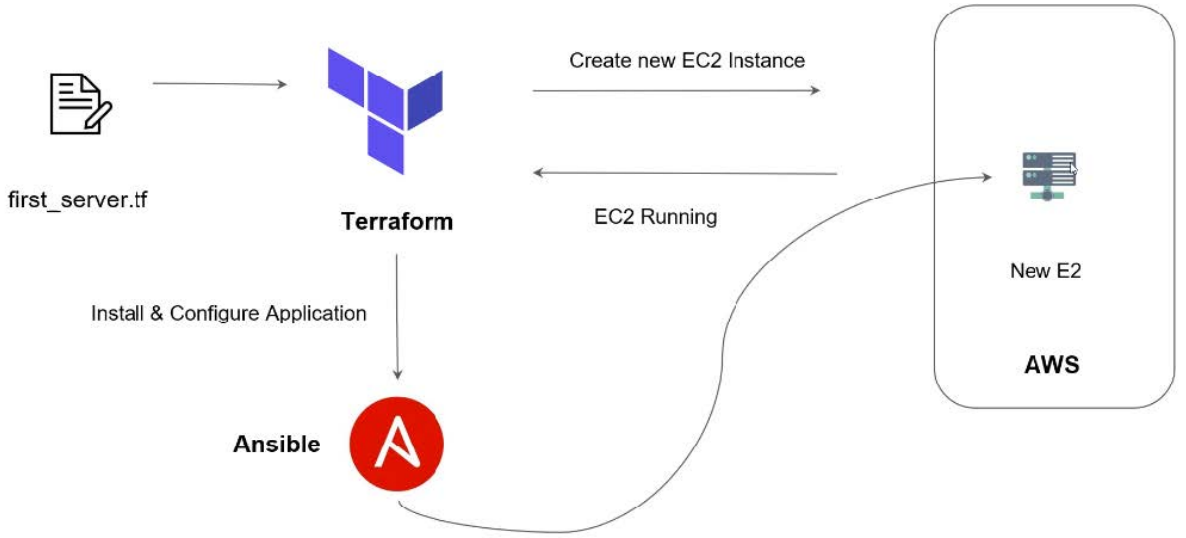
0. Getting Started

Choosing correct IAC tool: **Terraform, CloudFormation, heat, Ansible, SaltStack, Chef/Puppet**

Configuration Management Tools	Infrastructure orchestration tools
primarily designed to install and manage software on existing servers like Ansible, Chef/Puppet. Some degree of infrastructure provisioning as well	provision servers and infrastructure by themselves like Terraform and CF



Terraform:

- Support multiple platforms
- simple config language
- easy integration with configuration tools like Ansible
- Easily extensible with help of plugins
- Free !!

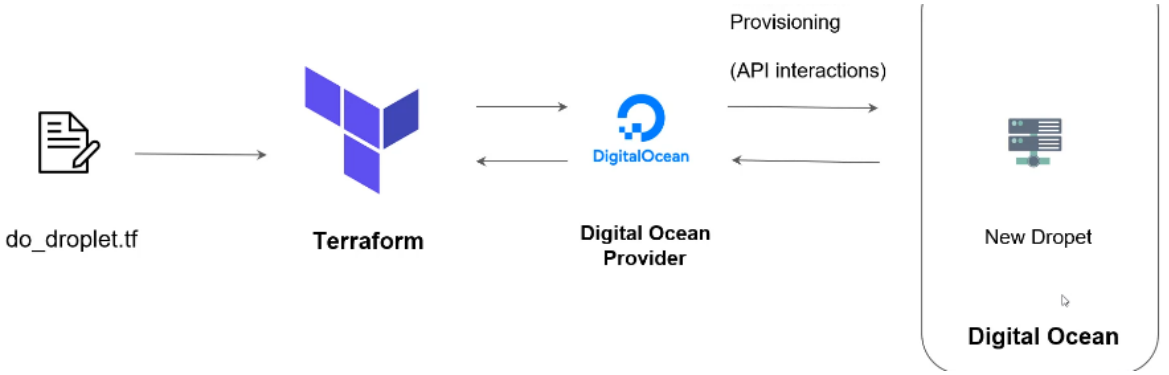
0.a Important Pointers

- Terraform ensures that deployed infrastructure is based on desired state
- If there is a difference between current and desires, **tf plan** presents description of **changes necessary to achieve desired state**
- **If parameters are force replacement like** if changing EC2 instances change ebs volumes as true or false, then in that case, the old instance will be destroyed and the new one will be created as nothing will be modified. Ex. t2.micro to t3.medium
- If parameters are not described in the configuration, and if they are changed manually, the terraform state will be changed but there will be no terraform plan changes as plan changes only come for parameters described in the configuration file

0.b Part 1 Pointers

<u>Part 1:</u>

- ***A provider***
  - responsible for understanding API interactions and exposing resource
  - correspond to one cloud or on-premises infrastructure platform
  - explicitly set a specific version of the provider within the provider block
  - to upgrade of latest version of each provider , run *terraform init -upgrade*



  - multiple provider instance with help of alias, without alias is default
- ***Terraform init***
  - initialize a working directory containing Terraform configuration files
  - search for module blocks and source code for referenced modules is retrieved from locations given in source arguments
  - must initialize provider first
  - it downloads and installs provider plugin to be later executed
- ***Terraform Plan***

    - create an execution plan
    - will NOT modify configuration
    - performs a refresh, unless disabled and then determines what actions are necessary to achieve desired state
    - step to check whether the execution plan for a set of changes matches your expectations without making any changes to real state
- ***Terraform Apply***
    - used to apply changes required to reach desired state of configuration
    - also write data to terraform.tfstate file
    - once completed, resources are modified
- ***Terraform Refresh***
    - used to reconcile the state Terraform knows about (via it's state file) with real-world infrastructure
    - doesn't modify the infrastructure but modify the state file
- ***Terraform Destroy***
    - used to destroy terraform managed infrastructure
- ***Terraform Format***
    - terraform fmt used to rewrite Terraform configuration files to a canonical format and style
- ***Terraform Validate***
    - validates configuration files in directory
    - checks that verify whether a configuration is syntactically valid and is useful for general verification of reusable modules
    - safe to run this command automatically
    - run before terraform plan
    - requires initialized working directory with any referenced plugins and modules installed
- ***Terraform Provisioners***
    - can be used to model specific actions on local machine or on remote machine in order to prepare servers or other infrastructure objects for service
    - last resort
    - are inside resource block
    - local-exec: allow us to invoke local executable after resource is created
        - ex, can be used to manually update glue job with more config parameters
        - Ansible Playbook
    - remote-exec: allow us to invoke scripts directly on remote server

    - Others: chef, connection, file, habitat, null_resource, salt-masterless
- Difference between provider and plugin:
    - Each provider can have their own sets of plugins. AWS provider will have its plugins, Azure provider will have different set of plugins.
    - Terraform providers and provisioners are provided via plugins. Each plugin exposes an implementation for a specific service, such as AWS, or provisioner, such as bash

0.c Part 2 Pointers

- ***Debugging terraform***
    - detailed logs that can be enabled by setting TF_LOG environment variable to any value
    - can set TF_LOG to one of log levels TRACE, DEBUG, INFO, WARN, ERROR
    - For persist logged output set TF_LOG_PATH
- ***Import***
    - allow you to take resources created by other means and bring it under terraform
    - can import only resources into state and doesn't generate configuration
    - It is important to run configuration file manually prior running terraform import command

```
terraform import aws_instance.myec2 instance-id
```

- ***Local Values***
    - assign a name to an expression allowing it to be used multiple times within a module without repeating it
    - Local can't refer to itself or to a variable that refers back to it
    - recommended to group together logically-related local values
- ***Data Types***
    - string, list, map, number

-

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

- ***Workspaces***
  - to have different set of environment variables associated
  - allow *multiple state files of a single configuration*
- **Modules**
  - centralize resources and can call out TF files when required
  - Root module (atleast one module, in main working directory .tf files)
  - Root module can call child modules which include child module resources into configuration and call is done with the help of source parameter
- ***Output Values***
  - calling modules can't access their attributes directly
  - child module can declare output values to selectively export certain values to be accessed by calling module
- ***Suppressing values in CLI output***
  - An output can be marked as containing sensitive material using optional sensitive argument
  - It only list it as *sensitive* at the end of terraform apply but stills records the sensitive information in the state

0.d Part 3 Pointers

- ***Functions***
  - include number of built-in functions that you can use to transform and combine values
  - element: retrieves a single element from a list
  - lookup:retrieves the value of a single element from a map, given its key. If the given key does not exist, the given default value is returned instead.

```
element(list,index)
lookup(map, key, default)
```

    - count: scale resources by simply incrementing a number
    - count.index: to modify the configuration of each resource (like changing the name of multiple resources)
- ***Lock***
    - lock state for all operations that could write state
    - force unlock to manually unlock the state
- ***Resource***
    - describe one or more infrastructure objects such as virtual networks, compute instances etc.
    - declares a resources of given type (aws_instance) with a given local name (web)
- ***Sentinel***
    - embedded policy as code framework integrated with HashiCorp Enterprise products
        - verify EC2 instance has tags
        - verify if S3 bucket has encryption enabled
- ***Sensitive Data in State File***
    - Cloud version always encrypts state at rest and protects it with TLS in transit
    - maintains the history of state changes
    - S3 backend supports encryption at rest when encrypt option is enabled
    - Storing credentials as part of environment variables is better approach than hard coding it in the system
- ***Remote Backend***
    - stores terraform state and maybe used to run operations in Terraform cloud
    - When using full remote operations, operations like terraform plan or apply can be executed in terraform cloud's run environment with log output streaming to local terminal

0.e Part 4 Pointers

- ***Graph***
    - to generate visual representation of either configuration or execution plan

    - output is in DOT format which can be easily converted to an image
- ***Splat Expressions***
    - allow us to get a list of all attributes (arguments and commands)
    - [*]
- ***Provider configuration***
    - not mandatory
- ***Output***
    - used to extract value of an output variable from state file
- Unlock
    - TF will lock state for all operations that could write state
    - Not all backend supports locking'

```
terraform force-unlock LOCK_ID [DIR]
```

- Automation, Versioning and Reusability (IaC benefits)
- Sentinel is a proactive service
- Terraform refresh doesn't modify infrastructure but it modifies the state file
- Slice function is not part of string function unlike join, split, chomp
- not mandatory to include module version argument while pulling code from terraform registry
- Overuse of dynamic blocks can make configuration hard to read and maintain
- Terraform apply can change, destroy and provision resources but can't import any resource

0.f Part 5 Pointers

Terraform Enterprise vs Terraform Cloud

- Enterprise provides following advantages:
    - SSO
    - Auditing
    - Private Data Center Networking
    - Clustering
- Team and Governance is not available for Cloud Free (paid)

- Variables with undefined values
    - Not result in error
    - terraform will ask you to supply the value associated with them
    - variable custom_var{}
- Environment variables

    - can be used to set variables
    - must be in format TF_VAR_name
- Structural Data type
    - have multiple values of several distinct types to be grouped together
    - object, tuple
    - A collection of named attributes that each have their own type.
      object({<ATTR NAME> = <TYPE>, ... })
      object({ name=string, age=number })
- Backend configuration
    - are configured directly
    - after configuration, it needs to be initialized
    - FIRST TIME
        - *moving from no defined backend to explicitly configuring one, terraform will give an option to migrate state to new backend*
        - *let you adopt backend without losing any existing state*
    - PARTIAL TIME
        - *not mandatory to mention all arguments and can be skipped*
        - *remaining config arguments must be provided as part of initialization process*
        -
```
terraform init \
    -backend-config="address=demo.consul.io" \
    -backend-config="path=example_app/terraform_state" \
    -backend-config="scheme=https"
```
- Terraform Taint
    - manually marks Terraform managed as tainted, forcing it to be destroyed and recreated on next apply
    - *next plan* will show resource will be destroyed and recreated and *next apply* will implement this change
    - terraform taint [options] address - can go more granular
- Provisioners:
    - local-exec: allow us to invoke local executable after resource is created
        - ex, can be used to manually update glue job with more config parameters
        - Ansible Playbook
    - remote-exec: allow us to invoke scripts directly on remote server
        - ssh, winrm
        - we have an inline (list of commands to run)

    - Others: chef, connection, file, habitat, null_resource, salt-masterless
    - Failure behaviour:
        - By default provisioners that fail will cause terraform apply to fail
        - on_failure setting
            - continue: ignore error and continue with creation/destruction
            - fail: Raise an error and stop applying, can taint the resource
    - **Creation-Time Provisioner:** only run during creation, not during updating or any other lifecycle. If it fails then resource is tainted
    - **Destroy-Time Provisioner:** are run before the source is destroyed
- Variables
    - values associated with variables can be defined via CLI as well as in tfvars file

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

- Variable Definition Precedence
    - Env variables
    - terraform.tfvars file
    - terraform.tfvars.json file
    - Any *.auto.tfvars or *auto.tfvars.json files
    - Any -var and -var-file options
- Local backend
    - stores state on local filesystem and locks that state using system APIs and performs operations locally
    - by default, local backend

0.g Part 6 Pointers

- Fetching values from map

```
var.ami_ids["mumbai"]

variable "ami_ids" {
default = {
    "mumbai" = "image-a"
    "germany" = "image-b"
    }
}
```

- TF and GIT
    - .gitignore should be configured to ignore certain terraform files that contain sensitive info

        - terraform.tfstate file
        - *.tfvars
    - repo can be used by prefixing the address with special git:: prefix and can use any valid git URL
    - will clone and use default branch (referenced by (HEAD))
        - can override using ref argument ?ref=v1.2.0
        - can be branch and tag names
- Dependency types:
    - Implicit
        - TF can automatically find references of the object and create an implicit ordering requirement between 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
          }
          ```
        - Here EIP is created first and then instance
    - Explicit
        - It is necessary only resource relies on some other resource behaviour but doesn't access any of that resource data in it's arguments
        - ```
          resource "aws_s3_bucket" "example" {
            acl    = "private"
          }

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

-

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.

- 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
-

0.h Extra Pointers latest

- Requirements for publishing module in terraform Registry
  - Published modules support versioning, automatically generate documentation, allow browsing version histories, show examples and READMEs, and more. We recommend publishing reusable modules to a registry.
  - Public modules are managed via Git and GitHub. Publishing a module takes only a few minutes. Once a module is published, you can release a new version of a module by simply pushing a properly formed Git tag.
    - **Github:** The module must be on GitHub and must be a public repo. This is only a requirement for the public registry.
    - **Named**: **terraform-<PROVIDER>-<NAME>**
    - Repo 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.
- **List:**

    - sequence of values identified by consecutive whole numbers starting with zero.
    - Terraform reserves some additional names that can no longer be used as input variable names for modules. These reserved names are:

```
count
depends_on
for_each
lifecycle
providers
source
```

- Air Gapped:
    - If terraform needs to be installed in an environment without internet access, the installation is referred to as air-gapped
- Terraform Enterprise supports the following data storage:
    - PostgresSQL
    - Any S3-compatible object storage service, GCP Cloud Storage or Azure blob storage meets Terraform Enterprise's object storage requirements.

1. Review Guide (HashiCorp Official)

- [x] ~~Understand Infrastructure as Code (IaC) concepts~~
- [x] ~~Understand Terraform's purpose (vs other IaC)~~
- [x] ~~Understand Terraform basics~~
- [x] ~~Use the Terraform CLI (outside of core workflow)~~
- [x] ~~Interact with Terraform modules~~
- [x] ~~Navigate Terraform workflow~~
- [x] ~~Implement and maintain state~~
- [x] ~~Read, generate, and modify configuration~~
- [x] ~~Understand Terraform Cloud and Enterprise capabilities~~

2. Authentication Terraform

Authentication and Configuration with AWS

1. Parameters in the provider configuration
2. Environment variables
3. Shared credentials files
4. Shared configuration files
5. Container credentials
6. Instance profile credentials and region

3. Terraform basic CLI commands

```
terraform init
```

This command performs several different initialization steps in order to prepare the current working directory for use with Terraform

```
terraform plan
```

The terraform plan command creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure.

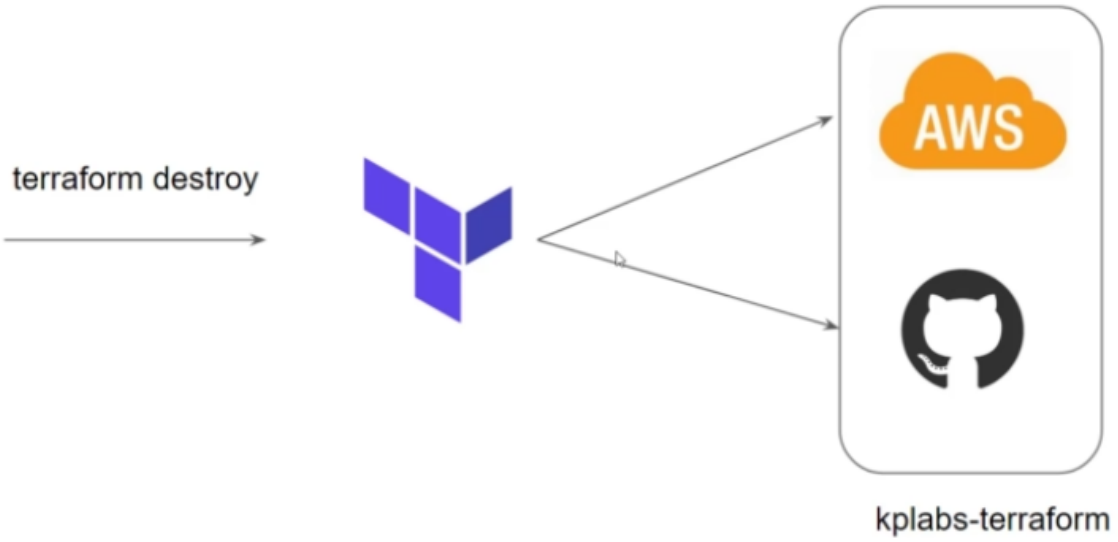
- Reads the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date
- Compares the current configuration to the prior state and noting any differences
- Proposes a set of change actions that should, if applied, make the remote objects match the configuration.

```
terraform apply
```

executes the actions proposed in a Terraform plan.
terraform apply [options] [plan file]

```
terraform destroy
```

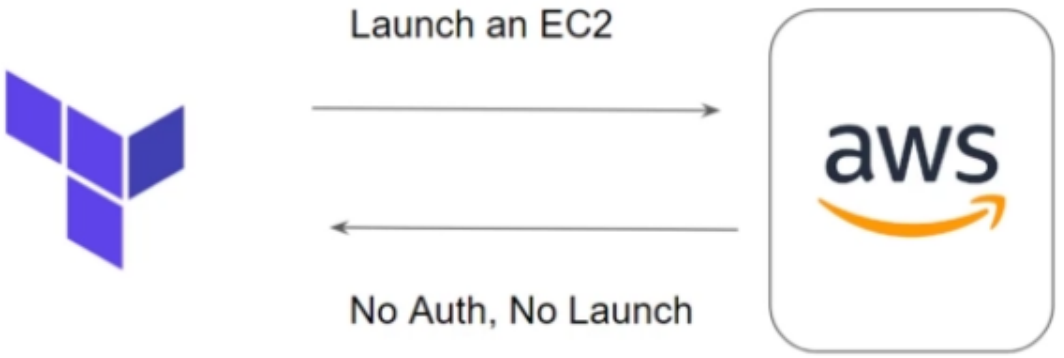
Destroy or deletes all resources as mentioned in the plan



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

```
terraform destroy -target <resource>.<local-resource-name>
```

4. EC2 with Terraform



- Authentication
- Region where resource needs to be launched
- Which resource needs to be launched

```
# Configure the AWS Provider
provider "aws" {
  region = "ca-central-1"
  access_key = "access_key"
  secret_key = "secret_key"
}
resource "aws_instance" "first_ec2" {
    ami = "ami-0f6b3aca8444b4f04"
    instance_type = "t2.micro"
}
```

```
terraform init
terraform plan
terraform apply
```

```
    }

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

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

  Enter a value: yes

aws_instance.first_ec2: Creating...
aws_instance.first_ec2: Still creating... [10s elapsed]
aws_instance.first_ec2: Creation complete after 15s [id=i-08846fac5325debc1]

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

```
terraform destroy
```

```
    }

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

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

aws_instance.first_ec2: Destroying... [id=i-08846fac5325debc1]
aws_instance.first_ec2: Still destroying... [id=i-08846fac5325debc1, 10s elapsed]
aws_instance.first_ec2: Still destroying... [id=i-08846fac5325debc1, 20s elapsed]
aws_instance.first_ec2: Destruction complete after 30s

Destroy complete! Resources: 1 destroyed.
```

5. Providers and Resources

Provider - support multiple providers - AWS, GCP, Azure, Kubernetes, Oracle etc. (2032 providers)

Initialization Phase - to run terraform init which in-turn will download plugins associated with the provider

```
terraform init
```

E:\Terraform\.terraform\providers\registry.terraform.io\hashicorp\aws\4.11.0\windows_amd64

Resource - Individual services which provider has to offer

```
provider "github" {
  token = var.token # or `GITHUB_TOKEN`
}
```

```
provider "github" {
  app_auth {
    id              = var.app_id              # or `GITHUB_APP_ID`
    installation_id = var.app_installation_id # or `GITHUB_APP_INSTALLATION_ID`
    pem_file        = var.app_pem_file        # or `GITHUB_APP_PEM_FILE`
  }
}
```

```
provider "kubernetes" {
  config_path = "~/.kube/config"
}
```

```
provider "kubernetes" {
  config_paths = [
    "/path/to/config_a.yaml",
    "/path/to/config_b.yaml"
  ]
}
```

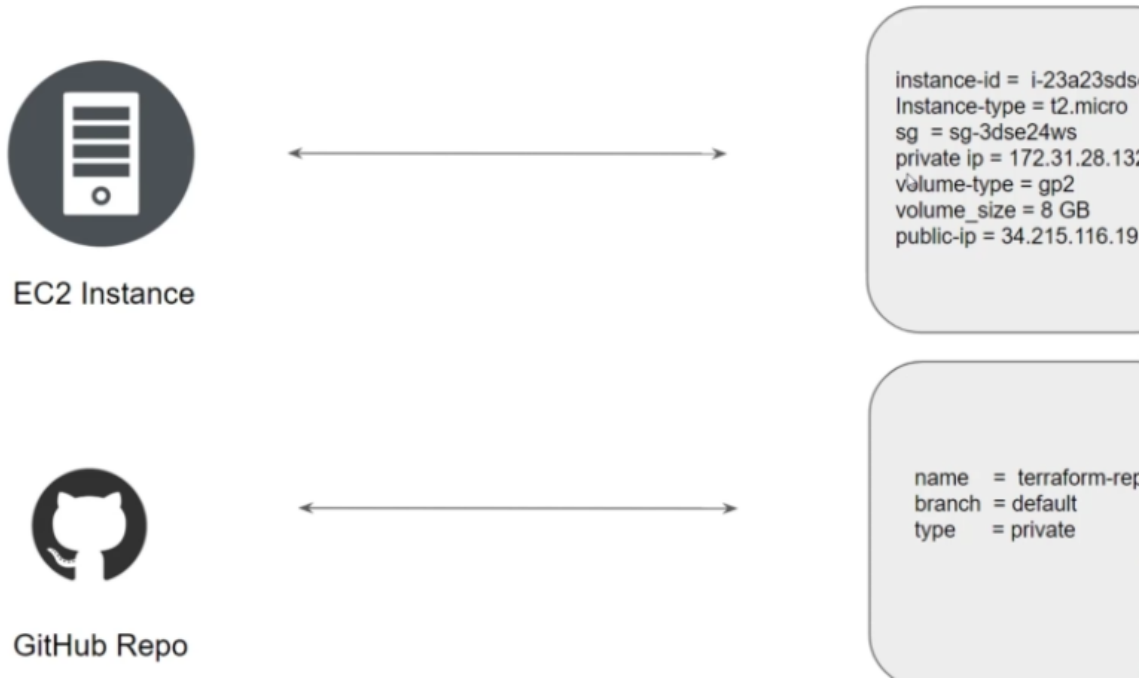
```
provider "kubernetes" {
  host = "https://cluster_endpoint:port"

  client_certificate     = file("~/.kube/client-cert.pem")
  client_key             = file("~/.kube/client-key.pem")
  cluster_ca_certificate = file("~/.kube/cluster-ca-cert.pem")
}
```

6. Terraform State File

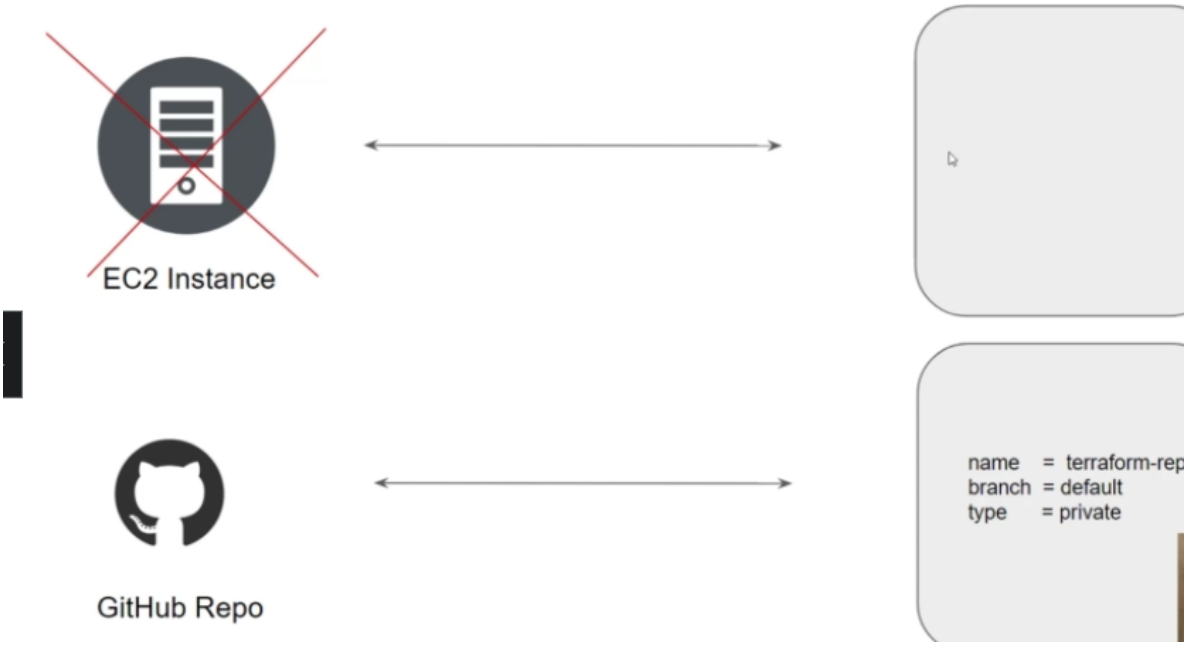
How does terraform determines if the files are created or destroyed and needs to be in the plan?

*Terraform s**tores the state of the infrastructure** that is being created from TF files. ---> **This state allows terraform to map real world resource** to your existing configuration*

- *When we create resource, TF stores the resource parameters (exact resource data) in the backend in TF state file*



If ec2 is destroyed



Desired and Current State

Terraform primary function is to create, modify, and destroy infrastructure resources to **match desired state d**escribed in terraform configuration

Current state - actual state of resource that is currently deployed

- Terraform ensures that deployed infrastructure is based on desired state
- If there is a difference between current and desires, **tf plan** presents description of **changes necessary to achieve desired state**

Ex.

```
aws_instance.first_ec2: Creating...
aws_instance.first_ec2: Still creating... [10s elapsed]
aws_instance.first_ec2: Still creating... [20s elapsed]
aws_instance.first_ec2: Still creating... [30s elapsed]
aws_instance.first_ec2: Creation complete after 32s [id=i-0f8fed2a18bc12dde]

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

Name	Instance ID	Instance state	Instance type
–	i-0f8fed2a18bc12dde	Stopped	t2.micro

```
No changes. Your infrastructure matches the configuration.

Your configuration already matches the changes detected above. If you'd like to update the Terraform state to match,
create and apply a refresh-only plan:
  terraform apply -refresh-only

E:\Terraform>
```

Name	Instance ID	Instance state	Instance type	St
–	i-0f8fed2a18bc12dde	Stopped	t2.medium	–

```
E:\Terraform>terraform refresh
github_repository.example: Refreshing state... [id=terraform_repo]
aws_instance.first_ec2: Refreshing state... [id=i-0f8fed2a18bc12dde]
```

```
code terraform.tfstate
```

```
    ],
    "ephemeral_block_device": [],
    "get_password_data": false,
    "hibernation": false,
    "host_id": null,
    "iam_instance_profile": "",
    "id": "i-0f8fed2a18bc12dde",
    "instance_initiated_shutdown_behavior": "stop",
    "instance_state": "stopped",
    "instance_type": "t2.medium",
    "ipv6_address_count": 0,
    "ipv6_addresses": [],
    "key_name": "",
```

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

Do you want to perform these actions?
```

Name	Instance ID	Instance state	Insta
–	i-0f8fed2a18bc12dde	Running	t2.m

Challenges with current state compute values

If parameters are not described in the configuration, and if they are changed manually, the terraform state will be changed but there will be no terraform plan changes as plan changes only come for parameters described in the configuration file

7. Provider Versioning

Provider plugins are released separately from Terraform itself

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

~>2.0 means Any version in 2.X range

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

When we do terraform init, .terraform.lock.hcl is created which has provider installed full path

.terraform.lock.hcl ×

> Terraform > provider-versioning > .terraform.lock.hcl

```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/aws" {
  version     = "3.75.1"
  constraints = "~> 3.0"
  hashes = [
    "h1:OuaWibxL7IoGHmX6xuhO2XvYJcr+DJNr9pnVjaUiMrw=",
    "zh:11c2ee541ca1da923356c9225575ba294523d7b6af82d6171c912470ef0f90cd",
    "zh:19fe975993664252b4a2ff1079546f2b186b01d1a025a94a4f15c37e023806c5",
    "zh:442e7fc145b2debebe9279b283d07f5f736dc1776c2e5b1702728a6eb03789d0",
    "zh:7a77991b204ae2c16ac29a32226135d5fdbda40c8dafa77c5adf5439a346be77",
    "zh:89a257933181c15293c15a858fbfe7252129cc57cc2ec05b6c0b595d1bfe9d38",
    "zh:b1813ea5b6b0fd88ea85b1b21b8e4119566d1bc34feca297b4fb39d0536893cb",
    "zh:c519f3292ae431bd2381f88a95bd37c52f7a56d91feef88511e929344c180549",
    "zh:d3dbe88b661c073c174f04f73adc2720372143bdfa12f4fe8f411332e64662cf",
    "zh:e92a27e3c7295b031b5d62dd9428966c96e3157fc768b3d848a9ac60d1661c8e",
    "zh:ecd664c0d664fcf2d8a89a01462cb00bcae37da200305aef2de1b8fe185c9cd8",
    "zh:ed6ce1f9fa96aa28dd65842f852abed25f919d20b5cf53d26cec5b3f4d845725",
  ]
}
```

- If we change the constraint (version now), then lock file might not change instead we need to use terraform init -upgrade or delete the file and again foe init with modified config file

8. Read, Generate, Modify Configurations

The three-act structure is a classic method of storytelling with clear beginning, middle, and ending phases. The second act is typically much longer than the others, and can be treated as two sections divided by a climactic midpoint. This template covers the basics, but feel free to add as many details as you wish.

[Understanding Attributes and Output Values in Terraform]

- Terraform has capability to output the attribute of a resource with the output values.
- We can explicitly say that which output values needs to be printed when we apply terraform plan like ip address or s3 bucket identifier etc.
- **An output attribute can be used for user reference and also act as input to other resources being created via terraform**

- Ex: After EIP is created, it's IP address should automatically get whitelisted in the security group

```
# Configure the AWS Provider
provider "aws" {
  region = "ca-central-1"
  access_key = "AKIA3RNOADYTSNSXHU47"
  secret_key = "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"
}
resource "aws_eip" "lb" {
  vpc      = true
}
/*
output "eip" {
  value = aws_eip.lb
}
*/
resource "aws_s3_bucket" "mys3" {
  bucket = "shraman-attribute-demo-001"
}
/*
output "mys3bucket" {
  value = aws_s3_bucket.mys3
}
*/
```

```
aws_eip.lb: Creating...
aws_s3_bucket.mys3: Creating...
aws_eip.lb: Creation complete after 0s [id=eipalloc-0635483883c356bfa]
aws_s3_bucket.mys3: Creation complete after 1s [id=shraman-attribute-demo-001]
```

```
# Configure the AWS Provider
provider "aws" {
  region = "ca-central-1"
  access_key = "AKIA3RNOADYTSNSXHU47"
  secret_key = "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"
}
resource "aws_eip" "lb" {
  vpc      = true
}
output "eip" {
  value = aws_eip.lb.public_ip
}
resource "aws_s3_bucket" "mys3" {
  bucket = "shraman-attribute-demo-001"
}
output "mys3bucket" {
  value = aws_s3_bucket.mys3.bucket_domain_name
```

```
}
```

```
aws_eip.lb: Refreshing state... [id=eipalloc-09534c259af08b22f]
aws_s3_bucket.mys3: Refreshing state... [id=shraman-attribute-dem

Outputs:

eip = "15.222.104.128"
mys3bucket = "shraman-attribute-demo-001.s3.amazonaws.com"
```

Referencing Cross-Account Resource Attributes

```
#In this exercise, we want to attach EIP automatically with EC2 instance on the go
# Configure the AWS Provider
provider "aws" {
  region = "ca-central-1"
  access_key = "AKIA3RNOADYTSNSXHU47"
  secret_key = "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"
}
#Creating EC2
resource "aws_instance" "first_ec2" {
    ami = "ami-0f6b3aca8444b4f04"
    instance_type = "t2.micro"
}
#Create EIP
resource "aws_eip" "lb" {
  vpc      = true
}
#Associate EIP with EC2
resource "aws_eip_association" "eip_assoc" {
  instance_id   = aws_instance.first_ec2.id
  allocation_id = aws_eip.lb.id
}
#Associate EIP with SG
resource "aws_security_group" "allow_tls" {
  name        = "shraman-security-group"
  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["${aws_eip.lb.public_ip}/32"]
#    cidr_blocks = [aws_eip.lb.public_ip/32]
  }
}
output "Instance-ID" {
  value = aws_instance.first_ec2.id
}
output "Instance-Type" {
  value = aws_instance.first_ec2.instance_type
}
output "Instance-State" {
```

```
  value = aws_instance.first_ec2.instance_state
}
output "Public-IP" {
  value = aws_eip.lb.public_ip
}
output "Ingress-Parameters" {
  value = aws_security_group.allow_tls.ingress
}
```

```
Outputs:

Ingress-Parameters = toset([
  {
    "cidr_blocks" = tolist([
      "35.182.144.96/32",
    ])
    "description" = ""
    "from_port" = 443
    "ipv6_cidr_blocks" = tolist([])
    "prefix_list_ids" = tolist([])
    "protocol" = "tcp"
    "security_groups" = toset([])
    "self" = false
    "to_port" = 443
  },
])
Instance-ID = "i-0557167179fe07033"
Instance-State = "running"
Instance-Type = "t2.micro"
Public-IP = "35.182.144.96"
```

[Terraform Variables]

- Repeated static values can create more work
- Central source (variables) to supply the values and from terraform we can reference

```
var.source
```

- Example:

variables.tf

```
variable "vpn_ip" {
  default = "116.50.30.50/32"
}
```

varsdemo.tf

```
resource "aws_security_group" "var_demo" {
```

```
  name        = "shraman-variables"
  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = [var.vpn_ip]
  }
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = [var.vpn_ip]
  }
  ingress {
    from_port   = 53
    to_port     = 53
    protocol    = "tcp"
    cidr_blocks = [var.vpn_ip]
  }
}
```

Approaches for Variable Assignment

- Variables can be declared in different ways
  - **Environment variables**

In Windows it is (but needs to load new command prompt):

```
setx TF_VAR_<name> <value>
```

In Linux:

```
export TF_VAR_<name>="value"
```

  - **Command Line Flags**

```
#To mention explicit value, we can write

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

```
#If you want to type the variable value when doing terraform plan, we can hav empty
variable section

variable "instancetype" {}
```

```
E:\Terraform\section2\terraformvariables>terraform plan
var.instancetype
  Enter a value: t2.medium
```

- **From a File (make terraform.tfvars but we can have custom as well)**

```
--terraform.tfvars
instancetype="t2.large"
#terraform plan

--custom.tfvars
instancetype="t2.large"
#terraform plan -var-file="custom.tfvars"
```

- **Variable default**

```
#Creating EC2
resource "aws_instance" "first_ec2" {
    ami = "ami-0f6b3aca8444b4f04"
    instance_type = var.instancetype
}
```

```
variable "instancetype"{
    default = "t2.micro"
}
```

[Data Types for Variables]

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

No check as instance name has to be only #,

variables.tf	terraform.tfvars

variable “instance_name” {
type=number
}
	instance_name=”john-123”

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

[Fetching Data for Maps and List in Variable]

Basically, we want to map the variable values from the variable default values through a key

```
resource "aws_instance" "myec2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = var.types["us-west-2"]
}

variable "list" {
  type = list
  default = ["m5.large","m5.xlarge","t2.medium"]
}

variable "types" {
  type = map
  default = {
    us-east-1 = "t2.micro"
    us-west-2 = "t2.nano"
    ap-south-1 = "t2.small"
  }
}
```

- We can reference the default value to a file and then map to var.types[0] (ex for first one). The index starts from 0

[Count and Count Index]

- Count parameter can simplify configurations and let you scale resources by simply incrementing a number
- So when we need to create multiple resources, common approach is create them multiple times but here we can scale it easily

```
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
}


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

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

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
}
```

But the problem is that in new cases, it will be create the resources with same name multiple timesas it fails and is not good approach in those cases

To resolve it, we can use count.index

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

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

Then count.index of this variable will give variables names

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

resource "aws_iam_user" "lb" {
  name = var.elb_names[count.index]
  count = 5
  path = "/system/"
}
```

[Conditional Expressions]

- Uses the value of a bool expression to select one of two values

```
condition ? true_val : false_val
```

Can be used where we have multiple env under one account like UAT/SIT/DEV and can use conditional resources to determine where to deploy these resources

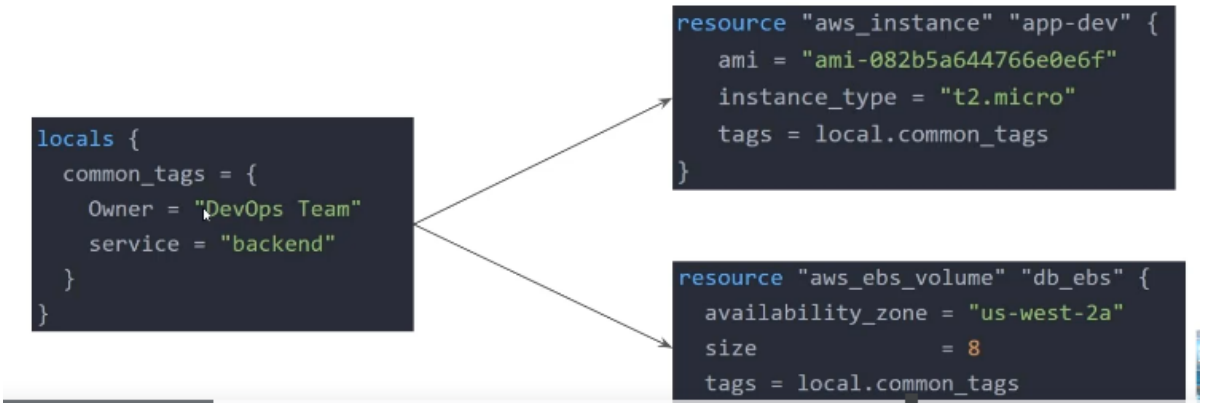
```
variable "istest" {}

#Under resource block
```

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

```
#terraform.tfvars

istest = true
```

[Local Values]

- Local value assigns a name to an expression, allowing it to be used multiple times within a module without repeating it (tagging)\



We can use as conditional expression

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

[Terraform Functions]

- in built functions
- Not user defined functions, only built in functions

```
function(arg1, arg2)
```

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

https://www.terraform.io/language/functions

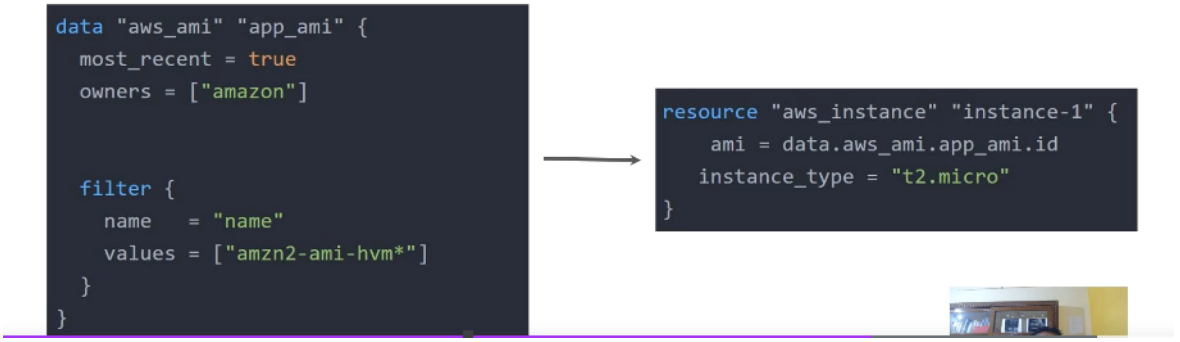
- lookup - to map between resource and variable values
- formatdate
- file - reads the contents of a file at the given path and returns them as a string.
- element - retrieves a single element from a list.

```
provider "aws" {
  region     = var.region
  access_key = "YOUR-ACCESS-KEY"
  secret_key = "YOUR-SECRET-KEY"
}
locals {
  time = formatdate("DD MMM YYYY hh:mm ZZZ", timestamp())
}
variable "region" {
  default = "ap-south-1"
}
variable "tags" {
  type = list
  default = ["firstec2","secondec2"]
}
variable "ami" {
  type = map
  default = {
    "us-east-1" = "ami-0323c3dd2da7fb37d"
    "us-west-2" = "ami-0d6621c01e8c2de2c"
    "ap-south-1" = "ami-0470e33cd681b2476"
  }
}
resource "aws_key_pair" "loginkey" {
  key_name   = "login-key"
  public_key = file("${path.module}/id_rsa.pub")
}
resource "aws_instance" "app-dev" {
```

```
    ami = lookup(var.ami,var.region)
    instance_type = "t2.micro"
    key_name = aws_key_pair.loginkey.key_name
    count = 2
    tags = {
      Name = element(var.tags,count.index)
    }
}
output "timestamp" {
  value = local.time
}
```

Data Sources

Referencing data (metafile names) from AWS to fetch same name to terraform config



Debugging In Terraform

- Terraform has detailed logs which can be enabled by settings TF_LOG environment variable to any value
- It can be set to TRACE, DEBUG, INFO, WARN, ERROR

```
export TF_LOG=TRACE
```

Terraform Format

Automatic formatting code: terrform fmt

Terraform Validate

- Checks whether a configuration is syntactically valid
- can check different aspects such as unsupported arguments, undeclared variables and others

Load Order and Semantics

- Generally load all configuration files within the directory in alphabetical order
- .tf .tf.json
- Not recommended to write everything in single file as to get more cleaner code through split code through multiple files
  - providers.tf
  - variables.tf
  - <service>.tf

Dynamic Blocks

- Repeatable nested blocks - long code and difficult to manage in a longer time
- dynamically construct repeatable nested blocks which is supported inside resource, data, provider and provisioned blocks

```
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"]
}
```

```
variable "sg_ports" {
  type        = list(number)
  description = "list of ingress ports"
  default     = [8200, 8201,8300, 9200, 9500]
}
```

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

- Iterator - set the name of temp variable that represents the current element of complex value
- Easier to read the code

```
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
  }
}
```

Tainting Resources

- Multiple users of the organization have done manual changes
  - Import the changes to terraform
  - Delete & Recreate resource
- terraform taint - marks Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply
- If Terraform currently considers a particular object as tainted but you've determined that it's actually functioning correctly and need *not* be replaced, you can use terraform untaint to remove the taint marker from that object.

Splat Expression

- 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

- generate visual representation of either a configuration or execution plan

- DOT format --> can be converted to Image

```
terraform graph > graph.dot
yum install graphviz
cat graph.dot | dot -Tsvg > graph.svg
```

Saving Terraform Plan to File

```
terraform plan -out=path
```

To revert to original settings through file (binary file)

Terraform Output

- Extract the value of an output variable from state file
- We can see the output in cli and also in tfplan

```
terraform output <name>
```

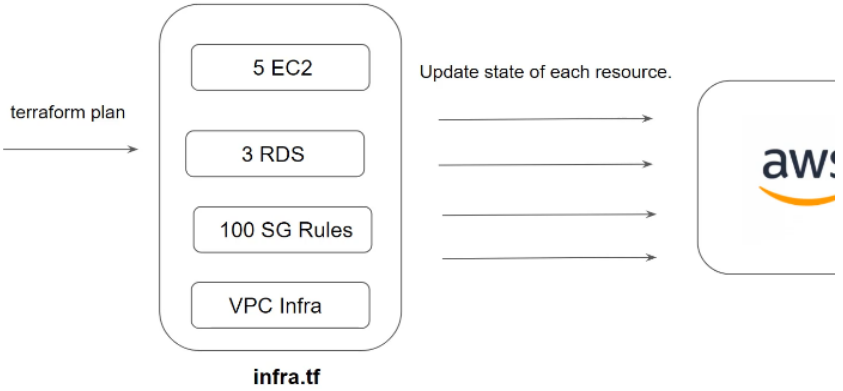
Terraform Settings

- Terraform version - which versions of tf can be used with your configuration

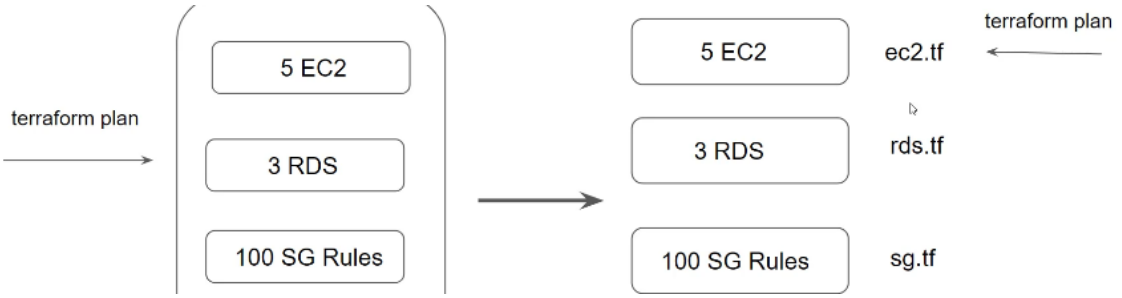
```
  terraform {
    required_version = "> 0.12.0"
    required_providers {
       aws = "~> 2.0"
   }
}
```

Dealing with Large Infrastructure

- When you have a larger infrastructure, you will face issue related to API limits for a provider
- Can slow down the processes



- smaller configuration must be applied



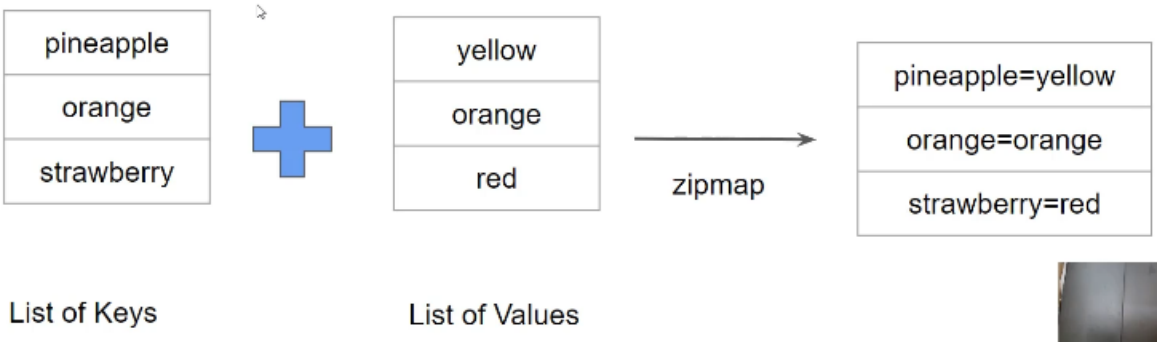
- If someone has already created a single file, then we can use -refresh=false flag, which will prevent tf from querying the current state during operations like terraform plan
- Also, we can specify the target directly -target=resource

```
terraform plan -target=ec2
```

- update in-place = there is an update which is going in place

Zipmap Function

- constructs a map from a list of keys and a corresponding list of values




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

- use-case: creating mulitple IAM users
- need output that contains direct mapping of IAM names and ARNs

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

output "name" {
  value = aws_iam_user.lb[*].name
}

output "combined" {
  value = zipmap(aws_iam_user.lb[*].name,aws_iam_user.lb[*].arn)
}
```

9. Provisioners

Terraform is not only about creating or deleting infrastructure resources but also configuring those resources as well.

- To create a complete end to end solution
- ex. execute a script on local or remote machines as part of resource creation or deletion

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

  provisioner "remote-exec" {
    inline = [
      "sudo amazon-linux-extras install -y nginx1.12",
      "sudo systemctl start nginx"
    ]

    connection {
      type        = "ssh"
      host        = self.public_ip
      user        = "ec2-user"
      private_key = "${file("./terraform.pem")}"
    }
  }
}
```

Types:

- local-exec: allow us to invoke local executable after resource is created
  - ex, can be used to manually update glue job with more config parameters
  - Ansible Playbook
- remote-exec: allow us to invoke scripts directly on remote server
- Others: chef, connection, file, habitat, null_resource, salt-masterless

Let's take an example:

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

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

Creation-Time Provisioner: only run during creation, not during updating or any other lifecycle. If it fails then resource is tainted

Destroy-Time Provisioner: are run before the source is destroyed

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

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

Failure Behaviour

If provisioners fail, then terraform apply will also fail

on_failure setting can be used to change this

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.

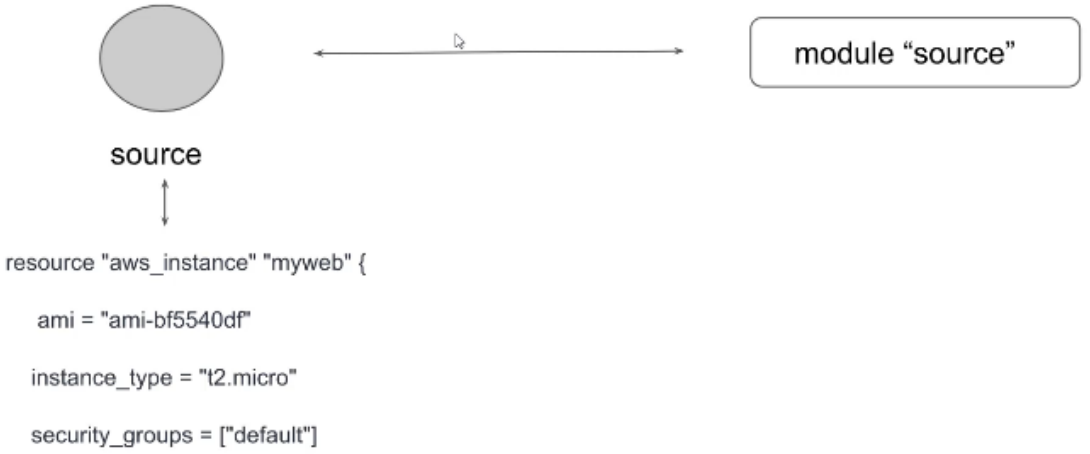
```
resource "aws_instance" "web" {
  # ...

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

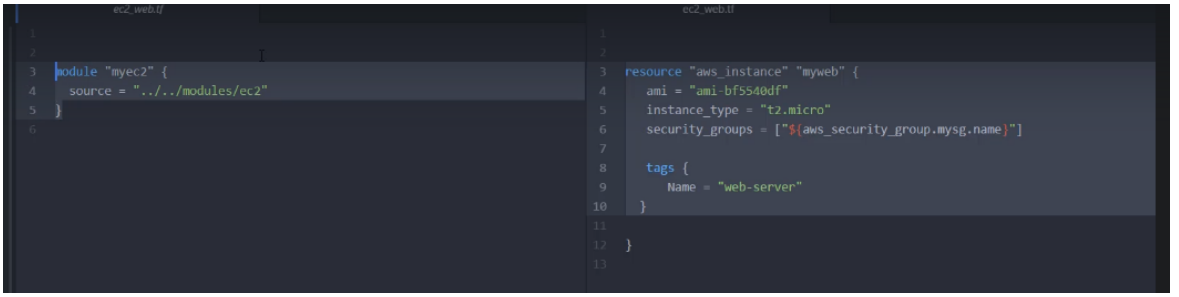
10. Modules & Workspaces

DRY Principle:

Don't Repeat Yourself aimed at reducing repetition of software patterns (remove static pattern into variables)

- Centralize terraform resource and can call our from TF files whenever required.

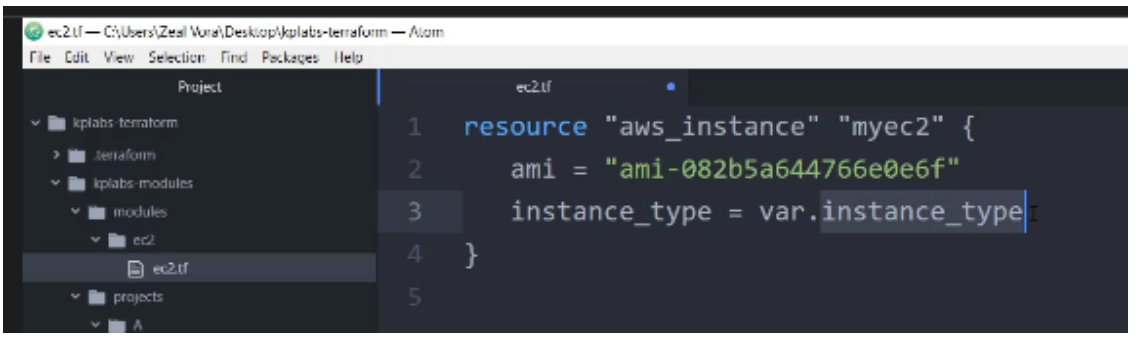


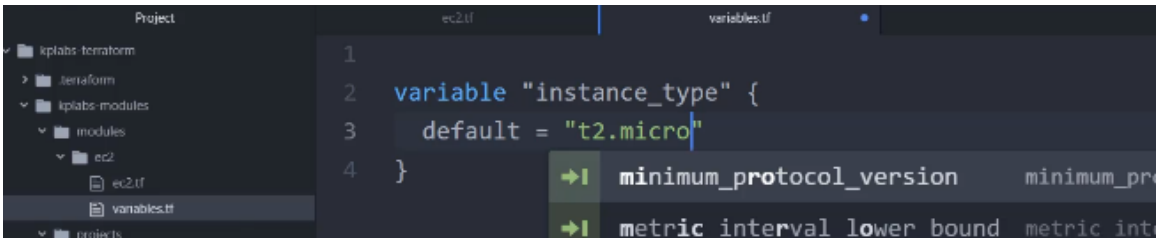


Challenges with modules

- build multiple environments (same setup but with diff variables)
- When we use modules directly, the resources will be replica of code in the module.

- Once we have hard coded in module, we can't overwrite it in destination
- Create variables to overcome this issue, need to create variables.tf in ec2 folder





Using Locals with Modules

- Allow users to override values which you might not want

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

Because if users override the value, it can change everywhere thus risk is bit higher so we can use **locals**

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

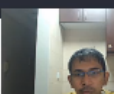
```
resource "aws_security_group" "ec2-sg" {
  name        = "myec2-sg"

  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
}
```

Referencing Output values

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

repository of modules written by Terraform community

- can use source argument within module

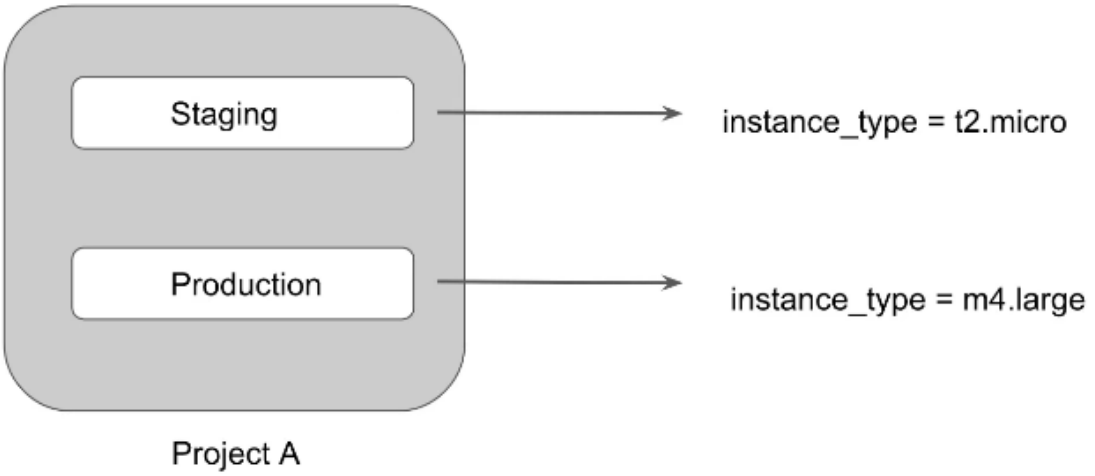
To use Terraform Registry module within the code, we can make use of the source argument that contains the module path.

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

```
module "ec2-instance" {
  source  = "terraform-aws-modules/ec2-instance/aws"
  version = "2.13.0"
  # insert the 10 required variables here
}
```

Terraform Workspace

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




```
terraform workspace

terraform workspace show

terraform workspace list

terraform workspace select <name>



terraform workspace -h

terraform workspace new dev
```

11. Remote State Management

Team collaboration using Git (Bitbucket)

```
cd Desktop/myrepo

git clone https://shramang@bitbucket.org/shramang/terraform-learning.git

cp -r <source> .

git add .

git commit -m "First Commit"

git push origin master
```

#Never store any api keys, secret keys in the code, .ake file outside GIT

```
"${file("../secretkey.txt")}"
```

```
name:                          "mydb"
option_group_name:             <computed>
parameter_group_name:          "default.mysql5.7"
password:                      <sensitive>
port:                          <computed>
```

- But when we do terraform apply, it still shoes password as sensitive but when we git commit and push to GIT with this command,

terraform.tfstate is committed and added to git repo, the password is stored in clear text which is not best security practice

<u>Module Sources:</u>
Local path - must begin with either ./ or ../ to indicate that a local path is intended

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

Git module source: - prefixing address with special git prefix (https or ssh)

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

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

Terraform will clone and use default branch (can override using ref argument)

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

Terraform & Gitignore

Don't want to commit 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

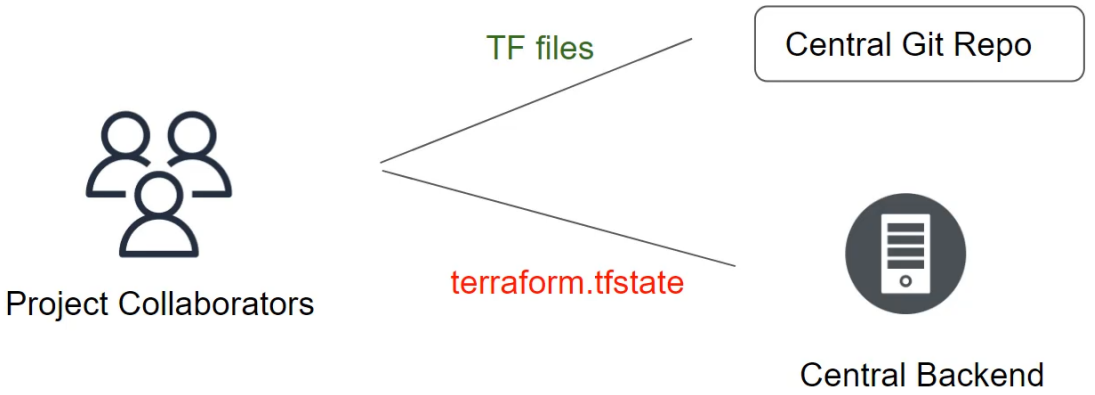
```
# Local .terraform directories
**/.terraform/*
# .tfstate files
*.tfstate
*.tfstate.*
# Crash log files
crash.log
crash.*.log
# Exclude all .tfvars files, which are likely to contain sensitive data, such as
# password, private keys, and other secrets. These should not be part of version
# control as they are data points which are potentially sensitive and subject
# to change depending on the environment.
*.tfvars
*.tfvars.json
# Ignore override files as they are usually used to override resources locally and so
# are not checked in
override.tf
override.tf.json
*_override.tf
*_override.tf.json
# Include override files you do wish to add to version control using negated pattern
# !example_override.tf
# Include tfplan files to ignore the plan output of command: terraform plan -out=tfplan
# example: *tfplan*
# Ignore CLI configuration files
.terraformrc
terraform.rc
```

Terraform Backends

- primarily determine where Terraform stores its state
- By default, it uses a backend called local to store state as a local file on disk

```
demo.tf ----> terraform.tfstate
```

- in larger industries, storing local state file is not good for team collaboration



Popular backend:

s3, consul, azurerm, kebernetes, https, etcd, ARTIFACTORY

- TF uses this persisted state data to keep track of resources it manages.

```
backend.tf

---->

terraform {
  backend "s3" {
    bucket = "kplabs-terraform-backend"
    key    = "network/demo.tfstate"
    region = "us-east-1"
    dynamodb_table = "terraform-state-locking"
  }
}
resource "time_sleep" "wait_150_seconds" {
  create_duration = "150s"
}
```

State Locking

With write operation, terraform would lock the state file. Otherwise your ongoing apply operations, if others also try for the same, it can corrupt the 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.
```

- Not all backends support locking

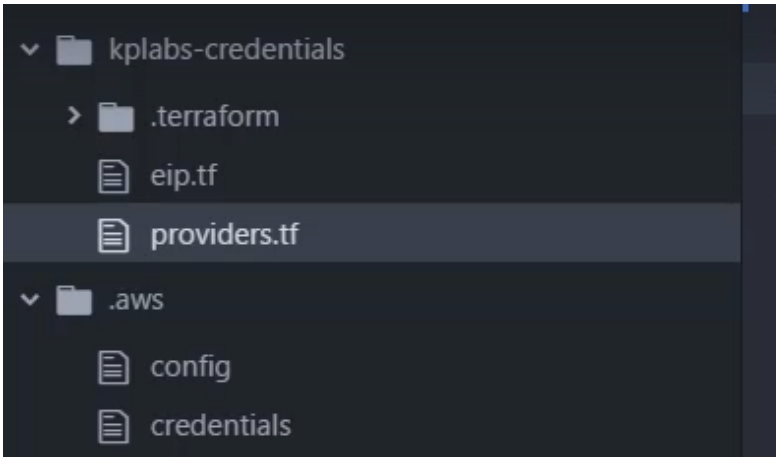
If state unlocking is failed and you need to unlock it, we can use (use only when you want to unlock your own lock):

```
force-unlock
```

12. Security Primer

Handling Access & Secret Keys

- Not to hard code



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

Q/A

- *Terraform requires GO as a pre-requisite for installation? - False*
  - (Terraform Core is a statically-compiled binary written in the Go programming language. However it is not required to install GO language to be installed as a pre-requisite to run Terraform.)
- *Referring to the following terraform code, what is the local name for the resource that is defined?*

```
resource "aws_instance" "instance-1" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
   count = 3
}
```

  - instance-1
- The sensitive values are still recorded in the state file and anyone who has access to the state file will be able to see the value.
- *What is the name of a file where all your infrastructure-related details managed by terraform is stored?*
  - terraform.tfstate
  - (Terraform by default will store the state in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment.)
- M*atthew is implementing Terraform in production. He realized that every region in AWS has a different AMI ID for CentOS 7 OS. He wants to create a Terraform code that works for all the regions. He has already created the EC2 resource but needs to figure on how he can deal with different AMI IDs based on regions? What is the best approach?*
  - Make use of Data source
  - (Using Data source, we can query for latest AMI ID based on specific conditions like availability, Operating System or other.)

- *James is having an issue with his Terraform code. He wants to enable debugging. How can he do that?*
    - TF_LOG=TRACE as env variable
- *Which of the following command will export the debugging logs to a file named /tmp/terraform-crash.log?*
    - export TF_LOG_PATH=/tmp/terraform-crash.log
- *Which of the following command will format your terraform code?*
    - terraform fmt
- *Terraform Plan validates the overall syntax of terraform code and will error if aspects like an undefined variable, missing arguments are part of the code? - True*
- *Which feature allows multiple state files for a single configuration file depending upon the environment? -*
    - Workspaces
- The locals block defines one or more local variables within a module. Each locals block can have as many locals as needed.
- Is defining the module version argument mandatory while pulling code from Terraform Registry? - False
- A variable will need a value assigned in the following manner. Terraform Administrator needs to set a default value for the data type. What is the value that he can set?

```
az=["us-west-1a","us-west-1b"]
```

    - List
- Is provider configuration block is mandatory in every configuration? - True

Information about Answers:

- Private Network Connectivity is available for Terraform Enterprise, since it is installed locally in your data center or cloud environment. However, it is also available in Terraform Cloud for Business since you can use a self-hosted agent to communicate with local/private hosts in your environment.

- SAML/SSO is available in BOTH Terraform Enterprise and Terraform Cloud for Business, therefore it is NOT exclusive to just Terraform Enterprise.

- Private Module Registry is available in every version of Terraform except for Open-Source. Therefore it is NOT exclusive to Terraform Enterprise

- Locally hosted installation is available for Terraform Enterprise, but technically you install Terraform open-source on your local machine or server, therefore it is NOT exclusive to Terraform Enterprise.

- Clustering is the ONLY answer that is available ONLY in Terraform Enterprise. You can't cluster Terraform open-source, and the other options are hosted solutions. This makes clustering the only correct answer. Note that Clustering was available for Enterprise for a while, then HashiCorp removed it. As of January 15, 2021, it's back and you can read more about it at this link.

Which Terraform command will force a resource to be destroyed and recreated even if there are no configuration changes that would require it?

A workspace can only be configured to a single VCS repo, however, multiple workspaces can use the same repo, if needed.

terraform apply -replace=<address>

The terraform validate command validates the configuration files in a directory, referring only to the configuration and not accessing any remote services such as remote state, provider APIs, etc.
Validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state. It is thus primarily useful for general verification of reusable modules, including correctness of attribute names and value types.

- Anytime you need to reference a resource that is NOT part of your Terraform configuration, you need to query that resource using a data block
- Whenever a configuration's backend changes, you must run terraform init again to validate and configure the backend before you can perform any plans, applies, or state operations. Re-running init with an already-initialized backend will update the working directory to use the new backend settings. Either -reconfigure or -migrate-state must be supplied to update the backend configuration
- Terraform OSS and Terraform Enterprise are versions of Terraform that can be installed locally on your own servers
- The most common source of dependencies is an implicit dependency between two resources or modules. That means that Terraform builds a dependency map (aka resource graph) to help determine what resources it can create in parallel, and what resources are dependent on others based on interpolation used within the configuration.

- Child modules can only access variables that are passed in the calling module block.
- As you continue to work with Terraform, you're going to need a way to organize and structure data. This data could be input variables that you are giving to Terraform, or it could be the result of resource creation, like having Terraform create a fleet of web servers or other resources. Either way, you'll find that data needs to be organized yet accessible so it is referenceable throughout your configuration. The Terraform language uses the following types for values:
- * **string**: a sequence of Unicode characters representing some text, like "hello".
- * **number**: a numeric value. The number type can represent both whole numbers like 15 and fractional values like 6.283185.
- * **bool**: a boolean value, either true or false. bool values can be used in conditional logic.
- * **list (or tuple)**: a sequence of values, like ["us-west-1a", "us-west-1c"]. Elements in a list or tuple are identified by consecutive whole numbers, starting with zero.
- * **map (or object)**: a group of values identified by named labels, like {name = "Mabel", age = 52}. Maps are used to store key/value pairs.
- Strings, numbers, and bools are sometimes called primitive types. Lists/tuples and maps/objects are sometimes called complex types, structural types, or collection types.

Terraform v0.13 and later vs 0.12 and previous