# Terraform
- https://www.terraform.io/docs
- Import AWS Organization- https://juzerali.com/importing-an-existing-aws-organizations-root-ou-into-terraform

- HashiCorp Terraform is an infrastructure as code tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share. You can then use a consistent workflow to provision and manage all of your infrastructure throughout its lifecycle. Terraform can manage low-level components like compute, storage, and networking resources, as well as high-level components like DNS entries and SaaS features.

 ![image](https://user-images.githubusercontent.com/57224583/156105778-80a5de93-e9c6-434f-81c5-17ce1097be5d.png)

- HashiCorp and the Terraform community have already written more than 1700 providers to manage thousands of different types of resources and services, and this number continues to grow. You can find all publicly available providers on the Terraform Registry, including Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, DataDog, and many more.

- The core Terraform workflow consists of three stages:

    - Write: You define resources, which may be across multiple cloud providers and services. For example, you might create a configuration to deploy an application on virtual machines in a Virtual Private Cloud (VPC) network with security groups and a load balancer.

    - Plan: Terraform creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration.

    - Apply: On approval, Terraform performs the proposed operations in the correct order, respecting any resource dependencies. For example, if you update the properties of a VPC and change the number of virtual machines in that VPC, Terraform will recreate the VPC before scaling the virtual machines.

    ![image](https://user-images.githubusercontent.com/57224583/156105854-d1842c4a-ecc1-4996-a1ba-ff2f7b3e04ea.png)

# Terraform-Projects
Practice Terraform Projects
Docs- https://registry.terraform.io/providers/cloudflare/cloudflare/latest/docs

# let's begin with intro
- Cloudflare Terraform
    - Configure Cloudflare using HashiCorp's “Infrastructure as Code” tool, Terraform. With
        Cloudflare’s Terraform provider, you can manage your edge using the same familiar tools you use to automate the rest of your infrastructure. Define and store configuration in source code repositories like GitHub, track and version changes over time, and roll back when needed all without needing to use the Cloudflare APIs.

# Some of the Tool-set for (Infrastructure as a code)

# Choosing a right Infrastructure as Code tool
- there are various type IAAC to deploy:

    <img width="596" alt="image" src="https://user-images.githubusercontent.com/57224583/155925430-0ae2cceb-0aec-4c1a-9aa2-30c397891b05.png">

    <img width="605" alt="image" src="https://user-images.githubusercontent.com/57224583/155925568-1104f00a-3b72-45b2-8c60-284b6e9bdbcb.png">

- https://docs.ansible.com/ansible/latest/index.html

<img width="636" alt="image" src="https://user-images.githubusercontent.com/57224583/155926159-32516017-3b71-46ba-92cd-ac6555480000.png">

<img width="612" alt="image" src="https://user-images.githubusercontent.com/57224583/155926236-1d64a07d-286f-4d7c-add2-f18896c010d8.png">

- Now, for this, there are four important questions.
- that you need to ask. First, the important question is: Will your infrastructure be vendor-specific? in the long ter m?

- So, for example, will your infrastructure stay longer in AWS?

- So if organizations say that, yes, our infrastructure will be needed for the next 10 years.

- years, then, in such cases, the ability to choose a different infrastructure, a code tool can be an advantage.

- So, for example, in AWS, they have a dedicated infrastructure as a code tool called AWS.

- CloudFormation that is available.

- And in fact, CloudFormation is also great.

- Terraform. So let's look at the 2nd pointer here.

- In which states do you plan to have a multi-cloud, which is an infrastructure based on a hybrid cloud?

- So, because this is a very common pattern these days, many members of the organization have AWS+ Azure

- or AWS+ Google OR AWS + DigitalOcean and so on.

- So plan to have a cloud-based hybrid architecture.

- Third, how well does your infrastructure as a code-based tool integrate with configuration management?

- This is a very important question and the fourth is the overall price and support of a specific tool.

- Based on this, so you can choose the appropriate IAC tool.

- Now, let me share with you.

<img width="407" alt="image" src="https://user-images.githubusercontent.com/57224583/155927080-8d9f0ab6-bef5-4b7b-958f-e68323741f49.png">

- Terraform works pretty well in all of this. So let's take a quick look at it.

- The first important point, however, is that TerraForm supports multiple platforms.

- It has hundreds of suppliers.

- So if you're in AWS, if you're in Azure, if you're in Google Cloud, if you're in DigitalOcean, Terraform supports them all.

- And that's the big advantage of Terraform over CloudFormation. CloudFormation is

- so a specific tool for AWS. Terraform works on multiple platforms. It is therefore not related to a single platform .

- https://www.terraform.io/language/providers

- The second important advantage of Terraform is its simple setup, language and learning curve more

- Fast, it is much easier to learn Terraform than another language.

- Now, I'll share an example with you.

- So we had few interns in our organization and in just two weeks they learned Terraform at

- starting from scratch and they started writing good Terraform code to deploy the infrastructure.

- it's pretty simple and you really like it in the longer run.

- The third advantage is its easy integration with configuration management tools like Ansible.

- 4th is that it is easily expandable using plug-ins.

- So you can create your own plugins, if there are certain things you see that are not taken

- in charge or if there are certain things you want to develop specifically for your organization

- and 5th, it's totally free.

## Download the Exe from official website

- https://www.terraform.io/downloads

- place it into folder and edit in the Enviornment variable
- check with cmd by
    - terraform -v

## Initialize Terraform
- Introduction to Terraform init
    - This tutorial shows you how to get started with Terraform. The tutorial uses an example scenario where you have a web server for your domain that is accessible on 203.0.113.10 and you just signed up your domain (example.com) on Cloudflare to manage everything in Terraform.

- Before you begin, ensure you installed Terraform. You will also need to create an API Token with permissions to edit resources for this tutorial.

1. Define your first Terraform config file
    - Create an initial Terraform config file. Any files ending in .tf will be processed by Terraform. As the configuration becomes more complex, you will want to split the config into separate files and modules. For now, proceed with a single file.

    - then add this configuration according to your domain

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

provider "cloudflare" {
    email = "you@example.com"
    api_token = "your-api-token"
}

variable "zone_id" {
    default = "e097e1136dc79bc1149e32a8a6bde5ef"
}

variable "domain" {
    default = "example.com"
}

resource "cloudflare_record" "www" {
    zone_id = var.zone_id
    name    = "www"
    value   = "203.0.113.10"
    type    = "A"
    proxied = true
}
```

2.Initialize Terraform and the Cloudflare provider
- then type "terraform init" in CMD of project directory
    - When you run terraform init, any plugins required, such as the Cloudflare Terraform provider, are automati cally downloaded and saved locally to a .terraform directory.

3. Review the execution plan
- then type "terraform plan" to see the execution file that will be exicuted
    - In the “execution plan”, Terraform will create a new DNS record as requested. Values that you explicitly sp ecified are displayed, such as the the value of the A record — 203.0.113.10. Values display as <computed> when the y are derived based on other API calls, for example, looking up the metadata, or if the values are returned after the o bject is created.

4. Apply your changes
- Terraform apply/terraform apply --auto-approve
- The plan command is important because it allows you to preview the changes for accuracy before actually ma king them. After you review the execution plan, apply your changes.

- You can use --auto-approve on the command line for a briefer output. Without this flag, Terraform will displa y the output of the Terraform plan and then ask for confirmation before applying it.

5. Verify the results
- terraform show
- Log in to the Cloudflare Dashboard and select the DNS tab. You should see the record created by Terraform.

- To see the full results returned from the API call, including the default values that you did not specify but let Terraform compute, you can run terraform show.

# Update the code and then apply
- if want to update the code
  - update the file first
  - then check the updated things with "terraform plan"
  - then hit the "terraform apply"

# Better reference
- Argument Reference
  - The following arguments are supported:
    - email
      - (Optional) The email associated with the account. This can also be specified with the CLOUDFLARE_EM
AIL shell environment variable.
    - api_key
      - (Optional) The Cloudflare API key. This can also be specified with the CLOUDFLARE_API_KEY shell en
vironment variable.
    - api_token
      - (Optional) The Cloudflare API Token. This can also be specified with the CLOUDFLARE_API_TOKEN s
hell environment variable. This is an alternative to email+api_key. If both are specified, api_token will be used over email+api_key fields.
    - api_user_service_key
      - (Optional) The Cloudflare API User Service Key. This can also be specified with the CLOUDFLARE_API
_USER_SERVICE_KEY shell environment variable. The value is to be used in combination with an api_token, or e
mail and api_key. This is used for a specific set of endpoints, such as creating Origin CA certificates.
    - rps
      - (Optional) RPS limit to apply when making calls to the API. Default: 4. This can also be specified with the CLOUDFLARE_RPS shell environment variable.
    - retries
      - (Optional) Maximum number of retries to perform when an API request fails. Default: 3. This can also be s
pecified with the CLOUDFLARE_RETRIES shell environment variable.
    - min_backoff
      - (Optional) Minimum backoff period in seconds after failed API calls. Default: 1. This can also be specified with the CLOUDFLARE_MIN_BACKOFF shell environment variable.
    - max_backoff
      - (Optional) Maximum backoff period in seconds after failed API calls Default: 30. This can also be specified with the CLOUDFLARE_MAX_BACKOFF shell environment variable.
    - api_client_logging
      - (Optional) Whether to print logs from the API client (using the default log library logger). Default: false. T
his can also be specified with the CLOUDFLARE_API_CLIENT_LOGGING shell environment variable.
    - account_id
      - (Optional) Configure API client with this account ID, so calls use the account API rather than the (default) user API. This is required for other users in your account to have access to the resources you manage. This can also be specified with the CLOUDFLARE_ACCOUNT_ID shell environment variable.
    - api_hostname
      - (Optional) Configure the API client to use a specific hostname. Default: "api.cloudflare.com"
    - api_base_path
      - (Optional) Configure the API client to use a specific base path. Default: "/client/v4"

      - last update

# Section-3 :Deploying Infrastructure with terraform
## Providers and Resources
- Providers
  - Terraform Supports Multiple Providers
  - Depending on what type of infrastructure you want to launch, we ave to use appropriate providers accordingly.
  - there are more than 700 providers.

- Example
    - AWS (if want to use AWS we need AWS Providers)
        - EC2, Dynamodb etc
- This is how 3rd party managed providers will look like

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

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

        - there is a .terraform folder in your project that contains your terraform provider.

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

    - Example
        - resource "cloudflare_record"
        - resource "cloudflare_page_rule"

## Destroying Infrastructure with Terraform
- 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.

- Multiple Approach to destroy
    - Approach 1
        - terraform destroy allows us to destroy all the resources that are created with in the folder.
        - Command
            - terraform destroy

    - Approach 2
        - terraform destroy with -target flag allows us to destroy specific resource.

        - Terraform Destroy with target
            - the -target option can e used to focus terraform's attention on only a subset of resources.

            - Combination of: Resource Type + Local Resources Name

        - Command
            - terraform destroy -target cloudflare_record.DNS_entry

- terraform plan

- Remove the code
    - Either remove the code or comment out(/*Code*/) the part that you have removed, otherwise terraform plan w

ill show you to create again, due to configuration file mismatch.

## Understand Terraform State File
- Terraform stores the state of the infrastructure that is being created from the .tf files.

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

- when ever we are going to run terraform apply/pla/destroy, this file keeps changing.

Note: Please Do not touch your terraform.tfstate, because it contains original configuration.

- But if you ever deleted both state file even backup as well, then the resources you have created earlier they will get conflict or may be they will create another resources.

## Understanding Desired and Current State
- Desired State
  - Terraform's primary function is to create, modify, and destroy infrastructure resources to match the desired state described in a terraform configuration.

- Current State
  - Current state is the actual state of a resource.

- 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.

- Terraform refresh
  - For an instance, if any of the infrastructure is deleted or updated from the GUI perspective.
  - then this command will get the changes and update the tf state file.
  - Example
    - Terraform apply
    - then change something from Dashboard/GUI, in a particular resource.
    - then terraform refresh
    - terraform plan, you will get to see that there is some change in configuration, but the terraform will tell to go back to desired state that has been set by terraform.
    - terraform apply, to save the changes.

## Note
- Always read the terraform carefully before apply, because sometime due to different configurations or architectural things, it may destroy and create a new resource(also it may destroy insider configuration of a resource).

## Challenges with the current state on computed values
- Current State
  - Current state is the actual state of a resources that is currently deployed.

  - when someone apply some changes in the dashboard and it doesn't reflect in terraform.tfstate.
    - try to run below command for getting changes
      - terraform refresh

  - So, when you discuss the desired state, if you look at the first specific PPT, it indicates that the main function of Terraforms is to create, modify and destroy the infrastructure resource to match the state described in the Terraform configuration.

  - The second part is therefore very important. The desired state described in a Terraform configuration.

  - Now, in this Terraform configuration, we have never described that the security group should be by default.

  - Thus, the security group is not part of their desired state at all. Thus, even if for the infrastructure you are modifying, configurations that are not part of their desired state, the next time you make a "terraform plan", the terraform plan will not show you any details to cancel these amendments.

  - And this is the reason why it is generally recommended that whenever you go ahead and create a resource as an instance, do not specify only minimal things, specify all the important things that are necessary, including the IAM role, security groups, and various other areas as part of your Terraform configuration so that it always matches the desired state whenever you add a Terraform plan in the future.

## Deploying Infrastructure with Terraform

- https://registry.terraform.io/providers/hashicorp/aws/latest/docs

- Basically to create any resource we need to create authentication method:
  - static auth
    - Access key and secret key
    - go to IAM, then users -> select yours
    - security credentials -> or use last credentials
    - save it

- Now you have to be careful about the providers and the resources we are using, it may change for others.

- Now crete a .tf file in folder and start writing down the code.

- After all the configuration, run terraform init to initialize the terraform inside this folder.

- now run terraform plan, to see the changes going to be take place in actual env.

- check if everything is good, then terraform apply
- okay

## Provider Versioning
- Overview
  - Provider plugins are released separately from Terraform itself.
  - They have different set of version numbers.

- 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.

- 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

- if tou face any difficulty just delete the .terraform.lock.hcl file and change the provider version accordingly.

- Dependency Lock File
    - Terraform dependency lock file allows us to lock to a specific version of 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 in terraform init.
        - Terraform init -upgrade.

    - https://docs.google.com/document/d/179clqsxOGQa-iGKu1dcmz89Vpso9-7Of8opIkXwPr_k/edit?usp=sharing

# Section-4 :Read, Generate, Modify Configuration
- Overview of the Format
    - We tend to use a different folder for each 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 EC@ Instance	Folder 1
Tainting Resources	Folder 2
Conditional Expression	Folder 3

- Delete/Destroy Resources after Practical

## Attributes & Output Values
- Terraform has capability to output the attribute of a resource with the output values.

    - Example
        - ec2_public_ip = 35.161.21.197
        - bucket_identifier = terraform-test-kplabs.s3.amazonaws.com

- An outputted attributes can not only be used for the user reference but it can also act as a input to other resources.
    - Example
        - After Elastic Ip is created, it's IP address should automatically get whitelisted in the security group.

- Let's try with code
    - create a tf file and add this code after provider

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

resource "aws_s3_bucket" "mys3" {
    bucket = "nayan-practice-training-s3"
}
```
    - then terraform plan
    - terraform apply
    - then check in the AWS Dashboard

- Now we will work with Output
    - add the below code after there own resources

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

        output "mys3bucket" {
           value = aws_s3_bucket.mys3.bucket_domain_name
        }
```

- then terraform destroy
- terraform apply
    - on the cmd you will be able to see the outputs
        - eip = "3.111.253.15"
        - mys3bucket = "nayan-practice-training-s3.s3.amazonaws.com"
- we can get more things(check Attribute reference documentation for it)

- Now, the output basically contains all the output attributes that you might have set in your TF file and

- Resources are the actual underlying resources that were created through Terraform.

## Referencing Cross-Resource Attributes

- This is basically to associate or attach the the things to other resources
    - First, We are going to create EC2 instance and attaching a Elastic Ip to it via Associating

```
        resource "aws_eip_association" "eip_assoc" {
           instance_id   = aws_instance.web.id
           allocation_id = aws_eip.example.id
        }
```

- Second, We are going to create Security group and attaching the Elastic Ip to it via Associating.

```
        resource "aws_security_group" "allow_tls" {
           name        = "Nayan-sample"

           ingress {
              from_port        = 443
              to_port          = 443
              protocol    = "tcp"
              cidr_blocks      = ["Add elastic ip name"/subnetmask]
           }

           egress {
              from_port        = 0
              to_port          = 0
              protocol         = "-1"
              cidr_blocks      = ["0.0.0.0/0"]
              ipv6_cidr_blocks = ["::/0"]
           }

           tags = {
              Name = "allow_tls"
           }
        }
```

- terraform apply

  - terraform destroy

## Terraform Variable

- we can have central source from which we can import the values from. the static files should be kept at one place.

- please avoid doing hard coding.

## Multiple Approaches to variable assignment
- variables in terraform can be assigned values in multiple ways.

  - Environment variable
    - in cmd add below command for this
      - setx TF_VAR_instancetype m5.large
    - reopen the cmd
    - type below commmand to check
      - echo %TF_VAR_instancetype%
  - Command line flags
    - terraform plan -var="instancetype=t2.small"
  - From a file
    - create a variables.tf file add below code
      - variable "instancetype" {}
    - create a file named as "terraform.tfvars"
    - add the below code
      - instancetype="t2.large"
  - Variables Defaults
    - creating a variable as a default value

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

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

  - if no type is given, it will accept any type.

  - string
    - Sequence of a unicode character
  - list
    - Sequence list of values identified by their position starts with 0. ["nayan", "Rajani"]
  - map
    - a group of values identified by named labels. {Key, value}
  - number
    - 200, 400, 349

## Fetch data from List and Map
- Use case- how we can reference the value from one of the part of list and map
- Check section 4 -> attributes.tf

## Count Parameter and Count Index
- Count Parameter
  - The count parameter on resources can simplify configurations and let you scale resources by simply incrementin
g a number.

- let's assume, you need to create two EC2 instances. One of the common approach is to define two separate blocks for aws_instance.

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

```
resource "aws_instance" "myec21" {
    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:

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

- With naming convention

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

## Conditional Expression
- A conditional expression uses the value of a bool expression to select one of two values.
- syntax:
    - condition ? true_val : false_val

## Local Values
- A local value assigns a name to an expression, allowing it to be used multiple times within a module without repeating it.
- syntax
    - create a local file and add tags inside "locals {}".
    - then access this tags in other file by
        - local.tagname
    - terraform plan

- 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 Points for Local Values
- Local values can be helpful to avoid repeating same values or expression multiple times.
- If overused they can also make a configuration hard to read by future maintainers by hiding the actual values used.
- Use locals values only in moderation, in situation where a single value or result is used in many places and that value is likely to be changed in future.

## Terraform Functions (https://developer.hashicorp.com/terraform/language/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 name followed by comma-separated arguments in parentheses:

```
function(argument 1, argument 2)
```

- Example
```
> max(5,12,9)
12
```
- List of available functions
    - The terraform language does not support user-defined functions, and so only functions built-in to the language ar e available for use.
        - Numeric
        - String
        - Collection
        - Encoding
        - Filesystem
        - Date and Time
        - Hash and Crypto
        - IP Network
        - Type Conversion

- try it out with "terraform console" command in cmd.
- and check the main.tf for more info.

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

```
data "aws_ami" "app_ami" {
most_recent = true
owners = ["amazon"]  // Owner= google/amazon/azure/Self etc


    filter {        // adding extra filter to filter-out linux images only
        name   = "name"
        values = ["amzn2-ami-hvm*"]
    }
}

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

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

- Note - Filters in Data Sources
    - If you need to find more details related to options that can be used in filters, you can refer to the following AWS documentation:

    https://docs.aws.amazon.com/cli/latest/reference/ec2/describe-instances.html

    Refer to the --filters option

Note: This additional details is beyond the scope of certification.

## Debugging in 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.

- pointers
  - TRACE is the most verbose and it is 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 log to always be append to specific file when logging is enabled.

## Terraform Format
- Anyone whi is into programming knows the importance of formatting the code for readability.

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

  - cli
  - terraform fmt

## Terraform Validate
- terraform validate primarily checks whether a configuration is syntactically valid.
- it can check various aspects including unsupported arguments, undeclared variables and others.

  - terraform validate

## Load Order & Semantics
- Terraform generally load all the configuartion files within the directory specified in alphabetical order.

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

## Dynamic Blocks
- 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.

- Dynamic
  - Dynamic block allows us to dynamically construct repeatable nested blocks which is supported inside resources, data, provider, and provisioner block.

- Iterator (Optional)
  - The iterator argument 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.

## Tainting Resources
- you created a new resources via terraform
- users have made a lot of manual changes.
- Two ways to deal with it
  - Import the changes to terraform
  - delete & recreate the resource
    - Terraform Taint
      - The terraform taint command manually marks a terraform-managed resources as tainted, forcing it to be destroyed and recreated on the next apply.

        - terraform taint aws_instance.myec2
        - only the local state will be marked as tainted until you apply.

- Pointers
  - This command will not modify the infra it will modify the state file in order to mark a resource as tainted
  - Once a resource is marked as tainted the next plan will show that the resource will be destroyed and recreated.
  - NOTE: That the tainting a resource for recreation may affect resources that depends on the newly tainted resourc es.

## Splat Expression
- Splat Expression allows us to get a list of all attributes.
- You can use in the output section to list all the attributes that are being created.
  - with the help of "*".
  - Example

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

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

## Terraform Graph
- This allows us 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.
  - terraform graph > graph.dot
  - download graphviz.gitlab.io
  - dot -Tpng InputFile.dot -o OutputFile.png (to convert in png)

## Terraform Plan File
- the generated terraform plan can be saved to a specific path.
- this plan can be used with terraform apply to be certain that only changes shown in this plan are applied.
  - Syntax
    - terraform plan -out=path
    - terraform apply "nameoffile"

## Terraform Output
- terraform output command is used to extract the value of an output from the state file.
  - terraform output iam_name

## Terraform Settings
- The special terraform configuration block types is used to configured some behaviors of terraform itself, such as re quiring version to apply your configuration.

  - Terraform Version

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

  - Provider Version

```
- terraform {
    required_providers {
        mycloud = {
            source = "mycorp/cloud"
```

```
                version = "~>1.0"
            }
        }
    }
```

## Dealing with larger Infrastructure
- -refresh=false
- -target=ec2
- -auto-approve (no need to type "yes" again)
- ~ => Update resource
- + => Add resource
- - => Destroy resource

## Zipmap Function
- The zipmap function construct a map from a list of keys and a corresponding list of values.
- syntax
    - cli
    - terraform console
    - zipmap([keys], [values])

## Comments in terraform
- //Comments
- #single line comments
- /*Comments*/

## Data Type - SET
- SET is used to stored multiple items in a single variable.
- SET items are unordered and no duplicates allowed.
    - toset dunction
        - toset function will convert the list of values to set.
            - toset(["a", "b", "c", "a"])
              > toset(["a", "b", "c"])

## for_each (Meta-argument)
- make use of map/set as an index value of the created resource.

```
    resource "aws_iam_user" "iam" {
        for_each = toset(["user-1", "user-2", "user-3"])
        name = each.key
    }
```

# Section-5 :Terraform Provisioners

## Understanding Provisioners in terraform
- Till now, we were just creating, updating, and destroying the empty resources.

- what if we want end to end solution?
- where we have to create a EC2 instance and install a server over it via running a script?

- Here comes the provisioners

```
    resource "aws_instance" "myec2" {

        ami           = "ami-01216e7612243e0ef" //If it is not working change the region and then try
        instance_type = "t2.micro"
        key_name      = "keyname"
        connection {
            type        = "ssh"
            user        = "ec2-user"
            private_key = file("./keyname.pem")
            host        = self.public_ip
        }

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

- commands
    - terraform init
    - terraform plan
    - terraform apply -auto-approve
    - terraform destroy -auto-approve

## Types of Provisioners
- Terraform has capability to turn provisioners both at the same time of resources creation as well as destruction.

- there are more types but mainly we are using two types of provisioners:
    - local-exec
        - Allow us to invoke executable after resources is created.
    - remote-exec
        - Allow us to invoke scripts Directly on the remote server.

## remote-exec provisioners
- Manual Way
    - create a key-pair
    - launch an ec2 with that key-pair
    - store the key-pair
    - CMD -> go the folder where you have store key-pair
    - ssh -i keyname ec2-user@public_ip
    - enter to connect
    - sudo su -
    - yum -y install nginx
    - sudo amazon-linux-extras install nginx1
        - yes
    - systemctl start nginx.service
    - add a rule on port 80 in the security group
        - access through public ip

- via terraform
    - copy the ami id as well
    - if you are not adding the security group then the default security group will be applied, please add ssh and port 8
0 port to it.
    - resource "aws_instance" "myec2" {

```
        ami           = "ami-01216e7612243e0ef" //If it is not working change the region and then try
        instance_type = "t2.micro"
        key_name      = "keyname"

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

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

  - terraform plan
  - terraform apply -auto-approve
  - check via ssh and via public_ip

## local-exec provisioners

```
resource "aws_instance" "myec2" {
  ami           = "ami-0e6329e222e662a52" //If it is not working change the region and then try
  instance_type = "t2.micro"

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

  }
}
```

- cmd
- goto terraform running folder
- echo, if it show echo is on then great
- terraform plan
- terraform apply -auto-approve

## Creation-Time Provisioner
- This only run during the time of creation, not during updating or any other lifecycle.
- If a C-t provisioner fails, the resource is marked as tainted.

## Destroy-Time Provisioners
- This run before the resource is destroyed.

## Failure Behavior for provisioners
- By default, if provisioner fails then terraform apply will fail itself.
- the on_failure setting can used to change this behavior
  - continue
    - ignore the error and continue with creation or destruction.
  - fail
    - raise an error and stop applying. if this is a creation provisioner, taint the resource.

## Null Response
- https://registry.terraform.io/providers/hashicorp/null/latest/docs/resources/resource

# Section-6 : Terraform Modules & Workspaces

## Understanding DRY Principle
- In software engineering, don't repeat yourself (DRY) is a principle of software development aimed at reducing repetition of software patterns.
- In the earlier lecture, we were making static content into variables so that there can be a single source of information.
- We do repeat multiple times various terraform resources for multiple projects.
    - Example
        - we were creating multiple Ec2 resource

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

## Implementing EC2 module with Terraform
- just create two folders
    - modules (create according to your requirement)
        - ec2
```
resource "aws_instance" "myec2" {
    ami           = "ami-0e6329e222e662a52" //If it is not working change the region and then try
    instance_type = "t2.micro"
}
```
        - cloudformation
    - projects
        - A
            - ec2.tf
```
module "ec2module" {
   source = "../../modules/ec2"
}
```

- by doing this whenever you need to create the ec2 instance now you can call with module to create it.
- terraform init
- terraform plan
## Variables & Terraform Modules
- Challenges with modules
    - One common need for infrastructure management is to build environments like staging, production with a simila
r setup but keeping environment variables different.

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

    - If you have hard coded the resources then you will not be able to change it from the root module.

    - Solution
        - it is better to create variable for this

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

        - Now if you add the instance_type in root module folder you will be able to override the default value.

```
module "ec2module" {
    source = "../../modules/ec2"
    instance_type = "t2.large"
```

}

## Using Locals with Modules
- Understanding the Challenge
    - Using variables in Modules can also allow users to override the values which you might not want.

- Setting the Context
    - There can be many repetitive values in modules and this can make your code difficult to maintain.
    - You can centralize these using variables but users will be able to override it.

- NOTE: if you don't want users to override it, then you better use Locals.

    - locals
        - Instead of variables, you can make use of locals to assign the values.
        - You can centralize these using variables but users will be able to override it.

```
locals {
    app_port = 8443
}
```

- More details-> look into project -B

## Module Outputs
- Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use.
- Accessing child module output
    - Adding the resource id to other resource
    - like creating an EC2 instance and assigning the security group id to it via Outputs
    - you need to provide the output in the resources creation file.

## Terraform Registry
- The Terraform Registry is a repository of modules written by the Terraform community.
- The registry can help you get started with Terraform more quickly
- Module Location
    - If we intend to use a module, we need to define the path where the module files are present.
    - The module files can be stored in multiple locations, some of these include:
        ● Local Path
        ● GitHub
        ● Terraform Registry
        ● S3 Bucket
        ● HTTP URLs

- Verified Modules in Terraform Registry
    - Within Terraform Registry, you can find verified modules that are maintained by various third-party vendors.
    - These modules are available for various resources like AWS VPC, RDS, ELB, and others

    - Verified modules are reviewed by HashiCorp and actively maintained by contributors to stay up-to-date and compatible with both Terraform and their respective providers.
    - The blue verification badge appears next to modules that are verified.
    - Module verification is currently a manual process restricted to a small group of trusted HashiCorp partners.

# Section-7: Remote State Management

## Integrating with GIT for team management
- download git for windows

- create an account on Bit-bucket/Github
- crete a repo and clone it in your directory
- add files into that repo and commit and push the changes.

- Little Warning:
    - While committing data to the Git repository, please avoid pushing your access/secret keys with the code. This is very important.

## Security Challenges in committing TFState to GIT
- Even if you create a different file for function, TFSTATE file will be storing the passwords
- be careful when pushing to the repo.
- will be discussing more in section 8

## Module Sources in Terraform.
- The source argument in a module block tells Terraform here to find the source code for the desired child module.
    ● Local paths
        - A local path must begin with either ./ or ../ to indicate that a local path is intended.
    - Git Module
        - Arbitrary Git repositories can be used by refixing the address with the special git:: prefix.
        - After this prefix, any valid Git URL can be specified to select one of the protocols supported by Git.
        - Referencing to a branch
            - By default, Terraform will clone and use the default branch (referenced by HEAD) in the selected repositor y.

            - You can override this using the ref argument:

```
module "demo" {
    source = "git::https://github.com/xyz/xyz.git?ref=development"
}
```

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

    ● Terraform Registry
    ● GitHub
    ● Bitbucket
    ● Generic Git, Mercurial repositories
    ● HTTP URLs
    ● S3 buckets
    ● GCS buckets

## Terraform and GitIgnore
- The .gitignore file is a text file that tells Git which files or folders to ignore in a project.
- Depending on the environment, it is recommended to avoid committing certain files to GIT.
    - .terraform
    - terraform.tfvars
    - terraform.tfstate
    - crash.log

## Terraform Backend
- Backends primarily determine where Terraform stores its state.
- By default, Terraform implicitly uses a backend called local to store state as a local file on disk.

- Challenges
    - Nowadays Terraform projects are handled and collaborated by an entire team.

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

- Ideal Architecture
    - Following describes one of the recommended architectures:
        - The Terraform Code is stored in Git Repository.
        - The State file is stored in a Central backend.

- Terraform supports multiple backends that allow remote service related operations.
Some of the popular backends include:
    - S3
    - Consul
    - Azurerm
    - Kubernetes
    - HTTP
    - ETCD

- Important Note
    - Accessing state in a remote service generally requires some kind of access credentials.

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

## Implementing S3 backends
- go for documentation
- create a bucket in AWS
- add a sub-folder for security and network and many more.
- then go the s3backend folder add required files and specify the names and all
- then download AWS CLI, and open cmd
- aws configure
- then add you keys, default zone and enter
    - then type this command to check you are able to see the sub-folders you have created.
        - aws s3 ls s3://{bucket-name}
- then in cmd got the s3 backend folder
- terraform init
- terraform apply -auto-approve
- then go to the AWS Console and check the terraform.tf state file.

## State locking
- Whenever you are performing write operation, terraform would lock the state file.
- This is very important as otherwise during your ongoing terraform apply operations, if others also try for the same, it can corrupt your state file.

- Important Note
    - State locking happens automatically on all operations that could write state. You won't see any message that it is happening
    - If state locking fails, Terraform will not continue Not all backends support locking. The documentation for each backend includes details on whether it supports locking or not.

- Force Unlocking State
    - Terraform has a force-unlock command to manually unlock the state if unlocking failed.
    - If you unlock the state when someone else is holding the lock it could cause multiple writers.
    - Force unlock should only be used to unlock your own lock in the situation where automatic unlocking failed.

## State locking in S3 backend
- By default, S3 does not support State Locking functionality.

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

- steps:
  - create a backend file and add a resource for 200 seconds sleep.
  - add provider of aws.
  - terraform init
  - then create a dynamodb table with with any name and add the partitionkey as: LockID with string.
  - copy the name and add into the backend code.
  - change the key from tmp to demo.tf
  - terraform init
    - if you get error
      - then
        - terraform init -reconfigure

  - terraform apply -auto-approve
  - open another cmd fast
  - got to the same directory
  - terraform plan
    - you will get an error "Error: Error acquiring the state lock"
  - check the dynamodb as well.
- delete the table as well.

## Terraform State Management
- As your Terraform usage becomes more advanced, there are some cases where you may need to modify the Terrafo rm state.
- It is important to never modify the state file directly. Instead, make use of terraform state command.
- There are multiple sub-commands that can be used with terraform state, these include:
  - list
    - The terraform state list command is used to list resources within a Terraform state.

  - mv
    - The terraform state mv command is used to move items in a Terraform state.
    - This command is used in many cases in which you want to rename an existing resource without destroying an d recreating it.
    - Due to the destructive nature of this command, this command will output a backup copy of the state prior to sa ving any changes
    - Overall Syntax:
      - terraform state mv [options] SOURCE DESTINATIO
  - pull
    - The terraform state pull command is used to manually download and output the state from a remote state.
    - This is useful for reading values out of state (potentially pairing this command with something like jq).
  - push
    - The terraform state push command is used to manually upload a local state file to remote state.
  - rm
    - The terraform state rm command is used to remove items from the Terraform state.
    - Items removed from the Terraform state are not physically destroyed.
    - Items removed from the Terraform state are only no longer managed by Terraform
    - For example, if you remove an AWS instance from the state, the AWS instance will continue running, but terr aform plan will no longer see that instance.

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

## Connecting the Remote states
- The terraform_remote_state data source retrieves the root module output values from some other Terraform config

uration, using the latest state snapshot from the remote backend

- Practical
    - go to the network folder
        - terraform init
        - terraform apply -auto-approve
        - go to aws and cross-check
            - all the things are done
    - now go to security folder
        - and create sg.tf for creating the security group
        - create a file remotestate.tf
        - terraform init
        - terraform apply -auto-approve
        - go to aws and cross-check
            - all the things are done

    - If the ip is changed for that practical
        - network folder
            - terraform destroy -auto-approve
            - terraform apply -auto-approve
            - open the state file
        - security folder
            - terraform apply -auto-approve

    - Completion
        - network folder
            - terraform destroy -auto-approve
        - security folder
            - terraform destroy -auto-approve

## Terraform Import
- It might happen that there is a resource that is already created manually.
- In such a case, any change you want to make to that resource must be done manually

- Terraform is able to import existing infrastructure. This allows you to take resources you've created by some other means and bring it under Terraform management.
- The current implementation of Terraform import can only import resources into the state. It does not generate configuration. A future version of Terraform will also generate configuration.
- Because of this, prior to running terraform import it is necessary to write manually a resource configuration block for the resource, to which the imported object will be mapped.

    - Important Pointer
        - The current implementation of terraform import can only import resources into the state. it does not generate configuration.
        - A future version of terraform will also generate configuration.
        - Because of this, prior to running terraform import it is necessary to write manually a resource configuration block for the resources, to which the imported object will be mapped.

# Security Primer

## Handling Access & Secret Key
- So if you have configured the AWS CLI then you don't have to write Access and secret keys in file and resources.

- please try avoiding the use of Access and secret key in same file or folder better use of variable and tfvars.
and use .gitignore to not send the key files publicly

## Terraform Provider Usecase - Resources in Multiple Regions

- Provider Configuration
    - Till now, we have been hardcoding the aws-region parameter within the providers.tf
    - This means that resources would be created in the region specified in the providers.tf file.
    - By default, resources use a default provider configuration inferred from the first word of the resource type name.

    - For example, a resource of type aws_instance uses the default (un-aliased) aws provider configuration unless otherwise stated.
    - To select an aliased provider for a resource or data source, set its provider meta-argument to a <PROVIDER NAME>.<ALIAS> reference

    - For multiple region
        - alias = "Singapore" in provider Block
        - provider = aws.Singapore in resource Block
        - terraform init
        - terraform apply -auto-approve

- For Multiple Accounts

## Handling Multiple AWS Profiles with Terraform Provider

- You can optionally define multiple configurations for the same provider, and select which one to use on a per-resource or per-module basis.
- The primary reason for this is to support multiple regions for a cloud platform.
- To include multiple configurations for a given provider, include multiple provider blocks with the same provider name, but set the alias meta-argument to an alias name to use for each additional configuration.

- Profiles
    - just go to your .aws folder in C drive and check for credentials file.
    - add one more for [account02] after default
    - to reference this
        - add in provider
            - profile = "account02"

```
provider "aws" {
    alias   = "account02"
    region  = "ap-southeast-1"
    profile = "account02"
}
```

    - terraform init
    - terraform apply -auto-approve

## Note - STS

- The below video named "Terraform & Assume Role with AWS STS" is beyond the scope of official Terraform exams.

- If you are not familiar with AWS in-detail, you can skip the next video.

- We have kept this specific video available in the course on behalf of multiple requests by students from AWS background who needed to understand integration of Terraform with AWS STS.

## Sensitive parameter

- With the organization managing its entire infrastructure in terraform, it is likely that you will see some sensitive inf

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

- Setting the sensitive to “true” will prevent the field's values from showing up in CLI output and in
Terraform Cloud
- It will not encrypt or obscure the value in the state, however.

## Note - HashiCorp Vault
- The below video of "Overview of HashiCorp Vault" is taken from our HashiCorp Certified Vault Associate 2022 v
ideo course.

- Since in the Terraform exams, you can expect a question about Vault Integration, we have decided to add a overvie
w of Vault video in this course.

- Once you understand the basics of what Vault is, we will then discuss about it's integration with Terraform.

## HashiCorp Vault
- HashiCorp Vault allows organizations to securely store secrets like tokens, passwords, certificates along with acces
s management for protecting secrets.
- One of the common challenges nowadays in an organization is “Secrets Management”
- Secrets can include database passwords, AWS access/secret keys, API Tokens, encryption keys and others Vault ca
n also generate dynamic secrets like AWS Access/Secret keys that has validity for a limited time
- Dynamic Secrets
    - Once Vault is integrated with multiple backends, your life will become much easier and you can focus more on t
he right work.
    - Major aspects related to Access Management can be taken over by vault.

## Terraform & Vault Integration
- Vault Provider
    - The Vault provider allows Terraform to read from, write to, and configure HashiCorp Vault.

- Important Note:
    - Interacting with Vault from Terraform causes any secrets that you read and write to be persisted in Terraform's st
ate file

# Section-9: Terraform Cloud & Enterprise Capabilities

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

- Terraform Cloud is available as a hosted service at https://app.terraform.io.

- Create a terraform cloud Account for free

## Creating infra with terraform cloud
- create a repo in github private
- add code below in ec2.tf

```
    provider "aws" {
        region     = "ap-south-1"
    }

    resource "aws_instance" "myec2" {
        ami = "ami-01216e7612243e0ef"
        instance_type = "t2.micro"
    }
```

- add a workspace with "Version Control" in terraform cloud connect with github with that repo

- set variable
    - add access key and secret key as a environment variable
        - https://registry.terraform.io/providers/hashicorp/aws/latest/docs
- then got to action
    - plan and apply
        - start run
    - it will show you the plan
    - then confirm and apply to create the ec2 instance
    - to destroy
        - go to destruction & deletion
            - queue destroy plan and delete

## Overview of Sentinel
- Sentinel is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products.
- It enables fine-grained, logic-based policy decisions, and can be extended to use information from external sources.

- Note: Sentinel policies are paid feature

## Remote Backends
- The remote backend stores terraform state and may be used to run operations in terraform cloud.
- Terraform cloud can also be used with local operations, in which case only state is stored in the terraform cloud.

- Terraform supports various types of remote backends which can be used to store state data.
- As of now, we were storing state data in local and GIT repository.
- Depending on remote backends that are being used, there can be various features.
    ● Standard BackEnd Type: State Storage and Locking
    ● Enhanced BackEnd Type: All features of Standard + Remote Management
- When using full remote operations, operations like terraform plan or terraform apply can be executed in Terraform Cloud's run environment, with log output streaming to the local terminal.
- Remote plans and applies use variable values from the associated Terraform Cloud workspace.
- Terraform Cloud can also be used with local operations, in which case only state is stored in the Terraform Cloud b ackend.

## Remote operation Implementation
- create a new workspace of "CLI driven", then
- read CLI Driven Runs
- copy the provided code and create a new file remote-backend.tf.

```
    terraform {
        cloud {
            organization = "Organisation name"

            workspaces {
                name = "Remote-Operation"
            }
        }
    }
```

- create iam.tf file and add below code with provider and keys

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

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

- Now we need to login via CLi into terraform cloud
    - go to folder in cmd
    - terraform login
        - If login is successful, Terraform will store the token in plain text in the following file for use by subsequent commands:
            - C:\Users\{username}\AppData\Roaming\terraform.d\credentials.tfrc.json
        - yes
        - it will automatically open a web page and create a token and copy and paste in cmd
    - terraform init
    - terraform plan
    - terraform apply -auto-approve
    - terraform destroy -auto-approve

## Air Gapped Environments
- Understanding Concept of Air Gap
    - An air gap is a network security measure employed to ensure that a secure computer network is physically isolated from unsecured networks, such as the public Internet.

- Usage of Air Gapped Systems
    - Air Gapped Environments are used in various areas Some of these include:
        - Military/governmental computer networks/systems
        - Financial computer systems, such as stock exchanges
        - Industrial control systems, such as SCADA in Oil & Gas fields

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