Terraform Workflow Overview

**Terraform Init**

Working Directory- When init is run its runs the code for aws from the aws registry with the provider files needed to run the files

**State files**

Representation of the world in JSON

Info about the resource and data object

Contain sensitive Info eg. Passwords

Stored locally or remotely

**Terraform Plan**

Terraform Config (Desired State)

Terraform State (Actual State)

**Terraform Apply**

Brings the desired state to the actual state by making sure they are the same

**Terraform Destroy**

**Remote Backend**

Terraform State files can be stored locally or remotely.

If remotely there are advantages of collaboration, encryption etc

**Terraform Cloud**

Terraform {

Backend “remote” {

Organization= “name-of-organization”

Workspaces {

name = “ terraform-course”

}

}

}

**AWS (S3 bucket and DynamoDB used for locking)**

Terraform {

backend “s3” {

bucket = “name-of-bucket”

key = “tf-infra/terraform.tfstate

region = “us-east-1”

dynamodb\_table =”terraform-state-locking” # prevents the terraform apply from running when it is already being done by another developer

encrypt = true

}

}

### Since Ideally will be using terraform/ Iac to provision the infrastructure we will have to use bootstrapping to create the s3 bucket and dynamo\_db environment for the rest of the team ###

1. Start with a local backend and provision the resources which is s3 bucket and dynamo db **NB** hash key has to be lock ID for it to work.
2. Apply the provision the dynamo db and s3 bucket
3. Specify the remote backend for the remaining configuration. This goes on top leaving the remaining configuration unchanged

**Data block references an existing resource within aws where as a resource block creates the resource**

**Variable Types**

**Input Variables**

variable “instance\_type” {

description = “ ec2 instance type”

type = string

default = “t2.micro”

}

var.<name> ----🡪 invocation

**Locals Variables**

For temporary repeatable variables within the code

locals {

service\_name = “My Service”

owner = “Name of Organization”

}

local.<name> ---🡪 invocation

**Output Variables**

output “instance\_ip\_addr” {

value = aws\_instance.instance.public\_ip

}

**Setting Up Input Variables lowest to highest precedence**

Manual entry during plan/apply

Default value in declaration block

TF\_VAR\_<name> environment variables

Terraform.tfvars file

\*.auto.tfvars file

Command line -var or -var-file

**Types and Validation**

Pimitive: strings numbers bool

Complex:

list(<TYPE>)

set (<TYPE>)

map (<TYPE>)

object ({<ATTR NAME> = <TYPE.,..})

tuple ([<TYPE>,..])

**Validation:**

Type checking happens automatically

Custom conditions can be enforced

**Sensitive Data**

**Mark variable as sensitive:**

Sensitive = true

**Pass to terraform apply with :**

TV\_VAR\_variable

-var (retrieved from secret manager at run time

**External secret store**

Eg AWS Secrets Manager

**variables.tf 🡪**

variable “ami” {

description = “Ami for ec2”

type = string

default = “ ami-12312321” #Ubuntu and for us-east-1

sensitive= true # include this for sensitive stuff

}

**Tf vars file** used for non sensitive data

**terraform.tfvars 🡪**

instance\_name = “hello world”

ami = “ami-1243125412512”

instance\_type = “t2.micro”

NB you can have more than one tfvars file, however by default it will refer to the terraform.tfvars file. Unless explicitly stated in the terraform apply code

Eg. Terraform apply -var-file=<name of the file>

**Outputs file** can be used as a place holder for a information yet to be created such as instance ip address

**Outputs.tf🡪**

output “instance\_ip\_addr” {

value = aws\_instance.instance.private\_ip

}

NB for like variables created to hold sensitive info like pass words which are declared in the variables.tf fine include the sensitive = true and input the value during the terraform apply. This is not ideall for automation sake so use secrets manager or some other way

You can also use

Terraform apply -var = “db\_user = myuser” --var =”db\_pass =Wd2SFg”

**Meta-Arguments**

**depends\_on**

Terraform automatically generates dependency graph based on references

If two resources depend on each other, depends\_on specifies that dependency to enforce ordering.

Eg. If i need to provision a role to access an s3 bucket i will have to tell terraform to provision the role before the instance since it depends on it

Eg. resource “aws\_iam\_role\_policy” “name” {

}

resource “aws\_instance” “example” {

.....

....

depends\_on = [

aws\_iam\_role\_policy.name,

]

}

**Count**

Allows for creation of multiple resources/modules from a single block

Useful when the multiple necessary resources are nearly identical

Eg. resource “aws\_instance” “server” {

Count =4 # creates four ec2 instnaces

ami = “ami-12412412”

instance\_type =” t2.micro”

tags = {

Name= “Server ${count.index}”

}

}

**For\_each**

Allows for creation of multiple resources/modules from a single block

Allows more control to customize each resource than count

Eg. locals {

subnet\_ids = toset ( [

“subnet-abcderf”,

“subnet-012442”,

])

}

Resource “aws\_instance” “server” {

for\_each = local.subnet\_ids

ami =”ami1241242”

tabs ={

Name = “Server ${each.key}”

}

**Lifecycle**

* A set of meta arguments to control terraform behaviour for specific
* **create\_before\_destroy can** help with zero downtime deployments
* **ignore\_changes** prevents Terraform from trying to revert metadata being set elsewhere
* **prevent\_destroy** causes Terraform to reject any plans which would destroy this resource
* this is also declared in the resources such aws\_instance block

**Provisioners**

**Eg. Terraform + Ansible**

Perform action on local or remote machine

**Project Organization + Modules**

Modules are containers for multiple resources that are used together. A module consists of a collection of .tf and / .tf.json files kept together in a directory

Modules are the main way to package and reuse resource configuration with terraform

**Types of Modules**

**Root Module**: Default module containing all .tf files in main working directory

**Child Module:** A separate external module referred to a .tf file

Module sources:

**local paths**

module “name” {

source = “../name”

}

**terraform registry**

module “consul’ {

source = “hashicorp/consul/aws”

version = “0.1.0”

}

**Github**

* **HTTPS**

module “example’ {

source = github.com/hashicorp/example?ref231”

}

* **SSH**

module “example” {

source = [git@github.com](mailto:git@github.com): hashicorp/example.git”

}

* **Generic git repo**

module “example” {

source = “git::ssh://username@example.com/storage.git

}

**bitbucket, git, http urls, s3 buckets**

**Input+ Meta-arguments**

Input variables are passed in via module block

**Meta-Arguments:**

**Count**

**For\_each**

**Providers**

**Depends\_on**

**Characteristics of a Good Module**

* Raises the abstraction level from base resource types
* Groups resources in a logical fashion
* Exposes input variables to allow necessary customization + composition
* Provides useful defaults
* Returns outputs to make further integrations possible