Terraform

IaC – Infrastructure as Code

Create S3 bucket on aws

Api-> make a call

VPC + VPC config+EC2+S3

(CFT) Cloud formation template -> in json and yaml

AWS -> cloud formation template

Azure -> Resource manager

Openstack -> Heat template

Automate infrastructure

Terraform -> API as code ->

Install terraform -> [Install Terraform | Terraform | HashiCorp Developer](https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli)

Like virtual machine -> code space -> [GitHub - iam-veeramalla/terraform-zero-to-hero: Master Terraform in 7 days using this Zero to Hero course.](https://github.com/iam-veeramalla/terraform-zero-to-hero)

Click on code -> codespaces -> Add(+button) -> Not Successful

Cmd

terraform –version

aws cli install and check

if necessary create an Iam role

aws configure

enter access key

secret access key

region us-east-1

aws s3 ls

main.tf

provider "aws" {

region = "us-east-1" # Set your desired AWS region

}

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0" # Specify an appropriate AMI ID

instance\_type = "t2.micro"

}

Terraform template : [aws\_instance | Resources | hashicorp/aws | Terraform | Terraform Registry](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance)

terraform plan -> dry run

terraform apply

In visual studio -> Install plugin hashicorp terraform -> HashiCorp HCL

Terraform destroy

Providers -> EC2 (AWS)

Main.tf -> Provider”aws”{

Region = “-”}

Resource “AWS” ={}

Terraform in 1 video

1. **Provider**: A provider is a plugin for Terraform that defines and manages resources for a specific cloud or infrastructure platform. Examples of providers include AWS, Azure, Google Cloud, and many others. You configure providers in your Terraform code to interact with the desired infrastructure platform.
2. **Resource**: A resource is a specific infrastructure component that you want to create and manage using Terraform. Resources can include virtual machines, databases, storage buckets, network components, and more. Each resource has a type and configuration parameters that you define in your Terraform code.
3. **Module**: A module is a reusable and encapsulated unit of Terraform code. Modules allow you to package infrastructure configurations, making it easier to maintain, share, and reuse them across different parts of your infrastructure. Modules can be your own creations or come from the Terraform Registry, which hosts community-contributed modules.
4. **Configuration File**: Terraform uses configuration files (often with a .tf extension) to define the desired infrastructure state. These files specify providers, resources, variables, and other settings. The primary configuration file is usually named main.tf, but you can use multiple configuration files as well.
5. **Variable**: Variables in Terraform are placeholders for values that can be passed into your configurations. They make your code more flexible and reusable by allowing you to define values outside of your code and pass them in when you apply the Terraform configuration.
6. **Output**: Outputs are values generated by Terraform after the infrastructure has been created or updated. Outputs are typically used to display information or provide values to other parts of your infrastructure stack.
7. **State File**: Terraform maintains a state file (often named terraform.tfstate) that keeps track of the current state of your infrastructure. This file is crucial for Terraform to understand what resources have been created and what changes need to be made during updates.
8. **Plan**: A Terraform plan is a preview of changes that Terraform will make to your infrastructure. When you run terraform plan, Terraform analyzes your configuration and current state, then generates a plan detailing what actions it will take during the apply step.
9. **Apply**: The terraform apply command is used to execute the changes specified in the plan. It creates, updates, or destroys resources based on the Terraform configuration.
10. **Workspace**: Workspaces in Terraform are a way to manage multiple environments (e.g., development, staging, production) with separate configurations and state files. Workspaces help keep infrastructure configurations isolated and organized.
11. **Remote Backend**: A remote backend is a storage location for your Terraform state files that is not stored locally. Popular choices for remote backends include Amazon S3, Azure Blob Storage, or HashiCorp Terraform Cloud. Remote backends enhance collaboration and provide better security and reliability for your state files.

**Setup Terraform for AWS**

1. **Install AWS CLI (Command Line Interface)**:
2. **Create an AWS IAM User**:
3. **Configure AWS CLI Credentials**:
4. **Multiple Providers**

You can use multiple providers in one single terraform project. For example,

1. Create a providers.tf file in the root directory of your Terraform project.
2. In the providers.tf file, define the AWS and Azure providers. For example:

provider "aws" {

region = "us-east-1"

}

provider "azurerm" {

subscription\_id = "your-azure-subscription-id"

client\_id = "your-azure-client-id"

client\_secret = "your-azure-client-secret"

tenant\_id = "your-azure-tenant-id"

}

**Modules**

1. **Modularity**: Terraform modules allow you to break down your infrastructure configuration into smaller, self-contained components. This modularity makes it easier to manage and reason about your infrastructure because each module handles a specific piece of functionality, such as an EC2 instance, a database, or a network configuration.

**Terraform destroy**

Terraform plan

Terraform Registry -> like docker hub registry

[Terraform Registry](https://registry.terraform.io/) -> ec2 instance -> module -> examples -> complete

**Terraform State File**

Terraform is an Infrastructure as Code (IaC) tool used to define and provision infrastructure resources. The Terraform state file is a crucial component of Terraform that helps it keep track of the resources it manages and their current state. This file, often named terraform.tfstate, is a JSON or HCL (HashiCorp Configuration Language) formatted file that contains important information about the infrastructure's current state, such as resource attributes, dependencies, and metadata.

**Advantages of Terraform State File:**

**Resource Tracking:** The state file keeps track of all the resources managed by Terraform, including their attributes and dependencies. This ensures that Terraform can accurately update or destroy resources when necessary.

**Concurrency Control:** Terraform uses the state file to lock resources, preventing multiple users or processes from modifying the same resource simultaneously. This helps avoid conflicts and ensures data consistency.

**Plan Calculation:** Terraform uses the state file to calculate and display the difference between the desired configuration (defined in your Terraform code) and the current infrastructure state. This helps you understand what changes Terraform will make before applying them.

**Resource Metadata:** The state file stores metadata about each resource, such as unique identifiers, which is crucial for managing resources and understanding their relationships.

Statefile -> record, Store

Ec2 -> t2.micro

When you type terraform apply or terraform plan ->

Statefile is heart of terraform -> without state file you will not know to update infrastructure rather than creating it.

Terraform destroy -> will destroy infrastructure

Disadvantage/ drawback ->

1. Github access is compromised statefile is compromised.
2. Everytime change in code updated there should be change in statefile.

In vcs(version control system) -> you have to upload statefile

Lets say a developer takes access of the statefile . Puts a tag on it ec2 and this is pushed back to Github (tag). If the statefile is not pushed to Github. Terraform will not know the logic of tag creation.

If there is difference in terraform file it will tell to delete files.

Note: Everytime a change is made to the code. statefile gets updated. This statefile should be uploaded to terraform after applying.

Remote backend

If you don’t want to store statefile in git you can store it in S3 bucket.

In this statefile won’t get created on Laptop or VM. It will get creted in S3 bucket. This will remove the 2 drawback of terraform.

In this condition when you run terraform apply automatically the state file is updated.

If it gets updated in local. It gets updated in Ec2 instance.

When you run terraform init. It will understand that we are using the logic of remote backend. It will compare between the S3 bucket and the Github repository.

S3 -> remote backend

You can host your terraform remote backend In the terraform cloud.

Azure -> Azure storage

Devops team -> github repository with terraform code. To create EKS or VPC 3 tier architecture -> using remote backend S3 -> Any one will clone on laptop -> when you do terraform apply will apply the logic to S3 bucket -> SO now statefile is updated in S3 bucket

Github

Devops



EC2

Ansible

Ansible is an open source IT automation engine that automates

* provisioning
* configuration management
* application deployment
* orchestration

and many other IT processes. It is free to use, and the project benefits from the experience and intelligence of its thousands of contributors.

**How Ansible works**

Ansible is agentless in nature, which means you don't need install any software on the manage nodes.

For automating Linux and Windows, Ansible connects to managed nodes and pushes out small programs—called Ansible modules—to them. These programs are written to be resource models of the desired state of the system. Ansible then executes these modules (over SSH by default), and removes them when finished. These modules are designed to be idempotent when possible, so that they only make changes to a system when necessary.

For automating network devices and other IT appliances where modules cannot be executed, Ansible runs on the control node. Since Ansible is agentless, it can still communicate with devices without requiring an application or service to be installed on the managed node.

Ansible uses python internally -> yaml file written gets converted to python ->

System

Admins

Configuration



Management

Linux Windows

1. Up to date (Supported)
2. OpenSSh/wget/curlshared -> check packages version -> up to date
3. Java
4. Maintenance

Things to check if are up to date

1. OS
2. System
3. Application
4. Servers

Similar tools -> Puppet, Chef, Salt, Ansible

Puppet, chef, salt are written in ruby

They just need to write one programming that is puppet or chef. It will automate all the things -> Example : Linux, Alpine, Windows, Debian Gnux etc

* Drawback -> have to learn ruby -> complesx way of writing code.
* Have to install software like agent on each device or VM
* New machines created have to be connected to the master.

We write Chef books or cookbooks ->

Ansible playbook

Need to write program in yaml

It is agentless so there is no need to install any software on the target ->

Control Node -> On control nodes Ansible is installed -> Generally VMs are used with this virtual machine you can control configuration of the rest of the machines ->

Manage nodes -> Machines whose configuration is getting managed by the control node

Ansible ->

Provisioning ->

Config Managemnent

Deploy

Network Automation -> CISCO/FSC/Zscaler

Kafka -> Data is managed on server-> data is routed for every second

Primary to secondary data transfer -> every second data is updated ->

Example : ola ,Zomato

Kafka can send many messages -> high through out

Kafka provides flexibility -> open source tool

Different microservices -> if you make changes in one place ->

Changes made in one module will be reflected in all module.

Triilion message can be handled by kafka.