}"

→ we can create CUSTOM MODULE

TERRAFORM STATE

→ keeps track of current state of your infrastructure as defined in TF configuration files.

→ map Real world Resources to TF configurations

→ State is stored locally by default in JSON

→ Backup is → terraform.tfstate.backup.

★) Importing Resources in TF

1]

```
resource "aws_instance" "web_server" {

}
```

2]

$ terraform import aws_instance.web_server resource_id

↳ get information about the Running Resources

a] Remote TF State

→ TF State can be stored in Remote data store.