Terraform-Basics

- Infrastructure as Code:
  - Introduction: Further Read: [Infrastructure as Code | IBM]
    - Concept where the infrastructure requirements and dependencies are written in a file (in a predefined language).
    - The file is maintained as the definition for infrastructure
      - Optionally stored & maintained in a source code repository[best practise].
    - The definition is used by a tool to provision infrastructure.
  - Advantage:
    - Automated deployment
    - Consistent environments
    - Repeatable process
    - Reusable components
    - Documented architecture
- Generic Terms:
  -

Idempotent	denoting an element of a set which is unchanged in value when multiplied or otherwise operated on by itself.
Mutable	Something that keeps changing or fluctuating.
Declarative	denoting high-level programming languages which can be used to solve problems without requiring the programmer to specify an exact procedure to be followed
Procedural	a type of computer programming language that specifies a series of well-structured steps and procedures within its programming context to compose a program.

- Terraform:
  - Introduction:
    - Built by Hashicorp. It is an Infrastructure provisioning tool.
    - Example of implementation of Infrastructure as Code.
    - Three Editions:

Terraform CLI	Open Source - Free to use. Single code binary file
Terraform Cloud	Commercial public as a service offering. Hosted @ https://app.terraform.io.
Terraform Enterprise	Terraform Cloud implementation in Private instance

    - Note: Terraform documentation is well written and self-explanatory to understand terraform.
    - Links:
      - Generic - Terraform by HashiCorp
      - Registry - Terraform Registry
  - Components

Executable	• Binary that performs the terraform activity

State file	• A source of truth file for terraform. Contains the details of the infrastructure that needs to work on.
	• Has extension of .tfstate . Is in JSON format.
	• Changes will be updated by terraform. **Not to be updated externally(corrupts file)**.
	• Types:
	  • Local - on same host where terraform is running
	  • Remote - on a different host (can be public cloud, terraform cloud etc)

Plugins	• Code modules that serve as Extensions to terraform.

	• One such plugin is called Provider, which defines which component to be worked on . Browse Providers | Terraform Registry
Definition files (terraform files)	• Written in HCL (Hashicorp config language) to define what infrastructure is required.

- Typical type of files:
  -

*.tf	Actual code in HCL format
*.var	variable definitions
*.tstate	State file
*.tfvars	Variable value assignments

- How it works (core workflow & commands):

Terraform Init	Initializes the directory structure to be used by Terraform. Command: $ terraform init
Terraform validate	Validates if the configuration files are correct and there are no errors.
Terraform Plan	The first part of the execution [with option to save the plan]. Steps Involved: • Uses the definition to connect to the provider • collect the necessary information • Compare with state file • Identify dependencies • list out actions that will be performed by terraform to get the infrastructure to the desired state. • Lists out the changes that will be performed ○ Will be denoted by ▪ (+) add (colour coded as green) ▪ (-) remove (colour coded as red) ▪ (-/+) update - usually would mean remove and then add. (colour coded as yellow) Command : $ terraform plan [options] terraform file Note: • If run with -destroy switch instead of creating the resources, will destroy the resources. • -var <variable>=<value> is another way to pass variables to tf files. • -out <filename> will store the plan in an external file.
Terraform Apply	• Apply the terraform plan (i.e. make changes to the infrastructure) • Can be from the last plan generated or from a plan file (stored as *.tfplan) • Will make changes to the tfstate file

- **Important** :
  - Changes made outside of terraform will not be tracked.
  - If there is a change to an infrastructure that is provisioned by terraform outside of terraform. Upon execution, terraform will see it as a variation from desired state and will re-apply per the terraform definition.
  - If there are multiple terraform configuration files, terraform will stitch them all together and then generate the plan. So, you do not have to link each file/ provide any dependencies. For coding simplicity, you can segregate each file based on purpose & terraform will pick them all together.
  - Another important feature is the execution sequence, terraform does not execute blocks completely based on the sequence they are written. While it does start top-down, it also

checks for dependencies and works on the resources automatically based on those dependencies.
    - This dependencies can also be configured manually.
- Hashicorp Configuration Language: [Overview - Configuration Language - Terraform by HashiCorp]
    - Generic Block format:
        - Basic block

```
block_type label_one label_two {
key = value
embedded_block {
key = value
}
}
```

    - Recognised datatypes:

string	- strings Example: string = "blah blah blah"
Number	- numeric values (including decimal) Example: number = 5
Boolian	- true / false Example: bool = true
List	- For storing a list of values of same data type. Example: list = ["abc","def"] Referenced as list[1] (starts with 0)
Map	- Key - Value pair (for any recognised data type) Example: map = {name = "abc", age = 4, school=true} Referenced as map["name"] . Key is the index input.

    - Variable
        - Purpose:
            - To store dynamic content
        - Definitions:

```
variable <variable name>{}
Variable <variable name> { value assignment}
```

            - Example:

```
variable "aws_secret_key" {}
variable "aws_region" { default = "us-east-1" }
```

            - Example:
                - #Specify default variable and type

```
variable "environment_name" {
type = string
default = "development"
}
#Specify variable in file
environment_name = "uat"
#Specify variable in-line
terraform plan –var 'environment_name=production'   <-passing variable during plan
```

                - #Create variable map

```
variable "cidr" {
type = map(string)
default = {
```

```
development = "10.0.0.0/16"
uat = "10.1.0.0/16"
production = "10.2.0.0/16"
}
}
#Use map based on environment
cidr_block = lookup(var.cidr, var.environment_name)
```

- Usage:
  var.<variable name>
  Example: region = "var.aws_region"
- Interpolation can be done using regular operations, but variable needs to be enclosed in ${}
  myimage_name = "myproj-${var.env_type}"

- Expressions, Functions & Loops:
  - HCL has provisions to work with varibles and different data types.
    - Functions: =Functions - Configuration Language - Terraform by HashiCorp
    - Expressions - Expressions - Configuration Language - Terraform by HashiCorp
    - Note: HCL is not a full blown program language, so while there are loops & conditions, they are minimalistic and with constraints.
  - For & IF:
    - For Expression:
      - Usage:
        [for <var> in <list/tuple/map/object>: [actions - with variable referenced as ${var}]
      - Example [for s in var.list : upper(s)]
    - IF Expression :
      - Usage: <CONDITION> ? <TRUE VAL> : <FALSE VAL>
      - Example:
        var.a != "" ? var.a : "default-a"
- Locals: - Simplify Terraform Configuration with Locals | Terraform - HashiCorp Learn
  - Purpose :
    - To represent repeated values in a config file. Something like a constant variable definition.
  - Definition :
    ```
    local {
    variable1 = value
    variable2 = value
    }
    ```
    - Referenced using local.<variable>
  - Example:
    ```
    locals {
     required_tags = {
      project     = var.project_name,
      environment = var.environment
     }
     tags = merge(var.resource_tags, local.required_tags)
    }
    ```
- Provider:
  - Purpose:
    - Defines what type of infrastructure & where it is being provisioned.
    - Mandatory. This block establishes connection to the target.
  - Establishing connection
    - Definition:
      ```
      terraform  {
          required_providers {
          <provider name> {
      ```

source = <path for provider plugin information in terraform registry>
Version = <version of provider plugin>
[options]
} }}

- Example:

```
terraform {
 required_providers {
  aws = {
   source = "hashicorp/aws"
   version = "3.52.0"
  }
 }
}
```

- Provider Block:
  - Purpose:
    - Provides configuration options for the provider connection.
  - Definition:

```
provider <provider name> {
   provider specific parameters  / configuration options
 }
```

  - Example:

```
provider "aws" {
access_key = "var.access_key"
secret_key = "var.secret_key"
region = "var.aws_region"
}
```

- Data Block:
  - Purpose:
    - Store data collected from the target
    - Options and module names are specific to the provider, please check the provider documentation .
      - Data source to use, need to be published by the provider.
  - Definition:

```
data <module name> <variable name> {
   [options]
}
```

  - Example: - aws_ami | Data Sources | hashicorp/aws | Terraform Registry

```
data "aws_ami" "alx" {
most_recent = true
owners = ["amazon"]
filters {} }
```

  - Value is referenced as data.<data sourcename>.<data returned> (chck doc on data type of returned value, as how it is referenced will vary based on that)
    Example: data.aws_ami.architecture
- Resource Block: Resources Overview - Configuration Language - Terraform by HashiCorp
  - Purpose:
    - Reference a resource for working with. For data collection, use the data resource in data block.
      - Example (for same type - AWS AMI):
        - Data source aws_ami | Data Sources | hashicorp/aws | Terraform Registry
        - Resource - aws_ami | Resources | hashicorp/aws | Terraform Registry
    - Options and resource names are specific to the provider, please check the provider documentation .
      - Resource to be used need to be published by the provider

- Definition

  Resource <resource_name> <variable name> {
  [options]
  }
- Example: aws_instance | Resources | hashicorp/aws | Terraform Registry

  resource "aws_instance" "ex"{
  ami = "data.aws_ami.alx.id"
  instance_type = "t2.micro"
  }
- Additional resource manipulation can be done using the documented meta data
  - Count (for total no of resources needed)
    - count.index - represents the running count number (max = count)
  - depends_on = to manually specify inter resource dependencies.
    Example:

    resource "aws_instance" "myservers" {
    count = 2
    tags {
    Name = "customer-${count.index}"
    }
    depends_on = [aws_iam_role_policy.allow_s3]
    }
  - For_each - Resource creations based on conditions. "Each" variable will have the content per iteration.
    Example: (if two S3 buckets to be created with name DEV_MyProj & PROD_MyProj). First iteration creates DEV_MyProj with public_read ACL and then creates PROD_MyProj with private ACL .

    resource "aws_s3_bucket" "mybuckets" {
    for_each = {
    DEV= "public-read"
    PROD = "private"
    }
    bucket = "${each.key}_MyProj"
    acl = each.value
    }

- Output Block:
  - Purpose:
    - If there is a need to display a result after the plan execution.
    - Like a return value.
  - Definition:

    output <variable name> {
    value = <value to store>}
  - Example :

    output "aws_public_ip"
    {
    value = "aws_instance.ex.public_dns "
    }
- Provisioners:
  - Further read - Provisioners - Terraform by HashiCorp
  - Purpose:
    - An alternate method to perform operations or executions.
    - To be used as a last resort only (as these do not maintain state and hard for tracking)
    - Written within the resource block & usually used to perform some task after the resource is created or destroyed
    - Commonly used provisioners:

File	- File Operations , like copy etc.

Local Exec	- Runs commands/ scripts on local machine
Remote Exec	- Runs commands on target machine

- Definition:

```
provisioner <provisioner type> {
[options]
}
```

- Example:

```
resource "aws_instance" "web" {
 # ...

 provisioner "file" {
  source      = "script.sh"
  destination = "/tmp/script.sh"
 }

 provisioner "remote-exec" {
  inline = [
   "chmod +x /tmp/script.sh",
   "/tmp/script.sh args",
  ]
 }
}
```

- **Workspaces**: State: Workspaces - Terraform by HashiCorp
  - Purpose - A Concept in terraform where the terraform statefile is seperated while keeping the other config files same.
    - Common scenario is if we want to use the same configuration for different environments (like DEV/UAT/PROD), you would different state files for each environment. In which case you would create 3 workspaces for each environment.
    - You would create the config files to be dynamic so that execution is based on which workspace it is working in . Typically referencing it using ${terraform.workspace} to identify current workspace
    - Based on workspace, load different variables to which are environment specific
  - Commands:

    terraform workspace new <workspace name>

    Terraform workspace select <workspace name> -> to go to respective workspace.
- **Module:**
  - Two types:
    - Provided by 3rd Party (in terraform registry)
    - User defined
  - Purpose: reusuable components of code. Think functions or modules in other programs, that take some parameters, do the work and return some values.
  - Creating a module:
    - Create a directory and put the necessary code in that directory.
    - If you want a return value, make sure you provide it in the output block.
  - Using a module :

```
module <module name> {
   source = <path of the code files>  # can be local, consul, github or other locations
}
```

    Output is referenced using module.<resourcename>.<output variable>