|  |  |
| --- | --- |
| Terraform  RAJESH MUMMIDI 18/5/2023 | HashiCorp Terraform - Provision & Manage any Infrastructure | Cloud ... |
|  |  |

1. What is Infrastructure as a Code?

A diagram of a cloud

Description automatically generated with low confidence

Infrastructure as Code (IaC) is an approach to managing and provisioning infrastructure resources using machine-readable configuration files or code. It enables the automation and reproducibility of infrastructure deployment and management.

In IaC, infrastructure resources such as servers, networks, storage, and other components are defined and managed through code, typically using declarative configuration files. These files describe the desired state of the infrastructure and are typically version-controlled, allowing for collaboration, change tracking, and rollback.

Popular tools for implementing Infrastructure as Code include Terraform, AWS CloudFormation, Azure Resource Manager, and Ansible.

2) what are the benefits of IaC?

Infrastructure as Code (IaC) offers several benefits that contribute to efficient and reliable infrastructure management. Here are the key advantages of implementing IaC:

1. **Consistency and Standardization**: IaC allows for consistent and standardized infrastructure deployments. Infrastructure resources are defined and provisioned using code, ensuring that the same configuration and settings are applied every time. This eliminates manual errors and reduces configuration drift.
2. **Automation and Efficiency**: IaC enables automation of infrastructure provisioning and configuration. Infrastructure code can be executed programmatically, reducing the need for manual interventions and saving time. Automated deployments also minimize human error and increase overall efficiency.
3. **Reproducibility and Scalability**: With IaC, infrastructure deployments become easily reproducible. The same code can be used to create identical infrastructure environments, whether it's for development, testing, staging, or production. Scaling infrastructure resources up or down can also be achieved through code, simplifying the process.
4. **Version Control and Collaboration**: IaC leverages version control systems, allowing infrastructure code to be managed, tracked, and collaborated on by multiple team members. Version control ensures a history of changes, simplifies collaboration, facilitates code reviews, and enables easy rollbacks if necessary.
5. **Auditing and Compliance**: Infrastructure code provides a clear audit trail, documenting all changes made to the infrastructure over time. This helps with compliance requirements and provides visibility into who made specific changes, when they were made, and why.
6. **Infrastructure Documentation**: Infrastructure code serves as a form of documentation. It provides a clear and concise representation of the infrastructure setup, configurations, and dependencies. This helps in understanding the infrastructure architecture and facilitates troubleshooting and maintenance activities.
7. **Portability and Multi-Cloud Support**: IaC promotes portability by abstracting the infrastructure configuration from the underlying infrastructure provider. This makes it easier to migrate infrastructure across different cloud providers or environments. It also enables multi-cloud deployments, allowing infrastructure to be provisioned on various cloud platforms using the same code.

Overall, Infrastructure as Code brings greater efficiency, reliability, scalability, and collaboration to infrastructure management, enabling organizations to achieve more consistent, automated, and agile deployments.

3) Terraform Advantages and Disadvantages ?

Advantages

1. Terraform internally uses the DAG(direct acyclic graph) technique to get the best results.
2. Terraform supports a variety of cloud options, and switching providers is a breeze.
3. Because the whole infrastructure is managed as code, incremental resource changes are not a problem.
4. Supports scripts that span many regions. For instance, we can search for an ami in us-east-1 and use that information to build an ec2 instance in us-east-2.
5. Effective networking assistance. It might take months to build an on-premise data center, but using Terraform, it can be done in a matter of hours.
6. Integrates easily with the build and deployment processes.
7. Modular architecture.
8. State upkeep. Terraform will reconstruct any objects produced by it if another process removes them.
9. Allows for the import of existing resources to convert them to a Terraform state.

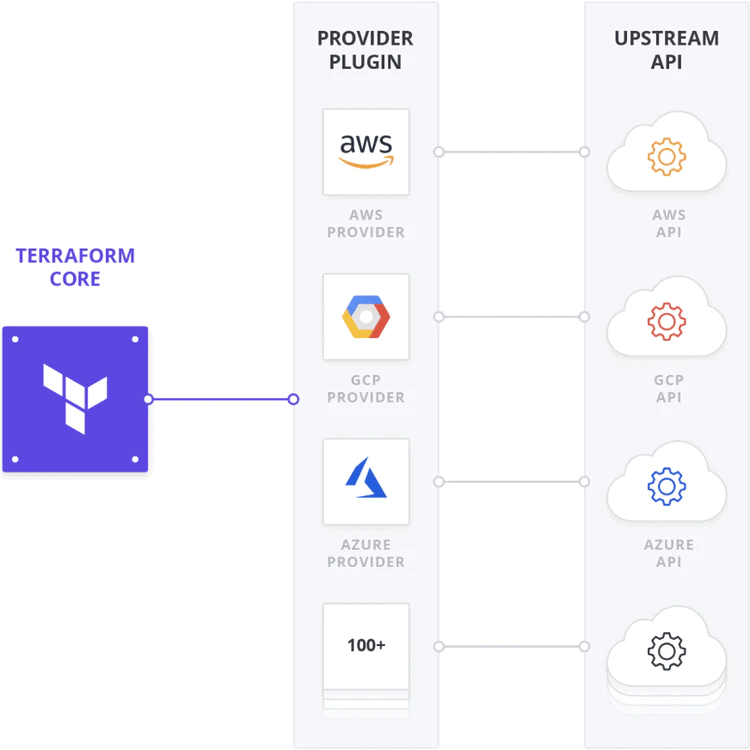
Disadvantages

1. Currently under development. Each month, we release a beta version.
2. The concerns are more connected to Terraform’s (AWS) provider teams. For example, Terraform AWS’s quick sight does not yet support all features.
3. Technology with a narrow application. To write loops or if blocks, intuition is required. Nonetheless, several hacks are accessible online.
4. Specific configurations, such as the terraform backend, are not accessible through var files. Therefore, either give the information in place or construct a backend-config block during Terraform’s initialization.
5. There is no error handling. This implies that we cannot utilize try-catch in the manner we do in other languages.
6. There is no way to roll back. As a result, we must delete everything and re-run if necessary.
7. A few things are prohibited from import.
8. Terraform does not support script generation from the state.
9. Terraform acknowledges that specific versions may include bugs.

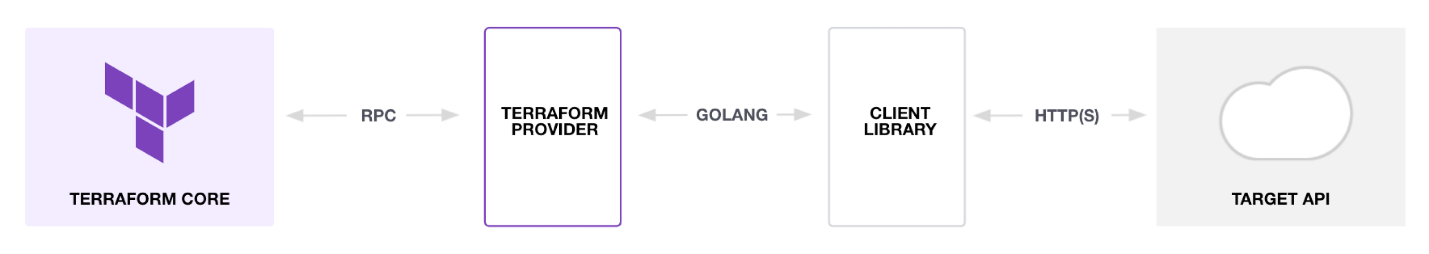
4) What are the providers in Terraform ?

**Providers** are the plugins that Terraform uses to manage those resources. Every supported service or infrastructure platform has a provider that defines which resources are available and performs API calls to manage those resources.

A provider is responsible for understanding API interactions and exposing resources. It interacts with the various APIs required to create, update, and delete various resources. Terraform configurations must declare which providers they require so that Terraform can install and use them.

****

Provider plugins like the [AWS provider](https://registry.terraform.io/providers/hashicorp/aws/latest) or the [cloud-init provider](https://registry.terraform.io/providers/hashicorp/cloudinit/latest/docs) act as a translation layer that allows Terraform to communicate with many different cloud providers, databases, and services.



Terraform uses providers to provision [resources](https://developer.hashicorp.com/terraform/language/resources), which describe one or more infrastructure objects like virtual networks and compute instances. Each provider on the [Terraform Registry](https://registry.terraform.io/) has documentation detailing available resources and their configuration options.

5) what are the regular used Commands used in Terraform ? (Minimum 10 Commands)

1. **terraform init**: Initializes a Terraform working directory, downloads provider plugins, and sets up the backend configuration.
2. **terraform plan**: Generates an execution plan that shows what Terraform will do when applying the configuration. It provides an overview of changes to be made without actually applying them.
3. **terraform apply**: Applies the changes described in the Terraform configuration and provisions or modifies the infrastructure accordingly. It prompts for confirmation before making any changes.
4. **terraform destroy**: Destroys the infrastructure created by Terraform, releasing all allocated resources. It prompts for confirmation before destroying.
5. **terraform validate**: Validates the syntax and configuration of the Terraform files without making any changes. It checks for errors and ensures the configuration is properly structured.
6. **terraform refresh**: Updates the Terraform state file with the current state of the infrastructure. It reconciles the state file with the actual resources in the environment.
7. **terraform state**: Performs various operations on the Terraform state file, such as listing resources, moving resources, or removing resources from the state.
8. **terraform output**: Displays the values of output variables defined in the Terraform configuration. It can be used to retrieve information about the provisioned infrastructure.
9. **terraform import**: Imports existing infrastructure resources into the Terraform state. It allows for managing resources that were not initially created using Terraform.
10. **terraform workspace**: Manages multiple workspaces in Terraform, which allow you to work with different sets of infrastructure configurations or environments.

**Tasks 1& 2) How to configure the providers ? how do configure multiple providers?**

a) Install the Terra form installation in Ubuntu by the following commands shown below :

wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o /usr/share/keyrings

/hashicorp-archive-keyring.gpg

echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg]

https://apt.releases.hashicorp.com $(lsb\_release -cs) main" | sudo tee /etc/apt/ sources.list.d/hashicorp.list

sudo apt update && sudo apt install terraform

b) create a directory of your choice “ “in root directory and then go into the directory and create an new file which need to be saved in main.tf

c) Type the script in main.tf for creating the single provider

A screen shot of a computer program

Description automatically generated with low confidence

e) Type the script in main.tf for creating the multiple provider

A picture containing text, screenshot, font

Description automatically generated

f) Type the command to start the terraform by giving the below command inside the directory we created

🡪 **terraform init**

e) then type the below command to check if of script is correct and ready to create the necessary resources

🡪 **terraform plan**

**A screenshot of a computer screen

Description automatically generated with medium confidence**

By the above steps it is necessary steps available to create the resources.

f) if there is no errors we will be applying the next command to create the resources by typing the command:

🡪 **terraform apply**

g) Then to delete the resources that we have creates we use the command

🡪 **terraform destroy**

**Task 3) Create Ec2 machine using terraform with security source and open security group only 22 should be open ?**

Below is the script for the above task

# Terraform Block

terraform {

required\_providers {

aws = {

source = "hashicorp/aws"

version = "3.0.0"

}

}

}

# Provider Block

provider "aws" {

region = "us-east-1"

#profile = "default"

}

# Website

resource "aws\_instance" "web" {

ami = "ami-0557a15b87f6559cf"

instance\_type = "t2.micro"

associate\_public\_ip\_address = true

vpc\_security\_group\_ids = [aws\_security\_group.pubsgp.id]

tags = {

Name = "king"

}

}

# Pub SGP

resource "aws\_security\_group" "pubsgp" {

name = "allow\_tls"

description = "Allow TLS inbound traffic"

ingress {

description = "SSH"

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

**Task 4) Create a VPC with 2 subnets one is private and another one public create a natgate way setup and associate the same in via terraform ?**

Below is the Yaml Script written for creating a VPC

terraform {

  required\_providers {

    aws = {

      source  = "hashicorp/aws"

      version = "~> 3.0"

    }

  }

}

# Configure the AWS Provider

provider "aws" {

  region = "us-east-1"

}

# Create VPC

resource "aws\_vpc" "vpc1" {

  cidr\_block       = "10.0.0.0/16"

  instance\_tenancy = "default"

  tags = {

    Name = "MY-VPC"

  }

}

# SUBNET PUBLIC

resource "aws\_subnet" "subpub" {

  vpc\_id            = aws\_vpc.vpc1.id

  cidr\_block        = "10.0.1.0/24"

  availability\_zone = "us-east-1a"

  tags = {

    Name = "SUB-PUB"

  }

}

# SUBNET PRIVATE

resource "aws\_subnet" "subpvt" {

  vpc\_id            = aws\_vpc.vpc1.id

  cidr\_block        = "10.0.2.0/24"

  availability\_zone = "us-east-1b"

  tags = {

    Name = "SUB-PVT"

  }

}

# INTERNET GATEWAY

resource "aws\_internet\_gateway" "rama" {

  vpc\_id = aws\_vpc.vpc1.id

  tags = {

    Name = "RAMA"

  }

}

#RT PUB

resource "aws\_route\_table" "rtpub" {

  vpc\_id = aws\_vpc.vpc1.id

  route {

    cidr\_block = "0.0.0.0/0"

    gateway\_id = aws\_internet\_gateway.rama.id

  }

  tags = {

    Name = "RT-PUB"

  }

}

# RTSUB-PUB - ASSOCIATION

resource "aws\_route\_table\_association" "rtassopub" {

  subnet\_id      = aws\_subnet.subpub.id

  route\_table\_id = aws\_route\_table.rtpub.id

}

#RT PVT

resource "aws\_route\_table" "rtpvt" {

  vpc\_id = aws\_vpc.vpc1.id

  tags = {

    Name = "RT-PVT"

  }

}

# RTSUB-PVT - ASSOCIATION

resource "aws\_route\_table\_association" "rtassopvt" {

  subnet\_id      = aws\_subnet.subpvt.id

  route\_table\_id = aws\_route\_table.rtpvt.id

}

# Create Security group

resource "aws\_security\_group" "mysgpub" {

  name        = "allow\_tls"

  description = "Allow TLS inbound traffic"

  vpc\_id      = aws\_vpc.vpc1.id

  ingress {

    description      = "http"

    from\_port        = 80

    to\_port          = 80

    protocol         = "tcp"

    cidr\_blocks      = ["0.0.0.0/0"]

  }

  ingress {

    description      = "ssh"

    from\_port        = 22

    to\_port          = 22

    protocol         = "tcp"

    cidr\_blocks      = ["0.0.0.0/0"]

  }

  egress {

    from\_port        = 0

    to\_port          = 0

    protocol         = "-1"

    cidr\_blocks      = ["0.0.0.0/0"]

    ipv6\_cidr\_blocks = ["::/0"]

  }

  tags = {

    Name = "allow\_tls"

  }

}

#EIP

resource "aws\_eip" "myeip" {

  vpc = true

}

# NAT GATEWAY

resource "aws\_nat\_gateway" "tnat" {

  allocation\_id = aws\_eip.myeip.id

  subnet\_id     = aws\_subnet.subpub.id

  tags = {

    Name = "gw NAT"

  }

}

# EC2-PUBLIC- Creation

resource "aws\_instance" "web" {

  ami                          = "ami-007855ac798b5175e"

  instance\_type                = "t2.micro"

  associate\_public\_ip\_address  = true

  vpc\_security\_group\_ids       = [aws\_security\_group.mysgpub.id]

  subnet\_id                    = aws\_subnet.subpub.id

  tags = {

    Name = "king"

  }

}

# EC2-PRIVATE-Creation

resource "aws\_instance" "DB" {

  ami                          = "ami-06a0cd9728546d178"

  instance\_type                = "t2.micro"

  vpc\_security\_group\_ids       = [aws\_security\_group.mysgpub.id]

  subnet\_id                    = aws\_subnet.subpvt.id

  tags = {

    Name = "DATABASE"

  }

}

**Task 5) Create S3 bucket with version enabling bucket should be public ?**

Below is the Yaml Script written for creating a s3 bucket with version enabling

# Terraform Block

terraform {

required\_providers {

aws = {

source = "hashicorp/aws"

version = "~> 3.0"

}

}

}

# Provider Block

provider "aws" {

region = "us-east-1"

#profile = "default"

}

# Create S3 bucket

resource "aws\_s3\_bucket" "buckzzzzzzzzzz" {

bucket = "mybuckzzzzzzzzzz"

versioning {

enabled = true

}

tags = {

Name = "My buckzz"

#Environment = "Dev"

}

}

**Task 6) Create a EC2 machine using Variable files?**

Create three different files in the directory where u are going to execute and the files are mentioned below in the name of **provider.tf** , **var.tf** and **main.tf** along with the yaml script that is meant to execute by using terraform

**provider.tf**

# Terraform Block

terraform {

  required\_providers {

    aws = {

      source  = "hashicorp/aws"

      version = "~> 3.0"

    }

  }

}

# Provider Block

provider "aws" {

  region  = "us-east-1"

  profile = "default"

}

**var.tf**

# Input Variables

variable "aws\_regions\_mumbai" {

  description = "Region in which AWS resources to be created"

  type        = string

  default     = "us-east-1"

}

variable "ec2\_ami" {

  description = "AMI ID"

  type        = string

  default     = "ami-007855ac798b5175e" # Amazon2 Linux AMI ID

}

variable "ec2\_instance\_type" {

  description = "EC2 Instance Type"

  type        = string

  default     = "t2.micro"

}

variable "ec2\_instance\_count" {

  description = "EC2 Instance Count"

  type        = number

  default     = 1

}

**main.tf**

# Create EC2 instance

resource "aws\_instance" "my-ec2-vm" {

 ami                     = var.ec2\_ami

 instance\_type           = var.ec2\_instance\_type

 count                   = var.ec2\_instance\_count

 tags = {

  "Name"  = "myec2vm"

 }

}

Use all the files and execute the terraform by using **terraform plan** 🡪**terraform apply**

And after successful creation be sure sure to destroy the script 🡪 **terraform destroy**

Top of Form

Bottom of Form