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)

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**:

**Multiple Providers**

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"

}

Three types of providers

Official -> maintained by hashicorp actively

Partner -> maintained by that particular provider

Community -> anyone can create the entire provider configuration

Multiple region -> Code changes

provider ”aws” {

alias =”us-east-1”

region =”us-east-1”

}

provider ”aws” {

alias =”us-west-2”

region =”us-west-2”

}

resource ”aws\_instance” ”example” {

ami = “ami-0123”

instance\_type=”t2.micro”

provider=”aws.us-east-1”

}

resource ”aws\_instance” ”example2” {

ami = “ami-0123”

instance\_type=”t2.micro”

provider=”aws.us-west-1”

}

Variable are used to parameterize okay so they can be used as params

* + Input variable
  + Output variables (public.ip)

# Define an output variable to expose the public IP Address of the EC2 instance

output ”public\_ip” {

description =” Public IP address of the EC2 instance”

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

}

Everything can’t be written in main.tf file so split

* + Provider.tf
  + Input.tf
  + Output.tf
  + Terraform.tfvars

Inside terraform.tfvars

ami\_value=”ami-23534”

instance\_type\_value=”t2.micro”

subnet\_id\_value = “”

terraform init

main.tf file

provider “aws” {

Variables

Input and output variables in Terraform are essential for parameterizing and sharing values within your Terraform configurations and modules. They allow you to make your configurations more dynamic, reusable, and flexible.

**Input Variables**

Input variables are used to parameterize your Terraform configurations. They allow you to pass values into your modules or configurations from the outside. Input variables can be defined within a module or at the root level of your configuration. Here's how you define an input variable:

variable "example\_var" {

description = "An example input variable"

type = string

default = "default\_value"

}

In this example:

* variable is used to declare an input variable named example\_var.
* description provides a human-readable description of the variable.
* type specifies the data type of the variable (e.g., string, number, list, map, etc.).
* default provides a default value for the variable, which is optional.
* resource "example\_resource" "example" {
* name = var.example\_var
* # other resource configurations
* }

resource "example\_resource" "example" {

name = var.example\_var

# other resource configurations

}

Output Variables

Output variables allow you to expose values from your module or configuration, making them available for use in other parts of your Terraform setup. Here's how you define an output variable:

output "example\_output" {

description = "An example output variable"

value = resource.example\_resource.example.id

}

In this example:

* output is used to declare an output variable named example\_output.
* description provides a description of the output variable.
* value specifies the value that you want to expose as an output variable. This value can be a resource attribute, a computed value, or any other expression.

You can reference output variables in the root module or in other modules by using the syntax module.module\_name.output\_name, where module\_name is the name of the module containing the output variable.

For example, if you have an output variable named example\_output in a module called example\_module, you can access it in the root module like this:

output "root\_output" {

value = module.example\_module.example\_output

}

This allows you to share data and values between different parts of your Terraform configuration and create more modular and maintainable infrastructure-as-code setups.

**Terraform tfvars**

In Terraform, .tfvars files (typically with a .tfvars extension) are used to set specific values for input variables defined in your Terraform configuration.

They allow you to separate configuration values from your Terraform code, making it easier to manage different configurations for different environments (e.g., development, staging, production) or to store sensitive information without exposing it in your code.

Here's the purpose of .tfvars files:

1. **Separation of Configuration from Code**: Input variables in Terraform are meant to be configurable so that you can use the same code with different sets of values. Instead of hardcoding these values directly into your .tf files, you use .tfvars files to keep the configuration separate. This makes it easier to maintain and manage configurations for different environments.
2. **Sensitive Information**: .tfvars files are a common place to store sensitive information like API keys, access credentials, or secrets. These sensitive values can be kept outside the version control system, enhancing security and preventing accidental exposure of secrets in your codebase.
3. **Reusability**: By keeping configuration values in separate .tfvars files, you can reuse the same Terraform code with different sets of variables. This is useful for creating infrastructure for different projects or environments using a single set of Terraform modules.

**Conditional Expressions**

Conditional expressions in Terraform are used to define conditional logic within your configurations. They allow you to make decisions or set values based on conditions. Conditional expressions are typically used to control whether resources are created or configured based on the evaluation of a condition.

The syntax for a conditional expression in Terraform is:

condition ? true\_val : false\_val

* condition is an expression that evaluates to either true or false.
* true\_val is the value that is returned if the condition is true.
* false\_val is the value that is returned if the condition is false.
* **Conditional Resource Creation Example**

resource "aws\_instance" "example" {

count = var.create\_instance ? 1 : 0

ami = "ami-XXXXXXXXXXXXXXXXX"

instance\_type = "t2.micro"

}

**How does Terraform work?**

Terraform uses plugins called the Terraform providers to interact with APIs on Cloud Platforms and provision our resources. As an end-user, terraform workflow has three steps.

**Write**: Author the infrastructure as code.

**Plan**: Preview changes Terraform will make before applying.

**Apply**: Provision the infrastructure and apply the changes.