**Build Infrastructure**

**Write Your own Configuration**

The set of files used to describe infrastructure in Terraform is known as a **Terraform configuration**. You'll write your first configuration now to launch a single AWS EC2 instance.

Before we start, your AWS credentials must be configured locally;

* With your account created and the CLI installed configure the AWS CLI.

$ aws configure

* Follow the prompts to input your AWS Access Key ID and Secret Access Key, which you'll find on [this page](https://console.aws.amazon.com/iam/home?#security_credential) .

Each configuration should be in its own directory. Create a directory for the new configuration and change into the directory.

$ mkdir learn-terraform-aws-instance && cd learn-terraform-aws-instance

Create a file for the configuration code.

$ touch example.tf

Paste the configuration below into example.tf and save it. Terraform loads all files in the working directory that end in .tf.

provider "aws" {

access\_key = "Your AWS account's access\_key provided by credentials\_aws.csv"

secret\_key = "Your AWS account's secret\_key provided by credentials\_aws.csv"

profile = "default"

region = "us-east-1"

}

resource "aws\_instance" "example" {

ami = "ami-2757f631"

instance\_type = "t2.micro"

}

This is a complete configuration that Terraform is ready to apply. In the following sections, we'll review each block of the configuration in more detail.

**Providers**

* The provider block configures the named provider, in our case aws, which is responsible for creating and managing resources. A provider is a plugin that Terraform uses to translate the API interactions with the service.
* The **profile** attribute in your provider block refers Terraform to the AWS credentials stored in your AWS Config File, which you created when you configured the AWS CLI.

**Note:**

* If you leave out your AWS credentials, Terraform will automatically search for saved API credentials (for example, in **~/.aws/credentials**) or IAM instance profile credentials. This is cleaner when .tf files are checked into source control or if there is more than one admin user.
* Terraform Engineers recommends that you never hard-code credentials into \*.tf configuration files. We are explicitly defining the default AWS config profile here to illustrate how Terraform should access sensitive credentials.

**Resources**

* The **resource** block defines a piece of 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 the block: the **resource type** and the **resource name**. In the 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.
* The arguments for the resource are within the resource block. The arguments could be things like machine sizes, disk image names, or VPC IDs. Our providers reference documents the required and optional arguments for each resource provider. For your EC2 instance, you specified an AMI for Ubuntu, and requested a t2.micro instance so you qualify under the free tier.

### Initialize the directory

When you create a new configuration, or check out an existing configuration from version control, you need to initialize the directory with terraform init command.

Terraform uses a plugin-based architecture to support hundreds of infrastructure and service providers. Initializing a configuration directory downloads and installs providers used in the configuration, which in this case is the aws provider. Subsequent commands will use local settings and data during initialization.

Initialize the directory.

$ terraform init

Initializing the backend...

Initializing provider plugins...

- Checking for available provider plugins...

- Downloading plugin for provider "aws" (terraform-providers/aws) 2.10.0...

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 = "~> 2.10"

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 working directory. If you forget,

    other

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

*Terraform downloads the aws provider and installs it in a hidden subdirectory of the current working directory. The output shows which version of the plugin was installed.*

*During terraform init, if version argument is not specified, the most recent provider will be downloaded during initialization. For production use, you should constrain the acceptable provider versions via configuration, to ensure that new versions with breaking changes will not be automatically installed.*

provider "aws" {

region = "us-east-1"

version = "2.7"

}

### Format and Validate the Configuration

Terraform (Hashicorp Team) recommends using consistent formatting in files and modules written by different teams. The terraform fmt command automatically updates configurations in the current directory for easy readability and consistency.

Format your configuration. Terraform will return the names of the files it formatted. For our case, which we shared the configuration file, your configuration file was already formatted correctly, so Terraform won't return any file names.

$ terraform fmt

If you are copying configuration snippets or just want to make sure your configuration is syntactically **valid** and internally **consistent**, the built in terraform validate command will **check** and **report** errors within modules, attribute names, and value types.

Validate your configuration. If your configuration is valid, Terraform will return a success message.

$ terraform validate

Success! The configuration is valid.

Formun Üstü

### Create infrastructure

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

$ terraform apply

## ... Output truncated ...

An execution plan has been generated and is shown below.

Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

# aws\_instance.example will be created

+ resource "aws\_instance" "example" {

+ ami = "ami-2757f631"

+ arn = (known after apply)

+ associate\_public\_ip\_address = (known after apply)

+ availability\_zone = (known after apply)

+ cpu\_core\_count = (known after apply)

+ cpu\_threads\_per\_core = (known after apply)

+ get\_password\_data = false

+ host\_id = (known after apply)

+ id = (known after apply)

+ instance\_state = (known after apply)

+ instance\_type = "t2.micro"

+ ipv6\_address\_count = (known after apply)

+ ipv6\_addresses = (known after apply)

## ... Output truncated ...

Plan: 1 to add, 0 to change, 0 to destroy.

You can 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.

Note: Terraform 0.11 and earlier require running **terraform plan** before **terraform apply**. Use **terraform version** to confirm your running version.

You've now created infrastructure using Terraform! Visit 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!

### Inspect State

When you applied your configuration, Terraform wrote data into a file called terraform.tfstate. This file now contains the IDs and properties of the resources Terraform created so that it can manage or destroy those resources going forward. You must save your state file securely and distribute it only to trusted team members who need to manage your infrastructure.

Inspect the current state using terraform show.

$ terraform show

# aws\_instance.example:

resource "aws\_instance" "example" {

ami = "ami-2757f631"

arn = "arn:aws:ec2:us-east-1:130490850807

        :instance/i-0bbf06244e44211d1"

associate\_public\_ip\_address = true

availability\_zone = "us-east-1c"

cpu\_core\_count = 1

cpu\_threads\_per\_core = 1

disable\_api\_termination = false

ebs\_optimized = false

get\_password\_data = false

id = "i-0bbf06244e44211d1"

instance\_state = "running"

instance\_type = "t2.micro"

ipv6\_address\_count = 0

ipv6\_addresses = []

monitoring = false

primary\_network\_interface\_id = "eni-0f1ce5bdae258b015"

private\_dns = "ip-172-31-61-141.ec2.internal"

private\_ip = "172.31.61.141"

public\_dns = "ec2-54-166-19-244.compute-1.amazonaws.com"

public\_ip = "54.166.19.244"

security\_groups = [

"default",

]

source\_dest\_check = true

subnet\_id = "subnet-1facdf35"

tenancy = "default"

volume\_tags = {}

vpc\_security\_group\_ids = [

"sg-5255f429",

]

credit\_specification {

cpu\_credits = "standard"

}

root\_block\_device {

delete\_on\_termination = true

iops = 100

volume\_id = "vol-0079e485d9e28a8e5"

volume\_size = 8

volume\_type = "gp2"

}

}

***Note:***  
When Terraform created this EC2 instance, it also gathered a lot of information about it. These values can be referenced to configure other resources or outputs.

Terraform has a built in command called terraform state for advanced state management. For example, if you have a long state file, you may want a list of the resources in state, which you can get by using the list subcommand.

$ terraform state list

aws\_instance.example

Formun Üstü

### Getting Error?

If you are getting error please read this content too, but if you did all the process with success, you can click to **Next**.

If terraform validate was successful and your apply still failed, you may be encountering a common error.

* If you use a region **other than us-east-1**, you will also need to **change your ami**, since AMI IDs are region specific. Choose an AMI ID specific to your region by following these instructions, and modify example.tf with this ID. Then re-run terraform apply.
* If you do not have a default VPC in your AWS account in the correct region, navigate to the AWS VPC Dashboard in the web UI, create a new VPC in your region, and associate a subnet and security group to that VPC. Then add the security group ID (vpc\_security\_group\_ids) and subnet ID (subnet\_id) into your aws\_instance resource, and replace the values with the ones from your new security group and subnet.

resource "aws\_instance" "example" {

ami = "ami-2757f631"

instance\_type = "t2.micro"

+ vpc\_security\_group\_ids = ["sg-0077..."]

+ subnet\_id = "subnet-923a..."

}

Save the changes to example.tf, and re-run terraform apply.