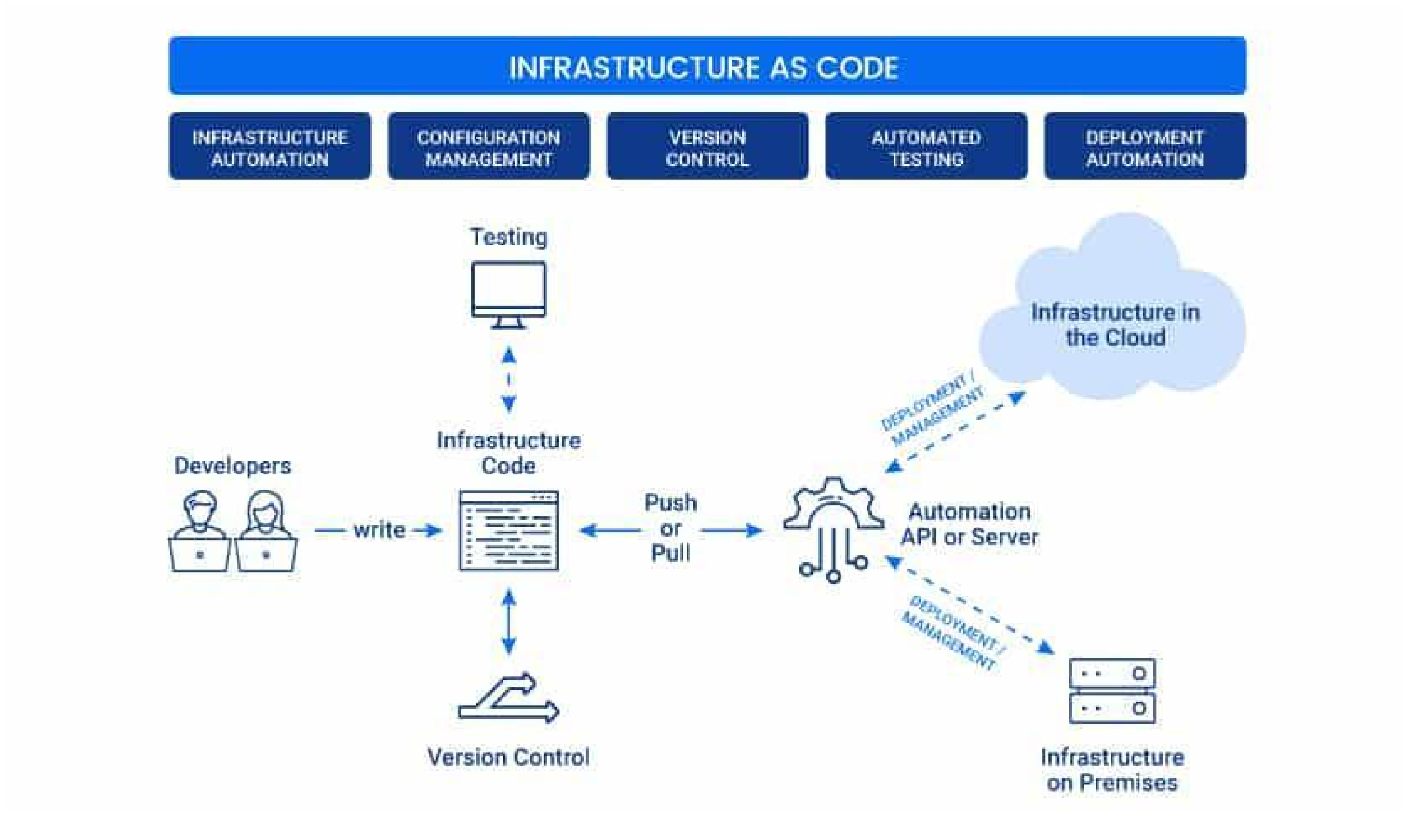
What Is Infrastructure as Code (IaC)?

Infrastructure as Code (IaC) is a widespread terminology among DevOps professionals and a key DevOps practice in the industry. It is the process of managing and provisioning the complete IT infrastructure (comprises both physical and virtual machines) using machine-readable definition files. It helps in automating the complete data center by using programming scripts.



Popular IaC Tools:

1. Terraform An open-source declarative tool that offers pre-written modules to build and manage an infrastructure.
2. Chef: A configuration management tool that uses cookbooks and recipes to deploy the desired environment. Best used for Deploying and configuring applications using a pull-based approach.
3. Puppet: Popular tool for configuration management that follows a Client-Server Model. Puppet needs agents to be deployed on the target machines before the puppet can start managing them.
4. Ansible: Ansible is used for building infrastructure as well as deploying and configuring applications on top of them. Best used for Ad hoc analysis.
5. Packer: Unique tool that generates VM images (not running VMs) based on steps you provide. Best used for Baking compute images.
6. Vagrant: Builds VMs using a workflow. Best used for Creating pre-configured developer VMs within VirtualBox.

Read our blog to know why Terraform is preferred over other IaC tools **Terraform vs Ansible**

What Is Terraform?

Terraform is one of the most popular **Infrastructure-as-code (IaC) tool**, used by DevOps teams to automate infrastructure tasks. It is used to automate the provisioning of your cloud resources. Terraform is an open-source, cloud-agnostic provisioning tool developed by HashiCorp and written in GO language.

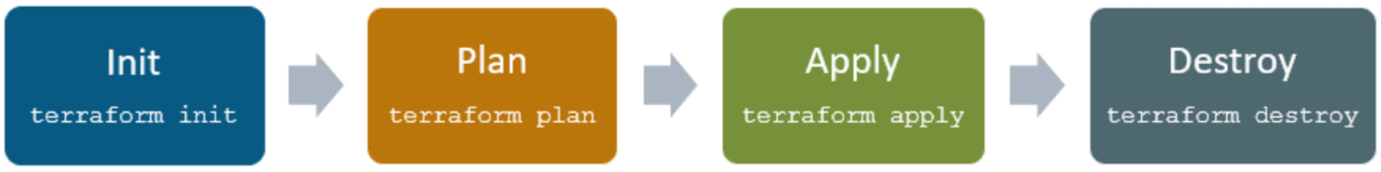
Benefits of using Terraform:

- Does orchestration, not just configuration management
- Supports multiple providers such as AWS, Azure, Oracle, GCP, and many more
- Provide immutable infrastructure where configuration changes smoothly
- Uses easy to understand language, HCL (HashiCorp configuration language)
- Easily portable to any other provider

Check out our blog for everything you need to know about Terraform Certification **Terraform Certification**

Terraform Lifecycle

Terraform lifecycle consists of – **init**, **plan**, **apply**, and **destroy**.



1. **Terraform init** initializes the (local) Terraform environment. Usually executed only once per session.
2. **Terraform plan** compares the Terraform state with the as-is state in the cloud, build and display an
execution plan. This does not change the deployment (read-only).
3. **Terraform apply** executes the plan. This potentially changes the deployment.
4. **Terraform destroy** deletes all resources that are governed by this specific terraform environment.

Terraform Core Concepts

1. Variables: Terraform has input and output variables, it is a key-value pair. Input variables are used as parameters to input values at run time to customize our deployments. Output variables are return values of a terraform module that can be used by other configurations.
Read our blog on **Terraform Variables**

2. Provider: Terraform users provision their infrastructure on the major cloud providers such as AWS, Azure, OCI, and others. A *provider* is a plugin that interacts with the various APIs required to create, update, and delete various resources.
Read our blog to know more about **Terraform Providers**

3. Module: Any set of Terraform configuration files in a folder is a *module*. Every Terraform configuration has at least one module, known as its ***root module.***

4. State: Terraform records information about what infrastructure is created in a Terraform *state* file. With the state file, Terraform is able to find the resources it created previously, supposed to manage and update them accordingly.

5. Resources: Cloud Providers provides various services in their offerings, they are referenced as Resources in Terraform. Terraform resources can be anything from compute instances, virtual networks to higher-level components such as DNS records. Each resource has its own attributes to define that resource.

6. Data Source: Data source performs a read-only operation. It allows data to be fetched or computed from resources/entities that are not defined or managed by Terraform or the current Terraform configuration.

7. Plan: It is one of the stages in the Terraform lifecycle where it determines what needs to be created, updated, or destroyed to move from the real/current state of the infrastructure to the desired state.

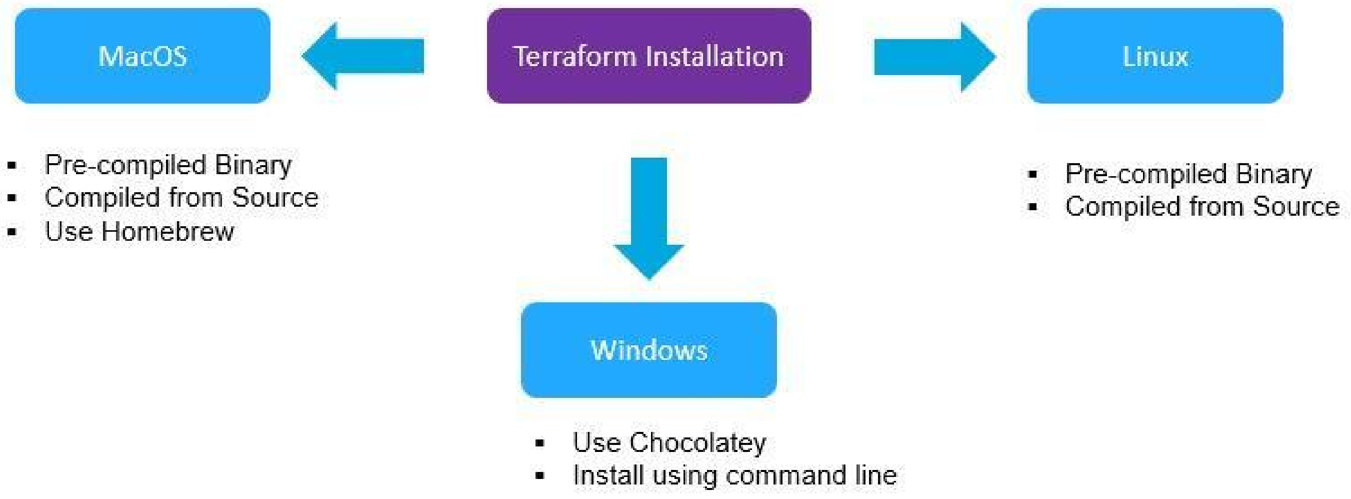
8. Apply: It is one of the stages in the Terraform lifecycle where it applies the changes real/current state of the infrastructure in order to achieve the desired state.

Check Out: Our previous blog post on **Terraform Cheat Sheet**.

Terraform Installation

Before you start working, make sure you have Terraform installed on your machine, it can be installed on any OS, say Windows, macOS, Linux, or others. Terraform installation is an easy process and can be done in a few minutes.

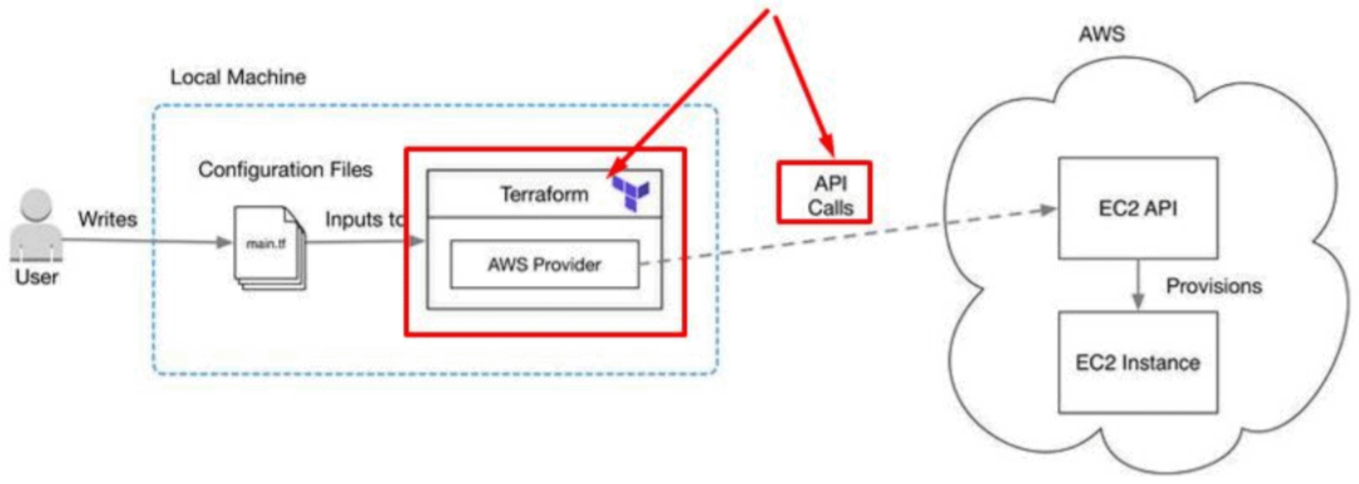
Read our blog to know how to install Terraform in Linux, Mac, Windows



We cover the step-by-step Terraform installation in all these ways in our Terraform training. Check out our blog for all the Hands-on Labs that we cover in our training HashiCorp Certified Terraform Associate-Step By Step Activity Guides.

Terraform Providers

A provider is responsible for understanding API interactions and exposing resources. It is an executable plug-in that contains code necessary to interact with the API of the service. Terraform configurations must declare which providers they require so that Terraform can install and use them.

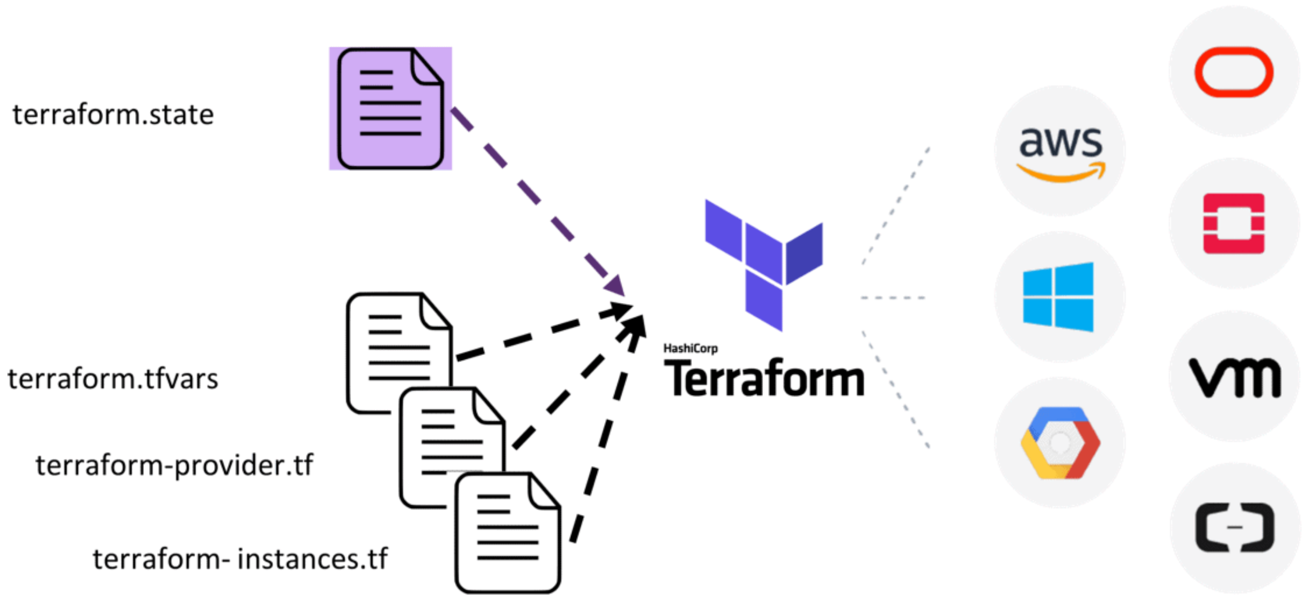


Terraform has over a hundred providers for different technologies, and each provider then gives terraform user access to its resources. So through AWS provider, for example, you have access to hundreds of AWS resources like EC2 instances, the AWS users, etc.

Read More: About **Terraform Workflow**.

Terraform Configuration Files

Configuration files are a set of files used to describe infrastructure in Terraform and have the file extensions **.tf** and **.tf.json**. Terraform uses a declarative model for defining infrastructure. Configuration files let you write a configuration that declares your desired state. Configuration files are made up of resources with settings and values representing the desired state of your infrastructure.



A Terraform configuration is made up of one or more files in a directory, provider binaries, plan files, and state files once Terraform has run the configuration.

1. Configuration file (*.tf files): Here we declare the provider and resources to be deployed along with the type of resource and all resources specific settings

2. Variable declaration file (variables.tf or variables.tf.json): Here we declare the input variables required to provision resources

3. Variable definition files (terraform.tfvars): Here we assign values to the input variables

4. State file (terraform.tfstate): a state file is created once after Terraform is run. It stores state about our managed infrastructure.

Also Read: Our blog post on **Terraform Create VM**.

Getting started using Terraform

To get started building infrastructure resources using Terraform, there are few things that you should take care of. The general steps to deploy a resource(s) in the cloud are:

1. Set up a Cloud Account on any cloud provider (AWS, Azure, OCI)
2. Install Terraform

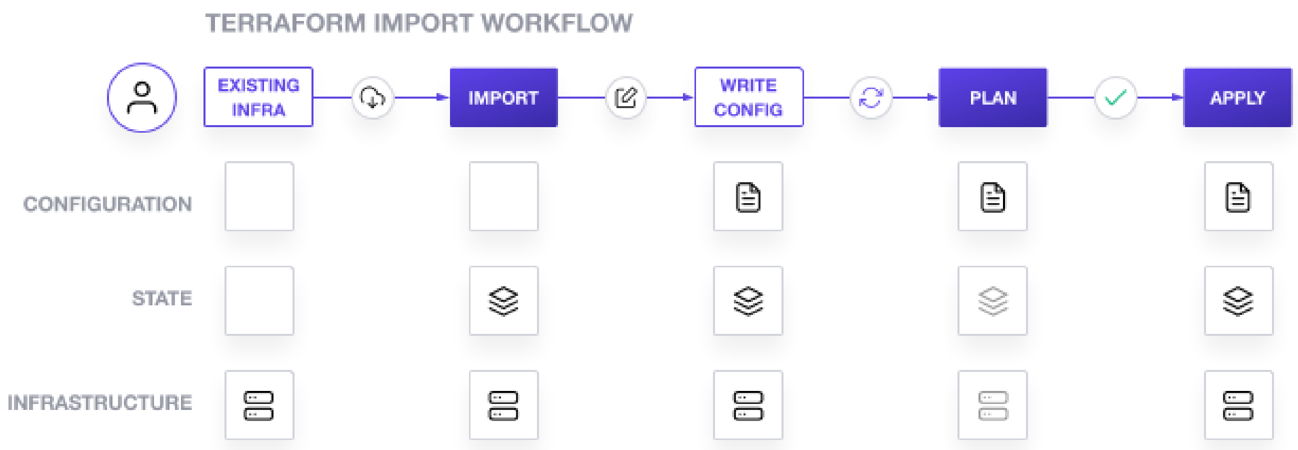
3. Add a provider – AWS, Azure, OCI, GCP, or others
4. Write configuration files
5. Initialize Terraform Providers
6. PLAN (DRY RUN) using terraform plan
7. APPLY (Create a Resource) using terraform apply
8. DESTROY (Delete a Resource) using terraform destroy

Also Check: Our blog post on **Terraform Interview Question**.

Import Existing Infrastructure

Terraform is one of the great IaC tools with which, you can deploy all your infrastructure's resources. In addition to that, you can manage infrastructures from different cloud providers, such as AWS, Google Cloud, etc. But what if you have already created your infrastructure manually?

Terraform has a really nice feature for importing existing resources, which makes the migration of existing infrastructure into Terraform a lot easier.



Currently, Terraform can only import resources into the state. It does not generate a configuration for them. Because of this, prior to running **terraform import** it is necessary to write manually a resource configuration block for the resource, to which the imported object will be mapped. For example:

```
resource "aws_instance" "import_example" {
  # ...instance configuration...
}
```

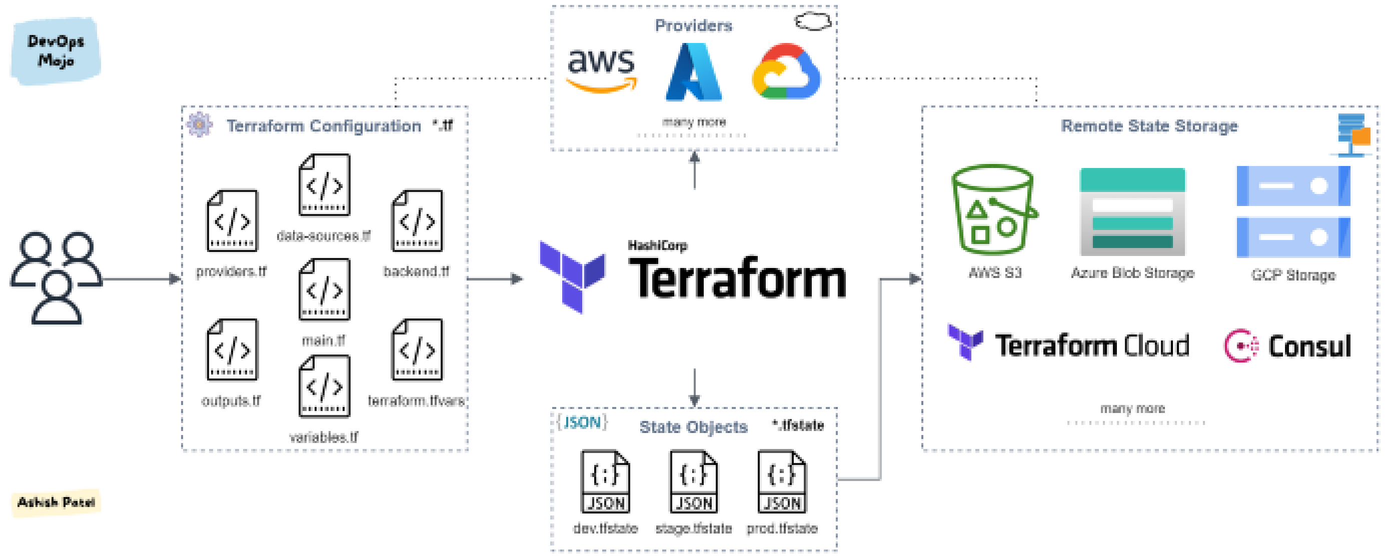
Now **terraform import** can be run to attach an existing instance to this resource configuration:

```
$ terraform import aws_instance.import_example i-03efafa258104165f
```

This command locates the AWS instance with ID i-03efafa258104165f (which has been created outside Terraform) and attaches it to the name **aws_instance.import_example** in the Terraform state.

Terraform — Remote States Overview

What is Terraform Remote State — Introduction to Terraform Remote Storage!



Terraform — Remote States

TL;DR

With *remote* state, Terraform writes the state data to a remote data store, which can be shared between all team members.

Why you need Terraform Remote States?

By default, Terraform stores its state in the file `terraform.tfstate` in local filesystem. This works well for personal projects, but working with Terraform in a team, use of a local file makes Terraform usage complicated because each user must make sure they always have the latest state data before running Terraform and make sure that nobody else runs Terraform at the same time.

The best way to do this is by running Terraform in a remote environment with shared access to state. Remote state solves those challenges. Remote state is simply storing that state file remotely, rather than on your local filesystem. With a single state file stored remotely, teams can ensure they always have the most up to date state file.

Best practice: In a enterprise project and/or if Terraform is used by a team, it is recommended to setup and use remote state.

What is Terraform Backend?

Terraform backends enable you to store the state file in a shared remote store.

- Remote state is implemented by a backend, which you can configure in configuration’s root module.
- Backends determine where state is stored. For example, the local (default) backend stores state in a local JSON file on disk.
- Backends provide an API for state locking. It is optional.

When using a non-local backend, Terraform will not persist the state anywhere on disk except in the case of a non-recoverable error where writing the state to the backend failed.

Benefits of using Terraform Remote State

- **Shared Storage:** Remote state (Backend) allow each of your team members to access same Terraform state files to manage infrastructure.
- **Locking:** With fully-featured remote backends, Terraform can lock the state file while changes are being made. This ensures all changes are captured, even if concurrent changes are being attempted against the same state.
  Without locking, if two team members are running Terraform at the same time, you may run into race conditions as multiple Terraform processes make concurrent updates to the state files, leading to conflicts, data loss, and state file corruption.
- **Versioning:** Some backends support versioning. This maintains versions of your Terraform state files allowing you to download an old version if needed. Likewise, it provides audit logs to know who changed what and when.
- **Encryption:** Many backends support encryption of the state file both in transit and at rest.
- **Security:** A local state file save the content in plain text. It is very common to have secrets or sensitive data in the state, so local state files are insecure. Remote state resolves this issue.
- **Remote operations:** Some backends allow to manage operations remotely (Terraform plan and apply execution). You don’t need to use

terraform on your system to apply the changes. You could either trigger it from a Web UI, API call or CLI tool.

- **Less Manual Errors**: Using a local state file as a shared storage, manually sync the changes could cause someone forget to sync the state file. Remote state will always sync the state automatically whenever it change.
  There are less changes of accidentally deletion of state file when it is stored remotely than on the local machine.

Terraform Remote State Storage Options

Terraform supports storing state in

1. Amazon S3
2. Azure Blob Storage
3. Google Cloud Storage
4. Terraform Cloud
5. HashiCorp Consul
6. Many more.

Read more about: Terraform Workspaces

Backend Configuration Example

Remote state setup can be achieved by setting up backends specific to the cloud.

1. Setting up a remote state in Amazon S3:

```
terraform {
  backend "s3" {
    bucket         = "YOUR_S3_BUCKET_NAME"
    dynamodb_table = "YOUR_DYNAMODB_TABLE_NAME"
    key            = "prod_terraform.tfstate"
    region         = "us-east-1"

    #  Authentication
    profile        = "MY_PROFILE"
  }
}
```

Terraform Variables - *string, number, bool*

Let’s take a simple example in which we are going to set up an EC2 instance on AWS. So to create an EC2 instance we need two things -

1. provider
2. resource

Here is the main.tf which we are going to parameterized using terraform variables.

```
provider "aws" {
  region     = "eu-central-1"
  access_key = "AKIATQ37NXB2OBQHAALW"
  secret_key = "ilKygurap8zSErv7jySTDi2796WGqMkEtN6txNHf"
}

resource "aws_instance" "ec2_example" {

  ami           = "ami-0767046d1677be5a0"
  instance_type = "t2.micro"

  tags = {
          Name = "Terraform EC2"
  }
}
```

BASH

2.1 string variable type - We are going parameterized instance_type = “t2.micro”

The first rule to create a parameter in terraform file is by defining *variable block*

Example -

```
variable "instance_type" {
  description = "Instance type t2.micro"
  type        = string
  default     = "t2.micro"
}
```

BASH

For defining ***variable block*** you need

1. *description* : Small or short description about the purpose of the variable
2. *type* : What type of variable it is going to be ex - string, bool, number …
3. *default* : What would be the default value of the variable

Let’s replace the hardcoded value of instance_type with variable

```
 instance_type = var.instance_type
```

BASH

Here is our final terraform file after replacing the hardcoded value of a variable -

```
provider "aws" {
  region     = "eu-central-1"
  access_key = "AKIATQ37NXB2OBQHAALW"
  secret_key = "ilKygurap8zSErv7jySTDi2796WGqMkEtN6txNHf"
}
```

```
resource "aws_instance" "ec2_example" {

   ami           = "ami-0767046d1677be5a0"
   instance_type =  var.instance_type

   tags = {
           Name = "Terraform EC2"
   }
}

variable "instance_type" {
   description = "Instance type t2.micro"
   type        = string
   default     = "t2.micro"
}
```
BASH

And now you can apply your terraform configuration

```
terraform apply
```
BASH

2.2 number variable type - We are going parameterized instance_count = 2

The next variable type we are going to take is number.
For example, we are going to increase the instance_count of the ec2_instances.
Let's create the variable first -

```
variable "instance_count" {
  description = "EC2 instance count"
  type        = number
  default     = 2
}
```
BASH

Here is the final terraform file with instance count -

```
provider "aws" {
   region     = "eu-central-1"
   access_key = "AKIATQ37NXB2AYK7R6PQ"
   secret_key = "S1Yg1Qm2JNSej8EHdhPTiu5I5ZD36URs3ed2NyYT"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0767046d1677be5a0"
   instance_type =  "t2.micro"
   count = var.instance_count

   tags = {
           Name = "Terraform EC2"
   }
}

variable "instance_count" {
  description = "EC2 instance count"
  type        = number
  default     = 2
}
```

BASH

2.3 boolean variable type - We are going parameterized enable_vpn_gateway = false

The next variable type which we are going to discuss is bool.
The bool variable can be used to set true or false values inside your terraform file.
Here is an example to create your bool variable -

```
variable "enable_public_ip" {
  description = "Enable public IP address"
  type        = bool
  default      = true
}
```

BASH

Let's create a complete terraform file with bool variable -

```
provider "aws" {
   region     = "eu-central-1"
   access_key = "AKIATQ37NXB2AYK7R6PQ"
   secret_key = "S1Yg1Qm2JNSej8EHdhPTiu5I5ZD36URs3ed2NyYT"
}


resource "aws_instance" "ec2_example" {

   ami           = "ami-0767046d1677be5a0"
   instance_type =  "t2.micro"
   count = 1
  associate_public_ip_address = var.enable_public_ip

   tags = {
           Name = "Terraform EC2"
   }

}

variable "enable_public_ip" {
  description = "Enable public IP address"
  type        = bool
  default      = true
}
```

BASH

3. Terraform Variables - *list, set, map*

When it comes to collection input variables then we are talking about -

1. List
2. Map
3. Set

3.1 List variable type

As the name suggests we are going to define a list that will contain more than one element in it.

Let’s define our first List variable –

Here is the list of IAM users

```
variable "user_names" {
  description = "IAM usernames"
  type        = list(string)
  default     = ["user1", "user2", "user3s"]
}
```

BASH

Here is our final terraform file with List variables -

```
provider "aws" {
   region     = "eu-central-1"
   access_key = "AKIATQ37NXB2OBQHAALW"
   secret_key = "ilKygurap8zSErv7jySTDi2796WGqMkEtN6txNHf"
}
resource "aws_instance" "ec2_example" {

   ami           = "ami-0767046d1677be5a0"
   instance_type =  "t2.micro"
   count = 1

   tags = {
           Name = "Terraform EC2"
   }

}

resource "aws_iam_user" "example" {
  count = length(var.user_names)
  name  = var.user_names[count.index]
}

variable "user_names" {
  description = "IAM usernames"
  type        = list(string)
  default     = ["user1", "user2", "user3s"]
}
```

BASH

3.2 Map variable type

Terraform also supports the map variable type where you can define the key-valye pair. Let’s take an example where we need to define project and environment, so we can use the map variable to achieve that.

Here is an example of map variable -

```
variable "project_environment" {
  description = "project name and environment"
  type        = map(string)
  default     = {
    project     = "project-alpha",
    environment = "dev"
  }
}
```

BASH

Let’s create a Terraform file

```
provider "aws" {
   region     = "eu-central-1"
   access_key = "AKIATQ37NXB2OBQHAALW"
   secret_key = "ilKygurap8zSErv7jySTDi2796WGqMkEtN6txNHf"
}
resource "aws_instance" "ec2_example" {

   ami           = "ami-0767046d1677be5a0"
   instance_type =  "t2.micro"

   tags = var.project_environment

}


variable "project_environment" {
  description = "project name and environment"
  type        = map(string)
  default     = {
    project     = "project-alpha",
    environment = "dev"
  }
}
```

Output.tf file :

. How to print the public_ip of aws_instance?

This is one of the most classic examples for terraform output values. As we know terraform is used as Infrastructure as code, so with the help of terraform you can provision many resources such - aws_instances, aws_vpc etc.
But is there a way by which you can know the public_ip address of the instance which you are planning to provision using terraform?
Yes, there is a really convenient and easy way to achieve that.
Here is my aws_instance which I have defined as inside my main.tf -

```
provider "aws" {
   region     = "eu-central-1"
   access_key = "AKIATQ37NXB2G2LXXXXX"
   secret_key = "r1oaShokKPw+YY7qaHxj8mD2T8BpxRUVXXXXXXXX"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0767046d1677be5a0"
   instance_type = "t2.micro"
   subnet_id = aws_subnet.staging-subnet.id
```

```
  tags = {
        Name = "test - Terraform EC2"
  }
}
```

You can simply append following terraform output value code inside your main.tf file so that it can print the public ip of your aws_instance

```
output "my_console_output" {
  value = aws_instance.ec2_example.public_ip
}
```

BASH

Break down of the above script

1. aws_instance - It is the keyword which you need write as is.
2. ec2_example - The name which we have given at the time of creating aws_instance
3. public_ip - You can use **attributes reference** page to get all the attribute which you want to print on console.

The same approach can be followed to print
- arn,instance_state,outpost_arn, password_data, primary_network_interface_id, private_dns, public_dns etc.

2. How to create output.tf for terraform output values?

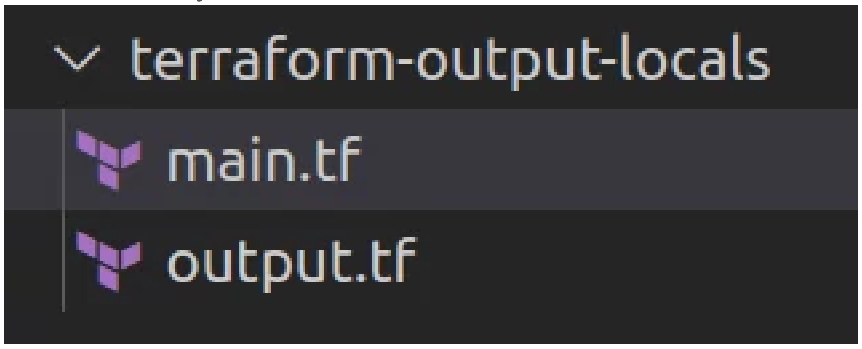
In the previous point we have seen how to create your first **terraform output values**. But if you have noticed we have created the terraform output values inside the same main.tffile.
It is completely okay to create **terraform output values** inside the main.tf but this practice is certainly not recommended in the industry.
The recommended way is to create a separate output.tf specially of the **terraform output values**, so that all the output values can be stored there.
The only change which you need to do over here is to create a new output.tf and store the following terraform code init -

output.tf

```
output "my_console_output" {
  value = aws_instance.ec2_example.public_ip
}
```

BASH

Your directory structure should look like this -



```
terraform-output-locals
  main.tf
  output.tf
```

How to use terraform output locals?

3. How to prevent printing sensitive info on the console?

As terraform output values help us to print the attributes reference values but sometimes you can not print all the values on console.
So to prevent from printing sensitive values on the console you need to set `sensitive = true`.
Here is an example of the terraform code where we want to prevent showing the **public ip** to console -

```
output "my_console_output" {
  value = aws_instance.ec2_example.public_ip
  sensitive = true
}
```

BASH

In the above code if you noticed we are using `sensitive = true` which tells terraform not to show `public ip` on console.