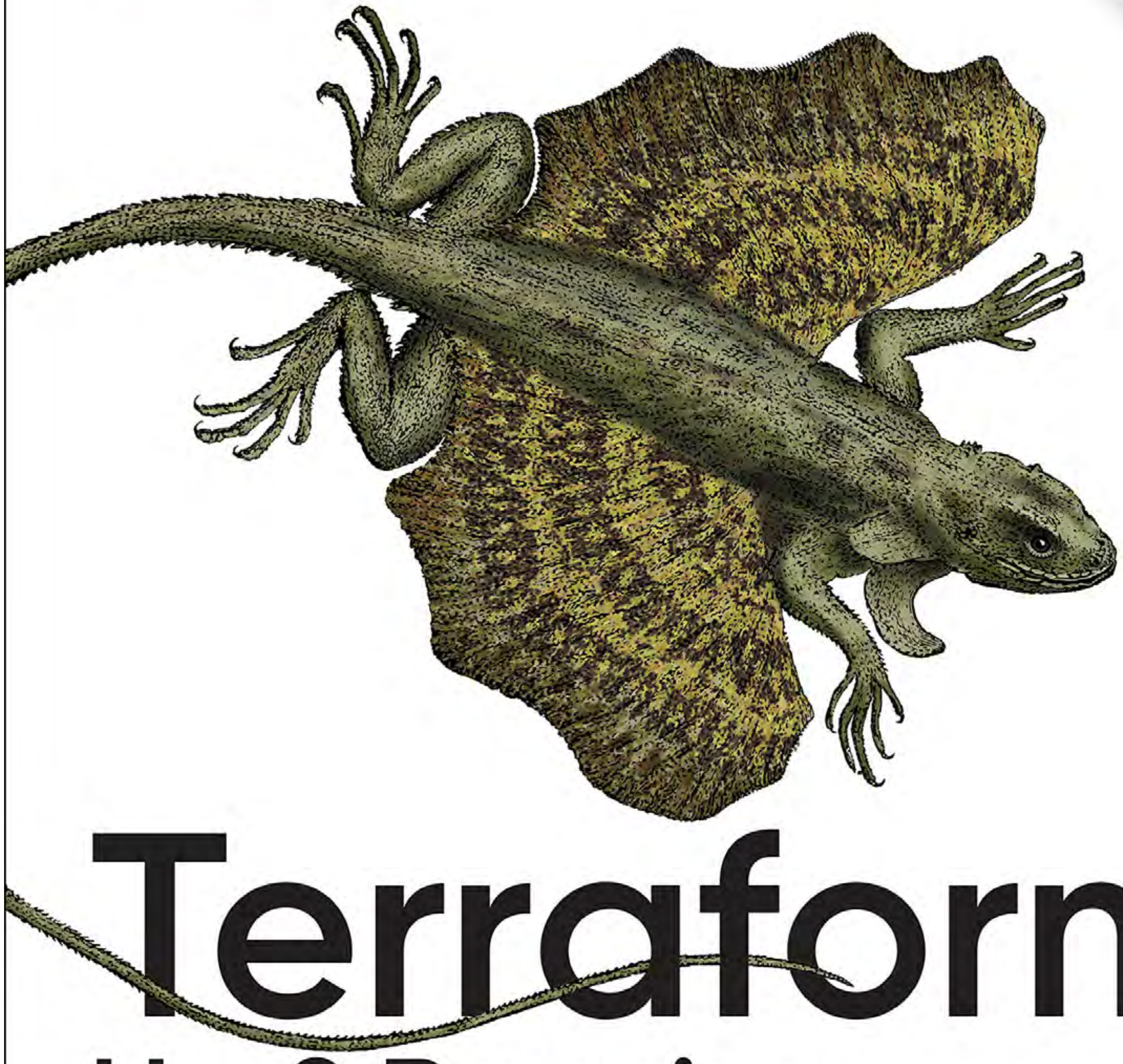
O'REILLY®

Second Edition



Terraform
Up & Running

Writing Infrastructure as Code

Yevgeniy Brikman

Table of Contents

Preface

A long time ago, in a data center far, far away, an ancient group of powerful beings known as sysadmins used to deploy infrastructure manually. Every server, every database, every load balancer, and every bit of network configuration was created and managed by hand. It was a dark and fearful age: fear of downtime, fear of accidental misconfiguration, fear of slow and fragile deployments, and fear of what would happen if the sysadmins fell to the dark side (i.e., took a vacation). The good news is that thanks to the DevOps movement, there is now a better way to do things: Terraform.

Terraform is an open source tool created by HashiCorp that allows you to define your infrastructure as code using a simple, declarative language, and to deploy and manage that infrastructure across a variety of public cloud providers (e.g., Amazon Web Services, Azure, Google Cloud, DigitalOcean) and private cloud and virtualization platforms (e.g., OpenStack, VMWare) using a few commands. For example, instead of manually clicking around a web page or running dozens of commands, here is all the code it takes to configure a server on Amazon Web Services:

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

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

And to deploy it, you just run the following:

```
$ terraform init
$ terraform apply
```

Thanks to its simplicity and power, Terraform has emerged as a key player in the DevOps world. It allows you to replace the tedious, fragile, and manual parts of infrastructure management with a solid, automated foundation upon which you can build all your other DevOps practices (e.g., automated testing, continuous integration, continuous delivery) and tooling (e.g., Docker, Chef, Puppet).

This book is the fastest way to get up and running with Terraform.

You'll go from deploying the most basic "Hello, World" Terraform example (in fact, you just saw it!) all the way up to running a full tech stack (server cluster, load balancer, database) capable of supporting a large amount of traffic and a large team of developers—all in the span of just a few chapters. This is a hands-on tutorial that not only teaches you DevOps and infrastructure as code principles, but also walks you through dozens of code examples that you can try at home, so make sure you have your computer handy.

By the time you're done, you'll be ready to use Terraform in the real world.

Who Should Read This Book

This book is for anyone responsible for the code after it has been written. That includes Sysadmins, Operations Engineers, Release Engineers, Site Reliability Engineers, DevOps Engineers, Infrastructure Developers, Full Stack Developers, Engineering Managers, and CTOs. No matter what your title is, if you're the one managing infrastructure, deploying code, configuring servers, scaling clusters, backing up data, monitoring apps, and responding to alerts at 3 a.m., then this book is for you.

Collectively, all of these tasks are usually referred to as "operations." In the past, it was common to find developers who knew how to write code, but did not understand operations; likewise, it was common to find sysadmins who understood operations, but did not know how to write code. You could get away with that divide in the past, but in the modern world, as cloud computing and the DevOps movement become ubiquitous, just about every developer will need to learn operational skills and every sysadmin will need to learn coding skills.

This book does not assume you're already an expert coder or expert sysadmin—a basic familiarity with programming, the command line, and server-based software (e.g., websites) should suffice. Everything else you need you'll be able to pick up as you go, so that by the end of the book, you will have a solid grasp of one of the most critical aspects of modern development and operations: managing infrastructure as code.

In fact, you'll learn not only how to manage infrastructure as code using Terraform, but also how this fits into the overall DevOps world. Here are some of the questions you'll be able to answer by the end of the book:

- Why use infrastructure as code at all?
- What are the differences between configuration management, orchestration, provisioning, and server templating?
- When should you use Terraform, Chef, Ansible, Puppet, Salt, CloudFormation, Docker, Packer, or Kubernetes?
- How does Terraform work and how do you use it to manage your infrastructure?
- How do you create reusable Terraform modules?
- How do you write Terraform code that's reliable enough for production usage?

- How do you test your Terraform code?
- How do you make Terraform a part of your automated deployment process?
- What are the best practices for using Terraform as a team?

The only tools you need are a computer (Terraform runs on most operating systems), an internet connection, and the desire to learn.

Why I Wrote This Book

Terraform is a powerful tool. It works with all popular cloud providers. It uses a clean, simple language and has strong support for reuse, testing, and versioning. It's open source and has a friendly, active community. But there is one area where it's lacking: maturity.

Terraform is a relatively new technology. As of May, 2019, it has not hit a 1.0.0 release yet, and despite Terraform's growing popularity, it's still hard to find books, blog posts, or experts to help you master the tool. The official Terraform documentation does a good job of introducing the basic syntax and features, but it includes little information on idiomatic patterns, best practices, testing, reusability, or team workflows. It's like trying to become fluent in French by studying only the vocabulary and not any of the grammar or idioms.

The reason I wrote this book is to help developers become fluent in Terraform. I've been using Terraform for 4 out of the 5 years it has existed, mostly at my company Gruntwork, where Terraform is one of the core tools we've used to create a library of over 300,000 lines of reusable, battle-tested infrastructure code that's used in production by hundreds of companies. Writing and maintaining this much infrastructure code, over this many years, and using it with so many different companies and use cases, has taught us a lot of hard lessons. My goal is to share these lessons with you, so you can cut this lengthy process down and become fluent in a matter of days.

Of course, you can't become fluent just by reading. To become fluent in French, you'll have to spend time talking with native French speakers, watching French TV shows, and listening to French music. To become fluent in Terraform, you'll have to write real Terraform code, use it to manage real software, and deploy that software on real servers. Therefore, be ready to read, write, and execute a lot of code.

What You Will Find in This Book

Here's an outline of what the book covers:

Chapter 1, *Why Terraform*

How DevOps is transforming the way we run software; an overview of infrastructure as code tools, including configuration management, server templating, orchestration, and provisioning tools; the benefits of infrastructure as code; a comparison of Terraform, Chef, Puppet, Ansible, SaltStack, OpenStack Heat, and CloudFormation; how to combine tools such as Terraform, Packer, Docker, Ansible, and Kubernetes.

Chapter 2, *Getting Started with Terraform*

Installing Terraform; an overview of Terraform syntax; an overview of the Terraform CLI tool; how to deploy a single server; how to deploy a web server; how to deploy a cluster of web servers; how to deploy a load balancer; how to clean up resources you've created.

Chapter 3, *How to Manage Terraform State*

What is Terraform state; how to store state so multiple team members can access it; how to lock state files to prevent race conditions; how to manage secrets with Terraform; how to isolate state files to limit the damage from errors; how to use Terraform workspaces; a best-practices file and folder layout for Terraform projects; how to use read-only state.

Chapter 4, *How to Create Reusable Infrastructure with Terraform Modules*

What are modules; how to create a basic module; how to make a module configurable with inputs and outputs; local values; versioned modules; module gotchas; using modules to define reusable, configurable pieces of infrastructure.

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

Loops with the `count` parameter, `for_each` and `for` expressions, and `for`string directive; conditionals with the `count` parameter, `for_each` and `for`expressions, and the `if` string directive; built-in functions; zero-downtime deployment; common Terraform gotchas and pitfalls, including count limitations, zero-downtime deployment gotchas, valid plans can fail, refactoring problems, and eventual consistency.

Chapter 6, *Production-grade Terraform code*

Why DevOps projects always take longer than you expect; the production-grade infrastructure checklist; how to build Terraform modules for production; small modules; composable modules; testable modules; releasable modules; Terraform Registry; Terraform escape hatches.

Chapter 7, *How to test Terraform code*

Manual tests for Terraform code; sandbox environments and cleanup; automated tests for Terraform code; Terratest; unit tests; integration tests; end-to-end tests; dependency injection; running tests in parallel; test stages; retries; the test pyramid; static analysis; property checking.

Chapter 8, *How to Use Terraform as a Team*

How to adopt Terraform as team; how to convince your boss; a workflow for deploying application code; a workflow for deploying infrastructure code; version control; the golden rule of Terraform; code reviews; coding guidelines; Terraform style; CI / CD for Terraform; deployment process.

Feel free to read the book from start to finish or jump around to the chapters that interest you the most. Note that the examples in each chapter reference and build upon the examples from the previous chapters, so if you skip around, use the open source code examples (as described in "Open Source Code Examples") to get your bearings. At the end of the book, in Appendix A, you'll find a list of recommended reading where you can learn more about Terraform, operations, infrastructure as code, and DevOps.

What's new in the 2nd edition

The first edition of this book came out in 2017; I'm now writing the second edition in May, 2019, and it's remarkable how much has changed in just a couple years! The second edition is almost double the length of the first edition (~160 more pages), including two completely new chapters, and major updates to all the original chapters and code examples.

If you read the first edition of the book and want to know what's new, or if you're just curious to see how Terraform has evolved, here are some of the highlights:

Four major Terraform releases

Terraform was at version 0.8 when this book first came out; between then and the time I'm writing this second edition now, Terraform has had four major releases, and is now at version 0.12. These releases introduced some amazing new functionality, as described below, as well as a fair amount of upgrade work for users![1]

Automated testing improvements

The tooling and practices for writing automated tests for Terraform code have evolved considerably. Chapter 7 is a completely new chapter dedicated to testing, covering topics such as unit tests, integration tests, end-to-end tests, dependency injection, test parallelism, static analysis, and more.

Module improvements

The tooling and practices for creating Terraform modules have also evolved considerably. In the brand new Chapter 6, you'll find a guide to building reusable, battle-tested, production-grade Terraform modules—the kind of modules you'd bet your company on.

Workflow improvements

Chapter 8 has been completely rewritten to reflect the changes in how teams integrate Terraform into their workflow, including a detailed guide on how to take application code and infrastructure code from development through testing and all the way to production.

Terraform state revamp

Terraform 0.9 introduced backends as a first-class way to store and share Terraform state, including built-in support for locking. Terraform 0.9 also introduced state environments as a way to manage deployments across multiple environments. In Terraform 0.10, state environments were replaced with Terraform workspaces. I cover all of these topics in Chapter 3.

HCL 2

Terraform 0.12 overhauled the underlying language from HCL to HCL 2. This included support for first-class expressions (so you don't have to wrap everything with `${…}`!), rich type constraints, lazily-evaluated conditional expressions, support for `null`, `for_each` and `for` expressions, dynamic inline blocks, and more. All the code examples in this book have been updated to use HCL2 and the new language features are covered extensively inChapter 5 and Chapter 6.

Terraform providers split

In Terraform 0.10, the core Terraform code was split up from the code for all the providers (i.e., the code for AWS, GCP, Azure, etc). This allowed providers to be developed in their own repos, at their own cadence, with their own versioning. However, you now have to run `terraform init` to download the provider code every time you start working with a new module, as discussed in Chapter 2 and Chapter 7.

Massive provider growth

Since 2016, Terraform has grown from a handful of major cloud providers (the usual suspects, such as AWS, GCP, and Azure) to over 100 official providers and many more community providers.[2] That means you can now use Terraform to not only manage many other types of clouds (e.g., there are now providers for Alicloud, Oracle Cloud Infrastructure, VMware vSphere, and others), but also to manage many other aspects of your world as code, including version control systems (e.g., using the GitHub, GitLab, or BitBucket providers), data stores (e.g., using the MySQL, PostreSQL, or InfluxDB providers), monitoring and alerting systems (e.g., using the DataDog, New Relic, or Grafana providers), platform tools (e.g., using the Kubernetes, Helm, Heroku, Rundeck, or Rightscale providers), and much more. Moreover, each provider has much better coverage these days: e.g., the AWS provider now covers the majority of important AWS services and often adds support for new services even before CloudFormation!

Terraform Registry

HashiCorp launched the Terraform Registry in 2017, a UI that made it easy to browse and consume open source, reusable Terraform modules contributed by the community. In 2018, HashiCorp added the ability to run a Private Terraform Registry within your own organization. Terraform 0.11 added first-class syntax support for consuming modules from a Terraform Registry. The Registry is discussed in "Releasable modules".

Better error handling

Terraform 0.9 updated state error handling: if there was an error writing state to a remote backend, the state would be saved locally in an `errored.tfstate` file. Terraform 0.11 did a better job of handling errors with outputs. Terraform 0.12 completely overhauled error handling, catch errors earlier, showing clearer error messages, and including the file path, line number, and a code snippet in the error message.

Many other small changes

There were many other smaller changes along the way, including the introduction of local values ("Module Locals"), new "escape hatches" for having Terraform interact with the outside world via scripts (e.g., "Beyond Terraform modules"), running `plan` as part of the `apply` command ("Deploy a Single Server"), fixes for the `create_before_destroy` cycle issues, major improvements to the `count` parameter so it can now include references to data sources and resources ("Loops"), dozens of new built-in functions, an overhaul in `provider` inheritance, and much more.

What You Won't Find in This Book

This book is not meant to be an exhaustive reference manual for Terraform. I do not cover all the cloud providers, or all of the resources supported by each cloud provider, or every available Terraform command. For these nitty-gritty details, I refer you instead to the Terraform documentation.

The documentation contains many useful answers, but if you're new to Terraform, infrastructure as code, or operations, you won't even know what questions to ask. Therefore, this book is focused on what the documentation does *not* cover: namely, how to go beyond introductory examples and use Terraform in a real-world setting. My goal is to get you up and running quickly by discussing why you may want to use Terraform in the first place, how to fit it into your workflow, and what practices and patterns tend to work best.

To demonstrate these patterns, I've included a number of code examples. I've tried to make it as easy as possible for you to try these examples at home by minimizing dependencies on any third parties. This is why almost all the examples use just a single cloud provider, Amazon Web Services (AWS), so you only have to sign up for a single third-party service (also, AWS offers a generous free tier, so running the example code shouldn't cost you anything). This is why the book and the example code do not cover or require HashiCorp's paid services, Terraform Pro and Terraform Enterprise. And this is why I've released all the code examples as open source.

Open Source Code Examples

All of the code samples in the book can be found at the following URL:

https://github.com/brikis98/terraform-up-and-running-code

You may want to check out this repo before you start reading so you can follow along with all the examples on your own computer:

```
git clone https://github.com/brikis98/terraform-up-and-running-code.git
```

The code examples in that repo are broken down chapter by chapter. It's worth noting that most of the examples show you what the code looks like at the *end* of a chapter. If you want to maximize your learning, you're better off writing the code yourself, from scratch.

You'll start coding in Chapter 2, where you'll learn how to use Terraform to deploy a basic cluster of web servers from scratch. After that, follow the instructions in each subsequent chapter on how to evolve and improve this web server cluster example. Make the changes as instructed, try to write all the code yourself, and only use the sample code in the GitHub repo as a way to check your work or get yourself unstuck.

A NOTE ABOUT VERSIONS

All of the examples in this book were tested against Terraform 0.12.x, which was the most recent major release at the time of writing. Since Terraform is a relatively new tool and has not hit version 1.0.0 yet, it is likely that future releases will contain backward incompatible changes and that some of the best practices will change and evolve over time.

I'll try to release updates as often as I can, but the Terraform project moves fast, so you'll have to do some work to keep up with it on your own. For the latest news, blog posts, and talks on Terraform and DevOps, be sure to check out this book's website and subscribe to the newsletter!

Using the Code Examples

This book is here to help you get your job done and you are welcome to use the sample code in your programs and documentation. You do not need to contact O'Reilly for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

Attribution is appreciated, but not required. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Terraform: Up and Running* by Yevgeniy Brikman (O'Reilly). Copyright 2017 Yevgeniy Brikman, 978-1-491-97708-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact O'Reilly Media at *permissions@oreilly.com*.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

O'Reilly Online Learning

NOTE

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

How to Contact O’Reilly Media

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/terraform-up-and-running*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Josh Padnick

This book would not have been possible without you. You were the one who introduced me to Terraform in the first place, taught me all the basics, and helped me figure out all the advanced parts. Thank you for supporting me while I took our collective learnings and turned them into a book. Thank you for being an awesome co-founder and making it possible to run a startup while still living a fun life. And thank you most of all for being a good friend and a good person.

O’Reilly Media

Thank you for publishing another one of my books. Reading and writing have profoundly transformed my life and I’m proud to have your help in sharing some of my writing with others. A special thanks to Brian Anderson for helping me get the first edition of this book out in record time, and to Virginia Wilson for somehow breaking that record for the second edition.

Gruntwork employees

I can’t thank you all enough for (a) joining our tiny startup, (b) building amazing software, (c) holding down the fort while I worked on the 2nd edition of this book, and (d) being amazing colleagues and friends.

Gruntwork customers

Thank you for taking a chance on a small, unknown company, and volunteering to be guinea pigs for our Terraform experiments. Gruntwork’s mission is to make it 10x easier to understand, develop, and deploy software. We haven’t always succeeded at that mission (I’ve captured many of our mistakes in this book!), so I’m grateful for your patience and willingness to be part of our audacious attempt to improve the world of software.

HashiCorp

Thank you for building an amazing collection of DevOps tools, including Terraform, Packer, Consul, and Vault. You’ve improved the world of DevOps and with it, the lives of millions of software developers.

Kief Morris, Seth Vargo, Mattias Gees, Ricardo Ferreira, Akash Mahajan, Moritz Heiber

Thank you for reading early versions of this book and providing lots of detailed, constructive feedback. Your suggestions have made this book significantly better.

Readers of the 1st edition

Those of you who bought the 1st edition of this book made the 2nd edition possible. Thank you. Your feedback, questions, pull requests, and constant prodding for updates motivated nearly 160 pages of new content. I hope you find the new content useful and I’m looking forward to the continued prodding.

Mom, Dad, Larisa, Molly

I accidentally wrote another book. That probably means I didn’t spend as much time with you as I wanted. Thank you for putting up with me anyway. I love you.

---

[1] Check out the Terraform upgrade guides for details.

[2] You can find the list of Terraform providers at *https://www.terraform.io/docs/providers/*

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 increase the number of features it delivered per month by 100%, 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 (check out Appendix A for recommended reading), 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, update, and destroy 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.

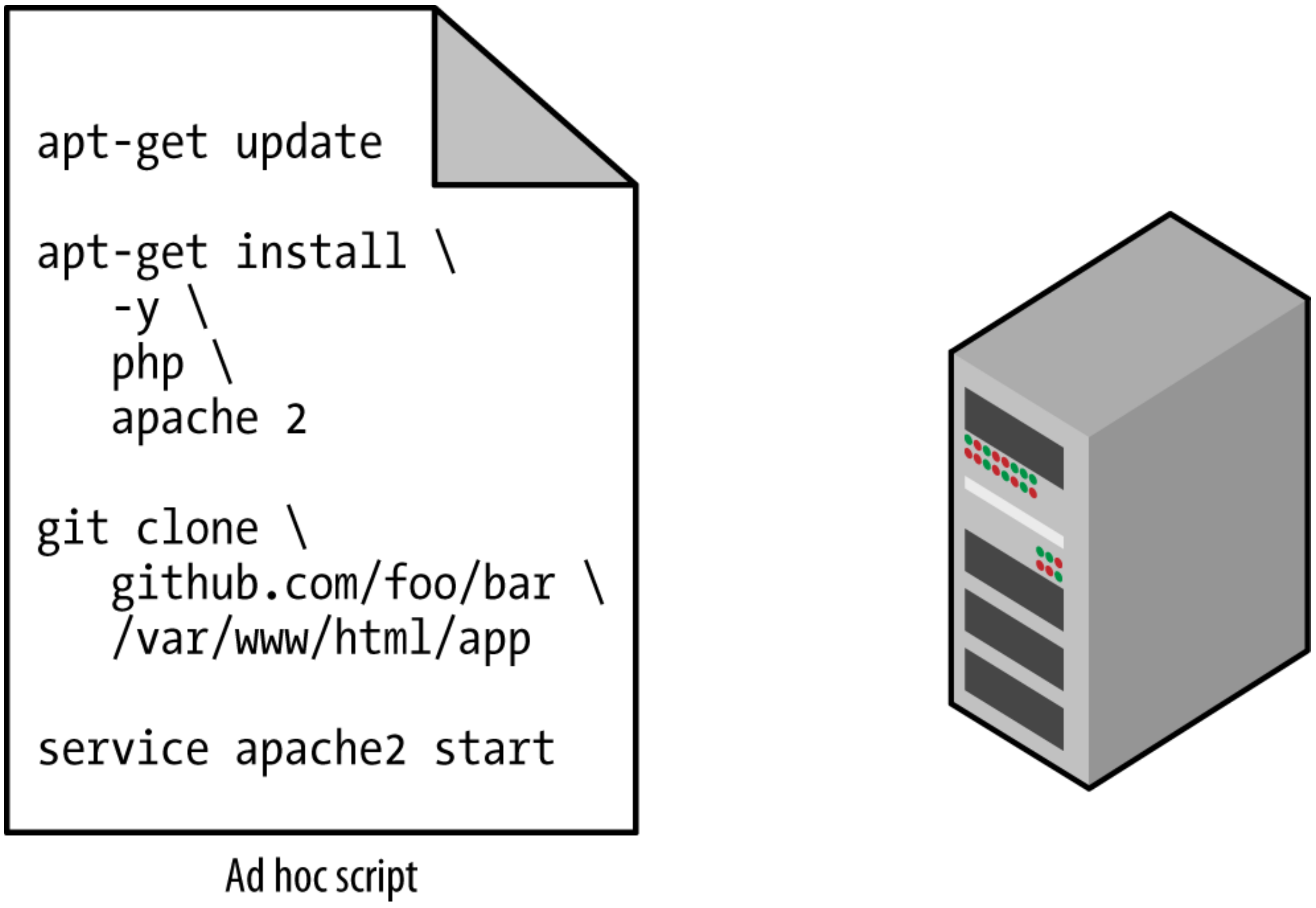
There are five broad categories of IAC tools:

- Ad hoc scripts
- Configuration management tools
- Server templating tools
- Orchestration tools
- 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.



```
apt-get update

apt-get install \
  -y \
  php \
  apache 2

git clone \
  github.com/foo/bar \
  /var/www/html/app

service apache2 start
```

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 an 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 a large repository of Bash 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.

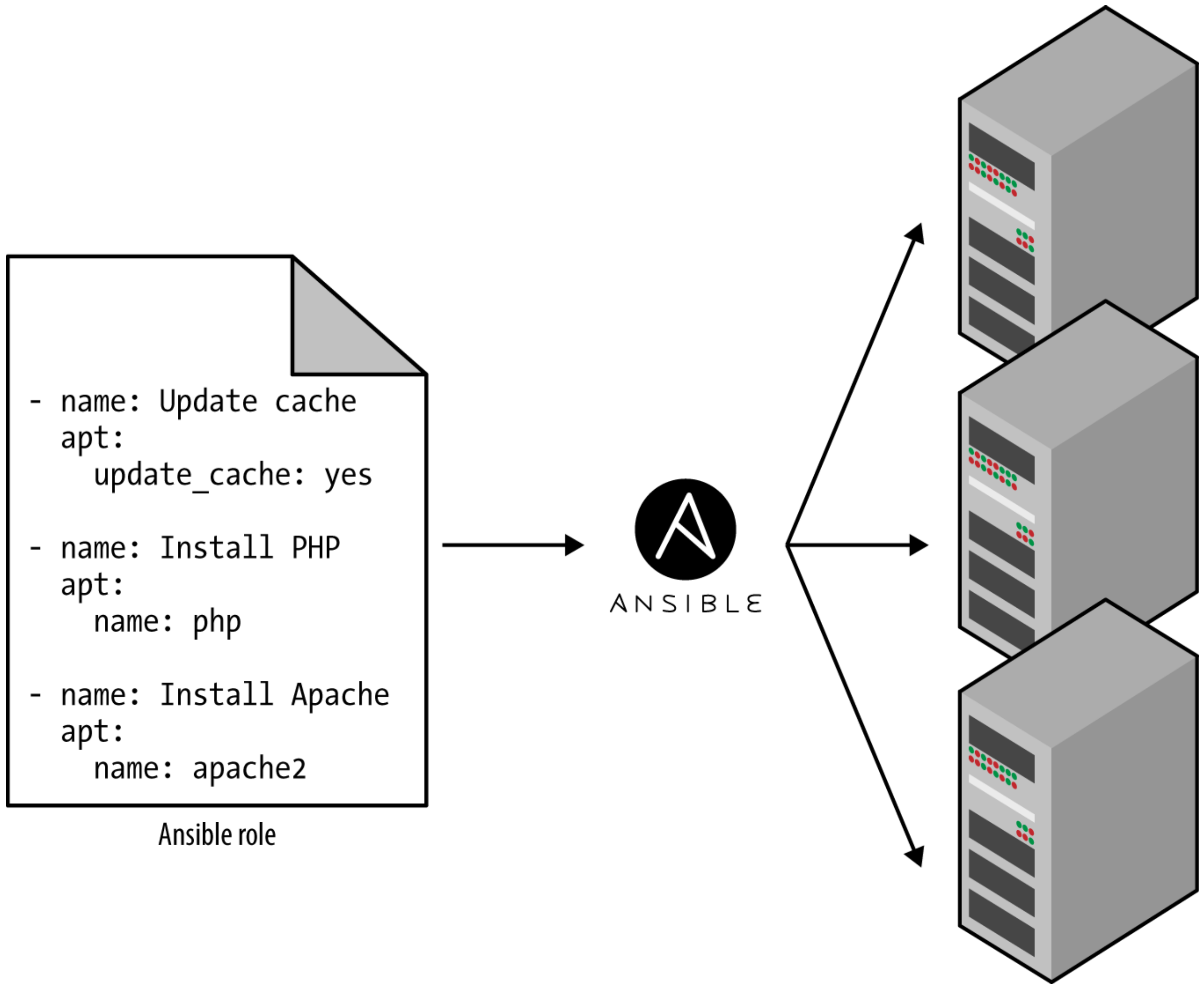


Figure 1-2. A configuration management tool like Ansible can execute your code across a large number of servers

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
```

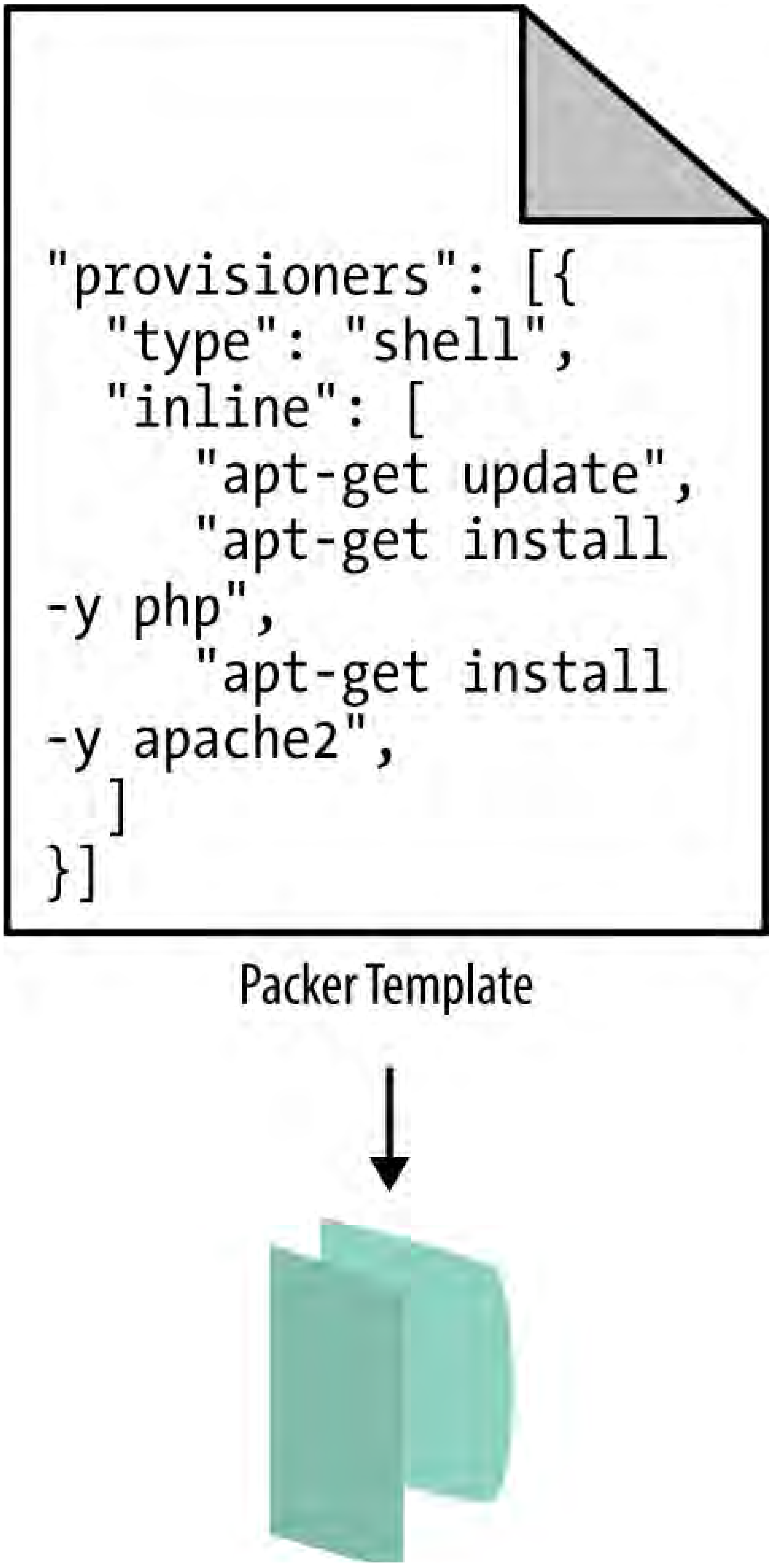
Finally, you execute the playbook as follows:

```
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",
  ]
}]
```

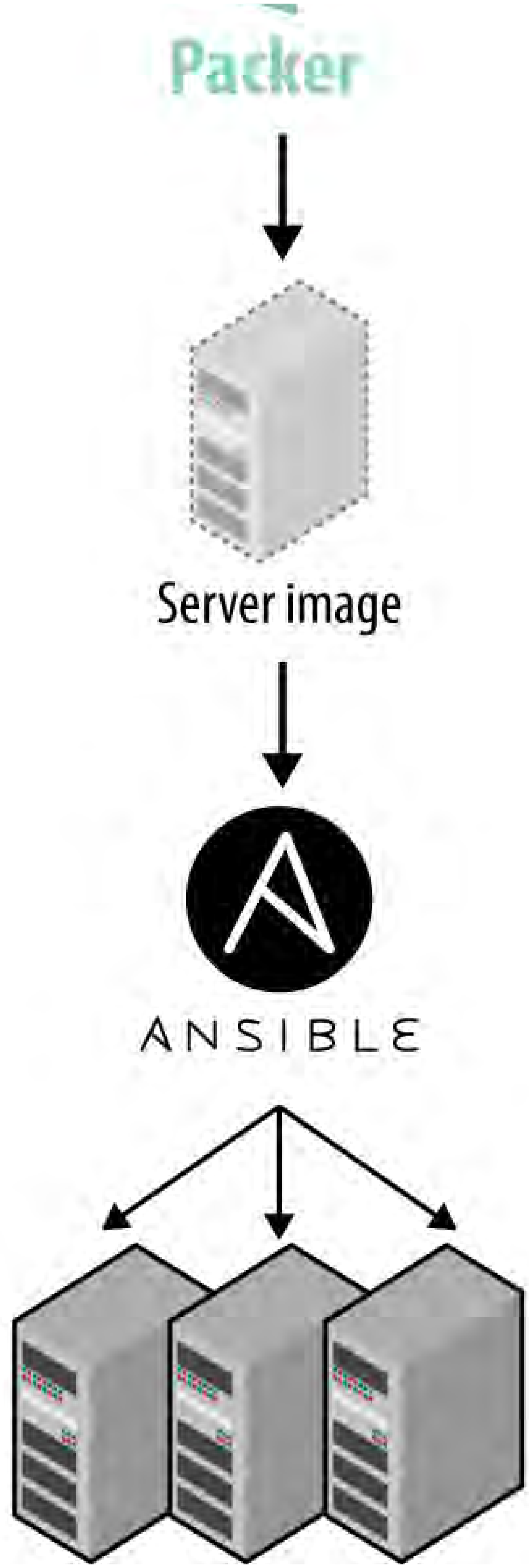

Packer
Server image
ANSIBLE

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, CoreOS rkt, or cri-o, 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 it's much harder to achieve the level of isolation and security you get with a VM.[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.

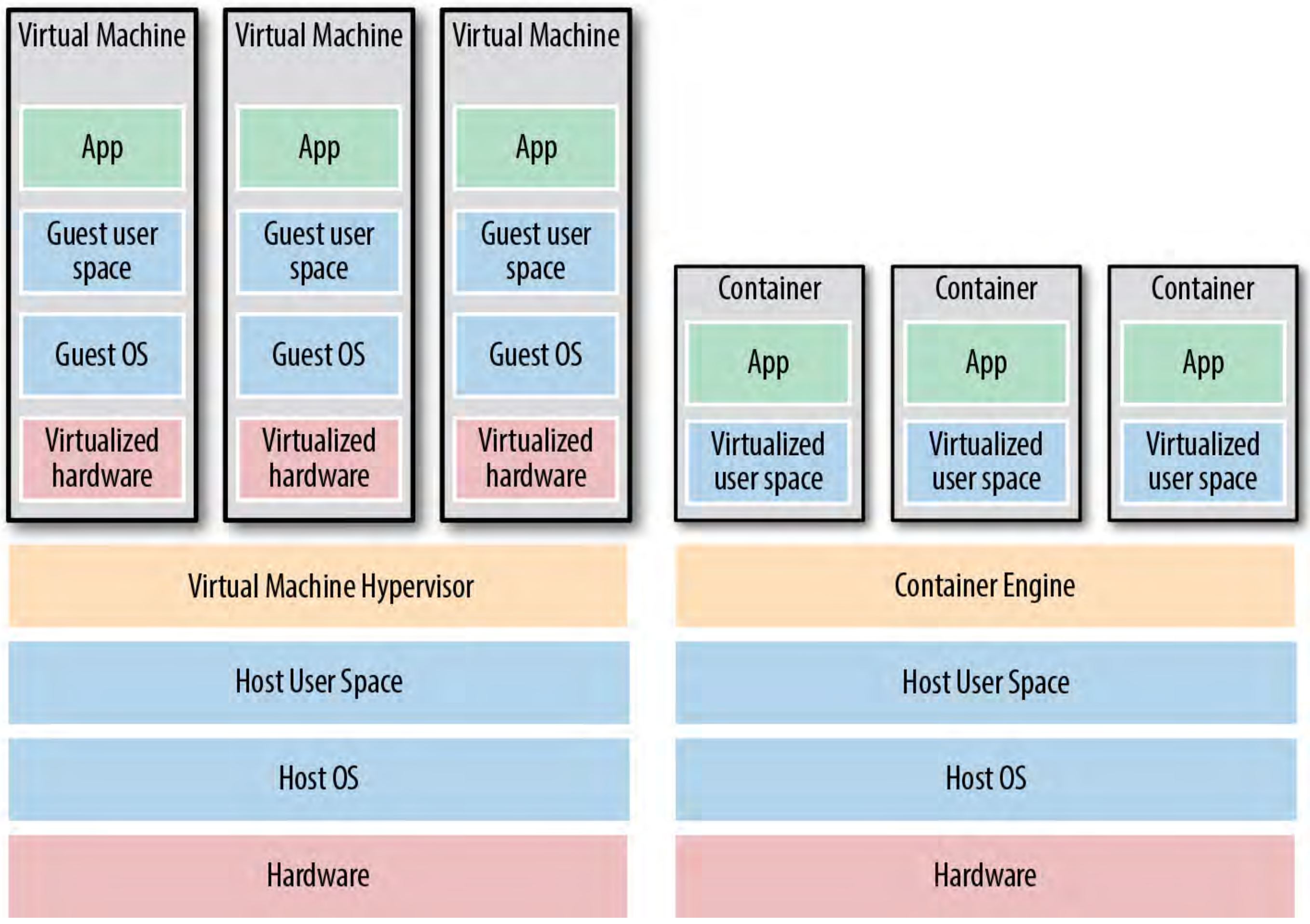


Figure 1-4. The two main types of images: VMs, on the left, and containers, on the right. VMs virtualize the hardware, whereas containers only virtualize user space.

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.

Orchestration Tools

Server templating tools are great for creating VMs and containers, but how do you actually manage them? For most real-world use cases, you'll need a way to:

1. Deploy VMs and containers, making efficient use of your hardware.
2. Roll out updates to an existing fleet of VMs and containers using strategies such as rolling deployment, blue-green deployment, and canary deployment.
3. Monitor the health of your VMs and containers and automatically replace unhealthy ones (auto healing).
4. Scale the number of VMs and containers up or down in response to load (auto scaling).
5. Distribute traffic across your VMs and containers (load balancing).
6. Allow your VMs and containers to find and talk to each other over the network (service discovery).

Handling these tasks is the realm of *orchestration tools* such as Kubernetes, Marathon/Mesos, Amazon ECS, Docker Swarm, and Nomad. For example, Kubernetes allows you to define how to manage your Docker containers as code. You first deploy a *Kubernetes cluster*, which is a group of servers that Kubernetes will manage and use to run your Docker containers. Most major cloud providers have native support for deploying managed Kubernetes clusters, such as EKS (AWS), GKE (Google Cloud), and AKS (Azure).

Once you have a working cluster, you can define how to run your Docker container as code in a YAML file:

```
apiVersion: apps/v1

# Use a Deployment to deploy multiple replicas of your Docker
# container(s) and to declaratively roll out updates to them
kind: Deployment

# Metadata about this Deployment, including its name
metadata:
  name: example-app

# The specification that configures this Deployment
spec:
  # This tells the Deployment how to find your container(s)
  selector:
    matchLabels:
      app: example-app

  # This tells the Deployment to run three replicas of your
  # Docker container(s)
  replicas: 3

  # Specifies how to update the Deployment. Here, we
  # configure a rolling update.
  strategy:
    rollingUpdate:
      maxSurge: 3
      maxUnavailable: 0
    type: RollingUpdate

  # This is the template for what container(s) to deploy
  template:

    # The metadata for these container(s), including labels
    metadata:
      labels:
        app: example-app

    # The specification for your container(s)
    spec:
      containers:

        # Run Apache listening on port 80
        - name: example-app
          image: httpd:2.4.39
          ports:
            - containerPort: 80
```

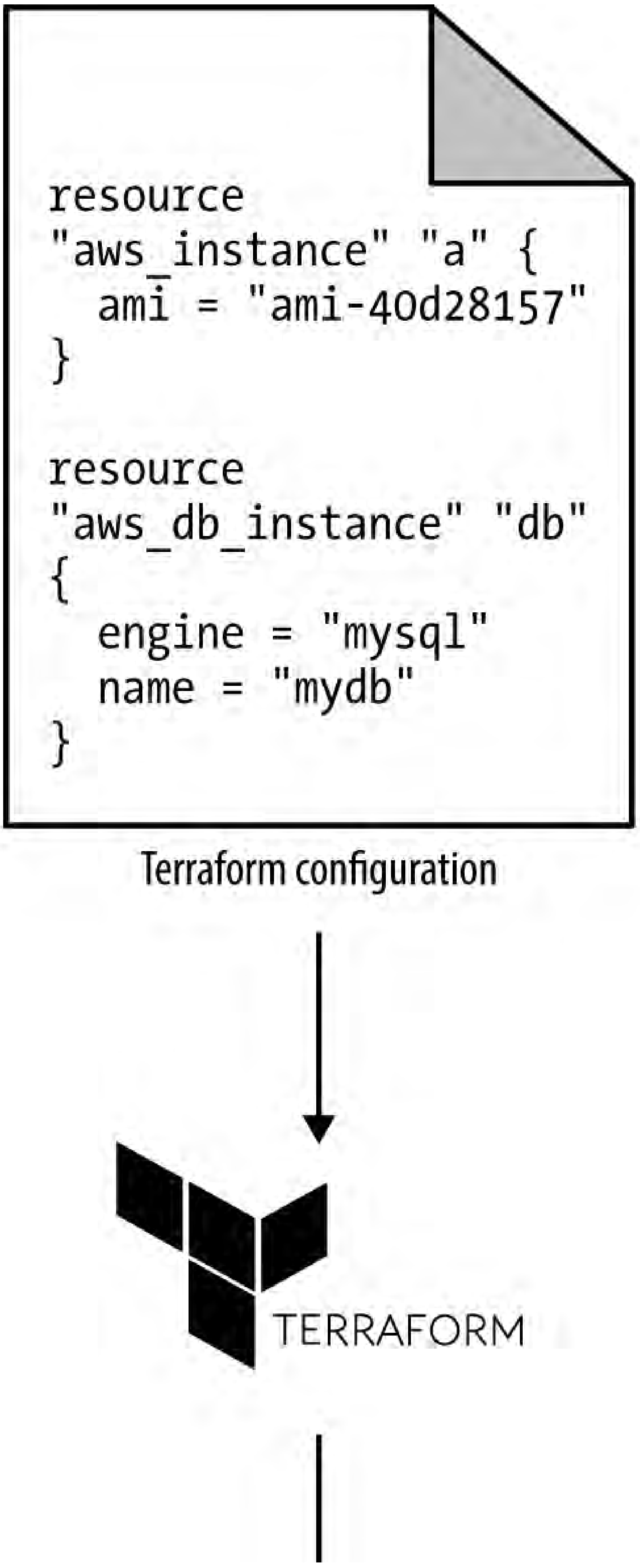
This file tells Kubernetes to create a *Deployment*, which is a declarative way to define:

1. One or more Docker containers to run together. This group of containers is called a *Pod*. The Pod defined above contains a single Docker container that runs Apache.
2. The settings for each Docker container in the Pod. The Pod above configures Apache to listen on port 80.
3. How many copies (AKA *replicas*) of the Pod to run in your cluster. The code above configures 3 replicas. Kubernetes will automatically figure out where in your cluster to deploy each Pod, using a scheduling algorithm to pick the optimal servers in terms of high availability (e.g., try to run each Pod on a separate server so a single server crash doesn't take down your app), resources (e.g., pick servers that have available the ports, CPU, memory, and other resources required by your containers), performance (e.g., try to pick servers with the least load and fewest container on them), and so on. Kubernetes will also constantly monitor the cluster to ensure there are always 3 replicas running, automatically replacing any Pods that crash or stop responding.
4. How to deploy updates. When deploying a new version of the Docker container, the code above will roll out 3 new replicas, wait for them to be healthy, and then undeploy the 3 old replicas.

That's a lot of power in just a few lines of YAML! You run `kubectl apply -f example-app.yml` to tell Kubernetes to deploy your app. You can then make changes to the YAML file and run `kubectl apply` again to roll out the updates.

Provisioning Tools

Whereas configuration management and server templating tools define the code that runs on each 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"
}
```

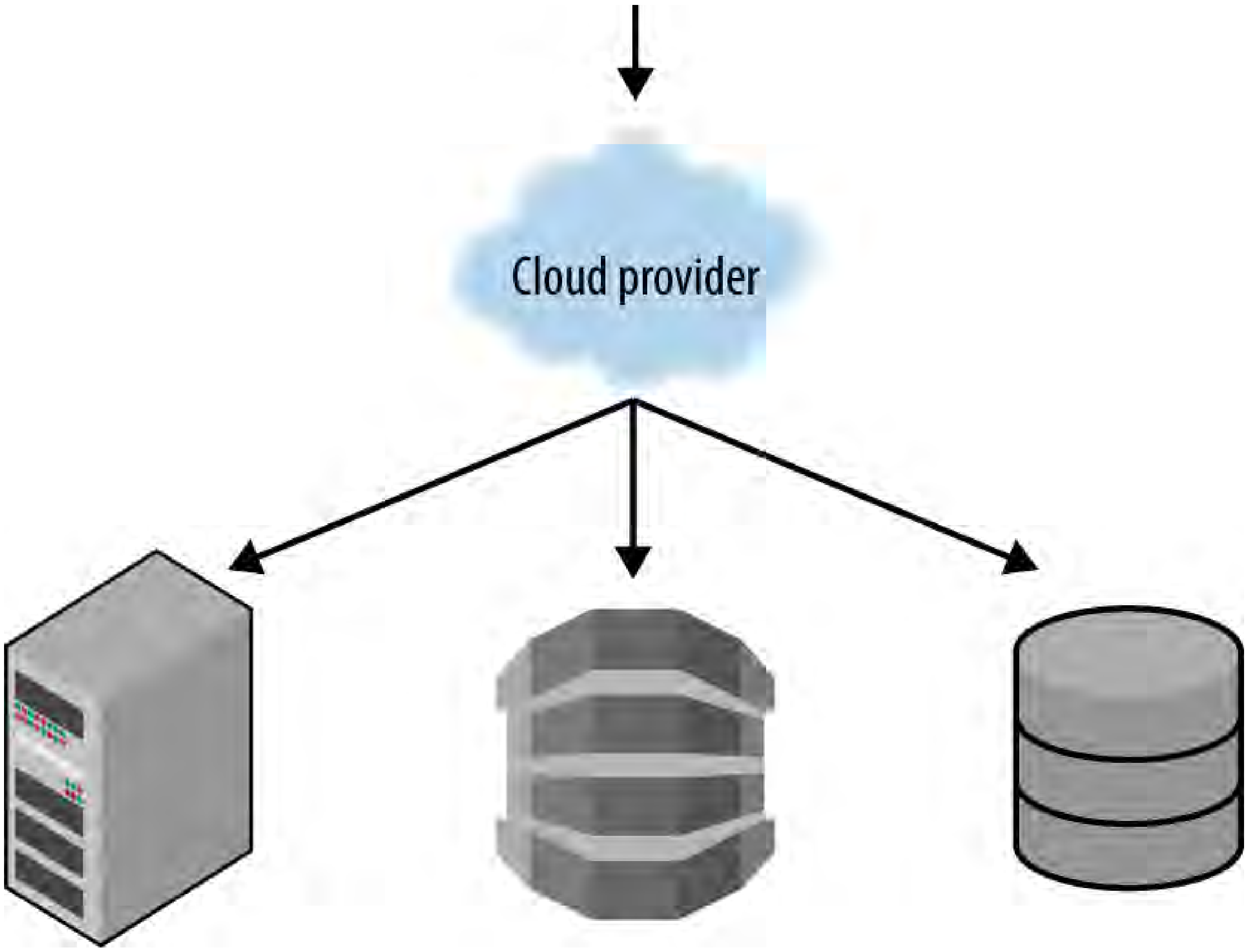


Figure 1-5. Provisioning tools can be used with your cloud provider to create servers, databases, load balancers, and all other parts of your infrastructure.

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 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 (which you’ll learn in Chapter 2), 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.

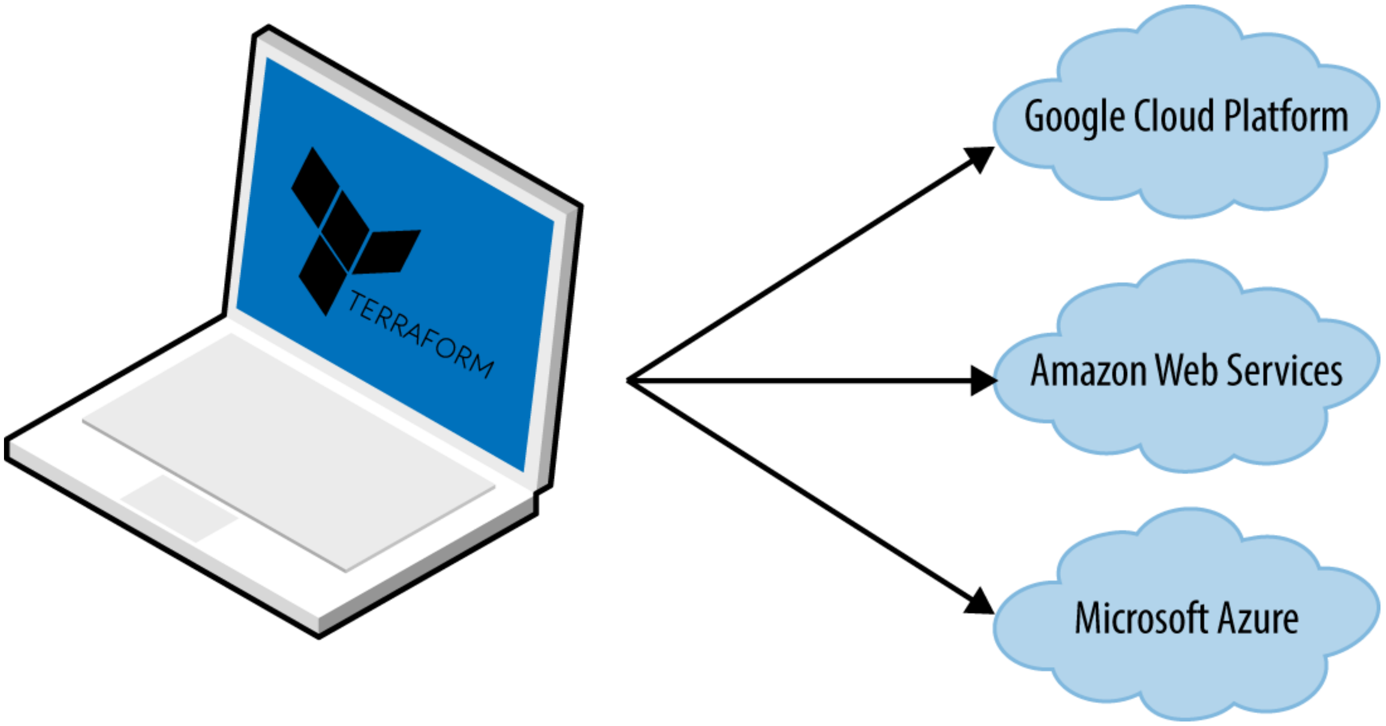


Figure 1-6. Terraform is a binary that translates the contents of your configurations into API calls to cloud providers

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 easy 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
- Using multiple tools together

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 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 terminate 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"
}
```

On 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 inChapter 5). Similarly, with limited 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 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.

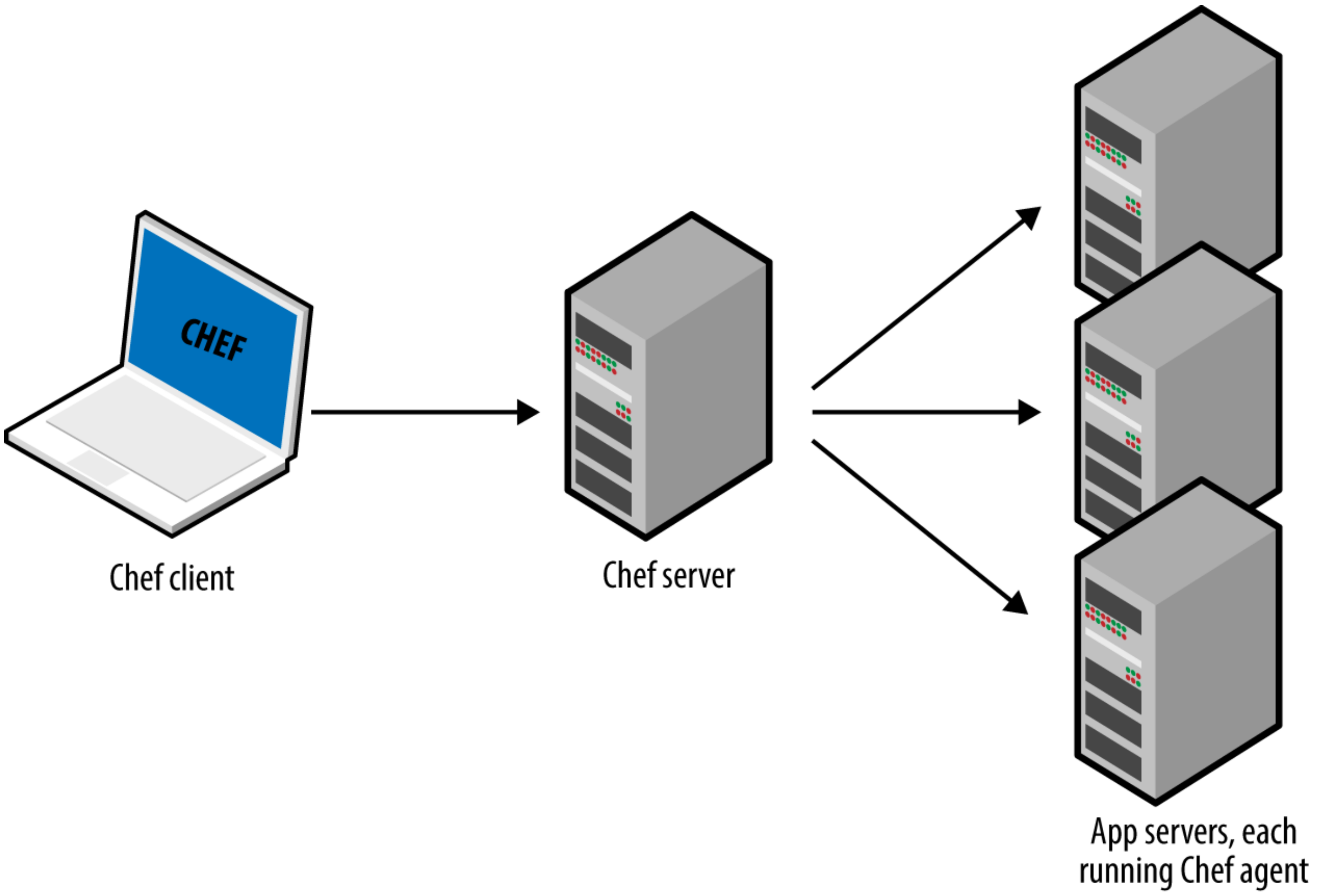


Figure 1-7. The typical architecture for Chef, Puppet, and SaltStack involves many moving parts. For example, the default setup for Chef is to run the Chef client on your computer, which talks to a Chef master server, which deploys changes by talking to Chef agents running on all your other servers.

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.

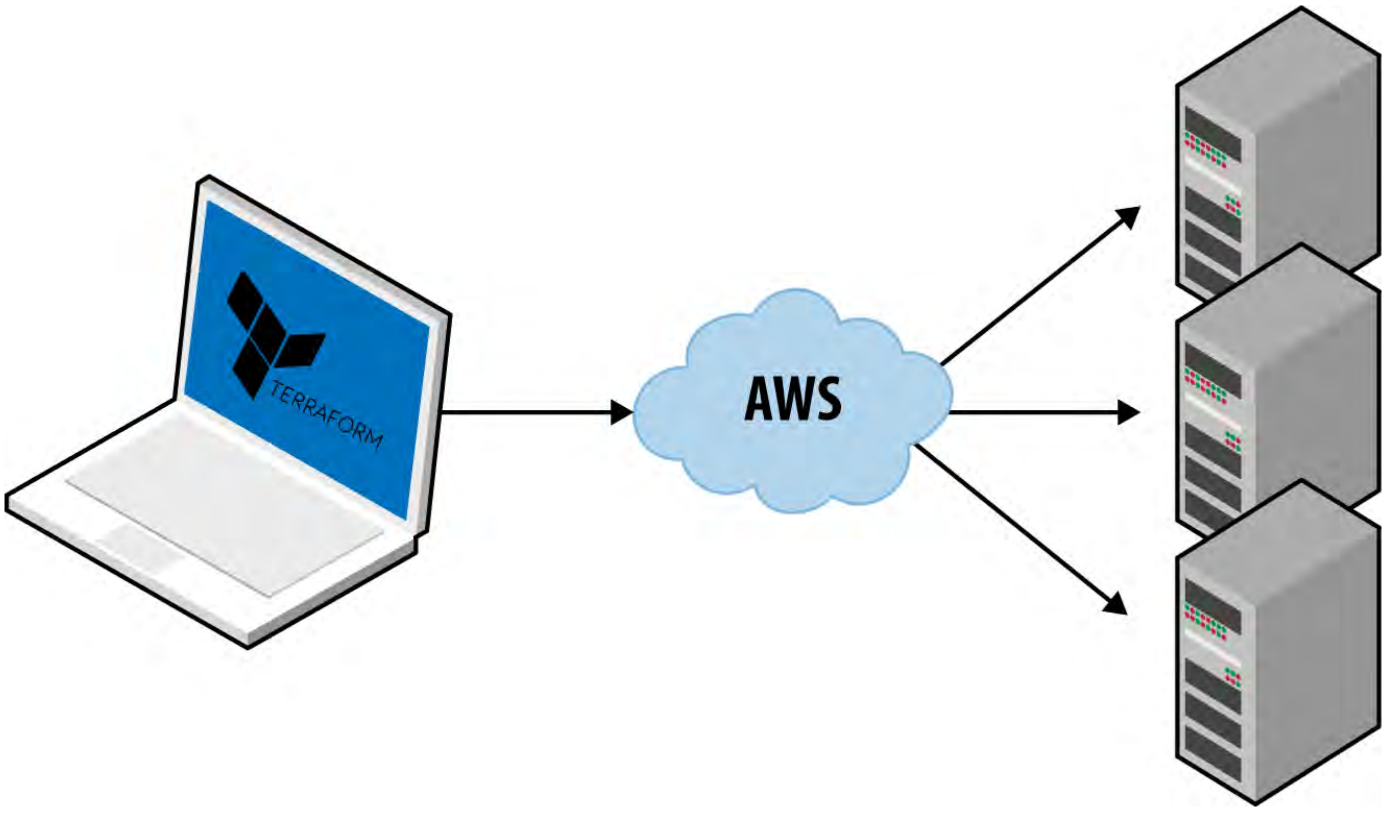


Figure 1-8. Terraform uses a masterless, agent-only architecture. All you need to run is the Terraform client and it takes care of the rest by using the APIs of cloud providers, such as AWS.

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)	Libraries	StackOverflow	Jobs
Chef	Open	All	562	5,794	435	86	3,832[a]	5,982	4,378[b]
Puppet	Open	All	515	5,299	94	314[c]	6,110[d]	3,585	4,200[e]
Ansible	Open	All	4,386	37,161	506	523	20,677[f]	11,746	8,787
SaltStack	Open	All	2,237	9,901	608	441	318[g]	1,062	1,622
CloudFormation	Closed	AWS	?	?	?	?	377[h]	3,315	2,318
Heat	Open	All	361	349	12	600[i]	0[j]	88	2,201[k]
Terraform	Open	All	1,261	16,837	173	204	1,462[l]	2,730	3,641

[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)	Libraries	StackOverflow	Jobs
Chef	Open	All	+18%	+31%	+139%	+48%	+26%	+43%	-22%
Puppet	Open	All	+19%	+27%	+19%	+42%	+38%	+36%	-19%
Ansible	Open	All	+195%	+97%	+49%	+66%	+157%	+223%	+125%
SaltStack	Open	All	+40%	+44%	+79%	+27%	+33%	+73%	+257%
CloudFormation	Closed	AWS	?	?	?	?	+57%	+441%	+249%
Heat	Open	All	+28%	+23%	-85%	+1,566%	0	+69%	+2,957%
Terraform	Open	All	+93%	+194%	-61%	-58%	+3,555%	+1,984%	+8,288%

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.

Using Multiple Tools Together

Although I've been comparing IAC tools this entire chapter, the reality is that you will likely need to use multiple tools to build your infrastructure. Each of the tools you've seen has strengths and weaknesses, so it's your job to pick the right tool for the right job.

Here are three common combinations I've seen work well at a number of companies:

Provisioning plus configuration management

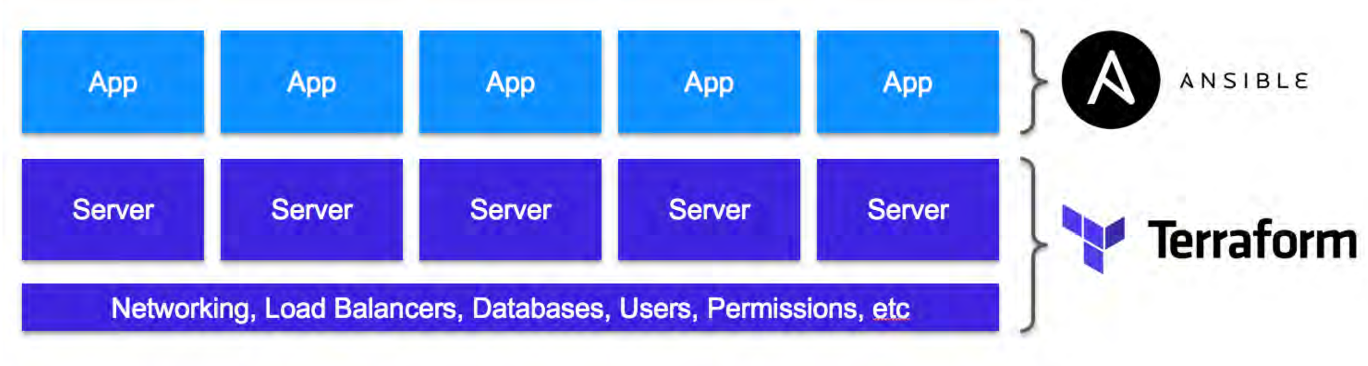

App
App
App
App
App
ANSIBLE
Server
Server
Server
Server
Server
Terraform
Networking, Load Balancers, Databases, Users, Permissions, etc

Figure 1-9. Using Terraform and Ansible together

Example: Terraform and Ansible. You use Terraform to deploy all the underlying infrastructure, including the network topology (i.e., VPCs, subnets, route tables), data stores (e.g., MySQL, Redis), load balancers, and servers. You then use Ansible to deploy your apps on top of those servers. See Figure 1-9.

This is an easy approach to start with, as there is no extra infrastructure to run (Terraform and Ansible are both client-only applications) and there are many ways to get Ansible and Terraform to work together (e.g., Terraform adds special tags to your servers and Ansible uses those tags to find the server and configure them). The major downside is that using Ansible typically means you're writing a lot of procedural code, with mutable servers, so as your code base, infrastructure, and team grow, maintenance may become more difficult.

Provisioning plus server templating

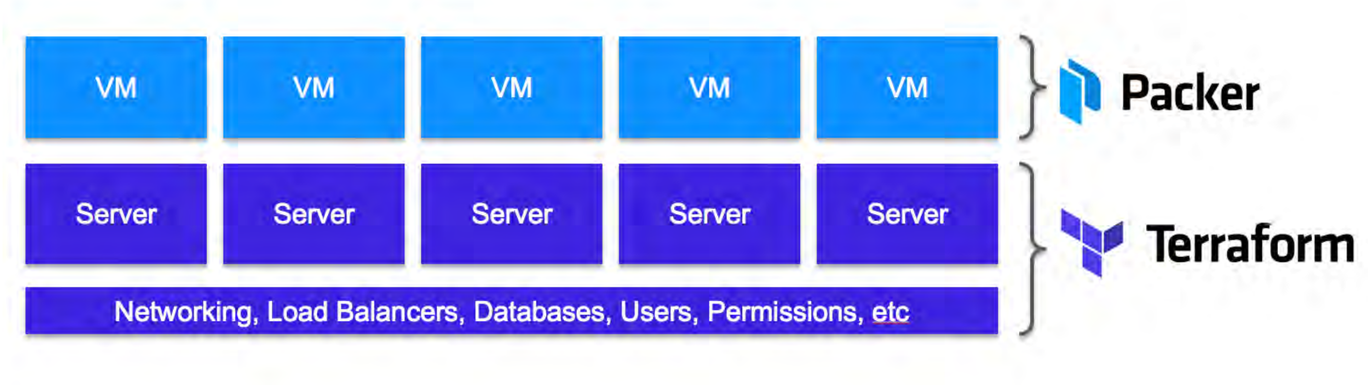

VM
VM
VM
VM
VM
Packer
Server
Server
Server
Server
Server
Terraform
Networking, Load Balancers, Databases, Users, Permissions, etc

Figure 1-10. Using Terraform and Packer together

Example: Terraform and Packer. You use Packer to package your apps as virtual machine images. You then use Terraform to deploy (a) servers with these virtual machine images and (b) the rest of your infrastructure, including the network topology (i.e., VPCs, subnets, route tables), data stores (e.g., MySQL, Redis), and load balancers. See Figure 1-10.

This is also an easy approach to start with, as there is no extra infrastructure to run (Terraform and Packer are both client-only applications) and you'll get plenty of practice deploying virtual machine images using Terraform later in this book. Moreover, this is an immutable infrastructure approach, which will make maintenance easier. However, there are two major drawbacks. First, virtual machines can take a long time to build and deploy, which will slow down your iteration speed. Second, as you'll see in later chapters, the deployment strategies you can implement with Terraform are limited (e.g., you can't implement blue-green deployment natively in Terraform), so you either end up writing lots of complicated deployment scripts, or you turn to orchestration tools, as described next.

Provisioning plus server templating plus orchestration

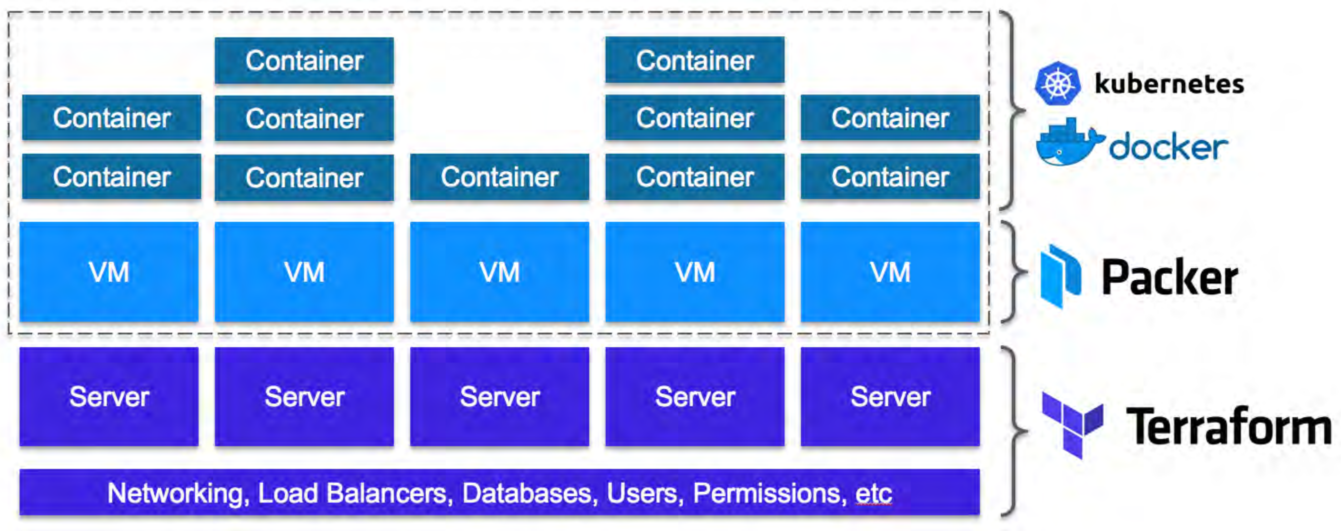


Figure 1-11. Using Terraform, Packer, Docker, and Kubernetes together

Example: Terraform, Packer, Docker, and Kubernetes. You use Packer to create a virtual machine image that has Docker and Kubernetes installed. You then use Terraform to deploy (a) a cluster of servers, each of which runs this virtual machine image and (b) the rest of your infrastructure, including the network topology (i.e., VPCs, subnets, route tables), data stores (e.g., MySQL, Redis), and load balancers. Finally, when the cluster of servers boots up, it forms a Kubernetes cluster that you use to run and manage your Dockerized applications. See Figure 1-11.

The advantage of this approach is that Docker images build fairly quickly, you can run and test them on your local computer, and you can take advantage of all the built-in functionality of Kubernetes, including various deployment strategies, auto healing, auto scaling, and so on. The drawback is the added complexity, both in terms of extra infrastructure to run (Kubernetes clusters are difficult and expensive to deploy and operate, though most major cloud providers now provide managed Kubernetes services, which can offload some of this work), and in terms of several extra layers of abstraction (Kubernetes, Docker, Packer) to learn, manage, and debug.

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**	**Agent**	**Master**	**Community**	**Maturity**
Chef	Open	All	Config Mgmt	Mutable	Procedural	Yes	Yes	Large	High
Puppet	Open	All	Config Mgmt	Mutable	Declarative	Yes	Yes	Large	High
Ansible	Open	All	Config Mgmt	Mutable	Procedural	No	No	Huge	Medium
SaltStack	Open	All	Config Mgmt	Mutable	Declarative	Yes	Yes	Large	Medium
CloudFormation	Closed	AWS	Provisioning	Immutable	Declarative	No	No	Small	Medium
Heat	Open	All	Provisioning	Immutable	Declarative	No	No	Small	Low
Terraform	Open	All	Provisioning	Immutable	Declarative	No	No	Huge	Low

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, Packer, and Kubernetes 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 is less clear cut these days, as some of the major configuration management tools have gradually improved their 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 40 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. 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 for the first year 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

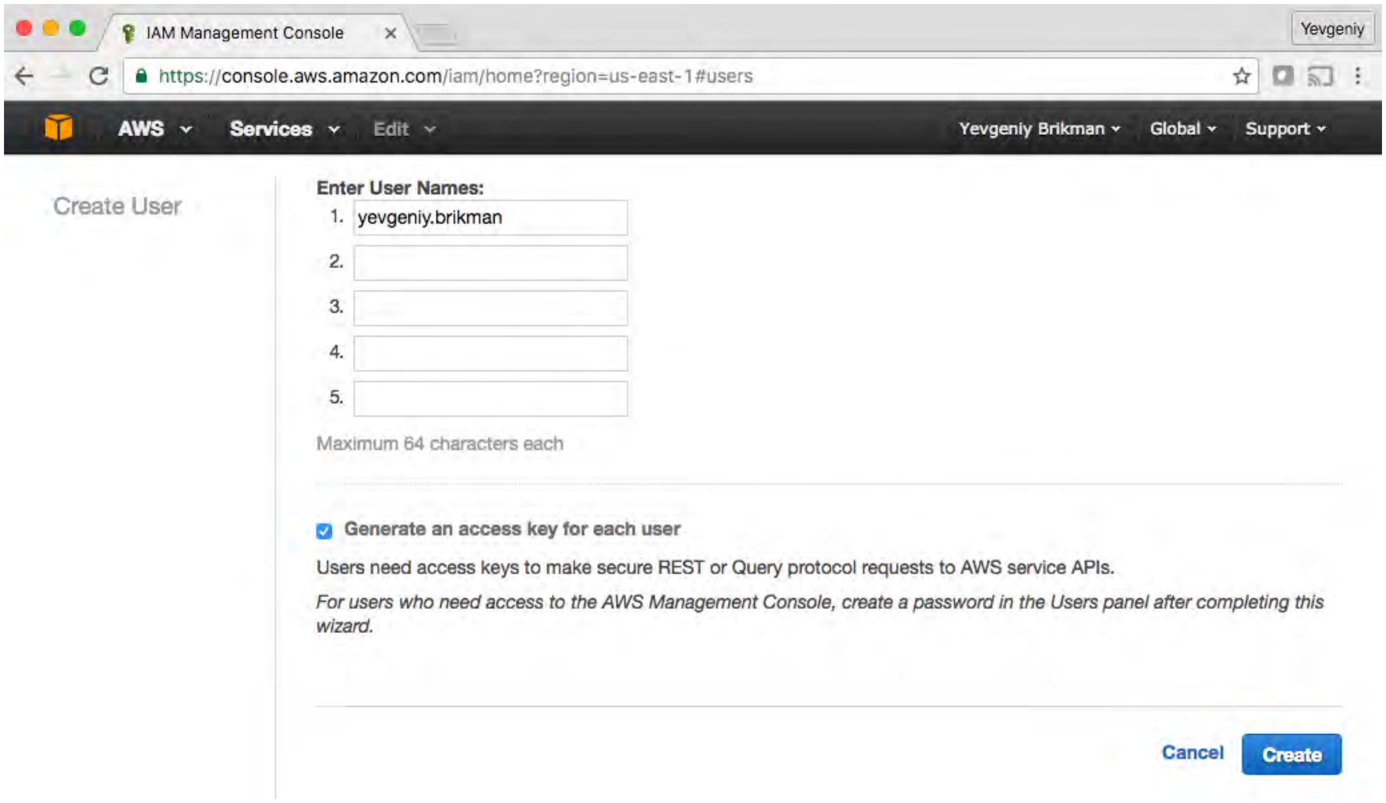
EXAMPLE CODE

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 "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 (note, AWS has been making design changes to its web console, so the IAM pages may look slightly different when you are reading this book).



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

Maximum 64 characters each

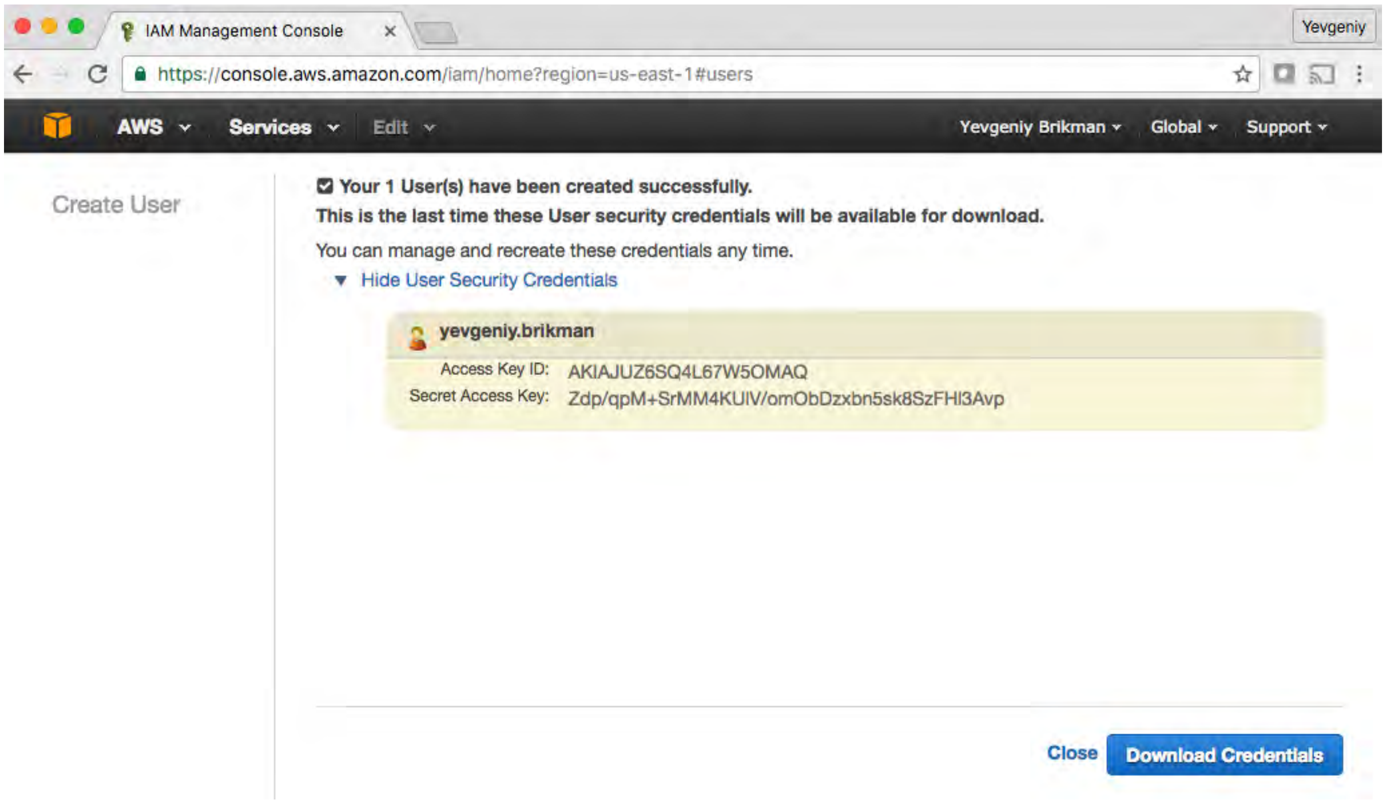
☑ Generate an access key for each user

Users need access keys to make secure REST or Query protocol requests to AWS service APIs.

For users who need access to the AWS Management Console, create a password in the Users panel after completing this wizard.

Figure 2-1. Create a new IAM user

Click the "Create" button and AWS will show you the security credentials for that user, which consist of an *Access Key ID* and a *Secret Access Key*, as shown in Figure 2-2. You must save these immediately, as they will never be shown again, and you'll need them later on in this tutorial. Remember that these credentials give access to your AWS account, so store them somewhere secure (e.g., a password manager such as 1Password, LastPass, or OS X Keychain) and never share them with anyone.



☑ Your 1 User(s) have been created successfully.
This is the last time these User security credentials will be available for download.

You can manage and recreate these credentials any time.

▼ Hide User Security Credentials

yevgeniy.brikman

Access Key ID: AKIAJUZ6SQ4L67W5OMAQ
Secret Access Key: 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

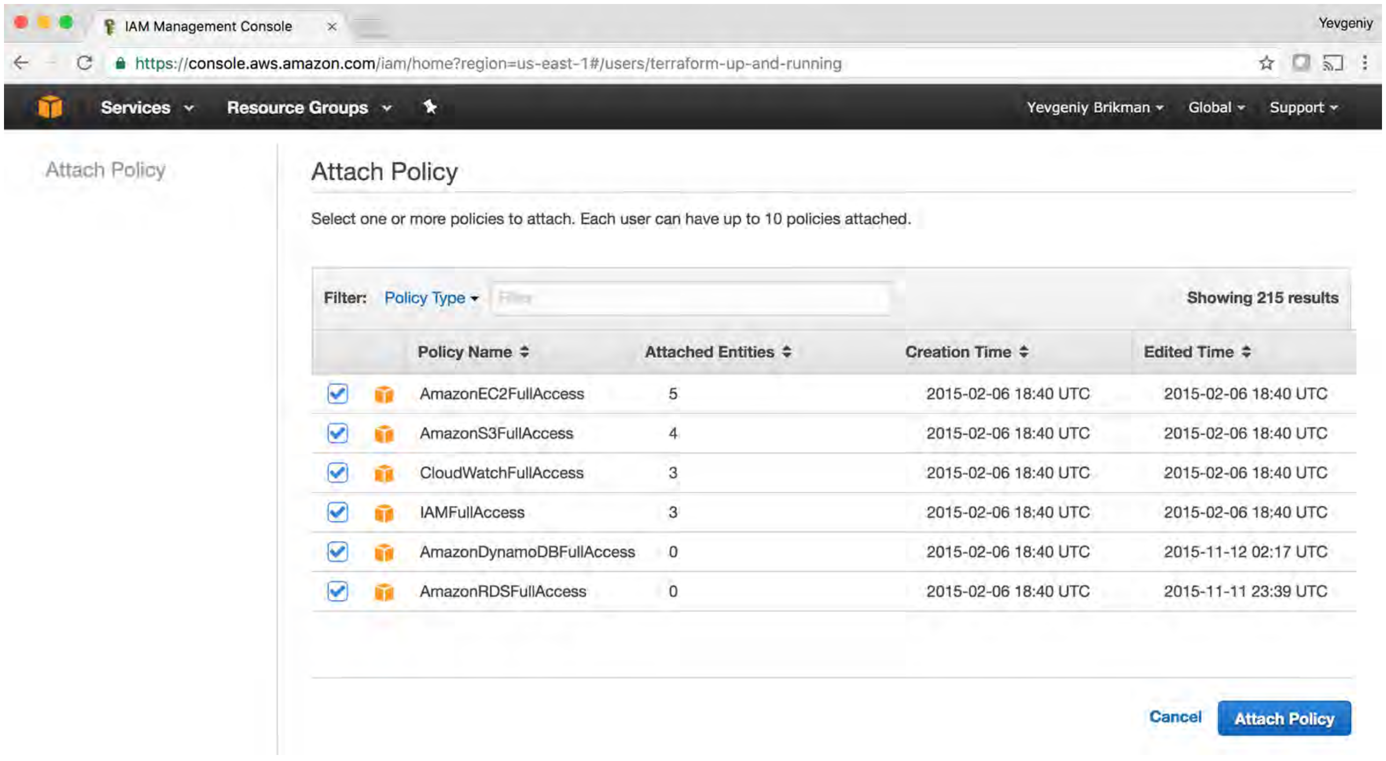
Figure 2-2. Store your AWS credentials somewhere secure. Never share them with anyone. Don't worry, the ones in the screenshot are fake.

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

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

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

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



Attach Policy

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

Filter: Policy Type — Showing 215 results

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

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

A NOTE ON DEFAULT VPCS

Please note that if you are using an existing AWS account, it must have a *Default VPC* in it. A *VPC*, or Virtual Private Cloud, is an isolated area of your AWS account that has its own virtual network and IP address space. Just about every AWS resource deploys into a VPC. If you don't explicitly specify a VPC, the resource will be deployed into the *Default VPC*, which is part of every new AWS account. All the examples in this book rely on this Default VPC, so if for some reason you deleted the one in your account, either use a different region (each region has its own Default VPC) or 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. Alternatively, Terraform may be available in your operating system's package manager, such as `brew install terraform` on OS X.

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. The general syntax for create a resource in Terraform is:

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

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

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

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

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

ami

The Amazon Machine Image (AMI) to run on the EC2 Instance. You can find free and paid AMIs in the AWS Marketplace or create your own using tools such as Packer (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 Terraform, you need to run `terraform init` to tell Terraform to scan the code, figure out what providers you're using, and download the code for them. By default, the provider code will be downloaded into a `.terraform` folder, which is Terraform's scratch directory (you may want to add it to `.gitignore`). You'll see a few other uses for the `init`command and `.terraform` folder in later chapters. For now, just be aware that you need to run `init` any time you start with new Terraform code, and that it's safe to run `init` multiple times (the command is idempotent).

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

```
$ terraform plan

(...)
```

```
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:

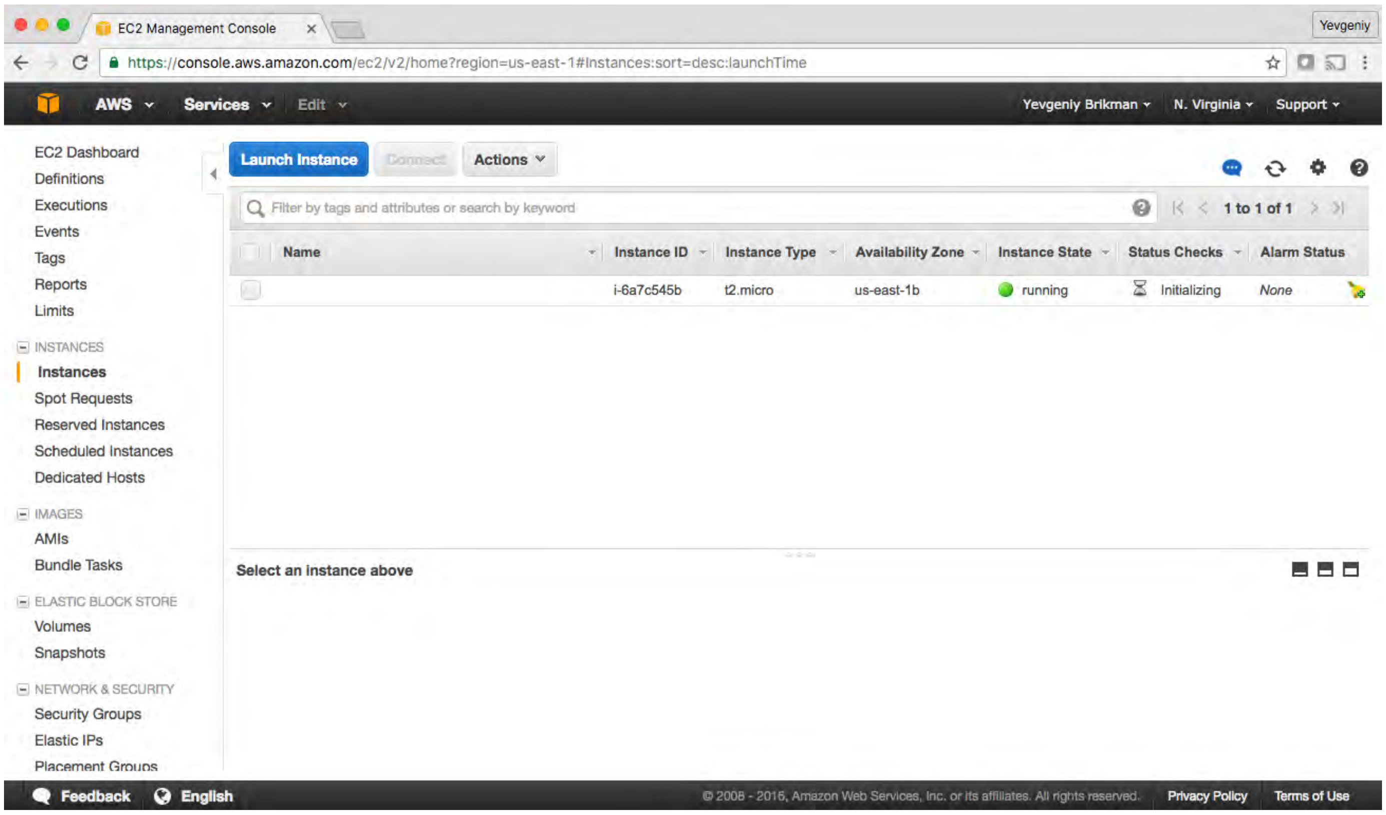
```
$ terraform apply

(...)

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 (a topic you'll see more of in Chapter 8), 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 an EC2 Instance in your AWS account using Terraform! To verify this, head over to the EC2 console, and you should see something similar to Figure 2-4.



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

Figure 2-4. A single EC2 Instance

Sure enough the Instance is there, though admittedly, this isn't the most exciting example. Let's make it a bit more interesting. First, notice that the EC2 Instance doesn't have a name. To add one, you can add `tags` to the `aws_instance`resource:

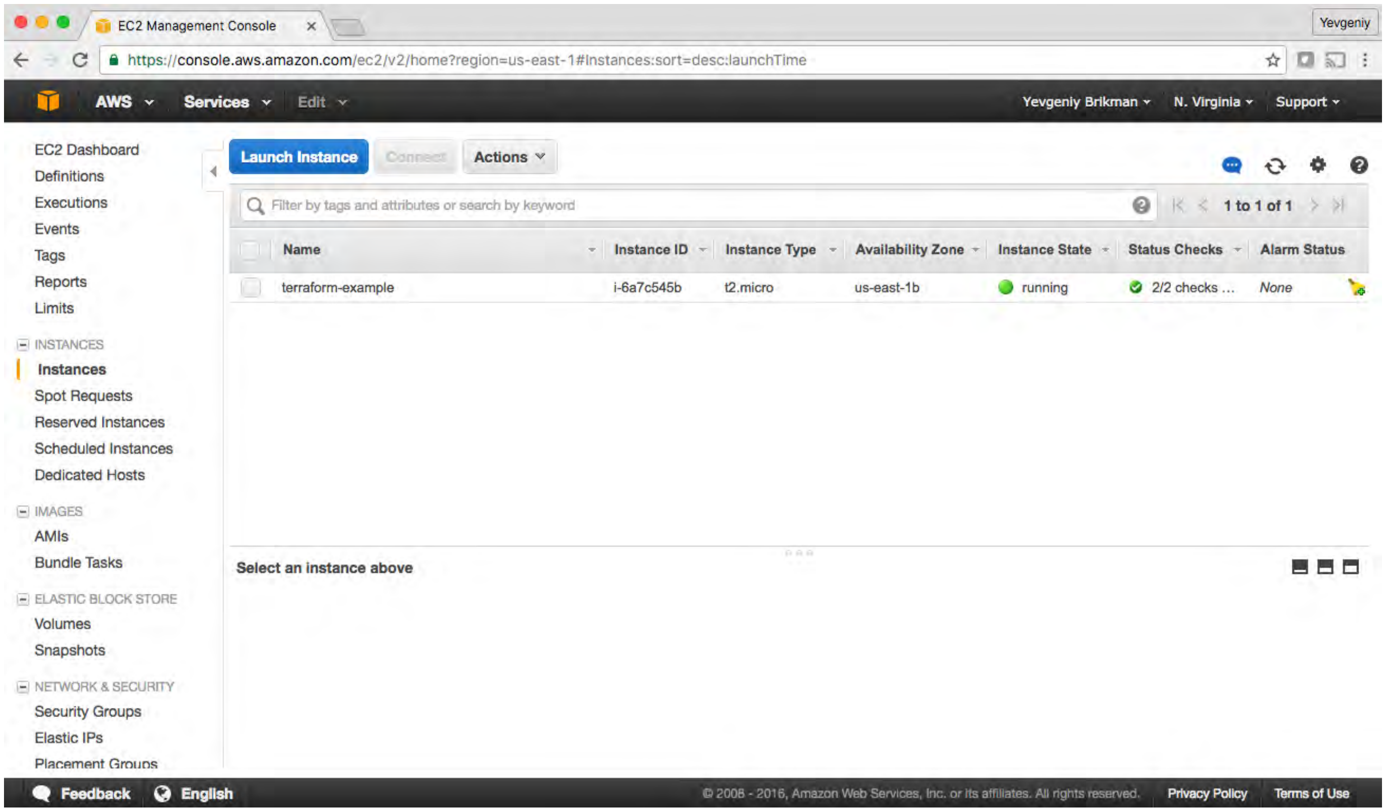
```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  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 `apply` command), and it can show you a diff between what's currently deployed and what's in your Terraform code (this is one of the advantages of using a declarative language over a procedural one, as discussed in "How 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.



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

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

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

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

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

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

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

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

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

```
git remote add origin git@github.com:<YOUR_USERNAME>/<YOUR_REPO_NAME>.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. You'll learn more about using Terraform as a team in Chapter 8.

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.

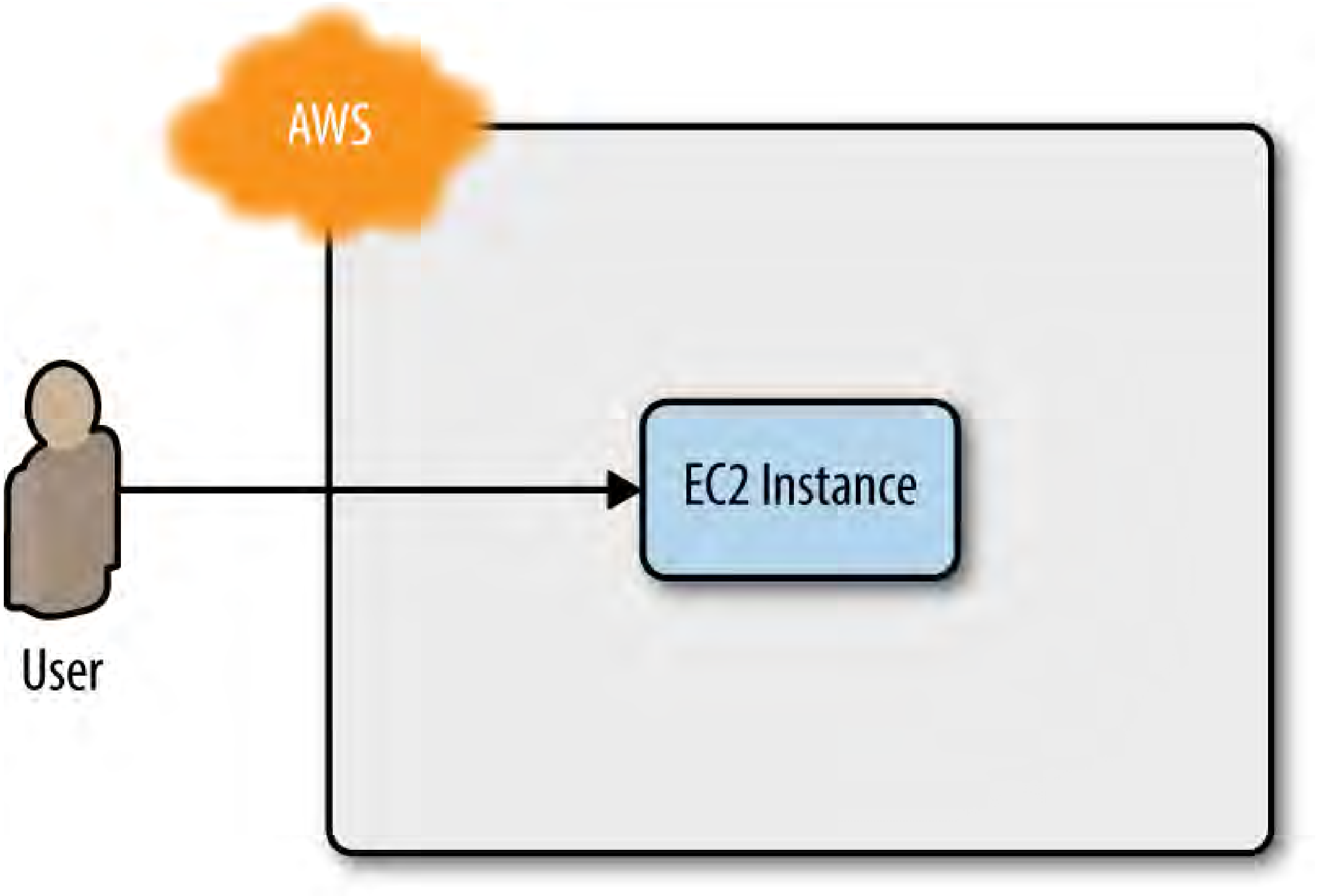


Figure 2-6. Start with a simple architecture: a single web server running in AWS that responds to HTTP requests

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. When you launch an EC2 Instance, you have the option of passing either a shell script or cloud-init directive to User Data, and the EC2 Instance will execute it during boot. Pass a shell script to User Data by setting the `user_data`argument in your Terraform code as follows:

```
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` argument of the `aws_instance` resource. To do that, you first need to learn about Terraform *expressions*.

An expression in Terraform is anything that returns a value. You've already seen the simplest type of expressions, *literals*, such as strings (e.g., `"ami-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 argument 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, if you were deploying this code from scratch, Terraform would know that 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.

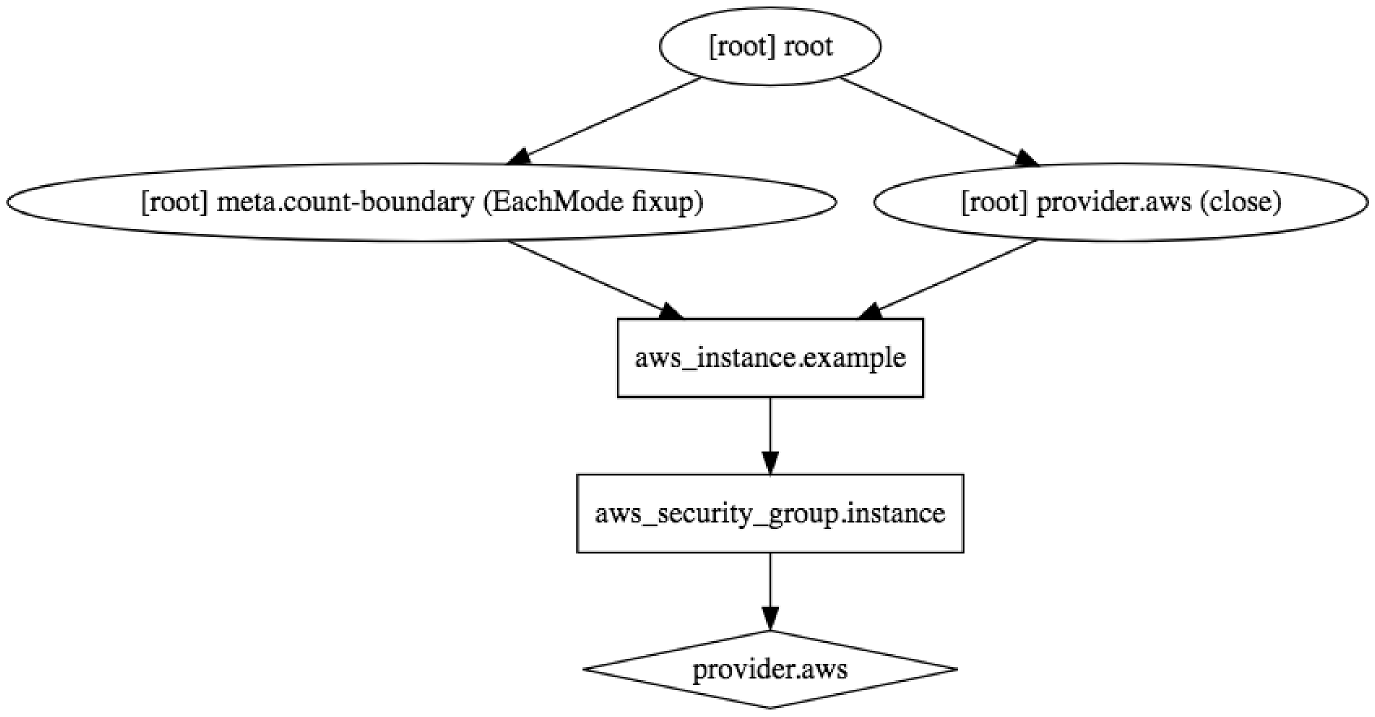

[root] root
[root] meta.count-boundary (EachMode fixup)
[root] provider.aws (close)
aws_instance.example
aws_security_group.instance
provider.aws

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

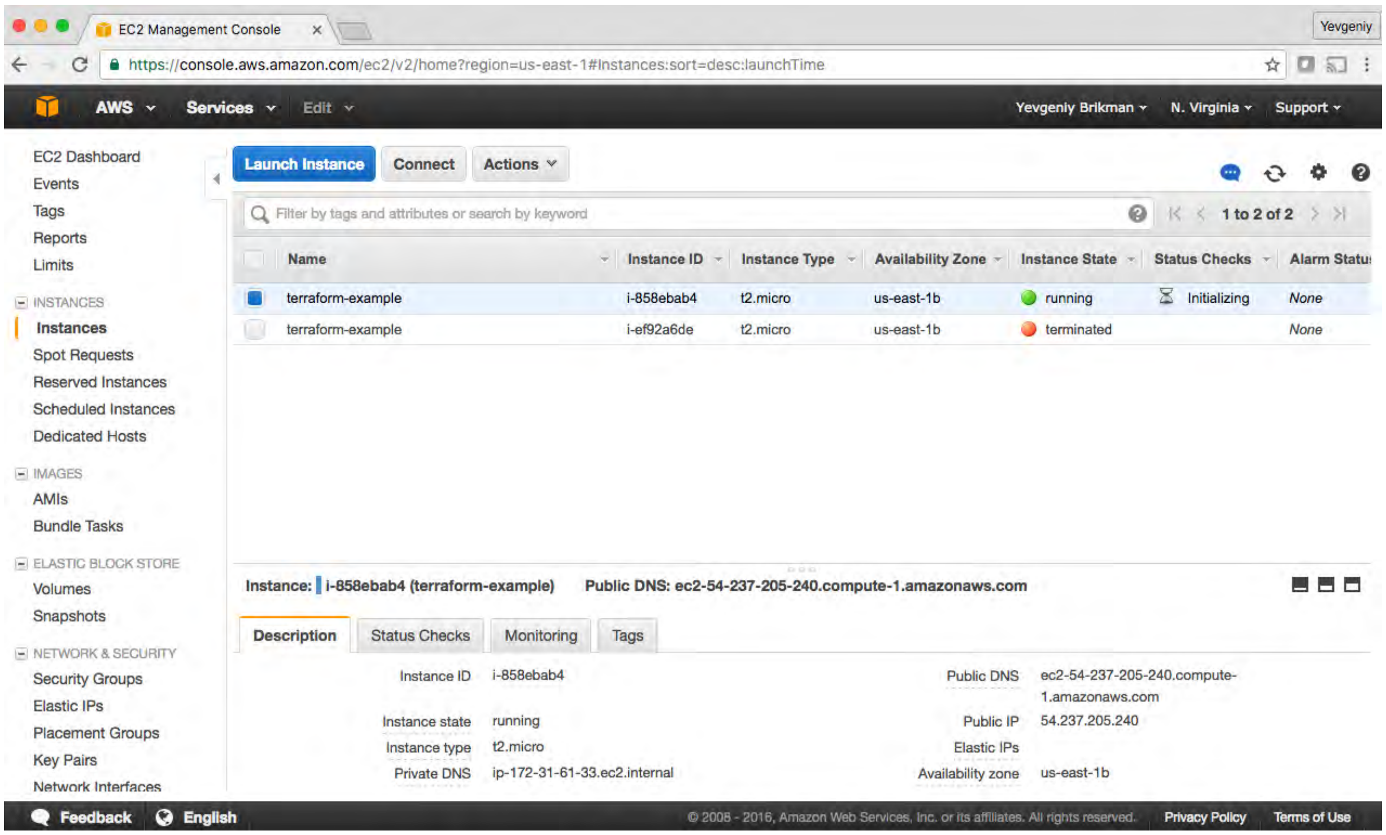
When Terraform walks your dependency tree, it will create as many resources in parallel as it can, which means it can apply your changes fairly efficiently. That's the beauty of a declarative language: you just specify what you want and Terraform figures out the most efficient way to make it happen.

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" in the plan output to figure out what is forcing Terraform to do a replacement. Many of the arguments on the `aws_instance` resource will force a replacement if changed, which means the original EC2 Instance will be terminated and a completely new Instance will 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.



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

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

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

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

If you click on your new Instance, you can find its public IP address in the description panel at the bottom of the screen. Give the Instance 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 that you lock down as much as possible (you'll see an example of how to deploy a load balancer later in this chapter).

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 *output variables* with the following syntax:

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

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

`description`

It’s always a good idea to use this parameter to document what type of data is contained in the output variable.

`sensitive`

Set this parameter to `true` to tell Terraform not to log this output at the end of `terraform apply`. This is useful if the output variable contains sensitive material or secrets, such as passwords or private keys.

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

```
output "public_ip" {
  value       = aws_instance.example.public_ip
  description = "The public IP 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 deployment script that runs `terraform apply` to deploy the web server, uses `terraform output public_ip` to grab its public IP, and runs `curl` on the IP as a quick smoke test to validate that the deployment worked.

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

Deploy a Cluster of Web Servers

Running a single server is a good start, but in the real world, a single server is a single point of failure. If that server crashes, or if it becomes overloaded from too much traffic, users will be unable to access your site. The solution is to run a cluster of servers, routing around servers that go down, and adjusting the size of the cluster up or down based on traffic.[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.

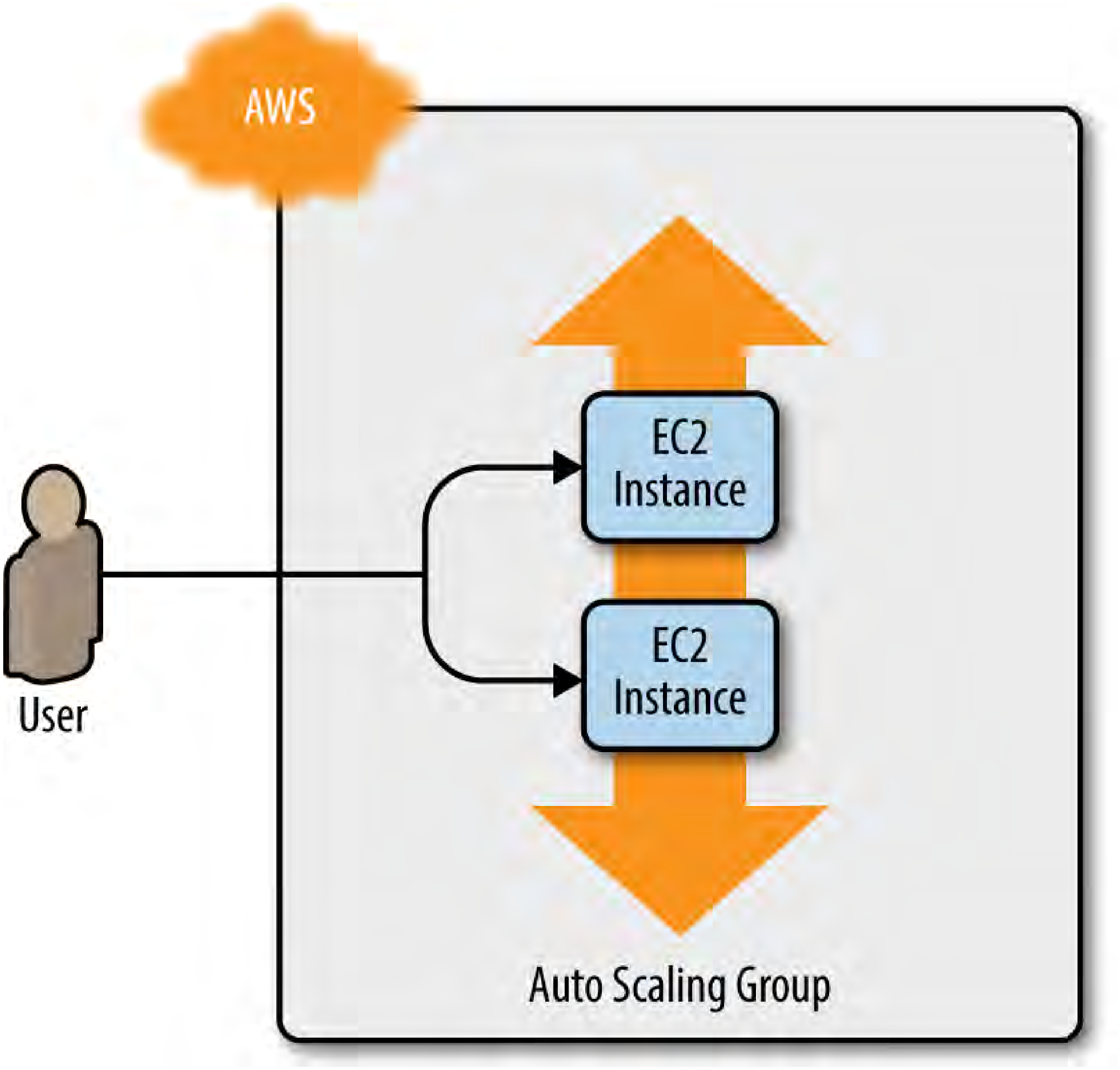


Figure 2-9. Instead of a single web server, run a cluster of web servers using an Auto Scaling Group

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 (two of the parameters have different names: `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”. Note that the ASG uses a reference to fill in the launch configuration name. This leads to a problem: launch configurations are immutable, so if you change any parameter of your launch configuration, Terraform will try to replace it. Normally, when replacing a resource, Terraform deletes the old resource first and then creates its replacement, but since your ASG now has a reference to the old resource, Terraform won’t be able to delete it.

To solve this problem, you can use a *lifecycle* setting. Every Terraform resource supports several lifecycle settings that configure how that resource is created, updated, and/or deleted. A particularly useful lifecycle setting is `create_before_destroy`. If you set `create_before_destroy` to `true`, Terraform will invert the order in which it replaces resources, creating the replacement resource first (including updating any references that were pointing at the old resource to point to the replacement) and then deleting the old resource. Add the `lifecycle` block to your `aws_launch_configuration` as follows:

```
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

  # Required when using a launch configuration with an auto scaling group.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

There’s also one other parameter you need to add to your ASG to make it work: `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 ...]
}
```

where `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.

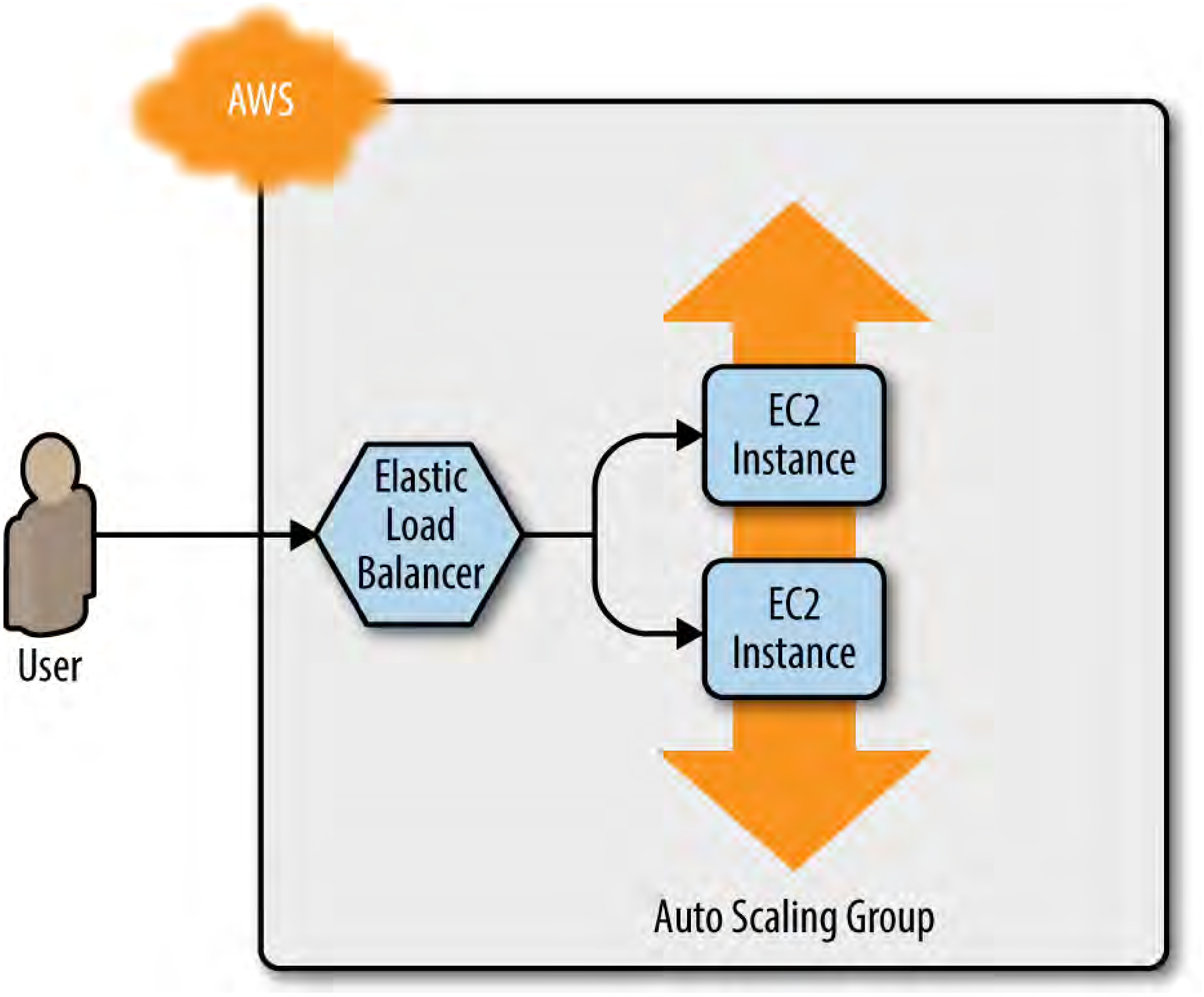


Figure 2-10. Use an Elastic Load Balancer to distribute traffic across the Auto Scaling Group

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, UDP, and TLS traffic. Can scale up and down in response to load faster than the ALB (the NLB is designed to scale to tens of millions of requests per second). 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.

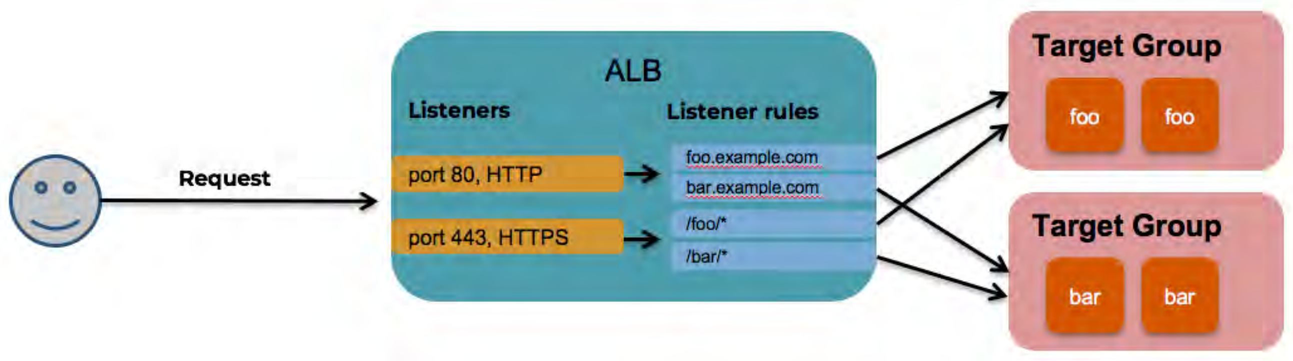


Figure 2-11. Application Load Balancer (ALB) overview

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] 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. This security group should allow incoming requests on port 80 so you can access the load balancer over HTTP and outgoing requests on all ports, so the load balancer can perform health checks:

```
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:

```
resource "aws_lb_target_group" "asg" {
  name     = "terraform-asg-example"
  port     = var.server_port
  protocol = "HTTP"
  vpc_id   = data.aws_vpc.default.id

  health_check {
    path                = "/"
    protocol            = "HTTP"
    matcher             = "200"
    interval            = 15
    timeout             = 3
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
}
```

Note that this target group will health check your Instances by periodically sending an HTTP request to each Instance and only considering the Instance “healthy” if the Instance returns a response that matches the configured `matcher`(e.g., you can configure a matcher to look for a 200 OK response). If an Instance fails to respond, perhaps because that Instance 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
  }
}
```

You should also update the `health_check_type` to `"ELB"`. The default `health_check_type` is `"EC2"`, which is a minimal health check that only a considers Instance unhealthy if the AWS hypervisor says the VM is completely down or unreachable. The `"ELB"` health check is 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.

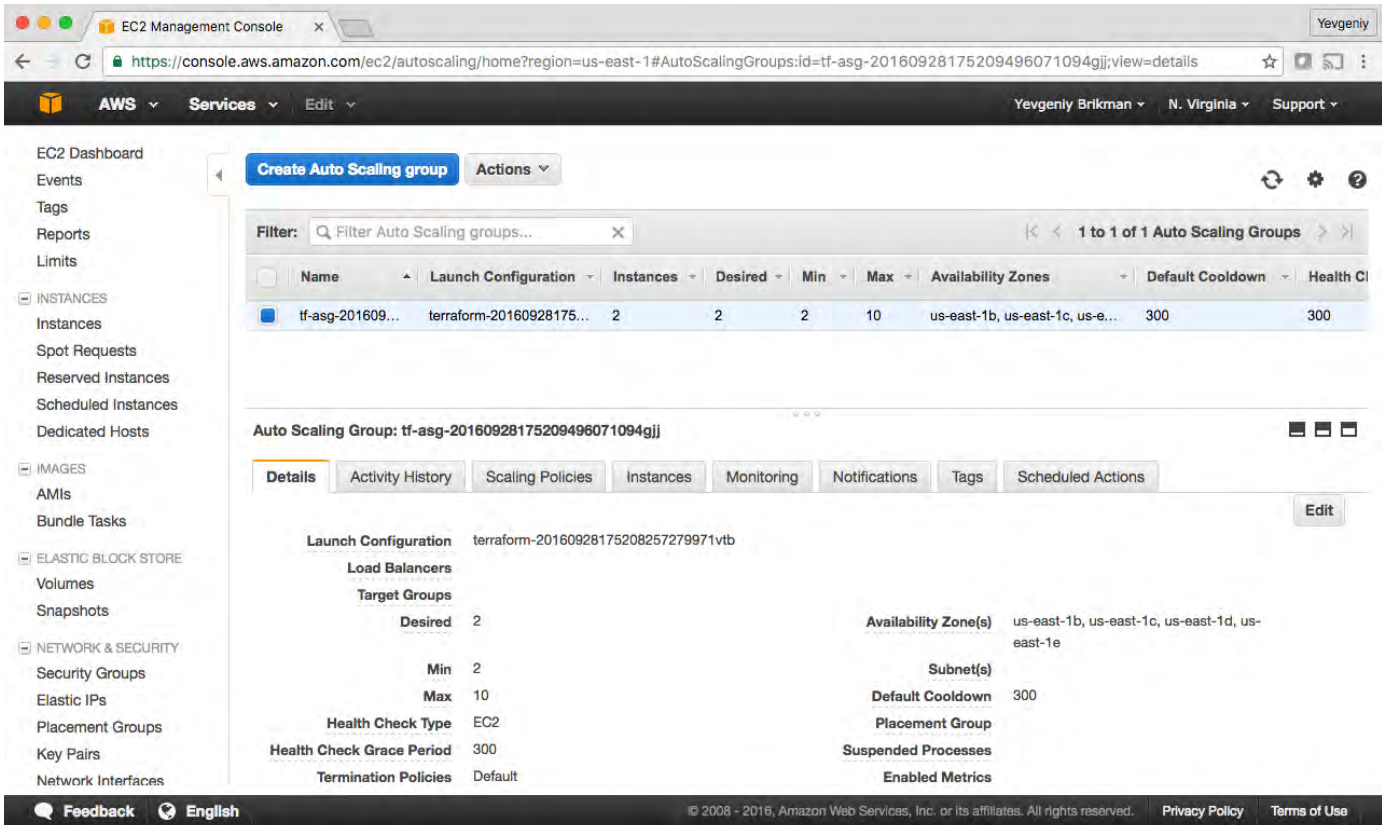
One last thing to do before deploying the load balancer: replace the old `public_ip` output of the single EC2 Instance you had before with an output that shows the DNS name of the ALB:

```
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.



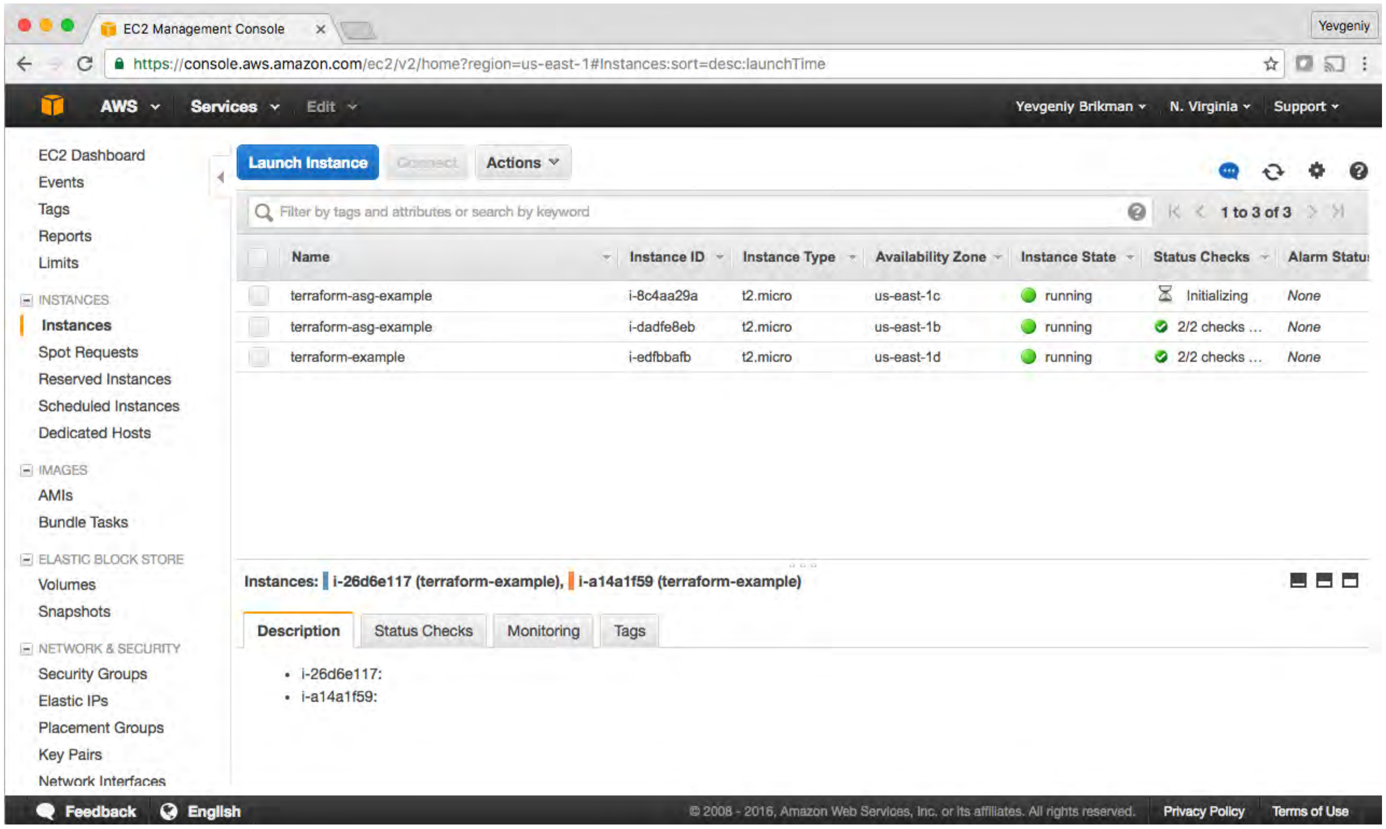
Name	Launch Configuration	Instances	Desired	Min	Max	Availability Zones	Default Cooldown	Health Cl
tf-asg-201609...	terraform-20160928175...	2	2	2	10	us-east-1b, us-east-1c, us-e...	300	300

Auto Scaling Group: tf-asg-20160928175209496071094gjj

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

Figure 2-12. The Auto Scaling Group

If you switch over to the Instances tab, you'll see the two EC2 Instances launching, as shown in Figure 2-13.



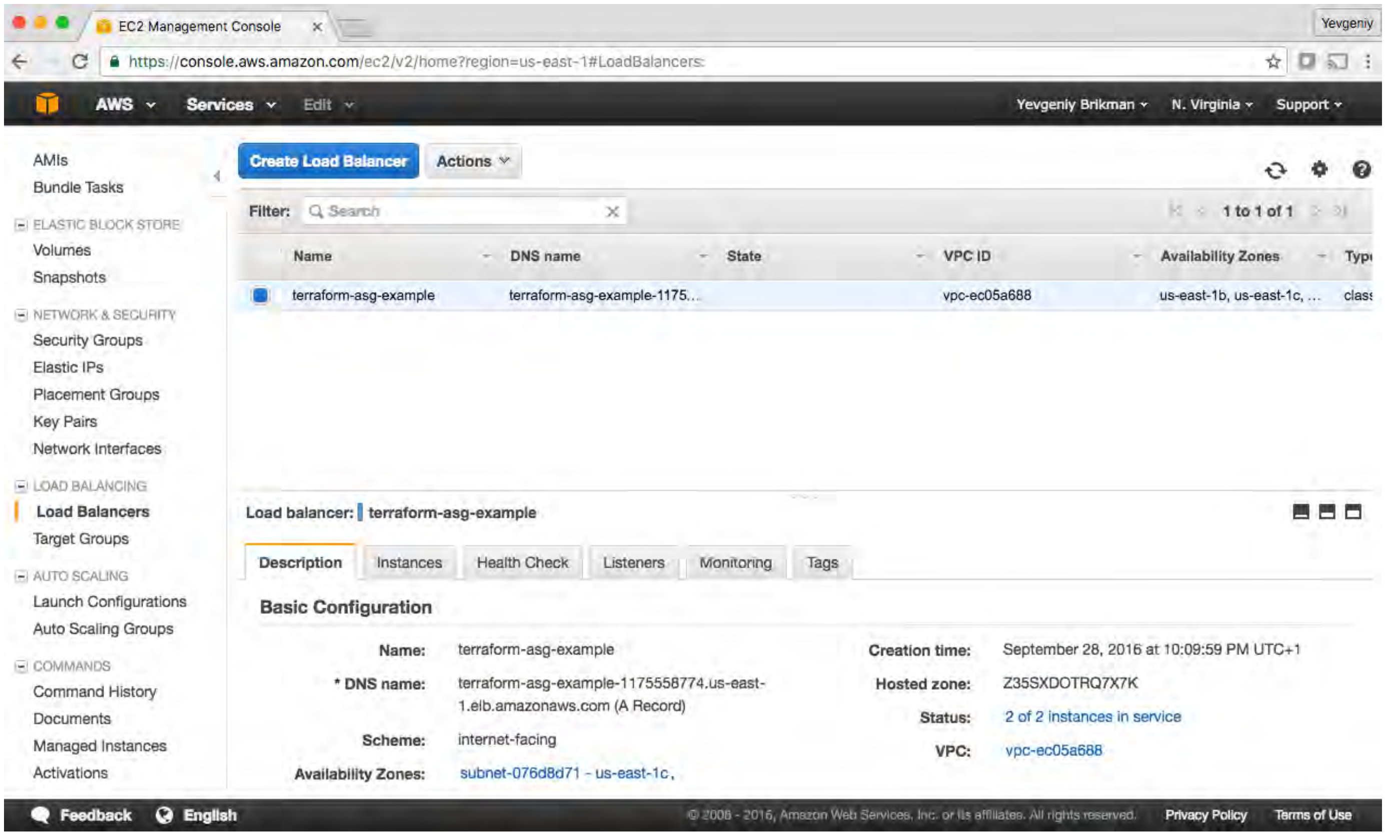
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Statu
terraform-asg-example	i-8c4aa29a	t2.micro	us-east-1c	● running	Initializing	None
terraform-asg-example	i-dadfe8eb	t2.micro	us-east-1b	● running	✓ 2/2 checks ...	None
terraform-example	i-edfbbafb	t2.micro	us-east-1d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

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

- i-26d6e117:
- i-a14a1f59:

Figure 2-13. 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.



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

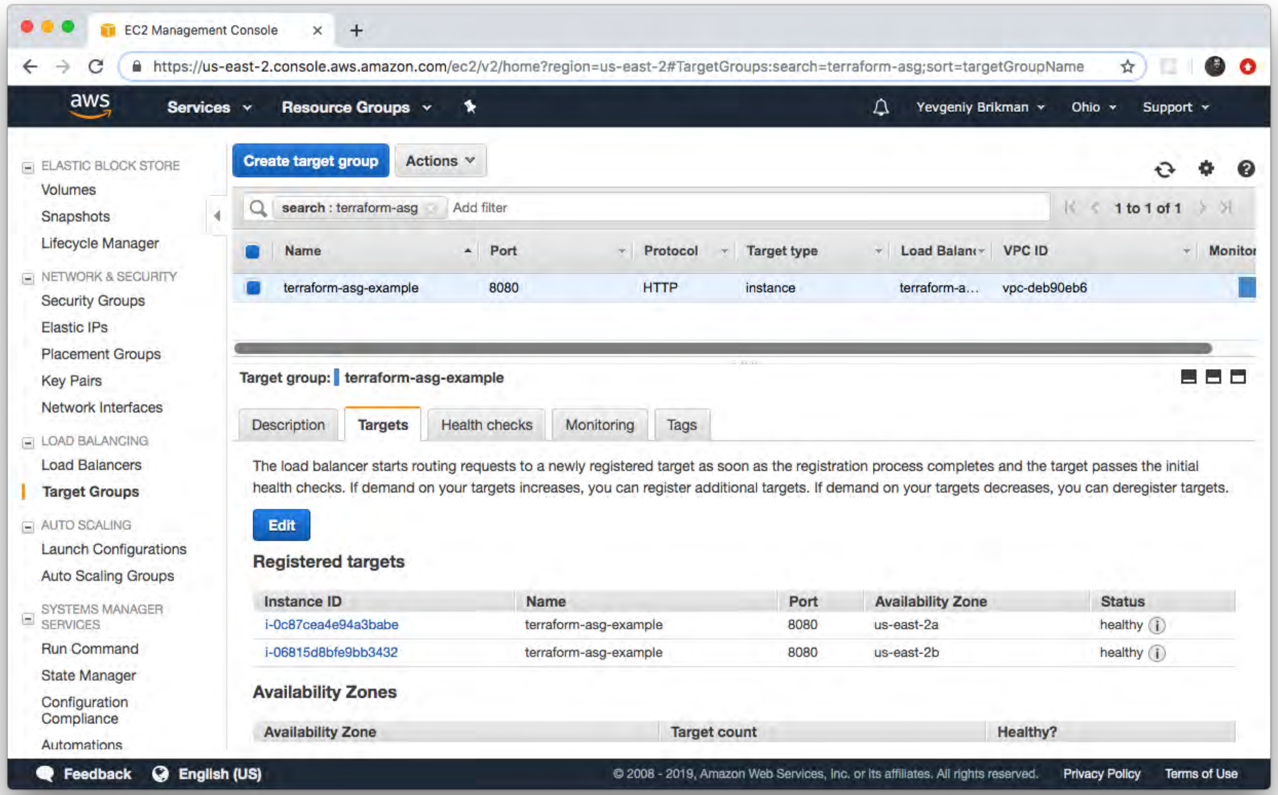
Load balancer: terraform-asg-example

Basic Configuration

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

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.



Name	Port	Protocol	Target type	Load Balan	VPC ID	Monitor
terraform-asg-example	8080	HTTP	instance	terraform-a...	vpc-deb90eb6	

Target group: terraform-asg-example

The load balancer starts routing requests to a newly registered target as soon as the registration process completes and the target passes the initial health checks. If demand on your targets increases, you can register additional targets. If demand on your targets decreases, you can deregister targets.

Registered targets

Instance ID	Name	Port	Availability Zone	Status
i-0c87cea4e94a3babe	terraform-asg-example	8080	us-east-2a	healthy
i-06815d8bfe9bb3432	terraform-asg-example	8080	us-east-2b	healthy

Availability Zones

Availability Zone	Target count	Healthy?

Figure 2-15. The Target Group

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 a production environment! 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, use either *http://www.ipaddressguide.com/cidr* in your browser or install the `ipcalc`command in your terminal.

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
- Limitations with Terraform's backends
- Isolating state files
    - Isolation via workspaces
    - Isolation via file layout
- The `terraform_remote_state` data source

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 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` or `terraform state` commands (you'll see examples of both in Chapter 5).

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 the following 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. A Terraform *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 Cloud, 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 acquire 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 your 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 concerns, 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 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 second lifecycle setting you've seen (the first was `create_before_destroy` in Chapter 2). 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 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 provider code and then run `terraform apply` to deploy (note: to deploy this code, your IAM User will need permissions to create S3 buckets and DynamoDB tables, as specified in "Set Up Your AWS Account"). 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 lives. Make sure to replace this with the region of the S3 bucket you created earlier.

`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 is here as a second layer to ensure that the data is always encrypted.

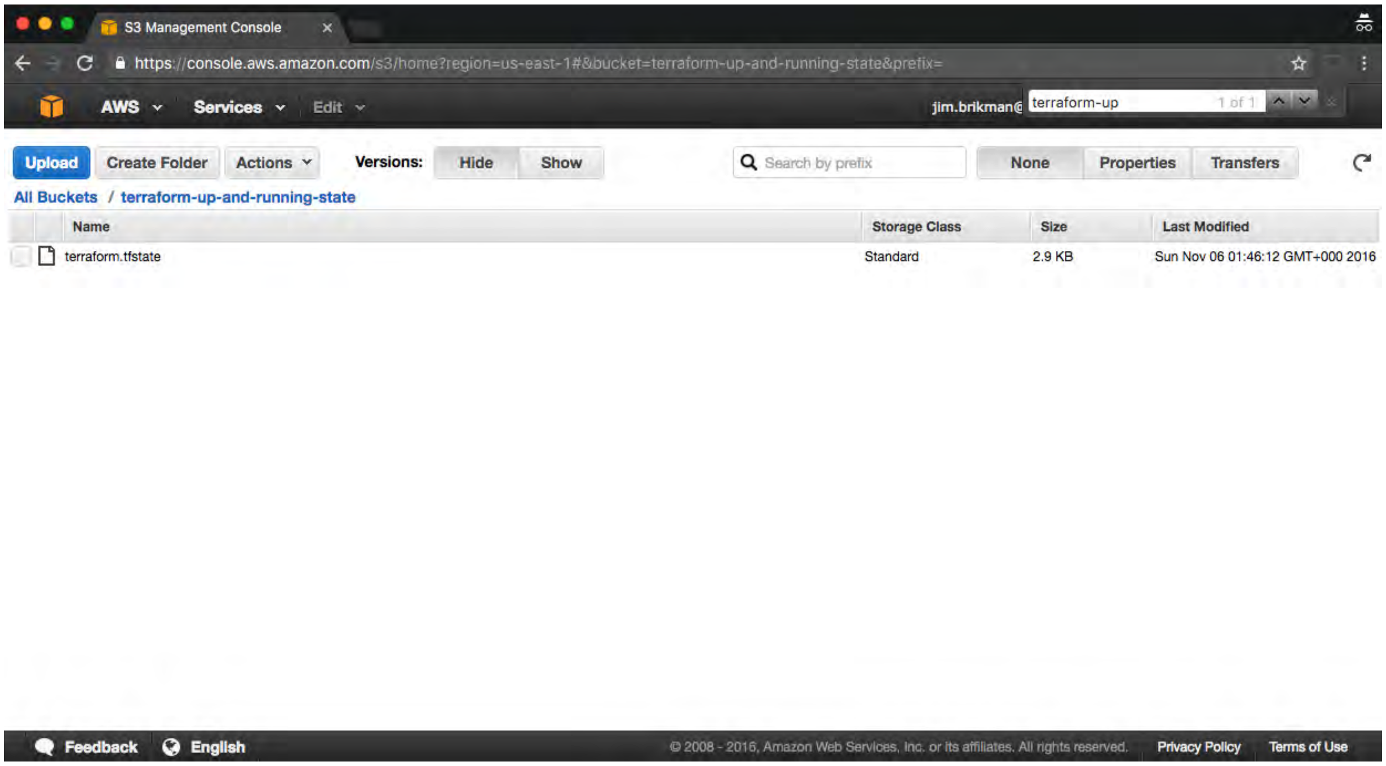
To tell Terraform to store your state file in this S3 bucket, you're going to use the `terraform init` command again. This command can not only download provider code, but also configure your Terraform backend (and you'll see yet another use later on too). Moreover, the `init` command is idempotent,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


S3 Management Console
https://console.aws.amazon.com/s3/home?region=us-east-1#&bucket=terraform-up-and-running-state&prefix=
AWS
Services
Edit
jim.brikman@
terraform-up
1 of 1
Upload
Create Folder
Actions
Versions:
Hide
Show
Search by prefix
None
Properties
Transfers
All Buckets / terraform-up-and-running-state
Feedback
English
© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Name	Storage Class	Size	Last Modified
terraform.tfstate	Standard	2.9 KB	Sun Nov 06 01:46:12 GMT+000 2016

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:

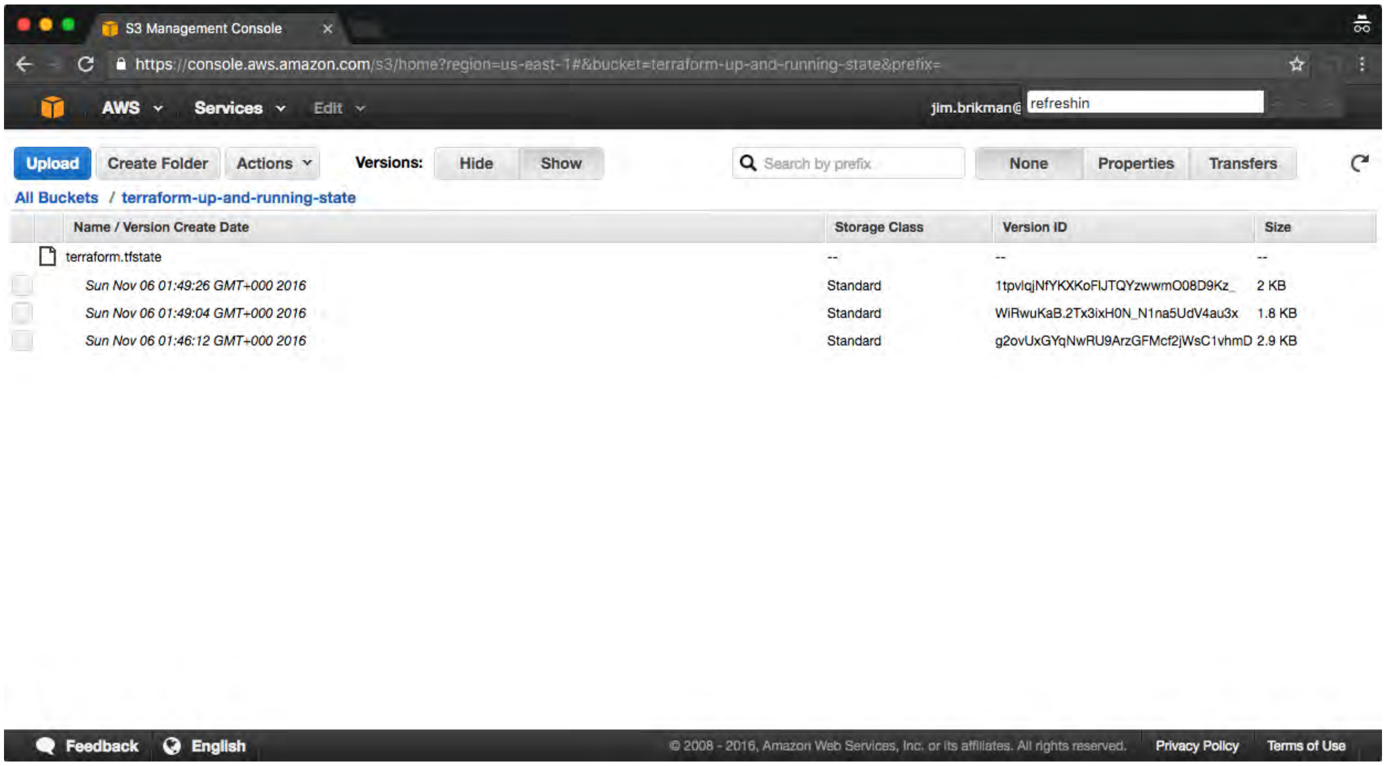
```
output "s3_bucket_arn" {
  value       = aws_s3_bucket.terraform_state.arn
  description = "The ARN of the S3 bucket"
}

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

These variables 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...
aws_s3_bucket.terraform_state: Refreshing 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.


S3 Management Console
https://console.aws.amazon.com/s3/home?region=us-east-1#&bucket=terraform-up-and-running-state&prefix=
AWS
Services
Edit
jim.brikman@
refreshin
Upload
Create Folder
Actions
Versions:
Hide
Show
Search by prefix
None
Properties
Transfers
All Buckets / terraform-up-and-running-state
Feedback
English
© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Name / Version Create Date	Storage Class	Version ID	Size
terraform.tfstate	--	--	--
Sun Nov 06 01:49:26 GMT+000 2016	Standard	1tpvlqjNfYKXKoFlJTQYzwwmO08D9Kz_	2 KB
Sun Nov 06 01:49:04 GMT+000 2016	Standard	WiRwuKaB.2Tx3ixH0N_N1na5UdV4au3x	1.8 KB
Sun Nov 06 01:46:12 GMT+000 2016	Standard	g2ovUxGYqNwRU9ArzGFMcf2jWsC1vhmD	2.9 KB

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.

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 all of your Terraform code, 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 (you'll learn all about Terraform modules in Chapter 4 and that Chapter 6; for now, it's enough to understand that modules are a way to organize and reuse Terraform code, and that real-world Terraform code typically consists of many small 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! Having 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`:

```
# 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. You'll see an example of how to use Terragrunt in Chapter 8.

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.

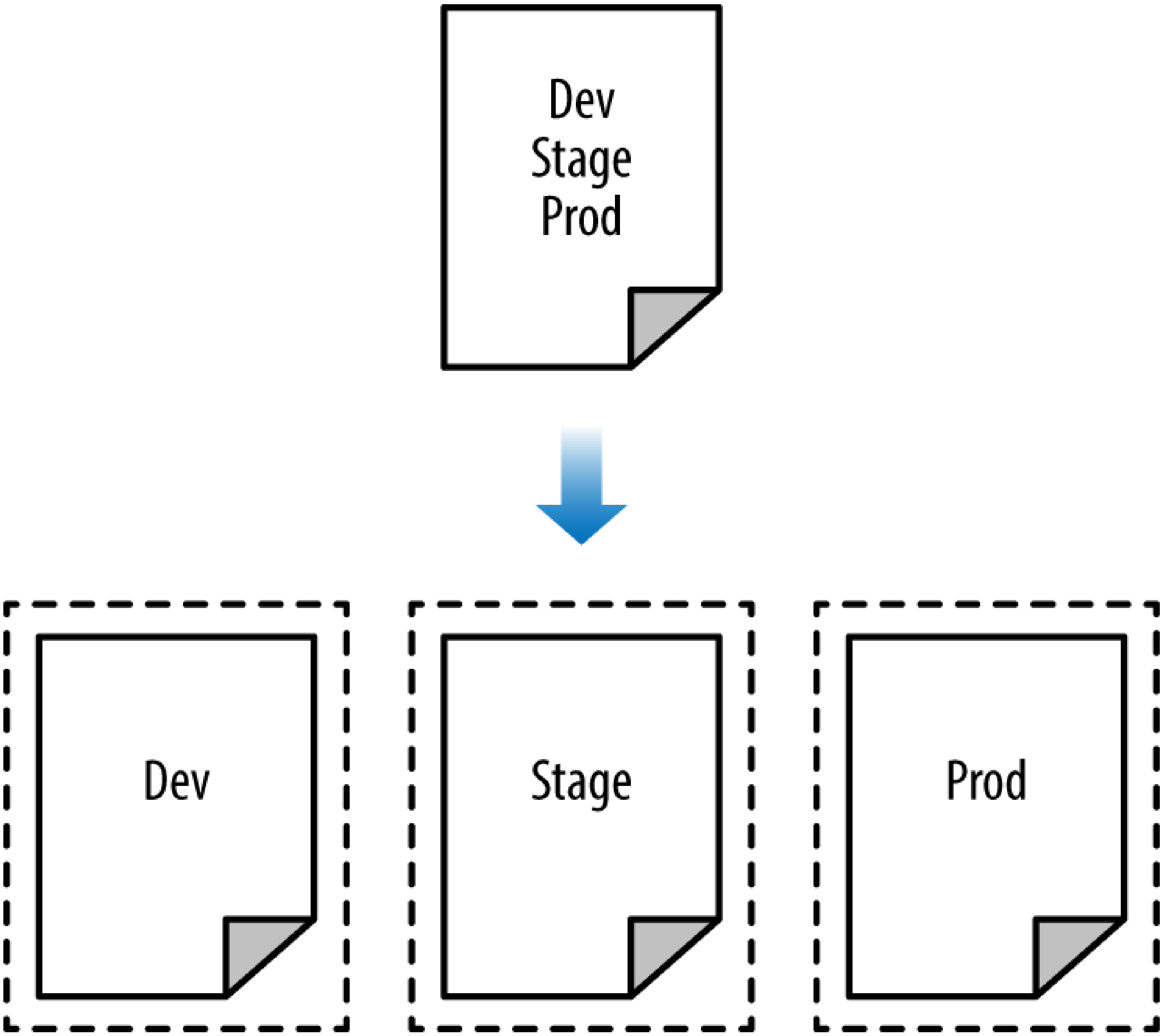


Figure 3-3. Instead of defining all your environments in a single set of Terraform configurations (top), you want to define each environment in a separate set of configurations (bottom), so a problem in one environment is completely isolated from the others.

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 never 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 some Terraform code 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 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 code:

```
$ 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.

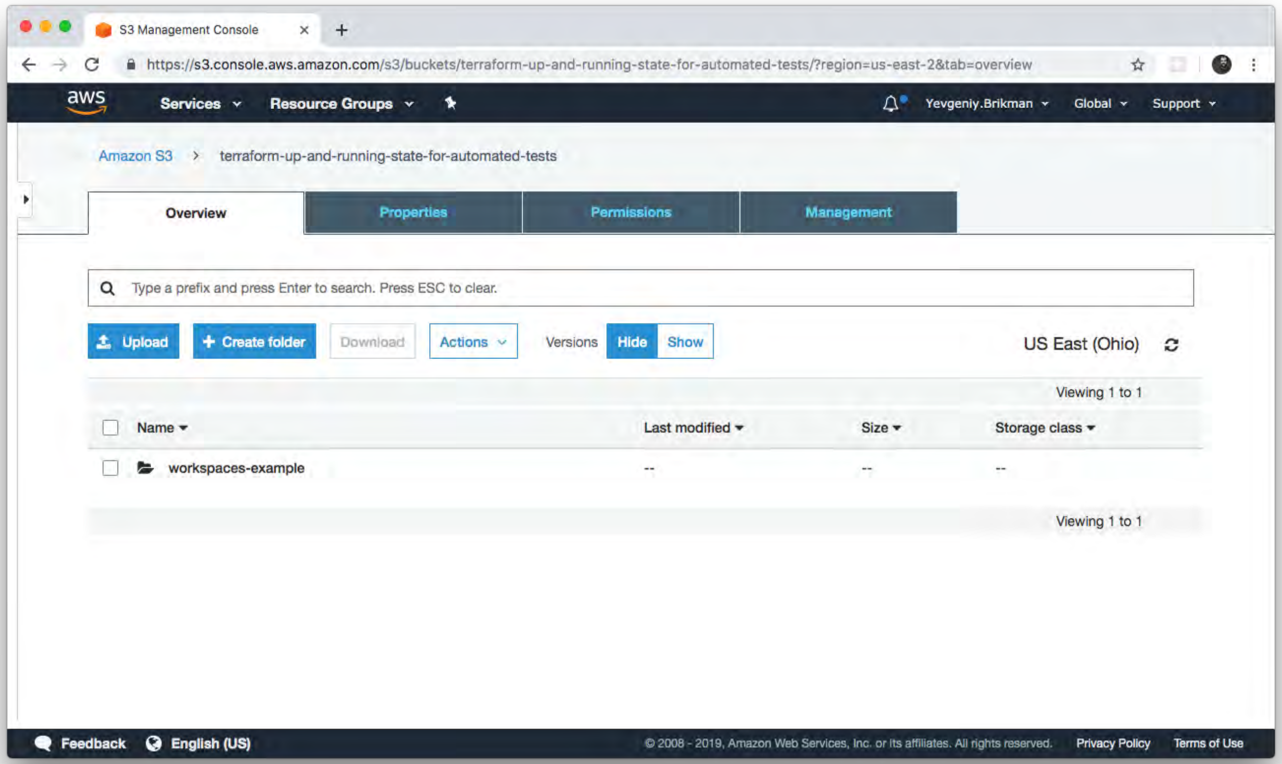


Figure 3-4. The S3 bucket after the state was stored in the default workspace

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`:

```
$ 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.
```

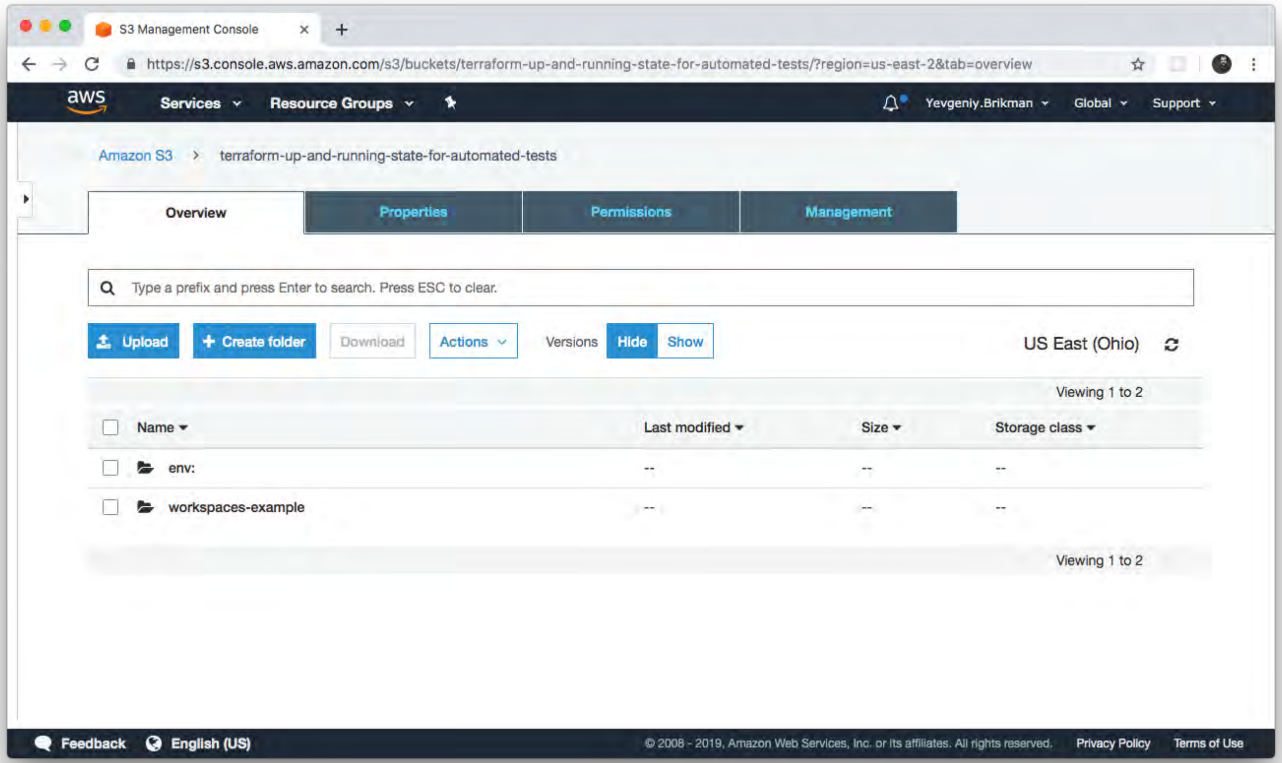
You now have three workspaces available, which you can see with the `terraform workspace list` command:

```
$ 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".
```

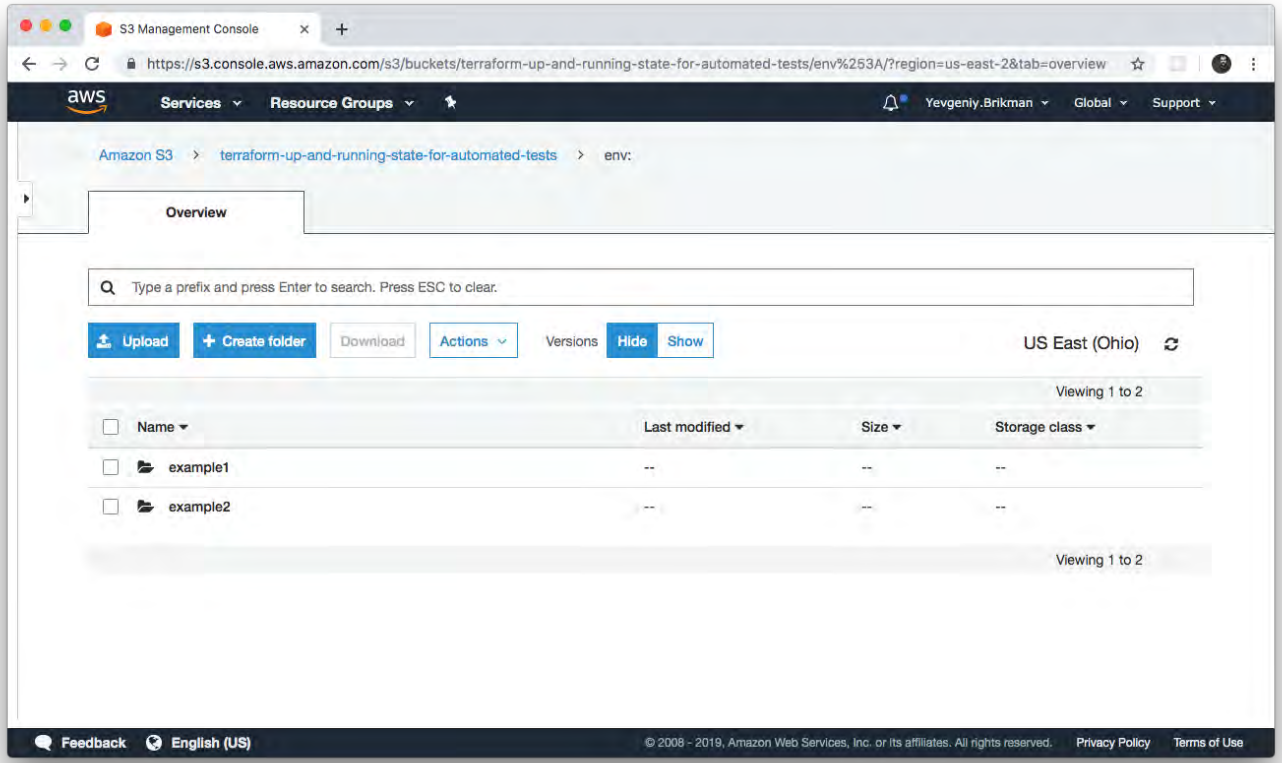
To understand how this works under the hood, take a look again in your S3 bucket, and you should now see a new folder called `env:`, as shown in Figure 3-5.


S3 Management Console
https://s3.console.aws.amazon.com/s3/buckets/terraform-up-and-running-state-for-automated-tests/?region=us-east-2&tab=overview
aws
Services
Resource Groups
Yevgeniy.Brikman
Global
Support
Amazon S3 > terraform-up-and-running-state-for-automated-tests
Overview
Properties
Permissions
Management
Type a prefix and press Enter to search. Press ESC to clear.
Upload
Create folder
Download
Actions
Versions
Hide
Show
US East (Ohio)
Viewing 1 to 2
Feedback
English (US)
© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Name	Last modified	Size	Storage class
env:	--	--	--
workspaces-example	--	--	--

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.


S3 Management Console
https://s3.console.aws.amazon.com/s3/buckets/terraform-up-and-running-state-for-automated-tests/env%253A/?region=us-east-2&tab=overview
aws
Services
Resource Groups
Yevgeniy.Brikman
Global
Support
Amazon S3 > terraform-up-and-running-state-for-automated-tests > env:
Overview
Type a prefix and press Enter to search. Press ESC to clear.
Upload
Create folder
Download
Actions
Versions
Hide
Show
US East (Ohio)
Viewing 1 to 2
Feedback
English (US)
© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Name	Last modified	Size	Storage class
example1	--	--	--
example2	--	--	--

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 an `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 set the instance type to `t2.medium` in the default workspace and `t2.micro` in all other workspaces (e.g., to save money when experimenting):

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

The code above uses *ternary syntax* to conditionally set `instance_type` to either `t2.medium` or `t2.micro`, depending on the value of `terraform.workspace`. You'll see the full details of ternary syntax and conditional logic in Terraform in Chapter 5.

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 staging from production).

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 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*.

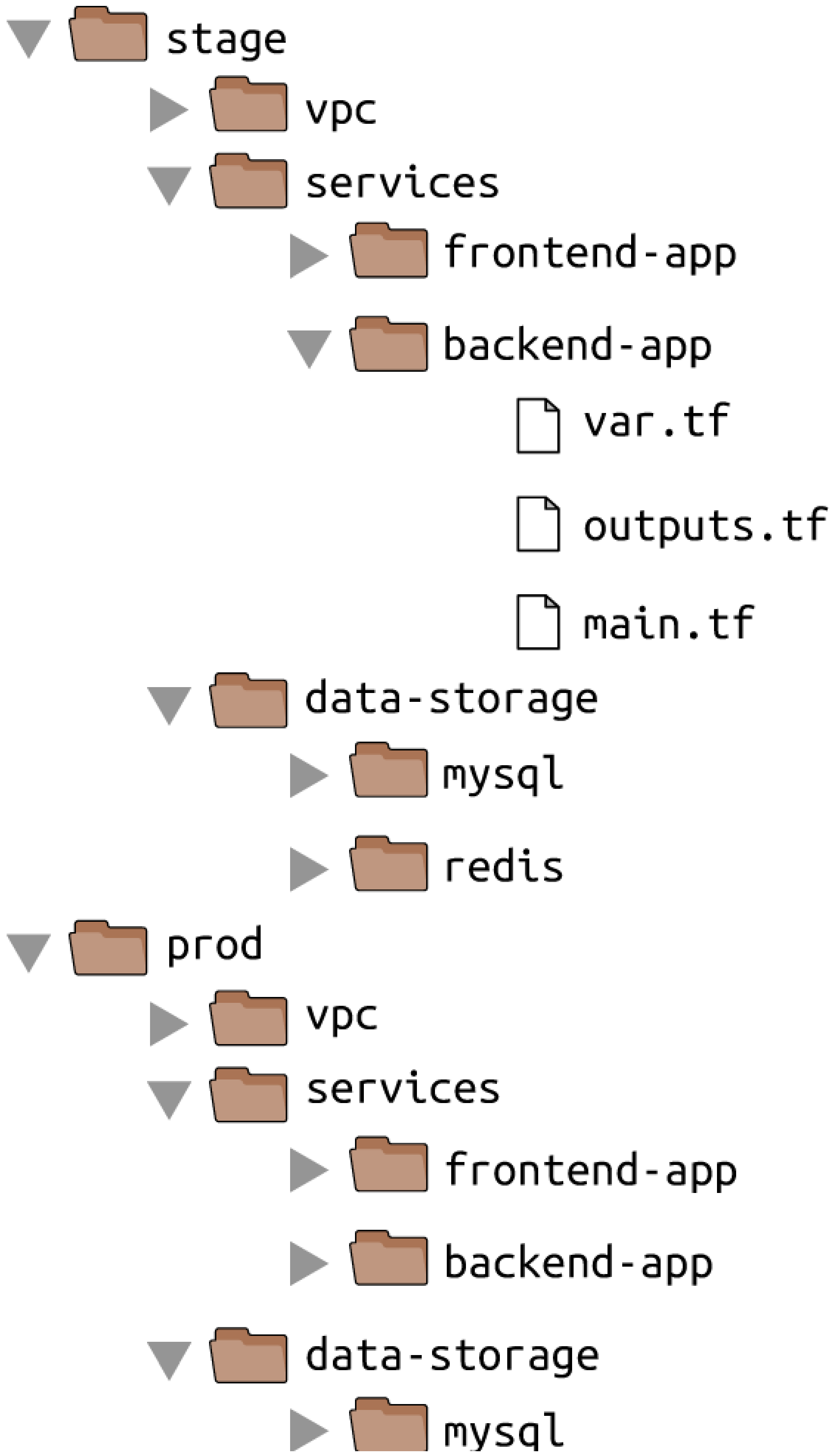
2. Configure a different backend for each environment, using different authentication mechanisms and access controls (e.g., each environment could live in a separate AWS account with a separate S3 bucket as a backend).

With this approach, the use of separate folders makes it much clearer which environments you're deploying 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

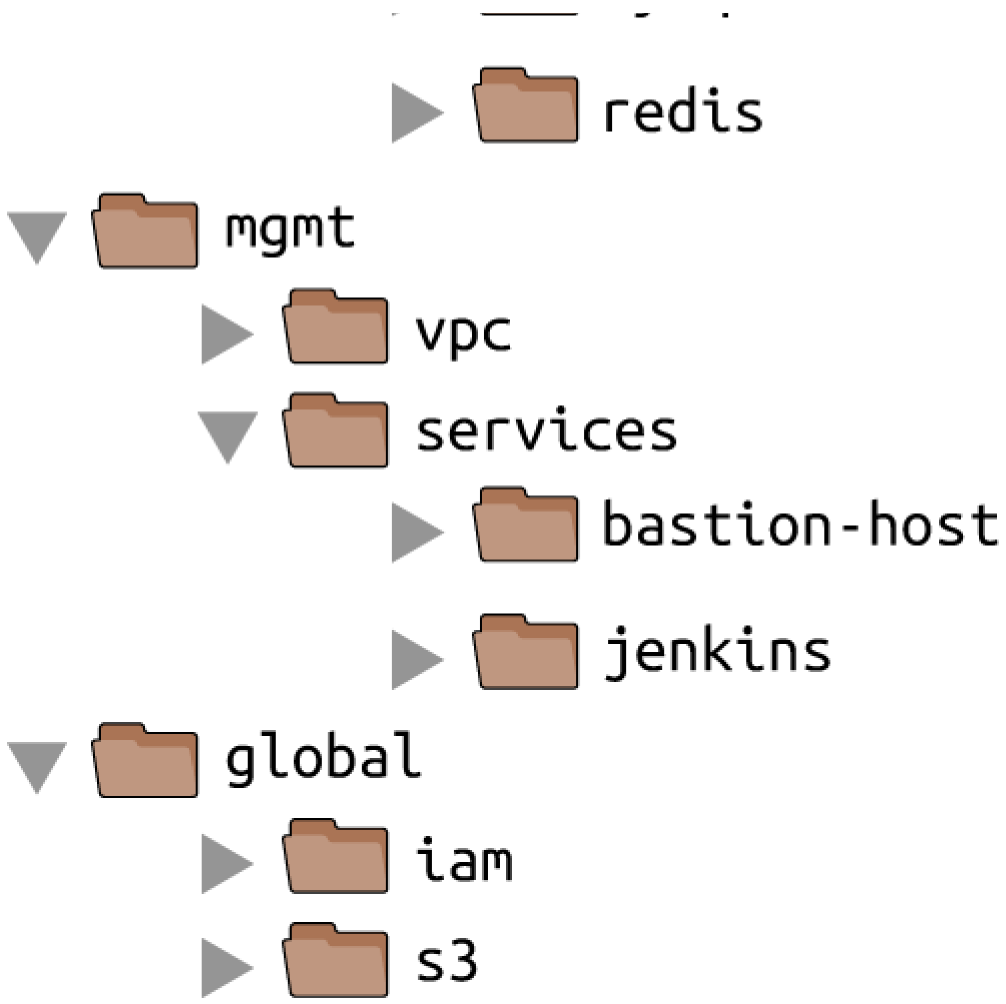


Figure 3-7. Typical file layout for a Terraform project

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 pre-production 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 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.

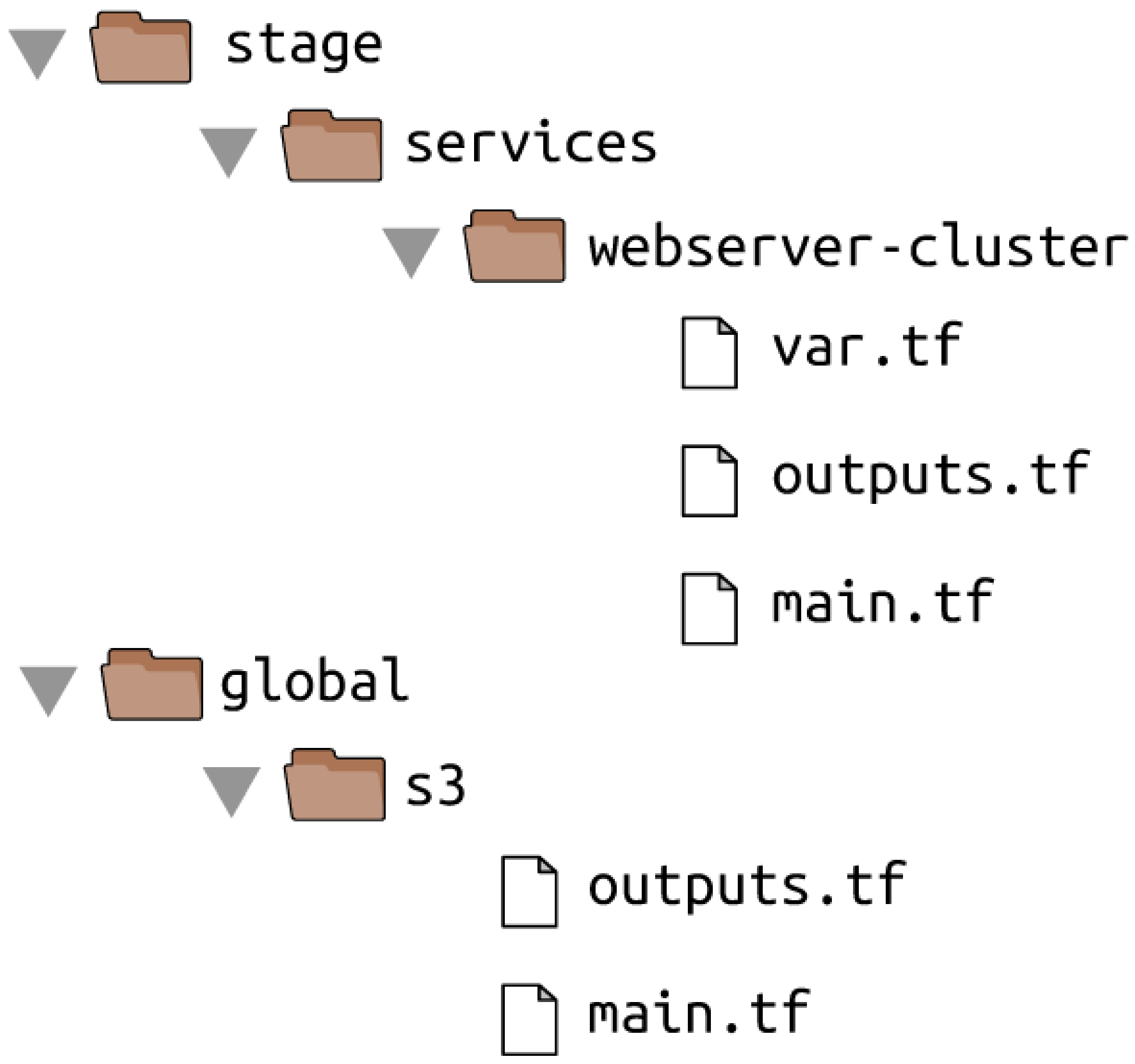


Figure 3-8. File layout for the web server cluster code

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 using an attribute reference (e.g.,access the database address via`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.

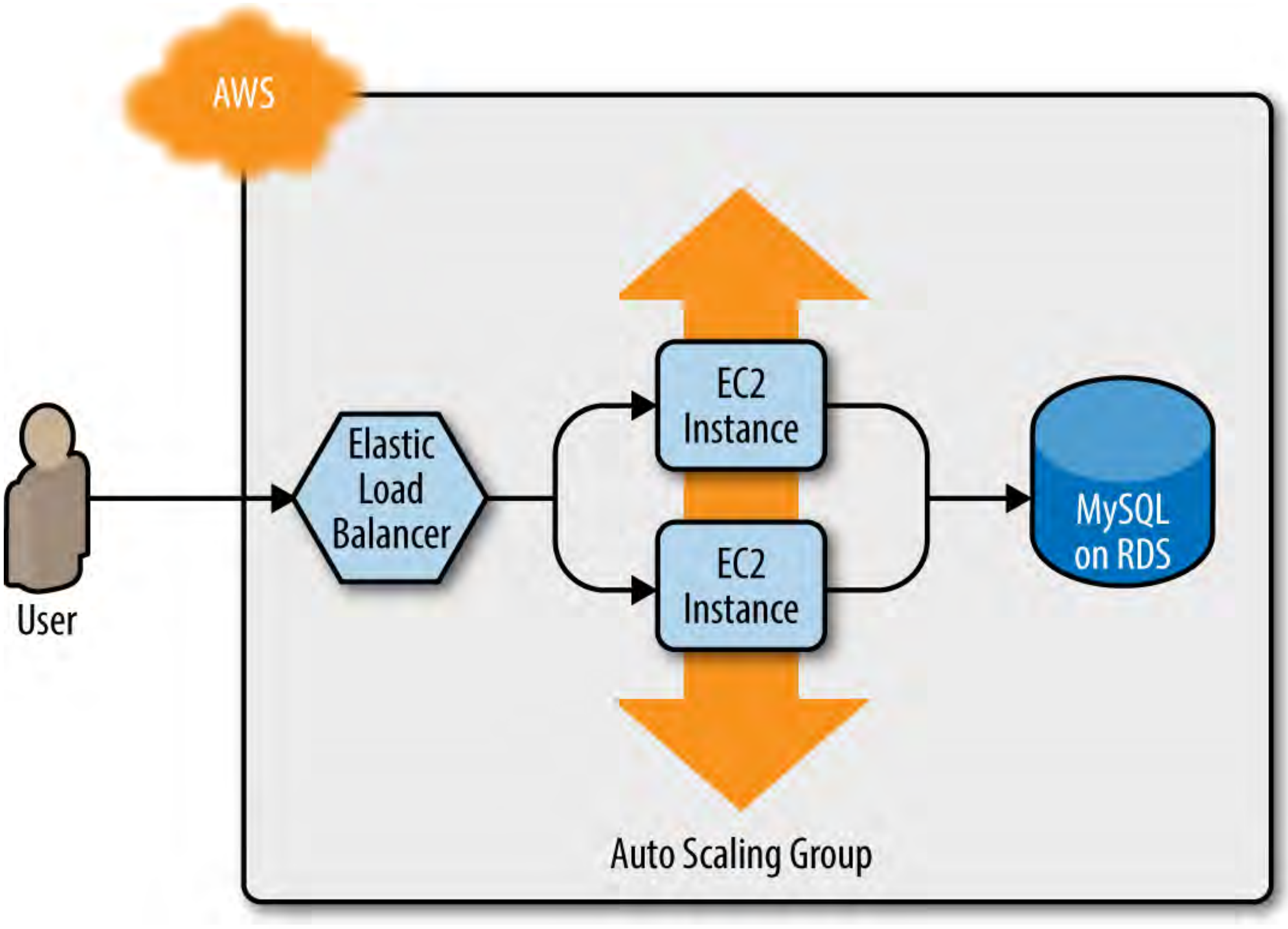


Figure 3-9. The web server cluster talks to MySQL, which is deployed on top of Amazon’s Relational Database Service

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            = "???"
}
```

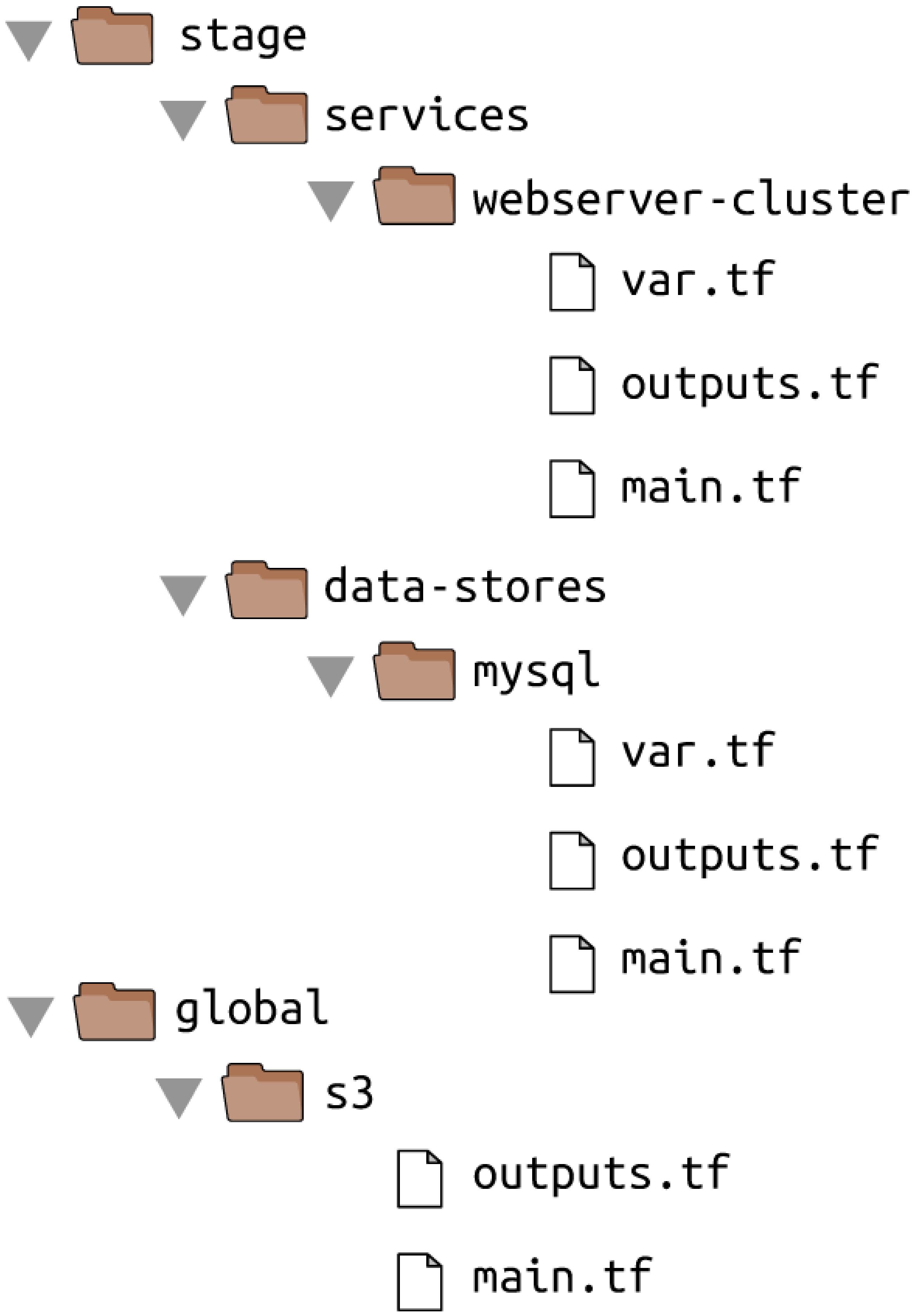


Figure 3-10. Create the database code in the stage/data-stores folder

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.secret_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:

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

(...)
```

Note there is intentionally a space before the `export` command to prevent the secret from being stored on disk in your Bash history[7]. An even better way to keep secrets from accidentally being stored on disk in plaintext is to store them in a command-line-friendly secret store, such as pass, and to use a subshell to securely read the secret from `pass` and into an environment variable:

```
$ export TF_VAR_db_password=$(pass database-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 init` and `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.

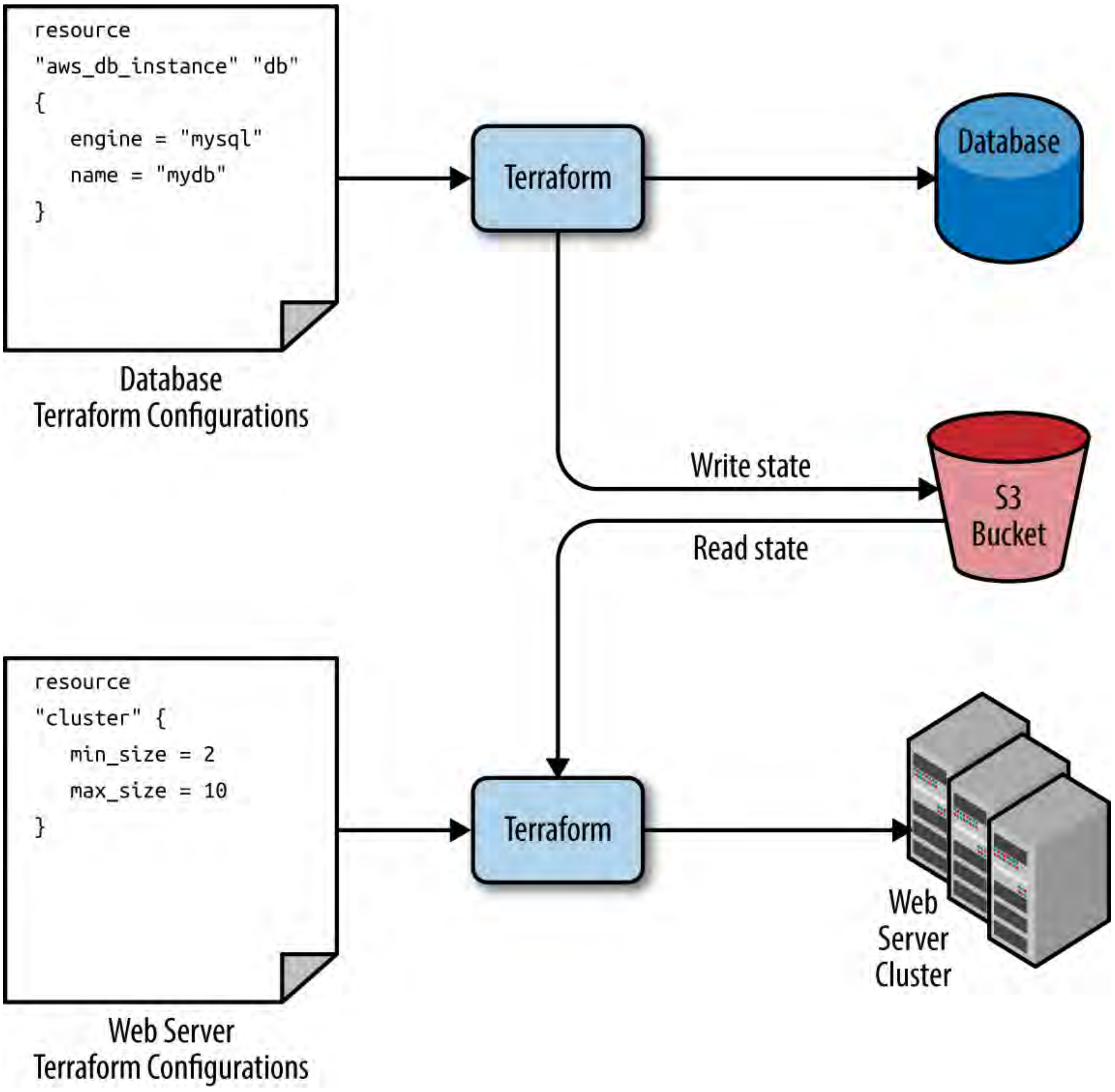


Figure 3-11. The database writes its state to an S3 bucket (top) and the web server cluster reads that state from the same bucket (bottom)

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 functions* 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 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`. 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:

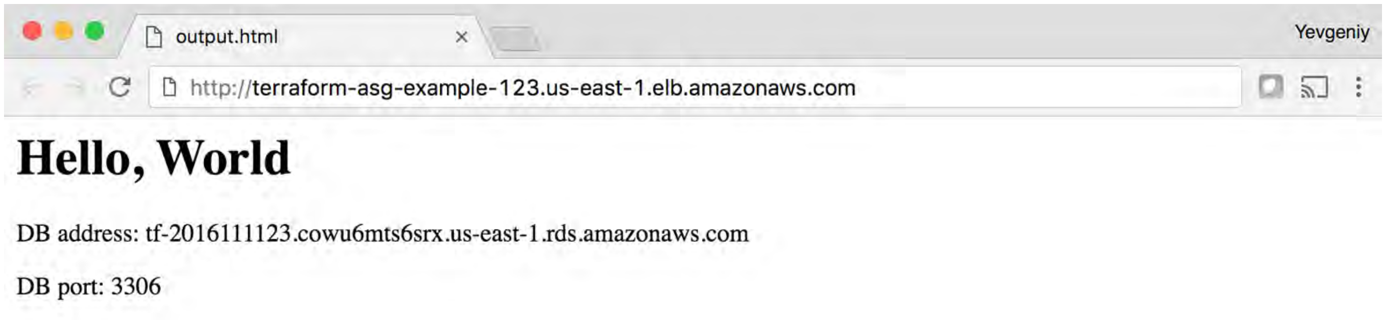
```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
  instance_type   = "t2.micro"
  security_groups = [aws_security_group.instance.id]
```

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

    # Required when using a launch configuration with an auto scaling group.
    # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
    lifecycle {
      create_before_destroy = true
    }
  }
```

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.


output.html
Yevgeniy
http://terraform-asg-example-123.us-east-1.elb.amazonaws.com

Hello, World

DB address: tf-2016111123.cowu6mts6srx.us-east-1.rds.amazonaws.com

DB port: 3306

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.

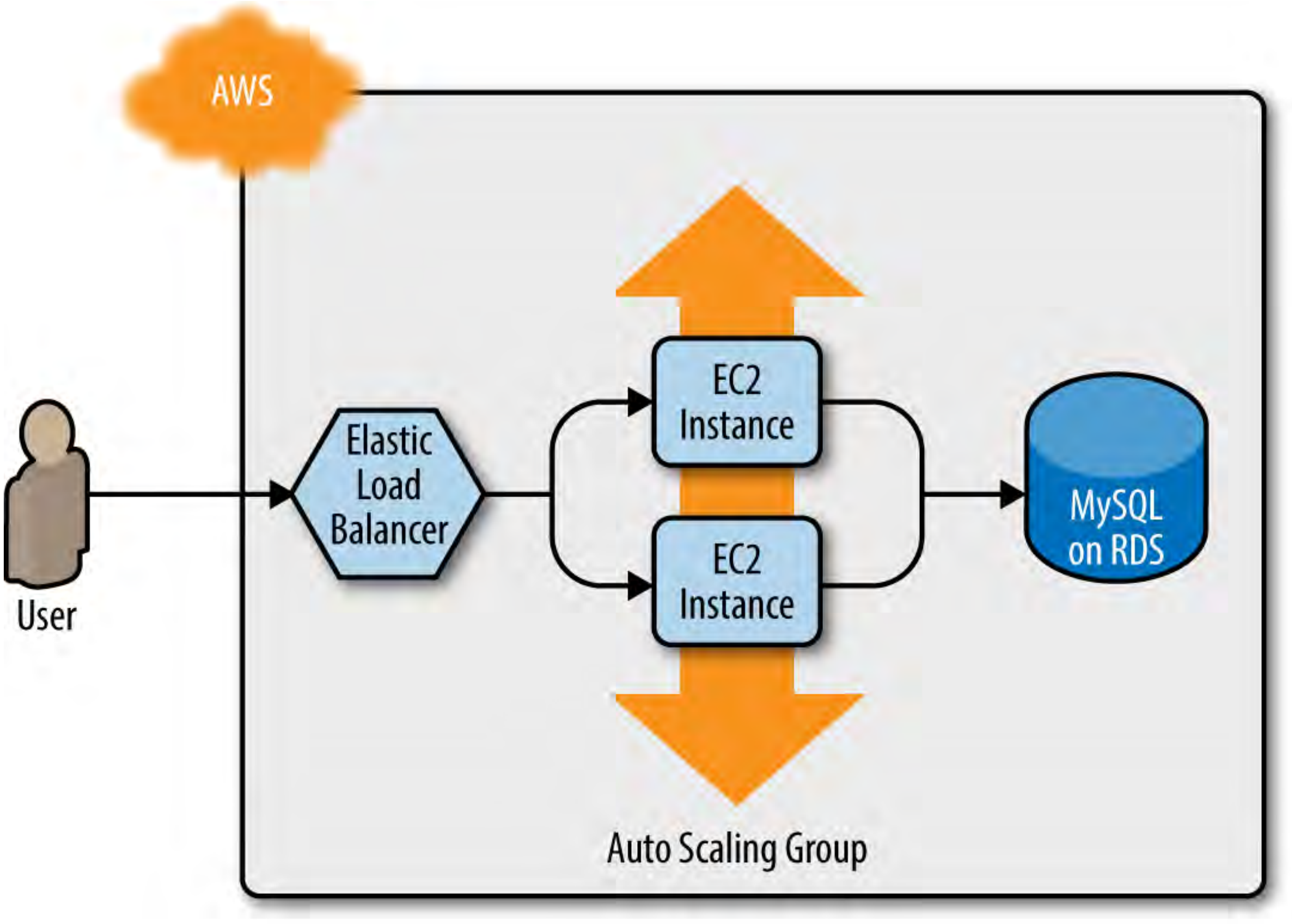


Figure 4-1. A load balancer, web server cluster, and database

This works great as a first environment, but you typically need at least two environments: one for your team's internal testing ("staging") and one that real users can access ("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.

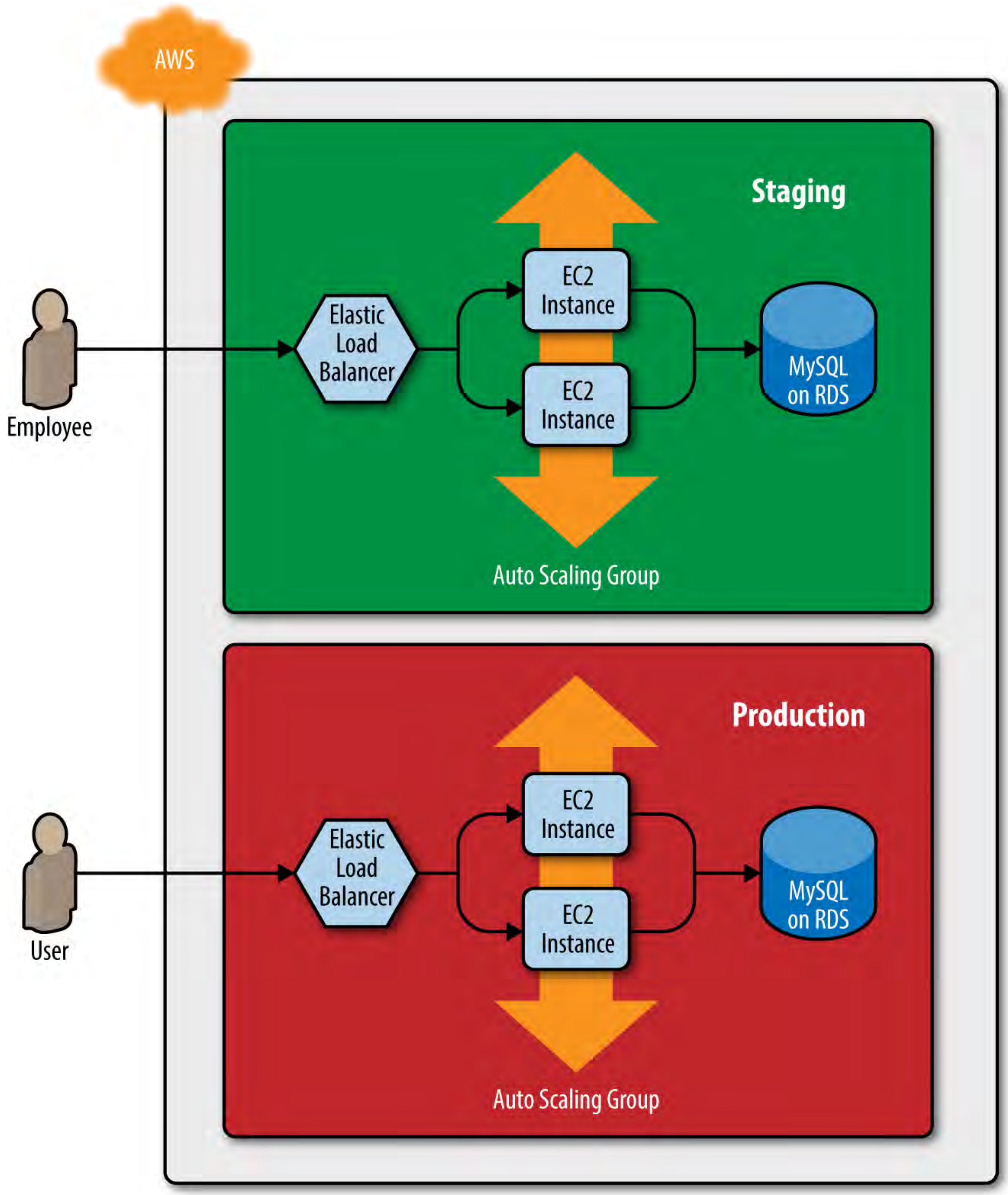


Figure 4-2. Two environments, each with its own load balancer, web server cluster, and database

How do you add this production environment without having to copy/paste all of the code from staging? For example, 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, such as Ruby, 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 everywhere:

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

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

With Terraform, you can put your code inside of a *Terraform module* and reuse that module in multiple places throughout your code. Instead of having the same code copy/pasted in the staging and production environments, you'll be able to have both environments reuse code from the same module, as shown in Figure 4-3.

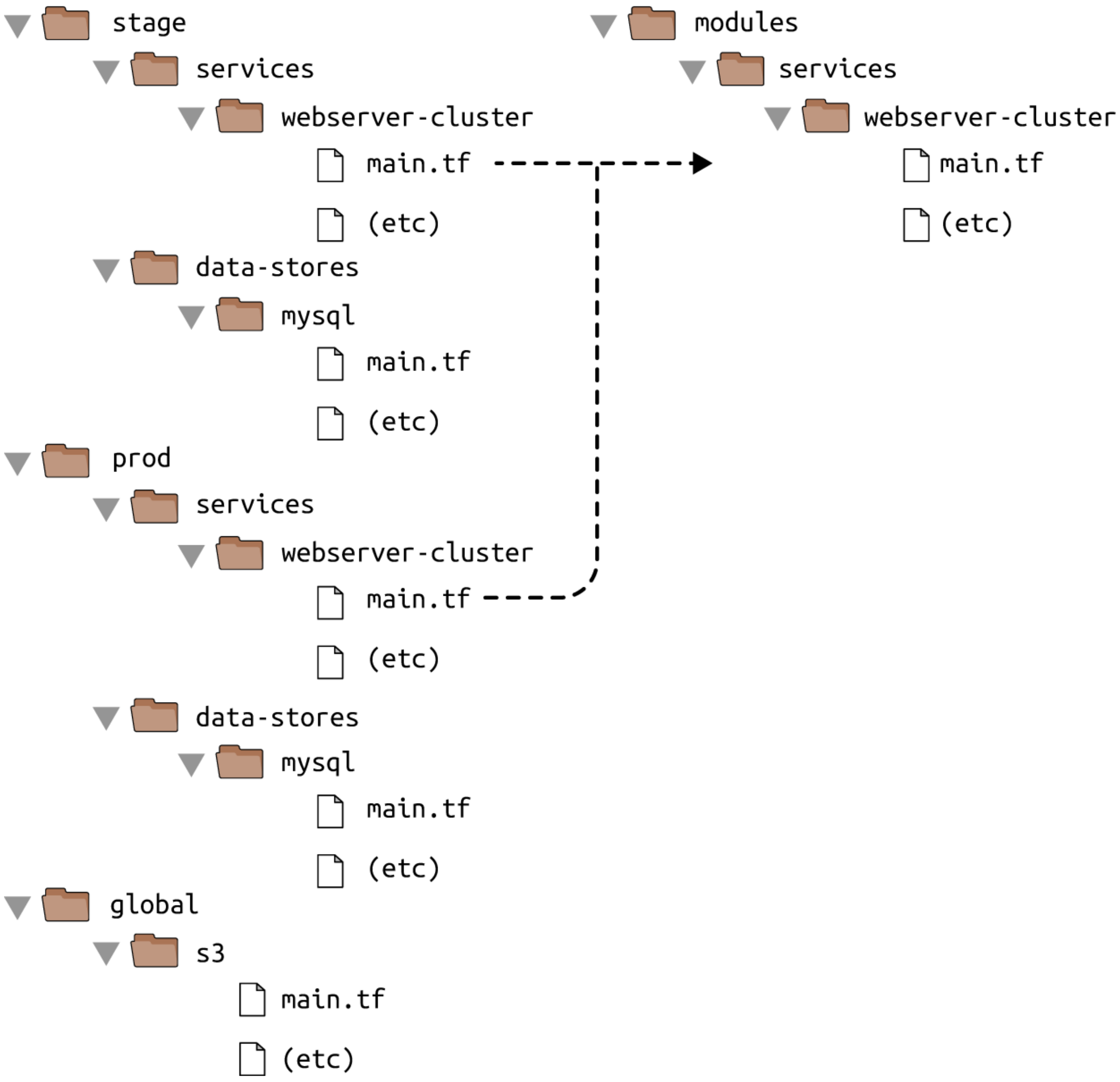


Figure 4-3. Putting code into modules allows you to reuse that code from multiple environments

This is a big deal. Modules are the key ingredient to writing reusable, maintainable, and testable Terraform code. Once you start using them, there's no going back. You'll start building everything as a module, creating a library of modules to share within your company, start leveraging modules you find online, and start thinking of your entire infrastructure as a collection of reusable modules.

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

- Module basics
- Module inputs
- Module locals
- 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*.

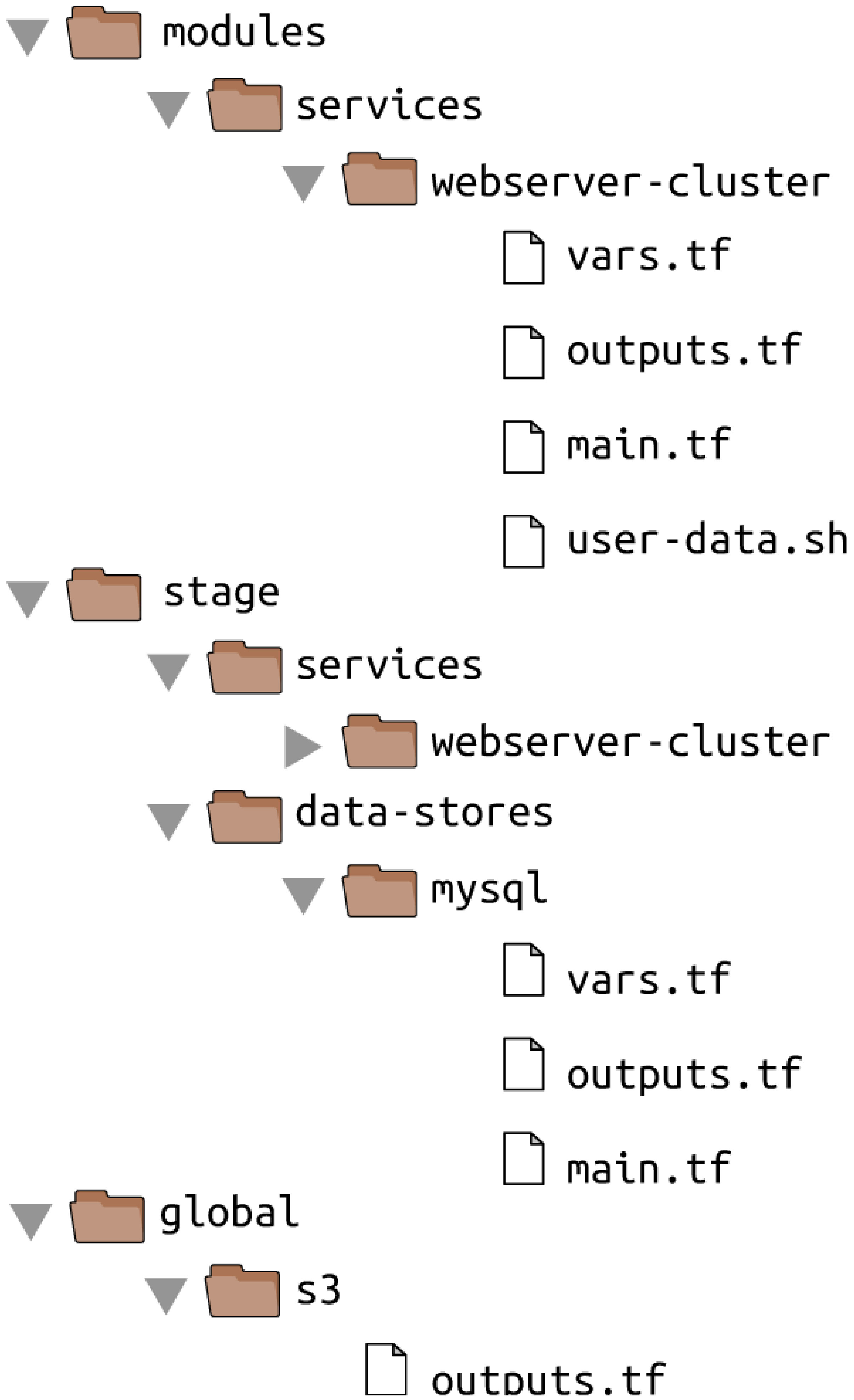
Module Basics

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 4-4.

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

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

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

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

  [CONFIG ...]
}
```

where `NAME` is an identifier you can use throughout the Terraform code to refer to this module (e.g., `web-service`), `SOURCE` is the path where the module code can be found (e.g., *modules/services/webserver-cluster*), and `CONFIG` consists of one or more arguments that are specific to that module. 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 with minimal 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 sleeve. 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, such as Ruby, 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 add a production database similar to the staging one.

As you can see, you set input variables for a module using the same syntax as setting arguments 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

  # Required when using a launch configuration with an auto scaling group.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

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, stick with `"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 `webserver-cluster` 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* in a `locals` block:

```
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, such as Ruby, 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 using the following syntax:

```
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 defined.

`path.root`

Returns the filesystem path of the root module.

`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 4-5.

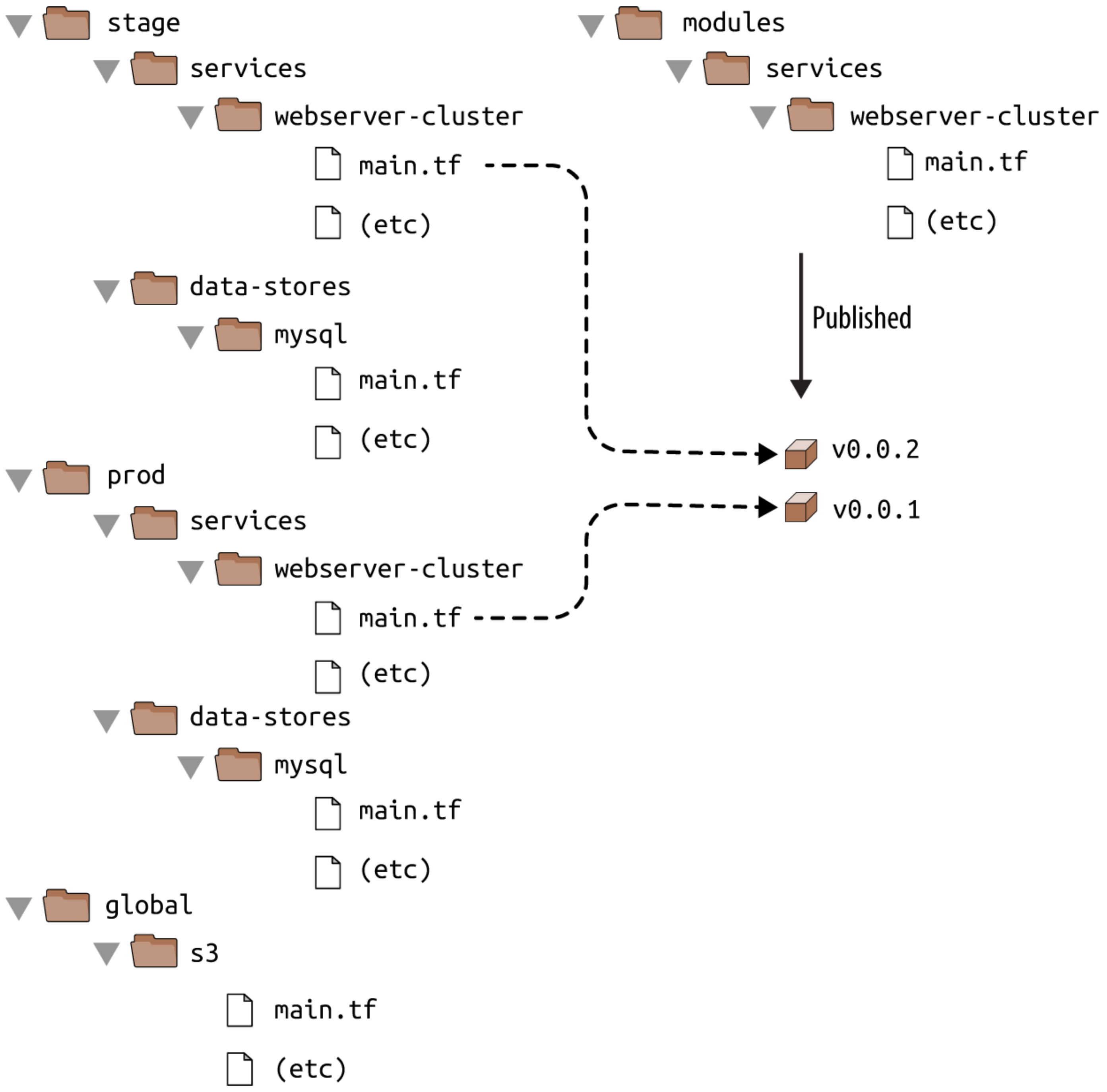


Figure 4-5. Using different versions of a module in different environments

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 4-6.

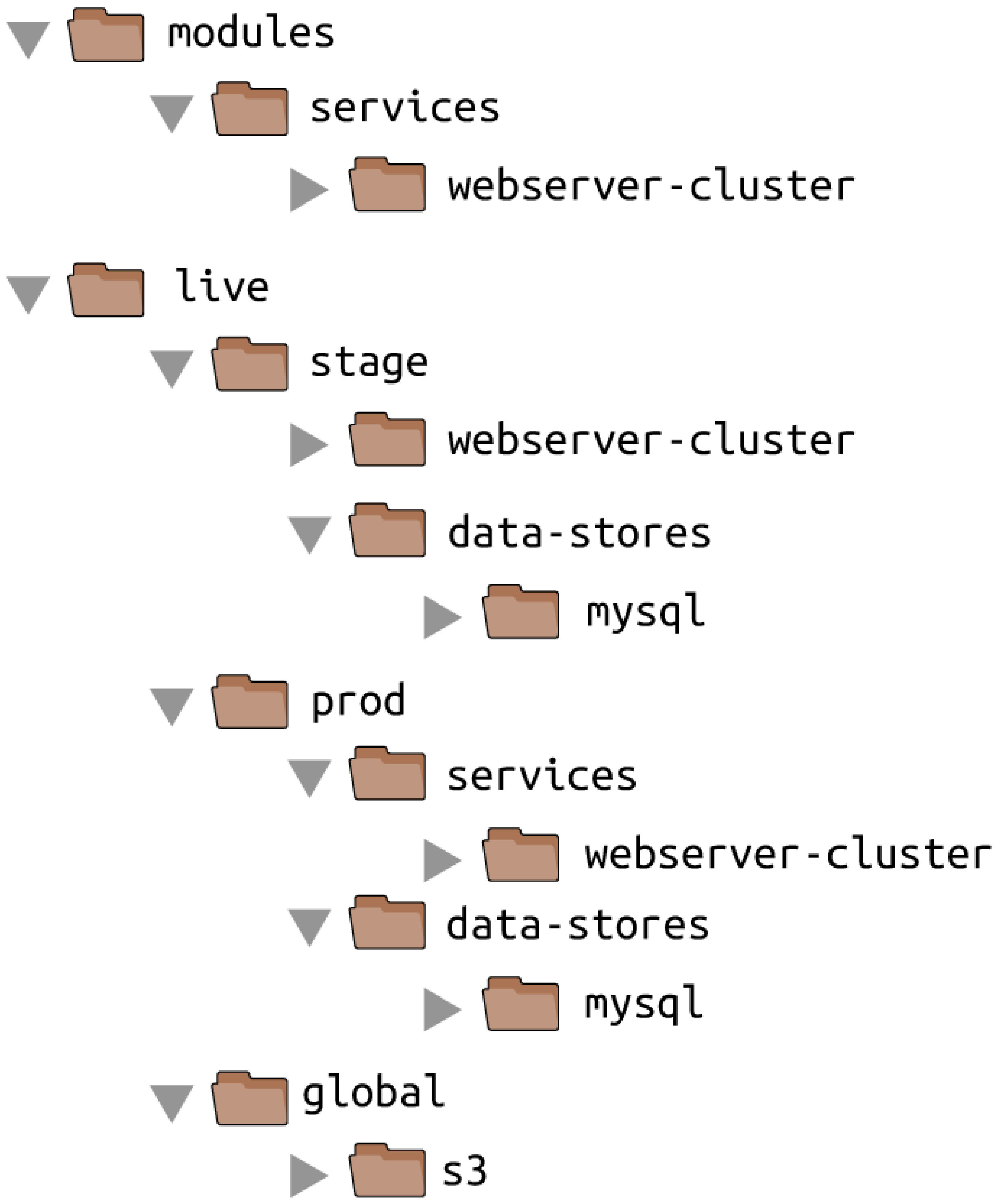


Figure 4-6. File layout with multiple repositories

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 below is required):

```
module "webserver_cluster" {
  source = "github.com/foo/modules//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 = "github.com/brikis98/terraform-up-and-running-code//
  code/terraform/04-terraform-module/module-example/modules/
  services/webserver-cluster?ref=v0.1.0"
```

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 `init` 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=v0.1.0
for 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:

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

For example:

```
git::git@github.com:acme/modules.git//example?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 that you're using versioned modules, let's walk through the process of making changes. Let's say 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, the `count` meta-parameter, `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 scenario:

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` along with:

1. **Array lookup syntax**: The syntax for looking up members of a list or array in Terraform is similar to most other programming languages:

   ```
   LIST[<INDEX>]
   ```

   For example, here’s how you would look up the element at index 1 of `var.user_names`:

   ```
   var.user_names[1]
   ```

2. **The length function**: Terraform has a built-in function called `length` that has the following syntax:

   ```
   length(<LIST>)
   ```

   As you can probably guess, the `length` function returns the number of items in the given `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:

neo_arn = arn:aws:iam::123456789012:user/neo
all_arns = [
  "arn:aws:iam::123456789012:user/neo",
  "arn:aws:iam::123456789012:user/trinity",
  "arn:aws:iam::123456789012:user/morpheus",
]
```

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 entire resource, but how do you do “loops” for inline blocks within a resource? For example, consider 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 you've specified your tags, how do you actually set them on the `aws_autoscaling_group` resource? What you need is 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. Moreover, the `count` parameter only works on entire resources, and not on inline blocks. Fortunately, in Terraform 0.12 and above, you can use 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 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 that 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.

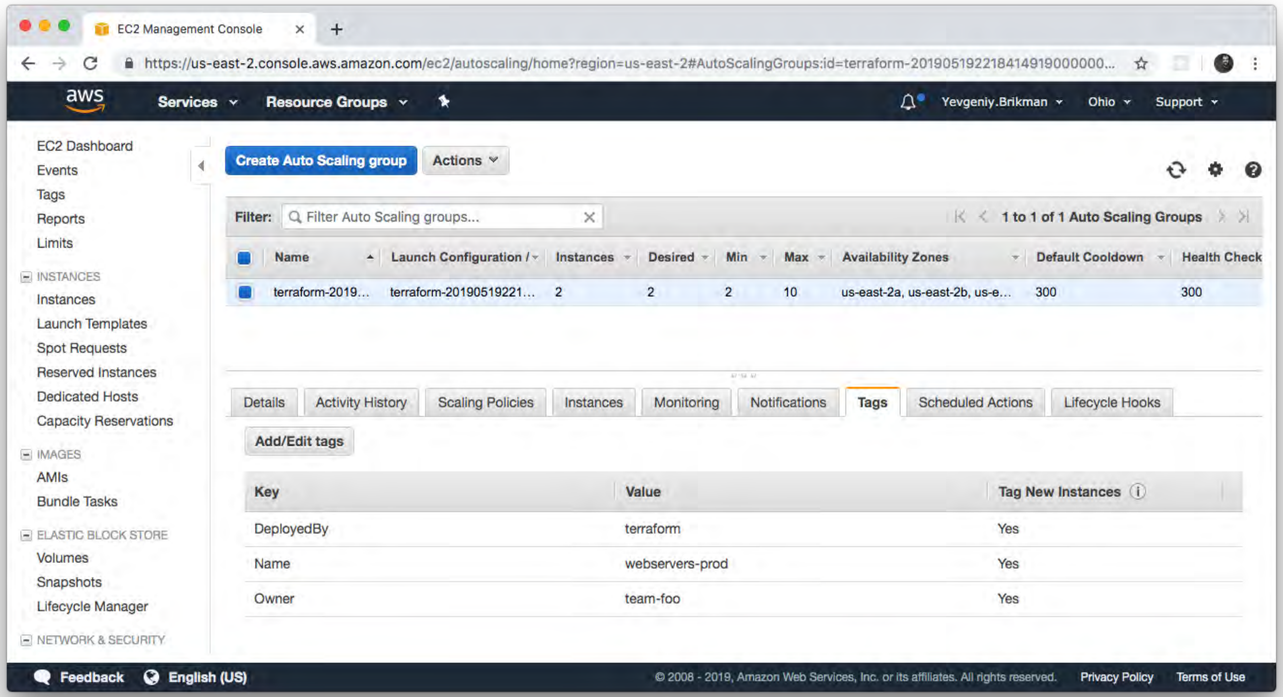
Putting this all together, here is how you can dynamically generate `tag` blocks using `for_each` in the `aws_autoscaling_group` resource:

```
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:

```
$ 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.



Name	Launch Configuration	Instances	Desired	Min	Max	Availability Zones	Default Cooldown	Health Check
terraform-2019...	terraform-20190519221...	2	2	2	10	us-east-2a, us-east-2b, us-e...	300	300

Key	Value	Tag New Instances
DeployedBy	terraform	Yes
Name	webservers-prod	Yes
Owner	team-foo	Yes

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 similar functionality in the form of a *for expression* (not to be confused with the `for_each` expression you saw in the previous section). 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 a 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 (e.g., for loops and if-statements) within strings using a syntax similar 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 scenario:

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.

Let’s go through each of these one at a time.

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  = "${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" {
    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
  }
}
```

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
  )

  # Required when using a launch configuration with an auto scaling group.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

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 string 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

  # Required when using a launch configuration with an auto scaling group.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

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:

```
Terraform will perform the following actions:

  # module.webserver_cluster.aws_autoscaling_group.ex will be updated in-place
  ~ resource "aws_autoscaling_group" "example" {
        id                        = "webservers-stage-terraform-20190516"
      ~ launch_configuration      = "terraform-20190516" -> (known after apply)
        (...)
    }

  # module.webserver_cluster.aws_launch_configuration.ex must be replaced
+/- resource "aws_launch_configuration" "example" {
      ~ id                          = "terraform-20190516" -> (known after apply)
        image_id                    = "ami-0c55b159cbfafe1f0"
        instance_type               = "t2.micro"
      ~ name                        = "terraform-20190516" -> (known after apply)
      ~ user_data                   = "bd7c0a6" -> "4919a13" # 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, the `create_before_destroy` lifecycle setting you first saw in Chapter 2 does exactly this.

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. Each time the launch configuration changes (which it will when you update the AMI or User Data), its name changes, 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                   = "example-2019" -> (known after apply)
      ~ launch_configuration = "example-2019" -> (known after apply)
      ~ name                 = "example-2019" -> (known after apply) # forces replacement
        (...)
```

```
    }

  # module.webserver_cluster.aws_launch_configuration.example must be replaced
+/- resource "aws_launch_configuration" "example" {
      ~ id                          = "terraform-2019" -> (known after apply)
        image_id                    = "ami-0c55b159cbfafe1f0"
        instance_type               = "t2.micro"
      ~ name                        = "terraform-2019" -> (known after apply)
      ~ user_data                   = "bd7c0a" -> "4919a" # 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).

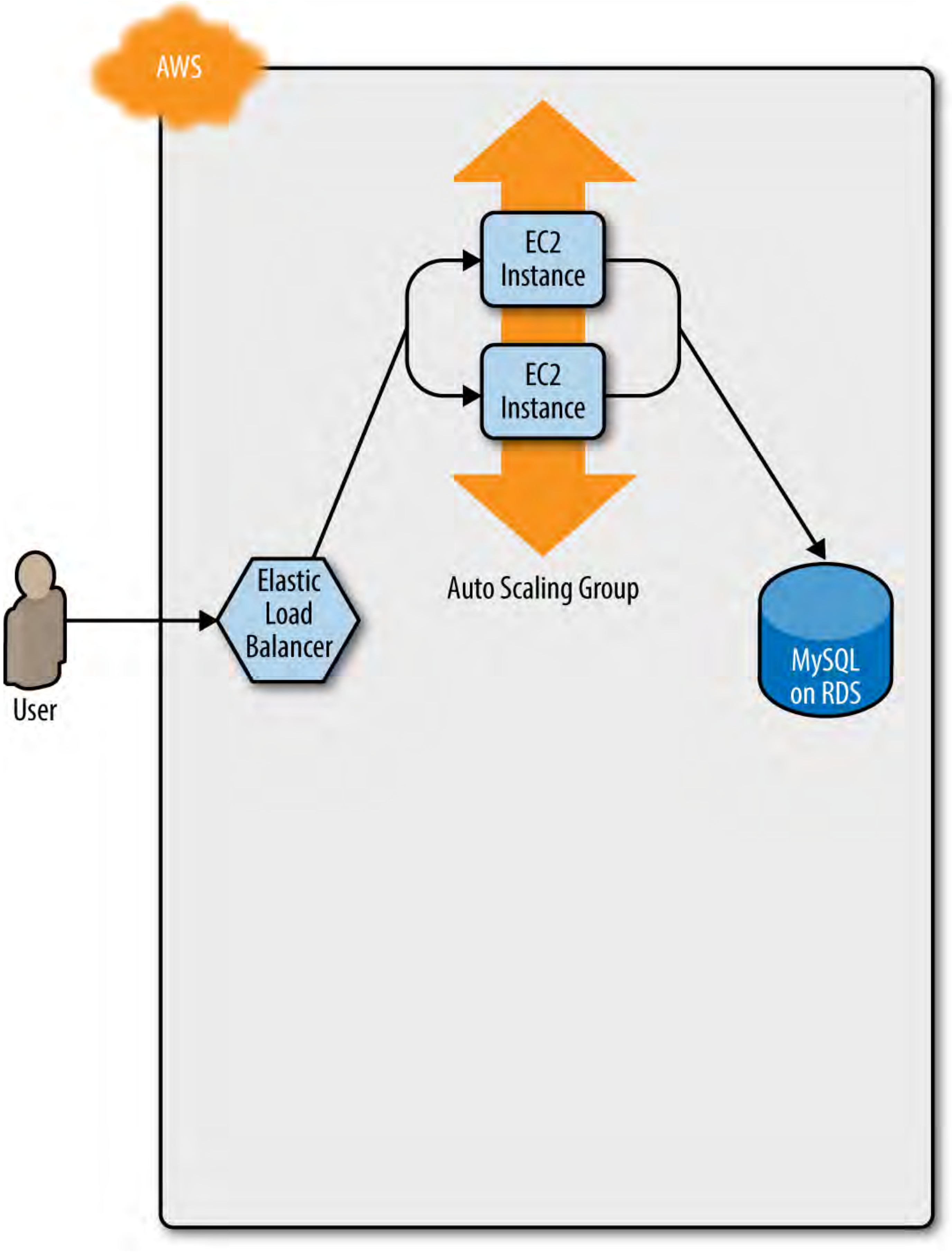


Figure 5-2. Initially, you have the original ASG running v1 of your code

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).

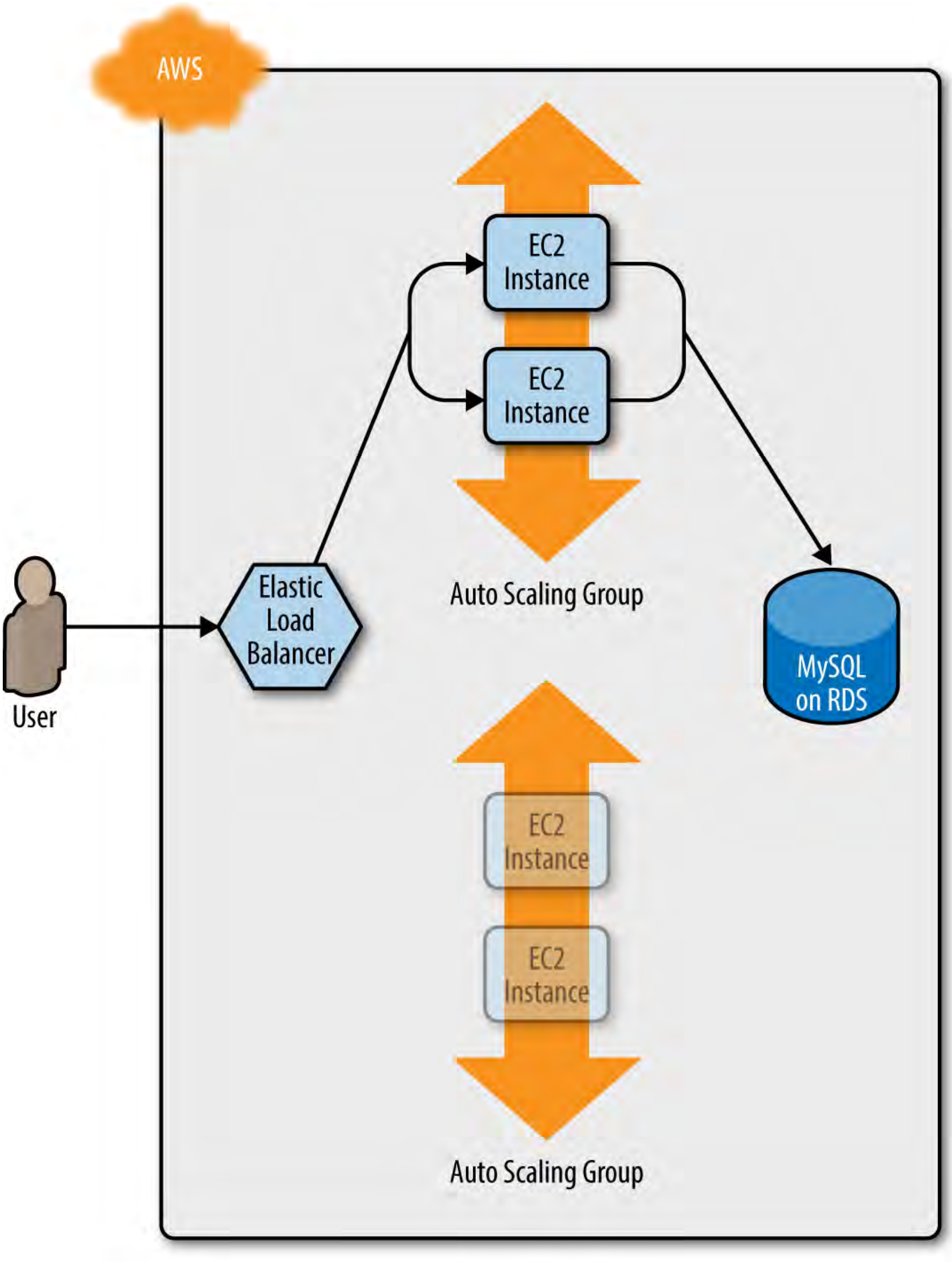


Figure 5-3. Terraform begins deploying the new ASG with v2 of your code

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).

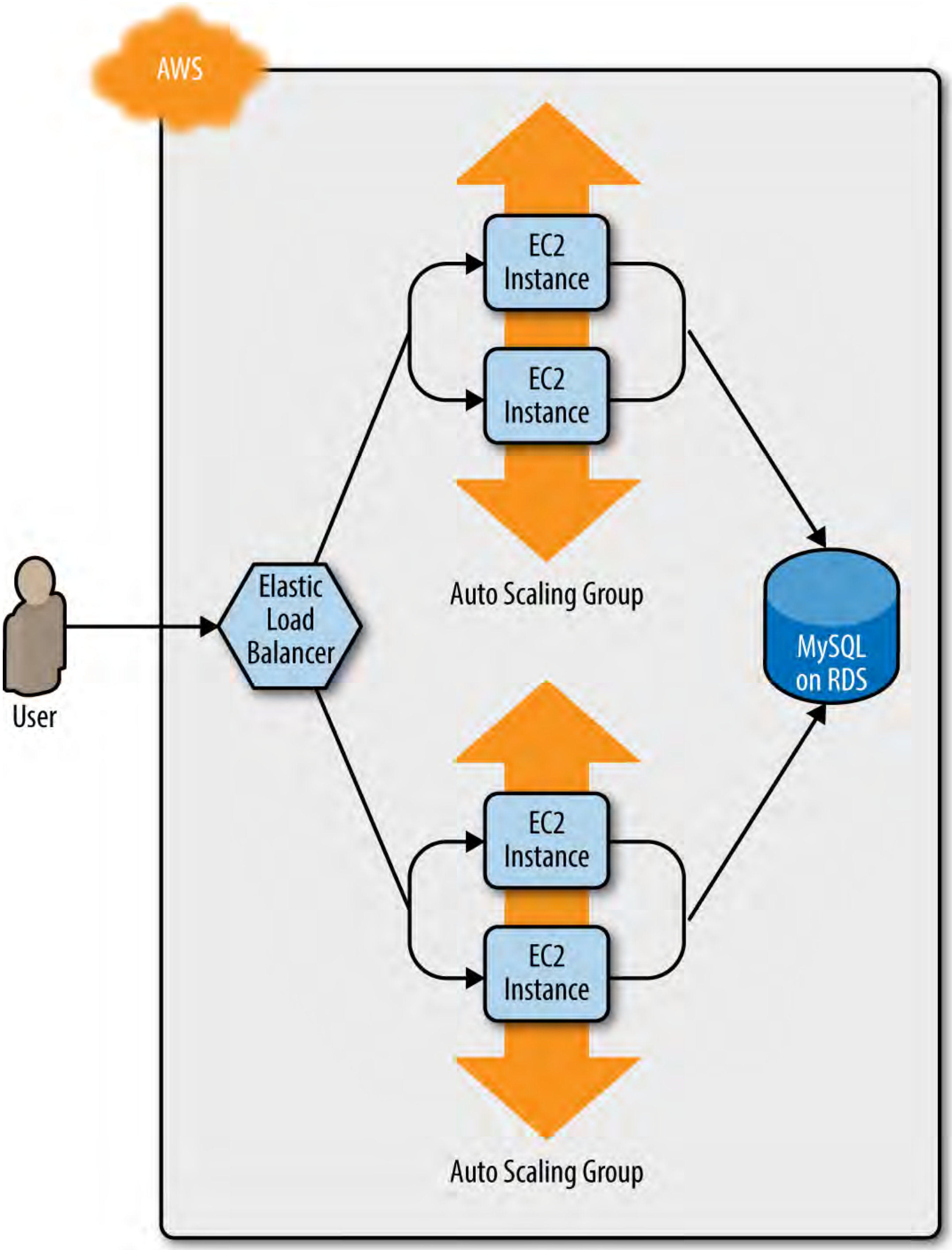


Figure 5-4. The servers in the new ASG boot up, connect to the DB, register in the ALB, and start serving traffic

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).

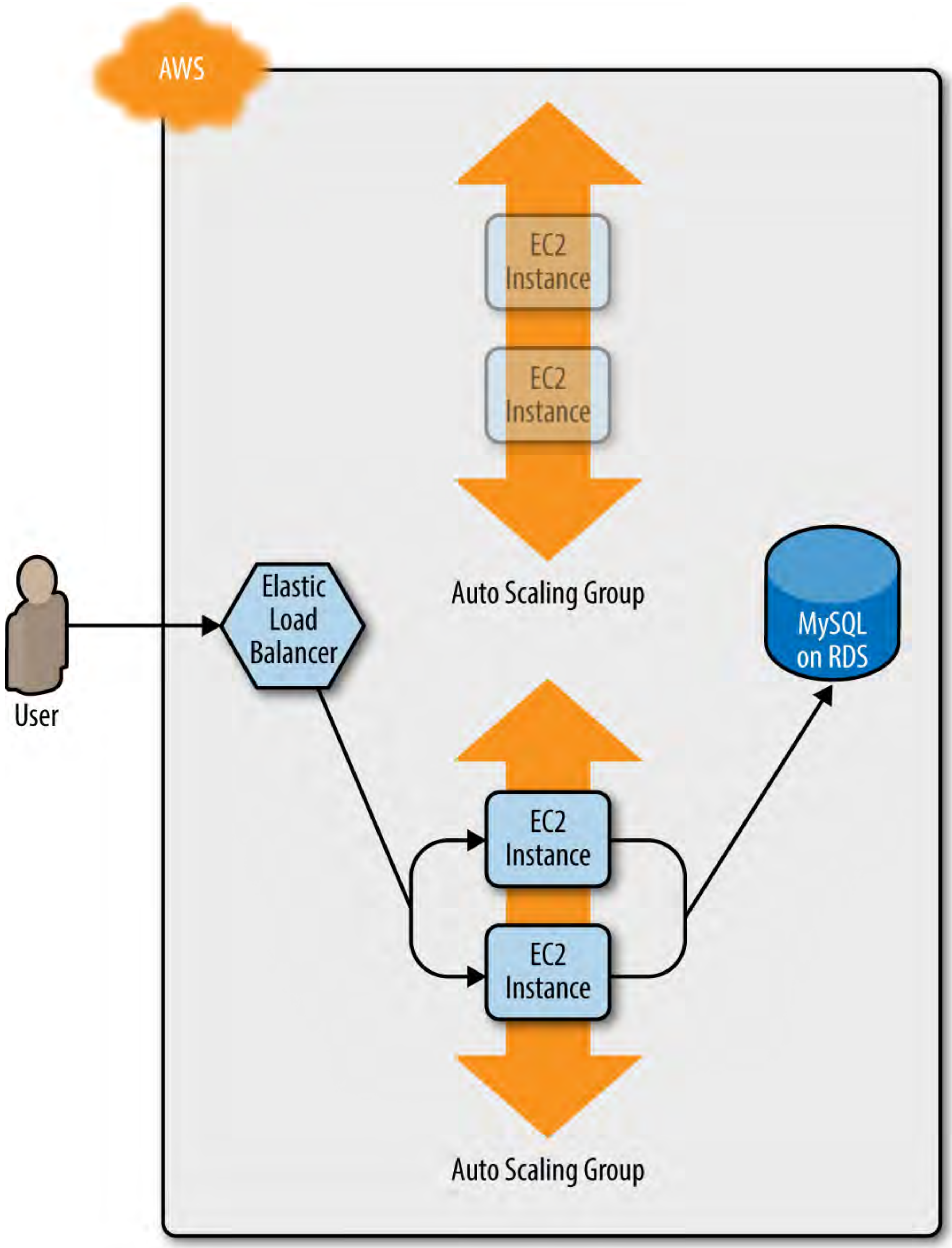


Figure 5-5. The servers in the old ASG begin to shut down

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).

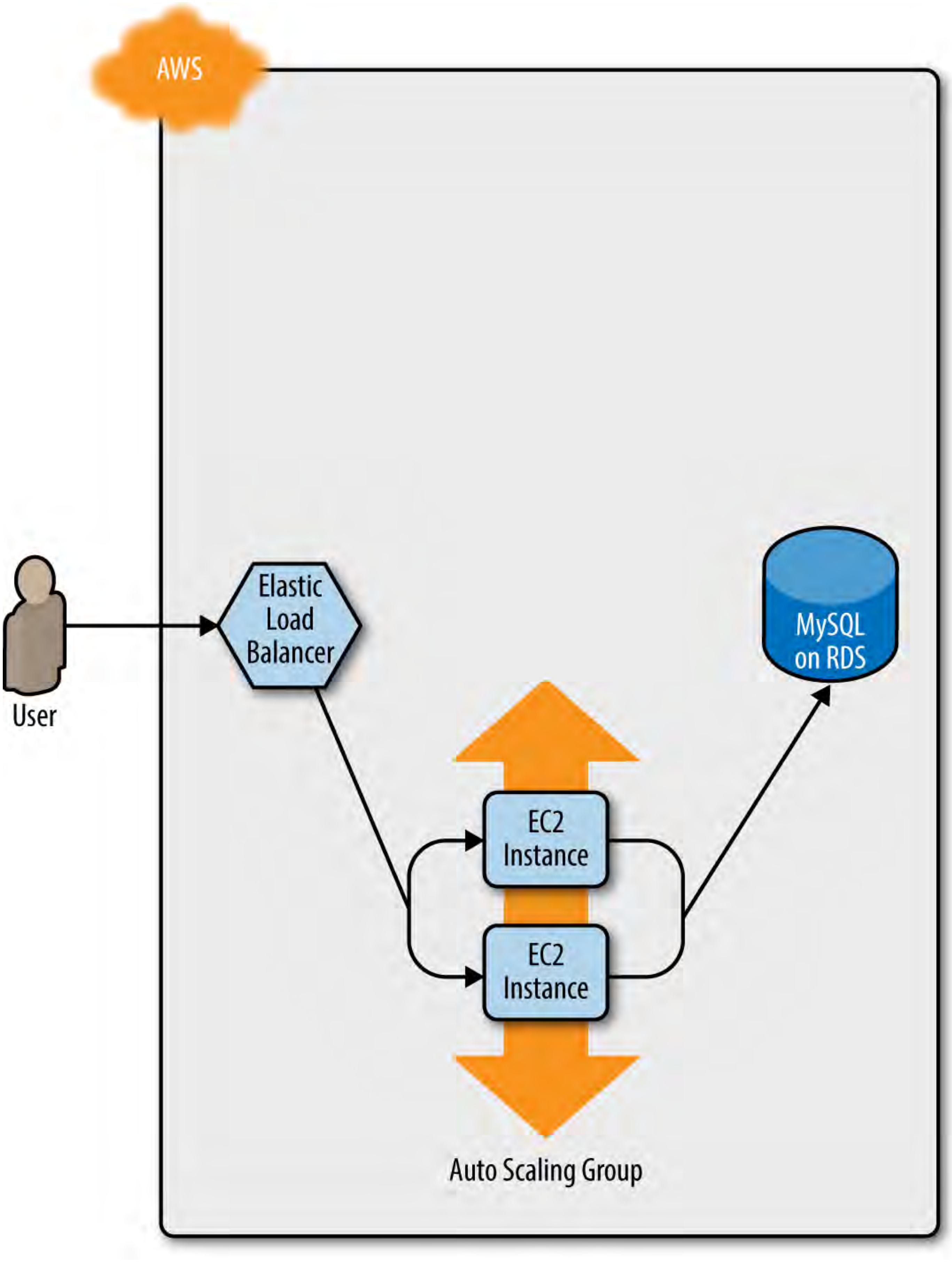
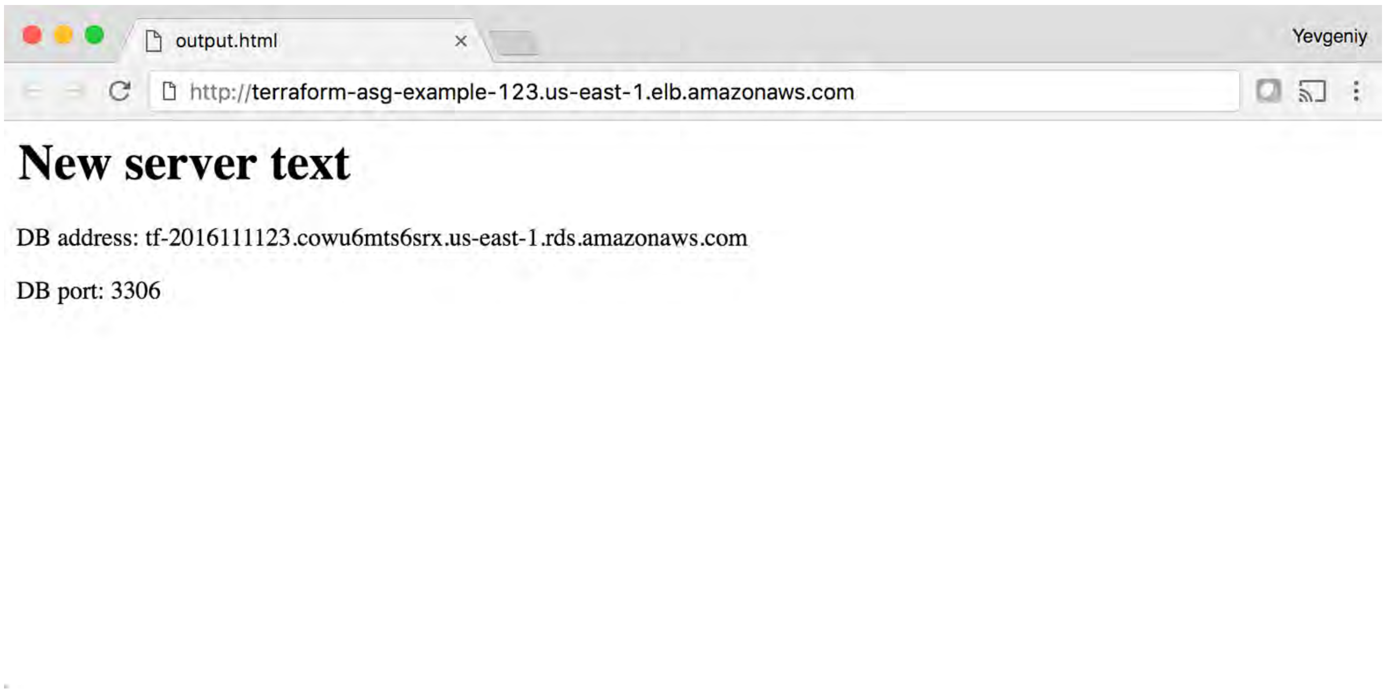


Figure 5-6. Now, only the new ASG remains, which is running v2 of your code

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.


output.html
Yevgeniy
http://terraform-asg-example-123.us-east-1.elb.amazonaws.com

New server text

DB address: tf-2016111123.cowu6mts6srx.us-east-1.rds.amazonaws.com

DB port: 3306

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 with `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:

```
$ 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! Instead of just deleting the trinity bucket, 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:

```
$ 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 planned 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 (see "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.

  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 off-guard 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 asTerraforming, 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 two security groups and the ALB:

```
resource "aws_lb" "example" {
  name               = var.cluster_name
  load_balancer_type = "application"
  subnets            = data.aws_subnet_ids.default.ids
  security_groups    = [aws_security_group.alb.id]
}
```

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" {
  # (...)
}
```

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`.

```
resource "aws_security_group" "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 get used to checking the docs. 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 modules.

Let's now move on to the next chapter, where I'll go over how create not just clean and beautiful modules, but also production-grade modules—the kind of modules you could bet your company on.

---

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*

Chapter 6. Production-grade Terraform code

Building production-grade infrastructure is hard. And stressful. And time consuming. By *production-grade infrastructure*, I mean the servers, data stores, load balancers, security functionality, monitoring and alerting tools, build pipeline, and all the other pieces of your technology that are necessary to run a business. Your company is placing a bet on you: it's betting that your infrastructure won't fall over if traffic goes up, or lose your data if there's an outage, or allow that data to be compromised when hackers try to break in—and if that bet doesn't work out, your company may go out of business. That's whats at stake when I refer to production-grade infrastructure in this chapter.

I've had the opportunity to work with hundreds of companies, and based on all of these experiences, here's roughly how long you should expect your next production-grade infrastructure project to take:

- If you want to deploy a service fully managed by a third party, such as running MySQL using the AWS Relational Database Service (RDS), you can expect it to take you 1-2 weeks to get that service ready for production.
- If you want to run your own stateless distributed app, such as a cluster of Node.js apps that don't store any data locally (e.g., they store all their data in RDS) running on top of an AWS Auto Scaling Group (ASG), that will take roughly twice as long, or about 2-4 weeks to get ready for production.
- If you want to run your own stateful distributed app, such as an Elasticsearch cluster that runs on top of an ASG and stores data on local disks, that will be another order of magnitude increase, or about 2-4 months to get ready for production.
- If you want to build out your entire architecture, including all of your apps, data stores, load balancers, monitoring, alerting, security, and so on, that's another order of magnitude (or two) increase, or about 6 - 36 months of work, with small companies typically being closer to 6 months and larger companies typically taking several years.

Table 6-1 shows a summary of this data in table format.

Table 6-1. How long it takes to build production-grade infrastructure from scratch

Type of infrastructure	Example	Time Estimate
Managed Service	Amazon RDS	1-2 weeks
Self-managed distributed system (stateless)	A cluster of Node.js apps	2-4 weeks
Self-managed distributed system (stateful)	Elasticsearch	2-4 months
Entire architecture	Apps, data stores, load balancers, monitoring, etc	6-36 months

If you haven't gone through the process of building out production-grade infrastructure, you may be surprised by these numbers. I often hear reactions like:

- "How can it possibly take that long?"
- "I can deploy a server on <cloud> in a few minutes. Surely it can't take months to get the rest done!"
- And all too often, from many-an-over-confident-engineer, "I'm sure those numbers apply to other people, but *I* will be able to get this done in a few days."

And yet, anyone who has gone through a major cloud migration or assembled a brand new infrastructure from scratch knows that these numbers, if anything, are optimistic—a best case scenario, really. If you don't have people on your team with deep expertise in building production-grade infrastructure, or if your team is being pulled in a dozen different directions and you can't find the time to focus on it, then it may take you significantly longer.

In this chapter, I'll go over why it takes so long to build production-grade infrastructure, what production-grade really means, and what patterns work best for creating reusable, production-grade modules:

1. Why it takes so long to build production-grade infrastructure

2. The production-grade infrastructure checklist

3. Production-grade infrastructure modules

    a. Small modules

    b. Composable modules

    c. Testable modules

    d. Releasable modules

    e. Beyond Terraform modules

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*.

Why it takes so long to build production-grade infrastructure

Time estimates for software projects are notoriously inaccurate. Time estimates for DevOps projects, doubly so. That quick tweak you thought would take 5 minutes takes up the whole day; the minor new feature that you estimated at a day of work takes two weeks; the app you thought would be in production in two weeks is still not quite there six months later. Infrastructure and DevOps projects, perhaps more than any other type of software, are the ultimate examples of Hofstadter's law[1]:

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

Douglas Hofstadter

I think there are three major reasons for this. The first reason is that DevOps, as an industry, is still in the stone ages. I don't mean that as an insult, but rather, in the sense that the industry is still in its infancy. The terms "cloud computing," "infrastructure as code," and "DevOps" only appeared in the mid to late 2000's and tools like Terraform, Docker, Packer, and Kubernetes were all initially released in the mid to late 2010's. All of these tools and techniques are relatively new and all of them are changing rapidly. That means they are not particularly mature and few people have deep experience with them, so it's no surprise that projects take longer than expected.

The second reason is that DevOps seems to be particularly susceptible to *yak shaving*. If you haven't heard of "yak shaving" before, I assure you, this is a term that you will grow to love (and hate). The best definition I've seen of this term comes from a blog post by Seth Godin[2]:

"*I want to wax the car today.*"

"*Oops, the hose is still broken from the winter. I'll need to buy a new one at Home Depot.*"

"*But Home Depot is on the other side of the Tappan Zee bridge and getting there without my EZPass is miserable because of the tolls.*"

"*But, wait! I could borrow my neighbor's EZPass…*"

"*Bob won't lend me his EZPass until I return the mooshi pillow my son borrowed, though.*"

"*And we haven't returned it because some of the stuffing fell out and we need to get some yak hair to restuff it.*"

And the next thing you know, you're at the zoo, shaving a yak, all so you can wax your car.

Seth Godin

Yak shaving consists of all the tiny, seemingly-unrelated tasks you have to do before you can do the task you originally wanted to do. If you develop software, and especially if you work in the DevOps industry, you've probably seen this sort of thing a thousand times. You go to deploy a fix for a small typo, only to trigger a bug in your app configuration. You roll out the fix for the app configuration and suddenly hit a TLS certificate issue. After spending hours on StackOverflow, you have the TLS issue sorted, try the deployment again, and this time it fails due to some issue with your deployment system. You spend hours digging into that problem and find out it's due to an out of date Linux version. The next thing you know, you're updating the operating system on your entire fleet of servers, all so you can deploy a "quick" one-character typo fix.

DevOps seems to be especially prone to these sorts of yak shave incidents. In part, this is a consequence of the immaturity of DevOps technologies and modern system design, which often involves lots of tight coupling and duplication in the infrastructure. Every change you make in the DevOps world is a bit like trying to pull out a single USB cable from a box of tangled wires—it just tends pull up everything else in the box with it. In part, this is because the term "DevOps" covers an astonishingly broad set of topics: everything from build to deployment to security and so on.

This brings us to the third reason for why DevOps work takes so long. The first two reasons—DevOps is in the stone ages and yak shaving—can be classified as accidental complexity. *Accidental complexity* refers to the problems imposed by the particular tools and processes you've chose, as opposed to *essential complexity*, which refers to the problems inherent to whatever it is that you're working on.[3] For example, if you're using C++ to write stock trading algorithms, then dealing with memory allocation bugs is accidental complexity (had you picked a different programming language with automatic memory management, you wouldn't have this as a problem at all), whereas figuring out an algorithm that can beat the market is essential complexity (you'd have to solve this problem no matter what programming language you picked).

The third reason why DevOps takes so long—the essential complexity of this problem—is that there is a genuinely long checklist of tasks that you must do to prepare infrastructure for production. The problem is that the vast majority of developers don't know about most of the items on the checklist, so when they estimate a project, they forget about a huge number of critical —and time-consuming—details. This checklist is the focus of the next section.

The production-grade infrastructure checklist

Here's a fun experiment: go around your company and ask, "what are the requirements for going to production?" In most companies, if you ask this question to five people, you'll get five different answers. One person will mention the need for metrics and alerts; another will talk about capacity planning and high availability; someone else will go on a rant about automated tests and code reviews; yet another person will bring up encryption, authentication, and server hardening; if you're lucky, someone might remember to bring up data backups and log aggregation. Most companies do not have a clear definition of the requirements for going to production, which means each piece of infrastructure is deployed a little differently, and may be missing some critical functionality.

To help improve this situation, I'd like to share with you the production-grade infrastructure checklist, as shown in Table 6-2. This list covers most of the key items you'll need to consider to deploy infrastructure to production.

Table 6-2. The production-grade infrastructure checklist

Task	Description	Example Tools
Install	Install the software binaries and all dependencies.	Bash, Chef, Ansible, Puppet
Configure	Configure the software at runtime. Includes port settings, TLS certs, service discovery, leaders, followers, replication, etc.	Bash, Chef, Ansible, Puppet
Provision	Provision the infrastructure. Includes servers, load balancers, network configuration, firewall settings, IAM permissions, etc.	Terraform, CloudFormation
Deploy	Deploy the service on top of the infrastructure. Roll out updates with no downtime. Includes blue-green, rolling, and canary deployments.	Terraform, CloudFormation, Kubernetes, ECS
High availability	Withstand outages of individual processes, servers, services, data centers, and regions.	Multi data center, multi-region, replication, auto scaling, load balancing
Scalability	Scale up and down in response to load. Scale horizontally (more servers) and/or vertically (bigger servers).	Auto scaling, replication, sharding, caching, divide and conquer
Performance	Optimize CPU, memory, disk, network, and GPU usage. Includes query tuning, benchmarking, load testing, and profiling.	Dynatrace, valgrind, VisualVM, ab, Jmeter
Networking	Configure static and dynamic IPs, ports, service discovery, firewalls, DNS, SSH access, and VPN access.	VPCs, firewalls, routers, DNS registrars, OpenVPN
Security	Encryption in transit (TLS) and on disk, authentication, authorization, secrets management, server hardening.	ACM, Let’s Encrypt, KMS, Cognito, Vault, CIS
Metrics	Availability metrics, business metrics, app metrics, server metrics, events, observability, tracing, and alerting.	CloudWatch, DataDog, New Relic, Honeycomb

Task	Description	Example Tools
Logs	Rotate logs on disk. Aggregate log data to a central location.	CloudWatch Logs, ELK, Sumo Logic, Papertrail
Backup and Restore	Make backups of DBs, caches, and other data on a scheduled basis. Replicate to separate region/account.	RDS, ElastiCache, replication
Cost optimization	Pick proper instance types, use spot and reserved instances, use auto scaling, and nuke unused resources.	Auto scaling, spot instances, reserved instances
Documentation	Document your code, architecture, and practices. Create playbooks to respond to incidents.	READMEs, wikis, Slack
Tests	Write automated tests for your infrastructure code. Run tests after every commit and nightly.	Terratest, inspec, serverspec, kitchen-terraform

Most developers are aware of the first few tasks: install, configure, provision, and deploy. It's all the ones that come after it that catch people off guard. For example, did you think through the resilience of your service and what happens if a server goes down? Or a load balancer goes down? Or an entire data center goes dark? Networking tasks are also notoriously tricky: setting up VPCs, VPNs, service discovery, and SSH access are all essential tasks that can take months, and yet are often entirely left out of many project plans and time estimates. Security tasks, such as encrypting data in transit using TLS, dealing with authentication, and figuring out how to store secrets are also often forgotten about until the last minute.

Every time you're working on a new piece of infrastructure, go through this checklist. Not every single piece of infrastructure needs every single item on the list, but you should consciously and explicitly document which items you've implemented, which ones you've decided to skip, and why.

Production-grade infrastructure modules

Now that you know the list of tasks you have to do for each piece of infrastructure, let's talk about the best practices for building reusable modules to implement these tasks. Here are the topics we'll cover:

1. Small modules
2. Composable modules
3. Testable modules
4. Releasable modules
5. Beyond Terraform modules

Small modules

Developers who are new to Terraform, and infrastructure as code in general, often define all of their infrastructure for all of their environments (dev, stage, prod, etc) in a single file or single module. As discussed in "Isolating State Files", this is a bad idea. In fact, I'll go even further, and make the following claim: large modules—modules that contain more than a few hundred lines of code or that deploy more than a few closely related pieces of infrastructure—should be considered harmful.

Here are just a few of the downsides of large modules:

Large modules are slow

If all your infrastructure is defined in one Terraform module, running any command will take a long time. I’ve seen modules get so large that `terraform plan` takes 5–6 minutes to run!

Large modules are insecure

If all your infrastructure is managed in a single large module, then to change anything you need permissions to access everything. That means that almost every user has to be an admin, which goes against the principle of least privilege.

Large modules are risky

If all your eggs are in one basket, then a mistake anywhere could break everything. You might be making a minor change to a frontend app in staging, but due to a typo or running the wrong command, you delete the production DB.

Large modules are hard to understand

The more code you have in one place, the harder it is for any one person to understand it all. And when you don’t understand the infrastructure you’re dealing with, you end up making costly mistakes.

Large modules are hard to review

Reviewing a module that consists of several dozen lines of is easy; reviewing a module that consists of several thousand lines of code is nearly impossible. Moreover, `terraform plan` not only takes longer to run, but if the output of the `plan` command is several thousand lines, no one will bother to read it. And that means no one will notice that one little red line that means your database is being deleted.

Large modules are hard to test

Testing infrastructure code is hard; testing a large amount of infrastructure code is nearly impossible. We’ll come back to this point in Chapter 7.

In short, you should build your code out of small modules that each do one thing. This is not a new or controversial insight. You’ve probably heard it many times before, albeit in slightly different contexts, such as this version from *Clean Code*[4]:

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.

Robert C. Martin

Imagine you were using a general purpose programming language such as Java or Python or Ruby, and came across a single, massive function that was 20,000 lines long:

```
def huge_function(data_set)
  x_pca = PCA(n_components=2).fit_transform(X_train)
  clusters = clf.fit_predict(X_train)
  ax = plt.subplots(1, 2, figsize=(4))
  ay = plt.subplots(0, 2, figsize=(2))
  fig = plt.subplots(3, 4, figsize=(5))
  fig.subplots_adjust(top=0.85)

  predicted = svc_model.predict(X_test)
  images_and_predictions = list(zip(images_test, predicted))

  for x in 0..xlimit
    ax[0].scatter(X_pca[x], X_pca[1], c=clusters)
    ax[0].set_title('Predicted Training Labels')
    ax[1].scatter(X_pca[x], X_pca[1], c=y_train)
    ax[1].set_title('Actual Training Labels')
    ax[2].scatter(X_pca[x], X_pca[1], c=clusters)
  end

  for y in 0..ylimit
    ay[0].scatter(X_pca[y], X_pca[1], c=clusters)
    ay[0].set_title('Predicted Training Labels')
    ay[1].scatter(X_pca[y], X_pca[1], c=y_train)
    ay[1].set_title('Actual Training Labels')
    ay[2].scatter(X_pca[y], X_pca[1], c=clusters)
  end

  #
  # ... 20,000 more lines...
  #
end
```

You immediately know this is a code smell and that the better approach is to refactor it into a number of small, standalone functions that each do one thing:

```
def calculate_images_and_predictions(images_test, predicted)
  x_pca = PCA(n_components=2).fit_transform(X_train)
  clusters = clf.fit_predict(X_train)
  ax = plt.subplots(1, 2, figsize=(4))
  fig = plt.subplots(3, 4, figsize=(5))
  fig.subplots_adjust(top=0.85)

  predicted = svc_model.predict(X_test)
  return list(zip(images_test, predicted))
end

def process_x_coords(ax)
  for x in 0..xlimit
    ax[0].scatter(X_pca[x], X_pca[1], c=clusters)
    ax[0].set_title('Predicted Training Labels')
    ax[1].scatter(X_pca[x], X_pca[1], c=y_train)
    ax[1].set_title('Actual Training Labels')
    ax[2].scatter(X_pca[x], X_pca[1], c=clusters)
  end

  return ax
end

def process_y_coords(ax)
  for y in 0..ylimit
    ay[0].scatter(X_pca[y], X_pca[1], c=clusters)
    ay[0].set_title('Predicted Training Labels')
    ay[1].scatter(X_pca[y], X_pca[1], c=y_train)
    ay[1].set_title('Actual Training Labels')
    ay[2].scatter(X_pca[y], X_pca[1], c=clusters)
  end

  return ay
end

#
# ... Lots more small functions...
#
```

You should use the same strategy with Terraform. Imagine you came across the architecture shown in Figure 6-1.

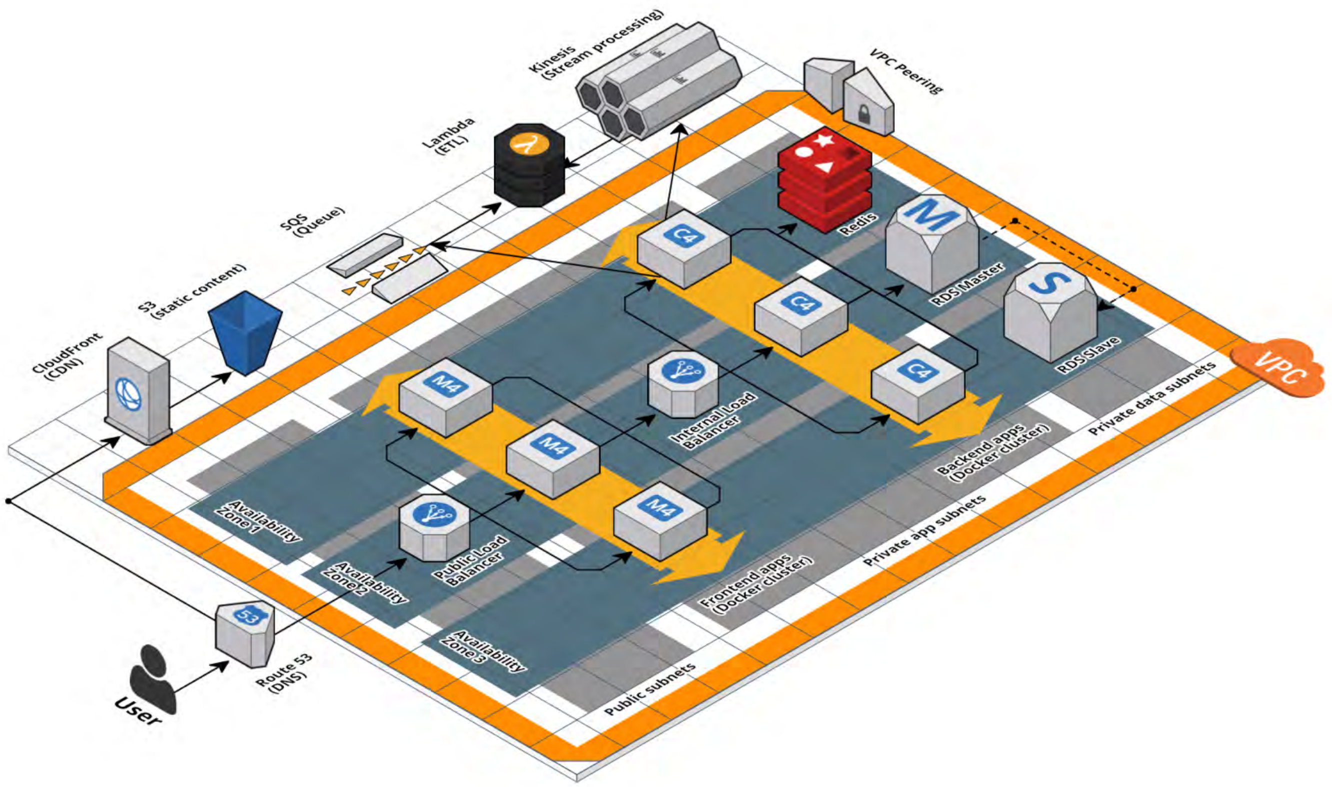


Figure 6-1. A relatively complicated AWS architecture

If this architecture was defined in a single, massive Terraform module that was 20,000 lines long, you should immediately think of it as a code smell. The better approach is to refactor into a number of small, standalone modules that each do one thing, as shown in Figure 6-2.

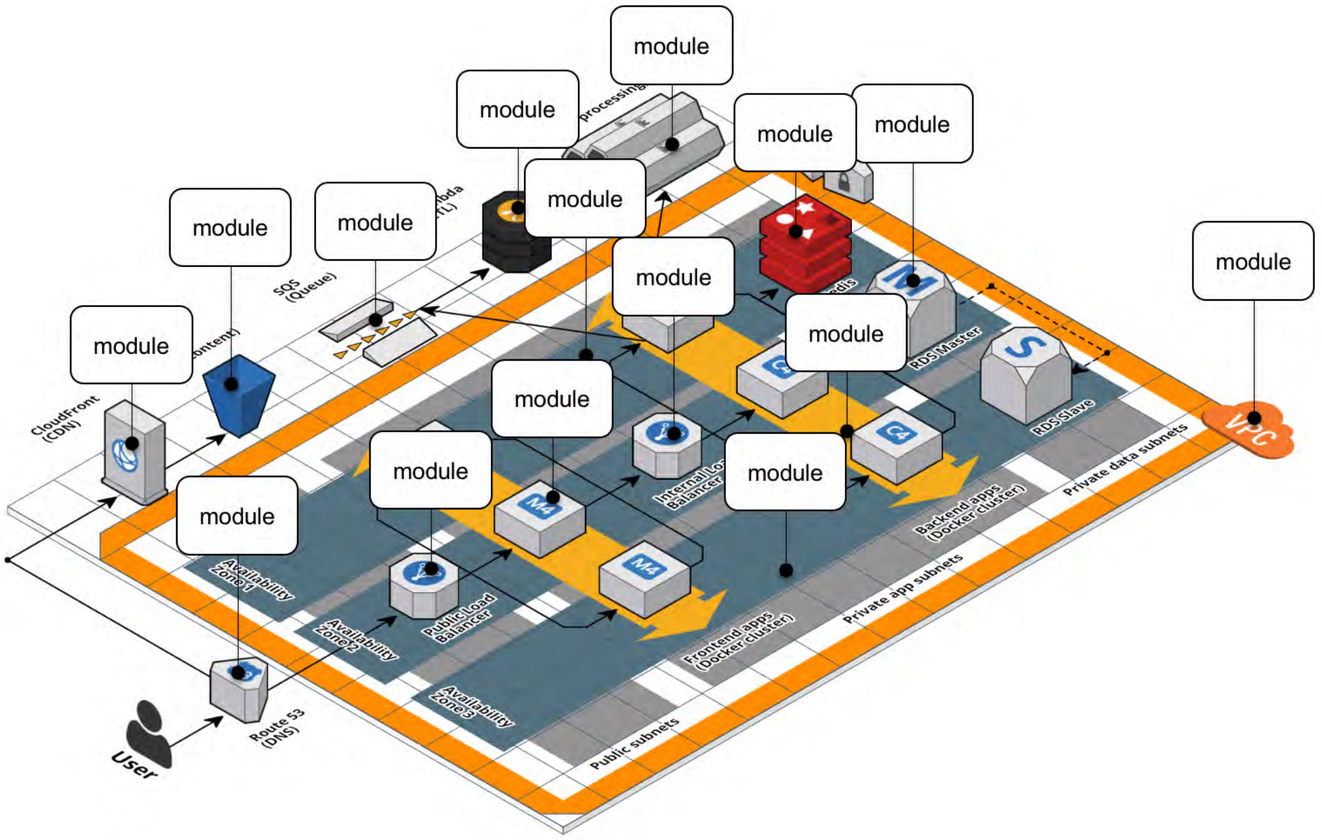


Figure 6-2. A relatively complicated AWS architecture refactored into many small modules

The `webserver-cluster` module you've built up to this point is starting to get a bit big, and it's handling three somewhat unrelated tasks:

1. **Auto Scaling Group (ASG)**: the `webserver-cluster` module deploys an ASG that can do a zero-downtime, rolling deployment.
2. **Application Load Balancer (ALB)**: the `webserver-cluster` deploys an ALB.
3. **Hello, World app**: the `webserver-cluster` module also deploys a simple "Hello, World" app.

Let's refactor the code accordingly into three smaller modules:

1. **modules/cluster/asg-rolling-deploy**: a generic, reusable, standalone module for deploying an ASG that can do a zero-downtime, rolling deployment.
2. **modules/networking/alb**: a generic, reusable, standalone module for deploying an ALB.
3. **modules/networking/hello-world-app**: a module specifically for deploying the "Hello, World" app.

Before getting started, make sure to run `terraform destroy` on any `webserver-cluster` deployments you currently have. Once that's done, create a new folder at *modules/cluster/asg-rolling-deploy* and move (i.e., cut and paste) the following resources from *module/services/webserver-cluster/main.tf* to *modules/cluster/asg-rolling-deploy/main.tf*:

- `aws_launch_configuration`
- `aws_autoscaling_group`
- `aws_autoscaling_schedule` (both of them)
- `aws_security_group` (for the instances, but not for the ALB)
- `aws_security_group_rule` (both of the rules for the instances, but not those for the ALB)
- `aws_cloudwatch_metric_alarm` (both of them)

Next, move the following variables from to *module/services/webserver-cluster/variables.tf* to *modules/cluster/asg-rolling-deploy/variables.tf*:

- `cluster_name`
- `ami`
- `instance_type`
- `min_size`
- `max_size`
- `enable_autoscaling`
- `custom_tags`
- `server_port`

Let's now move on to the ALB module. Create a new folder at *modules/networking/alb* and move the following resources from *module/services/webserver-cluster/main.tf* to *modules/networking/alb/main.tf*:

- `aws_lb`
- `aws_lb_listener`
- `aws_security_group` (the one for the ALB, but not for the instances)
- `aws_security_group_rule` (both of the rules for the ALB, but not for the instances)

Create *modules/networking/alb/variables.tf* and define a single variable within:

```
variable "alb_name" {
  description = "The name to use for this ALB"
  type        = string
}
```

Use this variable as the `name` argument of the `aws_lb` resource:

```
resource "aws_lb" "example" {
  name               = var.alb_name
  load_balancer_type = "application"
  subnets            = data.aws_subnet_ids.default.ids
  security_groups    = [aws_security_group.alb.id]
}
```

And the `name` argument of the `aws_security_group` resource:

```
resource "aws_security_group" "alb" {
  name = var.alb_name
}
```

This is a lot of code to shuffle around, so feel free to use the code examples for this chapter from *https://github.com/brikis98/terraform-up-and-running-code*.

Composable modules

You now have two small modules that do one thing and do it well. How do you make them work together? How do you build modules that are reusable and composable? This question is not unique to Terraform, but something programmers have been thinking about for decades. To quote Doug McIlroy[5], the original developer of Unix pipes and a number of other Unix tools, including`diff`, `sort`, `join`, and `tr`:

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together.

Doug McIlroy

One way to do this is through *function composition*, where you can take the outputs of one function and pass them as the inputs to another. For example, if you had the following small functions in Ruby:

```
# Simple function to do addition
def add(x, y)
  return x + y
end

# Simple function to do subtraction
def sub(x, y)
  return x - y
end

# Simple function to do multiplication
def multiply(x, y)
  return x * y
end
```

You can use function composition to put them together by taking the outputs from `add` and `sub` and passing them as the inputs to `multiply`:

```
# Complex function that composes several simpler functions
def do_calculation(x, y)
  return multiply(add(x, y), sub(x, y))
end
```

One of the main ways to make functions composable is to minimize *side effects*: that is, where possible, avoid reading state from the outside world, and instead have it passed in via input parameters, and avoid writing state to the outside world, and instead return the result of your computations via output parameters. Minimizing side effects is one of the core tenets of functional programming because it makes the code easier to reason about, easier to test, and easier to reuse. The reuse story is particularly compelling, as function composition allows you to gradually build up more complicated functions by combining simpler functions.

Although you can’t avoid side effects when working with infrastructure code, you can still follow the same basic principles in your Terraform modules: pass everything in through input variables, returning everything through output variables, and build more complicated modules by combining simpler modules.

Open up *modules/cluster/asg-rolling-deploy/variables.tf* and add four new input variables:

```
variable "subnet_ids" {
  description = "The subnet IDs to deploy to"
  type        = list(string)
}

variable "target_group_arns" {
  description = "The ARNs of load balancer target groups in which to register Instances"
  type        = list(string)
  default     = []
}

variable "health_check_type" {
  description = "The type of health check to perform. Must be one of: EC2, ELB."
  type        = string
  default     = "EC2"
}

variable "user_data" {
  description = "The User Data script to run in each Instance at boot"
  type        = string
  default     = ""
}
```

The first variable, `subnet_ids`, tells the `asg-rolling-deploy` module which subnets to deploy into. Whereas the `webserver-cluster` module was hard-coded to deploy into the Default VPC and subnets, by exposing the `subnet_ids`variable, you allow this module to be used with any VPC or subnets. The next two variables, `target_group_arns` and `health_check_type` configure how the ASG integrates with load balancers. Whereas the `webserver-cluster` module had a built-in ALB, the `asg-rolling-deploy` module is meant to be a generic module, so exposing the load balancer settings as input variables allows you to use the ASG with a wide variety of use cases: e.g., no load balancer, one ALB, multiple NLBs, etc.

Take these three input variables and pass them through to the `aws_autoscaling_group` resource in *modules/cluster/asg-rolling-deploy/main.tf*, replacing the previously hard-coded settings that are referencing resources (e.g., the ALB) and data sources (e.g., `aws_subnet_ids`) that we did not bother copying into the `asg-rolling-deploy` module (the code snippet below is slightly truncated for readability):

```
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  = var.subnet_ids

  # Configure integrations with a load balancer
  target_group_arns    = var.target_group_arns
  health_check_type    = var.health_check_type

  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

  # (...)
}
```

The fourth variable, `user_data`, is for passing in a User Data script. Whereas the `webserver-cluster` module had a hard-coded User Data script, and could only be used to deploy the “Hello, World” app, by taking in a User Data script as an input variable, the `asg-rolling-deploy` module can be used to deploy any app across an ASG. Pass this `user_data` variable through to the `aws_launch_configuration` resource (replacing the reference to the `template_file` data source we didn’t copy into the `asg-rolling-deploy`module):

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

  # Required when using a launch configuration with an auto scaling group.
  # https://www.terraform.io/docs/providers/aws/r/launch_configuration.html
  lifecycle {
    create_before_destroy = true
  }
}
```

You'll also want to add a couple useful output variables to *modules/cluster/asg-rolling-deploy/outputs.tf*:

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

output "instance_security_group_id" {
  value       = aws_security_group.instance.id
  description = "The ID of the EC2 Instance Security Group"
}
```

Outputting this data makes the `asg-rolling-deploy` module even more reusable, as consumers of the module can use these outputs to add new behaviors, such as attaching custom rules to the Security Group.

For similar reasons, you should add several output variables to*modules/networking/alb/outputs.tf*:

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

output "alb_http_listener_arn" {
  value       = aws_lb_listener.http.arn
  description = "The ARN of the HTTP listener"
}

output "alb_security_group_id" {
  value       = aws_security_group.alb.id
  description = "The ALB Security Group ID"
}
```

You'll see how these are used shortly.

The last step is to convert the `webserver-cluster` module into a `hello-world-app` module that can deploy a "Hello, World" app using the `asg-rolling-deploy` and `alb` modules. To do this, rename *module/services/webserver-cluster*to *module/services/hello-world-app*. After all the changes in the previous steps, you should only have the following resources and data sources left in *module/services/hello-world-app/main.tf*:

- `template_file` (for User Data)
- `aws_lb_target_group`
- `aws_lb_listener_rule`
- `terraform_remote_state` (for the DB)
- `aws_vpc`
- `aws_subnet_ids`

Add the following variable to *modules/services/hello-world-app/variables.tf*:

```
variable "environment" {
  description = "The name of the environment we're deploying to"
  type        = string
}
```

Now, add `asg-rolling-deploy` module you created earlier to the `hello-world-app` module to deploy an ASG:

```
module "asg" {
  source = "../../cluster/asg-rolling-deploy"

  cluster_name  = "hello-world-${var.environment}"
  ami           = var.ami
  user_data     = data.template_file.user_data.rendered
  instance_type = var.instance_type

  min_size           = var.min_size
  max_size           = var.max_size
  enable_autoscaling = var.enable_autoscaling

  subnet_ids        = data.aws_subnet_ids.default.ids
  target_group_arns = [aws_lb_target_group.asg.arn]
  health_check_type = "ELB"

  custom_tags = var.custom_tags
}
```

And add the `alb` module you created earlier to the `hello-world-app` module to deploy an ALB:

```
module "alb" {
  source = "../../networking/alb"

  alb_name   = "hello-world-${var.environment}"
  subnet_ids = data.aws_subnet_ids.default.ids
}
```

Note the use of the the input variable `environment` as a way to enforce a naming convention, so all of your resources will be namespaced based on the environment (e.g., `hello-world-stage`, `hello-world-prod`). This code also sets the new `subnet_ids`, `target_group_arns`, `health_check_type`, and `user_data` variables you added earlier to appropriate values.

Next, you'll need to configure the ALB Target Group and Listener Rule for this app. Update the `aws_lb_target_group` resource in *modules/services/hello-world-app/main.tf* to use `environment` in its `name`:

```
resource "aws_lb_target_group" "asg" {
  name     = "hello-world-${var.environment}"
  port     = var.server_port
  protocol = "HTTP"
  vpc_id   = data.aws_vpc.default.id

  health_check {
    path                = "/"
    protocol            = "HTTP"
    matcher             = "200"
    interval            = 15
    timeout             = 3
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
}
```

And update the `listener_arn` parameter of the `aws_lb_listener_rule`resource to point at the `alb_http_listener_arn` output of the ALB module:

```
resource "aws_lb_listener_rule" "asg" {
  listener_arn = module.alb.alb_http_listener_arn
```

```
    priority     = 100

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

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

Finally, pass through the important outputs from the `asg-rolling-deploy` and `alb` modules as outputs of the `hello-world-app` module:

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

  output "asg_name" {
    value       = module.asg.asg_name
    description = "The name of the Auto Scaling Group"
  }

  output "instance_security_group_id" {
    value       = module.asg.instance_security_group_id
    description = "The ID of the EC2 Instance Security Group"
  }
```

This is function composition at work: you're building up more complicated behavior (a "Hello, World" app) from simpler parts (ASG and ALB modules). A fairly common pattern you'll see with Terraform is that you'll have at least two types of modules:

1. **Generic modules**: Modules such as `asg-rolling-deploy` and `alb` are the basic building blocks of your code, reusable across wide variety of use cases. You've already seen them used to deploy a "Hello, World" app, but you could also use the exact same modules to deploy, for example, an ASG to run a Kafka cluster, or a completely standalone ALB that can distribute load across many different apps (running one ALB for all apps is cheaper than one ALB for each app).
2. **Use-case specific modules**: Modules such as `hello-world-app` combine multiple generic modules to serve one specific use case, such as deploying the "Hello, World" app.

In real-world usage, you may need to break your modules down even more to support better composition and reuse. For example, you can find a set of open-source, reusable modules for running HashiCorp Consul at*https://github.com/hashicorp/terraform-aws-consul*. Consul is an open-source, distributed, key-value store that sometimes needs to listen for network requests on a large number of different ports (server RPC, CLI RPC, Serf WAN, HTTP API, DNS, etc), so it ends up with about ~20 security group rules. In the `terraform-aws-consul` repo, these security group rules are defined in a standalone `consul-security-group-rules` module[6].

To understand why, it's useful to understand how Consul is deployed. One common use case is to deploy Consul as the data store for HashiCorp Vault, an open-source, distributed secrets store you can use to securely store passwords, API keys, TLS certs, etc. You can find a set of open-source, reusable modules for running Vault at *https://github.com/hashicorp/terraform-aws-vault/*, including the diagram in Figure 6-3, which shows the typical production architecture for Vault.

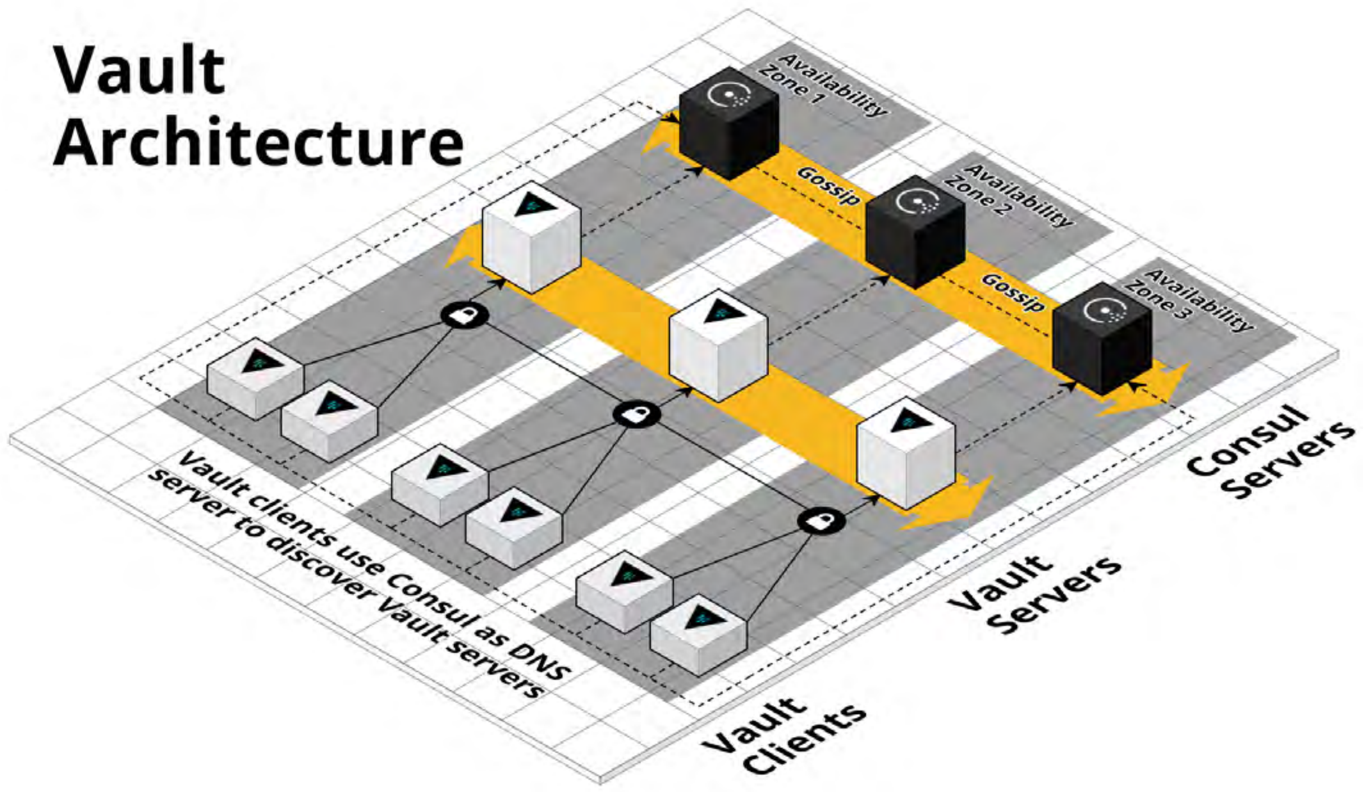


Figure 6-3. HashiCorp Vault and Consul architecture

In production, you typically run Vault on one ASG of 3-5 servers (deployed via the `vault-cluster` module[7]), and Consul on a separate ASG of 3-5 servers (deployed via the `consul-cluster` module[8]), so each can be scaled and secured separately. However, in a staging environment, running this many servers is overkill, and to save money, you may want to run both Vault and Consul on the same ASG, perhaps of size 1, deployed by the `vault-cluster` module. If all of Consul's security group rules were defined in the `consul-cluster` module, then you'd have no way to reuse them (without copy/pasting the 20+ rules) if you're actually deploying Consul using the `vault-cluster` module. However, because the rules are defined in separate `consul-security-group-rules` module, you can attach them to the `vault-cluster` or almost any other type of cluster.

This sort of breakdown would overkill for a simple "Hello, World" app, but for complicated real-world infrastructure, breaking out security group rules, IAM policies, and other cross-cutting concerns into separate, standalone modules is often essential to supporting different deployment patterns. We've used this pattern not only with Consul and Vault, but also the ELK stack (i.e., run Elasticsearch, Logstash, and Kibana on separate clusters in prod but the same cluster in dev), the Confluent Platform (i.e., run Kafka, ZooKeeper, REST Proxy, and Schema Registry on separate clusters in prod but the same cluster in dev), the TICK stack (i.e., run Telegraf, InfluxDB, Chronograf, Kapacitor on separate clusters in prod or the same cluster in dev), and so on.

Testable modules

At this stage, you've written a whole lot of code in the form of three modules: `asg-rolling-deploy`, `alb`, and `hello-world-app`. The next step is to check that your code actually works!

The modules you've created aren't root modules meant to be deployed directly. To deploy them, you need to write some Terraform code to plug in the arguments you want, set up the `provider`, configure the `backend`, and so on. A great way to do this is to create an `examples` folder that, as the name suggests, shows examples of how to use your modules. Let's try it out.

Create `examples/asg/main.tf` with the following contents:

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

module "asg" {
  source = "../../modules/cluster/asg-rolling-deploy"

  cluster_name  = var.cluster_name
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  min_size           = 1
  max_size           = 1
  enable_autoscaling = false
  subnet_ids        = data.aws_subnet_ids.default.ids
}

data "aws_vpc" "default" {
  default = true
}

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

This bit of code uses the `asg-rolling-deploy` module to deploy an ASG of size 1. Try it out by running `terraform init` and `terraform apply` and checking to see that it runs without errors and actually spins up an ASG. Now, add in a `README.md` file with these instructions, and suddenly, this tiny little example takes on a whole lot of power. In just several files and lines of code, you now have:

1. **A manual test harness**: you can use this example code while working on the `asg-rolling-deploy` module to repeatedly deploy and undeploy it by manually running `terraform apply` and `terraform destroy` to check that it works as you expect.
2. **An automated test harness**: as you'll see in Chapter 7, this example code is also how you'll create automated tests for your modules. I typically recommend that tests go into the `test` folder.
3. **Executable documentation**: if you commit this example (including the `README.md`) into version control, then other members of your team can find it, use it to understand how your module works, and take the module for a spin without writing a line of code. It's both a way to teach the rest of your team and, if you add automated tests around it, a way to ensure that your teaching materials always work as expected.

Every Terraform module you have in the `modules` folder should have a corresponding example in the `examples` folder. And every example in the `examples` folder should have a corresponding test in the `test` folder. In fact, you'll most likely have multiple examples (and therefore, multiple tests) for each module, with each example showing different configurations and permutations of how that module can be used. For example, you may want to add other examples for the `asg-rolling-deploy` module that show (a) how to use it with auto scaling policies, (b) how to hook up load balancers to it, (c) how to set custom tags, and so on.

Putting this all together, the folder structure for a typical `modules` repo will look something like this:

```
modules
 └ examples
   └ alb
   └ asg-rolling-deploy
     └ one-instance
     └ auto-scaling
     └ with-load-balancer
     └ custom-tags
   └ hello-world-app
   └ mysql
 └ modules
   └ alb
   └ asg-rolling-deploy
   └ hello-world-app
   └ mysql
 └ test
   └ alb
   └ asg-rolling-deploy
   └ hello-world-app
   └ mysql
```

As an exercise for the reader, I leave it up to you to turn the RDS code you wrote earlier into a MySQL module, and to add lots of examples for the `alb`, `asg-rolling-deploy`, `mysql`, and `hello-world-app` modules.

A great practice to follow when developing a new module is to write the example code *first*, before you write even a line of module code. If you start with the implementation, it's too easy to get lost deep in the implementation details, and by the time you resurface and make it back to the API, you end up with a module that is unintuitive and hard to use. On the other hand, if you start with the example code, you're free to think through the ideal user experience and come up with a clean API for your module, and then work backwards to the implementation. Since the example code is the primary way of testing modules anyway, this is a form of *Test Driven Development (TDD)* (we'll dive more into testing in Chapter 7).

There's one other practice that you will find useful as soon as you start regularly testing your modules: version pinning. You should pin all of your Terraform modules to a specific version of Terraform using the `required_version`argument. At a bare minimum, you should require a specific major version of Terraform:

```
terraform {
  # Require any 0.12.x version of Terraform
  required_version = ">= 0.12, < 0.13"
}
```

The code above will only allow you to use any 0.12.x version of Terraform with that module, but not 0.11.x and not 0.13.x. This is critical, because each major release of Terraform is backwards incompatible: e.g., upgrading from 0.11.x to 0.12.x required many code changes, so you don't want to do it by accident. By adding the `required_version` setting, if you try run `terraform apply` with a different version, you immediately get an error:

```
$ terraform apply

Error: Unsupported Terraform Core version

This configuration does not support Terraform version 0.11.11. To proceed,
either choose another supported Terraform version or update the root module's
version constraint. Version constraints are normally set for good reason, so
updating the constraint may lead to other errors or unexpected behavior.
```

For production-grade code, I recommend pinning the version even more strictly:

```
terraform {
  # Require Terraform at exactly version 0.12.0
  required_version = "= 0.12.0"
}
```

The reason for this is that even patch version number bumps (e.g., 0.12.0 → 0.12.1) can cause problems. Occasionally, they are buggy; occasionally, they are backwards incompatible (though that is more rare these days). But an even bigger issue is that once a Terraform state file has been written with a newer version of Terraform, you can no longer use that state file with any older version of Terraform. For example, let's say all of your code is deployed with Terraform 0.12.0, and one day, a developer who just happens have 0.12.1 installed comes along and runs `terraform apply` on a few of your modules. The state files for those modules are now no longer usable with 0.12.0, so now you're forced to update all your developer computers and all of your CI servers to 0.12.1!

This situation will likely get better when Terraform hits 1.0.0 and starts enforcing backwards compatibility, but until then, I recommend pinning yourself to an *exact* Terraform version. That way, there will be no accidental updates. Instead, you choose when you're ready to update, and when you do it, you can update all your code, all your developer computers, and all your CI servers at once.

I also recommend pinning all of your provider versions:

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

  # Allow any 2.x version of the AWS provider
  version = "~> 2.0"
}
```

This code pins the AWS provider code to any 2.x version (the `~> 2.0` syntax is equivalent to `>= 2.0, < 3.0`). Again, the bare minimum is to pin to a specific major version number to avoid accidentally pulling in backwards incompatible changes. Whether you need to pin the provider to an exact version depends on the provider. For example, the AWS provider updates often and does a good job of maintaining backwards compatibility, so you typically want to pin to a specific major version, but allow new patch versions to be picked up automatically so you get easy access to new features. However, each provider is different, so pay attention to how good of a job they do at maintaining backwards compatibility and pin the version number accordingly.

Releasable modules

Once your modules have been written and tested, the next step is to release them. As you saw in "Module Versioning", you can use Git tags with semantic versioning as follows:

```
$ git tag -a "v0.0.5" -m "Create new hello-world-app module"
$ git push --follow-tags
```

For example, to deploy version `v0.0.5` of your `hello-world-app` module in the staging environment, put the following code into *live/stage/services/hello-world-app/main.tf*:

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

  # Allow any 2.x version of the AWS provider
  version = "~> 2.0"
}

module "hello_world_app" {
  # TODO: replace this with your own module URL and version!!
  source = "git::git@github.com:foo/modules.git//services/hello-world-app?ref=v0.0.5"

  server_text            = "New server text"
  environment            = "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
}
```

And pass through the ALB DNS name as an output in *live/stage/services/hello-world-app/outputs.tf*:

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

And now you can deploy your versioned module by running `terraform init`and `terraform apply`:

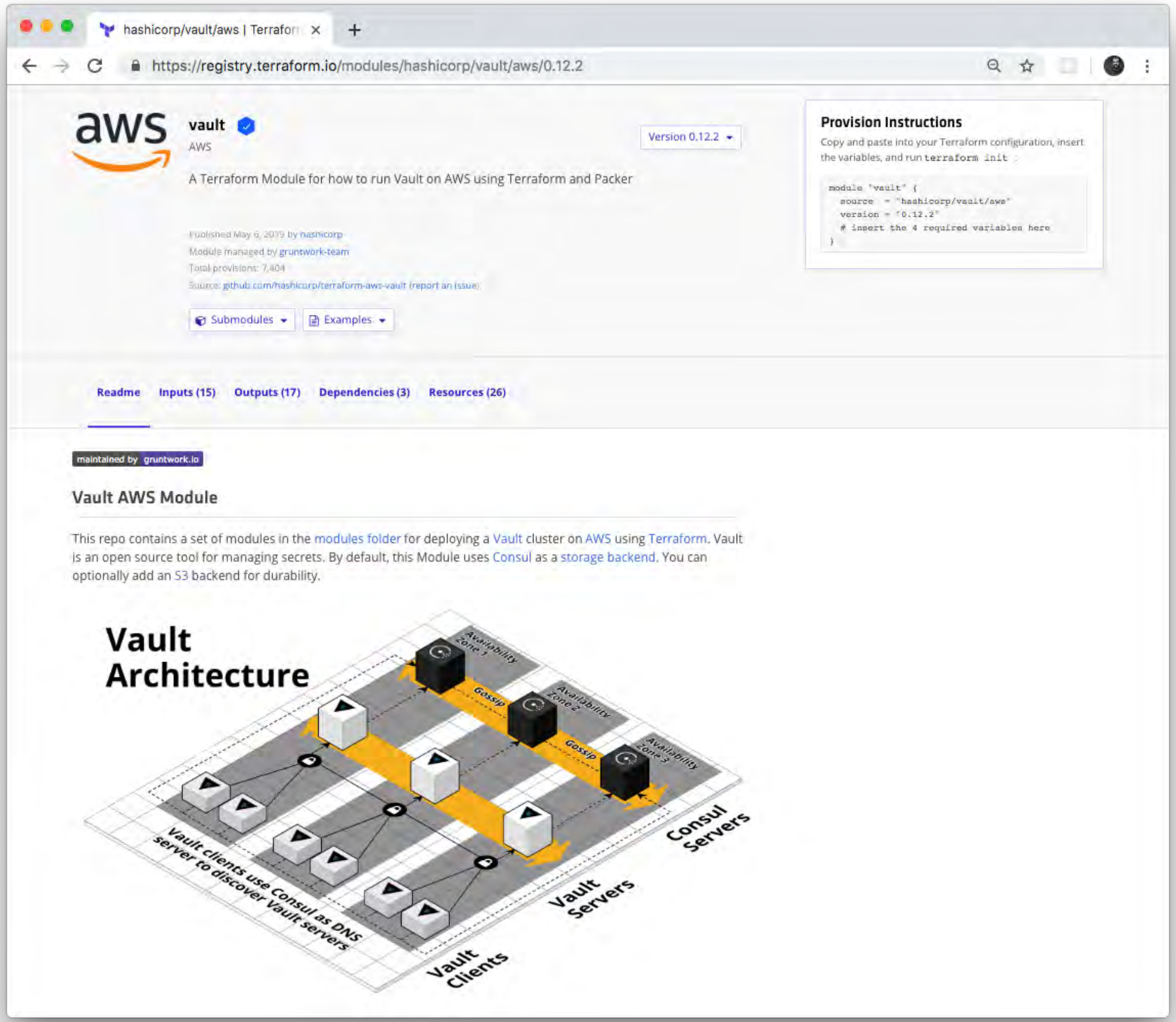
```
$ terraform apply

(...)

Apply complete! Resources: 13 added, 0 changed, 0 destroyed.

Outputs:

alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

If that works well, you can then deploy the exact same version—and therefore, the exact same code—to other environments, including production. If you ever hit an issue, versioning also gives you the option to roll back by deploying an older version.

Another option for releasing modules is to publish them in the Terraform Registry. The Public Terraform Registry lives at *https://registry.terraform.io/* and includes hundreds of reusable, community-maintained, open source modules for AWS, Google Cloud, Azure, and many other providers. There are a few requirements to publish a module to the Public Terraform Registry:[9]

1. The module must live in a public GitHub repo
2. The repo must be named `terraform-<PROVIDER>-<NAME>`, where `PROVIDER` is the provider the module is targeting (e.g., `aws`) and `NAME` is the name of the module (e.g., `vault`).
3. The module must follow a specific file structure, including defining Terraform code in the root of the repo, providing a `README.md`, and using the convention of `main.tf`, `variables.tf`, and `outputs.tf` as file names.
4. The repo must use Git tags with semantic versioning (`x.y.z`) for releases.

If your module meets those requirements, you can share it with the world by logging into the Terraform Registry with your GitHub account and using the web UI to publish the module. Once your module is in the Registry, you get a nice UI to browse the module, as shown in Figure 6-4.



vault

AWS

A Terraform Module for how to run Vault on AWS using Terraform and Packer

Published May 6, 2019 by hashicorp
Module managed by gruntwork-team
Total provisions: 7,404
Source: github.com/hashicorp/terraform-aws-vault (report an issue)

Provision Instructions

Copy and paste into your Terraform configuration, insert the variables, and run `terraform init` :

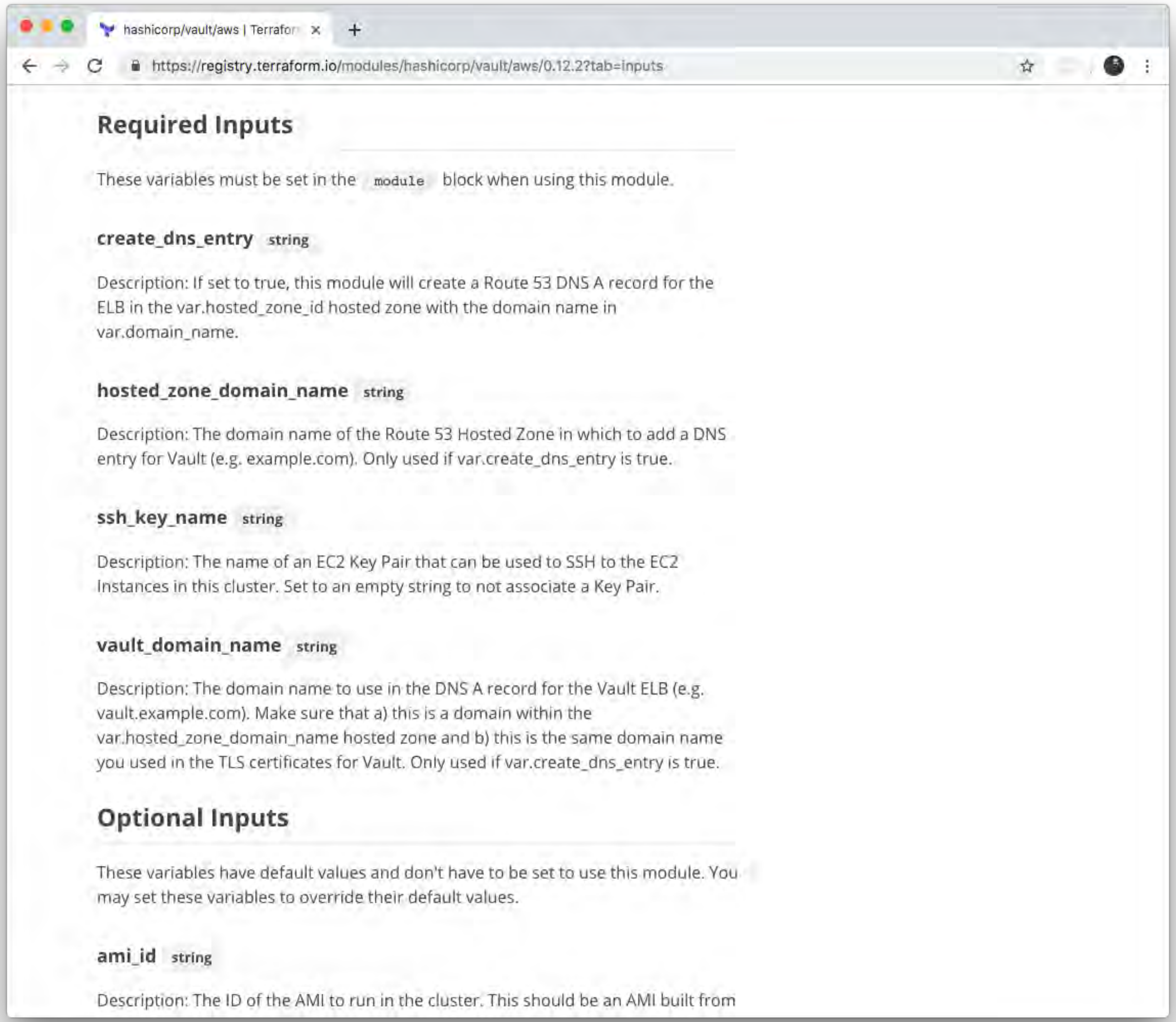
```
module "vault" {
  source  = "hashicorp/vault/aws"
  version = "0.12.2"
  # insert the 4 required variables here
}
```

Vault AWS Module

This repo contains a set of modules in the modules folder for deploying a Vault cluster on AWS using Terraform. Vault is an open source tool for managing secrets. By default, this Module uses Consul as a storage backend. You can optionally add an S3 backend for durability.

Figure 6-4. HashiCorp Vault module in the Terraform Registry

The Terraform Registry knows how to parse module inputs and outputs, so those show up in the UI as well, including the `type` and `description` fields, as shown in Figure 6-5.


hashicorp/vault/aws | Terrafor
https://registry.terraform.io/modules/hashicorp/vault/aws/0.12.2?tab=inputs

Required Inputs

These variables must be set in the `module` block when using this module.

create_dns_entry string

Description: If set to true, this module will create a Route 53 DNS A record for the ELB in the var.hosted_zone_id hosted zone with the domain name in var.domain_name.

hosted_zone_domain_name string

Description: The domain name of the Route 53 Hosted Zone in which to add a DNS entry for Vault (e.g. example.com). Only used if var.create_dns_entry is true.

ssh_key_name string

Description: The name of an EC2 Key Pair that can be used to SSH to the EC2 Instances in this cluster. Set to an empty string to not associate a Key Pair.

vault_domain_name string

Description: The domain name to use in the DNS A record for the Vault ELB (e.g. vault.example.com). Make sure that a) this is a domain within the var.hosted_zone_domain_name hosted zone and b) this is the same domain name you used in the TLS certificates for Vault. Only used if var.create_dns_entry is true.

Optional Inputs

These variables have default values and don't have to be set to use this module. You may set these variables to override their default values.

ami_id string

Description: The ID of the AMI to run in the cluster. This should be an AMI built from

Figure 6-5. The Terraform Registry automatically parses and displays module inputs and outputs

Terraform even supports a special syntax for consuming modules from the Terraform Registry. Instead of long Git URLs with hard-to-spot `ref` parameters, you can use a special shorter registry URL in the `source` argument and specify the version via a separate `version` argument using the following syntax:

```
module "<NAME>" {
  source  = "<OWNER>/<REPO>/<PROVIDER>"
  version = "<VERSION>"

  # (...)
}
```

where `NAME` is the identifier to use for the module in your Terraform code, `OWNER`is the owner of the GitHub repo (e.g., in `github.com/foo/bar`, the owner is `foo`), `REPO` is the name of the GitHub repo (e.g., in `github.com/foo/bar`, the repo is `bar`), `PROVIDER` is the provider you're targeting (e.g., `aws`), and `VERSION`is the version of the module to use. Here's an example of how to use the Vault module from the Terraform Registry:

```
module "vault" {
  source  = "hashicorp/vault/aws"
  version = "0.12.2"

  # (...)
}
```

If you are a customer of HashiCorp's Terraform Enterprise, you can have this same experience with a Private Terraform Registry: that is, a registry that lives in your private Git repos, and is only accessible to your team. This can be a great way to share modules within your company.

Beyond Terraform modules

Although this book is all about Terraform, to build out your entire production-grade infrastructure, you'll need to use other tools too, such as Docker, Packer, Chef, Puppet, and, of course, the duct tape, glue, and work horse of the DevOps world, the trusty Bash script. Most of this code can live in the `modules` folder right alongside your Terraform code. For example, in the HashiCorp Vault repo you saw earlier, the `modules` folder contains not only Terraform code, such as the `vault-cluster` module mentioned earlier, but Bash scripts, such as the `run-vault` Bash script[10] that can be run on a Linux server during boot (e.g., via User Data) to configure and start Vault.

However, every now and then, you may need to go further, and run some non Terraform code (e.g., a script) directly from a Terraform module. Sometimes, this is to integrate Terraform with another system (e.g., you've already used Terraform to configure User Data scripts for execution on EC2 Instances); other times, it's to work around a limitation of Terraform, such as a missing provider API, or the inability to implement complicated logic due to Terraform's declarative nature. If you search around, you can find a few "escape hatches" within Terraform that make this possible:

1. Provisioners

2. Provisioners with null_resource

3. External data source

Let's go through these one a time.

PROVISIONERS

Terraform *provisioners* are used to execute scripts either on the local machine or a remote machine when you run Terraform, typically to do the work of bootstrapping, configuration management, or cleanup. There are several different kinds of provisioners, including `local-exec` (execute a script on the local machine), `remote-exec` (execute a script on a remote resource), `chef` (run Chef Client on a remote resource), and `file` (copy files to a remote resource).[11]

Provisioners can be added to a resource by using a `provisioner` block. For example, here is how you can use the `local-exec` provisioner to execute a script on your local machine:

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

  provisioner "local-exec" {
    command = "echo \"Hello, World from $(uname -smp)\""
  }
}
```

When you run `terraform apply` on this code, it will print "Hello, World from" and then the local operating system details using the `uname` command:

```
$ terraform apply

(...)

aws_instance.example (local-exec): Hello, World from Darwin x86_64 i386

(...)

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

Trying out a `remote-exec` provisioner is a little more complicated. To execute code on a remote resource, such as an EC2 Instance, your Terraform client must be able to:

1. **Talk to the EC2 Instance over the network**: you already know how to allow this with a Security Group.

2. **Authenticate to the EC2 Instance**: The `remote-exec` provisioner supports SSH and WinRM connections. Since you'll be launching a Linux EC2 Instance (Ubuntu), you'll want to use SSH authentication. And that means you'll need to configure SSH keys.

Let's start by creating a Security Group that allows inbound connections to port 22, the default port for SSH:

```
resource "aws_security_group" "instance" {
  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"

    # To make this example easy to try out, we allow all SSH connections.
    # In real world usage, you should lock this down to solely trusted IPs.
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

With SSH keys, the normal process would be for you to generate an SSH key-pair on your computer, upload the public key to AWS, and store the private key somewhere secure where your Terraform code can access it. However, to make it easier for you to try this code out, you can use a resource called `tls_private_key` to automatically generate a private key:

```
# To make this example easy to try out, we generate a private key in Terraform.
# In real-world usage, you should manage SSH keys outside of Terraform.
resource "tls_private_key" "example" {
  algorithm = "RSA"
  rsa_bits  = 4096
}
```

This private key will be stored in Terraform state, which is not great for production use cases, but is fine for this learning exercise. Next, upload the public key to AWS using the `aws_key_pair` resource:

```
resource "aws_key_pair" "generated_key" {
  public_key = tls_private_key.example.public_key_openssh
}
```

Finally, let's start writing the code for the EC2 Instance:

```
resource "aws_instance" "example" {
  ami                    = "ami-0c55b159cbfafe1f0"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name               = aws_key_pair.generated_key.key_name
}
```

The first few lines of this code should be familiar to you: it's deploying an Ubuntu AMI on a `t2.micro` and associating the Security Group you created earlier with this EC2 Instance. The only new item is the use of the `key_name`attribute to tell AWS to associate your public key with this EC2 Instance. AWS will add that public key to the server's `authorized_keys` file, which will allow you to SSH to that server with the corresponding private key.

Next, let's add the `remote-exec` provisioner to this EC2 Instance:

```
resource "aws_instance" "example" {
  ami                    = "ami-0c55b159cbfafe1f0"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name               = aws_key_pair.generated_key.key_name

  provisioner "remote-exec" {
    inline = ["echo \"Hello, World from $(uname -smp)\""]
  }
}
```

This looks nearly identical to the `local-exec` provisioner, except you use an `inline` argument to pass a list of commands to execute, instead of a single `command` argument. Finally, you need to configure Terraform to use SSH to connect to this EC2 Instance when running the `remote-exec` provisioner. This is done using a `connection` block:

```
resource "aws_instance" "example" {
  ami                    = "ami-0c55b159cbfafe1f0"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name               = aws_key_pair.generated_key.key_name

  provisioner "remote-exec" {
    inline = ["echo \"Hello, World from $(uname -smp)\""]
  }

  connection {
    type        = "ssh"
    host        = self.public_ip
    user        = "ubuntu"
    private_key = tls_private_key.example.private_key_pem
  }
}
```

This `connection` block tells Terraform to connect to the EC2 Instance's public IP address using SSH with "ubuntu" as the username (this is the default username for the root user on Ubuntu AMIs) and the auto-generated private key. Note the use of the `self` keyword to set the `host` parameter. *Self expressions* use the following syntax:

```
self.<ATTRIBUTE>
```

You can use this special syntax solely in `connection` and `provisioner` blocks to refer to an output `ATTRIBUTE` of the surrounding resource. If you tried to use the standard `aws_instance.example.<ATTRIBUTE>` syntax, you'd get a circular dependency error, as resources can't have references to themselves, so the self expression is a workaround added specifically for provisioners.

If you run `terraform apply` on this code, you'll see the following:

```
$ terraform apply

(...)

aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Provisioning with 'remote-exec'...
aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connecting to remote host via SSH...

(... repeats a few more times ...)

aws_instance.example (remote-exec): Connecting to remote host via SSH...
aws_instance.example (remote-exec): Connected!
aws_instance.example (remote-exec): Hello, World from Linux x86_64 x86_64

Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
```

The `remote-exec` provisioner doesn't know exactly when the EC2 Instance will be booted and ready to accept connections, so it will retry the SSH connection multiple times until it succeeds or hits a timeout (the default timeout is 5 minutes, but it's configurable). Eventually, the connection succeeds, and you get a "Hello, World" from the server.

Note that, by default, when you specify a provisioner, it is a *creation-time provisioner*, which means it runs (a) during `terraform apply` and (b) only during the initial creation of a resource. The provisioner will *not* run on any subsequent calls to `terraform apply`, so creation-time provisioners are mainly useful for running initial bootstrap code. If you set the `when = "destroy"`argument on a provisioner, it will be a *destroy-time provisioner* which will run (a) after you run `terraform destroy`, (b) just before the resource is deleted.

You can specify multiple provisioners on the same resource and Terraform will run them one at a time, in order, from top to bottom. You can use the `on_failure`argument to tell Terraform how to handle errors from the provisioner: if set to `"continue"` Terraform will ignore the error and continue with resource creation or destruction; if set to `"abort"`, Terraform will abort the creation or destruction.

PROVISIONERS VERSUS USER DATA

You’ve now seen two different ways to execute scripts on a server using Terraform: one is to use a `remote-exec`provisioner and the other is to use a User Data script. I’ve generally found User Data to be the more useful tool for the following reasons:

1. A `remote-exec` provisioner requires that you open up SSH or WinRM access to your servers, which is more complicated to manage (as you saw above with all the security group and SSH key work) and less secure than User Data, which solely requires AWS API access (which you must have anyway to use Terraform).

2. You can use User Data scripts with ASGs, ensuring that all servers in that ASG execute the script during boot, including servers launched due to an auto scaling or auto recovery event. Provisioners only take effect while Terraform is running and don’t work with Auto Scaling Groups at all.

3. The User Data script can be seen in the EC2 Console (select an Instance, click Actions → Instance Settings → View/Change User Data) and you can find its execution log on the EC2 Instance itself (typically in */var/log/cloud-init*.log*), both of which are useful for debugging, and neither of which is available with provisioners.

The only advantages to provisioners are:

1. User Data scripts are limited to a length of 16 KB, whereas provisioner scripts can be arbitrarily long.

2. The Chef, Puppet, and Salt provisioners install, configure, and run the Chef, Puppet, and Salt clients, respectively, automatically on the server. This makes it easier to use configuration management tools instead of ad-hoc scripts to configure your servers.

PROVISIONERS WITH NULL_RESOURCE

Provisioners can only be defined within a resource, but sometimes, you want to execute a provisioner without tying it to a specific resource. You can do this using something called the `null_resource`, which acts just like a normal Terraform resource, except it doesn’t create anything. By defining provisioners on the `null_resource`, you can run your scripts as part of the Terraform lifecycle, but without being attached to any “real” resource:

```
resource "null_resource" "example" {
  provisioner "local-exec" {
    command = "echo \"Hello, World from $(uname -smp)\""
  }
}
```

The `null_resource` even has a handy argument called `triggers`, which takes in a map of keys and values. Whenever the values change, the `null_resource` will be recreated, therefore forcing any provisioners within it to be re-executed. For example, if you wanted to execute a provisioner within a `null_resource` every single time you ran `terraform apply`, you could use the `uuid()` built-in function, which returns a new, randomly generated UUID each time it’s called, within the `triggers` argument:

```
resource "null_resource" "example" {
  # Use UUID to force this null_resource to be recreated on every
  # call to 'terraform apply'
  triggers = {
    uuid = uuid()
  }

  provisioner "local-exec" {
    command = "echo \"Hello, World from $(uname -smp)\""
  }
}
```

Now, every time you call `terraform apply`, the `local-exec` provisioner will execute:

```
$ terraform apply

(...)

null_resource.example (local-exec): Hello, World from Darwin x86_64 i386

$ terraform apply
```

```
null_resource.example (local-exec): Hello, World from Darwin x86_64 i386
```

EXTERNAL DATA SOURCE

Provisioners will typically be your go-to for executing scripts from Terraform, but they aren't always the right fit. Sometimes, what you're really looking to do is execute a script to fetch some data and make that data available within the Terraform code itself. To do this, you can use the `external` data source, which allows an external command that implements a specific protocol to act as a data source.

The protocol is as follows:

- You can pass data from Terraform to the external program using the `query`argument of the `external` data source. The external program can read in these arguments as JSON from stdin.
- The external program can pass data back to Terraform by writing JSON to stdout. The rest of your Terraform code can then pull data out of this JSON by using the `result` output attribute of the external data source.

Here's an example:

```
data "external" "echo" {
  program = ["bash", "-c", "cat /dev/stdin"]

  query = {
    foo = "bar"
  }
}

output "echo" {
  value = data.external.echo.result
}

output "echo_foo" {
  value = data.external.echo.result.foo
}
```

This example uses the `external` data source to execute a Bash script that echos back to stdout any data it receives on stdin. Therefore, any data we pass in via the `query` argument should come back as-is via the `result` output attribute. Here's what happens when you run `terraform apply` on this code:

```
$ terraform apply

(...)

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

Outputs:

echo = {
  "foo" = "bar"
}
echo_foo = bar
```

You can see that `data.external.<NAME>.result` contains the JSON returned by the external program and that you can navigate within that JSON using the syntax `data.external.<NAME>.result.<PATH>` (e.g., `data.external.echo.result.foo`).

The `external` data source is a lovely escape hatch if you need to access data in your Terraform code and there's no existing data source that knows how to retrieve that data. However, be conservative with your use of `external` data sources and all of the other Terraform "escape hatches," as they make your code less portable and more brittle. For example, the `external` data source code you just saw relies on Bash, which means you won't be able to deploy that Terraform module from Windows.

Conclusion

Now that you've seen all the ingredients of creating production-grade Terraform code, it's time to put them together. The next time you begin to work on a new module, use the following process:

1. Go through the production-grade infrastructure checklist in Table 6-2 and explicitly identify the items you'll be implementing and the items you'll be skipping. Use the results of this checklist, plus Table 6-1, to come up with a time estimate for your boss.

2. Create an `examples` folder and write the example code first, using it to define the best user experience and cleanest API you can think of for your modules. Create an example for each important permutation of your module and include enough documentation and reasonable defaults to make the example as easy to deploy as possible.

3. Create a `modules` folder and implement the API you came up with as a collection of small, reusable, composable modules. Use a combination of Terraform and other tools like Docker, Packer, and Bash to implement these modules. Make sure to pin your Terraform and provider versions.

4. Create a `test` folder and write automated tests for each example.

Move on to Chapter 7 to learn how to write automated tests for your infrastructure code.

---

1 Hofstadter, Douglas R. *Gödel, Escher, Bach: An Eternal Golden Braid*. 20 Anv edition. New York: Basic Books, 1999.

2 Godin, Seth. "Don't Shave That Yak!" Seth's Blog, March 5, 2005. *https://seths.blog/2005/03/dont_shave_that/*.

3 Brooks, Frederick P. Jr. *The Mythical Man-Month: Essays on Software Engineering*. Anniversary edition. Reading, MA: Addison-Wesley Professional, 1995.

4 Martin, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st edition. Upper Saddle River, NJ: Prentice Hall, 2008.

5 Salus, Peter H. *A Quarter-Century of Unix*. New York: Addison-Wesley Professional, 1994.

6 *https://github.com/hashicorp/terraform-aws-consul/tree/master/modules/consul-security-group-rules*

7 *https://github.com/hashicorp/terraform-aws-vault/tree/master/modules/vault-cluster*

8 *https://github.com/hashicorp/terraform-aws-consul/tree/master/modules/consul-cluster*

9 You can find the full details on publishing modules at *https://www.terraform.io/docs/registry/modules/publish.html*

10 *https://github.com/hashicorp/terraform-aws-vault/tree/master/modules/run-vault*

11 You can find the full list of provisioners at *https://www.terraform.io/docs/provisioners/index.html*

Chapter 7. How to test Terraform code

The DevOps world is full of fear. Fear of downtime. Fear of data loss. Fear of security breaches. Every time you go to make a change, you're always wondering, what will this affect? Will it work the same way in every environment? Will this cause another outage? And if there is an outage, how late into the night will you have to stay up to fix it this time? As companies grow, there is more and more at stake, which makes the deployment process even scarier, and even more error prone. Many companies try to mitigate this risk by doing deployments less frequently, but the result is that each deployment is larger, and actually more prone to breakage.

If you manage your infrastructure as code, you have a better way to mitigate risk: tests. The goal of testing is to give you the confidence to make changes. The key word here is *confidence*: no form of testing can guarantee that your code is free of bugs, so it's more of a game of probability. If you can capture all of your infrastructure and deployment processes as code, then you can test that code in a pre-production environment, and if it works there, there's a high probability that when you use the exact same code in production, it'll work there too. And in a world of fear and uncertainty, high probability and high confidence go a long way.

In this chapter, I'll go over the process of testing infrastructure code, including both manual testing and automated testing, with the bulk of the chapter spent on the latter topic:

- Manual tests
  - Manual testing basics
  - Cleaning up after tests
- Automated tests
  - Unit tests
  - Integration tests
  - End-to-end tests
  - Other testing approaches

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*.

Manual tests

When thinking about how to test Terraform code, it can be helpful to draw some parallels with how you would test code written in a general purpose programming language, such as Ruby. Let's say you were writing a simple web server in Ruby in *web-server.rb*:

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Hello, World'
    else
      response.status = 404
      response['Content-Type'] = 'text/plain'
      response.body = 'Not Found'
    end
  end
end
```

This code will send a 200 OK response with the body "Hello, World" for the `/`URL and a 404 for all other URLs. How would you manually test this code? The typical answer is to add a bit of code to run the web server on localhost:

```
# This will only run if this script was called directly from the CLI, but
# not if it was required from another file
if __FILE__ == $0
  # Run the server on localhost at port 8000
  server = WEBrick::HTTPServer.new :Port => 8000
  server.mount '/', WebServer

  # Shut down the server on CTRL+C
  trap 'INT' do server.shutdown end

  # Start the server
  server.start
end
```

When you run this file from the CLI, it will start the web server on port 8000:

```
$ ruby web-server.rb
[2019-05-25 14:11:52] INFO  WEBrick 1.3.1
[2019-05-25 14:11:52] INFO  ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-05-25 14:11:52] INFO  WEBrick::HTTPServer#start: pid=19767 port=8000
```

And you can test this server using `curl`:

```
$ curl localhost:8000/
Hello, World

$ curl localhost:8000/invalid-path
Not Found
```

Now, let’s say you made a change to this code, adding a `/api` endpoint that responds with a 201 Created and a JSON body:

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Hello, World'
    when "/api"
      response.status = 201
      response['Content-Type'] = 'application/json'
      response.body = '{"foo":"bar"}'
    else
      response.status = 404
      response['Content-Type'] = 'text/plain'
      response.body = 'Not Found'
    end
  end
end
```

To manually test your updated code, you’d hit `CTRL+C` and re-run the script to restart the server:

```
$ ruby web-server.rb
[2019-05-25 14:11:52] INFO  WEBrick 1.3.1
[2019-05-25 14:11:52] INFO  ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-05-25 14:11:52] INFO  WEBrick::HTTPServer#start: pid=19767 port=8000
^C
[2019-05-25 14:15:54] INFO  going to shutdown ...
[2019-05-25 14:15:54] INFO  WEBrick::HTTPServer#start done.
```

```
$ ruby web-server.rb
[2019-05-25 14:11:52] INFO  WEBrick 1.3.1
[2019-05-25 14:11:52] INFO  ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-05-25 14:11:52] INFO  WEBrick::HTTPServer#start: pid=19767 port=8000
```

And you'd again use `curl` to test the new version:

```
$ curl localhost:8000/api
{"foo":"bar"}
```

Manual testing basics

What is the equivalent of this sort of manual testing with Terraform code? For example, from the previous chapters, you already have Terraform code for deploying an ALB. Here's a snippet from *modules/networking/alb/main.tf*:

```
resource "aws_lb" "example" {
  name               = var.alb_name
  load_balancer_type = "application"
  subnets            = var.subnet_ids
  security_groups    = [aws_security_group.alb.id]
}

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
    }
  }
}

resource "aws_security_group" "alb" {
  name = var.alb_name
}

# (...)
```

If you compare this code to the Ruby code, one difference should be fairly obvious: you can't deploy AWS ALBs, target groups, listeners, security groups, and all the other infrastructure on your own computer.

This brings us to *key testing takeaway #1*: when testing Terraform code, there is no localhost.

This applies to most infrastructure as code tools, and not just Terraform. The only practical way to do manual testing with Terraform is to deploy to a real environment (i.e., deploy to AWS). In other words, the way you've been manually running `terraform apply` and `terraform destroy` throughout the book is how you do manual testing with Terraform.

This is one of the reasons it's essential to have easy-to-deploy examples in the `examples` folder for each module, as described in Chapter 6. The easiest way to manually test the `alb` module is to use the example code you created for it in *examples/alb*:

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

  # Allow any 2.x version of the AWS provider
  version = "~> 2.0"
}
```

```
module "alb" {
  source = "../../modules/networking/alb"

  alb_name   = "terraform-up-and-running"
  subnet_ids = data.aws_subnet_ids.default.ids
}
```

As you’ve done many times throughout the book, you deploy this example code using `terraform apply`:

```
$ terraform apply

(...)

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:

alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

When the deploy is done, you can use a tool such as `curl` to test, for example, that the default action of the ALB is to return a 404:

```
$ curl \
  -s \
  -o /dev/null \
  -w "%{http_code}" \
  hello-world-stage-477699288.us-east-2.elb.amazonaws.com

404
```

VALIDATING INFRASTRUCTURE

The examples in this chapter use `curl` and HTTP requests to validate that the infrastructure is working, as the infrastructure we’re testing includes a load balancer that responds to HTTP requests. For other types of infrastructure, you’ll have to replace `curl` and HTTP requests with a different form of validation. For example, if your infrastructure code deploys a MySQL database, you’ll need to use a MySQL client to validate it; if your infrastructure code deploys a VPN server, you’ll need to use a VPN client to validate it; if your infrastructure code deploys a server that isn’t listening for requests at all, you may need to SSH to the server and execute some commands locally to test it; and so on. So while the same basic test structure described in this chapter can be used with any type of infrastructure, the validation steps will change depending on what you’re testing.

As a reminder, the ALB returns a 404 because it has no other listener rules configured, and the default action in the `alb` module is to return a 404:

```
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
    }
  }
}
```

Now that you have a way to run and test your code, you can start making changes. Every time you make a change—e.g., change the default action to return a 401—you re-run `terraform apply` to deploy the new changes:

```
$ terraform apply

(...)

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

Outputs:

alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

And re-run `curl` to test the new version:

```
$ curl \
  -s \
  -o /dev/null \
  -w "%{http_code}" \
  hello-world-stage-477699288.us-east-2.elb.amazonaws.com

401
```

When you're done, you run `terraform destroy` to clean up:

```
$ terraform destroy

(...)

Apply complete! Resources: 0 added, 0 changed, 5 destroyed.
```

In other words, when working with Terraform, every developer needs good example code to test and a real deployment environment (e.g., an AWS account) to use as an equivalent to localhost for running those tests. In the process of manual testing, you're likely to bring up and tear down a lot of infrastructure, and likely make lots of mistakes along the way, so this environment should be completely isolated from your other, more stable environments, such as staging, and especially production.

Therefore, I strongly recommend that every team sets up an isolated *sandbox environment* where developers can bring up and tear down any infrastructure they want without worrying about affecting others. In fact, to reduce the chances of conflicts between multiple developers (e.g., two developers trying to create a load balancer with the same name), the gold standard is that each developer gets their own completely isolated sandbox environment. For example, if you're using Terraform with AWS, the gold standard is for each developer to have their own AWS account that they can use to test anything they want.[1]

Cleaning up after tests

Having many sandbox environments is essential for developer productivity, but if you're not careful, you may end up with infrastructure running all over the place, cluttering up all of your environments, and costing you a lot of money.

To keep costs from spiraling out of control, *key testing takeaway #2* is: regularly clean up your sandbox environments.

At a minimum, your should create a culture where developers clean up whatever they deployed when they are done testing by running `terraform destroy`. Depending on your deployment environment, you may also be able to find tools you can run on a regular schedule (e.g., as a cron job) to automatically clean up unused or old resources, such as:

1. cloud-nuke: an open source tool that can delete all the resources in your cloud environment. It currently supports a number of resources in AWS (e.g., EC2 Instances, ASGs, ELBs, etc), with support for other resources and other clouds (Google Cloud, Azure) coming in the future. The key feature is the ability to delete all resources older than a certain age. For example, a common pattern is to run `cloud-nuke` as a cron job once per day in each sandbox environment to delete all resources that are more than 2 days old, based on the assumption that any infrastructure a developer fired up for manual testing is no longer necessary after a couple days:

   ```
   $ cloud-nuke aws --older-than 48h
   ```

2. Janitor Monkey: an open source tool that cleans up AWS resources on a configurable schedule (default is once per week). Supports configurable rules to determine if a resource should be cleaned up and even the ability to send a notification to the owner of the resource a few days before deletion. This is part of the Netflix Simian Army project, which also includes Chaos

Monkey, a tool for testing application resiliency. Note that the Simian Army project is no longer actively maintained, but various parts of it are being picked up by new projects, such as Janitor Monkey being replaced bySwabbie.

3. aws-nuke: an open source tool dedicated to deleting everything in an AWS account. You tell `aws-nuke` which accounts and resources to delete using a YAML configuration file:

```
# Regions to nuke
regions:
- us-east-2

# Accounts to nuke
accounts:
  "111111111111": {}

# Only nuke these resources
resource-types:
  targets:
  - S3Object
  - S3Bucket
  - IAMRole
```

And you run it as follows:

```
$ aws-nuke -c config.yml
```

Automated tests

WARNING: LOTS OF CODING AHEAD

Writing automated tests for infrastructure code is not for the feint of heart. This automated testing section is arguably the most complicated part of the book, and does not make for light reading. If you're just skimming, feel free to skip this part. On the other hand, if you really want to learn how to test your infrastructure code, then roll up your sleeves, and get ready to write some code! You don't need to run any of the Ruby code (it's just there to help build up your mental model), but you'll want to write and run as much Go code as you can.

The idea with automated testing is to write test code that validates that your real code behaves the way it should. As you'll see in Chapter 8, you can set up a CI server to run these tests after every single commit, and immediately revert or fix any commits that cause the tests to fail, thereby always keeping your code in a working state.

Broadly speaking, there are three types of automated tests:

1. **Unit tests**: Unit tests verify the functionality of a single, small unit of code. The definition of *unit* varies, but in a general-purpose programming language, it's typically a single function or class. Usually, any external dependencies—e.g., databases, web services, even the file system—are replaced with *test doubles* or *mocks* that allow you to finely control the behavior of those dependencies (e.g., by returning a hard-coded response from a database mock) to test that your code handles a variety of scenarios.

2. **Integration tests**: Integration tests verify that multiple units work together correctly. In a general-purpose programming language, an integration test consist of code that validates if several functions or classes work together correctly. Integration tests typically use a mix of real dependencies and mocks: for example, if you're testing the part of your app that talks to the database, you may want to test it with a real database, but mock out other dependencies, such as the app's authentication system.

3. **End-to-end tests**: End-to-end tests involve running your entire architecture—e.g., your apps, your data stores, your load balancers, etc—and validating that your system works as a whole. Usually, these tests are done from the end-user's perspective, such as using Selenium to automate interacting with your product via a web browser. End-to-end tests typically use real systems everywhere, without any mocks, in an architecture that mirrors production (albeit with fewer/smaller servers to save money).

Each type of test serves a different purpose, and can catch different types of bugs, so you'll likely want to use a mix of all three types. The purpose of unit tests is to have tests that run quickly, so you can get fast feedback on your changes and validate a variety of different permutations to build up confidence that the basic building blocks of your code (the individual units) work as expected. But just because individual units work correctly in isolation doesn't mean that they will work correctly when combined, so you need integration tests to make sure the basic building blocks fit together correctly. And just because different parts of your system work correctly doesn't mean it'll work correctly when deployed in the real world, so you need end-to-end tests to validate that your code behaves as expected in conditions similar to production.

Let's now go through how to write each type of test for Terraform code.

Unit tests

To understand how to write unit tests for Terraform code, it’s helpful to first look at what it takes to write unit tests for a general purpose programming language such as Ruby. Take a look again at the Ruby web server code:

```ruby
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Hello, World'
    when "/api"
      response.status = 201
      response['Content-Type'] = 'application/json'
      response.body = '{"foo":"bar"}'
    else
      response.status = 404
      response['Content-Type'] = 'text/plain'
      response.body = 'Not Found'
    end
  end
end
```

Writing a unit test for this code is a little tricky, as you would need to:

1. Instantiate the `WebServer` class. This is harder than it sounds, as the constructor for `WebServer`, which extends `AbstractServlet`, requires passing in a full WEBrick `HTTPServer` class. You could create a mock of it, but that’s a lot of work.
2. Create a `request` object, which is of type `HTTPRequest`. There’s no easy way to instantiate this class and creating a mock of it is a fair amount of work.
3. Create a `response` object, which is of type `HTTPResponse`. Again, there’s no easy way to instantiate this class and creating a mock of it is a fair amount of work.

When you find it hard to write unit tests, that’s often a code smell, and indicates the code needs to be refactored. One way to refactor this Ruby code to make unit testing easier is to extract the “handlers”—that is, the code that handles the `/`, `/api`, and not found paths—into its own `Handlers` class:

```ruby
class Handlers
  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Hello, World']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    else
      [404, 'text/plain', 'Not Found']
    end
  end
end
```

There are two key properties to notice about this new `Handlers` class:

1. **Simple values as inputs**: The `Handlers` class does not depend on `HTTPServer`, `HTTPRequest`, or `HTTPResponse`. Instead, all of its inputs are basic parameters, such as the `path` of the URL, which is a string.
2. **Simple values as outputs**: Instead of setting values on a mutable `HTTPResponse` object (a side effect), the methods in the `Handlers` class return the HTTP response as a simple value (an array that contains the HTTP status code, content type, and body).

Code that takes in simple values as inputs and returns simple values as outputs is typically easier to understand, update, and test. Let’s first update the `WebServer`class to use the new `Handlers` class to respond to requests:

```ruby
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    handlers = Handlers.new
    status_code, content_type, body = handlers.handle(request.path)

    response.status = status_code
    response['Content-Type'] = content_type
    response.body = body
  end
end
```

This code calls the `handle` method of the `Handlers` class and sends back the status code, content type, and body returned by that method as an HTTP response. As you can see, using the `Handlers` class is clean and simple. This same property will make testing easy too. Here's what a unit test for the `/`endpoint looks like:

```
class TestWebServer < Test::Unit::TestCase
  def initialize(test_method_name)
    super(test_method_name)
    @handlers = Handlers.new
  end

  def test_unit_hello
    status_code, content_type, body = @handlers.handle("/")
    assert_equal(200, status_code)
    assert_equal('text/plain', content_type)
    assert_equal('Hello, World', body)
  end
end
```

The test code calls the same `handle` method of the `Handlers` class and uses several `assert` methods to validate the response that comes back from the `/`endpoint. Here's how you run the test:

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Finished in 0.000287 seconds.
-------------------------------------------
1 tests, 3 assertions, 0 failures, 0 errors
100% passed
-------------------------------------------
```

Looks like the test passed. Let's now add unit tests for the `/api` and 404 endpoints:

```
  def test_unit_api
    status_code, content_type, body = @handlers.handle("/api")
    assert_equal(201, status_code)
    assert_equal('application/json', content_type)
    assert_equal('{"foo":"bar"}', body)
  end

  def test_unit_404
    status_code, content_type, body = @handlers.handle("/invalid-path")
    assert_equal(404, status_code)
    assert_equal('text/plain', content_type)
    assert_equal('Not Found', body)
  end
```

Run the tests again:

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Finished in 0.000572 seconds.
-------------------------------------------
3 tests, 9 assertions, 0 failures, 0 errors
100% passed
-------------------------------------------
```

In 0.0005272 seconds, you can now find out if your web server code works as expected. That's the power of unit testing: a fast feedback loop that helps you build confidence in your code. If you make any mistake in your code—e.g., you unintentionally changed the response of the `/api` endpoint—you find out about that almost immediately:

```
$ ruby web-server-test.rb
Loaded suite web-server-test
===========================================================================
Failure: test_unit_api(TestWebServer)
web-server-test.rb:25:in `test_unit_api'
     22:     status_code, content_type, body = Handlers.new.handle("/api")
     23:     assert_equal(201, status_code)
     24:     assert_equal('application/json', content_type)
  => 25:     assert_equal('{"foo":"bar"}', body)
     26:   end
     27:
     28:   def test_unit_404
<"{\"foo\":\"bar\"}"> expected but was
<"{\"foo\":\"whoops\"}">

diff:
? {"foo":"bar   "}
?         whoops
===========================================================================

Finished in 0.007904 seconds.
-------------------------------------------
3 tests, 9 assertions, 1 failures, 0 errors
66.6667% passed
-------------------------------------------
```

UNIT TESTING BASICS

What is the equivalent of this sort of unit testing with Terraform code? The first step is to identify what a "unit" is in the Terraform world. The closest equivalent to a single function or class in Terraform is a single generic module (using the term "generic module" as defined in "Composable modules"), such as the `alb`module you created in Chapter 6. How would you test this module?

With Ruby, to write unit tests, you had to refactor the code so you could run it without complicated dependencies, such as `HTTPServer`, `HTTPRequest`, or `HTTPResponse`. If you think about what your Terraform code is doing—making API calls to AWS to create the load balancer, listeners, target groups, etc—you'll realize that 99% of what this code is doing is talking to complicated dependencies! There's no practical way to reduce the number of external dependencies to zero, and even if you could, you'd effectively be left with no code to test.[2]

That brings us to *key testing takeaway #3*: you cannot do *pure* unit testing for Terraform code.

But don't despair. You can still build confidence that your Terraform code behaves as expected by writing automated tests that use your code to deploy real infrastructure into a real environment (e.g., into a real AWS account). In other words, unit tests for Terraform are really integration tests. However, I prefer to still call them unit tests to emphasize that the goal is to test a single unit (i.e., a single generic module) to get feedback as quickly as possible.

That means that the basic strategy for writing unit tests for Terraform is:

1. Create a generic, standalone module.
2. Create an easy-to-deploy example for that module.
3. Run `terraform apply` to deploy the example into a real environment.
4. Validate that what you just deployed works as expected. This step is specific to the type of infrastructure you're testing: e.g., for an ALB, you'd validate it by sending an HTTP request and checking you get back the expected response.
5. Run `terraform destroy` at the end of the test to clean up.

In other words, you do *exactly* the same steps as you would when doing manual testing, but you capture those steps as code. In fact, that's a good mental model for creating automated tests for your Terraform code: ask yourself, "How would I have tested this manually to be confident it works?" and then implement that test in code.

You can use any programming language you want to write the test code. In this book, you'll see that all the tests are written in the Go programming language to take advantage of an open source Go library called Terratest, which supports testing a wide variety of infrastructure as code tools (e.g., Terraform, Packer, Docker, Helm, etc.) across a wide variety of environments (e.g., AWS, Google Cloud, Kubernetes, etc). Terratest is a bit like a Swiss Army Knife, with hundreds of tools built-in that make it significantly easier to test infrastructure code, including first-class support for the test strategy described above, where you `terraform apply` some code, validate it works, and then run `terraform destroy` at the end to clean up.

To use Terratest, you need to:

1. Install Go: *https://golang.org/doc/install*.
2. Configure the `GOPATH` environment variable: *https://golang.org/doc/code.html#GOPATH*.
3. Add `$GOPATH/bin` to your `PATH` environment variable.
4. Install Dep, a dependency manager for Go: *https://golang.github.io/dep/docs/installation.html*. [3]
5. Create a folder within your `GOPATH` for your test code: e.g., the default `GOPATH` is *$HOME/go*, so you could create *$HOME/go/src/terraform-up-and-running*.
6. Run `dep init` in the folder you just created. This should create `Gopkg.toml`, `Gopkg.lock`, and an empty `vendors` folder.

As a quick sanity check that your environment is set up correctly, create *go_sanity_test.go* in your new folder with the following contents:

```go
package test

import (
	"fmt"
	"testing"
)

func TestGoIsWorking(t *testing.T)  {
	fmt.Println()
	fmt.Println("If you see this text, it's working!")
	fmt.Println()
}
```

Run this test using the `go test` command and make sure you see the following output:

```
$ go test -v

If you see this text, it's working!

PASS
ok      terraform-up-and-running        0.004s
```

(The `-v` flag means verbose, which ensures the test always shows all log output).

If that's working, feel free to delete *go_sanity_test.go*, and let's move on to writing a unit test for the `alb` module. Create *alb_example_test.go* in your test folder with the following skeleton of a unit test:

```go
package test

import (
	"testing"
)

func TestAlbExample(t *testing.T)  {
}
```

The first step is to tell Terratest where your Terraform code lives using the `terraform.Options` type:

```go
package test

import (
	"github.com/gruntwork-io/terratest/modules/terraform"
	"testing"
)

func TestAlbExample(t *testing.T)  {
	opts := &terraform.Options{
		// You should update this relative path to point at your alb
		// example directory!
```

```
                TerraformDir: "../examples/alb",
        }
}
```

Note that to test the `alb` module, you actually test the example code in your `examples` folder (note: you should update the relative path in `TerraformDir` to point to the folder where you created that example). That means that example code now serves three roles: (1) executable documentation, (2) a way to run manual tests for your modules, (3) a way to run automated tests for your modules.

Also, note that there's now a new import for the Terratest library at the top of the file. To download this dependency to your computer, run `dep ensure`:

```
$ dep ensure
```

The `dep ensure` command will scan your Go code, find any new imports, automatically download them and all their dependencies to the `vendor` folder, and add them to `Gopkg.lock`. If that's a bit too magical for you, you can alternatively use the `dep ensure -add` command to explicitly add the dependencies you want:

```
$ dep ensure -add github.com/gruntwork-io/terratest/modules/terraform
```

The next step in the automated test is to run `terraform init` and `terraform apply` to deploy the code. Terratest has handy helpers for doing that:

```
func TestAlbExample(t *testing.T)  {
        opts := &terraform.Options{
                // You should update this relative path to point at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }

        terraform.Init(t, opts)
        terraform.Apply(t, opts)
}
```

In fact, running `init` and `apply` is such a common operation with Terratest, that there is a convenient helper method that does both in one command:

```
func TestAlbExample(t *testing.T)  {
        opts := &terraform.Options{
                // You should update this relative path to point at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }

        // Deploy the example
        terraform.InitAndApply(t, opts)
}
```

The code above is already a fairly useful unit test, as it will run `terraform init`and `terraform apply`, and fail the test if those commands don't complete successfully (e.g., due to a problem with your Terraform code). However, you can go even further, by making HTTP requests to the deployed load balancer, and checking that it returns the data you expect. To do that, you need a way to get the domain name of the deployed load balancer. Fortunately, that's available as an ouptut variable in the `alb` example:

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

Terratest has helpers built-in to read outputs from your Terraform code:

```
func TestAlbExample(t *testing.T)  {
        opts := &terraform.Options{
                // You should update this relative path to point at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }

        // Deploy the example
        terraform.InitAndApply(t, opts)

        // Get the URL of the ALB
        albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)
}
```

The `OutputRequired` function will return the output of the given name, or fail the test if that output doesn't exist or is empty. The code above builds a URL from this output using the `fmt.Sprintf` function that's built into Go (don't forget to import the `fmt` package). The next step is to make some HTTP requests to this URL:

```
package test

import (
        "fmt"
        "github.com/gruntwork-io/terratest/modules/http-helper"
        "github.com/gruntwork-io/terratest/modules/terraform"
        "testing"
)

func TestAlbExample(t *testing.T)  {
        opts := &terraform.Options{
                // You should update this relative path to point at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }

        // Deploy the example
        terraform.InitAndApply(t, opts)

        // Get the URL of the ALB
        albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)

        // Test that the ALB's default action is working and returns a 404

        expectedStatus := 404
        expectedBody := "404: page not found"

        http_helper.HttpGetWithValidation(t, url, expectedStatus, expectedBody)
}
```

This new code uses a new import from Terratest, the `http_helper` package, so you'll need to run `dep ensure` one more time to download it. The `http_helper.HttpGetWithValidation` method will make an HTTP GET request to the URL you pass in and fail the test if the response doesn't have the status code and body you specified.

There's one problem with this code: there is a brief period of time between when `terraform apply` finishes and the DNS name of the load balancer is working (i.e, has propagated). If you run `http_helper.HttpGetWithValidation`immediately, there's a chance that it will fail, even though 30 seconds or a minute later, the ALB would be working just fine. As discussed in "Eventual Consistency Is Consistent…Eventually", this sort of asynchronous and eventually consistent behavior is normal in AWS—normal in most distributed systems, actually—and the solution is to add retries. Terratest has a helper for that too.

```
func TestAlbExample(t *testing.T)  {
        opts := &terraform.Options{
                // You should update this relative path to point at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }

        // Deploy the example
        terraform.InitAndApply(t, opts)
        // Get the URL of the ALB
        albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)

        // Test that the ALB's default action is working and returns a 404

        expectedStatus := 404
        expectedBody := "404: page not found"

        maxRetries := 10
        timeBetweenRetries := 10 * time.Second

        http_helper.HttpGetWithRetry(
                t,
                url,
                expectedStatus,
                expectedBody,
                maxRetries,
                timeBetweenRetries,
        )
}
```

The `http_helper.HttpGetWithRetry` method is nearly identical to `http_helper.HttpGetWithValidation`, except if it doesn't get back the expected status code or body, it will retry up to the specified max number of retries (10), with the specified amount of time between retries (10 seconds). If it eventually gets the expected response, the test will pass; if the max number of retries is reached without the expected response, the test will fail.

The last thing you need to do is to run `terraform destroy` at the end of the test to clean up. As you can guess, there is a Terratest helper for this, `terraform.Destroy`. However, if you call `terraform.Destroy` at the very end of the test, then if any of the code before that causes a test failure (e.g., `HttpGetWithRetry` fails because the ALB is misconfigured), then the test code will exit before getting to `terraform.Destroy`, and the infrastructure deployed for the test will never be cleaned up.

Therefore, you want to ensure that you *always* run `terraform.Destroy`, even if the test fails. In many programming languages, this is done with a `try` / `finally`or `try` / `ensure` construct, but in Go, this is done using the `defer` statement, which will guarantee that the code you pass to it will be executed when the surrounding function returns (no matter how that return happens):

```
func TestAlbExample(t *testing.T)  {
        opts := &terraform.Options{
                // You should update this relative path to point at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }

        // Clean up everything at the end of the test
        defer terraform.Destroy(t, opts)

        // Deploy the example
        terraform.InitAndApply(t, opts)

        // Get the URL of the ALB
        albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)

        // Test that the ALB's default action is working and returns a 404

        expectedStatus := 404
        expectedBody := "404: page not found"

        maxRetries := 10
        timeBetweenRetries := 10 * time.Second

        http_helper.HttpGetWithRetry(
                t,
                url,
                expectedStatus,
                expectedBody,
                maxRetries,
                timeBetweenRetries,
        )
}
```

Note that the `defer` is added early in the code, even before the call to `terraform.InitAndApply`, to ensure nothing can cause the test to fail before getting to the `defer` statement, and preventing it from queueing up the call to`terraform.Destroy`.

OK, this unit test is finally ready to run. Since this test deploys infrastructure to AWS, before running the test, you'll need to authenticate to your AWS account as usual (see Authentication Options). You saw earlier in this chapter that you should do manual testing in a sandbox account; for automated testing, this is even more important, so I recommend authenticating to a totally separate account. As your automated test suite grows, you may be spinning up hundreds or thousands of resources in every test suite, so keeping them isolated from everything else is essential.

I typically recommend that teams have a completely separate environment (e.g., completely separate AWS account) just for automated testing—separate even from the sandbox environments you use for manual testing. That way, you can safely delete all resources that are more than a few hours old in the testing environment, based on the assumption that no test will run that long.

Once you've authenticated to an AWS account you can safely use for testing, you can run the test as follows:

```
$ go test -v -timeout 30m

TestAlbExample 2019-05-26T13:29:32+01:00 command.go:53:
Running command terraform with args [init -upgrade=false]

(...)

TestAlbExample 2019-05-26T13:29:33+01:00 command.go:53:
Running command terraform with args [apply -input=false -lock=false -auto-approve]

(...)

TestAlbExample 2019-05-26T13:32:06+01:00 command.go:121:
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

(...)

TestAlbExample 2019-05-26T13:32:06+01:00 command.go:53:
Running command terraform with args [output -no-color alb_dns_name]

(...)

TestAlbExample 2019-05-26T13:38:32+01:00 http_helper.go:27:
Making an HTTP GET call to URL
http://terraform-up-and-running-1892693519.us-east-2.elb.amazonaws.com

(...)

TestAlbExample 2019-05-26T13:38:32+01:00 command.go:53:
Running command terraform with args
[destroy -auto-approve -input=false -lock=false]

(...)

TestAlbExample 2019-05-26T13:39:16+01:00 command.go:121:
Destroy complete! Resources: 5 destroyed.

(...)

PASS
ok      terraform-up-and-running        229.492s
```

Note the use of the `-timeout 30m` argument with `go test`. By default, Go imposes a time limit of 10 minutes for tests, after which it forcibly kills the test run, causing the tests to not only fail, but even preventing the cleanup code (i.e., `terraform destroy`) from running. This ALB test should take closer to 5 minutes, but whenever running a Go test that deploys real infrastructure, it's safer to set an extra large timeout to avoid the test being killed part way through, and leaving all sorts of infrastructure still running.

The test will produce a lot of log output, but if you read through it carefully, you should be able to spot all the key parts of the test:

1. Running `terraform init`
2. Running `terraform apply`
3. Reading output variables using `terraform output`
4. Repeatedly making HTTP requests to the ALB
5. Running `terraform destroy`

It’s nowhere near as fast as the Ruby unit tests, but in less than 5 minutes, you can now automatically find out if your `alb` module works as expected. This is about as fast of a feedback loop as you can get with infrastructure in AWS, and it should give you a lot of confidence that your code works as expected. If you make any mistake in your code—e.g., you unintentionally changed the status code in the default action to a 401—you find out about that fairly quickly:

```
$ go test -v -timeout 30m

(...)

Validation failed for URL
http://terraform-up-and-running-931760451.us-east-2.elb.amazonaws.com.
Response status: 401. Response body: 404: page not found.

(...)

Sleeping for 10s and will try again.

(...)

Validation failed for URL
http://terraform-up-and-running-h2ezYz-931760451.us-east-2.elb.amazonaws.com.
Response status: 401. Response body: 404: page not found.

(...)

Sleeping for 10s and will try again.

(...)

--- FAIL: TestAlbExample (310.19s)
    http_helper.go:94:
    HTTP GET to URL
    http://terraform-up-and-running-931760451.us-east-2.elb.amazonaws.com
    unsuccessful after 10 retries

FAIL    terraform-up-and-running        310.204s
```

DEPENDENCY INJECTION

Let’s now see what it would take to add a unit test for some slightly more complicated code. Going back to the Ruby web server example once more, consider what would happen if you needed to add a new `/web-service` endpoint that made HTTP calls to an external dependency:

```
class Handlers
  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Hello, World']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    when "/web-service"
      # New endpoint that calls a web service
      uri = URI("http://www.example.org")
      response = Net::HTTP.get_response(uri)
      [response.code.to_i, response['Content-Type'], response.body]
    else
      [404, 'text/plain', 'Not Found']
    end
  end
end
```

The updated `Handlers` class now handles the `/web-service` URL by calling a new method call `web_service`, which makes an HTTP GET to `example.org` and proxies the response. When you `curl` this endpoint, you get:

```
$ curl localhost:8000/web-service

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    <-- (...) -->
</head>
<body>
<div>
    <h1>Example Domain</h1>
    <p>
      This domain is established to be used for illustrative
      examples in documents. You may use this domain in
      examples without prior coordination or asking for permission.
    </p>
    <!-- (...) -->
</div>
</body>
</html>
```

How would you add a unit test for this new method? If you tried to test the code as-is, then your unit tests would be subject to the behavior of an external dependency (in this case, `example.org`). This has a number of downsides:

1. If that dependency has an outage, your tests will fail, even though there's nothing wrong with your code.
2. If that dependency changed its behavior from time to time (e.g., returned a different response body), your tests would fail from time to time, and you'd have to constantly keep updating the test code, even though there's nothing wrong with the implementation.
3. If that dependency was slow, your tests would be slow, which negates one of the main benefits of unit tests, the fast feedback loop.
4. If you wanted to test that your code handles various corner cases based on how that dependency behaves (e.g., does the code handle redirects?), you'd have no way to do it without control of that external dependency.

While working with real dependencies may make sense for integration and end-to-end tests, with unit tests, you should try to minimize external dependencies as much as possible. The typical strategy for doing this is *dependency injection*, where you make it possible to pass (or "inject") external dependencies from outside of your code, rather than hard-coding them within your code.

For example, the `Handlers` class shouldn't have to deal with all the details of how to call a web service. Instead, you can extract that logic into a separate `WebService` class:

```
class WebService
  def initialize(url)
    @uri = URI(url)
  end

  def proxy
    response = Net::HTTP.get_response(@uri)
    [response.code.to_i, response['Content-Type'], response.body]
  end
end
```

This class takes a URL as an input and exposes a `proxy` method to proxy the HTTP GET response from that URL. You can then update the `Handlers` class to take a `WebService` instance as an input and use that instance in the `web_service`method:

```
class Handlers
  def initialize(web_service)
```

```ruby
      @web_service = web_service
    end

    def handle(path)
      case path
      when "/"
        [200, 'text/plain', 'Hello, World']
      when "/api"
        [201, 'application/json', '{"foo":"bar"}']
      when "/web-service"
        # New endpoint that calls a web service
        @web_service.proxy
      else
        [404, 'text/plain', 'Not Found']
      end
    end
  end
```

Now, in your implementation code, you can inject a real `WebService` instance that makes HTTP calls to `example.org`:

```ruby
  class WebServer < WEBrick::HTTPServlet::AbstractServlet
    def do_GET(request, response)
      web_service = WebService.new("http://www.example.org")
      handlers = Handlers.new(web_service)

      status_code, content_type, body = handlers.handle(request.path)

      response.status = status_code
      response['Content-Type'] = content_type
      response.body = body
    end
  end
```

In your test code, you can create a mock version of the `WebService` class that allows you to specify a mock response to return:

```ruby
  class MockWebService
    def initialize(response)
      @response = response
    end

    def proxy
      @response
    end
  end
```

And now you can create an instance of this `MockWebService` class and inject it into the `Handlers` class in your unit tests:

```ruby
    def test_unit_web_service
      expected_status = 200
      expected_content_type = 'text/html'
      expected_body = 'mock example.org'
      mock_response = [expected_status, expected_content_type, expected_body]

      mock_web_service = MockWebService.new(mock_response)
      handlers = Handlers.new(mock_web_service)

      status_code, content_type, body = handlers.handle("/web-service")
      assert_equal(expected_status, status_code)
      assert_equal(expected_content_type, content_type)
      assert_equal(expected_body, body)
    end
```

Re-run the tests to make sure it all still works:

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Started
...

Finished in 0.000645 seconds.
--------------------------------------------
4 tests, 12 assertions, 0 failures, 0 errors
100% passed
--------------------------------------------
```

Fantastic. Using dependency injection to minimize external dependencies allows you to write fast, reliable tests, and check all the various corner cases. And since the 3 test cases you added earlier are still passing, you can be confident that your refactoring hasn't broken anything.

Let's now turn our attention back to Terraform and see what dependency injection looks like with Terraform modules, starting with the `hello-world-app`module. If you haven't already, the first step is to create an easy-to-deploy example for it in the `examples` folder:

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

  # Allow any 2.x version of the AWS provider
  version = "~> 2.0"
}

module "hello_world_app" {
  source = "../../../modules/services/hello-world-app"

  server_text = "Hello, World"
  environment = "example"

  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "examples/terraform.tfstate"

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

The dependency problem becomes apparent immediately: the `hello-world-app`assumes you've already deployed the `mysql` module and requires that you pass in the details of the S3 bucket where the `mysql` module is storing state using the`db_remote_state_bucket` and `db_remote_state_key` arguments. The goal here is to create a unit test for the `hello-world-app` module, and while a pure unit test with 0 external dependencies isn't possible with Terraform, it's still a good idea to minimize external dependencies whenever possible.

One of the first steps with minimizing dependencies is to make it clearer what dependencies your module has. A file-naming convention you may want to adopt is to move all of the data sources and resources that represent external dependencies into a separate `dependencies.tf` file. For example, here's what*modules/services/hello-world-app/dependencies.tf* would look like:

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

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

data "aws_vpc" "default" {
  default = true
}
```

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

This convention makes it easier for users of your code to tell, at a glance, what this code depends on in the outside world. In the case of the `hello-world-app`module, you can quickly see that it depends on a database, VPC, and subnets. So, how can you inject these dependencies from outside the module so you can replace them at test time? You already know the answer to this: input variables.

For each of these dependencies, you should add a new input variable in *modules/services/hello-world-app/variables.tf*:

```
variable "vpc_id" {
  description = "The ID of the VPC to deploy into"
  type        = string
  default     = null
}

variable "subnet_ids" {
  description = "The IDs of the subnets to deploy into"
  type        = list(string)
  default     = null
}

variable "mysql_config" {
  description = "The config for the MySQL DB"
  type        = object({
    address = string
    port    = number
  })
  default     = null
}
```

There's now an input variable for the VPC ID, subnet IDs, and MySQL config. Each variable specifies a `default`, so they are *optional variables*, that the user can set to something custom or omit to get the `default` value. The `default` for each variable is using a value you haven't seen before: `null`. If you set the `default` value to an *empty value*, such as an empty string for `vpc_id` or an empty list for `subnet_ids`, you wouldn't be able to tell the difference between (a) an empty value you set as a default vs (b) if the user of your module intentionally passed in an empty value they wanted to use. The *null* value is handy in these cases, as it is used specifically to indicate that a variable is unset and that the user wants to fall back to the default behavior.

Note that the `mysql_config` variable uses the `object` type constructor to create a nested type with `address` and `port` keys. This type is intentionally designed to match the output types of the `mysql` module:

```
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"
}
```

One of the advantages of doing this is that, once the refactor is complete, one of the ways you'll be able to use the `hello-world-app` and `mysql` modules together is as follows:

```
module "hello_world_app" {
  source = "../../../modules/services/hello-world-app"

  server_text            = "Hello, World"
  environment            = "example"

  # Pass all the outputs from the mysql module straight through!
  mysql_config = module.mysql

  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
}

module "mysql" {
  source = "../../../modules/data-stores/mysql"

  db_name     = var.db_name
  db_username = var.db_username
  db_password = var.db_password
}
```

Since the `type` of `mysql_config` matches the type of the `mysql` module outputs, you can pass them all straight through in one line. And if the types are ever changed and no longer match, Terraform will give you an error right away so that you know to update them. This is not only function composition at work, but type-safe function composition.

But before that can work, you'll need to finish refactoring the code. Since the MySQL configuration can be passed in as an input, that means the `db_remote_state_bucket` and `db_remote_state_key` variables should now be optional, so set their `default` values to `null`:

```
variable "db_remote_state_bucket" {
  description = "The name of the S3 bucket for the DB's Terraform state"
  type        = string
  default     = null
}

variable "db_remote_state_key" {
  description = "The path in the S3 bucket for the DB's Terraform state"
  type        = string
  default     = null
}
```

Next, use the `count` parameter to optionally create the three data sources in*modules/services/hello-world-app/dependencies.tf*, based on whether the corresponding input variable is set to `null` or not:

```
data "terraform_remote_state" "db" {
  count = var.mysql_config == null ? 1 : 0

  backend = "s3"

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

data "aws_vpc" "default" {
  count = var.vpc_id == null ? 1 : 0
  default = true
}

data "aws_subnet_ids" "default" {
  count = var.subnet_ids == null ? 1 : 0
  vpc_id = data.aws_vpc.default.id
}
```

And now you need to update any references to these data sources to conditionally use either the input variable or the data source. Let's capture these as local values:

```
locals {
  mysql_config = (
    var.mysql_config == null
```

```
      ? data.terraform_remote_state.db[0].outputs
      : var.mysql_config
  )

  vpc_id = (
    var.vpc_id == null
      ? data.aws_vpc.default[0].id
      : var.vpc_id
  )

  subnet_ids = (
    var.subnet_ids == null
      ? data.aws_subnet_ids.default[0].ids
      : var.subnet_ids
  )
}
```

Note that since the data sources use the `count` parameters, they are now lists, so any time you reference them, you have to use array lookup syntax (i.e., `[0]`). Go through the code, and anywhere you find a reference to one of these data sources, replace it with a reference to the corresponding local value instead. Start by updating the `aws_subnet_ids` data source to use `local.vpc_id`:

```
data "aws_subnet_ids" "default" {
  count = var.subnet_ids == null ? 1 : 0
  vpc_id = local.vpc_id
}
```

Then set the `subnet_ids` parameters of the `asg` and `alb` modules to use the `local.subnet_ids`:

```
module "asg" {
  source = "../../cluster/asg-rolling-deploy"

  cluster_name  = "hello-world-${var.environment}"
  ami           = var.ami
  user_data     = data.template_file.user_data.rendered
  instance_type = var.instance_type

  min_size           = var.min_size
  max_size           = var.max_size
  enable_autoscaling = var.enable_autoscaling

  subnet_ids        = local.subnet_ids
  target_group_arns = [aws_lb_target_group.asg.arn]
  health_check_type = "ELB"

  custom_tags = var.custom_tags
}

module "alb" {
  source = "../../networking/alb"

  alb_name   = "hello-world-${var.environment}"
  subnet_ids = local.subnet_ids
}
```

Update the `db_address` and `db_port` variables in `user_data` to use `local.mysql_config`:

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

  vars = {
    server_port = var.server_port
    db_address  = local.mysql_config.address
    db_port     = local.mysql_config.port
    server_text = var.server_text
  }
}
```

And finally, update the `vpc_id` parameter of the `aws_lb_target_group` to use `local.vpc_id`:

```
resource "aws_lb_target_group" "asg" {
  name     = "hello-world-${var.environment}"
  port     = var.server_port
  protocol = "HTTP"
  vpc_id   = local.vpc_id

  health_check {
    path                = "/"
    protocol            = "HTTP"
    matcher             = "200"
    interval            = 15
    timeout             = 3
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
}
```

With these updates, you can now choose to inject the VPC ID, subnet IDs, and/or MySQL config parameters into the `hello-world-app` module, or omit any of those parameters, and the module will use an appropriate data source to fetch those values by itself. Let's update the "Hello, World" app example to allow the MySQL config to be injected, but omit the VPC ID and subnet ID params, as using the default VPC is good enough for testing. Add a new input variable to*examples/hello-world-app/variables.tf*:

```
variable "mysql_config" {
  description = "The config for the MySQL DB"

  type = object({
    address = string
    port    = number
  })

  default = {
    address = "mock-mysql-address"
    port    = 12345
  }
}
```

And pass this variable through to the `hello-world-app` module in *examples/hello-world-app/main.tf*:

```
module "hello_world_app" {
  source = "../../../modules/services/hello-world-app"

  server_text = "Hello, World"
  environment = "example"

  mysql_config = var.mysql_config

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

You can now set this `mysql_config` variable in a unit test to any value you want. Create a unit test in *test/hello_world_app_example_test.go* with the following contents:

```
func TestHelloWorldAppExample(t *testing.T)  {
        opts := &terraform.Options{
                // You should update this relative path to point at your
                // hello-world-app example directory!
                TerraformDir: "../examples/hello-world-app/standalone",
        }

        // Clean up everything at the end of the test
        defer terraform.Destroy(t, opts)
        terraform.InitAndApply(t, opts)

        albDnsName := terraform.OutputRequired(t, opts, "alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)

        expectedStatus := 200
        expectedBody := "Hello, World"

        maxRetries := 10
        timeBetweenRetries := 10 * time.Second

        http_helper.HttpGetWithRetry(
                t,
                url,
                expectedStatus,
                expectedBody,
                maxRetries,
                timeBetweenRetries,
        )
}
```

This code is nearly identical to the unit test for the `alb` example. The only difference is the `TerraformDir` setting is pointing to the `hello-world-app`example (be sure to update the path as necessary for your file system) and the expected response from the ALB is a 200 OK with the body “Hello, World.” There’s just one new thing you’ll need to add to this test: set the `mysql_config`variable.

```
        opts := &terraform.Options{
                // You should update this relative path to point at your
                // hello-world-app example directory!
                TerraformDir: "../examples/hello-world-app/standalone",

                Vars: map[string]interface{}{
                        "mysql_config": map[string]interface{}{
                                "address": "mock-value-for-test",
                                "port": 3306,
                        },
                },
        }
```

The `Vars` parameter in `terraform.Options` allows you to set variables in your Terraform code. This code is passing in some mock data for the `mysql_config`variable. Alternatively, you could set this value to anything you want: for example, you could fire up a small, in-memory database at test time and set the `address` to that database’s IP.

Run this new test using `go test`, specifying the `-run` argument to run *just* this test (otherwise, Go’s default behavior is to run all tests in the current folder, including the ALB example test you created earlier):

```
$ go test -v -timeout 30m -run TestHelloWorldAppExample

(...)

PASS
ok      terraform-up-and-running        204.113s
```

If all goes well, the test will run `terraform apply`, make repeated HTTP requests to the load balancer, and once it gets back the expected response, will run `terraform destroy` to clean everything up. All told, it should take only a few minutes, and you now have a reasonable unit test for the “Hello, World” app.

RUNNING TESTS IN PARALLEL

In the previous section, you ran just a single test using the `-run` argument of the `go test` command. If you had omitted that argument, Go would've run all of your tests—sequentially. While 4-5 minutes to run a single test isn't too bad for testing infrastructure code, if you have dozens of tests, and each one runs sequentially, it could take hours to run your entire test suite. To shorten the feedback loop, you'll want to run as many tests in parallel as you can.

To tell Go to run your tests in parallel, the only change you need to make is to add `t.Parallel()` to the top of each test. Here it is in *test/hello_world_app_example_test.go*:

```
func TestHelloWorldAppExample(t *testing.T)  {
        t.Parallel()

        opts := &terraform.Options{
                // You should update this relative path to point at your
                // hello-world-app example directory!
                TerraformDir: "../examples/hello-world-app/standalone",

                Vars: map[string]interface{}{
                        "mysql_config": map[string]interface{}{
                                "address": "mock-value-for-test",
                                "port": 3306,
                        },
                },
        }

        // (...)
}
```

And similarly in *test/alb_example_test.go*:

```
func TestAlbExample(t *testing.T)  {
        t.Parallel()

        opts := &terraform.Options{
                // You should update this relative path to point at your alb
                // example directory!
                TerraformDir: "../examples/alb",
        }

        // (...)
}
```

If you run `go test` now, both of those tests will execute in parallel. However, there's one gotcha: some of the resources created by those tests—e.g., the ASG, security group, ALB, etc—use the same name, which will cause the tests to fail due to the name clashes. Even if you weren't using `t.Parallel()` in your tests, if multiple people on your team were running the same tests or if you had tests running in a CI environment, these sorts of name clashes would be inevitable.

This leads to *key testing takeaway #4*: you must namespace all of your resources.

That is, design modules and examples so that the name of every resource is (optionally) configurable. With the `alb` example, this means you need to make the name of the ALB configurable. Add a new input variable in*examples/alb/variables.tf* with a reasonable default:

```
variable "alb_name" {
  description = "The name of the ALB and all its resources"
  type        = string
  default     = "terraform-up-and-running"
}
```

And pass this value through to the `alb` module in *examples/alb/main.tf*:

```
module "alb" {
  source = "../../modules/networking/alb"

  alb_name   = var.alb_name
  subnet_ids = data.aws_subnet_ids.default.ids
}
```

Now, set this variable to a unique value in *test/alb_example_test.go*:

```
package test

import (
        "fmt"
        "github.com/gruntwork-io/terratest/modules/http-helper"
        "github.com/gruntwork-io/terratest/modules/random"
        "github.com/gruntwork-io/terratest/modules/terraform"
        "testing"
        "time"
)

func TestAlbExample(t *testing.T)  {
        t.Parallel()

        opts := &terraform.Options{
                // You should update this relative path to point at your alb
                // example directory!
                TerraformDir: "../examples/alb",

                Vars: map[string]interface{}{
                        "alb_name": fmt.Sprintf("test-%s", random.UniqueId()),
                },
        }

        // (...)
}
```

(Note the use of a new helper from the Terratest `random` package; you’ll need to run `dep ensure` once more.)

This code sets the `alb_name` var to `test-<RANDOM_ID>` where `RANDOM_ID` is a random unique ID returned by the `random.UniqueId()` helper in Terratest. This helper returns a randomized, 6-character base-62 string. The idea is that it’s a short identifier you can add to the names of most resources without hitting length limit issues, but random enough to make conflicts very unlikely (62^6 = 56+ billion combinations). This ensures that you can run a huge number of ALB tests in parallel with no concern of having a name conflict.

Make a similar change to the “Hello, World” app example, first by adding a new input variable in *examples/hello-world-app/variables.tf*:

```
variable "environment" {
  description = "The name of the environment we're deploying to"
  type        = string
  default     = "example"
}
```

Then passing that variable through to the `hello-world-app` module:

```
module "hello_world_app" {
  source = "../../../modules/services/hello-world-app"

  server_text = "Hello, World"

  environment = var.environment

  mysql_config = var.mysql_config

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

And finally, setting `environment` to a value that includes `random.UniqueId()` in*test/hello_world_app_example_test.go*:

```
func TestHelloWorldAppExample(t *testing.T)  {
        t.Parallel()

        opts := &terraform.Options{
                // You should update this relative path to point at your
                // hello-world-app example directory!
                TerraformDir: "../examples/hello-world-app/standalone",

                Vars: map[string]interface{}{
                        "mysql_config": map[string]interface{}{
                                "address": "mock-value-for-test",
                                "port": 3306,
                        },
                        "environment": fmt.Sprintf("test-%s", random.UniqueId()),
                },
        }

        // (...)
}
```

With these changes complete, it should now be safe to run all your tests in parallel:

```
$ go test -v -timeout 30m

TestAlbExample 2019-05-26T17:57:21+01:00 (...)
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)
TestAlbExample 2019-05-26T17:57:21+01:00 (...)
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)
TestHelloWorldAppExample 2019-05-26T17:57:21+01:00 (...)

(...)

PASS
ok      terraform-up-and-running        216.090s
```

You should see both tests running at the same time so that the entire test suite takes roughly as long as the slowest of the tests, rather than the combined time of all the tests running back to back.

RUNNING TESTS IN PARALLEL IN THE SAME FOLDER

One other type of parallelism to take into account is what happens if you try to run multiple automated tests in parallel against the same Terraform folder. For example, perhaps you'd want to run several different tests against*examples/hello-world-app*, where each test sets different values for the input variables before running `terraform apply`. If you try this, you'll hit a problem: the tests will end up clashing, as they all try to run `terraform init` and end up overwriting each other's `.terraform` folder and Terraform state files.

If you want to run multiple tests against the same folder in parallel, the easiest solution is to have each test copy that folder to a unique temporary folder, and run Terraform in the temporary folder to avoid conflicts. Terratest, of course, has a built in helper to do this for you, and it even does it in a way that ensures relative file paths within those Terraform modules work correctly: check out the `test_structure.CopyTerraformFolderToTemp` method and its documentation for details.

Integration tests

Now that you've got some unit tests in place, let's move on to integration tests. Once again, it's helpful to start with the Ruby web server example to build up some intuition that you can later apply to the Terraform code. To do an integration test of the Ruby web server code, you need to:

1. Run the web server on localhost so it listens on a port.
2. Send HTTP requests to the web server.

3. Validate you get back the responses you expect.

Let’s create a helper method in *web-server-test.rb* that implements these steps:

```
def do_integration_test(path, check_response)
  port = 8000
  server = WEBrick::HTTPServer.new :Port => port
  server.mount '/', WebServer

  begin
    # Start the web server in a separate thread so it
    # doesn't block the test
    thread = Thread.new do
      server.start
    end

    # Make an HTTP request to the web server at the
    # specified path
    uri = URI("http://localhost:#{port}#{path}")
    response = Net::HTTP.get_response(uri)

    # Use the specified check_response lambda to validate
    # the response
    check_response.call(response)
  ensure
    # Shut the server and thread down at the end of the
    # test
    server.shutdown
    thread.join
  end
end
```

The `do_integration_test` method configures the web server on port 8000, starts it in a background thread (so the web server doesn’t block the test from running), sends an HTTP GET to the `path` specified, passes the HTTP response to the specified `check_response` function for validation, and at the end of the test, shuts the web server down. Here’s how you can use this method to write an integration test for the `/` endpoint of the web server:

```
def test_integration_hello
  do_integration_test('/', lambda { |response|
    assert_equal(200, response.code.to_i)
    assert_equal('text/plain', response['Content-Type'])
    assert_equal('Hello, World', response.body)
  })
end
```

This method calls the `do_integration_test` method with the `/` path and passes it a lambda (essentially, an inline function) that checks the response was a 200 OK with the body “Hello, World.” The integration tests for the other endpoints are analogous, although the tests for the `/web-service` endpoint do a less specific check (i.e., `assert_include` rather than `assert_equal`) to try to minimize possible disruption from the `example.org` endpoint changing:

```
def test_integration_api
  do_integration_test('/api', lambda { |response|
    assert_equal(201, response.code.to_i)
    assert_equal('application/json', response['Content-Type'])
    assert_equal('{"foo":"bar"}', response.body)
  })
end

def test_integration_404
  do_integration_test('/invalid-path', lambda { |response|
    assert_equal(404, response.code.to_i)
    assert_equal('text/plain', response['Content-Type'])
    assert_equal('Not Found', response.body)
  })
end

def test_integration_web_service
  do_integration_test('/web-service', lambda { |response|
    assert_equal(200, response.code.to_i)
    assert_include(response['Content-Type'], 'text/html')
    assert_include(response.body, 'Example Domain')
  })
end
```

Let’s run all the tests:

```
$ ruby web-server-test.rb

(...)

Finished in 0.221561 seconds.
--------------------------------------------
8 tests, 24 assertions, 0 failures, 0 errors
100% passed
--------------------------------------------
```

Note that before, with solely unit tests, the test suite took 0.000572 seconds to run, but now, with integration tests, it takes 0.221561 seconds, a ~387x slowdown. Of course, 0.221561 seconds is still blazing fast, but that’s only because the Ruby web server code is intentionally a minimal example that doesn’t do much. The important thing here is not the absolute numbers, but the relative trend: integration tests are typically slower than unit tests. We’ll come back to this point later.

Let’s now turn our attention to integration tests for Terraform code. If a “unit” in Terraform is a single module, then an integration test that validates how several units work together would need to deploy several modules and see that they work correctly. In the previous section, you deployed the “Hello, World” app example with mock data instead of a real MySQL DB. For an integration test, let’s deploy the MySQL module for real and make sure the “Hello, World” app integrates with it correctly. You should already have just such code under *live/stage/data-stores/mysql* and *live/stage/services/hello-world-app*. That is, you can create an integration test for (parts of) your staging environment.

Of course, as mentioned earlier in the chapter, all automated tests should run in an isolated AWS account. So while you’re testing the code that is meant for staging, you should authenticate to an isolated testing account and run the tests there. If your modules have anything in them hard-coded for the staging environment, this is the time to make those values configurable so you can inject test-friendly values. In particular, in *live/stage/data-stores/mysql/variables.tf*, expose the database name via a new `db_name` input variable:

```
variable "db_name" {
  description = "The name to use for the database"
  type        = string
  default     = "example_database_stage"
}
```

And pass that value through to the `mysql` module in *live/stage/data-stores/mysql/main.tf*:

```
module "mysql" {
  source = "../../../../modules/data-stores/mysql"

  db_name     = var.db_name
  db_username = var.db_username
  db_password = var.db_password
}
```

Let’s now create the skeleton of the integration test in *test/hello_world_integration_test.go* and fill in the implementation details later:

```
// Replace these with the proper paths to your modules
const dbDirStage = "../live/stage/data-stores/mysql"
const appDirStage = "../live/stage/services/hello-world-app"

func TestHelloWorldAppStage(t *testing.T)  {
        t.Parallel()
        // Deploy the MySQL DB
        dbOpts := createDbOpts(t, dbDirStage)
        defer terraform.Destroy(t, dbOpts)
        terraform.InitAndApply(t, dbOpts)

        // Deploy the hello-world-app
        helloOpts := createHelloOpts(dbOpts, appDirStage)
        defer terraform.Destroy(t, helloOpts)
        terraform.InitAndApply(t, helloOpts)

        // Validate the hello-world-app works
        validateHelloApp(t, helloOpts)
}
```

The test is structured as follows: deploy `mysql`, deploy the `hello-world-app`, validate the app, undeploy the `hello-world+app` (runs at the end due to `defer`), and finally, undeploy `mysql` (runs at the end due to `defer`). The `createDbOpts`, `createHelloOpts`, and `validateHelloApp` methods don't exist yet, so let's implement them one at a time, starting with the `createDbOpts` method:

```
func createDbOpts(t *testing.T, terraformDir string) *terraform.Options {
        uniqueId := random.UniqueId()

        return &terraform.Options{
                TerraformDir: terraformDir,

                Vars: map[string]interface{}{
                        "db_name": fmt.Sprintf("test%s", uniqueId),
                        "db_password": "password",
                },
        }
}
```

Not much new so far: the code points `terraform.Options` at the passed in directory and sets the `db_name` and `db_password` variables.

The next step is to deal with where this `mysql` module will store its state. Up to now, the `backend` configuration has been set to hard-coded values:

```
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
  }
}
```

These hard-coded values are a big problem for testing, for if you don't change them, you'll end up overwriting the real state file for staging! One option is to use Terraform Workspaces (as discussed in "Isolation via workspaces"), but that would still require access to S3 bucket in the staging account, whereas you should be running tests in a totally separate AWS account. The better option is to use partial configuration, as introduced in "Limitations with Terraform's backends". Move the entire `backend` configuration into an external file, such as `backend.hcl`:

```
bucket         = "terraform-up-and-running-state"
key            = "stage/data-stores/mysql/terraform.tfstate"
region         = "us-east-2"
dynamodb_table = "terraform-up-and-running-locks"
encrypt        = true
```

Leaving the `backend` configuration in *live/stage/data-stores/mysql/main.tf* empty:

```
terraform {
  backend "s3" {
  }
}
```

When you're deploying the `mysql` module to the real staging environment, you tell Terraform to use the `backend` configuration in `backend.hcl` via the `-backend-config` argument:

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

When you're running tests on the `mysql` module, you can tell Terratest to pass in test-time friendly values using the `BackendConfig` parameter of `terraform.Options`:

```
func createDbOpts(t *testing.T, terraformDir string) *terraform.Options {
        uniqueId := random.UniqueId()

        bucketForTesting := "YOUR_S3_BUCKET_FOR_TESTING"
        bucketRegionForTesting := "YOUR_S3_BUCKET_FOR_TESTING"
        dbStateKey := fmt.Sprintf("%s/%s/terraform.tfstate", t.Name(), uniqueId)

        return &terraform.Options{
                TerraformDir: terraformDir,

                Vars: map[string]interface{}{
                        "db_name": fmt.Sprintf("test%s", uniqueId),
                        "db_password": "password",
                },

                BackendConfig: map[string]interface{}{
                        "bucket":         bucketForTesting,
                        "region":         bucketRegionForTesting,
                        "key":            dbStateKey,
                        "encrypt":        true,
                },
        }
}
```

You'll need to update the `bucketForTesting` and `bucketRegionForTesting`variables with your own values. You can create a single S3 bucket in your test AWS account to use as a `backend`, as the `key` configuration (the path within the bucket) includes the `uniqueId`, which should be unique enough to have a different value for each test.

The next step is to make some updates to the `hello-world-app` module in the staging environment. Open up *live/stage/services/hello-world-app/variables.tf*and expose variables for `db_remote_state_bucket`, `db_remote_state_key`, and `environment`:

```
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
}

variable "environment" {
  description = "The name of the environment we're deploying to"
  type        = string
  default     = "stage"
}
```

Pass those value through to the `hello-world-app` module in *live/stage/services/hello-world-app/main.tf*:

```
module "hello_world_app" {
  source = "../../../../modules/services/hello-world-app"

  server_text            = "Hello, World"

  environment            = var.environment
  db_remote_state_bucket = var.db_remote_state_bucket
  db_remote_state_key    = var.db_remote_state_key

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

Now you can implement the `createHelloOpts` method:

```
func createHelloOpts(
	dbOpts *terraform.Options,
	terraformDir string) *terraform.Options {

	return &terraform.Options{
		TerraformDir: terraformDir,

		Vars: map[string]interface{}{
			"db_remote_state_bucket": dbOpts.BackendConfig["bucket"],
			"db_remote_state_key": dbOpts.BackendConfig["key"],
			"environment": dbOpts.Vars["db_name"],
		},
	}
}
```

Note that `db_remote_state_bucket` and `db_remote_state_key` are set to the values used in the `BackendConfig` for the `mysql` module to ensure that the `hello-world-app` module is reading from the exact same state that the `mysql`module just wrote to. The `environment` variable is set to the `db_name` just so all the resources are namespaced the same way.

Finally, you can implement the `validateHelloApp` method:

```
func validateHelloApp(t *testing.T, helloOpts *terraform.Options) {
	albDnsName := terraform.OutputRequired(t, helloOpts, "alb_dns_name")
	url := fmt.Sprintf("http://%s", albDnsName)

	maxRetries := 10
	timeBetweenRetries := 10 * time.Second

	http_helper.HttpGetWithRetryWithCustomValidation(
		t,
		url,
		maxRetries,
		timeBetweenRetries,
		func(status int, body string) bool {
			return status == 200 &&
				strings.Contains(body, "Hello, World")
		},
	)
}
```

This method uses the `http_helper` package, just as with the unit tests, except this time, it’s with the`http_helper.HttpGetWithRetryWithCustomValidation` method that allows you to specify custom validation rules for the HTTP response status code and body. This is necessary to check that the HTTP response *contains* the string “Hello, World”, rather than equals that string exactly, as the User Data script in the `hello-world-app` module returns an HTML response with other text in it as well.

Alright, run the integration test to see if it worked:

```
$ go test -v -timeout 30m -v "TestHelloWorldAppStage"

(...)

PASS
ok      terraform-up-and-running        795.63s
```

Excellent, you now have an integration test that you can use to check that several of your modules work correctly together. This integration test is more complicated than the unit test, and takes more than twice as long (10-15 minutes rather than 4-5 minutes). In general, there's not a whole lot you can do to make things *faster*—the bottleneck here is how long AWS takes to deploy and undeploy RDS, ASGs, ALBs, etc.—but in certain circumstances, you may be able to make the test code do *less* using test stages.

TEST STAGES

If you look at the code for your integration test, you may notice that it consists of five distinct "stages":

1. Run `terraform apply` on the `mysql` module.
2. Run `terraform apply` on the `hello-world-app` module.
3. Run validations to make sure everything is working.
4. Run `terraform destroy` on the `hello-world-app` module.
5. Run `terraform destroy` on the `mysql` module.

When you run these tests in a CI environment, you'll want to run all the stages, from start to finish. However, if you're running these tests in your local dev environment while iteratively making changes to the code, running all of these stages is unnecessary. For example, if you're only making changes to the `hello-world-app` module, re-running this entire test after every change means you're paying the price of deploying and undeploying the `mysql` module, even though none of your changes affect it. That adds 5-10 minutes of pure overhead to every test run.

Ideally, the workflow would look more like this:

1. Run `terraform apply` on the `mysql` module.
2. Run `terraform apply` on the `hello-world-app` module.
3. Now, you start doing iterative development:
   a. Make a change to the `hello-world-app` module.
   b. Re-run `terraform apply` on the `hello-world-app` module to deploy your updates.
   c. Run validations to make sure everything is working.
   d. If everything works, move on to the next step. If not, go back to step (3a).
4. Run `terraform destroy` on the `hello-world-app` module.
5. Run `terraform destroy` on the `mysql` module.

Having the ability to do that inner loop in step 3 quickly is the key to fast, iterative development with Terraform. To support this, you need to break your test code into *stages*, where you can choose which stages to execute and which to skip.

Terratest supports this natively with the `test_structure` package (remember to run `dep ensure` to add it). The idea is that you wrap each stage of your test in a function with a name and you can then tell Terratest to skip some of those names by setting environment variables. Each test stage stores test data on disk so that it can be read back from disk on subsequent test runs. Let's try this out on *test/hello_world_integration_test.go*, writing the skeleton of the test first, and filling in the underlying methods later:

```go
func TestHelloWorldAppStageWithStages(t *testing.T) {
        t.Parallel()

        // Store the function in a short variable name solely to make the
        // code examples fit better in the book.
        stage := test_structure.RunTestStage

        // Deploy the MySQL DB
        defer stage(t, "teardown_db", func() { teardownDb(t, dbDirStage) })
        stage(t, "deploy_db", func() { deployDb(t, dbDirStage) })
        // Deploy the hello-world-app
        defer stage(t, "teardown_app", func() { teardownApp(t, appDirStage) })
        stage(t, "deploy_app", func() { deployApp(t, dbDirStage, appDirStage) })

        // Validate the hello-world-app works
        stage(t, "validate_app", func() { validateApp(t, appDirStage) })
}
```

The structure is the same as before—deploy `mysql`, deploy `hello-world-app`, validate `hello-world-app`, undeploy `hello-world-app` (runs at the end due to `defer`), undeploy `mysql` (runs at the end due to `defer`)—except now, each stage is wrapped in `test_structure.RunTestStage`. The `RunTestStage` method takes three arguments:

1. **t**: The first argument is the `t` value that Go passes as an argument to every automated test. You can use this value to manage test state. For example, you can fail the test by calling `t.Fail()`.
2. **Stage name**: The second argument allows you to specify the name for this test stage. You'll see an example shortly of how to use this name to skip test stages.
3. **The code to execute**: The third argument is the code to execute for this test stage. This can be any function.

Let's now implement the functions for each test stage, starting with `deployDb`:

```
func deployDb(t *testing.T, dbDir string) {
        dbOpts := createDbOpts(t, dbDir)

        // Save data to disk so that other test stages executed at a later
        // time can read the data back in
        test_structure.SaveTerraformOptions(t, dbDir, dbOpts)

        terraform.InitAndApply(t, dbOpts)
}
```

Just as before, to deploy `mysql`, the code calls `createDbOpts` and `terraform.InitAndApply`. The only new thing is that, in between those two steps, there is a call to `test_structure.SaveTerraformOptions`. This writes the data in `dbOpts` to disk so that other test stages can read it later on. For example, here's the implementation of the `teardownDb` function:

```
func teardownDb(t *testing.T, dbDir string) {
        dbOpts := test_structure.LoadTerraformOptions(t, dbDir)
        defer terraform.Destroy(t, dbOpts)
}
```

This function uses `test_structure.LoadTerraformOptions` to load the `dbOps`data from disk that was earlier saved by the `deployDb` function. The reason you need to pass this data via the hard drive, rather than passing it in memory, is that you may run each test stage as part of different test run—and therefore, as part of a different process. As you'll see a little later in this chapter, on the first few runs of `go test`, you may want to run `deployDb`, but skip `teardownDb`, and in later runs, do the opposite, running `teardownDb`, but skipping `deployDb`. To ensure that you're using the same database across all those test runs, you must store that database's information on disk.

Let's now implement the `deployHelloApp` function:

```
func deployApp(t *testing.T, dbDir string, helloAppDir string) {
        dbOpts := test_structure.LoadTerraformOptions(t, dbDir)
        helloOpts := createHelloOpts(dbOpts, helloAppDir)

        // Save data to disk so that other test stages executed at a later
        // time can read the data back in
        test_structure.SaveTerraformOptions(t, helloAppDir, helloOpts)

        terraform.InitAndApply(t, helloOpts)
}
```

This function reuses the `createHelloOpts` function from before, and calls `terraform.InitAndApply` on it. Once again, the only new behavior is the use of `test_structure.LoadTerraformOptions` to load `dbOpts` from disk, and the use of `test_structure.SaveTerraformOptions` to save `helloOpts` to disk. At this point, you can probably guess what the implementation of the `teardownApp`method looks like:

```
func teardownApp(t *testing.T, helloAppDir string) {
        helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)
        defer terraform.Destroy(t, helloOpts)
}
```

And the implementation of the `validateApp` method:

```
func validateApp(t *testing.T, helloAppDir string)  {
        helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)
        validateHelloApp(t, helloOpts)
}
```

So, overall, the test code is identical to the original integration test, except each stage is wrapped in a call to `test_structure.RunTestStage`, and you have to do a little work to save and load data to and from disk. These simple changes unlock an important ability: you can tell Terratest to skip any test stage called `foo` by setting the environment variable `SKIP_foo=true`. Let's go through a typical coding workflow to see how this works.

Your first step will be to run the test, but to skip both of the teardown stages, so that the `mysql` and `hello-world-app` modules stay deployed at the end of the test. Since the teardown stages are called `teardown_db` and `teardown_app`, you need to set the `SKIP_teardown_db` and `SKIP_teardown_app` environment variables, respectively, to tell Terratest to skip those two stages:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'

(...)

The 'SKIP_deploy_db' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'deploy_app' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'validate_app' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'teardown_app' environment variable is set,
so skipping stage 'deploy_db'.

(...)

The 'teardown_db' environment variable is set,
so skipping stage 'deploy_db'.

(...)

PASS
ok      terraform-up-and-running        423.650s
```

Now you can start iterating on the `hello-world-app` module, and each time you make a change, you can re-run the tests, but this time, tell them to skip not only the teardown stages, but also the `mysql` module deploy stage (as `mysql` is still running), so that the only things that get executed are `terraform apply` and the validations for the `hello-world-app` module:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  SKIP_deploy_db=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'

(...)
```

```
The 'SKIP_deploy_db' environment variable is set,
so skipping stage 'deploy_db'.

(...)

The 'deploy_app' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'validate_app' environment variable is not set,
so executing stage 'deploy_db'.

(...)

The 'teardown_app' environment variable is set,
so skipping stage 'deploy_db'.

(...)

The 'teardown_db' environment variable is set,
so skipping stage 'deploy_db'.

(...)

PASS
ok      terraform-up-and-running        13.824s
```

Notice how fast each of these test runs is now: instead of waiting 10-15 minutes after every change, you can try out new changes in 10 - 60 seconds (depending on the change). Since you're likely to re-run these stages dozens or even hundreds of times during development, the time savings can be massive.

Once the `hello-world-app` module changes are working the way you expect, it's time to clean everything up. Run the tests once more, this time skipping the deploy and validation stages, so only the teardown stages get executed:

```
$ SKIP_deploy_db=true \
  SKIP_deploy_app=true \
  SKIP_validate_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'

(...)

The 'SKIP_deploy_db' environment variable is set,
so skipping stage 'deploy_db'.

(...)

The 'SKIP_deploy_app' environment variable is set,
so skipping stage 'deploy_app'.

(...)

The 'SKIP_validate_app' environment variable is set,
so skipping stage 'validate_app'.

(...)

The 'SKIP_teardown_app' environment variable is not set,
so executing stage 'teardown_app'.

(...)

The 'SKIP_teardown_db' environment variable is not set,
so executing stage 'teardown_db'.
```

```
(...)

PASS
ok      terraform-up-and-running        340.02s
```

Using test stages lets you get rapid feedback from your automated tests, dramatically increasing the speed and quality of iterative development. It won't make any difference in how long tests take in your CI environment, but the impact on the development environment is huge.

RETRIES

Once you start running automated tests for your infrastructure code on a regular basis, you're likely to run into a problem: flaky tests. That is, tests occasionally will fail for transient reasons, such as an EC2 Instance occasionally failing to launch, or a Terraform eventual consistency bug, or a TLS handshake error talking to S3. The infrastructure world is a messy place, so you should expect intermittent failures in your tests, and handle them accordingly.

To make your tests a bit more resilient, you can add retries for known errors. For example, while writing this book, I'd occasionally get the following type of error, especially when running many tests in parallel:

```
* error loading the remote state: RequestError: send request failed
Post https://xxx.amazonaws.com/: dial tcp xx.xx.xx.xx:443:
connect: connection refused
```

To make tests more reliable in the face of such errors, you can enable retries in Terratest using the `MaxRetries`, `TimeBetweenRetries`, and `RetryableTerraformErrors` arguments of `terraform.Options`:

```go
func createHelloOpts(
        dbOpts *terraform.Options,
        terraformDir string) *terraform.Options {

        return &terraform.Options{
                TerraformDir: terraformDir,

                Vars: map[string]interface{}{
                        "db_remote_state_bucket": dbOpts.BackendConfig["bucket"],
                        "db_remote_state_key": dbOpts.BackendConfig["key"],
                        "environment": dbOpts.Vars["db_name"],
                },

                // Retry up to 3 times, with 5 seconds between retries,
                // on known errors
                MaxRetries: 3,
                TimeBetweenRetries: 5 * time.Second,
                RetryableTerraformErrors: map[string]string{
                        "RequestError: send request failed": "Throttling issue?",
                },
        }
}
```

In the `RetryableTerraformErrors` argument, you can specify a map of known errors that warrant a retry: the keys of the map are the error messages to look for the in the logs (you can use regular expressions here) and the values are additional information to display in the logs when Terratest matches one of these errors and kicks off a retry. Now, whenever your test code hits one of these known errors, you should see a message in your logs, followed by a sleep of`TimeBetweenRetries`, and then your command will re-run:

```
$ go test -v -timeout 30m

(...)

Running command terraform with args [apply -input=false -lock=false -auto-approve]

(...)

* error loading the remote state: RequestError: send request failed
```

```
Post https://s3.amazonaws.com/: dial tcp 11.22.33.44:443:
connect: connection refused

(...)

'terraform [apply]' failed with the error 'exit status code 1'
but this error was expected and warrants a retry. Further details:
Intermittent error, possibly due to throttling?

(...)

Running command terraform with args [apply -input=false -lock=false -auto-approve]
```

End-to-end tests

Now that you have unit tests and integration tests in place, the final type of tests you may want to add are end-to-end tests. With the Ruby web server example, end-to-end tests might consist of deploying the web server, any data stores it depends on, and testing it from the web browser using a tool such as Selenium. The end-to-end tests for Terraform infrastructure will look similar: deploy everything into an environment that mimics production and test it from the end-user's perspective.

Although you could write your end-to-end tests using the exact same strategy as the integration tests—that is, create a few dozen test stages to run `terraform apply`, do some validations, and then `terraform destroy`—this is rarely done in practice. The reason for this has to do with the *test pyramid* shown in Figure 7-1:

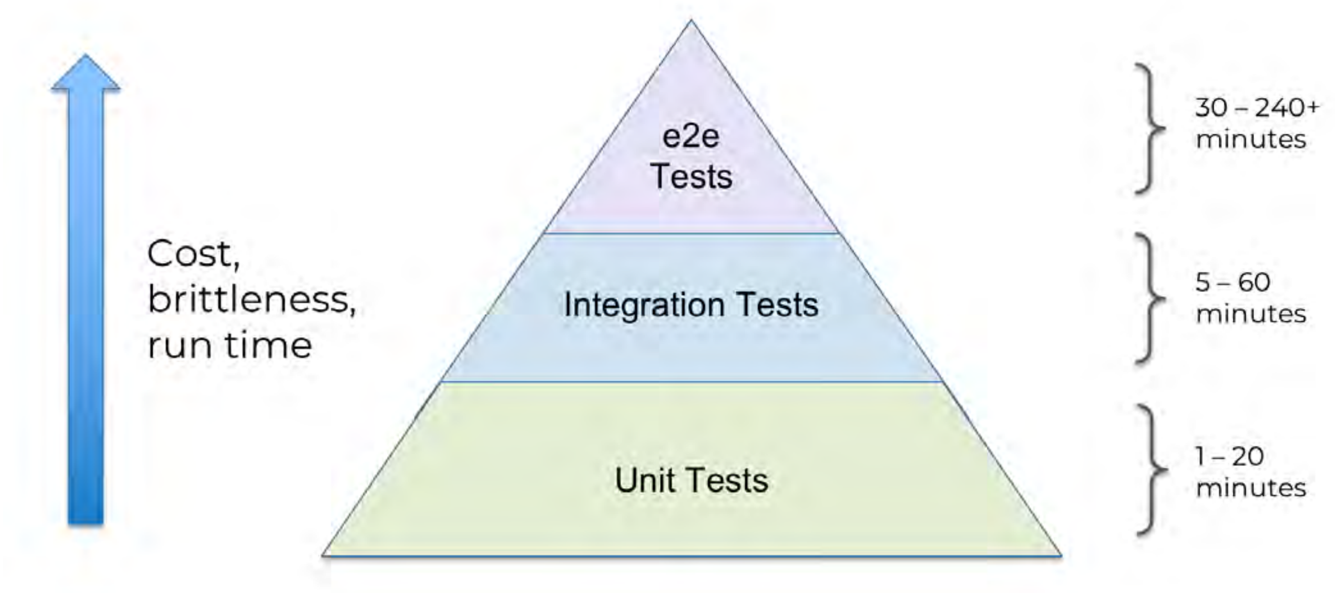


Figure 7-1. The test pyramid

The idea of the test pyramid is that you should typically be aiming for a large number of unit tests (the bottom of the pyramid), a smaller number of integration tests (the middle of the pyramid), and an even smaller number of end-to-end tests (the top of the pyramid). This is because, as you go up the pyramid, the cost and complexity of writing the tests, the brittleness of the tests, and the run time of the tests all increase.

That gives us *key testing takeaway #5*: smaller modules are easier and faster to test.

You saw in the previous sections that it took a fair amount of work with namespacing, dependency injection, retries, error handling, and test stages to test even a relatively simple `hello-world-app` module. With larger and more complicated infrastructure, it only gets harder. Therefore, You want to do as much of your testing as low in the pyramid as you can, as the bottom of the pyramid offers the fastest, most reliable feedback loop.

In fact, by the time you get to the top of the test pyramid, running tests to deploy a complicated architecture from scratch becomes untenable for two main reasons:

1. **Too slow**: Deploying your entire architecture from scratch, and then undeploying it all again, can take a very long time: on the order of several hours. Test suites that take that long provide relatively little value, as the feedback loop is simply too slow. You'd probably only run such a test suite overnight, which means in the morning, you'll get a report about a test failure, you'll investigate for a while, submit a fix, and then wait for the next day to see if it worked. That limits you to roughly one bug fix attempt per day. In these sorts of situations, what actually happens is developers start blaming others for test failures, convincing management to deploy despite the test failures, and eventually, ignoring the test failures entirely.
2. **Too brittle**: As mentioned in the previous section, the infrastructure world is messy. As the amount of infrastructure you're deploying goes up, the odds of hitting an intermittent, flaky issue goes up as well. For example, let's say that a single resource (e.g., such as an EC2 Instance) has a one in a thousand chance (0.1%) of failing due to an intermittent error (actual failure rates in the DevOps world are likely higher). That means that the probability that a test that deploys a single resource runs without any intermittent errors is 99.9%. So what about a test that

deploys two resources? In order for that test to succeed, you need both resources to deploy without intermittent errors, and to calculate those odds, you multiply the probabilities: $99.9\% \times 99.9\% = 99.8\%$ 99.9%×99.9%=99.8%. With three resources, the odds are $99.9\% \times 99.9\% \times 99.9\% = 99.7\%$ 99.9%×99.9%×99.9%=99.7%. With N resources, the formula is $99.9\%^N$ 99.9%N.

So now let’s consider different types of automated tests. If you had a unit test of a single module that deployed, say, 20 resources, the odds of success are $99.9\%^{20} = 98.0\%$ 99.9%20=98.0%. That means that 2 test runs out of 100 will fail; if you add a few retries, you can typically make these tests fairly reliable. Now, let’s say you had an integration test of 3 modules that deployed 60 resources. Now the odds of success are $99.9\%^{60} = 94.1\%$ 99.9%60=94.1%. Again, with enough retry logic, you can typically make these tests stable enough to be useful. So what happens if you wanted to write an end-to-end test that deploys your entire infrastructure, which consists of 30 modules, or about 600 resources? The odds of success are $99.9\%^{600} = 54.9\%$ 99.9%600=54.9%. That means that nearly half of your test runs will fail for transient reasons!

You’ll be able to handle some of these errors with retries, but it quickly turns into a never ending game of whack-a-mole. You add a retry for a TLS handshake timeout, only to be hit by an APT repo downtime in your Packer template; you add retries to the Packer build, only to have the build fail due to a Terraform eventual consistency bug; just as you are applying the band-aid to that, the build fails due to a brief GitHub outage. And since end-to-end tests take so long, you only get about one attempt per day to fix these issues.

In practice, very few companies with complicated infrastructure run end-to-end tests that deploy everything *from scratch*. Instead, the more common test strategy for end-to-end tests works as follows:

1. One time, you pay the cost of deploying a persistent, production-like environment called “test,” and you leave that environment running.
2. Every time someone makes a change to your infrastructure, the end-to-end test does the following:
    a. Apply the infrastructure change to the test environment.
    b. Run validations against the test environment (e.g., use Selenium to test your code from the end-user’s perspective) to make sure everything is working.

By changing your end-to-end test strategy to only applying incremental changes, you’re reducing the number of resources that are being deployed at test time from several hundred to just a handful, so these tests will be faster and less brittle.

Moreover, this approach to end-to-end testing more closely mimics how you’ll be deploying those changes in production. After all, it’s not like you tear down and bring up your production environment from scratch to roll out each change. Instead, you apply each change incrementally, so this style of end-to-end testing offers a huge advantage: you can test not only that your infrastructure works correctly, but that the *deployment process* for that infrastructure works correctly too.

For example, a critical property you will want to test is that you can roll out updates to your infrastructure without downtime. The `asg-rolling-deploy`module you created claims to be able to do a zero-downtime, rolling deployment, but how do you validate that claim? Let’s add an automated test to do just that.

You can do this with just a few tweaks to the test you just wrote in *test/hello_world_integration_test.go*, as the `hello-world-app` module uses the `asg-rolling-deploy` module under the hood. The first step is to expose a`server_text` variable in *live/stage/services/hello-world-app/variables.tf*:

```
variable "server_text" {
  description = "The text the web server should return"
  default     = "Hello, World"
  type        = string
}
```

Pass this variable through to the `hello-world-app` module in *live/stage/services/hello-world-app/main.tf*:

```
module "hello_world_app" {
  source = "../../../../modules/services/hello-world-app"

  server_text = var.server_text

  environment            = var.environment
  db_remote_state_bucket = var.db_remote_state_bucket
  db_remote_state_key    = var.db_remote_state_key

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

The `hello-world-app` module uses the `server_text` argument in its User Data script, so every time you change that variable, it will force a (hopefully) zero-downtime deployment. Let’s try this out by adding one more test stage called`redeploy_app` to *test/hello_world_integration_test.go* at the end of the test, right after the `validate_app` stage:

```
func TestHelloWorldAppStageWithStages(t *testing.T) {
        t.Parallel()

        // Store the function in a short variable name solely to make the
        // code examples fit better in the book.
        stage := test_structure.RunTestStage

        // Deploy the MySQL DB
        defer stage(t, "teardown_db", func() { teardownDb(t, dbDirStage) })
        stage(t, "deploy_db", func() { deployDb(t, dbDirStage) })

        // Deploy the hello-world-app
        defer stage(t, "teardown_app", func() { teardownApp(t, appDirStage) })
        stage(t, "deploy_app", func() { deployApp(t, dbDirStage, appDirStage) })

        // Validate the hello-world-app works
        stage(t, "validate_app", func() { validateApp(t, appDirStage) })

        // Redeploy the hello-world-app
        stage(t, "redeploy_app", func() { redeployApp(t, appDirStage) })
}
```

Next, implement the new `redeployApp` method:

```
func redeployApp(t *testing.T, helloAppDir string) {
        helloOpts := test_structure.LoadTerraformOptions(t, helloAppDir)

        albDnsName := terraform.OutputRequired(t, helloOpts, "alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)

        // Start checking every 1s that the app is responding with a 200 OK
        stopChecking := make(chan bool, 1)
        waitGroup, _ := http_helper.ContinuouslyCheckUrl(
                t,
                url,
                stopChecking,
                1*time.Second,
        )

        // Update the server text and redeploy
        newServerText := "Hello, World, v2!"
        helloOpts.Vars["server_text"] = newServerText
        terraform.Apply(t, helloOpts)

        // Make sure the new version deployed
        maxRetries := 10
        timeBetweenRetries := 10 * time.Second
        http_helper.HttpGetWithRetryWithCustomValidation(
                t,
                url,
                maxRetries,
                timeBetweenRetries,
                func(status int, body string) bool {
                        return status == 200 &&
                                strings.Contains(body, newServerText)
                },
        )

        // Stop checking
        stopChecking <- true
        waitGroup.Wait()
}
```

This method does the following:

1. Use the Terratest `http_helper.ContinuouslyCheckUrl` helper to run a goroutine (a lightweight thread managed by the Go runtime) in the background that, once every second, does an HTTP GET to the given ALB URL, and fails the test if at any point it gets back a response that isn't a 200 OK.
2. Update the new `server_text` variable and run `terraform apply` to kick off the rolling-deployment.
3. Once the deployment has completed, make sure the server is responding with the new `server_text` value.
4. Stop the goroutine that is continuously checking the ALB URL.

When you run this test and it gets to this final stage, you'll see log output that shows the continuous HTTP GET calls to the ALB interspersed with the `terraform apply` log output of the rolling deployment:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'

(...)

Making an HTTP GET call to URL
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com

Making an HTTP GET call to URL
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com

(...)

Running command terraform with args
[apply -input=false -lock=false -auto-approve]

(...)

Making an HTTP GET call to URL
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com

Making an HTTP GET call to URL
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com
Got response 200 and err <nil> from URL at
http://hello-world-test-thlMBF-168938547.us-east-2.elb.amazonaws.com

(...)

PASS
ok      terraform-up-and-running        551.04s
```

If this test passes, that means that the `hello-world-app` and `asg-rolling-deploy` module are able to do a zero-downtime, rolling deployment as promised! Every time your end-to-end test applies an incremental change to your test environment, you could use this strategy to make sure that the deployment does not cause downtime.

Other testing approaches

Most of this chapter has focused on using the Terratest approach for automated tests, but there are two other categories of testing approaches you may want to include as part of your testing tool belt:

1. Static analysis
2. Property testing

Just as different types of automated tests (unit, integration, end-to-end) catch different types of bugs, each of these testing approaches will catch different types of bugs as well, so you'll most likely want to use several of these together to get the best results. Let's go through these new categories one at a time.

STATIC ANALYSIS

There are several tools that can analyze your Terraform code without running it, including:

- `terraform validate`: This is a command built into Terraform that you can use to check your Terraform syntax and types (a bit like a compiler).
- tflint: A "lint" tool for Terraform that can scan your Terraform code and catch common errors and potential bugs based on a set of built-in rules.
- HashiCorp Sentinel: A "policy as code" framework that allows you to enforce rules across various HashiCorp tools. For example, you could create a policy to disallow security group rules in your Terraform code that allow inbound access `0.0.0.0/0`. At the time of writing, Sentinel is only available with HashiCorp Enterprise products, including Terraform Enterprise.

PROPERTY TESTING

There are a number of testing tools that focus on validating specific "properties" of your infrastructure, including:

- kitchen-terraform
- rspec-terraform
- serverspec
- inspec
- goss

Most of these tools provide a simple DSL (Domain Specific Language) for checking that the infrastructure you've deployed conforms to some sort of specification. For example, if you were testing a Terraform module that deployed an EC2 Instance, you could use the following `inspec` code to validate that the Instance has proper permissions on specific files, has certain dependencies installed, and is listening on a specific port:

```
describe file('/etc/myapp.conf') do
  it { should exist }
  its('mode') { should cmp 0644 }
end

describe apache_conf do
  its('Listen') { should cmp 8080 }
end

describe port(8080) do
  it { should be_listening }
end
```

The advantage of these tools is that the DLSs tend to be concise, easy to use, and offer an efficient, declarative way to validate a large number of properties about your infrastructure. This is great for enforcing checklists of requirements, especially around compliance requirements (e.g., PCI compliance, HIPAA compliance, etc). The disadvantage of these tools is that it's possible for all the property checks to pass and the infrastructure still doesn't work! For comparison, the "Terratest way" to validate these same properties would be to make an HTTP request to the server and see if you get back the expected response.

Conclusion

Everything in the infrastructure world is continuously changing: Terraform, Packer, Docker, Kubernetes, AWS, Google Cloud, Azure, and so on are all moving targets. That means that infrastructure code rots very quickly. Or to put it another way:

Infrastructure code without automated tests is broken.

I mean this both as an aphorism and as a literal statement. Every single time I've gone to write infrastructure code, no matter how much effort I put into keeping the code clean, testing it manually, and doing code reviews, as soon as I took the time to write automated tests, I found numerous, non-trivial bugs. Something magical happens when you take the time to automate the testing process and, almost without exception, it flushes out problems you otherwise would've never found yourself (but your customers would've). And not only do you find these bugs when you first add automated tests, but if you run your tests after every commit, you'll keep finding bugs over time, especially as the DevOps world changes all around you.

The automated tests I've added to my infrastructure code have not only caught bugs in my own code, but also bugs in the tools I was using, including non-trivial bugs in Terraform, Packer, Elasticsearch, Kafka, AWS, and so on. Writing automated tests as shown in this chapter is *not* easy: it takes considerable effort to write these tests; it takes even more effort to maintain them and add enough retry logic to make them reliable; and still more effort to keep your test environment clean to keep costs in check. But it's all worth it.

When I build a module to deploy a data store, for example, after every commit to that repo, my tests fire up a dozen copies of that data store in various configurations, write data, read data, and then tear everything back down. Each time those tests pass, that gives me huge confidence that my code still works. If nothing else, the automated tests let me sleep better. Those hours I spent dealing with retry logic and eventual consistency pay off in the hours I won't be spending at 3AM dealing with an outage.

THIS BOOK HAS TESTS TOO!

All the code examples in this book have tests too. You can find all the code examples, and all of their corresponding tests, at *https://github.com/brikis98/terraform-up-and-running-code*.

Throughout this chapter, you saw the basic process of testing Terraform code, including the following key takeaways:

1. **When testing Terraform code, there is no localhost.** Therefore, you have to do all of your manual testing by deploying real resources into one or more isolated sandbox environments.

2. **Regularly clean up your sandbox environments.** Otherwise, the environments will become unmanageable, and costs will spiral out of control.

3. **You cannot do pure unit testing for Terraform code.** Therefore, you have to do all of your automated testing by writing code that deploys real resources into one or more isolated sandbox environments.

4. **You must namespace all of your resources.** This ensures that multiple tests running in parallel do not conflict with each other.

5. **Smaller modules are easier and faster to test.** This was one of the key takeaways in Chapter 6 and it's worth repeating in this chapter too: smaller modules are easier to create, maintain, use, and test.

In the next chapter, you'll see how to incorporate Terraform code and your automated test code into your team's workflow, including how to manage environments, how to configure a CI / CD pipeline, and more.

---

1 AWS doesn't charge anything extra for additional AWS accounts, and if you use AWS Organizations, you can create multiple "child" accounts that all roll up their billing to a single "master" account.

2 In limited cases, it is possible to override the endpoints Terraform uses to talk to providers. For example, you can override the endpoint Terraform uses to talk to S3 and replace it with a mock endpoint that implements the S3 API. This works OK for a small number of endpoints, but most Terraform code makes *hundreds* of different API calls to the underlying provider, and mocking out all of them is impractical. Moreover, even if you do mock them all out, it's not clear that the resulting unit test can give you much confidence that your code works correctly: e.g., if you create mock endpoints for Auto Scaling Groups and ALBs, your `terraform apply`may succeed, but does that tell you anything useful about whether your code would've actually deployed a working app on top of that infrastructure?

3 Future versions of Terratest will most likely switch to using `go mod` for dependency management. At the time of writing, there was only preliminary support for `go mod`, but by the time Go 1.13 comes out, `go mod` should be enabled by default, so it'll likely become the standard tool for dependency management in Go—and remove the need for `GOPATH` as an added bonus. See *https://blog.golang.org/using-go-modules* for details.

Chapter 8. How to Use Terraform as a Team

As you've been reading this book and working through the code samples, you've most likely been working by yourself. In the real world, you'll most likely be working as part of a team, which introduces a number of new challenges. You may need to find a way to convince your team to use Terraform and other infrastructure as code (IAC) tools. You may need to deal with multiple people concurrently trying to understand, use, and modify the Terraform code you write. And you may need to figure out how to fit Terraform into the rest of your tech stack and make it part of your company's workflow.

In this chapter, I'll dive into the key processes you need to put in place to make Terraform and IAC work for your team:

- Adopting infrastructure as code in your team
- A workflow for deploying application code
- A workflow for deploying infrastructure code
- Putting it all together

Let's go through these topics one at a time.

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*.

Adopting infrastructure as code in your team

If your team is used to managing all of your infrastructure by hand, switching to infrastructure as code (IAC) requires more than just introducing a new tool or technology. It also requires changing the culture and processes of the team. Changing culture and process is a significant undertaking, especially at larger companies. Since every team's culture and process is a little different, there's no one-size-fits all way to do it, but here are a few tips that will be useful in most situations:

1. How to convince your boss
2. Work incrementally
3. Give your team the time to learn

How to convince your boss

I've seen this story play out many times at many companies: a developer discovers Terraform, becomes inspired by what it can do, shows up to work full of enthusiasm and excitement, shows Terraform to everyone… and the boss says "no." The developer, of course, becomes frustrated and discouraged. Why doesn't everyone else see the benefits of this? We could automate everything! We could avoid so many bugs! How else can we pay down all this tech debt? How can you all be so blind??

The problem is that while this developer sees all the benefits of adopting an IAC tool such as Terraform, they aren't seeing all the costs. Here are are just a few of the costs of adopting IAC:

Skills gap

The move to IAC means that your Ops team will have to spend most of its time writing large amounts of code: Terraform modules, Go tests, Chef recipes, and so on. While some Ops engineers are comfortable with coding all day and will love the change, others will find this a tough transition. Many Ops engineers and sysadmins are used to making changes manually, with perhaps an occasional short script here or there, and the move to doing software engineering nearly full-time may require learning a number of new skills or hiring new people.

New tools

Software developers can become attached to the tools they use; some are nearly religious about it. Every time you introduce a new tool, some developers will be thrilled at the opportunity to learn something new, but others will prefer to stick to what they know, and resist having to invest lots of time and energy learning new languages and techniques.

Change in mindset

If your team members are used to managing infrastructure manually, then they are used to making all of their changes *directly*: e.g., by SSHing to a server and executing a few commands. The move to IAC requires a shift in mindset where you make all of your changes *indirectly*, first by editing code, then checking it in, and then letting some automated process apply the changes. This layer of indirection can be frustrating, as for simple tasks, it'll feel slower than the direct option, especially when you're still learning a new IAC tool and are not efficient with it.

Opportunity cost

If you choose to invest your time in resources in one project, you are implicitly choosing not to invest that time and resources in other projects. What projects will you have to be put on hold so you can migrate to IAC? How important are those projects?

Some developers on your team will look at this list and become excited. But many others will groan—including your boss. Learning new skills, mastering new tools, and adopting new mindsets may or may not be beneficial, but one thing is certain: it is not free. Adopting IAC is a significant investment, and as with any investment, you have to consider not only the potential upside, but also the potential downsides.

Your boss in particular will be sensitive to the opportunity cost. One of the key responsibilities of any manager is to make sure the team is working on the highest priority projects. When you show up and excitedly start talking about Terraform, what your boss might really be hearing is, "oh no, this sounds like a massive undertaking, how much time is it going to take?" It's not that your boss is blind to what Terraform can do, but if you are spending time on that, you might not have time to deploy the new app the search team has been asking about for months, or to prepare for the PCI audit, or to dig into the outage from last week. So, if you want to convince your boss that your team should adopt IAC, your goal is not to prove that IAC has value, but that it'll bring more value to your team than anything else you could work on during that time.

One of the least effective ways to do this is to just list the features of your favorite IAC tool: e.g., Terraform is declarative; it's multi-cloud; it's open source. This is one of many areas where developers would do well to learn from sales people. Most sales people know that focusing on features is typically an ineffective way to sell products. A slightly better technique is to focus on benefits: that is, instead of talking about what a product can do ("product X can do Y!"), you should talk about what the customer can do by using that product ("you can do Y by using product X!"). In other words, show the customer what new superpowers your product can give them.

For example, instead of telling your boss that Terraform is declarative, talk about how your team will be able to get projects done faster. Instead of talking about the fact that Terraform is multi-cloud, talk about the peace of mind your boss can have knowing that if you migrate clouds some day, you won't have to change all your tooling; and instead of explaining to your boss that Terraform is open source, help your boss see how much easier it will be to hire new developers for the team from a large, active open source community.

Focusing on benefits is a great start, but the best sales people know an even more effective strategy: focus on the problems. If you watch a great sales person talking to a customer, you'll notice that it's actually the customer that does most of the talking. The sales person spends most of their time listening, and looking for one specific thing: what is the key problem that customer is trying to solve? What's the biggest pain point? Instead of trying to sell some sort of features or benefits, the best sales people try to solve their customer's problems. If that solution happens to include the product they are selling, all the better, but the real focus is on problem solving, not selling.

Talk to your boss and try to understand what are the most important problems they are working on that quarter or that year. You may find that those problems would not be solved by IAC. And that's OK! It may be slightly heretical for the author of a book on Terraform to say this, but not every team needs IAC. Adopting IAC has a relatively high cost, and while it'll pay off in the long term for some scenarios, it won't for others: e.g., if you're at a tiny startup with just one Ops person, or you're working on a prototype that may be thrown away in a few months, or you're just working on a side project for fun, then managing infrastructure by hand is often the right choice. Sometimes, even if IAC would be a great fit for your team, it won't be the highest priority, and given limited resources, working on other projects may still be the right choice.

If you do find that one of the key problems your boss is focused on can be solved with IAC, then your goal is to show your boss what that world looks like. For example, perhaps the biggest issue your boss is focused on this quarter is improving up-time. You've had numerous outages the last few months, many hours of downtime, customers are complaining, and the CEO is breathing down your manager's neck, checking in daily to see how things are going. You dig in and find out that more than half of these outages were caused by a manual error during deployment: e.g., someone accidentally skipped an important step during the roll out process, or a server was misconfigured, or the infrastructure in staging didn't match what you had in production.

Now, when you talk to your boss, instead of talking about Terraform features or benefits, lead with the following: "I have an idea for how to reduce our outages in half." I guarantee this will get your boss's attention. Use this opportunity to paint a picture for your boss of a world where your deployment process is fully automated, reliable, and repeatable, so that the manual errors that caused half of your previous outages are no longer possible. Not only that, but if deployment is automated, you can also add automated tests, reducing outages further, and allowing the whole company to deploy twice as often. Let your boss dream of being the one to tell the CEO that they've managed to cut outages in half and doubled deployments. And then mention that, based on your research, you believe you can deliver this future world using Terraform.

There's no guarantee your boss will say yes, but your odds are quite a bit higher with this approach. And your odds get even better if you work incrementally.

Work incrementally

One of the most important lessons I've learned in my career is that most large software projects fail. Whereas roughly three out of four of small IT projects (less than $1 million) are completed successfully, only one out of ten large projects (greater than $10 million) are completed on time and on budget, and more than one third of large projects are never completed at all.[1]

This is why I always get worried when I see a team try to not only adopt IAC, but to do so all at once, across a huge amount of infrastructure, across every team, and often as part of an even bigger initiative. I can't help but shake my head when I see the CEO or CTO of a large company give marching orders that everything must be migrated to the cloud, the old data centers must be shut down, and that everyone will "do DevOps" (whatever that means), all within 6 months. I'm not exaggerating when I say that I've seen this pattern several dozen times, and without exception, every single one of these initiatives has failed. Inevitably, 2-3 years later, every one of these companies is still working on the migration, the old data center is still running, and no one can tell if they are really doing DevOps or not.

If you want to successfully adopt IAC, or if you want to succeed at any other type of migration project, the only sane way to do it is incrementally. The key to *incrementalism* is not just splitting up the work into a series of small steps, but to split up the work in such a way where every step brings its own value—even if the later steps never happen.

To understand why this is so important, consider the opposite, *false incrementalism*.[2] Let's say you do a huge migration project, broken up into several small steps, but the project doesn't offer any real value until the very final step is completed. For example, the first step is to rewrite the frontend, but you don't launch it, as it relies on a new backend. Then you rewrite the backend, but you don’t launch that either, as it doesn’t work until data is migrated to a new data store. And then, finally, the last step is to do the data migration. Only after this last step do you finally launch everything and start getting any value from doing all this work. Waiting until the very end of a project to get any value is a big risk. If that project gets canceled or put on hold or significantly changed part way through, you might get zero value out of it, despite a lot of investment.

In fact, this is exactly what happens with many large migration projects. The project is big to begin with, and like most software projects, it takes much longer than expected. During that time, market conditions change, or the original stakeholders lose patience (e.g., the CEO was OK with spending 3 months to clean up tech debt, but after 12 months, it’s time to start shipping new products), and the project gets canceled before completion. With false incrementalism, this gives you the worst possible outcome: you’ve paid a huge cost and gotten absolutely nothing in return.

Therefore, incrementalism is essential. You want each part of the project to deliver some value so that even if the project doesn’t finish, no matter what step you got to, it was still worth doing. The best way to accomplish this is to focus on solving one, small, concrete problem at a time. For example, instead of trying to do a “big bang” migration to the cloud, try to identify one, small, specific app or team that is struggling, and work to migrate just them. Or instead of trying to do a “big bang” move to “DevOps,” try to identify a single, small, concrete problem (e.g., outages during deployment) and put in place a solution for that specific problem (e.g., automate the most problematic deployment with Terraform).

If you can get a quick win by fixing one real, concrete problem quickly, and making one team successful, then you’ll start to build momentum. That team may become your cheerleader and help convince other teams to migrate too. Fixing the specific deployment issue may make the CEO happy and get you support to use IAC for more projects. This will allow you to go for another quick win, and another one after that. And if you can keep repeating this process, delivering value early and often, you’ll be far more likely to succeed at the larger migration effort. But even if the larger migration doesn’t work out, then at least one team is more successful now, and one deployment process works better, so it was still worth the investment.

Give your team the time to learn

I hope that, at this point, it’s clear that adopting IAC can be a significant investment. It’s not something that will happen overnight. It’s not something that will happen magically, just because the manager gives you a nod. It will only happen through a deliberate effort of getting everyone on board, making learning resources (e.g., documentation, video tutorials, and of course, this book!) available, and providing dedicated time for team members to ramp up.

If your team doesn’t get the time and resources they need, then your IAC migration is unlikely to be successful. No matter how nice your code is, if your whole team isn’t on board with it, here’s how it’ll play out:

1. One developer on the team is passionate about IAC and spends a few months writing beautiful Terraform code, and using it to deploy lots of infrastructure.
2. The developer is happy and productive, but unfortunately, the rest of the team did not get the time to learn and adopt Terraform.
3. The, the inevitable happens: an outage.
4. Now one of the other team members needs to deal with the outage. They have two options: either (a) they fix the outage the way they’ve always done it, by making changes manually, which takes a few minutes or (b) they fix the outage by using Terraform, but they aren’t familiar with it, so this could take hours or days. Your team members are probably reasonable, rational people, and will almost always choose option (a).
5. Now, as a result of the manual change, the Terraform code no longer matches what’s actually deployed. Therefore, even if someone on the team picks option (b) and tries to use Terraform, there’s a chance they will get a weird error. If they do, they will lose trust in the Terraform code and once again fall back to option (a), making more manual changes. This makes the code even more out of sync with reality, so the odds of the next person getting a weird Terraform error are even higher, and you quickly get into a cycle where team members make more and more manual changes.
6. In a remarkably short time, everyone is back to doing everything manually, the Terraform code is completely unusable, and the months spent writing it are a total waste.

The scenario above isn’t hypothetical, but something I’ve seen happen at many different companies. They have large, expensive codebases full of beautiful Terraform code that is just gathering dust. To avoid this scenario, you need to not only convince your boss that you should use Terraform, but to also give everyone on the team the time they need to learn the tool and internalize how to use it, so when the next outage happens, it’s easier to fix it in code than it is to do it by hand.

One thing that can help teams adopt IAC faster is to have a well-defined process for using it. When you’re learning or using IAC on a small team, running it ad-hoc on a developer’s computer is good enough. But as your company and IAC usage grows, you will want to define a more systematic, repeatable, automated workflow for how deployments happen.

A workflow for deploying application code

In this section, I’ll introduce a typical workflow for taking application code (e.g., a Ruby on Rails or Java/Spring app) from development all the way to production. This workflow is reasonably well understood in the DevOps industry, so you’ll probably be familiar with parts of it. Later in this chapter, I’ll talk about a workflow for taking infrastructure code (e.g., Terraform modules) from development to production. This workflow is not nearly as well known in the industry, so it’ll be helpful to compare that workflow side by side with the application workflow, to understand how to translate each application code step to an analogous infrastructure code step.

Here’s what the application code workflow looks like:

1. Use version control
2. Run the code locally
3. Make code changes

4. Submit changes for review
5. Run automated tests
6. Merge and release
7. Deploy

Let’s go through these steps one at a time.

Use version control

All of your code should be in version control. No exceptions. It was the #1 item on the classic Joel Test when Joel Spolsky created it nearly 20 years ago, and the only things that have changed since then are that (a) with tools like GitHub, it’s easier than ever to use version control and (b) you can represent more and more things as code. This includes documentation (e.g., a README written in Markdown), application configuration (e.g., a config file written in YAML), specifications (e.g., test code written with RSpec), tests (e.g., automated tests written with JUnit), databases (e.g., schema migrations written in Active Record), and of course, infrastructure.

As in the rest of this book, I’m going to assume that you’re using Git for version control. For example, here is how you can checkout the code sample repos for this book:

```
$ git clone https://github.com/brikis98/terraform-up-and-running-code.git
```

By default, this checks out the `master` branch of your repo, but you’ll most likely do all of your work in a separate branch. Here’s how you can create a branch called `example-feature` and switch to it using the `git checkout` command:

```
$ cd terraform-up-and-running-code
$ git checkout -b example-feature
Switched to a new branch 'example-feature'
```

Run the code locally

Now that the code is on your computer, you can run it locally. You may recall the Ruby web server example from Chapter 7, which you can run as follows:

```
$ cd code/ruby/08-terraform/team
$ ruby web-server.rb

[2019-06-15 15:43:17] INFO  WEBrick 1.3.1
[2019-06-15 15:43:17] INFO  ruby 2.3.7 (2018-03-28) [universal.x86_64-darwin17]
[2019-06-15 15:43:17] INFO  WEBrick::HTTPServer#start: pid=28618 port=8000
```

And now you can manually test it with `curl`:

```
$ curl http://localhost:8000
Hello, World
```

Alternatively, you can run the automated tests:

```
$ ruby web-server-test.rb

(...)

Finished in 0.633175 seconds.
--------------------------------------------
8 tests, 24 assertions, 0 failures, 0 errors
100% passed
--------------------------------------------
```

The key thing to notice is that both manual and automated tests for application code can run completely locally on my own computer. You’ll see later in this chapter that this is not true for the same part of the workflow for infrastructure changes.

Make code changes

Now that you can run the application code, you can start making changes. This is an iterative process where you make a change, re-run your manual or automated tests to see if the change worked, make another change, re-run the tests, and so on.

For example, you can change the output of `web-server.rb` to “Hello, World v2”, restart the server, and see the result:

```
$ curl http://localhost:8000
Hello, World v2
```

And re-run the tests to see if they pass:

```
$ ruby web-server-test.rb

(...)

Failure: test_integration_hello(TestWebServer)
web-server-test.rb:53:in `block in test_integration_hello'
     50:     do_integration_test('/', lambda { |response|
     51:       assert_equal(200, response.code.to_i)
     52:       assert_equal('text/plain', response['Content-Type'])
  => 53:       assert_equal('Hello, World', response.body)
     54:     })
     55:   end
     56:
web-server-test.rb:100:in `do_integration_test'
web-server-test.rb:50:in `test_integration_hello'
<"Hello, World"> expected but was
<"Hello, World v2">

(...)

Finished in 0.236401 seconds.
--------------------------------------------
8 tests, 24 assertions, 2 failures, 0 errors
75% passed
--------------------------------------------
```

You immediately get feedback that the automated tests are still expecting the old value, so you can quickly go in and fix them. The idea in this part of the workflow is to optimize the feedback loop, so the time between making a change and seeing if it worked is minimized.

As you work, you should regularly be committing your code, with clear commit messages explaining the changes you’ve made:

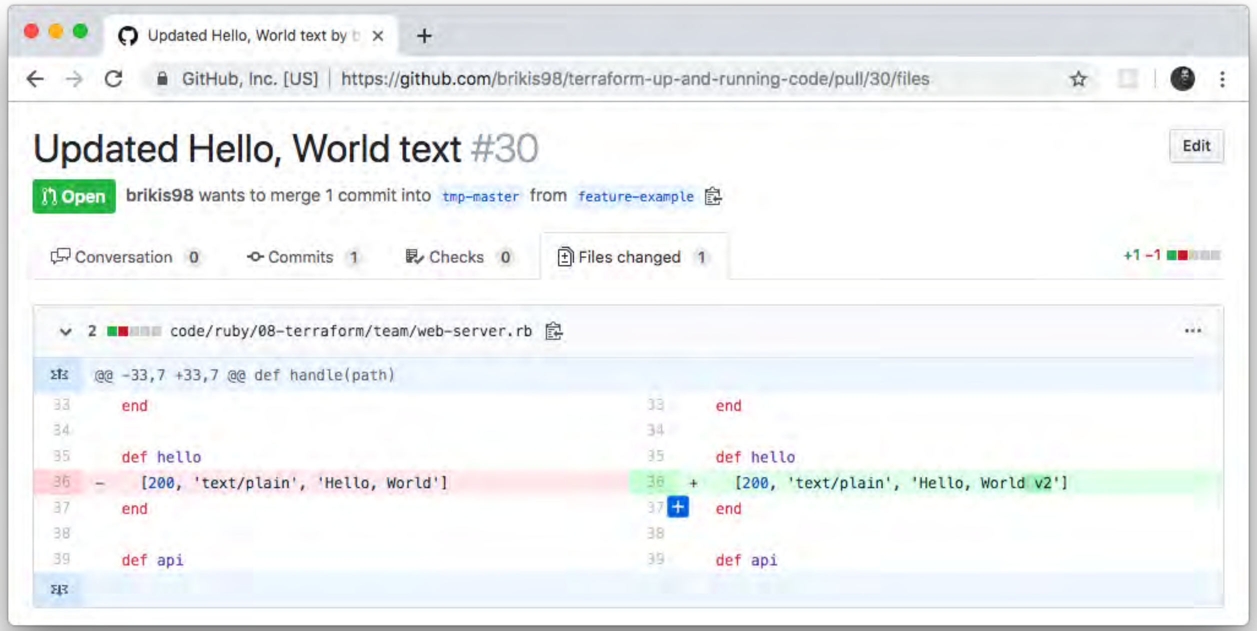
```
$ git commit -m "Updated Hello, World text"
```

Submit changes for review

Eventually, the code and tests will work the way you want them to, so it’s time to submit your changes for a code review. You can do this either with a separate code review tool (e.g., Phabricator or ReviewBoard) or, if you’re using GitHub, you can create a *pull request*. There are several different ways to create a pull request. One of the easiest is to `git push` your `example-feature` branch back to `origin` (that is, back to GitHub itself), and GitHub will automatically print out a pull request URL in the log output:

```
$ git push origin example-feature

(...)

remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'example-feature' on GitHub by visiting:
remote:      https://github.com/<OWNER>/<REPO>/pull/new/example-feature
remote:
```

Open that URL in your browser, fill out the pull request title and description, and click create. Your team members will now be able to review the changes, as shown in Figure 8-1.

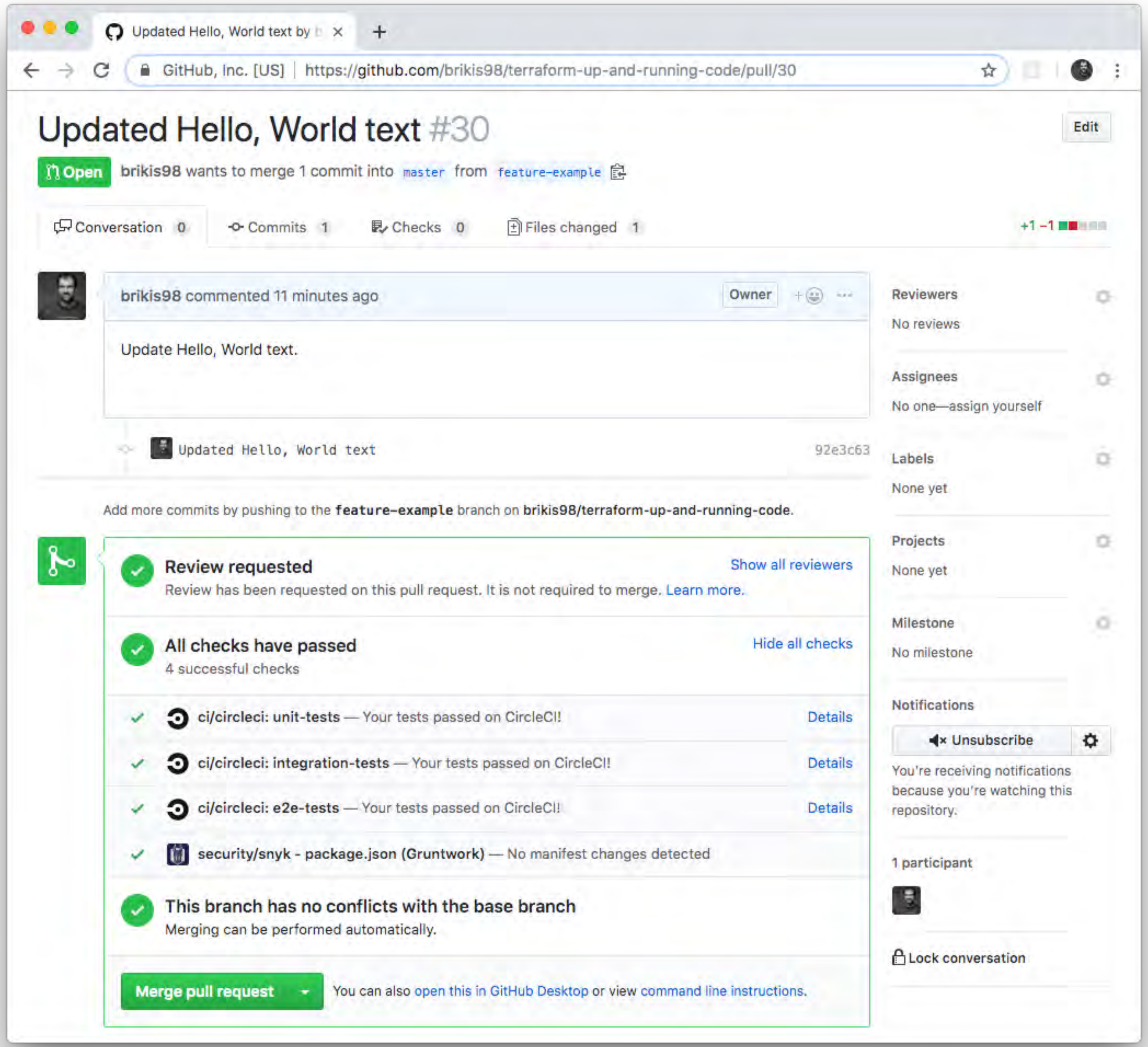

Updated Hello, World text by b
GitHub, Inc. [US] | https://github.com/brikis98/terraform-up-and-running-code/pull/30/files
Updated Hello, World text #30
Edit
Open
brikis98 wants to merge 1 commit into tmp-master from feature-example
Conversation 0
Commits 1
Checks 0
Files changed 1
+1 −1

```
2 code/ruby/08-terraform/team/web-server.rb
@@ -33,7 +33,7 @@ def handle(path)
   end                                            end

   def hello                                      def hello
-    [200, 'text/plain', 'Hello, World']     +      [200, 'text/plain', 'Hello, World v2']
   end                                            end

   def api                                        def api
```

Figure 8-1. GitHub Pull Request

Run automated tests

You should set up commit hooks to run automated tests for every commit you push to your version control system. The most common way to do this is to use a *Continuous Integration Server (CI Server)*, such as Jenkins, CircleCI, or TravisCI. Most popular CI Servers have integrations built-in specifically for GitHub, so not only does every commit automatically run tests, but the output of those tests shows up in the pull request itself, as shown in Figure 8-2.


Updated Hello, World text #30

Open brikis98 wants to merge 1 commit into master from feature-example

Conversation 0 · Commits 1 · Checks 0 · Files changed 1 · +1 −1

brikis98 commented 11 minutes ago · Owner

Update Hello, World text.

Updated Hello, World text · 92e3c63

Add more commits by pushing to the **feature-example** branch on **brikis98/terraform-up-and-running-code**.

Review requested — Show all reviewers
Review has been requested on this pull request. It is not required to merge. Learn more.

All checks have passed — Hide all checks
4 successful checks

✓ **ci/circleci: unit-tests** — Your tests passed on CircleCI! · Details
✓ **ci/circleci: integration-tests** — Your tests passed on CircleCI! · Details
✓ **ci/circleci: e2e-tests** — Your tests passed on CircleCI! · Details
✓ **security/snyk - package.json (Gruntwork)** — No manifest changes detected

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request · You can also open this in GitHub Desktop or view command line instructions.

Reviewers: No reviews · Assignees: No one—assign yourself · Labels: None yet · Projects: None yet · Milestone: No milestone · Notifications: Unsubscribe · You're receiving notifications because you're watching this repository. · 1 participant · Lock conversation

Figure 8-2. GitHub Pull Request showing automated test results from CircleCi

You can see in Figure 8-2 that CircleCi has run unit tests, integration tests, end-to-end tests, and some static analysis checks (in the form of security vulnerability scanning using a tool called `snyk`) against the code in the branch, and everything passed.

Merge and release

Your team members should review your code changes, looking for potential bugs, enforcing coding guidelines (more on this later in the chapter), checking the existing tests passed, and ensuring you've added tests for any new behavior you've added. If everything looks good, your code can be merged into the `master` branch.

The next step is to release the code. If you're using immutable infrastructure practices (as discussed in "Server Templating Tools"), releasing application code means packaging that code into a new, immutable, versioned artifact. Depending on how you wish to package and deploy your application, the artifact can be a new Docker image, a new virtual machine image (e.g., new AMI), a new `.jar`file, a new `.tar` file, etc. Whatever format you pick, make sure the artifact is immutable (i.e., you never change it), and that it has a unique version number (so you can distinguish this artifact from all the others).

For example, if you are packaging your application using Docker, you can store the version number in a Docker tag. You could use the ID of the commit (the sha1 hash) as the tag so that you can map the Docker image you're deploying back to the exact code it contains:

```
$ commit_id=$(git rev-parse HEAD)
$ docker build -t brikis98/ruby-web-server:$commit_id .
```

The code above will build a new Docker image called `brikis98/ruby-web-server` and tag it with the ID of the most recent commit, which will look something like `92e3c6380ba6d1e8c9134452ab6e26154e6ad849`. Later on, if you're debugging an issue in a Docker image, you can see the exact code it contains by checking out the Commit ID the Docker image has as a tag:

```
$ git checkout 92e3c6380ba6d1e8c9134452ab6e26154e6ad849
HEAD is now at 92e3c63 Updated Hello, World text
```

One downside to commit IDs is that they aren't very readable or memorable. An alternative is to create a Git tag:

```
$ git tag -a "v0.0.4" -m "Update Hello, World text"
$ git push --follow-tags
```

A tag is a pointer to a specific Git commit, but with a friendlier name. You can use this Git tag on your Docker images:

```
$ git_tag=$(git describe --tags)
$ docker build -t brikis98/ruby-web-server:$git_tag .
```

And when debugging, check out the code at a specific tag:

```
$ git checkout v0.0.4
Note: checking out 'v0.0.4'.
(...)
HEAD is now at 92e3c63 Updated Hello, World text
```

Deploy

Now that you have a versioned artifact, it's time to deploy it. There are many different ways to deploy application code, depending on the type of application, how you package it, how you wish to run it, your architecture, what tools you're using, and so on. Here are a few of the key considerations:

1. Deployment tooling
2. Deployment strategies
3. Deployment server
4. Promote artifacts across environments

DEPLOYMENT TOOLING

There are many different tools you can use to deploy your application, depending on how you package it, and how you wish to run it. A few examples:

Terraform

> As you've seen in this book, you can use Terraform to deploy certain types of applications. For example, in earlier chapters, you created a module called `asg-rolling-deploy` that could do a zero-downtime rolling deployment across an Auto Scaling Group. If you package your application as an AMI (e.g., using Packer), you could deploy new AMI versions with the `asg-rolling-deploy` module by updating the `ami` parameter in your Terraform code and running `terraform apply`.

Docker orchestration tools

> There are a number of orchestration tools designed to deploy and manage Dockerized applications, including Kubernetes (arguably the most popular one), Apache Mesos, HashiCorp Nomad, and Amazon ECS. If you package your application as a Docker image, you could deploy new Docker image versions with Kubernetes by running `kubectl apply` (`kubectl` is the command-line app you use to interact with Kubernetes) and passing in a YAML file that defines which Docker image name and tag to deploy.

Scripts

> Terraform and most orchestration tools only support a limited set of deployment strategies (discussed in the next section). If you have more complicated requirements, you will most likely have to write custom scripts in a general purpose programming language (e.g., Python, Ruby), a configuration management tool (e.g., Ansible, Chef), or other server automation tools (e.g., Capistrano). For example, to automatically deploy updates to an Apache Kafka cluster, you might create an Python script that knows how to update one broker node at a time: e.g., connect to the Kafka broker over SSH, unregister the broker from any load balancers using the AWS CLI, initiate a graceful shutdown using the `kafka-server-stop.sh`script, wait for the shutdown to complete, and so on.

The trickiest thing about writing these sorts of scripts is handling failure. For example, what happens if the computer running your deployment script loses Internet connectivity or crashes part way through the deployment? Writing the script so it's idempotent and can recover from failures and complete the deployment correctly is not easy. You may need to provide a way for the script to record its state somewhere (though sometimes you can derive the state by querying your infrastructure), and build a finite state machine that can handle all possible starting and transition states.

DEPLOYMENT STRATEGIES

There are a number of different strategies you can use for application deployment, depending on your requirements. Let's say you have 5 copies of the old version of your app running and you want to roll out a new version. Here are a few of the most common strategies you can use:

Rolling deployment with replacement

Take down 1 of the old copies of the app, deploy a new copy to replace it, wait for the new copy to come up and pass health checks, start sending the new copy live traffic, and then repeat the process until all the old copies have been replaced. Rolling deployment with replacement ensures that you never have more than 5 copies of the app running, which can be useful if you have limited capacity (e.g., if each copy of the app runs on a physical server), or if you're dealing with a stateful system where each app has a unique identity (e.g., this is often the case with consensus systems, such as Apache ZooKeeper). Note that this deployment strategy can work with larger batch sizes (i.e., you can replace more than one copy of the app at a time, assuming you can handle the load and won't lose data with fewer apps running) and that during deployment, you will have both the old and new versions of the app running at the same time.

Rolling deployment without replacement

Deploy 1 new copy of the app, wait for the new copy to come up and pass health checks, start sending the new copy live traffic, undeploy an old copy of the app, and then repeat the process until all the old copies have been replaced. Rolling deployment without replacement works only if you have flexible capacity (e.g., your apps run in the cloud, where you can spin up new virtual servers any time you want) and if your application can tolerate more than 5 copies of it running at the same time. The advantage is that you never have less than 5 copies of the app running, so you're not running at a reduced capacity during deployment. Note that this deployment strategy can work with larger batch sizes (i.e., you can deploy 5 new copies simultaneously, which is exactly what you did with the `asg-rolling-deploy` module) and that during deployment, you will have both the old and new versions of the app running at the same time.

Blue-green deployment

Deploy 5 new copies of the app, wait for all of them to come up and pass health checks, shift all live traffic to the new copies, and then undeploy the old copies. Blue-green deployment works only if you have flexible capacity (e.g., your apps run in the cloud, where you can spin up new virtual servers any time you want) and if your application can tolerate more than 5 copies of it running at the same time. The advantage is that only one version of your app is visible to users at any given time, and that you never have less than 5 copies of the app running, so you're not running at a reduced capacity during deployment.

Canary deployment

Deploy 1 new copy of the app, wait for it to come up and pass health checks, start sending live traffic to it, and then pause the deployment. During the pause, compare the new copy of the app, called the "canary," to one of the old copies, called the "control." You can compare the canary and control across a variety of dimensions: CPU usage, memory usage, latency, throughput, error rates in the logs, HTTP response codes, and so on. Ideally, there's no way to tell the two servers apart, which should give you confidence that the new code works just fine. In that case, you unpause the deployment, and use one of the rolling deployment strategies to complete it. On the other hand, if you spot any differences, then that may be a sign of problems in the new code, and you can cancel the deployment and undeploy the canary before the problem gets worse.

The name comes from the "canary in a coal mine," where miners would take canary birds with them down into the tunnels, and if the tunnels filled with dangerous gasses (e.g., carbon monoxide), those gasses would kill the canary before killing the miners, thus providing an early warning to the miners that something is wrong and that they need to exit immediately, before more damage is done. The canary deployment offers similar benefits, giving you a systematic way to test new code in production in a way that, if something goes wrong, you get a warning early on, when it has only affected a small portion of your users, and you still have enough time to react and prevent further damage.

Canary deployments are often combined with *feature toggles*, where you wrap all new features in an if-statement. By default, the if-statement defaults to false, so the new feature is toggled off when you initially deploy the code. Since all new functionality is off, when you deploy the canary server, it should behave identically to the control, and any differences can be automatically flagged as a problem, and trigger a rollback. If there were no problems, then later on, you can enable the feature toggle for a portion of your users via an internal web interface. For example, you might initially only enable the new feature for employees; if that works well, you can enable it for 1% of users; if that's still working well, you can ramp it up to 10%; and so on. If at any point there's a problem, you can use the feature toggle to ramp the feature back down. This process allows you to separate *deployment*of new code from *release* of new features.

DEPLOYMENT SERVER

You should run the deployment from a CI server and not from a developer's computer. This has the following benefits:

Fully automated

To run deployments from a CI server, you'll be forced to fully automate all deployment steps. This ensures that your deployment process is captured as code, that you don't miss any steps accidentally due to manual error, and that the deployment is fast and repeatable.

You are running from a consistent environment

If developers run deployments from their own computers, then you'll run into bugs due to differences in how their computer is configured: e.g., different operating systems, different dependency versions (e.g., different versions of Terraform), different configurations, and differences in what's actually being deployed (e.g., the developer accidentally deploys a change that wasn't committed to version control). You can eliminate all these issues by deploying everything from the same CI server.

Better permissions management

Instead of giving every developer permissions to deploy, you can solely give the CI server those permissions (especially for the production environment). It's a lot easier to enforce good security practices for a single server than it is to for dozens or hundreds of developers with production access.

PROMOTE ARTIFACTS ACROSS ENVIRONMENTS

If you're using immutable infrastructure practices, then the way to roll out new changes is to promote the exact same versioned artifact from one environment to another. For example, if you have dev, staging, and production environments, to roll out `v0.0.4` of your app, you would do the following:

1. Deploy `v0.0.4` of the app to dev.
2. Run your manual and automated tests in dev.
3. If `v0.0.4` works well in dev, repeat steps 1-2 to deploy `v0.0.4` to staging (this is known as *promoting* the artifact).
4. If `v0.0.4` works well in staging, repeat steps 1-2 again to promote `v0.0.4`to production.

Since you're running the exact same artifact everywhere, there's a good chance that if it works in one environment, it'll work in another. And if you do hit any issues, you can roll back any time by deploying an older artifact version.

A workflow for deploying infrastructure code

Now that you've seen the workflow for deploying application code, it's time to dive into the workflow for deploying infrastructure code. In this section, when I say "infrastructure code," I code written with any IAC tool (including, of course, Terraform) that can be used to deploy arbitrary infrastructure changes beyond a single application: e.g., deploying databases, load balancers, network configurations, DNS settings, and so on.

Here's what the infrastructure code workflow looks like:

1. Use version control
2. Run the code locally
3. Make code changes
4. Submit changes for review
5. Run automated tests
6. Merge and release
7. Deploy

At the surface, it looks identical to the application workflow, but under the hood, there are important differences. Deploying infrastructure code changes is more complicated, and the techniques are not as well understood, so being able to relate each step back to the analogous step from the application code workflow should make it easier to follow along. Let's dive in.

Use version control

Just as with your application code, all of your infrastructure code should be in version control. That means you'll use `git clone` to checkout your code, just as before. However, version control for infrastructure code has a few extra requirements:

1. Live repo and modules repo
2. Golden rule of Terraform
3. The trouble with branches

LIVE REPO AND MODULES REPO

As discussed in Chapter 4, you will typically want at least two separate version control repositories for your Terraform code: one repo for modules and one repo for live infrastructure. The repository for modules is where you create your reusable, versioned modules, such as all the modules you built in the previous chapters of this book (`cluster/asg-rolling-deploy`, `data-stores/mysql`, `networking/alb`, and `services/hello-world-app`). The repository for live infrastructure defines the live infrastructure you've deployed in each environment (dev, stage, prod, etc).

One pattern that works well is to have one infrastructure team in your company that specializes in creating reusable, robust, production-grade modules. This team can create remarkable leverage for your company by building a library of modules that implement the ideas from Chapter 6: i.e., each module has a composable API, is thoroughly documented (including executable documentation in the `examples` folder), has a comprehensive suite of automated tests, is versioned, and implements all of your company's requirements from the production-grade infrastructure checklist (i.e., security, compliance, scalability, high availability, monitoring, and so on).

If you build such a library (or you buy one off the shelf[3]), all the other teams at your company will be able to consume these modules, a bit like a service catalog, to deploy and manage their own infrastructure, without (a) each team having to spend months assembling that infrastructure from scratch or (b) the Ops team becoming a bottleneck because they have to deploy and manage the infrastructure for every team. Instead, the Ops team can spend most of its time writing infrastructure code, and all the other teams will be able to work independently, using these modules to get themselves up and running. And since every team is using the same canonical modules under the hood, as the company grows and requirements change, the Ops team can push out new versions of the modules to all teams, ensuring everything stays consistent and maintainable.

Or it will be maintainable, as long as you follow the Golden Rule of Terraform.

THE GOLDEN RULE OF TERRAFORM

Here's a quick way to check the health of your Terraform code: go into your *live*repository, pick several folders at random, and run `terraform plan` in each one. If the output is always, "no changes," that's great, as it means your infrastructure code matches what's actually deployed. If the output sometimes shows a small diff, and you get the occasional excuse from your team members ("oh, right, I tweaked that one thing by hand and forgot to update the code"), then your code doesn't match reality, and you may soon be in trouble. If `terraform plan` fails completely with weird errors, or every `plan` shows a gigantic diff, then your Terraform code has no relation at all to reality, and is likely useless.

The gold standard, or what you're really aiming for, is what I call the *The Golden Rule of Terraform*:

> *The master branch of the live repository should be a 1:1 representation of what's actually deployed in production.*

Let's break this sentence down, starting at the end and working our way back:

"…what's actually deployed"

The only way to ensure that the Terraform code in the *live* repository is an up-to-date representation of what's actually deployed is to *never make out-of-band changes*. Once you start using Terraform, do not make changes via a web UI, or manual API calls, or any other mechanism. As you saw in Chapter 5, out-of-band changes not only lead to complicated bugs, but they also void many of the benefits you get from using infrastructure as code in the first place.

"…a 1:1 representation…"

If I browse your *live* repository, I should be able to see, from a quick scan, what resources have been deployed in what environments. That is, every resource should have a 1:1 match with some line of code checked into the *live*repo. This seems obvious at first glance, but it's surprisingly easy to get it wrong. One way to get it wrong, as I just mentioned, is to make out band changes, so the code is there, but the live infrastructure is different. A more subtle way to get it wrong is to use Terraform workspaces to manage environments, so that the live infrastructure is there, but the code isn't. That is, if you use workspaces, your *live* repo will only have one copy of the code, even though you may have 3 or 30 environments deployed with it. From merely looking at the code, there will be no way to know what's actually deployed, which will lead to mistakes and make maintenance complicated. Therefore, as described in "Isolation via workspaces", instead of using workspaces to manage environments, you want each environment defined in a separate folder, using separate files, so that you can see exactly what environments have been deployed just by browsing the *live* repository. Later in this chapter, you'll see how to do this with minimal copy/paste.

"The master branch…"

You should only have to look at a single branch to understand what's actually deployed in production. Typically, that branch will be `master`. That means all changes that affect the production environment should go directly into`master` (you can create a separate branch, but only to create a pull request with the intention of merging that branch into `master`) and you should only run `terraform apply` for the production environment against the `master`branch. In the next section, I'll explain why.

THE TROUBLE WITH BRANCHES

In Chapter 3, you saw that you can use the locking mechanisms built into Terraform backends to ensure that if two team members are running `terraform apply` at the same time on the same set of Terraform configurations, their changes do not overwrite each other. Unfortunately, this only solves part of the problem. While Terraform backends provide locking for Terraform state, they cannot help you with locking at the level of the Terraform code itself. In particular, if two team members are deploying the same code to the same environment, but from different branches, you'll run into conflicts that locking can't prevent.

For example, let's say one of your team members, Anna, makes some changes to the Terraform configurations for an app called "foo" that consists of a single EC2 Instance:

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

The app is getting a lot of traffic, so Anna decides to change the `instance_type`from `t2.micro` to `t2.medium`:

```
resource "aws_instance" "foo" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.medium"
}
```

Here's what Anna sees when she runs `terraform plan` (the log output below is truncated for readability):

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_instance.foo will be updated in-place
  ~ resource "aws_instance" "foo" {
        ami                          = "ami-0c55b159cbfafe1f0"
        id                           = "i-096430d595c80cb53"
        instance_state               = "running"
      ~ instance_type                = "t2.micro" -> "t2.medium"
        (...)
    }

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

Those changes look good, so she deploys them to staging.

In the meantime, Bill comes along and also starts making changes to the Terraform configurations for the same app, but on a different branch. All Bill wants to do is to add a tag to the app:

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

  tags = {
    Name = "foo"
  }
}
```

Note that Anna's changes are already deployed in staging, but as they are on a different branch, Bill's code still has the `instance_type` set to the old value of `t2.micro` . Here's what Bill sees when he runs the `plan` command (the log output below is truncated for readability):

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_instance.foo will be updated in-place
  ~ resource "aws_instance" "foo" {
        ami                          = "ami-0c55b159cbfafe1f0"
        id                           = "i-096430d595c80cb53"
        instance_state               = "running"
      ~ instance_type                = "t2.medium" -> "t2.micro"
      + tags                         = {
          + "Name" = "foo"
        }
        (...)
    }

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

Uh oh, he's about to undo Anna's `instance_type` change! If Anna is still testing in staging, she'll be very confused when the server suddenly redeploys and starts behaving differently. The good news is that if Bill diligently reads the `plan`output, he can spot the error before it affects Anna. Nevertheless, the point of the example is to highlight what happens when you deploy changes to a shared environment from different branches.

The locking from Terraform backends doesn't help here because the conflict has nothing to do with concurrent modifications to the state file; Bill and Anna may be applying their changes weeks apart, and the problem would be the same. The underlying cause is that branching and Terraform are a bad combination. Terraform is implicitly a mapping from Terraform code to infrastructure deployed in the real world. Since there's only one real world, it doesn't make much sense to have multiple branches of your Terraform code. So for any shared environment (e.g., stage, prod), always deploy from a single branch.

Run the code locally

Now that you've got the code checked out onto your computer, the next step is to run it. The gotcha with Terraform is that, unlike application code, you don't have "localhost": e.g., you can't deploy an AWS Auto Scaling Group onto your own laptop. As discussed in "Manual testing basics", the only way to manually test Terraform code is to run it in a sandbox environment, such as an AWS account dedicated for developers (or better yet, one AWS account for each developer).

Once you have a sandbox environment, to test manually, you run `terraform apply`:

```
$ terraform apply

(...)

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:

alb_dns_name = hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

And you verify the deployed infrastructure works using tools such as `curl`:

```
$ curl hello-world-stage-477699288.us-east-2.elb.amazonaws.com
Hello, World
```

To run automated tests written in Go, you use `go test` in a sandbox account dedicated to testing:

```
$ go test -v -timeout 30m

(...)

PASS
ok      terraform-up-and-running        229.492s
```

Make code changes

Now that you can run your Terraform code, you can iteratively start making changes, just as with application code. Every time you make a change, you can re-run `terraform apply` to deploy those changes and re-run `curl` to see if those changes worked:

```
$ curl hello-world-stage-477699288.us-east-2.elb.amazonaws.com
Hello, World v2
```

Or you can re-run `go test` to make sure the tests are still passing:

```
$ go test -v -timeout 30m

(...)

PASS
ok      terraform-up-and-running        229.492s
```

The only difference from application code is that infrastructure code tests typically take longer, so you'll want to put more thought into how you can shorten the test cycle so you can get feedback on your changes as quickly as possible. In "Test stages", you saw that test stages can be used to only re-run specific stages of a test suite, dramatically shortening the feedback loop.

For example, if you have a test that deploys a database, deploys an app, validates both work, undeploys the app, and undeploys the databse, then when you do the initial test run, you can skip the two undeploy steps to leave the app and database running:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'

(...)

PASS
ok      terraform-up-and-running        423.650s
```

And then each time you make a change to your app, you skip the database deploy and both undeploy steps, so only the app deploy and validation steps run:

```
$ SKIP_teardown_db=true \
  SKIP_teardown_app=true \
  SKIP_deploy_db=true \
  go test -timeout 30m -run 'TestHelloWorldAppStageWithStages'

(...)

PASS
ok      terraform-up-and-running        13.824s
```

This shortens the feedback loop from minutes to seconds, which will provide a huge boost to developer productivity.

As you make changes, make sure to regularly commit your work:

```
$ git commit -m "Updated Hello, World text"
```

Submit changes for review

Once your code is working the way you expect, you can create a pull request to get your code reviewed, just as you would with application code. Your team will review your code changes, looking for bugs, as well as enforcing *coding guidelines*. Whenever you're writing code as a team, regardless of what type of code you're writing, you should define guidelines for everyone to follow. One of my favorite definitions of "clean code" comes from an interview I did with Nick Dellamaggiore for my earlier book, *Hello, Startup*:

> *If I look at a single file and it's written by 10 different engineers, it should be almost indistinguishable which part was written by which person. To me, that is clean code.*
>
> *The way you do that is through code reviews and publishing your style guide, your patterns, and your language idioms. Once you learn them, everybody is way more productive because you all know how to write code the same way. At that point, it's more about what you're writing and not how you write it.*
>
> Nick Dellamaggiore, Infrastructure Lead at Coursera

The Terraform coding guidelines that make sense for each team will be different, so here, I'll list a few of the common ones that are useful for most teams:

- Documentation
- Automated tests
- File layout
- Style guide

DOCUMENTATION

In some sense, Terraform code is, in and of itself, a form of documentation. It describes in a simple language exactly what infrastructure you deployed and how that infrastructure is configured. However, there is no such thing as self-documenting code. While well-written code can tell you *what* it does, no programming language I'm aware of (including Terraform) can tell you *why* it does it.

This is why all software, including IAC, needs documentation beyond the code itself. There are several types of documentation you can consider, and have your team members require as part of code reviews:

Written documentation

Most Terraform modules should have a Readme that explains what the module does, why it exists, how to use it, and how to modify it. In fact, you may want to write the Readme first, before any of the actual Terraform code, as that will force you to consider *what* you're building and *why* you're building it before you dive into the code and get lost in the details of *how* to build it.[4] Spending 20 minutes writing a Readme can often save you hours of writing code that solves the wrong problem. Beyond the basic Readme, you may also want to have tutorials, API documentation, wiki pages, and design documents that go deeper into how the code works and why it was built this way.

Code documentation

> Within the code itself, you can use comments as a form of documentation. Terraform treats any text that starts with a hash (`#`) as a comment. Don't use comments to explain what the code does; the code should do that itself. Only include comments to offer information that can't be expressed in code, such as how the code is meant to be used or why the code uses a particular design choice. Terraform also allows every input and output variable to declare a `description` parameter, which is a great place to describe how that variable should be used.

Example code

> As discussed in Chapter 6, every Terraform module should include example code that shows how that module is meant to be used. This is a great way to highlight the intended usage patterns, give your users a way to try your module without having to write any code, and it's the main way to add automated tests for the module.

AUTOMATED TESTS

All of Chapter 7 focuses on testing Terraform code, so I won't repeat any of that here, other than to say that infrastructure code without tests is broken. Therefore, one of the most important comments you can make in any code review is, "how did you test this?"

FILE LAYOUT

Your team should define conventions for where Terraform code is stored and the file layout you use. Since the file layout for Terraform also determines the way Terraform state is stored, you should be especially mindful of how file layout impacts your ability to provide isolation guarantees, such as ensuring changes in a staging environment cannot accidentally cause problems in production. In a code review, you may want to enforce the file layout described in "Isolation via file layout", which provides isolation between different environments (e.g., stage and prod) and different components (e.g., a network topology for the entire environment and a single app within that environment).

STYLE GUIDE

Every team should enforce a set of conventions about code style, including the use of whitespace, newlines, indentation, curly braces, variable naming, and so on. Although programmers love to debate spaces versus tabs and where the curly brace should go, the actual choice isn't that important. What really matters is that you are consistent throughout your codebase. Formatting tools are available for most text editors and IDEs, and as commit hooks for version control systems to help you enforce a common code layout.

Terraform even has a built-in `fmt` command that can reformat code to a consistent style automatically:

```
$ terraform fmt
```

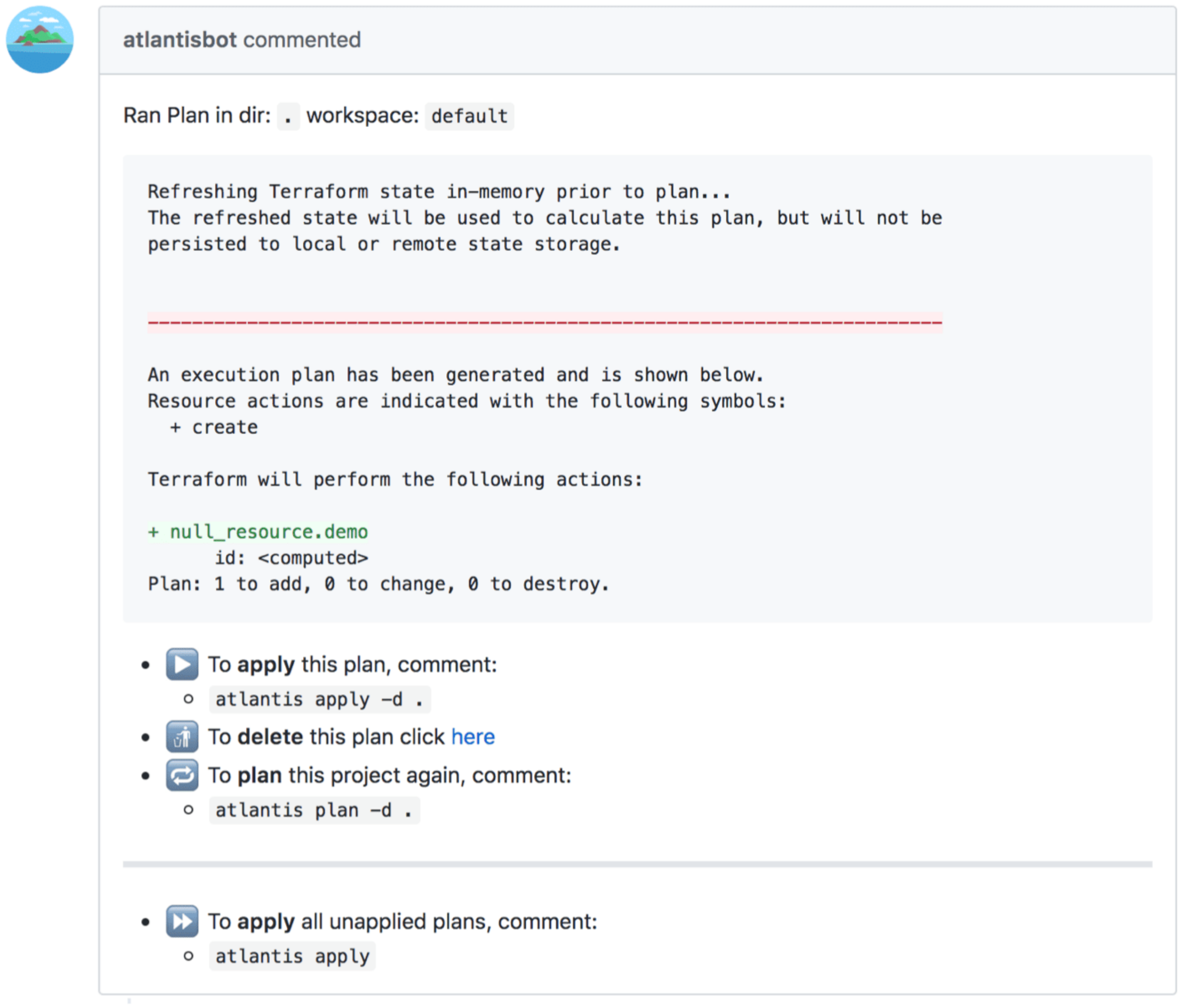
You could run this command as part of a commit hook to ensure that all code committed to version control automatically gets a consistent style.

Run automated tests

Just as with application code, your infrastructure code should have commit hooks that kick off automated tests in a CI server after every commit, and show the results of those tests in the pull request. You already saw how to write unit tests, integration tests, and end-to-end tests for your Terraform code in Chapter 7. There's one other critical type of test you should run: `terraform plan`. The rule here is simple:

> *Always run `plan` before `apply`.*

Terraform shows the `plan` output automatically when you run `apply`, so what this rule really means is that you should always pause and read the `plan` output! You'd be amazed at the type of errors you can catch by taking 30 seconds to scan the "diff" you get as an output. A great way to encourage this behavior is by integrating `plan` into your code review flow. For example, Atlantis is an open source tool that automatically runs `terraform plan` on commits and adds the `plan` output to pull requests as a comment, as shown in Figure 8-3.



atlantisbot commented

Ran Plan in dir: `.` workspace: `default`

```
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.


------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

+ null_resource.demo
      id: <computed>
Plan: 1 to add, 0 to change, 0 to destroy.
```

- ▶️ To **apply** this plan, comment:
  - `atlantis apply -d .`
- 🚮 To **delete** this plan click here
- 🔁 To **plan** this project again, comment:
  - `atlantis plan -d .`

- ⏩ To **apply** all unapplied plans, comment:
  - `atlantis apply`

Figure 8-3. Atlantis can automatically add the output of the the `terraform plan` command as a comment on your pull requests.

The `plan` command even allows you to store that diff output in a file:

```
$ terraform plan -out=example.plan
```

You can then run the `apply` command on this saved plan file to ensure that it applies *exactly* the changes you saw originally:

```
$ terraform apply example.plan
```

Note that, just like Terraform state, the saved plan files may contain secrets. For example, if you’re deploying a database with Terraform, the plan file may contain the database password. Since the plan files are not encrypted, if you want to store them for any length of time, you’ll have to provide your own encryption.

Merge and release

Once your team members have had a chance to review the code changes and `plan` output, and all the tests have passed, you can merge your changes into the `master` branch, and release the code. Similar to application code, you can use Git tags to create a versioned release:

```
$ git tag -a "v0.0.6" -m "Updated hello-world-example text"
$ git push --follow-tags
```

Whereas with application code, you often have a separate artifact to deploy, such as a Docker image or Virtual machine image, since Terraform natively supports downloading code from Git, the repository at a specific tag *is* the immutable, versioned artifact you will be deploying.

Deploy

Now that you have an immutable, versioned artifact, it's time to deploy it. Here are a few of the key considerations for deploying Terraform code:

1. Deployment tooling
2. Deployment strategies
3. Deployment server
4. Promote artifacts across environments

DEPLOYMENT TOOLING

When deploying Terraform code, Terraform itself is the main tool that you use. However, there are a few other tools that you may find useful:

Atlantis

The open source tool you saw earlier can not only add the `plan` output to your pull requests, but also allows you to trigger a `terraform apply` when you add a special comment to your pull request. While this provides a convenient web interface for Terraform deployments, be aware that it doesn't support versioning, which can make maintenance and debugging for larger projects more difficult.

Terraform Enterprise

HashiCorp's enterprise products provide a web UI you can use to run `terraform plan` and `terraform apply`, as well as manage variables, secrets, and access permissions.

Terragrunt

This is an open source wrapper for Terraform that fills in some gaps in Terraform. You'll see how to use it a bit later in this chapter to deploy versioned Terraform code across multiple environments with minimal copy/paste.

Scripts

As always, you can write scripts in a general purpose programming language such as Python or Ruby or Bash to customize how you use Terraform.

DEPLOYMENT STRATEGIES

Terraform in and of itself doesn't offer any deployment strategies. There's no built-in support for rolling deployments or blue-green deployments, and there's no way to feature toggle most Terraform changes (e.g., you can't feature toggle a database change; either you make the change or you don't). You're essentially limited to `terraform apply`, which executes whatever configuration you have in your code. Of course, in the code itself, you can sometimes implement your own deployment strategies, such as the zero-downtime rolling deployment in the `asg-rolling-deploy` module you built in previous chapters. However, since Terraform is a declarative language, your control over deployments is fairly limited.

Due to these limitations, it's critical to take into account what happens when a deployment goes wrong. With an application deployment, many types of errors are caught by the deployment strategy: e.g., if the app fails to pass health checks, the load balancer will never send it live traffic, so users won't be affected. Moreover, the rolling deployment or blue-green deployment strategy can automatically roll back to the previous version of the app in case of errors.

Terraform, on the other hand, *does not roll back automatically in case of errors*. In part, this is because with arbitrary infrastructure code, it's often not safe or possible to do so: e.g., if an app deployment failed, it's almost always safe to roll back to an older version of the app, but if the Terraform change you were deploying failed, and that change was to delete a database or terminate a server, you can't easily roll that back!

Moreover, as you saw in "Terraform Gotchas", errors are fairly common with Terraform. Therefore, your deployment strategy should assume errors are (relatively) normal and offer a first-class way to deal with them:

Retries

Certain types of Terraform errors are transient, and go away if you re-run `terraform apply`. The deployment tooling you use with Terraform should detect these known errors and automatically retry after a brief pause. Terragrunt has automatic retries on known errors as a built-in feature.

Terraform state errors

Occasionally, Terraform will fail to save state after running `terraform apply`. For example, if you lose Internet connectivity part way through an `apply`, not only will the `apply` fail, but Terraform won't be able to write the updated state file to your remote backend (e.g., to S3). In these cases, Terraform will save the state file on disk in a file called `errored.tfstate`. Make sure that your CI server does not delete these files (e.g., as part of cleaing up the workspace after a build)! If you can still access this file after a failed deployment, once Internet connectivity is restored, you can push this file to your remote backend (e.g., to S3) using the `state push` command so that the state information isn't lost:

```
$ terraform state push errored.tfstate
```

Errors releasing locks

Occasionally, Terraform will fail to release a lock. For example, if your CI server crashes in the middle of a `terraform apply`, the state will remain permanently locked. Anyone else who tries to run `apply` on the same module will get an error message saying the state is locked, and the ID of the lock. If you're absolutely sure this is an accidentally left-over lock, you can forcibly release it using the `force-unlock` command, passing it the ID of the lock from that error message:

```
$ terraform force-unlock <LOCK_ID>
```

DEPLOYMENT SERVER

Just as with your application code, all your infrastructure code changes should be applied from a CI server, and not from a developer's computer. You can run `terraform` from Jenkins, CircleCI, Terraform Enterprise, Atlantis, or any other reasonably secure automated platform. This gives you the same benefits as with application code: it forces you to fully automate your deployment process, it ensures deployment always happens from a consistent environment, and it gives you better control over who has permissions to access production environments.

That said, permissions to deploy infrastructure code are quite a bit trickier than for application code. With application code, you can usually give your CI server a minimal, fixed set of permissions to deploy your apps: e.g., to deploy to an Auto Scaling Group, the CI server typically only needs a few specific `ec2` and `autoscaling` permissions. However, to be able to deploy arbitrary infrastructure code changes (e.g., your Terraform code may try to deploy a database or a VPC or an entirely new AWS account), the CI server needs arbitrary permissions—that is, admin permissions.

Since CI servers are designed to execute arbitrary code, they are often difficult to secure (e.g., try joining the Jenkins security advisory list to see how often severe vulnerabilities are announced), so giving your CI server permanent admin credentials can be risky. There are a few things you can do to minimize this risk:

1. Don't expose your CI server on the public Internet. That is, run the CI server in private subnets, without any public IP, so it's only accessible over a VPN connection. This is significantly more secure, but it comes at a cost: webhooks from external systems won't work, so, for example, GitHub won't automatically be able to trigger builds in your CI server. Instead, you'll have to configure your CI server to poll your version control system for updates.

2. Lock the CI server down. Make it accessible solely over HTTPs, require all users to be authenticated, and follow server hardening practices (e.g., lock down the firewall, install fail2ban, enable audit logging, etc).

3. Consider not giving the CI server permanent admin credentials. Instead, give it credentials so it can deploy some types of infrastructure code changes automatically, but for anything more sensitive (e.g., for anything that can add or remove users or access secrets), require that a human administrator provide the server with temporary admin credentials (e.g, those that expire after 1 hour).

PROMOTE ARTIFACTS ACROSS ENVIRONMENTS

Just as with application artifacts, you'll want to promote your immutable, versioned infrastructure artifacts from environment to environment: e.g., promote `v0.0.6` from dev to stage to prod.[5] The rule here is also simple:

Always test Terraform changes in pre-prod before prod.

Since everything is automated with Terraform anyway, it doesn't cost you much extra effort to try a change in staging before production, but it will catch a huge number of errors. Testing in pre-prod is especially important because, as mentioned earlier in this chapter, Terraform does not roll back changes in case of errors. If you run `terraform apply` and something goes wrong, you have to fix it yourself. This is easier and less stressful to do if you catch the error in a pre-prod environment rather than prod.

The process for promoting Terraform code across environments is similar to the process of promoting application artifacts, except there is an extra step where you run `terraform plan` and manually review the output. This step isn't usually necessary for application deployments, as most application deployments are similar and relatively low risk. However, every infrastructure deployment can be completely different and mistakes can be very costly (e.g., deleting a database), so having one last chance to look at the `plan` output and review it is well worth the time.

Here's what the process looks like for promoting, say, `v0.0.6` of a Terraform module across the dev, stage, and prod environments:

1. Update the dev environment to `v0.0.6` and run `terraform plan`.

2. Prompt someone to review and approve the plan: e.g., send an automated message via Slack.

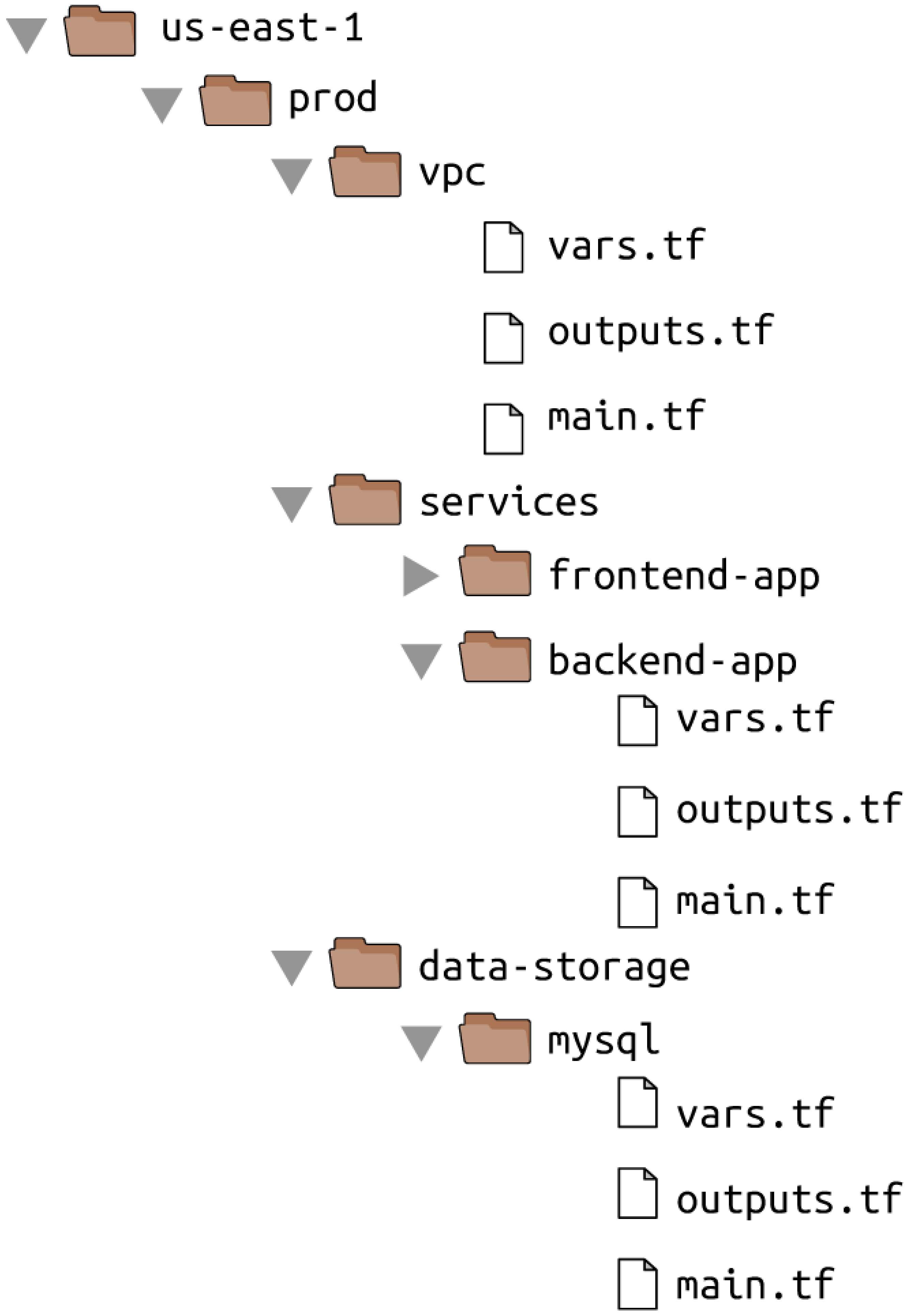
3. If the plan is approved, deploy `v0.0.6` to dev by running `terraform apply`.

4. Run your manual and automated tests in dev.

5. If `v0.0.6` works well in dev, repeat steps 1-4 to promote `v0.0.6` to staging.

6. If `v0.0.6` works well in staging, repeat steps 1-4 again to promote `v0.0.6`to production.

One important issue to deal with is all the code duplication between environments in the *live* repo. For example, consider the *live* repo shown in Figure 8-4.


us-east-1
prod
vpc
vars.tf
outputs.tf
main.tf
services
frontend-app
backend-app
vars.tf
outputs.tf
main.tf
data-storage
mysql
vars.tf
outputs.tf
main.tf

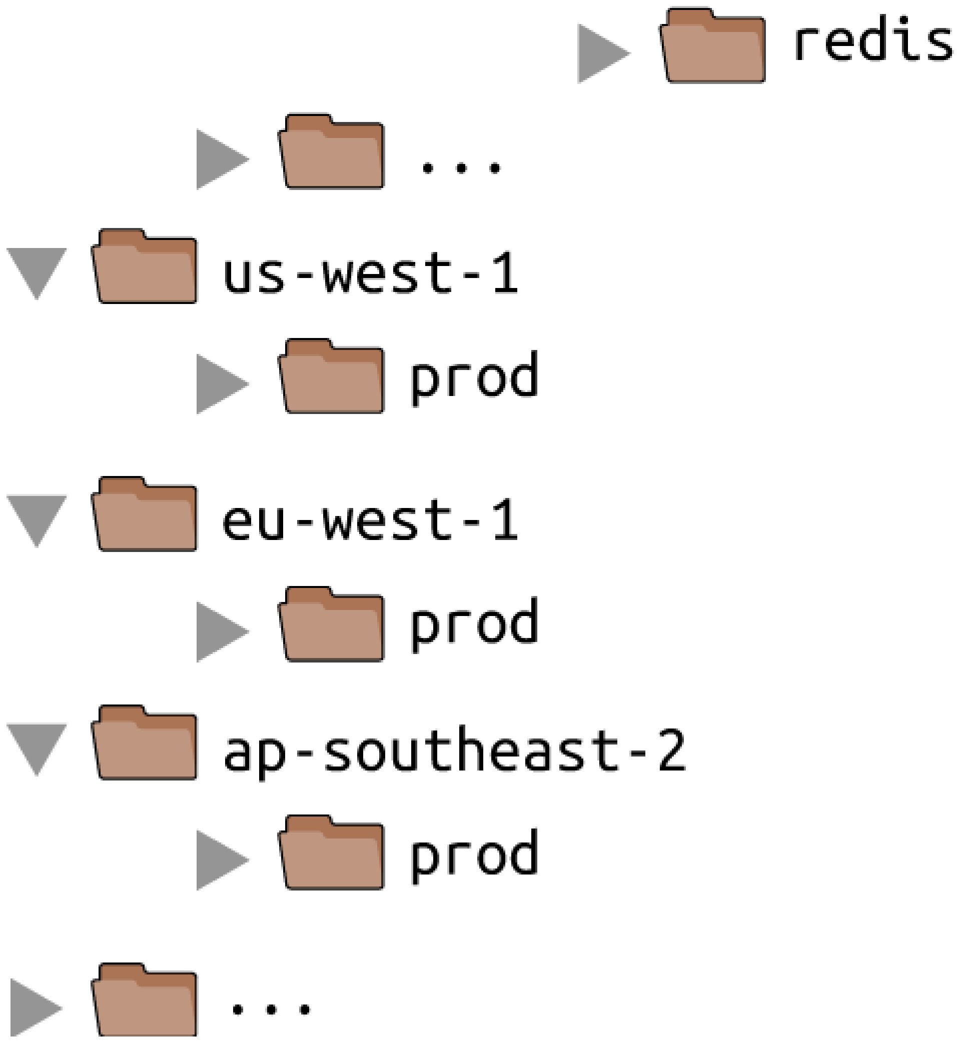


Figure 8-4. File layout with a large number of copy/pasted environments and modules within each environment

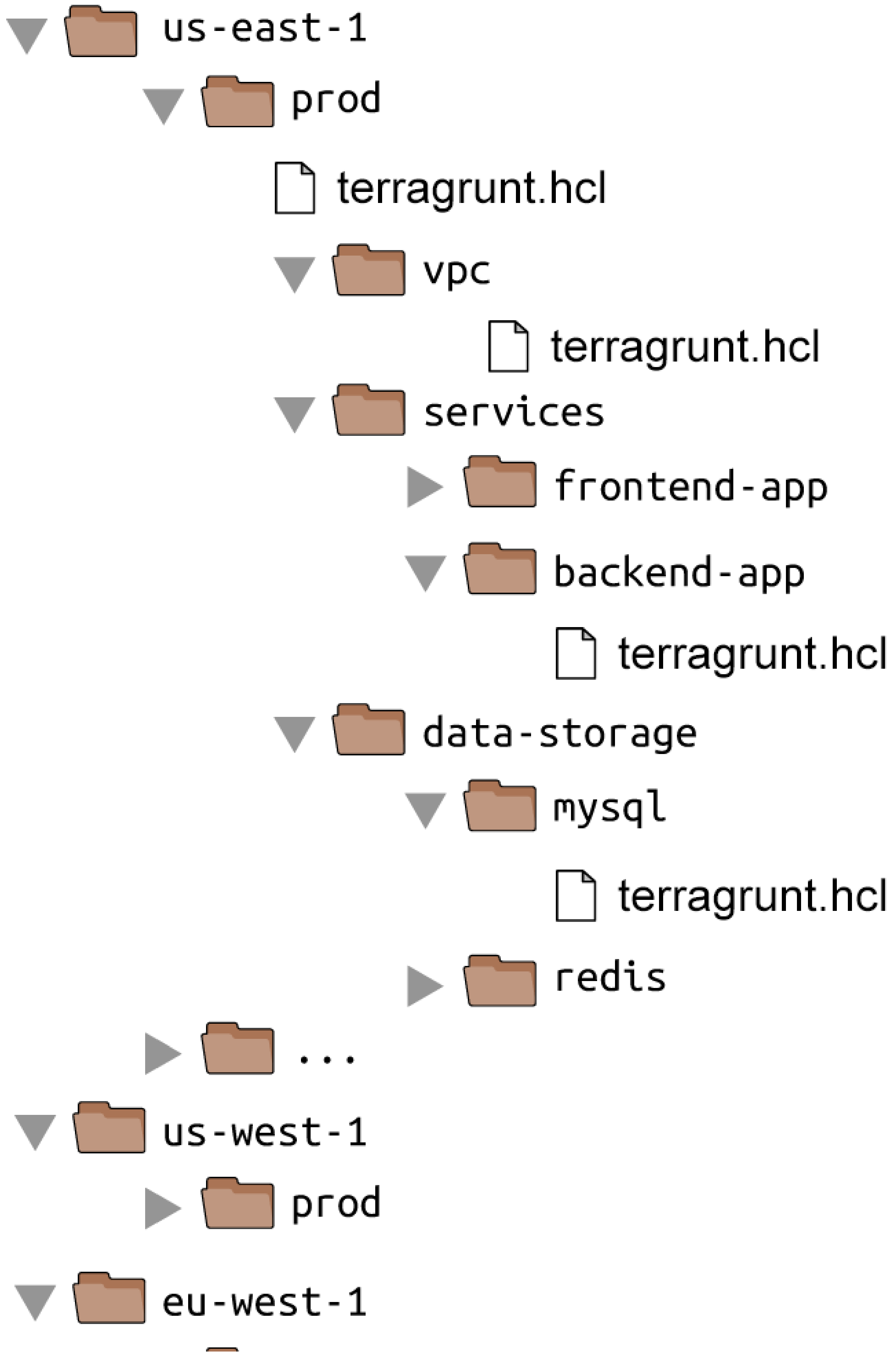
This *live* repo has a large number of regions, and within each region, a large number of modules, most of which are copy/pasted. Sure, each module has a `main.tf` that references a module in your `modules` repo, so it's not as much copy/paste as it could be, but even if all you're doing is instantiating a single module, there is still a large amount of boilerplate that needs to be duplicated between each environment:

1. The `provider` configuration.
2. The `backend` configuration.
3. Setting all the input variables for the module.
4. Outputting all the output variables from the module.

This can add up to dozens or hundreds of lines of mostly identical code in each module, copy/pasted into each environment. To make this code more DRY, and to make it easier to promote Terraform code across environments, you can use the open source tool I've mentioned earlier called Terragrunt. Terragrunt is a thin wrapper for Terraform, which means that once you install it (see the install instructions in the Terragrunt README), you can run all of your standard `terraform` commands, except you use `terragrunt` as the binary:

```
$ terragrunt plan
$ terragrunt apply
$ terragrunt output
```

Terragrunt will run Terraform with the command you specify, but based on configuration you specify in a `terragrunt.hcl` file, you can get some extra behavior. The basic idea is that you will define all of your Terraform code exactly once in the *modules* repo, while in the *live* repo, you will have solely `terragrunt.hcl` files that provide a DRY way to configure and deploy each module in each environment. This will result in a *live* repo with far fewer files and lines of code, as shown in Figure 8-5.


us-east-1
prod
terragrunt.hcl
vpc
terragrunt.hcl
services
frontend-app
backend-app
terragrunt.hcl
data-storage
mysql
terragrunt.hcl
redis
...
us-west-1
prod
eu-west-1

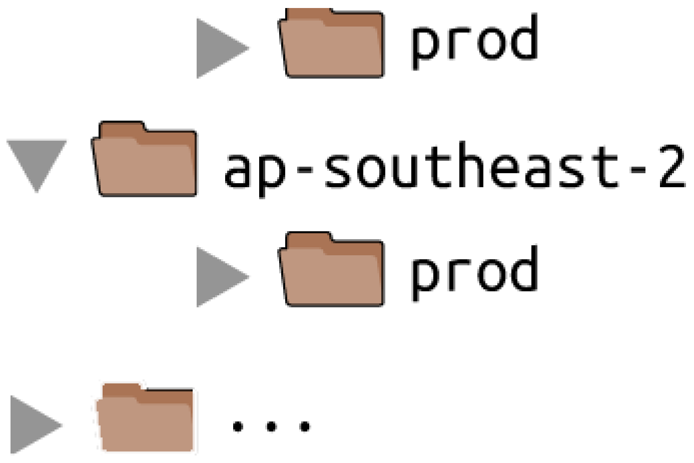


Figure 8-5. File layout with Terragrunt

To get started, add a `provider` configuration to *modules/data-stores/mysql/main.tf* and *modules/services/hello-world-app/main.tf*:

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

  # Allow any 2.x version of the AWS provider
  version = "~> 2.0"
}
```

Next, add a `backend` configuration *modules/data-stores/mysql/main.tf* and*modules/services/hello-world-app/main.tf*, but leave the `config` block empty (you'll see how to fill this in using Terragrunt, shortly):

```
terraform {
  # Require any 0.12.x version of Terraform
  required_version = ">= 0.12, < 0.13"

  # Partial configuration. The rest will be filled in by Terragrunt.
  backend "s3" {}
}
```

Commit these changes and release a new version of your `modules` repo:

```
$ git add modules/data-stores/mysql/main.tf
$ git add modules/services/hello-world-app/main.tf
$ git commit -m "Update mysql and hello-world-app for Terragrunt"
$ git tag -a "v0.0.7" -m "Update Hello, World text"
$ git push --follow-tags
```

Now, head over to the *live* repo, and delete all the `.tf` files. You're going to replace all that copy/pasted Terraform code with a single `terragrunt.hcl` file for each module. For example, here's `terragrunt.hcl` for *live/stage/data-stores/mysql/terragrunt.hcl*:

```
terraform {
  source = "github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7"
}

inputs = {
  db_name     = "example_stage"
  db_username = "admin"

  # Set the password using the TF_VAR_db_password environment variable
}
```

As you can see, `terragrunt.hcl` files use the same HCL syntax as Terraform itself. When you run `terragrunt apply`, and it finds the `source` parameter in `terragrunt.hcl` file, Terragrunt will do the following:

1. Check out the URL specified in `source` to a temporary folder. This supports the same URL syntax as the `source` parameter of Terraform modules, so you can use local file paths, Git URLs, versioned Git URLs (via the `ref` parameter, as in the example above), and so on.

2. Run `terraform apply` in the temporary folder, passing it the input variables that you've specified in the `inputs = { … }` block.

The benefit of this approach is that the code in the *live* repo is reduced to just a single `terragrunt.hcl` file per module, which only contains a pointer to the module to use (at a specific version), plus the input variables to set for that specific environment. That's about as DRY as you can get.

Terragrunt also helps you keep your `backend` configuration DRY. Instead of having to define the `bucket`, `key`, `dynamodb_table`, etc in every single module, you can define it in a single `terragrunt.hcl` file per environment. For example, create the following in *live/stage/terragrunt.hcl*:

```
remote_state {
  backend = "s3"
  config = {
    bucket         = "<YOUR BUCKET>"
    key            = "${path_relative_to_include()}/terraform.tfstate"
    region         = "us-east-2"
    encrypt        = true
    dynamodb_table = "<YOUR_TABLE>"
  }
}
```

In the `remote_state` block, you specify your `backend` configuration the same way as always, except for the `key` value, which uses a Terragrunt built-in function called *path_relative_to_include()*. This function will return the relative path between this root `terragrunt.hcl` file and any child module that includes it. For example, to include this root file in *live/stage/data-stores/mysql/terragrunt.hcl*, just add an `include` block:

```
terraform {
  source = "github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7"
}

include {
  path = find_in_parent_folders()
}

inputs = {
  db_name     = "example_stage"
  db_username = "admin"

  # Set the password using the TF_VAR_db_password environment variable
}
```

The `include` block finds the root `terragrunt.hcl` using the Terragrunt built-in function *find_in_parent_folders()*, automatically inheriting all the settings from that parent file, including the `remote_state` configuration. The result is that this `mysql` module will use all the same `backend` settings as the root file, and the `key`value will automatically resolve to *data-stores/mysql/terraform.tfstate*. This means that your Terraform state will be stored in the same folder structure as your *live* repo, which will make it easy to know which module produced which state files.

To deploy this module, run `terragrunt apply`:

```
$ terragrunt apply

[terragrunt] Reading Terragrunt config file at terragrunt.hcl

[terragrunt] Downloading Terraform configurations from
             github.com/<OWNER>/modules//data-stores/mysql?ref=v0.0.7

[terragrunt] Running command: terraform init -backend-config=(...)

(...)
```

```
[terragrunt] Running command: terraform apply

(...)

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

You can see in the log output that Terragrunt read your `terragrunt.hcl` file, downloaded the module you specified, ran `terraform init` to configure your `backend` (it'll even create your S3 bucket and DynamoDB table automatically if they don't exist already), and then ran `terraform apply` to deploy everything.

You can now deploy the `hello-world-app` module in staging by adding*live/stage/services/hello-world-app/terragrunt.hcl* and running `terragrunt apply`:

```
terraform {
  source = "github.com/<OWNER>/modules//services/hello-world-app?ref=v0.0.7"
}

include {
  path = find_in_parent_folders()
}

inputs = {
  environment = "stage"

  min_size = 2
  max_size = 2

  enable_autoscaling = false

  db_remote_state_bucket = "<YOUR_BUCKET>"
  db_remote_state_key    = "<YOUR_KEY>"
}
```

This module also uses `include` to pull in the settings from the root `terragrunt.hcl` file, so it will inherit the same `backend` settings, except for the `key`, which will be automatically set to *services/hello-world-app/terraform.tfstate*, just as you'd expect. Once everything is working correctly in staging, create analogous `terragrunt.hcl` files in *live/prod* and promote the exact same `v0.0.7` artifact to production by running `terragrunt apply` in each module.

Putting it all together

You've now seen how to take both application code and infrastructure code from development all the way through to production. Table 8-1 shows an overview of the two workflows side-by-side.

Table 8-1. Application and infrastructure code workflows

	Application code	**Infrastructure code**
Use version control	• `git clone` • One repo per app • Use branches	• `git clone` • *live* and *modules* repos • Don't use branches
Run the code locally	• Run on localhost • `ruby web-server.rb` • `ruby web-server-test.rb`	• Run in a sandbox environment • `terraform apply` • `go test`
Make code changes	• Change the code • `ruby web-server.rb` • `ruby web-server-test.rb`	• Change the code • `terraform apply` • `go test` • Use test stages
Submit changes for review	• Submit a pull request • Enforce coding guidelines	• Submit a pull request • Enforce coding guidelines
Run automated tests	• Tests run on CI server • Unit tests • Integration tests • End-to-end tests • Static analysis	• Tests run on CI server • Unit tests • Integration tests • End-to-end tests • Static analysis • `terraform plan`
Merge and release	• `git tag` • Create versioned, immutable artifact	• `git tag` • Use repo with tag as versioned, immutable artifact

	Application code	Infrastructure code
Deploy	• Deploy with Terraform, orchestration tool (e.g., Kubernetes, Mesos), scripts • Many deployment strategies: rolling deployment, blue-green, canary • Run deployment on a CI server • Give CI server limited permissions • Promote immutable, versioned artifacts across environments	• Deploy with Terraform, Atlantis, Terraform Enterprise, Terragrunt, scripts • Limited deployment strategies. Make sure to handle errors: retries, `errored.tfstate`! • Run deployment on a CI server • Give CI server admin permissions • Promote immutable, versioned artifacts across environments

If you follow this process, you will be able to run application and infrastructure code in dev, test it, review it, package it into versioned, immutable artifacts, and promote those artifacts from environment to environment, as shown in Figure 8-6.

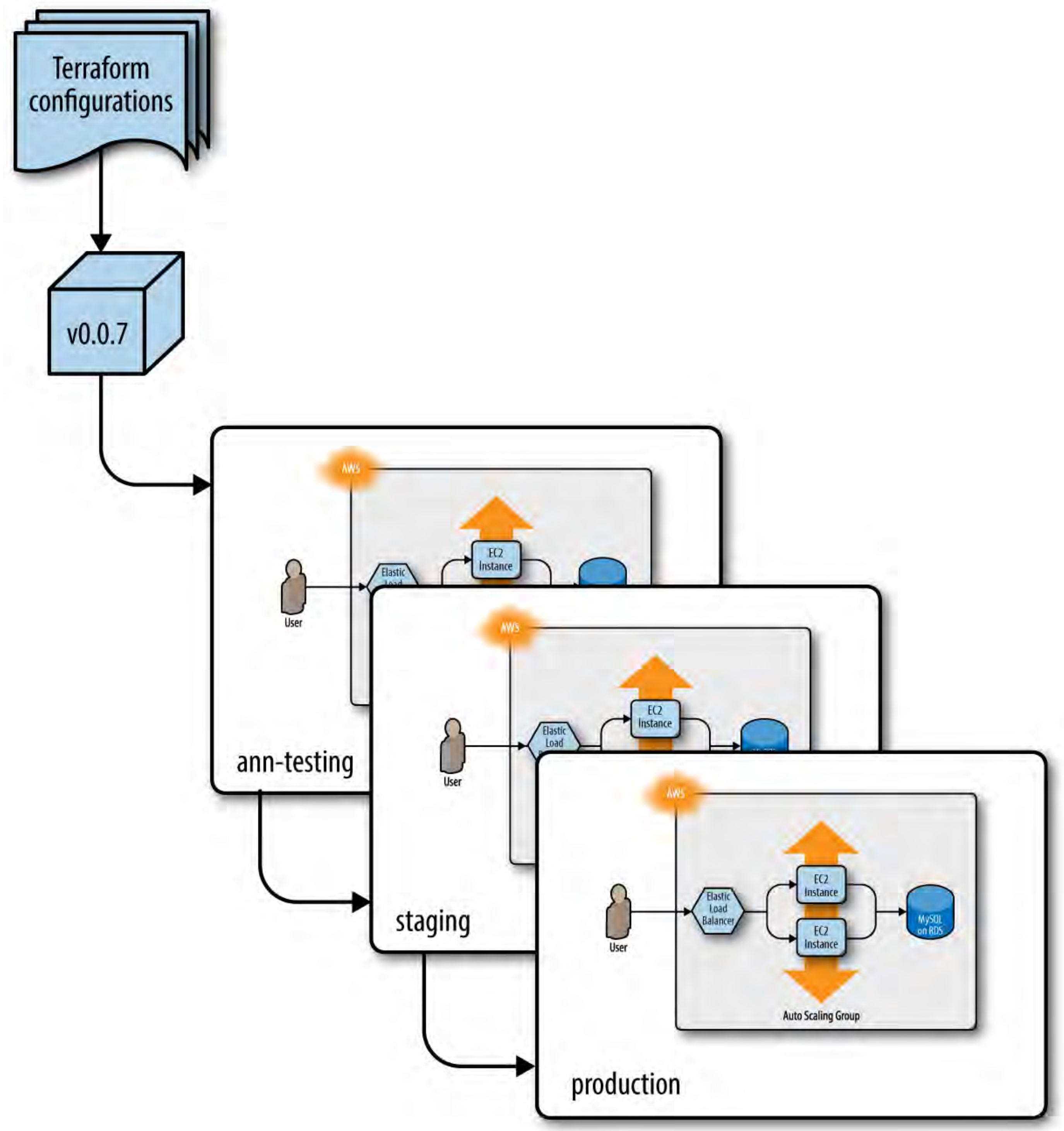


Figure 8-6. Promoting an immutable, versioned artifact of Terraform code from environment to environment

Conclusion

If you've made it to this point in the book, you now know just about everything you need to use Terraform in the real world, including how to write Terraform code, how to manage Terraform state, how to create reusable modules with Terraform, how to do loops, if-statements, and deployments, how to write production-grade Terraform code, how to test your Terraform code, and how to use Terraform as a team. You've worked through examples of deploying and managing servers, clusters of servers, load balancers, databases, auto scaling schedules, CloudWatch alarms, IAM users, reusable modules, zero-downtime deployment, automated tests, and more. Phew! Just don't forget to run `terraform destroy` in each module when you're all done.

The power of Terraform, and more generally, infrastructure as code, is that you can manage all the operational concerns around an application using the same coding principles as the application itself. This allows you to apply the full power of software engineering to your infrastructure, including modules, code reviews, version control, and automated testing.

If you use Terraform correctly, your team will be able to deploy faster and respond to changes more quickly. Hopefully, deployments will become routine and boring—and in the world of operations, boring is a very good thing. And if you really do your job right, rather than spending all your time managing infrastructure by hand, your team will be able to spend more and more time improving that infrastructure, allowing you to go even faster.

This is the end of the book, but just the beginning of your journey with Terraform. To learn more about Terraform, infrastructure as code, and DevOps, head over to Appendix A for a list of recommended reading. And if you've got feedback or questions, I'd love to hear from you at jim@ybrikman.com. Thank you for reading!

---

1 The Standish Group. "CHAOS Manifesto 2013: Think Big, Act Small." 2013. *http://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf*

2 Milstein, Dan. "How To Survive a Ground-Up Rewrite Without Losing Your Sanity." OnStartups.com, April 8, 2013. *http://onstartups.com/tabid/3339/bid/97052/How-To-Survive-a-Ground-Up-Rewrite-Without-Losing-Your-Sanity.aspx*.

3 The Gruntwork Infrastructure as Code Library is a collection of over 300,000 lines of production-grade, commercially-supported, reusable infrastructure code that has been proven in production with hundreds of companies.

4 Writing the Readme first is called Readme Driven Development, as described here: *http://bit.ly/1p8QBor*.

5 Credit for how to promote Terraform code across environments goes to Kief Morris: Using Pipelines to Manage Environments with Infrastructure as Code

Appendix A. Recommended Reading

The following are some of the best resources I've found on DevOps and infrastructure as code, including books, blog posts, newsletters, and talks.

Books

- *Infrastructure as Code: Managing Servers in the Cloud* by Kief Morris (O'Reilly)
- *Site Reliability Engineering: How Google Runs Production Systems* by Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy (O'Reilly)
- *The DevOps Handbook: How To Create World-Class Agility, Reliability, & Security in Technology Organizations* by Gene Kim, Jez Humble, Patrick Debois, and John Willis (IT Revolution Press)
- *Designing Data Intensive Applications* by Martin Kleppmann (O'Reilly)
- *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* by Jez Humble and David Farley (Addison-Wesley Professional)
- *Release It! Design and Deploy Production-Ready Software* by Michael T. Nygard (The Pragmatic Bookshelf)
- *Kubernetes In Action* by Marko Luksa (Manning)
- *Leading the Transformation: Applying Agile and DevOps Principles at Scale* by Gary Gruver and Tommy Mouser (IT Revolution Press)
- *Visible Ops Handbook* by by Kevin Behr, Gene Kim, and George Spafford (Information Technology Process Institute)
- *Effective DevOps* by Jennifer Davis and Katherine Daniels (O'Reilly)
- *Lean Enterprise* by Jez Humble, Joanne Molesky, Barry O'Reilly (O'Reilly)
- *Hello, Startup: A Programmer's Guide to Building Products, Technologies, and Teams* by Yevgeniy Brikman (O'Reilly)

Blogs

- High Scalability
- Code as Craft
- dev2ops
- AWS blog
- Kitchen Soap
- Paul Hammant's blog
- Martin Fowler's blog
- Gruntwork blog
- Yevgeniy Brikman blog

Talks

- Reusable, composable, battle-tested Terraform modules by Yevgeniy Brikman
- 5 Lessons Learned From Writing Over 300,000 Lines of Infrastructure Code by Yevgeniy Brikman
- "Infrastructure as code: running microservices on AWS using Docker, Terraform, and ECS" by Yevgeniy Brikman
- "Agility Requires Safety" by Yevgeniy Brikman
- "Adopting Continuous Delivery" by Jez Humble

- “Continuously Deploying Culture” by Michael Rembetsy and Patrick McDonnell
- “10+ Deploys Per Day: Dev and Ops Cooperation at Flickr” by John Allspaw and Paul Hammond
- “Why Google Stores Billions of Lines of Code in a Single Repository” by Rachel Potvin
- “The Language of the System” by Rich Hickey
- “Don’t Build a Distributed Monolith” by Ben Christensen
- “Real Software Engineering” by Glenn Vanderburg

Newsletters

- DevOps Weekly
- DevOpsLinks
- Gruntwork Newsletter
- Terraform: Up & Running Newsletter

Online Forums

- Terraform Google Group
- DevOps subreddit