O'REILLY

Terraform

Up & Running

Writing Infrastructure as Code

Yevgeniy Brikman

O'REILLY®

Second Edition

Early Release

RAW & UNEDITED

Terraform

Up & Running

Writing Infrastructure as Code

Yevgeniy Brikman

Chapter 1. Why Terraform

Software isn't done when the code is working on your computer. It's not done when the tests pass. And it's not done when someone gives you a "ship it" on a code review. Software isn't done until you *deliver* it to the user.

Software delivery consists of all the work you need to do to make the code available to a customer, such as running that code on production servers, making the code resilient to outages and traffic spikes, and protecting the code from attackers. Before you dive into the details of Terraform, it's worth taking a step back to see where Terraform fits into the bigger picture of software delivery.

In this chapter, I'll dive into the following topics:

- The rise of DevOps
- What is infrastructure as code?
- Benefits of infrastructure as code
- How Terraform works
- How Terraform compares to other infrastructure as code tools

The Rise of DevOps

In the not-so-distant past, if you wanted to build a software company, you also had to manage a lot of hardware. You would set up cabinets and racks, load them up with servers, hook up wiring, install cooling, build

redundant power systems, and so on. It made sense to have one team, typically called Operations (“Ops”), dedicated to managing this hardware, and a separate team, typically called Developers (“Devs”), dedicated to writing the software.

The typical Dev team would build an application and “toss it over the wall” to the Ops team. It was then up to Ops to figure out how to deploy and run that application. Most of this was done manually. In part, that was unavoidable, because much of the work had to do with physically hooking up hardware (e.g., racking servers, hooking up network cables). But even the work Ops did in software, such as installing the application and its dependencies, was often done by manually executing commands on a server.

This works well for a while, but as the company grows, you eventually run into problems. It typically plays out like this: since releases are done manually, as the number of servers increases, releases become slow, painful, and unpredictable. The Ops team occasionally makes mistakes, so you end up with *snowflake servers*, where each one has a subtly different configuration from all the others (a problem known as *configuration drift*). As a result, the number of bugs increases. Developers shrug and say “It works on my machine!” Outages and downtime become more frequent.

The Ops team, tired from their pagers going off at 3 a.m. after every release, reduce the release cadence to once per week. Then to once per month. Then once every six months. Weeks before the biannual release, teams start trying to merge all their projects together, leading to a huge mess of merge conflicts. No one can stabilize the release branch. Teams start blaming each other. Silos form. The company grinds to a halt.

Nowadays, a profound shift is taking place. Instead of managing their own data centers, many companies are moving to the cloud, taking advantage of services such as Amazon Web Services, Azure, and Google Cloud. Instead of investing heavily in hardware, many Ops teams are spending all their time working on software, using tools such as Chef, Puppet, Terraform, and Docker. Instead of racking servers and plugging in network cables, many sysadmins are writing code.

As a result, both Dev and Ops spend most of their time working on software, and the distinction between the two teams is blurring. It may still make sense to have a separate Dev team responsible for the application code and an Ops team responsible for the operational code, but it's clear that Dev and Ops need to work more closely together. This is where the *DevOps movement* comes from.

DevOps isn't the name of a team or a job title or a particular technology. Instead, it's a set of processes, ideas, and techniques. Everyone has a slightly different definition of DevOps, but for this book, I'm going to go with the following:

The goal of DevOps is to make software delivery vastly more efficient.

Instead of multiday merge nightmares, you integrate code continuously and always keep it in a deployable state. Instead of deploying code once per month, you can deploy code dozens of times per day, or even after every single commit. And instead of constant outages and downtime, you build resilient, self-healing systems, and use monitoring and alerting to catch problems that can't be resolved automatically.

The results from companies that have undergone DevOps transformations are astounding. For example, Nordstrom found that after applying DevOps

practices to its organization, it was able to double the number of features it delivered per month, reduce defects by 50%, reduce *lead times* (the time from coming up with an idea to running code in production) by 60%, and reduce the number of production incidents by 60% to 90%. After HP's LaserJet Firmware division began using DevOps practices, the amount of time its developers spent on developing new features went from 5% to 40% and overall development costs were reduced by 40%. Etsy used DevOps practices to go from stressful, infrequent deployments that caused numerous outages to deploying 25 to 50 times per day, with far fewer outages.[1]

There are four core values in the DevOps movement: Culture, Automation, Measurement, and Sharing (sometimes abbreviated as the acronym CAMS). This book is not meant as a comprehensive overview of DevOps, so I will just focus on one of these values: automation.

The goal is to automate as much of the software delivery process as possible. That means that you manage your infrastructure not by clicking around a web page or manually executing shell commands, but through code. This is a concept that is typically called infrastructure as code.

What Is Infrastructure as Code?

The idea behind *infrastructure as code (IAC)* is that you write and execute code to define, deploy, and update your infrastructure. This represents an important shift in mindset where you treat all aspects of operations as software—even those aspects that represent hardware (e.g., setting up physical servers). In fact, a key insight of DevOps is that you can manage almost *everything* in code, including servers, databases, networks, log files, application configuration, documentation, automated tests,

deployment processes, and so on.

There are four broad categories of IAC tools:

- Ad hoc scripts
- Configuration management tools
- Server templating tools
- Server provisioning tools

Let’s look at these one at a time.

Ad Hoc Scripts

The most straightforward approach to automating anything is to write an *ad hoc script*. You take whatever task you were doing manually, break it down into discrete steps, use your favorite scripting language (e.g., Bash, Ruby, Python) to define each of those steps in code, and execute that script on your server, as shown in Figure 1-1.

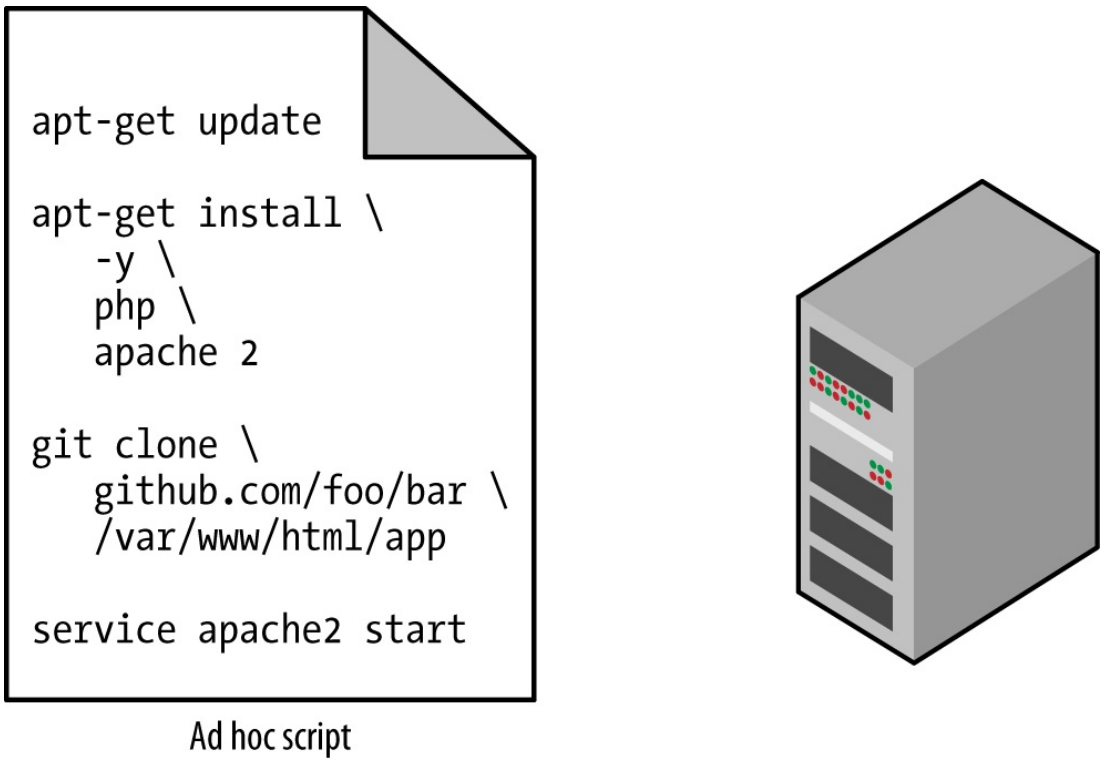

Figure 1-1. Running an ad hoc script on your server

For example, here is a Bash script called *setup-webserver.sh* that configures a web server by installing dependencies, checking out some code from a Git repo, and firing up the Apache web server:

```
# Update the apt-get cache
sudo apt-get update

# Install PHP and Apache
sudo apt-get install -y php apache2

# Copy the code from the repository
sudo git clone https://github.com/brikis98/php-app.git
/var/www/html/app

# Start Apache
sudo service apache2 start
```

The great thing about ad hoc scripts is that you can use popular, general-purpose programming languages and you can write the code however you

want. The terrible thing about ad hoc scripts is that you can use popular, general-purpose programming languages and you can write the code however you want.

Whereas tools that are purpose-built for IAC provide concise APIs for accomplishing complicated tasks, if you're using a general-purpose programming language, you have to write completely custom code for every task. Moreover, tools designed for IAC usually enforce a particular structure for your code, whereas with a general-purpose programming language, each developer will use his or her own style and do something different. Neither of these problems is a big deal for an eight-line script that installs Apache, but it gets messy if you try to use ad hoc scripts to manage dozens of servers, databases, load balancers, network configurations, and so on.

If you've ever had to maintain someone else's repository of ad hoc scripts, you know that it almost always devolves into a mess of unmaintainable spaghetti code. Ad hoc scripts are great for small, one-off tasks, but if you're going to be managing all of your infrastructure as code, then you should use an IAC tool that is purpose-built for the job.

Configuration Management Tools

Chef, Puppet, Ansible, and SaltStack are all *configuration management tools*, which means they are designed to install and manage software on existing servers. For example, here is an *Ansible Role* called *web-server.yml* that configures the same Apache web server as the *setup-webserver.sh* script:

```
- name: Update the apt-get cache
  apt:
```

```
    update_cache: yes

- name: Install PHP
  apt:
    name: php

- name: Install Apache
  apt:
    name: apache2

- name: Copy the code from the repository
  git: repo=https://github.com/brikis98/php-app.git
dest=/var/www/html/app

- name: Start Apache
  service: name=apache2 state=started enabled=yes
```

The code looks similar to the Bash script, but using a tool like Ansible offers a number of advantages:

Coding conventions

> Ansible enforces a consistent, predictable structure, including documentation, file layout, clearly named parameters, secrets management, and so on. While every developer organizes his or her ad hoc scripts in a different way, most configuration management tools come with a set of conventions that makes it easier to navigate the code.

Idempotence

> Writing an ad hoc script that works once isn't too difficult; writing an ad hoc script that works correctly even if you run it over and over again is a lot harder. Every time you go to create a folder in your script, you need to remember to check if that folder already exists; every time you add a line of configuration to a file, you need to check that line doesn't already exist; every time you want to run an app, you need to check that the app isn't already running.
>
> Code that works correctly no matter how many times you run it is

called *idempotent code*. To make the Bash script from the previous section idempotent, you'd have to add many lines of code, including lots of if-statements. Most Ansible functions, on the other hand, are idempotent by default. For example, the *web-server.yml* Ansible role will only install Apache if it isn't installed already and will only try to start the Apache web server if it isn't running already.

Distribution

Ad hoc scripts are designed to run on a single, local machine. Ansible and other configuration management tools are designed specifically for managing large numbers of remote servers, as shown in Figure 1-2.

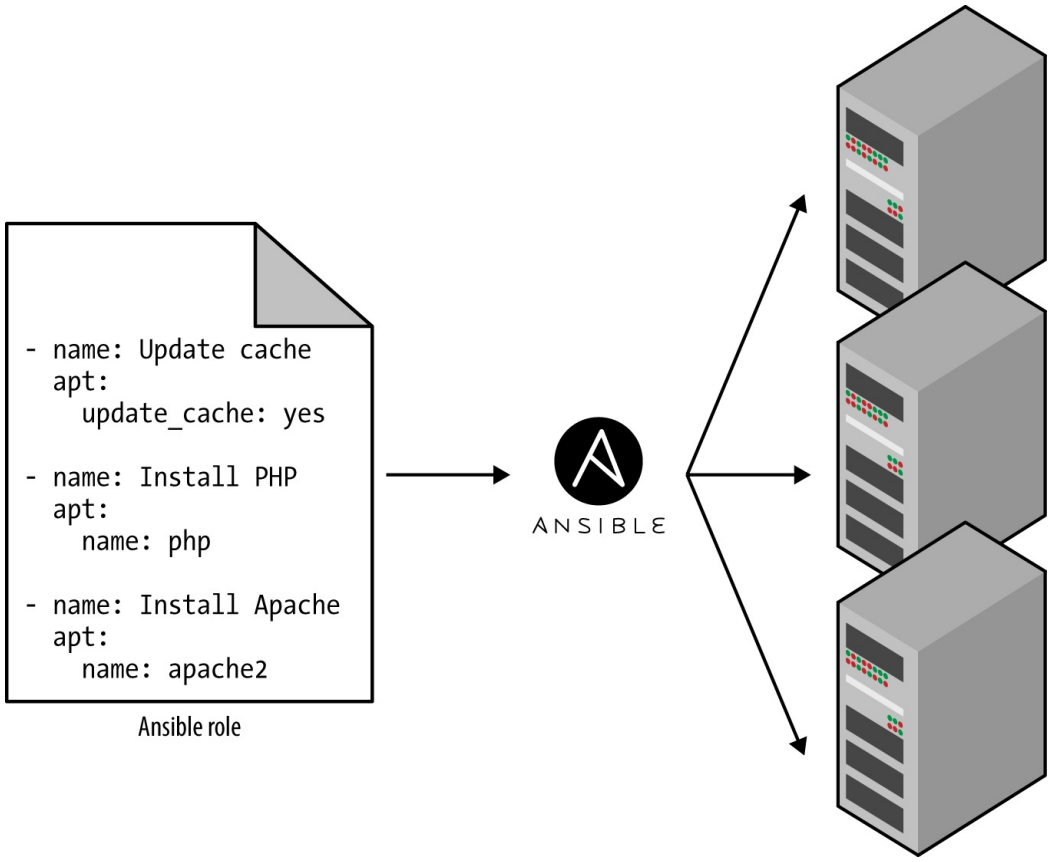

Figure 1-2. A configuration management tool like Ansible can execute your code across a large number of servers

For example, to apply the *web-server.yml* role to five servers, you first create a file called *hosts* that contains the IP addresses of those servers:

```
[webservers]
```

```
11.11.11.11
11.11.11.12
11.11.11.13
11.11.11.14
11.11.11.15
```

Next, you define the following *Ansible Playbook*:

```
- hosts: webservers
  roles:
  - webserver
```

Finally, you execute the playbook as follows:

```
ansible-playbook playbook.yml
```

This will tell Ansible to configure all five servers in parallel. Alternatively, by setting a single parameter called `serial` in the playbook, you can do a rolling deployment, which updates the servers in batches. For example, setting `serial` to 2 will tell Ansible to update two of the servers at a time, until all five are done. Duplicating any of this logic in an ad hoc script will take dozens or even hundreds of lines of code.

Server Templating Tools

An alternative to configuration management that has been growing in popularity recently are *server templating tools* such as Docker, Packer, and Vagrant. Instead of launching a bunch of servers and configuring them by running the same code on each one, the idea behind server templating tools is to create an *image* of a server that captures a fully self-contained "snapshot" of the operating system, the software, the files, and all other relevant details. You can then use some other IAC tool to install that image on all of your servers, as shown in Figure 1-3.

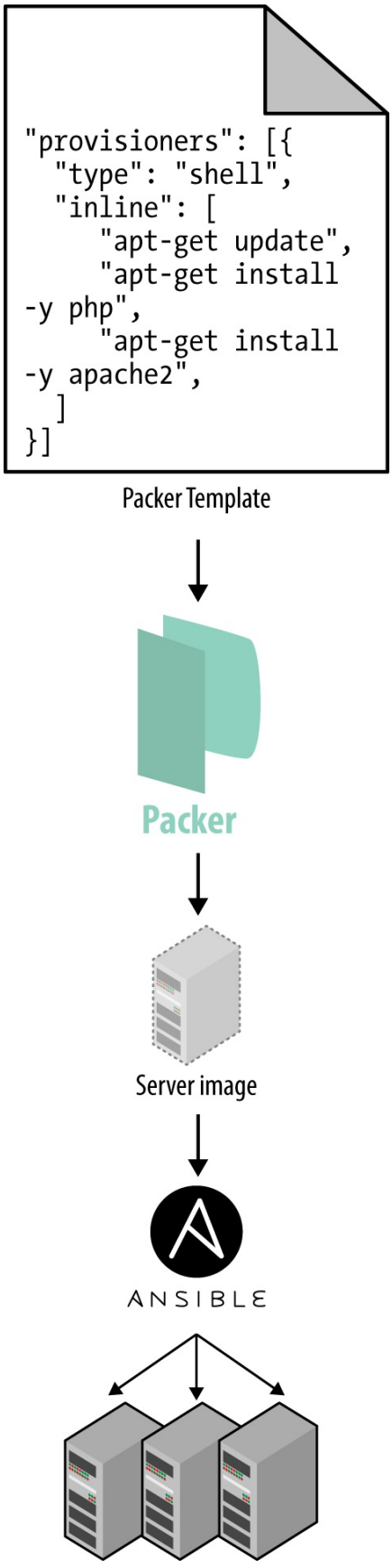
"provisioners": [{
"type": "shell",
"inline": [
"apt-get update",
"apt-get install
-y php",
"apt-get install
-y apache2",
]
}]
Packer Template
Packer
Server image
ANSIBLE

Figure 1-3. A server templating tool like Packer can be used to create a self-contained image of a server. You can then use other tools, such as Ansible, to install that image across all of your servers.

As shown in Figure 1-4, there are two broad categories of tools for working with images:

Virtual Machines

A *virtual machine (VM)* emulates an entire computer system, including the hardware. You run a *hypervisor*, such as VMWare, VirtualBox, or Parallels, to virtualize (i.e., simulate) the underlying CPU, memory, hard drive, and networking. The benefit of this is that any *VM Image* you run on top of the hypervisor can only see the virtualized hardware, so it's fully isolated from the host machine and any other VM Images, and will run exactly the same way in all environments (e.g., your computer, a QA server, a production server, etc). The drawback is that virtualizing all this hardware and running a totally separate operating system for each VM incurs a lot of overhead in terms of CPU usage, memory usage, and startup time. You can define VM Images as code using tools such as Packer and Vagrant.

Containers

A *container* emulates the user space of an operating system.[2] You run a *container engine*, such as Docker or CoreOS rkt, to create isolated processes, memory, mount points, and networking. The benefit of this is that any container you run on top of the container engine can only see its own user space, so it's isolated from the host machine and other containers, and will run exactly the same way in all environments (e.g., your computer, a QA server, a production server, etc.). The drawback is that all the containers running on a single server share that server's operating system kernel and hardware, so the isolation is not as secure as with VMs.[3] However, because the kernel and hardware are shared, your containers can boot up in milliseconds and have virtually no CPU or memory overhead. You can define Container Images as code using tools such as Docker and CoreOs rkt.

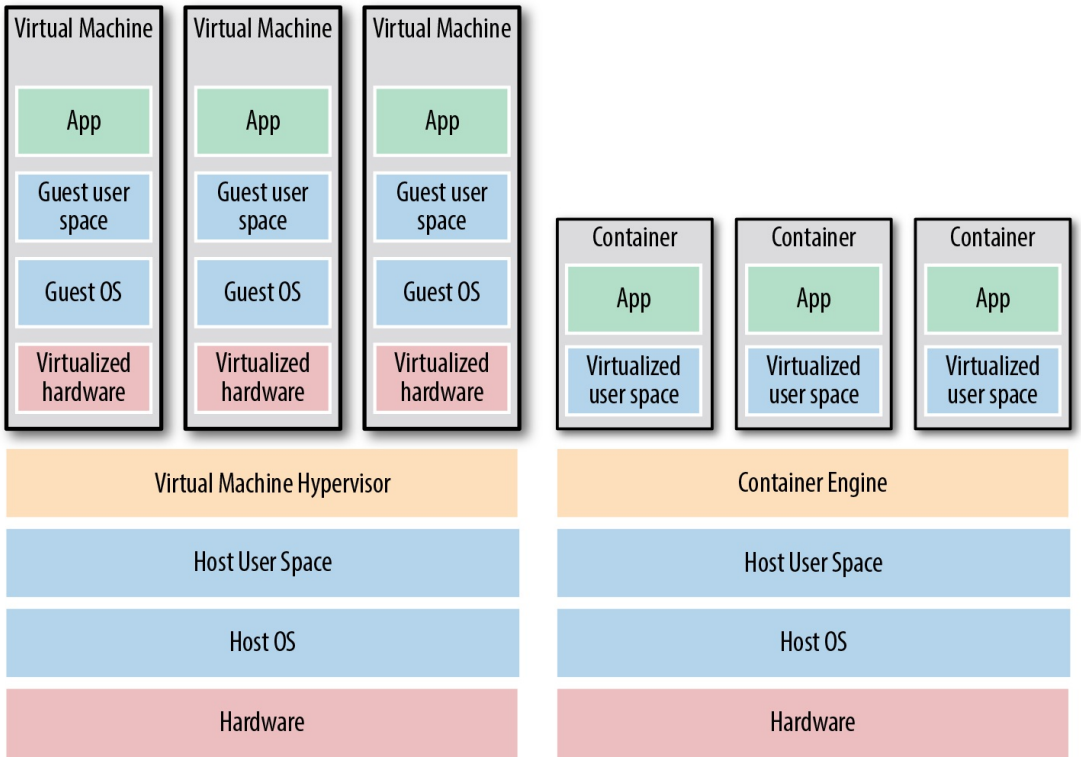

Figure 1-4. The two main types of images: VMs, on the left, and containers, on the right. VMs virtualize the hardware, whereas containers only virtualize user space.

For example, here is a Packer template called *web-server.json* that creates an *Amazon Machine Image* (AMI), which is a VM Image you can run on Amazon Web Services (AWS):

```
{
  "builders": [{
    "ami_name": "packer-example",
    "instance_type": "t2.micro",
    "region": "us-east-2",
    "type": "amazon-ebs",
    "source_ami": "ami-0c55b159cbfafe1f0",
    "ssh_username": "ubuntu"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sudo apt-get update",
```

```
      "sudo apt-get install -y php apache2",
      "sudo git clone https://github.com/brikis98/php-
app.git /var/www/html/app"
    ],
    "environment_vars": [
      "DEBIAN_FRONTEND=noninteractive"
    ]
  }]
}
```

This Packer template configures the same Apache web server you saw in *setup-webserver.sh* using the same Bash code.[4] The only difference between the preceding code and previous examples is that this Packer template does not start the Apache web server (e.g., by calling `sudo service apache2 start`). That's because server templates are typically used to install software in images, but it's only when you run the image (e.g., by deploying it on a server) that you should actually run that software.

You can build an AMI from this template by running `packer build webserver.json`, and once the build completes, you can install that AMI on all of your AWS servers, configure each server to run Apache when the server is booting (you'll see an example of this in the next section), and they will all run exactly the same way.

Note that the different server templating tools have slightly different purposes. Packer is typically used to create images that you run directly on top of production servers, such as an AMI that you run in your production AWS account. Vagrant is typically used to create images that you run on your development computers, such as a VirtualBox image that you run on your Mac or Windows laptop. Docker is typically used to create images of

individual applications. You can run the Docker images on production or development computers, so long as some other tool has configured that computer with the Docker Engine. For example, a common pattern is to use Packer to create an AMI that has the Docker Engine installed, deploy that AMI on a cluster of servers in your AWS account, and then deploy individual Docker containers across that cluster to run your applications.

Server templating is a key component of the shift to *immutable infrastructure*. This idea is inspired by functional programming, where variables are immutable, so once you've set a variable to a value, you can never change that variable again. If you need to update something, you create a new variable. Since variables never change, it's a lot easier to reason about your code.

The idea behind immutable infrastructure is similar: once you've deployed a server, you never make changes to it again. If you need to update something (e.g., deploy a new version of your code), you create a new image from your server template and you deploy it on a new server. Since servers never change, it's a lot easier to reason about what's deployed.

Server Provisioning Tools

Whereas configuration management and server templating tools define the code that runs on each server, *server provisioning tools* such as Terraform, CloudFormation, and OpenStack Heat are responsible for creating the servers themselves. In fact, you can use provisioning tools to not only create servers, but also databases, caches, load balancers, queues, monitoring, subnet configurations, firewall settings, routing rules, SSL certificates, and almost every other aspect of your infrastructure, as shown in Figure 1-5.

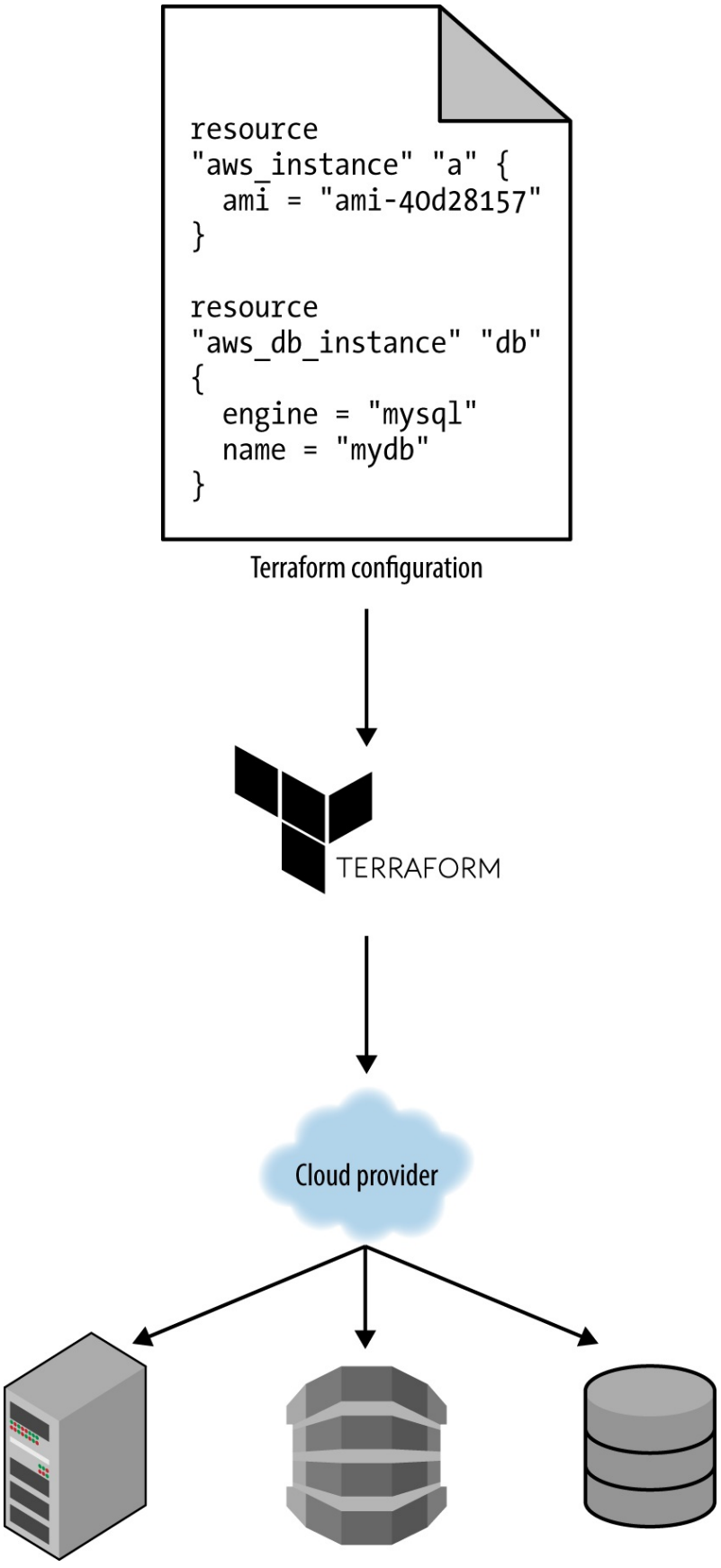
resource
"aws_instance" "a" {
ami = "ami-40d28157"
}
resource
"aws_db_instance" "db"
{
engine = "mysql"
name = "mydb"
}
Terraform configuration
TERRAFORM
Cloud provider

Figure 1-5. Server provisioning tools can be used with your cloud provider to create servers, databases, load balancers, and all other parts of your infrastructure.

For example, the following code deploys a web server using Terraform:

```
resource "aws_instance" "app" {
  instance_type     = "t2.micro"
  availability_zone = "us-east-2a"
  ami               = "ami-0c55b159cbfafe1f0"

  user_data = <<-EOF
              #!/bin/bash
              sudo service apache2 start
              EOF
}
```

Don't worry if some of the syntax isn't familiar to you yet. For now, just focus on two parameters:

`ami`

: This parameter specifies the ID of an AMI to deploy on the server. You could set this parameter to the ID of an AMI built from the *web-server.json* Packer template in the previous section, which has PHP, Apache, and the application source code.

`user_data`

: This is a Bash script that executes when the web server is booting. The preceding code uses this script to boot up Apache.

In other words, this code shows you server provisioning and server templating working together, which is a common pattern in immutable infrastructure.

Benefits of Infrastructure as Code

Now that you've seen all the different flavors of infrastructure as code, a good question to ask is, why bother? Why learn a bunch of new languages and tools and encumber yourself with more code to manage?

The answer is that code is powerful. In exchange for the up-front investment of converting your manual practices to code, you get dramatic improvements in your ability to deliver software. According to the 2016 State of DevOps Report, organizations that use DevOps practices, such as IAC, deploy 200 times more frequently, recover from failures 24 times faster, and have lead times that are 2,555 times lower.

When your infrastructure is defined as code, you are able to use a wide variety of software engineering practices to dramatically improve your software delivery process, including:

Self-service

> Most teams that deploy code manually have a small number of sysadmins (often, just one) who are the only ones who know all the magic incantations to make the deployment work and are the only ones with access to production. This becomes a major bottleneck as the company grows. If your infrastructure is defined in code, then the entire deployment process can be automated, and developers can kick off their own deployments whenever necessary.

Speed and safety

> If the deployment process is automated, it'll be significantly faster, since a computer can carry out the deployment steps far faster than a person; and safer, since an automated process will be more consistent, more repeatable, and not prone to manual error.

Documentation

> Instead of the state of your infrastructure being locked away in a single

sysadmin's head, you can represent the state of your infrastructure in source files that anyone can read. In other words, IAC acts as documentation, allowing everyone in the organization to understand how things work, even if the sysadmin goes on vacation.

Version control

You can store your IAC source files in version control, which means the entire history of your infrastructure is now captured in the commit log. This becomes a powerful tool for debugging issues, as any time a problem pops up, your first step will be to check the commit log and find out what changed in your infrastructure, and your second step may be to resolve the problem by simply reverting back to a previous, known-good version of your IAC code.

Validation

If the state of your infrastructure is defined in code, then for every single change, you can perform a code review, run a suite of automated tests, and pass the code through static analysis tools, all practices that are known to significantly reduce the chance of defects.

Reuse

You can package your infrastructure into reusable modules, so that instead of doing every deployment for every product in every environment from scratch, you can build on top of known, documented, battle-tested pieces.[5]

Happiness

There is one other very important, and often overlooked, reason for why you should use IAC: happiness. Deploying code and managing infrastructure manually is repetitive and tedious. Developers and sysadmins resent this type of work, as it involves no creativity, no challenge, and no recognition. You could deploy code perfectly for months, and no one will take notice—until that one day when you mess it up. That creates a stressful and unpleasant environment. IAC offers a better alternative that allows computers to do what they do best (automation) and developers to do what they do best (coding).

Now that you have a sense of why IAC is important, the next question is whether Terraform is the right IAC tool for you. To answer that, I'm first going to do a very quick primer on how Terraform works, and then I'll compare it to the other popular IAC options out there, such as Chef, Puppet, and Ansible.

How Terraform Works

Here is a high-level and somewhat simplified view of how Terraform works. Terraform is an open source tool created by HashiCorp and written in the Go programming language. The Go code compiles down into a single binary (or rather, one binary for each of the supported operating systems) called, not surprisingly, `terraform`.

You can use this binary to deploy infrastructure from your laptop or a build server or just about any other computer, and you don't need to run any extra infrastructure to make that happen. That's because under the hood, the `terraform` binary makes API calls on your behalf to one or more *providers*, such as Amazon Web Services (AWS), Azure, Google Cloud, DigitalOcean, OpenStack, etc. That means Terraform gets to leverage the infrastructure those providers are already running for their API servers, as well as the authentication mechanisms you're already using with those providers (e.g., the API keys you already have for AWS).

How does Terraform know what API calls to make? The answer is that you create *Terraform configurations*, which are text files that specify what infrastructure you wish to create. These configurations are the "code" in "infrastructure as code." Here's an example Terraform configuration:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
```

```
  instance_type = "t2.micro"
}

resource "google_dns_record_set" "a" {
  name         = "demo.google-example.com"
  managed_zone = "example-zone"
  type         = "A"
  ttl          = 300
  rrdatas      = [aws_instance.example.public_ip]
}
```

Even if you've never seen Terraform code before, you shouldn't have too much trouble reading it. This snippet tells Terraform to make API calls to AWS to deploy a server and then make API calls to Google Cloud to create a DNS entry pointing to the AWS server's IP address. In just a single, simple syntax, Terraform allows you to deploy interconnected resources across multiple cloud providers.

You can define your entire infrastructure—servers, databases, load balancers, network topology, and so on—in Terraform configuration files and commit those files to version control. You then run certain Terraform commands, such as `terraform apply`, to deploy that infrastructure. The `terraform` binary parses your code, translates it into a series of API calls to the cloud providers specified in the code, and makes those API calls as efficiently as possible on your behalf, as shown in Figure 1-6.

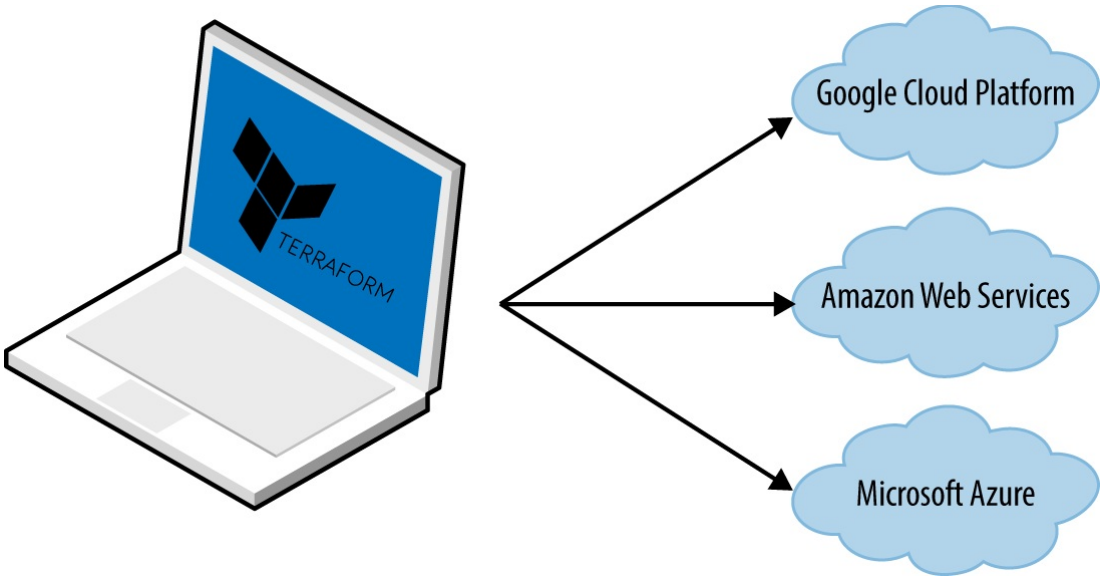

Figure 1-6. Terraform is a binary that translates the contents of your configurations into API calls to cloud providers

When someone on your team needs to make changes to the infrastructure, instead of updating the infrastructure manually and directly on the servers, they make their changes in the Terraform configuration files, validate those changes through automated tests and code reviews, commit the updated code to version control, and then run the `terraform apply` command to have Terraform make the necessary API calls to deploy the changes.

TRANSPARENT PORTABILITY BETWEEN CLOUD PROVIDERS

Since Terraform supports many different cloud providers, a common question that comes up is whether it supports *transparent portability* between them. For example, if you used Terraform to define a bunch of servers, databases, load balancers, and other infrastructure in AWS, could you tell Terraform to deploy exactly the same infrastructure in another cloud provider, such as Azure or Google Cloud, in just a few clicks?

This question turns out to be a bit of a red herring. The reality is that you can't deploy "exactly the same infrastructure" in a different cloud provider

because the cloud providers don't offer the same types of infrastructure! The servers, load balancers, and databases offered by AWS are very different than those in Azure and Google Cloud in terms of features, configuration, management, security, scalability, availability, observability, and so on. There is no way to "transparently" paper over these differences, especially as functionality in one cloud provider often doesn't exist at all in the others.

Terraform's approach is to allow you to write code that is specific to each provider, taking advantage of that provider's unique functionality, but to use the same language, toolset, and infrastructure as code practices under the hood for all providers.

How Terraform Compares to Other Infrastructure as Code Tools

Infrastructure as code is wonderful, but the process of picking an IAC tool is not. Many of the IAC tools overlap in what they do. Many of them are open source. Many of them offer commercial support. Unless you've used each one yourself, it's not clear what criteria you should use to pick one or the other.

What makes this even harder is that most of the comparisons you find between these tools do little more than list the general properties of each one and make it sound like you could be equally successful with any of them. And while that's technically true, it's not helpful. It's a bit like telling a programming newbie that you could be equally successful building a website with PHP, C, or assembly—a statement that's technically true, but one that omits a huge amount of information that is essential for making a good decision.

In the following sections, I'm going to do a detailed comparison between

the most popular configuration management and provisioning tools: Terraform, Chef, Puppet, Ansible, SaltStack, CloudFormation, and OpenStack Heat. My goal is to help you decide if Terraform is a good choice by explaining why my company, Gruntwork, picked Terraform as our IAC tool of choice and, in some sense, why I wrote this book.[6] As with all technology decisions, it's a question of trade-offs and priorities, and while your particular priorities may be different than mine, I hope that sharing this thought process will help you make your own decision.

Here are the main trade-offs to consider:

- Configuration management versus provisioning
- Mutable infrastructure versus immutable infrastructure
- Procedural language versus declarative language
- Master versus masterless
- Agent versus agentless
- Large community versus small community
- Mature versus cutting-edge

Configuration Management Versus Provisioning

As you saw earlier, Chef, Puppet, Ansible, and SaltStack are all configuration management tools, whereas CloudFormation, Terraform, and OpenStack Heat are all provisioning tools. Although the distinction is not entirely clear cut, as configuration management tools can typically do some degree of provisioning (e.g., you can deploy a server with Ansible) and provisioning tools can typically do some degree of configuration (e.g., you can run configuration scripts on each server you provision with Terraform), you typically want to pick the tool that's the best fit for your

use case.[7]

In particular, if you use server templating tools such as Docker or Packer, the vast majority of your configuration management needs are already taken care of. Once you have an image created from a Dockerfile or Packer template, all that's left to do is provision the infrastructure for running those images. And when it comes to provisioning, a server provisioning tool is going to be your best choice.

That said, if you're not using server templating tools, a good alternative is to use a configuration management and provisioning tool together. For example, you might use Terraform to provision your servers and run Chef to configure each one.

Mutable Infrastructure Versus Immutable Infrastructure

Configuration management tools such as Chef, Puppet, Ansible, and SaltStack typically default to a mutable infrastructure paradigm. For example, if you tell Chef to install a new version of OpenSSL, it'll run the software update on your existing servers and the changes will happen in place. Over time, as you apply more and more updates, each server builds up a unique history of changes. As a result, each server becomes slightly different than all the others, leading to subtle configuration bugs that are difficult to diagnose and reproduce (this is the same configuration drift problem that happens when you manage servers manually, although it's much less problematic when using a configuration management tool). Even with automated tests these bugs are hard to catch, as a configuration management change may work just fine on a test server, but that same change may behave differently on a production server because the

production server has accumulated months of changes that aren't reflected in the test environment.

If you're using a provisioning tool such as Terraform to deploy machine images created by Docker or Packer, then most "changes" are actually deployments of a completely new server. For example, to deploy a new version of OpenSSL, you would use Packer to create a new image with the new version of OpenSSL, deploy that image across a set of new servers, and then undeploy the old servers. Since every deployment uses immutable images on fresh servers, this approach reduces the likelihood of configuration drift bugs, makes it easier to know exactly what software is running on each server, and allows you to easily deploy any previous version of the software (any previous image) at any time. It also makes your automated testing more effective, as an immutable image that passes your tests in the test environment is likely to behave exactly the same way in the production environment.

Of course, it's possible to force configuration management tools to do immutable deployments too, but it's not the idiomatic approach for those tools, whereas it's a natural way to use provisioning tools. It's also worth mentioning that the immutable approach has downsides of its own. For example, rebuilding an image from a server template and redeploying all your servers for a trivial change can take a long time. Moreover, immutability only lasts until you actually run the image. Once a server is up and running, it'll start making changes on the hard drive and experiencing some degree of configuration drift (although this is mitigated if you deploy frequently).

Procedural Language Versus Declarative Language

Chef and Ansible encourage a *procedural* style where you write code that specifies, step by step, how to achieve some desired end state. Terraform, CloudFormation, SaltStack, Puppet, and Open Stack Heat all encourage a more *declarative* style where you write code that specifies your desired end state, and the IAC tool itself is responsible for figuring out how to achieve that state.

To demonstrate the difference, let's go through an example. Imagine you wanted to deploy 10 servers ("EC2 Instances" in AWS lingo) to run an AMI with ID `ami-0c55b159cbfafe1f0` (Ubuntu 18.04). Here is a simplified example of an Ansible template that does this using a procedural approach:

```
- ec2:
    count: 10
    image: ami-0c55b159cbfafe1f0
    instance_type: t2.micro
```

And here is a simplified example of a Terraform configuration that does the same thing using a declarative approach:

```
resource "aws_instance" "example" {
  count         = 10
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Now at the surface, these two approaches may look similar, and when you initially execute them with Ansible or Terraform, they will produce similar results. The interesting thing is what happens when you want to make a change.

For example, imagine traffic has gone up and you want to increase the number of servers to 15. With Ansible, the procedural code you wrote earlier is no longer useful; if you just updated the number of servers to 15 and reran that code, it would deploy 15 new servers, giving you 25 total! So instead, you have to be aware of what is already deployed and write a totally new procedural script to add the 5 new servers:

```
- ec2:
    count: 5
    image: ami-0c55b159cbfafe1f0
    instance_type: t2.micro
```

With declarative code, since all you do is declare the end state you want, and Terraform figures out how to get to that end state, Terraform will also be aware of any state it created in the past. Therefore, to deploy 5 more servers, all you have to do is go back to the same Terraform configuration and update the count from 10 to 15:

```
resource "aws_instance" "example" {
  count         = 15
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

If you applied this configuration, Terraform would realize it had already created 10 servers and therefore that all it needed to do was create 5 new servers. In fact, before applying this configuration, you can use Terraform's `plan` command to preview what changes it would make:

```
$ terraform plan

# aws_instance.example[11] will be created
+ resource "aws_instance" "example" {
    + ami            = "ami-0c55b159cbfafe1f0"
```

```
    + instance_type  = "t2.micro"
    + (...)
  }

# aws_instance.example[12] will be created
+ resource "aws_instance" "example" {
    + ami            = "ami-0c55b159cbfafe1f0"
    + instance_type  = "t2.micro"
    + (...)
  }

# aws_instance.example[13] will be created
+ resource "aws_instance" "example" {
    + ami            = "ami-0c55b159cbfafe1f0"
    + instance_type  = "t2.micro"
    + (...)
  }

# aws_instance.example[14] will be created
+ resource "aws_instance" "example" {
    + ami            = "ami-0c55b159cbfafe1f0"
    + instance_type  = "t2.micro"
    + (...)
  }

Plan: 5 to add, 0 to change, 0 to destroy.
```

Now what happens when you want to deploy a different version of the app, such as AMI ID `ami-02bcbb802e03574ba`? With the procedural approach, both of your previous Ansible templates are again not useful, so you have to write yet another template to track down the 10 servers you deployed previously (or was it 15 now?) and carefully update each one to the new version. With the declarative approach of Terraform, you go back to the exact same configuration file once again and simply change the `ami` parameter to `ami-02bcbb802e03574ba`:

```
resource "aws_instance" "example" {
  count         = 15
  ami           = "ami-02bcbb802e03574ba"
```

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

Obviously, these examples are simplified. Ansible does allow you to use tags to search for existing EC2 Instances before deploying new ones (e.g., using the `instance_tags` and `count_tag` parameters), but having to manually figure out this sort of logic for every single resource you manage with Ansible, based on each resource's past history, can be surprisingly complicated (e.g., finding existing instances not only by tag, but also image version, availability zone, etc.). This highlights two major problems with procedural IAC tools:

1. *Procedural code does* not *fully capture the state of the infrastructure*. Reading through the three preceding Ansible templates is not enough to know what's deployed. You'd also have to know the *order* in which those templates were applied. Had you applied them in a different order, you might have ended up with different infrastructure, and that's not something you can see in the code base itself. In other words, to reason about an Ansible or Chef codebase, you have to know the full history of every change that has ever happened.
2. *Procedural code limits reusability*. The reusability of procedural code is inherently limited because you have to manually take into account the current state of the infrastructure. Since that state is constantly changing, code you used a week ago may no longer be usable because it was designed to modify a state of your infrastructure that no longer exists. As a result, procedural codebases tend to grow large and complicated over time.

With Terraform's declarative approach, the code always represents the latest state of your infrastructure. At a glance, you can tell what's currently deployed and how it's configured, without having to worry about history or timing. This also makes it easy to create reusable code, as you don't

have to manually account for the current state of the world. Instead, you just focus on describing your desired state, and Terraform figures out how to get from one state to the other automatically. As a result, Terraform codebases tend to stay small and easy to understand.

Of course, there are downsides to declarative languages too. Without access to a full programming language, your expressive power is limited. For example, some types of infrastructure changes, such as a zero-downtime deployment, are hard to express in purely declarative terms (but not impossible, as you'll see in Chapter 5). Similarly, without the ability to do "logic" (e.g., if-statements, loops), creating generic, reusable code can be tricky. Fortunately, Terraform provides a number of powerful primitives—such as input variables, output variables, modules, `create_before_destroy`, `count`, ternary syntax, and built-in functions—that make it possible to create clean, configurable, modular code even in a declarative language. I'll come back to these topics in Chapter 4 and Chapter 5.

Master Versus Masterless

By default, Chef, Puppet, and SaltStack all require that you run a *master server* for storing the state of your infrastructure and distributing updates. Every time you want to update something in your infrastructure, you use a client (e.g., a command-line tool) to issue new commands to the master server, and the master server either pushes the updates out to all the other servers, or those servers pull the latest updates down from the master server on a regular basis.

A master server offers a few advantages. First, it's a single, central place where you can see and manage the status of your infrastructure. Many

configuration management tools even provide a web interface (e.g., the Chef Console, Puppet Enterprise Console) for the master server to make it easier to see what's going on. Second, some master servers can run continuously in the background, and enforce your configuration. That way, if someone makes a manual change on a server, the master server can revert that change to prevent configuration drift.

However, having to run a master server has some serious drawbacks:

Extra infrastructure

: You have to deploy an extra server, or even a cluster of extra servers (for high availability and scalability), just to run the master.

Maintenance

: You have to maintain, upgrade, back up, monitor, and scale the master server(s).

Security

: You have to provide a way for the client to communicate to the master server(s) and a way for the master server(s) to communicate with all the other servers, which typically means opening extra ports and configuring extra authentication systems, all of which increases your surface area to attackers.

Chef, Puppet, and SaltStack do have varying levels of support for masterless modes where you just run their agent software on each of your servers, typically on a periodic schedule (e.g., a cron job that runs every 5 minutes), and use that to pull down the latest updates from version control (rather than from a master server). This significantly reduces the number of moving parts, but, as discussed in the next section, this still leaves a number of unanswered questions, especially about how to provision the servers and install the agent software on them in the first place.

Ansible, CloudFormation, Heat, and Terraform are all masterless by default. Or, to be more accurate, some of them may rely on a master server, but it's already part of the infrastructure you're using and not an extra piece you have to manage. For example, Terraform communicates with cloud providers using the cloud provider's APIs, so in some sense, the API servers are master servers, except they don't require any extra infrastructure or any extra authentication mechanisms (i.e., just use your API keys). Ansible works by connecting directly to each server over SSH, so again, you don't have to run any extra infrastructure or manage extra authentication mechanisms (i.e., just use your SSH keys).

Agent Versus Agentless

Chef, Puppet, and SaltStack all require you to install *agent software* (e.g., Chef Client, Puppet Agent, Salt Minion) on each server you want to configure. The agent typically runs in the background on each server and is responsible for installing the latest configuration management updates.

This has a few drawbacks:

Bootstrapping

: How do you provision your servers and install the agent software on them in the first place? Some configuration management tools kick the can down the road, assuming some external process will take care of this for them (e.g., you first use Terraform to deploy a bunch of servers with an AMI that has the agent already installed); other configuration management tools have a special bootstrapping process where you run one-off commands to provision the servers using the cloud provider APIs and install the agent software on those servers over SSH.

Maintenance

: You have to carefully update the agent software on a periodic basis,

being careful to keep it in sync with the master server if there is one. You also have to monitor the agent software and restart it if it crashes.

Security

If the agent software pulls down configuration from a master server (or some other server if you're not using a master), then you have to open outbound ports on every server. If the master server pushes configuration to the agent, then you have to open inbound ports on every server. In either case, you have to figure out how to authenticate the agent to the server it's talking to. All of this increases your surface area to attackers.

Once again, Chef, Puppet, and SaltStack do have varying levels of support for agentless modes (e.g., salt-ssh), but these always feel like they were tacked on as an afterthought and don't support the full feature set of the configuration management tool. That's why in the wild, the default or idiomatic configuration for Chef, Puppet, and SaltStack almost always includes an agent and usually a master too, as shown in Figure 1-7.

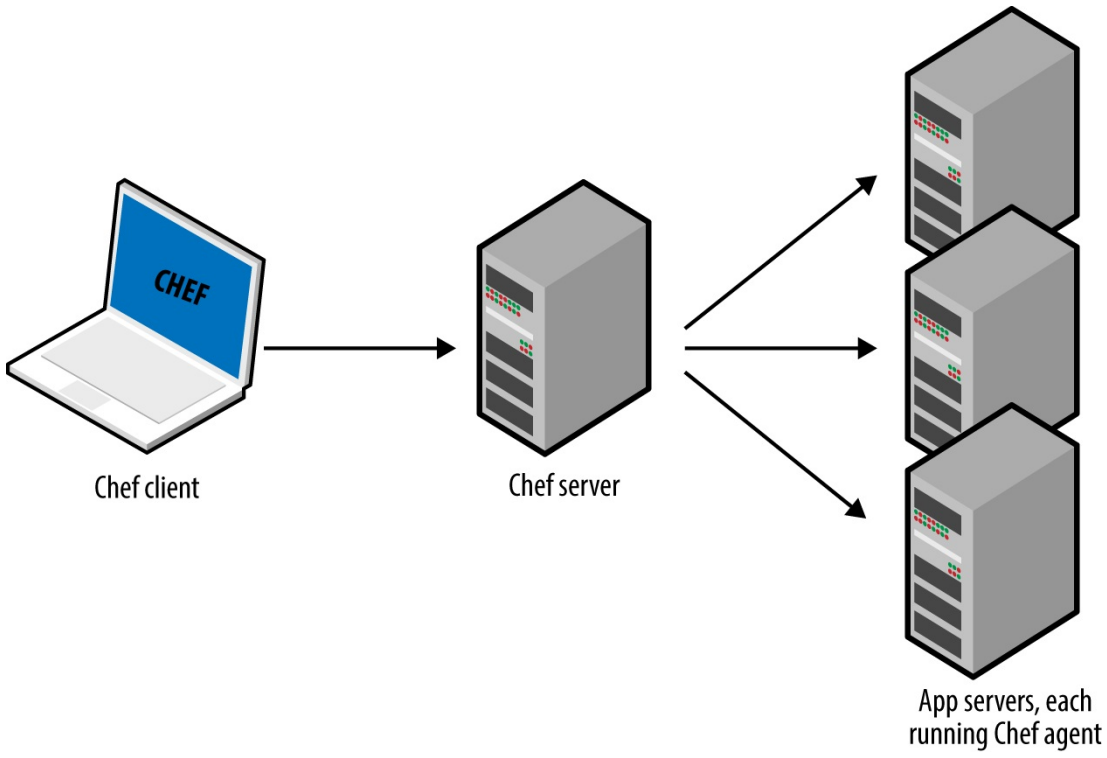

Figure 1-7. The typical architecture for Chef, Puppet, and SaltStack involves many moving parts. For example, the default setup for Chef is to run the Chef client on your computer, which talks to a Chef master server, which deploys changes by talking to Chef agents running on all your other servers.

All of these extra moving parts introduce a large number of new failure modes into your infrastructure. Each time you get a bug report at 3 a.m., you'll have to figure out if it's a bug in your application code, or your IAC code, or the configuration management client, or the master server(s), or the way the client talks to the master server(s), or the way other servers talk to the master server(s), or…

Ansible, CloudFormation, Heat, and Terraform do not require you to install any extra agents. Or, to be more accurate, some of them require agents, but these are typically already installed as part of the infrastructure you're using. For example, AWS, Azure, Google Cloud, and all other cloud providers take care of installing, managing, and authenticating agent software on each of their physical servers. As a user of Terraform, you don't have to worry about any of that: you just issue commands and the cloud provider's agents execute them for you on all of your servers, as shown in Figure 1-8. With Ansible, your servers need to run the SSH Daemon, which is common to run on most servers anyway.

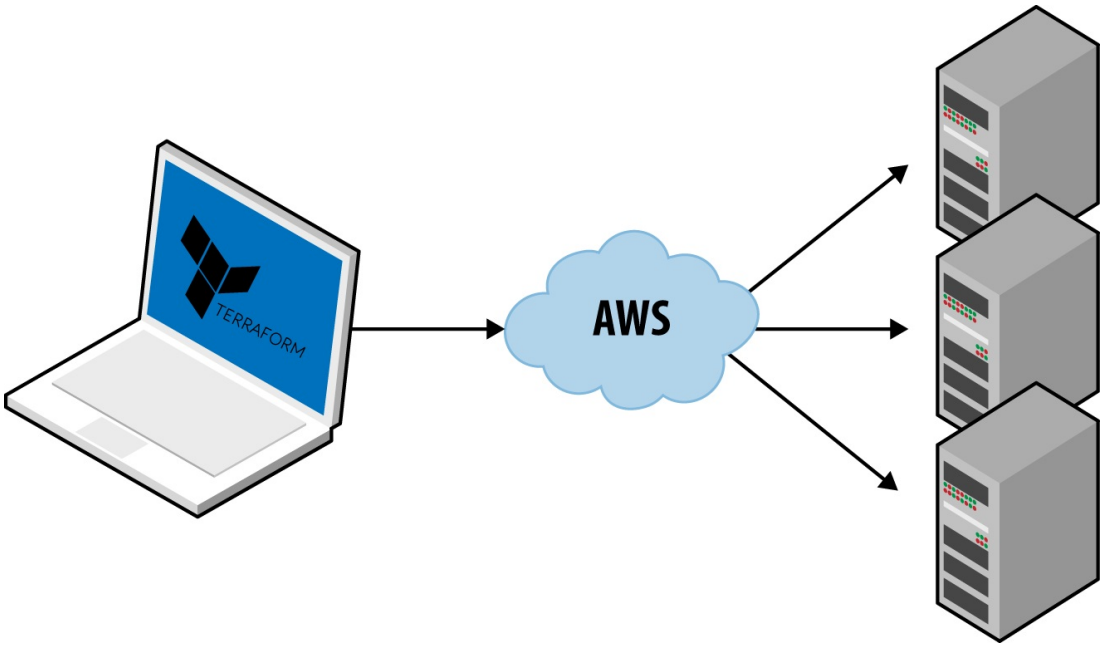

Figure 1-8. Terraform uses a masterless, agent-only architecture. All you need to run is the Terraform client and it takes care of the rest by using the APIs of cloud providers, such as AWS.

Large Community Versus Small Community

Whenever you pick a technology, you are also picking a community. In many cases, the ecosystem around the project can have a bigger impact on your experience than the inherent quality of the technology itself. The community determines how many people contribute to the project, how many plug-ins, integrations, and extensions are available, how easy it is to find help online (e.g., blog posts, questions on StackOverflow), and how easy it is to hire someone to help you (e.g., an employee, consultant, or support company).

It's hard to do an accurate comparison between communities, but you can spot some trends by searching online. Table 1-1 shows a comparison of popular IAC tools, with data I gathered during May 2019, including whether the IAC tool is open source or closed source, what cloud providers it supports, the total number of contributors and stars on GitHub,

how many commits and active issues there were over a one-month period from mid April to mid May, how many open source libraries are available for the tool, the number of questions listed for that tool on StackOverflow, and the number of jobs that mention the tool on Indeed.com.[8]

Table 1-1. A comparison of IAC communities

	Source	Cloud	Contributors	Stars	Commits (1 month)	Bugs (1 month
Chef	Open	All	562	5,794	435	86
Puppet	Open	All	515	5,299	94	314[c]
Ansible	Open	All	4,386	37,161	506	523
SaltStack	Open	All	2,237	9,901	608	441
CloudFormation	Closed	AWS	?	?	?	?
Heat	Open	All	361	349	12	600[i]
Terraform	Open	All	1,261	16,837	173	204

[a] This is the number of cookbooks in the Chef Supermarket.

[b] To avoid false positives for the term “chef”, I searched for “chef devops”.

[c] Based on the Puppet Labs JIRA account.

[d] This is the number of modules in Puppet Forge.

[e] To avoid false positives for the term “puppet”, I searched for “puppet devops”.

[f] This is the number of reusable roles in Ansible Galaxy.

[g] This is the number of formulas in the Salt Stack Formulas GitHub account.

[h] This is the number of templates in the awslabs GitHub account.

i Based on the OpenStack bug tracker.

j I could not find any collections of community Heat templates.

k To avoid false positives for the term “heat”, I searched for “openstack”.

l This is the number of modules in the Terraform Registry.

Obviously, this is not a perfect apples-to-apples comparison. For example, some of the tools have more than one repository, and some use other methods for bug tracking and questions; searching for jobs with common words like “chef” or “puppet” is tricky; Terraform split the provider code out into separate repos in 2017, so measuring activity on solely the core repo dramatically understates activity (by at least 10x); and so on.

That said, a few trends are obvious. First, all of the IAC tools in this comparison are open source and work with many cloud providers, except for CloudFormation, which is closed source, and only works with AWS. Second, Ansible leads the pack in terms of popularity, with Salt and Terraform not too far behind.

Another interesting trend to note is how these numbers have changed since the 1st edition of the book. Table 1-2 shows the percent change in each of the numbers from the values I gathered back in September, 2016.

Table 1-2. How the IAC communities have changed between September, 2016 and May, 2019

	Source	Cloud	Contributors	Stars	Commits (1 month)	Issues (1 month
Chef	Open	All	+18%	+31%	+139%	+48%
Puppet	Open	All	+19%	+27%	+19%	+42%
Ansible	Open	All	+195%	+97%	+49%	+66%
SaltStack	Open	All	+40%	+44%	+79%	+27%
CloudFormation	Closed	AWS	?	?	?	?
Heat	Open	All	+28%	+23%	-85%	+1,566
Terraform	Open	All	+93%	+194%	-61%	-58%

Again, the data here is not perfect, but it's good enough to spot a clear trend: Terraform and Ansible are experiencing explosive growth. The increase in the number of contributors, stars, open source libraries, StackOverflow posts, and jobs is through the roof.[9] Both of these tools have large, active communities today, and judging by these trends, it's likely that they will become even larger in the future.

Mature Versus Cutting Edge

Another key factor to consider when picking any technology is maturity. Table 1-3 shows the initial release dates and current version number (as of May, 2019) for of each of the IAC tools.

Table 1-3. A comparison of IAC maturity as of May, 2019

	Initial release	Current version
Puppet	2005	6.0.9

Chef	2009	12.19.31
CloudFormation	2011	2010-09-09
SaltStack	2011	2019.2.0
Ansible	2012	2.5.5
Heat	2012	12.0.0
Terraform	2014	0.12.0

Again, this is not an apples-to-apples comparison, since different tools have different versioning schemes, but some trends are clear. Terraform is, by far, the youngest IAC tool in this comparison. It's still pre 1.0.0, so there is no guarantee of a stable or backward compatible API, and bugs are relatively common (although most of them are minor). This is Terraform's biggest weakness: although it has gotten extremely popular in a short time, the price you pay for using this new, cutting-edge tool is that it is not as mature as some of the other IAC options.

Conclusion

Putting it all together, Table 1-4 shows how the most popular IAC tools stack up. Note that this table shows the *default* or *most common* way the various IAC tools are used, though as discussed earlier in this chapter, these IAC tools are flexible enough to be used in other configurations, too (e.g., Chef can be used without a master, Salt can be used to do immutable infrastructure).

Table 1-4. A comparison of the most common way to use the most popular IAC tools

	Source	Cloud	Type	Infrastructure	Language
Chef	Open	All	Config Mgmt	Mutable	Procedural
Puppet	Open	All	Config Mgmt	Mutable	Declarative
Ansible	Open	All	Config Mgmt	Mutable	Procedural
SaltStack	Open	All	Config Mgmt	Mutable	Declarative
CloudFormation	Closed	AWS	Provisioning	Immutable	Declarative
Heat	Open	All	Provisioning	Immutable	Declarative
Terraform	Open	All	Provisioning	Immutable	Declarative

At Gruntwork, what we wanted was an open source, cloud-agnostic provisioning tool that supported immutable infrastructure, a declarative language, a masterless and agentless architecture, and had a large community and a mature codebase. Table 1-4 shows that Terraform, while not perfect, comes the closest to meeting all of our criteria.

Does Terraform fit your criteria, too? If so, then head over to Chapter 2 to learn how to use it.

---

1 From *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations* (IT Revolution Press) by Gene Kim, Jez Humble, Patrick Debois, and John Willis.

2 On most modern operating systems, code runs in one of two "spaces": *kernel space* and *user space*. Code running in kernel space has direct, unrestricted access to all of the hardware. There are no security restrictions (i.e., you can execute any CPU instruction, access any part of the hard drive, write to any address in

memory) or safety restrictions (e.g., a crash in kernel space will typically crash the entire computer), so kernel space is generally reserved for the lowest-level, most trusted functions of the operating system (typically called the *kernel*). Code running in user space does not have any direct access to the hardware and must use APIs exposed by the operating system kernel instead. These APIs can enforce security restrictions (e.g., user permissions) and safety (e.g., a crash in a user space app typically only affects that app), so just about all application code runs in user space.

3 As a general rule, containers provide isolation that's good enough to run your own code, but if you need to run third-party code (e.g., you're building your own cloud provider) that may actively be performing malicious actions, you'll want the increased isolation guarantees of a VM.

4 As an alternative to Bash, Packer also allows you to configure your images using configuration management tools such as Ansible or Chef.

5 Check out the Gruntwork Infrastructure as Code Library for an example.

6 Docker and Packer are not part of the comparison because they can be used with any of the configuration management or provisioning tools.

7 The distinction between configuration management and provisioning has become even less clear cut in recent months, as some of the major configuration management tools have started to add better support for provisioning, such as Chef Provisioning and the Puppet AWS Module.

8 Most of this data, including the number of contributors, stars, changes, and issues, comes from the open source repositories and bug trackers (mostly GitHub) for each tool. Since CloudFormation is closed source, some of this information is not available.

9 the decline in Terraform's commits and issues is solely due to the fact that I'm only measuring the core Terraform repo, whereas in 2017, all the provider code was extracted into separate repos, so the vast amount of activity across the more than 100 provider repos is not being counted.

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.
- 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.

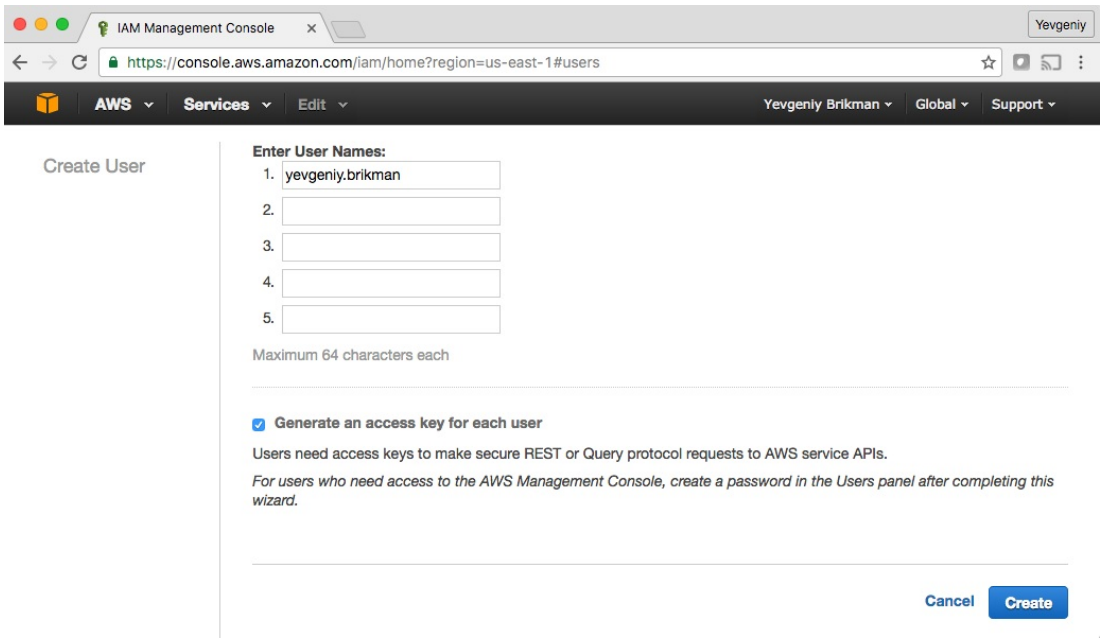

Figure 2-1. Create a new IAM user

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.

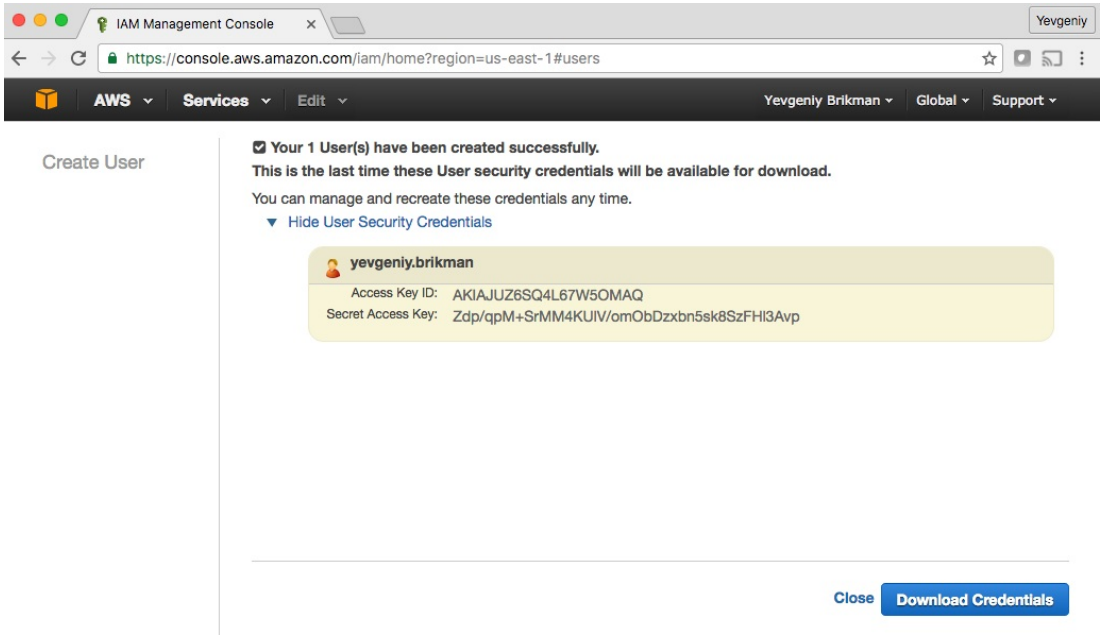

Figure 2-2. Store your AWS credentials somewhere secure. Never share them with anyone. Don't worry, the ones in the screenshot are fake.

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.

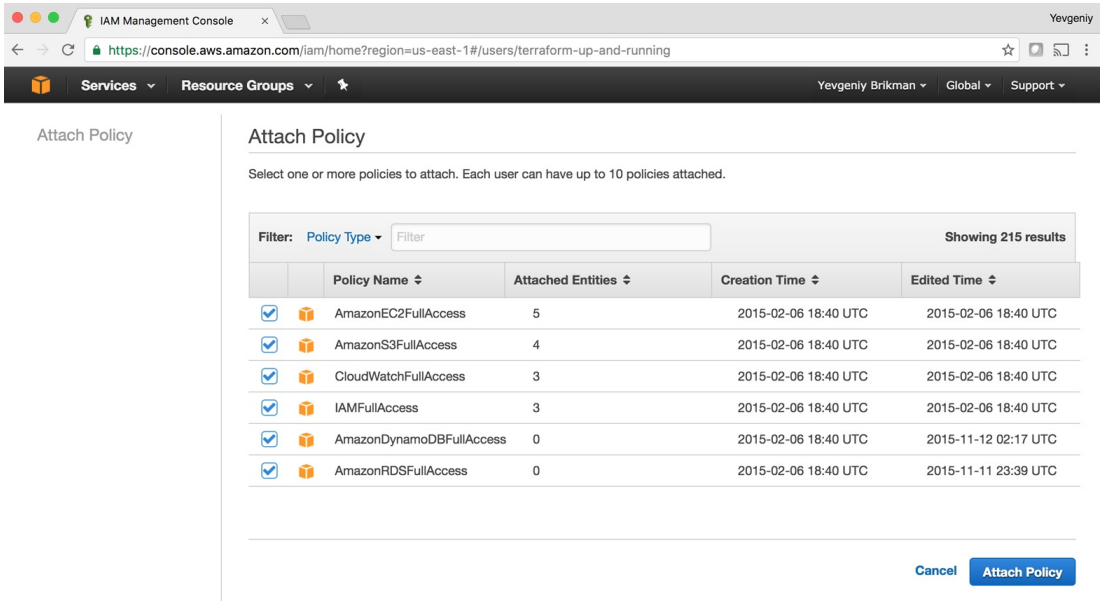

Figure 2-3. Add several Managed IAM Policies to your new IAM user

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 create a new Default VPC using the AWS Web Console. 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]

Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform
interpolations
    destroy            Destroy Terraform-managed
infrastructure
    env                Workspace management
    fmt                Rewrites config files to
canonical format
    (...)
```

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 A Comprehensive Guide to Authenticating to AWS on the Command Line.

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-2"
}
```

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-2` 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-2` (Ohio), `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-2a`, `us-east-2b`, 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-0c55b159cbfafe1f0"
  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-0c55b159cbfafe1f0"`). 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 (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 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. The preceding example uses `t2.micro`, which has one virtual CPU, 1GB of memory, and is part of the AWS free tier.

USE THE DOCS!

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

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

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (terraform-
providers/aws) 2.10.0...

The following providers do not have any version
constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that
may contain breaking
changes, it is recommended to add version = "..."
constraints to the
corresponding provider blocks in configuration, with the
constraint strings
suggested below.

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

Terraform has been successfully initialized!
```

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 a Terraform module, 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. You'll see a few other uses for the `init` command in later chapters.

Now that you have the provider code downloaded, run the `terraform plan` command (the log output below is truncated for readability):

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_instance.example will be created
  + resource "aws_instance" "example" {
      + ami                          = "ami-
0c55b159cbfafe1f0"
      + arn                          = (known after
apply)
      + associate_public_ip_address  = (known after
apply)
      + availability_zone            = (known after
apply)
      + cpu_core_count               = (known after
apply)
      + cpu_threads_per_core         = (known after
apply)
      + get_password_data            = false
      + host_id                      = (known after
apply)
      + id                           = (known after
apply)
      + instance_state               = (known after
apply)
      + instance_type                = "t2.micro"
```

```
      + ipv6_address_count            = (known after
apply)
      + ipv6_addresses                = (known after
apply)
      + key_name                      = (known after
apply)
      (...)
  }

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

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

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

```
$ terraform apply

(...)

Terraform will perform the following actions:

  # aws_instance.example will be created
  + resource "aws_instance" "example" {
      + ami                           = "ami-
0c55b159cbfafe1f0"
      + 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 [id=i-
07e2a3e006d785906]

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.

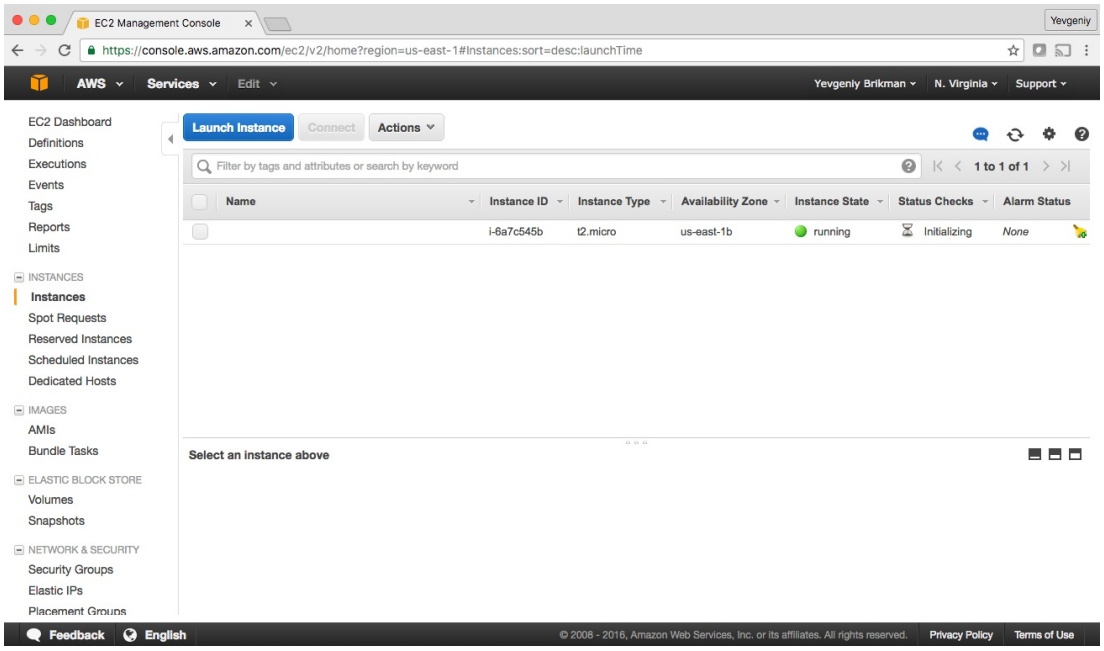

Figure 2-4. A single EC2 Instance

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-0c55b159cbfafe1f0"
  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-
0c55b159cbfafe1f0"
        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 `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 type in "yes" and hit enter.

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

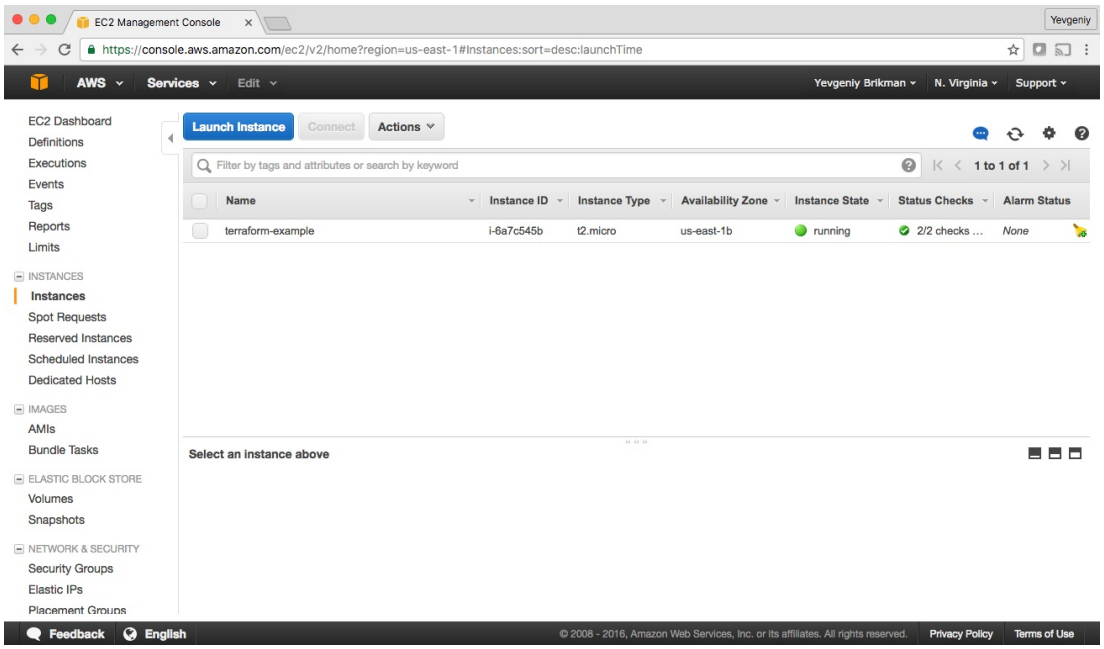

Figure 2-5. The EC2 Instance now has a name tag

Now that you have some working Terraform code, you may want to store it in version control. This allows you to share your code with other team members, track the history of all infrastructure changes, and use the commit log for debugging. For example, here is how you can create a

local Git repository and use it to store your Terraform configuration file:

```
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>.git
```

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.

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.

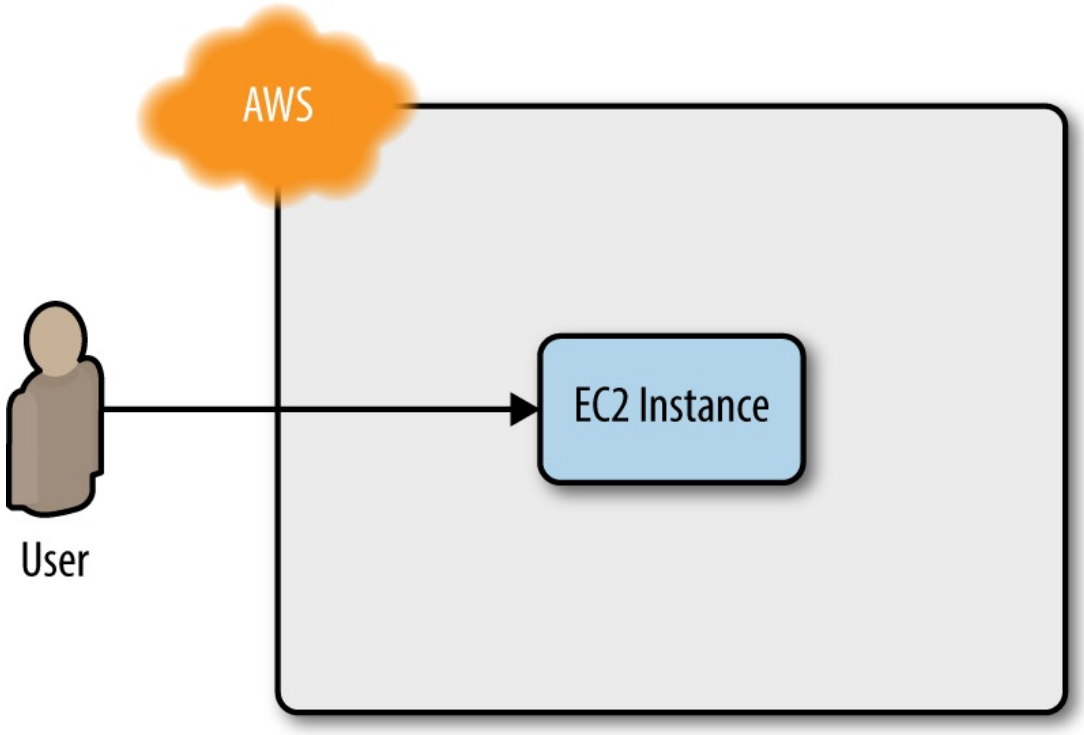

Figure 2-6. Start with a simple architecture: a single web server running in AWS that responds to HTTP requests

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":[6]

```
#!/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 18.04 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-0c55b159cbfafe1f0"
  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.[7]

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` parameter 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-0c55b159cbfafe1f0"`) and numbers (e.g., `5`). Terraform supports many other types of expressions that you'll see throughout the book.

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). 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-0c55b159cbfafe1f0"
  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, 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.instance"
                "[root] aws_security_group.instance" ->
"[root] provider.aws"
                "[root] meta.count-boundary (EachMode
```

```
fixup)" -> "[root] aws_instance.example"
                "[root] provider.aws (close)" -> "[root]
aws_instance.example"
                "[root] root" -> "[root] meta.count-
boundary (EachMode fixup)"
                "[root] root" -> "[root] provider.aws
(close)"
        }
}
```

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.

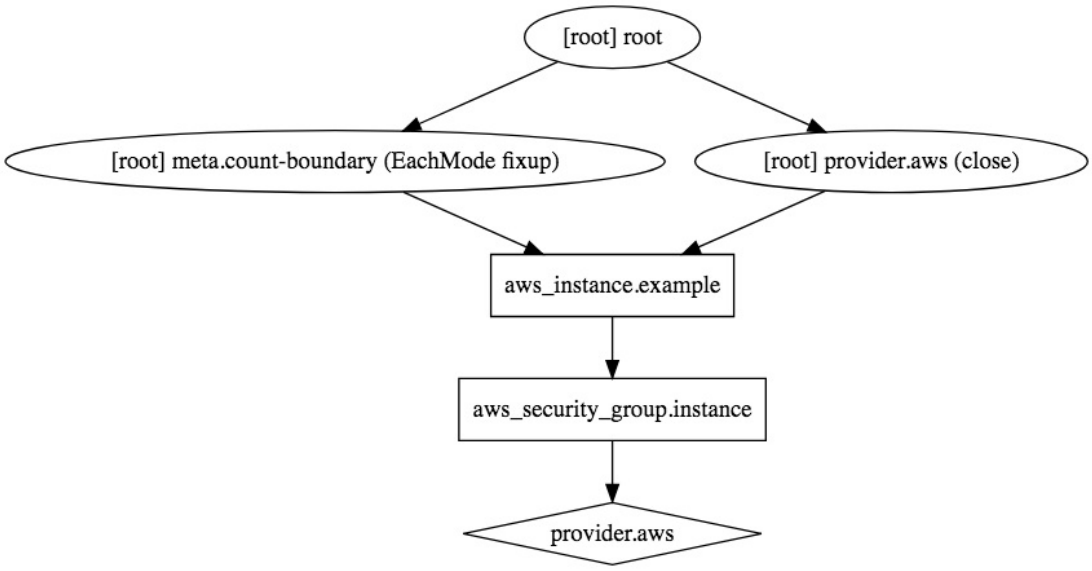

Figure 2-7. The dependency graph for the EC2 Instance and its security group

When Terraform walks your dependency tree, it will create as many resources in parallel as it can, which means it 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 update the EC2 Instance with a new one 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-
0c55b159cbfafe1f0"
      ~ availability_zone             = "us-east-2c" ->
(known after apply)
      ~ instance_state                = "running" ->
(known after apply)
        instance_type                 = "t2.micro"
        (...)
      + user_data                     = "c765373..." #
forces replacement
      ~ volume_tags                   = {} -> (known
after apply)
      ~ 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"
      + owner_id               = (known after apply)
      + revoke_rules_on_delete = false
      + vpc_id                 = (known after apply)
    }

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 most 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 discussed in "Server Templating Tools". 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 Chapter 5.

Since the plan looks good, enter "yes" and you'll see your new EC2 Instance deploying, as shown in Figure 2-8.

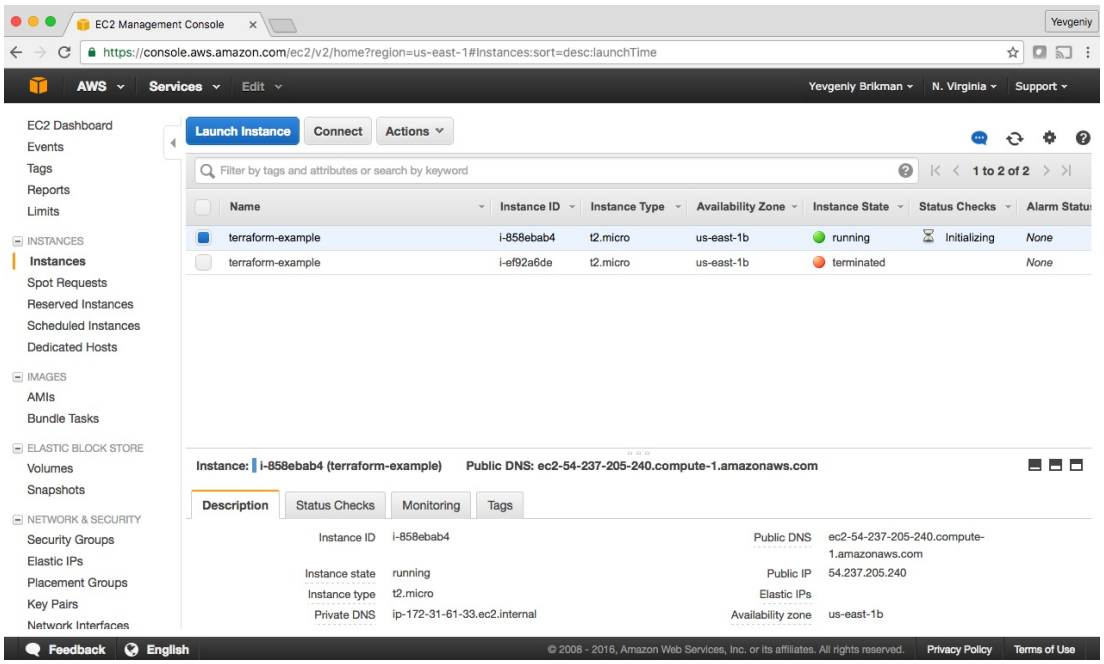

Figure 2-8. The new EC2 Instance with the web server code replaces the old Instance

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.[8] 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`.

Here is an example of an input variable that checks the value you pass in is a number:

```
variable "number_example" {
  description = "An example of a number variable in
Terraform"
  type        = number
  default     = 42
}
```

And here's an example of a variable that checks the value is a list:

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

You can combine type constraints, too. For example, here's a list input variable that requires all the items in the list to be numbers:

```
variable "list_numeric_example" {
  description = "An example of a numeric list in
Terraform"
  type        = list(number)
  default     = [1, 2, 3]
}
```

And here's a map that requires all the values to be strings:

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

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

You can also create more complicated *structural types* using the `object` and `tuple` type constraints:

```
variable "object_example" {
  description = "An example of a structural type in
Terraform"
  type        = object({
```

```
    name    = string
    age     = number
    tags    = list(string)
    enabled = bool
  })

  default = {
    name    = "value1"
    age     = 42
    tags    = ["a", "b", "c"]
    enabled = true
  }
}
```

The example above creates an input variable that will require the value to be an object with the keys `name` (which must be a string), `age` (which must be a number), `tags` (which must be a list of strings), and `enabled` (which must be a boolean). If you were to try to set this variable to a value that doesn't match this type, Terraform immediately gives you a type error. For example, if you try to set `enabled` to a string instead of a boolean:

```
variable "object_example_with_error" {
  description = "An example of a structural type in
Terraform with an error"
  type        = object({
    name    = string
    age     = number
    tags    = list(string)
    enabled = bool
  })

  default = {
    name    = "value1"
    age     = 42
    tags    = ["a", "b", "c"]
    enabled = "invalid"
  }
```

```
}
```

You get the following error:

```
$ terraform apply

Error: Invalid default value for variable

  on variables.tf line 78, in variable
"object_example_with_error":
  78:   default = {
  79:     name    = "value1"
  80:     age     = 42
  81:     tags    = ["a", "b", "c"]
  82:     enabled = "invalid"
  83:   }

This default value is not compatible with the variable's
type constraint: a
bool is required.
```

For the web server example, all you need is 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 plan -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 plan
```

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 *ouput 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 address 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... [id=sg-
078ccb4f9533d2c1a]
aws_instance.example: Refreshing state... [id=i-
028cad2d4e6bddec6]

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 simple 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.

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.[9]

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.

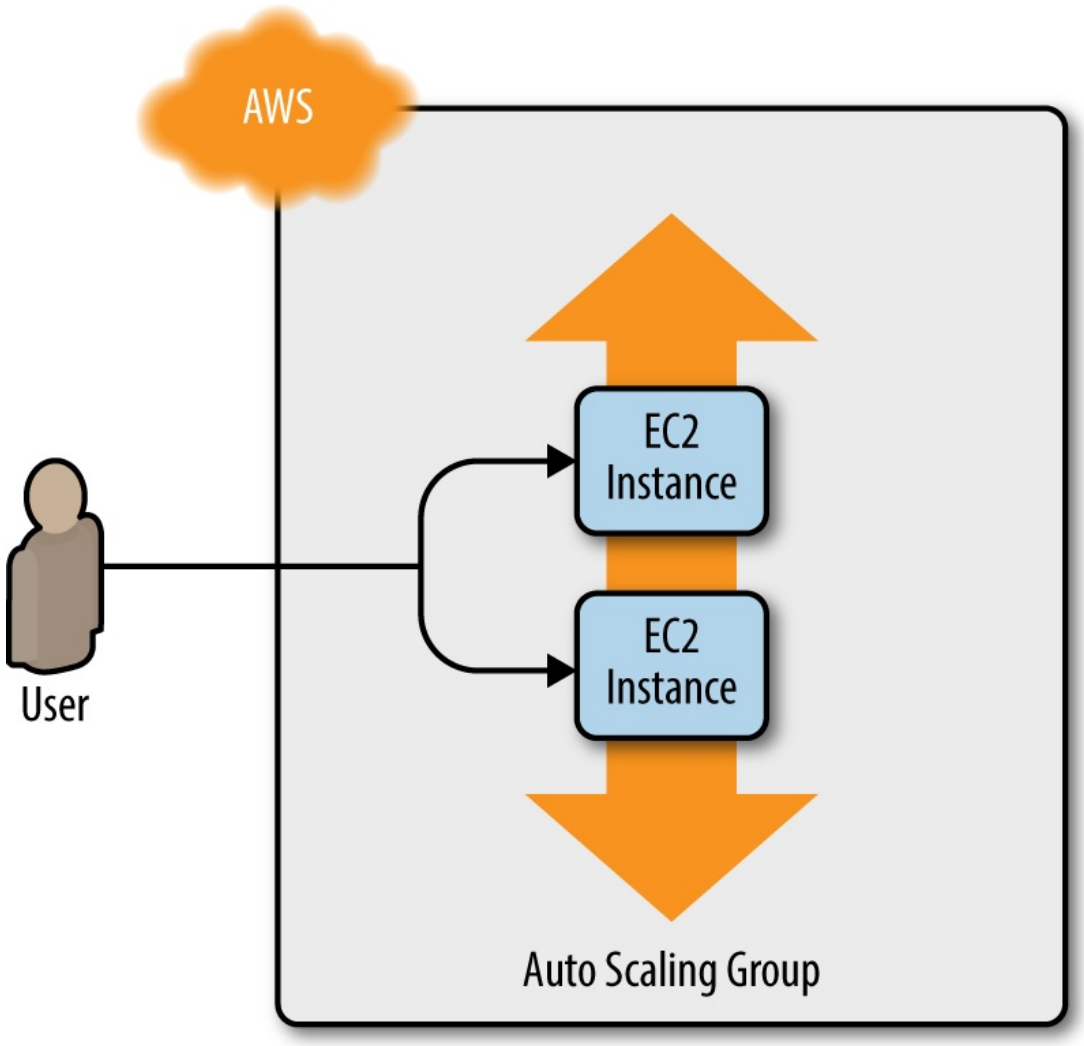

Figure 2-9. Instead of a single web server, run a cluster of web servers using an Auto Scaling Group

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 (only two parameters are different: `ami` is now `image_id` and `vpc_security_group_ids` is now `security_groups`), so you can cleanly replace the latter with the former:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
```

```
  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
}
```

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

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

  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: `subnet_ids`. This parameter tells the ASG into which VPC subnets the EC2 Instances should be deployed (see Network Security for background info on subnets). Each subnet lives in an isolated AWS Availability Zone

(that is, isolated data center), so by deploying your Instances across multiple subnets, you ensure that your service can keep running even if some of the data centers have an outage. You could hard-code the list of subnets, but that won't be maintainable or portable, 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_vpc` data source to look up the data for your Default VPC (see A Note on Default VPCs for background information):

```
data "aws_vpc" "default" {
```

```
  default = true
}
```

Note that with data sources, the arguments you pass in are typically search filters that tell the data source what information you're looking for. With the `aws_vpc` data source, the only filter you need is `default = true`, which tells Terraform to look up the default VPC in your AWS account.

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 ID of the VPC from the `aws_vpc` data source, you would use the following:

```
data.aws_vpc.default.id
```

You can combine this with another data source, `aws_subnet_ids`, to look up the subnets within that VPC:

```
data "aws_subnet_ids" "default" {
  vpc_id = data.aws_vpc.default.id
}
```

Finally, you can pull the subnet IDs out of the `aws_subnet_ids` data source and tell your ASG to use those subnets via the (somewhat oddly named) `vpc_zone_identifier` argument:

```
resource "aws_autoscaling_group" "example" {
```

```
  launch_configuration =
aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids

  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, as shown in Figure 2-10.

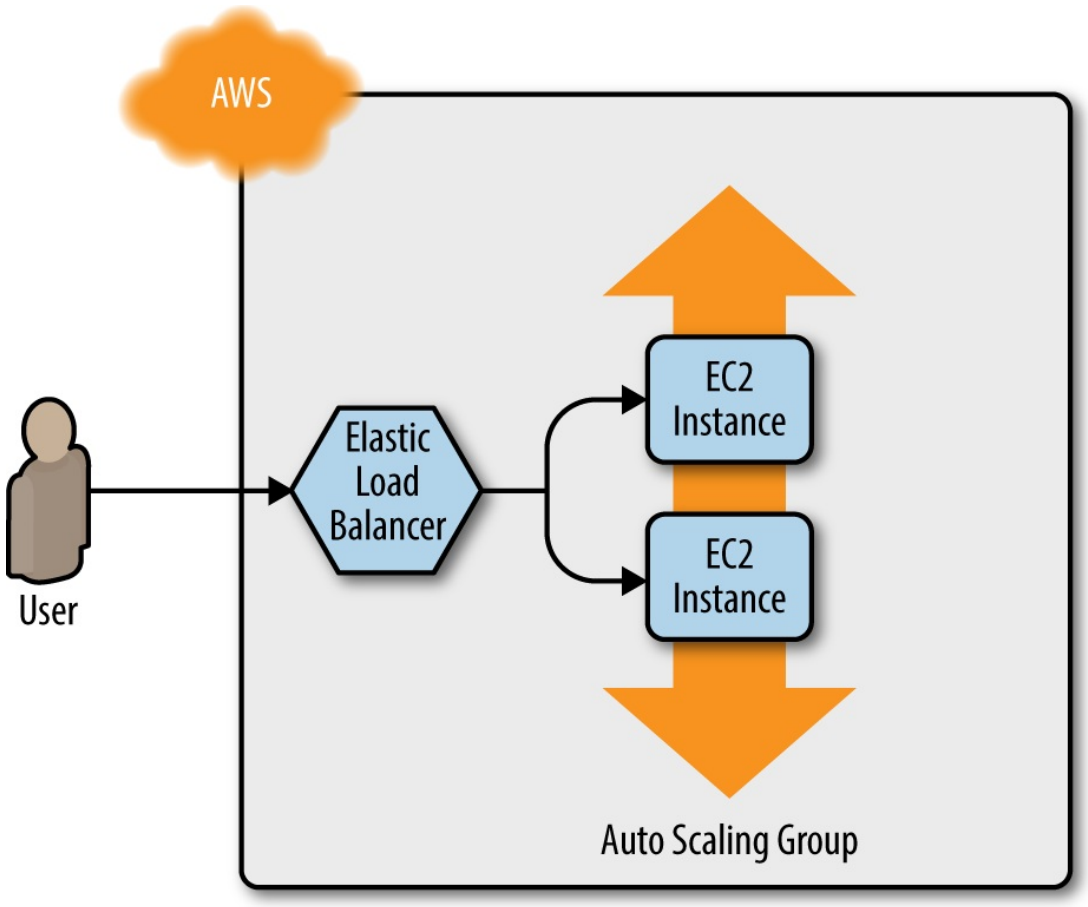

Figure 2-10. Use an Elastic Load Balancer to distribute traffic across the Auto Scaling Group

AWS offers three different types of load balancers:

Application Load Balancer (ALB)

: Best suited for load balancing of HTTP and HTTPS traffic. Operates at the application layer ("Layer 7") of the OSI model.

Network Load Balancer (NLB)

: Best suited for load balancing of TCP and TLS traffic, especially when extreme performance is required. Operates at the transport layer ("Layer 4") of the OSI model.

Classic Load Balancer (CLB)

: This is the "legacy" load balancer that predates both the ALB and

NLB. It can handle HTTP, HTTPS, TCP, and TLS traffic, but with far fewer features than either the ALB or NLB. Operates at both the application layer ("Layer 7") and transport layer ("Layer 4") of the OSI model.

Most applications these days should use either the ALB or the NLB. Since the simple web server example you're working on is an HTTP app without any extreme performance requirements, the ALB is going to be the best fit.

As shown in Figure 2-11, the ALB consists of several parts:

1. Listener: listens on a specific port (e.g., 80) and protocol (e.g., 443)
2. Listener rule: takes requests that come into a listener and sends those that match specific paths (e.g., `/foo` and `/bar`) or host names (e.g., `foo.example.com` and `bar.example.com`) to specific target groups.
3. Target groups: one or more servers that receive requests from the load balancer. The target group also performs health checks on these servers and only sends requests to healthy nodes.

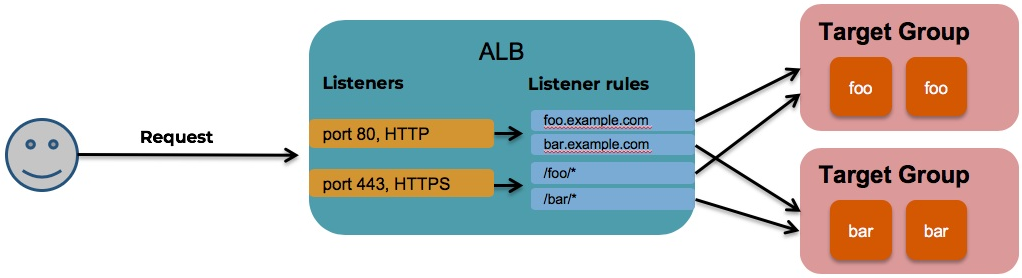

Figure 2-11. Application Load Balancer (ALB) overview

The first step is to create the ALB itself using the `aws_lb` resource:

```
resource "aws_lb" "example" {
  name               = "terraform-asg-example"
  load_balancer_type = "application"
  subnets            = data.aws_subnet_ids.default.ids
}
```

Note that the `subnets` parameter configures the load balancer to use all the subnets in your default VPC by using the `aws_subnet_ids` data source.[10] This is because AWS load balancers don't consist of a single server, but multiple servers that can run in separate subnets (and therefore, 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.

The next step is to define a listener for this ALB using the `aws_lb_listener` resource:

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = 80
  protocol          = "HTTP"

  # By default, return a simple 404 page
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

This listener configures the ALB to listen on the default HTTP port, port 80, use HTTP as the protocol, and send a simple 404 page as the default response for requests that don't match any listener rules.

Note that, by default, all AWS resources, including ALBs, don't allow any incoming or outgoing traffic, so you need to create a new security group specifically for the ALB to allow incoming requests on port 80, so you can access the load balancer over HTTP, as well as outgoing requests on all ports, so the load balancer can perform health checks, as you'll configure shortly:

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

  # Allow inbound HTTP requests
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

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

You'll need to tell the `aws_lb` resource to use this security group via the `security_groups` argument:

```
resource "aws_lb" "example" {
  name               = "terraform-asg-example"
  load_balancer_type = "application"
```

```
  subnets             = data.aws_subnet_ids.default.ids
  security_groups     = [aws_security_group.alb.id]
}
```

Next, you need to create a target group for your ASG using the `aws_lb_target_group` resource:

Note that this target group will health check your servers by periodically sending an HTTP request to each server and only considering the server "healthy" if the server returns a response that matches the configured `matcher` (i.e., returns a 200 OK). If a server fails to respond, perhaps because that server has gone down or is overloaded, it will be marked as "unhealthy," and the target group will automatically stop sending traffic to it to minimize disruption for your users.

How does the target group know which EC2 Instances to send requests to? You could attach a static list of EC2 Instances to the target group using the `aws_lb_target_group_attachment` resource, but with an ASG, Instances may launch or terminate at any time, so a static list won't work. Instead, you can take advantage of the first-class integration between the ASG and the ALB. Go back to the `aws_autoscaling_group` resource and set its `target_group_arns` argument to point at your new target group:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration =
aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids

  target_group_arns = [aws_lb_target_group.asg.arn]
  health_check_type = "ELB"
```

```
  min_size = 2
  max_size = 10

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

This will tell the ASG to register each Instance in the target group when that Instance is booting. Also, notice that the `health_check_type` is now `"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 target group's health check to determine if an Instance is healthy or not and to automatically replace Instances if the target group 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.

Finally, it's time to tie all these pieces together by creating listener rules using the `aws_lb_listener_rule` resource:

```
resource "aws_lb_listener_rule" "asg" {
  listener_arn = aws_lb_listener.http.arn
  priority     = 100

  condition {
    field  = "path-pattern"
    values = ["*"]
  }

  action {
```

```
    type              = "forward"
    target_group_arn = aws_lb_target_group.asg.arn
  }
}
```

The code above adds a listener rule that send requests that match any path to the target group that contains your ASG.

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 ALB:

```
output "alb_dns_name" {
  value       = aws_lb.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 `alb_dns_name` output:

```
Outputs:
alb_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 ALB. 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 2-12.

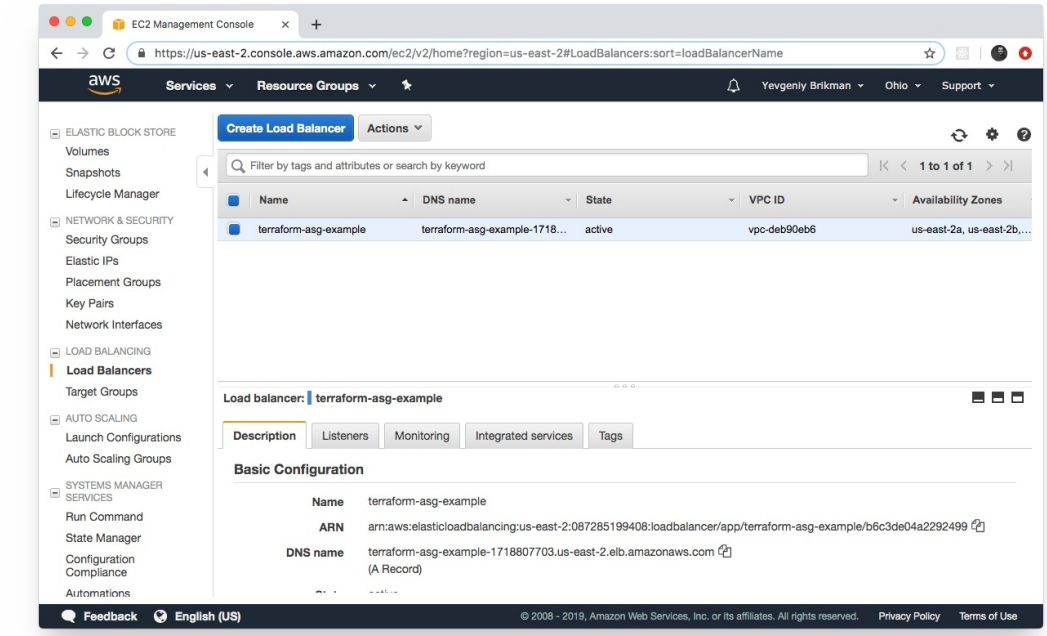

Figure 2-12. The Auto Scaling Group

If you switch over to the Instances tab, you'll see the two EC Instances launching, as shown in Figure 2-13.

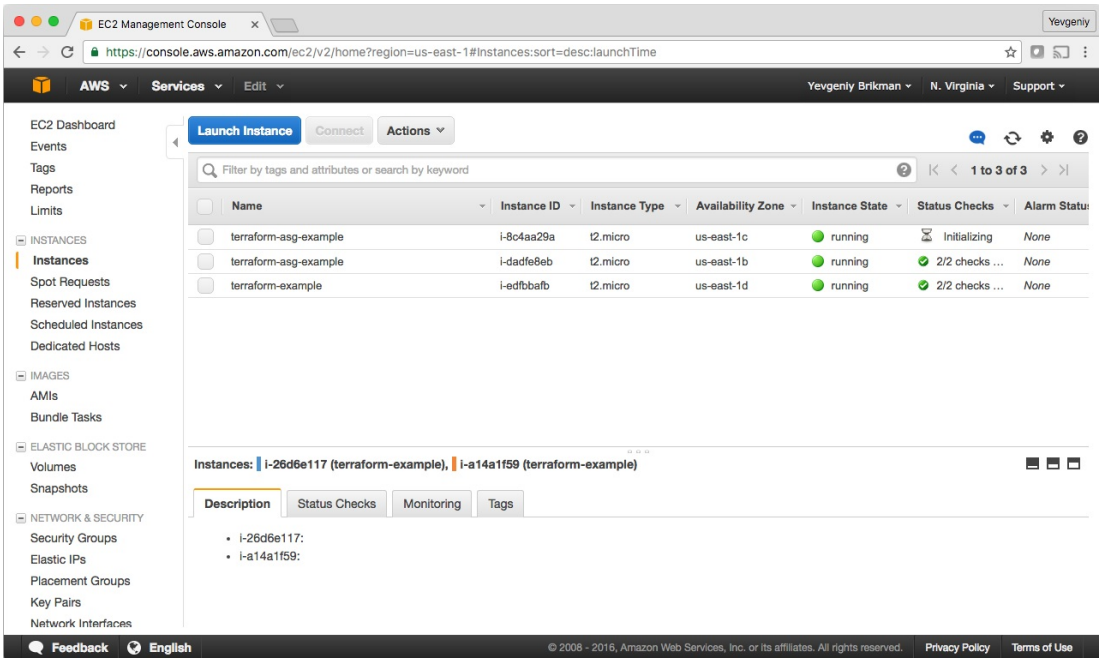

Figure 2-13. The EC2 Instances in the ASG are launching

If you click on the Load Balancers tab, you'll see your ALB, as shown in Figure 2-14.

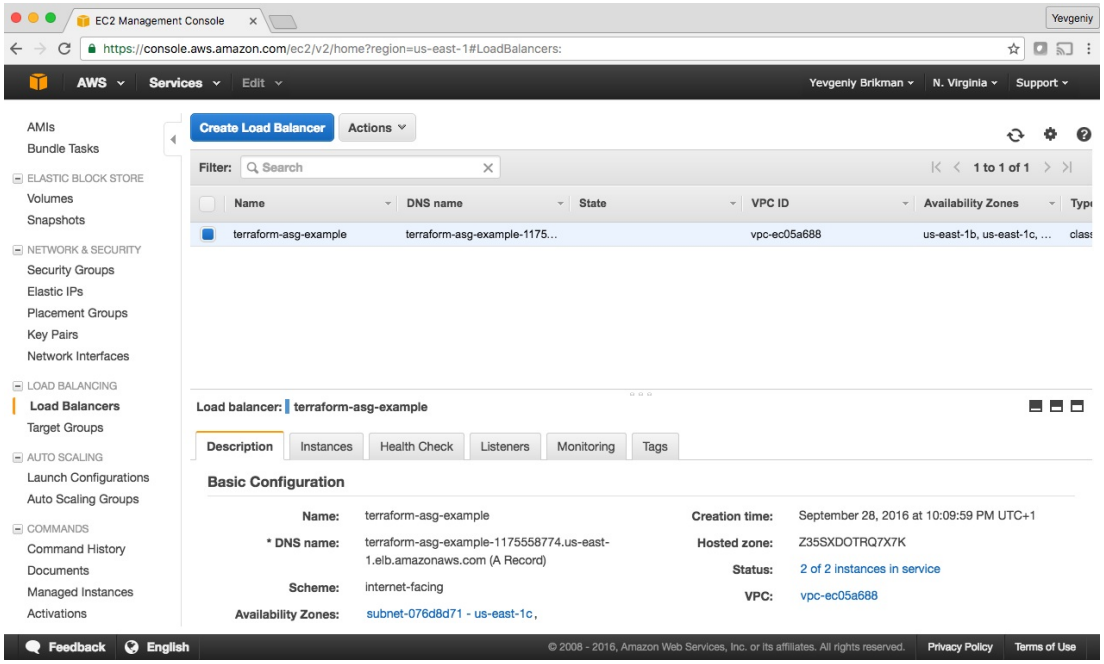

Figure 2-14. The Application Load Balancer

Finally, if you click on the Target Groups tab, you can find your target group, as shown in Figure 2-15.

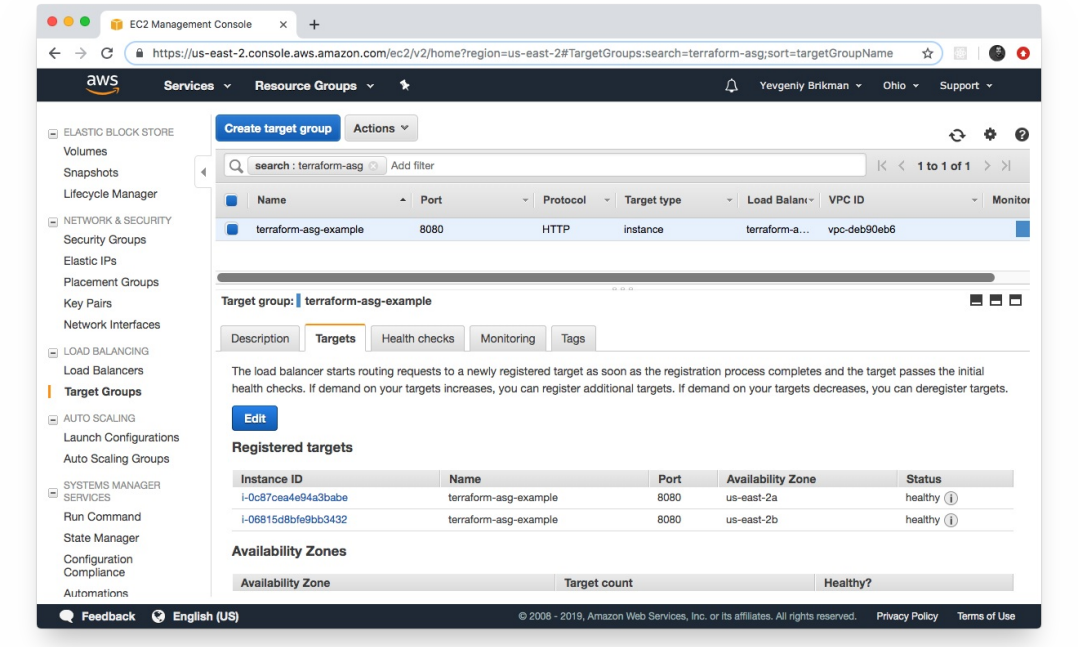

Figure 2-15. The Application Load Balancer

If you click on your target group and find the Targets tab in the bottom half of the screen, you can see your Instances registering with the target group and going through health checks. Wait for the "Status" indicator to say "healthy" for both of them. This typically takes 1 to 2 minutes. Once you see it, test the `alb_dns_name` output you copied earlier:

```
$ curl http://<alb_dns_name>
Hello, World
```

Success! The ALB 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 ALB URL and you should get a 200 OK for each request, even while terminating an Instance, as the ALB 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

(...)

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:
```

It goes without saying that you should rarely, if ever, run `destroy` in prod! There's no "undo" for the `destroy` command, so Terraform gives you one final chance to review what you're doing, showing you the list of all the resources you're about to delete, and prompting you to confirm the deletion. If everything looks good, type in "yes" and hit Enter, and 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 whenever you want. 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, references, 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*.

3 You can learn more about IAM Policies here: *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*.

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

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

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

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

10 To keep these examples simple, we're running the EC2 Instances and ALB in the same subnets. In production usage, you'd most likely run them in different subnets, with the EC2 Instances in private subnets (so they aren't directly accessible from the public Internet) and the ALBs in public subnets (so users can access them directly).

Chapter 3. How to Manage Terraform State

In Chapter 2, as you were using Terraform to create and update resources, you may have noticed that every time you ran `terraform plan` or `terraform apply`, Terraform was able to find the resources it created previously and update them accordingly. But how did Terraform know which resources it was supposed to manage? You could have all sorts of infrastructure in your AWS account, deployed through a variety of mechanisms (some manually, some via Terraform, some via the CLI), so how does Terraform know which infrastructure it's responsible for?

In this chapter, you're going to see how Terraform tracks the state of your infrastructure and the impact that has on file layout, isolation, and locking in a Terraform project. Here are the key topics I'll go over:

- What is Terraform state?
- Shared storage for state files
- Locking state files
- Isolating state files
    - Isolation via workspaces
    - Isolation via file layout
- Read-only state

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*.

What Is Terraform State?

Every time you run Terraform, it records information about what infrastructure it created in a *Terraform state file*. By default, when you run Terraform in the folder */foo/bar*, Terraform creates the file */foo/bar/terraform.tfstate*. This file contains a custom JSON format that records a mapping from the Terraform resources in your configuration files to the representation of those resources in the real world. For example, let's say your Terraform configuration contained the following:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

After running `terraform apply`, here is a small snippet of the contents of the *terraform.tfstate* file (truncated for readability):

```
{
  "version": 4,
  "terraform_version": "0.12.0",
  "serial": 1,
  "lineage": "1f2087f9-4b3c-1b66-65db-8b78faafc6fb",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
```

```
      "name": "example",
      "provider": "provider.aws",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0c55b159cbfafe1f0",
            "availability_zone": "us-east-2c",
            "id": "i-00d689a0acc43af0f",
            "instance_state": "running",
            "instance_type": "t2.micro",
            "(...)": "(truncated)"
          }
        }
      ]
    }
  ]
}
```

Using this simple JSON format, Terraform knows that a resource with type `aws_instance` and name `example` corresponds to an EC2 Instance in your AWS account with ID `i-00d689a0acc43af0f`. Every time you run Terraform, it can fetch the latest status of this EC2 Instance from AWS and compare that to what's in your Terraform configurations to determine what changes need to be applied. In other words, the output of the `plan` command is a diff between the code on your computer and the infrastructure deployed in the real world, as discovered via IDs in the state file.

THE STATE FILE IS A PRIVATE API

The state file format is a private API that changes with every release and is meant only for internal use within Terraform. You should never edit the

Terraform state files by hand or write code that reads them directly.

If for some reason you need to manipulate the state file—which should be a relatively rare occurrence—use the `terraform import` command (you'll see an example of this in Chapter 5) or the `terraform state` command (this is only for advanced use cases).

If you're using Terraform for a personal project, storing state in a single *terraform.tfstate* file that lives locally on your computer works just fine. But if you want to use Terraform as a team on a real product, you run into several problems:

Shared storage for state files

: To be able to use Terraform to update your infrastructure, each of your team members needs access to the same Terraform state files. That means you need to store those files in a shared location.

Locking state files

: As soon as data is shared, you run into a new problem: locking. Without locking, if two team members are running Terraform at the same time, you may run into race conditions as multiple Terraform processes make concurrent updates to the state files, leading to conflicts, data loss, and state file corruption.

Isolating state files

: When making changes to your infrastructure, it's a best practice to isolate different environments. For example, when making a change in a testing or staging environment, you want to be sure that there is no way you can accidentally break production. But how can you isolate your changes if all of your infrastructure is defined in the same Terraform state file?

In the following sections, I'll dive into each of these problems and show

you how to solve them.

Shared Storage for State Files

The most common technique for allowing multiple team members to access a common set of files is to put them in version control (e.g., Git). While you should definitely store your Terraform code in version control, storing Terraform state in version control is a *bad idea* for two reasons:

Manual error

: It's too easy to forget to pull down the latest changes from version control before running Terraform or to push your latest changes to version control after running Terraform. It's just a matter of time before someone on your team runs Terraform with out-of-date state files and as a result, accidentally rolls back or duplicates previous deployments.

Locking

: Most version control systems do not provide any form of locking that would prevent two team members from running `terraform apply` on the same state file at the same time.

Secrets

: All data in Terraform state files is stored in plain text. This is a problem because certain Terraform resources need to store sensitive data. For example, if you use the `aws_db_instance` resource to create a database, Terraform will store the username and password for the database in a state file in plain text. Storing plain-text secrets *anywhere* is a bad idea, including version control. As of May, 2019, this is an open issue in the Terraform community, although there are some reasonable workarounds, as I will discuss shortly.

Instead of using version control, the best way to manage shared storage for

state files is to use Terraform's built-in support for remote backends. Each *backend* determines how Terraform loads and stores state. The default backend, which you've been using this whole time, is the *local backend*, which stores the state file on your local disk. *Remote backends* allow you to store the state file in a remote, shared store. A number of remote backends are supported, including Amazon S3, Azure Storage, Google Cloud Storage, and HashiCorp's Terraform Pro and Terraform Enterprise.

Remote backends solve all three of the issues listed above:

Manual error

: Once you configure a remote backend, Terraform will automatically load the state file from that backend every time you run `plan` or `apply` and it'll automatically store the state file in that backend after each `apply`, so there's no chance of manual error.

Locking

: Most of the remote backends natively support locking. To run `terraform apply`, Terraform will automatically acuire a lock; if someone else is already running `apply`, they will already have the lock, and you will have to wait. You can run `apply` with the `-lock-timeout=<TIME>` parameter to tell Terraform to wait up to `TIME` for a lock to be released (e.g., `-lock-timeout=10m` will wait for 10 minutes).

Secrets

: Most of the remote backends natively support encryption in transit and encryption on disk of the state file. Moreover, those backends usually expose ways to configure access permissions (e.g., using IAM policies with an S3 bucket), so you can control who has access to you state files and the secrets the may contain. It would still be better if Terraform natively supported encrypting secrets within the state file, but these remote backends reduce most of the security concern, as at

least the state file isn't stored in plaintext on disk anywhere.

If you're using Terraform with AWS, Amazon S3 (Simple Storage Service), which is Amazon's managed file store, is typically your best bet as a remote backend for the following reasons:

- It's a managed service, so you don't have to deploy and manage extra infrastructure to use it.
- It's designed for 99.999999999% durability and 99.99% availability, which means you don't have to worry too much about data loss or outages.[1]
- It supports encryption, which reduces worries about storing sensitive data in state files. Anyone on your team who has access to that S3 bucket will be able to see the state files in an unencrypted form, so this is still a partial solution, but at least the data will be encrypted at rest (S3 supports server-side encryption using AES-256) and in transit (Terraform uses SSL to read and write data in S3).
- It supports locking via DynamoDB. More on this below.
- It supports *versioning*, so every revision of your state file is stored, and you can always roll back to an older version if something goes wrong.
- It's inexpensive, with most Terraform usage easily fitting into the free tier.[2]

To enable remote state storage with S3, the first step is to create an S3 bucket. Create a *main.tf* file in a new folder (it should be a different folder from where you store the configurations from Chapter 2) and at the top of the file, specify AWS as the provider:

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

Next, create an S3 bucket by using the `aws_s3_bucket` resource:

```
resource "aws_s3_bucket" "terraform_state" {
  bucket = "terraform-up-and-running-state"

  # Prevent accidental deletion of this S3 bucket
  lifecycle {
    prevent_destroy = true
  }

  # Enable versioning so we can see the full revision
history of our
  # state files
  versioning {
    enabled = true
  }

  # Enable server-side encryption by default
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}
```

This code sets four arguments:

`bucket`

> This is the name of the S3 bucket. Note that S3 bucket names must be *globally* unique amongst all AWS customers. Therefore, you will have to change the `bucket` parameter from `"terraform-up-and-running-state"` (which I already created) to your own name.[3] Make sure to remember this name and take note of what AWS region you're using, as you'll need both pieces of information again a little later on.

`prevent_destroy`

> `prevent_destroy` is the first *lifecycle* setting you've seen. Every Terraform resource supports several lifecycle settings that configure how that resource is created, updated, and/or deleted. When you set `prevent_destroy` to `true` on a resource, any attempt to delete that resource (e.g., by running `terraform destroy`) will cause Terraform to exit with an error. This is a good way to prevent accidental deletion of an important resource, such as this S3 bucket, which will store all of your Terraform state. Of course, if you really mean to delete it, you can just comment that setting out.

`versioning`

> This block enables versioning on the S3 bucket, so that every update to a file in the bucket actually creates a new version of that file. This allows you to see older versions of the file and revert to those older versions at any time.

`server_side_encryption_configuration`

> This block turns server-side encryption on by default for all data written to this S3 bucket. This ensures that your state files, and any secrets they may contain, are always encrypted on disk when stored in S3.

Next, you need to create a DynamoDB table to use for locking. DynamoDB is Amazon's managed, distributed key-value store. It supports strongly-consistent reads and conditional writes, which are all the ingredients you need for a distributed lock system. Moreover, it's completely managed, so you don't have any infrastructure to run yourself, and it's inexpensive, with most Terraform usage easily fitting into the free tier.[4]

To use DynamoDB for locking with Terraform, you must create a DynamoDB table that has a primary key called `LockID` (with this *exact*

spelling and capitalization!). You can create such a table using the `aws_dynamodb_table` resource:

```
resource "aws_dynamodb_table" "terraform_locks" {
  name         = "terraform-up-and-running-locks"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "LockID"

  attribute {
    name = "LockID"
    type = "S"
  }
}
```

Run `terraform init` to download the providers for this module and then run `terraform apply` to deploy your code. Once everything is deployed, you will have an S3 bucket and DynamoDB table, but your Terraform state will still be stored locally. To configure Terraform to store the state in your S3 bucket (with encryption and locking), you need to add a `backend` configuration to your Terraform code. This is configuration for Terraform itself, so it lives within a `terraform` block, and has the following syntax:

```
terraform {
  backend "<BACKEND_NAME>" {
    [CONFIG...]
  }
}
```

Where `BACKEND_NAME` is the name of the backend you want to use (e.g., `"s3"`) and `CONFIG` consists consists of one or more *arguments* that are specific to that backend (e.g., the name of the S3 bucket to use). Here's what the `backend` configuration looks like for an S3 bucket:

```
terraform {
  backend "s3" {
    # Replace this with your bucket name!
    bucket         = "terraform-up-and-running-state"
    key            = "global/s3/terraform.tfstate"
    region         = "us-east-2"

    # Replace this with your DynamoDB table name!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Let's go through these settings one at a time:

`bucket`

: The name of the S3 bucket to use. Make sure to replace this with the name of the S3 bucket you created earlier!

`key`

: The file path within the S3 bucket where the Terraform state file should be written. You'll see a little later on why the example code above sets this to `global/s3/terraform.tfstate`.

`region`

: The AWS region where the S3 bucket was created.

`dynamodb_table`

: The DynamoDB table to use for locking. Make sure to replace this with the name of the DynamoDB table you created earlier!

`encrypt`

: Setting this to true ensures your Terraform state will be encrypted on disk when stored in S3. We already enabled default encryption in the S3 bucket itself, so this here as a second layer to ensure that the data is always encrypted.

To tell Terraform to store your state file in this S3 bucket, you're going to use the `terraform init` command again. This little command can not only download provider code, but also configure your Terraform backend (and you'll see yet another use later on too!). Moreover, the `init` command is idempotent, so it's safe to run it over and over again:

```
$ terraform init

Initializing the backend...
Acquiring state lock. This may take a few moments...
Do you want to copy existing state to the new backend?
  Pre-existing state was found while migrating the
previous "local" backend to the
  newly configured "s3" backend. No existing state was
found in the newly
  configured "s3" backend. Do you want to copy this
state to the new "s3"
  backend? Enter "yes" to copy and "no" to start with an
empty state.

  Enter a value:
```

Terraform will automatically detect that you already have a state file locally and prompt you to copy it to the new S3 backend. If you type in "yes," you should see:

```
Successfully configured the backend "s3"! Terraform will
automatically
use this backend unless the backend configuration
changes.
```

After running this command, your Terraform state will be stored in the S3 bucket. You can check this by heading over to the S3 console in your browser and clicking your bucket. You should see something similar to Figure 3-1.

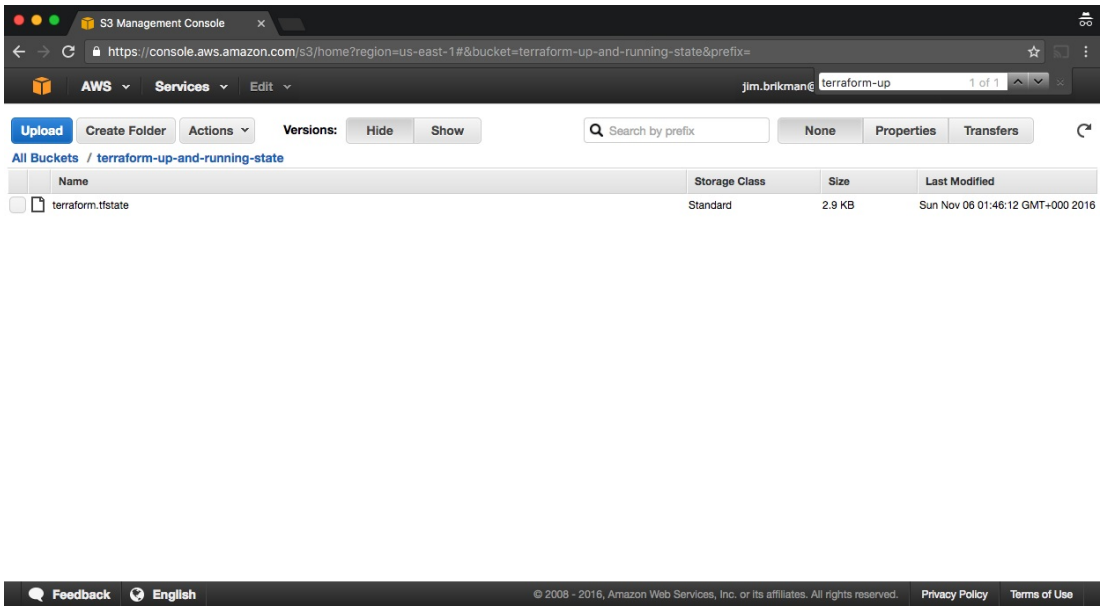

Figure 3-1. Terraform state file stored in S3

With this backend enabled, Terraform will automatically pull the latest state from this S3 bucket before running a command, and automatically push the latest state to the S3 bucket after running a command. To see this in action, add the following output variables:

```
output "s3_bucket_arn" {
  value       = aws_s3_bucket.terraform_state.arn
  description = "The ARN of the S3 bucket"
}

output "dynamodb_table_name" {
  value       = aws_dynamodb_table.terraform_locks.name
  description = "The name of the DynamoDB table"
}
```

These variable will print out the Amazon Resource Name (ARN) of your S3 bucket and the name of your DynamoDB table. Run `terraform apply` to see it:

```
$ terraform apply
```

```
Acquiring state lock. This may take a few moments...

aws_dynamodb_table.terraform_locks: Refreshing state...
[id=terraform-up-and-running-locks]
aws_s3_bucket.terraform_state: Refreshing state...
[id=terraform-up-and-running-state]

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

Releasing state lock. This may take a few moments...

Outputs:

dynamodb_table_name = terraform-up-and-running-locks
s3_bucket_arn = arn:aws:s3:::terraform-up-and-running-
state
```

(Note how Terraform is now acquiring a lock before running `apply` and releasing the lock after!)

Now, head over to the S3 console again, refresh the page, and click the gray "Show" button next to "Versions." You should now see several versions of your *terraform.tfstate* file in the S3 bucket, as shown in Figure 3-2.

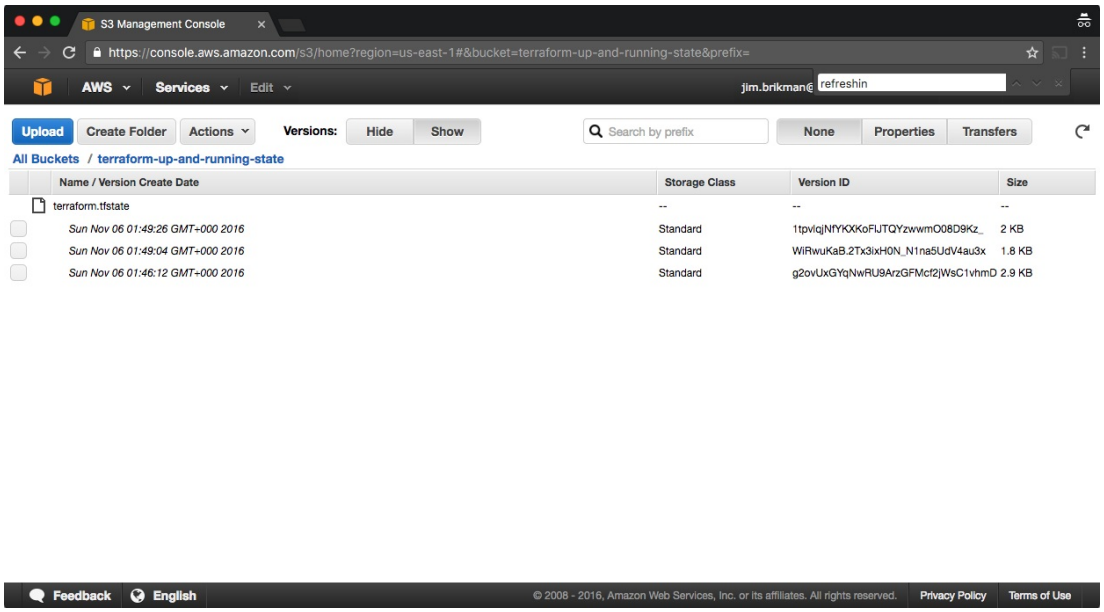

Figure 3-2. Multiple versions of the Terraform state file in S3

This means that Terraform is automatically pushing and pulling state data to and from S3 and S3 is storing every revision of the state file, which can be useful for debugging and rolling back to older versions if something goes wrong.

Some limitations with Terraform's backends

Terraform's backends have a few limitations / gotchas to be aware of. The first limitation is the chicken-and-egg situation of using Terraform to create the S3 bucket where you want to store your Terraform state. To make this work, you had to use a two-step process:

1. Write Terraform code to create the S3 bucket and DynamoDB table and deploy that code with a local backend.
2. Go back to the Terraform code, add a remote `backend` configuration to it to use the newly created S3 bucket and DynamoDB table, and run `terraform init` to copy your local state to S3.

If you ever wanted to delete the S3 bucket and DynamoDB table, you'd have to do this two-step process in reverse:

1. Go to the Terraform code, remove the `backend` configuration, and re-run `terraform init` to copy the Terraform state back to your local disk.
2. Run `terraform destroy` to delete the S3 bucket and DynamoDB table.

This two-step process is a bit awkward, but the good news is that you can share a single S3 bucket and DynamoDB table across many Terraform modules, so you'll probably only have to do it once (or once per AWS account if you have multiple accounts) Once the S3 bucket exists, in the rest of your Terraform code, you can specify the `backend` configuration right from the start without any extra steps.

The second limitation is more painful: the `backend` block in Terraform does not allow you to use any variables or references. The following code will NOT work:

```
# This will NOT work. Variables aren't allowed in a
backend configuration.
terraform {
  backend "s3" {
    bucket         = "${var.bucket}"
    region         = "${var.region}"
    dynamodb_table = "${var.dynamodb_table}"
    key            = "example/terraform.tfstate"
    encrypt        = true
  }
}
```

That means you have to manually copy and paste the S3 bucket name, region, DynamoDB table name, etc into every one of your Terraform

modules. Even worse, you have to very carefully *not* copy and paste the `key` value, but ensure a unique `key` for every Terraform module you deploy so that you don't accidentally overwrite the state of some other module! This need to do lots of copy / paste *and* lots of manual changes is error prone, especially if you have to deploy and manage many Terraform modules across many environments.

The only solution available as of May, 2019, is to take advantage of *partial configuration*, where you omit certain parameters from the `backend` configuration in your Terraform code, and instead, pass those in via `-backend-config` command line arguments when calling `terraform init`. For example, you could extract the repeated `backend` arguments, such as `bucket` and `region` into a separate file called `backend.hcl`:

```
bucket         = "terraform-up-and-running-state"
region         = "us-east-2"
dynamodb_table = "terraform-up-and-running-locks"
encrypt        = true
```

Only the `key` parameter remains in the Terraform code, as you still need to set a different `key` value for each module:

```
# Partial configuration. The other settings (e.g.,
bucket, region) will be
# passed in from a file via -backend-config arguments to
'terraform init'
terraform {
  backend "s3" {
    key = "example/terraform.tfstate"
  }
}
```

To put all your partial configurations together, run `terraform init` with the `-backend-config` argument:

```
$ terraform init -backend-config=backend.hcl
```

Terraform will merge the partial config in `backend.hcl` with the partial config in your Terraform code to produce the full configuration used by your module.

Another option is to use Terragrunt, an open source tool that tries to fill in a few gaps in Terraform. Terragrunt can help you keep your `backend` configuration DRY by defining all the basic backend settings (bucket name, region, DynamoDB table name) in one file and automatically setting the `key` argument to the relative folder path of the module. Check out the Terragurnt documentation for the details: *https://github.com/gruntwork-io/terragrunt*

Isolating State Files

With a remote backend and locking, collaboration is no longer a problem. However, there is still one more problem remaining: isolation. When you first start using Terraform, you may be tempted to define all of your infrastructure in a single Terraform file or a single set of Terraform files in one folder. The problem with this approach is that all of your Terraform state is now stored in a single file, too, and a mistake anywhere could break everything.

For example, while trying to deploy a new version of your app in staging, you might break the app in production. Or worse yet, you might corrupt your entire state file, either because you didn't use locking, or due to a rare

Terraform bug, and now all of your infrastructure in all environments is broken.[5]

The whole point of having separate environments is that they are isolated from each other, so if you are managing all the environments from a single set of Terraform configurations, you are breaking that isolation. Just as a ship has bulkheads that act as barriers to prevent a leak in one part of the ship from immediately flooding all the others, you should have "bulkheads" built into your Terraform design, as shown in Figure 3-3.

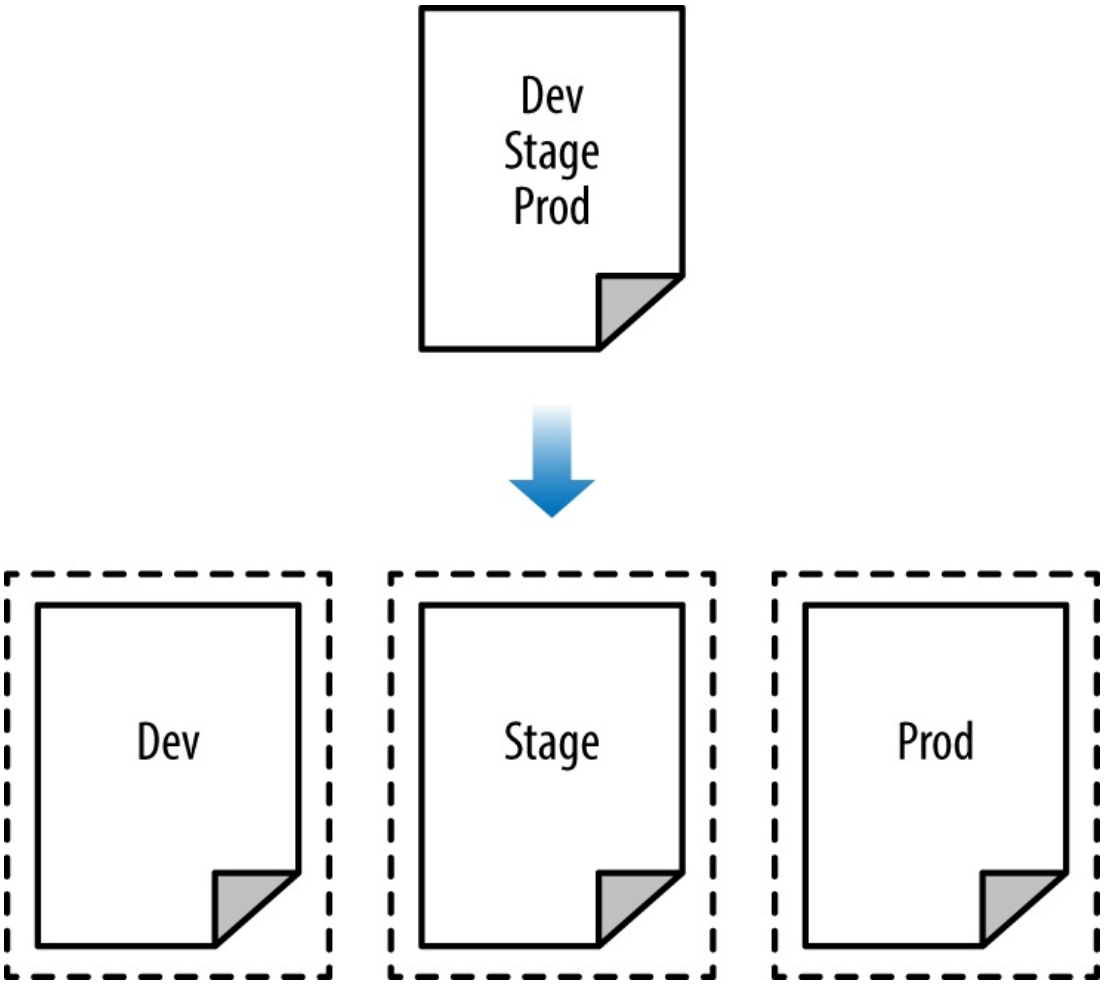

Figure 3-3. Instead of defining all your environments in a single set of Terraform configurations (top), you want to define each environment in a separate set of configurations (bottom), so a problem in one environment is completely isolated from the others.

There are two ways you could isolate state files:

1. Isolation via workspaces: useful for quick, isolated tests on the same configuration.
2. Isolation via file layout: useful for production use-cases where you need strong separation between environments.

Let's dive into each of these in the next two sections.

Isolation via workspaces

Terraform workspaces allow you to store your Terraform state in multiple, separate, named workspaces. Terraform starts with a single workspace called "default" and if you neve explicitly specify a workspace, then the default workspace is the one you'll use the entire time. To create a new workspace or switch between workspaces, you use the `terraform workspace` commands. Let's experiment with workspaces on a standalone module that deploys a single EC2 Instance:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Configure a backend for this instance using the S3 bucket and DynamoDB table you created earlier in the chapter, but with a the `key` value set to `workspaces-example/terraform.tfstate`:

```
terraform {
  backend "s3" {
    # Replace this with your bucket name!
    bucket         = "terraform-up-and-running-state"
    key            = "workspaces-
example/terraform.tfstate"
```

```
    region         = "us-east-2"

    # Replace this with your DynamoDB table name!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Run `terraform init` and `terraform apply` to deploy this module:

```
$ terraform init


Initializing the backend...

Successfully configured the backend "s3"! Terraform will
automatically
use this backend unless the backend configuration
changes.

Initializing provider plugins...

(...)

Terraform has been successfully initialized!


$ terraform apply

(...)

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

The state for this deployment is stored in the default workspace. You can confirm this by running the `terraform workspace show` command, which will tell you which workspace you're currently in:

```
$ terraform workspace show

default
```

The default workspace stores your state in exactly the location you specify via the `key` configuration. As shown in Figure 3-4, if you take a look in your S3 bucket, you'll find a `terraform.tfstate` file in the `workspaces-example` folder.

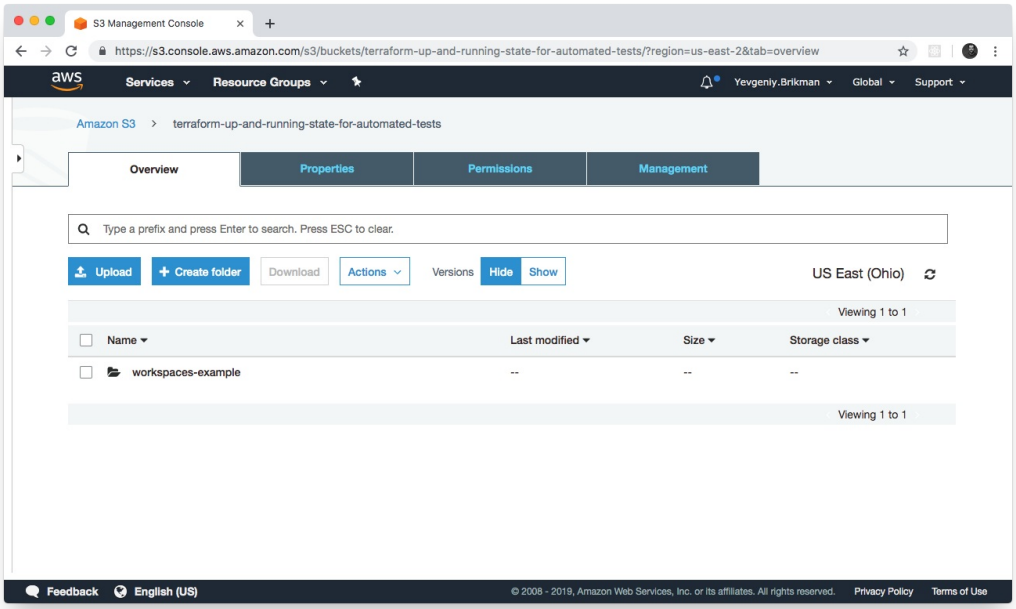

Figure 3-4. The S3 bucket after the state was stored in the default workspace

Let's create a new workspace called "example1" using the `terraform workspace new` command:

```
$ terraform workspace new example1
Created and switched to workspace "example1"!

You're now on a new, empty workspace. Workspaces isolate
their state,
so if you run "terraform plan" Terraform will not see
any existing state
```

```
for this configuration.
```

Now, note what happens if you try to run `terraform plan` (the log output below is truncated for readability):

```
$ terraform plan

Terraform will perform the following actions:

  # aws_instance.example will be created
  + resource "aws_instance" "example" {
      + ami                          = "ami-
0c55b159cbfafe1f0"
      + instance_type                = "t2.micro"
      (...)
    }

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

Terraform wants to create a totally new EC2 Instance from scratch! That's because the state files in each workspace are isolated from each other, and as you're now in the example1 workspace, Terraform isn't using the state file from the default workspace, and therefore, doesn't see the EC2 Instance was already created there.

Try running `terraform apply` to deploy this second EC2 Instance in the new workspace:

```
$ terraform apply

(...)

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

Let’s repeat the exercise one more time and create another workspace called “example2”:

```
$ terraform workspace new example2
Created and switched to workspace "example2"!

You're now on a new, empty workspace. Workspaces isolate
their state,
so if you run "terraform plan" Terraform will not see
any existing state
for this configuration.
```

And run `terraform apply` once again to deploy a third EC2 Instance:

```
$ terraform apply

(...)

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

You now have three workspaces available, which you can see with the `terraform workspace list` command:

```
$ terraform workspace list
  default
  example1
* example2
```

And you can switch between them at any time using the `terraform workspace select` command:

```
$ terraform workspace select example1

Switched to workspace "example1".
```

To understand how this works under the hood, take a look again in your S3 bucket, and you should now see a new folder called `env:`, as shown in Figure 3-5.

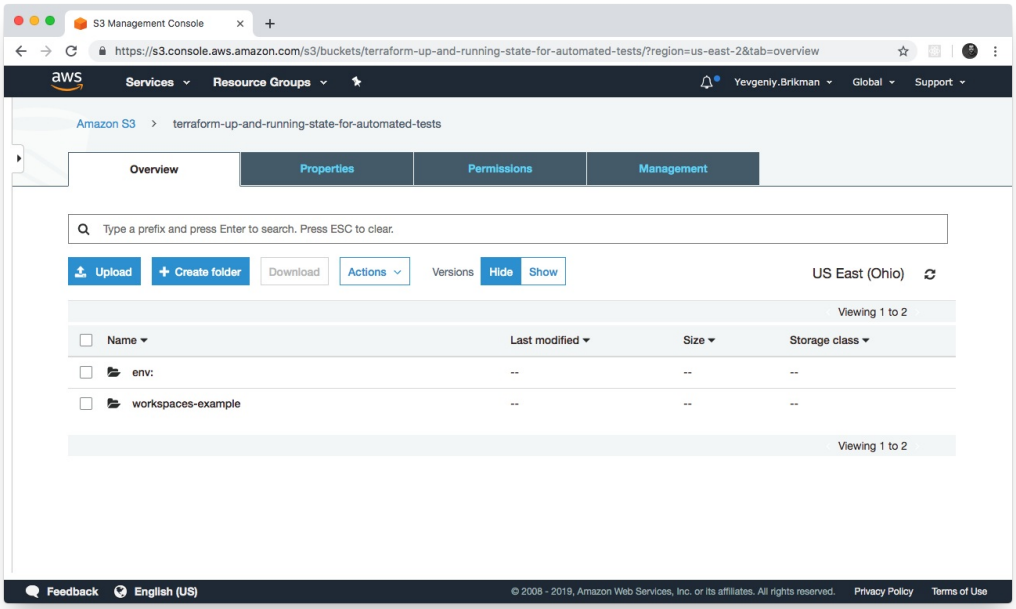

Figure 3-5. The S3 bucket after you've started using custom workspaces

Inside the `env:` folder, you'll find one folder for each of your workspaces, as shown in Figure 3-6.

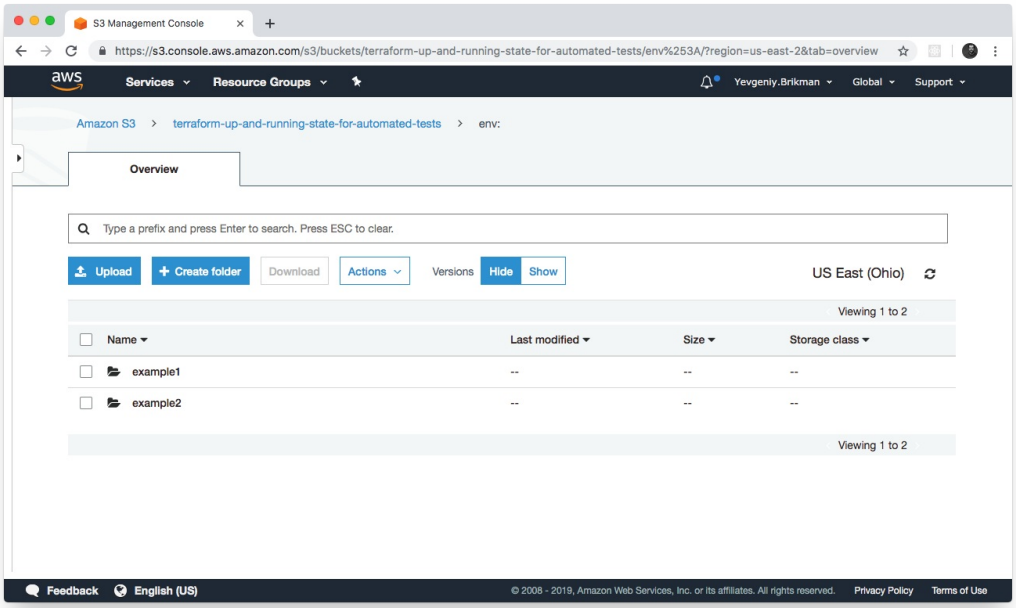

Figure 3-6. Terraform creates one folder per workspace

Inside each of those workspaces, Terraform uses the `key` you specified in your `backend` configuration, so you should find an `example1/workspaces-example/terraform.tfstate` and a `example2/workspaces-example/terraform.tfstate`. In other words, switching to a different workspace is equivalent to changing the path where your state file is stored.

This is handy when you already have a Terraform module deployed, and you want to do some experiments with it (e.g., try to refactor the code), but you don't want your experiments to affect the state of the already deployed infrastructure. Terraform workspaces allow you to run `terraform workspace new` and deploy a new copy of the exact same infrastructure, but storing the state in a separate file.

In fact, you can even change how that module behaves based on the workspace you're in by reading the workspace name using the expression

`terraform.workspace`. For example, here's how to change set the instance type to `t2.medium` in the default workspace and `t2.micro` in all other workspaces (to save money when experimenting):

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = terraform.workspace == "default" ?
"t2.medium" : "t2.micro"
}
```

Terraform workspaces can be a great way to quickly spin up and tear down different versions of your code, but they have a few drawbacks:

1. The state files for all of your workspaces are stored in the same backend (e.g., the same S3 bucket). That means you use the same authentication and access controls for all the workspaces, which is one major reason workspaces are an unsuitable mechanism for isolating environments (e.g., isolating stage from prod).
2. Workspaces are not visible in the code or on the terminal unless you run `terraform workspace` commands. When browsing the code, a module that has been deployed in one workspace looks exactly the same as a module deployed in ten workspaces. This makes maintenance harder, as you don't have a good picture of your infrastructure.
3. Putting the two previous items together, the result is that workspaces can be fairly error prone. The lack of visibility makes it easy to forget what workspace you're in and accidentally make changes in the wrong one (e.g., accidentally running `terraform destroy` in a "production" workspace rather than a "staging" workspace), and since you have to use the same authentication mechanism for all workspaces, you have no other layers of defense to protect against such errors.

To get proper isolation between environments, instead of workspaces,

you'll most likely want to use the file layout, which is the topic of the next section. But before moving on, make sure to clean up the three EC2 Instances you just deployed by running `terraform workspace select <name>` and `terraform destroy` in each of the three workspaces!

Isolation via file layout

To get full isolation between environments, you need to:

1. Put the Terraform configuration files for each environment into a separate folder. For example, all the configurations for the staging environment can be in a folder called *stage* and all the configurations for the production environment can be in a folder called *prod*.
2. Configure a different backend for each environment, using different authentication mechanisms and access controls (e.g., each environment could live in a separate AWS account with a separate S3 bucket as a backend).

With this approach, the use of separate folders makes it much clearer which environments you're deploy to, and the use of separate state files, with separate authentication mechanisms, makes it significantly less likely that a screw up in one environment can have any impact on another.

In fact, you may want to take the isolation concept beyond environments and down to the "component" level, where a component is a coherent set of resources that you typically deploy together. For example, once you've set up the basic network topology for your infrastructure—in AWS lingo, your Virtual Private Cloud (VPC) and all the associated subnets, routing rules, VPNs, and network ACLs—you will probably only change it once every few months, at most. On the other hand, you may deploy a new

version of a web server multiple times per day. If you manage the infrastructure for both the VPC component and the web server component in the same set of Terraform configurations, you are unnecessarily putting your entire network topology at risk of breakage (e.g., from a simple typo in the code or someone accidentally running the wrong command) multiple times per day.

Therefore, I recommend using separate Terraform folders (and therefore separate state files) for each environment (staging, production, etc.) and for each component (vpc, services, databases). To see what this looks like in practice, let's go through the recommended file layout for Terraform projects.

Figure 3-7 shows the file layout for my typical Terraform project.

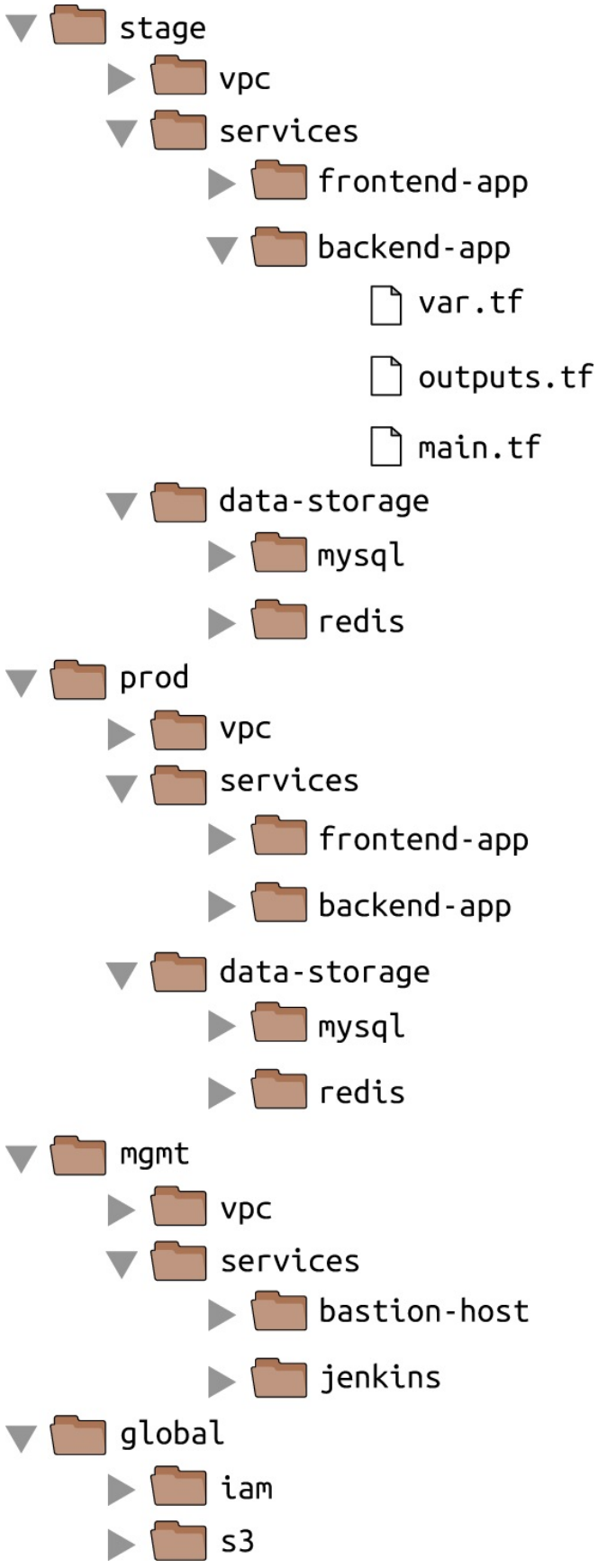
stage
vpc
services
frontend-app
backend-app
var.tf
outputs.tf
main.tf
data-storage
mysql
redis
prod
vpc
services
frontend-app
backend-app
data-storage
mysql
redis
mgmt
vpc
services
bastion-host
jenkins
global
iam
s3

Figure 3-7. Typical file layout for a Terraform project

At the top level, there are separate folders for each "environment." The exact environments differ for every project, but the typical ones are:

stage
: An environment for nonproduction workloads (i.e., testing).

prod
: An environment for production workloads (i.e., user-facing apps).

mgmt
: An environment for DevOps tooling (e.g., bastion host, Jenkins).

global
: A place to put resources that are used across all environments (e.g., S3, IAM).

Within each environment, there are separate folders for each "component." The components differ for every project, but the typical ones are:

vpc
: The network topology for this environment.

services
: The apps or microservices to run in this environment, such as a Ruby on Rails frontend or a Scala backend. Each app could even live in its own folder to isolate it from all the other apps.

data-storage
: The data stores to run in this environment, such as MySQL or Redis. Each data store could even live in its own folder to isolate it from all other data stores.

Within each component, there are the actual Terraform configuration files, which are organized according to the following naming conventions:

variables.tf
: Input variables.

outputs.tf
: Output variables.

main.tf
: The actual resources.

When you run Terraform, it simply looks for files in the current directory with the *.tf* extension, so you can use whatever filenames you want. However, while Terraform may not care about file names, your teammates probably do. Using a consistent, predictable naming convention makes your code easier to browse, as you'll always know where to look to find a variable, output, or resource. If individual Terraform files are becoming massive—especially `main.tf`—it's OK to break out certain functionality into separate files (e.g., *iam.tf*, *s3.tf*, *database.tf*), but that may also be a sign that you should break your code into smaller modules instead, a topic I'll dive into in Chapter 4.

AVOIDING COPY/PASTE

The file layout described in this section has a lot of duplication. For example, the same `frontend-app` and `backend-app` live in both the *stage* and *prod* folders. Don't worry, you won't need to copy/paste all of that code! In Chapter 4, you'll see how to use Terraform modules to keep all of this code DRY.

Let’s take the web server cluster code you wrote in Chapter 2, plus the S3 and DynamoDB code you wrote in this chapter, and rearrange it using the folder structure in Figure 3-8.

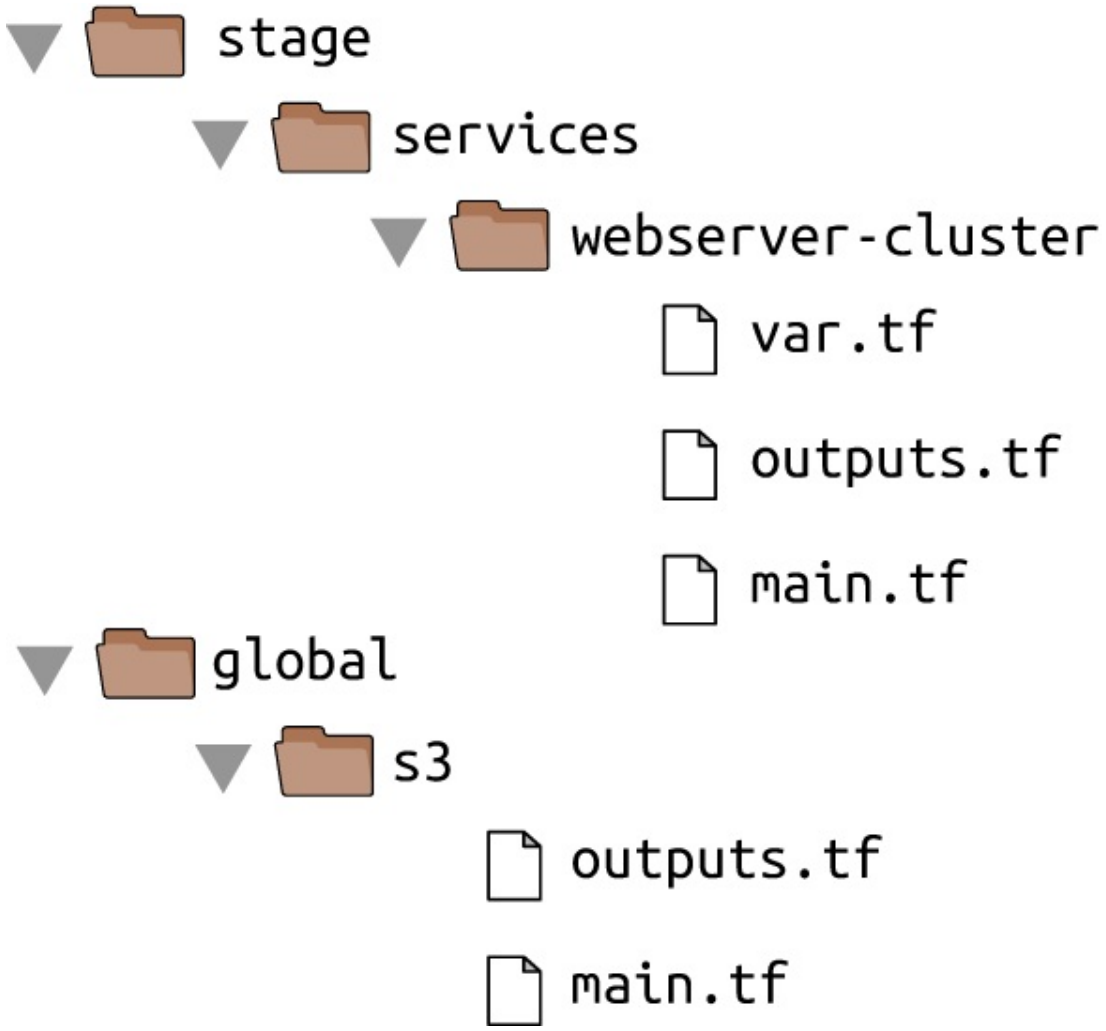

Figure 3-8. File layout for the web server cluster code

The S3 bucket you created in this chapter should be moved into the *global/s3* folder. Move the output variables (`s3_bucket_arn` and `dynamodb_table_name`) into *outputs.tf*. When moving the folder, make sure you don’t miss the (hidden) *.terraform* folder when copying files to the new location so you don’t have to re-initialize everything.

The web server cluster you created in Chapter 2 should be moved into

stage/services/webserver-cluster (think of this as the "testing" or "staging" version of that web server cluster; you'll add a "production" version in the next chapter). Again, make sure to copy over the *.terraform* folder, move input variables into *variables.tf*, and output variables into *outputs.tf*.

You should also update the web server cluster to use S3 as a `backend`. You can copy and paste the `backend` config from `global/s3/main.tf` more or less verbatim, but make sure to change the `key` to the same folder path as the web server Terraform code: *stage/services/webserver-cluster/terraform.tfstate*. This gives you a 1:1 mapping between the layout of your Terraform code in version control and your Terraform state files in S3, so it's obvious how the two are connected. The `s3` module already sets the `key` using this convention.

This file layout makes it easy to browse the code and understand exactly what components are deployed in each environment. It also provides a good amount of isolation between environments and between components within an environment, ensuring that if something goes wrong, the damage is contained as much as possible to just one small part of your entire infrastructure.

Of course, this very same property is, in some ways, a drawback, too: splitting components into separate folders prevents you from accidentally blowing up your entire infrastructure in one command, but it also prevents you from creating your entire infrastructure in one command. If all of the components for a single environment were defined in a single Terraform configuration, you could spin up an entire environment with a single call to `terraform apply`. But if all the components are in separate folders, then you need to run `terraform apply` separately in each one (note that if you're using Terragrunt, you can automate this process using the

`apply-all` command[6]).

There is another problem with this file layout: it makes it harder to use resource dependencies. If your app code was defined in the same Terraform configuration files as the database code, then that app could directly access attributes of the database (e.g., the database address and port) using an attribute reference (e.g., `aws_db_instance.foo.address`). But if the app code and database code live in different folders, as I've recommended, you can no longer do that. Fortunately, Terraform offers a solution: the `terraform_remote_state` data source.

The terraform_remote_state data source

In Chapter 2, you used data sources to fetch read-only information from AWS, such as the `aws_subnet_ids` data source, which returns a list of subnets in your VPC. There is another data source that is particularly useful when working with state: `terraform_remote_state`. You can use this data source to fetch the Terraform state file stored by another set of Terraform configurations in a completely read-only manner.

Let's go through an example. Imagine that your web server cluster needs to talk to a MySQL database. Running a database that is scalable, secure, durable, and highly available is a lot of work. Once again, you can let AWS take care of it for you, this time by using the *Relational Database Service (RDS)*, as shown in Figure 3-9. RDS supports a variety of databases, including MySQL, PostgreSQL, SQL Server, and Oracle.

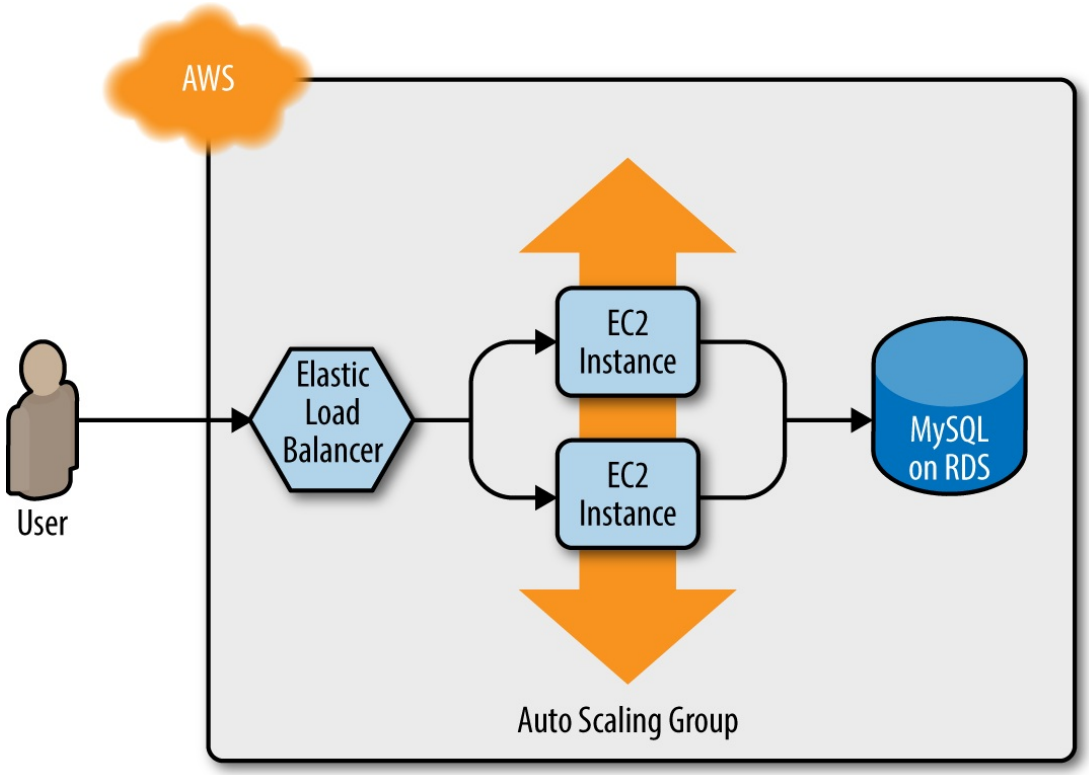

Figure 3-9. The web server cluster talks to MySQL, which is deployed on top of Amazon's Relational Database Service

You may not want to define the MySQL database in the same set of configuration files as the web server cluster, as you'll be deploying updates to the web server cluster far more frequently and don't want to risk accidentally breaking the database each time you do so. Therefore, your first step should be to create a new folder at *stage/data-stores/mysql* and create the basic Terraform files (*main.tf*, *variables.tf*, *outputs.tf*) within it, as shown in Figure 3-10.

Next, create the database resources in *stage/data-stores/mysql/main.tf*:

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

resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
```

```
  engine               = "mysql"
  allocated_storage    = 10
  instance_class       = "db.t2.micro"
  name                 = "example_database"
  username             = "admin"

  # How should we set the password?
  password             = "???"
}
```

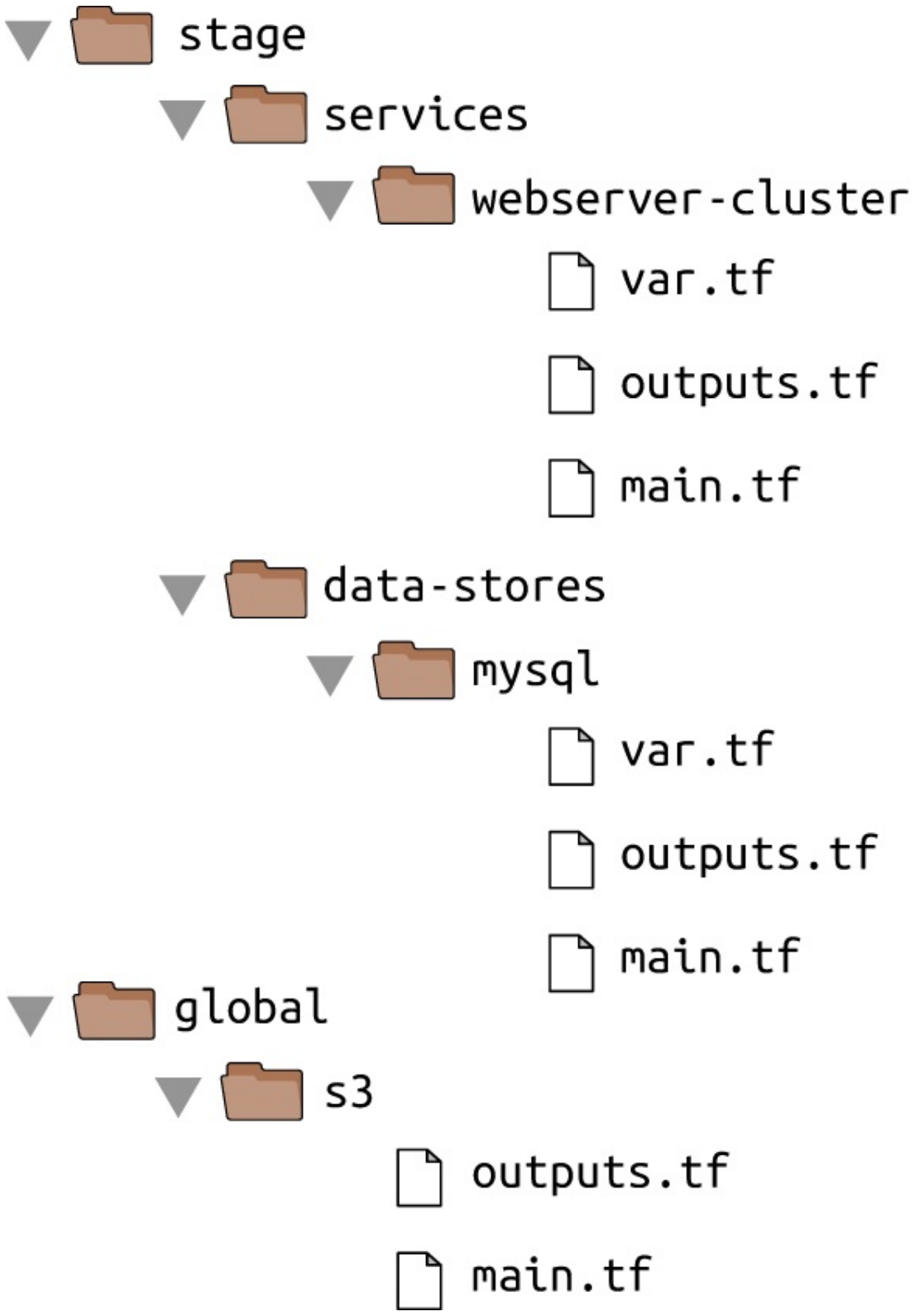

Figure 3-10. Create the database code in the stage/data-stores folder

At the top of the file, you see the typical `provider` resource, but just

below that is a new resource: `aws_db_instance`. This resource creates a database in RDS. The settings in this code configure RDS to run MySQL with 10GB of storage on a `db.t2.micro` instance, which has 1 virtual CPU, 1GB of memory, and is part of the AWS free tier.

Note that one of the parameters you have to pass to the `aws_db_instance` resource is the master password to use for the database. Since this is a secret, you should not put it directly into your code in plaintext! Instead, there are two better options for passing secrets to Terraform resources.

One option for handling secrets is to use a Terraform data source to read the secrets from a secret store. For example, you can store secrets, such as database passwords, in AWS Secrets Manager, a managed service AWS offers specifically for storing sensitive data. You could use the AWS Secrets Manager UI to store the secret and then read the secret back out in your Terraform code using the `aws_secretsmanager_secret_version` data source:

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

resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  name                = "example_database"
  username            = "admin"
  password            =

data.aws_secretsmanager_secret_version.db_password.secre
t_string
}
```

```
data "aws_secretsmanager_secret_version" "db_password" {
  secret_id = "mysql-master-password-stage"
}
```

Some of the supported secret stores and data source combos you could look into are:

1. AWS Secrets Manager and the `aws_secretsmanager_secret_version` data source (shown above).
2. AWS Systems Manager Parameter Store and the `aws_ssm_parameter` data source.
3. AWS KMS and the `aws_kms_secrets` data source.
4. Google Cloud KMS and the `google_kms_secret` data source.
5. Azure Key Vault and the `azurerm_key_vault_secret` data source.
6. HashiCorp Vault and the `vault_generic_secret` data source.

The second option for handling secrets is to manage them completely outside of Terraform (e.g., in a password manager such as 1Password, LastPass, or OS X Keychain) and to pass the secret into Terraform via an environment variable. To do that, declare a variable called `db_password` in *stage/data-stores/mysql/variables.tf*:

```
variable "db_password" {
  description = "The password for the database"
  type        = string
}
```

Note that this variable does not have a `default`. This is intentional. You should not store your database password or any sensitive information in plain text. Instead, you'll set this variable using an environment variable.

As a reminder, for each input variable `foo` defined in your Terraform configurations, you can provide Terraform the value of this variable using the environment variable `TF_VAR_foo`. For the `db_password` input variable, here is how you can set the `TF_VAR_db_password` environment variable on Linux/Unix/OS X systems (note there is a space before the `export` command to prevent the secret from being stored in Bash history[7]):

```
$  export TF_VAR_db_password="(YOUR_DB_PASSWORD)"
$ terraform apply

(...)
```

SECRETS ARE ALWAYS STORED IN TERRAFORM STATE

Reading secrets from a secrets store or environment variables is a good practice to ensure secrets aren't stored in plaintext in your code, but just a reminder: no matter how you read in the secret, if you pass it as an argument to a Terraform resource, such as `aws_db_instance`, that secret will be stored in the Terraform state file, in plain text.

This is a known weakness of Terraform, with no effective solutions available, so be extra paranoid with how you store your state files (e.g., always enable encryption) and who can access those state files (e.g., use IAM permissions to lock down access to your S3 bucket)!

Now that you've configured the password, the next step is to configure this module to store its state in the S3 bucket you created earlier at the path `stage/data-stores/mysql/terraform.tfstate`:

```
terraform {
  backend "s3" {
    # Replace this with your bucket name!
    bucket         = "terraform-up-and-running-state"
    key            = "stage/data-
stores/mysql/terraform.tfstate"
    region         = "us-east-2"

    # Replace this with your DynamoDB table name!
    dynamodb_table = "terraform-up-and-running-locks"
    encrypt        = true
  }
}
```

Run `terraform apply` to create the database. Note that RDS can take ~10 minutes to provision even a small database, so be patient!

Now that you have a database, how do you provide its address and port to your web server cluster? The first step is to add two output variables to *stage/data-stores/mysql/outputs.tf*:

```
output "address" {
  value       = aws_db_instance.example.address
  description = "Connect to the database at this
endpoint"
}

output "port" {
  value       = aws_db_instance.example.port
  description = "The port the database is listening on"
}
```

Run `terraform apply` one more time and you should see the outputs in the terminal:

```
$ terraform apply

(...)

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

Outputs:

address = tf-2016111123.cowu6mts6srx.us-east-
2.rds.amazonaws.com
port = 3306
```

These outputs are now also stored in the Terraform state for the database, which is in your S3 bucket at the path *stage/data-stores/mysql/terraform.tfstate*. You can get the web server cluster code to read the data from this state file by adding the `terraform_remote_state` data source in *stage/services/webserver-cluster/main.tf*:

```
data "terraform_remote_state" "db" {
  backend = "s3"

  config = {
    bucket = "(YOUR_BUCKET_NAME)"
    key    = "stage/data-stores/mysql/terraform.tfstate"
    region = "us-east-2"
  }
}
```

This `terraform_remote_state` data source configures the web server cluster code to read the state file from the same S3 bucket and folder where the database stores its state, as shown in Figure 3-11.

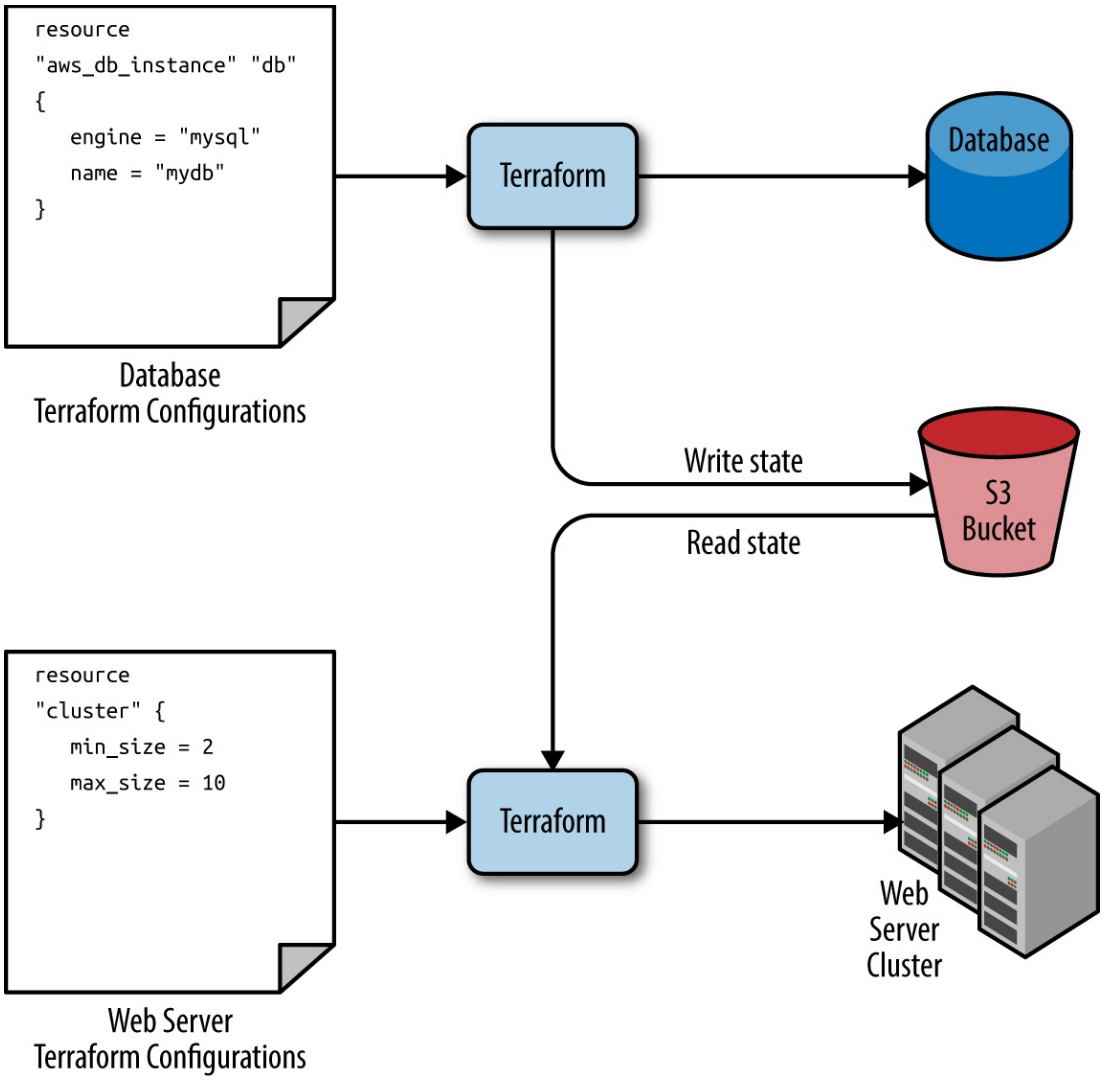

Figure 3-11. The database writes its state to an S3 bucket (top) and the web server cluster reads that state from the same bucket (bottom)

It's important to understand that, like all Terraform data sources, the data returned by `terraform_remote_state` is read-only. Nothing you do in your web server cluster Terraform code can modify that state, so you can pull in the database's state data with no risk of causing any problems in the database itself.

All the database's output variables are stored in the state file and you can read them from the `terraform_remote_state` data source using an attribute reference of the form:

```
data.terraform_remote_state.<NAME>.outputs.<ATTRIBUTE>
```

For example, here is how you can update the User Data of the web server cluster instances to pull the database address and port out of the `terraform_remote_state` data source and expose that information in the HTTP response:

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

As the User Data script is getting longer, defining it inline is getting messier and messier. In general, embedding one programming language (Bash) inside another (Terraform) makes it harder to maintain each one, so let's pause here for a moment to externalize the Bash script. To do that, you can use the `file` built-in function and the `template_file` data source. Let's talk about these one at a time.

Terraform includes a number of *built-in function* that you can execute using an expression of the form:

```
function_name(...)
```

For example, consider the `format` function:

```
format(<FMT>, <ARGS>, ...)
```

This function formats the arguments in `ARGS` according to the `sprintf` syntax in the string `FMT`.[8] A great way to experiment with built-in functions is to run the `terraform console` command to get an interactive console where you can try out different Terraform syntax, query the state of your infrastructure, and see the results instantly:

```
terraform console

$ format("%.3f", 3.14159265359)
3.142
```

Note that the Terraform console is read-only, so you don't have to worry about accidentally changing infrastructure or state!

There are a number of other built-in functions that can be used to manipulate strings, numbers, lists, and maps.[9] One of them is the `file` function:

```
file(<PATH>)
```

This function reads the file at PATH and returns its contents as a string. For example, you could put your User Data script into *stage/services/webserver-cluster/user-data.sh* and load its contents as a string as follows:

```
file("user-data.sh")
```

The catch is that the User Data script for the web server cluster needs some dynamic data from Terraform, including the server port, database address, and database port. When the User Data script was embedded in

the Terraform code, you used Terraform references and interpolation to fill in these values. This does not work with the `file` function. However, it does work if you use a `template_file` data source.

The `template_file` data source has two arguments: `template`, which is a string to render, and `vars`, which is a map of variables to make available while rendering. It has one output attribute called `rendered`, which is the result of rendering `template`, including any interpolation syntax in `template`, with the variables available in `vars`. To see this in action, add the following `template_file` data source to *stage/services/webserver-cluster/main.tf*:

```
data "template_file" "user_data" {
  template = file("user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  =
data.terraform_remote_state.db.outputs.address
    db_port     =
data.terraform_remote_state.db.outputs.port
  }
}
```

You can see that this code sets the `template` parameter to the contents of the *user-data.sh* script and the `vars` parameter to the three variables the User Data script needs: the server port, database address, and database port. To use these variables, you'll need to update *stage/services/webserver-cluster/user-data.sh* script as follows:

```
#!/bin/bash

cat > index.html <<EOF
```

```
<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

nohup busybox httpd -f -p ${server_port} &
```

Note that this Bash script has a few changes from the original:

- It looks up variables using Terraform's standard interpolation syntax, but the only available variables are the ones in the `vars` map of the `template_file` data source. Note that you don't need any prefix to access those variables: e.g., you should use `server_port` and not `var.server_port`.
- The script now includes some HTML syntax (e.g., `<h1>`) to make the output a bit more readable in a web browser.

A NOTE ON EXTERNALIZED FILES

One of the benefits of extracting the User Data script into its own file is that you can write unit tests for it. The test code can even fill in the interpolated variables by using environment variables, since the Bash syntax for looking up environment variables is the same as Terraform's interpolation syntax. For example, you could write an automated test for *user-data.sh* along the following lines:

```
export db_address=12.34.56.78
export db_port=5555
export server_port=8888

./user-data.sh

output=$(curl "http://localhost:$server_port")

if [[ $output == *"Hello, World"* ]]; then
  echo "Success! Got expected text from server."
else
  echo "Error. Did not get back expected text 'Hello,
```

```
  World'."
  fi
```

The final step is to update the `user_data` parameter of the `aws_launch_configuration` resource to point to the `rendered` output attribute of the `template_file` data source:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
  instance_type   = "t2.micro"
  security_groups = [aws_security_group.instance.id]
  user_data       =
data.template_file.user_data.rendered
}
```

Ah, that's much cleaner than writing Bash scripts inline!

If you deploy this cluster using `terraform apply`, wait for the Instances to register in the ALB, and open the ALB URL in a web browser, you'll see something similar to Figure 3-12.

Yay, your web server cluster can now programmatically access the database address and port via Terraform! If you were using a real web framework (e.g., Ruby on Rails), you could set the address and port as environment variables or write them to a config file so they could be used by your database library (e.g., ActiveRecord) to talk to the database.

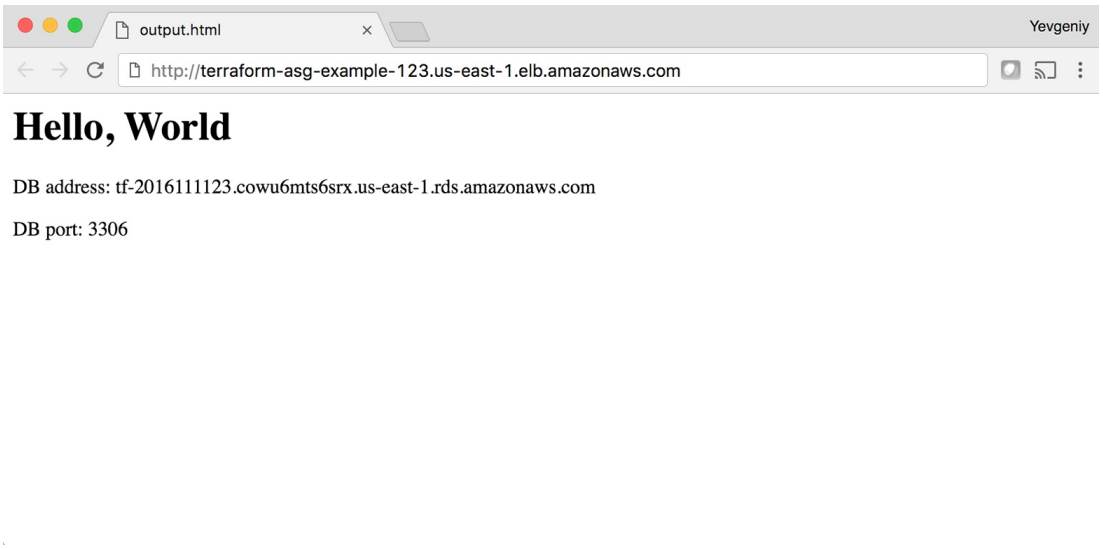

Figure 3-12. The web server cluster can programmatically access the database address and port

Conclusion

The reason you need to put so much thought into isolation, locking, and state is that infrastructure as code (IAC) has different trade-offs than normal coding. When you're writing code for a typical app, most bugs are relatively minor and only break a small part of a single app. When you're writing code that controls your infrastructure, bugs tend to be more severe, as they can break all of your apps—and all of your data stores and your entire network topology and just about everything else. Therefore, I recommend including more "safety mechanisms" when working on IAC than with typical code.[10]

A common concern of using the recommended file layout is that it leads to code duplication. If you want to run the web server cluster in both staging and production, how do you avoid having to copy and paste a lot of code between *stage/services/webserver-cluster* and *prod/services/webserver-cluster*? The answer is that you need to use Terraform modules, which are the main topic of Chapter 4.

1 Learn more about S3's guarantees here: *https://aws.amazon.com/s3/details/#durability*.

2 See pricing information for S3 here: *https://aws.amazon.com/s3/pricing/*.

3 See here for more information on S3 bucket names: *http://bit.ly/2b1s7eh*.

4 See pricing information for DynamoDB here: *https://aws.amazon.com/dynamodb/pricing/*

5 For a colorful example of what happens when you don't isolate Terraform state, see: *http://bit.ly/2lTsewM*.

6 For more information, see Terragrunt's documentation.

7 In most Linux/Unix/OS X shells, every command you type gets stored in some sort of history file (e.g., `~/.bash_history`). If you start your command with a space, then most shells will skip writing that command to the history file. Note that you may need to set the `HISTCONTROL` environment variable to "ignoreboth" to enable this if your shell doesn't enable it by default.

8 You can find documentation for the `sprintf` syntax here: *https://golang.org/pkg/fmt/*.

9 You can find the full list of built-in functions here: *https://www.terraform.io/docs/configuration/functions.html*.

10 For more information on software safety mechanisms, see *http://www.ybrikman.com/writing/2016/02/14/agility-requires-safety/*.

Chapter 4. How to Create Reusable Infrastructure with Terraform Modules

At the end of Chapter 3, you had deployed the architecture shown in Figure 4-1.

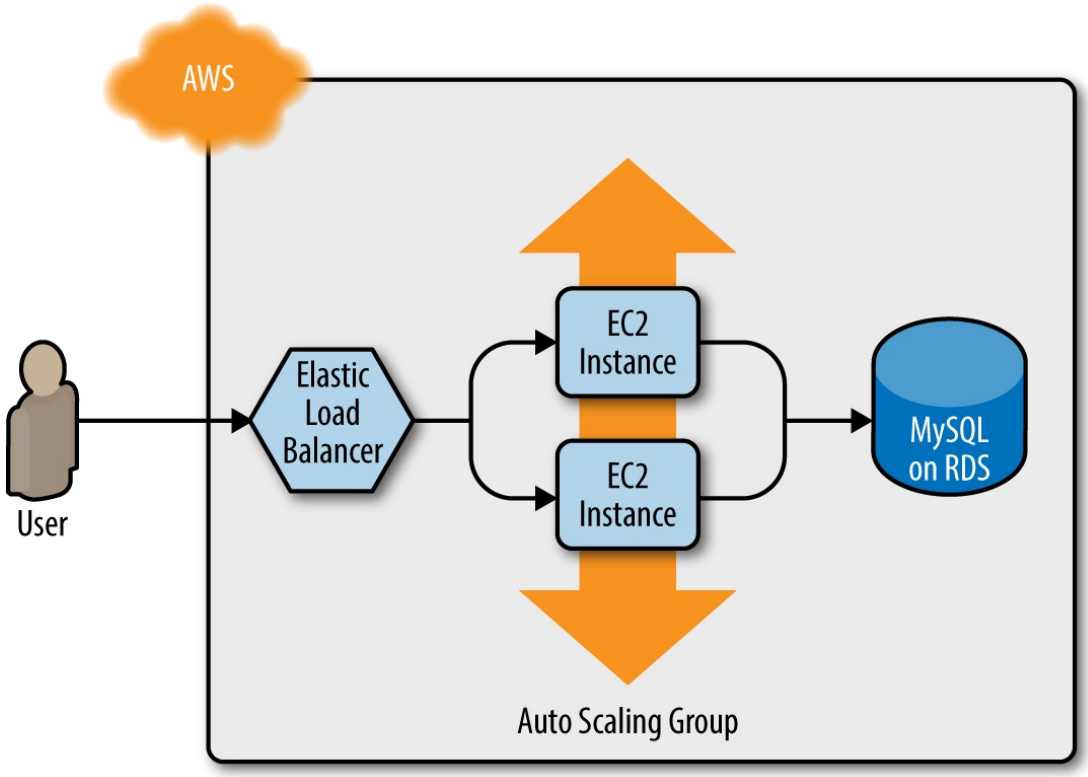

Figure 4-1. A load balancer, web server cluster, and database

This works great as a staging environment, but what about the production environment? You don't want your users accessing the same environment your employees use for testing, and it's too risky testing in production, so you typically need two environments, staging and production, as shown in

Figure 4-2. Ideally, the two environments are nearly identical, though you may run slightly fewer/smaller servers in staging to save money.

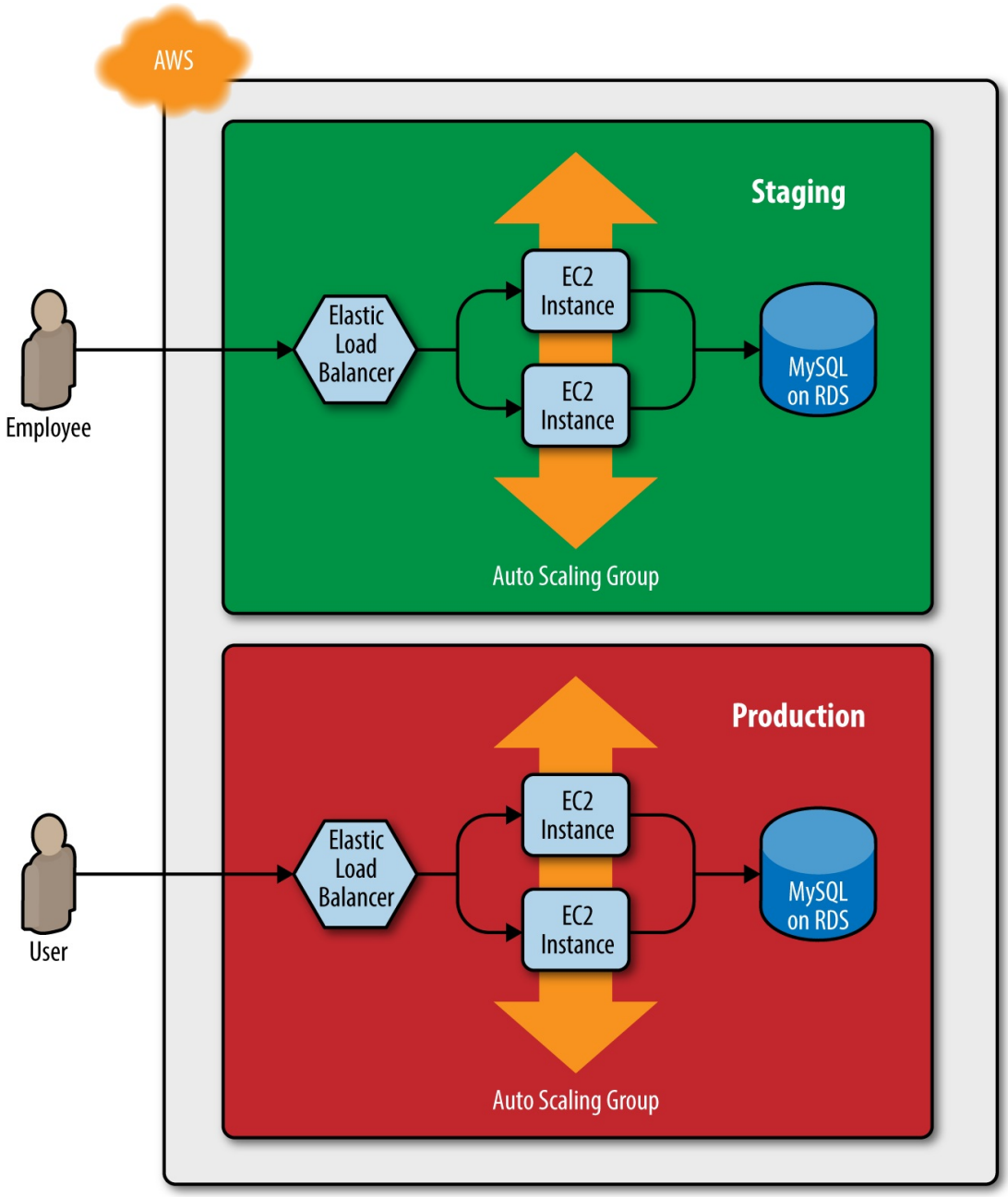

Figure 4-2. Two environments, each with its own load balancer, web server cluster, and database

With just a staging environment, the file layout for your Terraform code looked something like Figure 4-3.

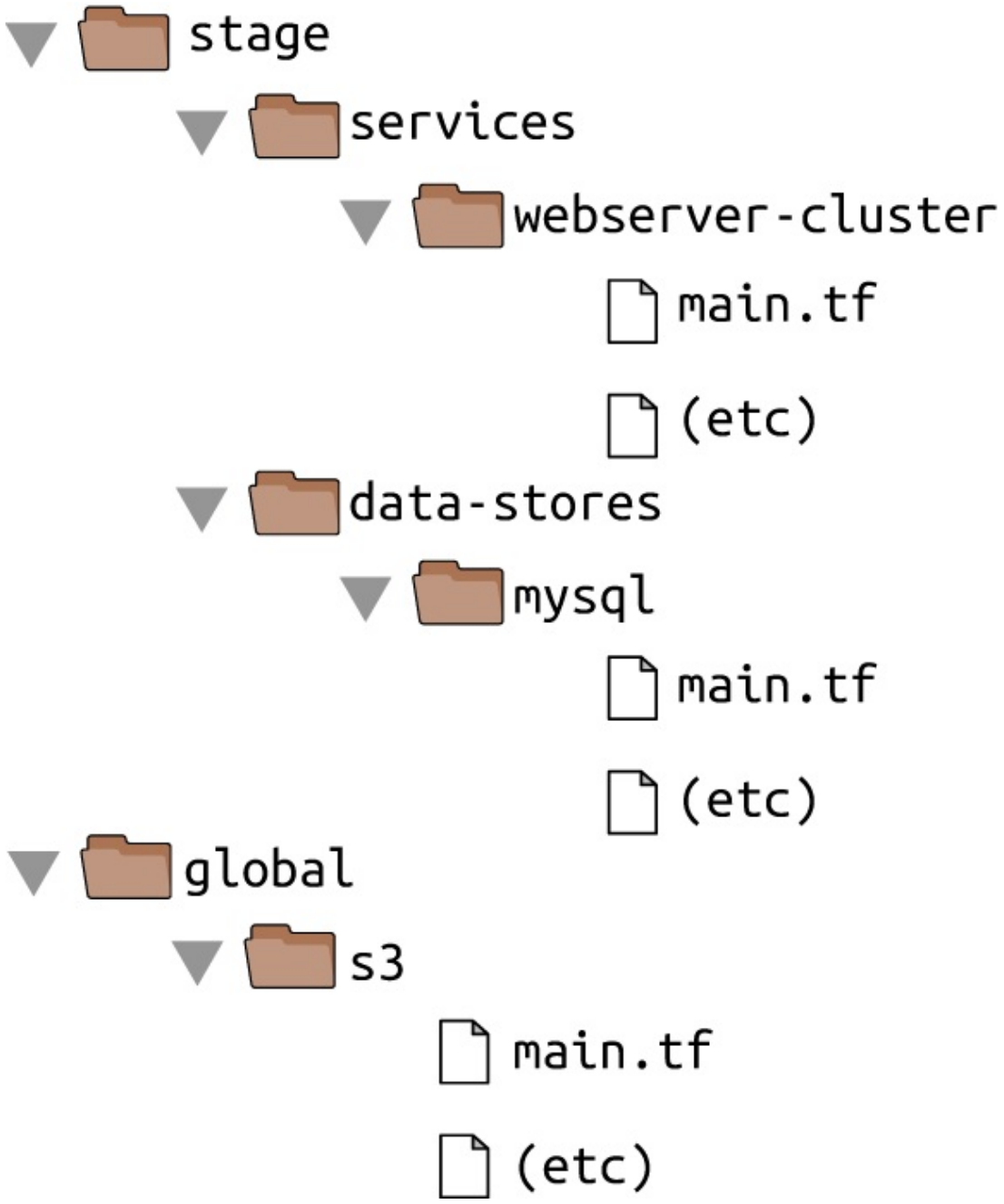

Figure 4-3. File layout with only a staging environment

If you were to add a production environment, you'd end up with the file layout in Figure 4-4.

How do you avoid duplication between the staging and production environments? How do you avoid having to copy and paste all the code in

stage/services/webserver-cluster into *prod/services/webserver-cluster* and all the code in *stage/data-stores/mysql* into *prod/data-stores/mysql*?

In a general-purpose programming language (e.g., Ruby, Python, Java), if you had the same code copied and pasted in several places, you could put that code inside of a function and reuse that function in multiple places throughout your code:

```
def example_function()
  puts "Hello, World"
end

# Other places in your code
example_function()
```

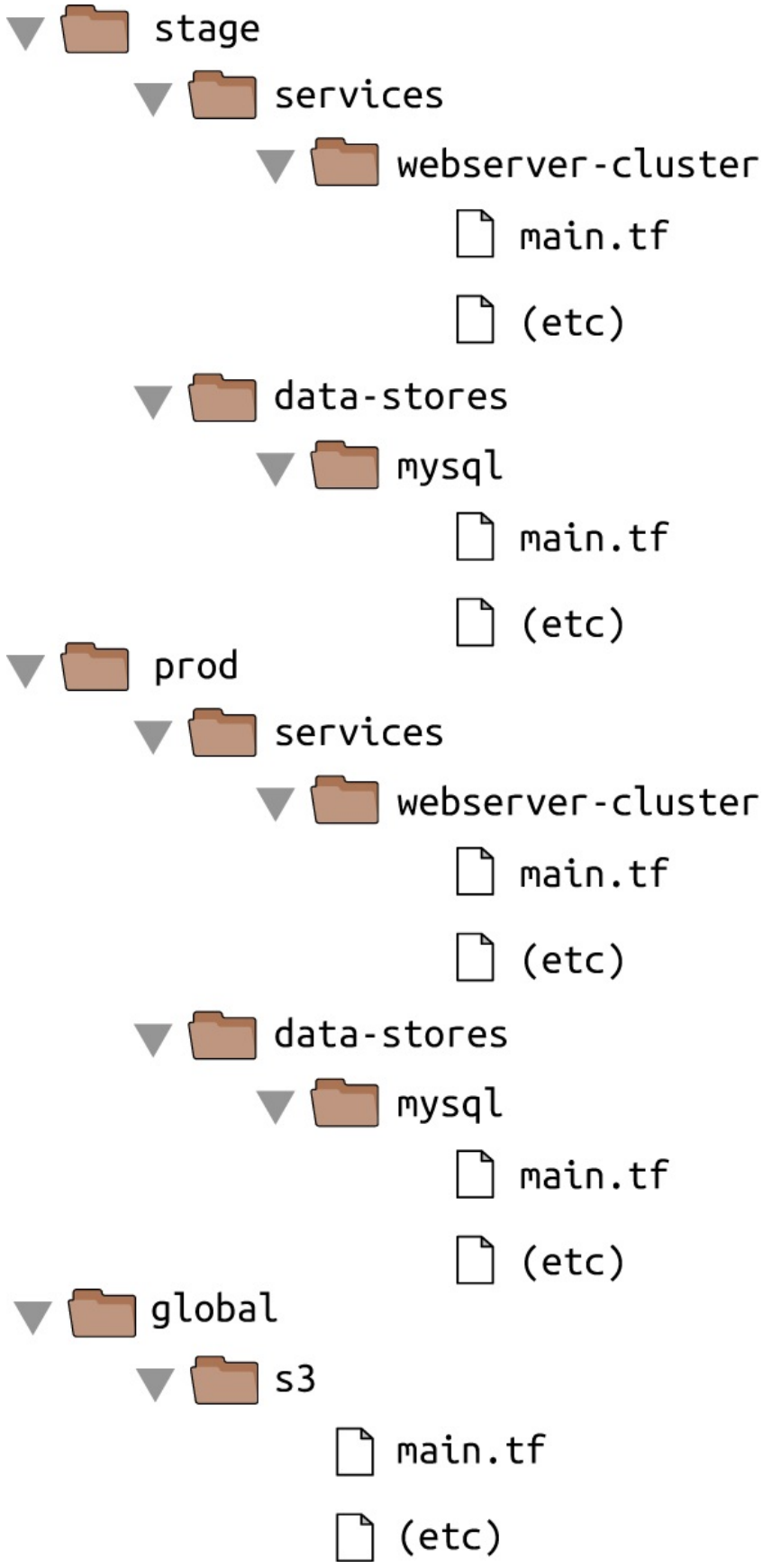
stage
services
webserver-cluster
main.tf
(etc)
data-stores
mysql
main.tf
(etc)
prod
services
webserver-cluster
main.tf
(etc)
data-stores
mysql
main.tf
(etc)
global
s3
main.tf
(etc)

Figure 4-4. File layout with a staging and production environment

With Terraform, you can put your code inside of a *Terraform module* and reuse that module in multiple places throughout your code. The *stage/services/webserver-cluster* and *prod/services/webserver-cluster* configurations can both reuse code from the same module without the need to copy and paste (see Figure 4-5).

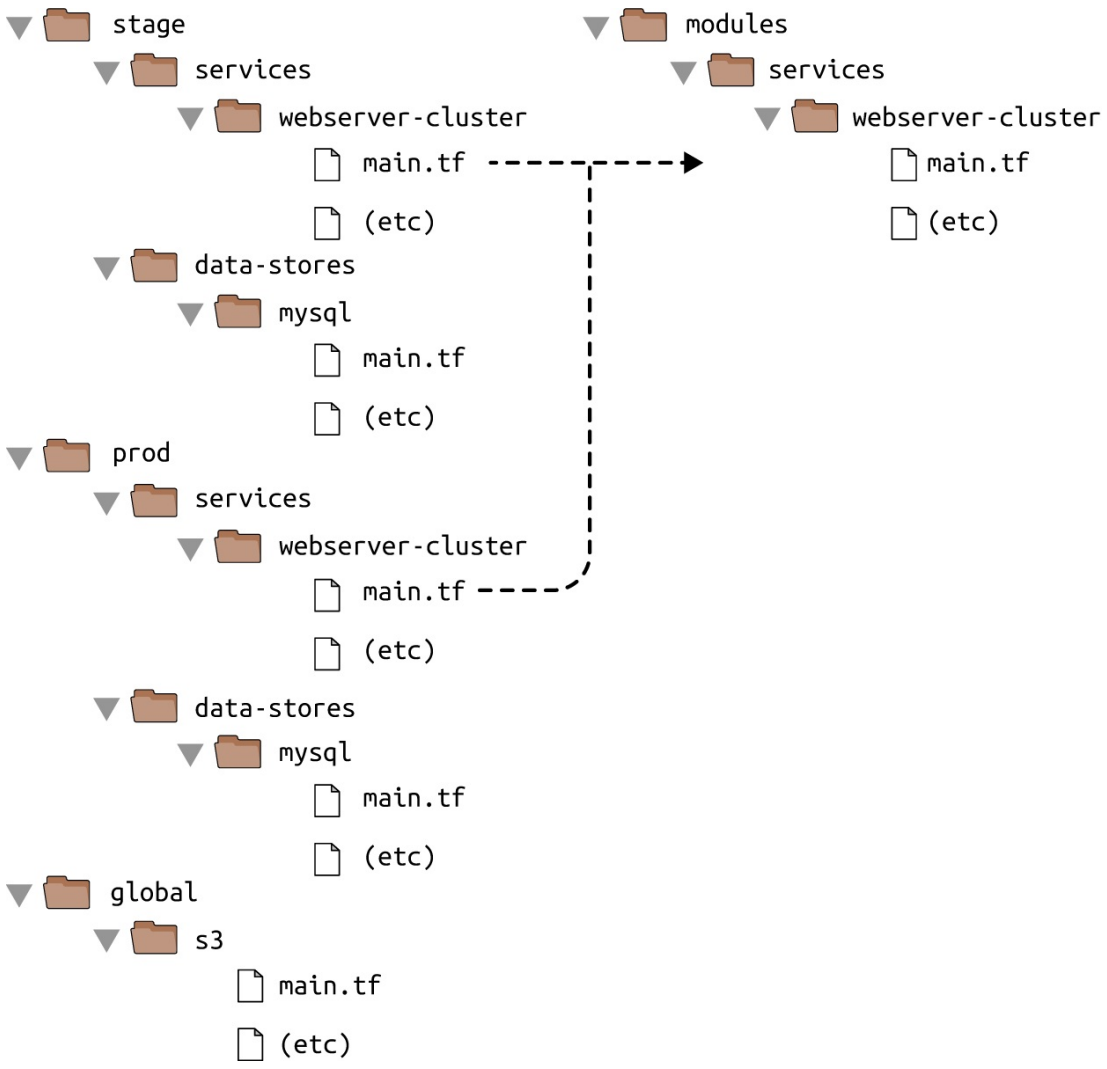

Figure 4-5. Putting code into modules allows you to reuse that code from multiple environments

In this chapter, I'll show you how to create and use Terraform modules by covering the following topics:

- Module basics
- Module inputs
- Module outputs
- Module gotchas
- Module versioning

> **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*.

Module Basics

A Terraform module is very simple: any set of Terraform configuration files in a folder is a module. All the configurations you've written so far have technically been modules, although not particularly interesting ones, since you deployed them directly (the module in the current working directory is called the *root module*). To see what modules are really capable of, you have to use one module from another module.

As an example, let's turn the code in *stage/services/webserver-cluster*, which includes an Auto Scaling Group (ASG), Application Load Balancer (ALB), security groups, and many other resources, into a reusable module.

As a first step, run `terraform destroy` in the *stage/services/webserver-cluster* to clean up any resources you created earlier. Next, create a new top-level folder called *modules* and move all

the files from *stage/services/webserver-cluster* to *modules/services/webserver-cluster*. You should end up with a folder structure that looks something like Figure 4-6.

Open up the *main.tf* file in *modules/services/webserver-cluster* and remove the `provider` definition. This should be defined by the user of the module and not in the module itself.

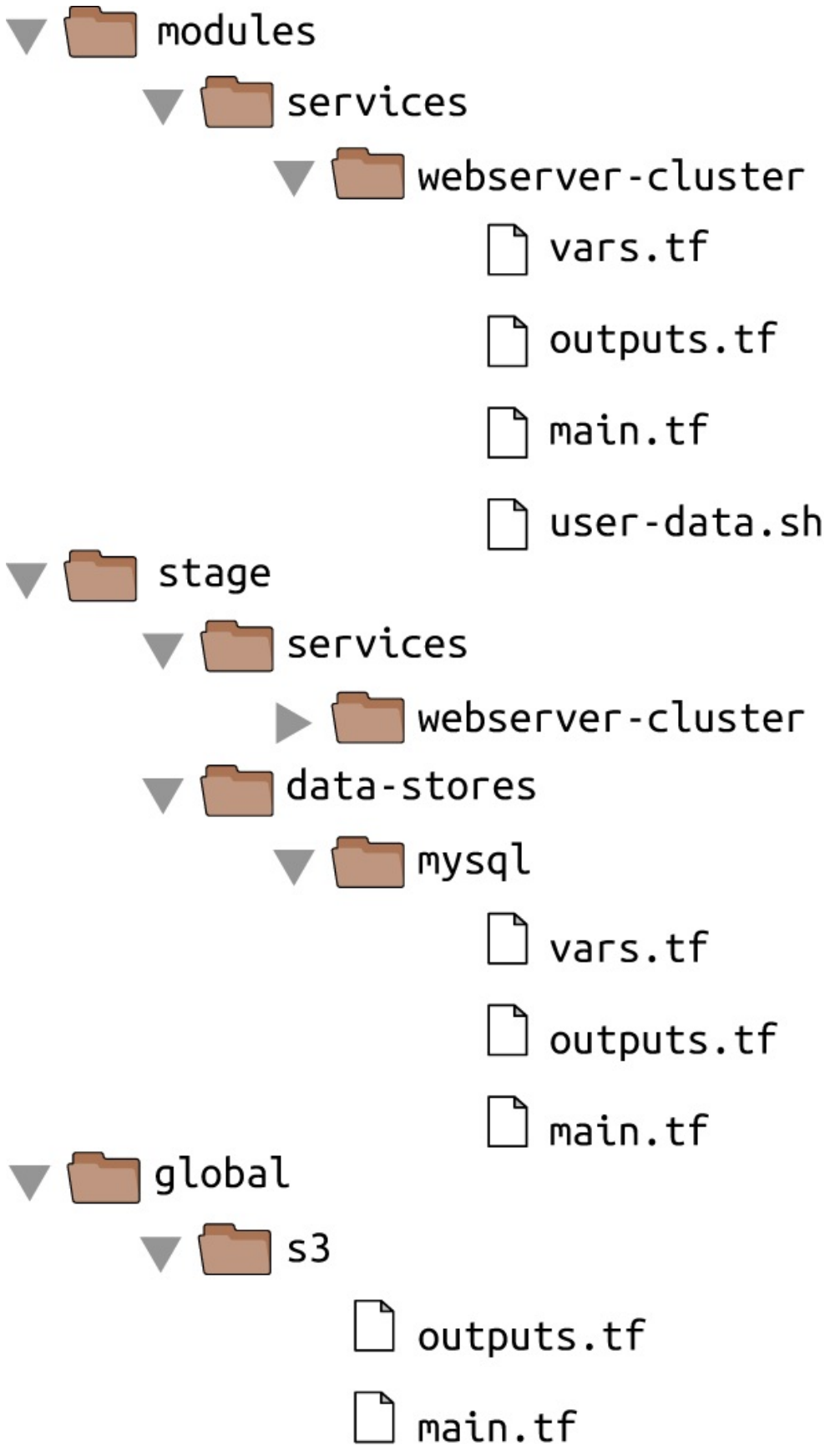
modules
services
webserver-cluster
vars.tf
outputs.tf
main.tf
user-data.sh
stage
services
webserver-cluster
data-stores
mysql
vars.tf
outputs.tf
main.tf
global
s3
outputs.tf
main.tf

Figure 4-6. The folder structure with a module and a staging environment

You can now make use of this module in the stage environment. The syntax for using a module is:

```
module "<NAME>" {
  source = "<SOURCE>"

  [CONFIG ...]
}
```

Within the module definition, the `source` parameter specifies the folder where the module's code can be found. For example, you can create a new file in *stage/services/webserver-cluster/main.tf* and use the `webserver-cluster` module in it as follows:

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

module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"
}
```

You can then reuse the exact same module in the production environment by creating a new *prod/services/webserver-cluster/main.tf* file with the following contents:

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

module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"
}
```

And there you have it: code reuse in multiple environments without any copy/paste! Note that whenever you add a module to your Terraform configurations or modify the `source` parameter of a module, you need to run the `init` command before you run `plan` or `apply`:

```
$ terraform init
Initializing modules...
- webserver_cluster in
../../../modules/services/webserver-cluster

Initializing the backend...

Initializing provider plugins...

Terraform has been successfully initialized!
```

Now you've seen all the tricks the `init` command has up its slieve! It downloads providers, modules, and configures your backends, all in one handy command.

Before you run the `apply` command on this code, you should note that there is a problem with the `webserver-cluster` module: all the names are hard-coded. That is, the name of the security groups, ALB, and other resources are all hard-coded, so if you use this module more than once, you'll get name conflict errors. Even the database details are hard-coded because the *main.tf* file you copied into *modules/services/webserver-cluster* is using a+terraform_remote_state+ data source to figure out the database address and port, and that `terraform_remote_state` is hard-coded to look at the staging environment.

To fix these issues, you need to add configurable inputs to the `webserver-cluster` module so it can behave differently in different

environments.

Module Inputs

To make a function configurable in a general-purpose programming language, you can add input parameters to that function:

```
def example_function(param1, param2)
  puts "Hello, #{param1} #{param2}"
end

# Other places in your code
example_function("foo", "bar")
```

In Terraform, modules can have input parameters, too. To define them, you use a mechanism you're already familiar with: input variables. Open up *modules/services/webserver-cluster/variables.tf* and add three new input variables:

```
variable "cluster_name" {
  description = "The name to use for all the cluster
resources"
  type        = string
}

variable "db_remote_state_bucket" {
  description = "The name of the S3 bucket for the
database's remote state"
  type        = string
}

variable "db_remote_state_key" {
  description = "The path for the database's remote
state in S3"
  type        = string
}
```

Next, go through *modules/services/webserver-cluster/main.tf* and use `var.cluster_name` instead of the hard-coded names (e.g., instead of `"terraform-asg-example"`). For example, here is how you do it for the ALB security group:

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  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"]
  }
}
```

Notice how the `name` parameter is set to `"${var.cluster_name}-alb"`. You'll need to make a similar change to the other `aws_security_group` resource (e.g., give it the name `"${var.cluster_name}-instance"`), the `aws_alb` resource, and the `tag` section of the `aws_autoscaling_group` resource.

You should also update the `terraform_remote_state` data source to use the `db_remote_state_bucket` and `db_remote_state_key` as its `bucket` and `key` parameter, respectively, to ensure you're reading the state file from the right environment:

```
data "terraform_remote_state" "db" {
  backend = "s3"

  config = {
    bucket = var.db_remote_state_bucket
    key    = var.db_remote_state_key
    region = "us-east-2"
  }
}
```

Now, in the staging environment, in *stage/services/webserver-cluster/main.tf*, you can set these new input variables accordingly:

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"

  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-
stores/mysql/terraform.tfstate"
}
```

You should do the same in the production environment in *prod/services/webserver-cluster/main.tf*:

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"

  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-
stores/mysql/terraform.tfstate"
}
```

Note: the production database doesn't actually exist yet. As an exercise, I leave it up to you to figure out how to deploy MySQL in both staging and production.

As you can see, you set input variables for a module using the same syntax as setting input parameters for a resource. The input variables are the API of the module, controlling how it will behave in different environments. This example uses different names in different environments, but you may want to make other parameters configurable, too. For example, in staging, you might want to run a small web server cluster to save money, but in production, you might want to run a larger cluster to handle lots of traffic. To do that, you can add three more input variables to *modules/services/webserver-cluster/variables.tf*:

```
variable "instance_type" {
  description = "The type of EC2 Instances to run (e.g.
t2.micro)"
  type        = string
}

variable "min_size" {
  description = "The minimum number of EC2 Instances in
the ASG"
  type        = number
}

variable "max_size" {
  description = "The maximum number of EC2 Instances in
the ASG"
  type        = number
}
```

Next, update the launch configuration in *modules/services/webserver-cluster/main.tf* to set its `instance_type` parameter to the new `var.instance_type` input variable:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]
```

```
  user_data       =
data.template_file.user_data.rendered
}
```

Similarly, you should update the ASG definition in the same file to set its `min_size` and `max_size` parameters to the new `var.min_size` and `var.max_size` input variables:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration =
aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }
}
```

Now, in the staging environment (*stage/services/webserver-cluster/main.tf*), you can keep the cluster small and inexpensive by setting `instance_type` to `"t2.micro"` and `min_size` and `max_size` to 2:

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"

  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-
stores/mysql/terraform.tfstate"
```

```
  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}
```

On the other hand, in the production environment, you can use a larger `instance_type` with more CPU and memory, such as m4.large (note: this instance type is *not* part of the AWS free tier, so if you're just using this for learning and don't want to be charged, use `"t2.micro"` for the `instance_type`), and you can set `max_size` to 10 to allow the cluster to shrink or grow depending on the load (don't worry, the cluster will launch with two Instances initially):

```
module "webserver_cluster" {
  source = "../../../modules/services/webserver-cluster"

  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-
stores/mysql/terraform.tfstate"

  instance_type = "m4.large"
  min_size      = 2
  max_size      = 10
}
```

Module Locals

Using input variables to define your module's inputs is great, but what if you need a way to define a variable in your module to do some intermediary calculation, or just to keep your code DRY, but you don't want to expose that variable as a configurable input? For example, the load balancer in the webservers module in *modules/services/webserver-*

cluster/main.tf listens on port 80, the default port for HTTP. This port number is currently copy/pasted in multiple places, including the load balancer listener:

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = 80
  protocol          = "HTTP"

  # By default, return a simple 404 page
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

And the load balancer security group:

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  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"]
  }
}
```

The values in the security group, including the “all IPs” CIDR block `0.0.0.0/0`, the “any port” value of 0, and the “any protocol” value of “-1” are also copy/pasted in several places throughout the module. Having these magical values hard-coded in multiple places makes the code harder to read and maintain. You could extract values into input variables, but then users of your module will be able to (accidentally) override these values, which you may not want. Instead of using input variables, you can define these as *local values*:

```
locals {
  http_port    = 80
  any_port     = 0
  any_protocol = "-1"
  tcp_protocol = "tcp"
  all_ips      = ["0.0.0.0/0"]
}
```

Local values allow you to assign a name to any Terraform expression, and to use that name throughout the module. These names are only visible within the module, so they will have no impact on other modules, and you can’t override these values from outside of the module. To read the value of a local, you need to use a *local reference*, which uses the following syntax:

```
local.<NAME>
```

Use this syntax to update your load balancer listener:

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = local.http_port
  protocol          = "HTTP"
```

```
  # By default, return a simple 404 page
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

And all the security groups in the module, including the load balancer security group:

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  ingress {
    from_port   = local.http_port
    to_port     = local.http_port
    protocol    = local.tcp_protocol
    cidr_blocks = local.all_ips
  }

  egress {
    from_port   = local.any_port
    to_port     = local.any_port
    protocol    = local.any_protocol
    cidr_blocks = local.all_ips
  }
}
```

Locals make your code easier to read and maintain, so use them often!

Module Outputs

A powerful feature of Auto Scaling Groups is that you can configure them to increase or decrease the number of servers you have running in response to load. One way to do this is to use an *auto scaling schedule*, which can change the size of the cluster at a scheduled time during the day. For example, if traffic to your cluster is much higher during normal business hours, you can use an auto scaling schedule to increase the number of servers at 9 a.m. and decrease it at 5 p.m.

If you define the auto scaling schedule in the `webserver-cluster` module, it would apply to both staging and production. Since you don't need to do this sort of scaling in your staging environment, for the time being, you can define the auto scaling schedule directly in the production configurations (in Chapter 5, you'll see how to conditionally define resources, which will allow you to move the auto scaling policy into the `webserver-cluster` module).

To define an auto scaling schedule, add the following two `aws_autoscaling_schedule` resources to *prod/services/webserver-cluster/main.tf*:

```
resource "aws_autoscaling_schedule"
"scale_out_during_business_hours" {
  scheduled_action_name = "scale-out-during-business-
hours"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 10
  recurrence            = "0 9 * * *"
}

resource "aws_autoscaling_schedule" "scale_in_at_night"
{
  scheduled_action_name = "scale-in-at-night"
  min_size              = 2
  max_size              = 10
```

```
  desired_capacity       = 2
  recurrence             = "0 17 * * *"
}
```

This code uses one `aws_autoscaling_schedule` resource to increase the number of servers to 10 during the morning hours (the `recurrence` parameter uses cron syntax, so `"0 9 * * *"` means "9 a.m. every day") and a second `aws_autoscaling_schedule` resource to decrease the number of servers at night (`"0 17 * * *"` means "5 p.m. every day"). However, both usages of `aws_autoscaling_schedule` are missing a required parameter, `autoscaling_group_name`, which specifies the name of the ASG. The ASG itself is defined within the `webserver-cluster` module, so how do you access its name? In a general-purpose programming language, functions can return values:

```
def example_function(param1, param2)
  return "Hello, #{param1} #{param2}"
end

# Other places in your code
return_value = example_function("foo", "bar")
```

In Terraform, a module can also return values. Again, this is done using a mechanism you already know: output variables. You can add the ASG name as an output variable in */modules/services/webserver-cluster/outputs.tf* as follows:

```
output "asg_name" {
  value       = aws_autoscaling_group.example.name
  description = "The name of the Auto Scaling Group"
}
```

You can access module output variables the same way as resource output attributes. The syntax is:

```
module.<MODULE_NAME>.<OUTPUT_NAME>
```

For example:

```
module.frontend.asg_name
```

In *prod/services/webserver-cluster/main.tf*, you can use this syntax to set the `autoscaling_group_name` parameter in each of the `aws_autoscaling_schedule` resources:

```
resource "aws_autoscaling_schedule"
"scale_out_during_business_hours" {
  scheduled_action_name = "scale-out-during-business-
hours"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 10
  recurrence            = "0 9 * * *"

  autoscaling_group_name =
module.webserver_cluster.asg_name
}

resource "aws_autoscaling_schedule" "scale_in_at_night"
{
  scheduled_action_name = "scale-in-at-night"
  min_size              = 2
  max_size              = 10
  desired_capacity      = 2
  recurrence            = "0 17 * * *"

  autoscaling_group_name =
module.webserver_cluster.asg_name
}
```

You may want to expose one other output in the `webserver-cluster` module: the DNS name of the ALB, so you know what URL to test when the cluster is deployed. To do that, you again add an output variable in */modules/services/webserver-cluster/outputs.tf*:

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

You can then "pass through" this output in *stage/services/webserver-cluster/outputs.tf* and *prod/services/webserver-cluster/outputs.tf* as follows:

```
output "alb_dns_name" {
  value       = module.webserver_cluster.alb_dns_name
  description = "The domain name of the load balancer"
}
```

Your web server cluster is almost ready to deploy. The only thing left is to take a few gotchas into account.

Module Gotchas

When creating modules, watch out for these gotchas:

- File paths
- Inline blocks

File Paths

In Chapter 3, you moved the User Data script for the web server cluster

into an external file, *user-data.sh*, and used the `file` built-in function to read this file from disk. The catch with the `file` function is that the file path you use has to be relative (since you could run Terraform on many different computers)—but what is it relative to?

By default, Terraform interprets the path relative to the current working directory. That works if you're using the `file` function in a Terraform configuration file that's in the same directory as where you're running `terraform apply` (that is, if you're using the `file` function in the root module), but that won't work when you're using `file` in a module that's defined in a separate folder.

To solve this issue, you can use an expression known as a *path reference*, which is of the form `path.<TYPE>`. Terraform supports the following types of path references:

`path.module`

: Returns the filesystem path of the module where the expression is placed.

`path.root`

: Returns the filesystem path of the root module of the configuration.

`path.cwd`

: Returns the filesystem path of the current working directory. In normal use of Terraform this is the same as `path.root`, but some advanced uses of Terraform run it from a directory other than the root module directory, causing these paths to be different.

For the User Data script, you need a path relative to the module itself, so you should use `path.module` in the `template_file` data source in

modules/services/webserver-cluster/main.tf:

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  =
data.terraform_remote_state.db.outputs.address
    db_port     =
data.terraform_remote_state.db.outputs.port
  }
}
```

Inline Blocks

The configuration for some Terraform resources can be defined either as inline blocks or as separate resources. When creating a module, you should always prefer using a separate resource.

For example, the `aws_security_group` resource allows you to define ingress and egress rules via inline blocks, as you saw in the `webserver-cluster` module (*modules/services/webserver-cluster/main.tf*):

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  ingress {
    from_port   = local.http_port
    to_port     = local.http_port
    protocol    = local.tcp_protocol
    cidr_blocks = local.all_ips
  }

  egress {
    from_port   = local.any_port
```

```
    to_port     = local.any_port
    protocol    = local.any_protocol
    cidr_blocks = local.all_ips
  }
}
```

You should change this module to define the exact same ingress and egress rules by using separate `aws_security_group_rule` resources (make sure to do this for both security groups in the module):

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"
}

resource "aws_security_group_rule" "allow_http_inbound"
{
  type              = "ingress"
  security_group_id = aws_security_group.alb.id

  from_port   = local.http_port
  to_port     = local.http_port
  protocol    = local.tcp_protocol
  cidr_blocks = local.all_ips
}

resource "aws_security_group_rule" "allow_all_outbound"
{
  type              = "egress"
  security_group_id = aws_security_group.alb.id

  from_port   = local.any_port
  to_port     = local.any_port
  protocol    = local.any_protocol
  cidr_blocks = local.all_ips
}
```

If you try to use a mix of *both* inline blocks and separate resources, you

will get errors where routing rules conflict and overwrite each other. Therefore, you must use one or the other. Because of this limitation, when creating a module, you should always try to use a separate resource instead of the inline block. Otherwise, your module will be less flexible and configurable.

For example, if all the ingress and egress rules within the `webserver-cluster` module are defined as separate `aws_security_group_rule` resources, you can make the module flexible enough to allow users to add custom rules from outside of the module. To do that, you export the ID of the `aws_security_group` as an output variable in *modules/services/webserver-cluster/outputs.tf*:

```
output "alb_security_group_id" {
  value       = aws_security_group.alb.id
  description = "The ID of the Security Group attached
to the load balancer"
}
```

Now, imagine that in the staging environment, you needed to expose an extra port just for testing. This is now easy to do by adding an `aws_security_group_rule` resource to *stage/services/webserver-cluster/main.tf*:

```
resource "aws_security_group_rule"
"allow_testing_inbound" {
  type              = "ingress"
  security_group_id =
module.webserver_cluster.alb_security_group_id

  from_port   = 12345
  to_port     = 12345
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
```

```
}
```

Had you defined even a single ingress or egress rule as an inline block, this code would not work. Note that this same type of problem affects a number of Terraform resources, such as:

- `aws_security_group` and `aws_security_group_rule`
- `aws_route_table` and `aws_route`
- `aws_network_acl` and `aws_network_acl_rule`

At this point, you are finally ready to deploy your web server cluster in both staging and production. Run `terraform apply` as usual and enjoy using two separate copies of your infrastructure.

NETWORK ISOLATION

The examples in this chapter create two environments that are isolated in your Terraform code, and isolated in terms of having separate load balancers, servers, and databases, but they are not isolated at the network level. To keep all the examples in this book simple, all the resources deploy into the same Virtual Private Cloud (VPC). That means a server in the staging environment can talk to a server in the production environment and vice versa.

In real-world usage, running both environments in one VPC opens you up to two risks. First, a mistake in one environment could affect the other. For example, if you're making changes in staging and accidentally mess up the configuration of the route tables, all the routing in production may be affected too. Second, if an attacker gets access to one environment, they also have access to the other. If you're making rapid changes in staging and accidentally leave a port exposed, any hacker that broke in would not only have access to your staging data, but also your production data.

Therefore, outside of simple examples and experiments, you should run each environment in a separate VPC. In fact, to be extra sure, you may even run each environment in totally separate AWS accounts!

Module Versioning

If both your staging and production environment are pointing to the same module folder, then as soon as you make a change in that folder, it will affect both environments on the very next deployment. This sort of coupling makes it harder to test a change in staging without any chance of affecting production. A better approach is to create *versioned modules* so that you can use one version in staging (e.g., v0.0.2) and a different version in production (e.g., v0.0.1), as shown in Figure 4-7.

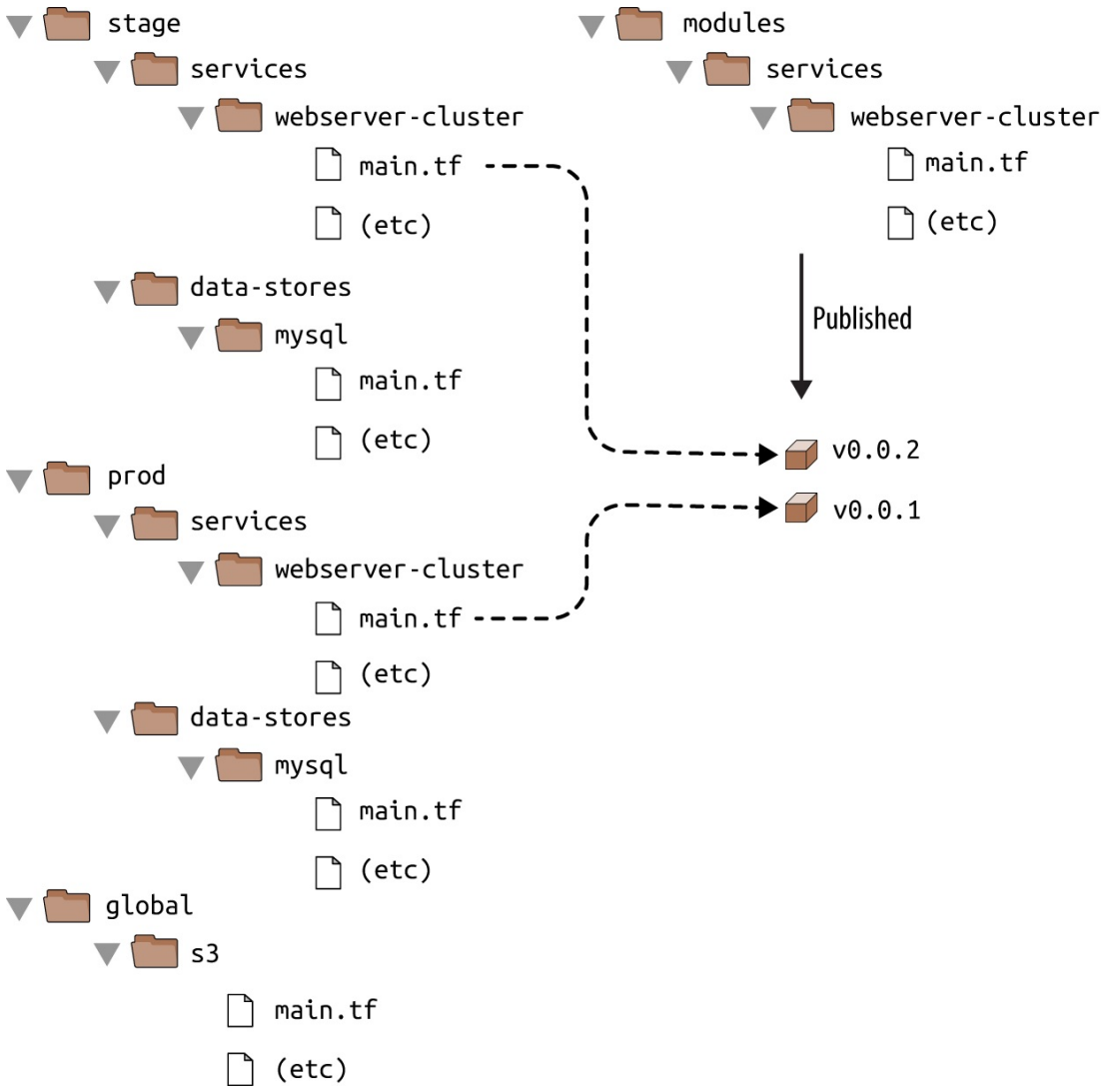

Figure 4-7. Using different versions of a module in different environments

In all the module examples you've seen so far, whenever you used a module, you set the `source` parameter of the module to a local file path. In addition to file paths, Terraform supports other types of module sources, such as Git URLs, Mercurial URLs, and arbitrary HTTP URLs.[1] The easiest way to create a versioned module is to put the code for the module in a separate Git repository and to set the `source` parameter to that repository's URL. That means your Terraform code will be spread out across (at least) two repositories:

modules

: This repo defines reusable modules. Think of each module as a "blueprint" that defines a specific part of your infrastructure.

live

: This repo defines the live infrastructure you're running in each environment (stage, prod, mgmt, etc). Think of this as the "houses" you built from the "blueprints" in the *modules* repo.

The updated folder structure for your Terraform code will now look something like Figure 4-8.

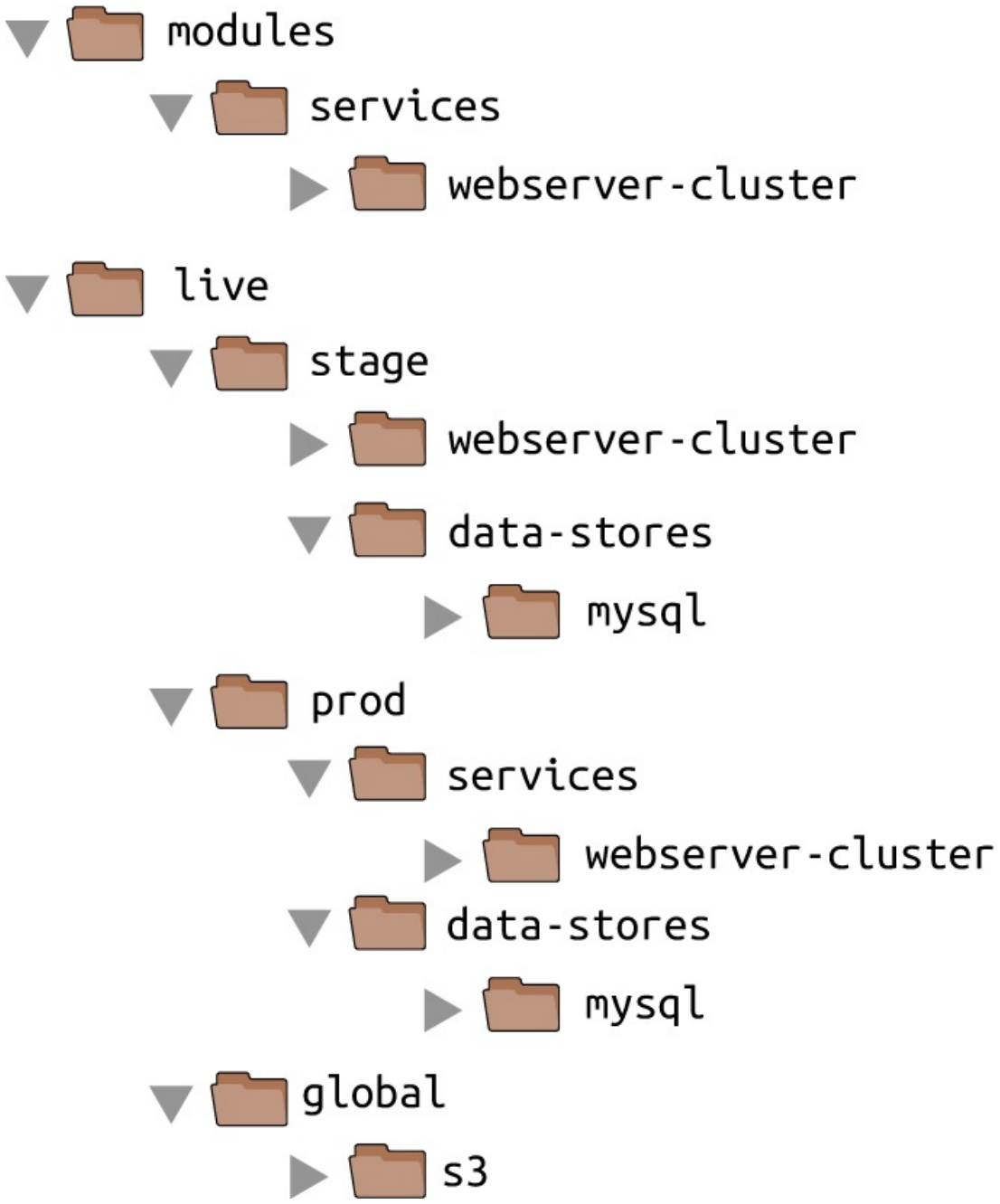

Figure 4-8. File layout with multiple repositories

To set up this folder structure, you'll first need to move the *stage*, *prod*, and *global* folders into a folder called *live*. Next, configure the *live* and *modules* folders as separate git repositories. Here is an example of how to do that for the *modules* folder:

```
$ cd modules
```

```
$ git init
$ git add .
$ git commit -m "Initial commit of modules repo"
$ git remote add origin "(URL OF REMOTE GIT REPOSITORY)"
$ git push origin master
```

You can also add a tag to the *modules* repo to use as a version number. If you're using GitHub, you can use the GitHub UI to create a release, which will create a tag under the hood. If you're not using GitHub, you can use the Git CLI:

```
$ git tag -a "v0.0.1" -m "First release of webserver-
cluster module"
$ git push --follow-tags
```

Now you can use this versioned module in both staging and production by specifying a Git URL in the `source` parameter. Here is what that would look like in *live/stage/services/webserver-cluster/main.tf* if your `modules` repo was in the GitHub repo *github.com/foo/modules* (note that the double-slash in the Git URL is required):

```
module "webserver_cluster" {
  source =
"git::git@github.com:foo/modules.git//webserver-cluster?
ref=v0.0.1"

  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-
stores/mysql/terraform.tfstate"

  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}
```

If you want to try out versioned modules without messing with Git repos, you can use a module from the code examples GitHub repo for this book (I had to break up the URL to make it fit in the book, but it should all be on one line):

```
source = "git@github.com:brikis98/terraform-up-and-
running-code.git//
  code/terraform/04-terraform-module/module-
example/modules/
  services/webserver-cluster?ref=v0.0.2"
```

The `ref` parameter allows you to specify a specific Git commit via its sha1 hash, a branch name, or, as in this example, a specific Git tag. I generally recommend using Git tags as version numbers for modules. Branch names are not stable, as you always get the latest commit on a branch, which may change every time you run the `get` command, and the sha1 hashes are not very human friendly. Git tags are as stable as a commit (in fact, a tag is just a pointer to a commit) but they allow you to use a friendly, readable name.

A particularly useful naming scheme for tags is *semantic versioning*. This is a versioning scheme of the format `MAJOR.MINOR.PATCH` (e.g., `1.0.4`) with specific rules on when you should increment each part of the version number. In particular, you should increment the…

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backward-compatible manner, and
- PATCH version when you make backward-compatible bug fixes.

Semantic versioning gives you a way to communicate to users of your

module what kind of changes you've made and the implications of upgrading.

Since you've updated your Terraform code to use a versioned module URL, you need to tell Terraform to download the module code by re-running `terraform init`:

```
$ terraform init
Initializing modules...
Downloading git@github.com:brikis98/terraform-up-and-
running-code.git?ref=v0.0.2 for webserver_cluster...
- webserver_cluster in
.terraform/modules/webserver_cluster/code/terraform/04-
terraform-module/module-
example/modules/services/webserver-cluster

(...)
```

This time, you can see that Terraform downloads the module code from Git rather than your local filesystem. Once the module code has been downloaded, you can run the `apply` command as usual.

PRIVATE GIT REPOS

If your Terraform module is in a private Git repository, to use that repo as a module `source`, you will need to give Terraform a way to authenticate to that Git repository. I recommend using SSH auth so that you don't have to hard-code the credentials for your repo in the code itself. With SSH auth, each developer can create an SSH key, associate it with their Git user, and add it to `ssh-agent`, and Terraform will automatically use that key for authentication if you use an SSH `source` URL.[2]

The `source` URL should be of the form:

```
source = "git::git@github.com:
<OWNER>/<REPO>.git//<PATH>?ref=<VERSION>"
```

For example:

```
source = "git::git@github.com:gruntwork-io/terraform-google-gke.git//modules/gke-cluster?ref=v0.1.2"
```

To check that you've formatted the URL correctly, try to `git clone` the base URL from your terminal:

```
$ git clone git@github.com:<OWNER>/<REPO>.git
```

If that command succeeds, Terraform should be able to use the private repo too.

Now, imagine you made some changes to the `webserver-cluster` module and you wanted to test them out in staging. First, you'd commit those changes to the *modules* repo:

```
$ cd modules
$ git add .
$ git commit -m "Made some changes to webserver-cluster"
$ git push origin master
```

Next, you would create a new tag in the *modules* repo:

```
$ git tag -a "v0.0.2" -m "Second release of webserver-cluster"
$ git push --follow-tags
```

And now you can update *just* the source URL used in the staging environment (*live/stage/services/webserver-cluster/main.tf*) to use this new version:

```
module "webserver_cluster" {
  source =
"git::git@github.com:foo/modules.git//webserver-cluster?
ref=v0.0.2"

  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-
stores/mysql/terraform.tfstate"

  instance_type = "t2.micro"
  min_size      = 2
  max_size      = 2
}
```

In production (*live/prod/services/webserver-cluster/main.tf*), you can happily continue to run v0.0.1 unchanged:

```
module "webserver_cluster" {
  source =
"git::git@github.com:foo/modules.git//webserver-cluster?
ref=v0.0.1"

  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-
stores/mysql/terraform.tfstate"

  instance_type = "m4.large"
  min_size      = 2
  max_size      = 10
}
```

Once v0.0.2 has been thoroughly tested and proven in staging, you can then update production, too. But if there turns out to be a bug in v0.0.2, no big deal, as it has no effect on the real users of your production environment. Fix the bug, release a new version, and repeat the whole process again until you have something stable enough for production.

DEVELOPING MODULES

Versioned modules are great when you're deploying to a shared environment (e.g., staging or production), but when you're just testing on your own computer, you'll want to use local file paths. This allows you to iterate faster, as you'll be able to make a change in the module folders and rerun the `plan` or `apply` command in the live folders immediately, rather than having to commit your code and publish a new version each time.

Since the goal of this book is to help you learn and experiment with Terraform as quickly as possible, the rest of the code examples will use local file paths for modules.

Conclusion

By defining infrastructure as code in modules, you can apply a variety of software engineering best practices to your infrastructure. You can validate each change to a module through code reviews and automated tests; you can create semantically versioned releases of each module; and you can safely try out different versions of a module in different environments and roll back to previous versions if you hit a problem.

All of this can dramatically increase your ability to build infrastructure quickly and reliably, as developers will be able to reuse entire pieces of proven, tested, documented infrastructure. For example, you could create a canonical module that defines how to deploy a single microservice—including how to run a cluster, how to scale the cluster in response to load, and how to distribute traffic requests across the cluster—and each team could use this module to manage their own microservices with just a few lines of code.

To make such a module work for multiple teams, the Terraform code in that module must be flexible and configurable. For example, one team may want to use your module to deploy a single instance of their microservice with no load balancer while another may want a dozen instances of their microservice with a load balancer to distribute traffic between those instances. How do you do conditional statements in Terraform? Is there a way to do a for-loop? Is there a way to use Terraform to roll out changes to this microservice without downtime? These advanced aspects of Terraform syntax are the topic of Chapter 5.

---

1 For the full details on source URLs, see *https://www.terraform.io/docs/modules/sources.html*.

2 See *https://help.github.com/en/articles/connecting-to-github-with-ssh* for a nice guide on working with SSH keys.

Chapter 5. Terraform Tips and Tricks: Loops, If-Statements, Deployment, and Gotchas

Terraform is a declarative language. As discussed in Chapter 1, infrastructure as code in a declarative language tends to provide a more accurate view of what's actually deployed than a procedural language, so it's easier to reason about and makes it easier to keep the codebase small. However, certain types of tasks are more difficult in a declarative language.

For example, since declarative languages typically don't have for-loops, how do you repeat a piece of logic—such as creating multiple similar resources—without copy and paste? And if the declarative language doesn't support if-statements, how can you conditionally configure resources, such as creating a Terraform module that can create certain resources for some users of that module but not for others? Finally, how do you express an inherently procedural idea, such as a zero-downtime deployment, in a declarative language?

Fortunately, Terraform provides a few primitives—namely, a meta-parameter called `count`, `for_each` and `for` expressions, a lifecycle block called `create_before_destroy`, a ternary operator, plus a large number of functions—that allow you to do certain types of loops, if-statements, and zero-downtime deployments. Here are the topics I'll cover in this chapter:

- Loops
- Conditionals
- Zero-downtime deployment
- Terraform gotchas

> **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*.

Loops

Terraform offers several different looping constructs, each intended to be used in a slightly different condition:

1. **`count` parameter**: loop over resources.
2. **`for_each` expressions**: loop over inline blocks within a resource.
3. **`for` expressions**: loop over lists and maps.
4. **`for` string directive**: loop over lists and maps within a string.

Let's go through these one at a time.

Loops with the count parameter

In Chapter 2, you created an IAM user by clicking around the AWS console. Now that you have this user, you can create and manage all future IAM users with Terraform. Consider the following Terraform code, which

should live in *live/global/iam/main.tf*:

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

resource "aws_iam_user" "example" {
  name = "neo"
}
```

This code uses the `aws_iam_user` resource to create a single new IAM user. What if you wanted to create three IAM users? In a general-purpose programming language, you'd probably use a for-loop:

```
# This is just pseudo code. It won't actually work in
Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = "neo"
  }
}
```

Terraform does not have for-loops or other traditional procedural logic built into the language, so this syntax will not work. However, every Terraform resource has a meta-parameter you can use called `count`. This parameter defines how many copies of the resource to create. Therefore, you can create three IAM users as follows:

```
resource "aws_iam_user" "example" {
  count = 3
  name  = "neo"
}
```

One problem with this code is that all three IAM users would have the same name, which would cause an error, since usernames must be unique.

If you had access to a standard for-loop, you might use the index in the for loop, `i`, to give each user a unique name:

```
# This is just pseudo code. It won't actually work in
Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = "neo.${i}"
  }
}
```

To accomplish the same thing in Terraform, you can use `count.index` to get the index of each "iteration" in the "loop":

```
resource "aws_iam_user" "example" {
  count = 3
  name  = "neo.${count.index}"
}
```

If you run the `plan` command on the preceding code, you will see that Terraform wants to create three IAM users, each with a different name ("neo.0", "neo.1", "neo.2"):

```
Terraform will perform the following actions:

  # aws_iam_user.example[0] will be created
  + resource "aws_iam_user" "example" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "neo.0"
      + path          = "/"
      + unique_id     = (known after apply)
    }

  # aws_iam_user.example[1] will be created
  + resource "aws_iam_user" "example" {
      + arn           = (known after apply)
```

```
      + force_destroy = false
      + id            = (known after apply)
      + name          = "neo.1"
      + path          = "/"
      + unique_id     = (known after apply)
    }

  # aws_iam_user.example[2] will be created
  + resource "aws_iam_user" "example" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "neo.2"
      + path          = "/"
      + unique_id     = (known after apply)
    }

Plan: 3 to add, 0 to change, 0 to destroy.
```

Of course, a username like "neo.0" isn't particularly usable. If you combine `count.index` with some built-in functions from Terraform, you can customize each "iteration" of the "loop" even more.

For example, you could define all of the IAM usernames you want in an input variable in *live/global/iam/variables.tf*:

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

If you were using a general-purpose programming language with loops and arrays, you would configure each IAM user to use a different name by looking up index `i` in the array `var.user_names`:

```
# This is just pseudo code. It won't actually work in
```

```
Terraform.
for (i = 0; i < 3; i++) {
  resource "aws_iam_user" "example" {
    name = vars.user_names[i]
  }
}
```

In Terraform, you can accomplish the same thing by using `count`, a new expression for working with lists, and the built-in function `length`:

```
list[<INDEX>]
length(list)
```

The array look up syntax is similar to most programming languages: it looks up `INDEX` in the given `list`. The `length` function returns the number of items in the `list` (it also works with strings and maps). Putting these together, you get:

```
resource "aws_iam_user" "example" {
  count = length(var.user_names)
  name  = var.user_names[count.index]
}
```

Now when you run the `plan` command, you'll see that Terraform wants to create three IAM users, each with a unique name:

```
Terraform will perform the following actions:

  # aws_iam_user.example[0] will be created
  + resource "aws_iam_user" "example" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "neo"
      + path          = "/"
```

```
      + unique_id     = (known after apply)
    }

  # aws_iam_user.example[1] will be created
  + resource "aws_iam_user" "example" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "trinity"
      + path          = "/"
      + unique_id     = (known after apply)
    }

  # aws_iam_user.example[2] will be created
  + resource "aws_iam_user" "example" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "morpheus"
      + path          = "/"
      + unique_id     = (known after apply)
    }

Plan: 3 to add, 0 to change, 0 to destroy.
```

Note that once you've used `count` on a resource, it becomes a list of resources, rather than just one resource. Since `aws_iam_user.example` is now a list of IAM users, instead of using the standard syntax to read an attribute from that resource (`<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>`), you have to specify which IAM user you're interested in by specifying its index in the list using the same array lookup syntax:

```
<PROVIDER>_<TYPE>.<NAME>[INDEX].ATTRIBUTE
```

For example, if you wanted to provide the Amazon Resource Name (ARN) of one of the IAM users as an output variable, you would need to

do the following:

```
output "neo_arn" {
  value       = aws_iam_user.example[0].arn
  description = "The ARN for user Neo"
}
```

If you want the ARNs of *all* the IAM users, you need to use a *splat* expression, "*", instead of the index:

```
output "all_arns" {
  value       = aws_iam_user.example[*].arn
  description = "The ARNs for all users"
}
```

When you run the `apply` command, the `neo_arn` output will contain just the ARN for Neo while the `all_arns` output will contain the list of all ARNs:

```
$ terraform apply

(...)

Apply complete! Resources: 3 added, 0 changed, 0
destroyed.

Outputs:

all_arns = [
  "arn:aws:iam::123456789012:user/neo",
  "arn:aws:iam::123456789012:user/trinity",
  "arn:aws:iam::123456789012:user/morpheus",
]
neo_arn = arn:aws:iam::123456789012:user/neo
```

Note that since the splat expression returns a list, you can combine it with

other expressions and built-in functions. For example, let's say you wanted to give each of these IAM users read-only access to EC2. You may remember from Chapter 2 that by default, new IAM users have no permissions whatsoever, and that to grant permissions, you can attach IAM policies to those IAM users. An IAM policy is a JSON document:

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["ec2:Describe*"],
      "Resource": ["*"]
    }
  ]
}
```

An IAM policy consists of one or more *statements*, each of which specifies an *effect* (either "Allow" or "Deny"), on one or more *actions* (e.g., `"ec2:Describe*"` allows all API calls to EC2 that start with the name `"Describe"`), on one or more *resources* (e.g., `"*"` means "all resources"). Although you can define IAM policies using a JSON string, Terraform also provides a handy data source called the `aws_iam_policy_document` that gives you a more concise way to define the same IAM policy:

```
data "aws_iam_policy_document" "ec2_read_only" {
  statement {
    effect    = "Allow"
    actions   = ["ec2:Describe*"]
    resources = ["*"]
  }
}
```

To create a new managed IAM policy from this document, you need to use the `aws_iam_policy` resource and set its `policy` parameter to the `json` output attribute of the `aws_iam_policy_document` you just created:

```
resource "aws_iam_policy" "ec2_read_only" {
  name   = "ec2-read-only"
  policy =
data.aws_iam_policy_document.ec2_read_only.json
}
```

Finally, to attach the IAM policy to your new IAM users, you use the `aws_iam_user_policy_attachment` resource:

```
resource "aws_iam_user_policy_attachment" "ec2_access" {
  count      = length(var.user_names)
  user       = element(aws_iam_user.example[*].name,
count.index)
  policy_arn = aws_iam_policy.ec2_read_only.arn
}
```

This code uses the `count` parameter to "loop" over each of your IAM users and a built-in function you haven't seen before called `element` to select each user's name from the list of names you get back from the splat expression (`aws_iam_user.example[*].name`). The `element` function has the following signature:

```
element(list, <INDEX>)
```

This function returns the item at `INDEX` in the given `list`, similar to an array lookup. In fact, the code to attach the IAM policy could've also been written as follows:

```
resource "aws_iam_user_policy_attachment" "ec2_access" {
  count      = length(var.user_names)
  user       = aws_iam_user.example[count.index].name
  policy_arn = aws_iam_policy.ec2_read_only.arn
}
```

The difference between `element` and array lookups is what happens if you try to access an index that is out of bounds. For example, if you tried look up index 4 in an array with only 3 items, the array lookup would give you an error, whereas the `element` function will loop around using a standard mod algorithm, returning the item at index 1.

Loops with for_each expressions

The `count` parameter is useful if you want to "loop" over an an entire resource, but how do you do "loops" for inline blocks within a resource? For example, how are tags are set in the `aws_autoscaling_group` resource:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration =
aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }
}
```

Each tag must be specified as an *inline block*—that is, an argument you set within a resource of the format:

```
resource "xxx" "yyy" {
  <NAME> {
    [CONFIG...]
  }
}
```

Where `NAME` is the name of the argument (e.g., `tag`) and `CONFIG` consists of one or more arguments that are specific to that argument (e.g., `key` and `value`). The ASG in the `webserver-cluster` module currently hard-codes a single tag, but you may want to allow users to pass in custom tags. For example, you could add a new map input variable called `custom_tags` in *modules/services/webserver-cluster/variables.tf*:

```
variable "custom_tags" {
  description = "Custom tags to set on the Instances in
the ASG"
  type        = map(string)
  default     = {}
}
```

And set some custom tags in the production environment, in *live/prod/services/webserver-cluster/main.tf*, as follows:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-
cluster"

  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-
stores/mysql/terraform.tfstate"

  instance_type        = "m4.large"
```

```
  min_size             = 2
  max_size             = 10
  enable_autoscaling   = true

  custom_tags = {
    Owner      = "team-foo"
    DeployedBy = "terraform"
  }
}
```

The code above sets a couple useful tags: the `Owner` tag to specifies which team owns this ASG and the `DeployedBy` tag specifies that this infrastructure was deployed using Terraform (indicating this infrastructure shouldn't be modified manually, as discussed in "Valid Plans Can Fail"). It's typically a good idea to come up with a tagging standard for your team and create Terraform modules that enforce this standard as code.

Now that the tags are set, how do you actually set them on the `aws_autoscaling_group` resource? What you need is to do a for loop over `var.custom_tags`, similar to the following pseudo code:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration =
aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }
```

```
  # This is just pseudo code. It won't actually work in
Terraform.
  for (tag in var.custom_tags) {
    tag {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
}
```

The psuedo code above won't work, but Terraform does support something similar in a *for_each expression*, which has the following syntax:

```
dynamic "<VAR_NAME>" {
  for_each = <COLLECTION>

  content {
    [CONFIG...]
  }
}
```

Where `VAR_NAME` is the name to use for the variable that will be used to store the value each "iteration" (e.g., `tag`), `COLLECTION` is a list or map to iterate over (e.g., `var.custom_tags`), and the `content` block is what to generate from each iteration. You can use `<VAR_NAME>.key` and `<VAR_NAME>.value` within the `content` block to access the key and value, respectively, of the current item in the `COLLECTION`. Note thatn when you're using `for_each` with a list, the `key` will be the index and the `value` will be the item in the list at that index, and when using `for_each` with a map, the `key` and `value` will be one of the key-value pairs in the map.

Putting this all together, here is how you can dynamically generate `tag` blocks using `for_each` in the `aws_autoscaling_group` resource:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration =
aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }

  dynamic "tag" {
    for_each = var.custom_tags

    content {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
}
```

If you run `terraform apply` now, you should see a plan that looks something like this (the log output below is truncated for readability):

```
$ terraform apply

Terraform will perform the following actions:

  # aws_autoscaling_group.example will be updated in-
place
```

```
  ~ resource "aws_autoscaling_group" "example" {
        (...)

        tag {
            key                 = "Name"
            propagate_at_launch = true
            value               = "webservers-prod"
        }
      + tag {
          + key                 = "Owner"
          + propagate_at_launch = true
          + value               = "team-foo"
        }
      + tag {
          + key                 = "DeployedBy"
          + propagate_at_launch = true
          + value               = "terraform"
        }
    }

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:
```

Enter “yes” to deploy the changes and you should see your new tags show up in the EC2 web console, as shown in

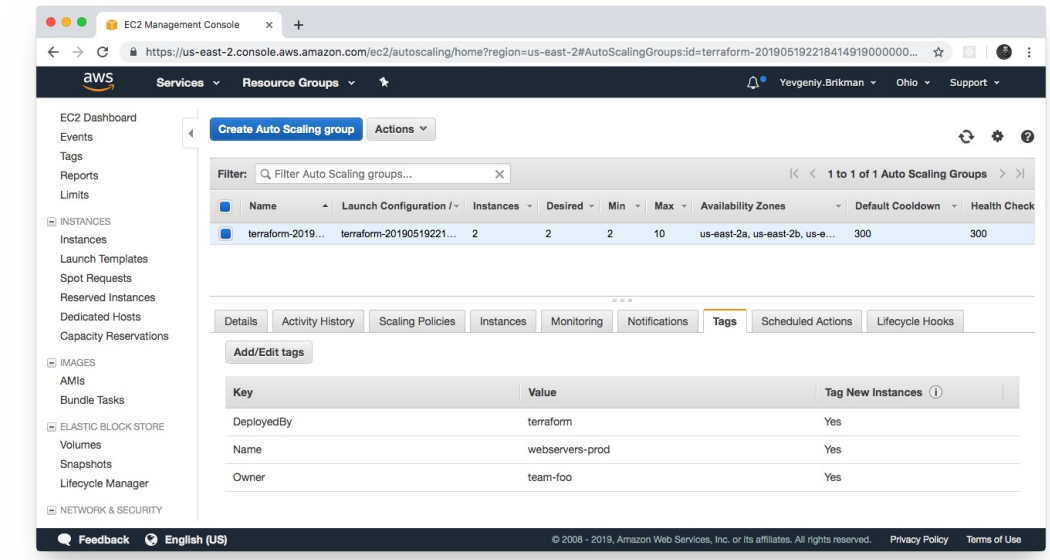

Figure 5-1. Dynamic Auto Scaling Group tags

Loops with for expressions

You've now seen how to loop over resources and inline blocks, but what if you need a loop to generate a single value? Let's take a brief aside to look at some examples unrelated to the web server cluster. Imagine you wrote some Terraform code that took in a list of names:

```
variable "names" {
  description = "A list of names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}
```

How could you convert all of these names to upper case? In a general purpose programming language, such as Python, you could write the following for-loop:

```
names = ["neo", "trinity", "morpheus"]
```

```
upper_case_names = []
for name in names:
    upper_case_names.append(name.upper())

print upper_case_names

# Prints out: ['NEO', 'TRINITY', 'MORPHEUS']
```

Python offers another way to write the exact same code in one line using a syntax known as a *list comprehension*:

```
names = ["neo", "trinity", "morpheus"]

upper_case_names = [name.upper() for name in names]

print upper_case_names

# Prints out: ['NEO', 'TRINITY', 'MORPHEUS']
```

Python also allows you to filter the resulting list by specifying a condition:

```
names = ["neo", "trinity", "morpheus"]

short_upper_case_names = [name.upper() for name in names
if len(name) < 5]

print short_upper_case_names

# Prints out: ['NEO']
```

Terraform offers very similar functionality in the form of a *for expression*. The basic syntax of a for expression is:

```
[for <ITEM> in <LIST> : <OUTPUT>]
```

Where `LIST` is a list to loop over, `ITEM` is the local variable name to assign to each item in `LIST`, and `OUTPUT` is an expression that transforms `ITEM` in some way. For example, here is the Terraform code to convert the list of names in `var.names` to upper case:

```
variable "names" {
  description = "A list of names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}

output "upper_names" {
  value = [for name in var.names : upper(name)]
}
```

If you run `terraform apply` on this code, you get the following output:

```
$ terraform apply

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

Outputs:

upper_names = [
  "NEO",
  "TRINITY",
  "MORPHEUS",
]
```

Just as with Python's list comprehensions, you can filter the resulting list by specifying a condition:

```
variable "names" {
  description = "A list of names"
```

```
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}

output "short_upper_names" {
  value = [for name in var.names : upper(name) if
length(name) < 5]
}
```

Running `terraform apply` on this code gives you:

```
short_upper_names = [
  "NEO",
]
```

Terraform's for expressions also allow you to loop over a map using the following syntax:

```
[for <KEY>, <VALUE> in <MAP> : <OUTPUT>]
```

Where MAP is a map to loop over, KEY and VALUE are the local variable names to assign to each key-value pair in MAP, and OUTPUT is an expression that transforms KEY and VALUE in some way. Here's an example:

```
variable "hero_thousand_faces" {
  description = "map"
  type        = map(string)
  default     = {
    neo      = "hero"
    trinity  = "love interest"
    morpheus = "mentor"
  }
}

output "bios" {
```

```
  value = [for name, role in var.hero_thousand_faces :
"${name} is the ${role}"]
}
```

When you run `terraform apply` on this code, you get:

```
map_example = [
  "morpheus is the mentor",
  "neo is the hero",
  "trinity is the love interest",
]
```

You can also use for expressions to output a map rather than list using the following syntax:

```
# For looping over lists
{for <ITEM> in <MAP> : <OUTPUT_KEY> => <OUTPUT_VALUE>}

# For looping over maps
{for <KEY>, <VALUE> in <MAP> : <OUTPUT_KEY> =>
<OUTPUT_VALUE>}
```

The only differences are that (a) you wrap the expression in curly braces rather than square brackets and (b) rather than outputting a single value each iteration, you output a key and value, separated by an arrow. For example, here is how you can transform map to make all the keys and values upper case:

```
variable "hero_thousand_faces" {
  description = "map"
  type        = map(string)
  default     = {
    neo      = "hero"
    trinity  = "love interest"
    morpheus = "mentor"
  }
```

```
}

output "upper_roles" {
  value = {for name, role in var.hero_thousand_faces :
upper(name) => upper(role)}
}
```

The output from running this code will be:

```
upper_roles = {
  "MORPHEUS" = "MENTOR"
  "NEO" = "HERO"
  "TRINITY" = "LOVE INTEREST"
}
```

Loops with the for string directive

Earlier in the book, you learned about string interpolations, which allow you to reference Terraform code within strings:

```
"Hello, ${var.name}"
```

String directives allow you to use control statements, such as for expressions and if-statements, within strings using a similar syntax to string interpolations, but instead of a dollar sign and curly braces (${…}), you use a percent sign and curly braces (%{…}).

Terraform supports two types of string directives: for loops and conditionals. In this section, we'll go over for loops; we'll come back to conditionals later in the chapter. The for string directive uses the following syntax:

```
%{ for <ITEM> in <COLLECTION> }<BODY>%{ endfor }
```

Where `COLLECTION` is a list or map to loop over, `ITEM` is the local variable name to assign to each item in `COLLECTION`, and `BODY` is what to render each iteration (which can reference `ITEM`). Here's an example:

```
variable "names" {
  description = "Names to render"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}

output "for_directive" {
  value = <<EOF
%{ for name in var.names }
  ${name}
%{ endfor }
EOF
}
```

When you run `terraform apply`,you get the output:

```
$ terraform apply

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

Outputs:

for_directive =
  neo

  trinity

  morpheus
```

Note all the extra newlines. You can use a *strip marker* (~) in your string directive to consume all of the whitespace (spaces and newlines) either before the string directive (if the marker appears at the beginning of the

string directive) or after (if the marker appears at the end of the string directive):

```
output "for_directive_strip_marker" {
  value = <<EOF
%{~ for name in var.names }
  ${name}
%{~ endfor }
EOF
}
```

This updated version gives you the following output:

```
for_directive_strip_marker =
  neo
  trinity
  morpheus
```

Conditionals

Just as Terraform offers several different ways to do loops, there are also several different ways to do conditionals, each intended to be used in a slightly different condition:

1. **`count` parameter**: conditional resources.
2. **`for_each` and `for` expressions**: conditional inline blocks within a resource.
3. **`if` string directive**: conditionals within a string.

Conditionals with the count parameter

The `count` parameter you saw earlier lets you do a basic loop. If you're clever, you can use the same mechanism to do a basic conditional. Let's

start by looking at if-statements in the next section and then move on to if-else statements in the section after.

IF-STATEMENTS WITH THE COUNT PARAMETER

In Chapter 4, you created a Terraform module that could be used as "blueprint" for deploying web server clusters. The module created an Auto Scaling Group (ASG), Application Load Balancer (ALB), security groups, and a number of other resources. One thing the module did *not* create was the auto scaling schedule. Since you only want to scale the cluster out in production, you defined the `aws_autoscaling_schedule` resources directly in the production configurations under *live/prod/services/webserver-cluster/main.tf*. Is there a way you could define the `aws_autoscaling_schedule` resources in the `webserver-cluster` module and conditionally create them for some users of the module and not create them for others?

Let's give it a shot. The first step is to add a boolean input variable in *modules/services/webserver-cluster/variables.tf* that can be used to specify whether the module should enable auto scaling:

```
variable "enable_autoscaling" {
  description = "If set to true, enable auto scaling"
  type        = bool
}
```

Now, if you had a general-purpose programming language, you could use this input variable in an if-statement:

```
# This is just pseudo code. It won't actually work in
Terraform.
if var.enable_autoscaling {
  resource "aws_autoscaling_schedule"
```

```
"scale_out_during_business_hours" {
    scheduled_action_name  = "scale-out-during-business-
hours"
    min_size               = 2
    max_size               = 10
    desired_capacity       = 10
    recurrence             = "0 9 * * *"
    autoscaling_group_name =
aws_autoscaling_group.example.name
  }

  resource "aws_autoscaling_schedule"
"scale_in_at_night" {
    scheduled_action_name  = "scale-in-at-night"
    min_size               = 2
    max_size               = 10
    desired_capacity       = 2
    recurrence             = "0 17 * * *"
    autoscaling_group_name =
aws_autoscaling_group.example.name
  }
}
```

Terraform doesn't support if-statements, so this code won't work. However, you can accomplish the same thing by using the `count` parameter and taking advantage of two properties:

1. If you set `count` to 1 on a resource, you get one copy of that resource; if you set `count` to 0, that resource is not created at all.
2. Terraform supports *conditional expressions* of the format `<CONDITION> ? <TRUE_VAL> : <FALSE_VAL>`. This *ternary syntax*, which may be familiar to you from other programming languages, will evaluate the boolean logic in `CONDITION`, and if the result is `true`, it will return `TRUE_VAL`, and if the result is `false`, it'll return `FALSE_VAL`.

Putting these two ideas together, you can update the `webserver-`

`cluster` module as follows:

```
resource "aws_autoscaling_schedule"
"scale_out_during_business_hours" {
  count = var.enable_autoscaling ? 1 : 0

  scheduled_action_name  = "${var.cluster_name}-scale-
out-during-business-hours"
  min_size               = 2
  max_size               = 10
  desired_capacity       = 10
  recurrence             = "0 9 * * *"
  autoscaling_group_name =
aws_autoscaling_group.example.name
}

resource "aws_autoscaling_schedule" "scale_in_at_night"
{
  count = var.enable_autoscaling ? 1 : 0

  scheduled_action_name  = "${var.cluster_name}-scale-
in-at-night"
  min_size               = 2
  max_size               = 10
  desired_capacity       = 2
  recurrence             = "0 17 * * *"
  autoscaling_group_name =
aws_autoscaling_group.example.name
}
```

If `var.enable_autoscaling` is `true`, the `count` parameter for each of the `aws_autoscaling_schedule` resources will be set to 1, so one of each will be created. If `var.enable_autoscaling` is `false`, the `count` parameter for each of the `aws_autoscaling_schedule` resources will be set to 0, so neither one will be created. This is exactly the conditional logic you want!

You can now update the usage of this module in staging (in

live/stage/services/webserver-cluster/main.tf) to disable auto scaling by setting `enable_autoscaling` to `false`:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-
cluster"

  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-
stores/mysql/terraform.tfstate"

  instance_type        = "t2.micro"
  min_size             = 2
  max_size             = 2
  enable_autoscaling   = false
}
```

Similarly, you can update the usage of this module in production (in *live/prod/services/webserver-cluster/main.tf*) to enable auto scaling by setting `enable_autoscaling` to `true` (make sure to also remove the custom `aws_autoscaling_schedule` resources that were in the production environment from Chapter 4):

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-
cluster"

  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-
stores/mysql/terraform.tfstate"

  instance_type        = "m4.large"
  min_size             = 2
  max_size             = 10
  enable_autoscaling   = true

  custom_tags = {
```

```
    Owner      = "team-foo"
    DeployedBy = "terraform"
  }
}
```

This approach works well if the user passes an explicit boolean value to your module, but what do you do if the boolean is the result of a more complicated comparison, such as string equality? Let's go through a more complicated example.

Imagine that as part of the `webserver-cluster` module, you wanted to create a set of CloudWatch alarms. A *CloudWatch alarm* can be configured to notify you via a variety of mechanisms (e.g., email, text message) if a specific metric exceeds a predefined threshold. For example, here is how you could use the `aws_cloudwatch_metric_alarm` resource in *modules/services/webserver-cluster/main.tf* to create an alarm that goes off if the average CPU utilization in the cluster is over 90% during a 5-minute period:

```
resource "aws_cloudwatch_metric_alarm"
"high_cpu_utilization" {
  alarm_name  = "${var.cluster_name}-high-cpu-
utilization"
  namespace   = "AWS/EC2"
  metric_name = "CPUUtilization"

  dimensions = {
    AutoScalingGroupName =
aws_autoscaling_group.example.name
  }

  comparison_operator = "GreaterThanThreshold"
  evaluation_periods  = 1
  period              = 300
  statistic           = "Average"
  threshold           = 90
  unit                = "Percent"
```

```
}
```

This works fine for a CPU Utilization alarm, but what if you wanted to add another alarm that goes off when CPU credits are low?[1] Here is a CloudWatch alarm that goes off if your web server cluster is almost out of CPU credits:

```
resource "aws_cloudwatch_metric_alarm"
"low_cpu_credit_balance" {
  alarm_name = "${var.cluster_name}-low-cpu-credit-
balance"
  namespace   = "AWS/EC2"
  metric_name = "CPUCreditBalance"

  dimensions = {
    AutoScalingGroupName =
aws_autoscaling_group.example.name
  }

  comparison_operator = "LessThanThreshold"
  evaluation_periods  = 1
  period              = 300
  statistic           = "Minimum"
  threshold           = 10
  unit                = "Count"
}
```

The catch is that CPU credits only apply to tXXX Instances (e.g., `t2.micro`, `t2.medium`, etc). Larger instance types (e.g., `m4.large`) don't use CPU credits and don't report a `CPUCreditBalance` metric, so if you create such an alarm for those instances, the alarm will always be stuck in the "INSUFFICIENT_DATA" state. Is there a way to create an alarm only if `var.instance_type` starts with the letter "t"?

You could add a new boolean input variable called

`var.is_t2_instance`, but that would be redundant with `var.instance_type`, and you'd most likely forget to update one when updating the other. A better alternative is to use a conditional:

```
resource "aws_cloudwatch_metric_alarm"
"low_cpu_credit_balance" {
  count = format("%.1s", var.instance_type) == "t" ? 1 :
0

  alarm_name = "${var.cluster_name}-low-cpu-credit-
balance"
  namespace   = "AWS/EC2"
  metric_name = "CPUCreditBalance"

  dimensions = {
    AutoScalingGroupName =
aws_autoscaling_group.example.name
  }

  comparison_operator = "LessThanThreshold"
  evaluation_periods  = 1
  period              = 300
  statistic           = "Minimum"
  threshold           = 10
  unit                = "Count"
}
```

The alarm code is the same as before, except for the relatively complicated `count` parameter:

```
  count = format("%.1s", var.instance_type) == "t" ? 1 :
0
```

This code uses the `format` function to extract just the first character from `var.instance_type`. If that character is a "t" (e.g., `t2.micro`), it sets the `count` to 1; otherwise, it sets the count to 0. This way, the alarm is only created for instance types that actually have a

`CPUCreditBalance` metric.

IF-ELSE-STATEMENTS WITH THE COUNT PARAMETER

Now that you know how to do an if-statement, what about an if-else-statement?

Earlier in this chapter, you created several IAM users with read-only access to EC2. Imagine that you wanted to give one of these users, neo, access to CloudWatch as well, but to allow the person applying the Terraform configurations to decide if neo got only read access or both read and write access. This is a slightly contrived example, but it makes it easy to demonstrate a simple type of if-else-statement, where all that matters is that one of the if or else branches gets executed, and the rest of the Terraform code doesn't need to know which one.

Here is an IAM policy that allows read-only access to CloudWatch:

```
resource "aws_iam_policy" "cloudwatch_read_only" {
  name   = "cloudwatch-read-only"
  policy =
data.aws_iam_policy_document.cloudwatch_read_only.json
}

data "aws_iam_policy_document" "cloudwatch_read_only" {
  statement {
    effect    = "Allow"
    actions   = [
      "cloudwatch:Describe*",
      "cloudwatch:Get*",
      "cloudwatch:List*"
    ]
    resources = ["*"]
  }
}
```

And here is an IAM policy that allows full (read and write) access to CloudWatch:

```
resource "aws_iam_policy" "cloudwatch_full_access" {
  name   = "cloudwatch-full-access"
  policy =
data.aws_iam_policy_document.cloudwatch_full_access.json
}

data "aws_iam_policy_document" "cloudwatch_full_access"
{
  statement {
    effect    = "Allow"
    actions   = ["cloudwatch:*"]
    resources = ["*"]
  }
}
```

The goal is to attach one of these IAM policies to neo, based on the value of a new input variable called `give_neo_cloudwatch_full_access`:

```
variable "give_neo_cloudwatch_full_access" {
  description = "If true, neo gets full access to
CloudWatch"
  type        = bool
}
```

If you were using a general-purpose programming language, you might write an if-else-statement that looks like this:

```
# This is just pseudo code. It won't actually work in
Terraform.
if var.give_neo_cloudwatch_full_access {
  resource "aws_iam_user_policy_attachment"
"neo_cloudwatch_full_access" {
    user       = aws_iam_user.example[0].name
```

```
    policy_arn =
aws_iam_policy.cloudwatch_full_access.arn
  }
} else {
  resource "aws_iam_user_policy_attachment"
"neo_cloudwatch_read_only" {
    user       = aws_iam_user.example[0].name
    policy_arn = aws_iam_policy.cloudwatch_read_only.arn
  }
}
```

To do this in Terraform, you can use the `count` parameter and a conditional expression on each of the resources:

```
resource "aws_iam_user_policy_attachment"
"neo_cloudwatch_full_access" {
  count = var.give_neo_cloudwatch_full_access ? 1 : 0

  user       = aws_iam_user.example[0].name
  policy_arn = aws_iam_policy.cloudwatch_full_access.arn
}

resource "aws_iam_user_policy_attachment"
"neo_cloudwatch_read_only" {
  count = var.give_neo_cloudwatch_full_access ? 0 : 1

  user       = aws_iam_user.example[0].name
  policy_arn = aws_iam_policy.cloudwatch_read_only.arn
}
```

This code contains two `aws_iam_user_policy_attachment` resources. The first one, which attaches the CloudWatch full access permissions, has a conditional expression that will evaluate to 1 if `var.give_neo_cloudwatch_full_access` is `true` and 0 otherwise (this is the if-clause). The second one, which attaches the CloudWatch read-only permissions, has a conditional expression that does

the exact opposite, evaluating to 0 if `var.give_neo_cloudwatch_full_access` is `true` and 1 otherwise (this is the else-clause).

This approach works well if your Terraform code doesn't need to know which of the if or else clauses actually got executed. But what if you need to access some output attribute on the resource that comes out of the if or else clause? For example, what if you wanted to offer two different User Data scripts in the `webserver-cluster` module and allow users to pick which one gets executed? Currently, the `webserver-cluster` module pulls in the *user-data.sh* script via a `template_file` data source:

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  =
data.terraform_remote_state.db.outputs.address
    db_port     =
data.terraform_remote_state.db.outputs.port
  }
}
```

The current *user-data.sh* script looks like this:

```
#!/bin/bash

cat > index.html <<EOF
<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF
```

```
nohup busybox httpd -f -p ${server_port} &
```

Now, imagine that you wanted to allow some of your web server clusters to use this alternative, shorter script, called *user-data-new.sh*:

```
#!/bin/bash

echo "Hello, World, v2" > index.html
nohup busybox httpd -f -p ${server_port} &
```

To use this script, you need a new `template_file` data source:

```
data "template_file" "user_data_new" {
  template = file("${path.module}/user-data-new.sh")

  vars = {
    server_port = var.server_port
  }
}
```

The question is, how can you allow the user of the `webserver-cluster` module to pick from one of these User Data scripts? As a first step, you could add a new boolean input variable in *modules/services/webserver-cluster/variables.tf*:

```
variable "enable_new_user_data" {
  description = "If set to true, use the new User Data
script"
  type        = bool
}
```

If you were using a general-purpose programming language, you could add an if-else-statement to the launch configuration to pick between the

two User Data `template_file` options as follows:

```
# This is just pseudo code. It won't actually work in
Terraform.
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]

  if var.enable_new_user_data {
    user_data =
data.template_file.user_data_new.rendered
  } else {
    user_data = data.template_file.user_data.rendered
  }
}
```

To make this work with real Terraform code, you first need to use the if-else-statement trick from before to ensure that only one of the `template_file` data sources is actually created:

```
data "template_file" "user_data" {
  count = var.enable_new_user_data ? 0 : 1

  template = file("${path.module}/user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  =
data.terraform_remote_state.db.outputs.address
    db_port     =
data.terraform_remote_state.db.outputs.port
  }
}

data "template_file" "user_data_new" {
  count = var.enable_new_user_data ? 1 : 0
```

```
  template = file("${path.module}/user-data-new.sh")

  vars = {
    server_port = var.server_port
  }
}
```

If `var.enable_new_user_data` is `true`, then `data.template_file.user_data_new` will be created and `data.template_file.user_data` will not; if it's `false`, it'll be the other way around. All you have to do now is to set the `user_data` parameter of the `aws_launch_configuration` resource to the `template_file` that actually exists. To do this, you can use another conditional expression:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]

  user_data = (
    length(data.template_file.user_data[*]) > 0
      ? data.template_file.user_data[0].rendered
      : data.template_file.user_data_new[0].rendered
  )
}
```

Let's break the large value for the `user_data` parameter down. First, take a look at the boolean condition being evaluated:

```
length(data.template_file.user_data[*]) > 0
```

Note that the two `template_file` data sources are both lists, as they

both use the `count` parameter, so you have to use array syntax with them. However, as one of these lists will be of length 1 and the other of length 0, you can't directly access a specific index (e.g., `data.template_file.user_data[0]`), as that list may be empty. The solution is to use using a splat expression, which will always return a list (albeit possibly an empty one), and to check that list's length.

Using that list's length, we then pick from one of the following expressions:

```
? data.template_file.user_data[0].rendered
: data.template_file.user_data_new[0].rendered
```

Terraform does lazy evaluation for conditional results, so the true value will only be evaluated if the condition was true and the false value will only be evaluated if the condition was false. That makes it safe to look up index 0 on `user_data` and `user_data_new`, as we know that only the one with the non-empty list will actually be evaluated.

You can now try out the new User Data script in the staging environment by setting the `enable_new_user_data` parameter to `true` in *live/stage/services/webserver-cluster/main.tf*:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-
cluster"

  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-
stores/mysql/terraform.tfstate"

  instance_type        = "t2.micro"
  min_size             = 2
```

```
  max_size              = 2
  enable_autoscaling    = false
  enable_new_user_data = true
}
```

In the production environment, you can stick with the old version of the script by setting `enable_new_user_data` to `false` in *live/prod/services/webserver-cluster/main.tf*:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-
cluster"

  cluster_name           = "webservers-prod"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "prod/data-
stores/mysql/terraform.tfstate"

  instance_type        = "m4.large"
  min_size             = 2
  max_size             = 10
  enable_autoscaling   = true
  enable_new_user_data = false

  custom_tags = {
    Owner      = "team-foo"
    DeployedBy = "terraform"
  }
}
```

Using `count` and built-in functions to simulate if-else-statements is a bit of a hack, but it's one that works fairly well, and as you can see from the code, it allows you to conceal lots of complexity from your users so that they get to work with a clean and simple API.

Conditionals with for_each and for expressions

Now that you understand how to do conditional logic with resources using the `count` parameter, you can probably guess that you can use a similar strategy to do conditional logic with inline blocks inside of a resource by using a `for_each` expression. If you pass a `for_each` expression an empty list, it will produce 0 inline blocks; if you pass it a non-empty list, it'll create one or more inline blocks. The only question is, how can you conditionally decide of the list should be empty or not?

The answer is to combine the `for_each` expression with the `for` expression! For example, recall the way the `webserver-cluster` module in *modules/services/webserver-cluster/main.tf* sets tags:

```
dynamic "tag" {
  for_each = var.custom_tags

  content {
    key                 = tag.key
    value               = tag.value
    propagate_at_launch = true
  }
}
```

If `var.custom_tags` is empty, then the `for_each` expression will have nothing to loop over, so no tags will be set. In other words, you already have some conditional logic here. But you can go even further, by combining the `for_each` expression with a `for` expression as follows:

```
dynamic "tag" {
  for_each = {
    for key, value in var.custom_tags:
    key => upper(value)
    if key != "Name"
  }
```

```
    content {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
```

The nested `for` expression loops over `var.custom_tags`, converts each value to upper case (e.g., perhaps for consistency), and uses a conditional in the `for` expression to filter out any `key` set to `Name`, as the module already sets its own `Name` tag. By filtering values in the `for` expression, you can implement any arbitrary conditional logic you want for inline blocks!

Conditionals with the if string directive

Earlier in the chapter, you used the `for` string directive to do loops within a string. Let's now look at a second type of for directive, which has the following form:

```
%{ if <CONDITION> }<TRUEVAL>%{ endif }
```

Where `CONDITION` is any expression that evaluates to a boolean and `TRUEVAL` is the expression to render if `CONDITION` evaluates to true. You can optionally include an `else` clause as follows:

```
%{ if <CONDITION> }<TRUEVAL>%{ else }<FALSEVAL>%{ endif
}
```

Where `FALSEVAL` is the expression to render if `CONDITION` evaluates to false. Here's an example:

```
variable "name" {
  description = "A name to render"
  type        = string
}

output "if_else_directive" {
  value = "Hello, %{ if var.name != "" }${var.name}%{
else }(unnamed)%{ endif }"
}
```

If you run `terraform apply`, setting the `name` variable to "World", you'll see:

```
$ terraform apply -var name="World"

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

Outputs:

if_else_directive = Hello, World
```

If you run `terraform apply` with `name` set to an empty string, you instead get:

```
$ terraform apply -var name=""

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

Outputs:

if_else_directive = Hello, (unnamed)
```

Zero-Downtime Deployment

Now that your module has a clean and simple API for deploying a web

server cluster, an important question to ask is, how do you update that cluster? That is, when you have made changes to your code, how do you deploy a new AMI across the cluster? And how do you do it without causing downtime for your users?

The first step is to expose the AMI as an input variable in *modules/services/webserver-cluster/variables.tf*. In real-world examples, this is all you would need, as the actual web server code would be defined in the AMI. However, in the simplified examples in this book, all of the web server code is actually in the User Data script, and the AMI is just a vanilla Ubuntu image. Switching to a different version of Ubuntu won't make for much of a demonstration, so in addition to the new AMI input variable, you can also add an input variable to control the text the User Data script returns from its one-liner HTTP server:

```
variable "ami" {
  description = "The AMI to run in the cluster"
  default     = "ami-0c55b159cbfafe1f0"
  type        = string
}

variable "server_text" {
  description = "The text the web server should return"
  default     = "Hello, World"
  type        = string
}
```

Earlier in the chapter, to practice with if-else-statements, you created two User Data scripts. Let's consolidate that back down to one to keep things simple. First, in *modules/services/webserver-cluster/variables.tf*, remove the `enable_new_user_data` input variable. Second, in *modules/services/webserver-cluster/main.tf*, remove the `template_file` resource called `user_data_new`. Third, in the same

file, update the other `template_file` resource, called `user_data`, to no longer use the `enable_new_user_data` input variable, and to add the new `server_text` input variable to its `vars` block:

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  =
data.terraform_remote_state.db.outputs.address
    db_port     =
data.terraform_remote_state.db.outputs.port
    server_text = var.server_text
  }
}
```

Now you need to update the *modules/services/webserver-cluster/user-data.sh* Bash script to use this `server_text` variable in the `<h1>` tag it returns:

```
#!/bin/bash

cat > index.html <<EOF
<h1>${server_text}</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

nohup busybox httpd -f -p ${server_port} &
```

Finally, find the launch configuration in *modules/services/webserver-cluster/main.tf*, set its `user_data` parameter to the remaining `template_file` (the one called `user_data`), and set its `ami` parameter to the new `ami` input variable:

```
resource "aws_launch_configuration" "example" {
  image_id        = var.ami
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]

  user_data = data.template_file.user_data.rendered
}
```

Now, in the staging environment, in *live/stage/services/webserver-cluster/main.tf*, you can set the new `ami` and `server_text` parameters and remove the `enable_new_user_data` parameter:

```
module "webserver_cluster" {
  source = "../../../../modules/services/webserver-
cluster"

  ami         = "ami-0c55b159cbfafe1f0"
  server_text = "New server text"

  cluster_name           = "webservers-stage"
  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "stage/data-
stores/mysql/terraform.tfstate"

  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}
```

This code uses the same Ubuntu AMI, but changes the `server_text` to a new value. If you run the `plan` command, you should see something like the following (I've omitted some of the output for clarity):

```
Terraform will perform the following actions:

  #
module.webserver_cluster.aws_autoscaling_group.example
```

```
will be updated in-place
  ~ resource "aws_autoscaling_group" "example" {
        id                          = "webservers-stage-
terraform-20190516131456645800000001"
      ~ launch_configuration      = "terraform-
20190516131456645800000001" -> (known after apply)
        (...)
    }

  #
module.webserver_cluster.aws_launch_configuration.example
 must be replaced
+/- resource "aws_launch_configuration" "example" {
      ~ id                          = "terraform-
20190516131456645800000001" -> (known after apply)
        image_id                    = "ami-
0c55b159cbfafe1f0"
        instance_type               = "t2.micro"
      ~ name                        = "terraform-
20190516131456645800000001" -> (known after apply)
      ~ user_data                   =
"bd7c0a6de4da4d6458e8b7447650b5616099dcb1" ->
"4919a13b8fb5226801746cfa14af45918b3db793" # forces
replacement
        (...)
    }

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

As you can see, Terraform wants to make two changes: first, replace the old launch configuration with a new one that has the updated `user_data`, and second, modify the Auto Scaling Group in place to reference the new launch configuration. The problem is that merely referencing the new launch configuration will have no effect until the Auto Scaling Group launches new EC2 Instances. So how do you tell the Auto Scaling Group to deploy new Instances?

One option is to destroy the ASG (e.g., by running `terraform`

`destroy`) and then re-create it (e.g., by running `terraform apply`). The problem is that after you delete the old ASG, your users will experience downtime until the new ASG comes up. What you want to do instead is a *zero-downtime deployment*. The way to accomplish that is to create the replacement ASG first and then destroy the original one. As it turns out, Terraform has a `lifecycle` setting that does exactly this!

You first saw `lifecycle` settings in Chapter 3 with `prevent_destroy`. Another `lifecycle` setting that is particularly useful is called `create_before_destroy`. Normally, when replacing a resource, Terraform deletes the old resource first and then creates its replacement. However, if you set `create_before_destroy` to `true`, Terraform will work in the opposite order, creating the replacement resource first and then deleting the old resource.

Here's how you can take advantage of this lifecycle setting to get a zero-downtime deployment:[2]

1. Configure the `name` parameter of the ASG to depend directly on the name of the launch configuration. That way, each time the launch configuration changes (which it will when you update the AMI or User Data), its name will change, and therefore the ASG's name will change, which forces Terraform to replace the ASG.
2. Set the `create_before_destroy` parameter of the ASG to `true`, so each time Terraform tries to replace it, it will create the replacement ASG before destroying the original.
3. Set the `min_elb_capacity` parameter of the ASG to the `min_size` of the cluster so that Terraform will wait for at least that many servers from the new ASG to pass health checks in the ALB before it'll start destroying the original ASG.

Here is what the updated `aws_autoscaling_group` resource should look like in *modules/services/webserver-cluster/main.tf*:

```
resource "aws_autoscaling_group" "example" {
  # Explicitly depend on the launch configuration's name
so each time it's replaced,
  # this ASG is also replaced
  name = "${var.cluster_name}-
${aws_launch_configuration.example.name}"

  launch_configuration =
aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  # Wait for at least this many instances to pass health
checks before
  # considering the ASG deployment complete
  min_elb_capacity = var.min_size

  # When replacing this ASG, create the replacement
first, and only delete the
  # original after
  lifecycle {
    create_before_destroy = true
  }

  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }

  dynamic "tag" {
    for_each = {
      for key, value in var.custom_tags:
      key => upper(value)
```

```
      if key != "Name"
    }

    content {
      key                 = tag.key
      value               = tag.value
      propagate_at_launch = true
    }
  }
}
```

If you rerun the `plan` command, you'll now see something that looks like this (I've omitted some of the output for clarity):

```
Terraform will perform the following actions:

  #
module.webserver_cluster.aws_autoscaling_group.example
must be replaced
+/- resource "aws_autoscaling_group" "example" {
      ~ id                          = "webservers-stage-
terraform-20190516131456645800000001" -> (known after
apply)
      ~ launch_configuration        = "terraform-
20190516131456645800000001" -> (known after apply)
      ~ name                        = "webservers-stage-
terraform-20190516131456645800000001" -> (known after
apply) # forces replacement
        (...)
    }

  #
module.webserver_cluster.aws_launch_configuration.example
 must be replaced
+/- resource "aws_launch_configuration" "example" {
      ~ id                              = "terraform-
20190516131456645800000001" -> (known after apply)
        image_id                        = "ami-
0c55b159cbfafe1f0"
        instance_type                   = "t2.micro"
      ~ name                            = "terraform-
20190516131456645800000001" -> (known after apply)
```

```
      ~ user_data                     =
"bd7c0a6de4da4d6458e8b7447650b5616099dcb1" ->
"4919a13b8fb5226801746cfa14af45918b3db793" # forces
replacement
        (...)
    }

    (...)

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

The key thing to notice is that the `aws_autoscaling_group` resource now says “must be replaced” next to its name parameter, which means Terraform will replace it with a new Auto Scaling Group running your new AMI or User Data. Run the `apply` command to kick off the deployment, and while it runs, consider how the process works.

You start with your original ASG running, say, v1 of your code (Figure 5-2).

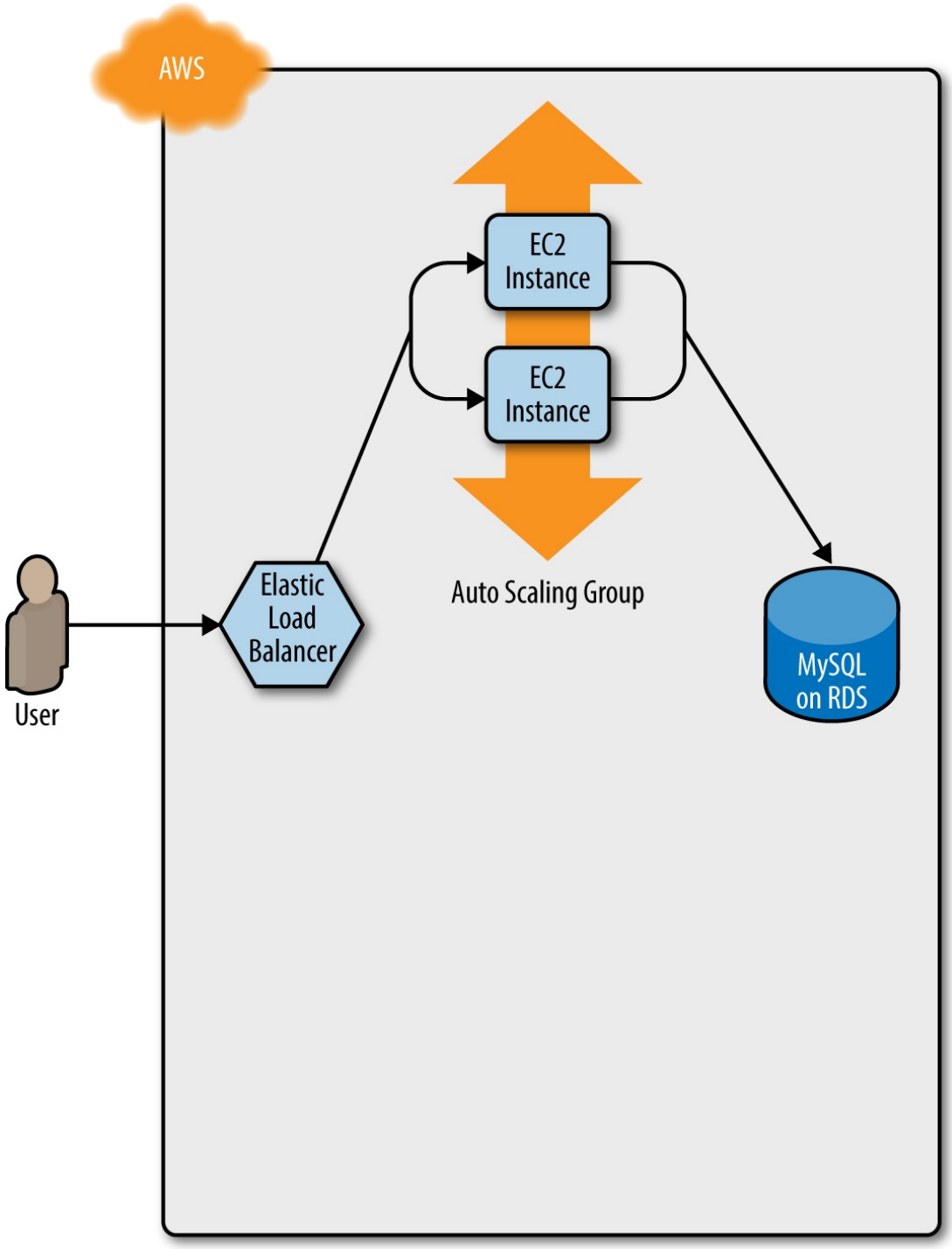

Figure 5-2. Initially, you have the original ASG running v1 of your code

You make an update to some aspect of the launch configuration, such as switching to an AMI that contains v2 of your code, and run the `apply` command. This forces Terraform to start deploying a new ASG with v2 of

your code (Figure 5-3).

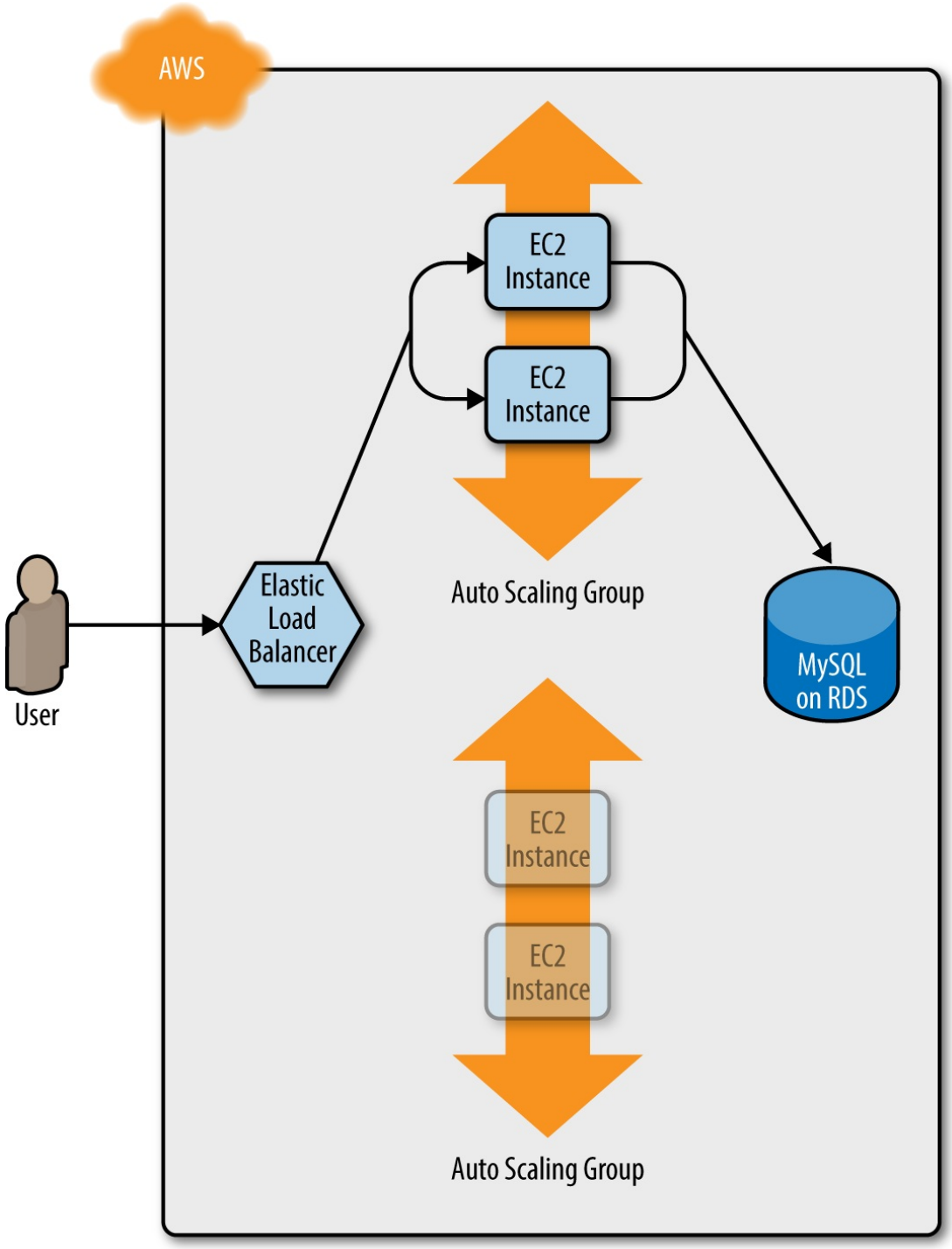

Figure 5-3. Terraform begins deploying the new ASG with v2 of your code

After a minute or two, the servers in the new ASG have booted, connected

to the database, registered in the ALB, and started to pass health checks. At this point, both the v1 and v2 versions of your app will be running simultaneously, and which one users see depends on where the ALB happens to route them (Figure 5-4).

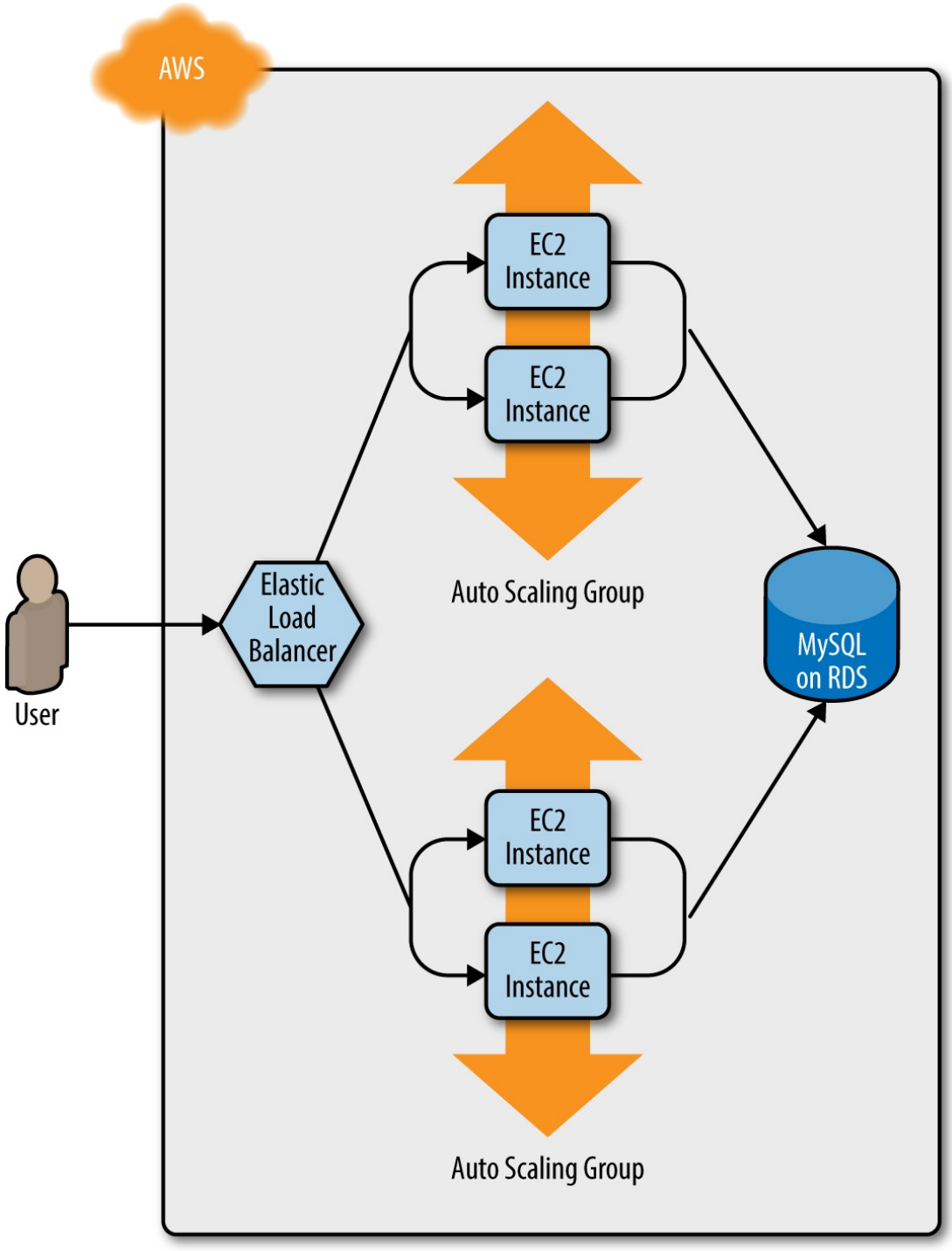

Figure 5-4. The servers in the new ASG boot up, connect to the DB, register in the ALB, and start serving traffic

Once `min_elb_capacity` servers from the v2 ASG cluster have registered in the ALB, Terraform will begin to undeploy the old ASG, first by deregistering the servers in that ASG from the ALB, and then by shutting them down (Figure 5-5).

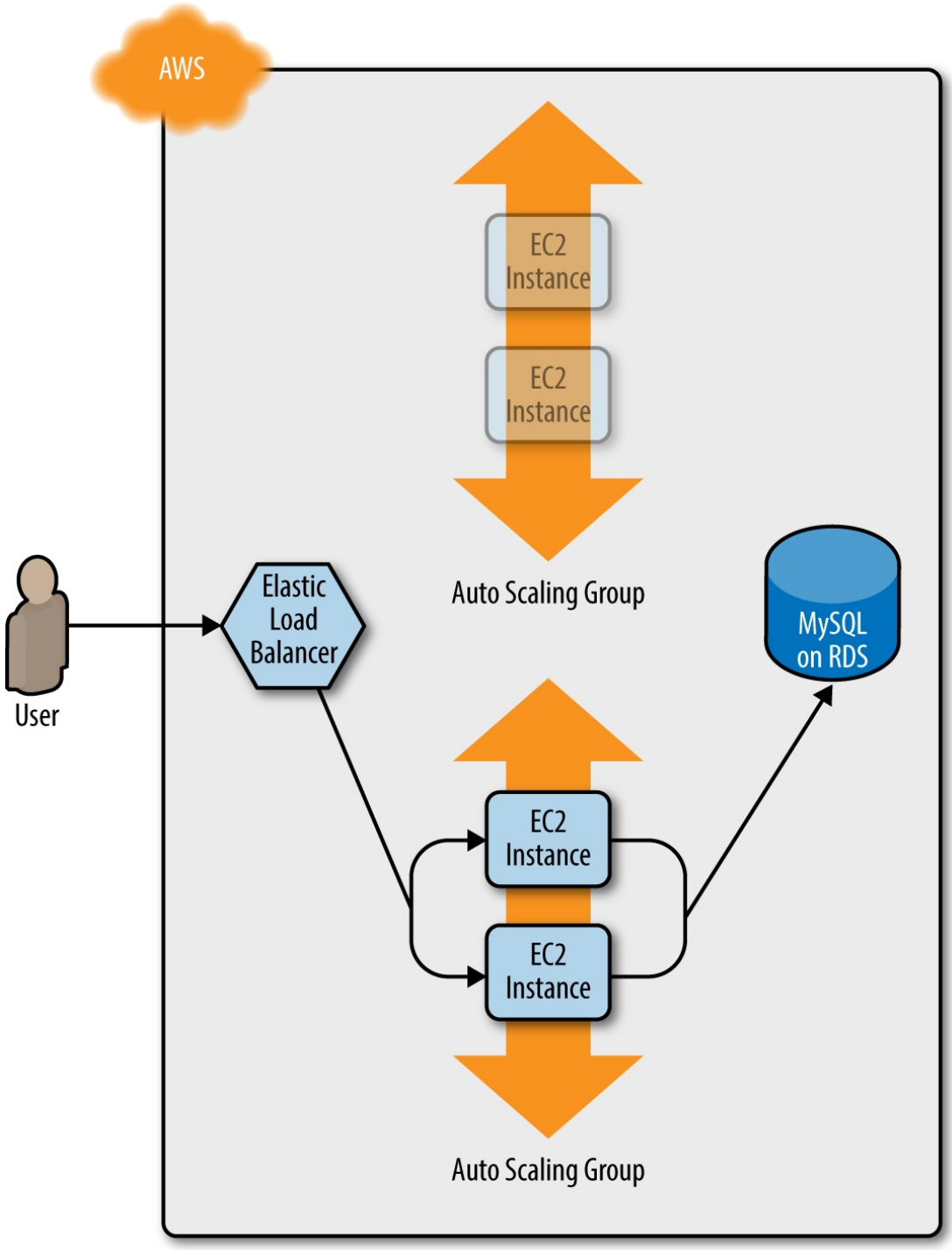

Figure 5-5. The servers in the old ASG begin to shut down

After a minute or two, the old ASG will be gone, and you will be left with just v2 of your app running in the new ASG (Figure 5-6).

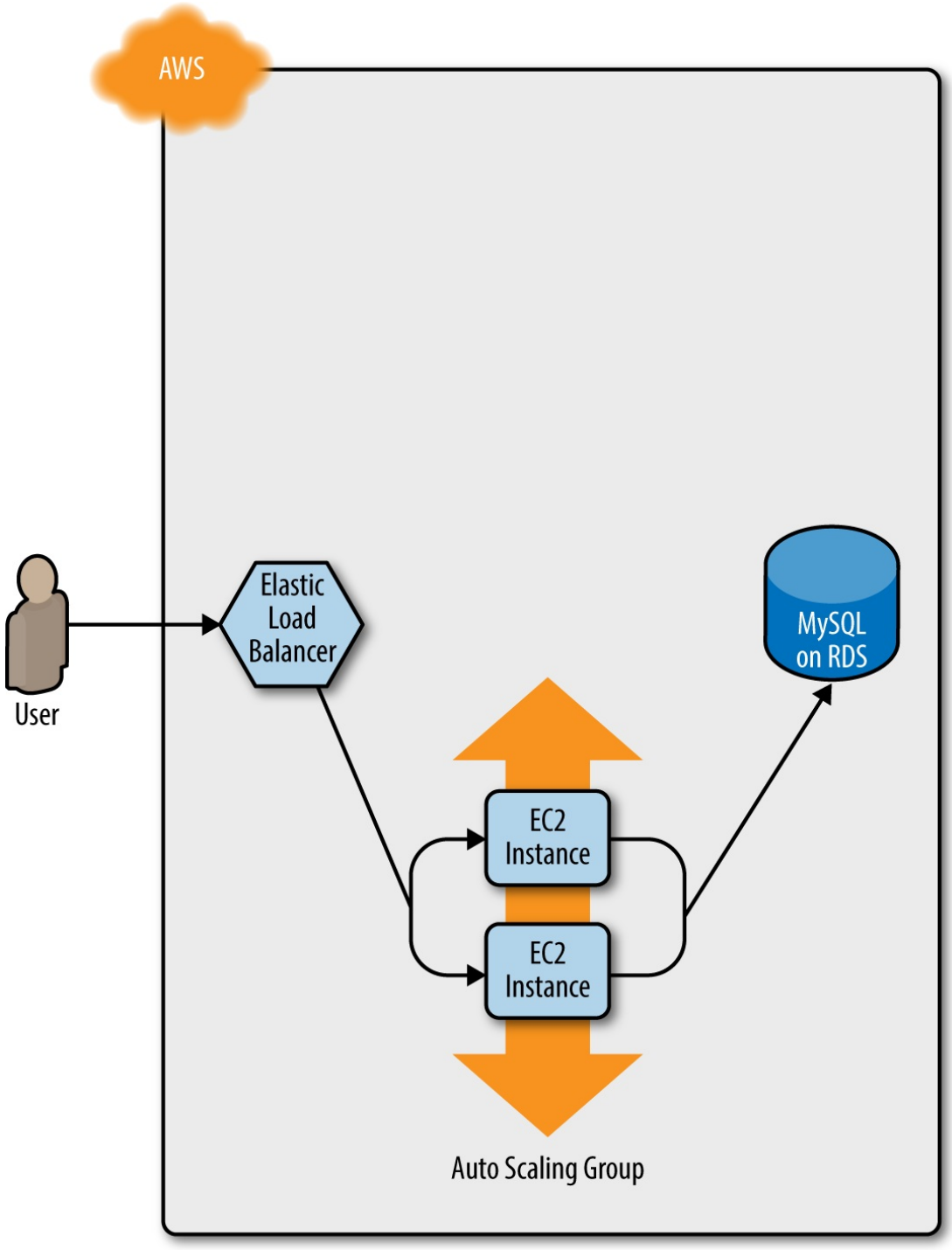

Figure 5-6. Now, only the new ASG remains, which is running v2 of your code

During this entire process, there are always servers running and handling requests from the ALB, so there is no downtime. Open the ALB URL in your browser and you should see something like Figure 5-7.

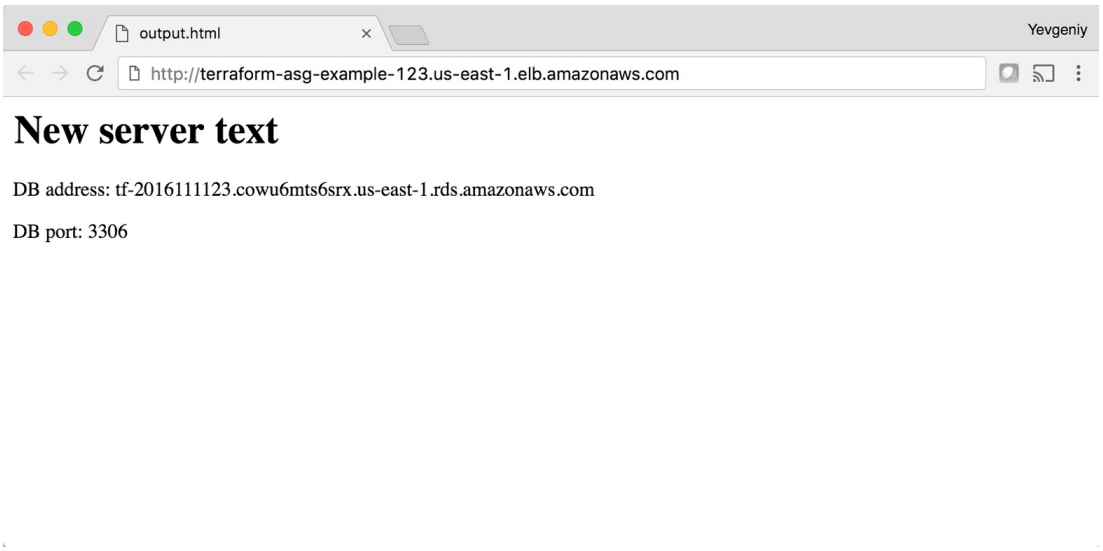

Figure 5-7. The new code is now deployed

Success! The new server text has deployed. As a fun experiment, make another change to the `server_text` parameter (e.g., update it to say "foo bar"), and run the `apply` command. In a separate terminal tab, if you're on Linux/Unix/OS X, you can use a Bash one-liner to run `curl` in a loop, hitting your ALB once per second, and allowing you to see the zero-downtime deployment in action:

```
$ while true; do curl http://<load_balancer_url>; sleep
1; done
```

For the first minute or so, you should see the same response that says "New server text". Then, you'll start seeing it alternate between the "New server text" and "foo bar". This means the new Instances have registered in the ALB and passed health checks. After another minute, the "New server text" will disappear, and you'll only see "foo bar", which means the old ASG has been shut down. The output will look something like this (for clarity, I'm listing only the contents of the `<h1>` tags):

```
New server text
New server text
```

```
New server text
New server text
New server text
New server text
foo bar
New server text
foo bar
New server text
foo bar
New server text
foo bar
New server text
foo bar
New server text
foo bar
foo bar
foo bar
foo bar
foo bar
foo bar
```

As an added bonus, if something went wrong during the deployment, Terraform will automatically roll back! For example, if there was a bug in v2 of your app and it failed to boot, then the Instances in the new ASG will not register with the ALB. Terraform will wait up to `wait_for_capacity_timeout` (default is 10 minutes) for `min_elb_capacity` servers of the v2 ASG to register in the ALB, after which it will consider the deployment a failure, delete the v2 ASG, and exit with an error (meanwhile, v1 of your app continues to run just fine in the original ASG).

Terraform Gotchas

After going through all these tips and tricks, it's worth taking a step back and pointing out a few gotchas, including those related to the loop, if-statement, and deployment techniques, as well as those related to more

general problems that affect Terraform as a whole:

- Count has limitations
- Zero-downtime deployment has limitations
- Valid plans can fail
- Refactoring can be tricky
- Eventual consistency is consistent…eventually

Count Has Limitations

In the examples in this chapter, you made extensive use of the `count` parameter in loops and if-statements. This works well, but there are three important limitations to `count` that you need to be aware of:

1. You cannot reference any resource outputs in `count`.
2. You cannot use `count` within a `module` configuration.
3. You cannot (easily) change `count`

Let's dig into these one at a time.

YOU CANNOT REFERENCE ANY RESOURCE OUTPUTS IN COUNT.

Imagine you wanted to deploy multiple EC2 Instances, and for some reason you didn't want to use an Auto Scaling Group. The code might look like this:

```
resource "aws_instance" "example_1" {
  count         = 3
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Since `count` is being set to a hard-coded value, this code will work without issues, and when you run `apply`, it will create 3 EC2 Instances. Now, what if you wanted to deploy one EC2 Instance per availability zone (AZ) in the current AWS region? You could update your code to fetch the list of AZs using the `aws_availability_zones` data source and use the `count` parameter and array lookups to "loop" over each AZ and create an EC2 Instance in it:

```
resource "aws_instance" "example_2" {
  count             =
length(data.aws_availability_zones.all.names)
  availability_zone =
data.aws_availability_zones.all.names[count.index]
  ami               = "ami-0c55b159cbfafe1f0"
  instance_type     = "t2.micro"
}

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

Again, this code will work just fine, as `count` can reference data sources without problems. However, what happens if the number of instances you needed to create depended on the output of some resource? The easiest way to experiment with this is to use the `random_integer` resource, which, as you can probably guess from the name, returns a random integer:

```
resource "random_integer" "num_instances" {
  min = 1
  max = 3
}
```

This code generates a random integer between 1 and 3. Let's see what happens if you try to use the `result` output from this resource in the

`count` parameter of your `aws_instance` resource:

```
resource "aws_instance" "example_3" {
  count         = random_integer.num_instances.result
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

If you run `terraform plan` on this code, you'll get the following error:

```
Error: Invalid count argument

  on main.tf line 30, in resource "aws_instance"
"example_3":
  30:   count         =
random_integer.num_instances.result

The "count" value depends on resource attributes that
cannot be determined
until apply, so Terraform cannot predict how many
instances will be created.
To work around this, use the -target argument to first
apply only the
resources that the count depends on.
```

The cause is that Terraform requires that it can compute the `count` parameter during the `plan` phase, *before* any resources are created or modified. That means `count` can reference hard-coded values, variables, data sources, and even lists of resources (so long as the length of the list can be determined during `plan`), but not computed resource outputs.

YOU CANNOT USE COUNT WITHIN A MODULE CONFIGURATION.

Something you may be tempted to try is to use the `count` parameter

within a `module` configuration:

```
module "count_example" {
  source = "../../../../modules/services/webserver-
cluster"

  count = 3

  cluster_name  = "terraform-up-and-running-example"
  server_port   = 8080
  instance_type = "t2.micro"
}
```

This code tries to use the `count` parameter on a `module` to create 3 copies of the webserver-cluster resources. Or, you may sometimes be tempted to try to set `count` to 0 on a `module` as a way to optionally include it or not based on some boolean condition. While the code looks perfectly reasonable, if you run `terraform plan`, you'll get the following error:

```
Error: Reserved argument name in module block

  on main.tf line 13, in module "count_example":
  13:   count = 3

The name "count" is reserved for use in a future version
of Terraform.
```

Unfortunately, as of Terraform 0.12, using `count` on `module` is not supported.

YOU CANNOT (EASILY) CHANGE `COUNT`

Perhaps the biggest gotcha with `count` is what happens when you try to change the value. Let's say you had a Terraform module that took in a list

of bucket names and created an S3 bucket for each one:

```
variable "bucket_names" {
  description = "Create S3 buckets with these names"
  type        = list(string)
}

resource "aws_s3_bucket" "example" {
  count  = length(var.bucket_names)
  bucket = var.bucket_names[count.index]
}

output "bucket_names" {
  value = aws_s3_bucket.example[*].bucket
}
```

Let's say you deployed this initially with three bucket names, "neo", "trinity", and "morpheus":

```
$ terraform apply -var 'bucket_names=["neo", "trinity",
"morpheus"]'

(...)

Apply complete! Resources: 3 added, 0 changed, 0
destroyed.

Outputs:

bucket_names = [
  "neo",
  "trinity",
  "morpheus",
]
```

Now, what happens if you wanted to remove one bucket from that list? Let's run `terraform plan`, but this time, with just "neo" and "morpheus" as the bucket names (the log output below is truncated for

clarity):

```
$ terraform plan -var 'bucket_names=["neo", "morpheus"]'

(...)

Terraform will perform the following actions:

  # aws_s3_bucket.example[1] must be replaced
-/+ resource "aws_s3_bucket" "example" {
      ~ bucket = "trinity" -> "morpheus" # forces
replacement
      (...)
    }

  # aws_s3_bucket.example[2] will be destroyed
  - resource "aws_s3_bucket" "example" {
      - bucket = "morpheus" -> null
      (...)
    }

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

Wait a second, that's probably not what you were expecting! The `plan` output is indicating that Terraform wants to delete *two* buckets—both trinity and morpheus—and to create a new one called morpheus. What's going on?

When you use the `count` parameter on a resource, that resource becomes a list or array of resources. Unfortunately, the way Terraform identifies each resource within the array is by its position (index) in that array. That is, after running `apply` the first time with three bucket names, Terraform's internal representation of these buckets looks something like this:

```
aws_s3_bucket.example[0]: neo
aws_s3_bucket.example[1]: trinity
```

```
aws_s3_bucket.example[2]: morpheus
```

When you remove an item from the middle of an array, all the items after it shift back by one, so after running `plan` with just two bucket names, Terraform's internal representation will look something like this:

```
aws_s3_bucket.example[0]: neo
aws_s3_bucket.example[1]: morpheus
```

Notice how morpheus has moved from index 2 to index 1. Since Terraform sees the index as a resource's identity, to Terraform, this change roughly translates to "rename the bucket at index 1 to morpheus and delete the bucket at index 2." Actually, since AWS doesn't allow you to rename buckets, what this really becomes is, "delete the buckets at index 1 and 2 and create a new bucket called morpheus to be stored at index 1."

In short, every time you use `count` to create a list of resources, if you remove an item from the middle of the list, Terraform will delete every resource after that item and then recreate those resources again from scratch. Ouch. The end result, of course, is exactly what you requested (i.e., two S3 buckets named morpheus and neo), but deleting and recreating resources is probably not how you want to get there!

The only solution currently available is to use the `terraform state mv` commands to update the indices in Terraform's state *before* running `terraform apply`. The `state mv` command has the following syntax:

```
terraform state mv <ORIGINAL_REFERENCE> <NEW_REFERENCE>
```

Where `ORIGINAL_REFERENCE` is the reference expression to the resource as it is now and `NEW_REFERENCE` is the new location you want to move it to. Using this command, you can move the bucket called "trinity" from index 1 to the end of the list at index 3:

```
$ terraform state mv aws_s3_bucket.example[1]
aws_s3_bucket.example[3]

Move "aws_s3_bucket.example[1]" to
"aws_s3_bucket.example[3]"
Successfully moved 1 object(s).
```

Next, you can use the command once more to move the bucket named "morpheus" from index 2 to index 1:

```
$ terraform state mv aws_s3_bucket.example[2]
aws_s3_bucket.example[1]

Move "aws_s3_bucket.example[2]" to
"aws_s3_bucket.example[1]"
Successfully moved 1 object(s).
```

Now, if you re-run `plan` with two bucket names, you should get the output you're expecting (the log output below is truncated for clarity):

```
$ terraform plan -var 'bucket_names=["neo", "morpheus"]'

(...)

Terraform will perform the following actions:

  # aws_s3_bucket.example[3] will be destroyed
  - resource "aws_s3_bucket" "example" {
      - bucket                      = "trinity" -> null
      (...)
    }
```

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

That means it's finally safe to run `apply` and be confident only the bucket you want deleted is removed, and not any of the others. Of course, if you had a list of 20 items and you needed to remove the 5th item, you'd have to run 16 `terraform state mv` commands: one to move the 5th item to the back of the list and 15 more to shift items 6 - 15 back one spot. For these sorts of use cases, you'll likely want to write a script to automate this process!

THE FUTURE OF COUNT AND FOR_EACH

As you can see, while `count` is very useful, it has a number of painful limitations. Terraform 0.12 introduced the `for_each` expression, but a few major features of `for_each` were not ready at the time of the release[3]:

1. Support for using `for_each` on resources instead of `count`.
2. Support for using `for_each` within `module` configurations.

In other words, in future versions of Terraform, `for_each` will most likely replace `count` and fix all of the limitations described in this section —including the gotcha with deleting items from the middle of a list, as `for_each` can loop over a map, using the keys of the map as a stable identity for each item, rather than the position within an array. Follow issue 17179 for progress.

Zero-Downtime Deployment has Limitations

Using `create_before_destroy` with an ASG is a great technique for zero-downtime deployment, but there is one limitation: it doesn't work

with auto scaling policies. Or, to be more accurate, it resets your ASG size back to its `min_size` after each deployment, which can be a problem if you had used auto scaling policies to increase the number of running servers.

For example, the web server cluster module includes a couple of `aws_autoscaling_schedule` resources that increase the number of servers in the cluster from 2 to 10 at 9 a.m. If you ran a deployment at, say, 11 a.m., the replacement ASG would boot up with only 2 servers, rather than 10, and would stay that way until 9 a.m. the next day.

There are several possible workarounds, including:

- Change the `recurrence` parameter on the `aws_autoscaling_schedule` from `0 9 * * *`, which means "run at 9 a.m.", to something like `0-59 9-17 * * *`, which means "run every minute from 9 a.m. to 5 p.m." If the ASG already has 10 servers, rerunning this auto scaling policy will have no effect, which is just fine; and if the ASG was just deployed, then running this policy ensures that the ASG won't be around for more than a minute before the number of Instances is increased to 10. This approach is a bit of a hack and the big jump from 10 servers to 2 servers back to 10 servers may still cause issues for your users.
- Create a custom script that uses the AWS API to figure out how many servers are running in the ASG, call this script using an `external` data source, and set the `desired_capacity` parameter of the ASG to the value returned by this script. That way, whenever a new ASG is launched, it'll always start with the capacity set to the same value as the ASG it is replacing. The downside is that using custom scripts makes your Terraform code less portable and harder to maintain.

Ideally, Terraform would have first-class support for zero-downtime deployment, but as of May 2019, the HashiCorp team has stated that they have no short-term plans to add this functionality (see *https://github.com/hashicorp/terraform/issues/1552* for details).

Valid Plans Can Fail

Sometimes, you run the `plan` command and it shows you a perfectly valid-looking plan, but when you run `apply`, you'll get an error. For example, try to add an `aws_iam_user` resource with the exact same name you used for the IAM user you created in Chapter 2:

```
resource "aws_iam_user" "existing_user" {
  # You should change this to the username of an IAM
user that already
  # exists so you can practice using the terraform
import command
  name = "yevgeniy.brikman"
}
```

If you now run the `plan` command, Terraform will show you a plan that looks reasonable:

```
Terraform will perform the following actions:

  # aws_iam_user.existing_user will be created
  + resource "aws_iam_user" "existing_user" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "yevgeniy.brikman"
      + path          = "/"
      + unique_id     = (known after apply)
    }

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

If you run the `apply` command, you'll get the following error:

```
Error: Error creating IAM User yevgeniy.brikman:
EntityAlreadyExists: User with name yevgeniy.brikman
already exists.
        status code: 409, request id: 71cd0053-77ef-
11e9-8831-41d1571eaa29

  on main.tf line 10, in resource "aws_iam_user"
"existing_user":
  10: resource "aws_iam_user" "existing_user" {
```

The problem, of course, is that an IAM user with that name already exists. This can happen not only with IAM users, but almost any resource. Perhaps someone created that resource manually or via CLI commands, but either way, some identifier is the same, and that leads to a conflict. There are many variations on this error, and Terraform newbies are often caught offguard by them.

The key realization is that `terraform plan` only looks at resources in its Terraform state file. If you create resources *out-of-band*—such as by manually clicking around the AWS console—they will not be in Terraform's state file, and therefore, Terraform will not take them into account when you run the `plan` command. As a result, a valid-looking plan may still fail.

There are two main lessons to take away from this:

Once you start using Terraform, you should only use Terraform

> Once a part of your infrastructure is managed by Terraform, you should never make changes manually to it. Otherwise, you not only set yourself up for weird Terraform errors, but you also void many of the benefits of using infrastructure as code in the first place, as that code

will no longer be an accurate representation of your infrastructure.

If you have existing infrastructure, use the `import` command

If you created infrastructure before you started using Terraform, you can use the `terraform import` command to add that infrastructure to Terraform's state file, so Terraform is aware of and can manage that infrastructure. The `import` command takes two arguments. The first argument is the "address" of the resource in your Terraform configuration files. This makes use of the same syntax as resource references, such as `<PROVIDER>_<TYPE>.<NAME>` (e.g., `aws_iam_user.existing_user`). The second argument is a resource-specific ID that identifies the resource to import. For example, the ID for an `aws_iam_user` resource is the name of the user (e.g., yevgeniy.brikman) and the ID for an `aws_instance` is the EC2 Instance ID (e.g., i-190e22e5). The documentation for each resource typically specifies how to import it at the bottom of the page.

For example, here is the `import` command you can use to sync the `aws_iam_user` you just added in your Terraform configurations with the IAM user you created back in Chapter 2 (obviously, you should replace "yevgeniy.brikman" with your own username in this command):

```
$ terraform import aws_iam_user.existing_user
yevgeniy.brikman
```

Terraform will use the AWS API to find your IAM user and create an association in its state file between that user and the `aws_iam_user.existing_user` resource in your Terraform configurations. From then on, when you run the `plan` command, Terraform will know that IAM user already exists and not try to create it again.

Note that if you have a lot of existing resources that you want to import into Terraform, writing the Terraform code for them from scratch and importing them one at a time can be painful, so you may

want to look into a tool such as Terraforming, which can import both code and state from an AWS account automatically.

Refactoring Can Be Tricky

A common programming practice is *refactoring*, where you restructure the internal details of an existing piece of code without changing its external behavior. The goal is to improve the readability, maintainability, and general hygiene of the code. Refactoring is an essential coding practice that you should do regularly. However, when it comes to Terraform, or any infrastructure as code tool, you have to be careful about what defines the "external behavior" of a piece of code, or you will run into unexpected problems.

For example, a common refactoring practice is to rename a variable or a function to give it a clearer name. Many IDEs even have built-in support for refactoring and can rename the variable or function for you, automatically, across the entire codebase. While such a renaming is something you might do without thinking twice in a general-purpose programming language, you have to be very careful in how you do it in Terraform, or it could lead to an outage.

For example, the `webserver-cluster` module has an input variable named `cluster_name`:

```
variable "cluster_name" {
  description = "The name to use for all the cluster
resources"
  type        = string
}
```

Perhaps you start using this module for deploying microservices, and

initially, you set your microservice's name to `foo`. Later on, you decide you want to rename the service to `bar`. This may seem like a trivial change, but it may actually cause an outage.

That's because the `webserver-cluster` module uses the `cluster_name` variable in a number of resources, including the `name` parameters of the ALB and two security groups. If you change the `name` parameter of certain resources, Terraform will delete the old version of the resource and create a new version to replace it. If the resource you are deleting happens to be an ALB, there will be nothing to route traffic to your web server cluster until the new ALB boots up. Similarly, if the resource you are deleting happens to be a security group, your servers will reject all network traffic until the new security group is created.

Another refactor you may be tempted to do is to change a Terraform identifier. For example, consider the `aws_security_group` resource in the `webserver-cluster` module:

```
resource "aws_security_group" "instance" {
  name = "${var.cluster_name}-instance"
}
```

The identifier for this resource is called `instance`. Perhaps you were doing a refactor and you thought it would be clearer to change this name to `cluster_instance`. What's the result? Yup, you guessed it: downtime.

Terraform associates each resource identifier with an identifier from the cloud provider, such as associating an `iam_user` resource with an AWS IAM User ID or an `aws_instance` resource with an AWS EC2

Instance ID. If you change the resource identifier, such as changing the `aws_security_group` identifier from `instance` to `cluster_instance`, then as far as Terraform knows, you deleted the old resource and have added a completely new one. As a result, if you `apply` these changes, Terraform will delete the old security group and create a new one, and in the time period in between, your servers will reject all network traffic.

There are four main lessons you should take away from this discussion:

Always use the plan command

> All of these gotchas can be caught by running the `plan` command, carefully scanning the output, and noticing that Terraform plans to delete a resource that you probably don't want deleted.

Create before destroy

> If you do want to replace a resource, then think carefully about whether its replacement should be created before you delete the original. If so, then you may be able to use `create_before_destroy` to make that happen. Alternatively, you can also accomplish the same effect through two manual steps: first, add the new resource to your configurations and run the `apply` command; second, remove the old resource from your configurations and run the `apply` command again.

Changing identifiers requires changing state

> If you want to change the identifier associated with a resource (e.g., rename an `aws_security_group` from `"instance"` to `"cluster_instance"`) without accidentally deleting and recreating that resource, you'll need to update the Terraform state accordingly. You should never update Terraform state files by hand—instead, use the `terraform state` commands to do it for you. In particular, when renaming identifiers, you'll need to run the

`terraform state mv` command introduced in "You cannot (easily) change `count`". For example, if you're renaming an `aws_security_group` group from `instance` to `cluster_instace`, you'll want to run:

```
terraform state mv aws_security_group.instance
aws_security_group.cluster_instance
```

This tells Terraform that the state that used to be associated with `aws_security_group.instance` should now be associated with `aws_security_group.cluster_instance`. If you rename an identifier, and run this command, you'll know you did it right if the subsequent `terraform plan` shows no changes.

Some parameters are immutable

The parameters of many resources are immutable, so if you change them, Terraform will delete the old resource and create a new one to replace it. The documentation for each resource often specifies what happens if you change a parameter, so RTFM. And, once again, make sure to always use the `plan` command, and consider whether you should use a create-before-destroy strategy.

Eventual Consistency Is Consistent…Eventually

The APIs for some cloud providers, such as AWS, are asynchronous and eventually consistent. *Asynchronous* means the API may send a response immediately, without waiting for the requested action to complete. *Eventually consistent* means it takes time for a change to propagate throughout the entire system, so for some period of time, you may get inconsistent responses depending on which data store replica happens to respond to your API calls.

For example, let's say you make an API call to AWS asking it to create an

EC2 Instance. The API will return a “success” (i.e., 201 Created) response more or less instantly, without waiting for the EC2 Instance creation to complete. If you tried to connect to that EC2 Instance immediately, you’d most likely fail because AWS is still provisioning it or the Instance hasn’t booted yet. Moreover, if you made another API call to fetch information about that EC2 Instance, you may get an error in return (i.e., 404 Not Found). That’s because the information about that EC2 Instance may still be propagating throughout AWS, and it’ll take a few seconds before it’s available everywhere.

In short, whenever you use an asynchronous and eventually consistent API, you are supposed to wait and retry for a while until that action has completed and propagated. Unfortunately, the AWS SDK does not provide good tools for doing this, and Terraform used to be plagued with a number of bugs similar to #6813:

```
$ terraform apply
aws_subnet.private-persistence.2:
InvalidSubnetID.NotFound:
The subnet ID 'subnet-xxxxxxx' does not exist
```

That is, you create a resource (e.g., a subnet), and then try to look up some data about that resource (e.g., the ID of the newly created subnet), and Terraform can’t find it. Most of these bugs (including #6813) have been fixed, but they still crop up from time to time, especially when Terraform adds support for a new type of resource. These bugs are annoying, but fortunately, most of them are harmless. If you just rerun `terraform apply`, everything will work fine, since by the time you rerun it, the information has propagated throughout the system.

Conclusion

Although Terraform is a declarative language, it includes a large number of tools, such as variables and modules, which you saw in Chapter 4, and `count`, `for_each`, `for`, `create_before_destroy`, and built-in functions, which you saw in this chapter, that give the language a surprising amount of flexibility and expressive power. There are many permutations of the if-statement tricks shown in this chapter, so spend some time browsing the functions documentation and let your inner hacker go wild. OK, maybe not too wild, as someone still has to maintain your code, but just wild enough that you can create clean, beautiful APIs for your users.

These users will be the focus of the next chapter, which describes how to use Terraform as a team. This includes a discussion of what workflows you can use, how to manage environments, how to test your Terraform configurations, and more.

---

1 You can learn about CPU credits here: *http://amzn.to/2lTuvs5*.

2 Credit for this technique goes to Paul Hinze.

3 *https://www.hashicorp.com/blog/hashicorp-terraform-0-12-preview-for-and-for-each*