HashiCorp Certified Terraform Associate 2024

Infrastructure as Code (IAC)

Understanding the Basics

There are two ways in which you can create and manage your infrastructure:

- Manually approach.
- Through Automation

Work Requirement: Database Backup

I was assigned a task to take database backup every day at 10 PM and the backup had to be stored in Amazon S3 Storage with appropriate timestamp.

- db-backup-01-01-2024.sql
- db-backup-02-01-2024.sql

Initially due to lack of time, I used to manually take DB backup at 10 PM and upload it to S3.

Learning from this Work Requirement

If a particular task has to be done in an repeatable manner, it MUST be automated.

Points to Note:

1. Depending on the type of task, the tools for automation will change.

2. There are wide variety of Tools & Technologies used for Automation like Ansible, CloudFormation, Terraform, Python etc.

Example of a Single Service

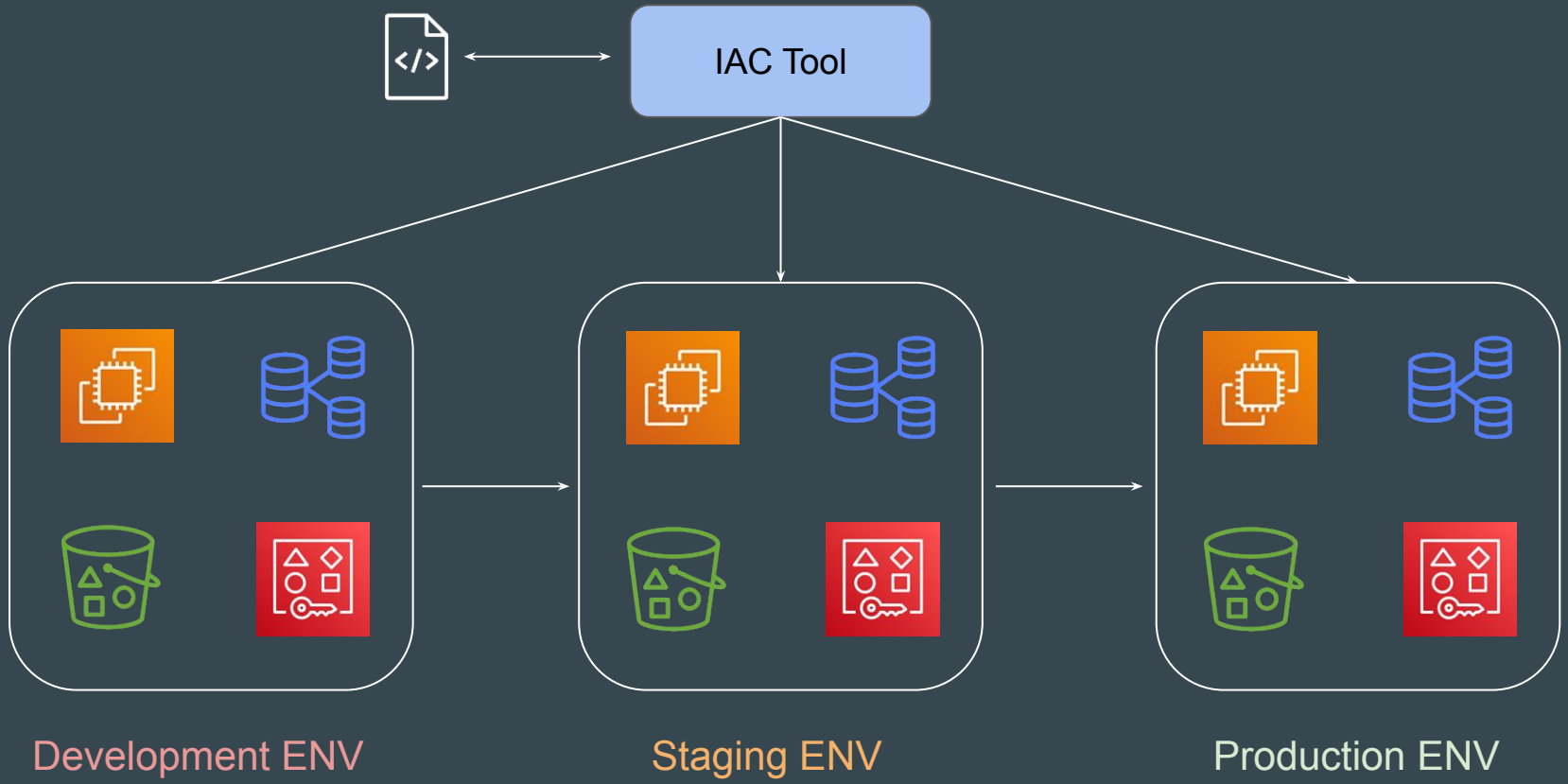
Set of resources (Virtual Machine, Database, S3, AWS Users) must be created with exact similar configuration in Dev, Stage and Production environment.

Development ENV

Staging ENV

Production ENV

Example of a Single Service - Automated Way



Basics of Infrastructure as Code

Infrastructure as Code (IaC) is the managing and provisioning of infrastructure through code instead of through manual processes.

test.yaml

```yaml
AWSTemplateFormatVersion: 2010-09-09
Description: Simple EC2

Resources:
  WebAppInstance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: ami-079db87dc4c10ac91
      InstanceType: t2.micro
```

vm.tf > ...

```
resource "aws_instance" "myec2" {
    ami = "ami-079db87dc4c10ac91"
    instance_type = "t2.micro"
}
```

Benefits of Infrastructure As Code

There are several benefits of designing your infrastructure as code:

- Speed of Infrastructure Management.
- Low Risk of Human Errors.
- Version Control.
- Easy collaboration between Teams.

Choosing Right IAC Tool

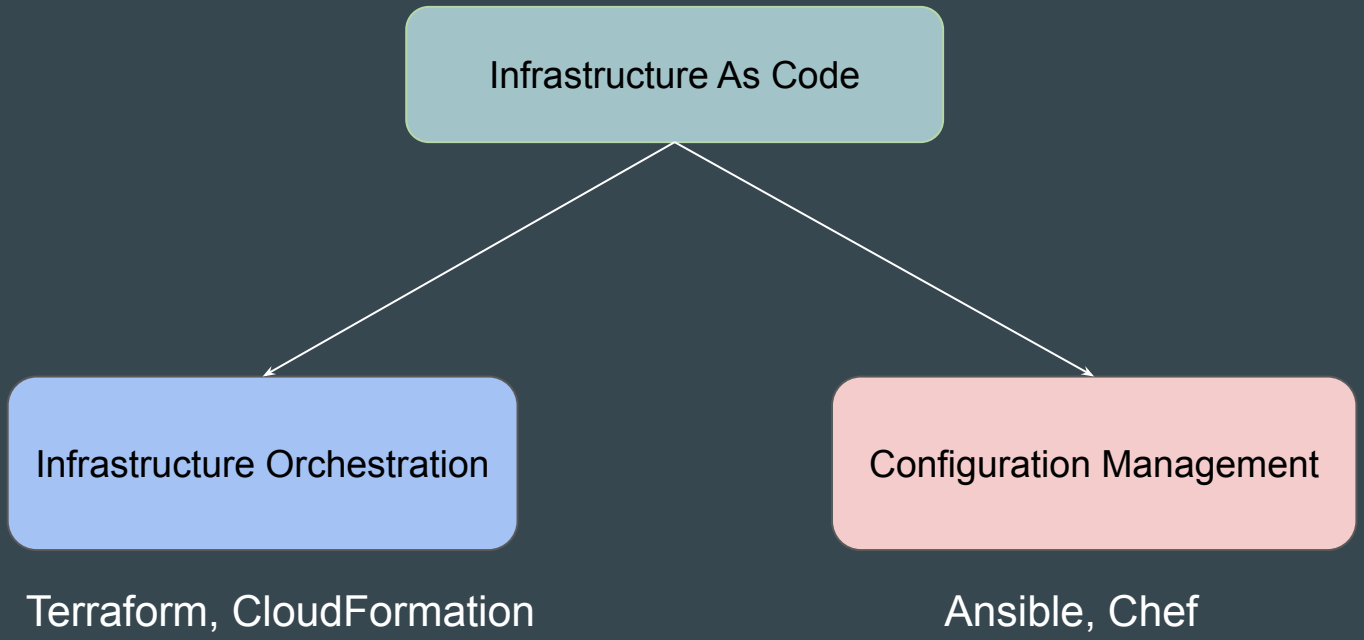
Available Tools

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

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

Categories of Tools

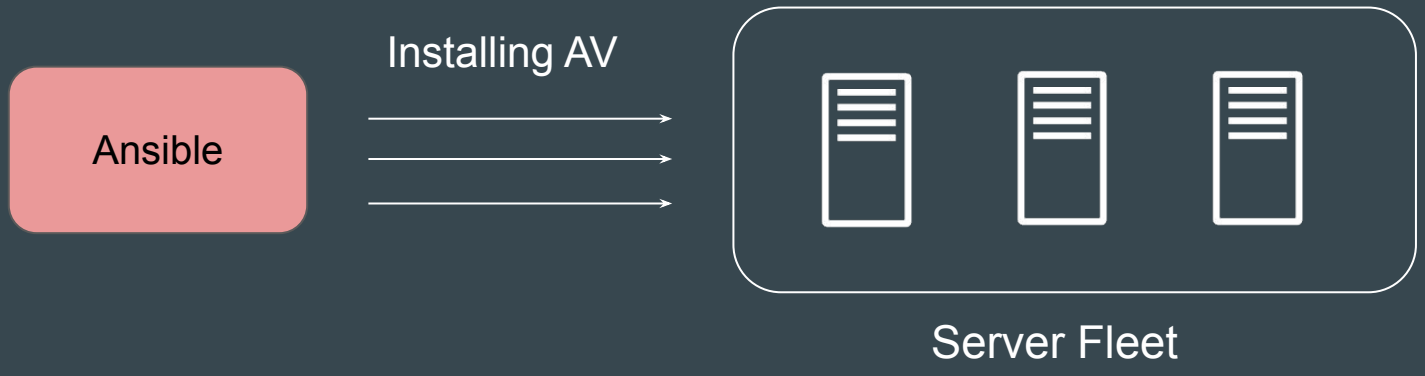
The tools are widely divided into two major categories



Configuration Management

Configuration Management tools are primarily used to maintain desired configuration of systems (inside servers)

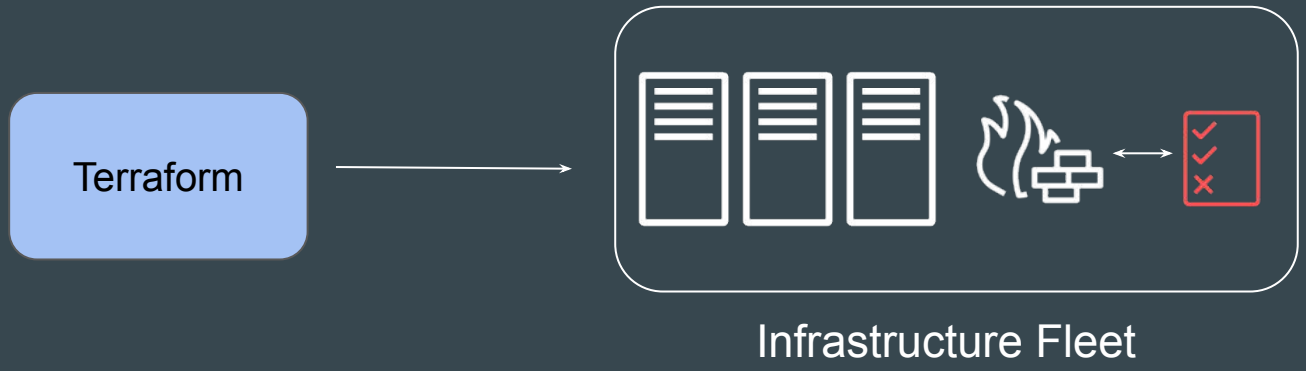
Example: ALL servers should have Antivirus installed with version 10.0.2



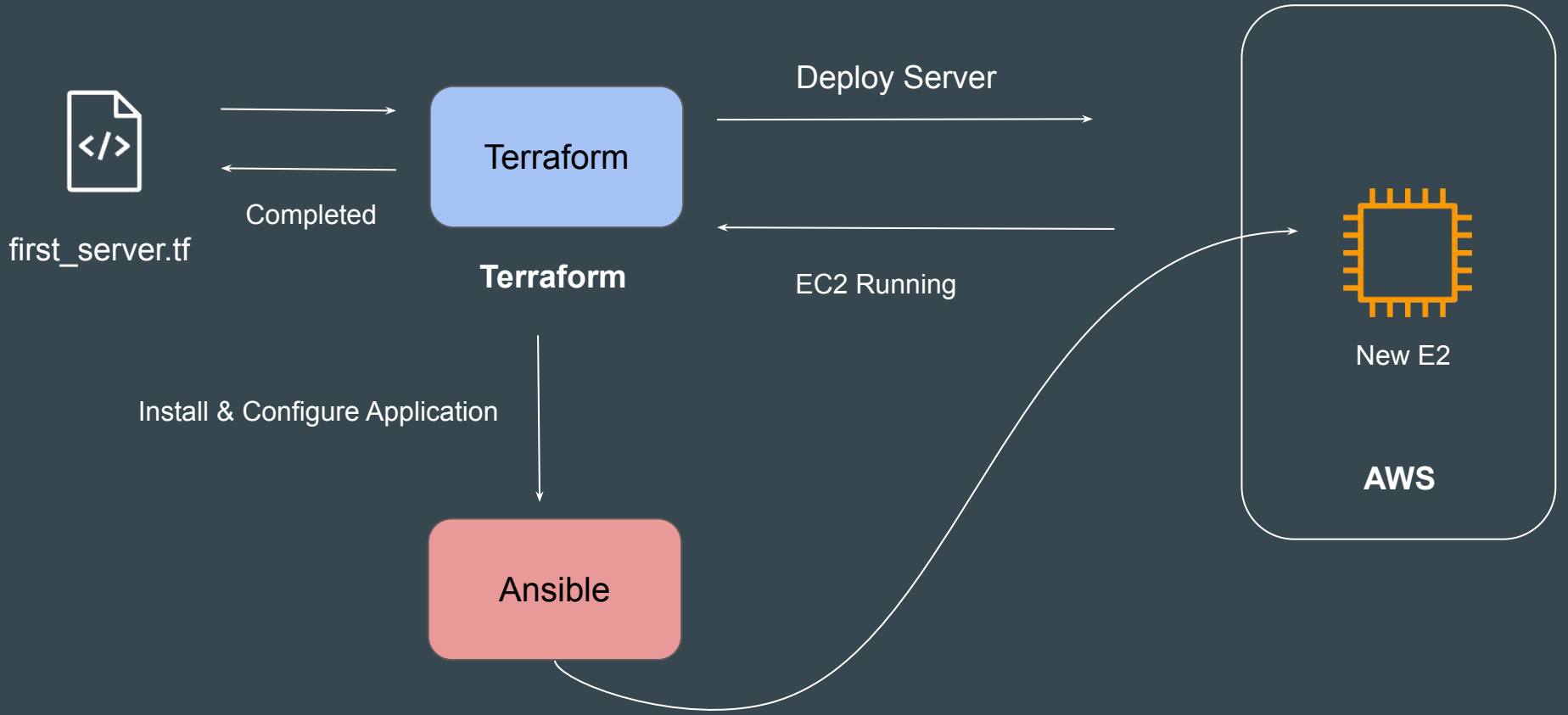
Infrastructure Orchestration

Infrastructure Orchestration is primarily used to create and manage infrastructure environments.

Example: Create 3 Servers with 4 GB RAM, 2 vCPUs. Each server should have firewall rule to allow SSH connection from Office IPs.



IAC & Configuration Management = Friends



How to choose IAC Tool?

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

Use-Case 1 - Requirement of Organization 1

1. Organization is going to be based on AWS for next 25 years.

2. Official support is required in-case if team face any issue related to IAC tool or code itself.

3. They want some kind of GUI interface that supports automatic code generation.

Use-Case 2 - Requirement of Organization 2

1. Organization is based on Hybrid Solution. They use VMware for on-premise setup; AWS, Azure and GCP for Cloud.

2. Official support is required in-case if IAC tool has any issues.

Installing Terraform

Terraform in detail

Overview of Installation Process

Terraform installation is very simple.

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

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

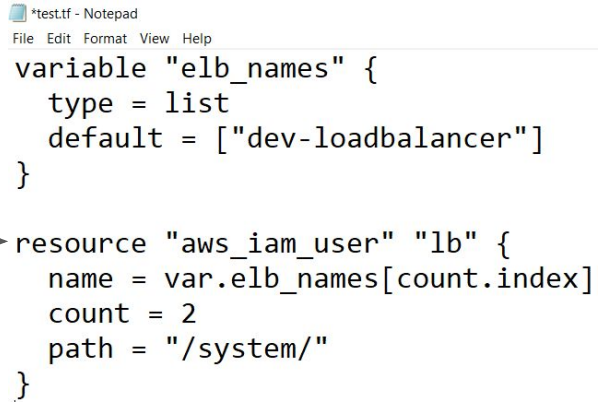
Terraform in detail

Terraform Code in NotePad!

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

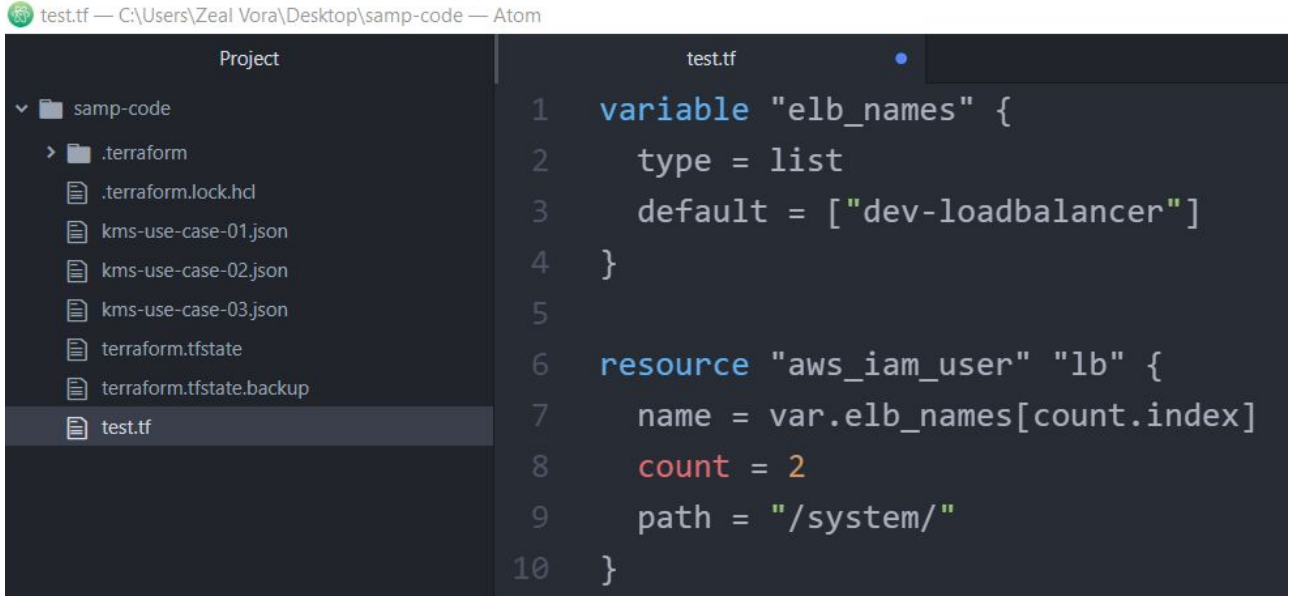
Downsides:

- Slower Development
- Limited Features



Need of a Better Software

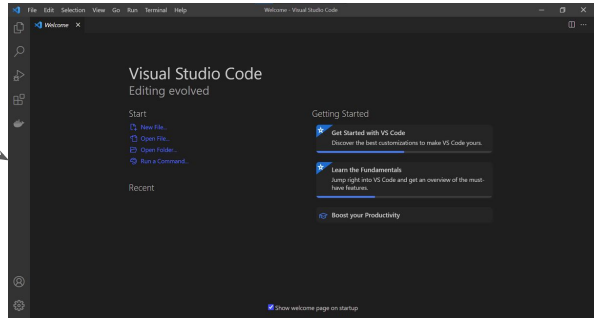
There is a need of a better application that allows us to develop code faster.



What are the Options!

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

Source Code Editors

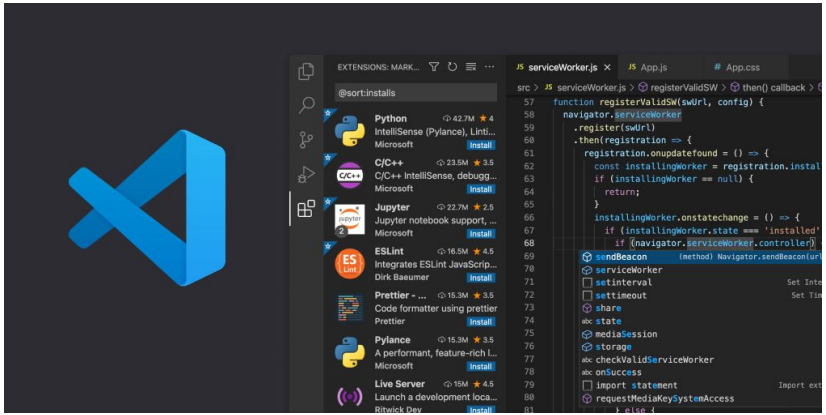


Editor for This Course

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.



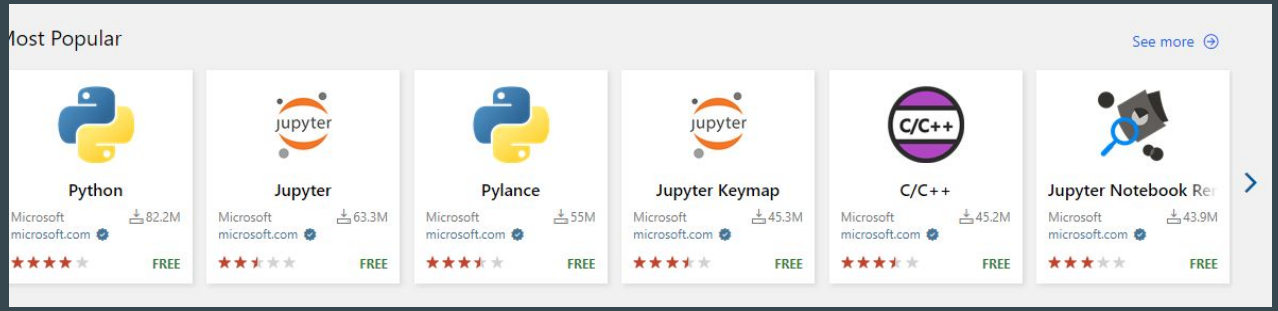

Using
Notepad
Using Visual
Studio Code
imgflip.com

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.

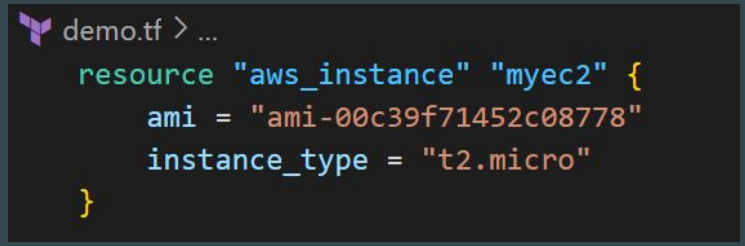


Terraform Extension

HashiCorp also provides extension for Terraform for Visual Studio Code.

demo.tf

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



Setting up the Lab

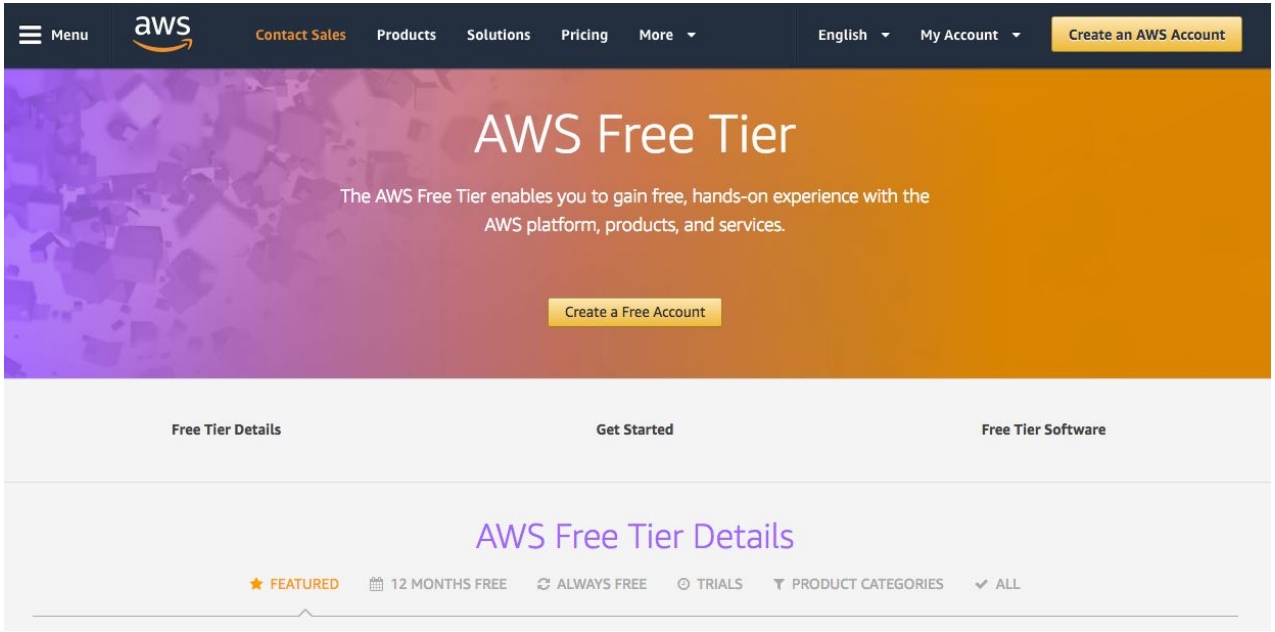
Let's start Rolling !

Let's Start

i) Create a new AWS Account.

ii) Begin the course

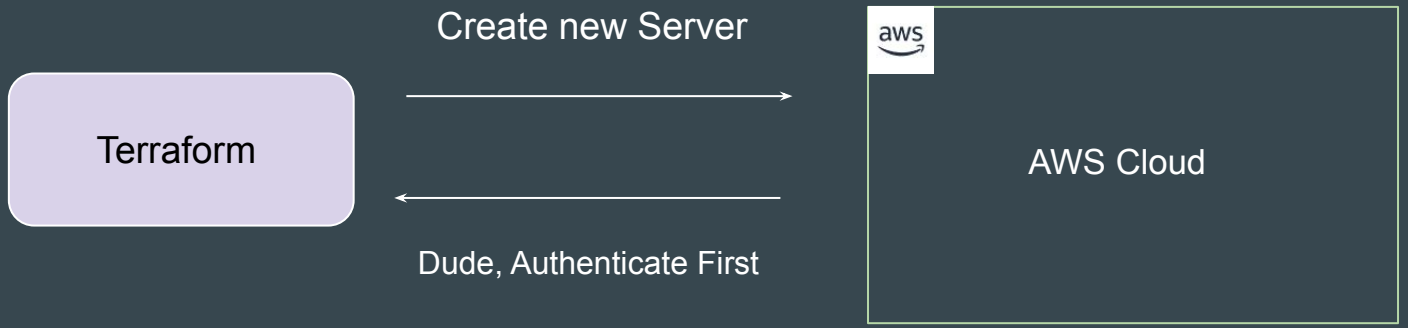
Registering an AWS Account



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.

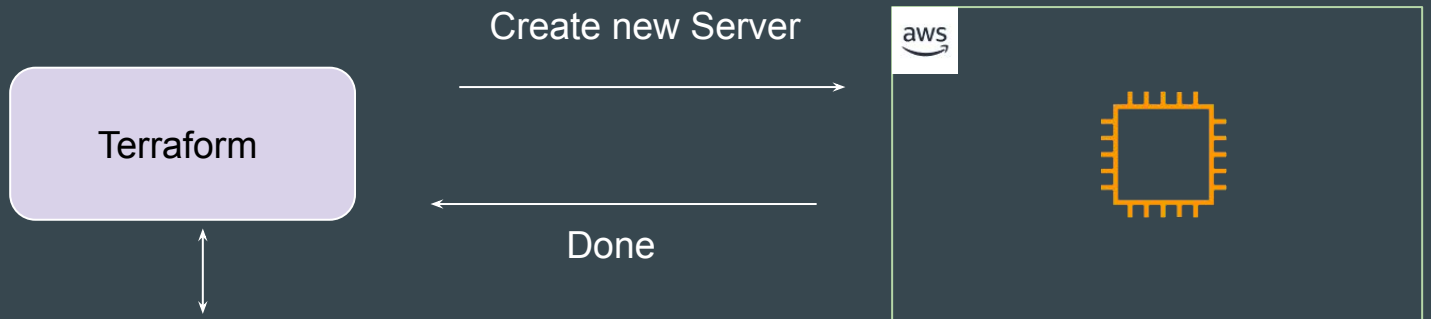
Authorization is the process of verifying what they have access to

Example:

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

Learning for Todays' Video

Terraform needs access credentials with relevant permissions to create and manage the environments.



username	password
Bob	pwd928#

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

First Virtual Machine Through Terraform

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.

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

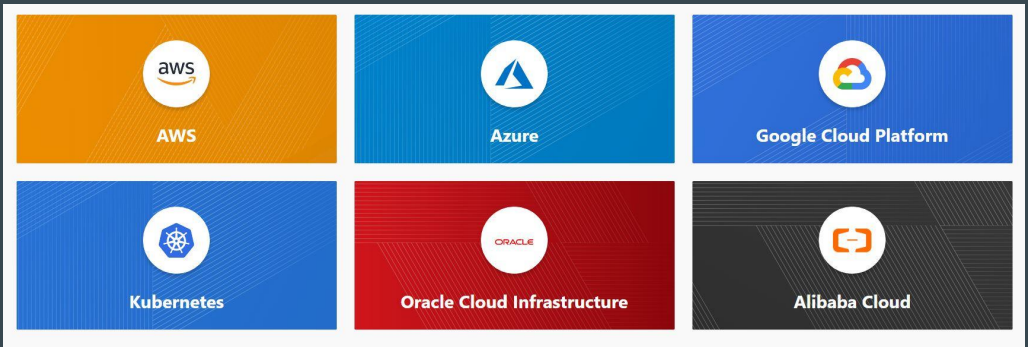
While creating VM through Terraform, you will need to define these.

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

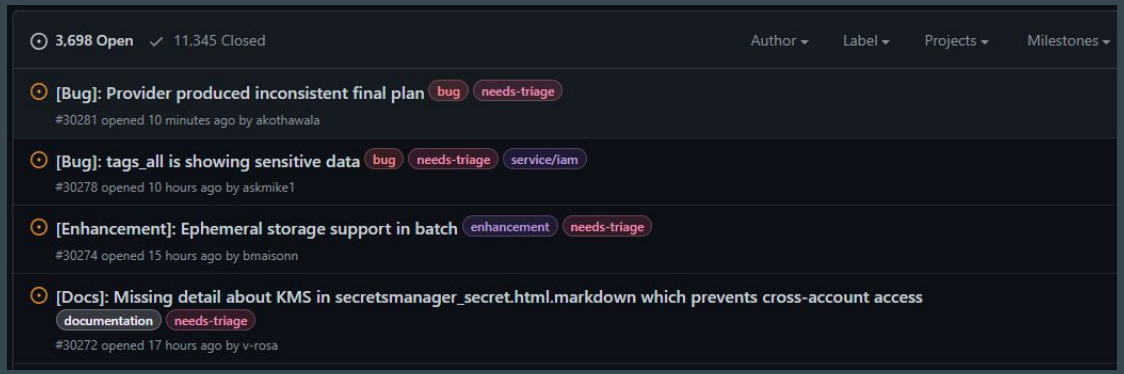
The core concepts, standard syntax remains similar across all providers.

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"
}
```

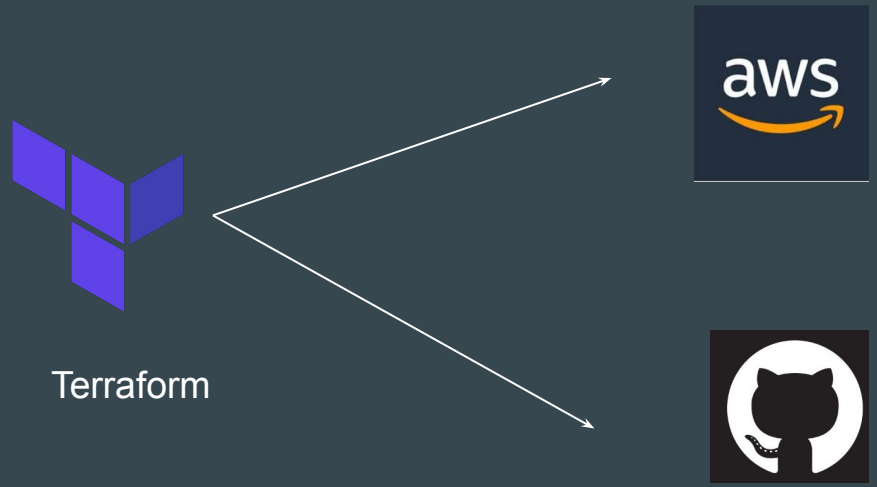
Non-HashiCorp Maintained

Terraform Destroy

Learning to Destroy Resources

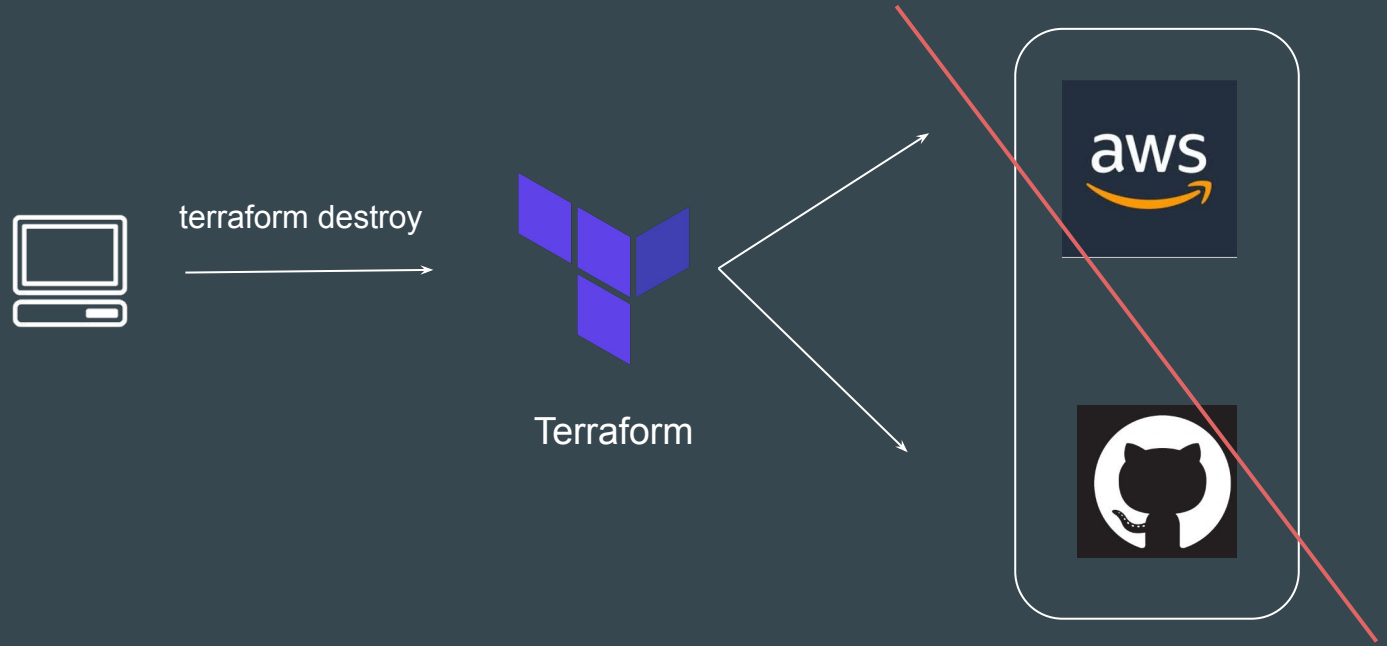
If you keep the infrastructure running, you will get charged for it.

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



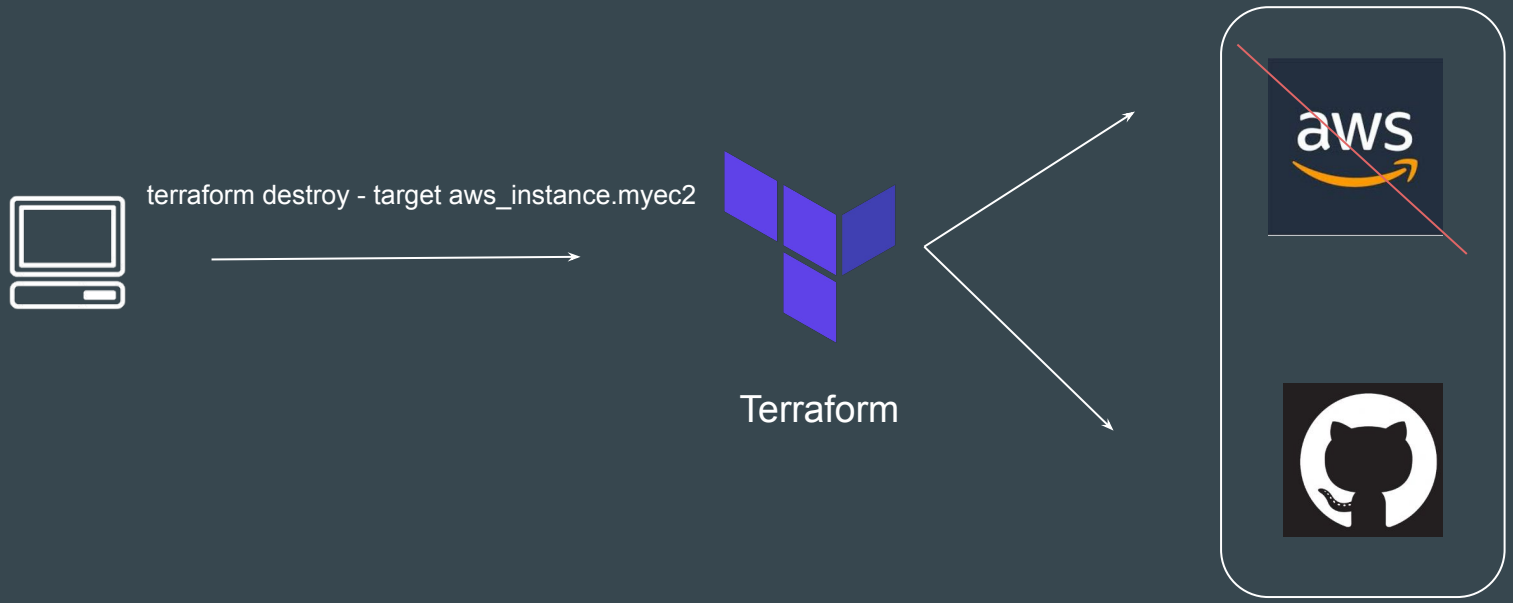
Approach 1 - Destroy ALL

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"

}
```

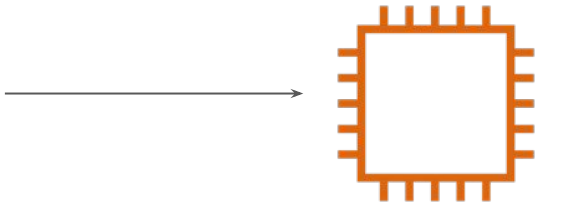
Desired & Current State

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

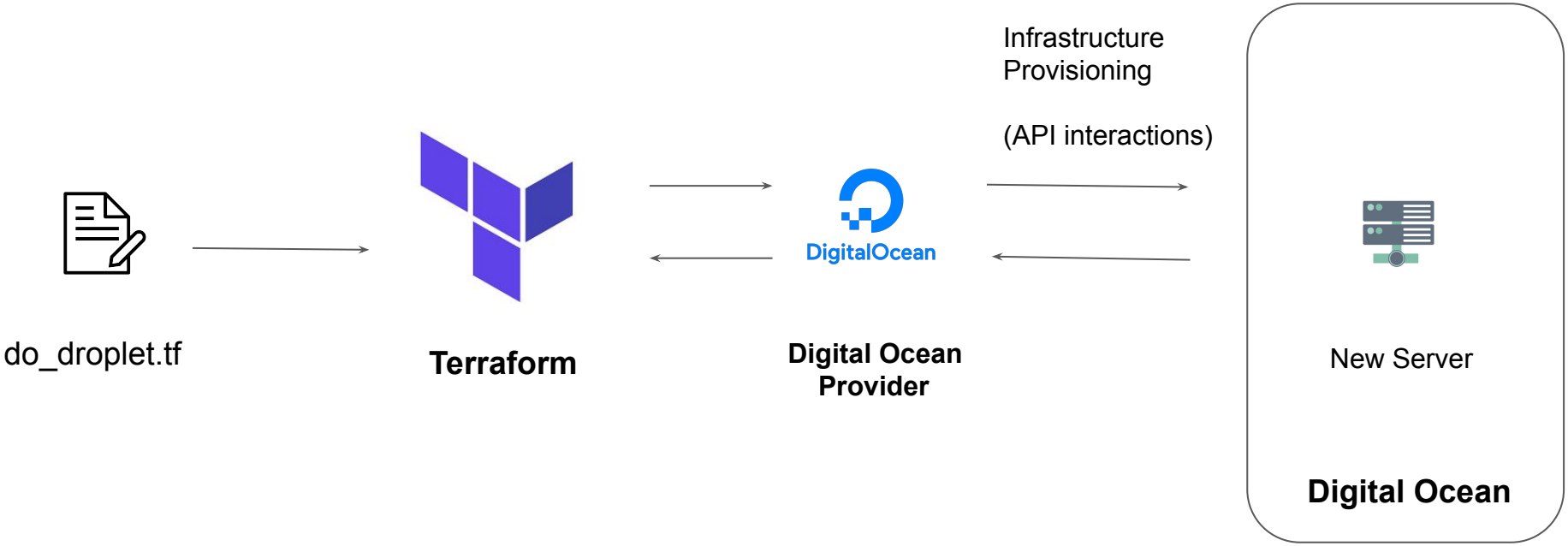
Terraform tries to ensure that the deployed infrastructure is based on the desired state.

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

Provider Versioning

Terraform in detail

Provider Architecture

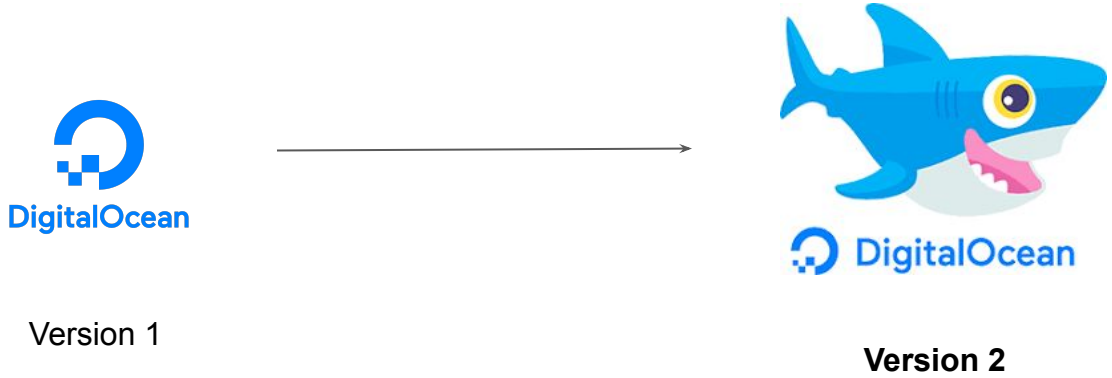


Overview of Provider Versioning

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,

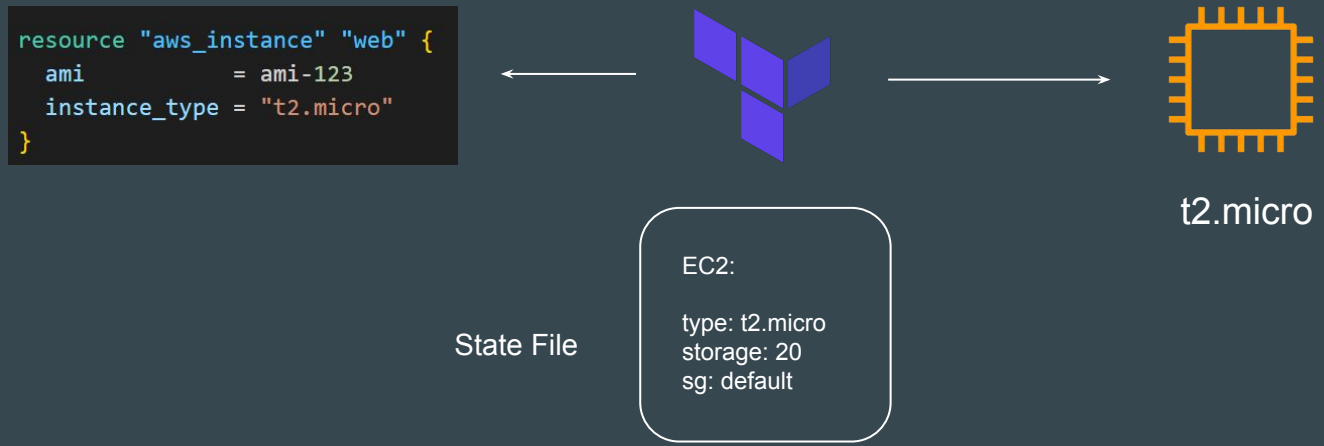
```
provider "registry.terraform.io/hashicorp/aws" {
  version     = "2.70.0"
  constraints = ">= 2.31.0, <= 2.70.0"
  hashes = [
    "h1:fx8tbGVwK1YIDI6UdHLnorC9PA1ZPSWEeW3V3aDCdWY=",
    "zh:01a5f351146434b418f9ff8d8cc956ddc801110f1cc8b139e01be2ff8c544605",
    "zh:1ec08abbaf09e3e0547511d48f77a1e2c89face2d55886b23f643011c76cb247",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

Terraform Refresh

Understanding the Challenge

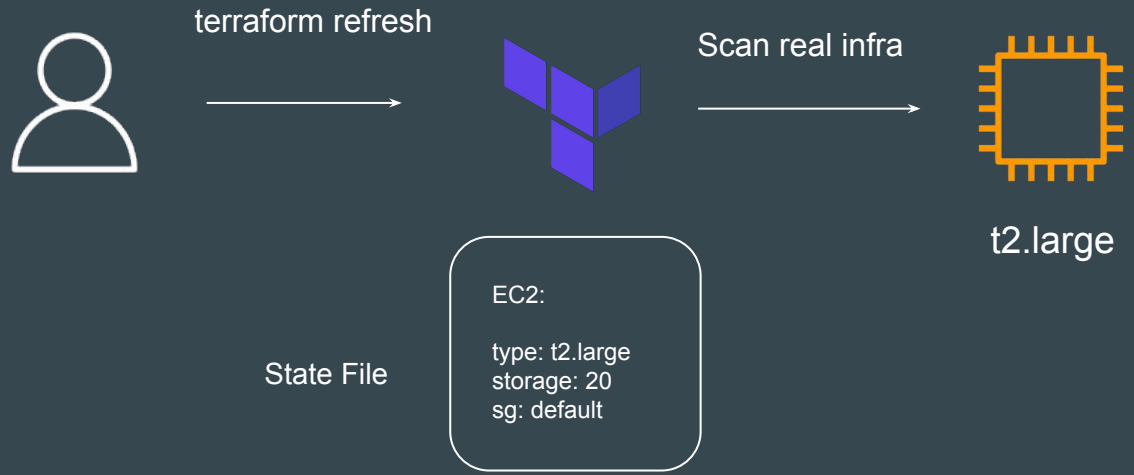
Terraform can create an infrastructure based on configuration you specified.

It can happen that the infrastructure gets modified manually.



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.

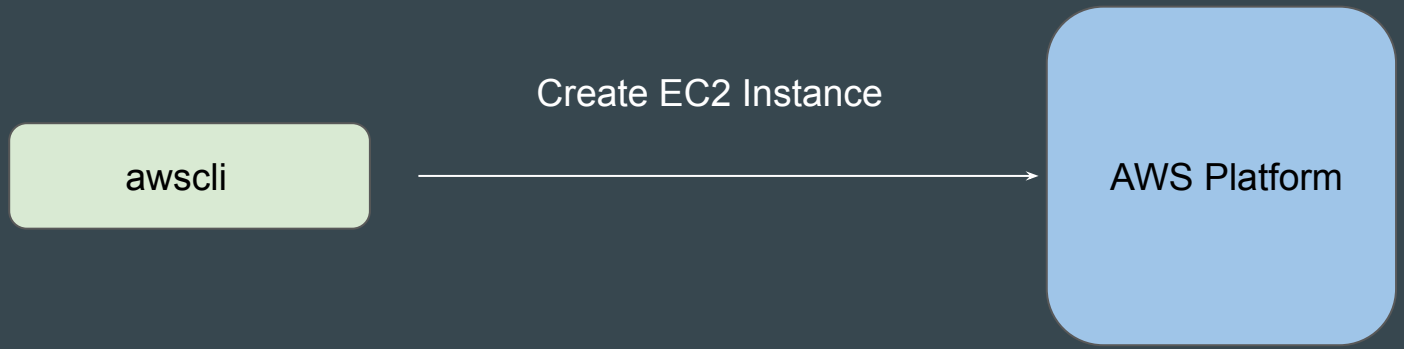
aws-provider-config.tf > ...

```
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.



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.

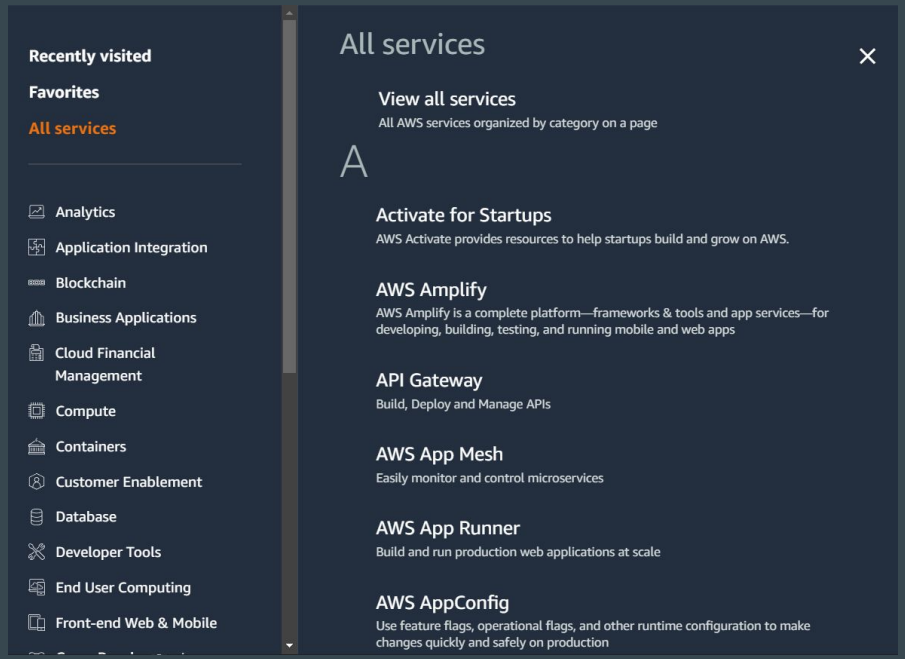
This is easier if you are maintaining separate folder for each practical.

Relax and Have a Meme Before Proceeding

Learning Scope - AWS Services for Terraform Course

Understanding the Basics

AWS has more than 200 services available.

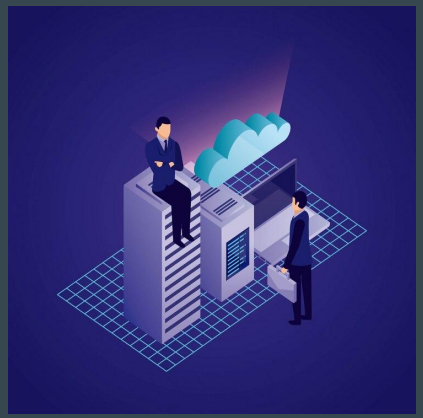


Aim of the Course

Primary aim of this course is to master the core concepts of Terraform.

Terraform = Infrastructure as Code Tool.

To learn Terraform, we need to create infrastructure somewhere.



Services that we Choose

Throughout the course, we use very basic AWS services to demonstrate Terraform concepts.

- Virtual Machine (EC2)
- Firewall (Security Groups)
- AWS Users (IAM Users)
- IP Address (Elastic IP)

Basics of These Services are Covered

We have 100,000+ students from different background who are learning Terraform.

Some are AWS Pros, Some are from Azure/GCP, Some are students

To align everyone on same page, we also cover basics of the AWS service that we use throughout the course.

Example - Creating Firewall Through Terraform

1. Basics of Firewalls in AWS

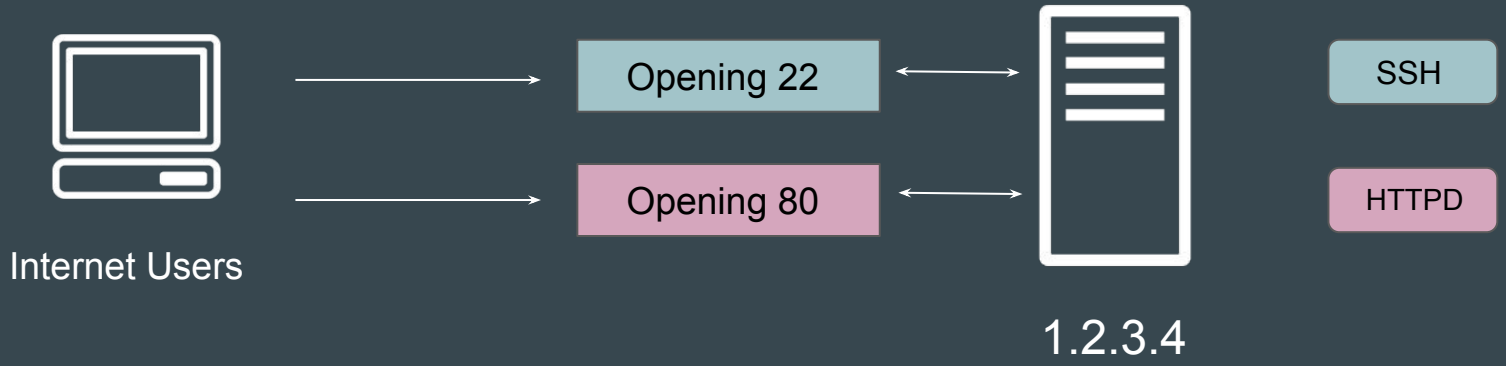
2. Firewall Practical in AWS (GUI Console)

3. Creating Firewall Rules Through Terraform.

Basics of Firewalls

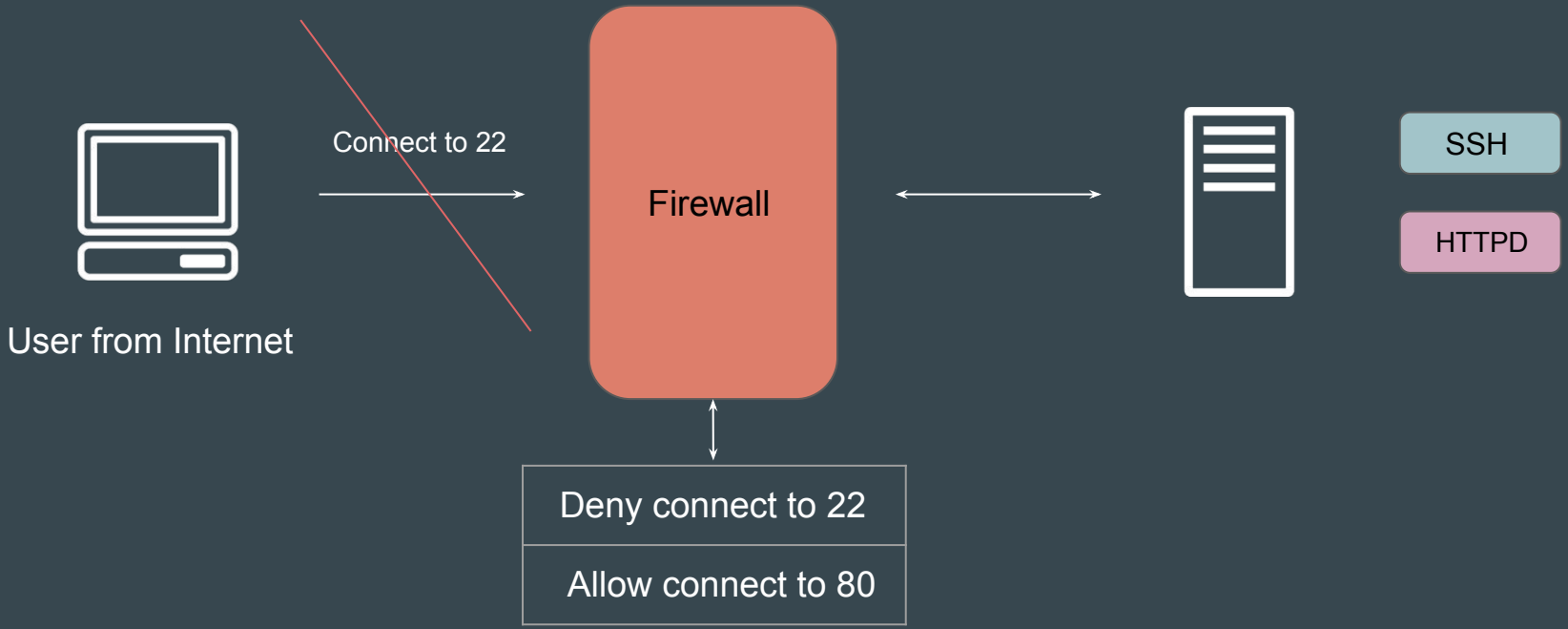
Basics of Ports

A port acts as a endpoint of communication to identify a given application or process on an Linux operating system



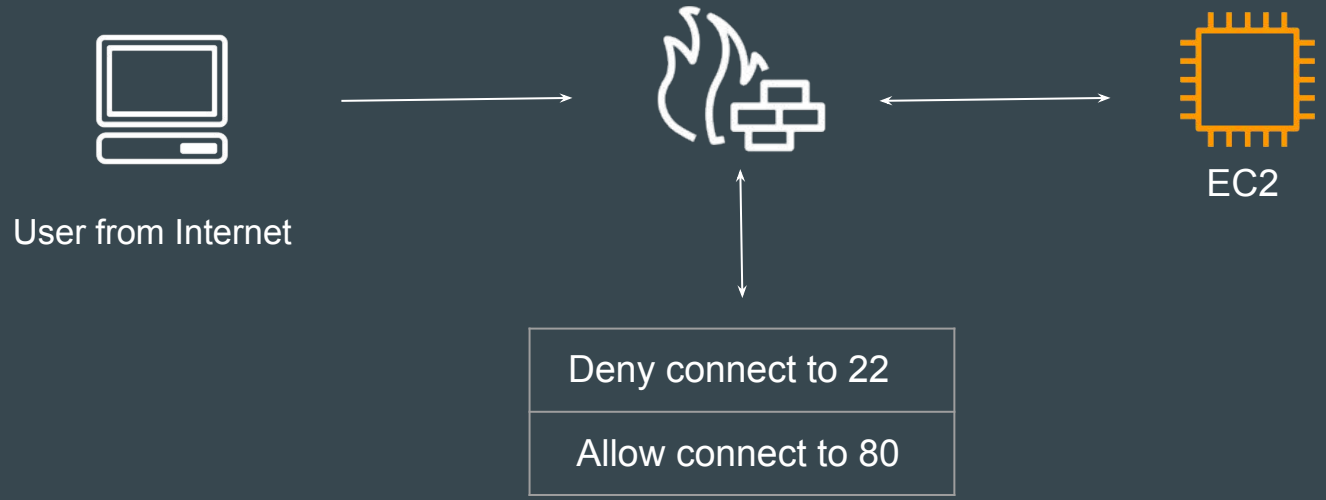
Basics of Firewall

Firewall is a network security system that monitors and controls incoming and outgoing network traffic based on predetermined security rules.

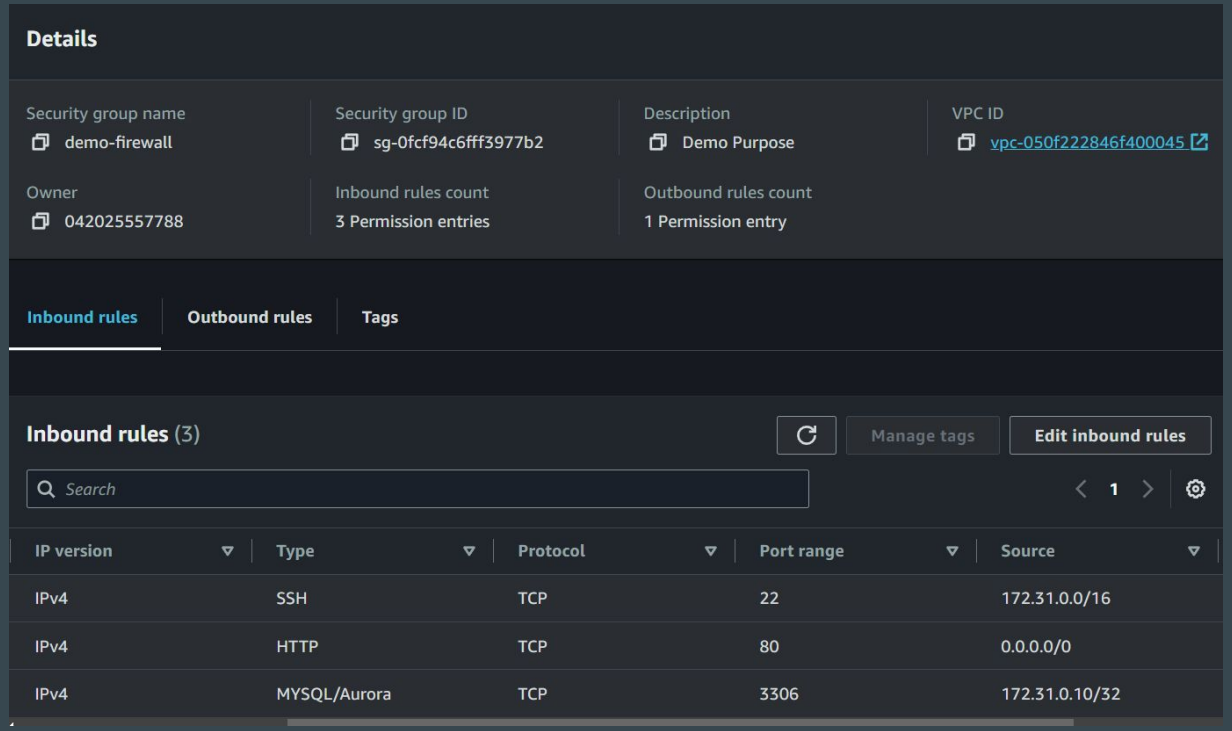


Firewall in AWS

A security group acts as a virtual firewall for your instance to control inbound and outbound traffic.

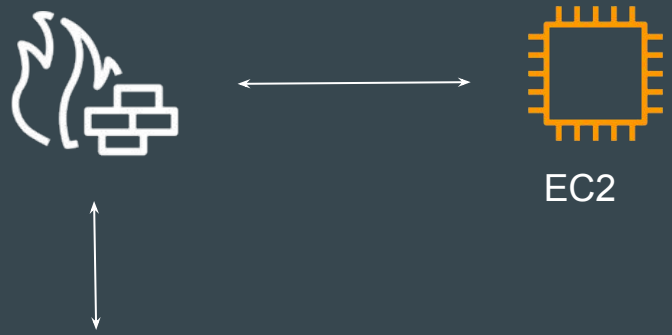


Sample Security Group with Rules



Inbound and Outbound Rules

Firewalls control both inbound and outbound connections to and from the server.



Inbound	Outbound
Allow 80 from 0.0.0.0/0	Allow 3306 to 172.31.10.50

Dealing with Documentation Code Updates - Terraform

Understanding the Challenge

Occasionally in the newer version of Providers, you will see some changes in the way you create a resource.

```
resource "aws_security_group" "allow_tls" {
  name        = "terraform-firewall"
  description = "Managed from Terraform"
}

resource "aws_vpc_security_group_ingress_rule" "allow_tls_ipv4" {
  security_group_id = aws_security_group.allow_tls.id
  cidr_ipv4         = "0.0.0.0/0"
  from_port         = 80
  ip_protocol       = "tcp"
  to_port           = 80
}
```

New Approach

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


  ingress {
    description      = "TLS from VPC"
    from_port        = 443
    to_port          = 443
    protocol         = "tcp"
    cidr_blocks      = ["10.77.32.50/32"]
  }
```

Old Approach

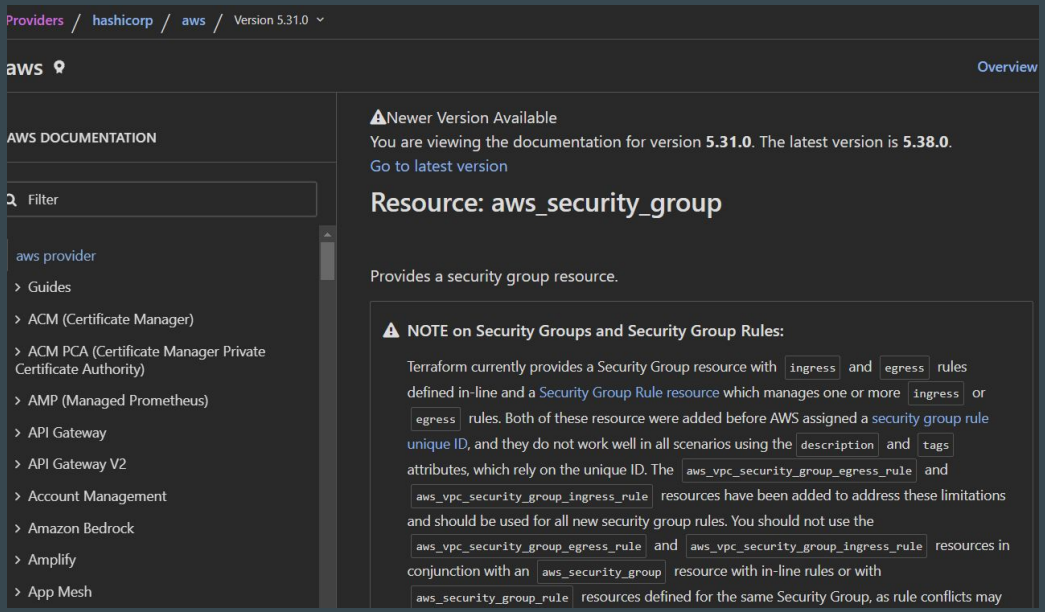
Points to Note

Just because a better approach is recommended, does NOT always mean that the older approach will stop working.

Organizations can continue to use the approach that suits best in it’s environment.

Switching to Older Provider Doc

You can always switch to the older version of provider documentation page to understand the changes.



Closing Pointers

For larger enterprises, it becomes difficult to upgrade their code base to the newer approach that provider recommends.

In such case, they stick with the appropriate provider version that supports the older approach of creating the resource.

Create Elastic IP with Terraform

Basics of Elastic IP in AWS

An Elastic IP address is a static IPv4 address in AWS.

You can create it and associate it with EC2 instance.

Aim of Today's Video

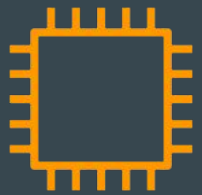
We are going to use Terraform to generate Elastic IP resource in AWS.

Attributes

Basics of Attributes

Each resource has its associated set of attributes.

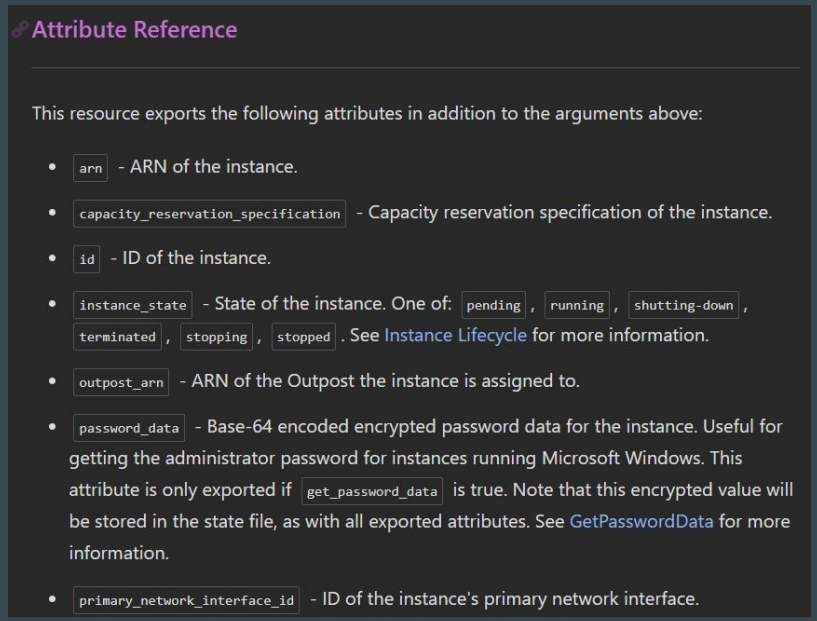
Attributes are the fields in a resource that hold the values that end up in state.



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

Points to Note

Each resource type has a predefined set of attributes determined by the provider.



Attribute Reference

This resource exports the following attributes in addition to the arguments above:

- `arn` - ARN of the instance.
- `capacity_reservation_specification` - Capacity reservation specification of the instance.
- `id` - ID of the instance.
- `instance_state` - State of the instance. One of: `pending`, `running`, `shutting-down`, `terminated`, `stopping`, `stopped`. See Instance Lifecycle for more information.
- `outpost_arn` - ARN of the Outpost the instance is assigned to.
- `password_data` - Base-64 encoded encrypted password data for the instance. Useful for getting the administrator password for instances running Microsoft Windows. This attribute is only exported if `get_password_data` is true. Note that this encrypted value will be stored in the state file, as with all exported attributes. See GetPasswordData for more information.
- `primary_network_interface_id` - ID of the instance's primary network interface.

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_vpc_security_group_ingress_rule" "allow_tls_ipv4" {
  security_group_id = aws_security_group.allow_tls.id
  cidr_ipv4         = "HOW-TO-ADD-ELASTIC-IP-ADDRESS-HERE"
  from_port         = 80
  ip_protocol       = "tcp"
  to_port           = 80
}
```

Analyzing the Attributes of EIP

We have to find which attribute stores the Public IP associated with EIP Resource.

Attribute Reference

This resource exports the following attributes in addition to the arguments above:

- `allocation_id` - ID that AWS assigns to represent the allocation of the Elastic IP address for use with instances in a VPC.
- `association_id` - ID representing the association of the address with an instance in a VPC.
- `carrier_ip` - Carrier IP address.
- `customer_owned_ip` - Customer owned IP.
- `id` - Contains the EIP allocation ID.
- `private_dns` - The Private DNS associated with the Elastic IP address (if in VPC).
- `private_ip` - Contains the private IP address (if in VPC).
- `public_dns` - Public DNS associated with the Elastic IP address.
- `public_ip` - Contains the public IP address.

Referencing Attribute in Other Resource

We have to find a way in which attribute value of “public_ip” is referenced to the cidr_ipv4 block of security group rule resource.

Elastic IP

Attribute	Value
public_ip	52.72.52.72

```
resource "aws_vpc_security_group_ingress_rule" "allow_tls_ipv4" {
  security_group_id = aws_security_group.allow_tls.id
  cidr_ipv4         = "REFERENCE-PUBLIC-IP-ATTRIBUTE-HERE"
  from_port         = 80
  ip_protocol       = "tcp"
  to_port           = 80
}
```

Cross Referencing Resource Attribute

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

Overall syntax:

<RESOURCE TYPE>.<NAME>.<ATTRIBUTE>

Cross Referencing Resource Attribute

We can specify the resource address with attribute for cross-referencing.

Elastic IP

Attribute	Value
public_ip	52.72.52.72

```
resource "aws_vpc_security_group_ingress_rule" "allow_tls_ipv4" {
  security_group_id = aws_security_group.allow_tls.id
  cidr_ipv4         = "aws_eip.lb.public_ip"
  from_port         = 80
  ip_protocol       = "tcp"
  to_port           = 80
}
```

String Interpolation in Terraform

${...}): This syntax indicates that Terraform will replace the expression inside the curly braces with its calculated value.

```
resource "aws_vpc_security_group_ingress_rule" "allow_tls_ipv4" {
  security_group_id = aws_security_group.allow_tls.id
  cidr_ipv4         = "${aws_eip.lb.public_ip}/32"
  from_port         = 80
  ip_protocol       = "tcp"
  to_port           = 80
}
```

Joke Time

Why did the Terraform attribute take a break?

...It was feeling over-referenced.

How did the Terraform attribute become a detective?

...It followed the resource trail.

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.

Sample Example

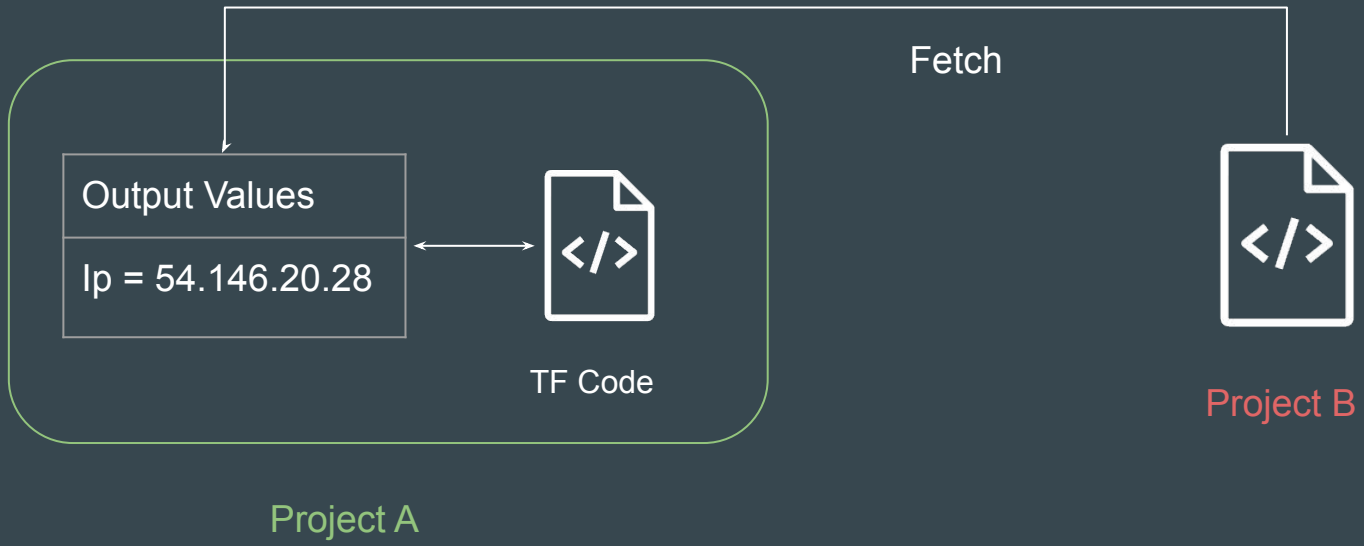
Use-Case:

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

Understanding the Challenge

Repeated static values in the code can create more work in the future.

Example: VPN IP needs to be whitelisted for 5 ports through Firewall Rules.

Port Number	CIDR Block	Description
80	101.0.62.210/32	VPN IP Whitelist
443	101.0.62.210/32	VPN IP Whitelist
22	101.0.62.210/32	VPN IP Whitelist
21	101.0.62.210/32	VPN IP Whitelist
8080	101.0.62.210/32	VPN IP Whitelist

Reference Screenshot

```
resource "aws_vpc_security_group_ingress_rule" "allow_tls_ipv4" {
  security_group_id = aws_security_group.allow_tls.id
  cidr_ipv4         = "101.0.62.210/32"
  from_port         = 80
  ip_protocol       = "tcp"
  to_port           = 80
}
```

Firewall Rule 1

```
resource "aws_vpc_security_group_ingress_rule" "allow_tls_ipv4" {
  security_group_id = aws_security_group.allow_tls.id
  cidr_ipv4         = "101.0.62.210/32"
  from_port         = 443
  ip_protocol       = "tcp"
  to_port           = 443
}
```

Firewall Rule 2

Better Approach

A better solution would be to define repeated static value in one central place.

Key	Value
vpn_ip	101.0.62.210/32

Central Location

```
resource "aws_vpc_security_group_ingress_rule" "allow_tls_ipv4" {
  security_group_id = aws_security_group.allow_tls.id
  cidr_ipv4         = "fetch-from-central-location"
  from_port         = 443
  ip_protocol       = "tcp"
  to_port           = 443
}
```

```
resource "aws_vpc_security_group_ingress_rule" "allow_tls_ipv4" {
  security_group_id = aws_security_group.allow_tls.id
  cidr_ipv4         = "fetch-from-central-location"
  from_port         = 8080
  ip_protocol       = "tcp"
  to_port           = 8080
}
```

Basics of Variables

Terraform input variables are used to pass certain values from outside of the configuration

Name	Value
vpn_ip	101.0.62.210/32
app_port	8080

Variable File


```
resource "aws_vpc_security_group_ingress_rule" "allow_tls_ipv4" {
  security_group_id = aws_security_group.allow_tls.id
  cidr_ipv4         = var.vpn_ip
  from_port         = var.app_port
  ip_protocol       = "tcp"
  to_port           = var.app_port
}
```

Benefits of Variables

1. Update important values in one central place instead of searching and replacing them throughout your code, saving time and potential mistakes.

2. No need to touch the core Terraform configuration file. This can avoid human mistakes while editing.

Variable Definitions File (TFVars)

Understanding the Base

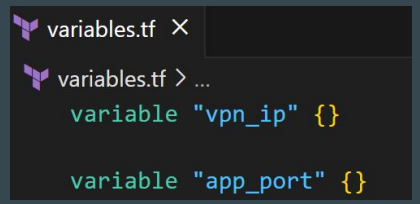
Managing variables in production environment is one of the very important aspect to keep code clean and reusable.

HashiCorp recommends creating a separate file with name of *.tfvars to define all variable value in a project.

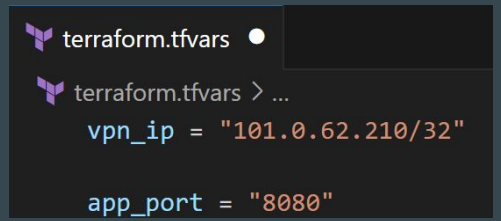
How Recommended Folder Structure Looks Like

1. Main Terraform Configuration File.
2. variables.tf file that defines all the variables.
3. terraform.tfvars file that defines value to all the variables.

Main Configuration File



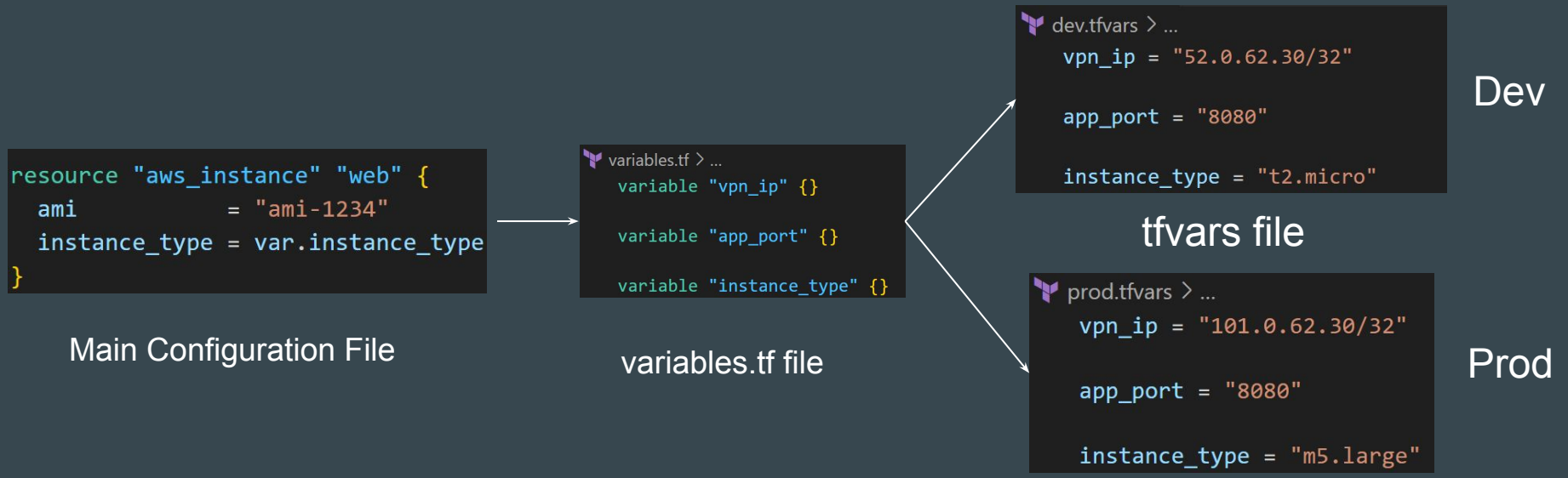
variables.tf File



terraform.tfvars file

Configuration for Different Environments

Organizations can have wide set of environments: Dev, Stage, Prod



Selecting tfvars File

If you have multiple variable definitions file (*.tfvars) file, you can manually define the file to use during command line.

```
C:\Users\zealv\kplabs-terraform>terraform plan -var-file="prod.tfvars"

Terraform used the selected providers to generate the following execution plan.
following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.web will be created
  + resource "aws_instance" "web" {
      + ami                                  = "ami-1234"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
```

Point to Note

If file name is terraform.tfvars → Terraform will automatically load values from it.

If file name is different like prod.tfvars → You have to explicitly define the file during plan / apply operation.

```
C:\Users\zealv\kplabs-terraform>terraform plan -var-file="prod.tfvars"

Terraform used the selected providers to generate the following execution plan.
following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.web will be created
  + resource "aws_instance" "web" {
      + ami                                  = "ami-1234"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
```

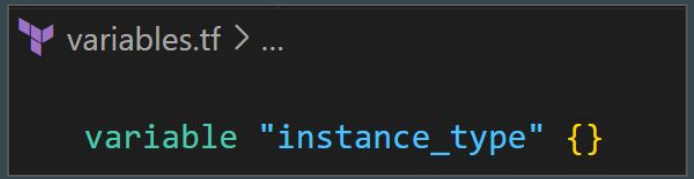
Approach to Variable Assignment

Understanding the Base

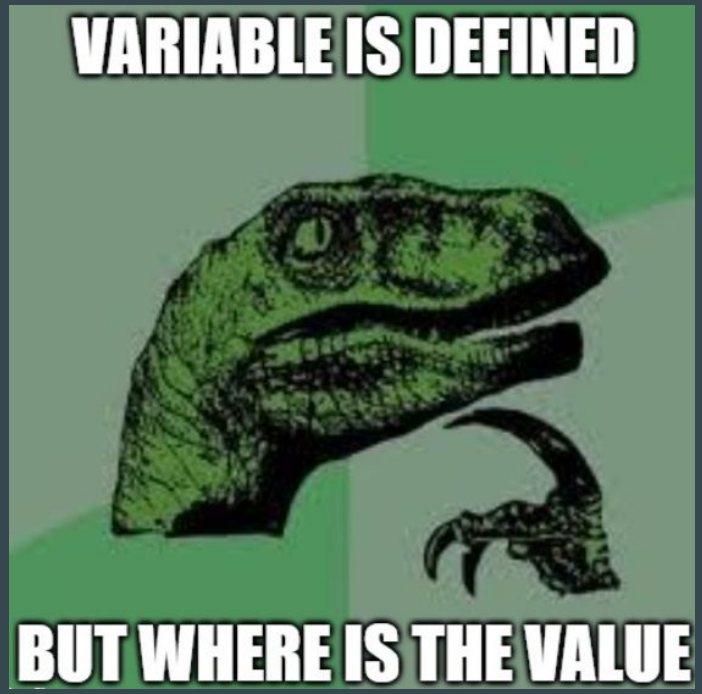
By default, whenever you define a variable, you must also set a value associated with it.

```
resource "aws_instance" "web" {
  ami           = "ami-1234"
  instance_type = var.instance_type
}
```

Main Configuration File



variables.tf


VARIABLE IS DEFINED
BUT WHERE IS THE VALUE

Add a Value in CLI

If you have not defined a value for a variable, Terraform will ask you to input the value in CLI Prompt when you run terraform plan / apply operation.

```
C:\Users\zealv\kplabs-terraform>terraform plan
var.instance_type
  Enter a value:
```

Declaring Variable Values

When variables are declared in your configuration, they can be set in a number of ways:

1. Variable Defaults.
2. Variable Definition File (*.tfvars)
3. Environment Variables
4. Setting Variables in the Command Line.

Variable Defaults

You can set a default value for a variable.

If there is no value supplied, the default value will be taken.

variables.tf > ...

```
variable "app_port" {
    default = "8080"
}
```

Variable Definition File (*.tfvars)

Variable Values can be defined in *.tfvars file.

prod.tfvars > ...

```
vpn_ip = "101.0.62.30/32"

app_port = "8080"

instance_type = "m5.large"
```

Setting Variable in Command Line

To specify individual variables on the command line, use the -var option when running the terraform plan and terraform apply commands:

```
C:\Users\zealv\kplabs-terraform>terraform plan -var="instance_type=m5.large"

Terraform used the selected providers to generate the following execution plan. Resource
following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.web will be created
  + resource "aws_instance" "web" {
      + ami                                  = "ami-1234"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_lifecycle                   = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "m5.large"
```

Setting Variable through Environment Variables

Terraform searches the environment of its own process for environment variables named TF_VAR_ followed by the name of a declared variable.

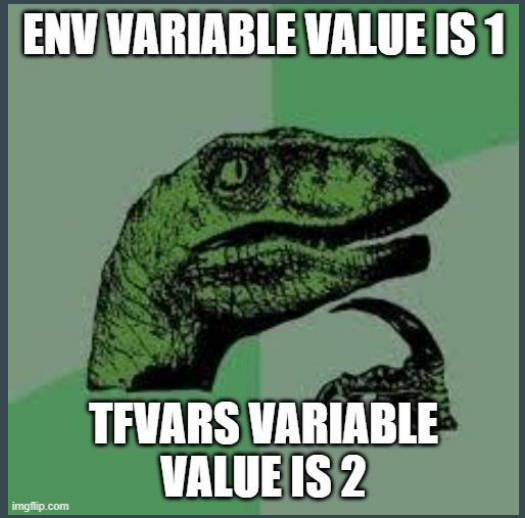
```
C:\Users\zealv\kplabs-terraform>echo %TF_VAR_instance_type%
t2.large

C:\Users\zealv\kplabs-terraform>terraform plan

Terraform used the selected providers to generate the following execution plan
following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.web will be created
  + resource "aws_instance" "web" {
      + ami                                  = "ami-1234"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
```

Variable Definition Precedence

Understanding the Base

Values for a variable can be defined at multiple different places.

What if values for a variable are different?



Simple Example

variable “instance_type” {}

1. Default Value is t2.micro
2. Terraform.tfvars value is “t2.small”
3. Environment Variable TF_VAR_instance_type = “t2.large”

Which value will Terraform take?

Variable Definition Precedence

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

1. Environment variables
2. The terraform.tfvars file, if present.
3. The terraform.tfvars.json file, if present.

4. Any *.auto.tfvars or *.auto.tfvars.json files, processed in lexical order of their filenames.

5. Any -var and -var-file options on the command line

Example 1

ENV Variable of TF_VAR_instance_type = “t2.micro”

Value in terraform.tfvars = “t2.large”

Final Result = “t2.large”

Example 2

1. ENV Variable of TF_VAR_instance_type = “t2.micro”
2. Value in terraform.tfvars = “t2.large”
3. terraform plan -var="instance_type=m5.large"

Final Result = “m5.large”

Data Types

Setting the Base

Data type refers to the type of value.

Depending on the requirement, you can use wide variety of values in Terraform configuration.

Example Data Type	Data Type
“Hello World”	String
7575	Number

Restricting Variable Value to Data Type

We can restrict the value of a variable to a data type.

Example:

Only numbers should be allowed in AWS Usernames.

```
variable "username" {
  type = number
}
```

Data Types in Terraform

Data Types	Description
string	a sequence of Unicode characters representing some text, like "hello".
number	A Numeric value
bool	a boolean value, either true or false
list	a sequence of values, like ["us-west-1a", "us-west-1c"]
set	a collection of unique values that do not have any secondary identifiers or ordering.
map	a group of values identified by named labels, like {name = "Mabel", age = 52}.
null	a value that represents absence or omission.

Data Type - List

List Data Type

Allows us to store collection of values for a single variable / argument.

Represented by a pair of square brackets containing a comma-separated sequence of values, like ["a", 15, true].

Useful when multiple values needs to be added for a specific argument

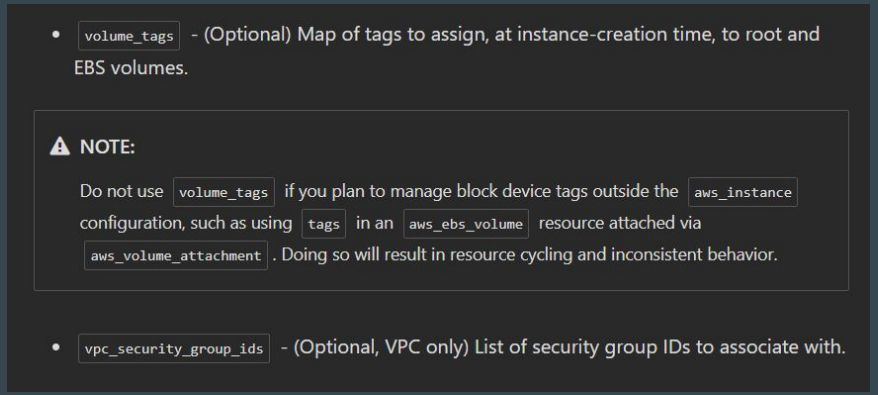
```
variable "my-list" {
  type = list
  default = ["mumbai","bangalore","delhi"]
}
```

Data Type and Documentation

Arguments for a resource requires specific data types.

Some argument requires list, some requires map and so on.

The details of data type expected for an argument is mentioned in documentation.



- `volume_tags` - (Optional) Map of tags to assign, at instance-creation time, to root and EBS volumes.

> ⚠ **NOTE:**
>
> Do not use `volume_tags` if you plan to manage block device tags outside the `aws_instance` configuration, such as using `tags` in an `aws_ebs_volume` resource attached via `aws_volume_attachment`. Doing so will result in resource cycling and inconsistent behavior.

- `vpc_security_group_ids` - (Optional, VPC only) List of security group IDs to associate with.

Use-Case 1: List Data Type

EC2 instance can have one or more security groups.

Requirement:

Create EC2 instance with 2 security groups attached.

Specify the Type of Values in List

We can also specify the type of values expected in a list.

```
variable "my-list" {
  type = list(number)
  default  = ["1","2","3"]
}
```

Map - Data Type

Map Data Type

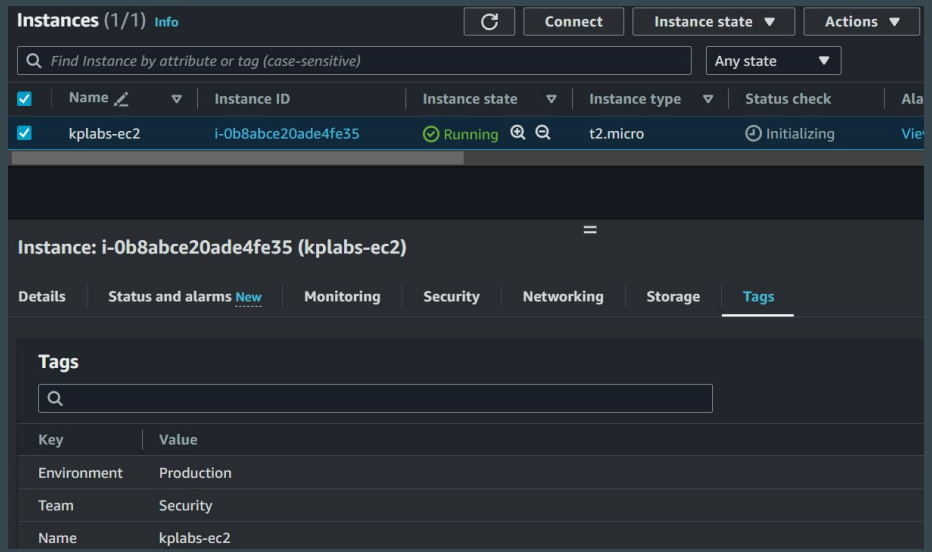
A map data type represents a collection of key-value pair elements

```
variable "instance_tags" {
  type = map
  default = {
    Name = "app-server"
    Environment = "development"
    Team = "payments"
  }
}
```

Use-Case of Map

We can add multiple tags to AWS resources.

These tags are key-value pairs.



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:

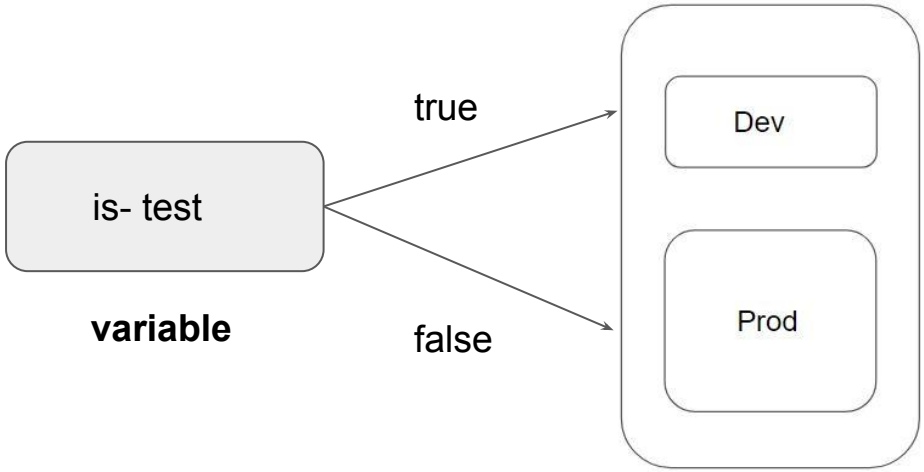
```
condition ? true_val : false_val
```

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.

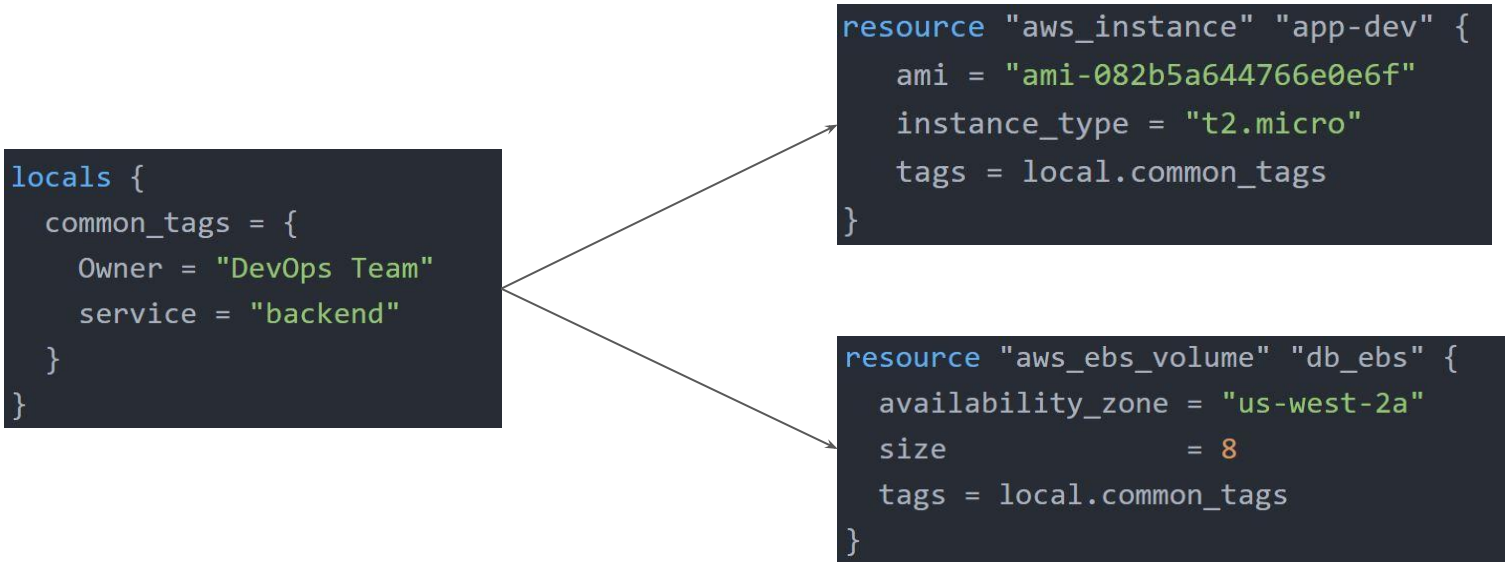


Local Values

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.



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

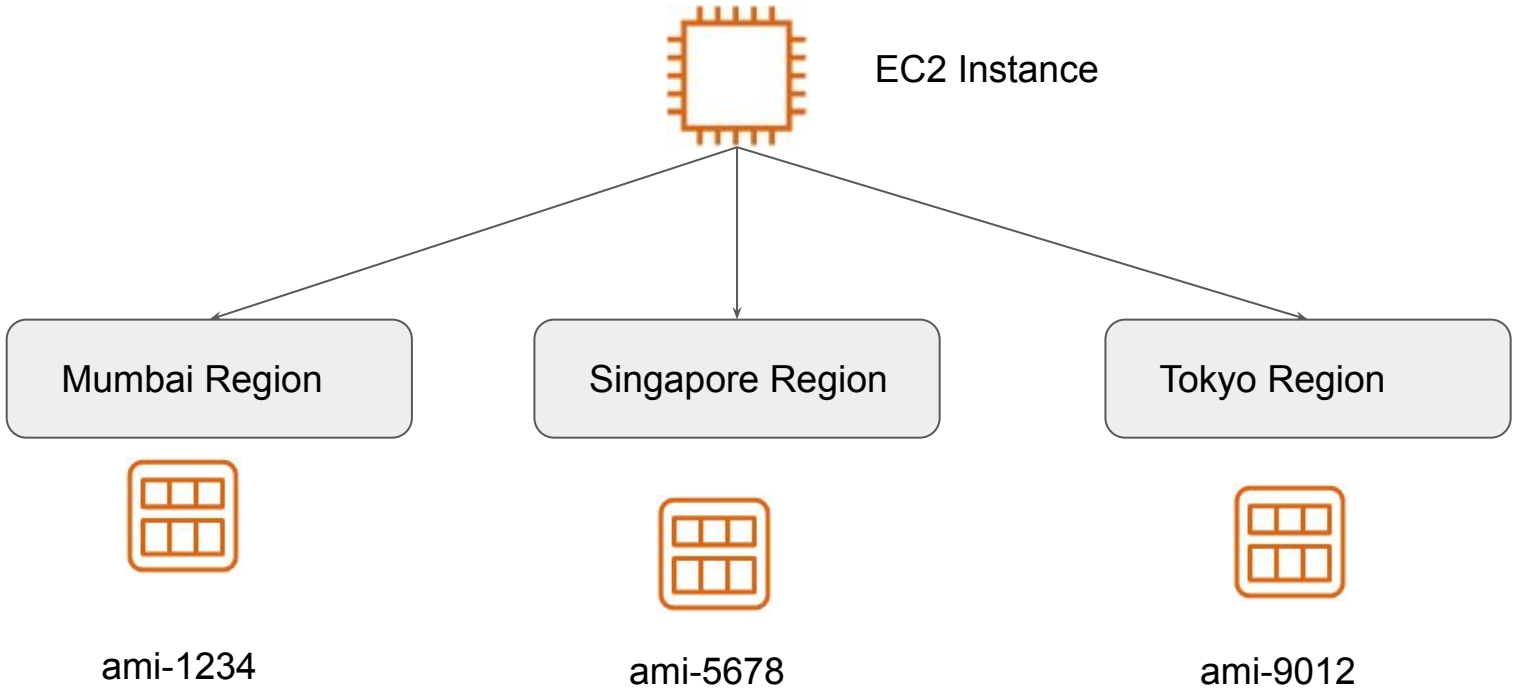
- Numeric
- String
- Collection
- Encoding
- Filesystem
- Date and Time
- Hash and Crypto
- IP Network
- Type Conversion

Data Sources

Terraform in detail

Overview of Data Sources

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



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

Do you have a special talent?

Me:

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.
```

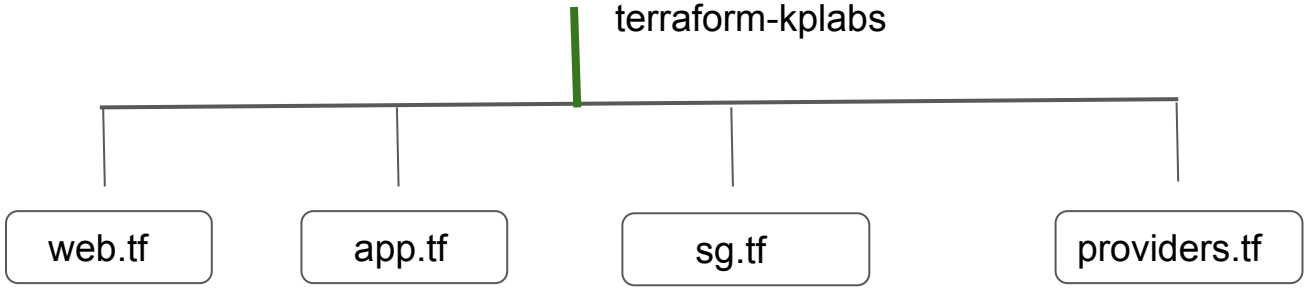
Load Order & Semantics

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.



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"]
  }
}
```

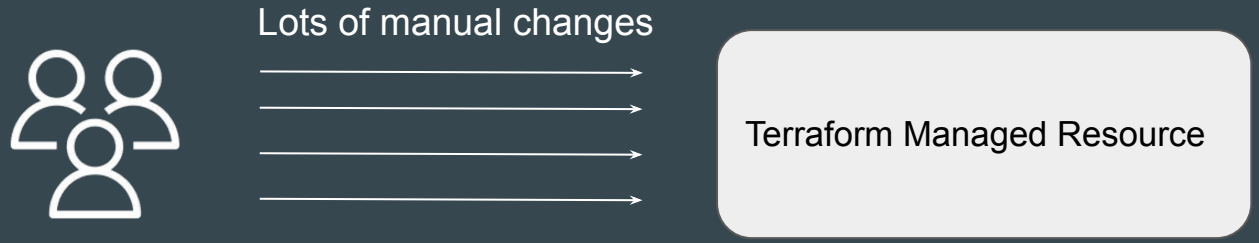
Terraform Taint

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)

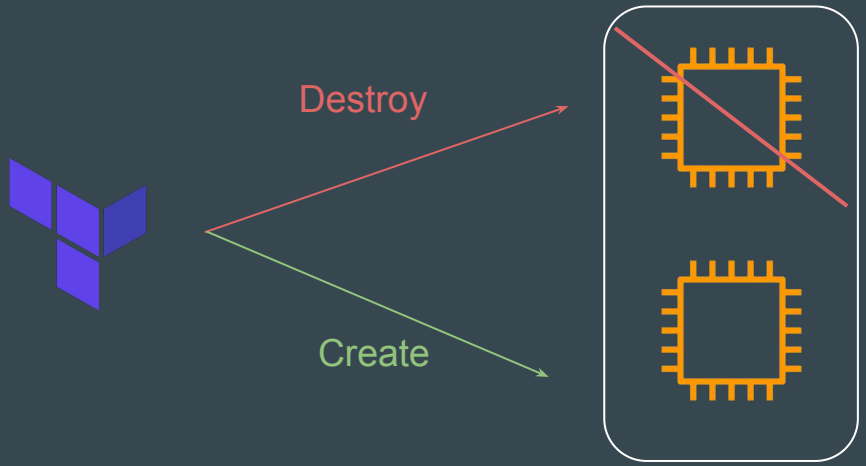
Two ways to deal with this: Import Changes to Terraform / Delete & Recreate the resource



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
}
```

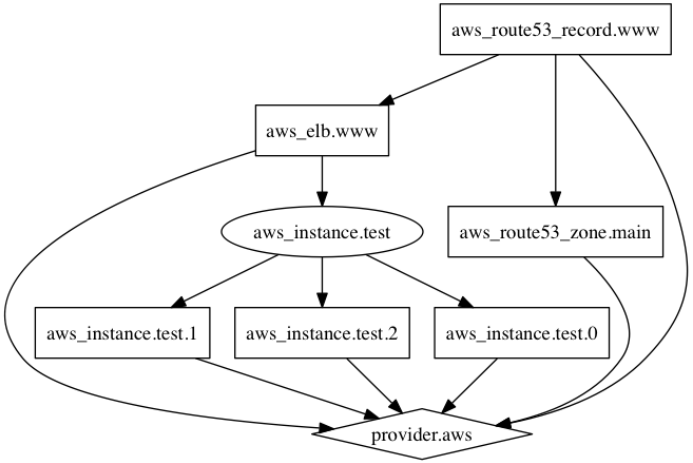
Terraform Graph

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.

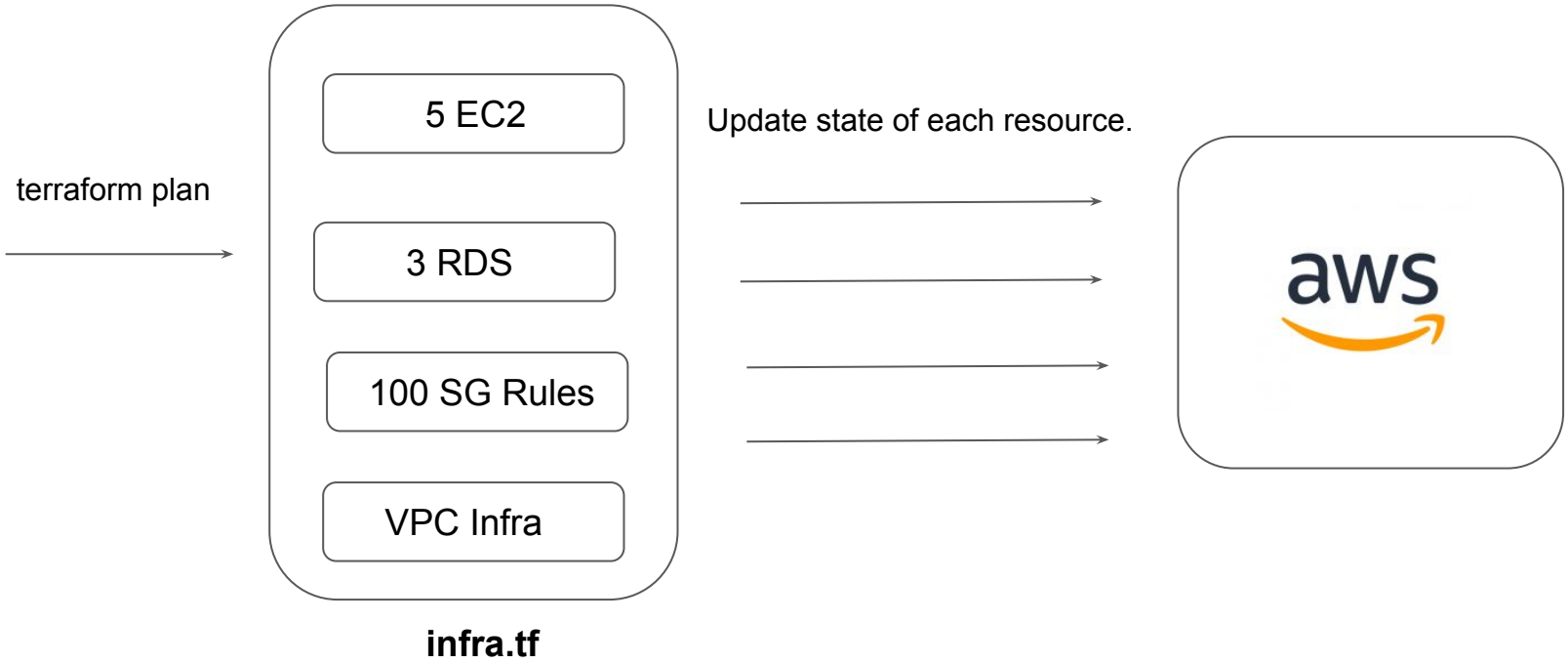
```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
    }
  }
}
```

Dealing with Larger Infrastructure

Terraform in detail

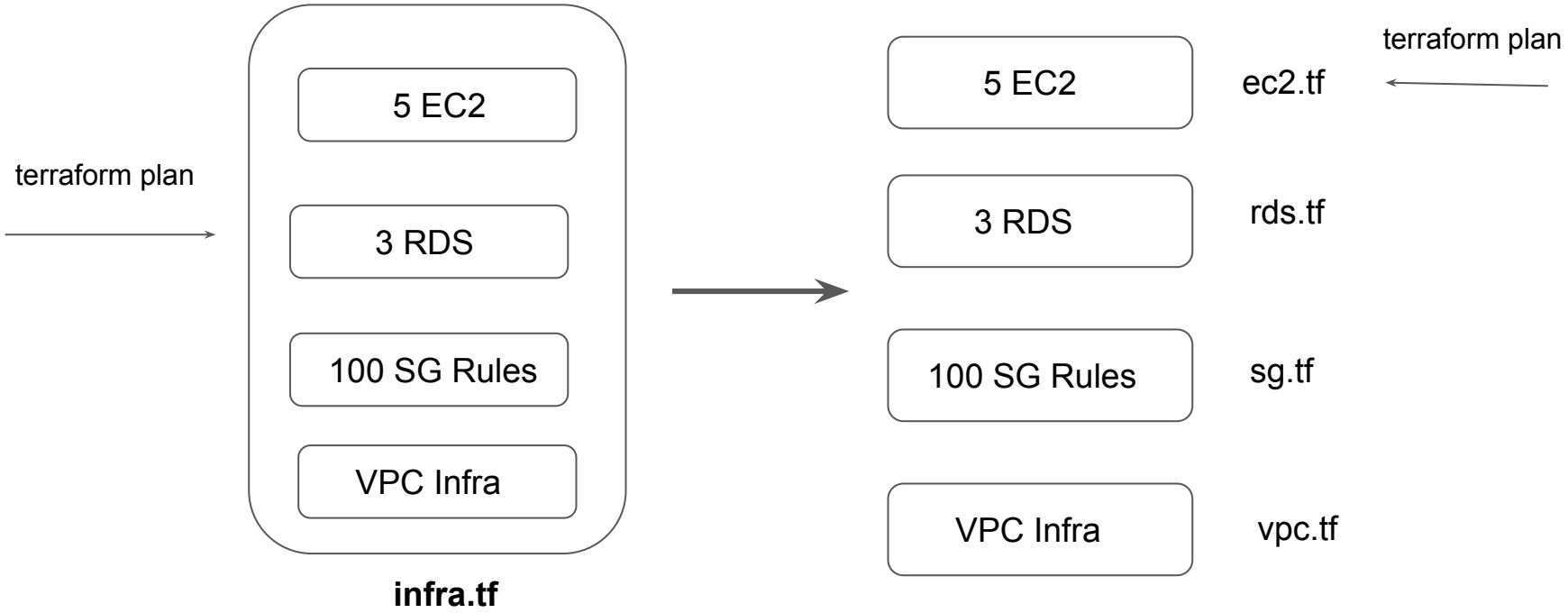
Challenges with Larger Infrastructure

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



Dealing With Larger Infrastructure

Switch to smaller configuration were each can be applied independently.



Slow Down, My Man

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

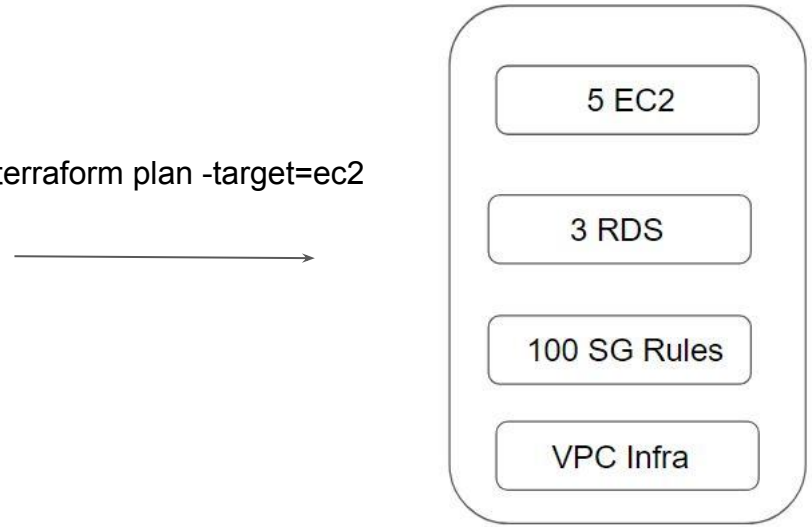
This can be achieved with the -refresh=false flag

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

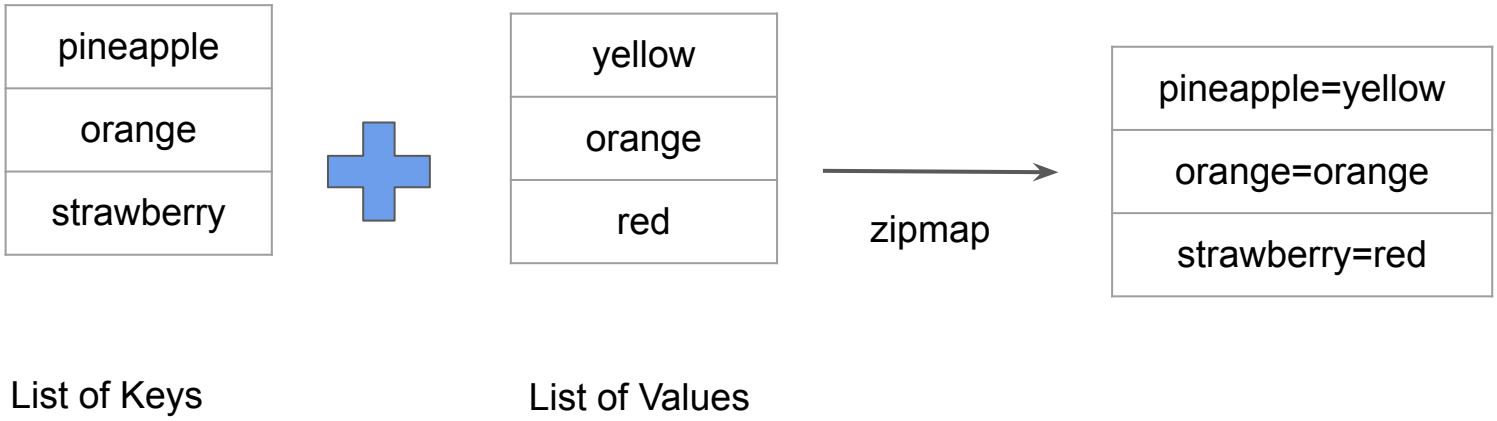


Zipmap

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

```
'''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.

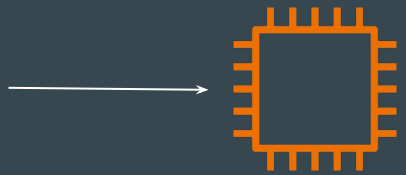
Destroy and re-create resources whose arguments have changed but which cannot be updated in-place due to remote API limitations.

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

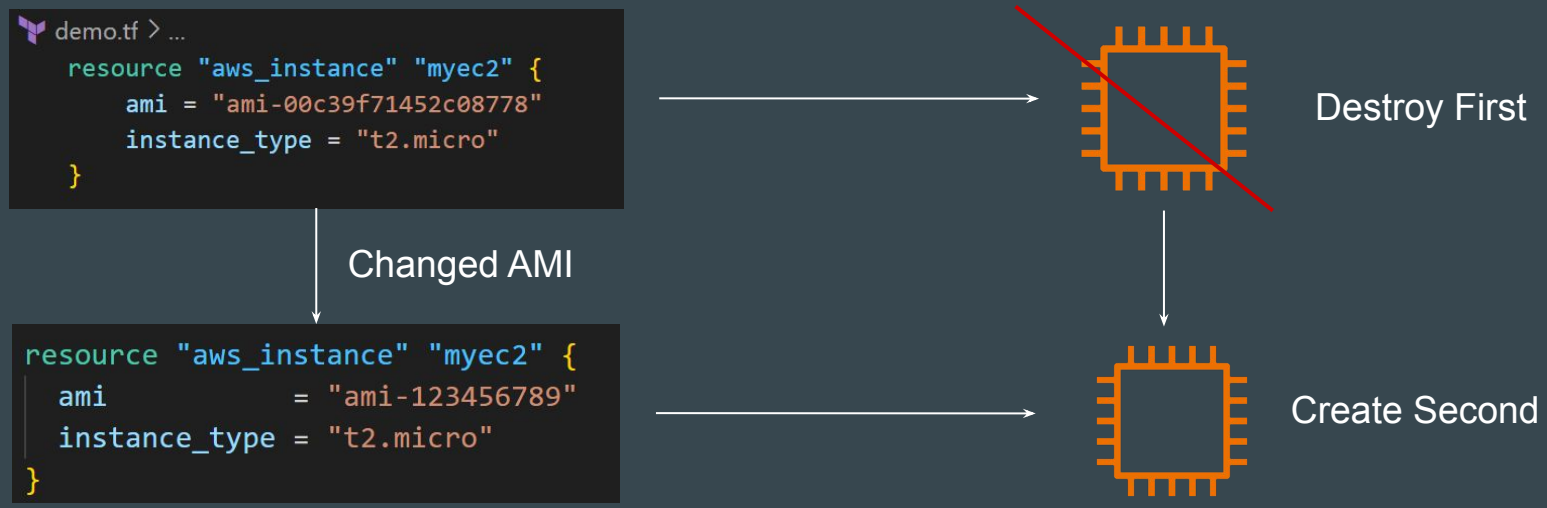
Replaces the resource when any of the referenced items change.

```
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.



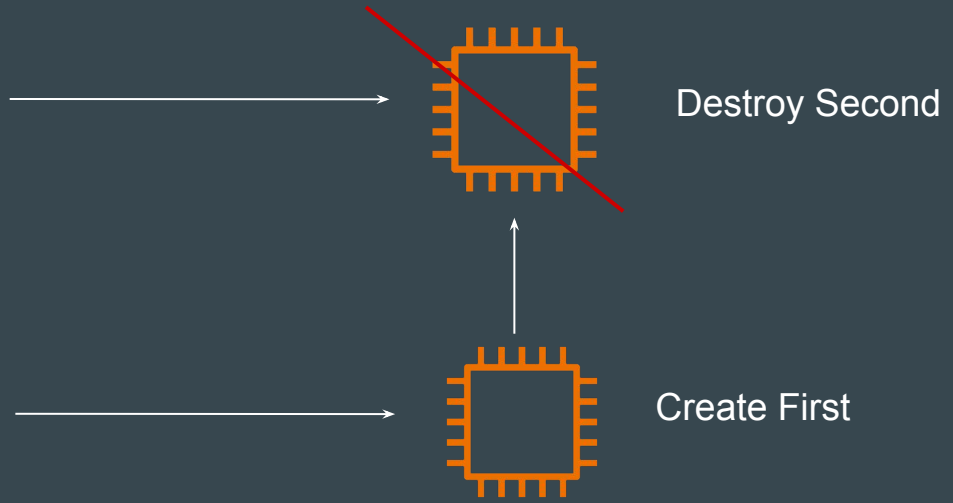
Create Before Destroy Argument

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.

```
resource "aws_instance" "myec2" {
    ami = "ami-053b0d53c279acc90"
    instance_type = "t2.micro"

    lifecycle {
      create_before_destroy = true
  }
}
```

Changed AMI

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

    lifecycle {
      create_before_destroy = true
  }
}
```



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.

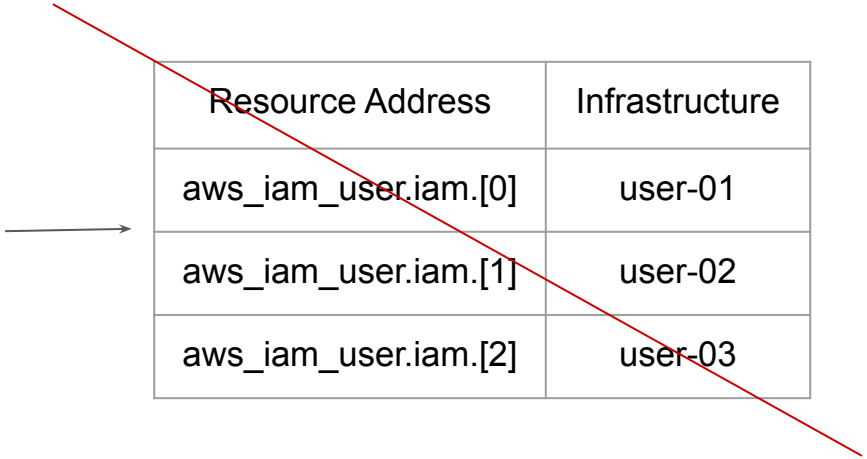
```
variable "iam_names" {
  type = list
  default = ["user-01","user-02","user-03"]
}

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.

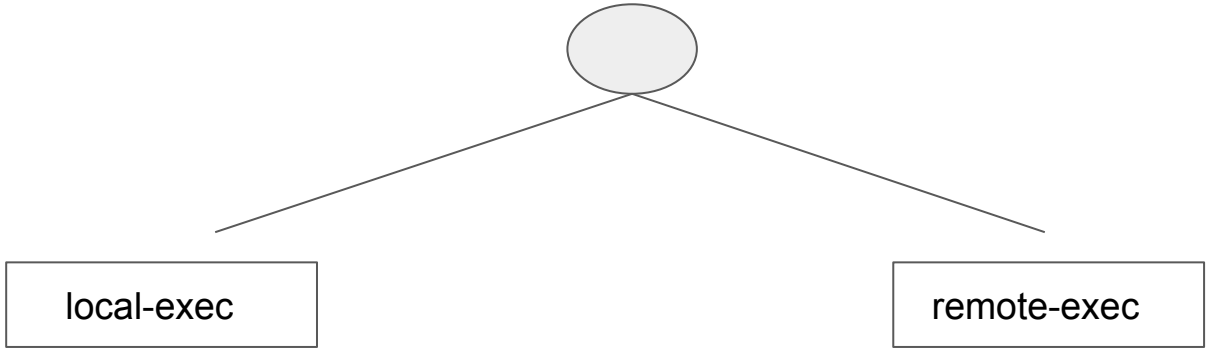
Types of Provisioners

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 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
  }
}
```