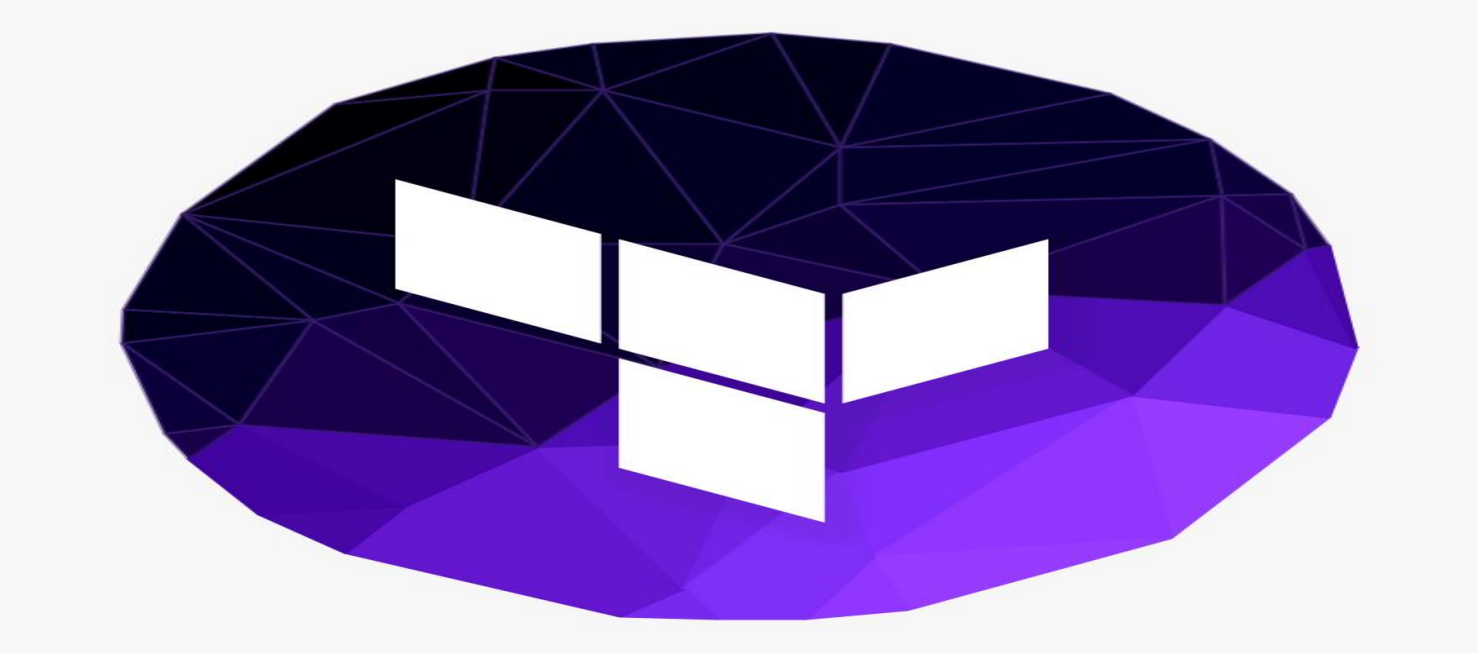


Infrastructure as code (IAC) is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools

What is Terraform?

- Terraform is an IT infrastructure automation tool for building, changing, and versioning infrastructure safely and efficiently
- Terraform can manage existing as well as custom in-house solutions
- It turns your infrastructure as code IAC i.e your computing environment has some of the same attributes as your application:
  - Your infrastructure is versionable.
  - Your infrastructure is reusable.
  - Your infrastructure is testable.
- Minimizes errors & security violations
- You only need to tell what the desired state should be, not how to achieve it

Why Terraform?

- Terraform is "Declarative" not "Procedural/Imperative

- Terraform gives Immutable Infrastructure
- Terraform is cloud-agnostic
- Scalable & Reproducible Infrastructure

Features of Terraform

Infrastructure as Code

- Infrastructure is described using a high-level configuration syntax
- Provides single unified syntax

Execution Plans

- Terraform has a "planning" step where it generates an execution plan
- The execution plan shows what Terraform will do before making the actual changes

Resource Graph

- Terraform builds a graph of all your resources, and parallelizes the creation and modification of any non-dependent resources
- Terraform builds infrastructure as efficiently as possible

Change Automation

- Complex change sets can be applied to your infrastructure with minimal human interaction
- With the previously mentioned execution plan and resource graph, you know exactly what Terraform will change and in what order, avoiding many possible human errors.

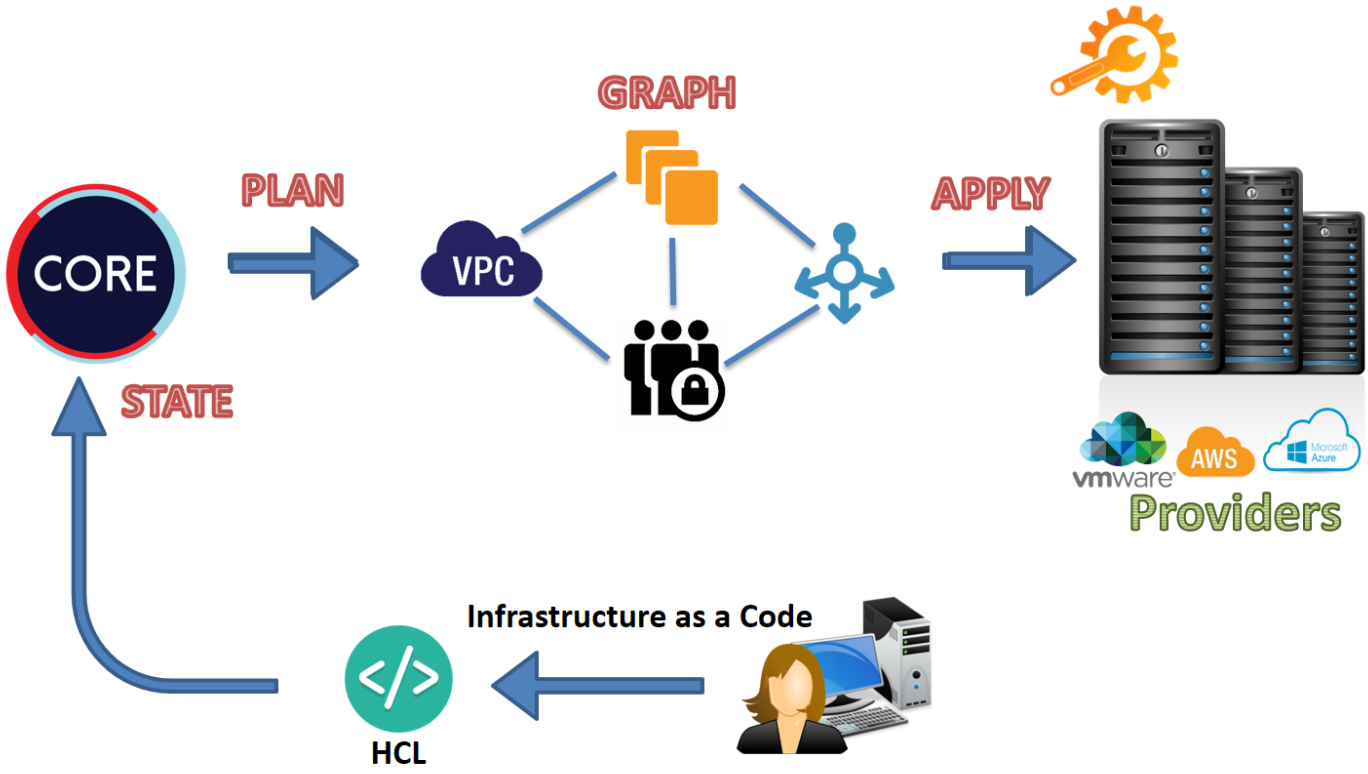
How Terraform Works?

Terraform allows infrastructure to be expressed as code in a simple, human readable language called HCL (HashiCorp Configuration Language). It reads configuration files and provides an execution plan of changes, which can be reviewed for safety and then applied and provisioned.

Extensible providers allow Terraform to manage a broad range of resources, including IaaS, PaaS, SaaS, and hardware services.

Terraform allows infrastructure to be expressed as code in a simple, human readable language called HCL (HashiCorp Configuration Language). It reads configuration files and provides an execution plan of changes, which can be reviewed for safety and then applied and provisioned.

Extensible providers allow Terraform to manage a broad range of resources, including IaaS, PaaS, SaaS, and hardware services.



Difference between Terraform vs Ansible vs Chef/Puppet

Factor	Terraform	Ansible	Puppet/Chef
Type	Infrastructure Orchestration tool	Configuration Management tool	Configuration Management tool
Infrastructure	Immutable	Mutable	Mutable
Language	Declarative	Procedural	Declarative/Procedural
Architecture	Agentless	Agentless	Client/Server
Syntax	HCL	YAML	Ruby
Method	Plan & Apply	Push	Pull

Setup Terraform on AWS(Ubuntu Server)

```
$ curl -O https://releases.hashicorp.com/terraform/0.13.0/terraform_0.13.0_linux_amd64.zip

$ apt install -y unzip

$ unzip terraform_0.13.0_linux_amd64.zip -d /usr/local/bin/

$ terraform version
```

HCL (HashiCorp Configuration Language)

- Terraform code is written in a language called HCL in files with the extension .tf
- Terraform can also read JSON configurations that is named with the .tf.json
- It is a declarative language, so your goal is to describe the infrastructure you want, and Terraform will figure out how to create it
- HCL syntax is composed of blocks that define a configuration in Terraform
- Blocks are comprised of key = value pairs
- Single line comments start with # & Multi-line are commented with /* and */
- Strings are in double-quotes, Boolean values: true, false
- Lists values are made with square brackets ([]), for example ["foo", "bar", "baz"]
- Maps can be made with braces ({}) and colons (:), for example { "foo": "bar", "bar": "baz" }
- Strings can interpolate other values using syntax wrapped in ${}, for example ${var.foo}

Terraform Providers

- Providers are API's which interact with the infrastructure components like AWS, Docker, Kubernetes
- Manages the lifecycle of the resources

- Providers generally are an IaaS (e.g. AWS, GCP, Azure, OpenStack), PaaS (e.g. Heroku), or SaaS services (e.g. Terraform Cloud, DNSimple, CloudFlare)

Syntax:

```
provider <PROVIDER> {
  [CONFIG ...]
    ARGS ...
}
```

- ❑ Where PROVIDER is the name of a provider (e.g., Docker, AWS),
- ❑ CONFIG consists of one or more arguments that are specific to that provider (e.g., host for Docker or Region for AWS").

Terraform Resources

- Resources are the components of your infrastructure
- It might be some low level component such as a physical server, virtual machine, or container. Or it can be a higher level component such as an email provider, DNS record, or database provider
- The resource block creates a resource of the given TYPE (first parameter) and NAME (second parameter). The combination of the type and name must be unique
- Within the block is the configuration for each resource. The configuration is dependent on the type

Syntax:

```
resource <PROVIDER_TYPE> <NAME> {
  [CONFIG ...]
    ARGS ...
}
```

- ❑ Where PROVIDER is the name of a provider (e.g., aws),
- ❑ TYPE is the type of resources to create in that provider (e.g., instance),
- ❑ NAME is an identifier you can use throughout the Terraform code to refer to this resource (e.g., example),
- ❑ CONFIG consists of one or more arguments that are specific to that resource (e.g., ami = "ami-0c55b159cbfafe1f0").

Terraform Commands

$ terraform init

- Used to initialize a working directory containing Terraform configuration files
- First command that should be run after writing a new Terraform configuration
- Will install Terraform modules & providers plugins

$ terraform validate

- Validate runs checks that verify whether a configuration is syntactically valid and internally consistent
- useful for general verification of reusable modules, including correctness of attribute names and value types
- Validation requires an initialized working directory with any referenced plugins and modules installed

$ terraform plan

- Used to create an execution plan
- By reading the configuration files terraform determines what actions are necessary to achieve the desired state specified
- Allows a user to see which actions Terraform will perform prior to making any changes, increasing confidence that a change will have the desired effect once applied
- Should be run before committing a change to version control, to create confidence that it will behave as expected
- The optional -out argument can be used to save the generated plan to a file for later execution

$ terraform providers

- Prints information about the providers used in the current configuration
- Command gives an overview of all of the current dependencies, as an aid to understanding why a particular provider is needed
- List providers in the folder: ls .terraform/plugins/linux_amd64/

$ terraform apply [plan]

- Used to apply the changes required to reach the desired state of the configuration
- Scans the current directory for the configuration and applies the changes on a default main.tf
- Useful flags for apply:

-auto-approve: This skips interactive approval of plan before applying.
-var 'foo=bar': This sets a variable in the Terraform configuration

$ terraform show

- Used to provide human-readable output from a state or plan file
- Inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it

$ terraform destroy

- Used to destroy the Terraform-managed infrastructure
- This will ask for confirmation before destroying. If -auto-approve is set, then the destroy confirmation will not be shown

$ terraform console

- Provides an interactive command-line console for evaluating expressions
- Useful for testing interpolations before using them in configurations, and for interacting with any values currently saved in state

Tainting and Updating Resources:

taint: Manually mark a resource for recreation

untaint: Manually unmark a resource as tainted

- This command will not modify infrastructure, but does modify the state file in order to mark a resource as tainted.
- Once a resource is marked as tainted, the next plan will show that the resource will be destroyed and recreated and the next apply will implement this change.
- Taint will only for the next immediate apply

Tainting a resource:

$ terraform taint [RESOURCE-TYPE.NAME]

Untainting a resource:

$ terraform untaint [RESOURCE-TYPE.NAME]

Example: Refer Class Notes

Resource Dependencies (Implicit & Explicit)

- Terraform, via interpolation syntax, allows us to reference any other resource it manages using the following syntax: RESOURCE_TYPE.RESOURCE_NAME.ATTRIBUTE_NAME
- By using this interpolation Terraform can automatically infer when one resource depends on another.
- When you add a reference from one resource to another, you create an implicit dependency.
- Terraform parses these dependencies, builds a dependency graph from them, and uses that to automatically figure out in what order it should create the resources

- Sometimes there are dependencies between resources that are not visible to Terraform.
- The depends_on argument is accepted by any resource and accepts a list of resources to create explicit dependencies

Example: Refer Class Notes

Terraform Variables

- Variables can be defined by the Terraform files and provided when executing a command.
- They give more flexibility to our configurations and let us deploy the same elements in different configurations
- We can also provide a value at runtime by using an environment variable or a command-line parameter.
- Use interpolation to obtain the value of variable using var.NAME
- Terraform variables are recommended to be stored in their own variables file
- Variables are normally defined within a file named variables.tf, Terraform will load all files ending in .tf, so you can also define variables in files with other names
- Each block declares a single variable.
  - ✓ type (Optional): If set, this defines the type of the variable. Valid values are string, list, and map.
  - ✓ default (Optional): This sets a default value for the variable. If no default is provided, Terraform will raise an error if a value is not provided by the caller.
  - ✓ description (Optional): A human-friendly description for the variable

Using Input Variables:

- Runtime Values should be passed using -var 'VARIABLENAME=VALUE“

```
$ terraform plan  -var 'ext_port=8080'
$ terraform destroy  -var 'ext_port=8080'
```

Using Variable Files:

- Values can also be passed through .tfvars files. These files use the same syntax as Terraform configuration & be invoked using -var-file=“file.tfvars”

```
$ terraform apply -var-file="testvars.tfvars“
$ terraform destroy -var-file="testvars.tfvars"
```

Example: Refer Class Notes

Terraform Output

- Used to extract the value of an variable from the state file
- Output variables can also be stored in their own output variables file named output.tf
- Output values return values of a Terraform module
- outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>

Example: Refer Class Notes

Data Source

- Data sources allows a Terraform configuration to make use of information defined outside of Terraform, or defined by another separate Terraform configuration
- Data sources present read-only views into pre-existing data, or they compute new values on the fly within Terraform itself
- Every data source in Terraform is mapped to a provider
- A data source is accessed via a special kind of resource known as a data resource, declared using a data block:
- The data block creates a data instance of the given TYPE (first parameter) and NAME (second parameter). The combination of the type and name must be unique

- Each data instance will export one or more attributes, which can be interpolated into other resources using variables of the form data.TYPE.NAME.ATTR

  Syntax:

```
data [type] [NAME] {
  ARG..
  filter{
     ARG...
  }
}
```

Example: Refer Class Notes

Terraform State

- Terraform stores state about your managed infrastructure and configuration.
- This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.
- This state is stored by default in a local file named "terraform.tfstate" (json file), but it can also be stored remotely, which works better in a team environment
- There is also a backup of the previous state in the terraform.tfstate.backup
- Terraform uses this local state to create plans and make changes to your infrastructure.
- Prior to any operation, Terraform does a refresh to update the state with the real infrastructure
- You can keep the terraform.tfstate in Git, which gives you the history
- Using a remote store will ensure you always have the latest version of the state
- It avoids commit & push
- Add a file backend.tf
- Terraform state can be saved remotely, using the backend functionality
  - ✓ s3
  - ✓ consul
  - ✓ terraform enterprise

Example: Refer Class Notes

Terraform Workspace

- Workspaces are core logical construct in which we operate
- A workspace consists of:
  - ✓ A Terraform configuration
  - ✓ Values for variables used by the configuration.
  - ✓ Persistent stored state for the resources the configuration manages
- Terraform starts with a single workspace named "default"
- With Multiple workspace allows to freely experiment with changes without affecting the default workspace
- The default workspace might correspond to the "master" branch, which describes the intended state of production infrastructure
- Non-default workspaces are often related to feature branches in version control
- For local state, Terraform stores the workspace states in a directory called terraform.tfstate.d

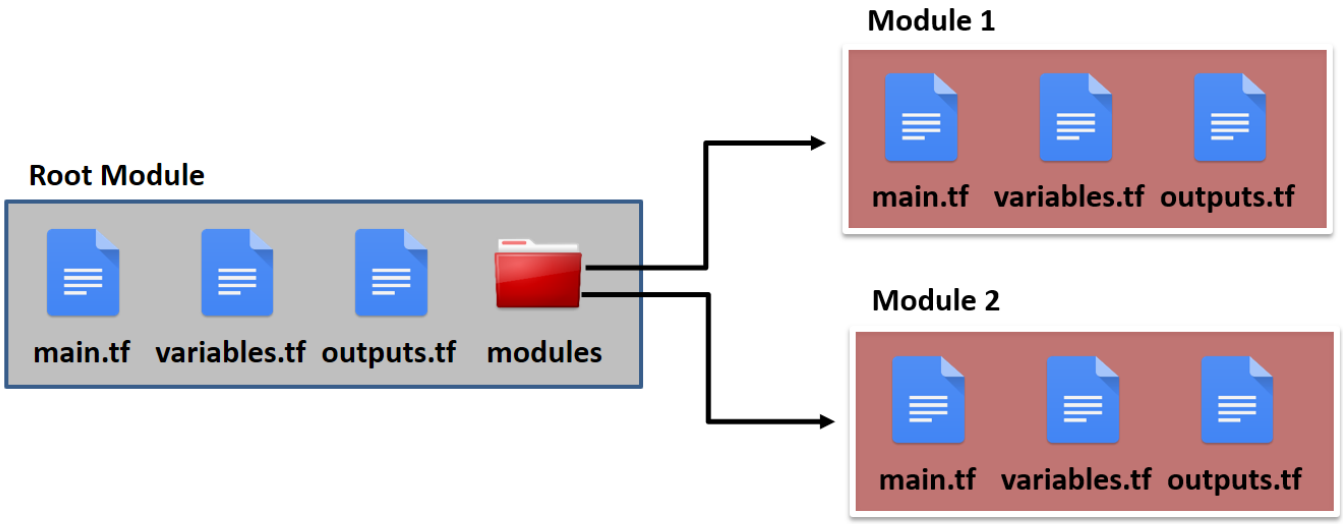
```
$ terraform workspace show
$ terraform workspace list
$ terraform workspace new <name>
$ terraform workspace select dev
$ terraform workspace delete dev
```

Example: Refer Class Notes

Terraform Modules

- For growing infra editing single configuration has a few key problems: a lack of organization, a lack of reusability, and difficulties in management for teams
- Modules in Terraform are self-contained packages of Terraform configurations that are managed as a group
- Modules are used to create reusable components, improve organization, and to treat pieces of infrastructure as a black box.
- Every terraform configuration has atleast 1 module which is "root" module
- The Terraform Registry includes a directory of ready-to-use modules for various common purposes, which can serve as larger building-blocks for your infrastructure.

- These are the recommended filenames for a minimal module
  - ❑ main.tf
  - ❑ variables.tf
  - ❑ outputs.tf
- main.tf should be the primary entrypoint
- Nested modules should exist under the modules/ subdirectory



Defining Root module:

- To call a module means to include the contents of that module into the configuration with specific values for its input variables.
- Modules are called from within other modules using module blocks:

```
module "MODULE_NAME" {
  ARG...
}
```

- The keyword immediately after the module keyword is the name of the module to be called
- Within the block body (between { and }) are the arguments for the module.
- Most of the arguments correspond to input variables defined by the module.
- root module should use relative paths like ./modules/MODULE-NAME

Example: Refer Class Notes

Terraform Loops

- A common problem in Terraform configurations for versions 0.11 and earlier is dealing with situations where the number of values or resources is decided by a dynamic expression rather than a fixed count
- The general problem of iteration is a big one to solve and Terraform 0.12 introduces a few different features to improve these capabilities, namely

✓ count parameter: loop over resources.
✓ for_each expressions: loop over resources and inline blocks within a resource.
✓ for expressions: loop over lists and maps.

Example: Refer Class Notes

Terraform Conditionals

- Terraform doesn't support if-statements directly, however you can accomplish the same thing by using the count parameter
- If you set count to 1 on a resource, you get one copy of that resource; if you set count to 0, that resource is not created at all
- Terraform supports conditional expressions of the format <CONDITION> ? <TRUE_VAL> : <FALSE_VAL>
- Will evaluate the boolean logic in CONDITION, and if the result is true, it will return TRUE_VAL, and if the result is false, it'll return FALSE_VAL

Example: Refer Class Notes

Terraform Provisioners

- Terraform not only helps us in infrastructure creation and management but also in provisioning them using a feature named Provisioner
- Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction.
- Provisioners help in:
  - ✓ local script execution on the machine from where we are running terraform
  - ✓ remote script execution on resource
  - ✓ file or directory copy on the remote resource
  - ✓ configure and run configuration management tools on the remote resource

Creation-Time Provisioners:

- By default, provisioners run when the resource they are defined within is created.
- Creation-time provisioners are only run during creation, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system.
- If a creation-time provisioner fails, the resource is marked as tainted. A tainted resource will be planned for destruction and recreation upon the next terraform apply
- Terraform does this because a failed provisioner can leave a resource in a semi-configured state, the only way to ensure proper creation of a resource is to recreate it

Destroy-Time Provisioners :

- If when = "destroy" is specified, the provisioner will run when the resource it is defined within is destroyed.
- Destroy provisioners are run before the resource is destroyed.
- If they fail, Terraform will error and rerun the provisioners again on the next terraform apply

Multiple Provisioners :

- Multiple provisioners can be specified within a resource.
- Multiple provisioners are executed in the order they're defined in the configuration file.
- You may also mix and match creation and destruction provisioners. Only the provisioners that are valid for a given operation will be run in order.

local-exec Provisioner

- The local-exec provisioner invokes a local executable after a resource is created.
- This invokes a process on the machine running Terraform, not on the resource.

Provisioner Connection Settings

- The provisioners help to interact with remote servers over SSH or WinRM
- You must include a connection block so that Terraform will know how to communicate with the server.

- Connection blocks don't take a block label, and can be nested within either a resource or a provisioner.
- A connection block nested directly within a resource affects all of that resource's provisioners.
- A connection block nested in a provisioner block only affects that provisioner, and overrides any resource-level connection settings.
- Expressions in connection blocks cannot refer to their parent resource by name.
- Instead, they can use the special self object. For example, use self.public_ip to reference an aws_instance's public_ip attribute.

```
connection {
  type     = "ssh|winwrm"
  user     = "username“
  private_key = “file”
  host     = “hostname”
  timeout  = sec
}
```

File Provisioner

- The file provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource.

Remote-Exec Provisioner

- The remote-exec provisioner invokes a script on a remote resource after it is created.
- This can be used to run a configuration management tool, bootstrap into a cluster, etc.

Provisioners without a Resource

- If you need to run provisioners that aren't directly associated with a specific resource, you can associate them with a null_resource.
- Instances of null_resource are treated like normal resources, but they don't do anything.

Debugging Terraform

- Terraform has detailed logs which can be enabled by setting the TF_LOG environment variable to any value.
- You can set TF_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs.
- To persist logged output you can set TF_LOG_PATH in order to force the log to always be appended to a specific file when logging is enabled.

```
$ export TF_LOG=TRACE

$ export TF_LOG_PATH=terraformlog.txt
```

Example: Refer Class Notes

Terraform Best Practices

- Versioning the terraform configuration file - git repo
- Structuring the configuration files - main.tf, variables.tf, output.tf, backend.tf
- Define Modules
- Distributed state - S3 versioning
- Credentials - Environment variable, file, assume role
- Pass runtime values using variables - file, -var option
- Set Log Level for debugging

PROJECTS : Refer Class Notes

- Project 1 - Setting VPC, IGW, RT, SG on AWS
- Project 2 - Setting ELB, ASG, SG on AWS
- Project 3 - Creating IAM, Policies & assigning to users