

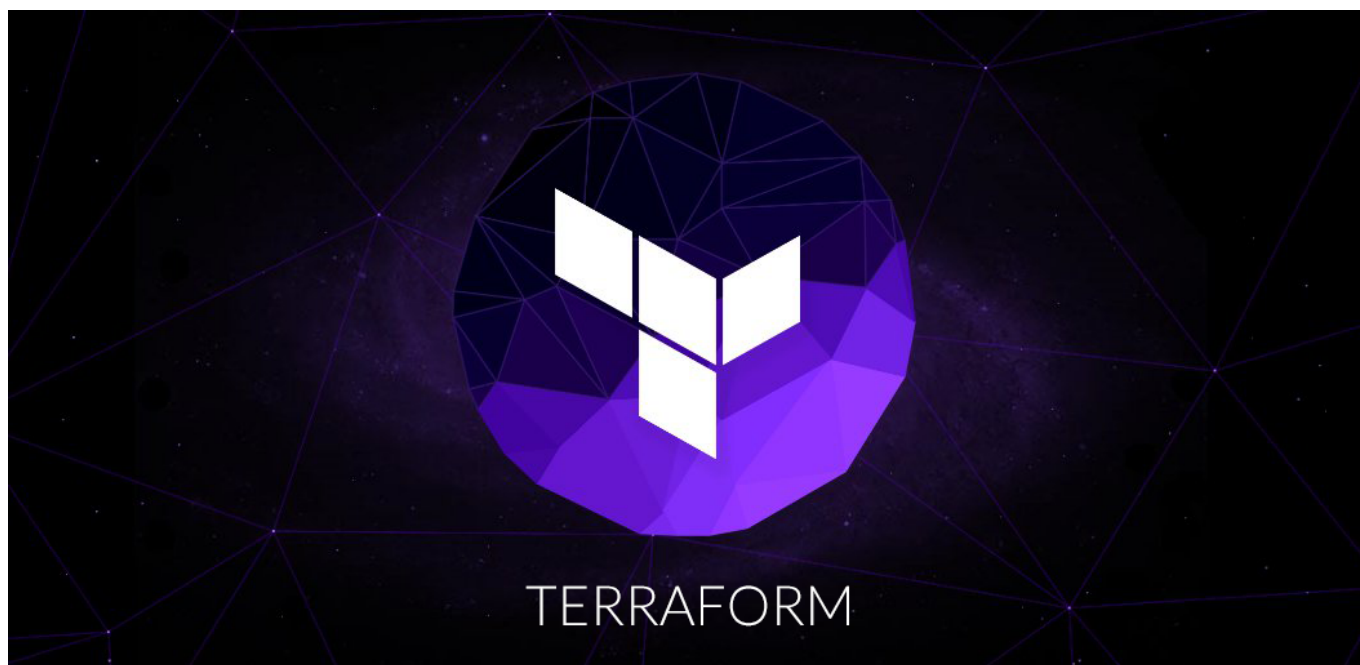
An Introduction to Terraform

Learn the basics of Terraform in this step-by-step tutorial of how to deploy a cluster of web servers and a load balancer on AWS



Yevgeniy Brikman [Follow](#)

Sep 29, 2016 · 29 min read



***Update, November 17, 2016:** We took this blog post series, expanded it, and turned it into a book called *Terraform: Up & Running!**

***Update, July 8, 2019:** We've updated this blog post series for Terraform 0.12 and released the 2nd edition of *Terraform: Up & Running!**

This is Part 2 of the Comprehensive Guide to Terraform series. In Part 1, we explained why we picked Terraform as our IAC tool of choice and not Chef, Puppet, Ansible, SaltStack, or CloudFormation. In this post, we're going to introduce the basics of how to use Terraform to define and manage your infrastructure.

The official Terraform Getting Started documentation does a good job of introducing the individual elements of Terraform (i.e. resources, input variables, output variables, etc), so in this guide, we're going to focus on how to put those elements together to create a fairly real-world example. In particular, we will provision several servers on AWS in a cluster and deploy a load balancer to distribute load across that cluster. The infrastructure you'll create in this example is a basic starting point for running scalable, highly-available web services and microservices.

This guide is targeted at AWS and Terraform newbies, so don't worry if you haven't used either one before. We'll walk you through the entire process, step-by-step:

1. Set up your AWS account
2. Install Terraform
3. Deploy a single server
4. Deploy a single web server
5. Deploy a configurable web server
6. Deploy a cluster of web servers
7. Deploy a load balancer
8. Clean up

You can find sample code for the examples below at: <https://github.com/gruntwork-io/intro-to-terraform>. Note that all the code samples are written for Terraform 0.12.x.

Set up your AWS account

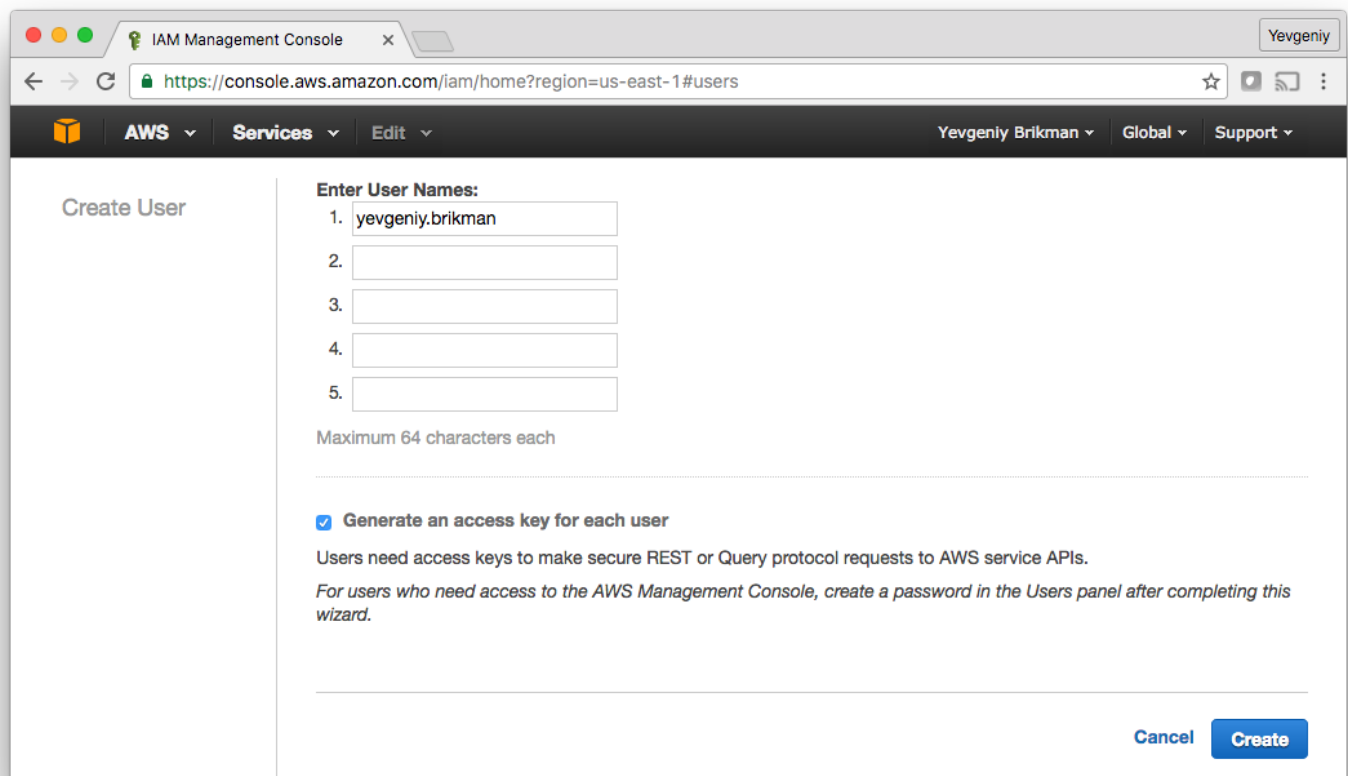
Terraform can provision infrastructure across many different types of cloud providers, including AWS, Azure, Google Cloud, DigitalOcean, and many others. For this tutorial, we picked Amazon Web Services (AWS) because:

- It provides a huge range of reliable and scalable cloud hosting services, including Elastic Compute Cloud (EC2), Auto Scaling Groups (ASGs), and Elastic Load

Balancing (ELB). If you find the AWS terminology confusing, be sure to check out [AWS in Plain English](#).

- AWS is the most popular cloud infrastructure provider, by far.
- AWS offers a generous Free Tier which should allow you to run all of these examples for free.

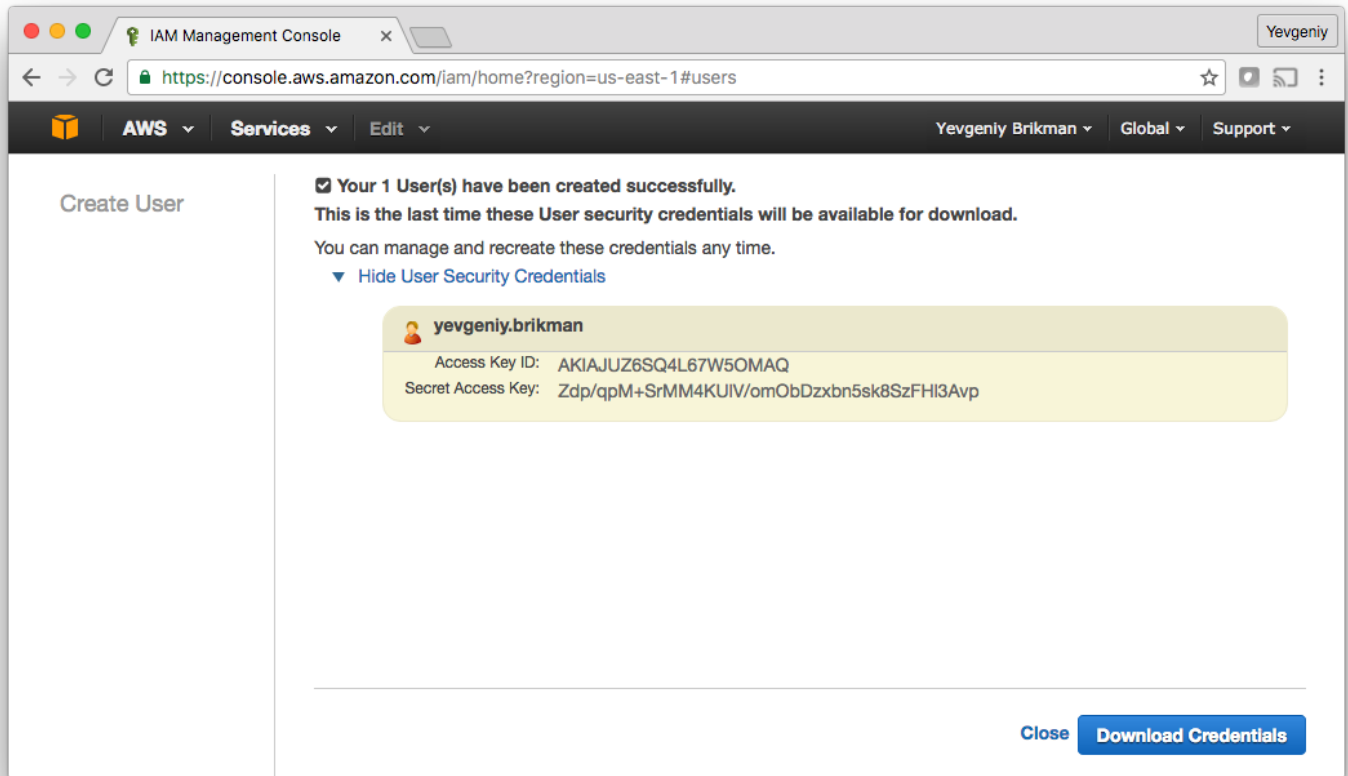
When you first register for AWS, you initially sign in as the root user. This user account has access permissions to everything, so from a security perspective, we recommend *only* using it to create other user accounts with more limited permissions (see IAM Best Practices). To create a more limited user account, head over to the Identity and Access Management (IAM) console, click “Users”, and click the blue “Create New Users” button. Enter a name for the user and make sure “Generate an access key for each user” is checked:



The screenshot shows the AWS IAM Management Console in a web browser. The browser's address bar displays the URL `https://console.aws.amazon.com/iam/home?region=us-east-1#users`. The page title is "IAM Management Console". The navigation bar includes the AWS logo, "AWS", "Services", "Edit", and user information "Yevgeniy Brikman", "Global", and "Support". The main content area is titled "Create User". It features a section "Enter User Names:" with five input fields. The first field contains the text "yevgeniy.brikman". Below the input fields, it states "Maximum 64 characters each". A checkbox labeled "Generate an access key for each user" is checked. Below this checkbox, there is explanatory text: "Users need access keys to make secure REST or Query protocol requests to AWS service APIs. For users who need access to the AWS Management Console, create a password in the Users panel after completing this wizard." At the bottom right of the form, there are two buttons: "Cancel" and "Create".

Note: the IAM user page may look a bit different when you try it, but the basic idea of creating an IAM User is the same.

Click the “Create” button and you’ll be able to see security credentials for that user, which consist of Access Key ID and a Secret Access Key. You **MUST** save these immediately, as they will never be shown again. We recommend storing them somewhere secure (e.g. a password manager such as Keychain or 1Password) so you can use them a little later in this tutorial.

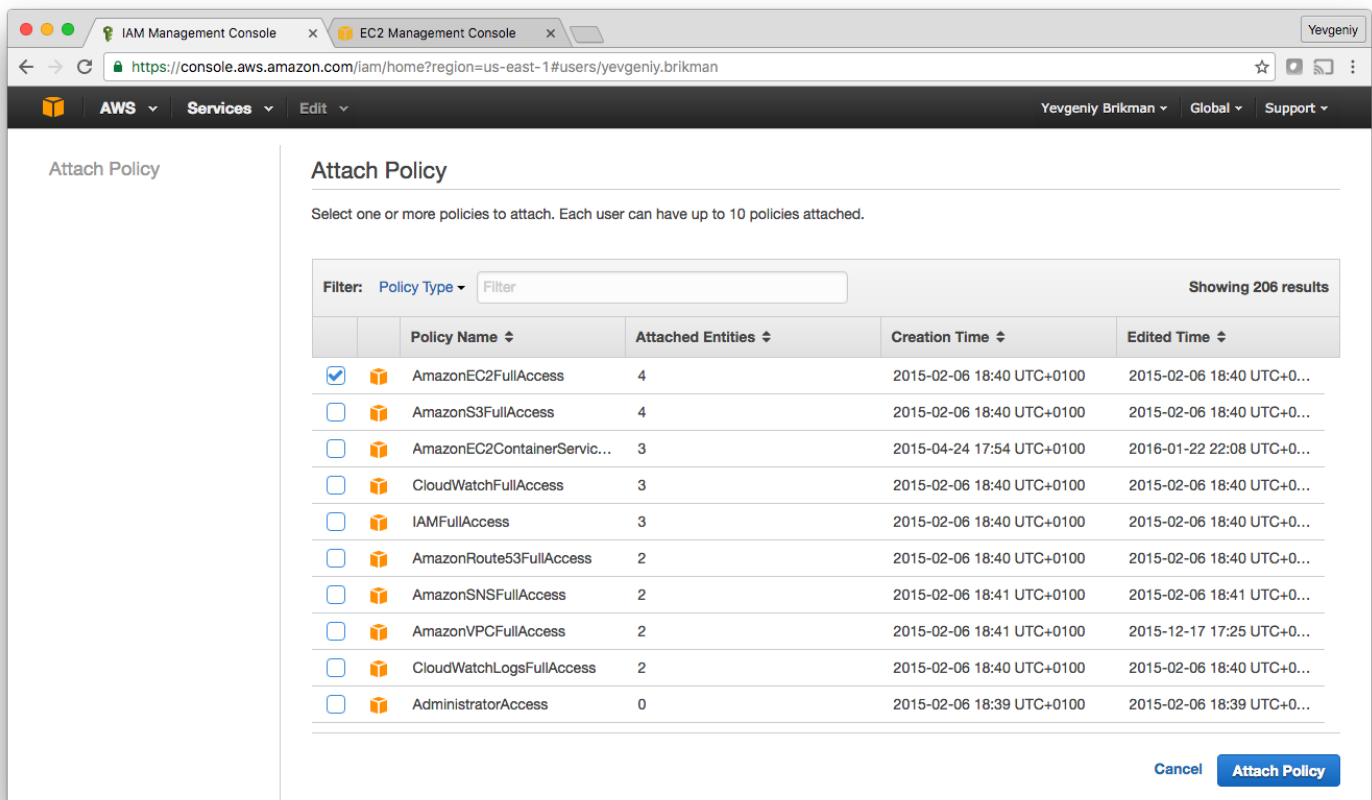


Save the credentials somewhere secure. Never share them with anyone. Don’t worry, the ones in the screenshot above are fake.

Once you’ve saved the credentials, click “Close” (twice) and you’ll be taken to the list of users. Click on the user you just created and select the “Permissions” tab. By default, a new IAM user does not have permissions to do anything in the AWS account. To be able to use Terraform for the examples in this blog post series, add the following permissions (learn more about Managed IAM Policies here):

- `AmazonEC2FullAccess` : required for this blog post.
- `AmazonS3FullAccess` : required for How to manage Terraform state.

- `AmazonDynamoDBFullAccess` : required for How to manage Terraform state.
- `AmazonRDSFullAccess` : required for How to create reusable infrastructure with Terraform modules.
- `CloudWatchFullAccess` : required for Terraform tips & tricks: loops, if-statements, and pitfalls.
- `IAMFullAccess` : required for Terraform tips & tricks: loops, if-statements, and pitfalls.



Install Terraform

Follow the instructions here to install Terraform. When you're done, you should be able to run the terraform command:

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

(...)

In order for Terraform to be able to make changes in your AWS account, you will need to configure the AWS credentials for the user you created earlier. There are several ways to do this (see [A Comprehensive Guide to Authenticating to AWS on the Command Line](#)), one of the easiest of which is to set the following environment variables:

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

Deploy a single server

Terraform code is written in a language called HCL in files with the extension `.tf`. 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, 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 a file called `main.tf` and put the following code in it:

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

This tells Terraform that you are going to be using the AWS provider and that you wish to deploy your infrastructure in the `us-east-2` region (AWS has data centers all over the world, grouped into regions and availability zones, and `us-east-2` is the name for data

centers in Ohio, USA). You can configure other settings for the AWS provider, but for this example, since you've already configured your credentials as environment variables, you only need to specify the region.

For each provider, there are many different kinds of *resources* you can create, such as servers, databases, and load balancers. Before we deploy a whole cluster of servers, let's first figure out how to deploy a single server that will run respond with "Hello, World" to HTTP requests. In AWS lingo, a server is called an *EC2 Instance*. Add the following code to `main.tf`, which uses the `aws_instance` resource to deploy an EC2 Instance:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfaffe1f0"
  instance_type = "t2.micro"
}
```

The general syntax for a Terraform resource is:

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

Where `PROVIDER` is the name of a provider (e.g., `aws`), `TYPE` is the type of resources 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., `example`), and `CONFIG` consists of one or more *arguments* that are specific to that resource (e.g., `ami = "ami-0c55b159cbfaffe1f0"`). For the `aws_instance` resource, there are many different arguments, but for now, you only need to set the following 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 a Ubuntu 18.04 AMI in `us-east-2`. This AMI is free to use.

- `instance_type` : The type of EC2 Instance to run. Each type of EC2 Instance provides a different amount CPU, memory, disk space, and networking capacity. The EC2 Instance Types page lists all the available options and how much each one costs. The preceding example uses `t2.micro`, which has one virtual CPU, 1GB of memory, and is part of the AWS free tier.

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...
- Checking for available provider plugins...
- Downloading plugin for provider "aws"

(...)

* provider.aws: version = "~> 2.10"

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 first starting to use Terraform, you need to run `terraform init` to tell Terraform to scan the code, figure out what 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`). You'll see a few other uses for the `init` command and `.terraform` folder later on. For now, just be aware that you need to run `init` any time 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
Refreshing Terraform state in-memory prior to plan...

(...)

+ aws_instance.example
  ami: "ami-2d39803a"
  availability_zone: "<computed>"
  ebs_block_device.#: "<computed>"
  ephemeral_block_device.#: "<computed>"
  instance_state: "<computed>"
  instance_type: "t2.micro"
  key_name: "<computed>"
  network_interface_id: "<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>"
```

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

The `plan` command lets you see what Terraform will do before actually doing it. This is a great way to sanity check your changes before unleashing them onto the world. The output of the `plan` command is a little like the output of the `diff` command: resources with a plus sign (`+`) are going to be created, resources with a minus sign (`-`) are going to be deleted, and resources with a tilde sign (`~`) are going to be modified in-place. In the output above, you can see that Terraform is planning on creating a single EC2 Instance and nothing else, which is exactly what we 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-0c55b159cbfafa1f0"
  + 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 if 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, and most of the time you'll run `apply` directly and review the plan output it shows you.

Type in "yes" and hit enter to deploy the EC2 Instance:

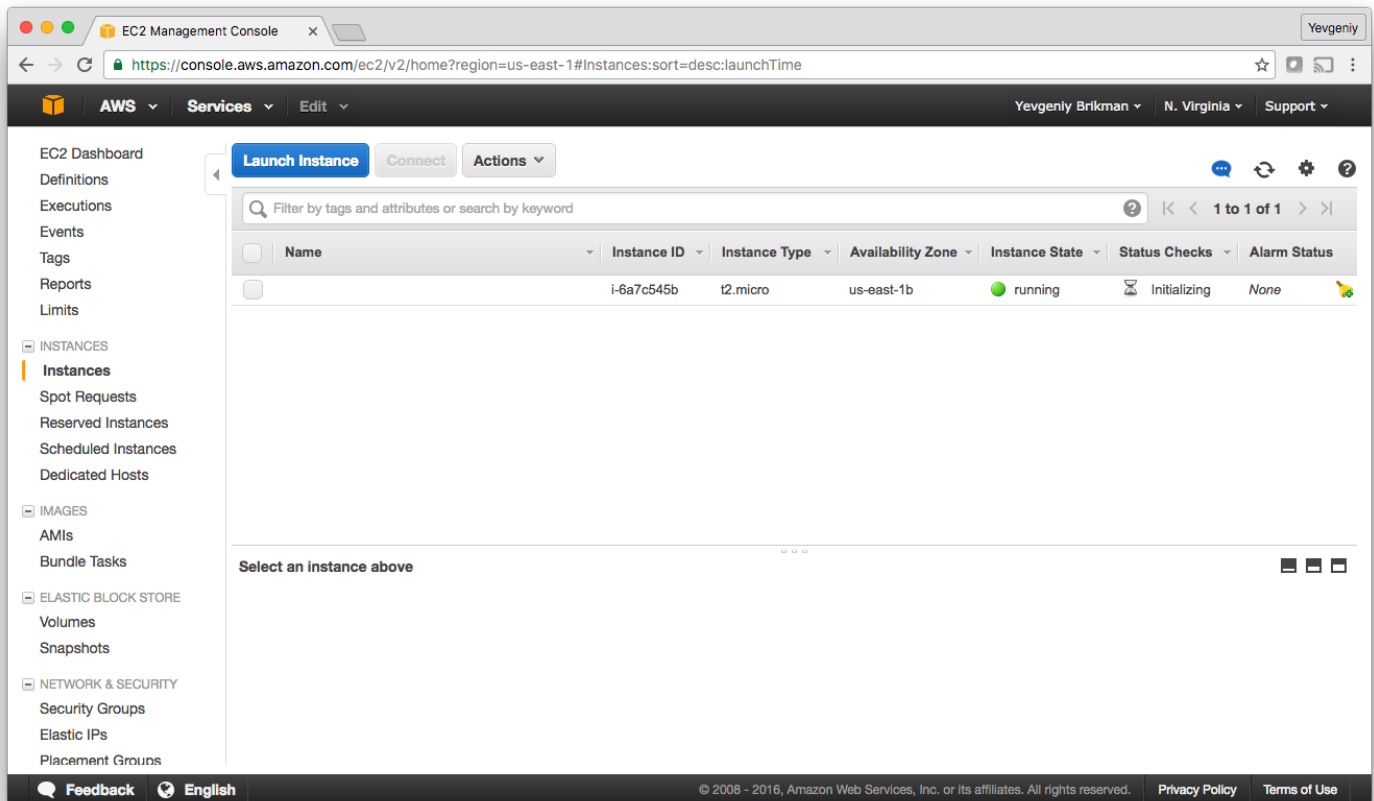
Do you want to perform these actions?
 Terraform will perform the actions described above.
 Only 'yes' will be accepted to approve.

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
```

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

Congrats, you've just deployed a server with Terraform! To verify this, you can login to the EC2 console, and you'll see something like this:



It's working, but it's not the most exciting example. For one thing, the Instance doesn't have a name. To add one, you can add a tag to the EC2 instance:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafa1f0"
  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-0c55b159cbfaffe1f0"
    availability_zone         = "us-east-2b"
    instance_state            = "running"
    (...)

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

    (...)
}
```

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

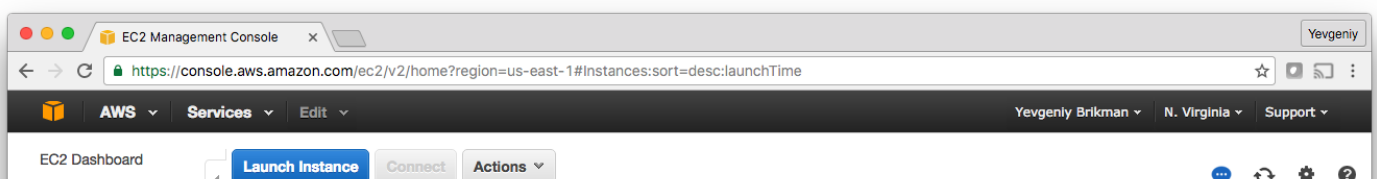
```
Do you want to perform these actions?
```

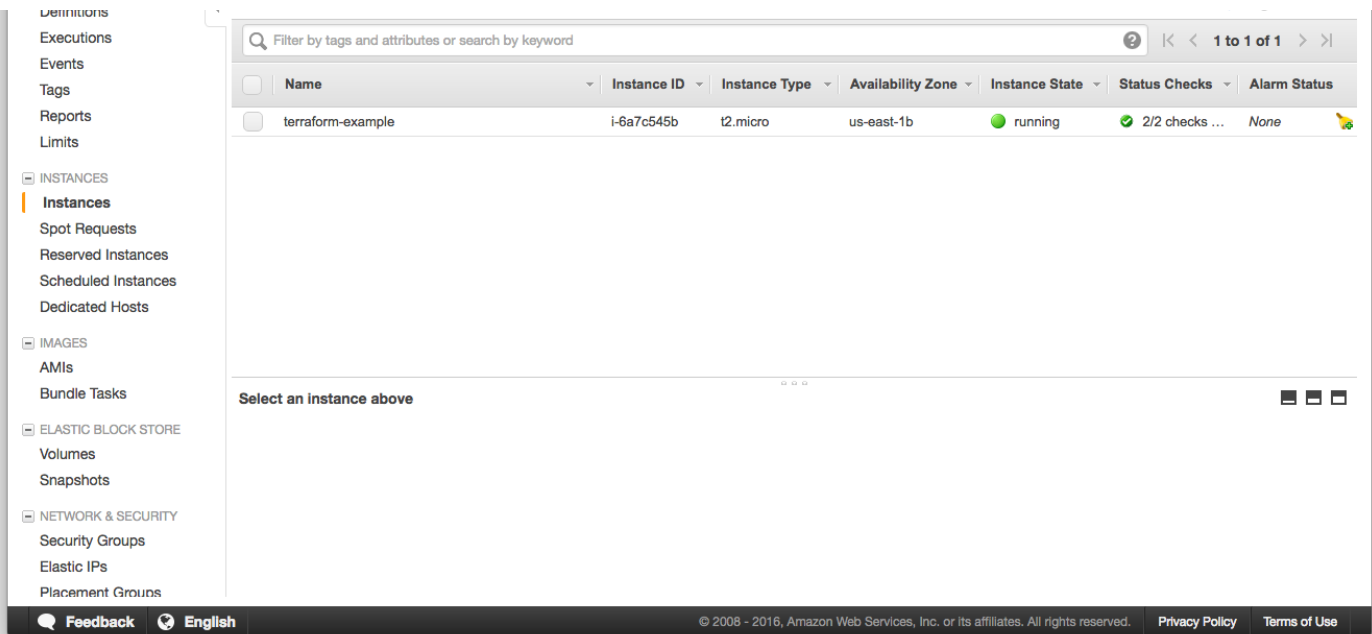
```
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
```

```
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). The preceding diff shows that Terraform wants to create a single tag called “Name,” which is exactly what you need, so type in “yes” and hit enter.

When you refresh your EC2 console, you’ll see:





Deploy a single web server

The next step is to run a web server on this Instance. In a real-world use case, you'd probably install a full-featured web framework like Ruby on Rails or Django, but to keep this example simple, we're going to run a dirt-simple web server that always returns the text "Hello, World" using a code borrowed from the big list of http static server one-liners:

```
#!/bin/bash
echo "Hello, World" > index.html
nohup busybox httpd -f -p 8080 &
```

This is a bash script that writes the text "Hello, World" into `index.html` and runs a web server on port 8080 using busybox (which is installed by default on Ubuntu) to serve that file at the URL `/`. We wrap the busybox command with `nohup` to ensure the web server keeps running even after this script exits and put an `&` at the end of the command so the web server runs in a background process and the script can exit rather than being blocked forever by the web server.

How do you get the EC2 Instance to run this script? Normally, instead of using an empty Ubuntu AMI, you would use a tool like Packer to create a custom AMI that has the web

server installed on it. But again, in the interest of keeping this example simple, we're going to run the script above as part of the EC2 Instance's User Data, which AWS will execute when the instance is booting:

```
resource "aws_instance" "example" {
  ami          = "ami-0c55b159cbfafa1f0"
  instance_type = "t2.micro"

  user_data = <<-EOF
    #!/bin/bash
    echo "Hello, World" > index.html
    nohup busybox httpd -f -p 8080 &
  EOF

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

The `<<-EOF` and `EOF` are Terraform's heredoc syntax, which allows you to create multiline strings without having to put `\n` all over the place (learn more about Terraform syntax [here](#)).

You need to do one more thing before this web server works. By default, AWS does not allow any incoming or outgoing traffic from an EC2 Instance. To allow the EC2 Instance to receive traffic on port 8080, you need to create a security group:

```
resource "aws_security_group" "instance" {
  name = "terraform-example-instance"

  ingress {
    from_port = 8080
    to_port   = 8080
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

This code creates a new resource called `aws_security_group` (notice how all resources for the AWS provider start with `aws_`) and specifies that this group allows incoming TCP

requests on port 8080 from the CIDR block `0.0.0.0/0`.

CIDR blocks are a concise way to specify IP address ranges. For example, a CIDR block of `10.0.0.0/24` represents all IP addresses between `10.0.0.0` and `10.0.0.255`. The CIDR block `0.0.0.0/0` is an IP address range that includes all possible IP addresses, so this security group allows incoming requests on port 8080 from any IP. For a handy calculator that converts between IP address ranges and CIDR notation, see <http://www.ipaddressguide.com/cidr>.

Simply creating a security group isn't enough; you also need to tell the EC2 Instance to actually use it by passing the ID of the security group into the `vpc_security_group_ids` argument of the `aws_instance` resource. To do that, you first need to learn about Terraform *expressions*.

An expression in Terraform is anything that returns a value. You've already seen the simplest type of expressions, *literals*, such as strings (e.g., `"ami-0c55b159cbfafa1f0"`) and numbers (e.g., `5`). Terraform supports many other types of expressions that you'll see throughout this blog post series.

One particularly useful type of expression is a *reference*, which allows you to access values from other parts of your code. To access the ID of the security group resource, you are going to need to use a *resource attribute reference*, which uses the following syntax:

```
<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>
```

Where `PROVIDER` is the name of the provider (e.g., `aws`), `TYPE` is the type of resource (e.g., `security_group`), `NAME` is the name of that resource (e.g., the security group is named `"instance"`), and `ATTRIBUTE` is either one of the arguments of that resource (e.g., `name`) or one of the attributes *exported* by the resource (you can find the list of available attributes in the documentation for each resource—e.g., here are the attributes for `aws_security_group`). The security group exports an attribute called `id`, so the expression to reference it will look like this:

```
aws_security_group.instance.id
```

You can use this security group ID in the `vpc_security_group_ids` parameter of the

`aws_instance`:

```
resource "aws_instance" "example" {
  ami                = "ami-0c55b159cbfafa1f0"
  instance_type      = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]

  user_data = <<-EOF
    #!/bin/bash
    echo "Hello, World" > index.html
    nohup busybox httpd -f -p 8080 &
  EOF

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

When you add a reference from one resource to another, you create an *implicit dependency*. Terraform parses these dependencies, builds a dependency graph from them, and uses that to automatically figure out in what order it should create resources. For example, if you were to deploy this code from scratch, Terraform would know it needs to create the security group before the EC2 Instance, since the EC2 Instance references the ID of the security group.

When Terraform walks your dependency tree, it will create as many resources in parallel as it can, which means it can apply your changes fairly efficiently. That's the beauty of a declarative language: you just specify what you want and Terraform figures out the most efficient way to make it happen.

If you run the `apply` command, you'll see that Terraform wants to add a security group and replace the EC2 Instance with a new Instance that has the new user data:

```
$ terraform apply
```


(...)

Terraform will perform the following actions:

```
# aws_instance.example must be replaced
-/+ resource "aws_instance" "example" {
    ami                = "ami-0c55b159cbfafa1f0"
    instance_type      = "t2.micro"

    (...)

    + user_data          = "c765373..." # forces replacement
    ~ vpc_security_group_ids = [
        - "sg-871fa9ec",
    ] -> (known after apply)

    (...)
}

# aws_security_group.instance will be created
+ resource "aws_security_group" "instance" {
    + arn                = (known after apply)
    + description        = "Managed by Terraform"
    + egress              = (known after apply)
    + id                 = (known after apply)
    + ingress             = [
        + {
            + cidr_blocks      = [
                + "0.0.0.0/0",
            ]
            + description      = ""
            + from_port        = 8080
            + ipv6_cidr_blocks = []
            + prefix_list_ids  = []
            + protocol         = "tcp"
            + security_groups  = []
            + self              = false
            + to_port          = 8080
        },
    ]
    + name                = "terraform-example-instance"
    (...)
}
```

Plan: 2 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?

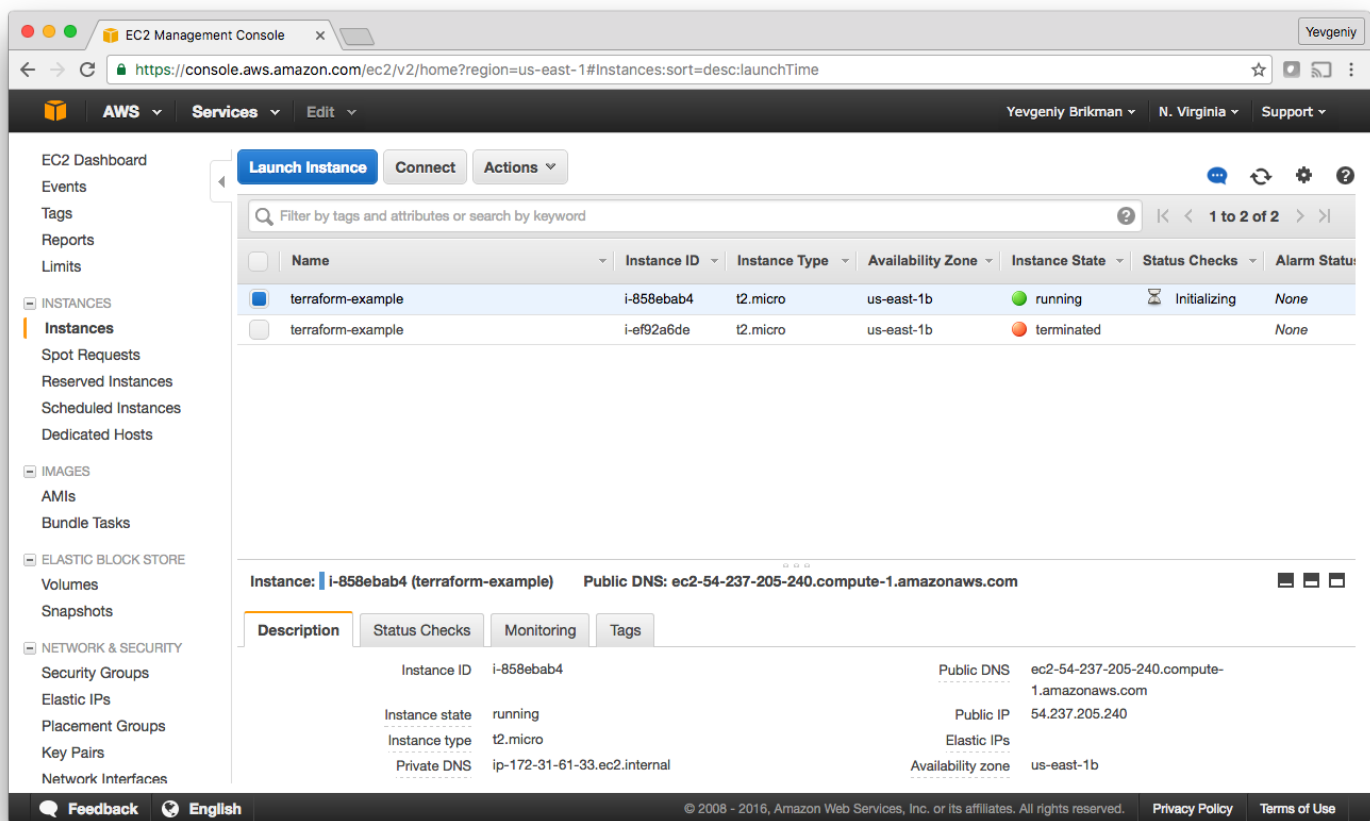
Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

Enter a value:

The `-/+` in the `plan` output means “replace”; look for the text “forces replacement” to figure out what is forcing Terraform to do a replacement. With EC2 Instances, changes to many attributes will force the original Instance to be terminated and a completely new Instance to be created (this is an example of the immutable infrastructure paradigm). It’s worth mentioning that while the web server is being replaced, any users of that web server would experience downtime; you’ll see how to do a zero-downtime deployment with Terraform in Terraform tips & tricks: loops, if-statements, and pitfalls.

Since the plan looks good, enter “yes” and you’ll see your new EC2 Instance deploying:



In the description panel at the bottom of the screen, you’ll also see the public IP address of this EC2 Instance. Give it a minute or two to boot up and then use a web browser or a tool like `curl` to make an HTTP request to this IP address at port 8080:

```
$ curl http://<EC2_INSTANCE_PUBLIC_IP>:8080
Hello, World
```

Yay, you now have a working web server running in AWS!

Deploy a Configurable Web Server

You may have noticed that the web server code has the port 8080 duplicated in both the security group and the User Data configuration. This violates the *Don't Repeat Yourself* (DRY) principle: every piece of knowledge must have a single, unambiguous, authoritative representation within a system. If you have the port number copy/pasted in two places, it's too easy to update it in one place but forget to make the same change in the other place.

To allow you to make your code more DRY and more configurable, Terraform allows you to define *input variables*. The syntax for declaring a variable is:

```
variable "NAME" {  
  [CONFIG ...]  
}
```

The body of the variable declaration can contain three parameters, all of them optional:

- `description` : It's always a good idea to use this parameter to document how a variable is used. Your teammates will not only be able to see this description while reading the code, but also when running the `plan` or `apply` commands (you'll see an example of this shortly).
- `default` : There are a number of ways to provide a value for the variable, including passing it in at the command line (using the `-var` option), via a file (using the `-var-file` option), or via an environment variable (Terraform looks for environment variables of the name `TF_VAR_<variable_name>`). If no value is passed in, the variable will fall back to this default value. If there is no default value, Terraform will interactively prompt the user for one.
- `type` : This allows you enforce *type constraints* on the variables a user passes in. Terraform supports a number of type constraints, including `string`, `number`, `bool`,

`list`, `map`, `set`, `object`, `tuple`, and `any`. If you don't specify a type, Terraform assumes the type is `any`.

For the web server example, here is how you can create a variable that stores the port number:

```
variable "server_port" {  
  description = "The port the server will use for HTTP requests"  
  type        = number  
}
```

Note that the `server_port` input variable has no `default`, so if you run the `apply` command now, Terraform will interactively prompt you to enter a value for `server_port` and show you the `description` of the variable:

```
$ terraform apply  
  
var.server_port  
  The port the server will use for HTTP requests  
  
Enter a value:
```

If you don't want to deal with an interactive prompt, you can provide a value for the variable via the `-var` command-line option:

```
$ terraform apply -var "server_port=8080"
```

You could also set the variable via an environment variable named `TF_VAR_<name>` where `<name>` is the name of the variable you're trying to set:

```
$ export TF_VAR_server_port=8080  
$ terraform apply
```

And if you don't want to deal with remembering extra command-line arguments every time you run `plan` or `apply`, you can specify a `default` value:

```
variable "server_port" {  
  description = "The port the server will use for HTTP requests"  
  type        = number  
  default     = 8080  
}
```

To use the value from an input variable in your Terraform code, you can use a new type of expression called a *variable reference*, which has the following syntax:

```
var.<VARIABLE_NAME>
```

For example, here is how you can set the `from_port` and `to_port` parameters of the security group to the value of the `server_port` variable:

```
resource "aws_security_group" "instance" {  
  name = "terraform-example-instance"  
  
  ingress {  
    from_port = var.server_port  
    to_port   = var.server_port  
    protocol  = "tcp"  
    cidr_blocks = ["0.0.0.0/0"]  
  }  
}
```

It's also a good idea to use the same variable when setting the port in the User Data script. To use a reference inside of a string literal, you need to use a new type of expression called an *interpolation*, which has the following syntax:

```
"${...}"
```

You can put any valid reference within the curly braces and Terraform will convert it to a string. For example, here's how you can use `var.server_port` inside of the User Data string:

```
user_data = <<-EOF
#!/bin/bash
echo "Hello, World" > index.html
nohup busybox httpd -f -p "${var.server_port}" &
EOF
```

In addition to input variables, Terraform also allows you to define *output variables* with the following syntax:

```
output "<NAME>" {
  value = <VALUE>
  [CONFIG ...]
}
```

The `NAME` is the name of the output variable and `VALUE` can be any Terraform expression that you would like to output. The `CONFIG` can contain two additional parameters, both optional:

- `description` : It's always a good idea to use this parameter to document what type of data is contained in the output variable.
- `sensitive` : Set this parameter to `true` to tell Terraform not to log this output at the end of `terraform apply`. This is useful if the output variable contains sensitive material or secrets, such as passwords or private keys.

For example, instead of having to manually poke around the EC2 console to find the IP address of your server, you can provide the IP address as an output variable:

```
output "public_ip" {
  value      = aws_instance.example.public_ip
  description = "The public IP of the web server"
}
```

This code uses an attribute reference again, this time referencing the `public_ip` attribute of the `aws_instance` resource. If you run the `apply` command again, Terraform will not apply any changes (since you haven't changed any resources), but it will show you the new output at the very end:

```
$ terraform apply

(...)

aws_security_group.instance: Refreshing state...
aws_instance.example: Refreshing state...

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

Outputs:

public_ip = 54.174.13.5
```

As you can see, output variables show up in the console after you run `terraform apply`, which users of your Terraform code may find useful (e.g., you now know what IP to test once the web server is deployed). You can also use the `terraform output` command to list all outputs without applying any changes:

```
$ terraform output
public_ip = 54.174.13.5
```

And you can run `terraform output <OUTPUT_NAME>` to see the value of a specific output called `<OUTPUT_NAME>`:

```
$ terraform output public_ip
54.174.13.5
```

This is particularly handy for scripting. For example, you could create a deployment script that runs `terraform apply` to deploy the web server, uses `terraform output public_ip` to grab its public IP, and runs `curl` on the IP as a quick smoke test to validate that the deployment worked.

Input and output variables are also essential ingredients in creating configurable and reusable infrastructure code, a topic you'll see more of in [How to create reusable infrastructure with Terraform modules](#).

Deploy a cluster of web servers

Running a single server is a good start, but in the real world, a single server is a single point of failure. If that server crashes, or if it becomes overwhelmed by too much traffic, users can no longer access your site. The solution is to run a cluster of servers, routing around servers that go down, and adjusting the size of the cluster up or down based on traffic (for more info, check out [A Comprehensive Guide to Building a Scalable Web App on Amazon Web Services](#)).

Managing such a cluster manually is a lot of work. Fortunately, you can let AWS take care of it for by you using an Auto Scaling Group (ASG). An ASG can automatically launch a cluster of EC2 Instances, monitor their health, automatically restart failed nodes, and adjust the size of the cluster in response to demand.

The first step in creating an ASG is to create a launch configuration, which specifies how to configure each EC2 Instance in the ASG. From deploying the single EC2 Instance earlier, you already know exactly how to configure it, and you can reuse almost exactly the same parameters in the `aws_launch_configuration` resource:

```
resource "aws_launch_configuration" "example" {
  image_id      = "ami-0c55b159cbfafelf0"
  instance_type = "t2.micro"
  security_groups = [aws_security_group.instance.id]

  user_data = <<-EOF
    #!/bin/bash
    echo "Hello, World" > index.html
    nohup busybox httpd -f -p "${var.server_port}" &
  EOF
```



```
    lifecycle {
      create_before_destroy = true
    }
  }
}
```

The only new thing here is the `lifecycle` setting. Terraform supports several *lifecycle settings* that let you customize how resources are created and destroyed. The `create_before_destroy` setting controls the order in which resources are recreated. The default order is to delete the old resource and then create the new one. Setting `create_before_destroy` to `true` reverses this order, creating the replacement first, and then deleting the old one. Since every change to a launch configuration creates a totally new launch configuration, you need this setting to ensure that the new configuration is created first, so any ASGs using this launch configuration can be updated to point to the new one, and then the old one can be deleted.

Now you can create the ASG itself using the `aws_autoscaling_group` resource:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.id

  min_size = 2
  max_size = 10

  tag {
    key           = "Name"
    value         = "terraform-asg-example"
    propagate_at_launch = true
  }
}
```

This ASG will run between 2 and 10 EC2 Instances (defaulting to 2 for the initial launch), each tagged with the name “terraform-asg-example”. The ASG uses a reference to fill in the launch configuration name.

To make this ASG work, you need to specify one more parameter: `availability_zones`. This parameter specifies into which availability zones (AZs) the EC2 Instances should be deployed. Each AZ represents an isolated AWS data center, so by deploying your Instances across multiple AZs, you ensure that your service can keep running even if

some of the AZs fail. You could hard-code the list of AZs (e.g. set it to `["us-east-2a", "us-east-2b"]`), but that won't be maintainable or portable (e.g., each AWS account has access to a slightly different set of AZs), so a better option is to use data sources to get the list of subnets in your AWS account.

A *data source* represents a piece of read-only information that is fetched from the provider (in this case, AWS) every time you run Terraform. Adding a data source to your Terraform configurations does not create anything new; it's just a way to query the provider's APIs for data and to make that data available to the rest of your Terraform code. Each Terraform provider exposes a variety of data sources. For example, the AWS provider includes data sources to look up VPC data, subnet data, AMI IDs, IP address ranges, the current user's identity, and much more.

The syntax for using a data source is very similar to the syntax of a resource:

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

`PROVIDER` is the name of a provider (e.g., `aws`), `TYPE` is the type of data source you want to use (e.g., `vpc`), `NAME` is an identifier you can use throughout the Terraform code to refer to this data source, and `CONFIG` consists of one or more arguments that are specific to that data source. For example, here is how you can use the `aws_availability_zones` data source to fetch the list of AZs in your AWS account:

```
data "aws_availability_zones" "all" {}
```

To get the data out of a data source, you use the following attribute reference syntax:

```
data.<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>
```

For example, to get the list of AZ names from the `aws_availability_zones` data source, you would use the following:

```
data.aws_availability_zones.all.names
```

Use this value to set the `availability_zone` argument of your `aws_autoscaling_group` resource:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.id
  availability_zones    = data.aws_availability_zones.all.names

  min_size = 2
  max_size = 10

  tag {
    key           = "Name"
    value         = "terraform-asg-example"
    propagate_at_launch = true
  }
}
```

Deploy a load balancer

At this point, you can deploy your ASG, but you'll have a small problem: you now have multiple servers, each with its own IP address, but you typically want to give your end users only a single IP to use. One way to solve this problem is to deploy a *load balancer* to distribute traffic across your servers and to give all your users the IP (actually, the DNS name) of the load balancer. Creating a load balancer that is highly available and scalable is a lot of work. Once again, you can let AWS take care of it for you, this time by using Amazon's *Elastic Load Balancer (ELB)* service.

AWS offers three different types of load balancers:

1. Application Load Balancer (ALB): best suited for HTTP and HTTPS traffic.
2. Network Load Balancer (NLB): best suited for TCP and UDP traffic.

3. Classic Load Balancer (CLB): this is the “legacy” load balancer that predates both the ALB and NLB. It can do HTTP, HTTPS, and TCP, but offers far fewer features than the ALB or NLB.

Since our web servers use HTTP, the ALB would be the best fit, but it requires more code and more explanation, so to keep this long blog post from getting even longer, we’re going to use the CLB, which is simpler to use.

You can create a CLB using the `aws_elb` resource:

```
resource "aws_elb" "example" {
  name                = "terraform-asg-example"
  availability_zones = data.aws_availability_zones.all.names
}
```

This creates an ELB that will be deployed across all of the AZs in your account. AWS load balancers don’t consist of a single server, but multiple servers that can run in separate AZs (that is, separate data centers). AWS will automatically scale the number of load balancer servers up and down based on traffic and handle failover if one of those servers goes down, so you get scalability and high availability out of the box.

Note that the `aws_elb` code above doesn’t do much until you tell the CLB how to route requests. To do that, you add one or more *listeners* which specify what port the CLB should listen on and what port it should route the request to:

```
resource "aws_elb" "example" {
  name                = "terraform-asg-example"
  availability_zones = data.aws_availability_zones.all.names

  # This adds a listener for incoming HTTP requests.
  listener {
    lb_port            = 80
    lb_protocol        = "http"
    instance_port      = var.server_port
    instance_protocol  = "http"
  }
}
```

In the code above, we are telling the CLB to receive HTTP requests on port 80 (the default port for HTTP) and to route them to the port used by the Instances in the ASG. Note that, by default, CLBs don't allow any incoming or outgoing traffic (just like EC2 Instances), so you need to add a new security group to explicitly allow inbound requests on port 80 and all outbound requests (the latter is to allow the CLB to perform health checks, as you'll see shortly):

```
resource "aws_security_group" "elb" {
  name = "terraform-example-elb"

  # Allow all outbound
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Inbound HTTP from anywhere
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

You now need to tell the CLB to use this security group by adding the `security_groups` parameter:

```
resource "aws_elb" "example" {
  name                = "terraform-asg-example"
  security_groups     = [aws_security_group.elb.id]
  availability_zones  = data.aws_availability_zones.all.names

  # This adds a listener for incoming HTTP requests.
  listener {
    lb_port           = var.elb_port
    lb_protocol       = "http"
    instance_port     = var.server_port
    instance_protocol = "http"
  }
}
```

The CLB has one other nifty trick up its sleeve: it can periodically check the health of your EC2 Instances and, if an instance is unhealthy, it will automatically stop routing traffic to it. Let's add an HTTP health check where the CLB will send an HTTP request every 30 seconds to the "/" URL of each of the EC2 Instances and only mark an Instance as healthy if it responds with a 200 OK:

```
resource "aws_elb" "example" {
  name                = "terraform-asg-example"
  security_groups     = [aws_security_group.elb.id]
  availability_zones  = data.aws_availability_zones.all.names

  health_check {
    target            = "HTTP:${var.server_port}/"
    interval          = 30
    timeout           = 3
    healthy_threshold = 2
    unhealthy_threshold = 2
  }

  # This adds a listener for incoming HTTP requests.
  listener {
    lb_port            = var.elb_port
    lb_protocol        = "http"
    instance_port      = var.server_port
    instance_protocol  = "http"
  }
}
```

How does the CLB know which EC2 Instances to send requests to? You can attach a static list of EC2 Instances to an ELB using the CLB's `instances` parameter, but with an ASG, Instances will be launching and terminating dynamically all the time, so that won't work. Instead, you can use the `load_balancers` parameter of the `aws_autoscaling_group` resource to tell the ASG to register each Instance in the CLB:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.id
  availability_zones   = data.aws_availability_zones.all.names

  min_size = 2
  max_size = 10
}
```

```
load_balancers    = [aws_elb.example.name]
health_check_type = "ELB"

tag {
  key          = "Name"
  value        = "terraform-asg-example"
  propagate_at_launch = true
}
```

Notice that we've also configured the `health_check_type` for the ASG to `"ELB"`. The default `health_check_type` is `"EC2"`, which is a minimal health check that only a considers Instance unhealthy if the AWS hypervisor says the server is completely down or unreachable. The `"ELB"` health check is much more robust, as it tells the ASG to use the CLB's health check to determine if an Instance is healthy or not and to automatically replace Instances if the CLB reports them as unhealthy. That way, Instances will be replaced not only if they are completely down, but also if, for example, they've stopped serving requests because they ran out of memory or a critical process crashed.

One last thing to do before deploying the load balancer: let's add its DNS name as an output so it's easier to test if things are working:

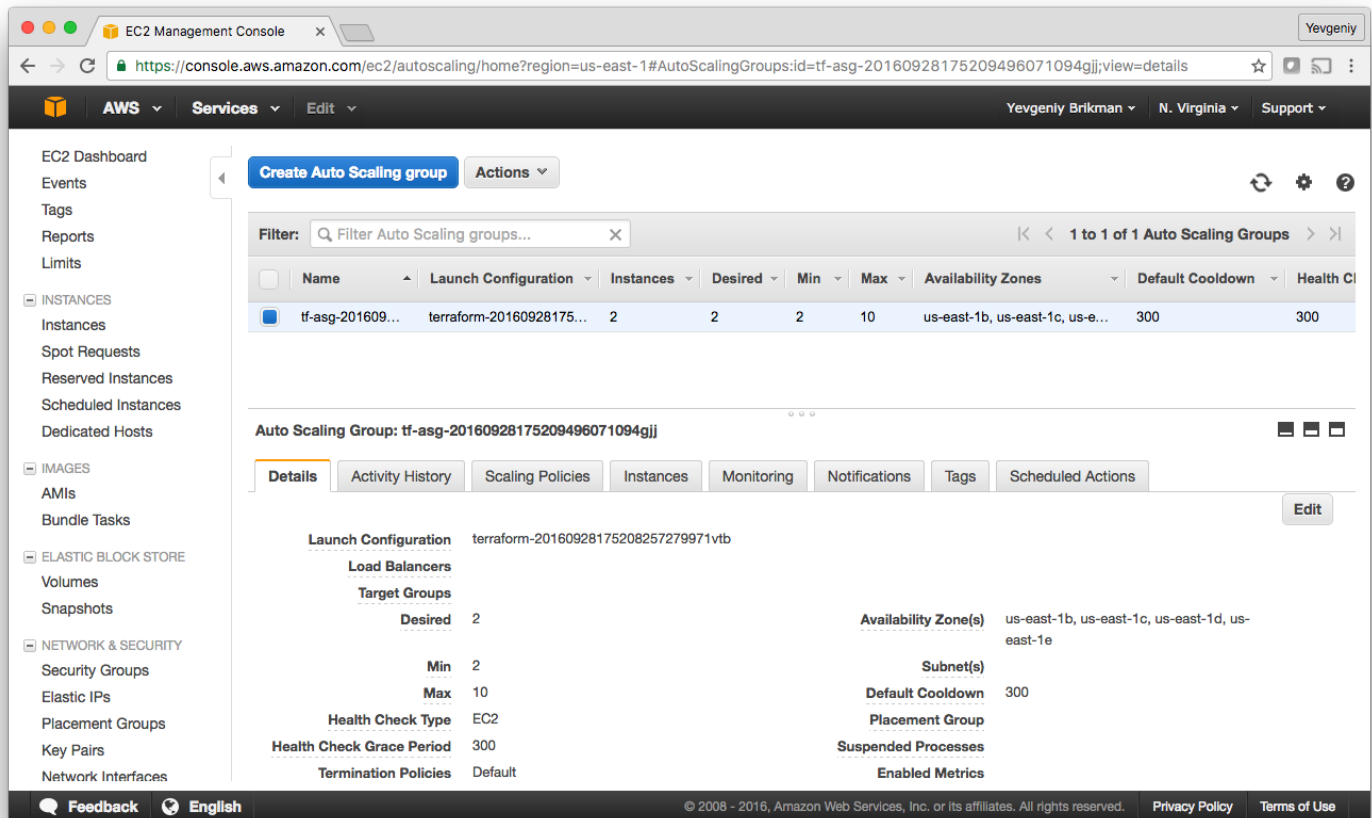
```
output "clb_dns_name" {
  value        = aws_elb.example.dns_name
  description = "The domain name of the load balancer"
}
```

Run `terraform apply` and read through the plan output. You should see that your original single EC2 Instance is being removed and in its place, Terraform will create a launch configuration, ASG, ALB, and a security group. If the plan looks good, type in "yes" and hit enter. When `apply` completes, you should see the `clb_dns_name` output:

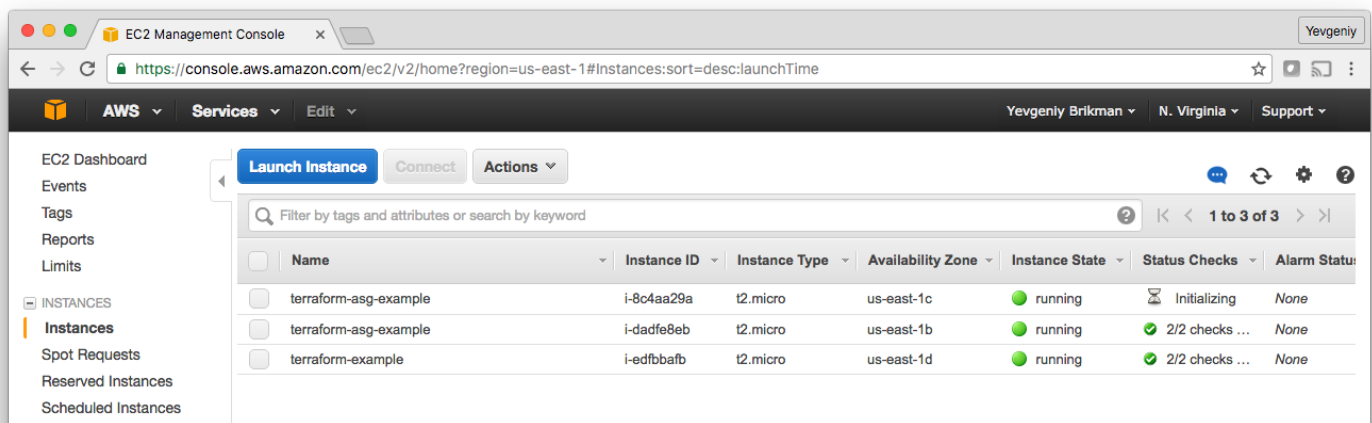
Outputs:

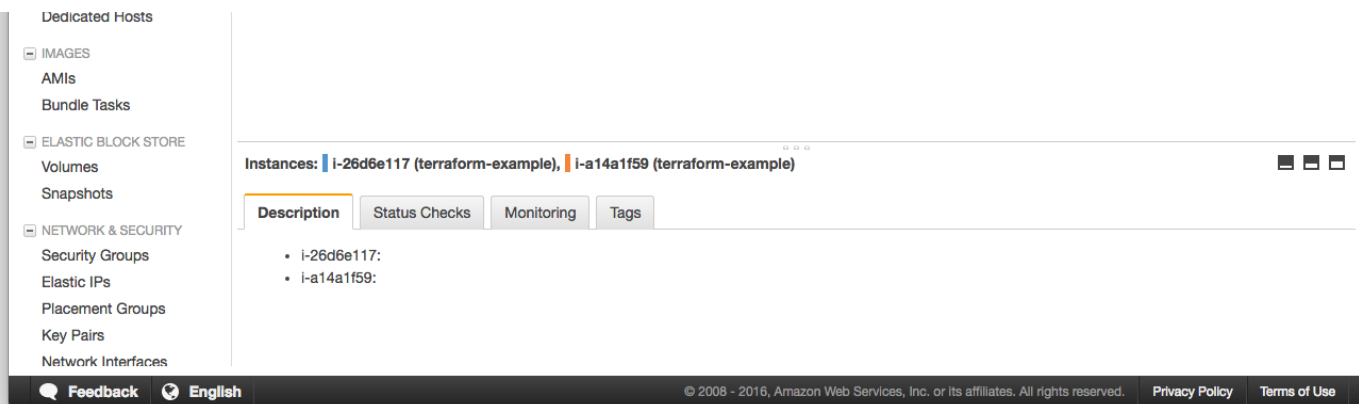
```
clb_dns_name = terraform-asg-example-123.us-east-2.elb.amazonaws.com
```

Copy this URL down. It'll take a couple minutes for the Instances to boot and show up as healthy in the CLB. In the meantime, you can inspect what you've deployed. Open up the ASG section of the EC2 console, and you should see that the ASG has been created:

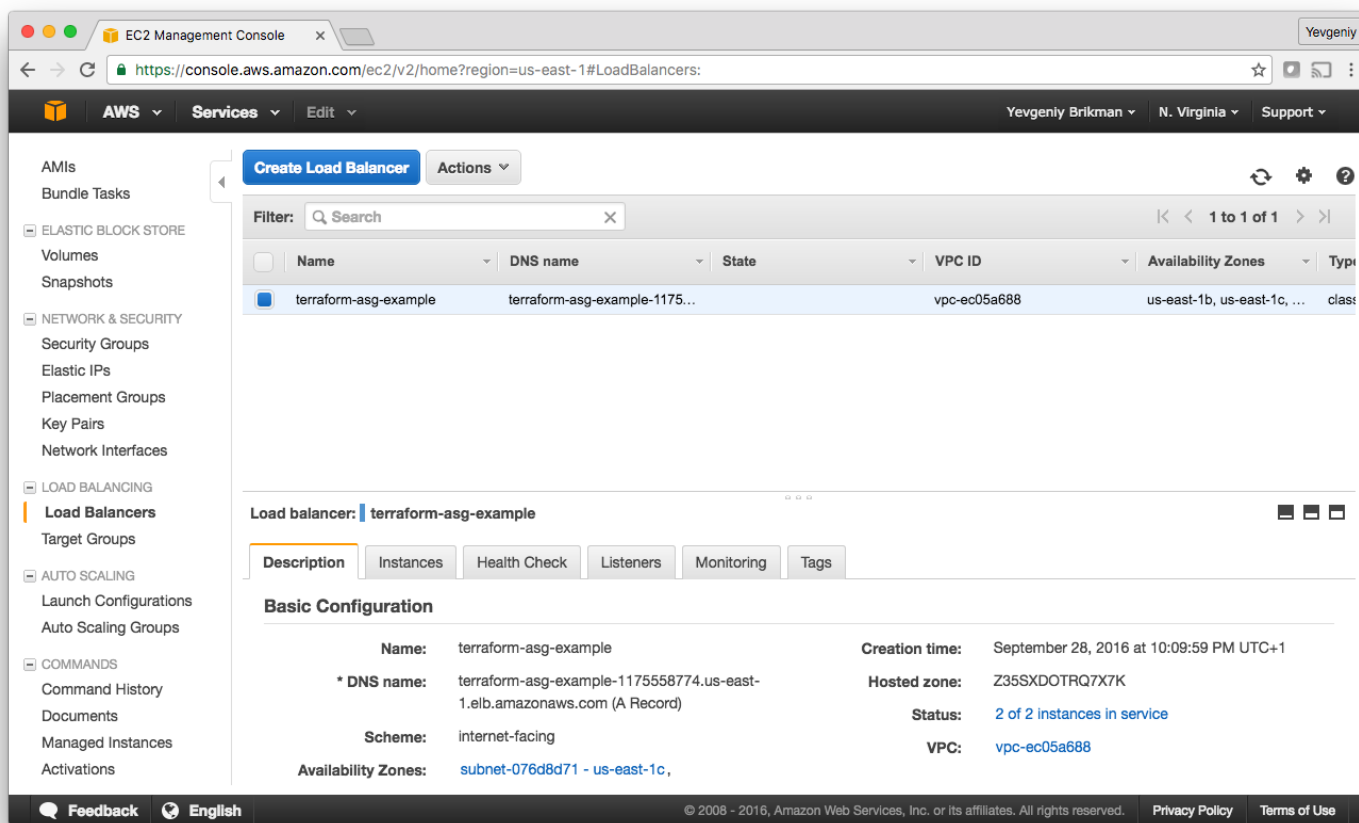


If you switch over to the Instances tab, you'll see the two instances in the process of launching:





And finally, if you switch over to the Load Balancers tab, you'll see your CLB:



Wait for the “Status” indicator to say “2 of 2 instances in service.” This typically takes 1–2 minutes. Once you see it, test the `clb_dns_name` output you copied earlier:

```
$ curl http://terraform-asg-example-123.us-east-2.elb.amazonaws.com
Hello, World
```

Success! The CLB is routing traffic to your EC2 Instances. Each time you hit the URL, it'll pick a different Instance to handle the request. You now have a fully working cluster of web servers! As a reminder, the sample code for the examples above is available at: <https://github.com/gruntwork-io/intro-to-terraform>.

At this point, you can see how your cluster responds to firing up new Instances or shutting down old ones. For example, go to the Instances tab, and terminate one of the Instances by selecting its checkbox, selecting the “Actions” button at the top, and setting the “Instance State” to “Terminate.” Continue to test the CLB URL and you should get a “200 OK” for each request, even while terminating an Instance, as the CLB will automatically detect that the Instance is down and stop routing to it. Even more interestingly, a short time after the Instance shuts down, the ASG will detect that fewer than 2 Instances are running, and automatically launch a new one to replace it (self healing!). You can also see how the ASG resizes itself by changing the `min_size` and `max_size` parameters or adding a `desired_size` parameter to your Terraform code, and re-running `apply`.

Of course, there are many other aspects to an ASG that we have not covered here. For a real deployment, you would need to attach IAM roles to the EC2 Instances, set up a mechanism to update the EC2 Instances in the ASG with zero downtime, and configure auto scaling policies to adjust the size of the ASG in response to load. For a fully pre-assembled, battle-tested, documented, production-ready version of the ASG, as well as other types of infrastructure such as Docker clusters, relational databases, VPCs, and more, check out the Gruntwork Infrastructure as Code Library.

Clean up

When you're done experimenting with Terraform, it's a good idea to remove all the resources you created so AWS doesn't charge you for them. Since Terraform keeps track of what resources you created, cleanup is simple. All you need to do is run the `destroy` command:

```
$ terraform destroy
```

```
(...)
```

Terraform will perform the following actions:

```
# aws_autoscaling_group.example will be destroyed
- resource "aws_autoscaling_group" "example" {
    (...)
}

# aws_launch_configuration.example will be destroyed
- resource "aws_launch_configuration" "example" {
    (...)
}

# aws_lb.example will be destroyed
- resource "aws_lb" "example" {
    (...)
}

(...)
```

Plan: 0 to add, 0 to change, 8 to destroy.

Do you really want to destroy all resources?

Terraform will destroy all your managed infrastructure, as shown above. There is no undo. Only 'yes' will be accepted to confirm.

Enter a value:

Once you type in “yes” and hit enter, Terraform will build the dependency graph and delete all the resources in the right order, using as much parallelism as possible. In about a minute, your AWS account should be clean again.

Conclusion

You now have a basic grasp of how to use Terraform. The declarative language makes it easy to describe exactly the infrastructure you want to create. The `plan` command allows you to verify your changes and catch bugs before deploying them. Variables, references, and dependencies allow you to keep code DRY and efficient.

However, we’ve only just scratched the surface. In Part 3 of the series, How to manage Terraform state, we’ll show how Terraform keeps track of what infrastructure it has

already created, and the profound impact that has on how you should structure your Terraform code. In Part 4 of the series, we'll show how to create reusable infrastructure with Terraform modules.

For an expanded version of this blog post series, pick up a copy of the book [Terraform: Up & Running](#) (2nd edition available now!). If you need help with Terraform, DevOps practices, or AWS at your company, feel free to reach out to us at Gruntwork.

Thanks to Josh Padnick.

[AWS](#) [Terraform](#) [DevOps](#) [Infrastructure As Code](#)

[About](#) [Help](#) [Legal](#)

Get the Medium app

