**Terraform**

There are various types of tools that can allow you to deploy infrastructure as a code:

* - Terraform
* - CloudFormation
* - Heat
* - Ansible
* - SaltStack
* - Chef,Puppet and others

**Terraform** is an open-source infrastructure as code software tool that provides a consistent CLI workflow to manage hundreds of cloud services. Terraform codifies cloud APIs into declarative configuration files.

**Features:**

1) Supports multiple platforms, has hundreds of providers

2) simple configuration language and faster learning curve

3) easy integration with configuration management tools like ansible

4) easily extensible with the help of plugins

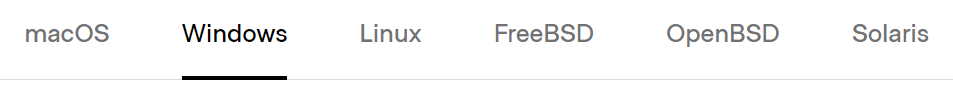
5) It's Free!!

**Installation:**

Terraform installation is very simple. You have to download one binary file and untar it.

[Downloads | Terraform by HashiCorp](https://www.terraform.io/downloads)

Terraform will support below operating systems to install.

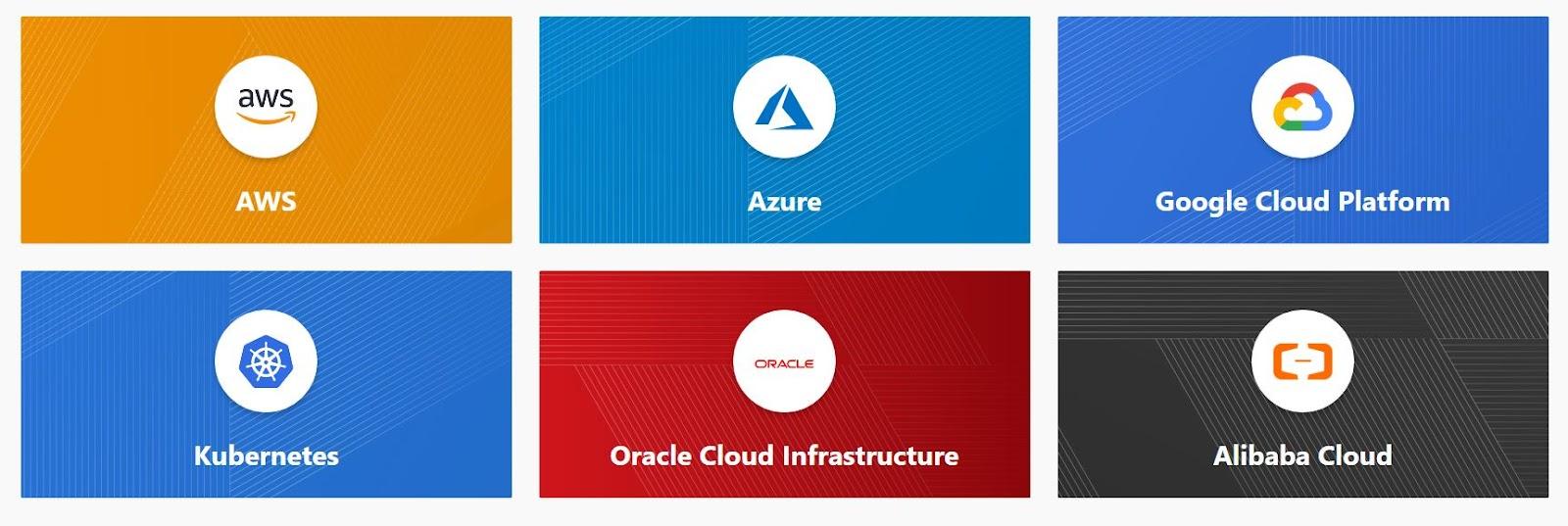


## **Provider and Resources**

Overview of Providers:

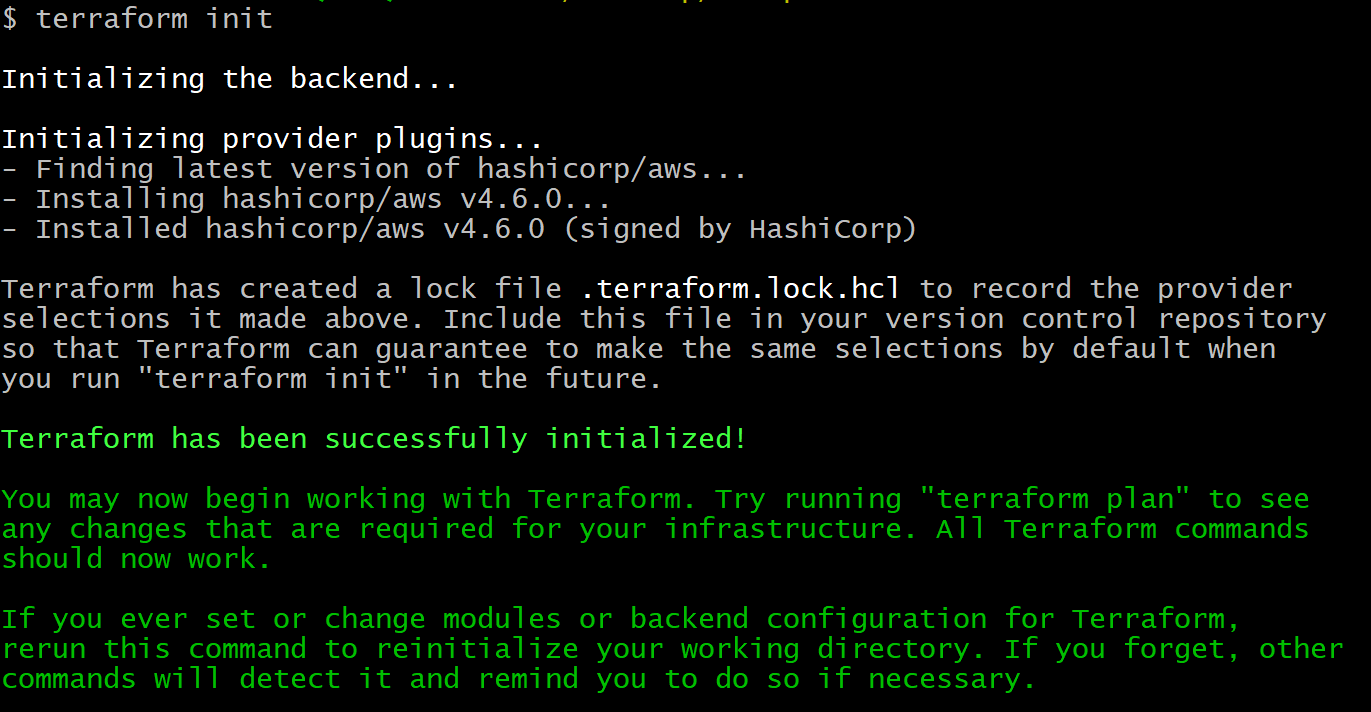
Terraform supports multiple providers.

Depending on what type of infrastructure we want to launch, we have to use appropriate providers accordingly.



initialization Phase

Upon adding a provider, it is important to run terraform init which in-turn will download plugins associated with the provider.

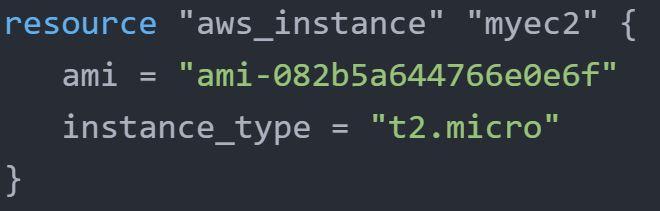


Resources

Resources are the reference to the individual services which the provider has to offer

Example:

* resource aws\_instance
* resource aws\_alb
* resource iam\_user
* resource digitalocean\_droplet



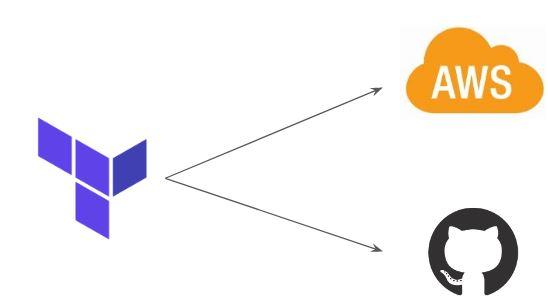
Important Update - Newer Version

From 0.13 onwards, Terraform requires explicit source information for any providers that are not HashiCorp-maintained, using a new syntax in the required\_providers nested block inside the Terraform configuration block



## **Destroying Infrastructure with Terraform (NEW)**

If you keep the infrastructure running, you will get charged for it. Hence it is important for us to also know how we can delete the infrastructure resources created via terraform.

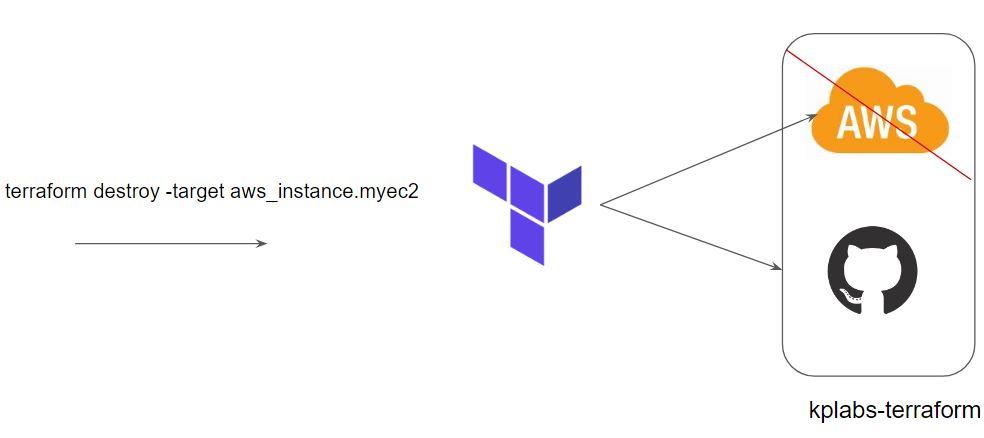


Approach 1

terraform destroy allows us to destroy all the resources that are created within the folder.

Approach 2

terraform destroy with -target flag allows us to destroy the specific resource.



provider "aws" {

region = "ap-south-1"

access\_key = ""

secret\_key = ""

}

resource "aws\_instance" "myfristec2-first" {

ami = "ami-068257025f72f470d"

instance\_type = "t2.micro"

key\_name = "XXX"

tags = {

Name = "Firstinstance created via Terraform"

}

}

resource "aws\_instance" "forDB" {

ami = "ami-068257025f72f470d"

instance\_type = "t2.micro"

key\_name = "XXX

"

tags = {

Name = "Firstinstance created via Terraform"

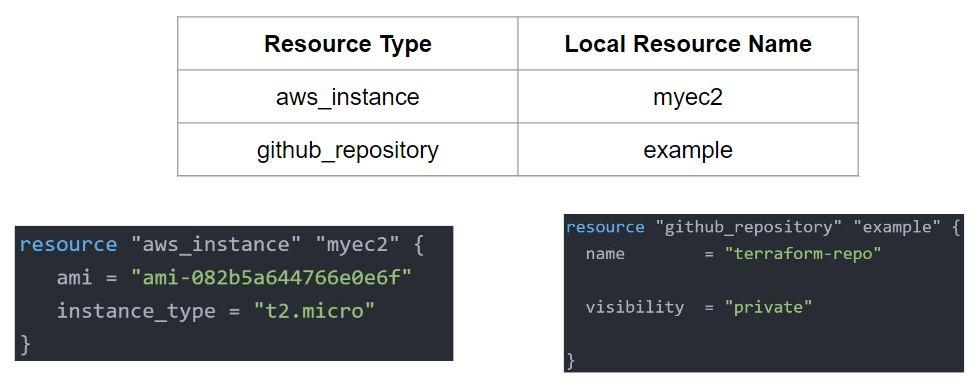
}

}

Terraform Destroy with Target

The -target option can be used to focus Terraform's attention on only a subset of resources.

Combination of: Resource Type + Local Resource Name

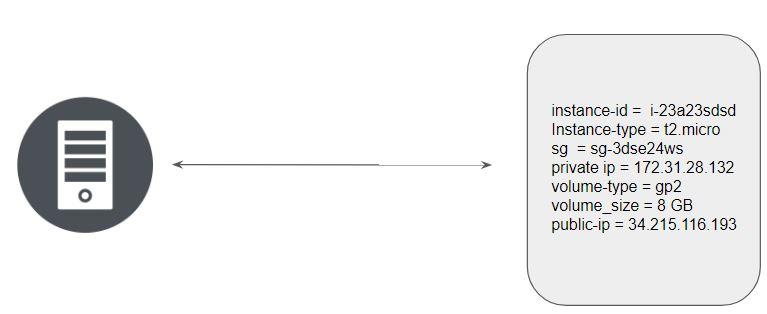


## **Terraform State File**

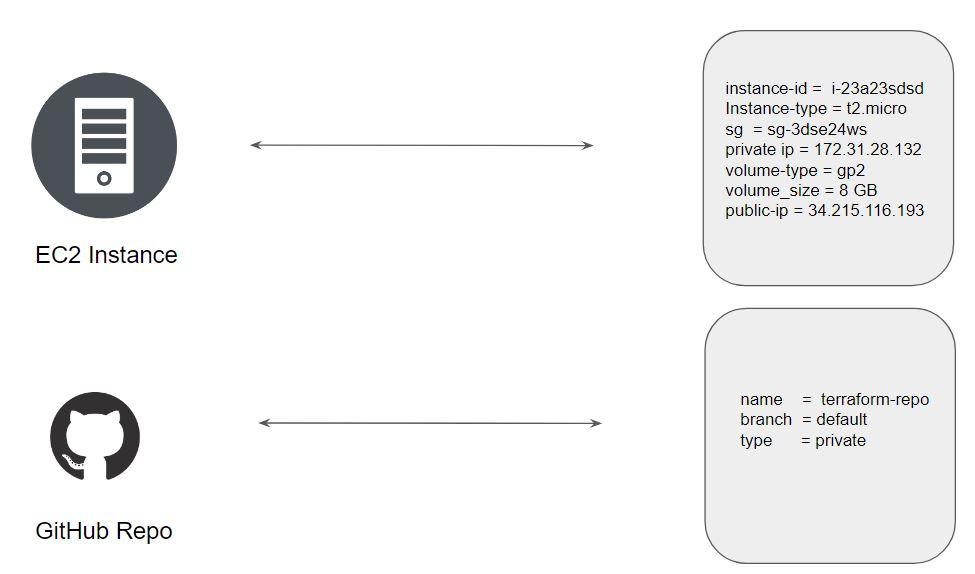
Overview of State Files:

Terraform stores the state of the infrastructure that is being created from the TF files.

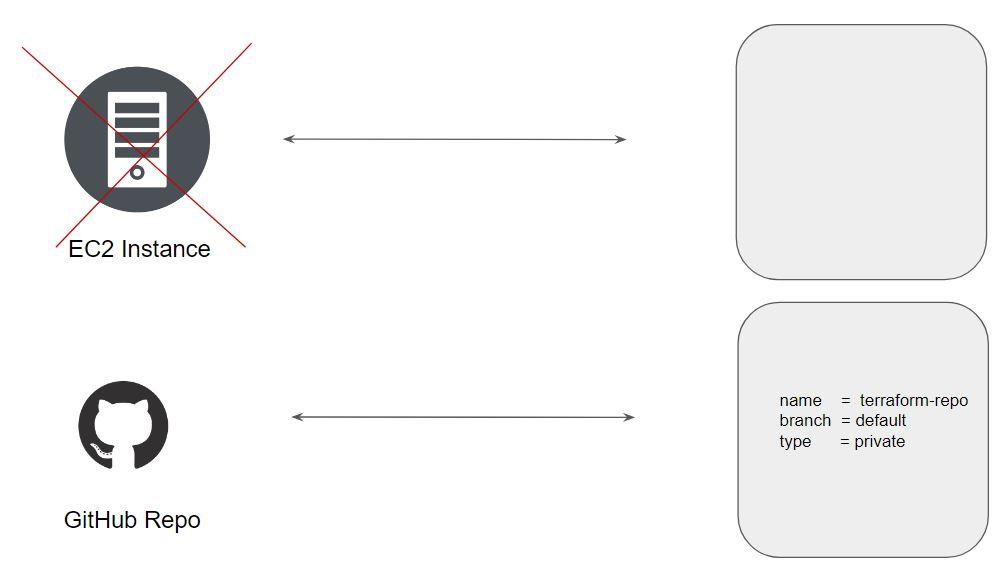
This state allows terraform to map real-world resources to your existing configuration.



Multiple resources in Terraform will have a separate block with the state file.



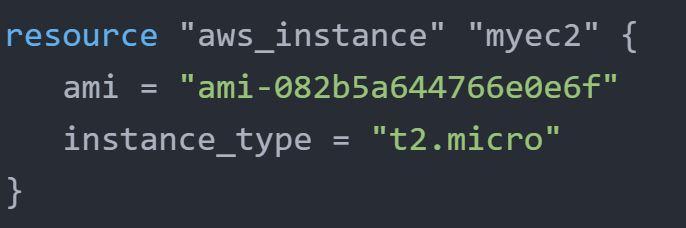
Whenever a resource is removed, its corresponding entry under the state file is also removed.



## **Desired & Current State**

Desired State

Terraform's primary function is to create, modify, and destroy infrastructure resources to match the desired state described in a Terraform configuration



Current State

The current state is the actual state of a resource that is currently deployed.



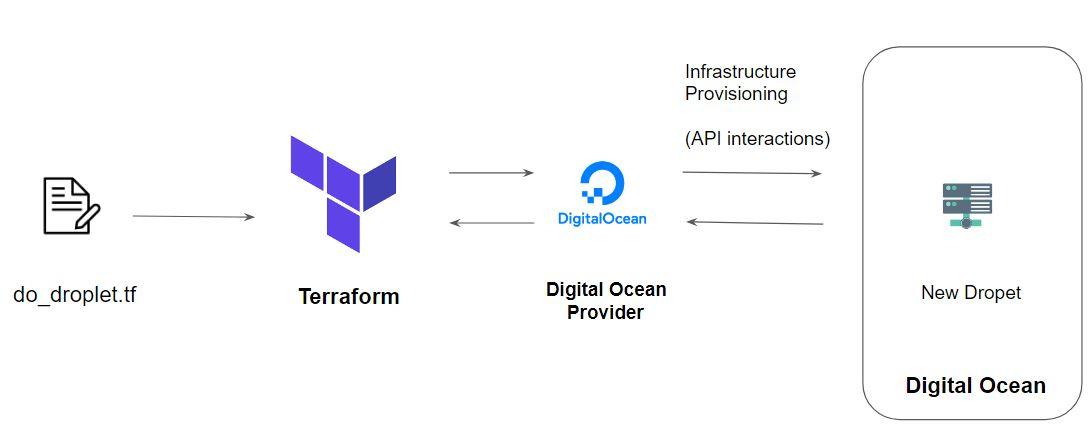
Important Pointer

Terraform tries to ensure that the deployed infrastructure is based on the desired state.

If there is a difference between the two, terraform plan presents a description of the changes necessary to achieve the desired state.

**Provider Versioning**

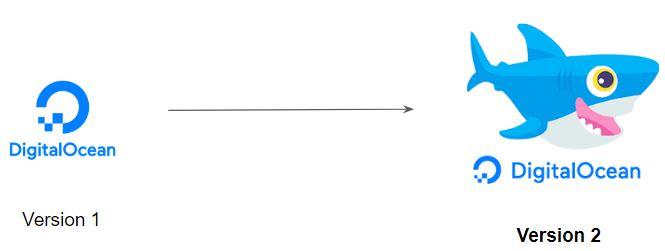
5.1 Overview of Provider Architecture



Provider Versioning

Provider plugins are released separately from Terraform itself.

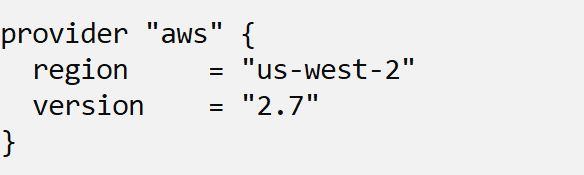
They have a different set of version numbers.



Explicitly Setting Provider Version

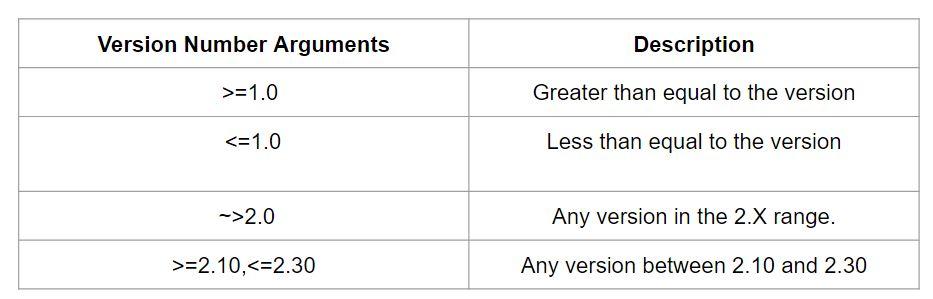
During terraform init, if the version argument is not specified, the most recent provider will be downloaded during initialization.

For production use, you should constrain the acceptable provider versions via configuration, to ensure that new versions with breaking changes will not be automatically installed



Arguments for Specifying the provider

There are multiple ways of specifying the version of a provider.



If you want to keep multiple providers in one Terraform configuration, below will be one example. In below I kept two providers AWS and GIT has used.

terraform {

required\_providers {

aws = {

source = "hashicorp/aws"

version = "~>4.0"

}

github = {

source = "integrations/github"

version = "4.22.0"

}

}

}

## **Attributes and Output Values**

Terraform has the capability to output the attribute of a resource with the output values.

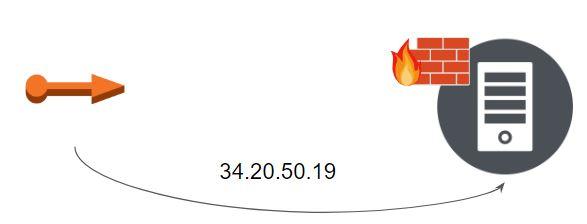
Example:

* ec2\_public\_ip = 35.161.21.197
* bucket\_identifier = terraform-test-repotech.s3.amazonaws.com

An outputed attributes can not only be used for the user reference but it can also act as an input to other resources being created via terraform

Let’s understand this with an example:

After EIP gets created, it’s IP address should automatically get whitelisted in the security group.



terraform {

required\_providers {

aws = {

source = "hashicorp/aws"

version = "4.6.0"

}

}

}

provider "aws" {

region = "us-east-1"

access\_key = "AWS\_ACCESS\_Kye"

secret\_key = "AWS\_Secret\_Key"

}

resource "aws\_eip" "lb" {

vpc = true

}

output "eip" {

value = aws\_eip.lb.public\_ip

}

resource "aws\_instance" "web" {

ami = "ami-0c02fb55956c7d316"

instance\_type = "t2.micro"

key\_name = "your\_sshkey"

tags = {

Name = "Firstinstance"

}

}

resource "aws\_eip\_association" "eip\_assoc" {

instance\_id = aws\_instance.web.id

allocation\_id = aws\_eip.lb.id

}

resource "aws\_security\_group" "allow\_all" {

name = "mycustom\_custom"

ingress {

from\_port = 443

to\_port = 443

protocol = "tcp"

cidr\_blocks = ["${aws\_eip.lb.public\_ip}/32"]

}

}

resource "aws\_network\_interface\_sg\_attachment" "sg\_attachment" {

security\_group\_id = aws\_security\_group.allow\_all.id

network\_interface\_id = aws\_instance.web.primary\_network\_interface\_id

}

## 

## 

## **Sensitive Parameter**

With the organization managing its entire infrastructure in terraform, it is likely that you will see some sensitive information embedded in the code.

When working with a field that contains information likely to be considered sensitive, it is best to set the Sensitive property on its schema to true



Setting the sensitive to “true” will prevent the field's values from showing up in CLI output and in Terraform Cloud

It will not encrypt or obscure the value in the state, however.

Example:

locals {

db\_password = {

admin = "password"

}

}

output "db\_password" {

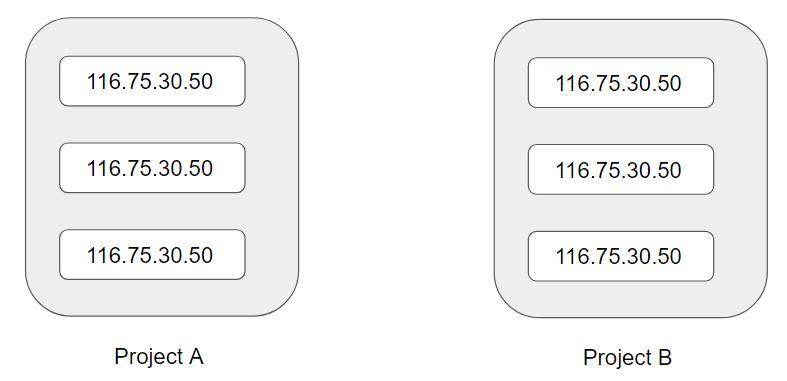
value = local.db\_password

sensitive = true

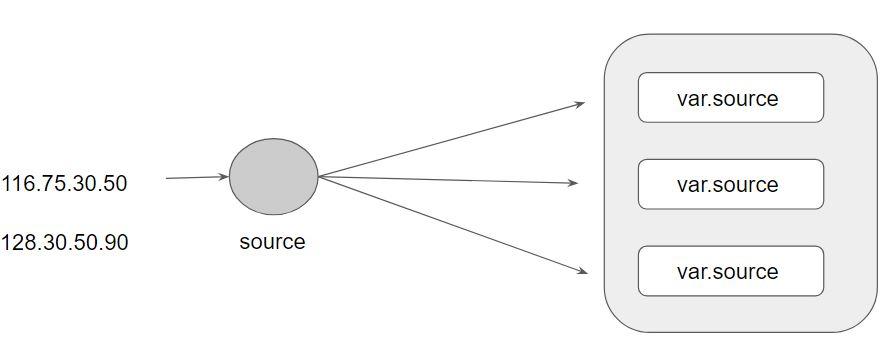
}

## **Terraform variables**

Repeated static values can create more work in the future.



Terraform Variables allows us to centrally define the values that can be used in multiple terraform configuration blocks.



## **Approach for Variable Assignment**

Variables in Terraform can be assigned values in multiple ways.

Some of these include:

* Environment variables
* Command Line Flags
* From a File
* Variable Defaults

Sample Commands for the following:

i) Environment Variables:

export TF\_VAR\_instancetype="t2.nano"

echo $TF\_VAR

ii) Command Line Flags:

terraform plan -var="instancetype=t2.small"

terraform plan -var-file="custom.tfvars"

iii) From a File (terraform.tfvars):

instancetype="t2.large"

iv) Variable Defaults:

variable "instancetype" {

default = "t2.micro"

}

## **Data Types for Variables**

Overview of Type Constraints

The type argument in a variable block allows you to restrict the type of value that will be accepted as the value for a variable

variable "image\_id" {

type = string

}

If no type constraint is set then a value of any type is accepted.

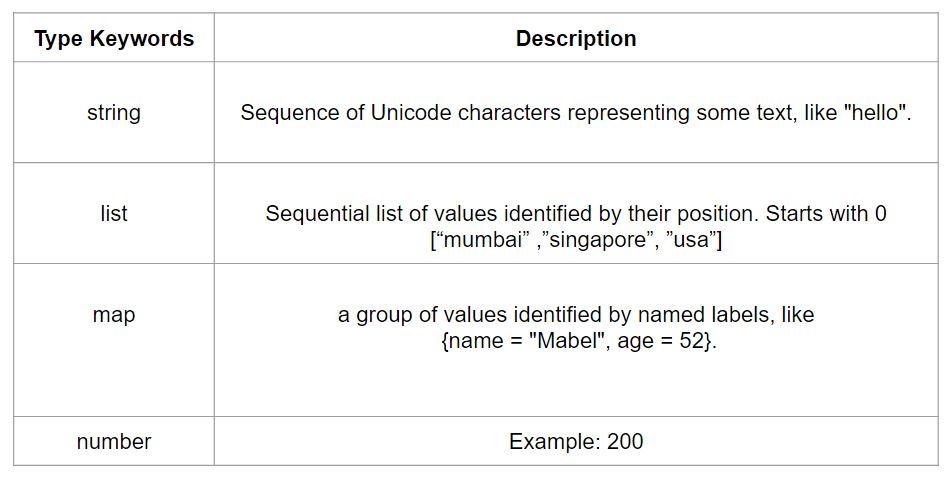
Example Use-Case 1

Every employee in Medium Corp is assigned an Identification Number.

Any resource that an employee creates should be created with the name of the identification number only.



Overview of Data Types



fetching values from map and list

variable "ec2type" {

type = list

default = ["m5.large",”m5.xlarge","t2.medium"]

}

variable "ec2types"{

type = map

default = {

us-east-1 = "t2.micro"

us-west-2 = "t2.nano"

ap-south-1 = "t2.small"

}

to call map

instance\_type = var.ec2types["us-east-1"]

Instacne\_type = lookup(ec2types, var.ec2type)

to call list

instance\_type = var.ec2type[0]

Instacne\_type = element(ec2type,0)

## 

## **Count Parameter**

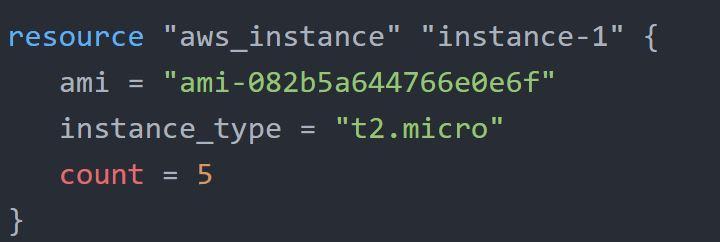
Overview of Count:

The count parameter on resources can simplify configurations and let you scale resources by simply incrementing a number.

Let’s assume, you need to create two EC2 instances. One of the common approaches is to define two separate resource blocks for aws\_instance.



With the count parameter, we can simply specify the count value and the resource can be scaled accordingly.



Count Index

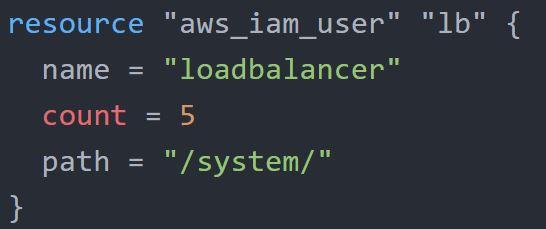
In resource blocks where the count is set, an additional count object is available in expressions, so you can modify the configuration of each instance.

This object has one attribute:

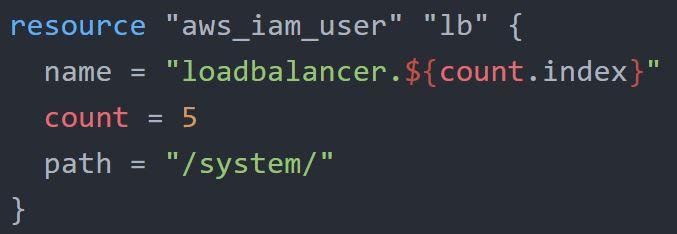
count.index — The distinct index number (starting with 0) corresponding to this instance.

Challenges with Count Parameter

With the below code, terraform will create 5 IAM users. But the problem is that all will have the same name.



count.index allows us to fetch the index of each iteration in the loop.

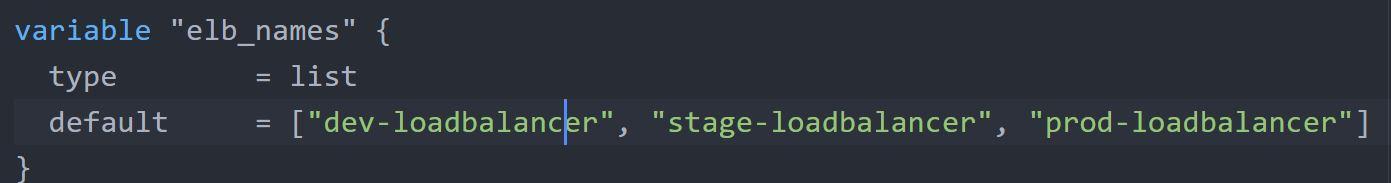


Understanding Challenge with Default Count Index

Having a username like loadbalancer0, loadbalancer1 might not always be suitable.

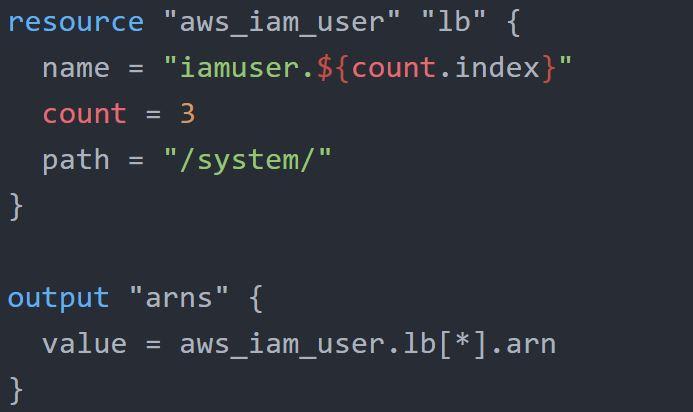
Better names like dev-loadbalancer, stage-loadbalancer, prod-loadbalancer is better.

count.index can help in such a scenario as well.



## **Splat Expression**

Splat Expression allows us to get a list of all the attributes.



Example:

provider "aws" {

region = "us-west-2"

access\_key = "YOUR-ACCESS-KEY"

secret\_key = "YOUR-SECRET-KEY"

}

resource "aws\_iam\_user" "lb" {

name = "iamuser.${count.index}"

count = 3

path = "/system/"

}

output "arns" {

value = aws\_iam\_user.lb[\*].arn

}

## **Conditional Expression**

A conditional expression uses the value of a bool expression to select one of two values.

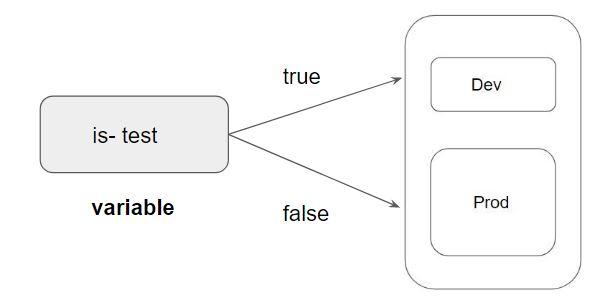
Syntax of Conditional expression:

condition ? true\_val : false\_val

If the condition is true then the result is true\_val. If the condition is false then the result is false\_val.

Let’s assume that there are two resource blocks as part of terraform configuration.

Depending on the variable value, one of the resource blocks will run.



terraform {

required\_providers {

aws = {

source = "hashicorp/aws"

version = "4.6.0"

}

}

}

provider "aws" {

region = "us-west-2"

access\_key = "AWS\_Access\_Key"

secret\_key = "AWS\_Secret\_key"

}

variable "istest" {}

resource "aws\_vpc" "my\_vpc\_test" {

cidr\_block = "172.16.0.0/16"

count = var.istest == true ? 1 : 0

tags = {

Name = "Dev-VPC"

}

}

resource "aws\_vpc" "my\_vpc\_prod" {

cidr\_block = "172.17.0.0/16"

count = var.istest == false ? 1 : 0

tags = {

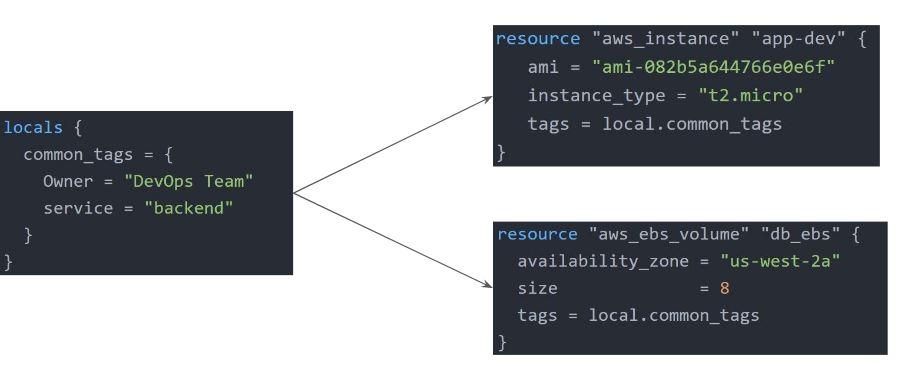
Name = "Prod-VPC"

}

}

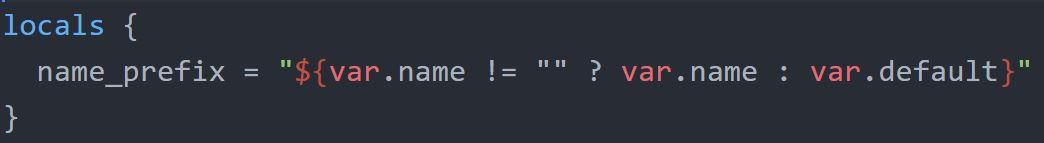
## **Local Values**

A local value assigns a name to an expression, allowing it to be used multiple times within a module without repeating it.



Local Values Support for Expression

Local Values can be used for multiple different use-cases like having a conditional expression.



Important Pointers for Local Values:

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration.

If overused they can also make a configuration hard to read by future maintainers by hiding the actual values used

Use local values only in moderation, in situations where a single value or result is used in many places and that value is likely to be changed in the future.

Example:

locals {

common\_tags = {

Owner = "DevOps Team"

service = "backend"

}

}

resource "aws\_instance" "app-dev" {

ami = "ami-082b5a644766e0e6f"

instance\_type = "t2.micro"

tags = local.common\_tags

}

resource "aws\_instance" "db-dev" {

ami = "ami-082b5a644766e0e6f"

instance\_type = "t2.small"

tags = local.common\_tags

}

resource "aws\_ebs\_volume" "db\_ebs" {

availability\_zone = "us-west-2a"

size = 8

tags = local.common\_tags

}

output "instaceid" {

value = aws\_instance.db-dev.id

}

## **Terraform Functions**

The Terraform language includes a number of built-in functions that you can use to transform and combine values.

The general syntax for function calls is a function name followed by comma-separated arguments in parentheses:

function (argument1, argument2)

Example:

> max(5, 12, 9)

12

The Terraform language does not support user-defined functions, and so only the functions built into the language are available for use

* Numeric
* String
* Collection
* Encoding
* Filesystem
* Date and Time
* Hash and Crypto
* IP Network
* Type Conversion

example:

locals {

time = formatdate("DD MMM YYYY hh:mm ZZZ", timestamp())

}

variable "region" {

default = "us-west-2"

}

variable "tags" {

type = list(any)

default = ["firstec2", "secondec2"]

}

variable "ami" {

type = map(any)

default = {

"us-east-1" = "ami-0dc2d3e4c0f9ebd18"

"us-west-2" = "ami-0dc8f589abe99f538"

"ap-south-1" = "ami-04f6f742e1d9012e3"

}

}

resource "aws\_key\_pair" "loginkey" {

key\_name = "myvirkey"

public\_key = file("${path.module}/id\_rsa.pub")

}

resource "aws\_instance" "app-dev" {

ami = lookup(var.ami, var.region)

instance\_type = "t2.micro"

key\_name = aws\_key\_pair.loginkey.key\_name

count = 2

tags = {

Name = element(var.tags, count.index)

}

}

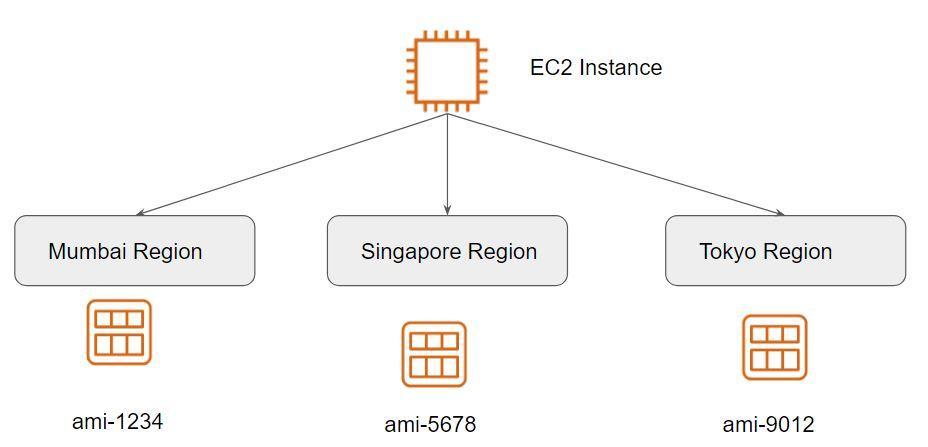
output "timestamp" {

value = local.time

}

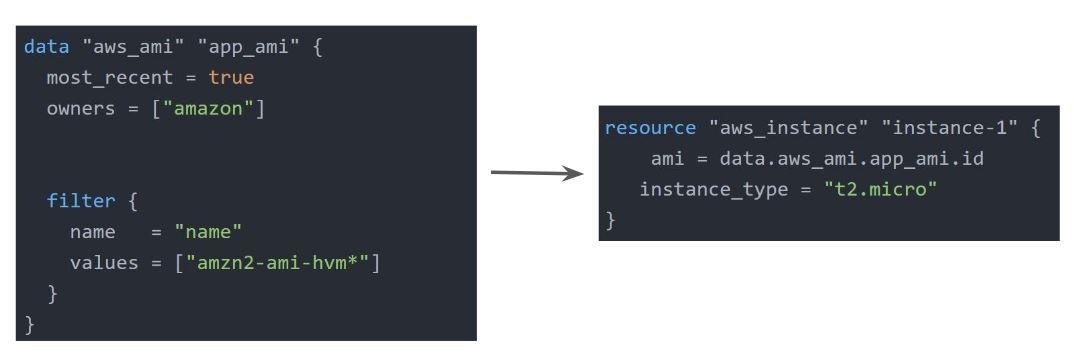
## **Data Sources**

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration.



A data source is defined under the data block.

It reads from a specific data source (aws\_ami) and exports results under “app\_ami”



Example

data "aws\_ami" "app\_ami" {

owners = ["amazon"]

most\_recent = true

filter {

name = "name"

values = ["amzn2-ami-kernel-5.10\*\*"]

}

}

resource "aws\_instance" "instance-1" {

ami = data.aws\_ami.app\_ami.id

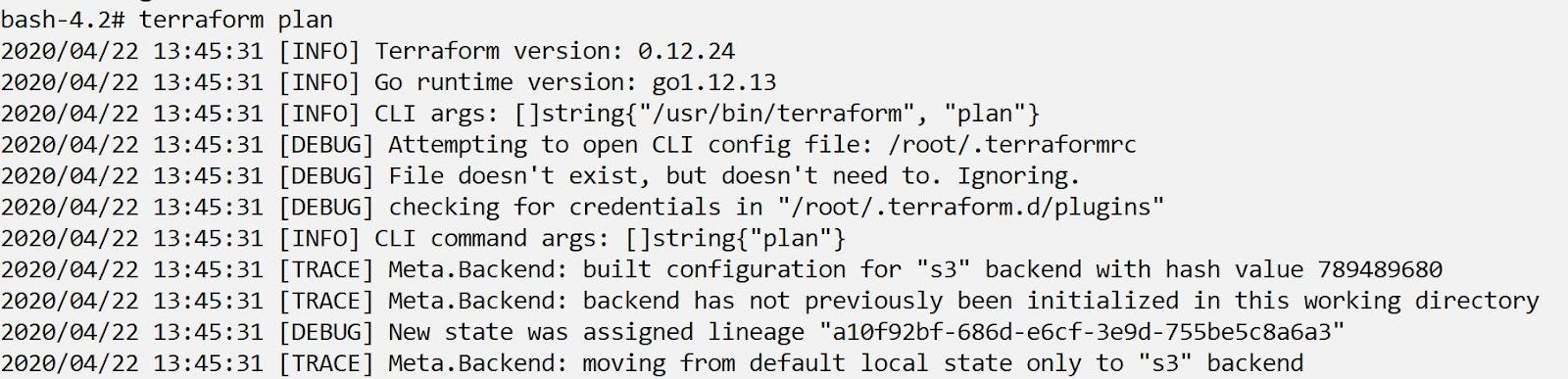
instance\_type = "t2.micro"

}

## **Debugging in Terraform**

Terraform has detailed logs that can be enabled by setting the TF\_LOG environment variable to any value.

You can set TF\_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs



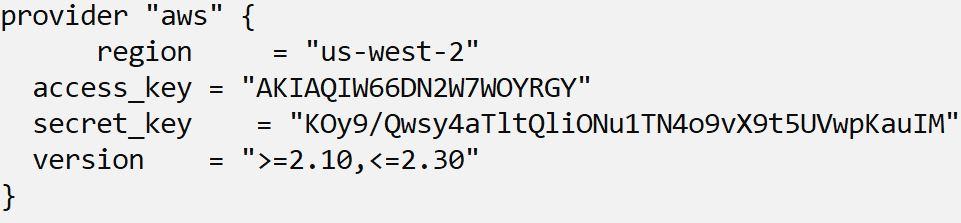
Important Pointers for Debugging:

TRACE is the most verbose and it is the default if TF\_LOG is set to something other than a log level name.

To persist logged output you can set TF\_LOG\_PATH in order to force the log to always be appended to a specific file when logging is enabled.

## **Terraform Format**

Anyone who is into programming knows the importance of formatting the code for readability.



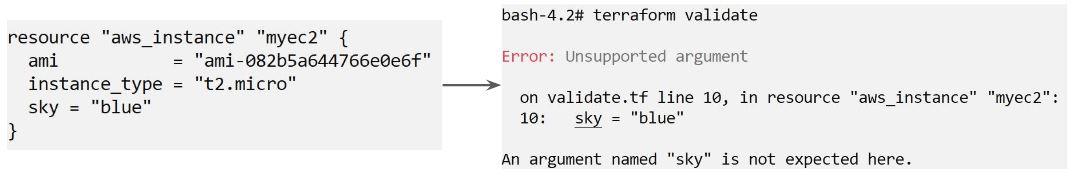
The terraform fmt command is used to rewrite Terraform configuration files to take care of the overall formatting



## **Terraform Validate**

Terraform Validate primarily checks whether a configuration is syntactically valid.

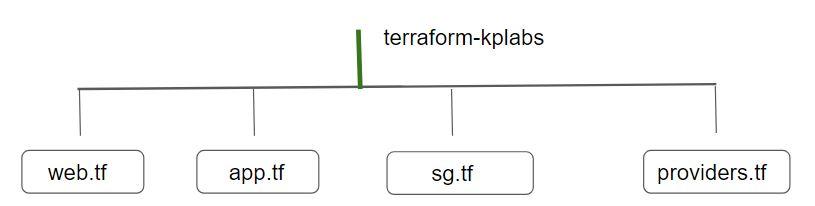
It can check various aspects including unsupported arguments, undeclared variables, and others.



## **Load Order & Semantics**

Terraform generally loads all the configuration files within the directory specified in alphabetical order.

The files loaded must end in either .tf or .tf.json to specify the format that is in use.



## 

## **Dynamic Blocks**

Understanding the Challenge:

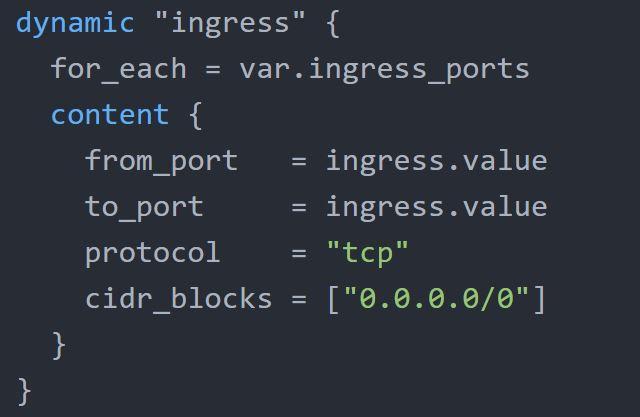
In many of the use-cases, there are repeatable nested blocks that need to be defined.

This can lead to a long code and it can be difficult to manage in a long time.



Overview of Dynamic Blocks

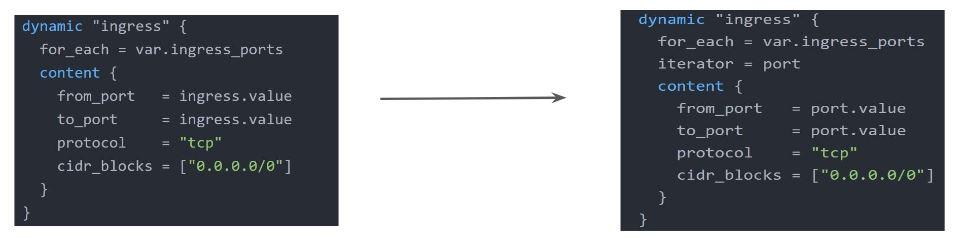
Dynamic Block allows us to dynamically construct repeatable nested blocks which is supported inside resource, data, provider, and provisioner blocks:



Overview of Iterators

The iterator argument (optional) sets the name of a temporary variable that represents the current element of the complex value

If omitted, the name of the variable defaults to the label of the dynamic block ("ingress" in the example above).



Example:

Before.tf

resource "aws\_security\_group" "demo\_sg" {

name = "sample-sg"

ingress {

from\_port = 8200

to\_port = 8200

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

from\_port = 8201

to\_port = 8201

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

from\_port = 8300

to\_port = 8300

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

from\_port = 9200

to\_port = 9200

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

from\_port = 9500

to\_port = 9500

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

Dynamic.tf

variable "sg\_ports" {

type = list(number)

description = "list of ingress ports"

default = [8200, 8201,8300, 9200, 9500]

}

resource "aws\_security\_group" "dynamicsg" {

name = "dynamic-sg"

description = "Ingress for Vault"

dynamic "ingress" {

for\_each = var.sg\_ports

iterator = port

content {

from\_port = port.value

to\_port = port.value

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

dynamic "egress" {

for\_each = var.sg\_ports

content {

from\_port = egress.value

to\_port = egress.value

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

}

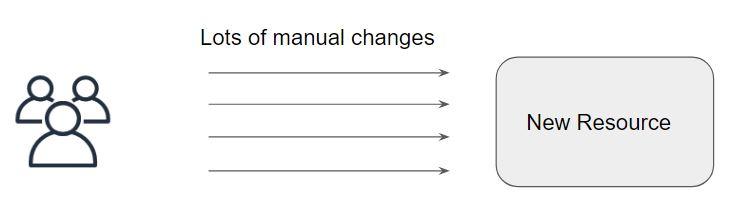
## **Terraform Taint**

Understanding the Challenge:

You have created a new resource via Terraform.

Users have made a lot of manual changes (both infrastructure and inside the server)

Two ways to deal with this: Import The Changes to Terraform / Delete & Recreate the resource



Overview of Terraform Taint

The terraform taint command manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply.



Important Pointers for Terraform Taint

This command will not modify infrastructure but does modify the state file in order to mark a resource as tainted.

Once a resource is marked as tainted, the next plan will show that the resource will be destroyed and recreated and the next apply will implement this change.

Note that tainting a resource for recreation may affect resources that depend on the newly tainted resource.

Example:

provider "aws" {

region = "us-west-2"

access\_key = "YOUR-ACCESS-KEY"

secret\_key = "YOUR-SECRET-KEY"

}

resource "aws\_instance" "myec2" {

ami = "ami-082b5a644766e0e6f"

instance\_type = "t2.micro"

}

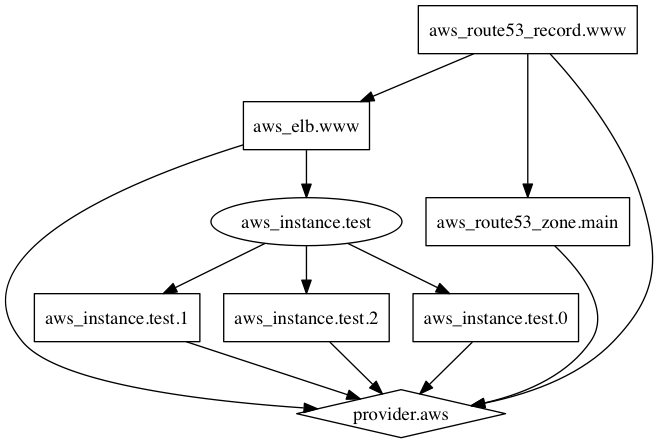
terraform taint aws\_instance.myec2

From the above myec2 instance will be tainted and will recreate next plan.

## **Terraform Graph**

The terraform graph command is used to generate a visual representation of either a configuration or execution plan

The output of terraform graph is in the DOT format, which can easily be converted to an image.



provider "aws" {

region = "us-west-2"

access\_key = "YOUR-ACCESS-KEY"

secret\_key = "YOUR-SECRET-KEY"

}

resource "aws\_instance" "myec2" {

ami = "ami-082b5a644766e0e6f"

instance\_type = "t2.micro"

}

resource "aws\_eip" "lb" {

instance = aws\_instance.myec2.id

vpc = true

}

resource "aws\_security\_group" "allow\_tls" {

name = "allow\_tls"

ingress {

description = "TLS from VPC"

from\_port = 443

to\_port = 443

protocol = "tcp"

cidr\_blocks = ["${aws\_eip.lb.private\_ip}/32"]

}

}

### **Commands Used:**

terraform graph > graph.dot

yum install graphviz

cat graph.dot | dot -Tsvg > graph.svg

For online:

[https://dreampuf.github.io/GraphvizOnline/#digraph%20%7B%0D%0A%09compound%20%3D%20%22true%22%0D%0A%09newrank%20%3D%20%22true%22%0D%0A%09subgraph%20%22root%22%20%7B%0D%0A%09%09%22%5Broot%5D%20aws\_eip.lb%20(expand)%22%20%5Blabel%20%3D%20%22aws\_eip.lb%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20aws\_eip\_association.eip\_assoc%20(expand)%22%20%5Blabel%20%3D%20%22aws\_eip\_association.eip\_assoc%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20aws\_instance.web%20(expand)%22%20%5Blabel%20%3D%20%22aws\_instance.web%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20aws\_internet\_gateway.gw%20(expand)%22%20%5Blabel%20%3D%20%22aws\_internet\_gateway.gw%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20aws\_subnet.main%20(expand)%22%20%5Blabel%20%3D%20%22aws\_subnet.main%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20aws\_vpc.myvpc%20(expand)%22%20%5Blabel%20%3D%20%22aws\_vpc.myvpc%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20provider%5B%5C%22registry.terraform.io%2Fhashicorp%2Faws%5C%22%5D%22%20%5Blabel%20%3D%20%22provider%5B%5C%22registry.terraform.io%2Fhashicorp%2Faws%5C%22%5D%22%2C%20shape%20%3D%20%22diamond%22%5D%0D%0A%09%09%22%5Broot%5D%20aws\_eip.lb%20(expand)%22%20-%3E%20%22%5Broot%5D%20provider%5B%5C%22registry.terraform.io%2Fhashicorp%2Faws%5C%22%5D%22%0D%0A%09%09%22%5Broot%5D%20aws\_eip\_association.eip\_assoc%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws\_eip.lb%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20aws\_eip\_association.eip\_assoc%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws\_instance.web%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20aws\_instance.web%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws\_subnet.main%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20aws\_internet\_gateway.gw%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws\_vpc.myvpc%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20aws\_subnet.main%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws\_vpc.myvpc%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20aws\_vpc.myvpc%20(expand)%22%20-%3E%20%22%5Broot%5D%20provider%5B%5C%22registry.terraform.io%2Fhashicorp%2Faws%5C%22%5D%22%0D%0A%09%09%22%5Broot%5D%20output.myec2privateip%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws\_instance.web%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20output.myec2publicIP%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws\_eip.lb%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20output.myvpccidr%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws\_vpc.myvpc%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20provider%5B%5C%22registry.terraform.io%2Fhashicorp%2Faws%5C%22%5D%20(close)%22%20-%3E%20%22%5Broot%5D%20aws\_eip\_association.eip\_assoc%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20provider%5B%5C%22registry.terraform.io%2Fhashicorp%2Faws%5C%22%5D%20(close)%22%20-%3E%20%22%5Broot%5D%20aws\_internet\_gateway.gw%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20root%22%20-%3E%20%22%5Broot%5D%20output.myec2privateip%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20root%22%20-%3E%20%22%5Broot%5D%20output.myec2publicIP%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20root%22%20-%3E%20%22%5Broot%5D%20output.myvpccidr%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20root%22%20-%3E%20%22%5Broot%5D%20provider%5B%5C%22registry.terraform.io%2Fhashicorp%2Faws%5C%22%5D%20(close)%22%0D%0A%09%7D%0D%0A%7D%0D%0A%0D%0A](https://dreampuf.github.io/GraphvizOnline/#digraph%20%7B%0D%0A%09compound%20%3D%20%22true%22%0D%0A%09newrank%20%3D%20%22true%22%0D%0A%09subgraph%20%22root%22%20%7B%0D%0A%09%09%22%5Broot%5D%20aws_eip.lb%20(expand)%22%20%5Blabel%20%3D%20%22aws_eip.lb%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20aws_eip_association.eip_assoc%20(expand)%22%20%5Blabel%20%3D%20%22aws_eip_association.eip_assoc%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20aws_instance.web%20(expand)%22%20%5Blabel%20%3D%20%22aws_instance.web%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20aws_internet_gateway.gw%20(expand)%22%20%5Blabel%20%3D%20%22aws_internet_gateway.gw%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20aws_subnet.main%20(expand)%22%20%5Blabel%20%3D%20%22aws_subnet.main%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20aws_vpc.myvpc%20(expand)%22%20%5Blabel%20%3D%20%22aws_vpc.myvpc%22%2C%20shape%20%3D%20%22box%22%5D%0D%0A%09%09%22%5Broot%5D%20provider%5B%5C%22registry.terraform.io%2Fhashicorp%2Faws%5C%22%5D%22%20%5Blabel%20%3D%20%22provider%5B%5C%22registry.terraform.io%2Fhashicorp%2Faws%5C%22%5D%22%2C%20shape%20%3D%20%22diamond%22%5D%0D%0A%09%09%22%5Broot%5D%20aws_eip.lb%20(expand)%22%20-%3E%20%22%5Broot%5D%20provider%5B%5C%22registry.terraform.io%2Fhashicorp%2Faws%5C%22%5D%22%0D%0A%09%09%22%5Broot%5D%20aws_eip_association.eip_assoc%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws_eip.lb%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20aws_eip_association.eip_assoc%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws_instance.web%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20aws_instance.web%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws_subnet.main%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20aws_internet_gateway.gw%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws_vpc.myvpc%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20aws_subnet.main%20(expand)%22%20-%3E%20%22%5Broot%5D%20aws_vpc.myvpc%20(expand)%22%0D%0A%09%09%22%5Broot%5D%20aws_vpc.myvpc%20(expand)%22%20-%3E%20%22%5Broot%5D%20provider%5B%5C%22registry.terraform.io%2Fhashicorp%2Faws%5C%22%5D%22%0D%0A%09%09%22%5Broot%5D%20output.myec2priva)

## **Saving Terraform Plan to a File**

The generated terraform plan can be saved to a specific path.

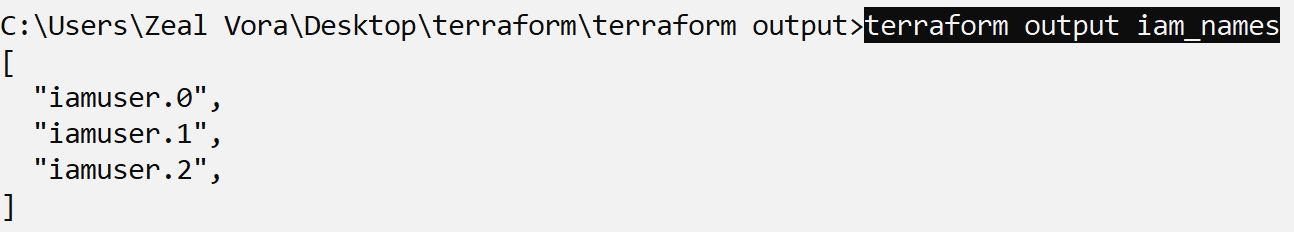
This plan can then be used with terraform apply to be certain that only the changes shown in this plan are applied.

Example:

terraform plan -out=path

## **Terraform Output**

The terraform output command is used to extract the value of an output variable from the state file.



## **Understanding Provisioners in Terraform**

Understanding the Challenge

Till now we have been working only on the creation and destruction of infrastructure scenarios.

Let’s take an example:

We created a web-server EC2 instance with Terraform.

Problem: It is only an EC2 instance, it does not have any software installed.

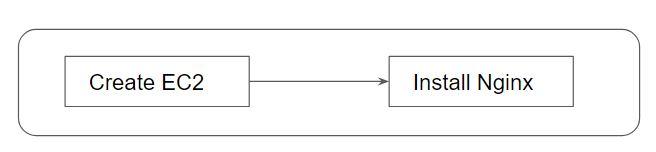
What if we want a complete end to end solution?

Introducing Terraform Provisioners

Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction.

Let’s take an example:

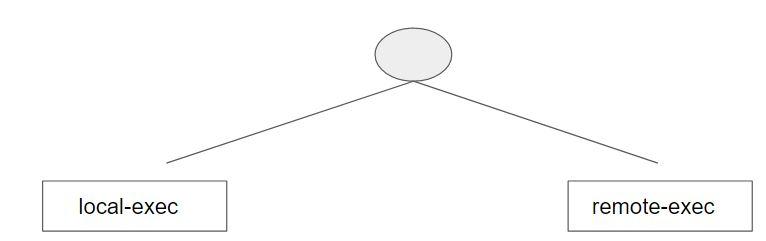
On creation of Web-Server, execute a script which installs Nginx web-server.



## **Types of Provisioners**

Terraform has the capability to turn provisioners both at the time of resource creation as well as destruction.

There are two main types of provisioners:



Local Exec Provisioners

local-exec provisioners allow us to invoke a local executable after the resource is created.

One of the most used approaches of local-exec is to run ansible-playbooks on the created server after the resource is created.

Let’s take an example:

provisioner "local-exec" {

command = "echo ${aws\_instance.web.private\_ip} >> private\_ips.txt"

}

Remote Exec Provisioners

Remote-exec provisioners allow invoking scripts directly on the remote server.

Let’s take an example:

resource "aws\_instance" "web" {

# …

provisioner "remote-exec" {

…………………………...

}

}

Example:

data "aws\_ami" "app\_ami" {

owners = ["amazon"]

most\_recent = true

filter {

name = "name"

values = ["amzn2-ami-kernel-5.10\*\*"]

}

}

resource "aws\_instance" "instance-1" {

ami = data.aws\_ami.app\_ami.id

instance\_type = "t2.micro"

key\_name = "alankey"

provisioner "remote-exec" {

inline = [

"sudo amazon-linux-extras install -y nginx1",

"sudo systemctl start nginx"

]

}

connection {

type = "ssh"

user = "ec2-user"

private\_key = file("alankey.pem")

host = self.public\_ip

}

provisioner "local-exec" {

command = "echo ${aws\_instance.instance-1.private\_ip} >> private\_ips.txt"

}

}

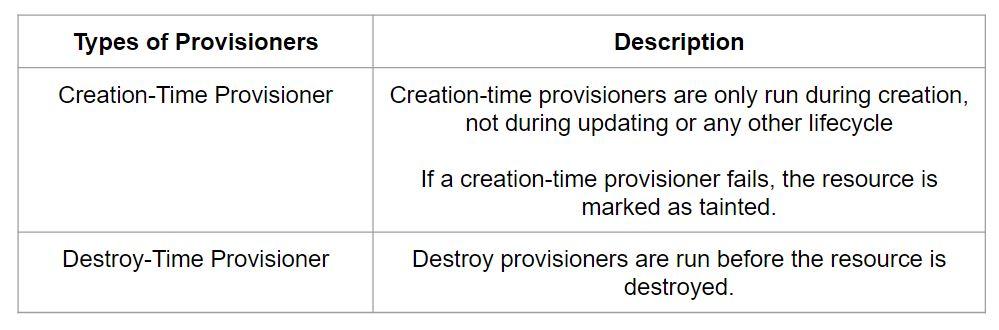
output "mypublicIP" {

value = aws\_instance.instance-1.public\_ip

}

## **Provisioner Types**

There are two primary types of provisioners:



If when = destroy is specified, the provisioner will run when the resource it is defined within is destroyed.



Example:

provider "aws" {

region = "ap-southeast-1"

access\_key = "YOUR-KEY"

secret\_key = "YOUR-KEY"

}

resource "aws\_security\_group" "allow\_ssh" {

name = "allow\_ssh"

description = "Allow SSH inbound traffic"

ingress {

description = "SSH into VPC"

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

egress {

description = "Outbound Allowed"

from\_port = 0

to\_port = 65535

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

resource "aws\_instance" "myec2" {

ami = "ami-0b1e534a4ff9019e0"

instance\_type = "t2.micro"

key\_name = "ec2-key"

vpc\_security\_group\_ids = [aws\_security\_group.allow\_ssh.id]

provisioner "remote-exec" {

inline = [

"sudo yum -y install nano"

]

}

provisioner "remote-exec" {

when = destroy

inline = [

"sudo yum -y remove nano"

]

}

connection {

type = "ssh"

user = "ec2-user"

private\_key = file("./ec2-key.pem")

host = self.public\_ip

}

}

=======

terraform {

required\_providers {

aws = {

source = "hashicorp/aws"

version = "4.6.0"

}

}

}

provider "aws" {

region = "ap-south-1"

access\_key = "ACCESSKEY"

secret\_key = "Secretkey"

}

resource "aws\_instance" "myec2" {

ami = "ami-08df646e18b182346"

instance\_type = "t2.micro"

key\_name = "ravi\_mumbai"

provisioner "remote-exec" {

inline = [

"sudo amazon-linux-extras install -y nginx1",

"sudo systemctl start nginx"

]

}

provisioner "remote-exec" {

when = destroy

inline = [

"sudo yum remove nginx -y",

]

}

connection {

type = "ssh"

user = "ec2-user"

private\_key = file("ravi\_mumbai.pem")

host = self.public\_ip

}

}

output "publicip" {

value = aws\_instance.myec2.public\_ip

}

**Failure Behaviour**

By default, provisioners that fail will also cause the terraform apply itself to fail.

The on\_failure setting can be used to change this. The allowed values are:





Example:

provider "aws" {

region = "ap-southeast-1"

access\_key = "YOUR-KEY"

secret\_key = "YOUR-KEY"

}

resource "aws\_security\_group" "allow\_ssh" {

name = "allow\_ssh"

description = "Allow SSH inbound traffic"

ingress {

description = "SSH into VPC"

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

resource "aws\_instance" "myec2" {

ami = "ami-0b1e534a4ff9019e0"

instance\_type = "t2.micro"

key\_name = "ec2-key"

vpc\_security\_group\_ids = [aws\_security\_group.allow\_ssh.id]

provisioner "remote-exec" {

on\_failure = continue

inline = [

"sudo yum -y install nano"

]

}

connection {

type = "ssh"

user = "ec2-user"

private\_key = file("./ec2-key.pem")

host = self.public\_ip

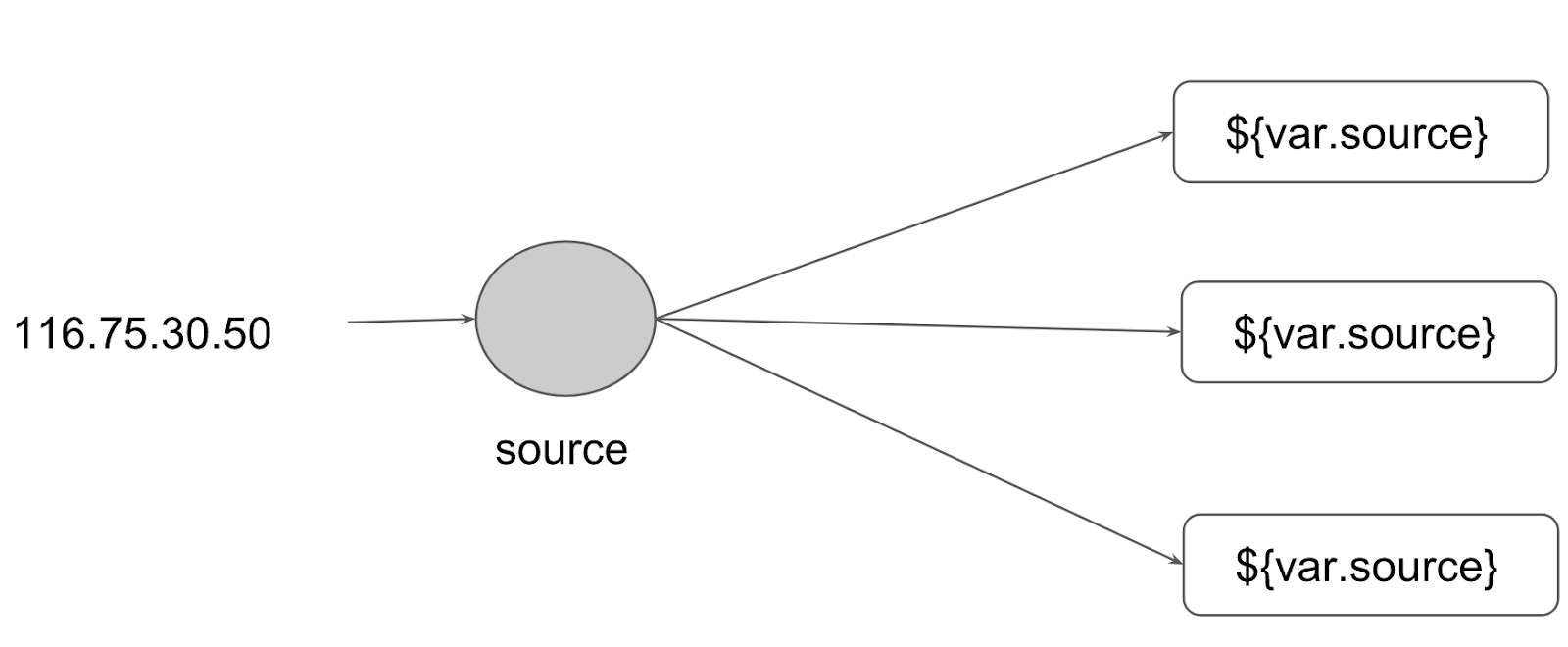
}

}

## **Understanding the DRY principle**

In software engineering, don't repeat yourself (DRY) is a principle of software development aimed at reducing repetition of software patterns.

In the earlier lecture, we were making static content into variables so that there can be a single source of information.



Generic Scenario:

We do repeat multiple times various terraform resources for multiple projects.

Sample EC2 Resource

resource "aws\_instance" "myweb" {

ami = "ami-bf5540df"

instance\_type = "t2.micro"

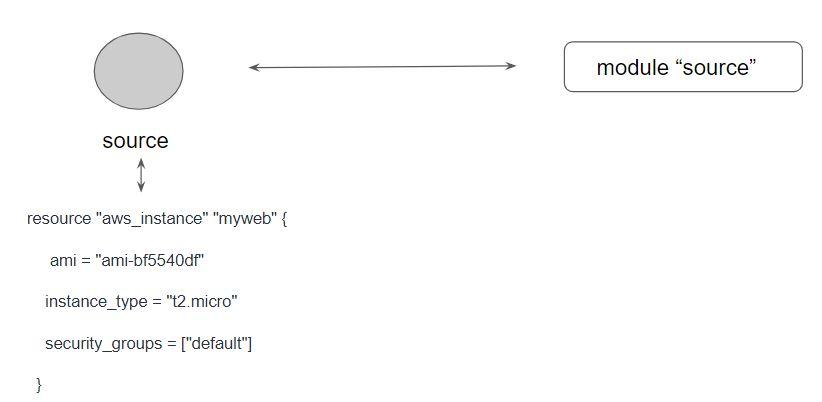
security\_groups = ["default"]

}

Instead of repeating a resource block multiple times, we can make use of a centralized structure.

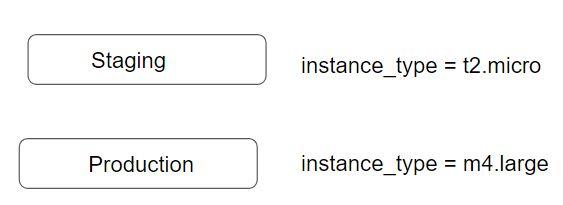
Centralized Structure:

We can centralize the terraform resources and can call out from TF files whenever required.



## **Challenges with Terraform Modules**

One common need for infrastructure management is to build environments like staging, production with a similar setup but keeping environment variables different.



When we use modules directly, the resources will be a replica of code in the module.



## 

## Exampe:

In module location keep below ec2.tf file.

resource "aws\_instance" "myec2" {

ami = "ami-04db49c0fb2215364" # us-west-2

instance\_type = var.instance\_type

}

Under same directory vars.tf

variable "instance\_type"{

default = "t2.micro"

}

To use module above created:

Under different directory u can keep myec2.tf and put below code with provider module.

module "myec2"{

source = "../../Ec2\_Module"

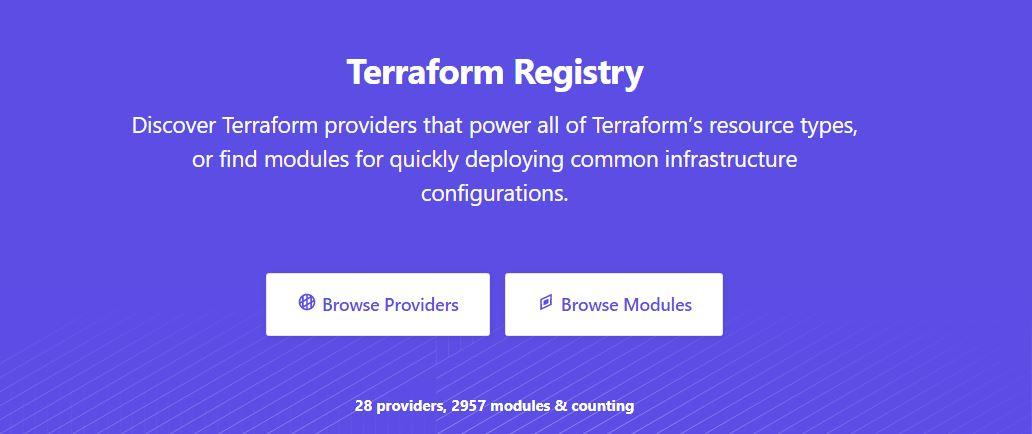
instance\_type = "t2.large"

}

## **Terraform Registry**

The Terraform Registry is a repository of modules written by the Terraform community.

The registry can help you get started with Terraform more quickly



Module Location

If we intend to use a module, we need to define the path where the module files are present.

The module files can be stored in multiple locations, some of these include:

* Local Path
* GitHub
* Terraform Registry
* S3 Bucket
* HTTP URLs

Verified Modules in Terraform Registry

Within Terraform Registry, you can find verified modules that are maintained by various third-party vendors.

These modules are available for various resources like AWS VPC, RDS, ELB, and others



Verified modules are reviewed by HashiCorp and actively maintained by contributors to stay up-to-date and compatible with both Terraform and their respective providers.

The blue verification badge appears next to modules that are verified.

Module verification is currently a manual process restricted to a small group of trusted HashiCorp partners.

Using Registry Modules in Terraform

To use Terraform Registry module within the code, we can make use of the source argument that contains the module path.

Below code references to the EC2 Instance module within terraform registry.

module "ec2-instance" {

source = "terraform-aws-modules/ec2-instance/aws"

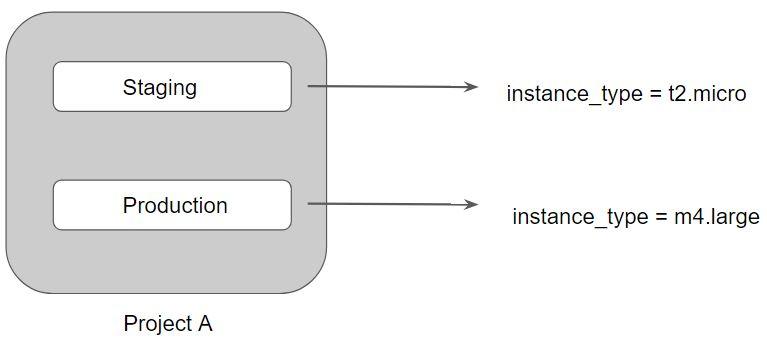
version = "2.13.0"

# insert the 10 required variables here

}

## **Terraform Workspace**

Terraform allows us to have multiple workspaces, with each of the workspaces we can have a different set of environment variables associated



Terraform starts with a single workspace named "default".

This workspace is special both because it is the default and also because it cannot ever be deleted.

If you've never explicitly used workspaces, then you've only ever worked on the "default" workspace.

Workspaces are managed with the terraform workspace set of commands.

To create a new workspace and switch to it, you can use terraform workspace new; to switch workspaces you can use terraform workspace select; etc.

Workspace commands:

terraform workspace -h

terraform workspace show

terraform workspace new dev

terraform workspace new prd

terraform workspace list

terraform workspace select dev

Terraform configuration to create multiple environments using work spaces.

provider "aws" {

region = "us-west-2"

access\_key = "YOUR-ACCESS-KEY"

secret\_key = "YOUR-SECRET-KEY"

}

resource "aws\_instance" "myec2" {

ami = "ami-082b5a644766e0e6f"

instance\_type = lookup(var.instance\_type,terraform.workspace)

}

variable "instance\_type" {

type = "map"

default = {

default = "t2.nano"

dev = "t2.micro"

prd = "t2.large"

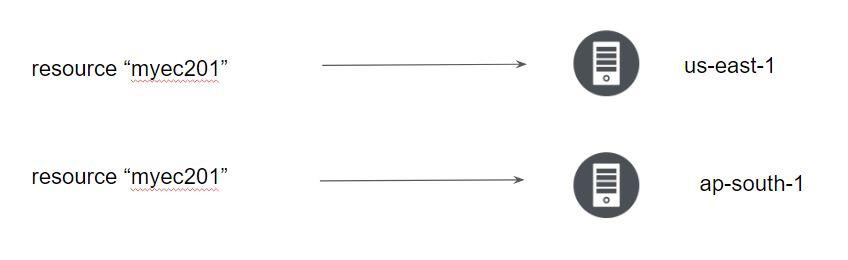
}

}

## **Provider Configuration**

Till now, we have been hardcoding the aws-region parameter within the providers.tf

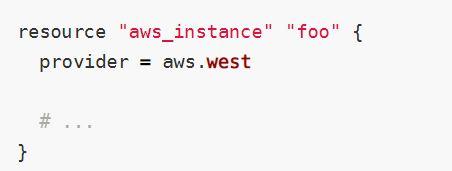
This means that resources would be created in the region specified in the providers.tf file.



By default, resources use a default provider configuration inferred from the first word of the resource type name.

For example, a resource of type aws\_instance uses the default (un-aliased) aws provider configuration unless otherwise stated.

To select an aliased provider for a resource or data source, set its provider meta-argument to a <PROVIDER NAME>.<ALIAS> reference:



### **eip.tf**

resource "aws\_eip" "myeip" {

vpc = "true"

}

resource "aws\_eip" "myeip01" {

vpc = "true"

provider = "aws.aws02"

}

#### **1st EIP -- one region**

#### **2nd EIP -- second region**

### **providers.tf**

provider "aws" {

region = "us-west-1"

}

provider "aws" {

alias = "aws02"

region = "ap-south-1"

Profile = “Prod”

}

## **Handling Multiple AWS Profiles in Terraform**

You can optionally define multiple configurations for the same provider, and select which one to use on a per-resource or per-module basis.

The primary reason for this is to support multiple regions for a cloud platform.

To include multiple configurations for a given provider, include multiple provider blocks with the same provider name, but set the alias meta-argument to an alias name to use for each additional configuration. For example:



The provider block without alias set is known as the default provider configuration.

When an alias is set, it creates an additional provider configuration.

For providers that have no required configuration arguments, the implied empty configuration is considered to be the default provider configuration.

### **eip.tf**

resource "aws\_eip" "myeip" {

vpc = "true"

}

resource "aws\_eip" "myeip01" {

vpc = "true"

provider = "aws.aws02"

}

#### **1st EIP -- one region**

#### **2nd EIP -- second region**

### **providers.tf**

provider "aws" {

region = "us-west-1"

}

provider "aws" {

alias = "aws02"

region = "ap-south-1"

profile = "account02"

}

## **Integrating with GIT for team management**

Till now, we have been working with terraform code locally.



However, storing your configuration files locally is not always an idea specifically in the scenario were other members of the team are also working on Terraform.

For such cases, it is important to store your Terraorm code to a centralized repository like in Git.



## **Module Sources in Terraform**

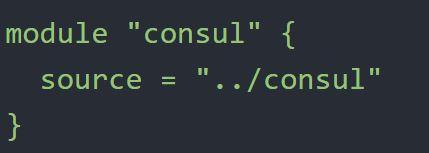
The source argument in a module block tells Terraform where to find the source code for the desired child module.

* Local paths
* Terraform Registry
* GitHub
* Bitbucket
* Generic Git, Mercurial repositories
* HTTP URLs
* S3 buckets
* GCS buckets

Let us explore some of the supported module sources.

Local Path

A local path must begin with either ./ or ../ to indicate that a local path is intended.



Git Module Source

Arbitrary Git repositories can be used by prefixing the address with the special git:: prefix.

After this prefix, any valid Git URL can be specified to select one of the protocols supported by Git.

.



Referencing to a Branch

By default, Terraform will clone and use the default branch (referenced by HEAD) in the selected repository.

You can override this using the ref argument:



.

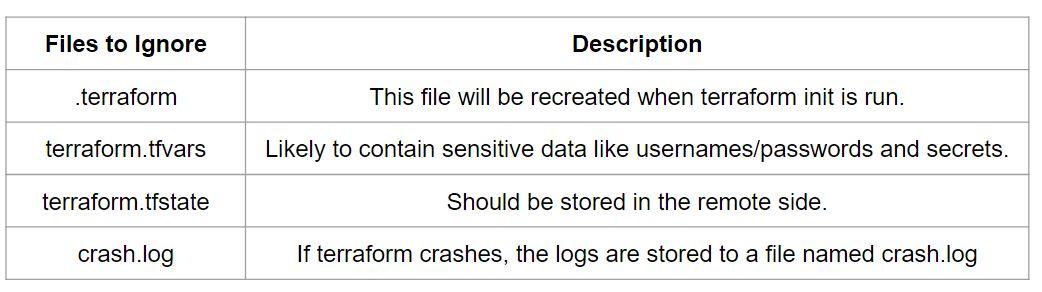
The value of the ref argument can be any reference that would be accepted by the git checkout command, including branch and tag names.

## **Terraform & GitIgnore**

The .gitignore file is a text file that tells Git which files or folders to ignore in a project.



Depending on the environment, it is recommended to avoid committing certain files to GIT.



## **Remote State Management**

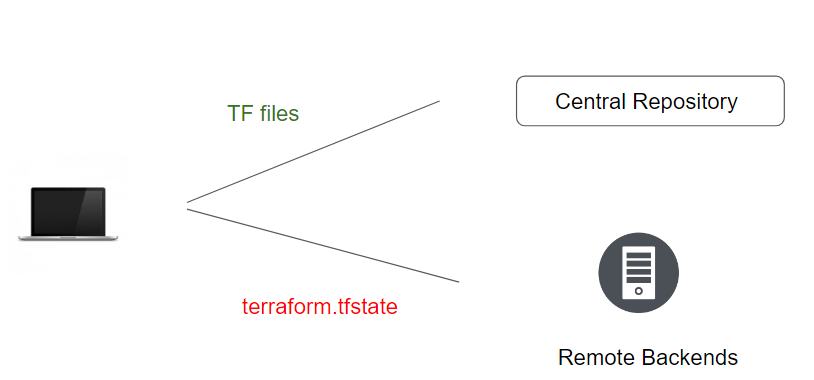
Terraform supports various types of remote backends which can be used to store state data.

As of now, we were storing state data in local and GIT repository.

Depending on remote backends that are being used, there can be various features.

* Standard BackEnd Type: State Storage and Locking
* Enhanced BackEnd Type: All features of Standard + Remote Management

In the ideal scenario, your terraform configuration code should be part of the centralized Git repositories and state file should be part of the remote backends.



During our demo, we had made use of S3 Backend to store our state file. Following is the sample configuration file for the S3 backend:



Example:

provider.tf

provider "aws" {

region = "us-west-1"

access\_key = "AWS\_Key"

secret\_key = "AWS\_Secret\_key"

}

main.tf

resource "aws\_eip" "myeip" {

vpc = "true"

}

backend.tf

terraform {

backend "s3" {

bucket = "your S3 bucket name"

key = "remotedemo.tfstate"

region = "us-west-1"

access\_key = "YOUR-ACCESS-KEY"

secret\_key = "YOUR-SECRET-KEY"

dynamodb\_table = "s3-state-lock"

}

}

### **DynamoDB State Locking**

The following configuration is optional:

* [dynamodb\_endpoint](https://developer.hashicorp.com/terraform/language/settings/backends/s3#dynamodb_endpoint) - (Optional) Custom endpoint for the AWS DynamoDB API. This can also be sourced from the AWS\_DYNAMODB\_ENDPOINT environment variable.
* [dynamodb\_table](https://developer.hashicorp.com/terraform/language/settings/backends/s3#dynamodb_table) - (Optional) Name of DynamoDB Table to use for state locking and consistency. The table must have a partition key named **LockID** with type of String. If not configured, state locking will be disabled.

<https://developer.hashicorp.com/terraform/language/settings/backends/s3>

## 

## **State File Locking**

Whenever you are performing a write operation, terraform would lock the state file.

This is very important as otherwise during your ongoing terraform apply operations, if others also try for the same, it would corrupt your state file.

Example:

* Person A is terminating the RDS resource which has associated rds.tfstate file
* Person B has now tried resizing the same RDS resource at the same time.

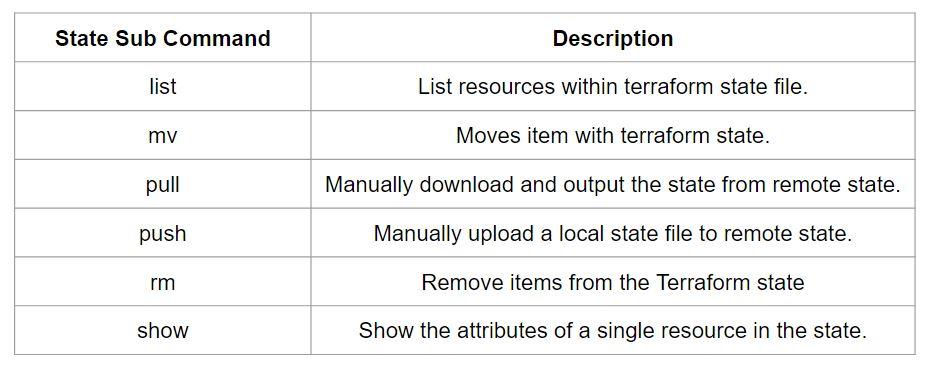
For the S3 backend, you can make use of the DynamoDB for state file locking functionality.

## **Terraform State Management**

As your Terraform usage becomes more advanced, there are some cases where you may need to modify the Terraform state.

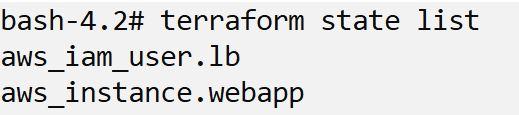
It is important to never modify the state file directly. Instead, make use of terraform state command.

There are multiple sub-commands that can be used with terraform state, these include:



Sub Command - List

The terraform state list command is used to list resources within a Terraform state.



Sub Command - Move

The terraform state mv command is used to move items in a Terraform state.

This command is used in many cases in which you want to rename an existing resource without destroying and recreating it.

Due to the destructive nature of this command, this command will output a backup copy of the state prior to saving any changes

Overall Syntax:

terraform state mv [options] SOURCE DESTINATION

Sub Command - Pull

The terraform state pull command is used to manually download and output the state from a remote state.

This is useful for reading values out of state (potentially pairing this command with something like jq).

Sub Command - Push

The terraform state push command is used to manually upload a local state file to remote state.

This command should rarely be used.

Sub Command - Remove

The terraform state rm command is used to remove items from the Terraform state.

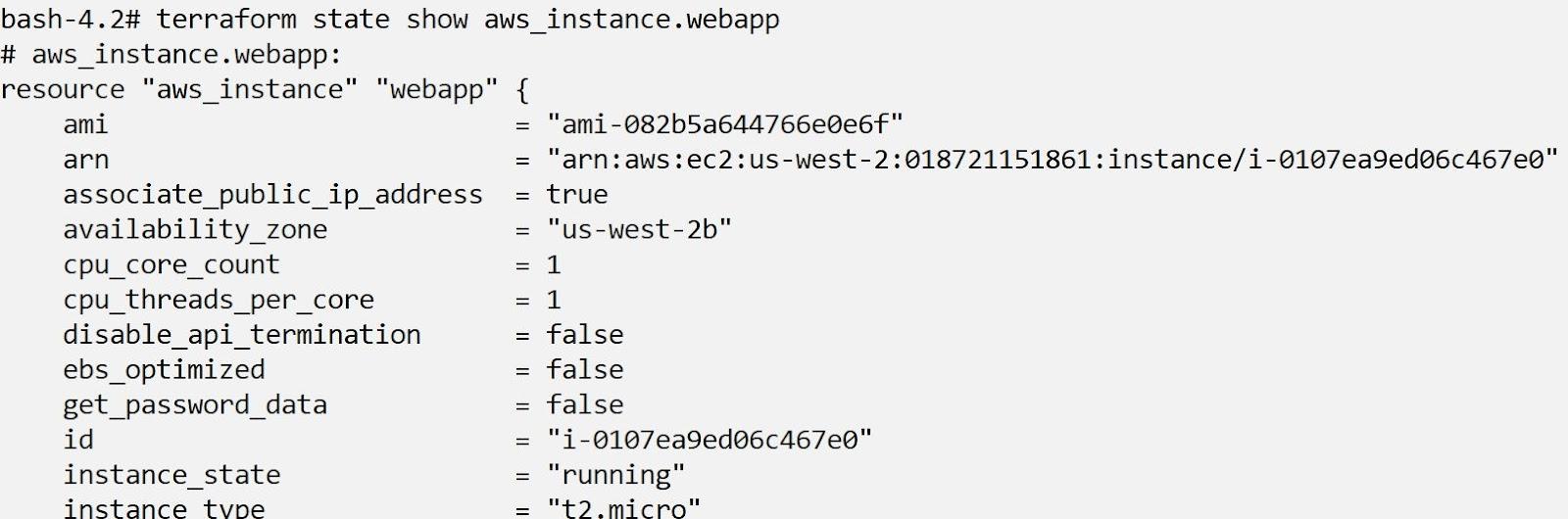
Items removed from the Terraform state are not physically destroyed.

Items removed from the Terraform state are only no longer managed by Terraform

For example, if you remove an AWS instance from the state, the AWS instance will continue running, but terraform plan will no longer see that instance.

Sub Command - Show

The terraform state show command is used to show the attributes of a single resource in the Terraform state.



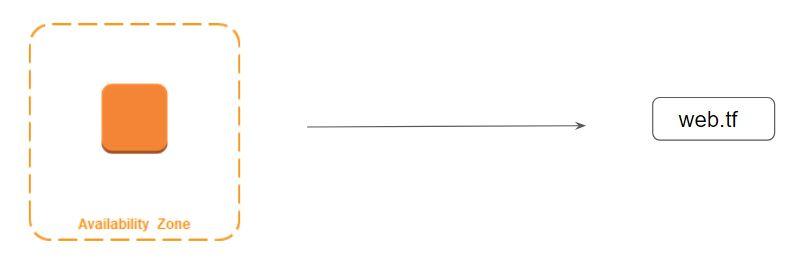
## 

## 

## **Terraform Import**

It might happen that there is a resource that is already created manually.

In such a case, any change you want to make to that resource must be done manually.



Terraform is able to import existing infrastructure. This allows you to take resources you've created by some other means and bring it under Terraform management.

The current implementation of Terraform import can only import resources into the state. It does not generate configuration. A future version of Terraform will also generate configuration.

Because of this, prior to running terraform import it is necessary to write manually a resource configuration block for the resource, to which the imported object will be mapped.

Example:

### ec2.tf

resource "aws\_instance" "myec2" {

ami = "ami-bf5540df"

instance\_type = "t2.micro"

vpc\_security\_group\_ids = ["sg-6ae7d613", "sg-53370035"]

key\_name = "remotepractical"

subnet\_id = "subnet-9e3cfbc5"

tags {

Name = "manual"

}

}

### providers.tf

provider "aws" {

region = "us-west-1"

}

### Command To Import Resource

terraform import aws\_instance.myec2 i-041886ebb7e9bd20