Terraform

Akshay Verma

Overview

=> Terraform - What and Why?

=> A Quick view.

=> Launching EC2 with comparing against Cloudformation.

=> State Files: Glance

=> S3 backend.

=> Use of Dynamodb.

=> Remote State (How data is fetched)

=> terraform workspace Vs Infra Environment

=> Terraform Functions

=> Modularization and Variabilization. Why and How?

=> Directory Structure Planning, a few cases.

=> Questions and Cases

KEEP
CALM
AND
AUTOMATE ALL
THE THINGS

Terraform

What:

=> Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently.

Why:

=> Terraform is cool.

=> Infrastructure as a Code.

=> It is completely platform agnostic.

=> A single tool to manage virtual clouds(AWS, VMWare, Azure), supporting services like DNS, Email, or managing some administration in your database.

=> Terraform's speed and operations are exceptional. Plan actually allows us to see what's gonna change.

What is Terraform

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions.

Configuration files describe to Terraform the components needed to run a single application or your entire datacenter. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.

https://www.terraform.io/intro/index.html
https://learn.hashicorp.com/terraform?track=getting-started#getting-started
https://www.terraform.io/docs/providers/aws/
https://github.com/terraform-aws-modules?

Installation

1. Install unzip
   `sudo apt-get install unzip`

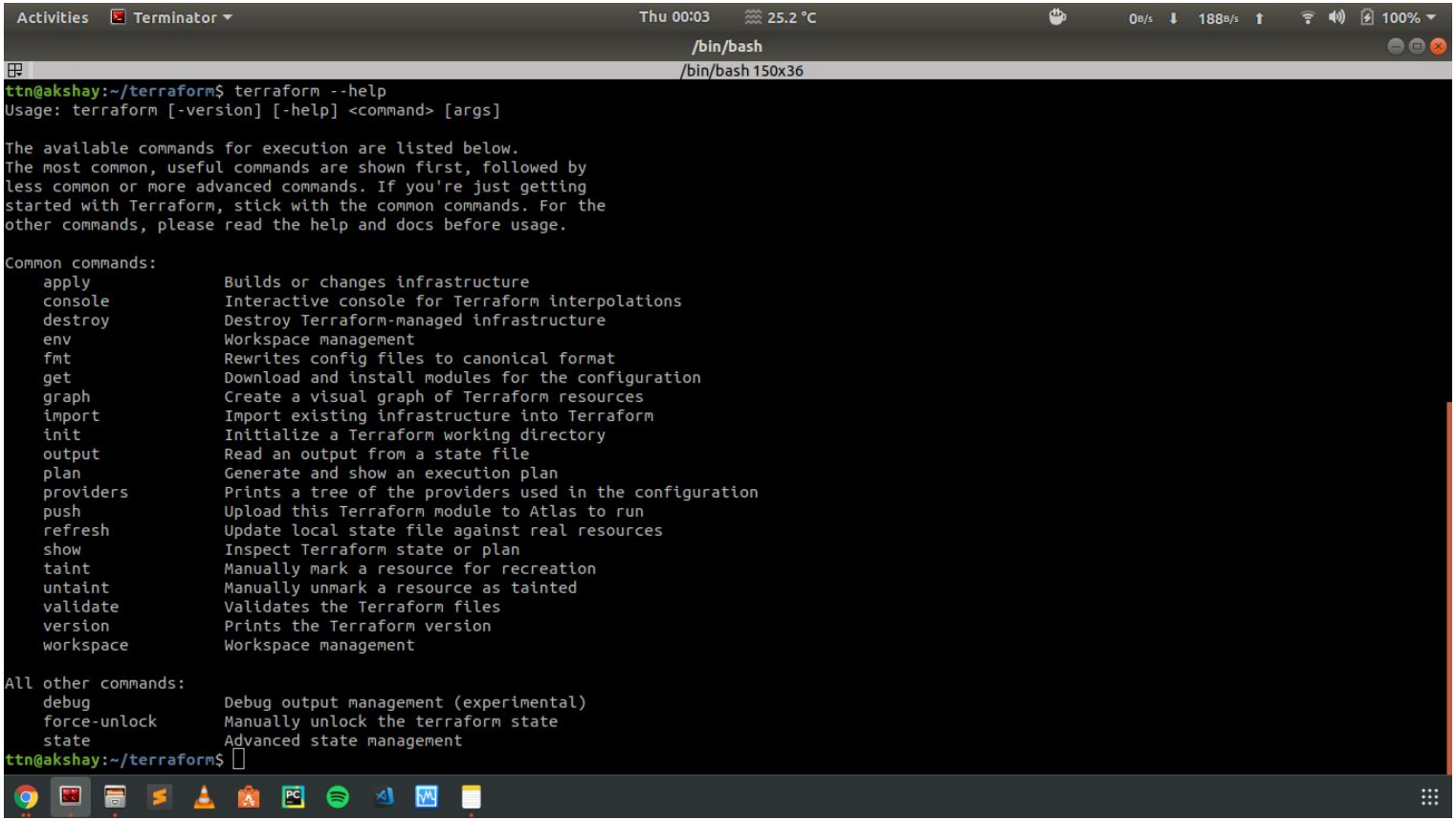
2. Download latest version of the terraform
   `wget https://releases.hashicorp.com/terraform/0.12.24/terraform_0.12.24_linux_amd64.zip`

3. Extract the downloaded file archive
   `unzip terraform_0.12.24_linux_amd64.zip`

4. Move the executable into a directory searched for executables
   `sudo mv terraform /usr/local/bin/`

5. Run it
   `terraform --version`

Terraform Commands



How Does Terraform look like?

```
provider “aws” {
        access_key = "ACCESS_KEY_HERE"
        secret_key = "SECRET_KEY_HERE"
        region     = "us-east-1"
}

resource "aws_instance" "test" {
  ami           = "ami-07ebfd5b3428b6f4d"
  instance_type = "t2.micro"

  tags = {
    Name = "terraform-test"
    owner = "akshay.verma@tothenew.com"
    purpose = "bootcamp-tf"
  }
}
```

The name to reference within Terraform files.

The actual name of bucket and other properties.

Here, the combination “aws_instance” and “test” must be unique.

Explaination

The provider block is used to configure the named provider, in our case "aws". A provider is responsible for creating and managing resources. Multiple provider blocks can exist if a Terraform configuration is composed of multiple providers, which is a common situation.

The resource block defines a resource that exists within the infrastructure. A resource might be a physical component such as an EC2 instance

Launching an EC2

Terraform

```
resource "aws_instance" "Nginx_Server" {
  ami                    = "${var.ami_id}"
  instance_type          = "${var.instance_type}"
  subnet_id              = "${var.subnet_id}"
  vpc_security_group_ids = ["${var.security_groups}"]
  key_name               = "${var.key_name}"
  iam_instance_profile   = "${var.role_name}"
  ebs_optimized          = true
  tags                   = "${var.tags}"
  volume_tags            = "${var.tags}"
}
```

Cloudformation

```
"Nginx_Server":{
        "Type":"AWS::EC2::Instance",
        "Properties":{
                "ImageId": {"Ref":"ESAMIId"},
                "InstanceType": {"Ref":"ESInstanceType"},
                "IamInstanceProfile" : { "Ref": "EC2ProfileForTag" },
                "KeyName": { "Ref":"KeyName"},
                "Tags":[
                {"Key":"Name", Value":  { "Fn::Join":["-",[{"Ref": "Envv"},
                        "nginx-server"]]}}
                ]
        }
}
```

Variable.tf

Variables to override the defaults

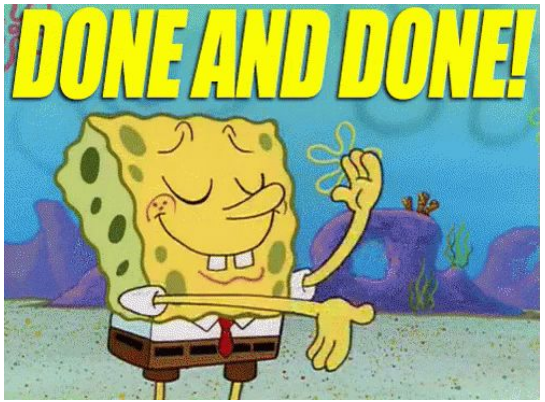
Variable.tf

```
variable "env" {}
```

So how you do it

First You See the Plan..

Then You Apply..



Creating Infrastructure

- terraform init
- terraform plan -var env=$env -out=tfplan
- terraform apply "tfplan"
- terraform destroy -var env=$env

Lets Dive Terraform

State and State Files

=> It is a JSON file which looks like:

State File


```
{
    "version": 3,
    "terraform_version": "0.11.13",
    "serial": 112,
    "lineage": "dcb1eebd-13c6-929e-4928-0486ffa9adb4",
    "modules": [
        { "resources": {
        "aws_instance.utility": {
            "type": "aws_instance",
            "depends_on": [],
            "primary": {
                "id": "i-0ce13100a7a2a2ed2",
                "attributes": {
                    "ami": "ami-0cfe7d2728d424846",
                    "arn": "arn:aws:ec2:ap-south-1:349554833603:instance/i-0ce13100a7a2a2ed2",
                    "associate_public_ip_address": "false",
                    "availability_zone": "ap-south-1b"    ..... (contd)
```

State and State Files

=> Terraform stores metadata and is needed for syncing to real world infrastructure.

=> Terraform must store state about your managed infrastructure and configuration.

=> This state is stored by default in a local file named "terraform.tfstate"

=> Terraform uses this local state to create plans and make changes to your infrastructure.

=> Prior to any operation, Terraform does a refresh to update the state with the real infrastructure.

=> It is a JSON file which looks like:

S3 backend and State Locking

Remote State

=> Allows multiple members of a team work on the same Terraform code.

=> Latest State File is always available to each member.

=> Remote Datastore: s3, consul etc

```
terraform {
  backend "s3" {
        bucket = "my_project_terraformbackend"
        key = "my_project/terraform.tfstate"
        region = "ap-south-1"
  }
}
```

Why This Key?

key = "my_project/terraform.tfstate"

DynamoDB

State Locking:

=> No two members should run TF at same time.

=> Prevents broken state file usage.

=> Uses DynamoDB to store lock state at any point of time.

=> Create a table with key type string named “LockID” which is also a hash key.

```
terraform {
 backend "s3" {
      bucket = "my_project_terraformbackend"
      key = "my_project/terraform.tfstate"
      region = "ap-south-1"
      dynamodb_table = "terraform-lock"
 }
}
```

Fetching data via Remote State

Create Resource in one terraform and use it everywhere??

Like import and export

YES its possible.. Let's see how.

Fetching data via Remote State

1. Export the resource of source TF that needs to be Used. ==>

Source Terraform

```
output "alb-https-listner-arn" {
 value = "${module.app-lb.https-listner-arn}"
}
```

2. Define the remote state details in destination terraform ==>

   The details are of source TF.

Destination Terraform

```
data "terraform_remote_state" "common" {
 backend   = "s3"
 workspace = "<Workspace of source TF directory"
 config {
   bucket = "tatasky-wallet-terrafom-tfstate"
   key    = "<Key of source TF directory"
   region = "ap-south-1"
 }
}
```

3. Extract and Use the exported value as ==>

```
${data.terraform_remote_state.common.alb-https-listner-arn}
```

Workspace and Infrastructure

=> Need to launch exact similar resources without affecting the current one, workspace is your savior.

=> As with other applications, workspace is a segregation between different versions of the same configuration.

=> Can compare with python environments

=> Different States can be maintained without coinciding with one another.

=> Works with a set of “terraform workspace” commands.
https://www.terraform.io/docs/state/workspaces.html

How this can be useful?

Workspace Vs Infra Environments

Workspace Vs AWS accounts

Workspace Vs Projects

Terraform Functions

To transform and combine values and to do a whole lot of different things.

Numeric Functions: abs(), ceil, floor, log, max, min etc

String: lower, upper, split, strrev, title, join, indent etc

Collection: concat, distinct, contains, element, index, keys, length, list, lookup, map,

Encoding: base64encode, base64decode

File System: File, path

Hash: md5

Conversion: tobool, tolist, tostring

Example

```
{for s in var.list : s => upper(s)}
${file("${path.module}/config/filename.tpl")}
map( "Name" ,"${var.project_name_prefix}"  )
 "${merge( var.tags, map( "Name" ,"${var.project_name_prefix}" ) )}"
Count ="${length(var.public_subnets)}"
Cidr = "${lookup(var.public_subnets[count.index], "cidr")}"
"${100 + (count.index * 100)}"


vpc_id    = "${var.create_new_vpc == "true" ? join("",aws_vpc.vpc.*.id) :
var.vpc_id}"
```

```
public_subnets =
[
  {
        az = "ap-south-1a"
        cidr = "10.23.34.0/27"
  },
  {
        az = "ap-south-1b"
        cidr = "10.23.34.32/27"
  }
]
```

Terraform Modules

Modules are the key factors for reusable terraform code.

But What is a module?

Any set of Terraform configuration files in a folder is a module.

We can pass the values while calling a module to provide custom data.

We all are using modules but are we really Modularizing?

Can this be treated as a module?

```
resource "aws_instance" "Nginx_Server" {
  ami                    = "${var.ami_id}"
  instance_type          = "${var.instance_type}"
  subnet_id              = "${var.subnet_id}"
  vpc_security_group_ids = ["${var.security_groups}"]
  key_name               = "${var.key_name}"
  iam_instance_profile   = "${var.role_name}"
  ebs_optimized          = true
  tags                   = "${var.tags}"
  volume_tags            = "${var.tags}"
}
```

A few examples of a good module

A good module must be a complete unit in itself. For example:

- VPC: Creates complete VPC including dynamic subnets, NAT's, route tables etc.
- ECS Service: Task Definition, Target Group, Service, Scaling and Alarms.
- ASG: ASG, Scaling, Alarms, Lifecycle Hooks, Spot configurations etc.
- Networking: CF, Route53, Lambda, WAF etc.

Should not use hard coded values.

Must contain all properties in a configurable form, i.e should allow all configurations to be passed externally.

Should not be dependent over any other external module.

Variabilization

Important part to reuse the code.

Terraform generally use .tfvars files to provide variable values.

Keep everything possible to be variablize so that same code can be used just by changing the variable values.

What structure are we using

- .modules
- terraform

|__ module_name (**sub-dir name**)

|___ aws.tf

|___ Main.tf / (module-wise segregation)

|___ outputs.tf

|___ variable.tf

|__ <env.tfvars>

cd module_name;
Terraform workspace select <env>
Terraform apply -var-file=../<env.tfvars>

Questions ?