```
$ export AWS_ACCESS_KEY_ID=(your access key id)
$ export AWS_SECRET_ACCESS_KEY=(your secret access key)
```

And here is how you can do it in a Windows command terminal:

```
$ set AWS_ACCESS_KEY_ID=(your access key id)
$ set AWS_SECRET_ACCESS_KEY=(your secret access key)
```

Note that these environment variables apply only to the current shell, so if you reboot your computer or open a new terminal window, you'll need to export these variables again.

### OTHER AWS AUTHENTICATION OPTIONS

In addition to environment variables, Terraform supports the same authentication mechanisms as all AWS CLI and SDK tools. Therefore, it'll also be able to use credentials in *$HOME/.aws/credentials*, which are automatically generated if you run `aws configure`, or IAM roles, which you can add to almost any resource in AWS. For more info, see A Comprehensive Guide to Authenticating to AWS on the Command Line. You'll also see more information on authenticating to Terraform providers in Chapter 6.

## Deploying a Single Server

Terraform code is written in the *HashiCorp Configuration Language* (HCL) in files with the extension *.tf*.[7] It is a declarative language, so your goal is to describe the infrastructure you want, and Terraform will figure out how to create it. Terraform can create infrastructure across a wide variety of platforms, or what it calls *providers*, including AWS, Azure, Google Cloud, DigitalOcean, and many others.

You can write Terraform code in just about any text editor. If you search around, you can find Terraform syntax highlighting support for most editors (note that you may have to search for the word *HCL* instead of *Terraform*), including vim, emacs, Sublime Text, Atom, Visual Studio Code, and IntelliJ (the latter even has support for refactoring, find usages, and go to declaration).

The first step to using Terraform is typically to configure the provider(s) you want to use. Create an empty folder and put a file in it called *main.tf* that contains the following contents:

```
provider "aws" {
  region = "us-east-2"
}
```

This tells Terraform that you are going to be using AWS as your provider and that you want to deploy your infrastructure into the `us-east-2` region. AWS has datacenters all over the world, grouped into regions. An *AWS region* is a separate geographic area, such as `us-east-2` (Ohio), `eu-west-1` (Ireland), and `ap-southeast-2` (Sydney). Within each region, there are multiple isolated datacenters known as *Availability Zones* (AZs), such as `us-east-2a`, `us-east-2b`, and so on.[8] There are many other settings you can configure on this provider, but for now, let's keep it simple, and we'll take a deeper look at provider configuration in Chapter 7.

For each type of provider, there are many different kinds of *resources* that you can create, such as servers, databases, and load balancers. The general syntax for creating a resource in Terraform is as follows:

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG ...]
}
```

where `PROVIDER` is the name of a provider (e.g., `aws`), `TYPE` is the type of resource to create in that provider (e.g., `instance`), `NAME` is an identifier you can use throughout the Terraform code to refer to this resource (e.g., `my_instance`), and `CONFIG` consists of one or more *arguments* that are specific to that resource.

For example, to deploy a single (virtual) server in AWS, known as an *EC2 Instance*, use the `aws_instance` resource in *main.tf* as follows:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
```

```
  instance_type = "t2.micro"
}
```

The `aws_instance` resource supports many different arguments, but for now, you only need to set the two required ones:

*`ami`*

The Amazon Machine Image (AMI) to run on the EC2 Instance. You can find free and paid AMIs in the AWS Marketplace or create your own using tools such as Packer. The preceding code sets the `ami` parameter to the ID of an Ubuntu 20.04 AMI in `us-east-2`. This AMI is free to use. Please note that AMI IDs are different in every AWS region, so if you change the `region` parameter to something other than `us-east-2`, you'll need to manually look up the corresponding Ubuntu AMI ID for that region,[9] and copy it into the `ami` parameter. In Chapter 7, you'll see how to fetch the AMI ID completely automatically.

*`instance_type`*

The type of EC2 Instance to run. Each type of EC2 Instance provides a different amount of CPU, memory, disk space, and networking capacity. The EC2 Instance Types page lists all the available options. The preceding example uses `t2.micro`, which has one virtual CPU, 1 GB of memory, and is part of the AWS Free Tier.

**USE THE DOCS!**

Terraform supports dozens of providers, each of which supports dozens of resources, and each resource has dozens of arguments. There is no way to remember them all. When you're writing Terraform code, you should be regularly referring to the Terraform documentation to look up what resources are available and how to use each one. For example, here's the documentation for the `aws_instance` resource. I've been using Terraform for years, and I still refer to these docs multiple times per day!

In a terminal, go into the folder where you created *main.tf* and run the `terraform init` command:

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency
lock file
- Using hashicorp/aws v4.19.0 from the shared cache directory

Terraform has been successfully initialized!
```

The `terraform` binary contains the basic functionality for Terraform, but it does not come with the code for any of the providers (e.g., the AWS Provider, Azure provider, GCP provider, etc.), so when you're first starting to use Terraform, you need to run `terraform init` to tell Terraform to scan the code, figure out which providers you're using, and download the code for them. By default, the provider code will be downloaded into a *.terraform* folder, which is Terraform's scratch directory (you may want to add it to *.gitignore*). Terraform will also record information about the provider code it downloaded into a *.terraform.lock.hcl* file (you'll learn more about this file in "Versioned Modules"). You'll see a few other uses for the `init` command and *.terraform* folder in later chapters. For now, just be aware that you need to run `init` anytime you start with new Terraform code and that it's safe to run `init` multiple times (the command is idempotent).

Now that you have the provider code downloaded, run the `terraform plan` command:

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_instance.example will be created
```

```
  + resource "aws_instance" "example" {
      + ami                           = "ami-0fb653ca2d3203ac1"
      + 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)
      + key_name                      = (known after apply)
      (...)
  }

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

The `plan` command lets you see what Terraform will do before actually making any changes. This is a great way to sanity-check your code before unleashing it onto the world. The output of the `plan` command is similar to the output of the `diff` command that is part of Unix, Linux, and `git`: anything with a plus sign (+) will be created, anything with a minus sign (–) will be deleted, and anything with a tilde sign (~) will be modified in place. In the preceding output, you can see that Terraform is planning on creating a single EC2 Instance and nothing else, which is exactly what you want.

To actually create the Instance, run the `terraform apply` command:

```
$ terraform apply

(...)

Terraform will perform the following actions:

  # aws_instance.example will be created
  + resource "aws_instance" "example" {
      + ami                           = "ami-0fb653ca2d3203ac1"
      + 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)
      + key_name                    = (known after apply)
      (...)
  }

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

You'll notice that the `apply` command shows you the same `plan` output and asks you to confirm whether you actually want to proceed with this plan. So, while `plan` is available as a separate command, it's mainly useful for quick sanity checks and during code reviews (a topic you'll see more of in Chapter 10), and most of the time you'll run `apply` directly and review the plan output it shows you.

Type **`yes`** and hit Enter to deploy the EC2 Instance:

```
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Still creating... [30s elapsed]
aws_instance.example: Creation complete after 38s [id=i-
07e2a3e006d785906]

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

Congrats, you've just deployed an EC2 Instance in your AWS account using Terraform! To verify this, head over to the EC2 console, and you should see something similar to Figure 2-4.

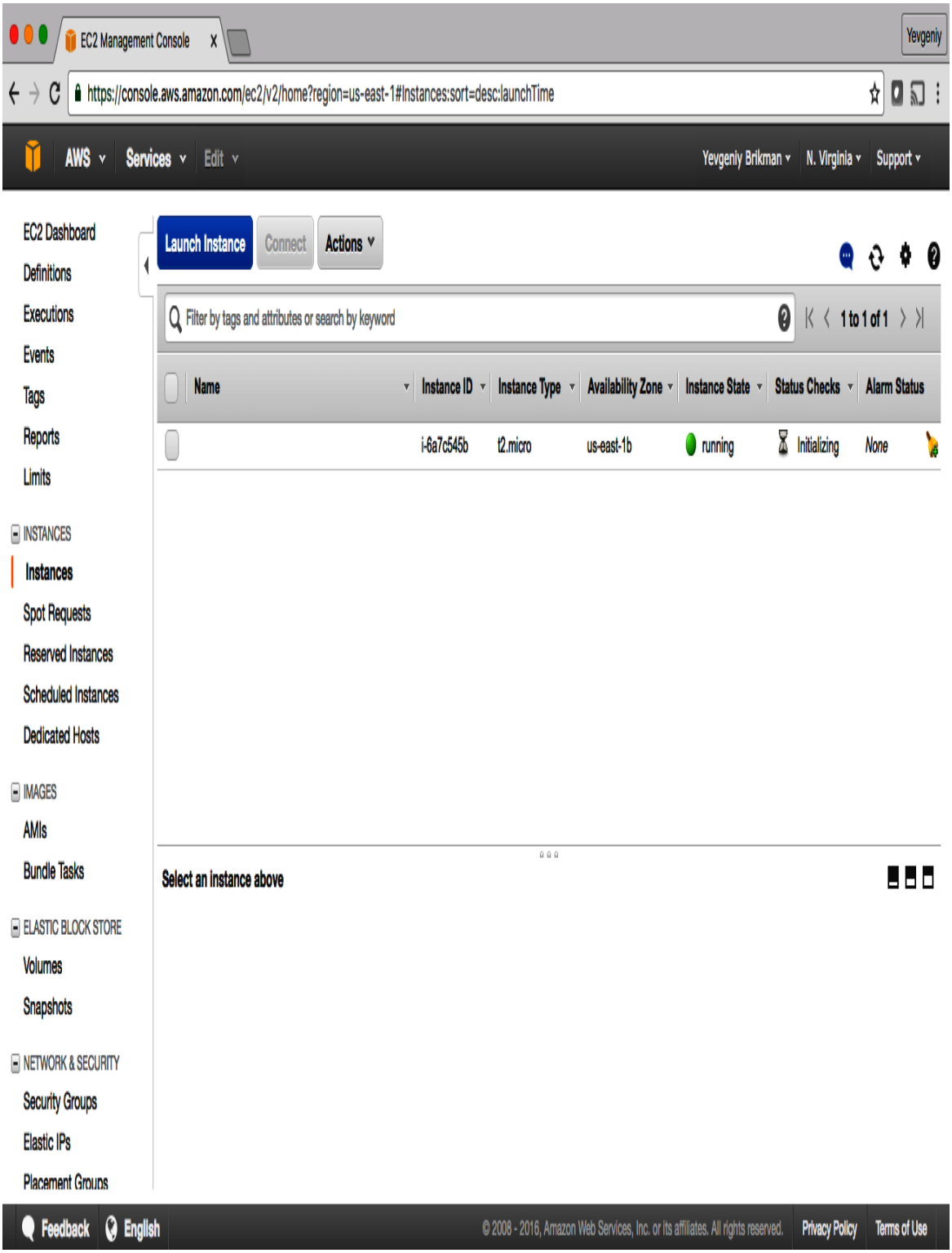


*Figure 2-4. The AWS Console shows the EC2 Instance you deployed.*

Sure enough, the Instance is there, though admittedly, this isn't the most exciting example. Let's make it a bit more interesting. First, notice that the

EC2 Instance doesn't have a name. To add one, you can add `tags` to the `aws_instance` resource:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"

  tags = {
    Name = "terraform-example"
  }
}
```

Run `terraform apply` again to see what this would do:

```
$ terraform apply

aws_instance.example: Refreshing state...
(...)

Terraform will perform the following actions:

  # aws_instance.example will be updated in-place
  ~ resource "aws_instance" "example" {
        ami                          = "ami-0fb653ca2d3203ac1"
        availability_zone            = "us-east-2b"
        instance_state               = "running"
        (...)
      + tags                         = {
          + "Name" = "terraform-example"
        }
        (...)
    }

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

Terraform keeps track of all the resources it already created for this set of configuration files, so it knows your EC2 Instance already exists (notice

Terraform says `Refreshing state...` when you run the `apply` command), and it can show you a diff between what's currently deployed and what's in your Terraform code (this is one of the advantages of using a declarative language over a procedural one, as discussed in "How Does Terraform Compare to Other IaC Tools?"). The preceding diff shows that Terraform wants to create a single tag called `Name`, which is exactly what you need, so type **`yes`** and hit Enter.

When you refresh your EC2 console, you'll see something similar to Figure 2-5.

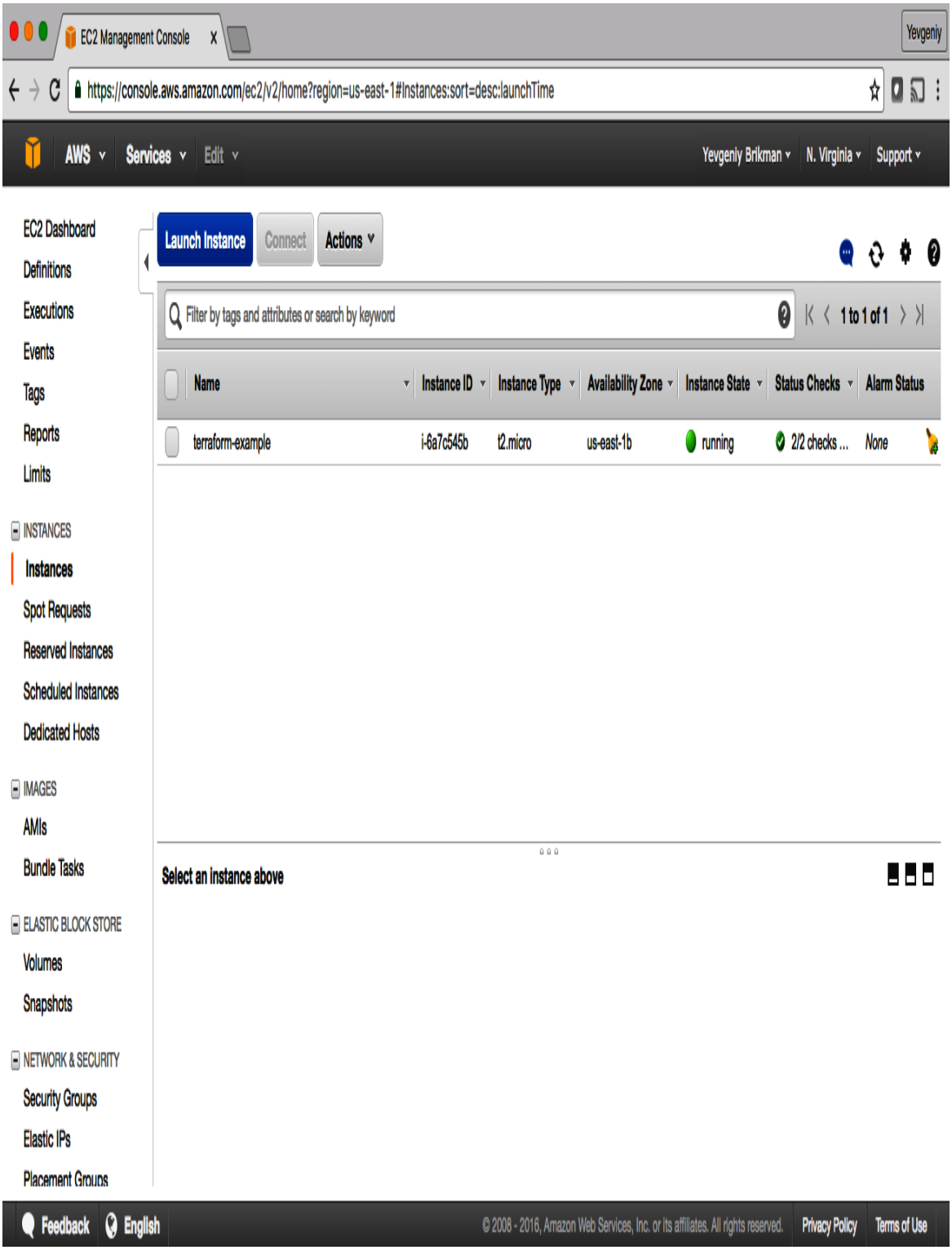


*Figure 2-5. The EC2 Instance now has a name tag.*

Now that you have some working Terraform code, you may want to store it in version control. This allows you to share your code with other team

members, track the history of all infrastructure changes, and use the commit log for debugging. For example, here is how you can create a local Git repository and use it to store your Terraform configuration file and the lock file (you'll learn all about the lock file in Chapter 8; for now, all you need to know is it should be added to version control along with your code):

```
git init
git add main.tf .terraform.lock.hcl
git commit -m "Initial commit"
```

You should also create a *.gitignore* file with the following contents:

```
.terraform
*.tfstate
*.tfstate.backup
```

The preceding *.gitignore* file instructs Git to ignore the *.terraform* folder, which Terraform uses as a temporary scratch directory, as well as **.tfstate* files, which Terraform uses to store state (in Chapter 3, you'll see why state files shouldn't be checked in). You should commit the *.gitignore* file, too:

```
git add .gitignore
git commit -m "Add a .gitignore file"
```

To share this code with your teammates, you'll want to create a shared Git repository that you can all access. One way to do this is to use GitHub. Head over to GitHub, create an account if you don't have one already, and create a new repository. Configure your local Git repository to use the new GitHub repository as a remote endpoint named `origin`, as follows:

```
git remote add origin git@github.com:
<YOUR_USERNAME>/<YOUR_REPO_NAME>.git
```

Now, whenever you want to share your commits with your teammates, you can *push* them to `origin`:

```
git push origin main
```