Chapter 2. Getting Started with Terraform

In this chapter, you're going to learn the basics of how to use Terraform. It's an easy tool to learn, so in the span of about 30 pages, you'll go from running your first Terraform commands all the way up to using Terraform to deploy a cluster of servers with a load balancer that distributes traffic across them. This infrastructure is a good starting point for running scalable, highly available web services and microservices. In subsequent chapters, you'll evolve this example even further.

Terraform can provision infrastructure across public cloud providers such as Amazon Web Services (AWS), Azure, Google Cloud, and DigitalOcean, as well as private cloud and virtualization platforms such as OpenStack and VMWare. For just about all of the code examples in this chapter and the rest of the book, you are going to use AWS. AWS is a good choice for learning Terraform because:

- AWS is the most popular cloud infrastructure provider, by far. It has a 45% share in the cloud infrastructure market, which is more than the next three biggest competitors (Microsoft, Google, and IBM) combined (http://bit.ly/2kWCuCm).
- AWS provides a huge range of reliable and scalable cloud hosting services, including: Elastic Compute Cloud (EC2), which you can use to deploy virtual servers; Auto Scaling Groups (ASGs), which make it easier to manage a cluster of virtual servers; and Elastic Load Balancers (ELBs), which you can use to distribute traffic across the cluster of virtual servers. [1]
- AWS offers a generous Free Tier that should allow you to run all of these examples for free. If you already used up your free tier credits, the examples in this book should still cost you no more than a few dollars.

If you've never used AWS or Terraform before, don't worry, as this tutorial is designed for novices to both technologies. I'll walk you through the following steps:

- Set up your AWS account
- Install Terraform
- Deploy a single server
- Deploy a single web server
- Deploy a configurable web server
- Deploy a cluster of web servers
- Deploy a load balancer
- Clean up

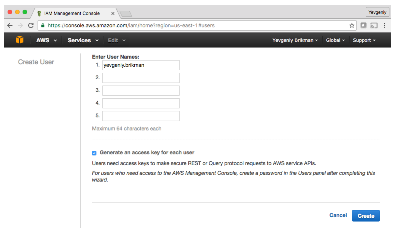
EXAMPLE CODE

As a reminder, all of the code examples in the book can be found at the following URL: *https://github.com/brikis98/terraform-up-and-running-code*.

Set Up Your AWS Account

If you don't already have an AWS account, head over to *https://aws.amazon.com* and sign up. When you first register for AWS, you initially sign in as *root user*. This user account has access permissions to do absolutely anything in the account, so from a security perspective, it's not a good idea to use the root user on a day-to-day basis. In fact, the *only* thing you should use the root user for is to create other user accounts with more limited permissions, and switch to one of those accounts immediately.[2]

To create a more limited user account, you will need to use the *Identity and Access Management (IAM)* service. IAM is where you manage user accounts as well as the permissions for each user. To create a new *IAM user*, head over to the 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, as shown in Figure 2-1.


IAM Management Console

https://console.aws.amazon.com/iam/home?region=us-east-1#users

Create User

Enter User Names:
1. yevgeniy.brikman
2.
3.
4.
5.

Maximum 64 characters each

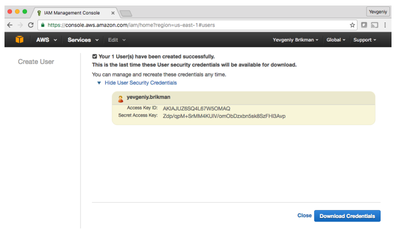
[✓] Generate an access key for each user

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.

Figure 2-1. Create a new IAM user

Click the "Create" button and AWS will show you the security credentials for that user, which consist of an *Access Key ID* and a *Secret Access Key*, as shown in Figure 2-2. You must save these immediately, as they will never be shown again. I recommend storing them somewhere secure (e.g., a password manager such as 1Password, LastPass, or OS X Keychain) so you can use them a little later in this tutorial.


IAM Management Console

https://console.aws.amazon.com/iam/home?region=us-east-1#users

Create User

Your 1 User(s) have been created successfully.
This is the last time these User security credentials will be available for download.
You can manage and recreate these credentials any time.

▼ Hide User Security Credentials

yevgeniy.brikman
Access Key ID: AKIAJUZ6SQ4L67W5OMAQ
Secret Access Key: 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

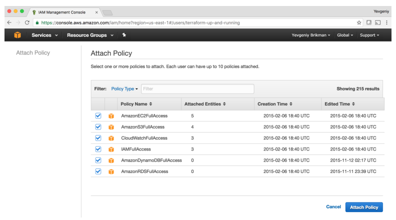
Figure 2-2. Store your AWS credentials somewhere secure. Never share them with anyone. Don't worry, the ones in the screenshot are fake.

Once you've saved your credentials, click the "Close" button (twice), and you'll be taken to the list of IAM users. Click the user you just created and select the "Permissions" tab. By default, new IAM users have no permissions whatsoever, and therefore cannot do anything in an AWS account.

To give an IAM user permissions to do something, you need to associate one or more IAM Policies with that user's account. An *IAM Policy* is a JSON document that defines what a user is or isn't allowed to do. You can create your own IAM Policies or use some of the predefined IAM Policies, which are known as *Managed Policies*.[3]

To run the examples in this book, you will need to add the following Managed Policies to your IAM user, as shown in Figure 2-3:

1. `AmazonEC2FullAccess`: required for this chapter.
2. `AmazonS3FullAccess`: required for Chapter 3.
3. `AmazonDynamoDBFullAccess`: required for Chapter 3.
4. `AmazonRDSFullAccess`: required for Chapter 3.
5. `CloudWatchFullAccess`: required for Chapter 5.
6. `IAMFullAccess`: required for Chapter 5.



Attach Policy

Select one or more policies to attach. Each user can have up to 10 policies attached.

Filter: Policy Type Filter Showing 215 results

		Policy Name	Attached Entities	Creation Time	Edited Time
✓		AmazonEC2FullAccess	5	2015-02-06 18:40 UTC	2015-02-06 18:40 UTC
✓		AmazonS3FullAccess	4	2015-02-06 18:40 UTC	2015-02-06 18:40 UTC
✓		CloudWatchFullAccess	3	2015-02-06 18:40 UTC	2015-02-06 18:40 UTC
✓		IAMFullAccess	3	2015-02-06 18:40 UTC	2015-02-06 18:40 UTC
✓		AmazonDynamoDBFullAccess	0	2015-02-06 18:40 UTC	2015-11-12 02:17 UTC
✓		AmazonRDSFullAccess	0	2015-02-06 18:40 UTC	2015-11-11 23:39 UTC

Figure 2-3. Add several Managed IAM Policies to your new IAM user

A NOTE ON DEFAULT VPCS

Please note that if you are using an existing AWS account, it must have a *Default VPC* in it. A *VPC*, or Virtual Private Cloud, is an isolated area of your AWS account that has its own virtual network and IP address space. Just about every AWS resource deploys into a VPC. If you don't explicitly specify a VPC, the resource will be deployed into the *Default VPC*, which is part of every new AWS account. All the examples in this book rely on this Default VPC, so if for some reason you deleted the one in your account, either use a different region (each region has its own Default VPC) or contact AWS customer support, and they can re-create a Default VPC for you (there is no way to mark a VPC as "Default" yourself). Otherwise, you'll need to update almost every example to include a `vpc_id` or `subnet_id` parameter pointing to a custom VPC.

Install Terraform

You can download Terraform from the Terraform homepage. Click the download link, select the appropriate package for your operating system, download the zip archive, and unzip it into the directory where you want Terraform to be installed. The archive will extract a single binary called `terraform`, which you'll want to add to your `PATH` environment variable.

To check if things are working, run the `terraform` command, and you should see the usage instructions:

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

Available commands are:
    apply       Builds or changes infrastructure
    destroy     Destroy Terraform-managed infrastructure
    get         Download and install modules for the configuration
    graph       Create a visual graph of Terraform resources
    (...)
```

In order for Terraform to be able to make changes in your AWS account, you will need to set the AWS credentials for the IAM user you created earlier as the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. For example, here is how you can do it in a Unix/Linux/OS X terminal:

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

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

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 the `configure` command on the AWS CLI, or IAM Roles, which you can add to almost any resource in AWS. For more info, see Configuring the AWS Command Line Interface (http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html).

Deploy a Single Server

Terraform code is written in the *HashiCorp Configuration Language (HCL)* in files with the extension *.tf*. [4] 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 an empty folder and put a file in it called *main.tf* with the following contents:

```
provider "aws" {
  region = "us-east-1"
}
```

This tells Terraform that you are going to be using AWS as your provider and that you wish to deploy your infrastructure into the `us-east-1` region. AWS has data centers all over the world, grouped into regions and availability zones. An *AWS region* is a separate geographic area, such as `us-east-1` (North Virginia), `eu-west-1` (Ireland), and `ap-southeast-2` (Sydney). Within each region, there are multiple isolated data centers known as *availability zones*, such as `us-east-1a`, `us-east-1b`, and so on. [5]

For each type of provider, there are many different kinds of *resources* you can create, such as servers, databases, and load balancers. For example, to deploy a single server in AWS, known as an `EC2 Instance`, you can add the `aws_instance` resource to *main.tf*:

```
resource "aws_instance" "example" {
  ami           = "ami-40d28157"
  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 configuration parameters that are specific to that resource (e.g., `ami = "ami-40d28157"`). For the `aws_instance` resource, there are many different configuration parameters, but for now, you only need to set the following ones: [6]

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 (see "Server Templating Tools" for a discussion of machine images and server templating). The preceding code example sets the `ami` parameter to the ID of an Ubuntu 16.04 AMI in `us-east-1`.

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. 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 plan` command:

```
> terraform plan

Refreshing Terraform state in-memory prior to plan...
(...)

+ aws_instance.example
    ami:                      "ami-40d28157"
    availability_zone:        "<computed>"
    instance_state:           "<computed>"
    instance_type:            "t2.micro"
    key_name:                 "<computed>"
    private_dns:              "<computed>"
    private_ip:               "<computed>"
    public_dns:               "<computed>"
    public_ip:                "<computed>"
    security_groups.#:        "<computed>"
    subnet_id:                "<computed>"
    vpc_security_group_ids.#: "<computed>"
    (...)
```

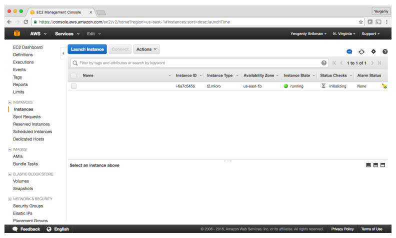
```
Plan: 1 to add, 0 to change, 0 to destroy.
```

The `plan` command lets you see what Terraform will do before actually 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`: 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 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

aws_instance.example: Creating...
  ami:                      "" => "ami-40d28157"
  availability_zone:        "" => "<computed>"
  instance_state:           "" => "<computed>"
  instance_type:            "" => "t2.micro"
  key_name:                 "" => "<computed>"
  private_dns:              "" => "<computed>"
  private_ip:               "" => "<computed>"
  public_dns:               "" => "<computed>"
  public_ip:                "" => "<computed>"
  security_groups.#:        "" => "<computed>"
  subnet_id:                "" => "<computed>"
  vpc_security_group_ids.#: "" => "<computed>"
  (...)

aws_instance.example: Still creating... (10s elapsed)
aws_instance.example: Still creating... (20s elapsed)
aws_instance.example: Creation complete

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

Congrats, you've just deployed a server with Terraform! To verify this, head over to the EC2 console, and you should see something similar to Figure 2-4.



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status
	i-6a7c545b	t2.micro	us-east-1b	running	Initializing	None

Figure 2-4. A single EC2 Instance

Sure enough the server 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:

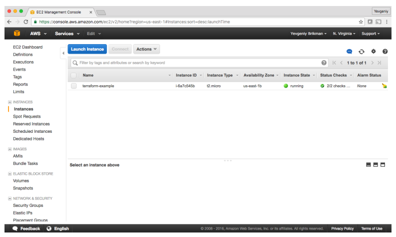
```
resource "aws_instance" "example" {
  ami           = "ami-40d28157"
  instance_type = "t2.micro"

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

Run the `plan` command again to see what this would do:

```
> terraform plan

aws_instance.example: Refreshing state... (ID: i-6a7c545b)
(...)

~ aws_instance.example
    tags.%:    "0" => "1"
    tags.Name: "" => "terraform-example"

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

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 `plan` 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 Terraform Compares to Other Infrastructure as Code Tools"). The preceding diff shows that Terraform wants to create a single tag called "Name," which is exactly what you need, so run the `apply` command again.

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



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status
terraform-example	i-6a7c545b	t2.micro	us-east-1b	running	2/2 checks ...	None

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:

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

You should also create a file called *.gitignore* that tells Git to ignore certain types of files so you don't accidentally check them in:

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

The preceding *.gitignore* file tells 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.com, 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>.g
```

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

```
git push origin master
```

And whenever you want to see changes your teammates have made, you can *pull* them from `origin`:

```
git pull origin master
```

As you go through the rest of this book, and as you use Terraform in general, make sure to regularly `git commit` and `git push` your changes. This way, you'll not only be able to collaborate with team members on this code, but all your infrastructure changes will also be captured in the commit log, which is very handy for debugging. You'll learn more about using Terraform as a team in Chapter 6.

Deploy a Single Web Server

The next step is to run a web server on this Instance. The goal is to deploy the simplest web architecture possible: a single web server that can respond to HTTP requests, as shown in Figure 2-6.

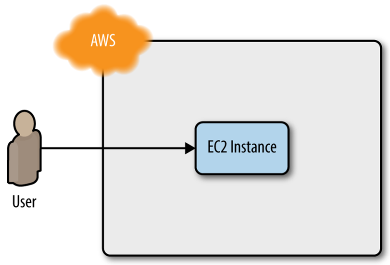


Figure 2-6. Start with a simple architecture: a single web server running in AWS that responds to HTTP requests

In a real-world use case, you'd probably build the web server using a web framework like Ruby on Rails or Django, but to keep this example simple, let's run a dirt-simple web server that always returns the text "Hello, World": [7]

```
#!/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 tool called busybox (which is installed by default on Ubuntu) to fire up a web server on port 8080 to serve that file. I wrapped the `busybox` command with `nohup` and `&` so that the web server runs permanently in the background, while the Bash script itself can exit.

PORT NUMBERS

The reason this example uses port 8080, rather than the default HTTP port 80, is that listening on any port less than 1024 requires root user privileges. This is a security risk, because any attacker who manages to compromise your server would get root privileges, too.

Therefore, it's a best practice to run your web server with a non-root user that has limited permissions. That means you have to listen on higher-numbered ports, but as you'll see later in this chapter, you can configure a load balancer to listen on port 80 and route traffic to the high-numbered ports on your server(s).

How do you get the EC2 Instance to run this script? Normally, as discussed in "Server Templating Tools", you would use a tool like Packer to create a custom AMI that has the web server installed on it. Since the dummy web server in this example is just a one-liner that uses `busybox`, you can use a plain Ubuntu 16.04 AMI, and run the "Hello, World" script as part of the EC2 Instance's *User Data* configuration, which AWS will execute when the Instance is booting:

```
resource "aws_instance" "example" {
  ami           = "ami-40d28157"
  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 insert newline characters all over the place.

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. [8]

Simply creating a security group isn't enough; you also need to tell the EC2 Instance to actually use it. To do that, you need to pass the ID of the security group into the `vpc_security_group_ids` parameter of the `aws_instance` resource.

To get the ID of the security group, you can use *interpolation syntax*, which looks like this:

```
"${something_to_interpolate}"
```

Whenever you see a dollar sign and curly braces inside of double quotes, that means Terraform is going to process the text within the curly braces in a special way. You'll see many different uses for this syntax throughout the book. The first use will be to look up an *attribute* of a resource.

In Terraform, every resource exposes attributes that you can access using interpolation (you can find the list of available attributes in the documentation for each resource). The syntax is:

```
"${TYPE.NAME.ATTRIBUTE}"
```

For example, here is how you can get the ID of the security group:

```
"${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-40d28157"
  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 use interpolation syntax to have one resource reference another resource, 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, Terraform knows it needs to create the security group before the EC2 Instance, since the EC2 Instance references the ID of the security group. You can even get Terraform to show you the dependency graph by running the `graph` command:

```
> terraform graph

digraph {
  compound = "true"
  newrank = "true"
  subgraph "root" {
    "[root] aws_instance.example"
      [label = "aws_instance.example", shape = "box"]
    "[root] aws_security_group.instance"
      [label = "aws_security_group.instance", shape = "box"]
    "[root] provider.aws"
      [label = "provider.aws", shape = "diamond"]
    "[root] aws_instance.example" -> "[root] aws_security_group.instanc
    "[root] aws_instance.example" -> "[root] provider.aws"
    "[root] aws_security_group.instance" -> "[root] provider.aws"
  }
}
```

The output is in a graph description language called DOT, which you can turn into an image, such as the dependency graph in Figure 2-7, by using a desktop app such as Graphviz or webapp such as GraphvizOnline (http://bit.ly/2mPbxmg).

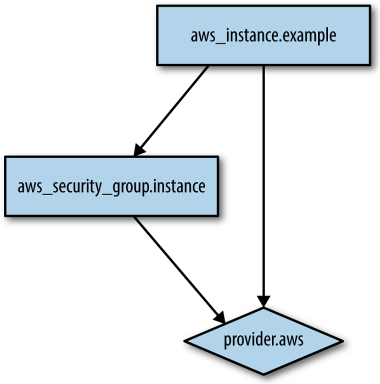


Figure 2-7. The dependency graph for the EC2 Instance and its security group

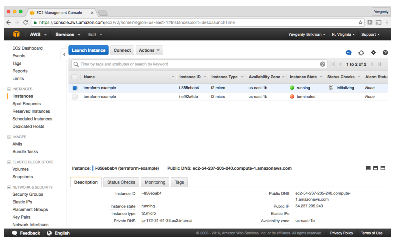
When Terraform walks your dependency tree, it will create as many resources in parallel as it can, which means it is very fast at applying your changes. That's the beauty of a declarative language: you just specify what you want and Terraform figures out the most efficient way to make it happen.

If you run the `plan` command, you'll see that Terraform wants to add a security group and replace the original EC2 Instance with a new one that has the new user data (the `-/+` means "replace"):

```
> terraform plan

(...)


+ aws_security_group.instance
    description:                         "Managed by Terraform"
    egress.#:                            "<computed>"
    ingress.#:                           "1"
    ingress.516175195.cidr_blocks.#:     "1"
    ingress.516175195.cidr_blocks.0:     "0.0.0.0/0"
    ingress.516175195.from_port:         "8080"
    ingress.516175195.protocol:          "tcp"
    ingress.516175195.security_groups.#: "0"
    ingress.516175195.self:              "false"
    ingress.516175195.to_port:           "8080"
    owner_id:                            "<computed>"
    vpc_id:                              "<computed>"

-/+ aws_instance.example
    ami:                      "ami-40d28157" => "ami-40d28157"
    instance_state:           "running" => "<computed>"
    instance_type:            "t2.micro" => "t2.micro"
    security_groups.#:        "0" => "<computed>"
    vpc_security_group_ids.#: "1" => "<computed>"
    (...)

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

In Terraform, most changes to an EC2 Instance, other than metadata such as tags, actually create a completely new Instance. This is an example of the immutable infrastructure paradigm discussed in "Server Templating Tools". It's worth mentioning that while the web server is being replaced, your users would experience downtime; you'll see how to do a zero-downtime deployment with Terraform in Chapter 5.

Since the plan looks good, run the `apply` command again and you'll see your new EC2 Instance deploying, as shown in Figure 2-8.



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Statu
terraform-example	i-858ebab4	t2.micro	us-east-1b	● running	Initializing	None
terraform-example	i-ef92a6de	t2.micro	us-east-1b	● terminated		None

Instance: i-858ebab4 (terraform-example) Public DNS: ec2-54-237-205-240.compute-1.amazonaws.com

Instance ID	i-858ebab4	Public DNS	ec2-54-237-205-240.compute-1.amazonaws.com
Instance state	running	Public IP	54.237.205.240
Instance type	t2.micro	Elastic IPs	
Private DNS	ip-172-31-61-33.ec2.internal	Availability zone	us-east-1b

Figure 2-8. The new EC2 Instance with the web server code replaces the old Instance

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!

NETWORK SECURITY

To keep all the examples in this book simple, they deploy not only into your Default VPC (as mentioned earlier), but also the default *subnets* of that VPC. A VPC is partitioned into one or more subnets, each with its own IP addresses. The subnets in the Default VPC are all *public subnets*, which means they get IP addresses that are accessible from the public internet. This is why you are able to test your EC2 Instance from your home computer.

Running a server in a public subnet is fine for a quick experiment, but in real-world usage, it's a security risk. Hackers all over the world are *constantly* scanning IP addresses at random for any weakness. If your servers are exposed publicly, all it takes is accidentally leaving a single port unprotected or running out-of-date code with a known vulnerability, and someone can break in.

Therefore, for production systems, you should deploy all of your servers, and certainly all of your data stores, in *private subnets*, which have IP addresses that can only be accessed from inside the VPC and not from the public internet. The only servers you should run in public subnets are a small number of reverse proxies and load balancers (you'll see an example of a load balancer later in this chapter) that you lock down as much as possible.

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.[9] 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`

Must be one of `"string"`, `"list"`, or `"map"`. If you don't specify a type, Terraform will try to guess the type from the `default` value. If there is no `default`, then Terraform will assume the variable is a string.

Here is an example of a list input variable in Terraform:

```
variable "list_example" {
  description = "An example of a list in Terraform"
  type        = "list"
  default     = [1, 2, 3]
}
```

And here's a map:

```
variable "map_example" {
  description = "An example of a map in Terraform"
  type        = "map"

  default = {
    key1 = "value1"
    key2 = "value2"
    key3 = "value3"
  }
}
```

For the web server example, all you need is a number, which in Terraform, are automatically coerced to strings, so you can omit the type:[10]

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

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

```
> terraform plan

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 plan -var server_port="8080"
```

And if you don't want to deal with remembering a command-line flag every time you run `plan` or `apply`, you're better off specifying a `default` value:

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

To extract values from these input variables in your Terraform code, you can use interpolation syntax again. The syntax for looking up a variable is:

```
"${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"]
  }
}
```

You can use the same syntax to set the port number used by `busybox` in the User Data of the EC2 Instance:

```
  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 *ouput variables* with the following syntax:

```
output "NAME" {
  value = VALUE
}
```

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}"
}
```

This code uses interpolation syntax again, this time to reference 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... (ID: sg-db91dba1)
aws_instance.example: Refreshing state... (ID: i-61744350)

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`. You can also use the `terraform output` command to list outputs without applying any changes and `terraform output OUTPUT_NAME` to see the value of a specific output:

```
> terraform output public_ip
54.174.13.5
```

Input and output variables are essential ingredients in creating configurable and reusable infrastructure code, a topic you'll see more of in Chapter 4.

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 overloaded from too much traffic, users will be unable to 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.[11]

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)*, as shown in Figure 2-9. An ASG takes care of a lot of tasks for you completely automatically, including launching a cluster of EC2 Instances, monitoring the health of each Instance, replacing failed Instances, and adjusting the size of the cluster in response to load.

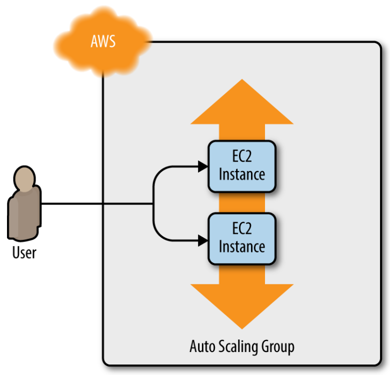


Figure 2-9. Instead of a single web server, run a cluster of web servers using an Auto Scaling Group

The first step in creating an ASG is to create a *launch configuration*, which specifies how to configure each EC2 Instance in the ASG. The `aws_launch_configuration` resource uses almost exactly the same parameters as the `aws_instance` resource, so you can replace the latter with the former pretty easily:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-40d28157"
  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 addition is the `lifecycle` parameter, which is required for using a launch configuration with an ASG. The `lifecycle` parameter is an example of a *meta-parameter*, or a parameter that exists on just about every resource in Terraform. You can add a `lifecycle` block to any resource to configure how that resource should be created, updated, or destroyed.

One of the available `lifecycle` settings is `create_before_destroy`, which, if set to `true`, tells Terraform to always create a replacement resource before destroying the original resource. For example, if you set `create_before_destroy` to `true` on an EC2 Instance, then whenever you make a change to that Instance, Terraform will first create a new EC2 Instance, wait for it to come up, and then remove the old EC2 Instance.

The catch with the `create_before_destroy` parameter is that if you set it to `true` on resource X, you also have to set it to `true` on every resource that X depends on (if you forget, you'll get errors about cyclical dependencies). In the case of the launch configuration, that means you need to set `create_before_destroy` to `true` on the security group:

```
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"]
  }

  lifecycle {
    create_before_destroy = true
  }
}
```

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-example". The ASG references the launch configuration you created earlier using Terraform's interpolation syntax.

To make this ASG work, you need to specify one more parameter: `availability_zones`. This parameter tells the ASG 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 have an outage. You could hard-code the list of AZs (e.g., set it to `["us-east-1a", "us-east-1b"]`), but each AWS account has access to a slightly different set of AZs, so a better option is to use the `aws_availability_zones` data source to fetch the AZs specific to your AWS account:

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

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. There are data sources to not only get the list of availability zones, but also AMI IDs, IP address ranges, and the current user's identity.

To use the data source, you reference it using the following syntax:

```
"${data.TYPE.NAME.ATTRIBUTE}"
```

For example, here is how you pass the names of the AZs from the `aws_availability_zones` data source into the `availability_zones` parameter of the ASG:

```
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 several different servers, each with its own IP addresses, but you typically want to give your end users only a single IP to hit. 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, as shown in Figure 2-10.[12]

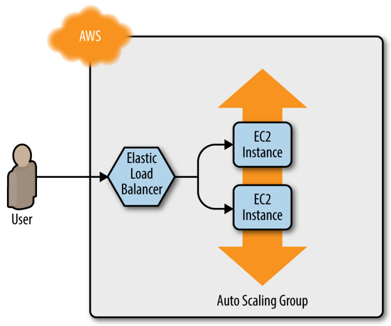


Figure 2-10. Use an Elastic Load Balancer to distribute traffic across the Auto Scaling Group

To create an ELB with Terraform, you use 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 work across all of the AZs in your account. Of course, this definition doesn't do much until you tell the ELB how to route requests. To do that, you add one or more *listeners* that specify what port the ELB 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}"]

  listener {
    lb_port           = 80
    lb_protocol       = "http"
    instance_port     = "${var.server_port}"
    instance_protocol = "http"
  }
}
```

This code tells the ELB 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, ELBs don't allow any incoming or outgoing traffic (just like EC2 Instances), so you need to create a new security group that explicitly allows incoming requests on port 80:

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

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

And now you need to tell the ELB to use this security group by using the `security_groups` parameter:

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

  listener {
    lb_port           = 80
    lb_protocol       = "http"
    instance_port     = "${var.server_port}"
    instance_protocol = "http"
  }
}
```

The ELB has one other 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. To configure a health check for the ELB, you add a `health_check` block. For example, here is a `health_check` block that sends an HTTP request every 30 seconds to the "/" URL of each of the EC2 Instances in the ASG and only considers an Instance healthy if it responds with a 200 OK:

```
resource "aws_elb" "example" {
  name               = "terraform-asg-example"
  availability_zones = ["${data.aws_availability_zones.all.names}"]
  security_groups    = ["${aws_security_group.elb.id}"]
  listener {
    lb_port           = 80
    lb_protocol       = "http"
    instance_port     = "${var.server_port}"
    instance_protocol = "http"
  }

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

To allow these health check requests, you need to modify the ELB's security group to allow outbound requests:

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

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

How does the ELB know which EC2 Instances to send requests to? You can attach a static list of EC2 Instances to an ELB using the ELB's `instances` parameter, but with an ASG, Instances may launch or terminate at any time, so a static list won't work. Instead, you can go back to the `aws_autoscaling_group` resource and set its `load_balancers` parameter to tell the ASG to register each Instance in the ELB when that Instance is booting:

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

  load_balancers    = ["${aws_elb.example.name}"]
  health_check_type = "ELB"

  min_size = 2
  max_size = 10

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

Notice that the `health_check_type` is now `"ELB"`. This tells the ASG to use the ELB's health check to determine if an Instance is healthy or not and to automatically replace Instances if the ELB reports them as unhealthy.

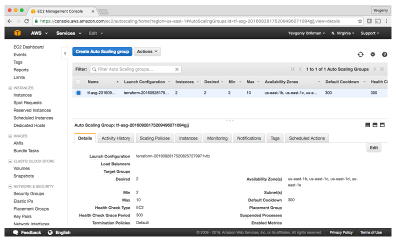
One last thing to do before deploying the load balancer: replace the old `public_ip` output of the single EC2 Instance you had before with an output that shows the DNS name of the ELB:

```
output "elb_dns_name" {
  value = "${aws_elb.example.dns_name}"
}
```

Run the `plan` command to verify your changes. You should see that your original single EC2 Instance is being removed and in its place, Terraform will create a launch configuration, ASG, ELB, and a security group. If the plan looks good, run `apply`. When `apply` completes, you should see the `elb_dns_name` output:

```
Outputs:
elb_dns_name = terraform-asg-example-123.us-east-1.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 ELB. 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, as shown in Figure 2-11.



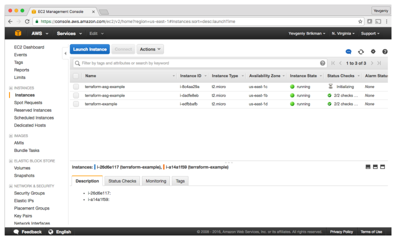
Name	Launch Configuration	Instances	Desired	Min	Max	Availability Zones	Default Cooldown	Health Cl
tf-asg-201609...	terraform-20160928175...	2	2	2	10	us-east-1b, us-east-1c, us-e...	300	300

Auto Scaling Group: tf-asg-20160928175209496071094gjj

Launch Configuration	terraform-20160928175208257279971vtb		
Load Balancers			
Target Groups			
Desired	2	Availability Zone(s)	us-east-1b, us-east-1c, us-east-1d, us-east-1e
Min	2	Subnet(s)	
Max	10	Default Cooldown	300
Health Check Type	EC2	Placement Group	
Health Check Grace Period	300	Suspended Processes	
Termination Policies	Default	Enabled Metrics	

Figure 2-11. The Auto Scaling Group

If you switch over to the Instances tab, you'll see the two EC Instances launching, as shown in Figure 2-12.



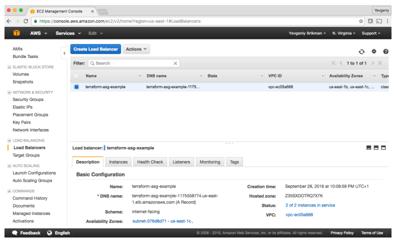
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Statu
terraform-asg-example	i-8c4aa29a	t2.micro	us-east-1c	running	Initializing	None
terraform-asg-example	i-dadfe8eb	t2.micro	us-east-1b	running	2/2 checks ...	None
terraform-example	i-edfbbafb	t2.micro	us-east-1d	running	2/2 checks ...	None

Instances: i-26d6e117 (terraform-example), i-a14a1f59 (terraform-example)

- i-26d6e117:
- i-a14a1f59:

Figure 2-12. The EC2 Instances in the ASG are launching

And finally, if you switch over to the Load Balancers tab, you'll see your ELB, as shown in Figure 2-13.



Name	DNS name	State	VPC ID	Availability Zones	Typ
terraform-asg-example	terraform-asg-example-1175...		vpc-ec05a688	us-east-1b, us-east-1c, ...	clas

Load balancer: terraform-asg-example

Basic Configuration

Name:	terraform-asg-example	Creation time:	September 28, 2016 at 10:09:59 PM UTC+1
* DNS name:	terraform-asg-example-1175558774.us-east-1.elb.amazonaws.com (A Record)	Hosted zone:	Z35SXDOTRQ7X7K
		Status:	2 of 2 instances in service
Scheme:	internet-facing	VPC:	vpc-ec05a688
Availability Zones:	subnet-076d8d71 - us-east-1c,		

Figure 2-13. The Elastic Load Balancer

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

```
> curl http://<elb_dns_name>
Hello, World
```

Success! The ELB 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!

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 ELB URL and you should get a 200 OK for each request, even while terminating an Instance, as the ELB 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 two Instances are running, and automatically launch a new one to replace it (self-healing!). You can also see how the ASG resizes itself by adding a `desired_capacity` parameter to your Terraform code and rerunning `apply`.

Cleanup

When you're done experimenting with Terraform, either at the end of this chapter, or at the end of future chapters, 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

Do you really want to destroy?
  Terraform will delete all your managed infrastructure.
```

```
    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 a minute or two, your AWS account should be clean again.

Note that later in the book, you will continue to evolve this example, so don't delete the Terraform code! However, feel free to run `destroy` on the actual deployed resources. After all, the beauty of infrastructure as code is that all of the information about those resources is captured in code, so you can re-create all of them at any time with a single command: `terraform apply`. In fact, you may want to commit your latest changes to Git so you can keep track of the history of your infrastructure.

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, interpolation, and dependencies allow you to remove duplication from your code and make it highly configurable.

However, you've only scratched the surface. In Chapter 3, you'll learn 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 Chapter 4, you'll see how to create reusable infrastructure with Terraform modules.

---

[1] If you find the AWS terminology confusing, be sure to check out AWS in Plain English.

[2] For more details on AWS user management best practices, see *http://amzn.to/2lvJ8Rf* (http://amzn.to/2lvJ8Rf).

[3] You can learn more about IAM Policies here: *http://amzn.to/2lQs1MA* (http://amzn.to/2lQs1MA).

[4] You can also write Terraform code in pure JSON in files with the extension *.tf.json*. You can learn more about Terraform's HCL and JSON syntax here: *https://www.terraform.io/docs/configuration/syntax.html*.

[5] You can learn more about AWS regions and availability zones here: *http://bit.ly/1NATGqS* (http://bit.ly/1NATGqS).

[6] You can find the full list of `aws_instance` configuration parameters here: *https://www.terraform.io/docs/providers/aws/r/instance.html*.

[7] You can find a handy list of HTTP server one-liners here: *https://gist.github.com/willurd/5720255*.

[8] To learn more about how CIDR works, see *http://bit.ly/2l8Ki9g* (http://bit.ly/2l8Ki9g). For a handy calculator that converts between IP address ranges and CIDR notation, see *http://www.ipaddressguide.com/cidr* (http://www.ipaddressguide.com/cidr).

[9] From *The Pragmatic Programmer* by Andy Hunt and Dave Thomas (Addison-Wesley Professional).

[10] Terraform allows you to specify numbers and booleans without quotes around them, but under the hood, it converts them all to strings. Numbers are converted more or less as you'd expect, where a 1 becomes "1". Booleans are first converted to a number and then a string, so a `true` becomes "1" and a `false` becomes "0".

[11] For a deeper look at how to build highly available and scalable systems on AWS, see: *http://bit.ly/2mpSXUZ* (http://bit.ly/2mpSXUZ).

[12] Although all the examples in this book use the ELB, AWS has recently released a new service called the Application Load Balancer (ALB) (http://amzn.to/2llkcee) that may be a better choice for HTTP-based services.