### 

[Installing Terraform](#_jia3mb9i2efv)

[What is Terraform?](#_a1ob1jfelqhq)

[How do I start using Terraform?](#_9zc9cwbolsx9)

[Where Are Configs Stored?](#_8zbuz9p5ermy)

[Terraform Blocks Broken Down](#_7vdmbi98wwkw)

[Terraform](#_q8s53n8dn38h)

[Provider](#_5c64jh7ld812)

[Resource](#_nlfbmgtobqfv)

[Data](#_7tzi9lo1m8hm)

[Output](#_b23dt3zbotpx)

[Module](#_taacphnxk2qq)

[Variable](#_dicndvgdpvoj)

[Locals](#_de8zd6kzldjw)

[Sample codes](#_bojz0ndzm7xs)

[Create S3 buckets](#_lg97n9otcqwq)

[Create 2 EC2 instances](#_y3okjgj3ib6u)

[Create AWS DynamoDB table](#_tmsnbmr5fkax)

### 

## Installing Terraform

<https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli>

## What is Terraform?

Terraform is an open-source infrastructure as code (IaC) tool developed by HashiCorp. It allows you to define, provision, and manage infrastructure resources such as virtual machines, networks, storage, and other cloud-based services in a declarative way. Terraform uses a configuration language called HashiCorp Configuration Language (HCL) or JSON to describe the desired state of your infrastructure.

With Terraform, you can define your infrastructure as code, version it, and apply changes in a consistent and repeatable way. Terraform supports multiple cloud platforms such as Amazon Web Services, Microsoft Azure, Google Cloud Platform, and many others, making it possible to manage infrastructure across multiple clouds and on-premises environments.

Look at this link to see what kind of online resources you can build with Terraform:

<https://registry.terraform.io/browse/providers>

## How do I start using Terraform?

<https://developer.hashicorp.com/terraform/tutorials/aws-get-started>

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs>

We need to run terraform in the following steps:

1. **Initialize** - We need to initialize our Terraform environment by running the terraform init command. This allows you to download the necessary providers and modules that you have specified in your configuration.
2. **Plan** - After initializing the Terraform environment, we can create an execution plan with the terraform plan command. This plan shows us what changes Terraform will make to our infrastructure.
3. **Apply** - After we have reviewed the execution plan from the previous step and are satisfied with the proposed changes, we can apply the changes proposed by running the terraform apply command. Terraform will make the necessary changes to our infrastructure based on the plan.
4. **Destroy** - When we don’t need our cloud resources anymore, we can use the terraform destroy command to tear down all the resources created by Terraform.

## Where Are Configs Stored?

Terraform state files store information about the infrastructure resources you manage using Terraform. They keep track of the actual resources created in your cloud provider environment, as well as the mapping between your configuration and those resources. This state is used to plan and manage updates to your infrastructure.

This file is normally named **\*.tfstate** e.g. **terraform.tfstate**

By default, Terraform stores state files locally in the same directory as your configuration files. However, this approach has limitations, especially in team environments or when working across multiple machines. To address these limitations, it's recommended to use remote state backends, which are external storage solutions for your Terraform state.

Here are some common remote state backends, and they can be particularly useful when working with Terraform in production or team settings:

* Amazon S3 and DynamoDB: AWS offers a remote state backend using Amazon S3 for storing the state file and Amazon DynamoDB for locking to prevent concurrent modifications. This is a popular choice when working with AWS resources. You can configure Terraform to use these services to store and manage your state.
* Azure Storage: If you're using Microsoft Azure, you can use Azure Storage as a remote state backend.
* Google Cloud Storage: For Google Cloud users, Google Cloud Storage can serve as a remote state backend.
* HashiCorp Consul: Consul is a service discovery and key-value store tool provided by HashiCorp. It can also be used as a remote state backend.
* Terraform Cloud: HashiCorp offers Terraform Cloud, a fully managed service that includes features like remote state storage, collaboration tools, and more. This is a convenient option for teams.
* Version Control Systems: While not a traditional backend, some teams opt to store their state files in version control systems (like Git) to maintain a history of changes. However, this approach might not offer the same level of state locking and management features as dedicated remote state backends.

To configure a remote state backend in your Terraform configuration, you need to modify your backend block in the configuration files. This configuration specifies the backend type (like "s3" for Amazon S3) and the necessary connection information.

Using remote state backends helps with better collaboration, state locking, and handling infrastructure as code at scale. It also provides a level of security by preventing sensitive infrastructure details from being stored in plain text on local systems.

Link: <https://developer.hashicorp.com/terraform/language/settings/backends/configuration>

## 

## Terraform Blocks Broken Down

### Terraform

The “terraform” block is used to specify settings for the Terraform execution environment, such as the required Terraform version and any backend configuration settings.

Example:

| terraform {  required\_providers {  aws = {  source = "hashicorp/aws"  version = "~> 4.0"  }  } } |
| --- |

### 

### Provider

A provider block specifies the details of the provider being used. The provider is responsible for creating and managing resources in a specific infrastructure. The provider block is required for every Terraform configuration file.

Example:

| provider "aws" {  region = "us-east-1"  # Edit below  access\_key = "my-access-key"  secret\_key = "my-secret-key" } |
| --- |

### Resource

A resource block specifies a single resource to be managed by Terraform. It includes the resource type, name, and its configuration options.

Example:

| resource "aws\_instance" "example" {  ami = "ami-0c55b159cbfafe1f0"  instance\_type = "t2.micro" } |
| --- |

### Data

A data block defines data sources that can be queried from an external system, such as a cloud provider or a database.

Example:

| data "aws\_ami" "ubuntu" {  most\_recent = true  filter {  name = "name"  values = ["ubuntu/images/\*ubuntu-xenial-16.04-amd64-server-\*"]  } } |
| --- |

### Output

An output block defines the values that Terraform should output after applying a configuration. Outputs are useful for retrieving information from Terraform to use in other parts of your infrastructure.

Example:

| output "public\_ip" {  value = aws\_instance.example.public\_ip } |
| --- |

### Module

A module block specifies a reusable set of resources and configurations. Modules can be used to organize and reuse code across multiple Terraform configurations.

Example:

| module "vpc" {  source = "terraform-aws-modules/vpc/aws"  version = "3.0.0"  name = "my-vpc"  cidr = "10.0.0.0/16"  azs = ["us-west-2a", "us-west-2b", "us-west-2c"] } |
| --- |

Link: <https://registry.terraform.io/browse/modules>

### Variable

A variable block defines variables that can be used in a Terraform configuration. Variables are used to provide values that may change depending on the environment.

Example:

| variable "aws\_region" {  type = string  default = "us-west-2" } |
| --- |

### Locals

The locals block defines local values that can be used within a Terraform module or configuration file. These values are computed once during Terraform execution and can be used to simplify complex expressions or provide more descriptive names for values.

Example:

| locals {  instance\_count = length(var.instance\_types)  instance\_names = [ "web-${count.index}" for count.index in range(local.instance\_count) ] } |
| --- |

## Sample codes

### Create S3 buckets

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/s3_bucket>

| terraform {  required\_providers {  aws = {  source = "hashicorp/aws"  version = "~> 4.0"  }  } }  # Configure the AWS Provider provider "aws" {  region = "us-east-1"  # Edit below  access\_key = "my-access-key"  secret\_key = "my-secret-key" }  # Create AWS S3 resource "aws\_s3\_bucket" "bucket" {  # Change to a unique name  bucket = "my-tf-test-bucket"   tags = {  Name = "<Name> Bucket"  Environment = "Dev"  } } |
| --- |

### Create 2 EC2 instances

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance>

| provider "aws" {  region = "us-east-1"  # Edit below  access\_key = "my-access-key"  secret\_key = "my-secret-key" }  resource "aws\_instance" "instances" {  ami = "ami-0fc61db8544a617ed" # Amazon Linux 2 LTS  instance\_type = "t2.micro"  count = 2   tags = {  Name = "<Name> EC2 Instance ${count.index + 1}"  } } |
| --- |

### Create AWS DynamoDB table

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/dynamodb_table>

| provider "aws" {  region = "us-east-1"  # Edit below  access\_key = "my-access-key"  secret\_key = "my-secret-key" }  resource "aws\_dynamodb\_table" "personal\_table" {  name = "<NAME>\_table"  billing\_mode = "PAY\_PER\_REQUEST"  hash\_key = "id"  range\_key = "name"  attribute {  name = "id"  type = "S"  }  attribute {  name = "name"  type = "S"  } } |
| --- |