TERRAFORM

1. **terraform init**: Initializes a Terraform working directory by installing the necessary plugins and setting up the backend.
2. **terraform plan**: Generates and shows an execution plan to deploy changes to the infrastructure.
3. **terraform apply**: Applies changes to the infrastructure as described in the Terraform plan.
4. **terraform show**: Displays the state of the infrastructure managed by Terraform.
5. **terraform destroy**: Destroys the infrastructure managed by Terraform.
6. **terraform validate**: Validates the Terraform configuration files.
7. **terraform fmt**: Formats the Terraform configuration files to adhere to the style conventions.
8. **terraform state**: Manages the Terraform state, including pushing, pulling, and removing state files.

The Terraform state file is a record of the infrastructure managed by Terraform. It keeps track of the resources that have been created, the properties of those resources, and the relationships between resources. The state file is used by Terraform to determine what actions are necessary to bring the infrastructure into the desired state specified in your Terraform configuration files.

The **terraform taint** command is used to mark a resource as "tainted," which tells Terraform that it should be destroyed and recreated on the next **terraform apply** run. Tainting a resource is useful when you need to make changes to a resource that Terraform is unable to automatically detect or manage, such as updating a custom script or configuration that is baked into a virtual machine image.

The **terraform lock** command in Terraform is used to lock the Terraform state file, preventing other users from making changes to the infrastructure while changes are being made. This helps to prevent collisions or conflicts between multiple users who are working on the same Terraform configuration.

Command-----terraform lock [options] [lock\_id]

A Terraform "null resource" The null\_resource in Terraform is a resource that does not create or manage any infrastructure. Instead, it is used as a placeholder resource that can be used to trigger arbitrary actions, such as running local or remote scripts, executing commands, or generating files.

Terraform provisioners are scripts or executables that are run after Terraform creates or updates a resource. Provisioners are used to perform additional configuration or setup tasks that are required for a resource to function correctly.

There are several types of provisioners available in Terraform, including:

* **local-exec**: Runs a command locally on the machine running Terraform.
* **remote-exec**: Runs a command on the target resource over a secure connection.
* **file**: Copies files to the target resource.

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

provisioner "local-exec" {

command = "echo 'Provisioner running on local machine.'"

}

}

Terraform backends are the way Terraform stores its state, the record of the infrastructure it manages. By default, Terraform stores the state locally in a file named **terraform.tfstate**. However, this can be problematic for larger infrastructure deployments where multiple people or processes need to access the state, or for teams who want to store their state in a centralized location.

Terraform backends provide a way to store the state in a more robust and scalable manner, such as in a remote object store like Amazon S3, or in a centralized database like Terraform Enterprise. Some common features of Terraform backends include:

* State locking: A mechanism to prevent multiple users or processes from making conflicting changes to the state at the same time.
* State versioning: A way to keep a history of the state over time, so that it's possible to revert to previous versions if necessary.
* Remote access: The ability for Terraform to access the state from remote locations, making it possible to manage infrastructure from multiple machines or processes.

terraform {

backend "s3" {

bucket = "my-terraform-state"

key = "state.tfstate"

region = "us-west-2"

}

}

Terraform meta-arguments are arguments that are used to modify the behavior of Terraform commands. Meta-arguments are passed to Terraform commands in order to change the way that Terraform behaves during the execution of the command.

Here are some common Terraform meta-arguments:

* **-var** or **-var-file**: Used to pass variables to Terraform, either directly on the command line or from a file.
* **-target**: Used to target a specific resource or set of resources for Terraform to manage, rather than managing all resources in the configuration.
* **-state**: Used to specify the location of the Terraform state file, if it's not stored in the default location.
* **-backend-config**: Used to specify configuration for the backend, such as the bucket name for an S3 backend.
* **-lock**: Used to lock or unlock the state.
* **-input**: Used to specify whether Terraform should prompt for missing variables or use default values.

Terraform functions are used to transform and manipulate data in Terraform configurations. They provide a way to perform operations on data, such as string manipulation, mathematical calculations, and logical operations, and they allow you to use the results of those operations in your Terraform configurations.

Here are some common Terraform functions:

* **length**: Returns the length of a string or list.
* **lookup**: Searches for a value in a map or list.
* **format**: Formats a string using a template.
* **concat**: Joins multiple strings or lists together.
* **lower**: Converts a string to lowercase.
* **upper**: Converts a string to uppercase.
* **replace**: Replaces all instances of one string with another.
* **md5**: Returns the MD5 hash of a string.

Terraform modules are self-contained packages of Terraform configurations that are designed to be easily reused and shared between different Terraform configurations. Modules encapsulate a specific set of infrastructure resources and their dependencies, and they provide a way to organize and abstract Terraform configurations in a modular and reusable manner

module "web\_server" {

source = "./web\_server\_module"

region = var.region

instance\_count = var.instance\_count

}

Modules are typically defined in a separate directory, and they contain a set of Terraform configuration files, variables, outputs, and other resources. They can be used in other Terraform configurations by calling the module and passing in the required inputs.

Outputs are defined in Terraform configuration files, and they can be used to display information about any Terraform resource, including values of Terraform variables, attributes of Terraform resources, or the result of Terraform expressions.

The **count** argument in Terraform is used to specify the number of instances of a resource that should be created. It provides a way to repeat the creation of a resource multiple times, and it allows you to manage large numbers of similar resources in a scalable and efficient manner.

he **for\_each** argument can be used with any Terraform resource, and it can be set to a map or a list of values. By using the **for\_each** argument, you can simplify and streamline the process of managing large numbers of similar resources, and you can reduce the amount of duplicated code in your Terraform configurations.

Terraform import is a command used to import existing infrastructure into Terraform. Terraform is an infrastructure as code (IaC) tool that allows you to manage your infrastructure resources in a versioned and automated way. By using Terraform import, you can take existing infrastructure resources, such as virtual machines or S3 buckets, and import them into Terraform state.

State file unlock

1. Identify the Terraform process holding the lock: To determine which Terraform process is holding the lock, you can run the following command in the directory where your Terraform files are located:

cssCopy code

terraform force-unlock [LOCK\_ID]

Replace **[LOCK\_ID]** with the ID of the lock that you want to remove.

1. Remove the state lock: Once you've identified the Terraform process holding the lock, you can remove the lock by running the following command:

cssCopy code

terraform force-unlock [LOCK\_ID]

Replace **[LOCK\_ID]** with the ID of the lock that you want to remove.

If you don't know the ID of the lock, you can remove all locks by running:

Copy code

terraform force-unlock

1. Retry the Terraform command: Once you've removed the lock, you can retry the Terraform command that was previously failing.

1.terraform destroy -target=aws\_security\_group.elb\_sg

1. Check your system's logs to see if there are any related errors or warnings. For example, you can check the output of the **dmesg** command or the system log file **/var/log/messages**.

To find the OS on Linux servers------Cat /etc/os-release

Memory-free

Disk free-df-h