HashiCorp Certified Terraform Associate 2023
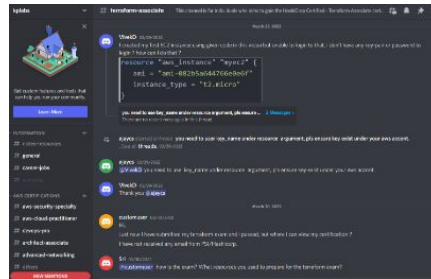
Instructed by Zeal Vora

Our Community

You can join our Discord community for any queries / discussions. You can also connect with other students going through the same course in Discord (Optional)

Discord Link: http://kplabs.in/chat

Group Page: #terraform-associate

PPT Version

PPT Release Date = 3rd August 2023

We regularly release new version of PPT when we update this course.

Please check regularly that you are using the latest version.

The Latest Version Details are mentioned in the PPT Lecture in Section 1.

IAC Tools

DevOps = Developers

Exploring Toolsets

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

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

Configuration Management vs Infrastructure Orchestration

Ansible, Chef, Puppet are configuration management tools which means that they are primarily designed to install and manage software on existing servers.

Terraform, CloudFormation are the infrastructure orchestration tools which basically means they can provision the servers and infrastructure by themselves.

Configuration Management tools can do some degree of infrastructure provisioning, but the focus here is that some tools are going to be better fit for certain type of tasks.

IAC & Configuration Management = Friends
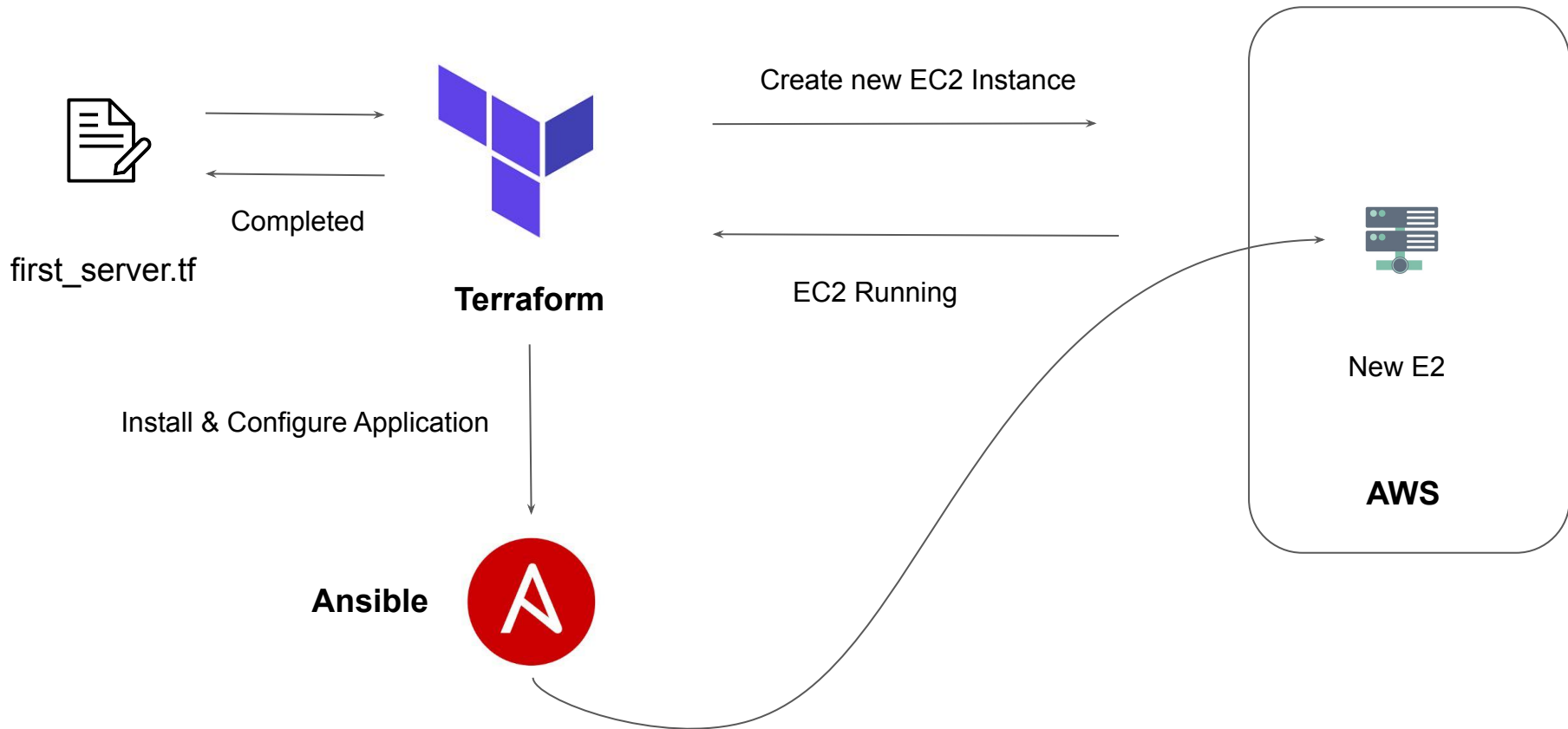
Which tool to choose ?

Question remains on how to choose right IAC tool for the organization

i) Is your infrastructure going to be vendor specific in longer term ? Example AWS.

ii) Are you planning to have multi-cloud / hybrid cloud based infrastructure ?

iii) How well does it integrate with configuration management tools ?

iv) Price and Support

Terraform

i) Supports multiple platforms, has hundreds of providers.

ii) Simple configuration language and faster learning curve.

iii) Easy integration with configuration management tools like Ansible.

iv) Easily extensible with the help of plugins.

v) Free !!!

Installing Terraform

Terraform in detail

Overview of Installation Process

Terraform installation is very simple.

You have a single binary file, download and use it.

terraform

Download

Supported Platforms

Terraform works on multiple platforms, these includes:

- Windows
- macOS
- Linux
- FreeBSD
- OpenBSD
- Solaris

Terraform Installation - Mac & Linux

There are two primary steps required to install terraform in Mac and Linux

1) Download the Terraform Binary File.

2) Move it in the right path.

Choosing IDE For Terraform

Terraform in detail

Terraform Code in NotePad!

You can write Terraform code in Notepad and it will not have any impact.
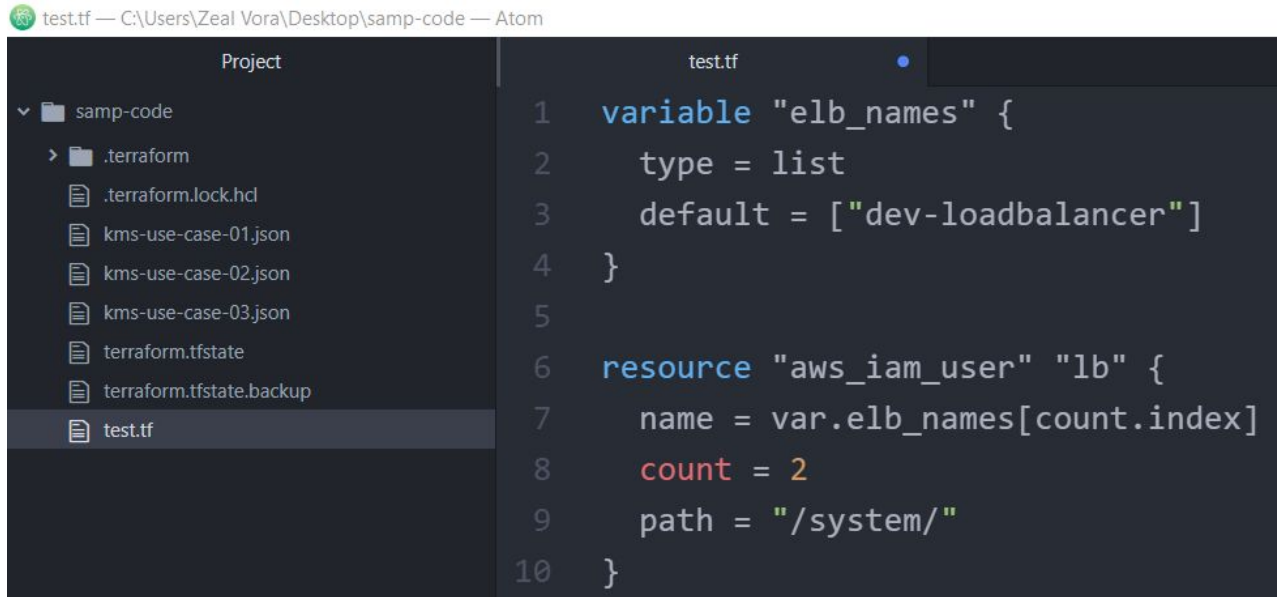
Downsides:

- Slower Development
- Limited Features


```
*test.tf - Notepad
File  Edit  Format  View  Help
variable "elb_names" {
  type = list
  default = ["dev-loadbalancer"]
}

resource "aws_iam_user" "lb" {
  name = var.elb_names[count.index]
  count = 2
  path = "/system/"
}
```

Need of a Better Software

There is a need of a better application that allows us to develop code faster.
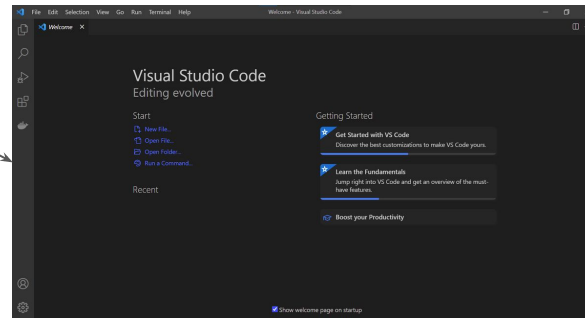
What are the Options!

There are many popular source code editors available in the market.

Source Code Editors

Editor for This Course

We are going to make use of Visual Studio Code as primary editor in this course.

Advantages:

1. Supports Windows, Mac, Linux
2. Supports Wide variety of programming languages.
3. Many Extensions.
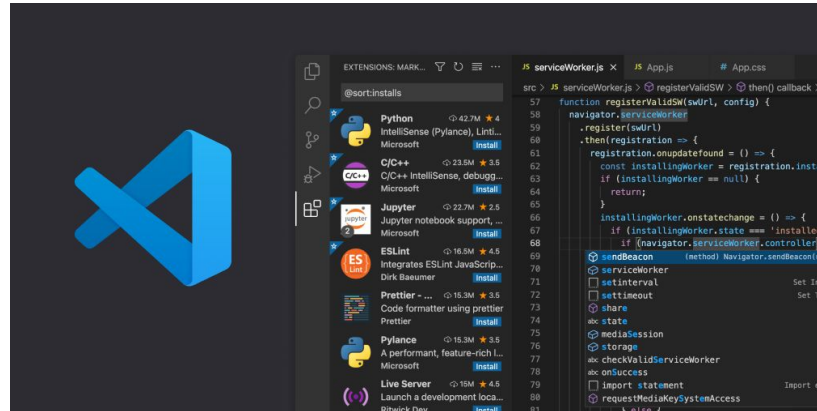
Visual Studio Code Extensions

Understanding the Basics

Extensions are add-ons that allow you to customize and enhance your experience in Visual Studio by adding new features or integrating existing tools

They offer wide range of functionality related to colors, auto-complete, report spelling errors etc.
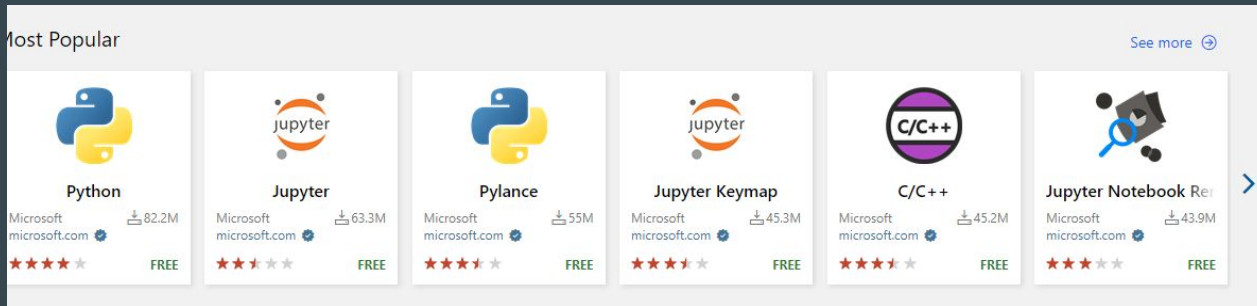
Terraform Extension

HashiCorp also provides extension for Terraform for Visual Studio Code.


```
demo.tf
  resource "aws_instance" "myec2" {
      ami = "ami-00c39f71452c08778"
      instance_type = "t2.micro"
  }
```



```
demo.tf > ...
  resource "aws_instance" "myec2" {
      ami = "ami-00c39f71452c08778"
      instance_type = "t2.micro"
  }
```

Setting up the Lab

Let's start Rolling !

Let's Start

i) Create a new AWS Account.

ii) Begin the course

Registering an AWS Account

Authentication and Authorization

Understanding the Basics

Before we start working on managing environments through Terraform, the first important step is related to Authentication and Authorization.
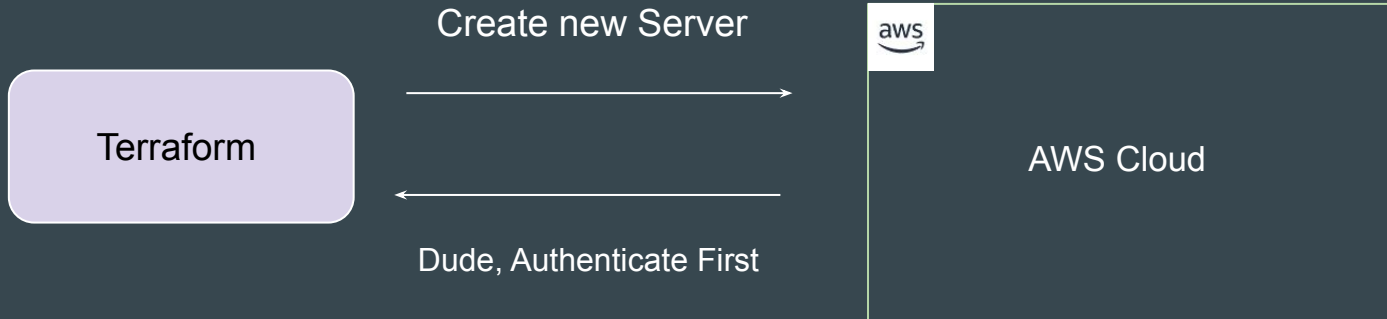
Basics of Authentication and Authorization

Authentication is the process of verifying who a user is.

Authorization is the process of verifying what they have access to

Example:

Alice is a user in AWS with no access to any service.

Access Credentials

Depending on the provider, the type of access credentials would change.

Provider	Access Credentials
AWS	Access Keys and Secret Keys
GitHub	Tokens
Kubernetes	Kubeconfig file, Credentials Config
Digital Ocean	Tokens
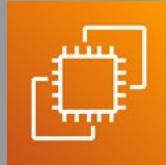
Revising the Basics of EC2

EC2 stands for Elastic Compute Cloud.

In-short, it's a name for a virtual server that you launch in AWS.
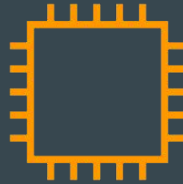
VM ←————————→ EC2 Instance

Available Regions

Cloud providers offers multiple regions in which we can create our resource.

You need to decide the region in which Terraform would create the resource.

Virtual Machine Configuration

A Virtual Machine would have it's own set of configurations.

- CPU
- Memory
- Storage
- Operating System

While creating VM through Terraform, you will need to define these.

Providers and Resources
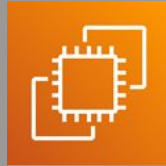
Basics of Providers

Terraform supports multiple providers.

Depending on what type of infrastructure we want to launch, we have to use appropriate providers accordingly.
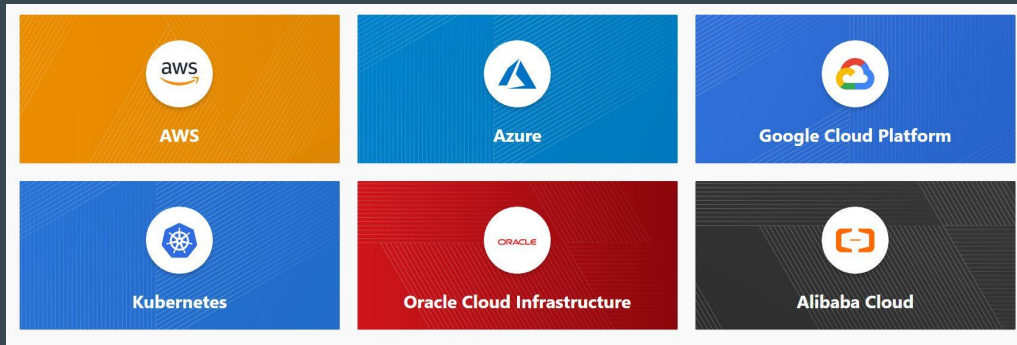
Learning 1 - Provider Plugins

A provider is a plugin that lets Terraform manage an external API.

When we run terraform init, plugins required for the provider are automatically downloaded and saved locally to a .terraform directory.

```
C:\Users\zealv\Desktop\kplabs-terraform>terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v4.60.0...
- Installed hashicorp/aws v4.60.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

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.
```

Learning 2 - Resource

Resource block describes one or more infrastructure objects

Example:

- resource aws_instance
- resource aws_alb
- resource iam_user
- resource digitalocean_droplet

```
resource "aws_instance" "myec2" {
    ami = "ami-082b5a644766e0e6f"
    instance_type = "t2.micro"
}
```

Learning 3 - Resource Blocks

A resource block declares a resource of a given type ("aws_instance") with a given local name ("myec2").

Resource type and Name together serve as an identifier for a given resource and so must be unique.

```
resource "aws_instance" "myec2" {
    ami = "ami-082b5a644766e0e6f"
    instance_type = "t2.micro"
}
```

EC2 Instance Number 1

```
resource "aws_instance" "web" {
    ami           = ami-123
    instance_type = "t2.micro"
}
```

EC2 Instance Number 2

Point to Note

You can only use the resource that are supported by a specific provider.

In the below example, provider of Azure is used with resource of aws_instance

```
provider "azurerm" {}

resource "aws_instance" "web" {
  ami           = ami-123
  instance_type = "t2.micro"

}
```

Important Question

The core concepts, standard syntax remains similar across all providers.
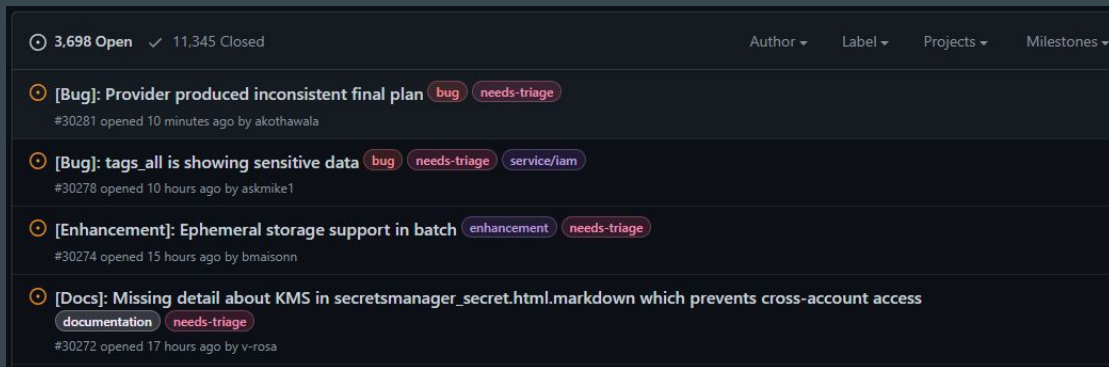
If you learn the basics, you should be able to work with all providers easily.

Issues and Bugs with Providers

A provider that is maintained by HashiCorp does not mean it has no bugs.

It can happen that there are inconsistencies from your output and things mentioned in documentation. You can raise issue at Provider page.

Relax and Have a Meme Before Proceeding

Provider Tiers

Provider Maintainers

There are 3 primary type of provider tiers in Terraform.

Provider Tiers	Description
Official	Owned and Maintained by HashiCorp.
Partner	Owned and Maintained by Technology Company that maintains direct partnership with HashiCorp.
Community	Owned and Maintained by Individual Contributors.

Provider Namespace

Namespaces are used to help users identify the organization or publisher responsible for the integration

Tier	Description
Official	hashicorp
Partner	Third-party organization e.g. mongodb/mongodbatlas
Community	Maintainer's individual or organization account, e.g. DeviaVir/gsuite

Important Learning

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

```
provider "aws" {
  region     = "us-west-2"
  access_key = "PUT-YOUR-ACCESS-KEY-HERE"
  secret_key = "PUT-YOUR-SECRET-KEY-HERE"
}
```

HashiCorp Maintained

```
terraform {
  required_providers {
    digitalocean = {
      source = "digitalocean/digitalocean"
    }
  }
}


provider "digitalocean" {
  token = "PUT-YOUR-TOKEN-HERE"
}
```

Non-HashiCorp Maintained

Terraform Destroy

Learning to Destroy Resources

If you keep the infrastructure running, you will get charged for it.

Hence it is important for us to also know on how we can delete the infrastructure resources created via terraform.
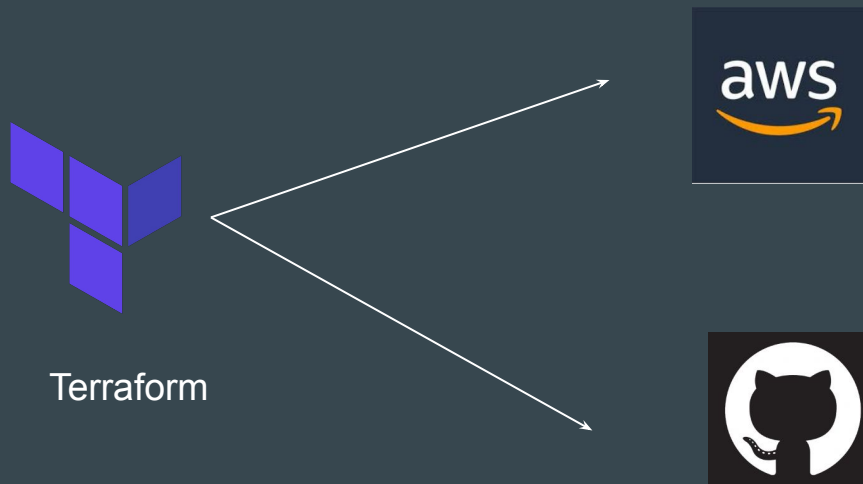
Terraform

Approach 1 - Destroy ALL

terraform destroy allows us to destroy all the resource that are created within the folder.
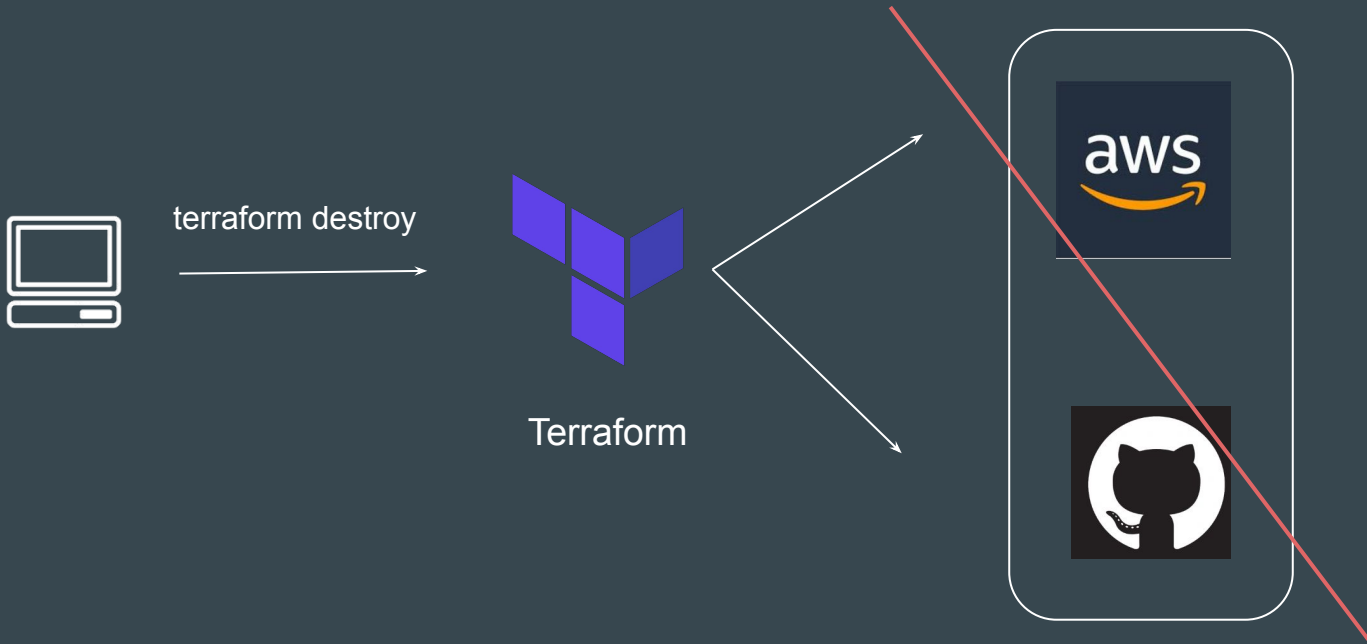
Approach 2 - Destroy Some

terraform destroy with -target flag allows us to destroy specific resource.
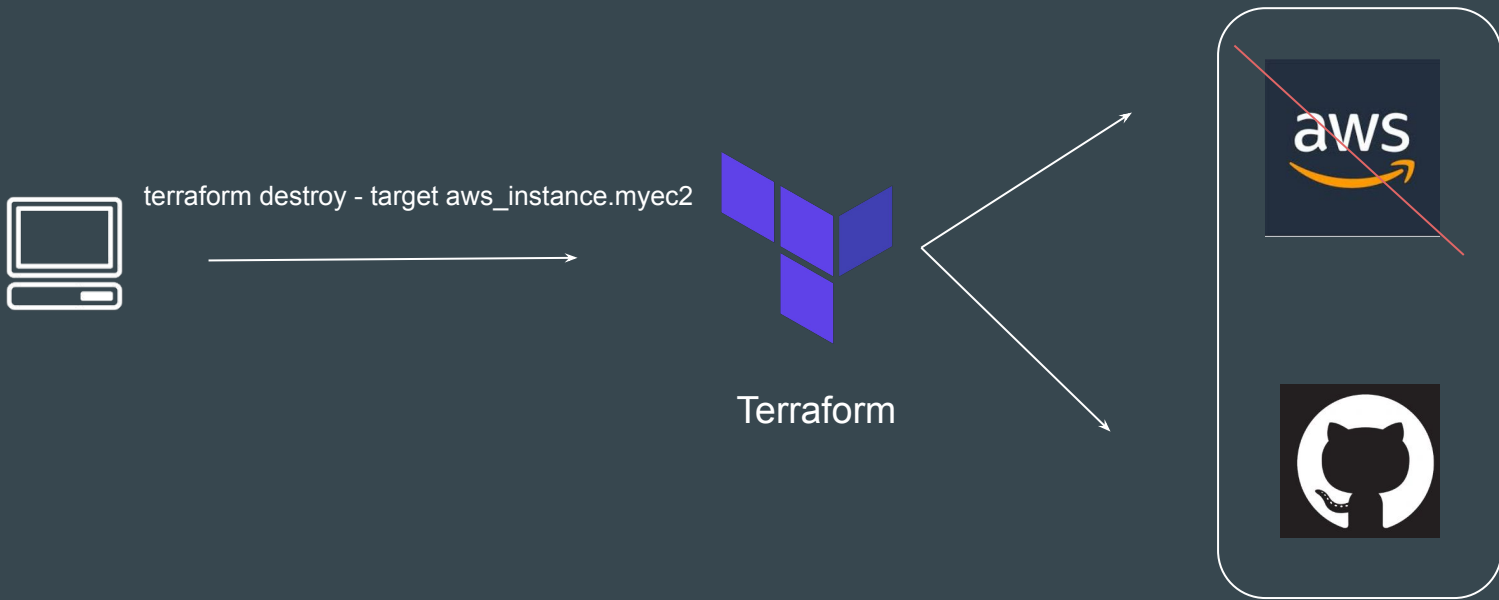
Terraform Destroy with Target

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

Combination of : Resource Type + Local Resource Name

Resource Type	Local Resource Name
aws_instance	myec2
github_repository	example

```
resource "aws_instance" "myec2" {
    ami = "ami-00c39f71452c08778"
    instance_type = "t2.micro"
}
```

```
resource "github_repository" "example" {
  name        = "example"
  description = "My awesome codebase"

  visibility = "public"

}
```

Desired & Current State

Terraform in detail

Desired State

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

```
resource "aws_instance" "myec2" {
    ami = "ami-082b5a644766e0e6f"
    instance_type = "t2.micro"
}
```
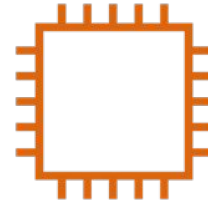
EC2 - t2.micro

Current State

Current state is the actual state of a resource that is currently deployed.

```
resource "aws_instance" "myec2" {
    ami = "ami-082b5a644766e0e6f"
    instance_type = "t2.micro"
}
```


t2.medium

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.

CURRENT
STATE

GET BACK TO
DESIRED STATE

Provider Versioning

Terraform in detail

Provider Architecture

do_droplet.tf

Terraform

Digital Ocean Provider

Infrastructure Provisioning

(API interactions)

New Server

Digital Ocean
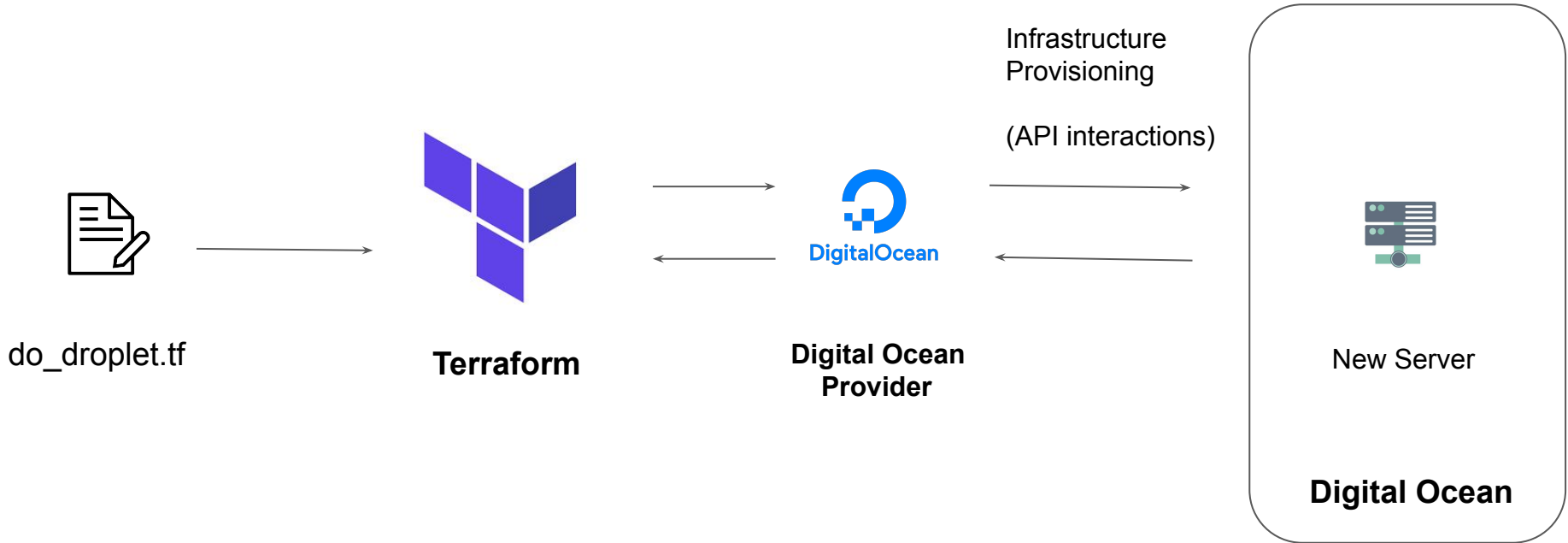
Overview of Provider Versioning

Provider plugins are released separately from Terraform itself.

They have different set of version numbers.

.

Version 1

Version 2

Explicitly Setting Provider Version

During terraform init, if 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.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}


provider "aws" {
  region = "us-east-1"
}
```

Arguments for Specifying provider

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

Version Number Arguments	Description
>=1.0	Greater than equal to the version
<=1.0	Less than equal to the version
~>2.0	Any version in the 2.X range.
>=2.10,<=2.30	Any version between 2.10 and 2.30

Dependency Lock File

Terraform dependency lock file allows us to lock to a specific version of the provider.

If a particular provider already has a selection recorded in the lock file, Terraform will always re-select that version for installation, even if a newer version has become available.

You can override that behavior by adding the -upgrade option when you run terraform init,

```
provider "registry.terraform.io/hashicorp/aws" {
  version     = "2.70.0"
  constraints = ">= 2.31.0, <= 2.70.0"
  hashes = [
    "h1:fx8tbGVwK1YIDI6UdHLnorC9PA1ZPSWEeW3V3aDCdWY=",
    "zh:01a5f351146434b418f9ff8d8cc956ddc801110f1cc8b139e01be2ff8c544605",
    "zh:1ec08abbaf09e3e0547511d48f77a1e2c89face2d55886b23f643011c76cb247",
    "zh:606d134fef7c1357c9d155aadbee6826bc22bc0115b6291d483bc1444291c3e1",
    "zh:67e31a71a5ecbbc96a1a6708c9cc300bbfe921c322320cdbb95b9002026387e1",
```

Terraform Refresh

Understanding the Challenge

Terraform can create an infrastructure based on configuration you specified.

It can happen that the infrastructure gets modified manually.

```
resource "aws_instance" "web" {
  ami           = ami-123
  instance_type = "t2.micro"
}
```

State File

EC2:

type: t2.micro
storage: 20
sg: default

t2.micro

Understanding the Challenge

The terraform refresh command will check the latest state of your infrastructure and update the state file accordingly.
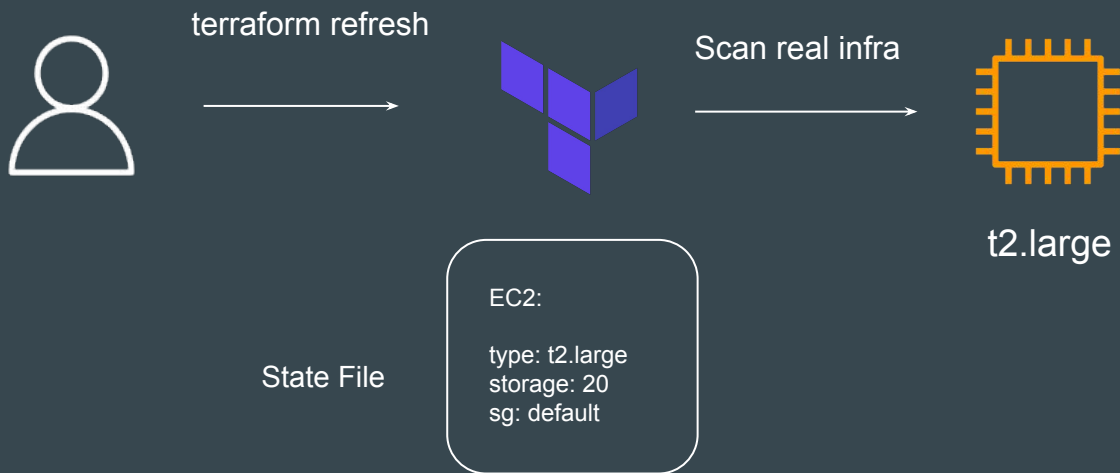
Points to Note

You shouldn't typically need to use this command, because Terraform automatically performs the same refreshing actions as a part of creating a plan in both the terraform plan and terraform apply commands.

Understanding the Usage

The terraform refresh command is deprecated in newer version of terraform.

The -refresh-only option for terraform plan and terraform apply was introduced in Terraform v0.15.4.

AWS Provider - Authentication Configuration

Understanding the Basics

At this stage, we have been manually hardcoding the access / secret keys within the provider block.

Although a working solution, but it is not optimal from security point of view.

```
aws-provider-config.tf > ...
    provider "aws" {
        region     = "us-east-1"
        access_key = "AKIAQTSHLD4OASDZ5QJI"
        secret_key = "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"
    }

    resource "aws_eip" "lb" {
        domain   = "vpc"
    }
```

Better Way

We want our code to run successfully without hardcoding the secrets in the provider block.

```
aws-provider-config.tf > ...
    provider "aws" {
      region     = "us-east-1"
    }

    resource "aws_eip" "lb" {
      domain   = "vpc"
    }
```

Better Approach

The AWS Provider can source credentials and other settings from the shared configuration and credentials files.

```
aws-provider-config.tf > ...
  provider "aws" {
    shared_config_files      = ["/Users/tf_user/.aws/conf"]
    shared_credentials_files = ["/Users/tf_user/.aws/creds"]
    profile                  = "customprofile"
  }

  resource "aws_eip" "lb" {
    domain   = "vpc"
  }
```

Default Configurations

If shared files lines are not added to provider block, by default, Terraform will locate these files at $HOME/.aws/config and $HOME/.aws/credentials on Linux and macOS.

"%USERPROFILE%\.aws\config" and "%USERPROFILE%\.aws\credentials" on Windows.

```
aws-provider-config.tf > ...
    provider "aws" {
      region      = "us-east-1"
    }


    resource "aws_eip" "lb" {
      domain   = "vpc"
    }
```

AWS CLI

AWS CLI allows customers to manage AWS resources directly from CLI.

When you configure Access/Secret keys in AWS CLI, the location in which these credentials are stored is the same default location that Terraform searches the credentials from.

Create EC2 Instance

awscli

AWS Platform

Lecture Format - Terraform Course

Terraform in detail

Overview of the Format

We tend to use a different folder for each practical that we do in the course.

This allows us to be more systematic and allows easier revisit in-case required.

Lecture Name	Folder Names
Create First EC2 Instance	folder1
Tainting resource	folder2
Conditional Expression	folder3

Find the appropriate code from GitHub

Code in GitHub is arranged according to sections that are matched to the domains in the course.

Every section in GitHub has easy Readme file for quick navigation.

Video-Document Mapper

Sr No	Document Link
1	Understanding Attributes and Output Values in Terraform
2	Referencing Cross-Account Resource Attributes
3	Terraform Variables
4	Approaches for Variable Assignment
5	Data Types for Variables

Destroy Resource After Practical

We know how to destroy resources by now

terraform destroy

After you have completed your practical, make sure you destroy the resource before moving to the next practical.

This is easier if you are maintaining separate folder for each practical.

Relax and Have a Meme Before Proceeding

Cross-Resource Attribute References

Typical Challenge

It can happen that in a single terraform file, you are defining two different resources.

However Resource 2 might be dependent on some value of Resource 1.

Elastic IP Address

Security group

Allow 443 from Elastic IP

Understanding The Workflow


```
 eip.tf > ...
   resource "aws_eip" "lb" {
     domain  = "vpc"
   }
```

```
resource "aws_security_group" "allow_tls" {
  name        = "allow_tls"
  description = "Allow TLS inbound traffic"

  ingress {
    description    = "TLS from VPC"
    from_port      = 443
    to_port        = 443
    protocol       = "tcp"
    cidr_blocks    = [HOW-TO-ADD-ELASTIC-IP-ADDRESS-HERE?]
  }
}
```

Elastic IP

52.72.30.50

Security group

Allow 443 from 52.72.30.50

Basics of Attributes

Each resource has its associated set of attributes.

Attributes are the fields in a resource that hold the values that end up in state.
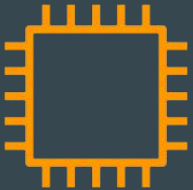
Attributes	Values
ID	i-abcd
public_ip	52.74.32.50
private_ip	172.31.10.50
private_dns	ip-172-31-10-50-.ec2.internal

Cross Referencing Resource Attribute

Terraform allows us to reference the attribute of one resource to be used in a different resource.

Elastic IP

Attribute	Value
public_ip	52.72.52.72

```
resource "aws_security_group" "allow_tls" {
  name        = "allow_tls"
  description = "Allow TLS inbound traffic"
  vpc_id      = aws_vpc.main.id

  ingress {
    description     = "TLS from VPC"
    from_port       = 443
    to_port         = 443
    protocol        = "tcp"
    cidr_blocks     = [aws_eip.myeip.public_ip]

  }
```

Output Values

Understanding the Basics

Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use.

Create EC2 and give me
it's Public IP

Create EC2

Terraform

IP Of EC2: 172.32.10.50

Fetch Info of EC2

Sample Example

Use-Case:

Create a Elastic IP (Public IP) resource in AWS and output the value of the EIP.

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

Changes to Outputs:
  + demo = (known after apply)
aws_eip.lb: Creating...
aws_eip.lb: Creation complete after 3s [id=eipalloc-0680508decfe8c252]

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

Outputs:

demo = "54.146.20.18"
```

Point to Note

Output values defined in Project A can be referenced from code in Project B as well.
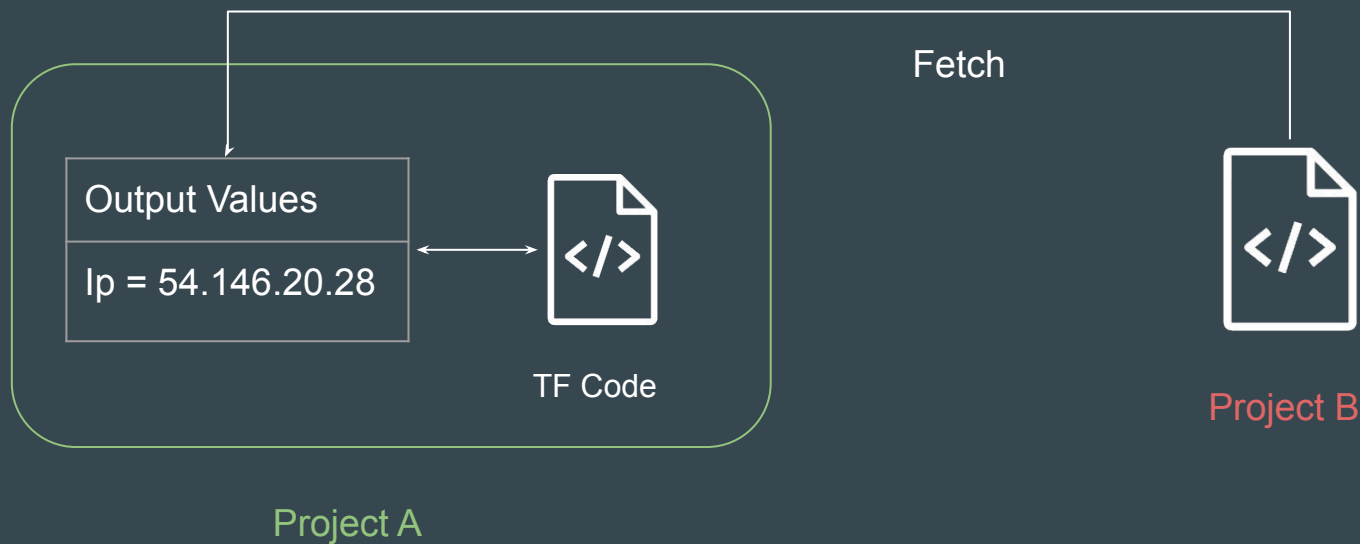
Terraform Variables

Terraform in detail

Static = Work

Repeated static values can create more work in the future.
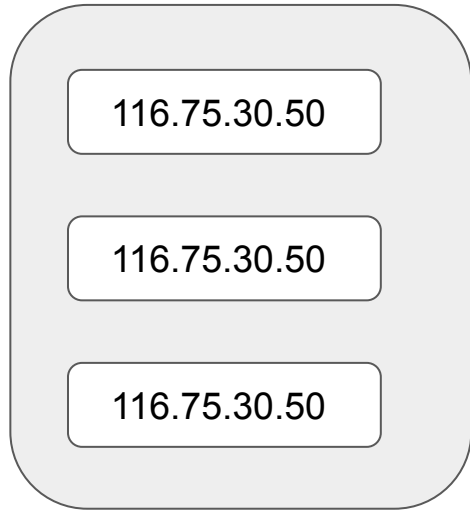
Project A
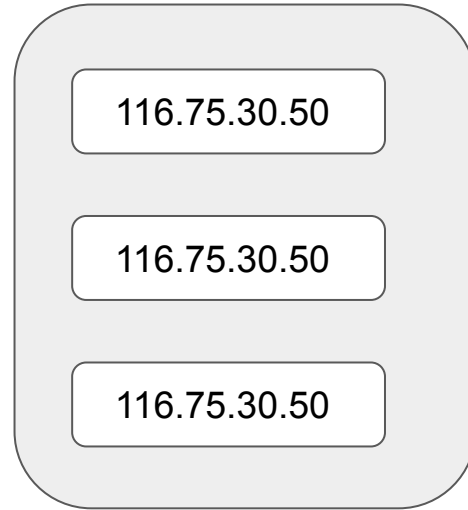
Project B

Variables are good

We can have a central source from which we can import the values from.

116.75.30.50

128.30.50.90

source

var.source

var.source

var.source
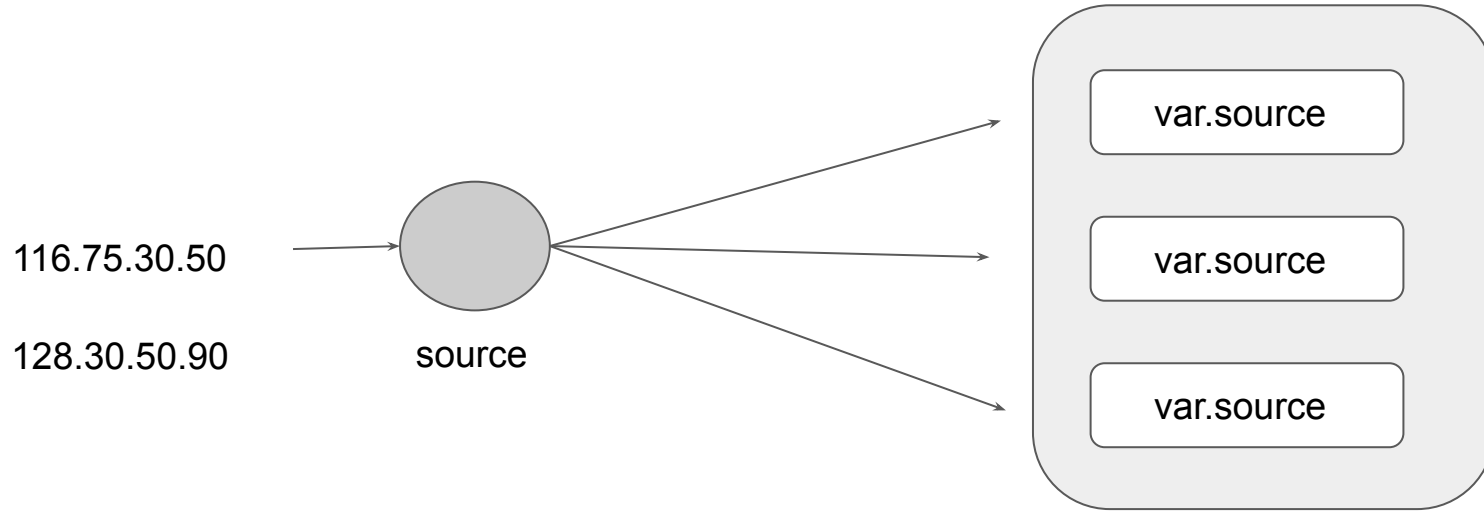
Variables are good

We can have a central source from which we can import the values from.
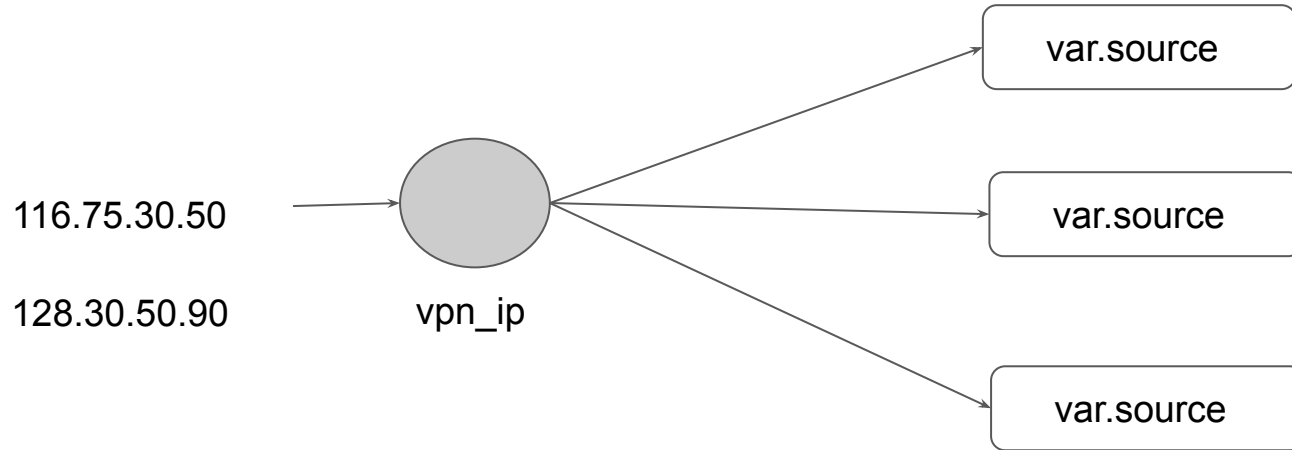
Approaches to Variable Assignment

Terraform in detail

Multiple Approaches to 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

Data Types for Variables

Terraform in detail

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

Every employee in Medium Corp is assigned a Identification Number.

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

variables.tf	terraform.tfvars
variable "instance_name" {}	instance_name="john-123"

Example Use-Case

Every employee in Medium Corp is assigned a Identification Number.

Any EC2 instance that employee creates should be created using the identification number only.

variables.tf	terraform.tfvars

variable "instance_name" {
 type=number
 }
	instance_name="john-123"

Overview of Data Types

Type Keywords	Description
string	Sequence of Unicode characters representing some text, like "hello".
list	Sequential list of values identified by their position. Starts with 0 ["mumbai" ,"singapore", "usa"]
map	a group of values identified by named labels, like {name = "Mabel", age = 52}.
number	Example: 200

Count Parameter

Terraform in detail

Overview of Count Parameter

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 approach is to define two separate resource blocks for aws_instance.

```
resource "aws_instance" "instance-1" {
    ami = "ami-082b5a644766e0e6f"
    instance_type = "t2.micro"
}
```

➕

```
resource "aws_instance" "instance-2" {
    ami = "ami-082b5a644766e0e6f"
    instance_type = "t2.micro"
}
```

Overview of Count Parameter

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

```
resource "aws_instance" "instance-1" {
    ami = "ami-082b5a644766e0e6f"
    instance_type = "t2.micro"
    count = 5
}
```

Count Index

In resource blocks where 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.

Understanding Challenge with Count

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

```
resource "aws_iam_user" "lb" {
  name = "loadbalancer"
  count = 5
  path = "/system/"
}
```

Understanding Challenge with Count

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

```
resource "aws_iam_user" "lb" {
  name = "loadbalancer.${count.index}"
  count = 5
  path = "/system/"
}
```

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 scenario as well.

```
variable "elb_names" {
  type          = list
  default       = ["dev-loadbalancer", "stage-loadbalancer", "prod-loadbalancer"]
}
```

Conditional Expression

Terraform in detail

Overview of 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 condition is true then the result is true_val. If condition is false then the result is false_val.

Example of Conditional Expression

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.
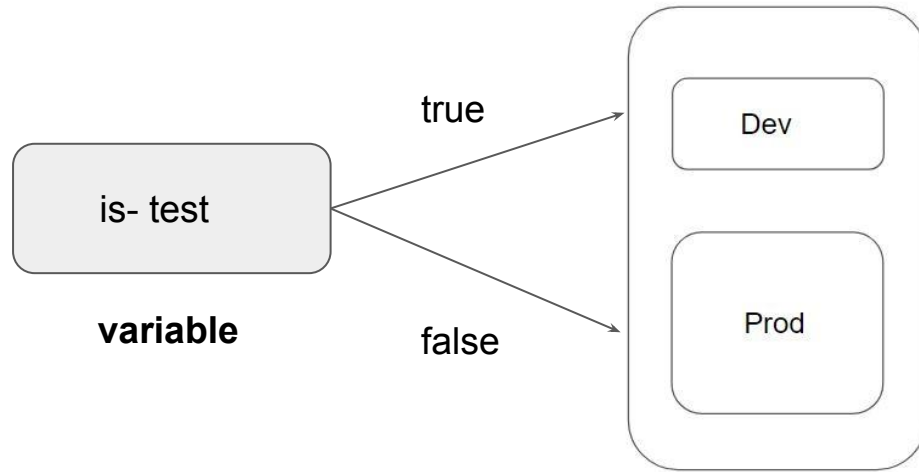
Local Values

Terraform in detail

Overview of Local Values

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

```
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_ebs_volume" "db_ebs" {
    availability_zone = "us-west-2a"
    size              = 8
    tags = local.common_tags
}
```

Local Values Support for Expression

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

```
locals {
  name_prefix = "${var.name != "" ? var.name : var.default}"
}
```

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 future.

Terraform Functions

Terraform in detail

Overview of 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
```

List of Available Functions

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

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

Data Sources

Terraform in detail

Overview of Data Sources

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

EC2 Instance

Mumbai Region

Singapore Region

Tokyo Region

ami-1234

ami-5678

ami-9012
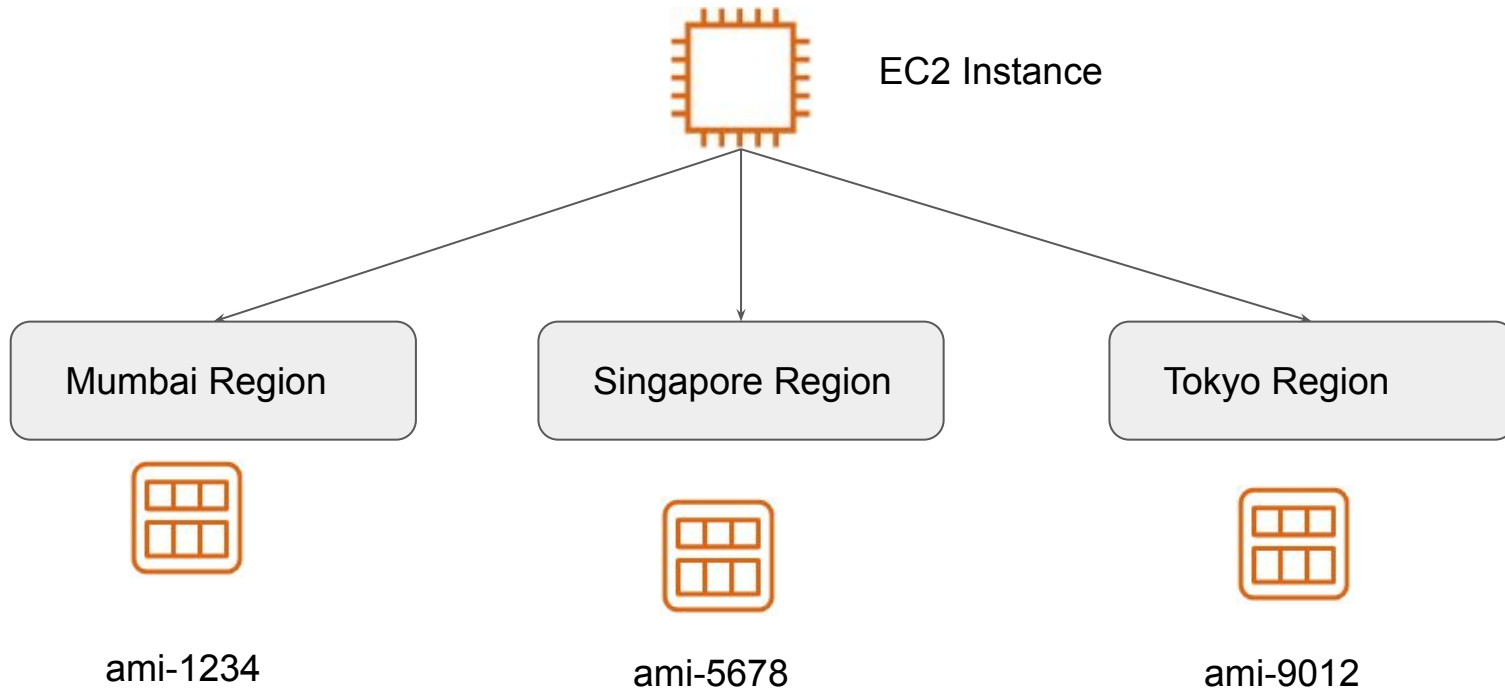
Data Source Code

- Defined under the data block.
- Reads from a specific data source (aws_ami) and exports results under "app_ami"

```
data "aws_ami" "app_ami" {
  most_recent = true
  owners = ["amazon"]


  filter {
    name   = "name"
    values = ["amzn2-ami-hvm*"]
  }
}
```

```
resource "aws_instance" "instance-1" {
    ami = data.aws_ami.app_ami.id
   instance_type = "t2.micro"
}
```

Debugging Terraform

Terraform in detail

Overview of Debugging Terraform

Terraform has detailed logs which 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

```
bash-4.2# terraform plan
2020/04/22 13:45:31 [INFO] Terraform version: 0.12.24
2020/04/22 13:45:31 [INFO] Go runtime version: go1.12.13
2020/04/22 13:45:31 [INFO] CLI args: []string{"/usr/bin/terraform", "plan"}
2020/04/22 13:45:31 [DEBUG] Attempting to open CLI config file: /root/.terraformrc
2020/04/22 13:45:31 [DEBUG] File doesn't exist, but doesn't need to. Ignoring.
2020/04/22 13:45:31 [DEBUG] checking for credentials in "/root/.terraform.d/plugins"
2020/04/22 13:45:31 [INFO] CLI command args: []string{"plan"}
2020/04/22 13:45:31 [TRACE] Meta.Backend: built configuration for "s3" backend with hash value 789489680
2020/04/22 13:45:31 [TRACE] Meta.Backend: backend has not previously been initialized in this working directory
2020/04/22 13:45:31 [DEBUG] New state was assigned lineage "a10f92bf-686d-e6cf-3e9d-755be5c8a6a3"
2020/04/22 13:45:31 [TRACE] Meta.Backend: moving from default local state only to "s3" backend
```

Important Pointers

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.

Lecture Format - Terraform Course

Terraform in detail

Overview of the Format

We tend to use a different folder for each practical that we do in the course.

This allows us to be more systematic and allows easier revisit in-case required.

Lecture Name	Folder Names
Create First EC2 Instance	folder1
Tainting resource	folder2
Conditional Expression	folder3

Find the appropriate code from GitHub

Code in GitHub is arranged according to sections that are matched to the domains in the course.

Every section in GitHub has easy Readme file for quick navigation.

Video-Document Mapper

Sr No	Document Link
1	Understanding Attributes and Output Values in Terraform
2	Referencing Cross-Account Resource Attributes
3	Terraform Variables
4	Approaches for Variable Assignment
5	Data Types for Variables

Destroy Resource After Practical

We know how to destroy resources by now

terraform destroy

After you have completed your practical, make sure you destroy the resource before moving to the next practical.

This is easier if you are maintaining separate folder for each practical.

Relax and Have a Meme Before Proceeding

Terraform Format

Terraform in detail

Importance of Readability

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.

```
provider "aws" {
      region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key    = "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"
  version    = ">=2.10,<=2.30"
}
```

Before fmt

```
provider "aws" {
      region       = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key    = "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"
  version     = ">=2.10,<=2.30"
}
```

After fmt

```
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
  version    = ">=2.10,<=2.30"
}
```

Terraform Validate

Terraform in detail

Overview of Terraform Validate

Terraform Validate primarily checks whether a configuration is syntactically valid.

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

```
resource "aws_instance" "myec2" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
  sky = "blue"
}
```

```
bash-4.2# terraform validate

Error: Unsupported argument

  on validate.tf line 10, in resource "aws_instance" "myec2":
  10:   sky = "blue"

An argument named "sky" is not expected here.
```

Load Order & Semantics

Terraform in detail

Understanding 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.

terraform-kplabs

```
web.tf          app.tf          sg.tf          providers.tf
```
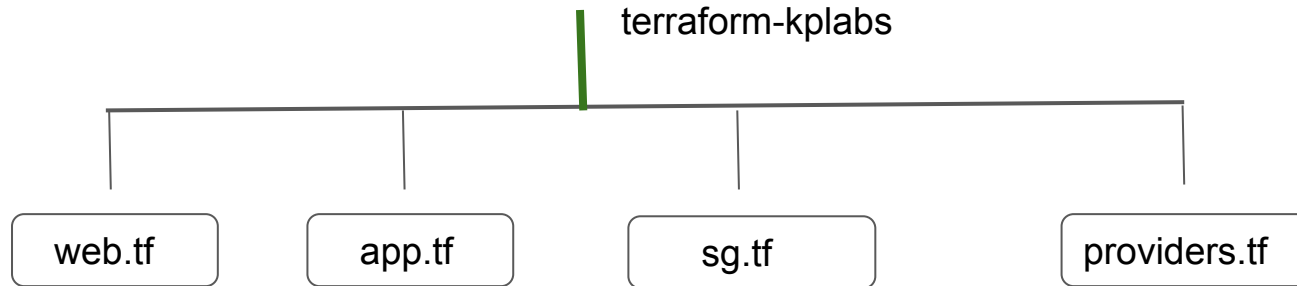
Dynamic Block

Terraform In Depth

Understanding the Challenge

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

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

```
ingress {
  from_port     = 9200
  to_port       = 9200
  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"]
}
```

Dynamic Blocks

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

```
dynamic "ingress" {
  for_each = var.ingress_ports
  content {
    from_port   = ingress.value
    to_port     = ingress.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

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).

```
dynamic "ingress" {
  for_each = var.ingress_ports
  content {
    from_port   = ingress.value
    to_port     = ingress.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

```
dynamic "ingress" {
  for_each = var.ingress_ports
  iterator = port
  content {
    from_port   = port.value
    to_port     = port.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Terraform Taint

Understanding the Use-Case

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 Changes to Terraform / Delete & Recreate the resource

Lots of manual changes
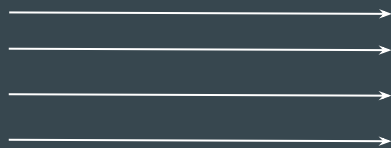
Terraform Managed Resource

Recreating the Resource

The -replace option with terraform apply to force Terraform to replace an object even though there are no configuration changes that would require it.

terraform apply -replace="aws_instance.web"
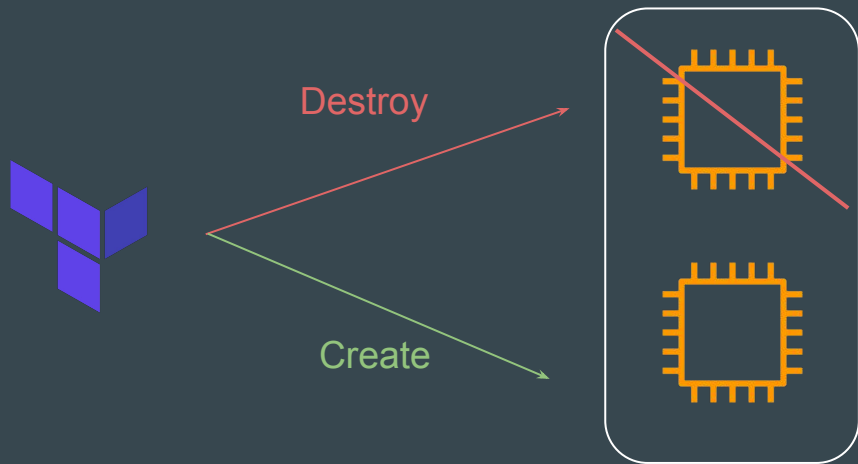
Points to Note

Similar kind of functionality was achieved using terraform taint command in older versions of Terraform.

For Terraform v0.15.2 and later, HashiCorp recommend using the -replace option with terraform apply

Splat Expression

Terraform Expressions

Overview of Spalat Expression

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

```
resource "aws_iam_user" "lb" {
  name = "iamuser.${count.index}"
  count = 3
  path = "/system/"
}

output "arns" {
  value = aws_iam_user.lb[*].arn
}
```

Terraform Graph

Terraform In Detail

Overview of 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.
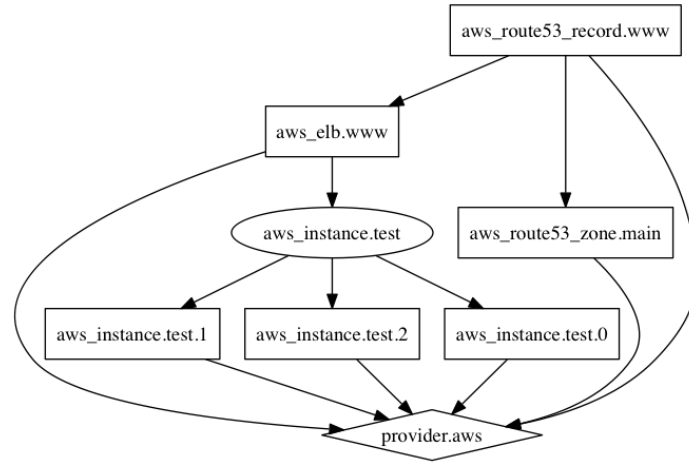
Saving Terraform Plan to a File

Terraform In Detail

Terraform Plan 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

Terraform in detail

Terraform Output

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

```
C:\Users\Zeal Vora\Desktop\terraform\terraform output>terraform output iam_names
[
  "iamuser.0",
  "iamuser.1",
  "iamuser.2",
]
```

Terraform Settings

Terraform in detail

Overview of Terraform Settings

The special terraform configuration block type is used to configure some behaviors of Terraform itself, such as requiring a minimum Terraform version to apply your configuration.

Terraform settings are gathered together into terraform blocks:

```
terraform {
  # ...
}
```

Setting 1 - Terraform Version

The required_version setting accepts a version constraint string, which specifies which versions of Terraform can be used with your configuration.

If the running version of Terraform doesn't match the constraints specified, Terraform will produce an error and exit without taking any further actions.

```
terraform {
    required_version = "> 0.12.0"
}
```

Setting 2 - Provider Version

The required_providers block specifies all of the providers required by the current module, mapping each local provider name to a source address and a version constraint.

```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
    }
  }
}
```

Dealing with Larger Infrastructure

Terraform in detail

Challenges with Larger Infrastructure

When you have a larger infrastructure, you will face issue related to API limits for a provider.

terraform plan

infra.tf
- 5 EC2
- 3 RDS
- 100 SG Rules
- VPC Infra

Update state of each resource.

aws
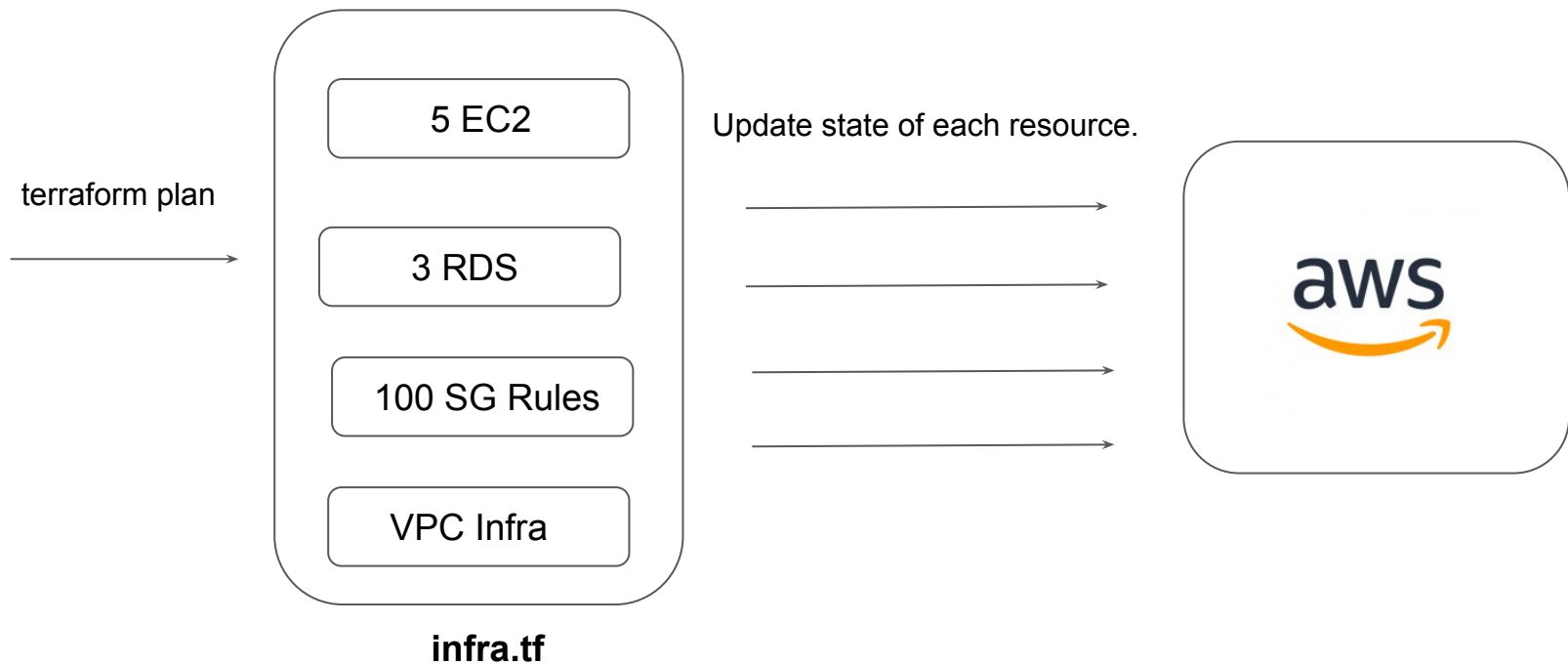
Dealing With Larger Infrastructure

Switch to smaller configuration were each can be applied independently.

5 EC2

3 RDS

100 SG Rules

VPC Infra

infra.tf

terraform plan

5 EC2   ec2.tf

3 RDS   rds.tf

100 SG Rules   sg.tf

VPC Infra   vpc.tf
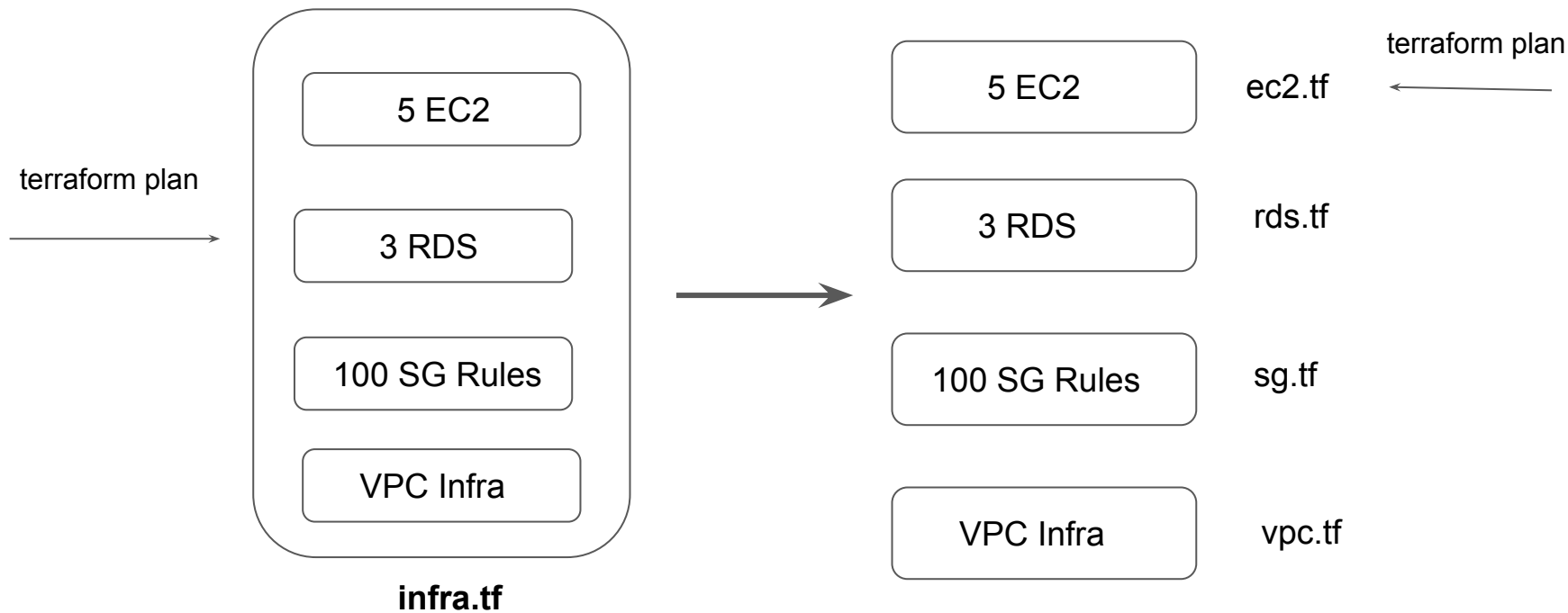
terraform plan

Slow Down, My Man

We can prevent terraform from querying the current state during operations like terraform plan.

This can be achieved with the -refresh=false flag

Specify the Target

The **-target=resource** flag can be used to target a specific resource.

Generally used as a means to operate on isolated portions of very large configurations

terraform plan -target=ec2

```
┌─────────────────┐
│  ┌───────────┐  │
│  │   5 EC2   │  │
│  └───────────┘  │
│                 │
│  ┌───────────┐  │
│  │   3 RDS   │  │
│  └───────────┘  │
│                 │
│  ┌───────────┐  │
│  │100 SG Rules│ │
│  └───────────┘  │
│                 │
│  ┌───────────┐  │
│  │ VPC Infra │  │
│  └───────────┘  │
└─────────────────┘
```

Zipmap

Terraform Function

Overview of Zipmap

The zipmap function constructs a map from a list of keys and a corresponding list of values.
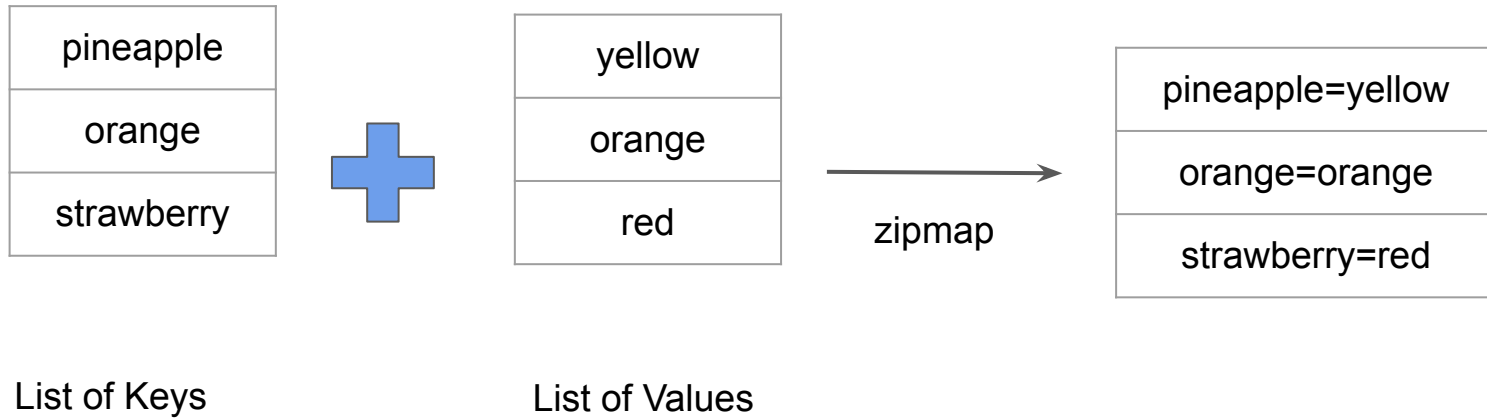
Sample Output of Zipmap Function

```
←[J> zipmap(["pineapple","oranges","strawberry"], ["yellow","orange","red"])
{
  "oranges" = "orange"
  "pineapple" = "yellow"
  "strawberry" = "red"
}
```

Simple Use-Case

You are creating multiple IAM users.

You need output which contains direct mapping of IAM names and ARNs

```
zipmap = {
  "demo-user.0" = "arn:aws:iam::018721151861:user/system/demo-user.0"
  "demo-user.1" = "arn:aws:iam::018721151861:user/system/demo-user.1"
  "demo-user.2" = "arn:aws:iam::018721151861:user/system/demo-user.2"
}
```

Comments in Terraform Code

Commenting the Code!

Overview of Comments

A comment is a text note added to source code to provide explanatory information, usually about the function of the code

```python
'''In this program, we check if the number is positive or
negative or zero and
display an appropriate message'''

num = 3.4

# Try these two variations as well:
# num = 0
# num = -4.5

if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

Comments in Terraform

The Terraform language supports three different syntaxes for comments:

Type	Description
#	begins a single-line comment, ending at the end of the line.
//	also begins a single-line comment, as an alternative to #.
/* and */	are start and end delimiters for a comment that might span over multiple lines.

Resource Behavior and Meta-Argument

Understanding the Basics

A resource block declares that you want a particular infrastructure object to exist with the given settings

```
resource "aws_instance" "myec2" {
    ami = "ami-00c39f71452c08778"
    instance_type = "t2.micro"
}
```

How Terraform Applies a Configuration

Create resources that exist in the configuration but are not associated with a real infrastructure object in the state.

Destroy resources that exist in the state but no longer exist in the configuration.

Update in-place resources whose arguments have changed.

Destroy and re-create resources whose arguments have changed but which cannot be updated in-place due to remote API limitations.

Understanding the Limitations

What happens if we want to change the default behavior?

Example: Some modification happened in Real Infrastructure object that is not part of Terraform but you want to ignore those changes during terraform apply.

```
resource "aws_instance" "web" {
  ami           = "ami-00c39f71452c08778"
  instance_type = "t3.micro"

  tags = {
    Name = "HelloWorld"
  }
}
```
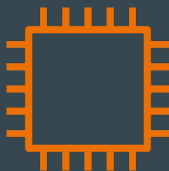
Name	HelloWorld

Env	Production

Solution - Using Meta Arguments

Terraform allows us to include meta-argument within the resource block which allows some details of this standard resource behavior to be customized on a per-resource basis.

Inside resource block

```
resource "aws_instance" "myec2" {
    ami = "ami-00c39f71452c08778"
    instance_type = "t2.micro"

    lifecycle {
      ignore_changes = [tags]
    }
}
```

Different Meta-Arguments

Meta-Argument	Description
depends_on	Handle hidden resource or module dependencies that Terraform cannot automatically infer.
count	Accepts a whole number, and creates that many instances of the resource
for_each	Accepts a map or a set of strings, and creates an instance for each item in that map or set.
lifecycle	Allows modification of the resource lifecycle.
provider	Specifies which provider configuration to use for a resource, overriding Terraform's default behavior of selecting one based on the resource type name

Meta Argument - LifeCycle

Basics of Lifecycle Meta-Argument

Some details of the default resource behavior can be customized using the special nested lifecycle block within a resource block body:

```
resource "aws_instance" "myec2" {
    ami = "ami-0f34c5ae932e6f0e4"
    instance_type = "t2.micro"

    tags = {
        Name = "HelloEarth"
    }

    lifecycle {
        ignore_changes = [tags]
    }
}
```

Arguments Available

There are four argument available within lifecycle block.

Arguments	Description
create_before_destroy	New replacement object is created first, and the prior object is destroyed after the replacement is created.
prevent_destroy	Terraform to reject with an error any plan that would destroy the infrastructure object associated with the resource
ignore_changes	Ignore certain changes to the live resource that does not match the configuration.
replace_triggered_by	Replaces the resource when any of the referenced items change

Replace Triggered By

Replaces the resource when any of the referenced items change.

```
resource "aws_appautoscaling_target" "ecs_target" {
  # ...

  lifecycle {
    replace_triggered_by = [
      # Replace `aws_appautoscaling_target` each time this instance of
      # the `aws_ecs_service` is replaced.
      aws_ecs_service.svc.id
    ]
  }
}
```
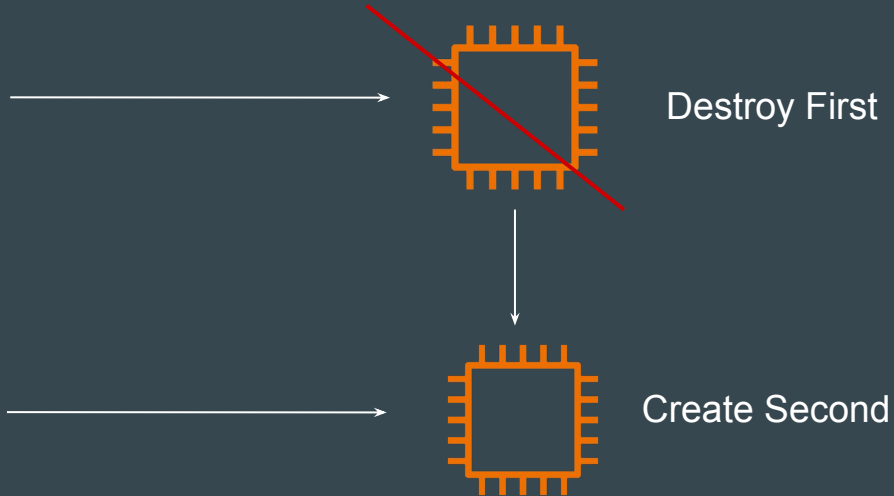
Create Before Destroy Argument

Understanding the Default Behavior

By default, when Terraform must change a resource argument that cannot be updated in-place due to remote API limitations, Terraform will instead destroy the existing object and then create a new replacement object with the new configured arguments.


```
demo.tf > ...
    resource "aws_instance" "myec2" {
        ami = "ami-00c39f71452c08778"
        instance_type = "t2.micro"
    }
```

Changed AMI

```
resource "aws_instance" "myec2" {
  ami           = "ami-123456789"
  instance_type = "t2.micro"
}
```
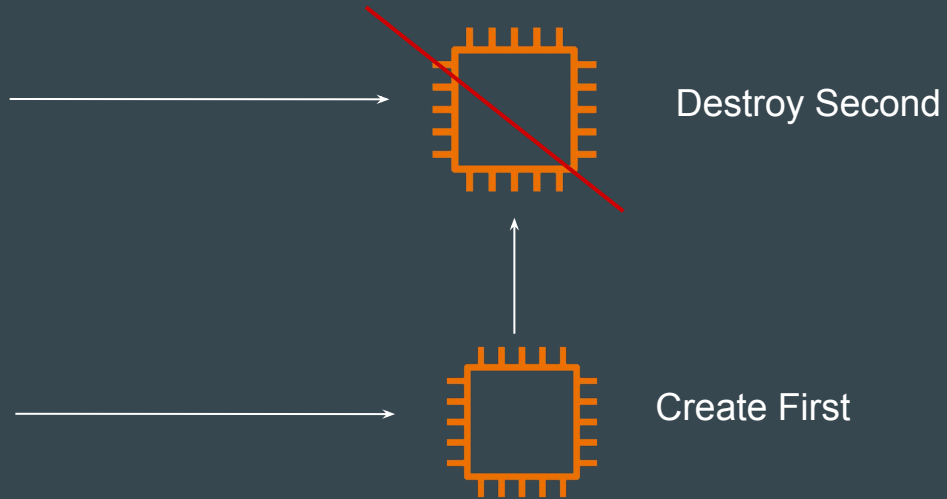
Destroy First

Create Second

Create Before Destroy Argument

The create_before_destroy meta-argument changes this behavior so that the new replacement object is created first, and the prior object is destroyed after the replacement is created.

Join us in our Adventure

Be Awesome

kplabs.in/chat

kplabs.in/linkedin

LifeCycle - Prevent Destroy Argument

Prevent Destroy Argument

This meta-argument, when set to true, will cause Terraform to reject with an error any plan that would destroy the infrastructure object associated with the resource, as long as the argument remains present in the configuration.
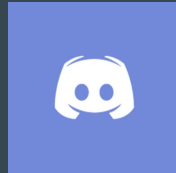
```
resource "aws_instance" "myec2" {
  ami           = "ami-123456789"
  instance_type = "t2.micro"

  lifecycle {
    prevent_destroy = true
  }
}
```

Points to Note

This can be used as a measure of safety against the accidental replacement of objects that may be costly to reproduce, such as database instances.

Since this argument must be present in configuration for the protection to apply, note that this setting does not prevent the remote object from being destroyed if the resource block were removed from configuration entirely.

LifeCycle - Ignore Changes Argument

Ignore Changes

In cases where settings of a remote object is modified by processes outside of Terraform, the Terraform would attempt to "fix" on the next run.

In order to change this behavior and ignore the manually applied change, we can make use of ignore_changes argument under lifecycle.

```
resource "aws_instance" "myec2" {
    ami = "ami-00c39f71452c08778"
    instance_type = "t2.micro"

    lifecycle {
      ignore_changes = [tags]
  }
}
```

Points to Note

Instead of a list, the special keyword all may be used to instruct Terraform to ignore all attributes, which means that Terraform can create and destroy the remote object but will never propose updates to it.

```
resource "aws_instance" "myec2" {
    ami = "ami-0f34c5ae932e6f0e4"
    instance_type = "t2.micro"

    tags = {
        Name = "HelloEarth"
    }

    lifecycle {
        ignore_changes = all
    }
}
```

Challenges with Count

Meta-Argument

Revising the Basics

Resource are identified by the index value from the list.

```
variable "iam_names" {
  type = list
  default = ["user-01","user-02","user-03"]
}

resource "aws_iam_user" "iam" {
  name = var.iam_names[count.index]
  count = 3
  path = "/system/"
}
```

Resource Address	Infrastructure
aws_iam_user.iam[0]	user-01
aws_iam_user.iam[1]	user-02
aws_iam_user.iam[2]	user-03

Challenge - 1

If the order of elements of index is changed, this can impact all of the other resources.

```
variable "iam_names" {
  type = list
  default = ["user-0","user-01","user-02","user-03"]
}

resource "aws_iam_user" "iam" {
  name = var.iam_names[count.index]
  count = 4
  path = "/system/"
}
```
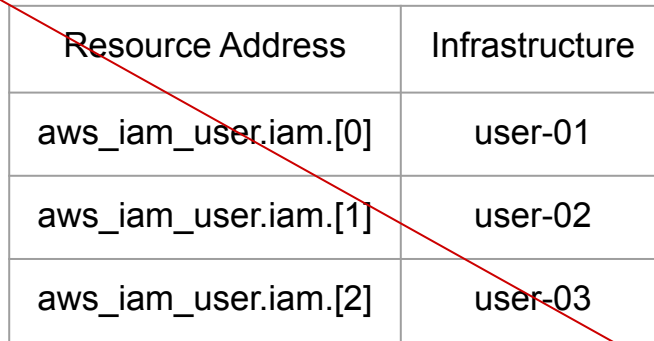
Resource Address	Infrastructure
aws_iam_user.iam.[0]	user-01
aws_iam_user.iam.[1]	user-02
aws_iam_user.iam.[2]	user-03

Important Note

If your resources are almost identical, count is appropriate.

If distinctive values are needed in the arguments, usage of for_each is recommended.

```
resource "aws_instance" "server" {
  count = 4 # create four similar EC2 instances
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

Data Type - SET

Let's Revise Programming

Basics of List

- Lists are used to store multiple items in a single variable.
- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0], the second item has index [1] etc.

```
variable "iam_names" {
  type = list
  default = ["user-01","user-02","user-03"]
}
```

Understanding SET

- SET is used to store multiple items in a single variable.

- SET items are unordered and no duplicates allowed.

Allowed

```
demoset = {"apple", "banana", "mango"}
```

Not-Allowed

```
demoset = {"apple", "banana", "mango", "apple"}
```

toset Function

toset function will convert the list of values to SET

```
> toset(["a", "b", "c","a"])
toset([
  "a",
  "b",
  "c",
])
```

for_each

Meta-Argument

Basics of For Each

for_each makes use of map/set as an index value of the created resource.

```
resource "aws_iam_user" "iam" {
  for_each = toset( ["user-01","user-02", "user-03"] )
  name     = each.key
}
```

Resource Address	Infrastructure
aws_iam_user.iam[user-01]	user-01
aws_iam_user.iam[user-02]	user-02
aws_iam_user.iam[user-03]	user-03

Replication Count Challenge

If a new element is added, it will not affect the other resources.

```
resource "aws_iam_user" "iam" {
  for_each = toset( ["user-0","user-01","user-02", "user-03"] )
  name     = each.key
}
```

Resource Address	Infrastructure
aws_iam_user.iam[user-01]	user-01
aws_iam_user.iam[user-02]	user-02
aws_iam_user.iam[user-03]	user-03
aws_iam_user.iam[user-0]	user-0

The each object

In blocks where for_each is set, an additional each object is available.

This object has two attributes:

Each object	Description
each.key	The map key (or set member) corresponding to this instance.
each.value	The map value corresponding to this instance

Relax and Have a Meme Before Proceeding

Provisioners

Interesting Part is here

Provisioners are interesting

Till now we have been working only on 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 ?

Welcome to 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:

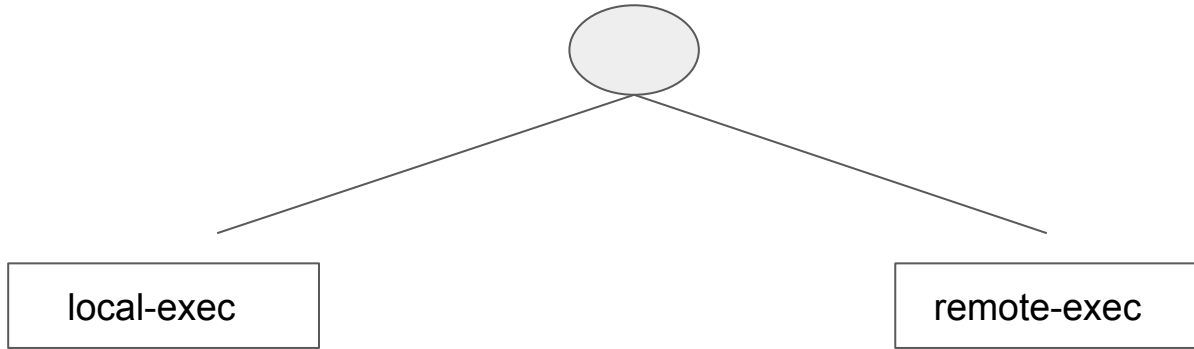On creation of Web-Server, execute a script which installs Nginx web-server.

Types of Provisioners

Interesting Part is here

Provisioners are interesting

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

There are two main types of provisioners:

| local-exec | remote-exec |

Local Exec Provisioners

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

Let's take an example:

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



  provisioner "local-exec" {
    command = "echo ${aws_instance.web.private_ip} >> private_ips.txt"
  }
}
```

Remote Exec Provisioners

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

Let's take an example:

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



  provisioner "remote-exec" {

                    ……………………………..

  }
}
```

Provisioner Types

Terraform in detail

Overview of Provisioner Types

There are two primary types of provisioners:

Types of Provisioners	Description
Creation-Time Provisioner	Creation-time provisioners are only run during creation, not during updating or any other lifecycle If a creation-time provisioner fails, the resource is marked as tainted.
Destroy-Time Provisioner	Destroy provisioners are run before the resource is destroyed.

Destroy Time Provisioner

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

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

  provisioner "local-exec" {
    when    = destroy
    command = "echo 'Destroy-time provisioner'"
  }
}
```

local-exec

Provisioners Time!

Provisioners are interesting

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

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

```
provisioner "local-exec" {
    command = "echo ${aws_instance.web.private_ip} >> private_ips.txt"
  }
```

Failure Behavior - Provisioners

Terraform in detail

Provisioner - 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:

Allowed Values	Description
continue	Ignore the error and continue with creation or destruction.
fail	Raise an error and stop applying (the default behavior). If this is a creation provisioner, taint the resource.

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

  provisioner "local-exec" {
    command    = "echo The server's IP address is ${self.private_ip}"
    on_failure = continue
  }
}
```
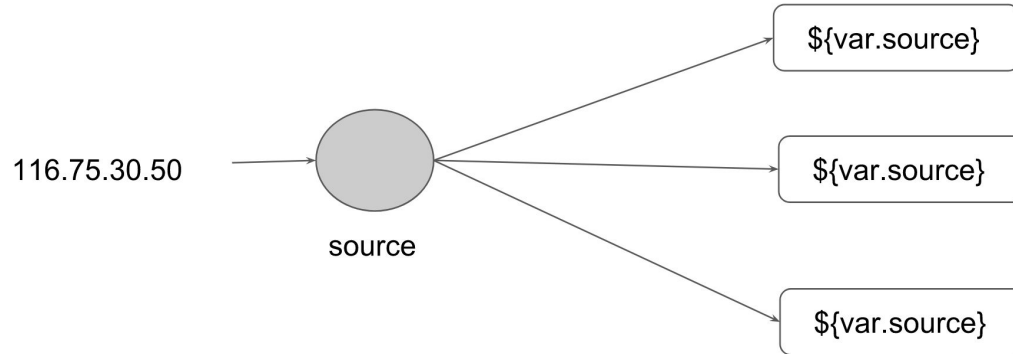
DRY Principle

Software Engineering

Understanding DRY Approach

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 single source of information.

We are repeating resource code

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"]

}
```

Centralized Structure

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

```
resource "aws_instance" "myweb" {

    ami = "ami-bf5540df"

    instance_type = "t2.micro"

    security_groups = ["default"]

}
```
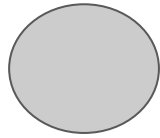
source

module "source"

Challenges with Modules

Software Engineering

Challenges

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

| Staging |

instance_type = t2.micro

| Production |

instance_type = m4.large

Challenges

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

source

resource "aws_instance" "myweb" {

   ami = "ami-bf5540df"

  instance_type = "t2.micro"

  security_groups = ["default"]

}

Development — t2.micro

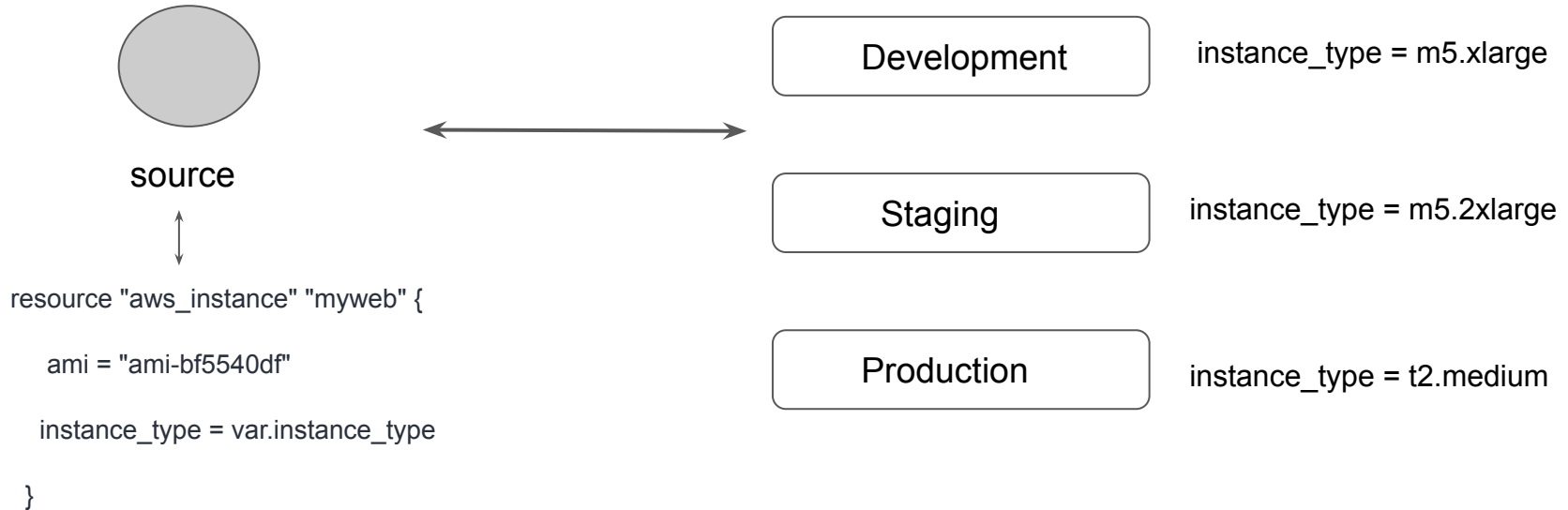Staging — t2.small

Production — m4.large

Using Locals with Modules

Terraform Function

Understanding the Challenge

Using variables in Modules can also allow users to override the values which you might not want.

source

resource "aws_instance" "myweb" {

  ami = "ami-bf5540df"

  instance_type = var.instance_type

}

Development — instance_type = m5.xlarge

Staging — instance_type = m5.2xlarge

Production — instance_type = t2.medium

Setting the Context

There can be many repetitive values in modules and this can make your code difficult to maintain.

You can centralize these using variables but users will be able to override it.

```
resource "aws_security_group" "elb-sg" {
  name        = "myelb-sg"


  ingress {
    description       = "Allow Inbound from Secret Application"
    from_port         = 8443
    to_port           = 8443
    protocol          = "tcp"
    cidr_blocks       = ["0.0.0.0/0"]
  }
}
```

```
resource "aws_security_group" "elb-sg" {
  name        = "myelb-sg"


  ingress {
    description       = "Allow Inbound from Secret Application"
    from_port         = var.app_port
    to_port           = var.app_port
    protocol          = "tcp"
    cidr_blocks       = ["0.0.0.0/0"]
  }
}
```

Hardcoded Port   Variable Port

Using Locals

Instead of variables, you can make use of locals to assign the values.

You can centralize these using variables but users will be able to override it.

```
resource "aws_security_group" "ec2-sg" {
  name         = "myec2-sg"

  ingress {
    description      = "Allow Inbound from Secret Application"
    from_port        = local.app_port
    to_port          = local.app_port
    protocol         = "tcp"
    cidr_blocks      = ["0.0.0.0/0"]
  }

locals {
  app_port = 8443
}
```

Module Outputs

Output the Data

Revising Output Values

Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use.

```
output "instance_ip_addr" {
  value = aws_instance.server.private_ip
}
```

Accessing Child Module Outputs

In a parent module, outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>

```
resource "aws_security_group" "ec2-sg" {
  name        = "myec2-sg"

  ingress {
    description     = "Allow Inbound from Secret Application"
    from_port       = 8433
    to_port         = 8433
    protocol        = "tcp"
    cidr_blocks     = ["0.0.0.0/0"]
  }

}

output "sg_id" {
  value = aws_security_group.ec2-sg.arn
}
```

```
module "sgmodule" {
  source = "../../modules/sg"
}


resource "aws_instance" "web" {
  ami             = "ami-0ca285d4c2cda3300"
  instance_type = "t3.micro"
  vpc_security_group_ids = [module.sgmodule.sg_id]
}
```
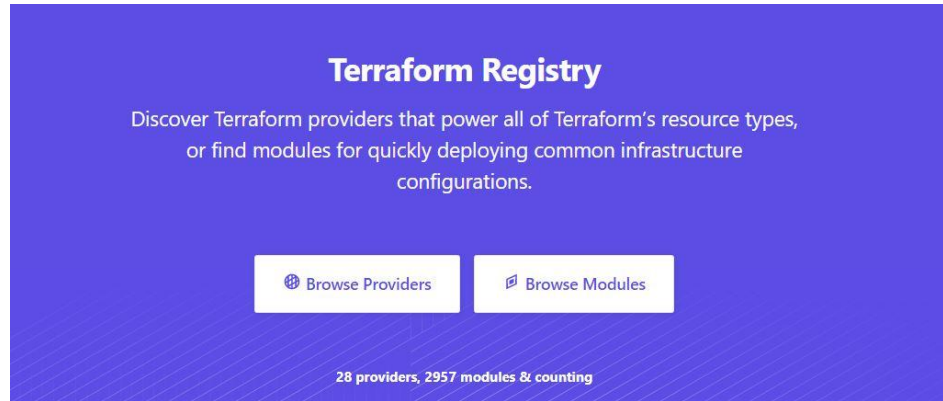
Terraform Registry

Terraform in detail

Overview of 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

Terraform Registry

Discover Terraform providers that power all of Terraform's resource types, or find modules for quickly deploying common infrastructure configurations.

⊕ Browse Providers   ⬧ Browse Modules

28 providers, 2957 modules & counting

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 in Terraform Registry

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.
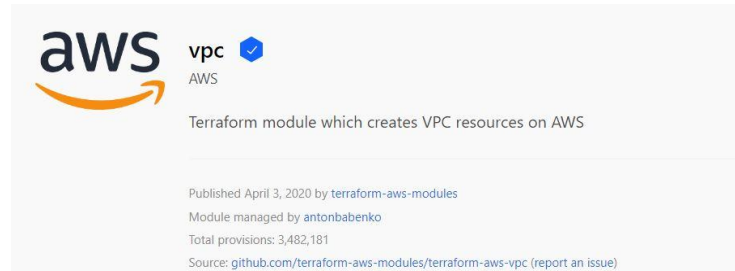
Using Registry Module in Terraform

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
}
```
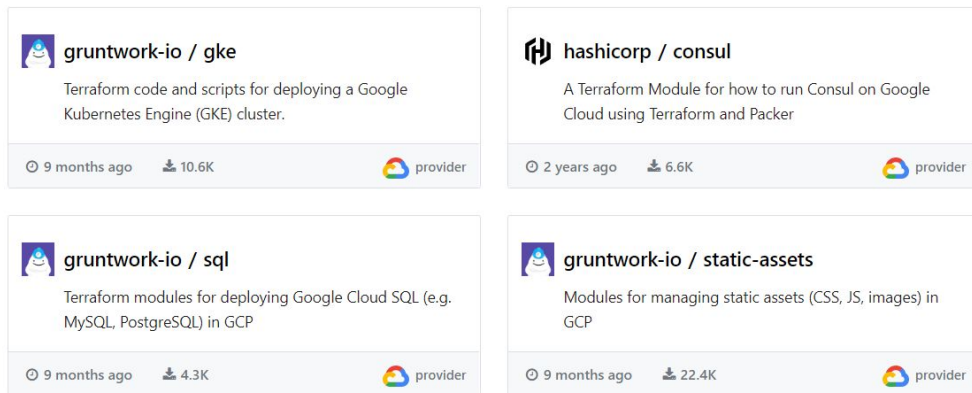
Publishing Modules

Publish Modules to Terraform Registry

Overview of Publishing Modules

Anyone can publish and share modules on the Terraform Registry.

Published modules support versioning, automatically generate documentation, allow browsing version histories, show examples and READMEs, and more.

Requirements for Publishing Module

Requirement	Description
GitHub	The module must be on GitHub and must be a public repo. This is only a requirement for the public registry.
Named	Module repositories must use this three-part name format terraform-<PROVIDER>-<NAME>
Repository description	The GitHub repository description is used to populate the short description of the module.
Standard module structure	The module must adhere to the standard module structure.
x.y.z tags for releases	The registry uses tags to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a v. For example, v1.0.4 and 0.9.2

Standard Module Structure

The standard module structure is a file and directory layout that is recommend for reusable modules distributed in separate repositories

```
$ tree minimal-module/

.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
```

```
$ tree complete-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── ...
├── modules/
│   ├── nestedA/
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   ├── nestedB/
│   ├── .../
├── examples/
│   ├── exampleA/
│   │   ├── main.tf
│   ├── exampleB/
│   ├── .../
```

Terraform Workspace

Interesting topics

Understanding WorkSpaces

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

Staging   →   instance_type = t2.micro

Production   →   instance_type = m4.large

Project A
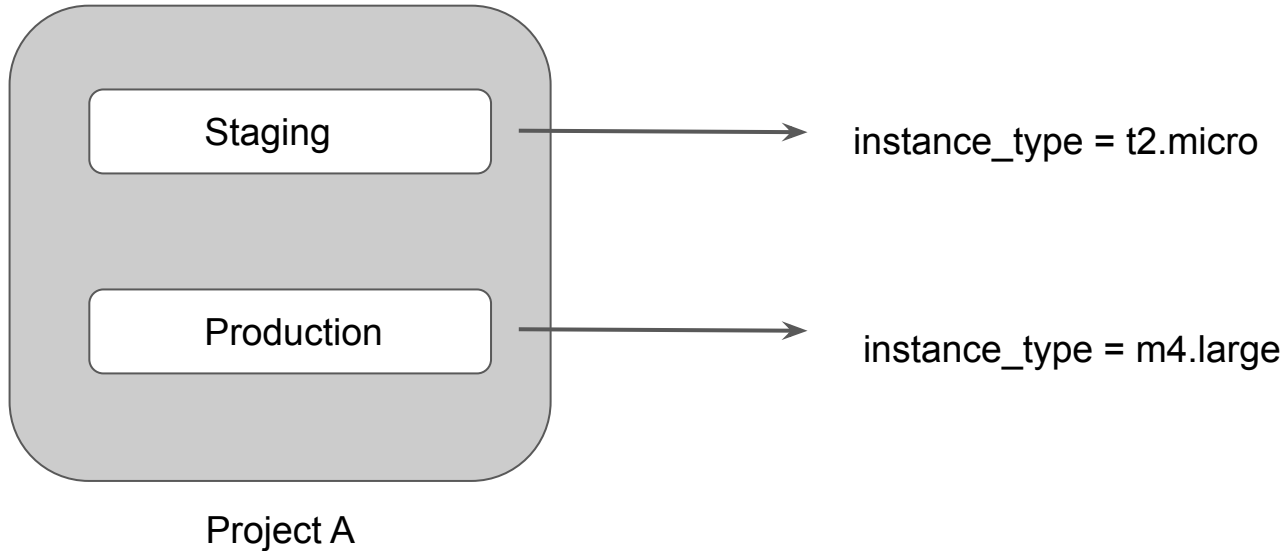
Team Collaboration

Terraform in detail

Local Changes are not always good

Currently we have been working with terraform code locally.

Terraform Code

………
………

Centralized Management

Central Repository

Terraform Code

....
....

Terraform Code

....
....
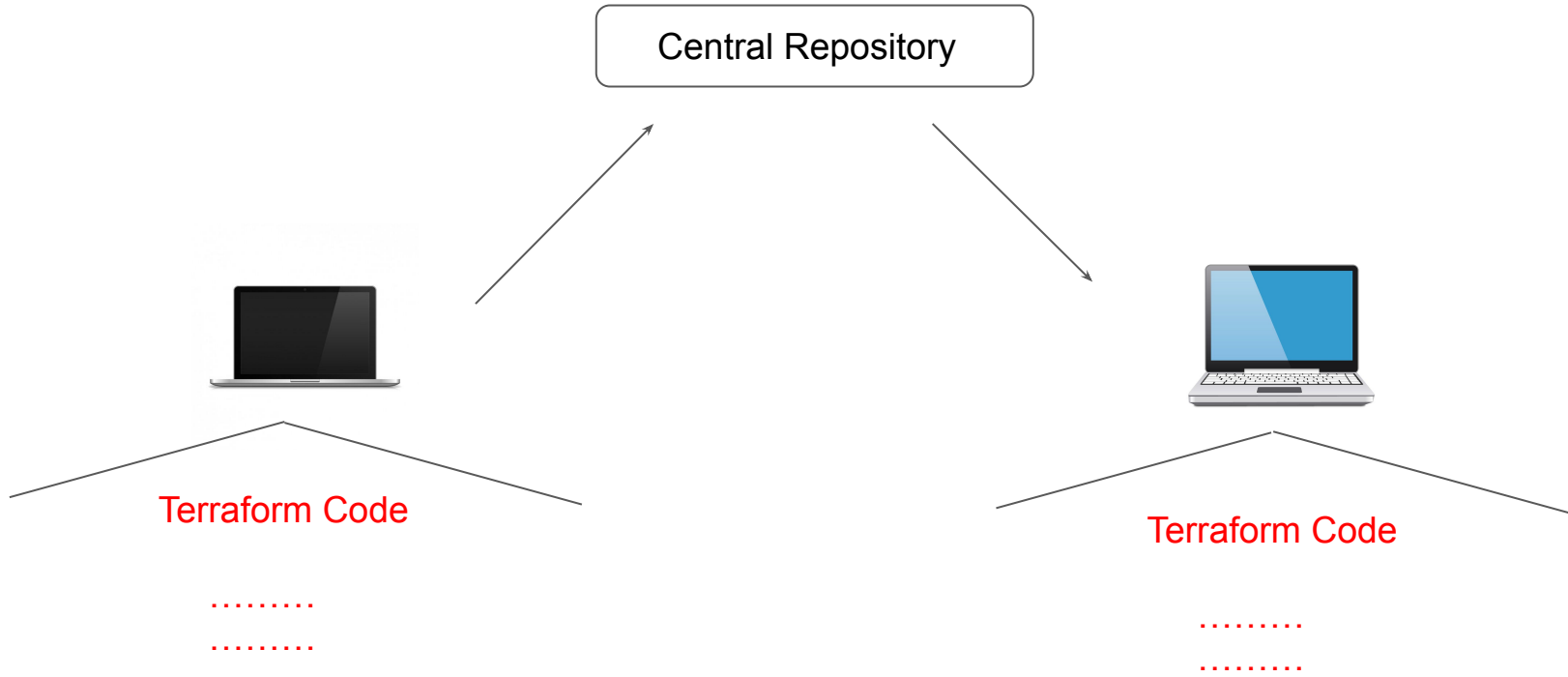
Relax and Have a Meme Before Proceeding

me: i'll do it at 6

time: 6:05

me: wow looks like i gotta wait til 7 now

Terraform Module Sources

Terraform in detail

Supported Module Sources

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

```
module "consul" {
  source = "app.terraform.io/example-corp/k8s-cluster/azurerm"
  version = "1.1.0"
}
```

Local Path

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

```
module "consul" {
  source = "../consul"
}
```

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.

.

```
module "vpc" {
  source = "git::https://example.com/vpc.git"
}


module "storage" {
  source = "git::ssh://username@example.com/storage.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:

```
module "vpc" {
  source = "git::https://example.com/vpc.git?ref=v1.2.0"
}
```

.
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

Terraform in detail

Overview of gitignore

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

.gitignore
conf/
*.artifacts
credentials

Terraform and .gitignore

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

Files to Ignore	Description
.terraform	This file will be recreated when terraform init is run.
terraform.tfvars	Likely to contain sensitive data like usernames/passwords and secrets.
terraform.tfstate	Should be stored in the remote side.
crash.log	If terraform crashes, the logs are stored to a file named crash.log

Terraform Backend

Terraform in detail

Basics of Backends

Backends primarily determine where Terraform stores its state.

By default, Terraform implicitly uses a backend called local to store state as a local file on disk.

demo.tf

```
provider "vault" {
  address = "http://127.0.0.1:8200"
}

data "vault_generic_secret" "demo" {
  path = "secret/db_creds"
}

output "vault_secrets" {
  value = data.vault_generic_secret.demo.data_json
  sensitive = "true"
}
```
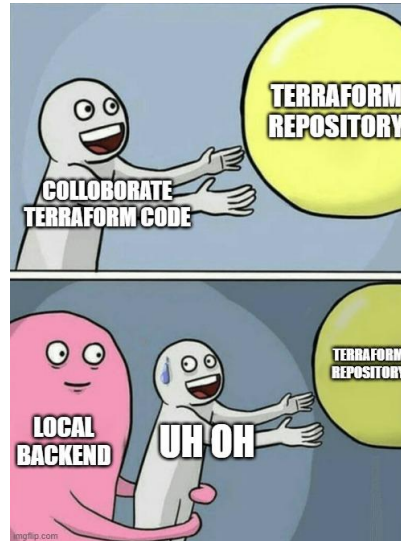
```
terraform.tfstate
1  {
2    "version": 4,
3    "terraform_version": "1.1.9",
4    "serial": 1,
5    "lineage": "f7ba581a-ab47-b03e-2e54-e683a2dc4ba2",
6    "outputs": {
7      "vault_secrets": {
8        "value": "{\"admin\":\"password123\"}",
9        "type": "string",
10       "sensitive": true
11     }
12   },
13   "resources": [
14     {
15       "mode": "data",
16       "type": "vault_generic_secret",
17       "name": "demo",
18       "provider": "provider[\"registry.terraform.io/hashicorp/vault\"]",
19       "instances": [
```

terraform.tfstate

Challenge with Local Backend

Nowadays Terraform project is handled and collaborated by an entire team.

Storing the state file in the local laptop will not allow collaboration.

Ideal Architecture

Following describes one of the recommended architectures:

1. The Terraform Code is stored in Git Repository.
2. The State file is stored in a Central backend.

Backends Supported in Terraform

Terraform supports multiple backends that allows remote service related operations.

Some of the popular backends include:

- S3
- Consul
- Azurerm
- Kubernetes
- HTTP
- ETCD

Important Note

Accessing state in a remote service generally requires some kind of access credentials

Some backends act like plain "remote disks" for state files; others support locking the state while operations are being performed, which helps prevent conflicts and inconsistencies.

Terraform User

Store State File

Authenticate First

S3 Bucket

State Locking

Let's Lock the State

Understanding State Lock

Whenever you are performing 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 can corrupt your state file.

```
C:\Users\Zeal Vora\Desktop\tf-demo\remote-backend>terraform plan

Error: Error acquiring the state lock

Error message: Failed to read state file: The state file could not be read: read terraform.tfstate: The process
cannot access the file because another process has locked a portion of the file.

Terraform acquires a state lock to protect the state from being written
by multiple users at the same time. Please resolve the issue above and try
again. For most commands, you can disable locking with the "-lock=false"
flag, but this is not recommended.
```

Basic Working
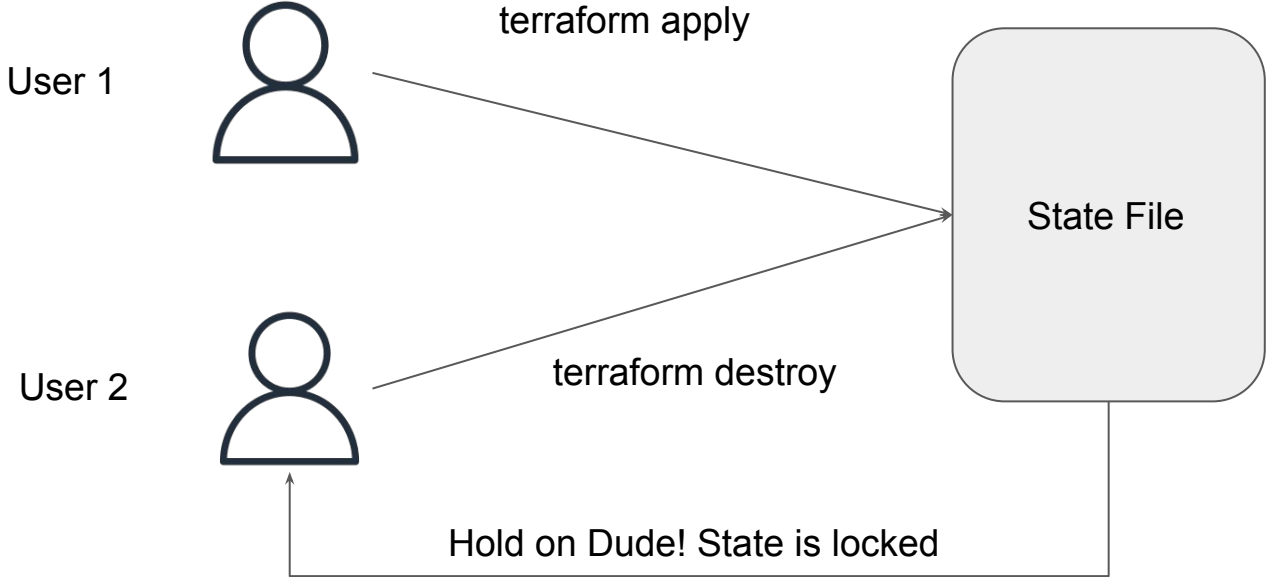
Important Note

State locking happens automatically on all operations that could write state. You won't see any message that it is happening

If state locking fails, Terraform will not continue

Not all backends support locking. The documentation for each backend includes details on whether it supports locking or not.

Force Unlocking State

Terraform has a force-unlock command to manually unlock the state if unlocking failed.

If you unlock the state when someone else is holding the lock it could cause multiple writers.

Force unlock should only be used to unlock your own lock in the situation where automatic unlocking failed.
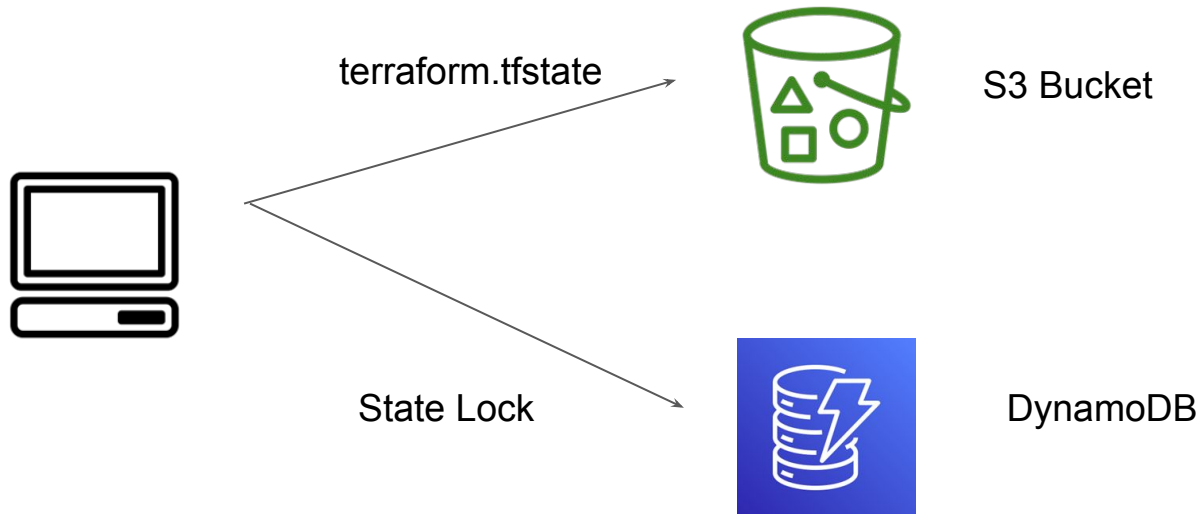
State Locking in S3 Backend

Back to Providers

State Locking in S3

By default, S3 does not support State Locking functionality.

You need to make use of DynamoDB table to achieve state locking functionality.

terraform.tfstate   S3 Bucket

State Lock   DynamoDB

Terraform State Management

Advanced State Management

Overview of State Modification

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.

Overview of State Modification

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

State Sub Command	Description
list	List resources within terraform state file.
mv	Moves item with terraform state.
pull	Manually download and output the state from remote state.
push	Manually upload a local state file to remote state.
rm	Remove items from the Terraform state
show	Show the attributes of a single resource in the state.

Sub Command - List

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

```
bash-4.2# terraform state list
aws_iam_user.lb
aws_instance.webapp
```

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 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.

```
bash-4.2# terraform state show aws_instance.webapp
# aws_instance.webapp:
resource "aws_instance" "webapp" {
    ami                          = "ami-082b5a644766e0e6f"
    arn                          = "arn:aws:ec2:us-west-2:018721151861:instance/i-0107ea9ed06c467e0"
    associate_public_ip_address  = true
    availability_zone            = "us-west-2b"
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
    disable_api_termination      = false
    ebs_optimized                = false
    get_password_data            = false
    id                           = "i-0107ea9ed06c467e0"
    instance_state               = "running"
    instance_type                = "t2.micro"
```

Connecting Remote States

Terraform in detail

Basics of Terraform Remote State

The terraform_remote_state data source retrieves the root module output values from some other Terraform configuration, using the latest state snapshot from the remote backend.
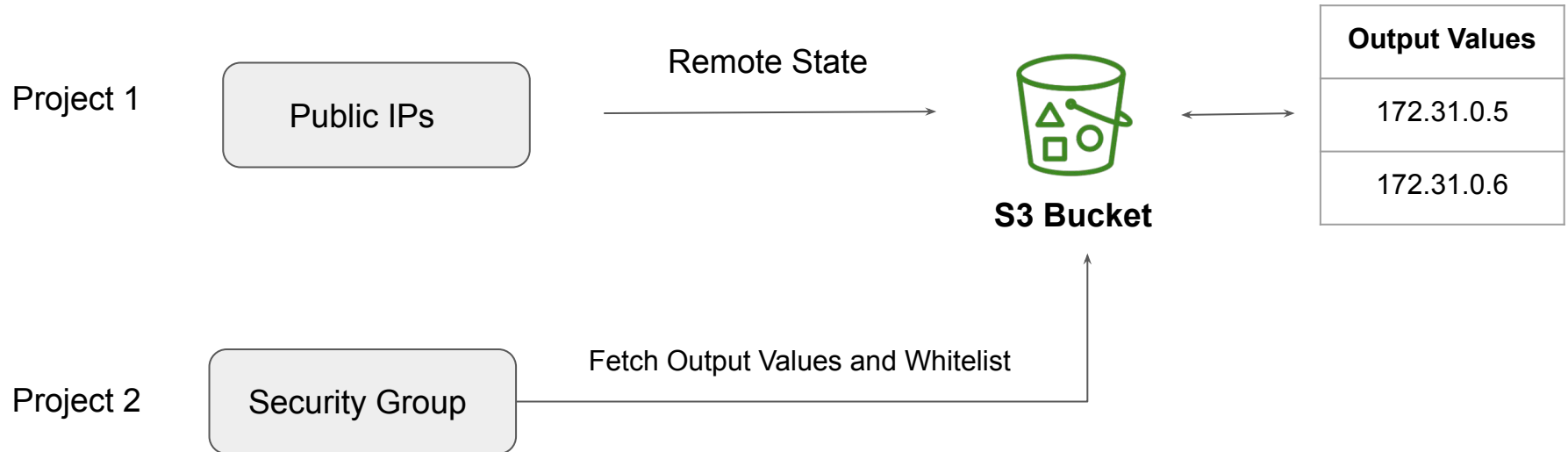
Step 1 - Create a Project with Output Values & S3 Backend

```
resource "aws_eip" "lb" {
  vpc        = true
}

output "eip_addr" {
  value = aws_eip.lb.public_ip
}
```

```
terraform {
  backend "s3" {
    bucket = "kplabs-terraform-backend"
    key    = "network/eip.tfstate"
    region = "us-east-1"
  }
}
```

Step 2 - Reference Output Values from Different Project

```
data "terraform_remote_state" "eip" {
  backend = "s3"
  config = {
    bucket = "kplabs-terraform-backend"
    key    = "network/eip.tfstate"
    region = "us-east-1"
  }
}
```

```
resource "aws_security_group" "allow_tls" {
  name        = "allow_tls"
  description = "Allow TLS inbound traffic"

  ingress {
    description      = "TLS from VPC"
    from_port        = 443
    to_port          = 443
    protocol         = "tcp"
    cidr_blocks      = ["${data.terraform_remote_state.eip.outputs.eip_addr}/32"]
  }

}
```

Terraform Import

Typical Challenge

It can happen that all the resources in an organization are created manually.

Organization now wants to start using Terraform and manage these resources via Terraform.
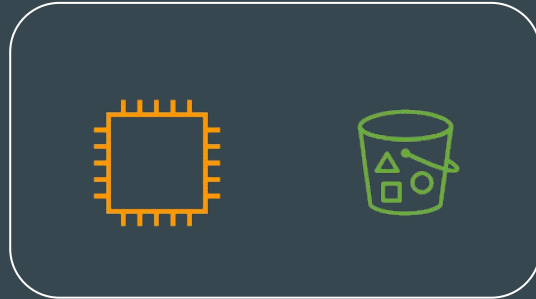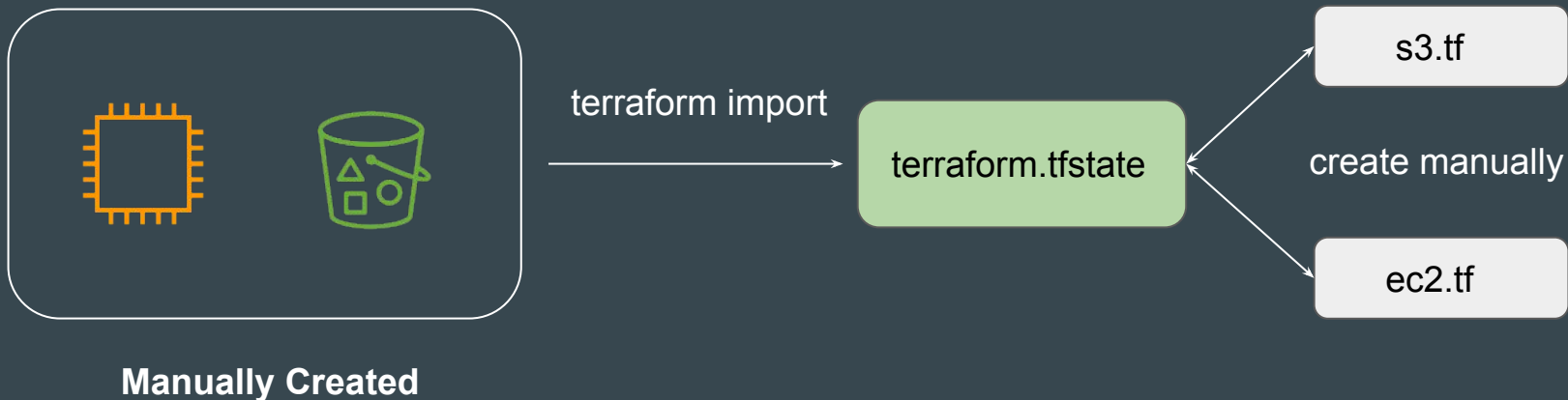

Manually Created

Earlier Approach

In the older approach, Terraform import would create the state file associated with the resources running in your environment.

Users still had to write the tf files from scratch.

Manually Created

terraform import

terraform.tfstate

s3.tf

create manually

ec2.tf

Newer Approach

In the newer approach, terraform import can automatically create the terraform configuration files for the resources you want to import.

Point to Note

Terraform 1.5 introduces automatic code generation for imported resources.

This dramatically reduces the amount of time you need to spend writing code to match the imported

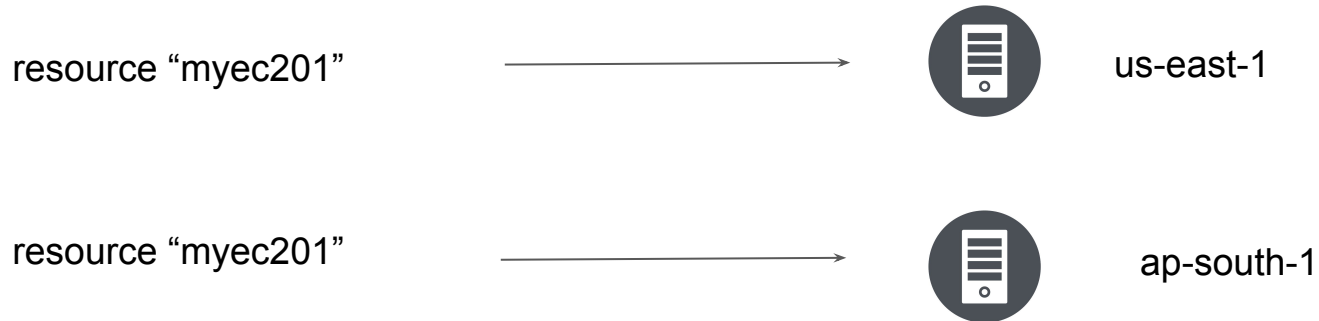This feature is not available in the older version of Terraform.

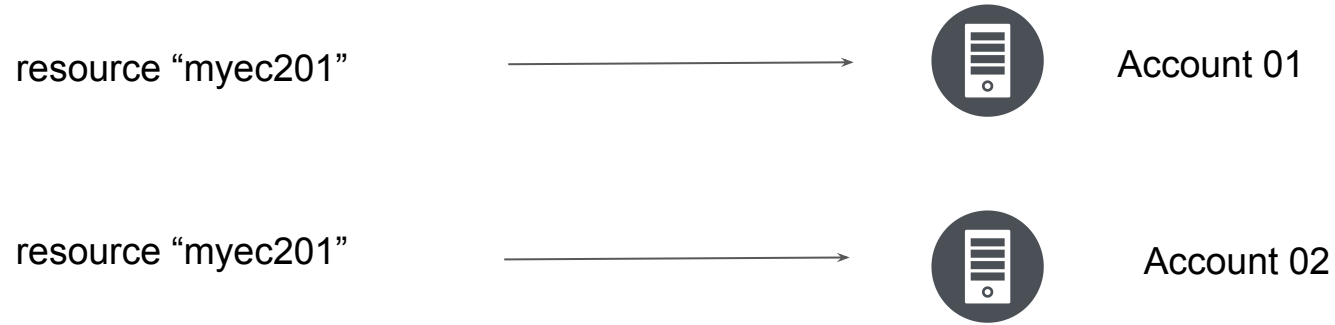# Provider Configuration

Terraform in detail

Single Provider Multiple 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.

resource "myec201" ⟶ us-east-1

resource "myec201" ⟶ ap-south-1

Single Provider Multiple Configuration

resource "myec201"   ⟶   🖥️ Account 01

resource "myec201"   ⟶   🖥️ Account 02

Terraform with STS

Terraform in detail

Definitive Use-Case
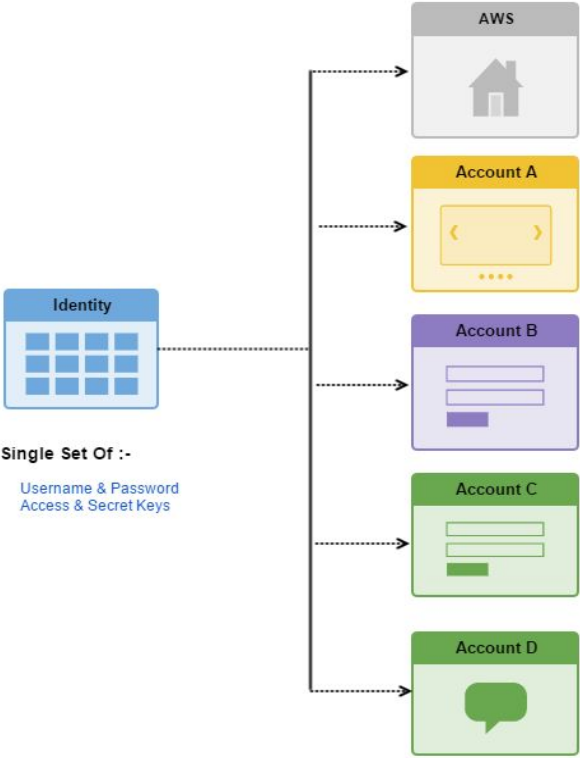
Sensitive Parameter

Terraform Security

Overview of Sensitive Parameter

With organization managing their 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

```
output "db_password" {
  value       = aws_db_instance.db.password
  description = "The password for logging in to the database.
  sensitive   = true
}
```

Overview of Sensitive Parameter

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.

```
C:\Users\Zeal Vora\Desktop\terraform\sensitive data>terraform apply

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

Outputs:

db_password = <sensitive>
```

Overview of Vault

HashiCorp Certified: Vault Associate

Let's get started

HashiCorp Vault allows organizations to securely store secrets like tokens, passwords, certificates along with access management for protecting secrets.

One of the common challenges nowadays in an organization is "Secrets Management"

Secrets can include, database passwords, AWS access/secret keys, API Tokens, encryption keys and others.
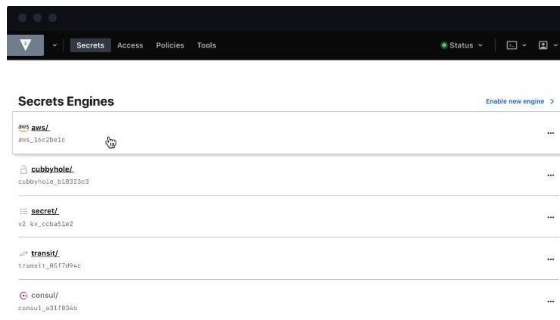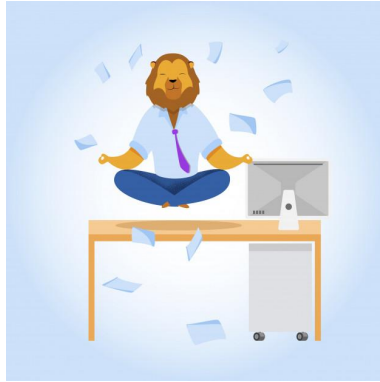
Dynamic Secrets
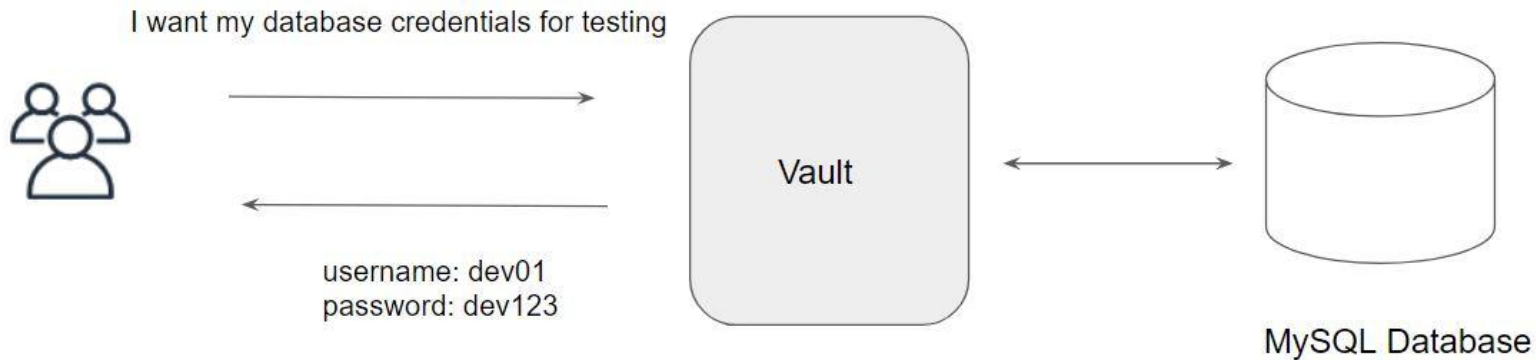
Life Becomes Easier

Once Vault is integrated with multiple backends, your life will become much easier and you can focus more on the right work.

Major aspect related to Access Management can be taken over by vault.

Vault Provider

Back to Providers

Vault Provider

The Vault provider allows Terraform to read from, write to, and configure HashiCorp Vault.


```
provider "vault" {
  address = "http://127.0.0.1:8200"
}

data "vault_generic_secret" "demo" {
  path = "secret/db-creds"
}
```
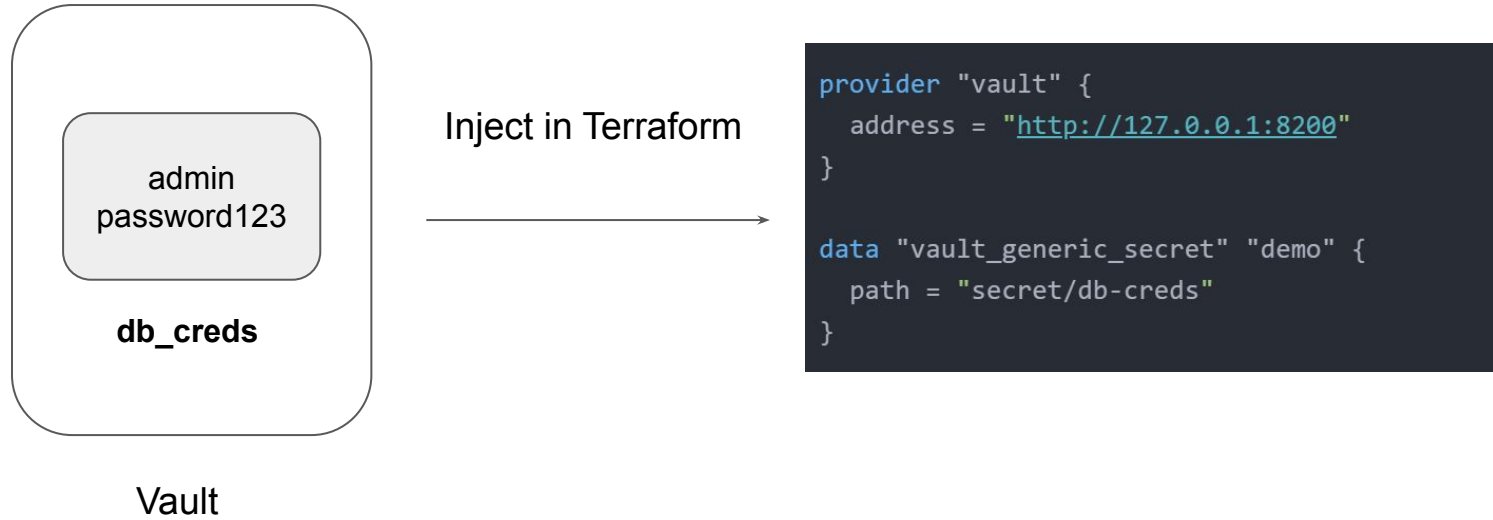
Important Note

Interacting with Vault from Terraform causes any secrets that you read and write to be persisted in both Terraform's state file.

Terraform Cloud

Terraform in detail

Overview of Terraform Cloud

Terraform Cloud manages Terraform runs in a consistent and reliable environment with various features like access controls, private registry for sharing modules, policy controls and others.

Sentinel

Terraform Cloud In Detail

Overview of the Sentinel

Sentinel is a policy-as-code framework integrated with the HashiCorp Enterprise products.

It enables fine-grained, logic-based policy decisions, and can be extended to use information from external sources.

Note: Sentinel policies are paid feature

terraform plan	→	sentinel checks	→	terraform apply

High Level Structure

```
┌─────────────┐      ┌─────────────┐      ┌─────────────┐
│             │      │             │      │             │
│   Policy    │─────▶│ Policy Sets │─────▶│  Workspace  │
│             │      │             │      │             │
└─────────────┘      └─────────────┘      └─────────────┘
       │
       ▼
┌─────────────┐
│             │
│             │
│ Block EC2 without tags │
│             │
│             │
└─────────────┘
```

Terraform Backend

Terraform in detail

Basics of Backends

Backends primarily determine where Terraform stores its state.

By default, Terraform implicitly uses a backend called local to store state as a local file on disk.

demo.tf

```
provider "vault" {
  address = "http://127.0.0.1:8200"
}

data "vault_generic_secret" "demo" {
  path = "secret/db_creds"
}

output "vault_secrets" {
  value = data.vault_generic_secret.demo.data_json
  sensitive = "true"
}
```

```
terraform.tfstate
1  {
2    "version": 4,
3    "terraform_version": "1.1.9",
4    "serial": 1,
5    "lineage": "f7ba581a-ab47-b03e-2e54-e683a2dc4ba2",
6    "outputs": {
7      "vault_secrets": {
8        "value": "{\"admin\":\"password123\"}",
9        "type": "string",
10       "sensitive": true
11     }
12   },
13   "resources": [
14     {
15       "mode": "data",
16       "type": "vault_generic_secret",
17       "name": "demo",
18       "provider": "provider[\"registry.terraform.io/hashicorp/vault\"]",
19       "instances": [
```

terraform.tfstate

Challenge with Local Backend

Nowadays Terraform project is handled and collaborated by an entire team.

Storing the state file in the local laptop will not allow collaboration.

Ideal Architecture

Following describes one of the recommended architectures:

1. The Terraform Code is stored in Git Repository.
2. The State file is stored in a Central backend.

TF files

Central Git Repo

Project Collaborators

terraform.tfstate

Central Backend

Backends Supported in Terraform

Terraform supports multiple backends that allows remote service related operations.

Some of the popular backends include:

- S3
- Consul
- Azurerm
- Kubernetes
- HTTP
- ETCD

Important Note

Accessing state in a remote service generally requires some kind of access credentials

Some backends act like plain "remote disks" for state files; others support locking the state while operations are being performed, which helps prevent conflicts and inconsistencies.

Store State File

Terraform User

Authenticate First

S3 Bucket

Air Gapped Environments

Installation Methods

Understanding Concept of Air Gap

An air gap is a network security measure employed to ensure that a secure computer network is physically isolated from unsecured networks, such as the public Internet.

Internet Gateway

Internal Router

Internet Connectivity
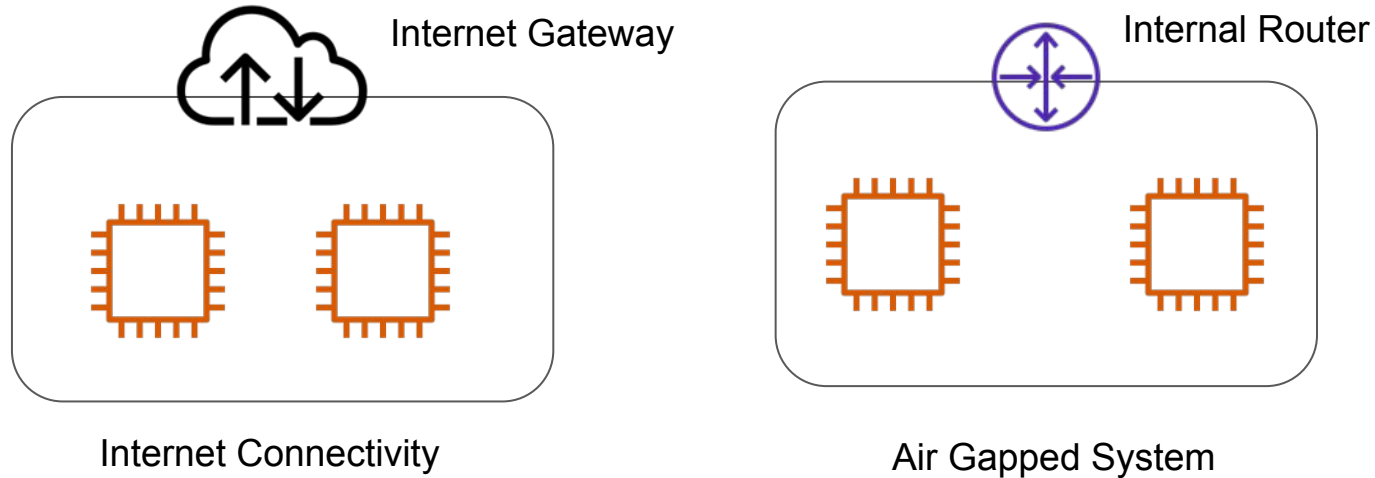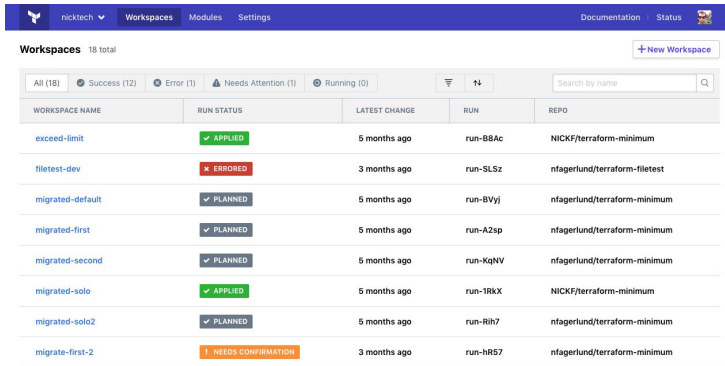
Air Gapped System

Usage of Air Gapped Systems

Air Gapped Environments are used in various areas. Some of these include:

- Military/governmental computer networks/systems

- Financial computer systems, such as stock exchanges

- Industrial control systems, such as SCADA in Oil & Gas fields

Terraform Enterprise Installation Methods

Terraform Enterprise installs using either an online or air gapped method and as the names infer, one requires internet connectivity, the other does not

Terraform Enterprise

Air Gap Install

Isolated Server

Choose your installation type

Online

Airgapped

Please choose an installation type to continue.

Continue »

Provide path or upload airgap bundle

Provide absolute path on this server to archive file

e.g. /mnt/installers/package.airgap

Continue »

Select file for upload

📂 Upload Airgap Bundle

To upload an app bundle, file must have a `.airgap` extension.

« Back

Relax and Have a Meme Before Proceeding

Overview of HashiCorp Exams

Let's Get Certified!

Overview of HashiCorp Associate Exams

Overview of the basic exam related information.

Assessment Type	Description
Type of Exams	Multiple Choice
Format	Online Proctored
Duration	1 hour
Questions	57
Price	70.50 USD + Taxes
Language	English
Expiration	2 years

Multiple Choice

This includes various sub-formats, including:

- True or False
- Multiple Choice
- Fill in the blank

Delta Type of Question

Example 1:

Demo Software stores information in which type of backend?

Format - Online Proctored

Important Rules to be followed:

- You are alone in the room
- Your desk and work area are clear
- You are connected to a power source
- No phones or headphones
- No dual monitors
- No leaving your seat
- No talking
- Webcam, speakers, and microphone must remain on throughout the test.
- The proctor must be able to see you for the duration of the test.

My Experience - Before Room

My Experience - After Room

My Experience - My Desk

Registration Process

The high-level steps for registering for the exams are as follows:

1. Login to the HashiCorp Certification Page.
2. Register for Exams.
3. Check System Requirements
4. Download PSI Software
5. Best of Luck & Good Luck!

Make sure to complete system check.

Registration Process

Registration Process

Registration Process

Registration Process

Photo ID Capture

Please take a picture of a current Photo ID.

- It must be a current, non-expired, government-issued identification.
- Military IDs are not currently accepted.

💡 Photo Tips

1. Position the FRONT of your ID card in the center of the frame so that all of the corners are visible.
2. Make sure that your photo is clear.
3. Make sure that all of the text can be read.
4. When you are ready, press the camera button.
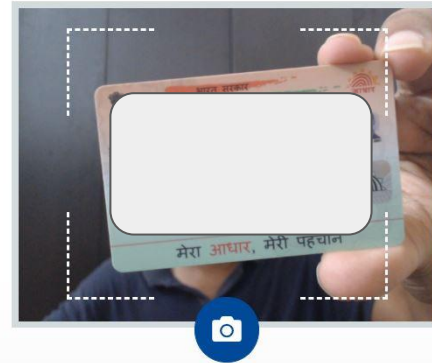
Registration Process

Scan Your Room and Workspace

Click the blue camera button and take a video of all four walls around your desk and your immediate workspace showing that you have no prohibited items within your reach. Click the red STOP button when you are done. Then, click CONTINUE.

💡 Scanning Requirements

1. Room Scan should be slow and thorough.
2. **Conduct a 360-degree scan all the way around the room.**
3. Slowly scan from the ceiling to the floor.
4. **Scan the entire desk/work-space area.**
5. Scan the area directly underneath where you will be placing laptop and/or keyboard.
6. After your laptop or keyboard is in place, show your cell phone to the camera and place it directly behind your seat and out of reach.
7. Roll up any sleeves and show both sides of arms up to the camera.
8. Show your ears to ensure there are no earbuds in use.
9. If wearing glasses, hold them up to the camera for visual inspection.
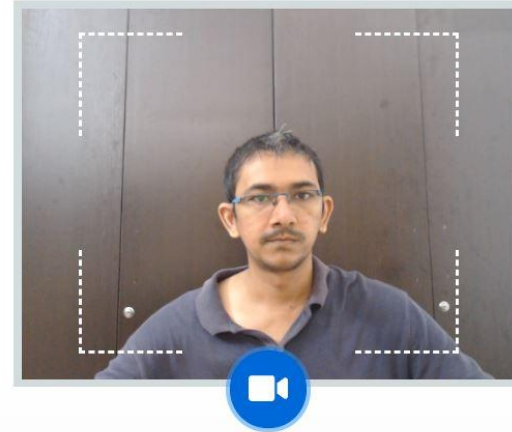
CONTINUE >

Important Pointers for Exams - 1

Let's get Certified

Terraform Providers

A provider is responsible for understanding API interactions and exposing resources.

Most of the available providers correspond to one cloud or on-premises infrastructure platform, and offer resource types that correspond to each of the features of that platform.

You can explicitly set a specific version of the provider within the provider block.

To upgrade to the latest acceptable version of each provider, run terraform init -upgrade

Provider Architecture

do_droplet.tf

Terraform

Digital Ocean Provider

Infrastructure Provisioning

(API interactions)

New Dropet

Digital Ocean
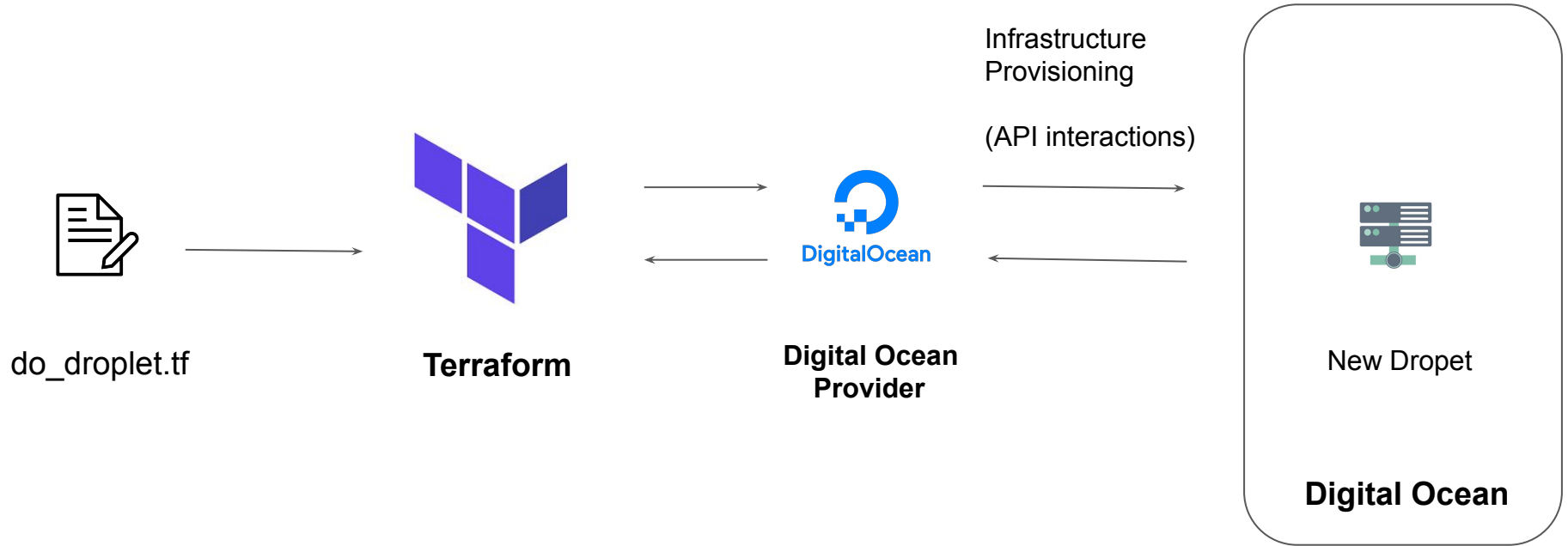
Terraform Providers - Part 2

You can have multiple provider instance with the help of alias

```
provider "aws" {
  region = "us-east-1"
  }

provider "aws" {
  alias  = "west"
  region = "us-west-2"
  }
```

The provider block without alias set is known as the default provider configuration. When an alias is set, it creates an additional provider configuration.

Terraform Init

The terraform init command is used to initialize a working directory containing Terraform configuration files.

During init, the configuration is searched for module blocks, and the source code for referenced modules is retrieved from the locations given in their source arguments.

Terraform must initialize the provider before it can be used.

Initialization downloads and installs the provider's plugin so that it can later be executed.

It will not create any sample files like example.tf

Terraform Plan

The terraform plan command is used to create an execution plan.

It will not modify things in infrastructure.

Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files.

This command is a convenient way to check whether the execution plan for a set of changes matches your expectations without making any changes to real resources or to the state.

Terraform Apply

The terraform apply command is used to apply the changes required to reach the desired state of the configuration.

Terraform apply will also write data to the terraform.tfstate file.

Once apply is completed, resources are immediately available.

Terraform Refresh

The terraform refresh command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure.

This does not modify infrastructure but does modify the state file.

Terraform Destroy

The terraform destroy command is used to destroy the Terraform-managed infrastructure.

terraform destroy command is not the only command through which infrastructure can be destroyed.

Terraform Format

The terraform fmt command is used to rewrite Terraform configuration files to a canonical format and style.

For use-case, where the all configuration written by team members needs to have a proper style of code, terraform fmt can be used.

Terraform Validate

The terraform validate command validates the configuration files in a directory.

Validate runs checks that verify whether a configuration is syntactically valid and thus primarily useful for general verification of reusable modules, including the correctness of attribute names and value types.

It is safe to run this command automatically, for example, as a post-save check in a text editor or as a test step for a reusable module in a CI system. It can run before terraform plan.

Validation requires an initialized working directory with any referenced plugins and modules installed

Terraform 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.

Provisioners should only be used as a last resort. For most common situations, there are better alternatives.

- Provisioners are inside the resource block.
- Have an overview of local and remote provisioner

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

  provisioner "local-exec" {
    command = "echo The server's IP address is ${self.private_ip}"
  }
}
```

Important Pointers for Exams - 2

Let's get Certified

Overview of Debugging 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.

Example:

TF_LOG=TRACE

To persist logged output, you can set TF_LOG_PATH

Terraform Import

Terraform is able to import existing infrastructure.

This allows you take resources that 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.

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

terraform import aws_instance.myec2 instance-id

Local Values

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

The expression of a local value can refer to other locals, but as usual reference cycles are not allowed. That is, a local cannot refer to itself or to a variable that refers (directly or indirectly) back to it.

It's recommended to group together logically-related local values into a single block, particularly if they depend on each other.

Overview of Data Types

Type Keywords	Description
string	Sequence of Unicode characters representing some text, like "hello".
list	Sequential list of values identified by their position. Starts with 0 ["mumbai" ,"singapore", "usa"]
map	a group of values identified by named labels, like {name = "Mabel", age = 52}.
number	Example: 200

Terraform WorkSpaces

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

Workspaces allow multiple state files of a single configuration.
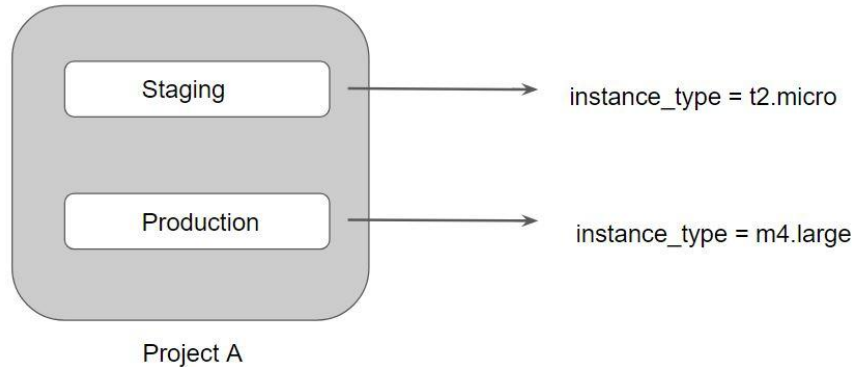
Terraform Modules

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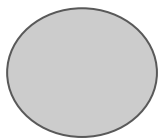
source

```
resource "aws_instance" "myweb" {

    ami = "ami-bf5540df"

    instance_type = "t2.micro"

    security_groups = ["default"]

}
```

module "source"

Terraform Modules - ROOT and Child

Every Terraform configuration has at least one module, known as its root module, which consists of the resources defined in the .tf files in the main working directory.

A module can call other modules, which lets you include the child module's resources into the configuration in a concise way.

A module that includes a module block like this is the calling module of the child module.

```
module "servers" {
  source = "./app-cluster"

  servers = 5
}
```

Module - Accessing Output Values

The resources defined in a module are encapsulated, so the calling module cannot access their attributes directly.

However, the child module can declare output values to selectively export certain values to be accessed by the calling module.

A module includes a module block like this is the calling module of the child module.

```
output "instance_ip_addr" {
  value = aws_instance.server.private_ip
}
```

Suppressing Values in CLI Output

An output can be marked as containing sensitive material using the optional sensitive argument:

```
output "db_password" {
  value       = aws_db_instance.db.password
  description = "The password for logging in to the database."
  sensitive   = true
}
```

Setting an output value in the root module as sensitive prevents Terraform from showing its value in the list of outputs at the end of terraform apply

Sensitive output values are still recorded in the state, and so will be visible to anyone who is able to access the state data.

Module Versions

It is recommended to explicitly constraining the acceptable version numbers for each external module to avoid unexpected or unwanted changes.

Version constraints are supported only for modules installed from a module registry, such as the Terraform Registry or Terraform Cloud's private module registry.

```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"

  servers = 3
}
```

Terraform Registry

The Terraform Registry is integrated directly into Terraform.

The syntax for referencing a registry module is

<NAMESPACE>/<NAME>/<PROVIDER>.

For example: hashicorp/consul/aws.

```
module "consul" {
  source = "hashicorp/consul/aws"
  version = "0.1.0"
}
```

Private Registry for Module Sources

You can also use modules from a private registry, like the one provided by Terraform Cloud.

Private registry modules have source strings of the following form:
<HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>.

This is the same format as the public registry, but with an added hostname prefix.

While fetching a module, having a version is required.

```
module "vpc" {
  source = "app.terraform.io/example_corp/vpc/aws"
  version = "0.9.3"
}
```

Important Pointers for Exams - 3

Let's get Certified

Terraform Functions

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

```
> 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

Be aware of basic functions like element, lookup.

Count and Count Index

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

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

```
resource "aws_iam_user" "lb" {
  name = "loadbalancer.${count.index}"
  count = 5
  path = "/system/"
}
```

Find the Issue - Use-Case

You can expect use-case with terraform code, and you have to find what should be removed as part of Terraform best practice.

```
terraform {
  backend "s3" {
    bucket = "mybucket"
    key     = "path/to/my/key"
    region = "us-east-1"
    access_key = 1234
    aecret_key = 1234567890
  }
}
```

Terraform Lock

If supported by your backend, Terraform will lock your state for all operations that could write state.

Terraform has a force-unlock command to manually unlock the state if unlocking failed.

Use-Case - Resources Deleted Out of Terraform

You have created an EC2 instance. Someone has modified the EC2 instance manually. What will happen if you do terraform plan yet again?

1. Someone has changed EC2 instance type from t2.micro to t2.large?
2. Someone has terminated the EC2 instance.

Answer 1. Terraform's current state will have t2.large, and the desired state is t2.micro. It will try to change back instance type to t2.micro.

Answer 2. Terraform will create a new EC2 instance.

Resource Block

Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records.

A resource block declares a resource of a given type ("aws_instance") with a given local name ("web").

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

Sentinel

Sentinel is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products.

Can be used for various use-cases like:

- Verify if EC2 instance has tags.
- Verify if the S3 bucket has encryption enabled.

```
terraform plan  →  sentinel checks  →  terraform apply
```
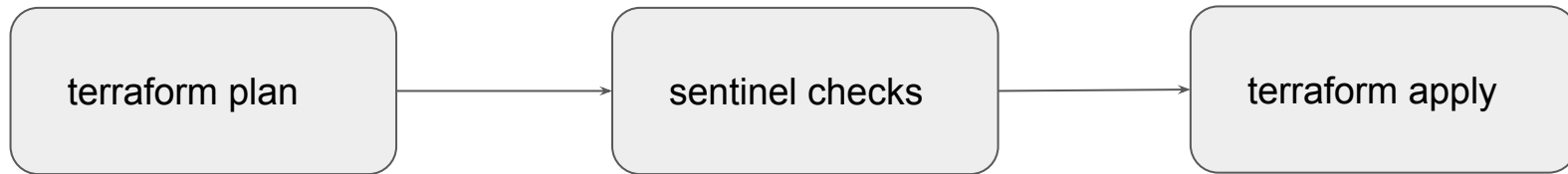
Sensitive Data in State File

If you manage any sensitive data with Terraform (like database passwords, user passwords, or private keys), treat the state itself as sensitive data.

Approaches in such a scenario:

Terraform Cloud always encrypts the state at rest and protects it with TLS in transit. Terraform Cloud also knows the identity of the user requesting state and maintains a history of state changes.

The S3 backend supports encryption at rest when the encrypt option is enabled.

Dealing with Credentials in Config

Hard-coding credentials into any Terraform configuration are not recommended, and risks the secret leakage should this file ever be committed to a public version control system.

You can store the credentials outside of terraform configuration.

Storing credentials as part of environment variables is also a much better approach than hard coding it in the system.

Remote Backend for Terraform Cloud

The remote backend stores Terraform state and may be used to run operations in Terraform Cloud.

When using full remote operations, operations like terraform plan or terraform apply can be executed in Terraform Cloud's run environment, with log output streaming to the local terminal.

Miscellaneous Pointers

Terraform does not require go as a prerequisite.

It works well in Windows, Linux, MAC.

Windows Server is not mandatory.

Important Pointers for Exams - 4

Let's get Certified

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.
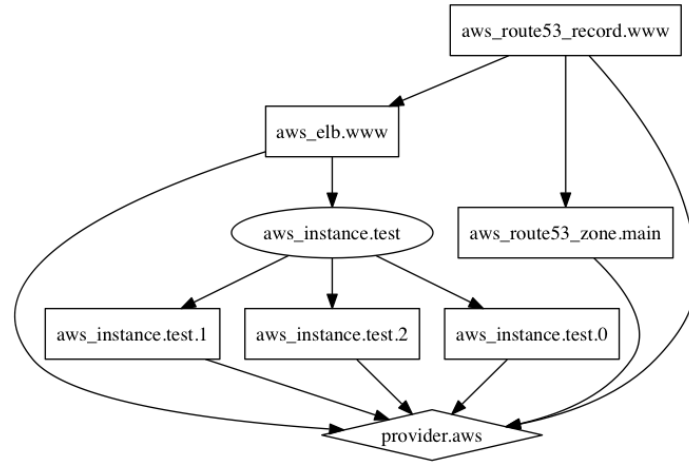
Splat Expressions

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

```
resource "aws_iam_user" "lb" {
  name = "iamuser.${count.index}"
  count = 3
  path = "/system/"
}

output "arns" {
  value = aws_iam_user.lb[*].arn
}
```

Terraform Terminologies

aws_instance	Resource Type
example	Local name for the resource
ami	Argument Name
abc123	Argument value

```
resource "aws_instance" "example" {
  ami = "abc123"
  }
```

Provider Configuration

Provider Configuration block is not mandatory for all the terraform configuration.

```
provider "aws" {
  region      = "us-east-1"
  access_key = "YOUR-KEY"
  secret_key = "YOUR-KEY"
}

resource "aws_iam_user" "iam" {
  name = "iamuser"
  path = "/system/"
}
```

```
locals {
  arr = ["value1", "value2"]
}


output "test" {
  value = local.arr
}
```

Terraform Output

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

```
C:\Users\Zeal Vora\Desktop\terraform\terraform output>terraform output iam_names
[
  "iamuser.0",
  "iamuser.1",
  "iamuser.2",
]
```

Terraform Unlock

If supported by your backend, Terraform will lock your state for all operations that could write state.

Not all backends supports locking functionality.

Terraform has a force-unlock command to manually unlock the state if unlocking failed.

terraform force-unlock LOCK_ID [DIR]

Miscellaneous Pointers - 1

There are three primary benefits of Infrastructure as Code tools:

Automation, Versioning, and Reusability.

Various IAC Tools Available in the market:

- Terraform
- CloudFormation
- Azure Resource Manager
- Google Cloud Deployment Manager

Miscellaneous Pointers - 2

Sentinel is a proactive service.

Terraform Refresh does not modify the infrastructure but it modifies the state file.

Slice Function is not part of the string function. Others like join, split, chomp are part of it.

It is not mandatory to include the module version argument while pulling the code from terraform registry.

Miscellaneous Pointers - 3

Overuse of dynamic blocks can make configuration hard to read and maintain.

Terraform Apply can change, destroy and provision resources but cannot import any resource.

Join us in our Adventure

Be Awesome

kplabs.in/twitter

kplabs.in/linkedin

Important Pointers for Exams - 5

Let's get Certified

Terraform Enterprise & Terraform Cloud

Terraform Enterprise provides several added advantage compared to Terraform Cloud.

Some of these include:

- Single Sign-On
- Auditing
- Private Data Center Networking
- Clustering

Team & Governance features are not available for Terraform Cloud Free (Paid)

Variables with undefined values

If you have variables with undefined values, it will not directly result in an error.

Terraform will ask you to supply the value associated with them.

Example Code:

variable custom_var { }

```
C:\Users\Zeal Vora\Desktop\terraform\tmp\1>terraform plan
var.custom_var
  Enter a value:
```

Environment Variables

Environment variables can be used to set variables.

The environment variables must be in the format TF_VAR_name

```
export TF_VAR_region=us-west-1

export TF_VAR_ami=ami-049d8641

export TF_VAR_alist='[1,2,3]'
```

Structural Data Types

A structural type allows multiple values of several distinct types to be grouped together as a single value.

List contains multiple values of same type while object can contain multiple values of different type.

Structural Type	Description

object	A collection of named attributes that each have their own type.
	object({<ATTR NAME> = <TYPE>, ... })
	object({ name=string, age=number })
	
	{
	name = "John"
	age = 52
	}

tuple	tuple([<TYPE>, ...])

BackEnd Configuration

Backends are configured directly in Terraform files in the terraform section.

After configuring a backend, it has to be initialized.

```
terraform {
  backend "s3" {
    bucket = "mybucket"
    key    = "path/to/my/key"
    region = "us-east-1"
  }
}
```

BackEnd Configuration Types - 1

First Time Configuration:

When configuring a backend for the first time (moving from no defined backend to explicitly configuring one), Terraform will give you the option to migrate your state to the new backend.

This lets you adopt backends without losing any existing state.

BackEnd Configuration Types - 2

Partial Time Configuration:

You do not need to specify every required argument in the backend configuration. Omitting certain arguments may be desirable to avoid storing secrets, such as access keys, within the main configuration.

With a partial configuration, the remaining configuration arguments must be provided as part of the initialization process.

```
terraform {
  backend "consul" {}
}
```

```
terraform init \
    -backend-config="address=demo.consul.io" \
    -backend-config="path=example_app/terraform_state" \
    -backend-config="scheme=https"
```

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.

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.
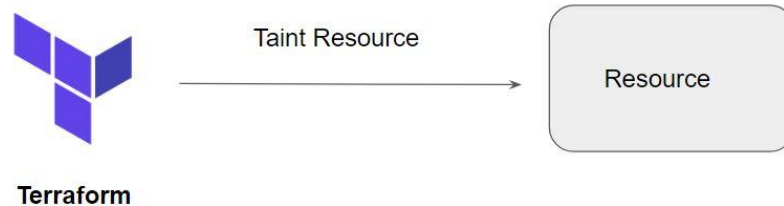
Input Variables

The value associated with a variable can be assigned via multiple approaches.

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

Value associated with the variables can be defined via CLI as well as in tfvars file.

Following is syntax to load custom tfvars file:

terraform apply -var-file="testing.tfvars"

Variable 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.

If the same variable is assigned multiple values, Terraform uses the last value it finds.

Terraform Local Backend

The local backend stores state on the local filesystem, locks that state using system APIs, and performs operations locally.

By default, Terraform uses the "local" backend, which is the normal behavior of Terraform you're used to

```
terraform {
  backend "local" {
    path = "relative/path/to/terraform.tfstate"
  }
}
```

Required Providers

Each Terraform module must declare which providers it requires, so that Terraform can install and use them.

Provider requirements are declared in a required_providers block.

```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
    }
  }
}
```

Required Version

The required_version setting accepts a version constraint string, which specifies which versions of Terraform can be used with your configuration.

If the running version of Terraform doesn't match the constraints specified, Terraform will produce an error and exit without taking any further actions.

```
terraform {
  required_version = "> 0.12.0"
}
```

Versioning Arguments

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

Version Number Arguments	Description
>=1.0	Greater than equal to the version
<=1.0	Less than equal to the version
~>2.0	Any version in the 2.X range.
>=2.10,<=2.30	Any version between 2.10 and 2.30

Important Pointers for Exams - 6

Let's get Certified

Fetching Values from Map

To reference to image-abc from the below map, following approaches needs to be used:

var.ami_ids["mumbai"]

```
variable "ami_ids" {
    type = "map"
    default = {
        "mumbai" = "image-abc"
        "germany" = "image-def"
        "states"  = "image-xyz"
     }
}
```

Terraform and GIT - Part 1

If you are making use of GIT repository for commiting terraform code, the .gitignore should be configured to ignore certain terraform files that might contain sensitive data.

Some of these can include:

- terraform.tfstate file (this can include sensitive information)

- *.tfvars (may contain sensitive data like passwords)

.

Terraform and GIT - Part 2

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.

```
module "vpc" {
  source = "git::https://example.com/vpc.git"
}


module "storage" {
  source = "git::ssh://username@example.com/storage.git"
}
```

.

Terraform and GIT - Part 3

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:

```
module "vpc" {
  source = "git::https://example.com/vpc.git?ref=v1.2.0"
}
```

.
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 Workspace

- Workspaces are managed with the terraform workspace set of commands.
- State File Directory = terraform.tfstate.d
- Not suitable for isolation for strong separation between workspace (stage/prod)

Use-Case	Command
Create New Workspace	terraform workspace new kplabs
Switch to a specific Workspace	terraform workspace select prod

.

```
$ terraform workspace new bar
Created and switched to workspace "bar"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

Dependency Types - Implicit

With implicit dependency, Terraform can automatically find references of the object, and create an implicit ordering requirement between the two resources.

.

```
resource "aws_eip" "my_eip"{
  vpc = "true"
}


resource "aws_instance" "my_ec2" {
    instance_type = "t2.micro"
    public_ip     = aws_eip.myeip.private_ip
}
```

Dependency Types - Explicit

Explicitly specifying a dependency is only necessary when a resource relies on some other resource's behavior but doesn't access any of that resource's data in its arguments.

.

```
resource "aws_s3_bucket" "example" {
  acl     = "private"
}


resource "aws_instance" "myec2" {
    instance_type = "t2.micro"
    depends_on = [aws_s3_bucket.example]
}
```

State Command

Rather than modify the state directly, the terraform state commands can be used in many cases instead.

State Command	Description
terraform state list	List resources within terraform state
terraform state mv	Move items within terraform state. Can be used to resource renaming.
terraform state pull	manually download and output the state from state file.
terraform state rm	Remove Items from terraform state file.
Terraform state show	Show the attributes of a single resource in the Terraform state.

Data Source Code

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

- Reads from a specific data source (aws_ami) and exports results under "app_ami"

```
data "aws_ami" "app_ami" {
  most_recent = true
  owners = ["amazon"]


  filter {
    name   = "name"
    values = ["amzn2-ami-hvm*"]
  }
}
```

```
resource "aws_instance" "instance-1" {
    ami = data.aws_ami.app_ami.id
    instance_type = "t2.micro"
}
```

Terraform taint

Terraform Taint can also be used to taint resource within a module.

terraform taint [options] address

```
$ terraform taint "module.couchbase.aws_instance.cb_node[9]"
Resource instance module.couchbase.aws_instance.cb_node[9] has been marked as tainted.
```

For multiple sub modules, following syntax-based example can be used

module.foo.module.bar.aws_instance.qux

Terraform Plan Destroy

The behavior of any terraform destroy command can be previewed at any time with an equivalent terraform plan -destroy command.

```
C:\Users\Zeal Vora\Desktop\terraform\tmp\1>terraform plan -destroy
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:

Terraform will perform the following actions:

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

Changes to Outputs:
  - test = [
      - "host1",
      - "host2",
      - "host3",
    ] -> null
```

Terraform Module Sources

The module installer supports installation from a number of different source types like Local paths, Terraform Registry, GitHub, S3 buckets and others.

Local path references allow for factoring out portions of a configuration within a single source repository.

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

```
module "consul" {
    source = "../consul"
}
```

Dealing with Larger Infrastructure

Cloud Providers has certain amount of rate limiting set so Terraform can only request certain amount of resources over a period of time.

It is important to break larger configurations into multiple smaller configurations that can be independently applied.

Alternatively, you can make use of -refresh=false and target flag for a workaround (not recommended)

Miscellaneous Pointers

lookup retrieves the value of a single element from a map

```
lookup(map, key, default)
```

Various commands runs terraform refresh implicitly, some of these include:

terraform [plan, apply, destroy]

Others like terraform [init, import] do not run refresh implictly.

Array Datatype is not supported in Terraform.

Miscellaneous Pointers -2

Various variable definition files will be loaded automatically in terraform. These include:

- terraform.tfvars
- terraform.tfvars.json
- Any files with names ending in .auto.tfvars.json

Both implicit and explicit dependency information is stored in terraform.tfstate file.

terraform init -upgrade updates all previously installed plugins to the newest version.

Miscellaneous Pointers -3

The terraform console command provides an interactive console for evaluating expressions.

Difference 0.11 and 0.12

- "${var.instance_type}" → 0.11
- var.instance_type → 0.12

Miscellaneous Pointers -3

If you have multiple modules and you want to export a value from one module to be imported into another module,

Difference 0.11 and 0.12

- "${var.instance_type}" → 0.11
- var.instance_type   → 0.12