**Terraform**

**Terraform** is an open-source infrastructure as code (IaC) tool developed by HashiCorp. It allows you to define and manage your infrastructure in a declarative manner, meaning you describe the desired state of your infrastructure using configuration files rather than scripting the individual steps to create and manage resources.

**Key features:**

**1. Declarative Configuration:** You define your infrastructure as code using HashiCorp Configuration Language (HCL) or JSON syntax. This code represents the desired state of your infrastructure.

**2. Provider-Neutral:** Terraform supports a wide range of cloud providers (AWS, Azure, Google Cloud, etc.) and other infrastructure technologies (Docker, Kubernetes, databases, etc.) through provider plugins.

**3. State Management:** Terraform maintains a state file that keeps track of the current state of your infrastructure. This helps Terraform understand what resources are managed and allows it to track changes over time.

**4. Plan and Apply:** Terraform provides a planning phase (`terraform plan`) that generates an execution plan of what changes will be made to your infrastructure before you apply them (`terraform apply`).

**5. Idempotent Operations:** Terraform ensures that applying a configuration multiple times results in the same outcome. It only makes necessary changes to bring the infrastructure to the desired state.

**6. Modules:** Terraform supports modularization, allowing you to encapsulate and reuse parts of your infrastructure code. Modules can represent entire systems or specific components.

**7. Versioning and Collaboration:** Infrastructure code can be stored in version control systems, enabling collaboration and code reviews just like software development.

**Here's a basic example of a Terraform configuration file that creates an AWS EC2 instance:**

***create-ec2.tf***

provider "aws" {

region = "us-west-1"

}

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

}

***To use this configuration:***

1. Run `terraform init` to initialize the directory.

2. Run `terraform plan` to see the changes that will be made.

3. Run `terraform apply` to create the AWS instance.

**Note:**

1. Terraform is an open source s/w created by HashiCorp and written in Go programming language
2. Terraform is an infrastructure as code (IaC) software tool,
3. Infrastructure as code is the process of managing infrastructure in a file or files rather than manually configuring resources using user interface (UI)
4. In Terraform resources are nothing but Virtual machines, Elastic IPs, Security Groups, Network interfaces, RDS, LBR etc..
5. Terraform code is written in the HashiCorp Configuration Language (HCL) in files with the extension ***.tf***
6. Terraform allows users to use HashiCorp Configuration Language (HCL) to create the files containing definitions of the desired resources.
7. Terraform Supports all most all cloud providers (AWS, AZURE, GCP, Openstack etc..)
8. To automate infrastructure creation in cloud platforms we will use Terraform.

**Terraform vs Cloud Formation**

* Terraform developed by HashiCorp
* CloudFormation developed by AWS
* Terraform supports many cloud providers
* Cloud Formation supports only in AWS
* Terraform uses HashiCorp configuration language (HCL) which built by HashiCorp. It is fully compatible with JSON.
* AWS Cloud Formation utilizes either JSON or YAML.

**Terraform Vs Ansible**

* Terraform developed by HashiCorp.
* Ansible is also an open source software.
* Terraform is an infrastructure as a Code, which means they are designed to provision the servers themselves.
* Ansible is a configuration management tool. Which means Ansible designed for install and manage software on existing servers.
* Terraform is ideal for creating, managing and improving infrastructure.
* Ansible is ideal for software provisioning, application deployment and configuration management.

**Terraform Setup - Pre-Requisites**

1) Cloud Platform Account (AWS, Azure, GCP, Openstack etc..)

2) IAM User account (Secret Key and Access Key)

3) IAM User should have resources Access

**Terraform Installation**

**Ref:** [**https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli**](https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli)

1) Create EC2 instance (RED HAT Linux)

2) Connect to EC2 VM using Mobaxterm

3) Swith to root user

$ sudo su -

4) Install unzip software

$ sudo yum install wget unzip vim -y

5) Download Terraform Software (https://www.terraform.io/downloads)

$ sudo yum install -y yum-utils

$ sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo

$ sudo yum -y install terraform

6) Check Terraform Version

$ terraform -v

**WORKING WITH EC2 INSTANCE USING TERRAFORM**

1) Create IAM user with Programmatic Access (IAM user should have EC2FullAccess)

2) Download Secret Key and Access Key

3) Write First Terraform Script

$ mkdir terraformscript

$ cd terraformscripts

$ vi FirstTFScript.tf

provider "aws" {

region = "ap-south-1"

access\_key = "AKIAWZADR3RJTUF7L76B"

secret\_key = "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"

}

resource "aws\_instance" "AWSServer" {

ami = "ami-0f5ee92e2d63afc18"

instance\_type = "t2.micro"

key\_name = "linux"

security\_groups = ["default"]

tags = {

Name = "MyEC2-VM-Ubuntu"

}

}

10) Initialize Terraform using init command

$ terraform init

11) Format your script (indent spaces)

$ terraform fmt

12) Validate Your Script

$ terraform validate

13) Create Execution Plan for Your Script

$ terraform plan

14) Create Infrastructure

$ terraform apply

Note: When the script got executed it will store that state in a file. If we execute script again it will not create. If you delete that state file and execute script again then it will create it.

15) Destroy Infrastructure

$ terraform destroy -auto-approve

In first script we kept provider and resources info in single script file. We can keep provider and resources information in separate files

Ex : proder.tf & main.tf

**Script to create multiple Ec2 instances**

***provider.tf***

provider "aws" {

region = "ap-south-1"

access\_key = "AKIA4MGQ5UW757KVKECC"

secret\_key = "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"

}

***main.tf***

resource "aws\_instance" "AWSVM\_Server" {

count = "2"

ami = "ami-05c8ca4485f8b138a"

instance\_type = "t2.micro"

key\_name = "awskp"

security\_groups = ["RaghuSG"]

tags = {

Name = "REDHAT-EC2-VM"

}

}

Note: Once it is created, then destory infrastructure using below command

$ terraform destroy -auto-approve

**Commonly used Terraform commands:**

**1. terraform init:**

- Initializes a Terraform configuration in a directory.

- Downloads provider plugins and sets up the environment.

**2. terraform plan:**

- Generates an execution plan for applying changes to the infrastructure.

- Shows what resources will be created, modified, or destroyed.

**3. terraform apply:**

- Applies the changes described in the configuration.

- Creates, updates, or deletes resources according to the configuration.

**4. terraform destroy:**

- Destroys all resources created by the configuration.

- Prompts for confirmation before taking any action.

**5. terraform validate:**

- Validates the syntax and structure of the configuration files.

- Checks for any errors or warnings in the configuration.

**6. terraform fmt:**

- Automatically formats the configuration files to follow Terraform's standard style.

**7. terraform get:**

- Downloads and installs modules from the module registry.

- Fetches modules specified in the configuration.

**8. terraform workspace:**

- Allows you to manage multiple workspaces (environments) for the same configuration.

- Useful for isolating different deployments or environments.

**9. terraform import:**

- Imports existing infrastructure resources into your Terraform state.

- Helps you manage existing resources using Terraform.

**10. terraform output:**

- Displays the values of output variables defined in the configuration.

- Useful for extracting information about the deployed resources.

**11. terraform state:**

- Provides various commands to manage the Terraform state, which tracks the current state of resources.

- Some subcommands include "list," "show," "mv," "rm," etc.

**12. terraform graph:**

- Generates a visual representation of the resource dependency graph.

- Helps you understand the relationships between resources.

**13. terraform taint:**

- Manually marks a resource as tainted, causing it to be recreated on the next apply.

- Useful for triggering the recreation of a specific resource.

**Terraform Variables:**

It enable you to parameterize your configuration and make it more dynamic and reusable. Variables allow you to pass values into your Terraform configurations without hardcoding them directly in the code.

**There are several types of Terraform variables:**

**1. Input Variables:** These are used to accept values from users or other parts of your codebase. Input variables are defined in your Terraform configuration and can be set when you run Terraform commands.

**variable "instance\_count" {**

**description = "Number of instances to create"**

**type        = number**

**default     = 1**

**}**

**2. Local Variables:** Local variables are used to compute intermediate values based on input variables or other factors within your configuration. They are not exposed to the outside world like input variables.

**locals {**

**instance\_suffix = "-web"**

**instance\_name   = "app${local.instance\_suffix}"**

**}**

**3. Output Variables:** Output variables are used to expose specific values from your infrastructure for reference by other parts of your system or other Terraform configurations. They can be queried after applying the configuration.

**output "instance\_ip" {**

**value = aws\_instance.example.public\_ip**

**}**

**Terraform variables can be set in various ways:**

**- Using Command Line Flags:** You can pass variable values via the command line using the `-var` flag.

*terraform apply -var="instance\_count=3"*

**- Using a Variables File:** You can define variable values in a separate `.tfvars` file and pass it to Terraform using the `-var-file` flag.

*terraform apply -var-file="variables.tfvars"*

**- Interactive Input:** If a variable doesn't have a default value and is not provided through the command line or a `.tfvars` file, Terraform will prompt you for the value during execution.

**- Environment Variables:** You can also set variables using environment variables with the format `TF\_VAR\_name`.

*Terraform will automatically merge and prioritize variable values from different sources, such as variable definitions, `.tfvars` files, command line flags, and environment variables.*

**Create an EC2 instance using variables:**

***project/***

│-main.tf

│-variables.tf

│-terraform.tfvars

**1. variables.tf:** Define your variables in this file.

**variable "aws\_region" {**

**description = "AWS region for resources"**

**type        = string**

**}**

**variable "instance\_type" {**

**description = "EC2 instance type"**

**type        = string**

**default     = "t2.micro"**

**}**

**variable "instance\_count" {**

**description = "Number of EC2 instances"**

**type        = number**

**default     = 1**

**}**

**2. main.tf:** Use the defined variables to create the EC2 instance.

**provider "aws" {**

**region = var.aws\_region**

**}**

**resource "aws\_instance" "example" {**

**count         = var.instance\_count**

**ami           = "ami-0c55b159cbfafe1f0"**

**instance\_type = var.instance\_type**

**}**

**output "instance\_ip" {**

**value = aws\_instance.example.\*.public\_ip**

**}**

**3. terraform.tfvars:** Set the values for your variables in this file.

**aws\_region     = "us-west-2"**

**instance\_type  = "t2.small"**

**instance\_count = 2**

Run the following commands in the `project` directory:

*$ terraform init*

*$ terraform apply*

**Terraform Modules:**

Terraform modules are a way to encapsulate and organize your infrastructure code into reusable components. They allow you to abstract and package resources, configurations, and logic so that you can create consistent and repeatable infrastructure patterns across different projects and environments. Modules promote modularity, code reusability, and maintainability in your Terraform configurations.

**A module in Terraform typically includes:**

**1. Input Variables:** These are parameters that users of the module can customize to configure its behavior.

**2. Output Values:** These are the values that the module exposes to the calling configuration, which can be used by other parts of the codebase.

**3. Resources:** These are the actual infrastructure resources that the module provisions.

**Example:** Wants to create an AWS S3 bucket, and wants to encapsulate this in a module for reuse:

**1. Module Structure:**

***s3\_module/***

│-main.tf

│-variables.tf

│-outputs.tf

**2. s3\_module/main.tf: Define the S3 bucket resource.**

**resource "aws\_s3\_bucket" "bucket" {**

**bucket = var.bucket\_name**

**acl    = "private"**

**}**

**3. s3\_module/variables.tf: Declare input variables for the module.**

**variable "bucket\_name" {**

**description = "Name of the S3 bucket"**

**type        = string**

**}**

**4. s3\_module/outputs.tf: Define output values for the module.**

**output "bucket\_id" {**

**value = aws\_s3\_bucket.bucket.id**

**}**

**5. Using the Module:**

In your main configuration directory, you can use the module like this:

**module "my\_s3\_bucket" {**

**source     = "./s3\_module"**

**bucket\_name = "my-unique-bucket-name"**

**}**

**output "my\_bucket\_id" {**

**value = module.my\_s3\_bucket.bucket\_id**

**}**

**Terraform Project:** Terraform project using modules and variables to create a VPC, EC2 instances, RDS instance, and S3 bucket in AWS.

**Folder System:**

terraform-aws-project/

│ main.tf

│ variables.tf

│ outputs.tf

│ providers.tf

│

└── modules/

├── vpc/

│ ├── main.tf

│ ├── variables.tf

│ └── outputs.tf

│

├── ec2/

│ ├── main.tf

│ ├── variables.tf

│ └── outputs.tf

│

├── rds/

│ ├── main.tf

│ ├── variables.tf

│ └── outputs.tf

│

└── s3/

├── main.tf

├── variables.tf

└── outputs.tf

**GitHub Link for Code:** [**https://github.com/javabyraghu/terraform-aws-project**](https://github.com/javabyraghu/terraform-aws-project)