Terraform must first be installed on your machine. Terraform is distributed as a [binary package](https://www.terraform.io/downloads.html) for all supported platforms and architectures. This page will not cover how to compile Terraform from source, but compiling from source is covered in the [documentation](https://www.terraform.io/docs/index.html) for those who want to be sure they're compiling source they trust into the final binary.

[**»**](https://learn.hashicorp.com/terraform/getting-started/install#installing-terraform) **Installing Terraform**

To install Terraform, find the [appropriate package](https://www.terraform.io/downloads.html) for your system and download it. Terraform is packaged as a zip archive.

After downloading Terraform, unzip the package. Terraform runs as a single binary named terraform. Any other files in the package can be safely removed and Terraform will still function.

The final step is to make sure that the terraform binary is available on the PATH. See [this page](https://stackoverflow.com/questions/14637979/how-to-permanently-set-path-on-linux) for instructions on setting the PATH on Linux and Mac. [This page](https://stackoverflow.com/questions/1618280/where-can-i-set-path-to-make-exe-on-windows) contains instructions for setting the PATH on Windows.

[**»**](https://learn.hashicorp.com/terraform/getting-started/install#verifying-the-installation) **Verifying the Installation**

After installing Terraform, verify the installation worked by opening a new terminal session and checking that terraform is available. By executing terraform you should see help output similar to this:

$ terraform

Usage: terraform [--version] [--help] <command> [args]

The available commands for execution are listed below.

The most common, useful commands are shown first, followed by

less common or more advanced commands. If you're just getting

started with Terraform, stick with the common commands. For the

other commands, please read the help and docs before usage.

Common commands:

apply Builds or changes infrastructure

console Interactive console for Terraform interpolations

# ...

If you get an error that terraform could not be found, your PATH environment variable was not set up properly. Please go back and ensure that your PATH variable contains the directory where Terraform was installed.

With Terraform installed, let's dive right into it and start creating some infrastructure.

We'll build infrastructure on [AWS](https://aws.amazon.com) for the getting started guide since it is popular and generally understood, but Terraform can [manage many providers](https://www.terraform.io/docs/providers/index.html), including multiple providers in a single configuration. Some examples of this are in the [use cases section](https://www.terraform.io/intro/use-cases.html).

If you don't have an AWS account, [create one now](https://aws.amazon.com/free/). For the getting started guide, we'll only be using resources which qualify under the AWS [free-tier](https://aws.amazon.com/free/), meaning it will be free. If you already have an AWS account, you may be charged some amount of money, but it shouldn't be more than a few dollars at most.

**Warning!** If you're not using an account that qualifies under the AWS [free-tier](https://aws.amazon.com/free/), you may be charged to run these examples. The most you should be charged should only be a few dollars, but we're not responsible for any charges that may incur.

## [**»**](https://learn.hashicorp.com/terraform/getting-started/build#configuration) **Configuration**

The set of files used to describe infrastructure in Terraform is simply known as a Terraform *configuration*. We're going to write our first configuration now to launch a single AWS EC2 instance.

The format of the configuration files is [documented here](https://www.terraform.io/docs/configuration/index.html). Configuration files can [also be JSON](https://www.terraform.io/docs/configuration/syntax.html), but we recommend only using JSON when the configuration is generated by a machine.

The entire configuration is shown below. We'll go over each part after. Save the contents to a file named example.tf. Verify that there are no other \*.tf files in your directory, since Terraform loads all of them.

provider "aws" {

access\_key = "ACCESS\_KEY\_HERE"

secret\_key = "SECRET\_KEY\_HERE"

region = "us-east-1"

}

resource "aws\_instance" "example" {

ami = "ami-2757f631"

instance\_type = "t2.micro"

}

**Note**: The above configuration is designed to work on most EC2 accounts, with access to a default VPC. For EC2 Classic users, please use t1.micro for instance\_type, and ami-408c7f28 for the ami. If you use a region other than us-east-1 then you will need to choose an AMI in that region as AMI IDs are region specific.

Replace the ACCESS\_KEY\_HERE and SECRET\_KEY\_HERE with your AWS access key and secret key, available from [this page](https://console.aws.amazon.com/iam/home?#security_credential). We're hardcoding them for now, but will extract these into variables later in the getting started guide.

**Note**: If you simply leave out AWS credentials, Terraform will automatically search for saved API credentials (for example, in ~/.aws/credentials) or IAM instance profile credentials. This option is much cleaner for situations where tf files are checked into source control or where there is more than one admin user. See details [here](https://aws.amazon.com/blogs/apn/terraform-beyond-the-basics-with-aws/). Leaving IAM credentials out of the Terraform configs allows you to leave those credentials out of source control, and also use different IAM credentials for each user without having to modify the configuration files.

This is a complete configuration that Terraform is ready to apply. The general structure should be intuitive and straightforward.

The provider block is used to configure the named provider, in our case "aws". A provider is responsible for creating and managing resources. Multiple provider blocks can exist if a Terraform configuration is composed of multiple providers, which is a common situation.

The resource block defines a resource that exists within the infrastructure. A resource might be a physical component such as an EC2 instance, or it can be a logical resource such as a Heroku application.

The resource block has two strings before opening the block: the resource type and the resource name. In our example, the resource type is "aws\_instance" and the name is "example." The prefix of the type maps to the provider. In our case "aws\_instance" automatically tells Terraform that it is managed by the "aws" provider.

Within the resource block itself is configuration for that resource. This is dependent on each resource provider and is fully documented within our [providers reference](https://www.terraform.io/docs/providers/index.html). For our EC2 instance, we specify an AMI for Ubuntu, and request a "t2.micro" instance so we qualify under the free tier.

## [**»**](https://learn.hashicorp.com/terraform/getting-started/build#initialization) **Initialization**

The first command to run for a new configuration -- or after checking out an existing configuration from version control -- is terraform init, which initializes various local settings and data that will be used by subsequent commands.

Terraform uses a plugin based architecture to support the numerous infrastructure and service providers available. As of Terraform version 0.10.0, each "Provider" is its own encapsulated binary distributed separately from Terraform itself. The terraform init command will automatically download and install any Provider binary for the providers in use within the configuration, which in this case is just the aws provider:

$ terraform init

Initializing the backend...

Initializing provider plugins...

- downloading plugin for provider "aws"...

The following providers do not have any version constraints in configuration,

so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking

changes, it is recommended to add version = "..." constraints to the

corresponding provider blocks in configuration, with the constraint strings

suggested below.

\* provider.aws: version = "~> 1.0"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see

any changes that are required for your infrastructure. All Terraform commands

should now work.

If you ever set or change modules or backend configuration for Terraform,

rerun this command to reinitialize your environment. If you forget, other

commands will detect it and remind you to do so if necessary.

The aws provider plugin is downloaded and installed in a subdirectory of the current working directory, along with various other book-keeping files.

The output specifies which version of the plugin was installed, and suggests specifying that version in configuration to ensure that running terraform init in future will install a compatible version. This step is not necessary for following the getting started guide, since this configuration will be discarded at the end.

## [**»**](https://learn.hashicorp.com/terraform/getting-started/build#apply-changes) **Apply Changes**

**Note:** The commands shown in this guide apply to Terraform 0.11 and above. Earlier versions require using the terraform plan command to see the execution plan before applying it. Use terraform version to confirm your running version.

In the same directory as the example.tf file you created, run terraform apply. You should see output similar to below, though we've truncated some of the output to save space:

$ terraform apply

# ...

+ aws\_instance.example

ami: "ami-2757f631"

availability\_zone: "<computed>"

ebs\_block\_device.#: "<computed>"

ephemeral\_block\_device.#: "<computed>"

instance\_state: "<computed>"

instance\_type: "t2.micro"

key\_name: "<computed>"

placement\_group: "<computed>"

private\_dns: "<computed>"

private\_ip: "<computed>"

public\_dns: "<computed>"

public\_ip: "<computed>"

root\_block\_device.#: "<computed>"

security\_groups.#: "<computed>"

source\_dest\_check: "true"

subnet\_id: "<computed>"

tenancy: "<computed>"

vpc\_security\_group\_ids.#: "<computed>"

This output shows the *execution plan*, describing which actions Terraform will take in order to change real infrastructure to match the configuration. The output format is similar to the diff format generated by tools such as Git. The output has a + next to aws\_instance.example, meaning that Terraform will create this resource. Beneath that, it shows the attributes that will be set. When the value displayed is <computed>, it means that the value won't be known until the resource is created.

If terraform apply failed with an error, read the error message and fix the error that occurred. At this stage, it is likely to be a syntax error in the configuration.

If the plan was created successfully, Terraform will now pause and wait for approval before proceeding. If anything in the plan seems incorrect or dangerous, it is safe to abort here with no changes made to your infrastructure. In this case the plan looks acceptable, so type yes at the confirmation prompt to proceed.

Executing the plan will take a few minutes since Terraform waits for the EC2 instance to become available:

# ...

aws\_instance.example: Creating...

ami: "" => "ami-2757f631"

instance\_type: "" => "t2.micro"

[...]

aws\_instance.example: Still creating... (10s elapsed)

aws\_instance.example: Creation complete

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

# ...

After this, Terraform is all done! You can go to the EC2 console to see the created EC2 instance. (Make sure you're looking at the same region that was configured in the provider configuration!)

Terraform also wrote some data into the terraform.tfstate file. This state file is extremely important; it keeps track of the IDs of created resources so that Terraform knows what it is managing. This file must be saved and distributed to anyone who might run Terraform. It is generally recommended to [setup remote state](https://www.terraform.io/docs/state/remote.html) when working with Terraform, to share the state automatically, but this is not necessary for simple situations like this Getting Started guide.

You can inspect the current state using terraform show:

$ terraform show

aws\_instance.example:

id = i-32cf65a8

ami = ami-2757f631

availability\_zone = us-east-1a

instance\_state = running

instance\_type = t2.micro

private\_ip = 172.31.30.244

public\_dns = ec2-52-90-212-55.compute-1.amazonaws.com

public\_ip = 52.90.212.55

subnet\_id = subnet-1497024d

vpc\_security\_group\_ids.# = 1

vpc\_security\_group\_ids.3348721628 = sg-67652003

You can see that by creating our resource, we've also gathered a lot of information about it. These values can actually be referenced to configure other resources or outputs, which will be covered later in the getting started guide.

## [**»**](https://learn.hashicorp.com/terraform/getting-started/build#provisioning) **Provisioning**

The EC2 instance we launched at this point is based on the AMI given, but has no additional software installed. If you're running an image-based infrastructure (perhaps creating images with [Packer](https://www.packer.io)), then this is all you need.

However, many infrastructures still require some sort of initialization or software provisioning step. Terraform supports provisioners, which we'll cover a little bit later in the getting started guide, in order to do this.

In the previous page, you created your first infrastructure with Terraform: a single EC2 instance. In this page, we're going to modify that resource, and see how Terraform handles change.

Infrastructure is continuously evolving, and Terraform was built to help manage and enact that change. As you change Terraform configurations, Terraform builds an execution plan that only modifies what is necessary to reach your desired state.

By using Terraform to change infrastructure, you can version control not only your configurations but also your state so you can see how the infrastructure evolved over time.

## [**»**](https://learn.hashicorp.com/terraform/getting-started/change#configuration) **Configuration**

Let's modify the ami of our instance. Edit the aws\_instance.example resource in your configuration and change it to the following:

resource "aws\_instance" "example" {

ami = "ami-b374d5a5"

instance\_type = "t2.micro"

}

**Note:** EC2 Classic users please use AMI ami-656be372 and type t1.micro

We've changed the AMI from being an Ubuntu 16.04 LTS AMI to being an Ubuntu 16.10 AMI. Terraform configurations are meant to be changed like this. You can also completely remove resources and Terraform will know to destroy the old one.

## [**»**](https://learn.hashicorp.com/terraform/getting-started/change#apply-changes) **Apply Changes**

After changing the configuration, run terraform apply again to see how Terraform will apply this change to the existing resources.

$ terraform apply

# ...

-/+ aws\_instance.example

ami: "ami-2757f631" => "ami-b374d5a5" (forces new resource)

availability\_zone: "us-east-1a" => "<computed>"

ebs\_block\_device.#: "0" => "<computed>"

ephemeral\_block\_device.#: "0" => "<computed>"

instance\_state: "running" => "<computed>"

instance\_type: "t2.micro" => "t2.micro"

private\_dns: "ip-172-31-17-94.ec2.internal" => "<computed>"

private\_ip: "172.31.17.94" => "<computed>"

public\_dns: "ec2-54-82-183-4.compute-1.amazonaws.com" => "<computed>"

public\_ip: "54.82.183.4" => "<computed>"

subnet\_id: "subnet-1497024d" => "<computed>"

vpc\_security\_group\_ids.#: "1" => "<computed>"

The prefix -/+ means that Terraform will destroy and recreate the resource, rather than updating it in-place. While some attributes can be updated in-place (which are shown with the ~ prefix), changing the AMI for an EC2 instance requires recreating it. Terraform handles these details for you, and the execution plan makes it clear what Terraform will do.

Additionally, the execution plan shows that the AMI change is what required resource to be replaced. Using this information, you can adjust your changes to possibly avoid destroy/create updates if they are not acceptable in some situations.

Once again, Terraform prompts for approval of the execution plan before proceeding. Answer yes to execute the planned steps:

# ...

aws\_instance.example: Refreshing state... (ID: i-64c268fe)

aws\_instance.example: Destroying...

aws\_instance.example: Destruction complete

aws\_instance.example: Creating...

ami: "" => "ami-b374d5a5"

availability\_zone: "" => "<computed>"

ebs\_block\_device.#: "" => "<computed>"

ephemeral\_block\_device.#: "" => "<computed>"

instance\_state: "" => "<computed>"

instance\_type: "" => "t2.micro"

key\_name: "" => "<computed>"

placement\_group: "" => "<computed>"

private\_dns: "" => "<computed>"

private\_ip: "" => "<computed>"

public\_dns: "" => "<computed>"

public\_ip: "" => "<computed>"

root\_block\_device.#: "" => "<computed>"

security\_groups.#: "" => "<computed>"

source\_dest\_check: "" => "true"

subnet\_id: "" => "<computed>"

tenancy: "" => "<computed>"

vpc\_security\_group\_ids.#: "" => "<computed>"

aws\_instance.example: Still creating... (10s elapsed)

aws\_instance.example: Still creating... (20s elapsed)

aws\_instance.example: Creation complete

We've now seen how to build and change infrastructure. Before we move on to creating multiple resources and showing resource dependencies, we're going to go over how to completely destroy the Terraform-managed infrastructure.

Destroying your infrastructure is a rare event in production environments. But if you're using Terraform to spin up multiple environments such as development, test, QA environments, then destroying is a useful action.

## [**»**](https://learn.hashicorp.com/terraform/getting-started/destroy#destroy) **Destroy**

Resources can be destroyed using the terraform destroy command, which is similar to terraform apply but it behaves as if all of the resources have been removed from the configuration.

$ terraform destroy

# ...

- aws\_instance.example

The - prefix indicates that the instance will be destroyed. As with apply, Terraform shows its execution plan and waits for approval before making any changes.

Answer yes to execute this plan and destroy the infrastructure:

# ...

aws\_instance.example: Destroying...

Apply complete! Resources: 0 added, 0 changed, 1 destroyed.

# ...

Just like with apply, Terraform determines the order in which things must be destroyed. In this case there was only one resource, so no ordering was necessary. In more complicated cases with multiple resources, Terraform will destroy them in a suitable order to respect dependencies, as we'll see later in this guide.

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.

# ...

As indicated by the execution plan, Terraform first destroyed the existing instance and then created a new one in its place. You can use terraform show again to see the new values associated with this instance.

In this page, we're going to introduce resource dependencies, where we'll not only see a configuration with multiple resources for the first time, but also scenarios where resource parameters use information from other resources.

Up to this point, our example has only contained a single resource. Real infrastructure has a diverse set of resources and resource types. Terraform configurations can contain multiple resources, multiple resource types, and these types can even span multiple providers.

On this page, we'll show a basic example of multiple resources and how to reference the attributes of other resources to configure subsequent resources.

## [**»**](https://learn.hashicorp.com/terraform/getting-started/dependencies#assigning-an-elastic-ip) **Assigning an Elastic IP**

We'll improve our configuration by assigning an elastic IP to the EC2 instance we're managing. Modify your example.tf and add the following:

resource "aws\_eip" "ip" {

instance = "${aws\_instance.example.id}"

}

This should look familiar from the earlier example of adding an EC2 instance resource, except this time we're building an "aws\_eip" resource type. This resource type allocates and associates an [elastic IP](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html) to an EC2 instance.

The only parameter for [aws\_eip](https://www.terraform.io/docs/providers/aws/r/eip.html) is "instance" which is the EC2 instance to assign the IP to. For this value, we use an interpolation to use an attribute from the EC2 instance we managed earlier.

The syntax for this interpolation should be straightforward: it requests the "id" attribute from the "aws\_instance.example" resource.

## [**»**](https://learn.hashicorp.com/terraform/getting-started/dependencies#apply-changes) **Apply Changes**

Run terraform apply to see how Terraform plans to apply this change. The output will look similar to the following:

$ terraform apply

+ aws\_eip.ip

allocation\_id: "<computed>"

association\_id: "<computed>"

domain: "<computed>"

instance: "${aws\_instance.example.id}"

network\_interface: "<computed>"

private\_ip: "<computed>"

public\_ip: "<computed>"

+ aws\_instance.example

ami: "ami-b374d5a5"

availability\_zone: "<computed>"

ebs\_block\_device.#: "<computed>"

ephemeral\_block\_device.#: "<computed>"

instance\_state: "<computed>"

instance\_type: "t2.micro"

key\_name: "<computed>"

placement\_group: "<computed>"

private\_dns: "<computed>"

private\_ip: "<computed>"

public\_dns: "<computed>"

public\_ip: "<computed>"

root\_block\_device.#: "<computed>"

security\_groups.#: "<computed>"

source\_dest\_check: "true"

subnet\_id: "<computed>"

tenancy: "<computed>"

vpc\_security\_group\_ids.#: "<computed>"

Terraform will create two resources: the instance and the elastic IP. In the "instance" value for the "aws\_eip", you can see the raw interpolation is still present. This is because this variable won't be known until the "aws\_instance" is created. It will be replaced at apply-time.

As usual, Terraform prompts for confirmation before making any changes. Answer yes to apply. The continued output will look similar to the following:

# ...

aws\_instance.example: Creating...

ami: "" => "ami-b374d5a5"

instance\_type: "" => "t2.micro"

[..]

aws\_instance.example: Still creating... (10s elapsed)

aws\_instance.example: Creation complete

aws\_eip.ip: Creating...

allocation\_id: "" => "<computed>"

association\_id: "" => "<computed>"

domain: "" => "<computed>"

instance: "" => "i-f3d77d69"

network\_interface: "" => "<computed>"

private\_ip: "" => "<computed>"

public\_ip: "" => "<computed>"

aws\_eip.ip: Creation complete

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

As shown above, Terraform created the EC2 instance before creating the Elastic IP address. Due to the interpolation expression that passes the ID of the EC2 instance to the Elastic IP address, Terraform is able to infer a dependency, and knows it must create the instance first.

## [**»**](https://learn.hashicorp.com/terraform/getting-started/dependencies#implicit-and-explicit-dependencies) **Implicit and Explicit Dependencies**

By studying the resource attributes used in interpolation expressions, Terraform can automatically infer when one resource depends on another. In the example above, the expression ${aws\_instance.example.id} creates an *implicit dependency* on the aws\_instance named example.

Terraform uses this dependency information to determine the correct order in which to create the different resources. In the example above, Terraform knows that the aws\_instance must be created before the aws\_eip.

Implicit dependencies via interpolation expressions are the primary way to inform Terraform about these relationships, and should be used whenever possible.

Sometimes there are dependencies between resources that are *not* visible to Terraform. The depends\_on argument is accepted by any resource and accepts a list of resources to create *explicit dependencies* for.

For example, perhaps an application we will run on our EC2 instance expects to use a specific Amazon S3 bucket, but that dependency is configured inside the application code and thus not visible to Terraform. In that case, we can use depends\_on to explicitly declare the dependency:

# New resource for the S3 bucket our application will use.

resource "aws\_s3\_bucket" "example" {

# NOTE: S3 bucket names must be unique across \_all\_ AWS accounts, so

# this name must be changed before applying this example to avoid naming

# conflicts.

bucket = "terraform-getting-started-guide"

acl = "private"

}

# Change the aws\_instance we declared earlier to now include "depends\_on"

resource "aws\_instance" "example" {

ami = "ami-2757f631"

instance\_type = "t2.micro"

# Tells Terraform that this EC2 instance must be created only after the

# S3 bucket has been created.

depends\_on = ["aws\_s3\_bucket.example"]

}

## [**»**](https://learn.hashicorp.com/terraform/getting-started/dependencies#non-dependent-resources) **Non-Dependent Resources**

We can continue to build this configuration by adding another EC2 instance:

resource "aws\_instance" "another" {

ami = "ami-b374d5a5"

instance\_type = "t2.micro"

}

Because this new instance does not depend on any other resource, it can be created in parallel with the other resources. Where possible, Terraform will perform operations concurrently to reduce the total time taken to apply changes.

Before moving on, remove this new resource from your configuration and run terraform apply again to destroy it. We won't use this second instance any further in the getting started guide.

**Terraform Modules**

## What are modules for?

Here are some of the ways that modules help solve the problems listed above:

* Organize configuration - Modules make it easier to navigate, understand, and update your configuration by keeping related parts of your configuration together. Even moderately complex infrastructure can require hundreds or thousands of lines of configuration to implement. By using modules, you can organize your configuration into logical components.
* Encapsulate configuration - Another benefit of using modules is to encapsulate configuration into distinct logical components. Encapsulation can help prevent unintended consequences, such as a change to one part of your configuration accidentally causing changes to other infrastructure, and reduce the chances of simple errors like using the same name for two different resources.
* Re-use configuration - Writing all of your configuration from scratch can be time consuming and error prone. Using modules can save time and reduce costly errors by re-using configuration written either by yourself, other members of your team, or other Terraform practitioners who have published modules for you to use. You can also share modules that you have written with your team or the general public, giving them the benefit of your hard work.
* Provide consistency and ensure best practices - Modules also help to provide consistency in your configurations. Not only does consistency make complex configurations easier to understand, it also helps to ensure that best practices are applied across all of your configuration. For instance, cloud providers give many options for configuring object storage services, such as Amazon S3 or Google Cloud Storage buckets. There have been many high-profile security incidents involving incorrectly secured object storage, and given the number of complex configuration options involved, it's easy to accidentally misconfigure these services.  
  Using modules can help reduce these errors. For example, you might create a module to describe how all of your organization's public website buckets will be configured, and another module for private buckets used for logging applications. Also, if a configuration for a type of resource needs to be updated, using modules allows you to make that update in a single place and have it be applied to all cases where you use that module.

## [»](https://learn.hashicorp.com/tutorials/terraform/module?in=terraform/modules#what-is-a-terraform-module)What is a Terraform module?

A Terraform module is a set of Terraform configuration files in a single directory. Even a simple configuration consisting of a single directory with one or more .tf files is a module. When you run Terraform commands directly from such a directory, it is considered the **root module**. So in this sense, every Terraform configuration is part of a module. You may have a simple set of Terraform configuration files such as:

$ tree minimal-module/

.

├── LICENSE

├── README.md

├── main.tf

├── variables.tf

├── outputs.tf

In this case, when you run terraform commands from within the minimal-module directory, the contents of that directory are considered the root module.

### [»](https://learn.hashicorp.com/tutorials/terraform/module?in=terraform/modules#calling-modules)Calling modules

Terraform commands will only directly use the configuration files in one directory, which is usually the current working directory. However, your configuration can use module blocks to call modules in other directories. When Terraform encounters a module block, it loads and processes that module's configuration files.

A module that is called by another configuration is sometimes referred to as a "child module" of that configuration.

### [»](https://learn.hashicorp.com/tutorials/terraform/module?in=terraform/modules#local-and-remote-modules)Local and remote modules

Modules can either be loaded from the local filesystem, or a remote source. Terraform supports a variety of remote sources, including the Terraform Registry, most version control systems, HTTP URLs, and Terraform Cloud or Terraform Enterprise private module registries.

## [»](https://learn.hashicorp.com/tutorials/terraform/module?in=terraform/modules#module-best-practices)Module best practices

In many ways, Terraform modules are similar to the concepts of libraries, packages, or modules found in most programming languages, and provide many of the same benefits. Just like almost any non-trivial computer program, real-world Terraform configurations should almost always use modules to provide the benefits mentioned above.

We recommend that every Terraform practitioner use modules by following these best practices:

1. Start writing your configuration with modules in mind. Even for modestly complex Terraform configurations managed by a single person, you'll find the benefits of using modules outweigh the time it takes to use them properly.
2. Use local modules to organize and encapsulate your code. Even if you aren't using or publishing remote modules, organizing your configuration in terms of modules from the beginning will significantlty reduce the burden of maintaining and updating your configuration as your infrastructure grows in complexity.
3. Use the public Terraform Registry to find useful modules. This way you can more quickly and confidently implement your configuration by relying on the work of others to implement common infrastructure scenarios.
4. Publish and share modules with your team. Most infrastructure is managed by a team of people, and modules are important way that teams can work together to create and maintain infrastructure.

Terraform Configuration:

Remember about 3 files

1. main.tf - Main tf file, which you have created in the last step
2. variables.tf - the values in this can be passed to the module block as arguments. Remember there are certain datatypes in this(string/list-which is array / boolean)
3. outputs.tf - the values which will be displayed as output(if you don't want to see the default output)

Example of These files:

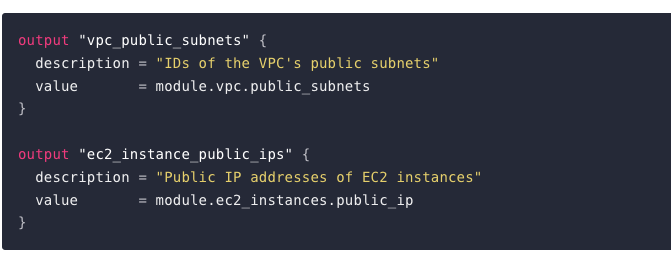
1. main.tf - This is the main file, which you have worked in the previous example. Example given below.



1. variables.tf - The values that you want to pass in the main.tf file above. Remember that you will have to refer it to as “var.” and so on example(var.vpc\_name) and so on. Example of file below.



1. outputs.tf - This contains the result the should be displayed. Example below.



1. Rest all is same as “terraform init “ and “terraform apply”