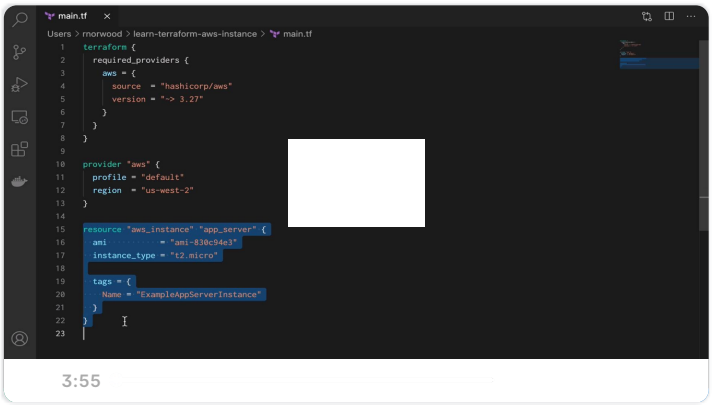
Build Infrastructure

11min |

Reference this often? Create an account to bookmark tutorials.



With Terraform installed, you are ready to create your first infrastructure.

In this tutorial, you will provision an EC2 instance on Amazon Web Services (AWS). EC2 instances are virtual machines running on AWS, and a common component of many infrastructure projects.

Prerequisites

To follow this tutorial you will need:

- The Terraform CLI (1.2.0+) installed.
- The AWS CLI installed.
- AWS account and associated credentials that allow you to create resources.

To use your IAM credentials to authenticate the Terraform AWS provider, set the `AWS_ACCESS_KEY_ID` environment variable.

```
$ export AWS_ACCESS_KEY_ID=
```

Now, set your secret key.

```
$ export AWS_SECRET_ACCESS_KEY=
```

Tip

If you don't have access to IAM user credentials, use another authentication method described in the AWS provider documentation.

This tutorial will provision resources that qualify under the AWS free tier. If your account does not qualify for free tier resources, we are not responsible for any charges that you may incur.

Write configuration

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

Each Terraform configuration must be in its own working directory. Create a directory for your configuration.

```
$ mkdir learn-terraform-aws-instance
```

Change into the directory.

```
$ cd learn-terraform-aws-instance
```

Create a file to define your infrastructure.

```
$ touch main.tf
```

Open `main.tf` in your text editor, paste in the configuration below, and save the file.

> **Tip**
>
> The AMI ID used in this configuration is specific to the `us-west-2` region. If you would like to use a different region, see the Troubleshooting section for guidance.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }

  required_version = ">= 1.2.0"
}

provider "aws" {
  region  = "us-west-2"
}

resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"

  tags = {
    Name = "ExampleAppServerInstance"
  }
}
```

This is a complete configuration that you can deploy with Terraform. The following sections review each block of this configuration in more detail.

AWS

The `terraform {}` block contains Terraform settings, including the required providers

Terraform will use to provision your infrastructure. For each provider, the `source` attribute defines an optional hostname, a namespace, and the provider type. Terraform installs providers from the Terraform Registry by default. In this example configuration, the `aws` provider's source is defined as `hashicorp/aws`, which is shorthand for `registry.terraform.io/hashicorp/aws`.

You can also set a version constraint for each provider defined in the `required_providers` block. The `version` attribute is optional, but we recommend using it to constrain the provider version so that Terraform does not install a version of the provider that does not work with your configuration. If you do not specify a provider version, Terraform will automatically download the most recent version during initialization.

To learn more, reference the provider source documentation.

Providers

The `provider` block configures the specified provider, in this case `aws`. A provider is a plugin that Terraform uses to create and manage your resources.

You can use multiple provider blocks in your Terraform configuration to manage resources from different providers. You can even use different providers together. For example,

you could pass the IP address of your AWS EC2 instance to a monitoring resource from DataDog.

Resources

Use `resource` blocks to define components of your infrastructure. A resource might be a physical or virtual component such as an EC2 instance, or it can be a logical resource such as a Heroku application.

Resource blocks have two strings before the block: the resource type and the resource name. In this example, the resource type is `aws_instance` and the name is `app_server`. The prefix of the type maps to the name of the provider. In the example configuration, Terraform manages the `aws_instance` resource with the `aws` provider. Together, the resource type and resource name form a unique ID for the resource. For example, the ID for your EC2 instance is `aws_instance.app_server`.

Resource blocks contain arguments which you use to configure the resource. Arguments can include things like machine sizes, disk image names, or VPC IDs. Our providers reference lists the required and optional arguments for each resource. For your EC2 instance, the example configuration sets the AMI ID to an Ubuntu image, and the instance type to `t2.micro`, which qualifies for AWS' free tier. It also sets a tag to give

the instance a name.

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`.

Initializing a configuration directory downloads and installs the providers defined in the configuration, which in this case is the `aws` provider.

Initialize the directory.

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions match
- Installing hashicorp/aws v4.17.0...
- Installed hashicorp/aws v4.17.0 (sig

Terraform has created a lock file .ter
selections it made above. Include this
so that Terraform can guarantee to mak
you run "terraform init" in the future

Terraform has been successfully initia

You may now begin working with Terrafo
any changes that are required for your
should now work.

If you ever set or change modules or b
rerun this command to reinitialize you
commands will detect it and remind you
```

Terraform downloads the `aws` provider and installs it in a hidden subdirectory of your current working directory, named `.terraform`. The `terraform init` command prints out which version of the provider was installed. Terraform also creates a lock file named `.terraform.lock.hcl` which specifies the exact provider versions used, so that you can control when you want

to update the providers used for your project.

Format and validate the configuration

We recommend using consistent formatting in all of your configuration files. The `terraform fmt` command automatically updates configurations in the current directory for readability and consistency.

Format your configuration. Terraform will print out the names of the files it modified, if any. In this case, your configuration file was already formatted correctly, so Terraform won't return any file names.

```
$ terraform fmt
```

You can also make sure your configuration is syntactically valid and internally consistent by using the `terraform validate` command.

Validate your configuration. The example configuration provided above is valid, so Terraform will return a success message.

```
$ terraform validate
Success! The configuration is valid.
```

Create infrastructure

Apply the configuration now with the `terraform apply` command. Terraform will print output similar to what is shown below. We have truncated some of the output to save space.

```
$ terraform apply

Terraform used the selected providers
Resource actions are indicated with th
  + create

Terraform will perform the following a

  # aws_instance.app_server will be cr
  + resource "aws_instance" "app_serve
      + ami                          =
      + arn                          =
##...

Plan: 1 to add, 0 to change, 0 to dest

Do you want to perform these actions?
  Terraform will perform the actions d
  Only 'yes' will be accepted to appro

  Enter a value:
```

> **Tip**
>
> If your configuration fails to apply, you may have customized your region or removed your default VPC. Refer to the troubleshooting section of this tutorial for help.

Before it applies any changes, Terraform prints out the *execution plan* which describes the actions Terraform will take in order to change your 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.app_server`, meaning that Terraform will create this resource. Beneath that, it shows the attributes that will be set. When the value displayed is `(known after apply)`, it means that the value will not be known until the resource is created. For example, AWS assigns Amazon Resource Names (ARNs) to instances upon creation, so Terraform cannot know the value of the `arn` attribute until you apply the change and the AWS provider returns that value from the AWS API.

Terraform will now pause and wait for your approval before proceeding. If anything in the plan seems incorrect or dangerous, it is safe to abort here before Terraform modifies your infrastructure.

In this case the plan is 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.

```
  Enter a value: yes

aws_instance.app_server: Creating...
aws_instance.app_server: Still creatir
aws_instance.app_server: Still creatir
aws_instance.app_server: Still creatir
aws_instance.app_server: Creation comp

Apply complete! Resources: 1 added, 0
```

You have now created infrastructure using Terraform! Visit the EC2 console and find your new EC2 instance.

Note

Per the `aws` provider block, your instance was created in the `us-west-2` region. Ensure that your AWS Console is set to this region.

Inspect state

When you applied your configuration, Terraform wrote data into a file called `terraform.tfstate`. Terraform stores the IDs and properties of the resources it manages in this file, so that it can update or destroy those resources going forward.

The Terraform state file is the only way Terraform can track which resources it manages, and often contains sensitive information, so you must store your state file securely and restrict access to only trusted team members who need to manage your infrastructure. In production, we recommend storing your state remotely with Terraform Cloud or Terraform Enterprise. Terraform also supports several other remote backends you can use to store and manage your state.

Inspect the current state using `terraform show`.

```
$ terraform show
# aws_instance.app_server:
resource "aws_instance" "app_server" {
    ami                          = "am
    arn                          = "ar
    associate_public_ip_address  = tru
    availability_zone            = "us
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
    disable_api_termination      = fal
    ebs_optimized                = fal
    get_password_data            = fal
    hibernation                  = fal
    id                           = "i-
    instance_state               = "ru
    instance_type                = "t2
    ipv6_address_count           = 0
    ipv6_addresses               = []
    monitoring                   = fal
    primary_network_interface_id = "er
```

```
private_dns                   = "ip
private_ip                    = "17
public_dns                    = "ec
public_ip                     = "18
secondary_private_ips         = []
security_groups               = [
    "default",
]
source_dest_check             = tru
subnet_id                     = "su
tags                          = {
    "Name" = "ExampleAppServerInst
}
tenancy                       = "de
vpc_security_group_ids        = [
    "sg-0edc8a5a",
]

credit_specification {
    cpu_credits = "standard"
}

enclave_options {
    enabled = false
}

metadata_options {
    http_endpoint               =
    http_put_response_hop_limit =
    http_tokens                 =
}

root_block_device {
    delete_on_termination = true
    device_name           = "/dev/
```

```
        encrypted             = false
        iops                  = 0
        tags                  = {}
        throughput            = 0
        volume_id             = "vol-0
        volume_size           = 8
        volume_type           = "stand
    }
}
```

When Terraform created this EC2 instance, it also gathered the resource's metadata from the AWS provider and wrote the metadata to the state file. In later tutorials, you will modify your configuration to reference these values to configure other resources and output values.

Manually Managing State

Terraform has a built-in command called `terraform state` for advanced state management. Use the `list` subcommand to list of the resources in your project's state.

```
$ terraform state list
aws_instance.app_server
```

Troubleshooting

If `terraform validate` was successful and your apply still failed, you may be

encountering one of these common errors.

- **If you use a region other than `us-west-2`**, 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 `main.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`) arguments to your `aws_instance` resource, and replace the values with the ones from your new security group and subnet.

```
 resource "aws_instance" "app_serve
   ami                    = "ami-83
   instance_type          = "t2.mic
+  vpc_security_group_ids = ["sg-00
+  subnet_id              = "subnet
 }
```

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

Remember to add these lines to your

configuration for later tutorials. For more information, review this document from AWS on working with VPCs.

Next Steps

Now that you have created your first infrastructure using Terraform, continue to the next tutorial to modify your infrastructure.

For more detail on the concepts used in this tutorial:

- Read about the Terraform configuration language in the Terraform documentation.
- Learn more about Terraform providers.
- Find examples of other uses for Terraform in the documentation use cases section.
- Read the AWS provider documentation to learn more about AWS authentication.
- For more information about the `terraform state` command and subcommands for moving or removing resources from state, see the CLI `state` command documentation.

Was this tutorial helpful? Yes No