Terraform Fundamentals

Providers

A provider is responsible for understanding API interactions and exposing resources. Most providers configure a specific infrastructure platform (either cloud or self-hosted).

Terraform automatically discovers provider requirements from your configuration, including providers used in child modules.

To see the requirements and constraints, run "terraform providers".

```
C:\Users\gagandeep\terra>terraform providers
Providers required by configuration:
.
└── provider[registry.terraform.io/hashicorp/aws]
C:\Users\gagandeep\terra>
```

A provider is responsible for creating and managing resources.

Multiple provider blocks can exist if a Terraform configuration manages resources from different providers.

Resources

- Resources are the most important element in the Terraform language. Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records.

```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

A resource block declares a resource of a given type ("aws_instance") with a given local name ("web"). The name is used to refer to this resource from elsewhere in the same Terraform module but has no significance outside that module's scope.

The resource type and name together serve as an identifier for a given resource and so must be unique within a module.

Resource names must start with a letter or underscore, and may contain only letters, digits, underscores, and dashes.

Provisioners

- Terraform uses provisioners to upload files, run shell scripts, or install and trigger other software like configuration management tools.
- Multiple provisioner blocks can be added to define multiple provisioning steps.
- Terraform treats provisioners differently from other arguments. Provisioners only run when a resource is created but adding a provisioner does not force that resource to be destroyed and recreated.

Configuration files

- Whatever you want to achieve(deploy) using terraform will be achieved with configuration files.
- Configuration files ends with .tf extension (tf.json for json version).
- Terraform uses its own configuration language, designed to allow concise descriptions of infrastructure.
- The Terraform language is declarative, describing an intended goal rather than the steps to reach that goal.
- A group of resources can be gathered into a module, which creates a larger unit of configuration.
- As Terraform's configuration language is declarative, the ordering of blocks is generally not significant. Terraform automatically processes resources in the correct order based on relationships defined between them in configuration

Example

- You can write up the terraform code in hashicorp Language – HCL.
- Your configuration file will always endup with .tf extension

```
provider "aws" {
  region = "us-east-2"
 access_key = "AKIAJB2KQBDLH56XQEYA"
  secret_key = "rNNWWuzvBpp+v//OXCB10Zr2OVuPI3iayxXXStPs"
}

resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"

  tags = {
    Name = "Techlanders-aws-ec2-instance"
  }
}

output "myawsserver" {
  value = "${aws_instance.myawsserver.public_ip}"
}
```

Terraform Workflow

Few Steps to work with terraform:

1) Set the Scope - Confirm what resources need to be created for a given project.

2) Author - Create the configuration file in HCL based on the scoped parameters

3) Run terraform init to initialize the plugins and modules

4) Run terraform validate to validate the template

5) Do terraform plan

6) Run terraform apply to apply the changes

Terraform validate

- Terraform validate will validate the terraform configuration file
- It'll through error for syntax issues:

```
[root@TechLanders aws]# terraform validate
Success! The configuration is valid.

[root@TechLanders aws]#
```

Terraform init

- Terraform init will initialize the modules and plugins.
- If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory.
- If you forget running init, terraform plan/apply will remind you about initialization.
- Terraform init will download the connection plugins from Repository “registry.terraform.io” under your current working directory/.terraform:

```
[root@TechLanders plugins]# pwd
/root/aws/.terraform/plugins
[root@TechLanders plugins]# ls -l
total 4
drwxr-xr-x. 3 root root  23 Aug 15 07:06 registry.terraform.io
-rw-r--r--. 1 root root 136 Aug 15 07:06 selections.json
[root@TechLanders plugins]#
```

- Important concept:
  - Always make a best practice to initialize the terraform modules with versions. i.e.
    hashicorp/aws: version = "~> 3.2.0"

Example

- Perform Terraform Init:

```
[root@TechLanders aws]# terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v3.2.0...
- Installed hashicorp/aws v3.2.0 (signed by HashiCorp)

The following providers do not have any version constraints in configuration,
so the latest version was installed.
To prevent automatic upgrades to new major versions that may contain breaking
changes, we recommend adding version constraints in a required_providers block
in your configuration, with the constraint strings suggested below.
* hashicorp/aws: version = "~> 3.2.0"

Terraform has been successfully initialized!

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
[root@TechLanders aws]#
```

Terraform plan

- terraform plan will create an execution plan and will update you what changes it going to make.
- It’ll update you upfront what its gonna add, change or destroy.
- Terraform will automatically resolve the dependency between components- which to be created first and which in last.

```
[root@TechLanders aws]# terraform plan

Refreshing Terraform state in-memory prior to plan...

The refreshed state will be used to calculate this plan but will not be persisted to local or remote state storage.

An execution plan has been generated and is shown below. Resource actions are indicated with the following symbols:

  + create

Terraform will perform the following actions:

  # aws_instance.myserver will be created

  + resource "aws_instance" "myserver" {

      + ami                          = "ami-06b35f67f1340a795"

      + arn                          = (known after apply)

Plan: 1 to add, 0 to change, 0 to destroy.
```

Terraform apply

- Terraform apply will apply the changes.
- Before it applies changes, it'll showcase changes again and will ask to confirm to move ahead:

```
[root@TechLanders aws]# terraform apply

An execution plan has been generated and is shown below. Resource actions are indicated with the following symbols:

  + create

Do you want to perform these actions?   Terraform will perform the actions described above.   Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_instance.myserver: Creating...

aws_instance.myserver: Still creating... [10s elapsed]

aws_instance.myserver: Still creating... [20s elapsed]

aws_instance.myserver: Creation complete after 21s [id=i-0a63756c96d338801]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

[root@TechLanders aws]#
```

Terraform apply

- Terraform apply will create **tfstate** file to maintain the desired state:

```
[root@TechLanders aws]# ls -l
total 8
-rw-r--r--. 1 root root  234 Aug 15 07:06 myinfra.tf
-rw-r--r--. 1 root root 3209 Aug 15 08:02 terraform.tfstate
[root@TechLanders aws]# cat terraform.tfstate
{
  "version": 4,
  "terraform_version": "0.13.0",
  "serial": 1,
  "lineage": "7f7e0e15-95ef-d8fa-b1cd-12024aed5fa6",
  "outputs": {},
  "resources": [
   "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-06b35f67f1340a795",
            "arn": "arn:aws:ec2:us-east-2:677729060277:instance/i-0a63756c96d338801",
```

- Note: -auto-approve option can be given alongwith terraform apply to avoid the human intervention.

Terraform show

- Terraform show will show the current state of the environment been created by your config file:

```
[root@ip-172-31-6-233 aws]# terraform show
# aws_instance.myserver:
resource "aws_instance" "myserver" {
   ami                          = "ami-06b35f67f1340a795"
   arn                          = "arn:aws:ec2:us-east-2:677729060277:instance/i-0a63756c96d338801"
   associate_public_ip_address  = true
   availability_zone            = "us-east-2a"
   cpu_core_count               = 1
   cpu_threads_per_core         = 1
----
----
```

Idempotency

- Run Terraform apply again and check the status of the server.

```
[root@TechLanders aws]# terraform apply
aws_instance.myserver: Refreshing state... [id=i-0a63756c96d338801]

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
[root@TechLanders aws]#
```

- Stop the server and then check. You'll have no change, as server still exists, its just stopped.
- Run Terraform Plan to check the status:

```
[root@TechLanders aws]# terraform plan

    Refreshing Terraform state in-memory prior to plan...

    The refreshed state will be used to calculate this plan, but will not be  persisted to local or remote state storage.

    aws_instance.myserver: Refreshing state... [id=i-0a63756c96d338801]

    No changes. Infrastructure is up-to-date.

    This means that Terraform did not detect any differences between your configuration and real physical resources that exist. As a result, no actions need to be performed.

    [root@TechLanders aws]#
```

Desired State Maintenance (DSC)

- Delete the newly created server and then check for the terraform plan

```
[root@TechLanders aws]# terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

aws_instance.myserver: Refreshing state... [id=i-0a63756c96d338801]
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
Terraform will perform the following actions:
  # aws_instance.myserver will be created
  + resource "aws_instance" "myserver" {
```

- Run terraform apply command again and witness the provisioning of new server on console.

```
[root@TechLanders aws]# terraform apply
aws_instance.myserver: Refreshing state... [id=i-0a63756c96d338801]
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
Terraform will perform the following actions:
  # aws_instance.myserver will be created
```

Infrastructure as Code

- Modify your template file to change the instance size from t2.micro to t2.small and plan/apply the changes:

```
[root@TechLanders aws]# cat myinfra.tf
resource "aws_instance" "myserver" {
ami = "ami-06b35f67f1340a795"
instance_type = "t2.small"
}
[root@TechLanders aws]#
```

- Run terraform plan and apply again to check the differences

```
[root@TechLanders aws]# terraform apply
aws_instance.myserver: Refreshing state... [id=i-0a1f8a600cb968c7c]
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  ~ update in-place
Plan: 0 to add, 1 to change, 0 to destroy.
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.
  Enter a value: yes
aws_instance.myserver: Modifying... [id=i-0a1f8a600cb968c7c]
```

Refreshing the state

- In case the requirement is to just check for any updates been done in the running environment, we can run terraform refresh command:

```
C:\Users\gagandeep\Desktop\terraform>terraform refresh

google_compute_network.vpc_network: Refreshing state... [id=projects/accenture-286519/global/networks/terraform-net3]

google_compute_address.vm_static_ip: Refreshing state... [id=projects/accenture-286519/regions/us-central1/addresses/terraform-static-ip1]

google_compute_instance.vm_instance1: Refreshing state... [id=projects/accenture-286519/zones/us-central1-b/instances/terraform-instance1]


C:\Users\gagandeep\Desktop\terraform>
```

Destroying Infra in one go

- Terraform destroy will destroy the infrastructure in one go by using your tfstate file.

```
[root@TechLanders aws]# terraform destroy
aws_instance.myserver: Refreshing state... [id=i-0a1f8a600cb968c7c]
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy
Terraform will perform the following actions:
  # aws_instance.myserver will be destroyed
  - resource "aws_instance" "myserver" {
      - ami  = "ami-06b35f67f1340a795“
 Enter a value: yes
aws_instance.myserver: Destroying... [id=i-0a1f8a600cb968c7c]
aws_instance.myserver: Still destroying... [id=i-0a1f8a600cb968c7c, 10s elapsed]
aws_instance.myserver: Still destroying... [id=i-0a1f8a600cb968c7c, 20s elapsed]
aws_instance.myserver: Destruction complete after 29s
Destroy complete! Resources: 1 destroyed.
```

Destroying Infra

- Terraform destroy can also delete selected resources given with –target option and can also be auto-approved with –auto-approve option. But it is always recommended to modify the configuration file instead of –target.

```
C:\Users\gagandeep\Desktop\terraform>terraform destroy -target=google_compute_instance.vm_instance2 -auto-approve

google_compute_network.vpc_network: Refreshing state... [id=projects/accenture-286519/global/networks/terraform-net3]

google_compute_instance.vm_instance2: Refreshing state... [id=projects/accenture-286519/zones/us-central1-b/instances/terraform-instance2]

google_compute_instance.vm_instance2: Destroying... [id=projects/accenture-286519/zones/us-central1-b/instances/terraform-instance2]

google_compute_instance.vm_instance2: Still destroying... [id=projects/accenture-286519/zones/us-central1-b/instances/terraform-instance2, 10s elapsed]

google_compute_instance.vm_instance2: Still destroying... [id=projects/accenture-286519/zones/us-central1-b/instances/terraform-instance2, 20s elapsed]

google_compute_instance.vm_instance2: Destruction complete after 24s

Warning: Resource targeting is in effect

You are creating a plan with the -target option, which means that the result of this plan may not represent all of the changes requested by the current
configuration.

The -target option is not for routine use and is provided only for  exceptional situations such as recovering from errors or mistakes, or when Terraform specifically
suggests to use it as part of an error message.
```

Note: Multiple –target options are supported as well.

Output from a run

Terraform provides output for every run and same can be used to list the resources details which are created using help of Terraform:

```
provider "aws" {
  region = "us-east-2"
 access_key = "AKIAJB2KQH56XQEYA"
  secret_key = "rNNWWuzvBpp+v"
}
resource "aws_instance" "myawsserver" {
  ami = "ami-0a54aef4ef3b5f881"
  instance_type = "t2.small"
  tags = {
    Name = "Techlanders-aws-ec2-instance"
    Env = "Prod"
  }
}
output "myawsserver-ip" {
  value = "${aws_instance.myawsserver.public_ip}"
}
```

Using Resource values

Create a GCP instance with network instance: Add below code to the file:

```
resource "google_compute_instance" "vm_instance" {
  name         = "terraform-instance"
  machine_type = "f1-micro"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
    }
  }

  network_interface {
    network = google_compute_network.vpc_network.name
    access_config {
    }
  }
}
```

Working with change

Modify your terraform file and add tags/labels to it and run terraform plan/apply again:

```
resource "google_compute_instance" "vm_instance" {
  name         = "terraform-instance"
  machine_type = "f1-micro“
  tags  = [“web”, “dev”]

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
    }
  }

  network_interface {
    network = google_compute_network.vpc_network.name
    access_config {
    }
  }
}
```

Notedown the output of terraform plan stating it’ll be an in-place upgrade

Working with change

Changes are of two types:

- Up-date In-place
- Disruptive

So always be careful with what you are adding/modifying

Update in-place

Update in-place will ensure your existing resources intact and modify the existing resources only. Here also based on what configuration is required to be changed, server may or may-not shutdown.

- For example, if you add IP address to a server, reboot will not be required.

```
  network_interface {
    network = google_compute_network.vpc_network.name
    access_config {
     nat_ip = google_compute_address.vm_static_ip.address
    }
  }
}
resource "google_compute_address" "vm_static_ip" {
  name = "terraform-static-ip"
}
```

- On the other side modifying the server size can't be done live. It needs a stop and start of the server. For same you need to grant permission in configuration file:

```
resource "google_compute_instance" "vm_instance" {
  name         = "terraform-instance"
  machine_type = "g1-small"
  tags = ["web", "dev", "client1"]
   allow_stopping_for_update = true
  boot_disk {
    initialize_params {
      image = "cos-cloud/cos-stable"
    }
  }
```

Update - Disruptive

Disruptive updates require a resource to be deleted and recreated.

For example, modifying the image type for an instance will require instance to be deleted and re-created.

Modify the image type to g1-small in config file and check the output of terraform plan:

```
machine_type = "g1-small"

C:\Users\gagandeep\terra>terraform plan

Refreshing Terraform state in-memory prior to plan...

The refreshed state will be used to calculate this plan, but will not be

persisted to local or remote state storage.

An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

-/+ destroy and then create replacement

Terraform will perform the following actions:

  # google_compute_instance.vm_instance must be replaced

Plan: 1 to add, 0 to change, 1 to destroy.
```

Working with change

Now let's see an example of **disruptive** change:

Replace the boot disk of your configuration with cos-cloud/cos-stable or any other AMI and re-run terraform plan:

```
C:\Users\gagandeep\terra>terraform plan

Resource actions are indicated with the following symbols:

-/+ destroy and then create replacement

Terraform will perform the following actions:

  # google_compute_instance.vm_instance must be replaced

-/+ resource "google_compute_instance" "vm_instance" {

--

 ~ initialize_params {

         ~ image    = "https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/debian-9-stretch-v20200805"  ->  "cos-cloud/cos-stable"  #
forces replacement

Plan: 1 to add, 0 to change, 1 to destroy.
```

Changes outside of terraform

Changes which occurred outside of terraform are unwanted changes and if anything which is modified outside of terraform is detected, same will be marked in state files and will be corrected at next apply.

- Run terraform show command to check current required state of infrastructure.
- Modify the Labels (add a label) of a terraform instance from GCP console.
- Run terraform plan to check the behavior of terraform against the changes
- Check the terraform show command to view state file
- Check terraform refresh command to update the state frontend
- Run terraform apply to revert the changes
- Check the terraform refresh/show command as well as console again to validate the reversion of changes.

Resource Dependencies

- There are two types of dependencies available in terraform:
  - ➢ Implicit - Dependency automatically detected and Hierarchy map automatically created by terraform
  - ➢ Explicit - The depends_on argument can be added to any resource and accepts a list of resources to create explicit dependencies on resources.
- Terraform uses dependency information to determine the correct order in which to create and update different resources.

Implicit Dependencies

- Real-world infrastructure has a diverse set of resources and resource types.
- Dependencies among resources are obvious and should be maintained during provisioning. For e.g. Creating a network first than a Virtual machine; and creating a static IP before a VM is initialized and attaching that IP to it.
- Try adding below resource to your configuration file and add a link of same in your Instance network interface:

```
network_interface {
network = google_compute_network.vpc_network.self_link
access_config {
nat_ip = google_compute_address.vm_static_ip.address
 }
 }
resource "google_compute_address" "vm_static_ip" {
 name = "terraform-static-ip"
}
```

Implicit Dependencies

In the previous example, when Terraform reads this configuration, it will:

- Ensure that vm_static_ip is created before vm_instance
- Save the properties of vm_static_ip in the state
- Set nat_ip to the value of the vm_static_ip.address property

You can put your resources here and there in configuration file and terraform will automatically build a dependency map between them.

Implicit dependencies via interpolation expressions are the primary way to inform Terraform about these relationships, and should be used whenever possible.

Explicit Dependencies

- Sometimes there are dependencies between resources that are not visible to Terraform. The depends_on argument can be added to any resource and accepts a list of resources to create explicit dependencies for.
- For example, perhaps an application we will run on our instance expects to use a specific Cloud Storage bucket, but that dependency is configured inside the application code and thus not visible to Terraform. In that case, we can use depends_on to explicitly declare the dependency.

```
resource "google_storage_bucket" "example_bucket" {
  name     = "<UNIQUE-BUCKET-NAME>"
  location = "US"
  website {
    main_page_suffix = "index.html"
    not_found_page   = "404.html"
  }
}
resource "google_compute_instance" "another_instance" {
depends_on = [google_storage_bucket.example_bucket]
  name         = "terraform-instance-2"
  machine_type = "f1-micro"
  boot_disk {
    initialize_params {
      image = "cos-cloud/cos-stable"
    }
  }
  network_interface {
    network = google_compute_network.vpc_network.self_link
    access_config {
    }
  }
}
```

Explicit Dependencies

- Multiple resource dependencies can also be created:

```
# Create a new instance that uses the bucket
resource "google_compute_instance" "another_instance" {
  # Tells Terraform that this VM instance must be created only after the
  # storage bucket has been created.
  depends_on = [google_storage_bucket.example_bucket1, google_compute_instance.vm_instance]
  name         = "terraform-instance-2"
  machine_type = "f1-micro"

  boot_disk {
    initialize_params {
      image = "cos-cloud/cos-stable"
    }
  }

  network_interface {
    network = google_compute_network.vpc_network.self_link
    access_config {
    }
  }
}
```

Backup

- Just run terraform destroy or terraform apply and cancel it. Cross-check for terraform.tfstate.backup file which is being created as backup for your statefile.

```
C:\Users\gagandeep\terra>dir

16-08-2020  00:24    <DIR>          .
16-08-2020  00:24    <DIR>          ..
16-08-2020  00:12    <DIR>          .terraform
16-08-2020  00:24               226 .terraform.tfstate.lock.info
16-08-2020  00:08               243 myinfra.tf
15-08-2020  11:45        85,426,504 terraform.exe
16-08-2020  00:22             3,203 terraform.tfstate
16-08-2020  00:22             3,205 terraform.tfstate.backup
               5 File(s)     85,433,381 bytes
               3 Dir(s)  735,488,614,400 bytes free

C:\Users\gagandeep\terra>
```

Note: Terraform determines the order in which things must be destroyed. For e.g. GCP/AWS won't allow a VPC network to be deleted if there are resources still in it, so Terraform waits until the instance is destroyed before destroying the network.

Terraform plan – saving plans

- Even you can save the terraform plan output for a later reference and then apply same to terraform apply command:

```
C:\Users\gagandeep\terra>terraform plan -out t1
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.
---
C:\Users\gagandeep\terra>terraform apply t1
google_compute_address.vm_static_ip: Creating...
google_compute_address.vm_static_ip: Creation complete after 5s [id=projects/accenture-286519/regions/us-central1/addresses/terraform-
static-ip]
google_compute_instance.vm_instance: Creating...
```

- You can create multiple plans and then execute one out of them, once you have finalized the stuff.

- After running terraform apply, your plan files become stale and can no longer be used.

```
C:\Users\gagandeep\terra>terraform apply t1
Error: Saved plan is stale
The given plan file can no longer be applied because the state was changed by
another operation after the plan was created.
C:\Users\gagandeep\terra>
```

Terraform Advanced

Provisioners

- Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.
- Running Provisioners can help you to execute stuff as per requirement
- The local-exec provisioner executes a command locally on the machine running Terraform, not the VM instance itself.
- Terraform don't encourage the use of provisioners, as they add complexity and uncertainty to terraform usage. Hashicorp recommends resolving your requirement using other techniques first, and use provisioners only if there is no other option left.
- When deploying virtual machines or other similar compute resources, we often need to pass in data about other related infrastructure that the software on that server will need to do its job.
- **Note:** Provisioners should only be used as a last resort. For most common situations there are better alternatives.

Provisioners

- Provisioners also add a considerable amount of complexity and uncertainty to Terraform usage.
- Firstly, Terraform cannot model the actions of provisioners as part of a plan because they can in principle take any action.
- Secondly, successful use of provisioners requires coordinating many more details than Terraform usage usually requires - direct network access to your servers, issuing Terraform credentials to log in, making sure that all of the necessary external software is installed, etc.
- Some use cases:
  - Passing data into virtual machines and other compute resources
  - Running configuration management software

Local-exec Provisioners

- Running Provisioners can help you to execute stuff as per requirement
- The local-exec provisioner executes a command locally on the machine running Terraform, not the VM instance itself.

```
 resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"
  key_name = "test1"

  tags = {
    Name = "Techlanders-aws-ec2-instance"
    env = "test"
  }
  provisioner "local-exec" {
    command = "echo The servers IP address is ${self.private_ip} && echo ${self.private_ip} myawsserver >> /etc/hosts"
  }
```

Remote-Exec Provisioners

- Remote-Exec provisioner helps you to execute commands on next machine:

```
resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"
  key_name = "test1"
provisioner "remote-exec" {
    inline = [
     "touch /tmp/gagandeep",
      "sudo mkdir /root/gagan"
     ]
 connection {
    type     = "ssh"
    user     = "ec2-user"
    insecure = "true"
    private_key = "${file("test1.pem")}"
    host     =  aws_instance.myawsserver.public_ip
  }
}
}
```

Node Tainting

Tainting a Node

- In case there is a requirement to delete and recreate a resource, you can mark same in Terraform to tell terraform to do so. Terraform taint does so. We can manually mark a resource as tainted, forcing a destroy and recreate on the next plan/apply.
- Forcing the recreation of a resource is useful when you want a certain side effect of recreation that is not visible in the attributes of a resource. For example: re-running provisioners will cause the node to be different or rebooting the machine from a base image will cause new startup scripts to run.
- Tainting a resource for recreation may affect resources that depend on the newly tainted resource

Tainting a Node

```
[root@ip-172-31-38-249 aws]# terraform taint aws_instance.myawsserver
Resource instance aws_instance.myawsserver has been marked as tainted.
[root@ip-172-31-38-249 aws]# terraform plan
Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement
Terraform will perform the following actions:
  # aws_instance.myawsserver is tainted, so must be replaced
-/+ resource "aws_instance" "myawsserver"
```

Note: This command on its own will not modify infrastructure. For same you'll have to run terraform apply.

- You can use untaint command to untaint a node.

```
[root@ip-172-31-38-249 aws]# terraform untaint aws_instance.myawsserver
Resource instance aws_instance.myawsserver has been successfully untainted.
[root@ip-172-31-38-249 aws]#
```

LAB

- Create a Server (in Ohio region) but code should work anywhere
- Need to fetch Public subnet ID and Private Subnet ID of default VPC of that region
- use public subnet if server is web (take it as input from client), or private subnet of its any other server

Multiple Providers

- Same Providers with multiple alias can be given for region or attributes change:

```
provider "aws" {
  region = "us-east-2"
  access_key = "AKIAJB2KQBDL56XQEYA"
  secret_key = "rNNWuzvBpp+v//XCB10Zr2OVuPI3iayxXXStPs"
  alias  = "useast2"
}

provider "aws" {
  region = "us-east-1"
  access_key = "AKIAJB2KQBD56XQEYA"
  secret_key = "rNNWuzvBpp//B10Zr2OVuPI3iayxXXStPs"
  alias  = "useast1"
}
```

Multiple Providers

- Provide the provider name in resource:

```
resource "aws_instance" "myawsserver1" {
  ami = "ami-0c94855ba95c71c99"
  instance_type = "t2.micro"
  provider = aws.useast1
  tags = {
    Name = "Techlanders-aws-ec2-instance1"
    Env = "Prod"
  }
}
resource "aws_instance" "myawsserver2" {
  ami = "ami-0603cbe34fd08cb81"
  provider = aws.useast2
  instance_type = "t2.micro"

  tags = {
    Name = "Techlanders-aws-ec2-instance2"
    Env = "Prod"
  }
}
```

Variables

- To become truly shareable and version controlled as well as to avoid hardcoding, we need to parameterize the configurations. Same can be achieved through input variables in Terraform. Variables can be defined in different .tf files and usually we define it in variable.tf or files ending with .tfvars file.

```
variable "project" { }
variable "credentials_file" { }
variable "region" {
  default = "us-central1"
}
variable "zone" {
  default = "us-central1-c"
}
```

Variables

- Variables can be of different types, based on terraform versions:
  - Strings

```
variable "project" {
 type = string }
```

  - Numbers

```
variable "web_instance_count" {
 type    = number
 default = 1 }
```

  - Lists

```
variable "cidrs" { default = ["10.0.0.0/16”] }
```

  - Maps

```
variable "machine_types" {
 type    = map
 default = {
   dev  = "f1-micro"
   test = "n1-highcpu-32"
   prod = "n1-highcpu-32"
 }
```

Variables

- Variables can be assigned via different ways:
  - Via UI
  - Via command line flags:

    ```
    terraform plan -var 'project=<PROJECT_ID>’
    ```

  – From .tfvars file
  – From environment variables like TF_VAR_name

Variables

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Lab: Variables

1) Declare AMI as variable and use same in your aws_instance resource

2) Define the value of AMI(with AMIID) inside terraform.tfvars file

3) terraform plan

4) Rename terraform.tfvars with abc.tfvars

5) Run terraform plan again

6) Run terraform plan with -var-file=abc.tfvars and see the outputs

7) Run terraform plan with -var ami="AMI_ID"

Variables

```
resource "aws_instance" "myawsserver1" {
  ami = var.ami["us-east-1"]
  instance_type = var.instance_type
  provider = aws.useast1
  tags = {
    Name = "Techlanders-aws-ec2-instance1"
    Env = "Prod"
  }
}
variable "instance_type" {
  default = "t2.micro"
}

variable "ami" {
  type = "map"
default = {
    us-east-1     = "ami-0c94855ba95c71c99"
    us-east-2     = "ami-0603cbe34fd08cb81"
  }
}
```

Variables Definition Precedence

Terraform loads variables in the following order, with later sources taking precedence over earlier ones:

- Environment variables
- The terraform.tfvars file, if present.
- The terraform.tfvars.json file, if present.
- Any *.auto.tfvars or *.auto.tfvars.json files, processed in lexical order of their filenames.
- Any -var and -var-file options on the command line, in the order they are provided. (This includes variables set by a Terraform Cloud workspace.)

Tfvars files

```
[root@ip-172-31-38-249 third-demo]# cat prod.tfvars
instance_type =  "t2.small"

ami = {
   us-east-1     = "ami-0dba2cb6798deb6d8"
   us-east-2     = "ami-07efac79022b86107 "
 }

[root@ip-172-31-38-249 third-demo]# cat dev.tfvars
instance_type =  "t2.medium"

ami = {
   us-east-1     = "ami-0dba2cb6798deb6d8"
   us-east-2     = "ami-07efac79022b86107 "
 }

[root@ip-172-31-38-249 third-demo]#
```

Executions

```
[root@ip-172-31-38-249 third-demo]# terraform plan
An execution plan has been generated and is shown below.
+ resource "aws_instance" "myawsserver1" {
     + ami                          = "ami-0c94855ba95c71c99"
+ instance_type                   = "t2.micro"

[root@ip-172-31-38-249 third-demo]# terraform plan -var-file="dev.tfvars"
Refreshing Terraform state in-memory prior to plan...
 # aws_instance.myawsserver1 will be created
  + resource "aws_instance" "myawsserver1" {
     + ami                          = "ami-0dba2cb6798deb6d8"
+ instance_type                   = "t2.medium”

[root@ip-172-31-38-249 third-demo]# terraform plan -var "instance_type=t2.nano"
Refreshing Terraform state in-memory prior to plan...
 # aws_instance.myawsserver1 will be created
  + resource "aws_instance" "myawsserver1" {
     + ami                          = "ami-0c94855ba95c71c99"
+ instance_type                   = "t2.nano"
```

Loops

- Terraform offers several different looping constructs, each intended to be used in a slightly different scenario:
    - count parameter: loop over resources.
    - for_each expressions: loop over resources and inline blocks within a resource.
    - for expressions: loop over lists and maps.

Loops - count

- Depending on resource types, it'll take count values to create number of resources:

```
provider "aws" {
  region = "us-east-2"
 access_key = "AKIAJB2KQBDLH56XQEYA"
  secret_key = "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"
}
resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"
  key_name = "test1"
  count = 2
  tags = {
    Name = "Techlanders-aws-ec2-instance.${count.index}"
    env = "test"
  }
}
output "Private-IP-0" {
 value = aws_instance.myawsserver.0.private_ip
}
output "Private-IP-1" {
 value = aws_instance.myawsserver.1.private_ip
}
```

Loops - count

```
variable "server_names" {
  description = "Create virtual machines with these names"
  type        = list(string)
  default     = ["myvm1", "myvm2"]
}

resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"
  key_name = "test1"
  count = length(var.server_names)
  tags = {
    Name = var.server_names[count.index]
    env = "test"
  }
}

output "Private-IP" {
 value = aws_instance.myawsserver[*].private_ip
}
```

for and for-each

- COUNT have its own limitations. Delete a string from count from previous example and then look at the Terraform behavior. If you remove an item from the middle of the list, Terraform will delete every resource after that item and then recreate those resources again from scratch.
- COUNT can't be used with-in resource.
- Based on complexity of your playbook, you can use for and for_each loops in your configuration files.
- This is similar to loops in Programming languages.
- https://www.hashicorp.com/blog/hashicorp-terraform-0-12-preview-for-and-for-each/

for-each

```
variable "server_names" {
  description = "Create virtual machines with these names"
  type        = list(string)
  default     = ["vm1", "vm2"]
}

resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"
  key_name = "test1"
  for_each = toset(var.server_names)
  tags = {
    Name = each.value
    env = "test"
  }
}

output "Private-IP" {
# As for_each loop is a map, you have to modify the syntax to get the values printed
 value = values(aws_instance.myawsserver)[*].private_ip
}
```