O'REILLY®

Third Edition

Terraform

Up & Running

Writing Infrastructure as Code

Yevgeniy Brikman

O'REILLY®

Terraform: Up and Running

Terraform has become a key player in the DevOps world for defining, launching, and managing infrastructure as code (IaC) across a variety of cloud and virtualization platforms, including AWS, Google Cloud, Azure, and more. This hands-on third edition, expanded and thoroughly updated for version 1.0 and beyond, shows you the fastest way to get up and running with Terraform.

Gruntwork cofounder Yevgeniy (Jim) Brikman takes you through code examples that demonstrate Terraform's simple, declarative programming language for deploying and managing infrastructure with a few commands. Veteran sysadmins, DevOps engineers, and novice developers will quickly go from Terraform basics to running a full stack that can support a massive amount of traffic and a large team of developers.

- **Compare Terraform with Chef, Puppet, Ansible, CloudFormation, and Pulumi**
- **Deploy servers, load balancers, and databases**
- **Create reusable infrastructure with Terraform modules**
- **Test your Terraform modules with static analysis, unit tests, and integration tests**
- **Configure CI/CD pipelines for both your apps and infrastructure code**
- **Use advanced Terraform syntax for loops, conditionals, and zero-downtime deployment**
- **Get up to speed on Terraform 0.13 to 1.0 and beyond**
- **Work with multiple clouds and providers (including Kubernetes!)**

"This book teaches you everything you need to know about Terraform to massively improve infrastructure provisioning efficiency and enjoyability across any platform."

—**Mitchell Hashimoto**
Creator of Terraform and Cofounder of HashiCorp

"If you are a DevOps practitioner and want to get started with infrastructure as code, this book is the perfect resource for you."

—**Akash Mahajan**
Cofounder and Director, Appsecco

Yevgeniy (Jim) Brikman is the cofounder of Gruntwork, a company with the mission of making it 10 times easier to build software. The author of *Hello, Startup: A Programmer's Guide to Building Products, Technologies, and Teams* (O'Reilly), he previously served as a software engineer at LinkedIn, TripAdvisor, Cisco, and Thomson Financial. For more information, check out ybrikman.com.

INFRASTRUCTURE AS CODE

US $59.99 CAN $74.99
ISBN: 978-1-098-11674-3

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

THIRD EDITION

Terraform: Up & Running

Writing Infrastructure as Code

Yevgeniy Brikman

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Terraform: Up & Running
by Yevgeniy Brikman

Copyright © 2022 Yevgeniy Brikman. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Developmental Editor: Corbin Collins
Production Editor: Kate Galloway
Copyeditor: Piper Editorial Consulting, LLC
Proofreader: Kim Cofer
Indexer: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

March 2017: First Edition
September 2019: Second Edition
September 2022: Third Edition

Revision History for the Third Edition
2022-09-19: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098116743* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Terraform: Up & Running*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-11674-3

[LSI]

To Mom, Dad, Lyalya, and Molly

Table of Contents

Preface ix

1. Why Terraform 1
- What Is DevOps? 1
- What Is Infrastructure as Code? 3
  - Ad Hoc Scripts 4
  - Configuration Management Tools 5
  - Server Templating Tools 7
  - Orchestration Tools 12
  - Provisioning Tools 14
- What Are the Benefits of Infrastructure as Code? 16
- How Does Terraform Work? 17
- How Does Terraform Compare to Other IaC Tools? 20
  - Configuration Management Versus Provisioning 21
  - Mutable Infrastructure Versus Immutable Infrastructure 21
  - Procedural Language Versus Declarative Language 22
  - General-Purpose Language Versus Domain-Specific Language 25
  - Master Versus Masterless 26
  - Agent Versus Agentless 28
  - Paid Versus Free Offering 30
  - Large Community Versus Small Community 31
  - Mature Versus Cutting Edge 33
  - Use of Multiple Tools Together 34
- Conclusion 36

2. Getting Started with Terraform 39
- Setting Up Your AWS Account 40
- Installing Terraform 43

Deploying a Single Server 44
Deploying a Single Web Server 52
Deploying a Configurable Web Server 60
Deploying a Cluster of Web Servers 66
Deploying a Load Balancer 70
Cleanup 79
Conclusion 80

3. How to Manage Terraform State. 81
What Is Terraform State? 82
Shared Storage for State Files 83
Limitations with Terraform's Backends 91
State File Isolation 93
Isolation via Workspaces 94
Isolation via File Layout 100
The terraform_remote_state Data Source 105
Conclusion 113

4. How to Create Reusable Infrastructure with Terraform Modules. 115
Module Basics 118
Module Inputs 121
Module Locals 125
Module Outputs 127
Module Gotchas 129
File Paths 129
Inline Blocks 130
Module Versioning 133
Conclusion 139

5. Terraform Tips and Tricks: Loops, If-Statements, Deployment, and Gotchas. 141
Loops 142
Loops with the count Parameter 142
Loops with for_each Expressions 149
Loops with for Expressions 156
Loops with the for String Directive 158
Conditionals 160
Conditionals with the count Parameter 160
Conditionals with for_each and for Expressions 165
Conditionals with the if String Directive 167
Zero-Downtime Deployment 169
Terraform Gotchas 179
count and for_each Have Limitations 179

Zero-Downtime Deployment Has Limitations 181
Valid Plans Can Fail 184
Refactoring Can Be Tricky 186
Conclusion 189

6. Managing Secrets with Terraform. 191
Secret Management Basics 192
Secret Management Tools 193
The Types of Secrets You Store 193
The Way You Store Secrets 194
The Interface You Use to Access Secrets 195
A Comparison of Secret Management Tools 195
Secret Management Tools with Terraform 196
Providers 196
Resources and Data Sources 206
State Files and Plan Files 217
Conclusion 219

7. Working with Multiple Providers. 221
Working with One Provider 221
What Is a Provider? 222
How Do You Install Providers? 223
How Do You Use Providers? 225
Working with Multiple Copies of the Same Provider 226
Working with Multiple AWS Regions 226
Working with Multiple AWS Accounts 238
Creating Modules That Can Work with Multiple Providers 245
Working with Multiple Different Providers 248
A Crash Course on Docker 249
A Crash Course on Kubernetes 252
Deploying Docker Containers in AWS Using Elastic Kubernetes Service 264
Conclusion 272

8. Production-Grade Terraform Code. 275
Why It Takes So Long to Build Production-Grade Infrastructure 277
The Production-Grade Infrastructure Checklist 279
Production-Grade Infrastructure Modules 280
Small Modules 281
Composable Modules 285
Testable Modules 291
Versioned Modules 298
Beyond Terraform Modules 305

Conclusion 313

9. How to Test Terraform Code. 315
Manual Tests 316
Manual Testing Basics 317
Cleaning Up After Tests 319
Automated Tests 320
Unit Tests 321
Integration Tests 348
End-to-End Tests 362
Other Testing Approaches 364
Conclusion 372

10. How to Use Terraform as a Team. 375
Adopting IaC in Your Team 376
Convince Your Boss 376
Work Incrementally 379
Give Your Team the Time to Learn 380
A Workflow for Deploying Application Code 382
Use Version Control 382
Run the Code Locally 383
Make Code Changes 383
Submit Changes for Review 384
Run Automated Tests 385
Merge and Release 386
Deploy 387
A Workflow for Deploying Infrastructure Code 390
Use Version Control 391
Run the Code Locally 395
Make Code Changes 396
Submit Changes for Review 397
Run Automated Tests 399
Merge and Release 400
Deploy 401
Putting It All Together 412
Conclusion 414

Appendix. Recommended Reading. 417

Index. 421

Preface

A long time ago, in a datacenter 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 [AWS], Microsoft Azure, Google Cloud Platform, 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 AWS:

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

resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  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 (virtual servers, Kubernetes clusters, Docker containers, load balancers, databases) 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 (IaC) 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., 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 that 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 IaC at all?
- What are the differences between configuration management, orchestration, provisioning, and server templating?
- When should you use Terraform, Chef, Ansible, Puppet, Pulumi, 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 securely manage secrets when working with Terraform?
- How do you use Terraform with multiple regions, accounts, and clouds?
- 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 has become wildly popular, but it's still a relatively new technology, and despite its popularity, it's still difficult to find books, blog posts, or experts to help you become proficient with 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 but 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 six out of the seven 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 more than 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 that 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 need to spend time conversing with native French speakers, watching French TV shows, and listening to French music. To become fluent in Terraform, you need 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, Pulumi, 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 Terraform state is; how to store state so that multiple team members can access it; how to lock state files to prevent race conditions; 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 modules are; 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 the `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` and

`for_each` limitations, zero-downtime deployment gotchas, how valid plans can fail, and how to refactor Terraform code safely.

Chapter 6, "Managing Secrets with Terraform"
An introduction to secrets management; an overview of the different types of secrets, different ways to store secrets, and different ways to access secrets; a comparison of common secret management tools such as HashiCorp Vault, AWS Secrets Manager, and Azure Key Vault; how to manage secrets when working with providers, including authentication via environment variables, IAM roles, and OIDC; how to manage secrets when working with resources and data sources, including how to use environment variables, encrypted files, and centralized secret stores; how to securely handle state files and plan files.

Chapter 7, "Working with Multiple Providers"
A closer look at how Terraform providers work, including how to install them, how to control the version, and how to use them in your code; how to use multiple copies of the same provider, including how to deploy to multiple AWS regions, how to deploy to multiple AWS accounts, and how to build reusable modules that can use multiple providers; how to use multiple different providers together, including an example of using Terraform to run a Kubernetes cluster (EKS) in AWS and deploy Dockerized apps into the cluster.

Chapter 8, "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; variable validation; versioning Terraform, Terraform providers, Terraform modules, and Terragrunt; Terraform escape hatches.

Chapter 9, "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; plan testing; server testing.

Chapter 10, "How to Use Terraform as a Team"
How to adopt Terraform as a 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; the deployment process.

Feel free to read the book from beginning to end 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" on page xix)

to get your bearings. At the end of the book, in the Appendix, you'll find a list of recommended reading where you can learn more about Terraform, operations, IaC, and DevOps.

Changes from the Second Edition to the Third Edition

The first edition of this book came out in 2017, the second edition came out in 2019, and although it's hard for me to believe it, I'm now working on the third edition in 2022. Time flies. It's remarkable how much has changed over the years!

If you've read the second edition of the book and want to know what's new, or if you're just curious to see how Terraform has evolved between 2019 and 2022, here are some of the highlights of what changed between the second and third editions:

Hundreds of pages of updated content

The third edition of the book is about a hundred pages longer than the second edition. I also estimate that roughly one-third to one-half of the pages originally in the second edition were updated as well. Why so much churn? Well, Terraform went through six major releases since the second edition came out: 0.13, 0.14, 0.15, 1.0, 1.1, and 1.2. Moreover, many Terraform providers went through major upgrades of their own, including the AWS Provider, which was at version 2 when the second edition came out and is now at version 4. Plus, the Terraform community has seen massive growth over the last few years, which has led to the emergence of many new best practices, tools, and modules. I've tried to capture as much of this change as I could in the third edition, adding two completely new chapters and making major updates to all the existing chapters, as described next.

New provider functionality

Terraform has significantly improved how you work with providers. In the third edition, I've added an entirely new chapter, Chapter 7, that describes how to work with multiple providers: e.g., how to deploy into multiple regions, multiple accounts, and multiple clouds. Also, by popular demand, this chapter includes a brand-new set of examples showing how to use Terraform, Kubernetes, Docker, AWS, and EKS to run containerized apps. Finally, I've also updated all the other chapters to highlight new provider features from the last several releases, including the `required_providers` block introduced in Terraform 0.13, the lock file introduced in Terraform 0.14, and the `configuration_aliases` parameter introduced in Terraform 0.15.

Better secrets management

When using Terraform code, you often have to deal with many types of secrets: database passwords, API keys, cloud provider credentials, TLS certificates, and so on. In the third edition, I added an entirely new chapter, Chapter 6, dedicated to this topic, including a comparison of common secret management tools, as

well as lots of new example code that shows a variety of techniques for securely using secrets with Terraform, including environment variables, encrypted files, centralized secret stores, IAM roles, OIDC, and more.

New module functionality

Terraform 0.13 added the ability to use `count`, `for_each`, and `depends_on` on `module` blocks, making modules considerably more powerful, flexible, and reusable. You can find examples of how to use these new features in Chapters 5 and 7.

New validation functionality

In Chapter 8, I've added examples of how to use the `validation` feature introduced in Terraform 0.13 to perform basic checks on variables (such as enforcing minimum or maximum values) and the `precondition` and `postcondition` features introduced in Terraform 1.2 to perform basic checks on resources and data sources, either before running `apply` (such as enforcing that the AMI a user selected uses the x86_64 architecture) or after running `apply` (such as checking that the EBS volume you're using was successfully encrypted). In Chapter 6, I show how to use the `sensitive` parameter introduced in Terraform 0.14 and 0.15, which ensures that secrets won't be logged when you run `plan` or `apply`.

New refactoring functionality

Terraform 1.1 introduced the `moved` block, which provides a much better way to handle certain types of refactoring, such as renaming a resource. In the past, this type of refactoring required users to manually run error-prone `terraform state mv` operations, whereas now, as you'll see in a new example in Chapter 5, this process can be fully automated, making upgrades safer and more compatible.

More testing options

The tools available for automated testing of Terraform code continue to improve. In Chapter 9, I've added example code and comparisons of static analysis tools for Terraform, including `tfsec`, `tflint`, `terrascan`, and the `validate` command; `plan` testing tools for Terraform, including Terratest, OPA, and Sentinel; and server testing tools, including `inspec`, `serverspec`, and `goss`. I also added a comparison of all the testing approaches out there, so you can pick the best ones for your use cases.

Improved stability

Terraform 1.0 was a big milestone for Terraform, not only signifying that the tool had reached a certain level of maturity but also coming with a number of compatibility promises. Namely, there is a promise that all the 1.x releases will be backward compatible, so upgrading between v1.x releases should no longer require changes to your code, workflows, or state files. Terraform state files are now cross-compatible with Terraform 0.14, 0.15, and all 1.x releases, and Terraform remote state data sources are cross-compatible with Terraform

0.12.30, 0.13.6, 0.14.0, 0.15.0, and all 1.x releases. I've also updated Chapter 8 with examples of how to better manage versioning of Terraform (including using `tfenv`), Terragrunt (including using `tgswitch`), and Terraform providers (including how to use the lock file).

Improved maturity
: Terraform has been downloaded over 100 million times, has had over 1,500 open source contributors, and is in use at ~79% of Fortune 500 companies,[1] so it's safe to say that the ecosystem has grown and matured significantly over the last several years. There are now more developers, providers, reusable modules, tools, plugins, classes, books, and tutorials for Terraform than ever before. Moreover, HashiCorp, the company that created Terraform, had its IPO (initial public offering) in 2021, so Terraform is no longer backed by a small startup but by a large, stable, publicly traded company, for which Terraform is its biggest business line.

Many other changes
: There were many other changes along the way, including the launch of Terraform Cloud (a web UI for using Terraform); the improved maturity of popular community tools such as Terragrunt, Terratest, and tfenv; the addition of many new provider features (including new ways to do zero-downtime deployment, such as instance refresh, which I've added to Chapter 5) and new functions (e.g., I added examples of how to use the `one` function in Chapter 5 and the `try` function in Chapter 7); the deprecation of many old features (e.g., `template_file` data source, many `aws_s3_bucket` parameters, `list` and `map`, support for external references on `destroy` provisioners); and much more.

Changes from the First Edition to the Second Edition

Going back in time even further, the second edition of the book added roughly 150 pages of new content on top of the first edition. Here is a summary of those changes, which also covers how Terraform changed between 2017 and 2019:

Four major Terraform releases
: Terraform was at version 0.8 when the first edition came out; between then and the time of the second edition, Terraform had four major releases, all the way up to version 0.12. These releases introduced some amazing new functionality, as I'll describe shortly, as well as a fair amount of upgrade work for users![2]

1 Per the HashiCorp S1.

2 Check out the Terraform upgrade guides for details.

Automated testing improvements

The tooling and practices for writing automated tests for Terraform code evolved considerably between 2017 and 2019. In the second edition, I added Chapter 9, 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 also evolved considerably. In the second edition, I added Chapter 8, a new chapter that contains a guide to building reusable, battle-tested, production-grade Terraform modules—the kind of modules you'd bet your company on.

Workflow improvements

Chapter 10 was completely rewritten in the second edition to reflect the changes in how teams integrate Terraform into their workflows, including a detailed guide on how to take application code and infrastructure code from development through testing and all the way to production.

HCL2

Terraform 0.12 overhauled the underlying language from HCL to HCL2. This included support for first-class expressions, 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 the second edition of the book were updated to use HCL2, and the new language features were covered extensively in Chapters 5 and 8.

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.

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 repositories, at their own cadence, with their own versioning. However, you now must run `terraform init` to download the provider code every time you start working with a new module, as discussed in Chapters 2 and 9.

Massive provider growth

From 2016 to 2019, Terraform grew from a handful of major cloud providers (the usual suspects, such as AWS, GCP, and Azure) to more than one hundred official providers and many more community providers.[3] This means that 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 with the GitHub, GitLab, and Bitbucket providers; data stores with the MySQL, PostgreSQL, and InfluxDB providers; monitoring and alerting systems with the Datadog, New Relic, and Grafana providers; platform tools with the Kubernetes, Helm, Heroku, Rundeck, and RightScale providers; and much more. Moreover, each provider has much better coverage these days: AWS now covers the majority of important AWS services and often adds support for new services even before CloudFormation does!

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. We look at the Registry in Chapter 8.

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 0.12 completely overhauled error handling, by catching errors earlier, showing clearer error messages, and including the filepath, line number, and a code snippet in the error message.

Many other changes

There were many other changes along the way, including the introduction of local values (see "Module Locals" on page 125), new "escape hatches" for having Terraform interact with the outside world via scripts (see "Beyond Terraform Modules" on page 305), running `plan` as part of the `apply` command, fixes for the `create_before_destroy` cycle issues, major improvements to the `count` parameter so that it can include references to data sources and resources, dozens of new built-in functions, an overhaul in `provider` inheritance, and much more.

3 You can find the full list of Terraform providers in the Terraform Registry.

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 of 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 might 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, AWS, so that you need to sign up only for a single third-party service (also, AWS offers a generous free tier, so running the example code shouldn't cost you much). This is why the book and the example code do not cover or require HashiCorp's paid services, Terraform Cloud or Terraform Enterprise. And this is why I've released all of the code examples as open source.

Open Source Code Examples

You can find all of the code samples in the book at the following URL:

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

You might want to check out this repo before you begin 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 in the *code* folder, and they are organized first by the tool or language (e.g., Terraform, Packer, OPA) and then by chapter. The one exception is the Go code used for automated tests in Chapter 9, which lives in the *terraform* folder to follow the *examples*, *modules*, and *test* folder layout recommended in that chapter. Table P-1 shows a few examples of where to find different types of code examples in the code samples repo.

Table P-1. Where to find different types of code examples in the code samples repo

Type of code	Chapter	Folder to look at in the samples repo
Terraform	Chapter 2	*code/terraform/02-intro-to-terraform-syntax*
Terraform	Chapter 5	*code/terraform/05-tips-and-tricks*
Packer	Chapter 1	*code/packer/01-why-terraform*
OPA	Chapter 9	*code/opa/09-testing-terraform-code*
Go	Chapter 9	*code/terraform/09-testing-terraform-code/test*

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, and checking the "official" solutions only at the very end.

You'll begin writing code 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 develop and improve this web server cluster example. Make the changes as instructed, try to write all the code yourself, and use the sample code in the GitHub repo only 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 1.x and AWS Provider 4.x, which were the most recent major releases as of this writing. Because Terraform is a relatively new tool, it is possible 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 need 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

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us 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 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.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Terraform: Up and Running*, Third Edition by Yevgeniy Brikman (O'Reilly). Copyright 2022 Yevgeniy Brikman, 978-1-098-11674-3."

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.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Online Learning

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

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

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 *https://oreil.ly/terraform-UR3*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

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

Watch us on YouTube: *https://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 cofounder 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, Virginia Wilson, and Corbin Collins for all your help on the first, second, and third editions, respectively.

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 third 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 10 times 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.

Reviewers
Thank you to Kief Morris, Seth Vargo, Mattias Gees, Ricardo Ferreira, Akash Mahajan, Moritz Heiber, Taylor Dolezal, and Anton Babenko 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 first and second editions
Those of you who bought the first and second editions of this book made the third edition possible. Thank you. Your feedback, questions, pull requests, and constant prodding for updates motivated a whole bunch of new and updated content. I hope you find the updates 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.

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 of 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, you'll dive into the following topics:

- What is DevOps?
- What is infrastructure as code?
- What are the benefits of infrastructure as code?
- How does Terraform work?
- How does Terraform compare to other infrastructure-as-code tools?

What Is DevOps?

In the not-so-distant past, if you wanted to build a software company, you also needed 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 Developers ("Devs"), dedicated to writing the software, and a separate team, typically called Operations ("Ops"), dedicated to managing this hardware.

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: because 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*, wherein 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 begin trying to merge all of their projects together, leading to a huge mess of merge conflicts. No one can stabilize the release branch. Teams begin blaming one another. Silos form. The company grinds to a halt.

Nowadays, a profound shift is taking place. Instead of managing their own datacenters, many companies are moving to the cloud, taking advantage of services such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Instead of investing heavily in hardware, many Ops teams are spending all their time working on software, using tools such as Chef, Puppet, Terraform, Docker, and Kubernetes. 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 might 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 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 in which 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, logfiles, 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

1 From *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations* (IT Revolution Press, 2016) by Gene Kim, Jez Humble, Patrick Debois, and John Willis.

- 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.

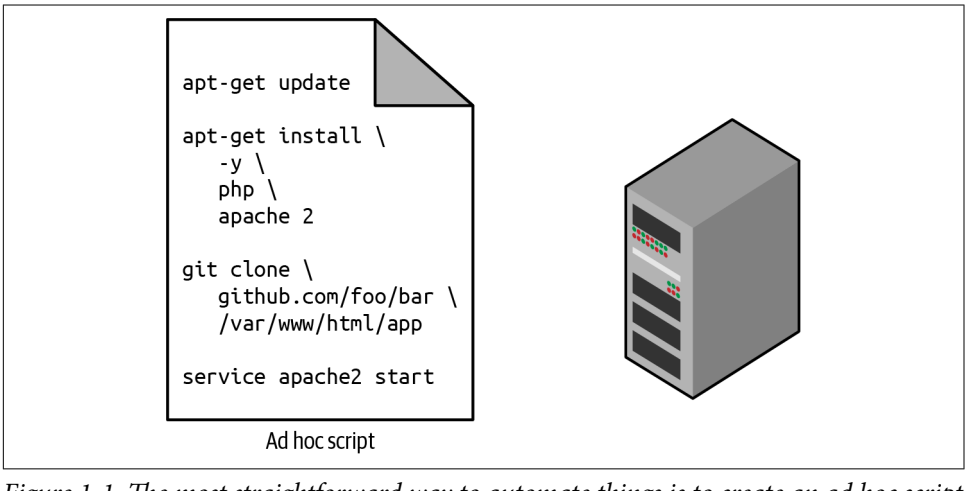


Figure 1-1. The most straightforward way to automate things is to create an ad hoc script that you run on your servers.

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 need 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 their 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, and Ansible are all *configuration management tools*, which means that 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 their 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 much harder. Every time you go to create a folder in your script, you need to remember to check whether 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 need 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 install Apache only if it isn't installed already and will try to start the Apache web server only 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.

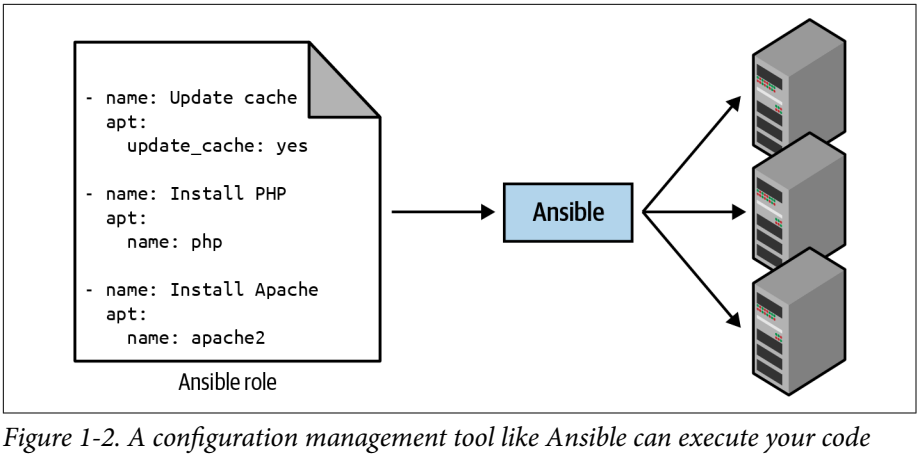


Figure 1-2. A configuration management tool like Ansible can execute your code across a large number of servers.

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 instructs Ansible to configure all five servers in parallel. Alternatively, by setting a parameter called `serial` in the playbook, you can do a *rolling deployment*, which updates the servers in batches. For example, setting `serial` to `2` directs 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 would 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 (OS), 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.

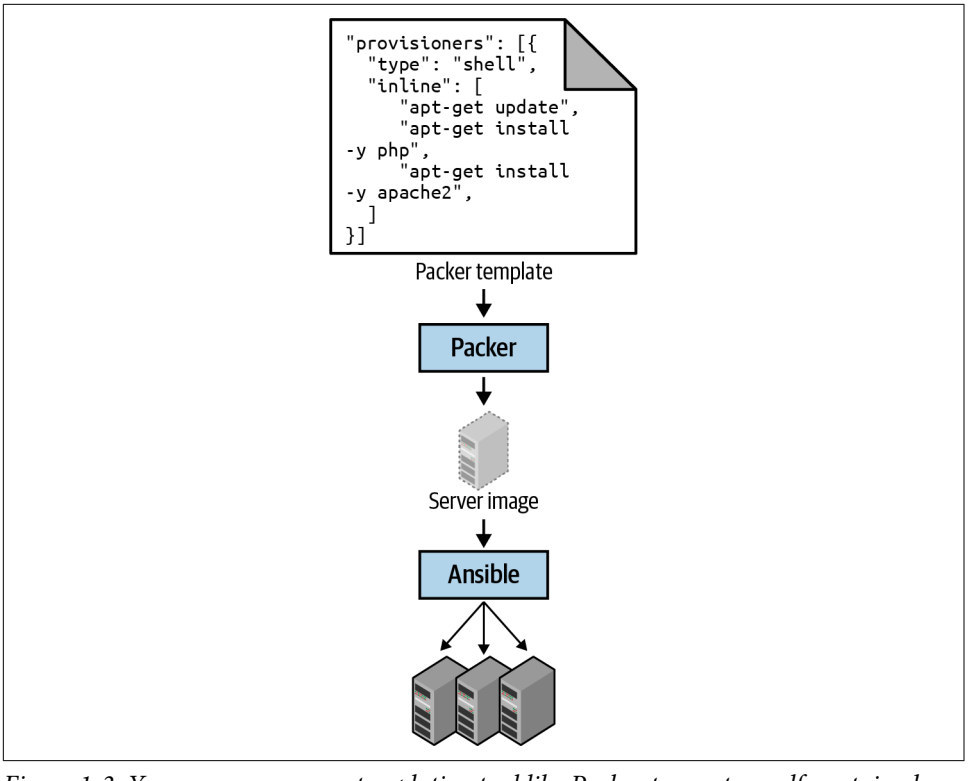


Figure 1-3. You can use a server templating tool like Packer to create a self-contained image of a server. You can then use other tools, such as Ansible, to install that image across all of your servers.

There are two broad categories of tools for working with images (Figure 1-4):

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* that you run on top of the hypervisor can see only the virtualized hardware, so it's fully isolated from the host machine and any other VM images, and it will run exactly the same way in all environments (e.g., your computer, a QA server, a production server). The drawback is that virtualizing all this hardware and running a totally separate OS 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 OS.[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 see only 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 (your computer, a QA server, a production server, etc.). The drawback is that all of the containers running on a single server share that server's OS kernel and hardware, so it's much more difficult 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; you'll see an example of how to use Docker in Chapter 7.

2 On most modern operating systems, code runs in one of two "spaces": *kernel space* or *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 OS (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 OS kernel instead. These APIs can enforce security restrictions (e.g., user permissions) and safety (e.g., a crash in a user space app typically affects only 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 might actively be performing malicious actions, you'll want the increased isolation guarantees of a VM.

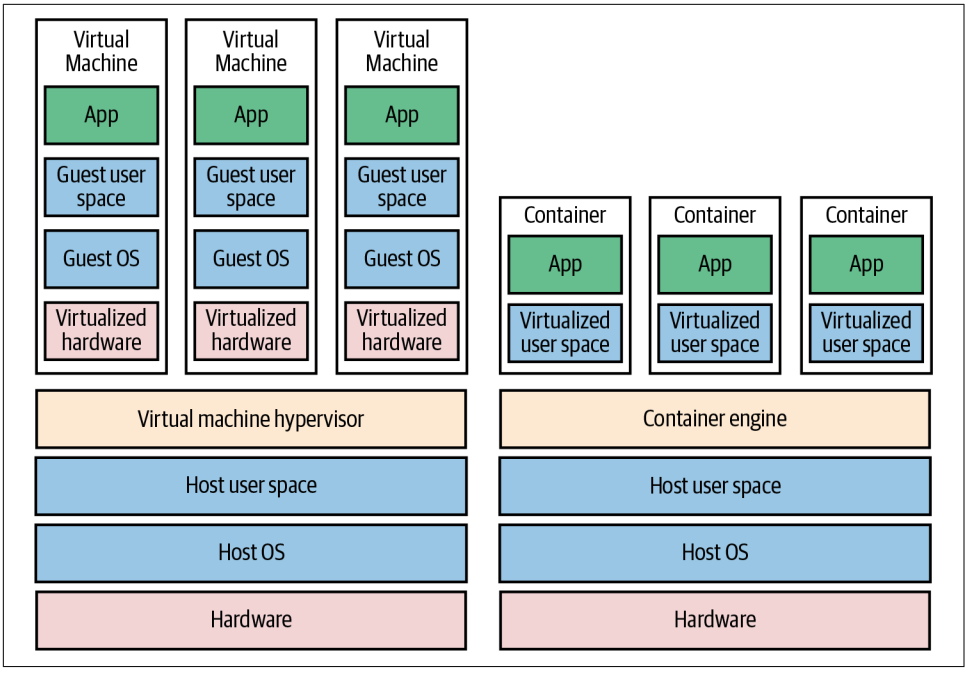


Figure 1-4. The two main types of images: VMs, on the left, and containers, on the right. VMs virtualize the hardware, whereas containers virtualize only user space.

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

```
{
  "builders": [{
    "ami_name": "packer-example-",
    "instance_type": "t2.micro",
    "region": "us-east-2",
    "type": "amazon-ebs",
    "source_ami": "ami-0fb653ca2d3203ac1",
    "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"
    ],
    "pause_before": "60s"
```

```
    }]
  }
```

This Packer template configures the same Apache web server that you saw in *setup-webserver.sh* using the same Bash code. The only difference between the code in the Packer template and the 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—for example, by deploying it on a server—that you should actually run that software.

To build an AMI from this template, run `packer build webserver.json`. After the build completes, you can install that AMI on all of your AWS servers and 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, as 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 after 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. Because 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, such as deploying a new version of your code, you create a new image from your server template and you deploy it on a new server. Because 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 do the following:

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

Handling these tasks is the realm of *orchestration tools* such as Kubernetes, Marathon/Mesos, Amazon Elastic Container Service (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 Amazon Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE), and Azure Kubernetes Service (AKS).

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

```yaml
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 instructs Kubernetes to create a *Deployment*, which is a declarative way to define the following:

- One or more Docker containers to run together. This group of containers is called a *Pod*. The Pod defined in the preceding code contains a single Docker container that runs Apache.
- The settings for each Docker container in the Pod. The Pod in the preceding code configures Apache to listen on port 80.
- How many copies (aka *replicas*) of the Pod to run in your cluster. The preceding code configures three replicas. Kubernetes automatically figures 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 containers on them), and so on. Kubernetes also constantly monitors the cluster to ensure that there are always three replicas running, automatically replacing any Pods that crash or stop responding.

- How to deploy updates. When deploying a new version of the Docker container, the preceding code rolls out three new replicas, waits for them to be healthy, and then undeploys the three old replicas.

That's a lot of power in just a few lines of YAML! You run `kubectl apply -f example-app.yml` to instruct Kubernetes to deploy your app. You can then make changes to the YAML file and run `kubectl apply` again to roll out the updates. You can also manage both the Kubernetes cluster and the apps within it using Terraform; you'll see an example of this in Chapter 7.

Provisioning Tools

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

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-0fb653ca2d3203ac1"

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

Don't worry if you're not yet familiar with some of the syntax. 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.

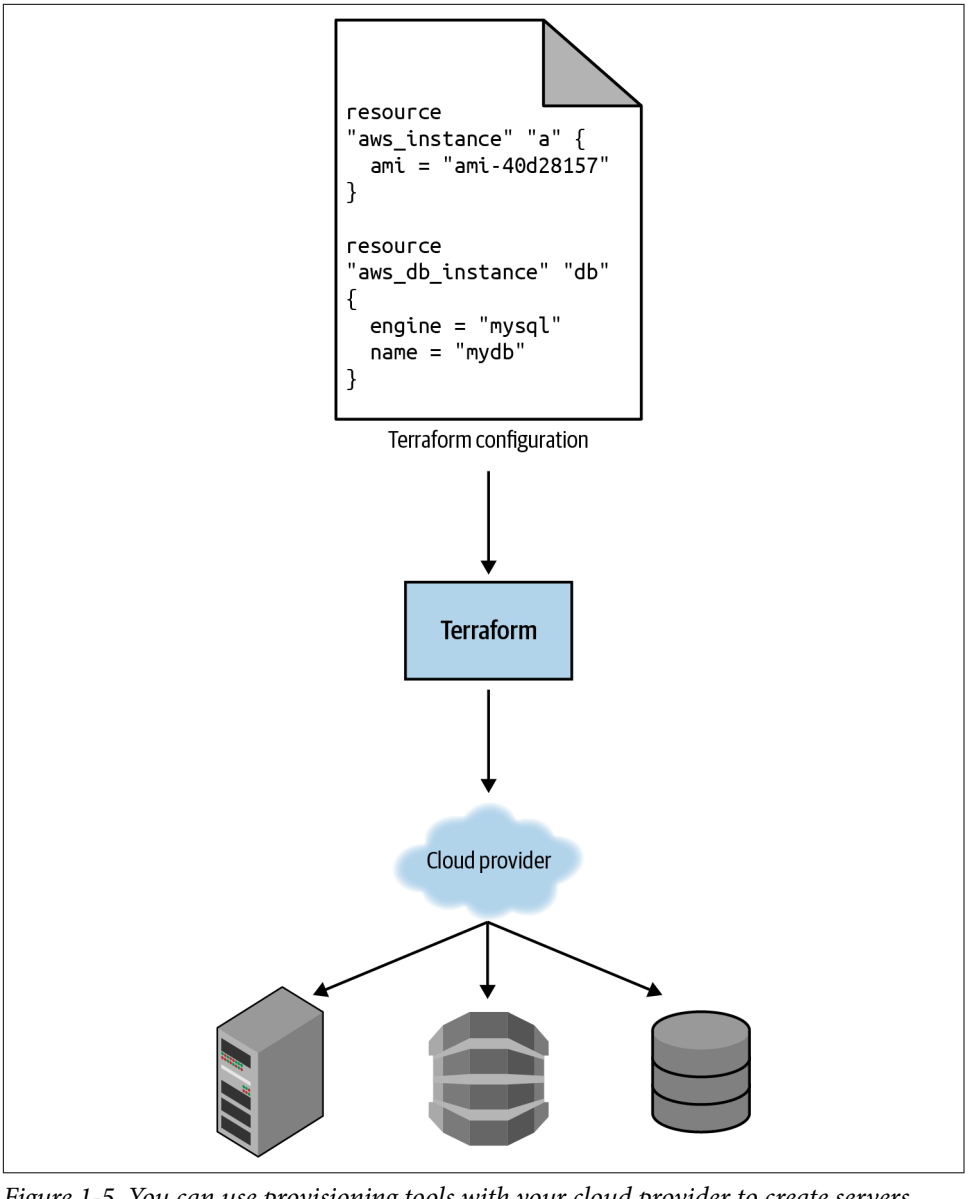


Figure 1-5. You can use provisioning tools with your cloud provider to create servers, databases, load balancers, and all other parts of your infrastructure.

What Are the Benefits of Infrastructure as Code?

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

The answer is that code is powerful. In exchange for the upfront 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 the following:

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, 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 will be significantly faster, since a computer can carry out the deployment steps far faster than a person, and safer, given that an automated process will be more consistent, more repeatable, and not prone to manual error.

Documentation

If the state of your infrastructure is locked away in a single sysadmin's head, and that sysadmin goes on vacation or leaves the company or gets hit by a bus,[4] you may suddenly realize you can no longer manage your own infrastructure. On the other hand, if your infrastructure is defined as code, then the state of your infrastructure is 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.

4 This is where the term *bus factor* comes from: your team's bus factor is the number of people you can lose (e.g., because they got hit by a bus) before you can no longer operate your business. You never want to have a bus factor of 1.

Version control

You can store your IaC source files in version control, which means that the entire history of your infrastructure is now captured in the commit log. This becomes a powerful tool for debugging issues, because 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 might 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, 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, since 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 best IaC tool for you. To answer that, I'm first going to go through 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 Does Terraform Work?

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`.

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

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 AWS, Azure, Google Cloud, DigitalOcean, OpenStack, and more. This means that 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 want to create. These configurations are the "code" in "infrastructure as code." Here's an example Terraform configuration:

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  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 instructs Terraform to make API calls to AWS to deploy a server, and then make API calls to Google Cloud to create a Domain Name System (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.

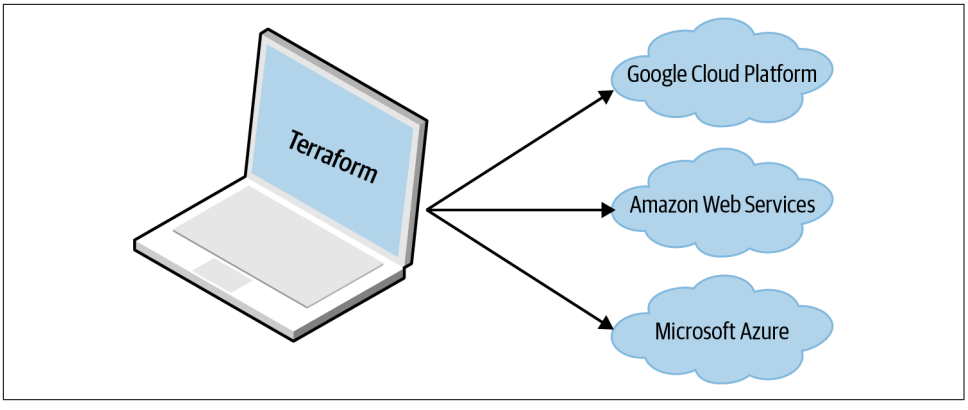


Figure 1-6. Terraform is a binary that translates the contents of your configurations into API calls to cloud providers.

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

Because Terraform supports many different cloud providers, a common question that arises 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 instruct Terraform to deploy exactly the same infrastructure in another cloud provider, such as Azure or Google Cloud, in just a few commands?

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 from 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 IaC practices under the hood for all providers.

How Does Terraform Compare to Other IaC 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 more difficult 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 as if you could be equally successful with any of them. And although 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, Pulumi, CloudFormation, and OpenStack Heat. My goal is to help you decide whether 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 even though your particular priorities might be different than mine, my hope is that sharing this thought process will help you to 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
- General-purpose language versus domain-specific language
- Master versus masterless
- Agent versus agentless
- Paid versus free offering
- Large community versus small community
- Mature versus cutting-edge
- Use of multiple tools together

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.

Configuration Management Versus Provisioning

As you saw earlier, Chef, Puppet, and Ansible are all configuration management tools, whereas CloudFormation, Terraform, OpenStack Heat, and Pulumi are all provisioning tools.

Although the distinction is not entirely clear cut, given that configuration management tools can typically do some degree of provisioning (e.g., you can deploy a server with Ansible) and that 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.

In particular, if you use server templating tools, 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. In Chapter 7, you'll see an example of how to use Terraform and Docker together, which is a particularly popular combination these days.

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, a popular combination is to use Terraform to provision your servers and Ansible to configure each one.

Mutable Infrastructure Versus Immutable Infrastructure

Configuration management tools such as Chef, Puppet, and Ansible typically default to a mutable infrastructure paradigm.

For example, if you instruct Chef to install a new version of OpenSSL, it will 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 difficult to catch; a configuration management change might work just fine on a test server, but that same change might 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, 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. Because 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, because 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 lasts only until you actually run the image. After a server is up and running, it will begin 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 in which you write code that specifies, step by step, how to achieve some desired end state.

Terraform, CloudFormation, Puppet, OpenStack Heat, and Pulumi all encourage a more *declarative* style in which 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 that you want to deploy 10 servers (*EC2 Instances* in AWS lingo) to run an AMI with ID `ami-0fb653ca2d3203ac1` (Ubuntu 20.04). Here is a simplified example of an Ansible template that does this using a procedural approach:

```
- ec2:
    count: 10
    image: ami-0fb653ca2d3203ac1
    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-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

On the surface, these two approaches might 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 need to be aware of what is already deployed and write a totally new procedural script to add the five new servers:

```
- ec2:
    count: 5
    image: ami-0fb653ca2d3203ac1
    instance_type: t2.micro
```

With declarative code, because all you do is declare the end state that 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 five more servers, all you need 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-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

If you applied this configuration, Terraform would realize it had already created 10 servers and therefore all it needs to do is create five 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-0fb653ca2d3203ac1"
    + instance_type  = "t2.micro"
    + (...)
  }

# aws_instance.example[12] will be created
+ resource "aws_instance" "example" {
    + ami            = "ami-0fb653ca2d3203ac1"
    + instance_type  = "t2.micro"
    + (...)
  }

# aws_instance.example[13] will be created
+ resource "aws_instance" "example" {
    + ami            = "ami-0fb653ca2d3203ac1"
```

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

# aws_instance.example[14] will be created
+ resource "aws_instance" "example" {
    + ami            = "ami-0fb653ca2d3203ac1"
    + 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 need 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 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: for example, you may have to manually configure your code to look up existing Instances not only by tag but also by image version, Availability Zone, and other parameters. This highlights two major problems with procedural IaC tools:

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 need 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 codebase itself. In other words, to reason about an Ansible or Chef codebase, you need to know the full history of every change that has ever happened.

Procedural code limits reusability

The reusability of procedural code is inherently limited because you must manually take into account the current state of the infrastructure. Because that state is constantly changing, code you used a week ago might 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 determine 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, since you don't need 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.

General-Purpose Language Versus Domain-Specific Language

Chef and Pulumi allow you to use a *general-purpose programming language* (GPL) to manage infrastructure as code: Chef supports Ruby; Pulumi supports a wide variety of GPLs, including JavaScript, TypeScript, Python, Go, C#, Java, and others. Terraform, Puppet, Ansible, CloudFormation, and OpenStack Heat each use a *domain-specific language* (DSL) to manage infrastructure as code: Terraform uses HCL; Puppet uses Puppet Language; Ansible, CloudFormation, and OpenStack Heat use YAML (CloudFormation also supports JSON).

The distinction between GPLs and DSLs is not entirely clear-cut—it's more of a helpful mental model than a clean, separate categorization—but the basic idea is that DSLs are designed for use in one specific domain, whereas GPLs can be used across a broad range of domains. For example, the HCL code you write for Terraform works only with Terraform and is limited solely to the functionality supported by Terraform, such as deploying infrastructure. This is in contrast to using a GPL such as JavaScript with Pulumi, where the code you write can not only manage infrastructure using Pulumi libraries but also perform almost any other programming task you wish, such as run a web app (in fact, Pulumi offers an Automation API you can use to embed Pulumi within your application code), perform complicated control logic (loops, conditionals, and abstraction are all easier to do in a GPL than a DSL), run various validations and tests, integrate with other tools and APIs, and so on.

DSLs have several advantages over GPLs:

Easier to learn
: Since DSLs, by design, deal with just one domain, they tend to be smaller and simpler languages than GPLs and therefore are easier to learn than GPLs. Most developers will be able to learn Terraform faster than, say, Java.

Clearer and more concise
: Since DSLs are designed for one specific purpose, with all the keywords in the language built to do that one thing, code written in DSLs tends to be easier to

understand and more concise than code written to do the exact same thing but written in a GPL. The code to deploy a single server in AWS is usually going to be shorter and easier to understand in Terraform than in Java.

More uniform
: Most DSLs are limited in what they allow you to do. This has some drawbacks, as I'll mention shortly, but one of the advantages is that code written in DSLs typically uses a uniform, predictable structure, so it's easier to navigate and understand than code written in GPLs, where every developer might solve the same problem in a completely different way. There's really only one way to deploy a server in AWS using Terraform; there are hundreds of ways to do the same thing with Java.

GPLs also have several advantages over DSLs:

Possibly no need to learn anything new
: Since GPLs are used in many domains, there's a chance you might not have to learn a new language at all. This is especially true of Pulumi, as it supports several of the most popular languages in the world, including JavaScript, Python, and Java. If you already know Java, you'll be able to jump into Pulumi faster than if you had to learn HCL to use Terraform.

Bigger ecosystem and more mature tooling
: Since GPLs are used in many domains, they have far bigger communities and much more mature tooling than a typical DSL. The number and quality of Integrated Development Environments (IDEs), libraries, patterns, testing tools, and so on for Java vastly exceeds what's available for Terraform.

More power
: GPLs, by design, can be used to do almost any programming task, so they offer much more power and functionality than DSLs. Certain tasks, such as control logic (loops and conditionals), automated testing, code reuse, abstraction, and integration with other tools, are far easier with Java than with Terraform.

Master Versus Masterless

By default, Chef and Puppet 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 of 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 need to deploy an extra server, or even a cluster of extra servers (for high availability and scalability), just to run the master.

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

Security
: You need 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 and Puppet do have varying levels of support for masterless modes where you run just their agent software on each of your servers, typically on a periodic schedule (e.g., a cron job that runs every five 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 I discuss 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, Terraform, and Pulumi are all masterless by default. Or, to be more accurate, some of them rely on a master server, but it's already part of the infrastructure you're using and not an extra piece that you need 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 that 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 need to run any extra infrastructure or manage extra authentication mechanisms (i.e., just use your SSH keys).

Agent Versus Agentless

Chef and Puppet require you to install *agent software* (e.g., Chef Client, Puppet Agent) on each server that 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 that 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 in which 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 need to update the agent software on a periodic basis, being careful to keep it synchronized with the master server if there is one. You also need 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), you need to open outbound ports on every server. If the master server pushes configuration to the agent, you need to open inbound ports on every server. In either case, you must figure out how to authenticate the agent to the server to which it's communicating. All of this increases your surface area to attackers.

Once again, Chef and Puppet do have varying levels of support for agentless modes, 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 and Puppet almost always includes an agent and usually a master, too, as shown in Figure 1-7.

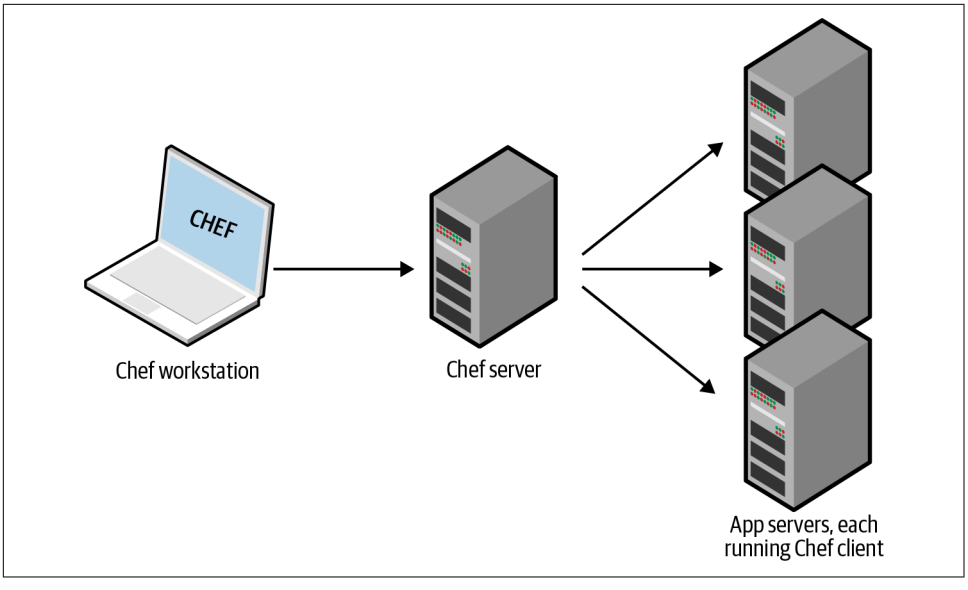


Figure 1-7. The typical architecture for Chef and Puppet involves many moving parts. For example, the default setup for Chef is to run the Chef client on your computer, which talks to a Chef master server, which deploys changes by communicating with Chef clients running on all your other servers.

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 need to figure out whether 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 communicates with the master server(s), or the way other servers communicate with the master server(s), or…

Ansible, CloudFormation, Heat, Terraform, and Pulumi 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 of the 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 need 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.

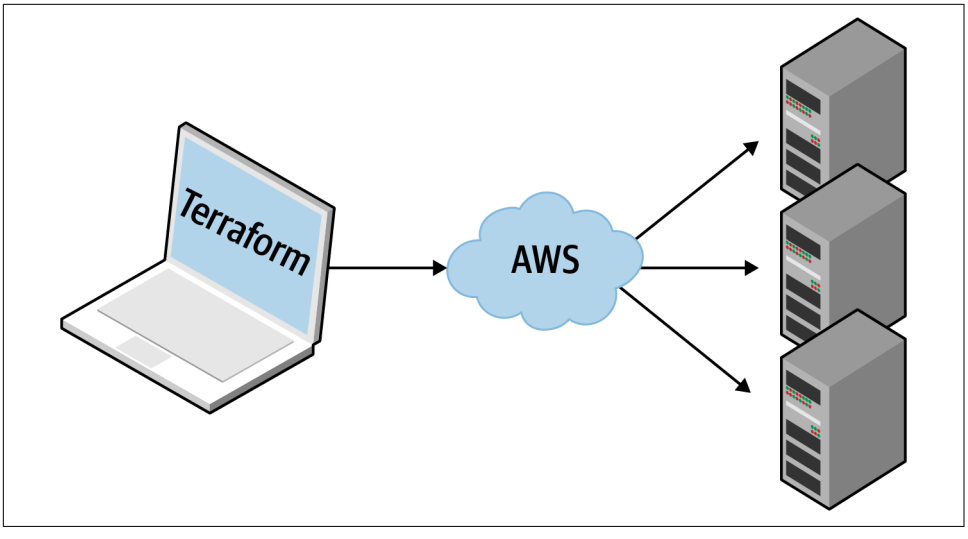


Figure 1-8. Terraform uses a masterless, agentless architecture. All you need to run is the Terraform client, and it takes care of the rest by using the APIs of cloud providers, such as AWS.

Paid Versus Free Offering

CloudFormation and OpenStack Heat are completely free: the resources you deploy with those tools may cost money, but you don't pay anything to use the tools themselves. Terraform, Chef, Puppet, Ansible, and Pulumi are all available in free versions and paid versions: for example, you can use the free and open source version of Terraform by itself, or you could choose to use it with HashiCorp's paid product, Terraform Cloud. The price points, packaging, and trade-offs with the paid versions are beyond the scope of this book. The one question I want to focus on here is whether the free version is so limited that you are effectively *forced* to use the paid offering for real-world, production use cases.

To be clear, there's nothing wrong with a company offering a paid service for one of these tools; in fact, if you're using these tools in production, I strongly recommend looking into the paid services, as many of them are well worth the money. However, you have to realize that those paid services aren't under your control—they could go out of business, or get acquired (e.g., Chef, Puppet, and Ansible have all gone through acquisitions that had significant impacts on their paid product offerings), or change their pricing model (e.g., Pulumi changed its pricing in 2021, which benefited some users but increased prices by ~10x for others), or change the product, or discontinue the product entirely—so it's important to know whether the IaC tool you picked would still be usable if, for some reason, you couldn't use one of these paid services.

In my experience, the free versions of Terraform, Chef, Puppet, and Ansible can all be used successfully for production use cases; the paid services can make these tools even better, but if they weren't available, you could still get by. Pulumi, on the other hand, is harder to use in production without the paid offering known as Pulumi Service.

A key part of managing infrastructure as code is managing state (you'll learn about how Terraform manages state in Chapter 3), and Pulumi, by default, uses Pulumi Service as the backend for state storage. You can switch to other supported backends for state storage, such as Amazon S3, Azure Blob Storage, or Google Cloud Storage, but the Pulumi backend documentation explains that only Pulumi Service supports transactional checkpointing (for fault tolerance and recovery), concurrent state locking (to prevent corrupting your infrastructure state in a team environment), and encrypted state in transit and at rest. In my opinion, without these features, it's not practical to use Pulumi in any sort of production environment (i.e., with more than one developer), so if you're going to use Pulumi, you more or less have to pay for Pulumi Service.

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 plugins, integrations, and extensions are available; how easy it is to find help online (e.g., blog posts, questions on Stack Overflow); and how easy it is to hire someone to help you (e.g., an employee, consultant, or support company).

It's difficult 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 in June 2022, 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 open source libraries are available for the tool, and the number of questions listed for that tool on Stack Overflow.[7]

7 The data on contributors and stars comes from the open source repositories (mostly GitHub) for each tool. Because CloudFormation is closed source, this information is not available.

Table 1-1. A comparison of IaC communities

	Source	Cloud	Contributors	Stars	Libraries	Stack Overflow
Chef	Open	All	640	6,910	3,695[a]	8,295
Puppet	Open	All	571	6,581	6,871[b]	3,996
Ansible	Open	All	5,328	53,479	31,329[c]	22,052
Pulumi	Open	All	1,402	12,723	15[d]	327
CloudFormation	Closed	AWS	?	?	369[e]	7,252
Heat	Open	All	395	379	0[f]	103
Terraform	Open	All	1,621	33,019	9,641[g]	13,370

[a] This is the number of cookbooks in the Chef Supermarket.
[b] This is the number of modules in Puppet Forge.
[c] This is the number of reusable roles in Ansible Galaxy.
[d] This is the number of packages in the Pulumi Registry.
[e] This is the number of templates in AWS Quick Starts.
[f] I could not find any collections of community Heat templates.
[g] 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: e.g., Terraform split the provider code (i.e., the code specific to AWS, Google Cloud, Azure, etc.) out into separate repos in 2017, so the preceding table significantly understates activity; some tools offer alternatives to Stack Overflow for questions; 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 works only with AWS. Second, Ansible and Terraform seem to be the clear leads in terms of popularity.

Another interesting trend to note is how these numbers have changed since the first edition of the book. Table 1-2 shows the percentage change in each of the numbers from the values I gathered in the first edition back in September 2016. (Note: Pulumi is not included in this table, as it wasn't part of this comparison in the first edition of the book.)

Table 1-2. How the IaC communities have changed between September 2016 and June 2022

	Source	Cloud	Contributors	Stars	Libraries	Stack Overflow
Chef	Open	All	+34%	+56%	+21%	+98%
Puppet	Open	All	+32%	+58%	+55%	+51%
Ansible	Open	All	+258%	+183%	+289%	+507%
CloudFormation	Closed	AWS	?	?	+54%[a]	+1,083%
Heat	Open	All	+40%	+34%	0	+98%
Terraform	Open	All	+148%	+476%	+24,003%	+10,106%

[a] In earlier editions of the book, I used CloudFormation templates in the awslabs GitHub repo, but these seem to be gone now, so I used AWS Quick Starts in this edition, so the numbers aren't directly comparable.

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, and Stack Overflow posts is through the roof. 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. Is this a technology that has been around for years, where all the usage patterns, best practices, problems, and failure modes are well understood? Or is this a new technology where you'll have to learn all those hard lessons from scratch? Table 1-3 shows the initial release dates, current version numbers (as of June 2022), and my own subjective perception of the maturity of each of the IaC tools.

Table 1-3. A comparison of IaC maturity as of June 2022

	Initial release	Current version	Perceived maturity
Chef	2009	17.10.3	High
Puppet	2005	7.17.0	High
Ansible	2012	5.9.0	Medium
Pulumi	2017	3.34.1	Low
CloudFormation	2011	???	Medium
Heat	2012	18.0.0	Low
Terraform	2014	1.2.3	Medium

Again, this is not an apples-to-apples comparison: age alone does not determine maturity—neither does a high version number (different tools have different versioning schemes). Still, some trends are clear. Pulumi is the youngest IaC tool in this comparison and, arguably, the least mature: this becomes apparent when you search for documentation, best practices, community modules, etc. Terraform is a bit more

mature these days: the tooling has improved, the best practices are better understood, there are far more learning resources available (including this book!), and now that it has reached the 1.0.0 milestone, it is a considerably more stable and reliable tool than when the first and second editions of this book came out. Chef and Puppet are the oldest and arguably most mature tools on this list.

Use of 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 tools for the job.

The following sections show three common combinations I've seen work well at a number of companies.

Provisioning plus configuration management

Example: Terraform and Ansible. You use Terraform to deploy all the underlying infrastructure, including the network topology (i.e., virtual private clouds [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, as depicted in Figure 1-9.

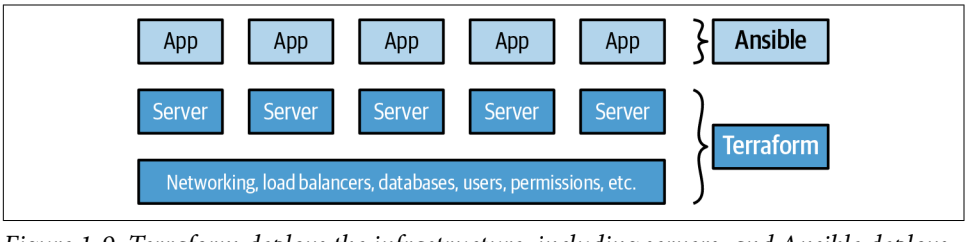


Figure 1-9. Terraform deploys the infrastructure, including servers, and Ansible deploys apps onto those servers.

This is an easy approach to get started with, because 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 servers and configure them). The major downside is that using Ansible typically means that you're writing a lot of procedural code, with mutable servers, so as your codebase, infrastructure, and team grow, maintenance can become more difficult.

Provisioning plus server templating

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

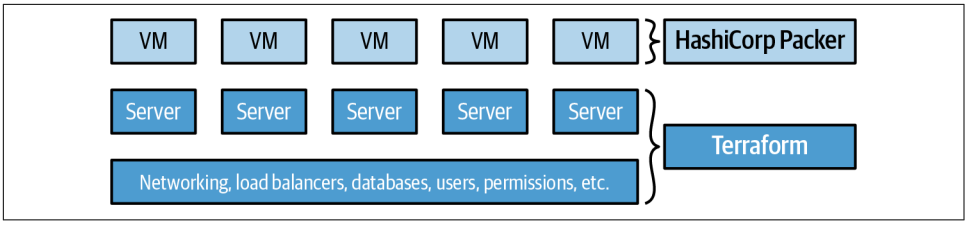


Figure 1-10. Terraform deploys the infrastructure, including servers, and Packer creates the VMs that run on those servers.

This is also an easy approach to get started with, because there is no extra infrastructure to run (Terraform and Packer are both client-only applications), and you'll get plenty of practice deploying VM 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, VMs 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

Example: Terraform, Packer, Docker, and Kubernetes. You use Packer to create a VM image that has Docker and Kubernetes agents installed. You then use Terraform to deploy a cluster of servers, each of which runs this VM image, and 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, as shown in Figure 1-11.

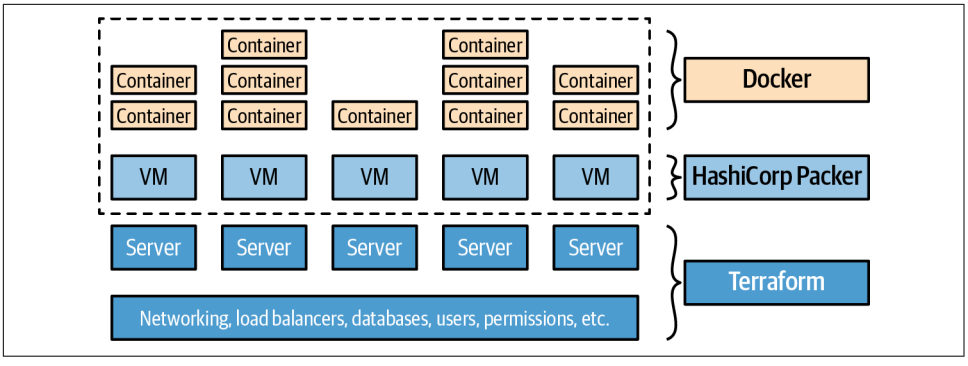


Figure 1-11. Terraform deploys the infrastructure, including servers; Packer creates the VMs that run on those servers; and Kubernetes manages those VMs as a cluster for running Docker containers.

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 of 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.

You'll see an example of this approach in Chapter 7.

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., you can use Chef without a master, you can use Puppet to do immutable infrastructure, etc.).

Table 1-4. A comparison of the most common ways to use the most popular IaC tools

	Chef	Puppet	Ansible	Pulumi	CloudFormation	Heat	Terraform
Source	Open	Open	Open	Open	Closed	Open	Open
Cloud	All	All	All	All	AWS	All	All
Type	Config mgmt	Config mgmt	Config mgmt	Provisioning	Provisioning	Provisioning	Provisioning
Infra	Mutable	Mutable	Mutable	Immutable	Immutable	Immutable	Immutable
Paradigm	Procedural	Declarative	Procedural	Declarative	Declarative	Declarative	Declarative
Language	GPL	DSL	DSL	GPL	DSL	DSL	DSL

	Chef	Puppet	Ansible	Pulumi	CloudFormation	Heat	Terraform
Master	Yes	Yes	No	No	No	No	No
Agent	Yes	Yes	No	No	No	No	No
Paid Service	Optional	Optional	Optional	Must-have	N/A	N/A	Optional
Community	Large	Large	Huge	Small	Small	Small	Huge
Maturity	High	High	Medium	Low	Medium	Low	Medium

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

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

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 develop this example even further.

Terraform can provision infrastructure across public cloud providers such as 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 of the following:

- AWS is the most popular cloud infrastructure provider, by far. It has a 32% share in the cloud infrastructure market, which is more than the next three biggest competitors (Microsoft, Google, and IBM) combined.[1]
- AWS provides a huge range of reliable and scalable cloud-hosting services, including Amazon Elastic Compute Cloud (Amazon 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.[2]

1 Source: "Global Cloud Services Spend Hits Record US$49.4 Billion in Q3 2021", Canalys, October 28, 2021.

2 If you find the AWS terminology confusing, be sure to check out Amazon Web Services in Plain English.

- AWS offers a Free Tier for the first year that should allow you to run all of these examples for free or a very low cost.[3] If you already used 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; this tutorial is designed for novices to both technologies. I'll walk you through the following steps:

- Setting up your AWS account
- Installing Terraform
- Deploying a single server
- Deploying a single web server
- Deploying a configurable web server
- Deploying a cluster of web servers
- Deploying a load balancer
- Cleaning up

Example Code

As a reminder, you can find all of the code examples in the book on GitHub.

Setting 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 the *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 then switch to one of those accounts immediately.[4]

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*, go to the IAM Console, click Users, and then click the Add Users button. Enter a name for the user, and make sure "Access key - Programmatic access" is selected, as shown in Figure 2-1 (note

3 Check out the AWS Free Tier documentation for details.

4 For more details on AWS user management best practices, see the documentation.

that AWS occasionally makes changes to its web console, so what you see may look slightly different than the screenshots in this book).

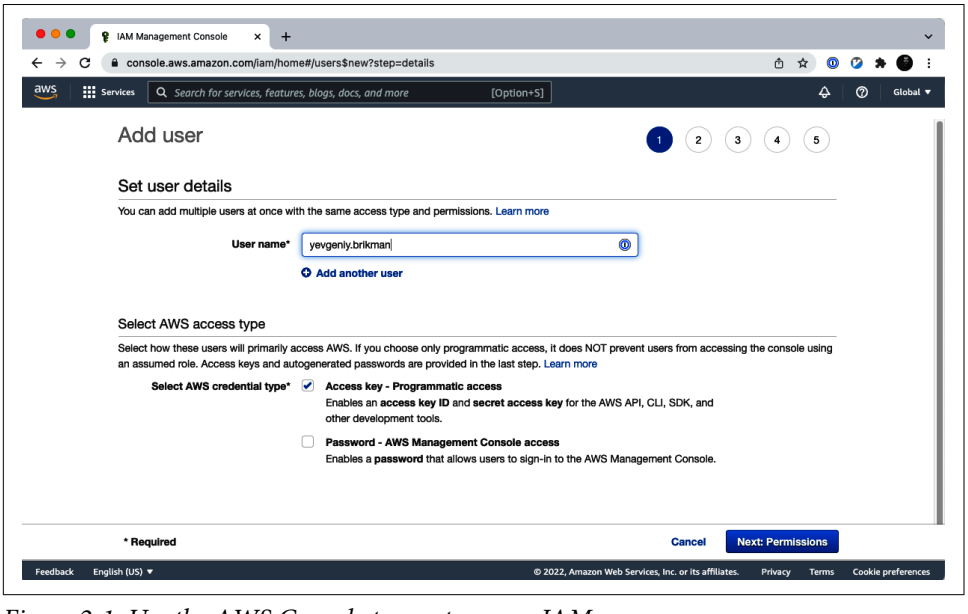


Figure 2-1. Use the AWS Console to create a new IAM user.

Click the Next button. AWS will ask you to add permissions to the user. By default, new IAM users have no permissions whatsoever and cannot do anything in an AWS account. To give your IAM user the ability 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 built into your AWS account, which are known as *Managed Policies*.[5]

To run the examples in this book, the easiest way to get started is to add the `AdministratorAccess` Managed Policy to your IAM user (search for it, and click the checkbox next to it), as shown in Figure 2-2.[6]

5 You can learn more about IAM Policies on the AWS website.

6 I'm assuming that you're running the examples in this book in an AWS account dedicated solely to learning and testing so that the broad permissions of the `AdministratorAccess` Managed Policy are not a big risk. If you are running these examples in a more sensitive environment—which, for the record, I don't recommend!—and you're comfortable with creating custom IAM Policies, you can find a more pared-down set of permissions in this book's code examples repo.

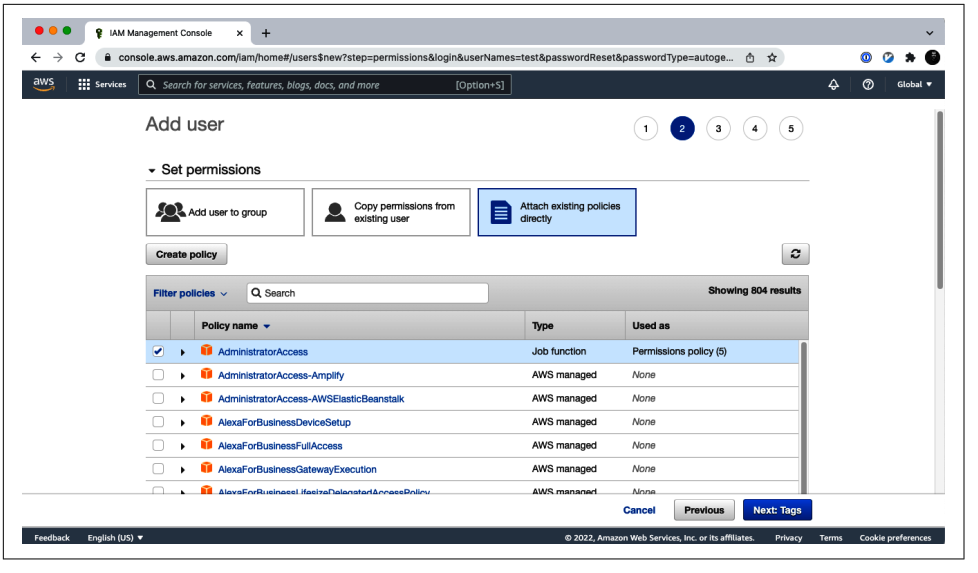


Figure 2-2. Add the `AdministratorAccess` Managed IAM Policy to your new IAM user.

Click Next a couple more times and then the "Create user" button. 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 2-3. You must save these immediately because 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 macOS Keychain), and never share them with anyone.

After you've saved your credentials, click the Close button. You're now ready to move on to using Terraform.

A Note on Default Virtual Private Clouds

All of the AWS examples in this book use the *Default VPC* in your AWS account. 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 AWS account created after 2013. If for some reason you deleted the Default VPC 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.

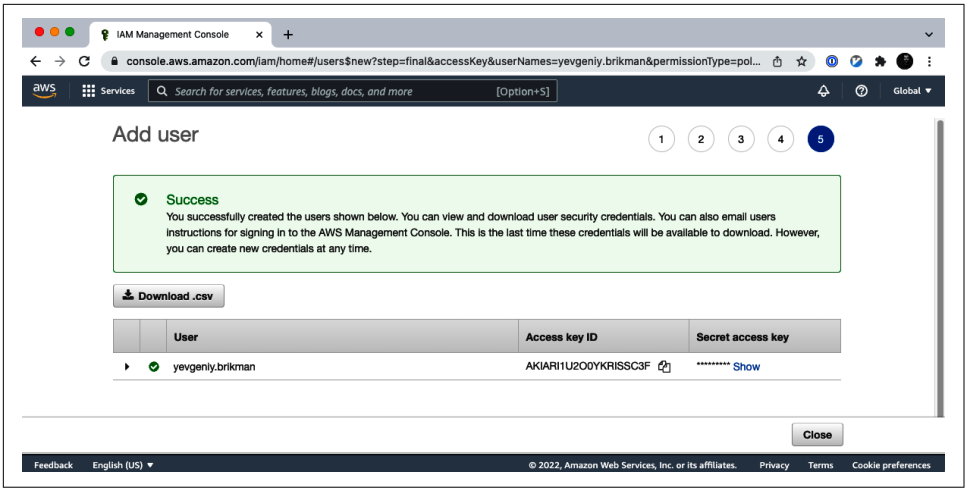


Figure 2-3. Store your AWS credentials somewhere secure. Never share them with anyone. (Don't worry, the ones in the screenshot are fake.)

Installing Terraform

The easiest way to install Terraform is to use your operating system's package manager. For example, on macOS, if you are a Homebrew user, you can run the following:

```
$ brew tap hashicorp/tap
$ brew install hashicorp/tap/terraform
```

On Windows, if you're a Chocolatey user, you can run the following:

```
$ choco install terraform
```

Check the Terraform documentation for installation instructions on other operating systems, including the various flavors of Linux.

Alternatively, you can install Terraform manually by going to the Terraform home page, clicking the download link, selecting the appropriate package for your operating system, downloading the ZIP archive, and unzipping it into the directory where you want Terraform to be installed. The archive will extract a single binary called *terraform*, which you'll want to add to your `PATH` environment variable.

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

```
$ terraform
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.
```

```
Main commands:
  init          Prepare your working directory for other commands
  validate      Check whether the configuration is valid
  plan          Show changes required by the current configuration
  apply         Create or update infrastructure
  destroy       Destroy previously-created infrastructure

(...)
```

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/macOS terminal:

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

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

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

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

Other AWS Authentication Options

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

Deploying a Single Server

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

7 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 in the Terraform documentation.

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

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

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

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

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

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

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

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

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

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

8 You can learn more about AWS regions and Availability Zones on the AWS website.

`ami`

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

`instance_type`

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

Use the Docs!

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

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

```
$ terraform init

Initializing the backend...

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

Terraform has been successfully initialized!
```

The `terraform` binary contains the basic functionality for Terraform, but it does not come with the code for any of the providers (e.g., the AWS Provider, Azure provider, GCP provider, etc.), so when you're first starting to use Terraform, you need to

9 Finding AMI IDs is surprisingly complicated, as documented in this Gruntwork blog post.

run `terraform init` to tell Terraform to scan the code, figure out which providers you're using, and download the code for them. By default, the provider code will be downloaded into a *.terraform* folder, which is Terraform's scratch directory (you may want to add it to *.gitignore*). Terraform will also record information about the provider code it downloaded into a *.terraform.lock.hcl* file (you'll learn more about this file in "Versioned Modules" on page 298). You'll see a few other uses for the `init` command and *.terraform* folder in later chapters. For now, just be aware that you need to run `init` anytime you start with new Terraform code and that it's safe to run `init` multiple times (the command is idempotent).

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

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_instance.example will be created
  + resource "aws_instance" "example" {
      + ami                          = "ami-0fb653ca2d3203ac1"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t2.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
      + key_name                     = (known after apply)
      (...)
  }

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

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

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

```
$ terraform apply

(...)

Terraform will perform the following actions:

  # aws_instance.example will be created
  + resource "aws_instance" "example" {
      + ami                          = "ami-0fb653ca2d3203ac1"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t2.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
      + key_name                     = (known after apply)
      (...)
  }

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

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

  Enter a value:
```

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

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

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

  Enter a value: yes

aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
```

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

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

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

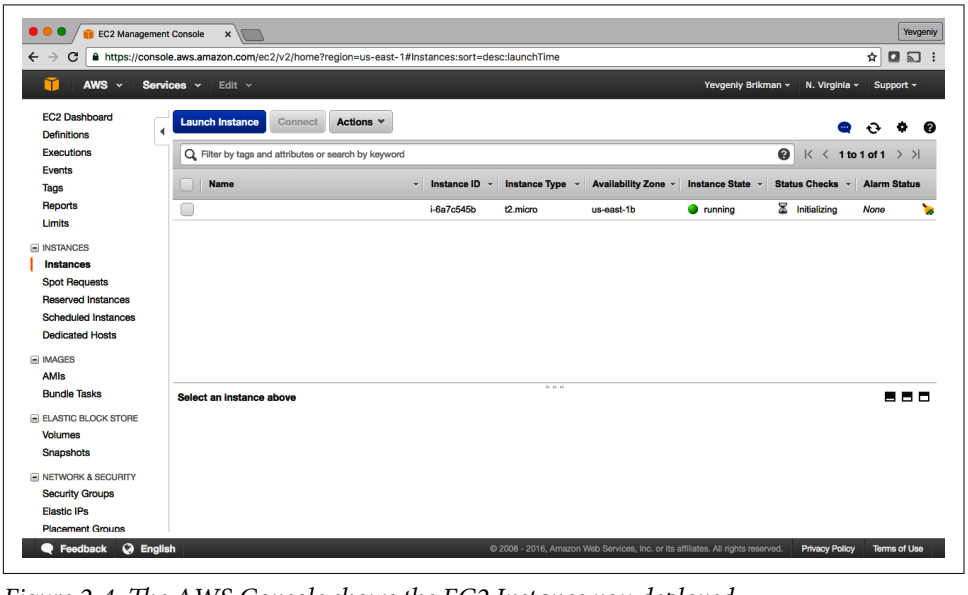


Figure 2-4. The AWS Console shows the EC2 Instance you deployed.

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

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

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

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

```
$ terraform apply

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

```
Terraform will perform the following actions:

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

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

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

  Enter a value:
```

Terraform keeps track of all the resources it already created for this set of configuration files, so it knows your EC2 Instance already exists (notice Terraform says `Refreshing state...` when you run the `apply` command), and it can show you a diff between what's currently deployed and what's in your Terraform code (this is one of the advantages of using a declarative language over a procedural one, as discussed in "How Does Terraform Compare to Other IaC Tools?" on page 20). The preceding diff shows that Terraform wants to create a single tag called `Name`, which is exactly what you need, so type **`yes`** and hit Enter.

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

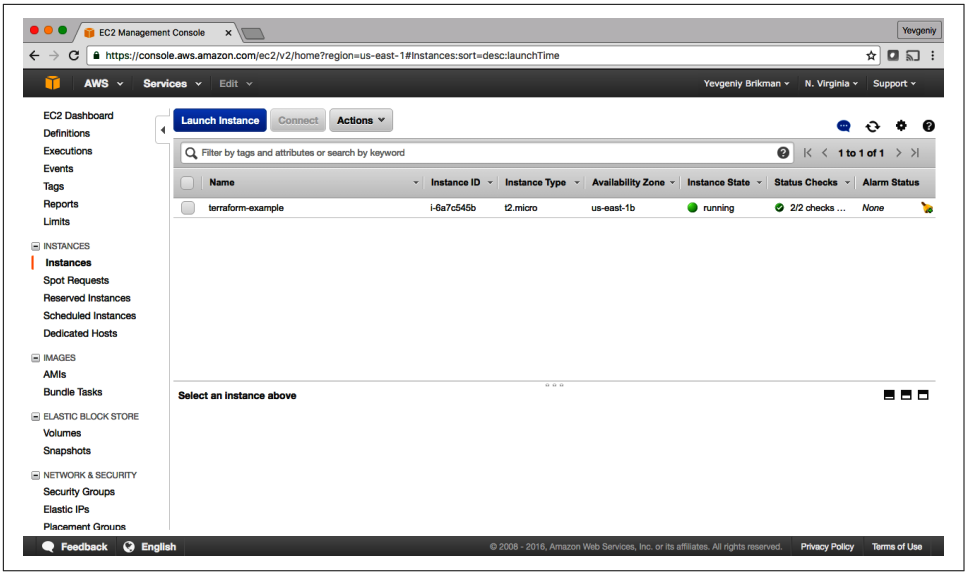


Figure 2-5. The EC2 Instance now has a name tag.

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

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

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

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

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

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

To share this code with your teammates, you'll want to create a shared Git repository that you can all access. One way to do this is to use GitHub. Head over to GitHub,

create an account if you don't have one already, and create a new repository. Configure your local Git repository to use the new GitHub repository as a remote endpoint named `origin`, as follows:

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

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

```
git push origin main
```

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

```
git pull origin main
```

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 of 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 10.

Deploying 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.

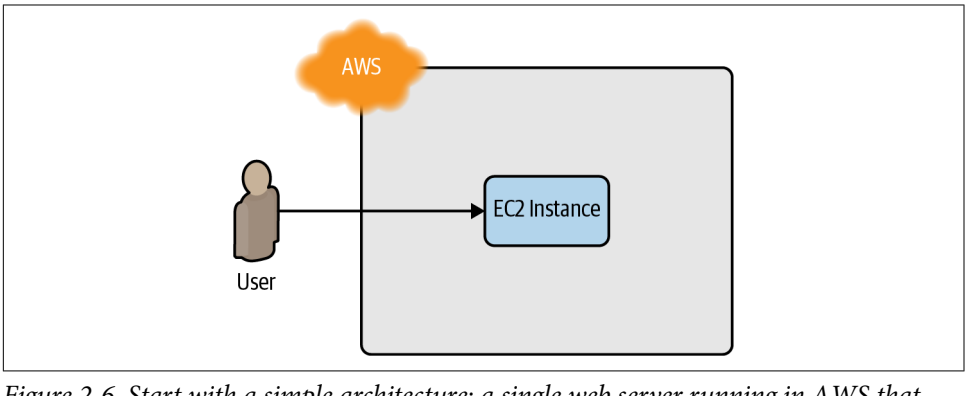


Figure 2-6. Start with a simple architecture: a single web server running in AWS that responds to HTTP requests.

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":[10]

```
#!/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 an ampersand (`&`) so that the web server runs permanently in the background, whereas 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 since 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" on page 7, 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 20.04 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 its very first boot. You 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-0fb653ca2d3203ac1"
  instance_type          = "t2.micro"

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

10 You can find a handy list of HTTP server one-liners on GitHub.

```
              EOF

  user_data_replace_on_change = true

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

Two things to notice about the preceding code:

- The `<<-EOF` and `EOF` are Terraform's *heredoc* syntax, which allows you to create multiline strings without having to insert `\n` characters all over the place.
- The `user_data_replace_on_change` parameter is set to `true` so that when you change the `user_data` parameter and run `apply`, Terraform will terminate the original instance and launch a totally new one. Terraform's default behavior is to update the original instance in place, but since User Data runs only on the very first boot, and your original instance already went through that boot process, you need to force the creation of a new instance to ensure your new User Data script actually gets executed.

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 begin 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.[11]

11 To learn more about how CIDR works, see its Wikipedia page. For a handy calculator that converts between IP address ranges and CIDR notation, use either *https://cidr.xyz/* in your browser or install the `ipcalc` command in your terminal.

Simply creating a security group isn't enough; you 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-0fb653ca2d3203ac1"`) 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` as follows:

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

  user_data_replace_on_change = true

  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 determine in which 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, because 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, similar to the dependency graph shown in Figure 2-7, by using a desktop app such as Graphviz or web app like GraphvizOnline.[12]

12 Note that while the `graph` command can be useful for visualizing the relationships between a small number of resources, with dozens or hundreds of resources, the graphs tend to become too large and messy to be useful.

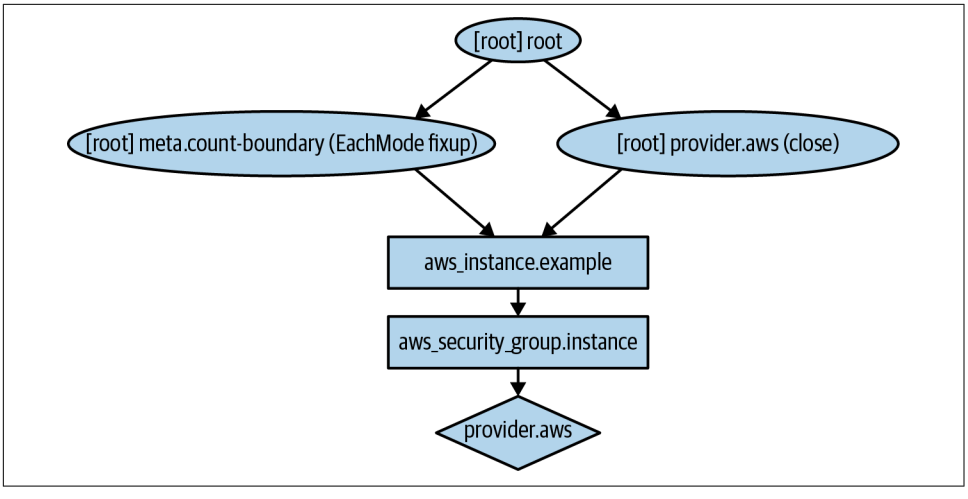


Figure 2-7. This is what the dependency graph for the EC2 Instance and its security group looks like when rendered with Graphviz.

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

If you run the `apply` command, you'll see that Terraform wants to create a security group and replace 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-0fb653ca2d3203ac1"
      ~ 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. Since you set `user_data_replace_on_change` to `true` and changed the `user_data` parameter, this will force a replacement, which means that 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" on page 7. It's worth mentioning that while the web server is being replaced, any users of that web server would experience downtime; you'll see how to do a zero-downtime deployment with Terraform in Chapter 5.

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

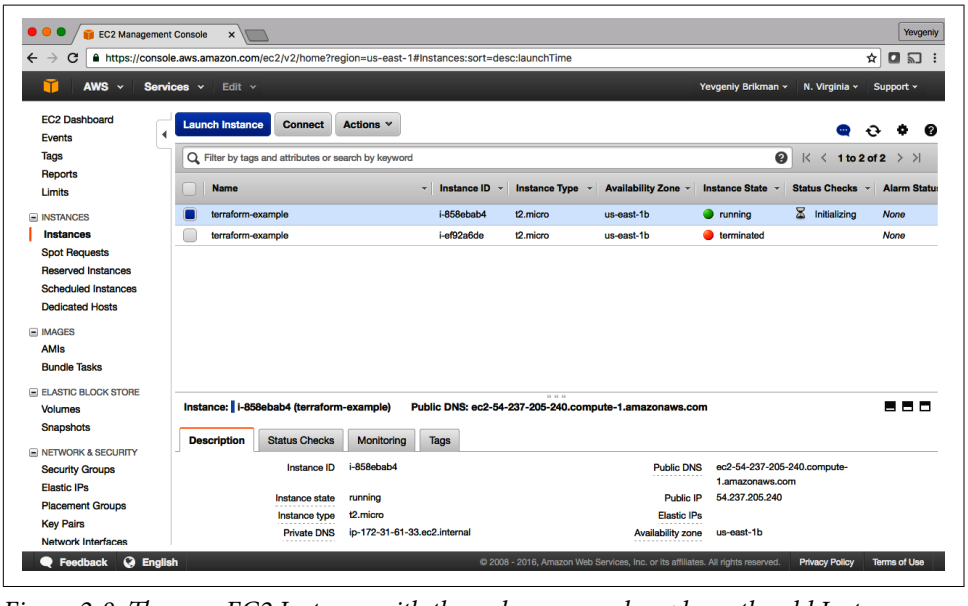


Figure 2-8. The new EC2 Instance with the web server code replaces the old Instance.

If you click 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 of the examples in this book simple, they deploy not only into your Default VPC (as mentioned earlier) but also into 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 be accessed only from within 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).

Deploying a Configurable Web Server

You might 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.[13] If you have the port number in two places, it's 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*. Here's the syntax for declaring a variable:

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

The body of the variable declaration can contain the following optional parameters:

`description`
: It's always a good idea to use this parameter to document how a variable is used. Your teammates will be able to see this description not only 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.

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

`type`
: This allows you to 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`. It's always a good idea to define a type constraint to catch simple errors. If you don't specify a type, Terraform assumes the type is `any`.

`validation`
: This allows you to define custom validation rules for the input variable that go beyond basic type checks, such as enforcing minimum or maximum values on a number. You'll see an example of validations in Chapter 8.

`sensitive`
: If you set this parameter to `true` on an input variable, Terraform will not log it when you run `plan` or `apply`. You should use this on any secrets you pass into your Terraform code via variables: e.g., passwords, API keys, etc. I'll talk more about secrets in Chapter 6.

Here is an example of an input variable that checks to verify that 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 whether 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 of 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 of 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` type constraint:

```
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 preceding example 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 try to set this variable to a value that doesn't match this type, Terraform immediately gives you a type error. The following example demonstrates trying 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.
```

Coming back to the web server example, what 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* by using 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 the following optional parameters:

`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 instruct Terraform not to log this output at the end of `plan` or `apply`. This is useful if the output variable contains secrets such as passwords or private keys. Note that if your output variable references an input variable or resource attribute marked with `sensitive = true`, you are *required* to mark the output variable with `sensitive = true` as well to indicate you are intentionally outputting a secret.

`depends_on`
: Normally, Terraform automatically figures out your dependency graph based on the references within your code, but in rare situations, you have to give it extra hints. For example, perhaps you have an output variable that returns the IP address of a server, but that IP won't be accessible until a security group (firewall) is properly configured for that server. In that case, you may explicitly tell Terraform there is a dependency between the IP address output variable and the security group resource using `depends_on`.

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 (because 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 might find useful (e.g., you now know what IP to test after 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.

Deploying 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.[14]

Managing such a cluster manually is a lot of work. Fortunately, you can let AWS take care of it for you by 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.

14 For a deeper look at how to build highly available and scalable systems on AWS, see "A Comprehensive Guide to Building a Scalable Web App on Amazon Web Services - Part 1" by Josh Padnick.

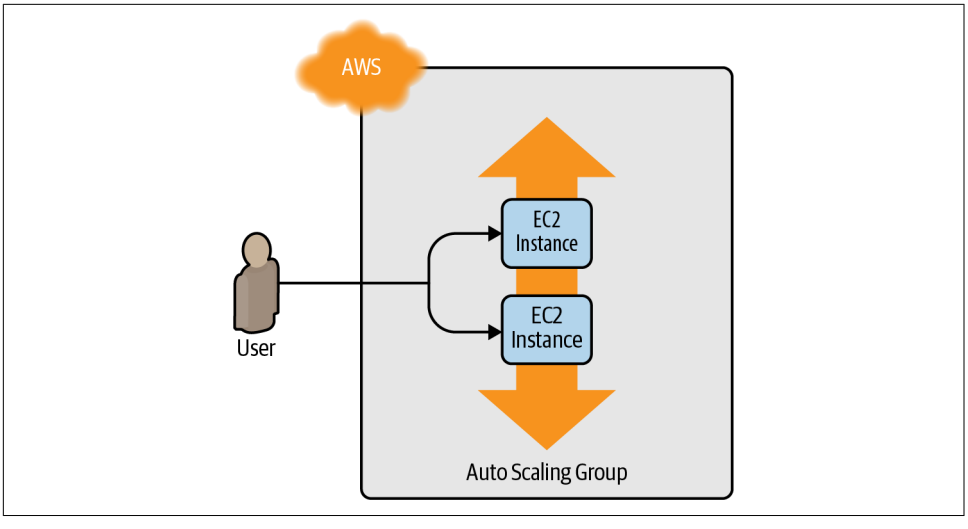


Figure 2-9. Instead of a single web server, run a cluster of web servers using an Auto Scaling Group.

The first step in creating an ASG is to create a *launch configuration*, which specifies how to configure each EC2 Instance in the ASG.[15] The `aws_launch_configuration` resource uses almost the same parameters as the `aws_instance resource`, although it doesn't support tags (you'll handle these in the `aws_autoscaling_group` resource later) or the `user_data_replace_on_change` parameter (ASGs launch new instances by default, so you don't need this parameter), and two of the parameters have different names (`ami` is now `image_id`, and `vpc_security_group_ids` is now `security_groups`), so replace `aws_instance` with `aws_launch_configuration` as follows:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0fb653ca2d3203ac1"
  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
}
```

15 These days, you should actually be using a *launch template* (and the `aws_launch_template` resource) with ASGs rather than a launch configuration. However, I've stuck with the launch configuration in the examples in this book as it is convenient for teaching some of the concepts in the zero-downtime deployment section of Chapter 5.

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 would delete the old resource first and then creates its replacement, but because 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-0fb653ca2d3203ac1"
  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.
  lifecycle {
    create_before_destroy = true
  }
}
```

There's also one other parameter that you need to add to your ASG to make it work: `subnet_ids`. This parameter specifies to the ASG into which VPC subnets the EC2 Instances should be deployed (see "Network Security" on page 59 for background info on subnets). Each subnet lives in an isolated AWS AZ (that is, isolated datacenter), so by deploying your Instances across multiple subnets, you ensure that your service can keep running even if some of the datacenters have an outage. You could hardcode 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 ...]
}
```

Here, `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 Virtual Private Clouds" on page 42 for background information):

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

Note that with data sources, the arguments you pass in are typically search filters that indicate to the data source what information you're looking for. With the `aws_vpc` data source, the only filter you need is `default = true`, which directs 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_subnets`, to look up the subnets within that VPC:

```
data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}
```

Finally, you can pull the subnet IDs out of the `aws_subnets` 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_subnets.default.ids

  min_size = 2
  max_size = 10

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

Deploying 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.

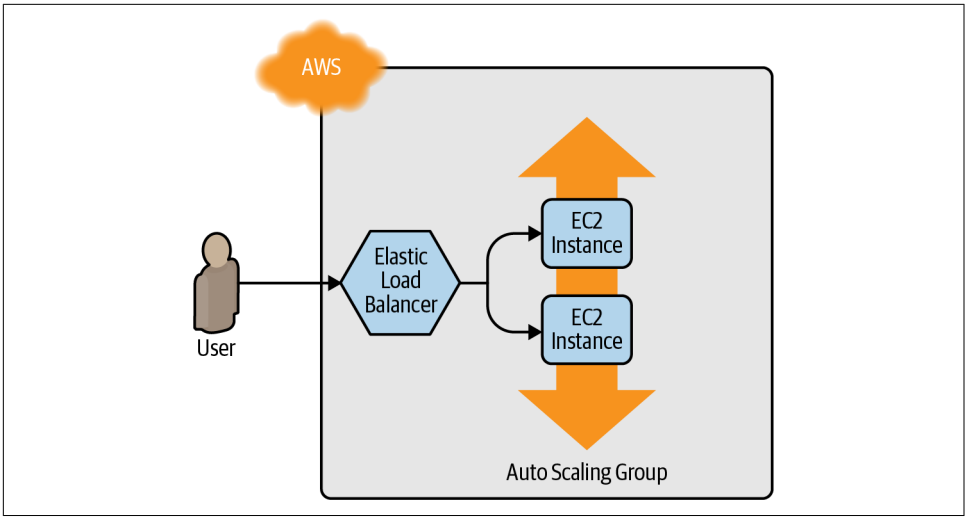


Figure 2-10. Use Amazon ELB to distribute traffic across the Auto Scaling Group.

AWS offers three 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 Open Systems Interconnection (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 (L7) and transport layer (L4) of the OSI model.

Most applications these days should use either the ALB or the NLB. Because 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.

The ALB consists of several parts, as shown in Figure 2-11:

Listener
: Listens on a specific port (e.g., 80) and protocol (e.g., HTTP).

Listener rule

Takes requests that come into a listener and sends those that match specific paths (e.g., `/foo` and `/bar`) or hostnames (e.g., `foo.example.com` and `bar.example.com`) to specific target groups.

Target groups

One or more servers that receive requests from the load balancer. The target group also performs health checks on these servers and sends requests only to healthy nodes.

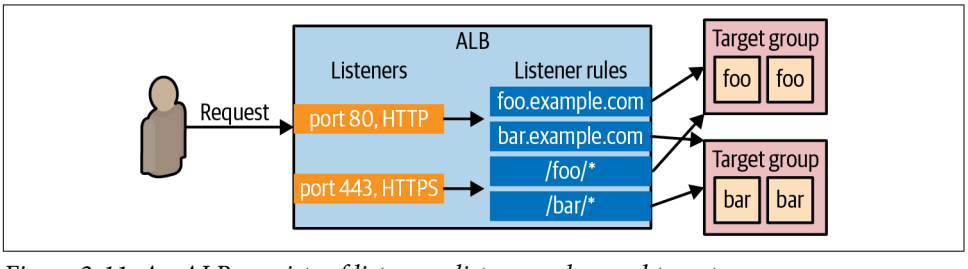


Figure 2-11. An ALB consists of listeners, listener rules, and target groups.

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_subnets.default.ids
}
```

Note that the `subnets` parameter configures the load balancer to use all the subnets in your Default VPC by using the `aws_subnets` data source.[16] AWS load balancers don't consist of a single server, but of multiple servers that can run in separate subnets (and, therefore, separate datacenters). AWS automatically scales the number of load balancer servers up and down based on traffic and handles 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"
```

16 To keep these examples simple, the EC2 Instances and ALB are running 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).

```
  # 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 that you can access the load balancer over HTTP, and allow outgoing requests on all ports so that 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_subnets.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
  }
}
```

This target group will health check your Instances by periodically sending an HTTP request to each Instance and will consider the Instance "healthy" only 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 can 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_subnets.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 considers an Instance unhealthy only if the AWS hypervisor says the VM is completely down or unreachable. The `"ELB"` health check is more robust, because it instructs the ASG to use the target group's health check to determine whether an Instance is healthy 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 {
    path_pattern {
      values = ["*"]
    }
  }

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

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

There's one last thing to do before you deploy 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 **`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 down this URL. It'll take a couple minutes for the Instances to boot and show up as healthy in the ALB. In the meantime, you can inspect what you've deployed. Open up the ASG section of the EC2 console, and you should see that the ASG has been created, as shown in Figure 2-12.

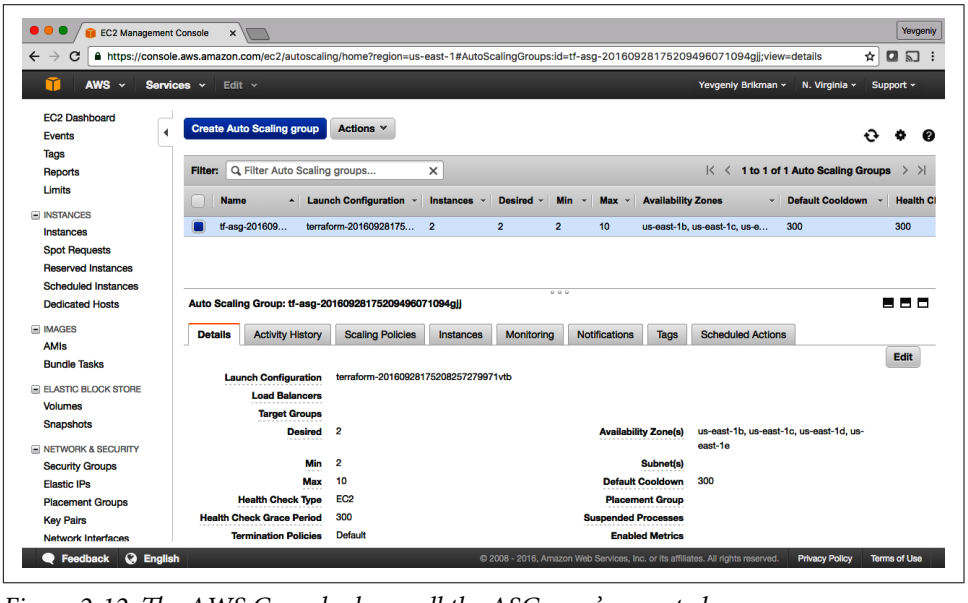


Figure 2-12. The AWS Console shows all the ASGs you've created.

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

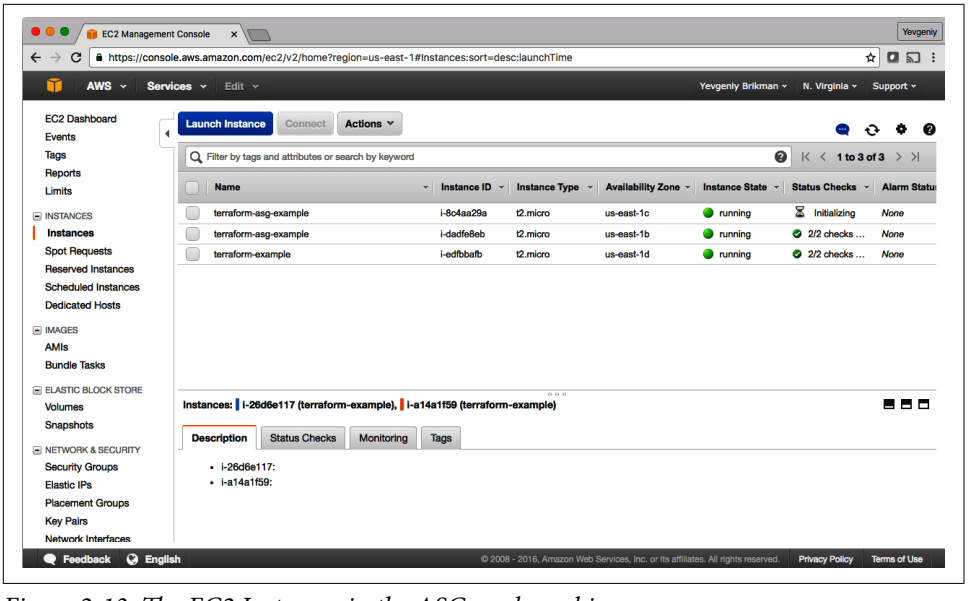


Figure 2-13. The EC2 Instances in the ASG are launching.

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

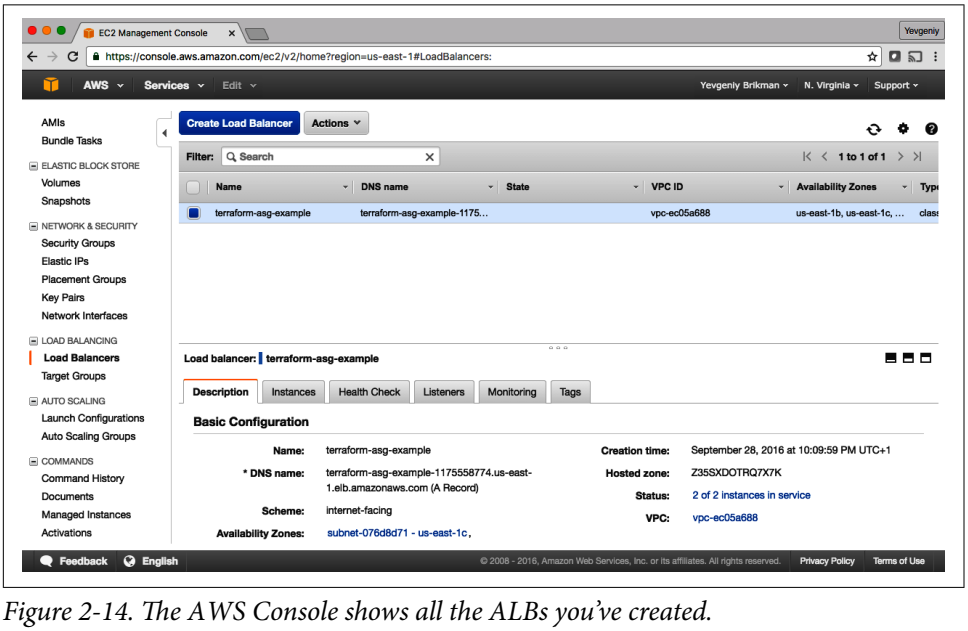


Figure 2-14. The AWS Console shows all the ALBs you've created.

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

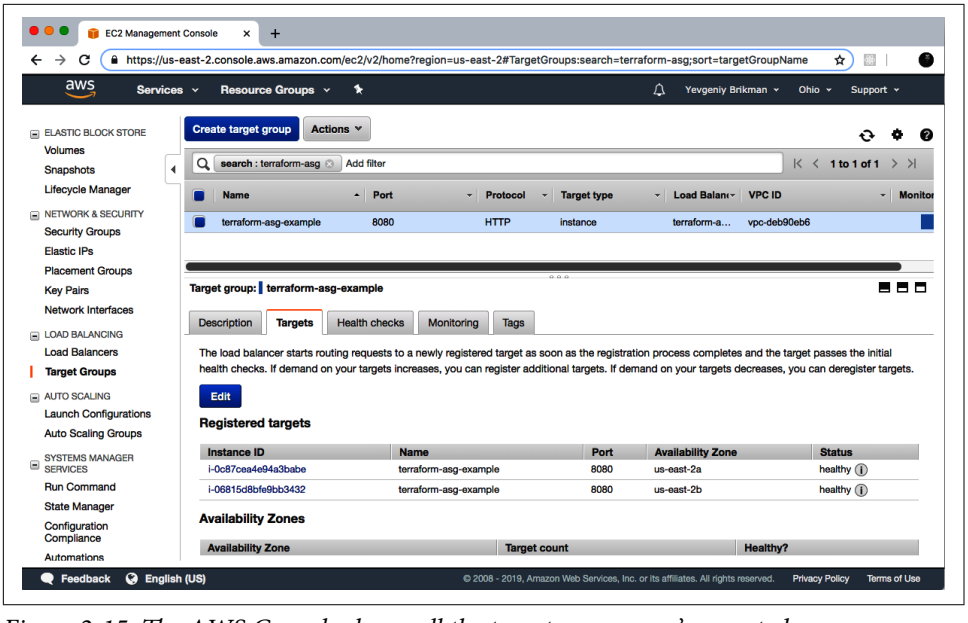


Figure 2-15. The AWS Console shows all the target groups you've created.

If you click 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 indicate "healthy" for both of them. This typically takes one to two minutes. When 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 access 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, clicking the Actions button at the top, and then 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, because the ALB will automatically detect that the Instance is down and stop routing to it. Even more interesting, 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 of the resources you created so that AWS doesn't charge you for them. Because 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 **`yes`** and hit Enter; Terraform will build the dependency graph and delete all of the resources in the correct 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 develop 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 might want to commit your latest changes to Git so that 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.

CHAPTER 3

How to Manage Terraform State

In Chapter 2, as you were using Terraform to create and update resources, you might 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
- State file isolation
  - Isolation via workspaces
  - Isolation via file layout
- The `terraform_remote_state` data source

Example Code

As a reminder, you can find all of the code examples in the book on GitHub.

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-0fb653ca2d3203ac1"
  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": "1.2.3",
  "serial": 1,
  "lineage": "86545604-7463-4aa5-e9e8-a2a221de98d2",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0fb653ca2d3203ac1",
            "availability_zone": "us-east-2b",
            "id": "i-0bc4bbe5b84387543",
            "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-0bc4bbe5b84387543`. 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 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 can 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). Although 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, and you shouldn't store plain text secrets in version control.

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 entire 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 and Terraform Enterprise.

Remote backends solve the three issues just listed:

Manual error

After 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 instruct 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 at rest of the state file. Moreover, those backends usually expose ways to configure access permissions (e.g., using IAM policies with an Amazon S3 bucket), so you can control who has access to your state files and the secrets they might contain. It would be better still if Terraform natively supported encrypting secrets within the state file, but these remote backends reduce most of the security concerns, given that at least the state file isn't stored in plain text 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 need 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 need to worry too much about data loss or outages.[1]
- It supports encryption, which reduces worries about storing sensitive data in state files. You still have to be very careful who on your team can access the S3 bucket, but at least the data will be encrypted at rest (Amazon S3 supports server-side encryption using AES-256) and in transit (Terraform uses TLS when talking to Amazon S3).
- It supports locking via DynamoDB. (More on this later.)
- 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 AWS Free Tier.[2]

To enable remote state storage with Amazon 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"
}
```

1 Learn more about S3's guarantees on the AWS website.

2 See pricing information for S3 on the AWS website.

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
  }
}
```

This code sets the following arguments:

`bucket`
: This is the name of the S3 bucket. Note that S3 bucket names must be *globally* unique among all AWS customers. Therefore, you will need to change the `bucket` parameter from `"terraform-up-and-running-state"` (which I already created) to your own name. Make sure to remember this name and take note of what AWS region you're using because 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.

Let's now add several extra layers of protection to this S3 bucket.

First, use the `aws_s3_bucket_versioning` resource to enable 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, which can be a useful fallback mechanism if something goes wrong:

```
# Enable versioning so you can see the full revision history of your
# state files
resource "aws_s3_bucket_versioning" "enabled" {
  bucket = aws_s3_bucket.terraform_state.id
  versioning_configuration {
    status = "Enabled"
  }
}
```

Second, use the `aws_s3_bucket_server_side_encryption_configuration` resource to turn server-side encryption on by default for all data written to this S3 bucket. This ensures that your state files, and any secrets they might contain, are always encrypted on disk when stored in S3:

```
# Enable server-side encryption by default
resource "aws_s3_bucket_server_side_encryption_configuration" "default" {
  bucket = aws_s3_bucket.terraform_state.id

  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}
```

Third, use the `aws_s3_bucket_public_access_block` resource to block all public access to the S3 bucket. S3 buckets are private by default, but as they are often used to serve static content—e.g., images, fonts, CSS, JS, HTML—it is possible, even easy, to make the buckets public. Since your Terraform state files may contain sensitive data and secrets, it's worth adding this extra layer of protection to ensure no one on your team can ever accidentally make this S3 bucket public:

```
# Explicitly block all public access to the S3 bucket
resource "aws_s3_bucket_public_access_block" "public_access" {
  bucket                  = aws_s3_bucket.terraform_state.id
  block_public_acls       = true
  block_public_policy     = true
  ignore_public_acls      = true
  restrict_public_buckets = true
}
```

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 AWS Free Tier.[3]

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"
```

3 Pricing information for DynamoDB is available on the AWS website.

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

Run `terraform init` to download the provider code, and then run `terraform apply` to deploy. After 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 resides 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 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 filepath within the S3 bucket where the Terraform state file should be written. You'll see a little later on why the preceding example code 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 that 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 instruct Terraform to store your state file in this S3 bucket, you're going to use the `terraform init` command again. This command not only can 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 multiple times:

```
$ 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 **`yes`**, you should see the following:

```
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 Management Console in your browser and clicking your bucket. You should see something similar to Figure 3-1.

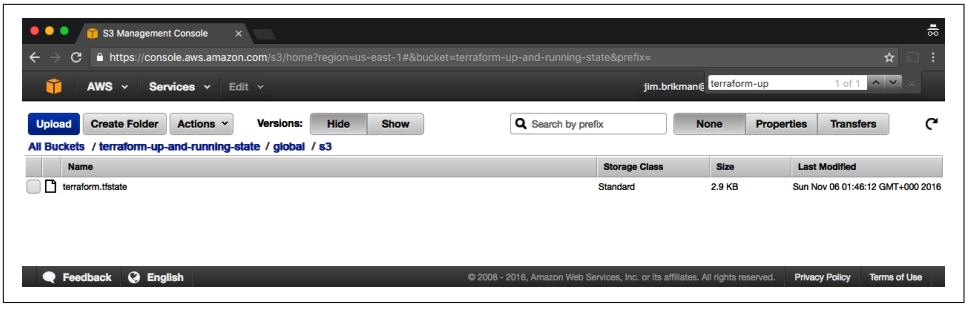


Figure 3-1. You can use the AWS Console to see how your state file is stored in an S3 bucket.

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.

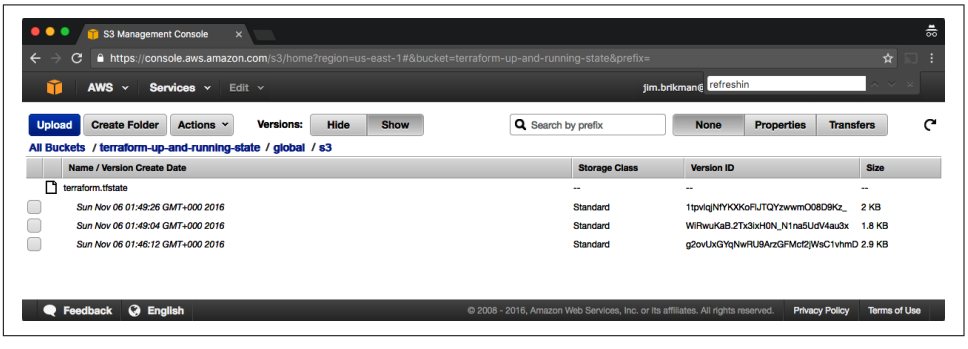


Figure 3-2. If you enable versioning for your S3 bucket, every change to the state file will be stored as a separate version.

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 and gotchas that you need 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 rerun `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 need to do it once (or once per AWS account if you have multiple accounts). After 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
  }
}
```

This means that you need 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 Chapters 4 and 8; 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 must 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-and-pastes *and* lots of manual changes is error prone, especially if you need to deploy and manage many Terraform modules across many environments.

One option for reducing copy-and-paste is to use *partial configurations*, 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, since 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 merges the partial configuration in *backend.hcl* with the partial configuration in your Terraform code to produce the full configuration used by your module. You can use the same *backend.hcl* file with all of your modules, which reduces duplication considerably; however, you'll still need to manually set a unique `key` value in every module.

Another option for reducing copy-and-paste is to use Terragrunt, an open source tool that tries to fill in a few gaps in Terraform. Terragrunt can help you keep your entire `backend` configuration DRY (Don't Repeat Yourself) 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 10.

State File Isolation

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 might 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.[4]

The whole point of having separate environments is that they are isolated from one another, 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

4 Here's a colorful example of what happens when you don't isolate Terraform state.

others, you should have "bulkheads" built into your Terraform design, as shown in Figure 3-3.

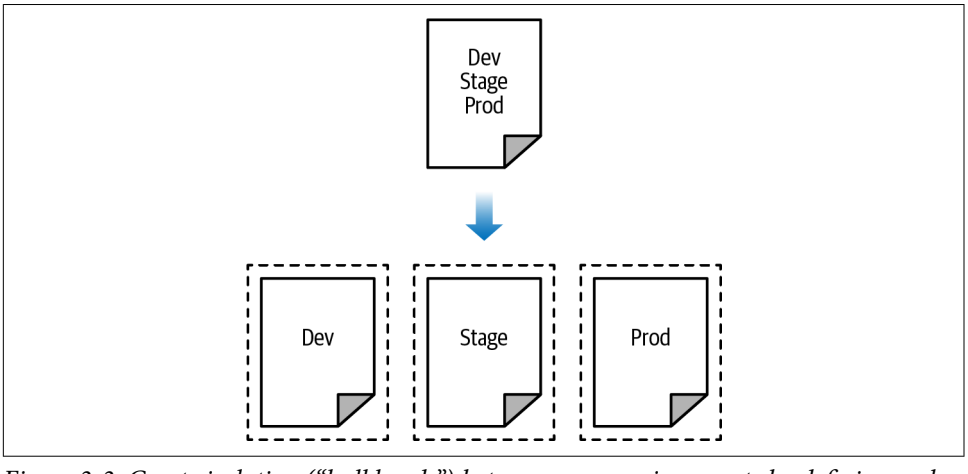


Figure 3-3. Create isolation ("bulkheads") between your environments by defining each environment in a separate Terraform configuration.

As Figure 3-3 illustrates, 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:

Isolation via workspaces
: Useful for quick, isolated tests on the same configuration

Isolation via file layout
: Useful for production use cases for which 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, 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-0fb653ca2d3203ac1"
```

```
  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` 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 identify 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.

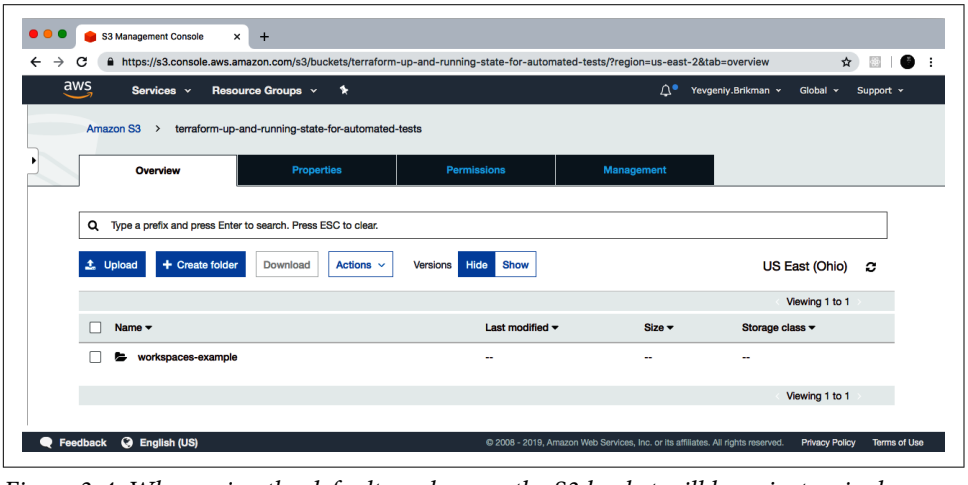


Figure 3-4. When using the default workspace, the S3 bucket will have just a single folder and state file in it.

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-0fb653ca2d3203ac1"
      + 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 one another, and because 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.
```

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.
```

Run `terraform apply` 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 by using 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; you should now see a new folder called *env:*, as shown in Figure 3-5.

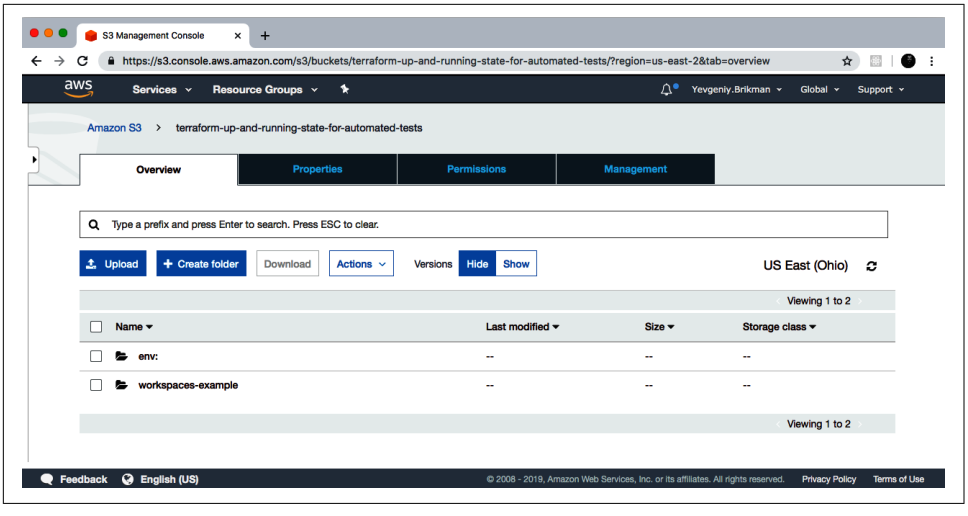


Figure 3-5. When using custom workspaces, the S3 bucket will have multiple folders and state files in it.

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

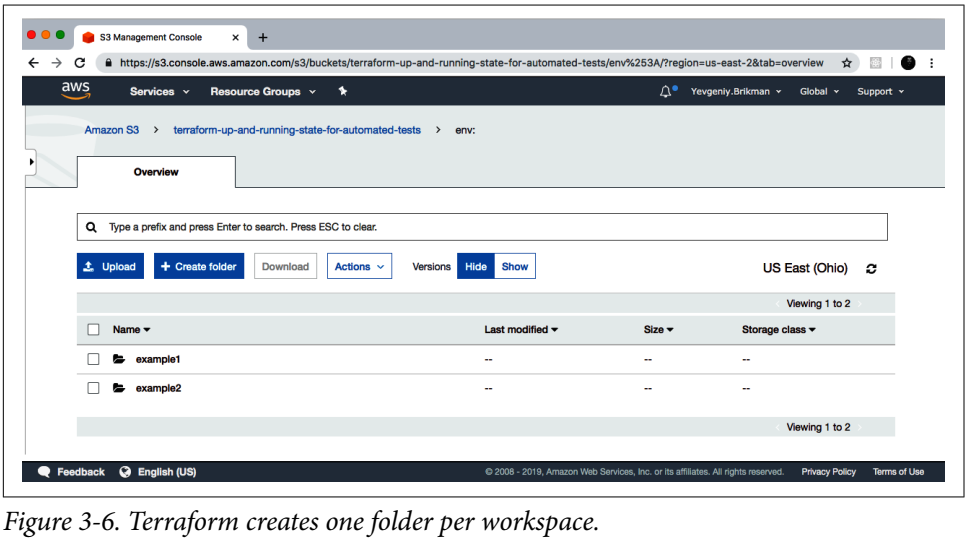


Figure 3-6. Terraform creates one folder per workspace.

Inside each of those workspaces, Terraform uses the `key` you specified in your `backend` configuration, so you should find an *example1/workspaces-example/terraform.tfstate* and 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-0fb653ca2d3203ac1"
  instance_type = terraform.workspace == "default" ? "t2.medium" : "t2.micro"
}
```

The preceding code 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:

- 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).
- 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 10 workspaces. This makes maintenance more difficult, because you don't have a good picture of your infrastructure.
- 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 deploy changes in the wrong one (e.g., accidentally running `terraform destroy` in a "production" workspace rather than a "staging" workspace), and because you must use the same authentication mechanism for all workspaces, you have no other layers of defense to protect against such errors.

Due to these drawbacks, workspaces are not a suitable mechanism for isolating one environment from another: e.g., isolating staging from production.[5] 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.

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 achieve full isolation between environments, you need to do the following:

- Put the Terraform configuration files for each environment into a separate folder. For example, all of 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*.
- 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 might 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, after 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 change it only once every few months, at most. On the other hand, you might 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.

5 The workspaces documentation makes this same exact point, but it's buried among several paragraphs of text, and as workspaces used to be called "environments," I find many users are still confused about when and when not to use workspaces.

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) within that environment. 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.

Figure 3-7. The typical file layout for a Terraform project uses separate folders for each environment and for each component within that environment.

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

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, CI server)

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 here are the typical ones:

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 reside 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
: Resources and data sources

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, although Terraform may not care about filenames, your teammates probably do. Using a consistent, predictable naming convention makes your code easier to browse: e.g., you'll always know where to look to find a variable, output, or resource.

Note that the preceding convention is the *minimum* convention you should follow, because in virtually all uses of Terraform, it's useful to be able to jump to the input variables, output variables, and resources very quickly, but you may want to go beyond this convention. Here are just a few examples:

dependencies.tf
: It's common to put all your data sources in a *dependencies.tf* file to make it easier to see what external things the code depends on.

providers.tf
: You may want to put your `provider` blocks into a *providers.tf* file so you can see, at a glance, what providers the code talks to and what authentication you'll have to provide.

main-xxx.tf
: If the *main.tf* file is getting really long because it contains a large number of resources, you could break it down into smaller files that group the resources in some logical way: e.g., *main-iam.tf* could contain all the IAM resources, *main-s3.tf* could contain all the S3 resources, and so on. Using the *main-* prefix makes it easier to scan the list of files in a folder when they are organized alphabetically, as all the resources will be grouped together. It's also worth noting that if you find yourself managing a very large number of resources and struggling to break them down across many files, that might be a sign that you should break your code into smaller modules instead, which is a topic I'll dive into in Chapter 4.

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

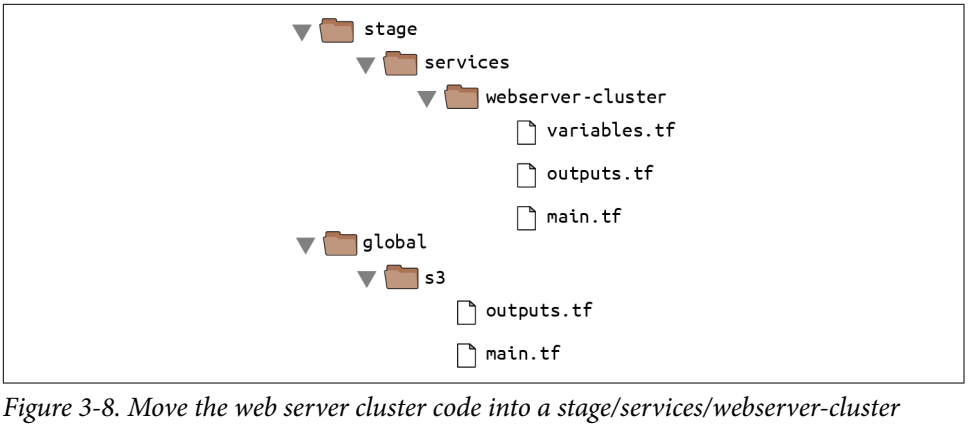


Figure 3-8. Move the web server cluster code into a stage/services/webserver-cluster folder to indicate that this is a testing or staging version of the web server.

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 that you don't miss the (hidden) *.terraform* folder when copying files to the new location so you don't need to reinitialize 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 move 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 has a number of advantages:

Clear code / environment layout
: It's easy to browse the code and understand exactly what components are deployed in each environment.

Isolation
: This layout 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.

In some ways, these advantages are drawbacks, too:

Working with multiple folders
: 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 of the components are in separate folders, then you need to run `terraform apply` separately in each one.

  Solution: If you use Terragrunt, you can run commands across multiple folders concurrently using the `run-all` command.

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.

Solution: You won't actually need to copy and paste all of that code! In Chapter 4, you'll see how to use Terraform modules to keep all of this code DRY.

Resource dependencies

Breaking the code into multiple folders makes it more difficult to use resource dependencies. If your app code was defined in the same Terraform configuration files as the database code, that app code 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.

Solution: One option is to use `dependency` blocks in Terragrunt, as you'll see in Chapter 10. Another option is to use the `terraform_remote_state` data source, as described in the next section.

The terraform_remote_state Data Source

In Chapter 2, you used data sources to fetch read-only information from AWS, such as the `aws_subnets` 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.

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

You might not want to define the MySQL database in the same set of configuration files as the web server cluster, because 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.

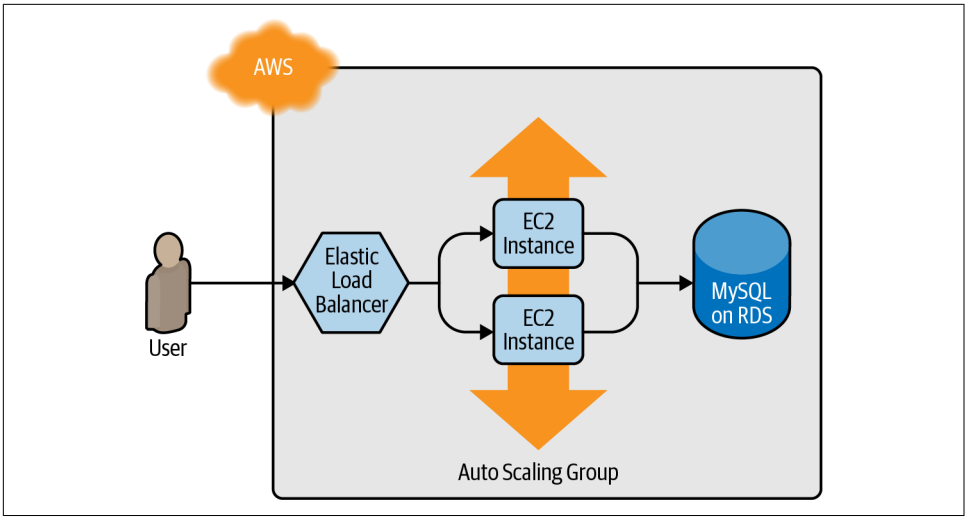


Figure 3-9. The web server cluster communicates with MySQL, which is deployed on top of Amazon RDS.

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.

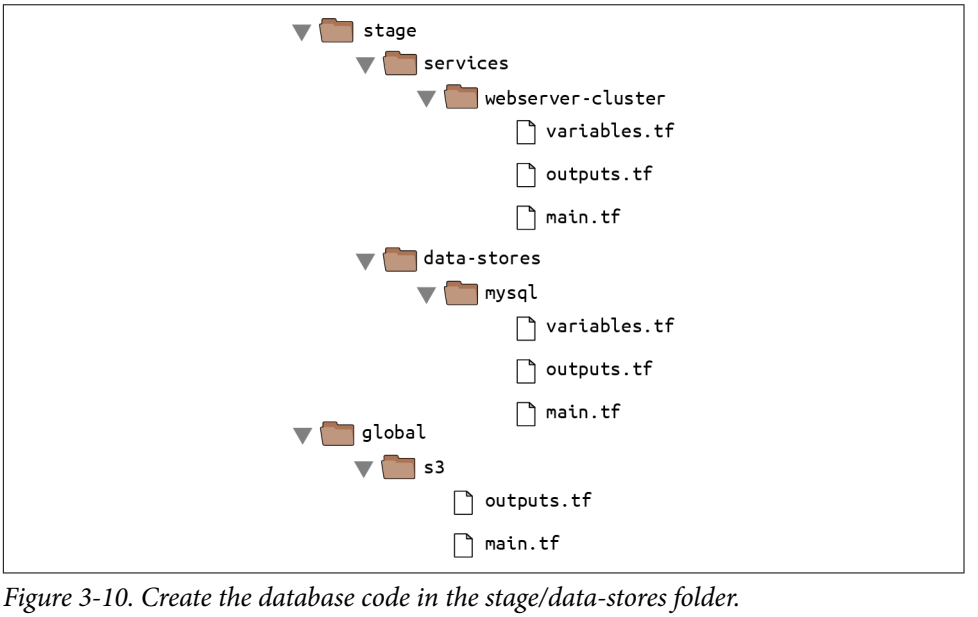


Figure 3-10. Create the database code in the stage/data-stores folder.

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"
  skip_final_snapshot = true
  db_name             = "example_database"

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

At the top of the file, you see the typical `provider` block, but just below that is a new resource: `aws_db_instance`. This resource creates a database in RDS with the following settings:

- MySQL as the database engine.
- 10 GB of storage.
- A `db.t2.micro` Instance, which has one virtual CPU, 1 GB of memory, and is part of the AWS Free Tier.
- The final snapshot is disabled, as this code is just for learning and testing (if you don't disable the snapshot, or don't provide a name for the snapshot via the `final_snapshot_identifier` parameter, `destroy` will fail).

Note that two of the parameters that you must pass to the `aws_db_instance` resource are the master username and master password. Because these are secrets, you should not put them directly into your code in plain text! In Chapter 6, I'll discuss a variety of options for how to securely handle secrets with Terraform. For now, let's use an option that avoids storing any secrets in plain text and is easy to use: you store your secrets, such as database passwords, outside of Terraform (e.g., in a password manager such as 1Password, LastPass, or macOS Keychain), and you pass those secrets into Terraform via environment variables.

To do that, declare variables called `db_username` and `db_password` in *stage/data-stores/mysql/variables.tf*:

```
variable "db_username" {
  description = "The username for the database"
  type        = string
  sensitive   = true
}
```

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

First, note that these variables are marked with `sensitive = true` to indicate they contain secrets. This ensures Terraform won't log the values when you run `plan` or `apply`. Second, note that these variables do not have a `default`. This is intentional. You should not store your database credentials or any sensitive information in plain text. Instead, you'll set these variables using environment variables.

Before doing that, let's finish the code. First, pass the two new input variables through to the `aws_db_instance` resource:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = "example_database"

  username = var.db_username
  password = var.db_password
}
```

Next, 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
  }
}
```

Finally, add two output variables in *stage/data-stores/mysql/outputs.tf* to return the database's address and port:

```
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"
}
```

You're now ready to pass in the database username and password using environment variables. 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_username` and `db_password` input variables, here is how you can set the `TF_VAR_db_username` and `TF_VAR_db_password` environment variables on Linux/Unix/macOS systems:

```
$ export TF_VAR_db_username="(YOUR_DB_USERNAME)"
$ export TF_VAR_db_password="(YOUR_DB_PASSWORD)"
```

And here is how you do it on Windows systems:

```
$ set TF_VAR_db_username="(YOUR_DB_USERNAME)"
$ set TF_VAR_db_password="(YOUR_DB_PASSWORD)"
```

Run `terraform init` and `terraform apply` to create the database. Note that Amazon RDS can take roughly 10 minutes to provision even a small database, so be patient. After `apply` completes, you should see the outputs in the terminal:

```
$ terraform apply

(...)

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

Outputs:

address = "terraform-up-and-running.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*.

If you go back to your web server cluster code, you can get the web server to read those outputs from the database's 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.

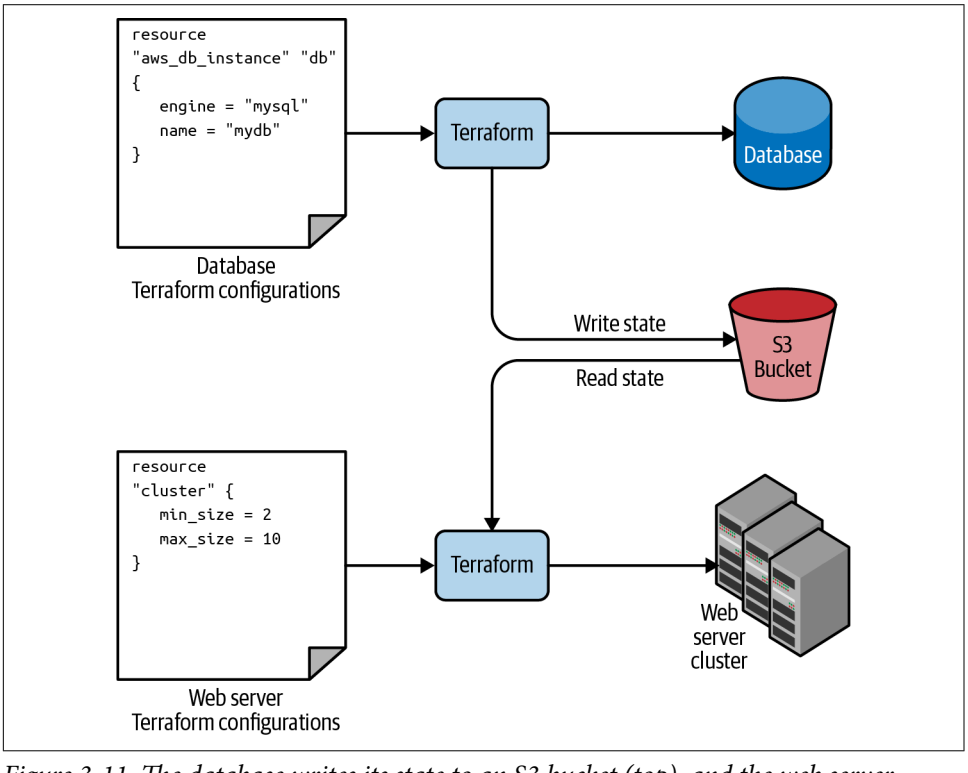


Figure 3-11. The database writes its state to an S3 bucket (top), and the web server cluster reads that state from the same bucket (bottom).

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 of 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 growing longer, defining it inline is becoming messier and messier. In general, embedding one programming language (Bash) inside another (Terraform) makes it more difficult to maintain each one, so let's pause here for a moment to externalize the Bash script. To do that, you can use the `templatefile` built-in function.

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`.[6] 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 need to worry about accidentally changing infrastructure or state.

There are a number of other built-in functions that you can use to manipulate strings, numbers, lists, and maps.[7] One of them is the `templatefile` function:

```
templatefile(<PATH>, <VARS>)
```

This function reads the file at `PATH`, renders it as a template, and returns the result as a string. When I say "renders it as a template," what I mean is that the file at `PATH` can use the string interpolation syntax in Terraform (`${...}`), and Terraform will render the contents of that file, filling variable references from `VARS`.

6 You can find documentation for the `sprintf` syntax on the Go website.

7 The full list of built-in functions is available on the Terraform website.

To see this in action, put the contents of the User Data script into the file *stage/services/webserver-cluster/user-data.sh* 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, except the only variables it has access to are those you pass in via the second parameter to `templatefile` (as you'll see shortly), so you don't need any prefix to access them: for example, 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.

The final step is to update the `user_data` parameter of the `aws_launch_configuration` resource to call the `templatefile` function and pass in the variables it needs as a map:

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0fb653ca2d3203ac1"
  instance_type   = "t2.micro"
  security_groups = [aws_security_group.instance.id]

  # Render the User Data script as a template
  user_data = templatefile("user-data.sh", {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  })

  # Required when using a launch configuration with an auto scaling group.
  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.

Congrats, 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 that they could be used by your database library (e.g., ActiveRecord) to communicate with the database.

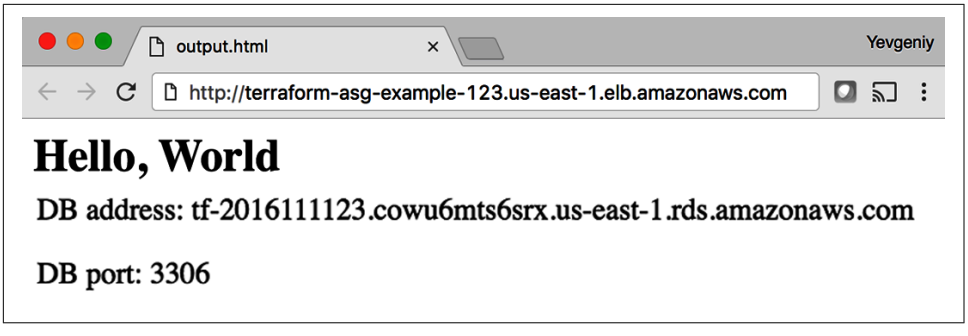


Figure 3-12. The web server cluster can programmatically access the database address and port.

Conclusion

The reason you need to put so much thought into isolation, locking, and state is that infrastructure as code (IaC) has different trade-offs than normal coding. When you're writing code for a typical app, most bugs are relatively minor and break only a small part of a single app. When you're writing code that controls your infrastructure, bugs tend to be more severe, given that 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.[8]

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.

8 For more information on software safety mechanisms, see Agility Requires Safety.

CHAPTER 4

How to Create Reusable Infrastructure with Terraform Modules

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

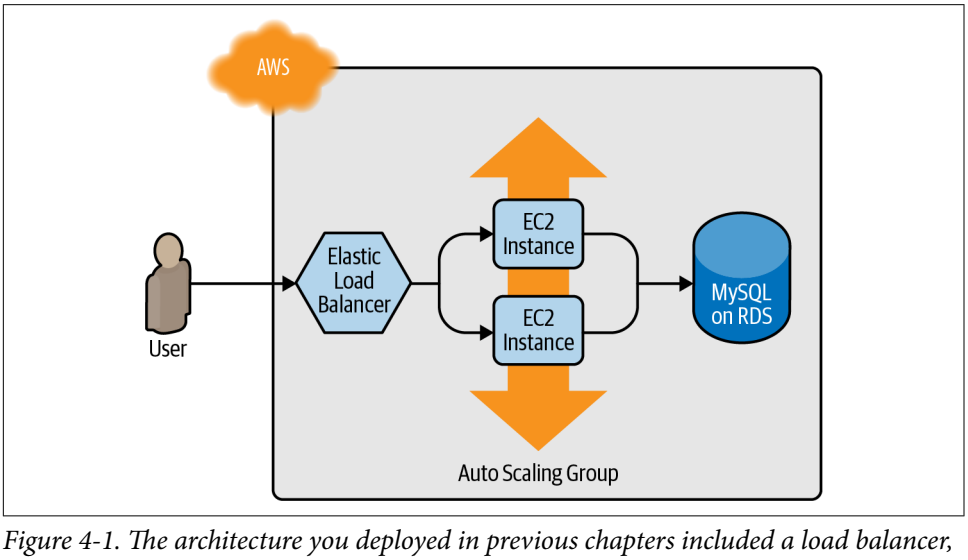


Figure 4-1. The architecture you deployed in previous chapters included a load balancer, web server cluster, and database.

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 might run slightly fewer/smaller servers in staging to save money.

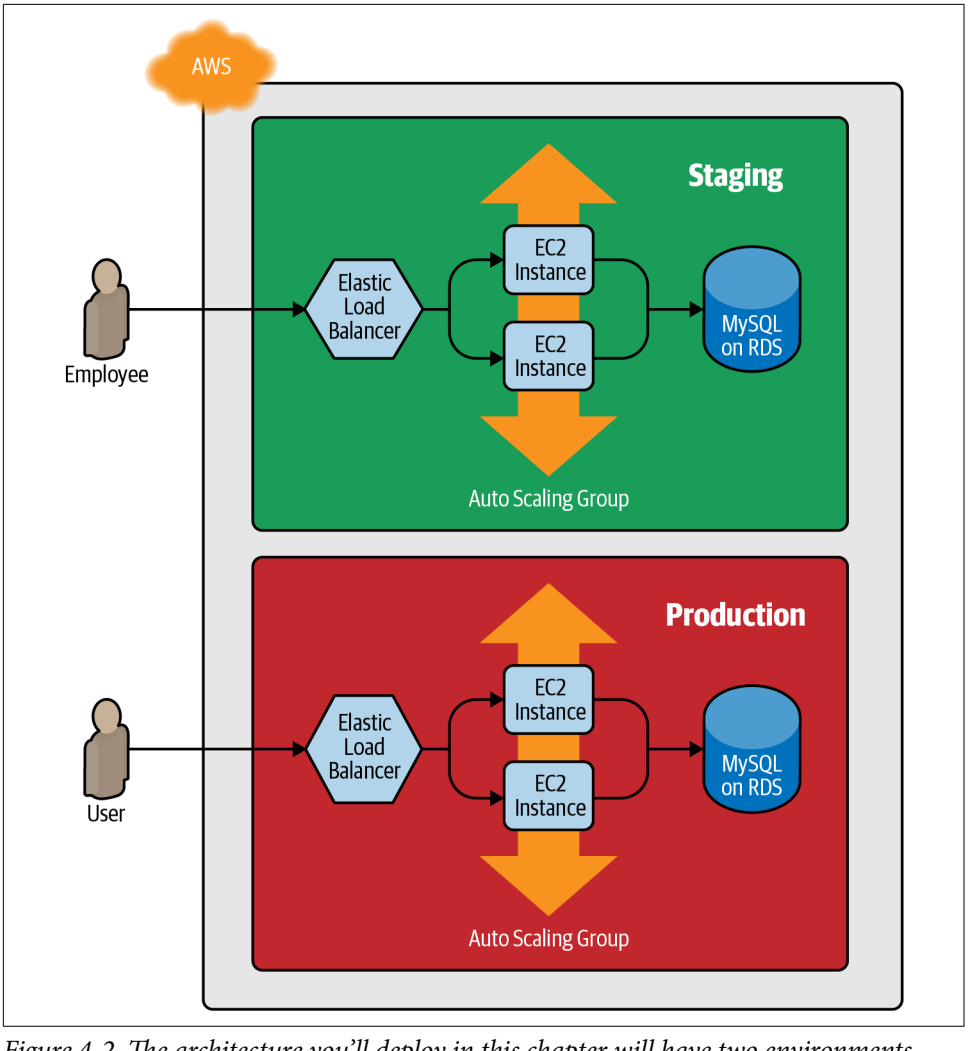


Figure 4-2. The architecture you'll deploy in this chapter will have two environments, each with its own load balancer, web server cluster, and database.

How do you add this production environment without having to copy and 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:

```
# Define the function in one place
def example_function()
  puts "Hello, World"
end

# Use the function in multiple other places
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 copied and 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.

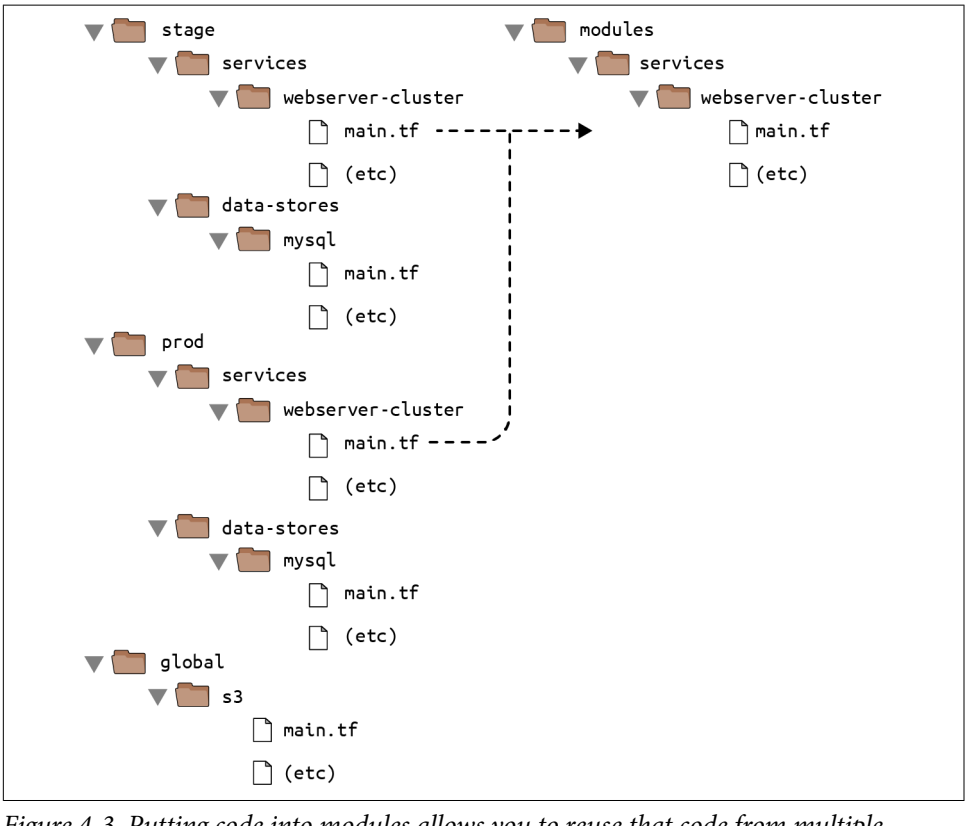


Figure 4-3. Putting code into modules allows you to reuse that code from multiple environments.

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, using modules that you find online, and 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, you can find all of the code examples in the book on GitHub.

Module Basics

A Terraform module is very simple: any set of Terraform configuration files in a folder is a module. All of the configurations you've written so far have technically been modules, although not particularly interesting ones, since you deployed them directly: if you run `apply` directly on a module, it's referred to as a *root module*. To see what modules are really capable of, you need to create a *reusable module*, which is a module that is meant to be used within other modules.

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 that you created earlier. Next, create a new top-level folder called *modules*, and move all of 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 only in root modules and not in reusable modules (you'll learn a lot more about working with providers in Chapter 7).

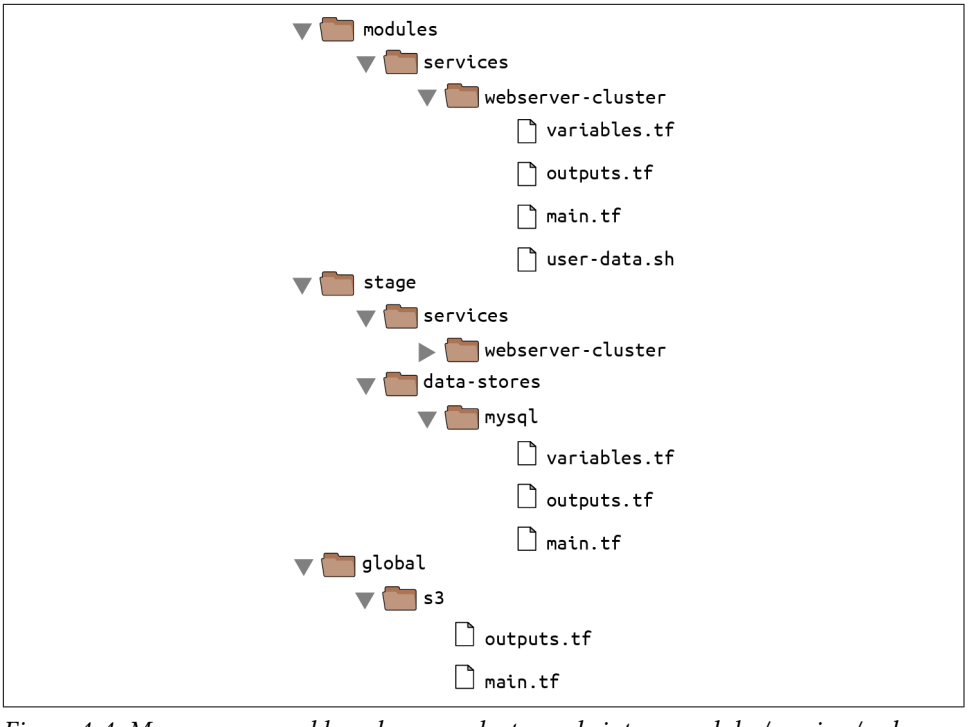


Figure 4-4. Move your reusable web server cluster code into a modules/services/web server-cluster folder.

You can now make use of this module in the staging environment. Here's the syntax for using a module:

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

  [CONFIG ...]
}
```

where `NAME` is an identifier you can use throughout the Terraform code to refer to this module (e.g., `webserver_cluster`), `SOURCE` is the path where the module code can be found (e.g., *modules/services/webserver-cluster*), and `CONFIG` consists of 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 that involves minimal duplication. 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 installs providers, it configures your backends, and it downloads modules, all in one handy command.

Before you run the `apply` command on this code, be aware that there is a problem with the `webserver-cluster` module: all of the names are hardcoded. That is, the name of the security groups, ALB, and other resources are all hardcoded, so if you use this module more than once in the same AWS account, you'll get name conflict errors. Even the details for how to read the database's state are hardcoded 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 hardcoded to look at the staging environment.

To fix these issues, you need to add configurable inputs to the `webserver-cluster` module so that 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:

```
# A function with two input parameters
def example_function(param1, param2)
  puts "Hello, #{param1} #{param2}"
end

# Pass two input parameters to the function
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 hardcoded 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 also set these variables in the production environment in *prod/services/webserver-cluster/main.tf* but to different values that correspond to that environment:

```
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"
}
```

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 by 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.

So far, you've added input variables for the name and database remote state, but you may want to make other parameters configurable in your module, 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-0fb653ca2d3203ac1"
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]

  user_data = templatefile("user-data.sh", {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  })

  # Required when using a launch configuration with an auto scaling group.
  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, respectively:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnets.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` (be aware that 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 copied and 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 copied and pasted in several places throughout the module. Having these magical values hardcoded in multiple places makes the code more difficult 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 might 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 visible only 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 the `port` parameter of 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
    }
  }
}
```

Similarly, update virtually all the parameters in 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 ASGs 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 a *scheduled action*, 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 a scheduled action to increase the number of servers at 9 a.m. and decrease it at 5 p.m.

If you define the scheduled action in the `webserver-cluster` module, it would apply to both staging and production. Because 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 lets you move the scheduled action into the `webserver-cluster` module).

To define a scheduled action, 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:

```
# A function that returns a value
def example_function(param1, param2)
  return "Hello, #{param1} #{param2}"
end

# Call the function and get the return value
return_value = example_function("foo", "bar")
```

In Terraform, a module can also return values. Again, you do this 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 might 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 `templatefile` built-in function to read this file from disk. The catch with the `templatefile` function is that the filepath you use must be a relative path (you don't want to use absolute file paths, as your Terraform code may run on many different computers, each with a different disk layout)—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 `templatefile` 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 `templatefile` function in the root module), but that won't work when you're using `templatefile` in a module that's defined in a separate folder (a reusable module).

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` when calling the `templatefile` function in *modules/services/webserver-cluster/main.tf*:

```
user_data = templatefile("${path.module}/user-data.sh", {
  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. An *inline block* is an argument you set within a resource of the format:

```
resource "xxx" "yyy" {
  <NAME> {
    [CONFIG...]
  }
}
```

where `NAME` is the name of the inline block (e.g., `ingress`) and `CONFIG` consists of one or more arguments that are specific to that inline block (e.g., `from_port` and `to_port`). For example, with the `aws_security_group_resource`, you can define ingress and egress rules using either inline blocks (e.g., `ingress { … }`) or separate `aws_security_group_rule` resources.

If you try to use a mix of *both* inline blocks and separate resources, due to how Terraform is designed, you will get errors where the configurations conflict and overwrite one another. Therefore, you must use one or the other. Here's my advice: when creating a module, you should always prefer using separate resources.

The advantage of using separate resources is that they can be added anywhere, whereas an inline block can only be added within the module that creates a resource. So using solely separate resources makes your module more flexible and configurable.

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

```
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
  }
}
```

With these inline blocks, a user of this module has no way to add additional ingress or egress rules from outside the module. To make your module more flexible, you should change it 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
}
```

You should also 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, if you needed to expose an extra port in just the staging environment (e.g., for testing), you can do this by adding an `aws_security_group_rule` resource to *stage/services/webserver-cluster/main.tf*:

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

  # (parameters hidden for clarity)
}

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 the following:

- `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, as well as 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 of the resources deploy into the same VPC. This means that a server in the staging environment can communicate with 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 can be affected, too. Second, if an attacker gains 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 have access to not only 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 might even run each environment in a totally separate AWS account.

Module Versioning

If both your staging and production environment are pointing to the same module folder, 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 more difficult 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.

In all of the module examples you've seen so far, whenever you used a module, you set the `source` parameter of the module to a local filepath. In addition to file paths, Terraform supports other types of module sources, such as Git URLs, Mercurial URLs, and arbitrary HTTP URLs.[1]

1 For the full details on source URLs, see the Terraform website.

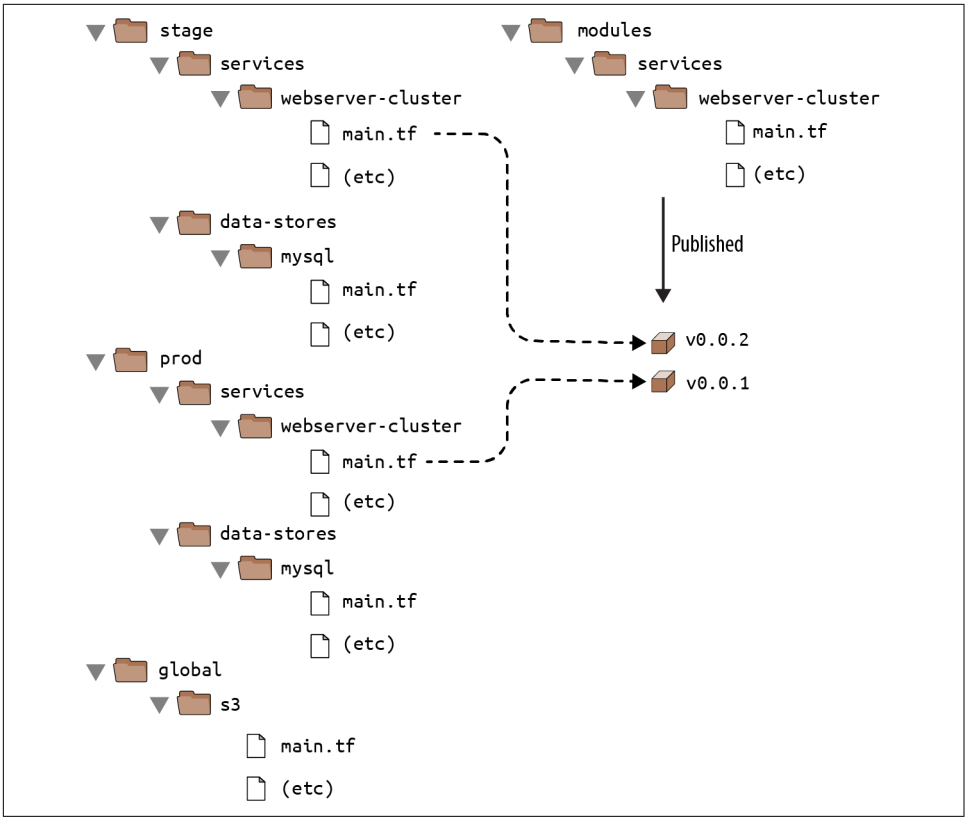


Figure 4-5. By versioning your modules, you can use different versions in different environments: e.g., v0.0.1 in prod and v0.0.2 in stage.

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 now looks something like Figure 4-6.

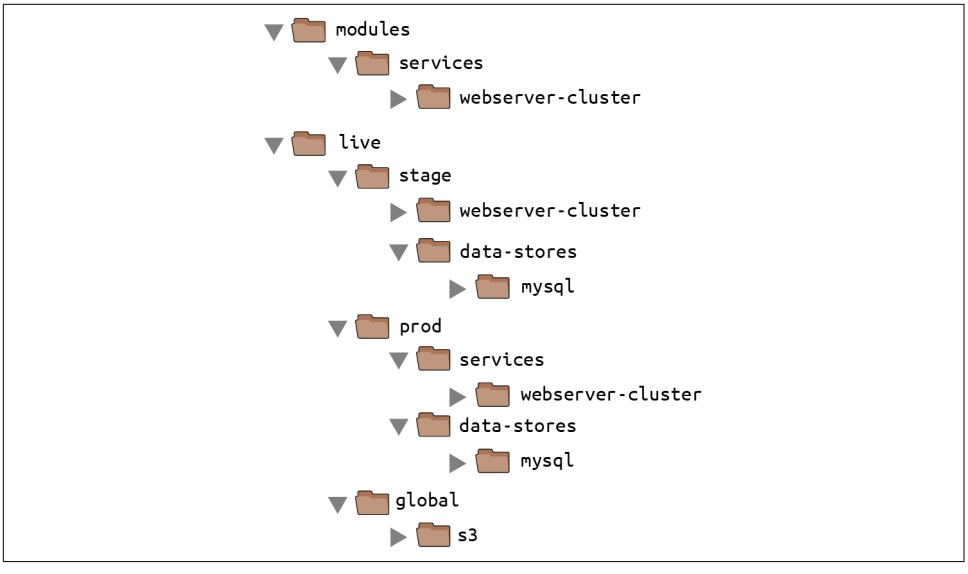


Figure 4-6. You should store reusable, versioned modules in one repo (modules) and the configuration for your live environments in another repo (live).

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 main
```

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 following Git URL is required):

```
module "webserver_cluster" {
  source = "github.com/foo/modules//services/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=v0.3.0"
```

The `ref` parameter allows you to specify a particular 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 following:

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

Semantic versioning gives you a way to communicate to users of your module what kinds of changes you've made and the implications of upgrading.

Because you've updated your Terraform code to use a versioned module URL, you need to instruct Terraform to download the module code by rerunning `terraform init`:

```
$ terraform init
Initializing modules...
Downloading git@github.com:brikis98/terraform-up-and-running-code.git?ref=v0.3.0
```

```
for webserver_cluster...

(...)
```

This time, you can see that Terraform downloads the module code from Git rather than your local filesystem. After 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 need to give Terraform a way to authenticate to that Git repository. I recommend using SSH auth so that you don't need to hardcode the credentials for your repo in the code itself. With SSH authentication, each developer can create an SSH key, associate it with their Git user, 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@github.com:<OWNER>/<REPO>.git//<PATH>?ref=<VERSION>
```

For example:

```
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:acme/modules.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 want 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 main
```

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
```

2 See the GitHub documentation for a nice guide on working with SSH keys.

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 = "github.com/foo/modules//services/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 = "github.com/foo/modules//services/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
}
```

After 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, because it has no effect on the real users of your production environment. Fix the bug, release a new version, and repeat the entire 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, because 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, publish a new version, and rerun `init` 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 because developers will be able to reuse entire pieces of proven, tested, and 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 might want to use your module to deploy a single Instance of their microservice with no load balancer, whereas another might 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.

CHAPTER 5

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

Terraform is a declarative language. As discussed in Chapter 1, IaC 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, because 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 meta-parameter `count`, `for_each` and `for` expressions, a ternary operator, a lifecycle block called `create_before_destroy`, and 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, you can find all of the code examples in the book on GitHub.

Loops

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

- `count` parameter, to loop over resources and modules
- `for_each` expressions, to loop over resources, inline blocks within a resource, and modules
- `for` expressions, to loop over lists and maps
- `for` string directive, to 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 AWS Identity and Access Management (IAM) user by clicking around the 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 want 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`. `count` is Terraform's oldest, simplest, and most limited iteration construct: all it does is define how many copies of the resource to create. Here's how you use `count` to create three IAM users:

```
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" {
      + name          = "neo.0"
      (...)
    }

  # aws_iam_user.example[1] will be created
  + resource "aws_iam_user" "example" {
      + name          = "neo.1"
      (...)
    }

  # aws_iam_user.example[2] will be created
  + resource "aws_iam_user" "example" {
      + name          = "neo.2"
```

```
      (...)
    }

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 the following:

Array lookup syntax

The syntax for looking up members of an array in Terraform is similar to most other programming languages:

```
ARRAY[<INDEX>]
```

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

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

The `length` *function*

Terraform has a built-in function called `length` that has the following syntax:

```
length(<ARRAY>)
```

As you can probably guess, the `length` function returns the number of items in the given `ARRAY`. It also works with strings and maps.

Putting these together, you get the following:

```
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, readable name:

```
Terraform will perform the following actions:

  # aws_iam_user.example[0] will be created
  + resource "aws_iam_user" "example" {
      + name          = "neo"
      (...)
    }

  # aws_iam_user.example[1] will be created
  + resource "aws_iam_user" "example" {
      + name          = "trinity"
      (...)
    }

  # aws_iam_user.example[2] will be created
  + resource "aws_iam_user" "example" {
      + name          = "morpheus"
      (...)
    }

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

Note that after you've used `count` on a resource, it becomes an array of resources rather than just one resource. Because `aws_iam_user.example` is now an array of IAM users, instead of using the standard syntax to read an attribute from that resource (`<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>`), you must specify which IAM user you're interested in by specifying its index in the array using the same array lookup syntax:

```
<PROVIDER>_<TYPE>.<NAME>[INDEX].ATTRIBUTE
```

For example, if you want to provide the Amazon Resource Name (ARN) of the first IAM user in the list as an output variable, you would need to do the following:

```
output "first_arn" {
  value       = aws_iam_user.example[0].arn
  description = "The ARN for the first user"
}
```

If you want the ARNs of *all* of 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 `first_arn` output will contain just the ARN for `neo`, whereas the `all_arns` output will contain the list of all ARNs:

```
$ terraform apply

(...)

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

first_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",
]
```

As of Terraform 0.13, the `count` parameter can also be used on modules. For example, imagine you had a module at *modules/landing-zone/iam-user* that can create a single IAM user:

```
resource "aws_iam_user" "example" {
  name = var.user_name
}
```

The username is passed into this module as an input variable:

```
variable "user_name" {
  description = "The user name to use"
  type        = string
}
```

And the module returns the ARN of the created IAM user as an output variable:

```
output "user_arn" {
  value       = aws_iam_user.example.arn
  description = "The ARN of the created IAM user"
}
```

You could use this module with a `count` parameter to create three IAM users as follows:

```
module "users" {
  source = "../../../modules/landing-zone/iam-user"

  count     = length(var.user_names)
  user_name = var.user_names[count.index]
}
```

The preceding code uses `count` to loop over this list of usernames:

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

And it outputs the ARNs of the created IAM users as follows:

```
output "user_arns" {
  value       = module.users[*].user_arn
  description = "The ARNs of the created IAM users"
}
```

Just as adding `count` to a resource turns it into an array of resources, adding `count` to a module turns it into an array of modules.

If you run `apply` on this code, you'll get the following output:

```
$ terraform apply

(...)

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

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

So, as you can see, `count` works more or less identically with resources and with modules.

Unfortunately, `count` has two limitations that significantly reduce its usefulness. First, although you can use `count` to loop over an entire resource, you can't use `count` within a resource to loop over inline blocks.

For example, consider how 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_subnets.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` requires you to create a new inline block with values for `key`, `value`, and `propagate_at_launch`. The preceding code hardcodes a single tag, but you might want to allow users to pass in custom tags. You might be tempted to try to use the `count` parameter to loop over these tags and generate dynamic inline `tag` blocks, but unfortunately, using `count` within an inline block is not supported.

The second limitation with `count` is what happens when you try to change its value. Consider the list of IAM users you created earlier:

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

Imagine that you removed `"trinity"` from this list. What happens when you run `terraform plan`?

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_iam_user.example[1] will be updated in-place
  ~ resource "aws_iam_user" "example" {
        id            = "trinity"
      ~ name          = "trinity" -> "morpheus"
    }

  # aws_iam_user.example[2] will be destroyed
  - resource "aws_iam_user" "example" {
      - id            = "morpheus" -> null
      - name          = "morpheus" -> null
    }

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

Wait a second, that's probably not what you were expecting! Instead of just deleting the `"trinity"` IAM user, the `plan` output is indicating that Terraform wants to rename the `"trinity"` IAM user to `"morpheus"` and delete the original `"morpheus"` user. What's going on?

When you use the `count` parameter on a resource, that resource becomes an 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 usernames, Terraform's internal representation of these IAM users looks something like this:

```
aws_iam_user.example[0]: neo
aws_iam_user.example[1]: trinity
aws_iam_user.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_iam_user.example[0]: neo
aws_iam_user.example[1]: morpheus
```

Notice how `"morpheus"` has moved from index 2 to index 1. Because it 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." In other words, 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 re-create those resources again from scratch. Ouch. The end result, of course, is exactly what you requested (i.e., two IAM users named `"morpheus"` and `"neo"`), but deleting resources is probably not how you want to get there, as you may lose availability (you can't use the IAM user during the `apply`), and, even worse, you may lose data (if the resource you're deleting is a database, you may lose all the data in it!).

To solve these two limitations, Terraform 0.12 introduced `for_each` expressions.

Loops with for_each Expressions

The *for_each* expression allows you to loop over lists, sets, and maps to create (a) multiple copies of an entire resource, (b) multiple copies of an inline block within a resource, or (c) multiple copies of a module. Let's first walk through how to use `for_each` to create multiple copies of a resource.

The syntax looks like this:

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  for_each = <COLLECTION>

  [CONFIG ...]
}
```

where `COLLECTION` is a set or map to loop over (lists are not supported when using `for_each` on a resource) and `CONFIG` consists of one or more arguments that are specific to that resource. Within `CONFIG`, you can use `each.key` and `each.value` to access the key and value of the current item in `COLLECTION`.

For example, here's how you can create the same three IAM users using `for_each` on a resource:

```
resource "aws_iam_user" "example" {
  for_each = toset(var.user_names)
  name     = each.value
}
```

Note the use of `toset` to convert the `var.user_names` list into a set. This is because `for_each` supports sets and maps only when used on a resource. When `for_each` loops over this set, it makes each username available in `each.value`. The username will also be available in `each.key`, though you typically use `each.key` only with maps of key-value pairs.

Once you've used `for_each` on a resource, it becomes a map of resources, rather than just one resource (or an array of resources as with `count`). To see what that means, remove the original `all_arns` and `first_arn` output variables, and add a new `all_users` output variable:

```
output "all_users" {
  value = aws_iam_user.example
}
```

Here's what happens when you run `terraform apply`:

```
$ terraform apply

(...)

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

all_users = {
  "morpheus" = {
    "arn" = "arn:aws:iam::123456789012:user/morpheus"
    "force_destroy" = false
    "id" = "morpheus"
    "name" = "morpheus"
    "path" = "/"
    "tags" = {}
  }
  "neo" = {
    "arn" = "arn:aws:iam::123456789012:user/neo"
    "force_destroy" = false
    "id" = "neo"
    "name" = "neo"
    "path" = "/"
    "tags" = {}
  }
  "trinity" = {
```

```
    "arn" = "arn:aws:iam::123456789012:user/trinity"
    "force_destroy" = false
    "id" = "trinity"
    "name" = "trinity"
    "path" = "/"
    "tags" = {}
  }
}
```

You can see that Terraform created three IAM users and that the `all_users` output variable contains a map where the keys are the keys in `for_each` (in this case, the usernames) and the values are all the outputs for that resource. If you want to bring back the `all_arns` output variable, you'd need to do a little extra work to extract those ARNs using the `values` built-in function (which returns just the values from a map) and a splat expression:

```
output "all_arns" {
  value = values(aws_iam_user.example)[*].arn
}
```

This gives you the expected output:

```
$ terraform apply

(...)

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

Outputs:

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

The fact that you now have a map of resources with `for_each` rather than an array of resources as with `count` is a big deal, because it allows you to remove items from the middle of a collection safely. For example, if you again remove `"trinity"` from the middle of the `var.user_names` list and run `terraform plan`, here's what you'll see:

```
$ terraform plan

Terraform will perform the following actions:

  # aws_iam_user.example["trinity"] will be destroyed
  - resource "aws_iam_user" "example" {
      - arn           = "arn:aws:iam::123456789012:user/trinity" -> null
      - name          = "trinity" -> null
    }

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

That's more like it! You're now deleting solely the exact resource you want, without shifting all of the other ones around. This is why you should almost always prefer to use `for_each` instead of `count` to create multiple copies of a resource.

`for_each` works with modules in a more or less identical fashion. Using the `iam-user` module from earlier, you can create three IAM users with it using `for_each` as follows:

```
module "users" {
  source = "../../../modules/landing-zone/iam-user"

  for_each  = toset(var.user_names)
  user_name = each.value
}
```

And you can output the ARNs of those users as follows:

```
output "user_arns" {
  value       = values(module.users)[*].user_arn
  description = "The ARNs of the created IAM users"
}
```

When you run `apply` on this code, you get the expected output:

```
$ terraform apply

(...)

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

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

Let's now turn our attention to another advantage of `for_each`: its ability to create multiple inline blocks within a resource. For example, you can use `for_each` to dynamically generate `tag` inline blocks for the ASG in the `webserver-cluster` module. First, to allow users to specify custom tags, 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     = {}
}
```

Next, 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

  custom_tags = {
    Owner     = "team-foo"
    ManagedBy = "terraform"
  }
}
```

The preceding code sets a couple of useful tags: the `Owner` tag specifies which team owns this ASG, and the `ManagedBy` tag specifies that this infrastructure is managed using Terraform (indicating that this infrastructure shouldn't be modified manually).

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 pseudocode:

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnets.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 preceding pseudocode won't work, but a `for_each` expression will. The syntax for using `for_each` to dynamically generate inline blocks looks like this:

```
dynamic "<VAR_NAME>" {
  for_each = <COLLECTION>

  content {
    [CONFIG...]
  }
}
```

where `VAR_NAME` is the name to use for the variable that will store the value of each "iteration," `COLLECTION` is a list or map to iterate over, 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_subnets.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 plan` now, you should see a plan that looks something like this:

```
$ terraform plan

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                 = "ManagedBy"
          + propagate_at_launch = true
          + value               = "terraform"
        }
    }

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

Enforcing Tagging Standards

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. One way to do this is to manually ensure that every resource in every module sets the proper tags, but with many resources, this is tedious and error prone. If there are tags that you want to apply to *all* of your AWS resources, a more reliable approach is to add the `default_tags` block to the `aws` provider in every one of your modules:

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

  # Tags to apply to all AWS resources by default
  default_tags {
    tags = {
      Owner     = "team-foo"
      ManagedBy = "Terraform"
    }
  }
}
```

The preceding code will ensure that every single AWS resource you create in this module will include the `Owner` and `ManagedBy` tags (the only exceptions are resources that don't support tags and the `aws_autoscaling_group` resource, which does support tags but doesn't work with `default_tags`, which is why you had to do all that work in the previous section to set tags in the `webserver-cluster` module). `default_tags` gives you a way to ensure all resources have a common baseline of tags while still

allowing you to override those tags on a resource-by-resource basis. In Chapter 9, you'll see how to define and enforce policies as code such as "all resources must have a `ManagedBy` tag" using tools such as OPA.

Loops with for Expressions

You've now seen how to use loops to create multiple copies of entire resources and inline blocks, but what if you need a loop to set a single variable or parameter?

Imagine that 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 uppercase? 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 as follows:

```
[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 uppercase:

```
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:

```
output "short_upper_names" {
  value = [for name in var.names : upper(name) if length(name) < 5]
}
```

Running `terraform apply` on this code gives you this:

```
short_upper_names = [
  "NEO",
]
```

Terraform's `for` expression also allows you to loop over a map using the following syntax:

```
[for <KEY>, <VALUE> in <MAP> : <OUTPUT>]
```

Here, `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 the following:

```
bios = [
  "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 a list using the following syntax:

```
# Loop over a list and output a map
{for <ITEM> in <LIST> : <OUTPUT_KEY> => <OUTPUT_VALUE>}

# Loop over a map and output a map
{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 uppercase:

```
output "upper_roles" {
  value = {for name, role in var.hero_thousand_faces : upper(name) => upper(role)}
}
```

Here's the output from running this code:

```
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 = "%{ for name in var.names }${name}, %{ endfor }"
}
```

When you run `terraform apply`, you get the following output:

```
$ terraform apply

(...)

Outputs:

for_directive = "neo, trinity, morpheus, "
```

There's also a version of the `for` string directive syntax that gives you the index in the for-loop:

```
%{ for <INDEX>, <ITEM> in <COLLECTION> }<BODY>%{ endfor }
```

Here's an example using the index:

```
output "for_directive_index" {
  value = "%{ for i, name in var.names }(${i}) ${name}, %{ endfor }"
}
```

When you run `terraform apply`, you get the following output:

```
$ terraform apply

(...)

Outputs:

for_directive_index = "(0) neo, (1) trinity, (2) morpheus, "
```

Note how in both outputs there is an extra trailing comma and space. You can fix this using conditionals—specifically, the `if` string directive—as described in the next section.

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:

`count` *parameter*
: Used for conditional resources

`for_each` *and* `for` *expressions*
: Used for conditional resources and inline blocks within a resource

`if` *string directive*
: Used for conditionals within a string

Let's go through 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 begin by looking at if-statements in the next section and then move on to if-else-statements in the section thereafter.

If-statements with the count parameter

In Chapter 4, you created a Terraform module that could be used as a "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 scheduled action. Because you want to scale the cluster out only 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 you can use 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:

- 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.
- 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"
    ManagedBy = "terraform"
```

```
  }
}
```

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 allow the person applying the Terraform configurations to decide whether neo is assigned only read access or both read and write access. This is a slightly contrived example, but a useful one to demonstrate a simple type of if-else-statement.

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). And there you are—you now know how to do if-else-statements!

Now that you have the ability to create one resource or the other based on an if/else condition, what do you do if you need to access an attribute on the resource that actually got created? For example, what if you wanted to add an output variable called `neo_cloudwatch_policy_arn`, which contains the ARN of the policy you actually attached?

The simplest option is to use ternary syntax:

```
output "neo_cloudwatch_policy_arn" {
  value = (
    var.give_neo_cloudwatch_full_access
    ? aws_iam_user_policy_attachment.neo_cloudwatch_full_access[0].policy_arn
    : aws_iam_user_policy_attachment.neo_cloudwatch_read_only[0].policy_arn
  )
}
```

This will work fine for now, but this code is a bit brittle: if you ever change the conditional in the `count` parameter of the `aws_iam_user_policy_attachment` resources—perhaps in the future, it'll depend on multiple variables and not solely on `var.give_neo_cloudwatch_full_access`—there's a risk that you'll forget to update the conditional in this output variable, and as a result, you'll get a very confusing error when trying to access an array element that might not exist.

A safer approach is to take advantage of the `concat` and `one` functions. The `concat` function takes two or more lists as inputs and combines them into a single list. The `one` function takes a list as input and if the list has 0 elements, it returns `null`; if the list has 1 element, it returns that element; and if the list has more than 1 element, it shows an error. Putting these two together, and combining them with a splat expression, you get the following:

```
output "neo_cloudwatch_policy_arn" {
  value = one(concat(
    aws_iam_user_policy_attachment.neo_cloudwatch_full_access[*].policy_arn,
    aws_iam_user_policy_attachment.neo_cloudwatch_read_only[*].policy_arn
  ))
}
```

Depending on the outcome of the if/else conditional, either `neo_cloudwatch_full_access` will be empty and `neo_cloudwatch_read_only` will contain one element or vice versa, so once you concatenate them together, you'll have a list with one element, and the `one` function will return that element. This will continue to work correctly no matter how you change your if/else conditional.

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 by using a `for_each` expression.

If you pass a `for_each` expression an empty collection, the result will be zero copies of the resource, inline block, or module where you have the `for_each`; if you pass it a nonempty collection, it will create one or more copies of the resource, inline block, or module. The only question is, how do you conditionally decide if the collection 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, 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 uppercase (perhaps for consistency), and uses a conditional in the `for` expression to filter out any `key` set to `Name` because the module already sets its own `Name` tag. By filtering values in the `for` expression, you can implement arbitrary conditional logic.

Note that even though you should almost always prefer `for_each` over `count` for creating multiple copies of a resource or module, when it comes to conditional logic, setting `count` to 0 or 1 tends to be simpler than setting `for_each` to an empty or nonempty collection. Therefore, I typically recommend using `count` to conditionally

create resources and modules, and using `for_each` for all other types of loops and conditionals.

Conditionals with the if String Directive

Let's now look at the `if` string directive, which has the following syntax:

```
%{ 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.

Earlier in the chapter, you used the `for` string directive to do loops within a string to output several comma-separated names. The problem was that there was an extra trailing comma and space at the end of the string. You can use the `if` string directive to fix this issue as follows:

```
output "for_directive_index_if" {
  value = <<EOF
%{ for i, name in var.names }
  ${name}%{ if i < length(var.names) - 1 }, %{ endif }
%{ endfor }
EOF
}
```

There are a few changes here from the original version:

- I put the code in a *HEREDOC*, which is a way to define multiline strings. This allows me to spread the code out across several lines so it is more readable.
- I used the `if` string directive to not output the comma and space for the last item in the list.

When you run `terraform apply`, you get the following output:

```
$ terraform apply

(...)

Outputs:

for_directive_index_if = <<EOT

  neo,

  trinity,

  morpheus


EOT
```

Whoops. The trailing comma is gone, but we've introduced a bunch of extra whitespace (spaces and newlines). Every whitespace you put in a HEREDOC ends up in the final string. You can fix this by adding *strip markers* (~) to your string directives, which will eat up the extra whitespace before or after the strip marker:

```
output "for_directive_index_if_strip" {
  value = <<EOF
%{~ for i, name in var.names ~}
${name}%{ if i < length(var.names) - 1 }, %{ endif }
%{~ endfor ~}
EOF
}
```

Let's give this version a try:

```
$ terraform apply

(...)

Outputs:

for_directive_index_if_strip = "neo, trinity, morpheus"
```

OK, that's a nice improvement: no extra whitespace or commas. You can make this output even prettier by adding an `else` to the string directive, which uses the following syntax:

```
%{ if <CONDITION> }<TRUEVAL>%{ else }<FALSEVAL>%{ endif }
```

where `FALSEVAL` is the expression to render if `CONDITION` evaluates to false. Here's an example of how to use the `else` clause to add a period at the end:

```
output "for_directive_index_if_else_strip" {
  value = <<EOF
%{~ for i, name in var.names ~}
${name}%{ if i < length(var.names) - 1 }, %{ else }.%{ endif }
%{~ endfor ~}
EOF
}
```

When you run `terraform apply`, you get the following output:

```
$ terraform apply

(...)

Outputs:

for_directive_index_if_else_strip = "neo, trinity, morpheus."
```

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 make changes to your code, how do you deploy a new Amazon Machine Image (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 because 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"
  type        = string
  default     = "ami-0fb653ca2d3203ac1"
}

variable "server_text" {
  description = "The text the web server should return"
  type        = string
  default     = "Hello, World"
}
```

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*, update the `image_id` parameter to use `var.ami`, and update the `templatefile` call in the `user_data` parameter to pass in `var.server_text`:

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

```
  user_data       = templatefile("${path.module}/user-data.sh", {
    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
  })

  # Required when using a launch configuration with an auto scaling group.
  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:

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

  ami         = "ami-0fb653ca2d3203ac1"
  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-0fb653ca2d3203ac1"
        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. There is a problem here: merely referencing the new launch configuration will have no effect until the ASG launches new EC2 Instances. So how do you instruct the ASG 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:[1]

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 that 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 will begin 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_subnets.default.ids
```

1 Credit for this technique goes to Paul Hinze.

```
  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 the following:

```
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)
      ~ 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-0fb653ca2d3203ac1"
```

```
      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 `forces replacement` next to its name parameter, which means that Terraform will replace it with a new ASG 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 5-1).

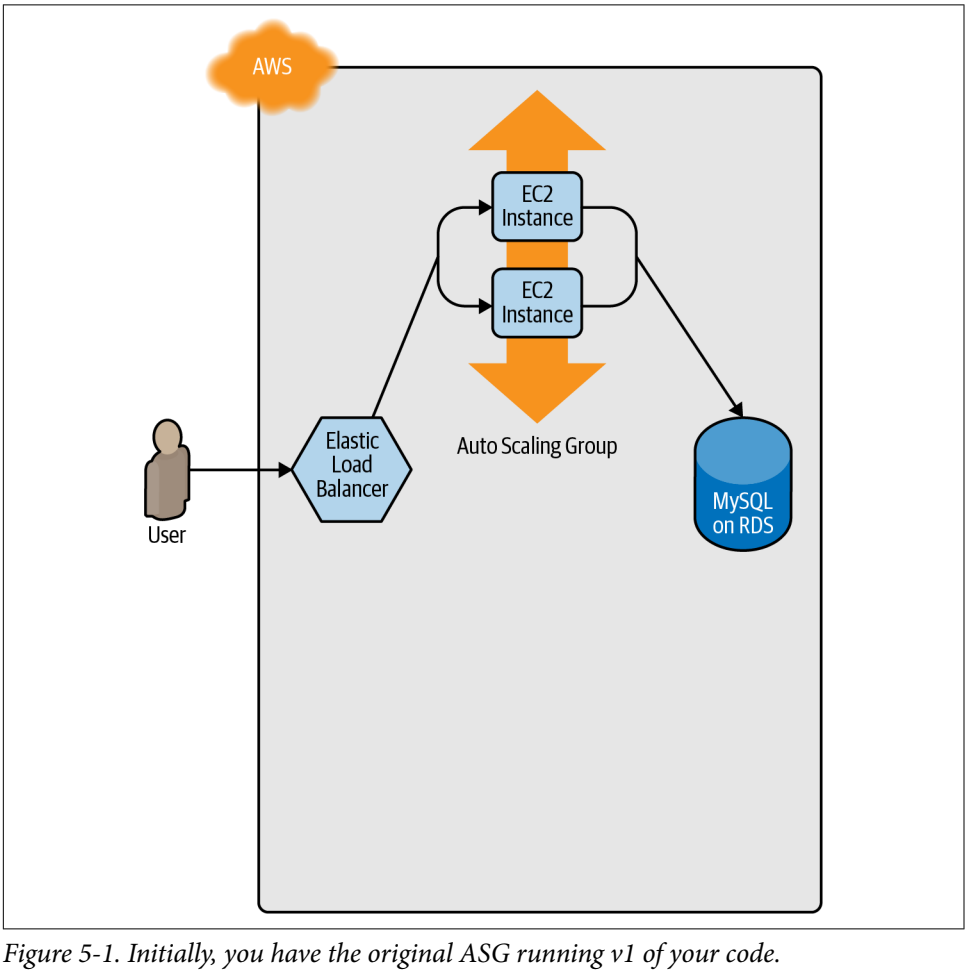


Figure 5-1. Initially, you have the original ASG running v1 of your code.

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 begin deploying a new ASG with v2 of your code (Figure 5-2).

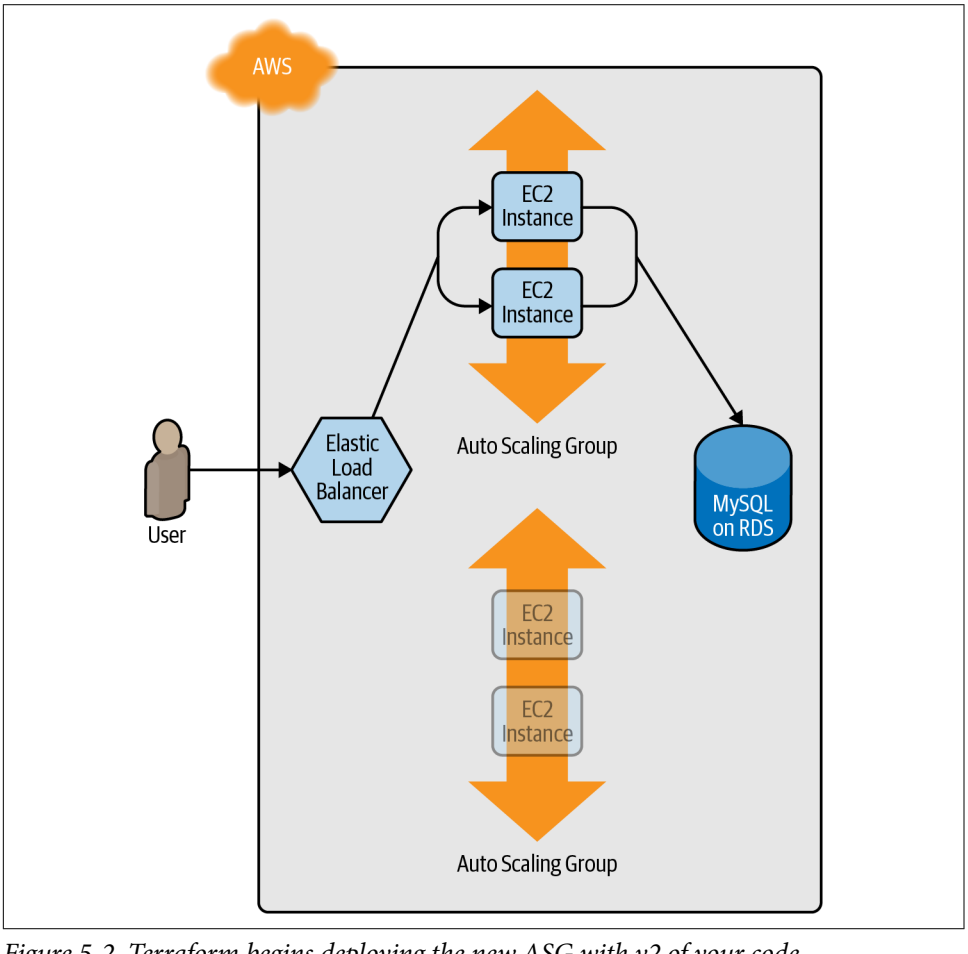


Figure 5-2. Terraform begins deploying the new ASG with v2 of your code.

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 5-3).

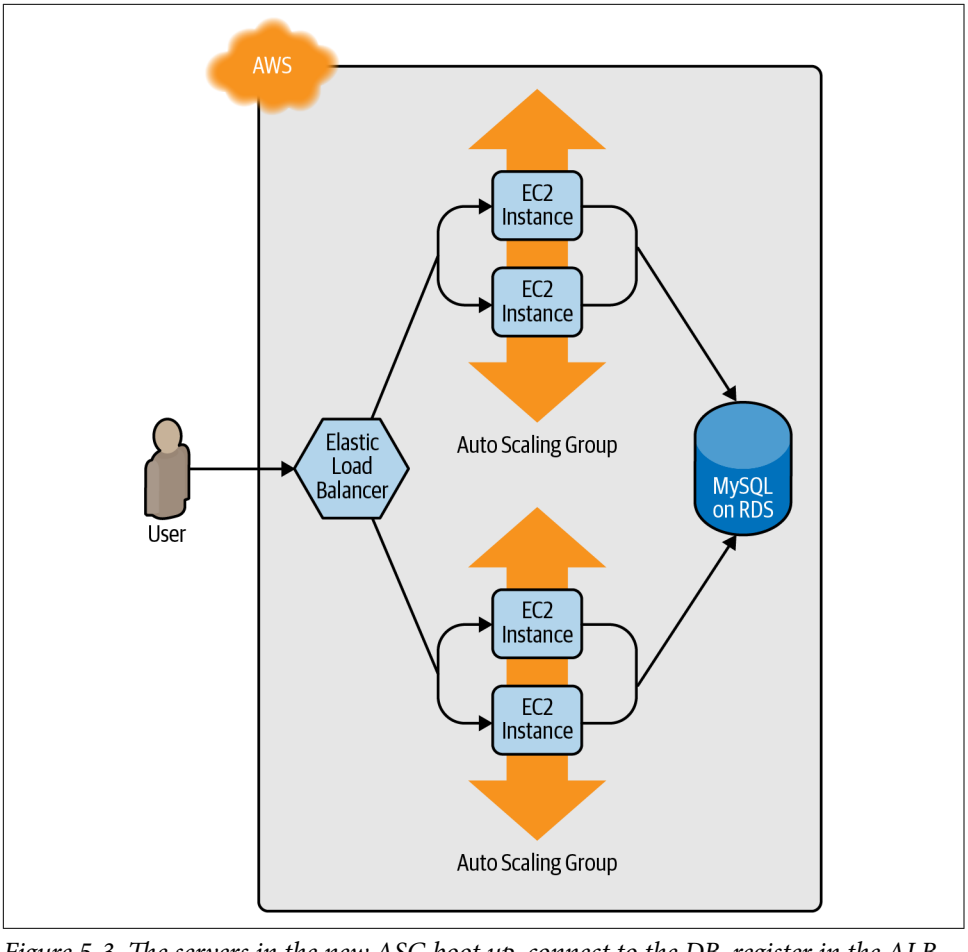


Figure 5-3. The servers in the new ASG boot up, connect to the DB, register in the ALB, and begin serving traffic.

After `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 5-4).

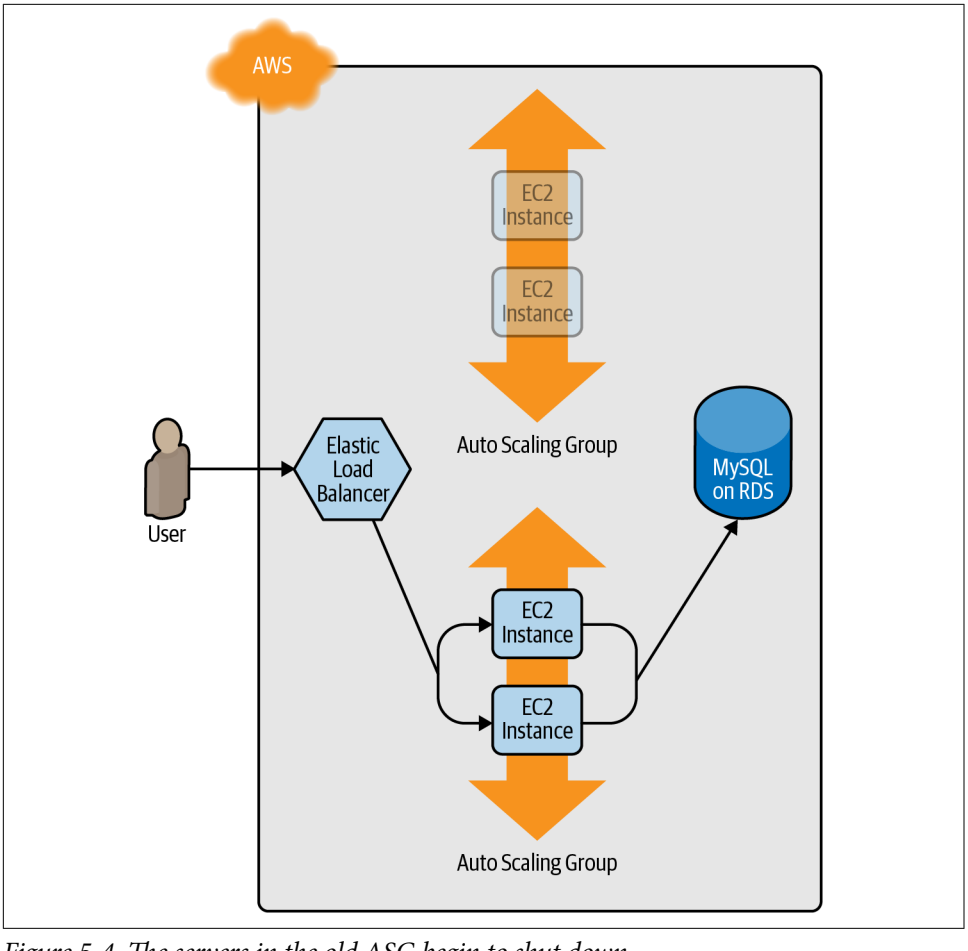


Figure 5-4. The servers in the old ASG begin to shut down.

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 5-5).

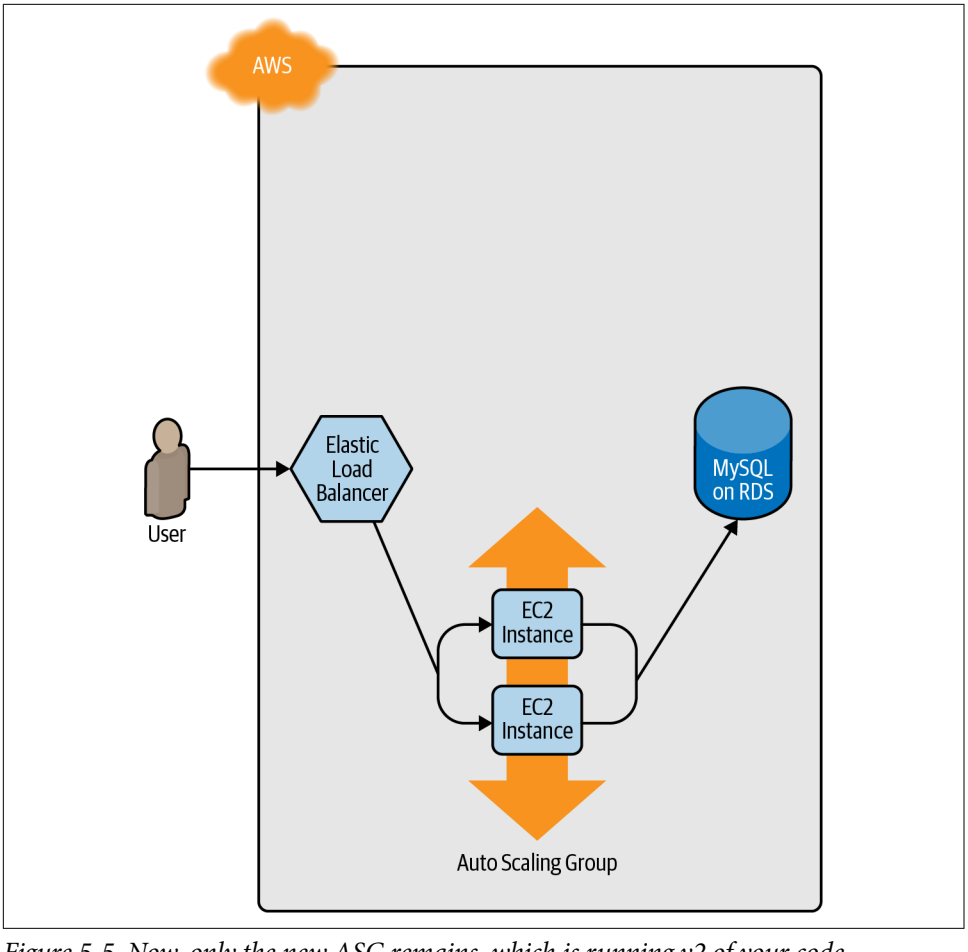


Figure 5-5. Now, only the new ASG remains, which is running v2 of your code.

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 5-6.

Figure 5-6. 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—for example, update it to say "foo bar"—and run the `apply` command. In a separate terminal tab, if you're on Linux/Unix/macOS, 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: `New server text`. Then, you'll begin seeing it alternate between `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` message will disappear, and you'll see only `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 were a bug in v2 of your app and it failed to boot, 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 considers the deployment a failure, deletes the v2 ASG, and exits 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` and `for_each` have limitations.
- Zero-downtime deployment has limitations.
- Valid plans can fail.
- Refactoring can be tricky.

count and for_each Have Limitations

In the examples in this chapter, you made extensive use of the `count` parameter and `for_each` expressions in loops and if-statements. This works well, but there's an important limitation that you need to be aware of: you cannot reference any resource outputs in `count` or `for_each`.

Imagine that you want to deploy multiple EC2 Instances, and for some reason you didn't want to use an ASG. The code might look like this:

```
resource "aws_instance" "example_1" {
  count         = 3
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

Because `count` is being set to a hardcoded value, this code will work without issues, and when you run `apply`, it will create three EC2 Instances. Now, what if you want 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-0fb653ca2d3203ac1"
  instance_type     = "t2.micro"
}

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

Again, this code works just fine, since `count` can reference data sources without problems. However, what happens if the number of instances you need to create depends 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-0fb653ca2d3203ac1"
  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.
```

Terraform requires that it can compute `count` and `for_each` during the `plan` phase, *before* any resources are created or modified. This means that `count` and `for_each` can reference hardcoded 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.

Zero-Downtime Deployment Has Limitations

There are a couple of gotchas with using `create_before_destroy` with an ASG to do zero-downtime deployment.

The first issue is that 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 `webserver-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 it would stay that way until 9 a.m. the next day. There are several possible workarounds, such as tweaking the `recurrence` parameter on the `aws_autoscaling_schedule` or setting the `desired_capacity` parameter of the ASG to get its value from a custom script that uses the AWS API to figure out how many instances were running before deployment.

However, the second, and bigger, issue is that, for important and complicated tasks like a zero-downtime deployment, you really want to use native, first-class solutions, and not workarounds that require you to haphazardly glue together `create_before_destroy`, `min_elb_capacity`, custom scripts, etc. As it turns out, for Auto Scaling Groups, AWS now offers a native solution called *instance refresh*.

Go back to your `aws_autoscaling_group` resource and undo the zero-downtime deployment changes:

- Set `name` back to `var.cluster_name`, instead of having it depend on the `aws_launch_configuration` name.
- Remove the `create_before_destroy` and `min_elb_capacity` settings.

And now, update the `aws_autoscaling_group` resource to instead use an `instance_refresh` block as follows:

```
resource "aws_autoscaling_group" "example" {
  name                 = var.cluster_name
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnets.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

  # Use instance refresh to roll out changes to the ASG
  instance_refresh {
    strategy = "Rolling"
```

```
    preferences {
      min_healthy_percentage = 50
    }
  }
}
```

If you deploy this ASG, and then later change some parameter (e.g., change `server_text`) and run `plan`, the diff will be back to just updating the `aws_launch_configuration`:

```
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-0fb653ca2d3203ac1"
        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.
```

If you run `apply`, it'll complete very quickly, and at first, nothing new will be deployed. However, in the background, because you modified the launch configuration, AWS will kick off the instance refresh process, as shown in Figure 5-7.

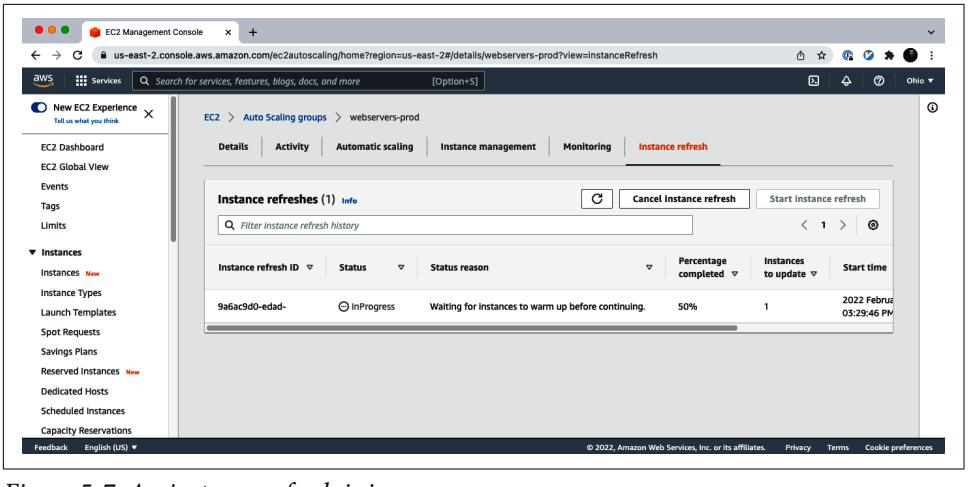


Figure 5-7. An instance refresh is in progress.

AWS will initially launch one new instance, wait for it to pass health checks, shut down one of the older instances, and then repeat the process with the second instance, until the instance refresh is completed, as shown in Figure 5-8.

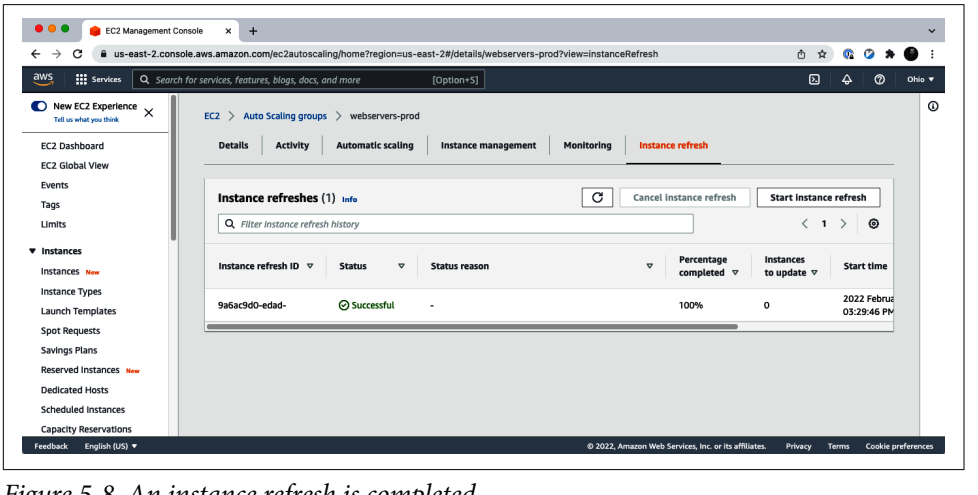


Figure 5-8. An instance refresh is completed.

This process is entirely managed by AWS, is reasonably configurable, handles errors pretty well, and requires no workarounds. The only drawback is the process can sometimes be slow (taking up to 20 minutes to replace just two servers), but other than that, it's a much more robust solution to use for most zero-downtime deployments.

In general, you should prefer to use first-class, native deployment options like instance refresh whenever possible. Although such options weren't always available in the earlier days of Terraform, these days, many resources support native deployment options. For example, if you're using Amazon Elastic Container Service (ECS) to deploy Docker containers, the `aws_ecs_service` resource natively supports zero-downtime deployments via the `deployment_maximum_percent` and `deployment_minimum_healthy_percent` parameters; if you're using Kubernetes to deploy Docker containers, the `kubernetes_deployment` resource natively supports zero-downtime deployments by setting the `strategy` parameter to `RollingUpdate` and providing configuration via the `rolling_update` block. Check the docs for the resources you're using, and make use of native functionality when you can!

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 manually in Chapter 2:

```
resource "aws_iam_user" "existing_user" {
  # Make sure to update this to your own user name!
  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 just with IAM users but with 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` looks only 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 will still fail.

There are two main lessons to take away from this:

After you start using Terraform, you should only use Terraform.
: When a part of your infrastructure is managed by Terraform, you should never manually make changes 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, given that the 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 that 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 at the bottom of the page for each resource typically specifies how to import it.

  For example, here is the `import` command that 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 an 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 might want to look into tools such as terraformer and terracognita, which can import both code and state from supported cloud environments automatically.

Refactoring Can Be Tricky

A common programming practice is *refactoring*, in which 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 IaC 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 automatically rename the variable or function for you, across the entire codebase. Although such a renaming is something you might do without thinking twice in a general-purpose programming language, you need to be very careful about 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 that you want to rename the service to `bar`. This might seem like a trivial change, but it can 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_subnets.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 that you might 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`, 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. You may run into similar problems if you change the identifier associated with a module, split one module into multiple modules, or add `count` or `for_each` to a resource or module that didn't have it before.

There are four main lessons that you should take away from this discussion:

Always use the `plan` *command*

You can catch all of these gotchas 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, think carefully about whether its replacement should be created before you delete the original. If so, you might 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.

Refactoring may require changing state

If you want to refactor your code without accidentally causing downtime, you'll need to update the Terraform state accordingly. However, you should never update Terraform state files by hand! Instead, you have two options: do it manually by running `terraform state mv` commands, or do it automatically by adding a `moved` block to your code.

Let's first look at the `terraform state mv` command, which 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. For example, if you're renaming an `aws_security_group` group from `instance` to `cluster_instance`, you could run the following:

```
$ terraform state mv \
  aws_security_group.instance \
  aws_security_group.cluster_instance
```

This instructs 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.

Having to remember to run CLI commands manually is error prone, especially if you refactored a module used by dozens of teams in your company, and each of those teams needs to remember to run `terraform state mv` to avoid downtime. Fortunately, Terraform 1.1 has added a way to handle this automatically: `moved` blocks. Any time you refactor your code, you should add a `moved` block to capture how the state should be updated. For example, to capture that the `aws_security_group` resource was renamed from `instance` to `cluster_instance`, you would add the following `moved` block:

```
moved {
  from = aws_security_group.instance
  to   = aws_security_group.cluster_instance
}
```

Now, whenever anyone runs `apply` on this code, Terraform will automatically detect if it needs to update the state file:

```
Terraform will perform the following actions:

  # aws_security_group.instance has moved to
  # aws_security_group.cluster_instance
    resource "aws_security_group" "cluster_instance" {
        name                   = "moved-example-security-group"
        tags                   = {}
        # (8 unchanged attributes hidden)
    }

Plan: 0 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:
```

If you enter **yes**, Terraform will update the state automatically, and as the plan shows no resources to add, change, or destroy, Terraform will make no other changes—which is exactly what you want!

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 documentation. And, once again, make sure to always use the `plan` command and consider whether you should use a `create_before_destroy` strategy.

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 needs to maintain your code, but just wild enough that you can create clean, beautiful APIs for your modules.

Let's now move on to Chapter 6, where I'll go over how create modules that are not only clean and beautiful but also handle secrets and sensitive data in a safe and secure manner.

CHAPTER 6

Managing Secrets with Terraform

At some point, you and your software will be entrusted with a variety of secrets, such as database passwords, API keys, TLS certificates, SSH keys, GPG keys, and so on. This is all sensitive data that, if it were to get into the wrong hands, could do a lot of damage to your company and its customers. If you build software, it is your responsibility to keep those secrets secure.

For example, consider the following Terraform code for deploying a database:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name

  # How to set these parameters securely?
  username = "???"
  password = "???"
}
```

This code requires you to set two secrets, the username and password, which are the credentials for the master user of the database. If the wrong person gets access to them, it could be catastrophic, as these credentials give you superuser access to that database and all the data within it. So, how do you keep these secrets secure?

This is part of the broader topic of *secrets management*, which is the focus of this chapter. This chapter will cover:

- Secret management basics
- Secret management tools

- Secret management tools with Terraform

Secret Management Basics

The first rule of secrets management is:

> Do not store secrets in plain text.

The second rule of secrets management is:

> DO NOT STORE SECRETS IN PLAIN TEXT.

Seriously, don't do it. For example, do *not* hardcode your database credentials directly in your Terraform code and check it into version control:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name

  # DO NOT DO THIS!!!
  username = "admin"
  password = "password"
  # DO NOT DO THIS!!!
}
```

Storing secrets in plain text in version control is a *bad idea*. Here are just a few of the reasons why:

Anyone who has access to the version control system has access to that secret.
: In the preceding example, every single developer at your company who can access that Terraform code will have access to the master credentials for your database.

Every computer that has access to the version control system keeps a copy of that secret.
: Every single computer that has ever checked out that repo may still have a copy of that secret on its local hard drive. That includes the computer of every developer on your team, every computer involved in CI (e.g., Jenkins, CircleCI, GitLab, etc.), every computer involved in version control (e.g., GitHub, GitLab, BitBucket), every computer involved in deployment (e.g., all your pre-prod and prod environments), every computer involved in backup (e.g., CrashPlan, Time Machine, etc.), and so on.

Every piece of software you run has access to that secret.
: Because the secrets are sitting in plain text on so many hard drives, every single piece of software running on any of those computers can potentially read that secret.

There's no way to audit or revoke access to that secret.
: When secrets are sitting on hundreds of hard drives in plain text, you have no way to know who accessed them (there's no audit log) and no easy way to revoke access.

In short, if you store secrets in plain text, you are giving malicious actors (e.g., hackers, competitors, disgruntled former employees) countless ways to access your company's most sensitive data—e.g., by compromising the version control system, or by compromising any of the computers you use, or by compromising any piece of software on any of those computers—and you'll have no idea if you were compromised or have any easy way to fix things if you were.

Therefore, it's essential that you use a proper *secret management tool* to store your secrets.

Secret Management Tools

A comprehensive overview of all aspects of secret management is beyond the scope of this book, but to be able to use secret management tools with Terraform, it's worth briefly touching on the following topics:

- The types of secrets you store
- The way you store secrets
- The interface you use to access secrets
- A comparison of secret management tools

The Types of Secrets You Store

There are three primary types of secrets: personal secrets, customer secrets, and infrastructure secrets.

Personal secrets
: Belong to an individual. Examples include the usernames and passwords for websites you visit, your SSH keys, and your Pretty Good Privacy (PGP) keys.

Customer secrets
: Belong to your customers. Note that if you run software for other employees of your company—e.g., you manage your company's internal Active Directory server—then those other employees are your customers. Examples include the

usernames and passwords that your customers use to log into your product, personally identifiable info (PII) for your customers, and personal health information (PHI) for your customers.

Infrastructure secrets
: Belong to your infrastructure. Examples include database passwords, API keys, and TLS certificates.

Most secret management tools are designed to store exactly one of these types of secrets, and while you could try to force it to store the other types, that's rarely a good idea from a security or usability standpoint. For example, the way you store passwords that are infrastructure secrets is completely different from how you store passwords that are customer secrets: for the former, you'd typically use an encryption algorithm such as AES (Advanced Encryption Standard), perhaps with a nonce, as you need to be able to decrypt the secrets and get back the original password; on the other hand, for the latter, you'd typically use a hashing algorithm (e.g., bcrypt) with a salt, as there should be no way to get back the original password. Using the wrong approach can be catastrophic, so use the right tool for the job!

The Way You Store Secrets

The two most common strategies for storing secrets are to use either a file-based secret store or a centralized secret store.

File-based secret stores store secrets in encrypted files, which are typically checked into version control. To encrypt the files, you need an encryption key. This key is itself a secret! This creates a bit of a conundrum: How do you securely store that key? You can't check the key into version control as plain text, as then there's no point of encrypting anything with it. You could encrypt the key with another key, but then all you've done is kicked the can down the road, as you still have to figure out how to securely store that second key.

The most common solution to this conundrum is to store the key in a *key management service* (KMS) provided by your cloud provider, such as AWS KMS, GCP KMS, or Azure Key Vault. This solves the kick-the-can-down-the-road problem by trusting the cloud provider to securely store the secret and manage access to it. Another option is to use PGP keys. Each developer can have their own PGP key, which consists of a *public key* and a *private key*. If you encrypt a secret with one or more public keys, only developers with the corresponding private keys will be able to decrypt those secrets. The private keys, in turn, are protected by a password that the developer either memorizes or stores in a personal secrets manager.

Centralized secret stores are typically web services that you talk to over the network that encrypt your secrets and store them in a data store such as MySQL, PostgreSQL, DynamoDB, etc. To encrypt these secrets, these centralized secret stores need an

encryption key. Typically, the encryption key is managed by the service itself, or the service relies on a cloud provider's KMS.

The Interface You Use to Access Secrets

Most secret management tools can be accessed via an API, CLI, and/or UI.

Just about all centralized secret stores expose an API that you can consume via network requests: e.g., a REST API you access over HTTP. The API is convenient for when your code needs to programmatically read secrets. For example, when an app is booting up, it can make an API call to your centralized secret store to retrieve a database password. Also, as you'll see later in this chapter, you can write Terraform code that, under the hood, uses a centralized secret store's API to retrieve secrets.

All the file-based secret stores work via a *command-line interface (CLI)*. Many of the centralized secret stores also provide CLI tools that, under the hood, make API calls to the service. CLI tools are a convenient way for developers to access secrets (e.g., using a few CLI commands to encrypt a file) and for scripting (e.g., writing a script to encrypt secrets).

Some of the centralized secret stores also expose a *user interface (UI)* via the web, desktop, or mobile. This is potentially an even more convenient way for everyone on your team to access secrets.

A Comparison of Secret Management Tools

Table 6-1 shows a comparison of popular secret management tools, broken down by the three considerations defined in the previous sections.

Table 6-1. A comparison of secret management tools

	Types of secrets	Secret storage	Secret interface
HashiCorp Vault	Infrastructure[a]	Centralized service	UI, API, CLI
AWS Secrets Manager	Infrastructure	Centralized service	UI, API, CLI
Google Secrets Manager	Infrastructure	Centralized service	UI, API, CLI
Azure Key Vault	Infrastructure	Centralized service	UI, API, CLI
Confidant	Infrastructure	Centralized service	UI, API, CLI
Keywhiz	Infrastructure	Centralized service	API, CLI
sops	Infrastructure	Files	CLI
git-secret	Infrastructure	Files	CLI
1Password	Personal	Centralized service	UI, API, CLI
LastPass	Personal	Centralized service	UI, API, CLI
Bitwarden	Personal	Centralized service	UI, API, CLI
KeePass	Personal	Files	UI, CLI

	Types of secrets	Secret storage	Secret interface
Keychain (macOS)	Personal	Files	UI, CLI
Credential Manager (Windows)	Personal	Files	UI, CLI
pass	Personal	Files	CLI
Active Directory	Customer	Centralized service	UI, API, CLI
Auth0	Customer	Centralized service	UI, API, CLI
Okta	Customer	Centralized service	UI, API, CLI
OneLogin	Customer	Centralized service	UI, API, CLI
Ping	Customer	Centralized service	UI, API, CLI
AWS Cognito	Customer	Centralized service	UI, API, CLI

[a] Vault supports multiple *secret engines*, most of which are designed for infrastructure secrets, but a few support customer secrets as well.

Since this is a book about Terraform, from here on out, I'll mostly be focusing on secret management tools designed for infrastructure secrets that are accessed through an API or the CLI (although I'll mention personal secret management tools from time to time too, as those often contain the secrets you need to authenticate to the infrastructure secret tools).

Secret Management Tools with Terraform

Let's now turn to how to use these secret management tools with Terraform, going through each of the three places where your Terraform code is likely to brush up against secrets:

- Providers
- Resources and data sources
- State files and plan files

Providers

Typically, your first exposure to secrets when working with Terraform is when you have to authenticate to a provider. For example, if you want to run `terraform apply` on code that uses the AWS Provider, you'll need to first authenticate to AWS, and that typically means using your access keys, which are secrets. How should you store those secrets? And how should you make them available to Terraform?

There are many ways to answer these questions. One way you should *not* do it, even though it occasionally comes up in the Terraform documentation, is by putting secrets directly into the code, in plain text:

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

  # DO NOT DO THIS!!!
  access_key = "(ACCESS_KEY)"
  secret_key = "(SECRET_KEY)"
  # DO NOT DO THIS!!!
}
```

Storing credentials this way, in plain text, is *not* secure, as discussed earlier in this chapter. Moreover, it's also not practical, as this would hardcode you to using one set of credentials for all users of this module, whereas in most cases, you'll need different credentials on different computers (e.g., when different developers or your CI server runs `apply`) and in different environments (dev, stage, prod).

There are several techniques that are far more secure for storing your credentials and making them accessible to Terraform providers. Let's take a look at these techniques, grouping them based on the user who is running Terraform:

Human users
: Developers running Terraform on their own computers.

Machine users
: Automated systems (e.g., a CI server) running Terraform with no humans present.

Human users

Just about all Terraform providers allow you to specify your credentials in some way other than putting them directly into the code. The most common option is to use environment variables. For example, here's how you use environment variables to authenticate to AWS:

```
$  export AWS_ACCESS_KEY_ID=(YOUR_ACCESS_KEY_ID)
$  export AWS_SECRET_ACCESS_KEY=(YOUR_SECRET_ACCESS_KEY)
```

Setting your credentials as environment variables keeps plain-text secrets out of your code, ensures that everyone running Terraform has to provide their own credentials, and ensures that credentials are only ever stored in memory, and not on disk.[1]

One important question you may ask is where to store the access key ID and secret access key in the first place. They are too long and random to memorize, but if you

1 Note that in most Linux/Unix/macOS shells, every command you type is stored on disk in some sort of history file (e.g., *~/.bash_history*). That's why the `export` commands shown here have a leading space: if you start your command with a space, most shells will skip writing that command to the history file. Note that you might need to set the `HISTCONTROL` environment variable to "ignoreboth" to enable this if your shell doesn't enable it by default.

store them on your computer in plain text, then you're still putting those secrets at risk. Since this section is focused on human users, the solution is to store your access keys (and other secrets) in a secret manager designed for personal secrets. For example, you could store your access keys in 1Password or LastPass and copy/paste them into the `export` commands in your terminal.

If you're using these credentials frequently on the CLI, an even more convenient option is to use a secret manager that supports a CLI interface. For example, 1Password offers a CLI tool called `op`. On Mac and Linux, you can use `op` to authenticate to 1Password on the CLI as follows:

```
$ eval $(op signin my)
```

Once you've authenticated, assuming you had used the 1Password app to store your access keys under the name "aws-dev" with fields "id" and "secret", here's how you can use `op` to set those access keys as environment variables:

```
$ export AWS_ACCESS_KEY_ID=$(op get item 'aws-dev' --fields 'id')
$ export AWS_SECRET_ACCESS_KEY=$(op get item 'aws-dev' --fields 'secret')
```

While tools like 1Password and `op` are great for general-purpose secrets management, for certain providers, there are dedicated CLI tools to make this even easier. For example, for authenticating to AWS, you can use the open source tool `aws-vault`. You can save your access keys using the `aws-vault add` command under a *profile* named `dev` as follows:

```
$ aws-vault add dev
Enter Access Key Id: (YOUR_ACCESS_KEY_ID)
Enter Secret Key: (YOUR_SECRET_ACCESS_KEY)
```

Under the hood, `aws-vault` will store these credentials securely in your operating system's native password manager (e.g., Keychain on macOS, Credential Manager on Windows). Once you've stored these credentials, now you can authenticate to AWS for any CLI command as follows:

```
$ aws-vault exec <PROFILE> -- <COMMAND>
```

where `PROFILE` is the name of a profile you created earlier via the `add` command (e.g., `dev`) and `COMMAND` is the command to execute. For example, here's how you can use the `dev` credentials you saved earlier to run `terraform apply`:

```
$ aws-vault exec dev -- terraform apply
```

The `exec` command automatically uses AWS STS to fetch temporary credentials and exposes them as environment variables to the command you're executing (in this case, `terraform apply`). This way, not only are your permanent credentials stored in a secure manner (in your operating system's native password manager), but now, you're also only exposing temporary credentials to any process you run, so the risk of leaking credentials is minimized. `aws-vault` also has native support for assuming

IAM roles, using multifactor authentication (MFA), logging into accounts on the web console, and more.

Machine users

Whereas a human user can rely on a memorized password, what do you do in cases where there's no human present? For example, if you're setting up a continuous integration / continuous delivery (CI/CD) pipeline to automatically run Terraform code, how do you securely authenticate that pipeline? In this case, you are dealing with authentication for a *machine user*. The question is, how do you get one machine (e.g., your CI server) to authenticate itself to another machine (e.g., AWS API servers) without storing any secrets in plain text?

The solution here heavily depends on the type of machines involved: that is, the machine you're authenticating *from* and the machine you're authenticating *to*. Let's go through three examples:

- CircleCI as a CI server, with stored secrets
- EC2 Instance running Jenkins as a CI server, with IAM roles
- GitHub Actions as a CI server, with OIDC

Warning: Simplified Examples

This section contains examples that fully flush out how to handle provider authentication in a CI/CD context, but all other aspects of the CI/CD workflow are highly simplified. You'll see more complete, end-to-end, production-ready CI/CD workflows in Chapter 9.

CircleCI as a CI server, with stored secrets. Let's imagine that you want to use CircleCI, a popular managed CI/CD platform, to run Terraform code. With CircleCI, you configure your build steps in a *.circleci/config.yml* file, where you might define a job to run `terraform apply` that looks like this:

```
version: '2.1'
orbs:
  # Install Terraform using a CircleCi Orb
  terraform: circleci/terraform@1.1.0
jobs:
  # Define a job to run 'terraform apply'
  terraform_apply:
    executor: terraform/default
    steps:
      - checkout          # git clone the code
      - terraform/init    # Run 'terraform init'
```

```
      - terraform/apply  # Run 'terraform apply'
workflows:
  # Create a workflow to run the 'terraform apply' job defined above
  deploy:
    jobs:
      - terraform_apply
    # Only run this workflow on commits to the main branch
    filters:
      branches:
        only:
          - main
```

With a tool like CircleCI, the way to authenticate to a provider is to create a machine user in that provider (that is, a user solely used for automation, and not by any human), store the credentials for that machine user in CircleCI in what's called a *CircleCI Context*, and when your build runs, CircleCI will expose the credentials in that Context to your workflows as environment variables. For example, if your Terraform code needs to authenticate to AWS, you would create a new IAM user in AWS, give that IAM user the permissions it needs to deploy your Terraform changes, and manually copy that IAM user's access keys into a CircleCI Context, as shown in Figure 6-1.

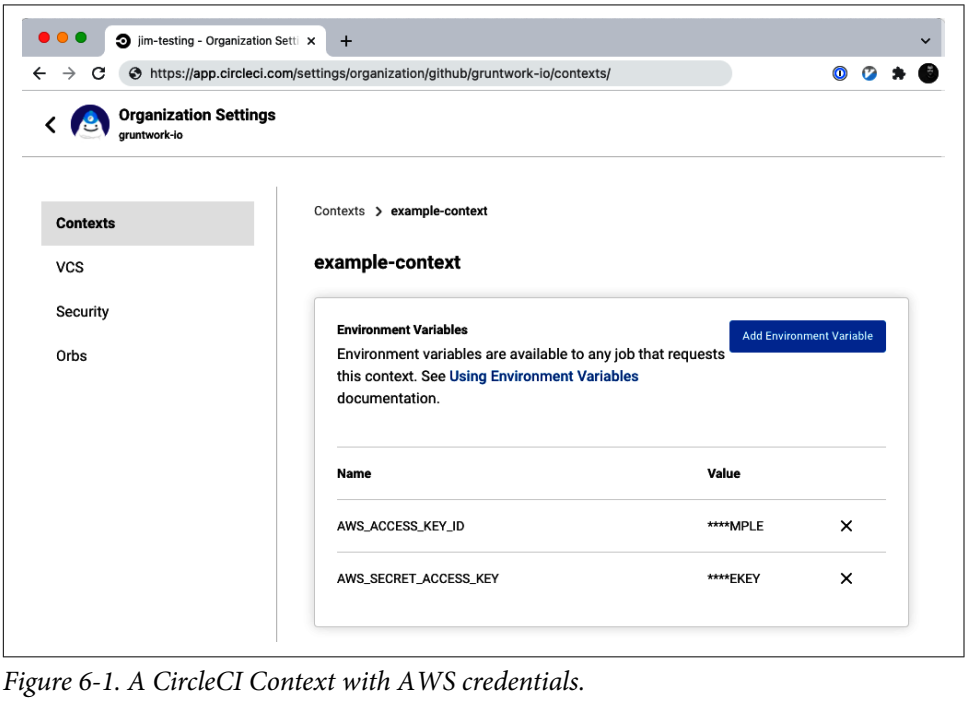


Figure 6-1. A CircleCI Context with AWS credentials.

Finally, you update the `workflows` in your *.circleci/config.yml* file to use your CircleCI Context via the `context` parameter:

```
workflows:
  # Create a workflow to run the 'terraform apply' job defined above
  deploy:
    jobs:
      - terraform_apply
    # Only run this workflow on commits to the main branch
    filters:
      branches:
        only:
          - main
    # Expose secrets in the CircleCI context as environment variables
    context:
      - example-context
```

When your build runs, CircleCI will automatically expose the secrets in that Context as environment variables—in this case, as the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`—and `terraform apply` will automatically use those environment variables to authenticate to your provider.

The biggest drawbacks to this approach are that (a) you have to manually manage credentials, and (b) as a result, you have to use permanent credentials, which once saved in CircleCI, rarely (if ever) change. The next two examples show alternative approaches.

EC2 Instance running Jenkins as a CI server, with IAM roles. If you're using an EC2 Instance to run Terraform code—e.g., you're running Jenkins on an EC2 Instance as a CI server—the solution I recommend for machine user authentication is to give that EC2 Instance an IAM role. An *IAM role* is similar to an IAM user, in that it's an entity in AWS that can be granted IAM permissions. However, unlike IAM users, IAM roles are not associated with any one person and do not have permanent credentials (password or access keys). Instead, a role can be *assumed* by other IAM entities: for example, an IAM user might assume a role to temporarily get access to different permissions than they normally have; many AWS services, such as EC2 Instances, can assume IAM roles to grant those services permissions in your AWS account.

For example, here's code you've seen many times to deploy an EC2 Instance:

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

To create an IAM role, you must first define an *assume role policy*, which is an IAM Policy that defines who is allowed to assume the IAM role. You could write the IAM Policy in raw JSON, but Terraform has a convenient `aws_iam_policy_document` data source that can create the JSON for you. Here's how you can use an `aws_iam_policy_document` to define an assume role policy that allows the EC2 service to assume an IAM role:

```
data "aws_iam_policy_document" "assume_role" {
  statement {
    effect  = "Allow"
    actions = ["sts:AssumeRole"]

    principals {
      type        = "Service"
      identifiers = ["ec2.amazonaws.com"]
    }
  }
}
```

Now, you can use the `aws_iam_role` resource to create an IAM role and pass it the JSON from your `aws_iam_policy_document` to use as the assume role policy:

```
resource "aws_iam_role" "instance" {
  name_prefix        = var.name
  assume_role_policy = data.aws_iam_policy_document.assume_role.json
}
```

You now have an IAM role, but by default, IAM roles don't give you any permissions. So, the next step is to attach one or more IAM policies to the IAM role that specify what you can actually do with the role once you've assumed it. Let's imagine that you're using Jenkins to run Terraform code that deploys EC2 Instances. You can use the `aws_iam_policy_document` data source to define an IAM Policy that gives admin permissions over EC2 Instances as follows:

```
data "aws_iam_policy_document" "ec2_admin_permissions" {
  statement {
    effect    = "Allow"
    actions   = ["ec2:*"]
    resources = ["*"]
  }
}
```

And you can attach this policy to your IAM role using the `aws_iam_role_policy` resource:

```
resource "aws_iam_role_policy" "example" {
  role   = aws_iam_role.instance.id
  policy = data.aws_iam_policy_document.ec2_admin_permissions.json
}
```

The final step is to allow your EC2 Instance to automatically assume that IAM role by creating an *instance profile*:

```
resource "aws_iam_instance_profile" "instance" {
  role = aws_iam_role.instance.name
}
```

And then tell your EC2 Instance to use that instance profile via the `iam_instance_profile` parameter:

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

  # Attach the instance profile
  iam_instance_profile = aws_iam_instance_profile.instance.name
}
```

Under the hood, AWS runs an *instance metadata endpoint* on every EC2 Instance at *http://169.254.169.254*. This is an endpoint that can only be reached by processes running on the instance itself, and those processes can use that endpoint to fetch metadata about the instance. For example, if you SSH to an EC2 Instance, you can query this endpoint using `curl`:

```
$ ssh ubuntu@<IP_OF_INSTANCE>
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.11.0-1022-aws x86_64)
(...)

$ curl http://169.254.169.254/latest/meta-data/
ami-id
ami-launch-index
ami-manifest-path
block-device-mapping/
events/
hibernation/
hostname
identity-credentials/
(...)
```

If the instance has an IAM role attached (via an instance profile), that metadata will include AWS credentials that can be used to authenticate to AWS and assume that IAM role. Any tool that uses the AWS SDK, such as Terraform, knows how to use these instance metadata endpoint credentials automatically, so as soon as you run `terraform apply` on the EC2 Instance with this IAM role, your Terraform code will authenticate as this IAM role, which will thereby grant your code the EC2 admin permissions it needs to run successfully.[2]

For any automated process running in AWS, such as a CI server, IAM roles provide a way to authenticate (a) without having to manage credentials manually, and (b) the credentials AWS provides via the instance metadata endpoint are always temporary,

2 By default, the instance metadata endpoint is open to all OS users running on your EC2 Instances. I recommend locking this endpoint down so that only specific OS users can access it: e.g., if you're running an app on the EC2 Instance as user *app*, you could use `iptables` or `nftables` to only allow *app* to access the instance metadata endpoint. That way, if an attacker finds some vulnerability and is able to execute code on your instance, they will only be able to access the IAM role permissions if they are able to authenticate as user *app* (rather than as any user). Better still, if you only need the IAM role permissions during boot (e.g., to read a database password), you could disable the instance metadata endpoint entirely after boot, so an attacker who gets access later can't use the endpoint at all.

and rotated automatically. These are two big advantages over the manually managed, permanent credentials with a tool like CircleCI that runs outside of your AWS account. However, as you'll see in the next example, in some cases, it's possible to have these same advantages for external tools, too.

GitHub Actions as a CI server, with OIDC. GitHub Actions is another popular managed CI/CD platform you might want to use to run Terraform. In the past, GitHub Actions required you to manually copy credentials around, just like CircleCI. However, as of 2021, GitHub Actions offers a better alternative: *Open ID Connect (OIDC)*. Using OIDC, you can establish a trusted link between the CI system and your cloud provider (GitHub Actions supports AWS, Azure, and Google Cloud) so that your CI system can authenticate to those providers without having to manage any credentials manually.

You define GitHub Actions workflows in YAML files in a *.github/workflows* folder, such as the *terraform.yml* file shown here:

```
name: Terraform Apply
# Only run this workflow on commits to the main branch
on:
  push:
    branches:
      - 'main'
jobs:
  TerraformApply:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      # Run Terraform using HashiCorp's setup-terraform Action
      - uses: hashicorp/setup-terraform@v1
          with:
            terraform_version: 1.1.0
            terraform_wrapper: false
        run: |
          terraform init
          terraform apply -auto-approve
```

If your Terraform code talks to a provider such as AWS, you need to provide a way for this workflow to authenticate to that provider. To do this using OIDC,[3] the first step is to create an *IAM OIDC identity provider* in your AWS account, using the `aws_iam_openid_connect_provider` resource, and to configure it to trust the GitHub Actions thumbprint, fetched via the `tls_certificate` data source:

3 At the time this book was written, OIDC support between GitHub Actions and AWS was fairly new and the details subject to change. Make sure to check the latest GitHub OIDC documentation for the latest updates.

```
# Create an IAM OIDC identity provider that trusts GitHub
resource "aws_iam_openid_connect_provider" "github_actions" {
  url             = "https://token.actions.githubusercontent.com"
  client_id_list  = ["sts.amazonaws.com"]
  thumbprint_list = [
    data.tls_certificate.github.certificates[0].sha1_fingerprint
  ]
}

# Fetch GitHub's OIDC thumbprint
data "tls_certificate" "github" {
  url = "https://token.actions.githubusercontent.com"
}
```

Now, you can create IAM roles exactly as in the previous section—e.g., an IAM role with EC2 admin permissions attached—except the assume role policy for those IAM roles will look different:

```
data "aws_iam_policy_document" "assume_role_policy" {
  statement {
    actions = ["sts:AssumeRoleWithWebIdentity"]
    effect  = "Allow"

    principals {
      identifiers = [aws_iam_openid_connect_provider.github_actions.arn]
      type        = "Federated"
    }

    condition {
      test     = "StringEquals"
      variable = "token.actions.githubusercontent.com:sub"
      # The repos and branches defined in var.allowed_repos_branches
      # will be able to assume this IAM role
      values = [
        for a in var.allowed_repos_branches :
        "repo:${a["org"]}/${a["repo"]}:ref:refs/heads/${a["branch"]}"
      ]
    }
  }
}
```

This policy allows the IAM OIDC identity provider to assume the IAM role via federated authentication. Note the `condition` block, which ensures that only the specific GitHub repos and branches you specify via the `allowed_repos_branches` input variable can assume this IAM role:

```
variable "allowed_repos_branches" {
  description = "GitHub repos/branches allowed to assume the IAM role."
  type = list(object({
    org    = string
    repo   = string
    branch = string
```

```
  }))
  # Example:
  # allowed_repos_branches = [
  #   {
  #     org    = "brikis98"
  #     repo   = "terraform-up-and-running-code"
  #     branch = "main"
  #   }
  # ]
}
```

This is important to ensure you don't accidentally allow *all* GitHub repos to authenticate to your AWS account! You can now configure your builds in GitHub Actions to assume this IAM role. First, at the top of your workflow, give your build the `id-token: write` permission:

```
permissions:
  id-token: write
```

Next, add a build step just before running Terraform to authenticate to AWS using the `configure-aws-credentials` action:

```
      # Authenticate to AWS using OIDC
      - uses: aws-actions/configure-aws-credentials@v1
        with:
          # Specify the IAM role to assume here
          role-to-assume: arn:aws:iam::123456789012:role/example-role
          aws-region: us-east-2

      # Run Terraform using HashiCorp's setup-terraform Action
      - uses: hashicorp/setup-terraform@v1
          with:
            terraform_version: 1.1.0
            terraform_wrapper: false
        run: |
          terraform init
          terraform apply -auto-approve
```

Now, when you run this build in one of the repos and branches in the `allowed_repos_branches` variable, GitHub will be able to assume your IAM role automatically, using temporary credentials, and Terraform will authenticate to AWS using that IAM role, all without having to manage any credentials manually.

Resources and Data Sources

The next place you'll run into secrets with your Terraform code is with resources and data sources. For example, you saw earlier in the chapter the example of passing database credentials to the `aws_db_instance` resource:

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

```
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name

  # DO NOT DO THIS!!!
  username = "admin"
  password = "password"
  # DO NOT DO THIS!!!
}
```

I've said it multiple times in this chapter already, but it's such an important point that it's worth repeating again: storing those credentials in the code, as plain text, is a bad idea. So, what's a better way to do it?

There are three main techniques you can use:

- Environment variables
- Encrypted files
- Secret stores

Environment variables

This first technique, which you saw back in Chapter 3, as well as earlier in this chapter when talking about providers, keeps plain-text secrets out of your code by taking advantage of Terraform's native support for reading environment variables.

To use this technique, declare variables for the secrets you wish to pass in:

```
variable "db_username" {
  description = "The username for the database"
  type        = string
  sensitive   = true
}

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

Just as in Chapter 3, these variables are marked with `sensitive = true` to indicate they contain secrets (so Terraform won't log the values when you run `plan` or `apply`), and these variables do not have a `default` (so as not to store secrets in plain text). Next, pass the variables to the Terraform resources that need those secrets:

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

```
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name

  # Pass the secrets to the resource
  username = var.db_username
  password = var.db_password
}
```

Now you can pass in a value for each variable `foo` by setting the environment variable `TF_VAR_foo`:

```
$  export TF_VAR_db_username=(DB_USERNAME)
$  export TF_VAR_db_password=(DB_PASSWORD)
```

Passing in secrets via environment variables helps you avoid storing secrets in plain text in your code, but it doesn't answer an important question: How do you store the secrets securely? One nice thing about using environment variables is that they work with almost any type of secrets management solution. For example, one option is to store the secrets in a personal secrets manager (e.g., 1Password) and manually set those secrets as environment variables in your terminal. Another option is to store the secrets in a centralized secret store (e.g., HashiCorp Vault) and write a script that uses that secret store's API or CLI to read those secrets out and set them as environment variables.

Using environment variables has the following advantages:

- Keep plain-text secrets out of your code and version control system.
- Storing secrets is easy, as you can use just about any other secret management solution. That is, if your company already has a way to manage secrets, you can typically find a way to make it work with environment variables.
- Retrieving secrets is easy, as reading environment variables is straightforward in every language.
- Integrating with automated tests is easy, as you can easily set the environment variables to mock values.
- Using environment variables doesn't cost any money, unlike some of the other secret management solutions discussed later.

Using environment variables has the following drawbacks:

- Not everything is defined in the Terraform code itself. This makes understanding and maintaining the code harder. Everyone using your code has to know to take extra steps to either manually set these environment variables or run a wrapper script.

- Standardizing secret management practices is harder. Since all the management of secrets happens outside of Terraform, the code doesn't enforce any security properties, and it's possible someone is still managing the secrets in an insecure way (e.g., storing them in plain text).
- Since the secrets are not versioned, packaged, and tested with your code, configuration errors are more likely, such as adding a new secret in one environment (e.g., staging) but forgetting to add it in another environment (e.g., production).

Encrypted files

The second technique relies on encrypting the secrets, storing the ciphertext in a file, and checking that file into version control.

To encrypt some data, such as some secrets in a file, you need an encryption key. As mentioned earlier in this chapter, this encryption key is itself a secret, so you need a secure way to store it. The typical solution is to either use your cloud provider's KMS (e.g., AWS KMS, Google KMS, Azure Key Vault) or to use the PGP keys of one or more developers on your team.

Let's look at an example that uses AWS KMS. First, you'll need to create a KMS *Customer Managed Key* (CMK), which is an encryption key that AWS manages for you. To create a CMK, you first have to define a *key policy*, which is an IAM Policy that defines who can use that CMK. To keep this example simple, let's create a key policy that gives the current user admin permissions over the CMK. You can fetch the current user's information—their username, ARN, etc.—using the `aws_caller_identity` data source:

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

data "aws_caller_identity" "self" {}
```

And now you can use the `aws_caller_identity` data source's outputs inside an `aws_iam_policy_document` data source to create a key policy that gives the current user admin permissions over the CMK:

```
data "aws_iam_policy_document" "cmk_admin_policy" {
  statement {
    effect    = "Allow"
    resources = ["*"]
    actions   = ["kms:*"]
    principals {
      type        = "AWS"
      identifiers = [data.aws_caller_identity.self.arn]
    }
  }
}
```

Next, you can create the CMK using the `aws_kms_key` resource:

```
resource "aws_kms_key" "cmk" {
  policy = data.aws_iam_policy_document.cmk_admin_policy.json
}
```

Note that, by default, KMS CMKs are only identified by a long numeric identifier (e.g., `b7670b0e-ed67-28e4-9b15-0d61e1485be3`), so it's a good practice to also create a human-friendly *alias* for your CMK using the `aws_kms_alias` resource:

```
resource "aws_kms_alias" "cmk" {
  name          = "alias/kms-cmk-example"
  target_key_id = aws_kms_key.cmk.id
}
```

The preceding alias will allow you to refer to your CMK as `alias/kms-cmk-example` when using the AWS API and CLI, rather than a long identifier such as `b7670b0e-ed67-28e4-9b15-0d61e1485be3`. Once you've created the CMK, you can start using it to encrypt and decrypt data. Note that, by design, you'll never be able to see (and, therefore, to accidentally leak) the underlying encryption key. Only AWS has access to that encryption key, but you can make use of it by using the AWS API and CLI, as described next.

First, create a file called *db-creds.yml* with some secrets in it, such as the database credentials:

```
username: admin
password: password
```

Note: do *not* check this file into version control, as you haven't encrypted it yet! To encrypt this data, you can use the `aws kms encrypt` command and write the resulting ciphertext to a new file. Here's a small Bash script (for Linux/Unix/macOS) called *encrypt.sh* that performs these steps using the AWS CLI:

```
CMK_ID="$1"
AWS_REGION="$2"
INPUT_FILE="$3"
OUTPUT_FILE="$4"

echo "Encrypting contents of $INPUT_FILE using CMK $CMK_ID..."
ciphertext=$(aws kms encrypt \
  --key-id "$CMK_ID" \
  --region "$AWS_REGION" \
  --plaintext "fileb://$INPUT_FILE" \
  --output text \
  --query CiphertextBlob)

echo "Writing result to $OUTPUT_FILE..."
echo "$ciphertext" > "$OUTPUT_FILE"

echo "Done!"
```

Here's how you can use *encrypt.sh* to encrypt the *db-creds.yml* file with the KMS CMK you created earlier and store the resulting ciphertext in a new file called *db-creds.yml.encrypted*:

```
$ ./encrypt.sh \
  alias/kms-cmk-example \
  us-east-2 \
  db-creds.yml \
  db-creds.yml.encrypted

Encrypting contents of db-creds.yml using CMK alias/kms-cmk-example...
Writing result to db-creds.yml.encrypted...
Done!
```

You can now delete *db-creds.yml* (the plain-text file) and safely check *db-creds.yml.encrypted* (the encrypted file) into version control. At this point, you have an encrypted file with some secrets inside of it, but how do you make use of that file in your Terraform code?

The first step is to decrypt the secrets in this file using the `aws_kms_secrets` data source:

```
data "aws_kms_secrets" "creds" {
  secret {
    name    = "db"
    payload = file("${path.module}/db-creds.yml.encrypted")
  }
}
```

The preceding code reads *db-creds.yml.encrypted* from disk using the `file` helper function and, assuming you have permissions to access the corresponding key in KMS, decrypts the contents. That gives you back the contents of the original *db-creds.yml* file, so the next step is to parse the YAML as follows:

```
locals {
  db_creds = yamldecode(data.aws_kms_secrets.creds.plaintext["db"])
}
```

This code pulls out the database secrets from the `aws_kms_secrets` data source, parses the YAML, and stores the results in a local variable called `db_creds`. Finally, you can read the username and password from `db_creds` and pass those credentials to the `aws_db_instance` resource:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name
```

```
  # Pass the secrets to the resource
  username = local.db_creds.username
  password = local.db_creds.password
}
```

So now you have a way to store secrets in an encrypted file, which are safe to check into version control, and you have a way to read those secrets back out of the file in your Terraform code automatically.

One thing to note with this approach is that working with encrypted files is awkward. To make a change, you have to locally decrypt the file with a long `aws kms decrypt` command, make some edits, re-encrypt the file with another long `aws kms encrypt` command, and the whole time, be extremely careful to not accidentally check the plain-text data into version control or leave it sitting behind forever on your computer. This is a tedious and error-prone process.

One way to make this less awkward is to use an open source tool called sops. When you run `sops <FILE>`, sops will automatically decrypt `FILE` and open your default text editor with the plain-text contents. When you're done editing and exit the text editor, sops will automatically encrypt the contents. This way, the encryption and decryption are mostly transparent, with no need to run long `aws kms` commands and less chance of accidentally checking plain-text secrets into version control. As of 2022, sops can work with files encrypted via AWS KMS, GCP KMS, Azure Key Vault, or PGP keys. Note that Terraform doesn't yet have native support for decrypting files that were encrypted by sops, so you'll either need to use a third-party provider such as carlpett/sops or, if you're a Terragrunt user, you can use the built-in `sops_decrypt_file` function.

Using encrypted files has the following advantages:

- Keep plain-text secrets out of your code and version control system.
- Your secrets are stored in an encrypted format in version control, so they are versioned, packaged, and tested with the rest of your code. This helps reduce configuration errors, such as adding a new secret in one environment (e.g., staging) but forgetting to add it in another environment (e.g., production).
- Retrieving secrets is easy, assuming the encryption format you're using is natively supported by Terraform or a third-party plugin.
- It works with a variety of different encryption options: AWS KMS, GCP KMS, PGP, etc.
- Everything is defined in the code. There are no extra manual steps or wrapper scripts required (although sops integration does require a third-party plugin).

Using encrypted files has the following drawbacks:

- Storing secrets is harder. You either have to run lots of commands (e.g., `aws kms encrypt`) or use an external tool such as sops. There's a learning curve to using these tools correctly and securely.
- Integrating with automated tests is harder, as you will need to do extra work to make encryption keys and encrypted test data available for your test environments.
- The secrets are now encrypted, but as they are still stored in version control, rotating and revoking secrets is hard. If anyone ever compromises the encryption key, they can go back and decrypt all the secrets that were ever encrypted with it.
- The ability to audit who accessed secrets is minimal. If you're using a cloud key management service (e.g., AWS KMS), it will likely maintain an audit log of who used an encryption key, but you won't be able to tell what the key was actually used for (i.e., what secrets were accessed).
- Most managed key services cost a small amount of money. For example, each key you store in AWS KMS costs $1/month, plus $0.03 per 10,000 API calls, where each decryption and encryption operation requires one API call. A typical usage pattern, where you have a small number of keys in KMS and your apps use those keys to decrypt secrets during boot, usually costs $1–$10/month. For larger deployments, where you have dozens of apps and hundreds of secrets, the price is typically in the $10–$50/month range.
- Standardizing secret management practices is harder. Different developers or teams may use different ways to store encryption keys or manage encrypted files, and mistakes are relatively common, such as not using encryption correctly or accidentally checking in a plain-text file into version control.

Secret stores

The third technique relies on storing your secrets in a centralized secret store.

Some of the more popular secret stores are AWS Secrets Manager, Google Secret Manager, Azure Key Vault, and HashiCorp Vault. Let's look at an example using AWS Secrets Manager. The first step is to store your database credentials in AWS Secrets Manager, which you can do using the AWS Web Console, as shown in Figure 6-2.

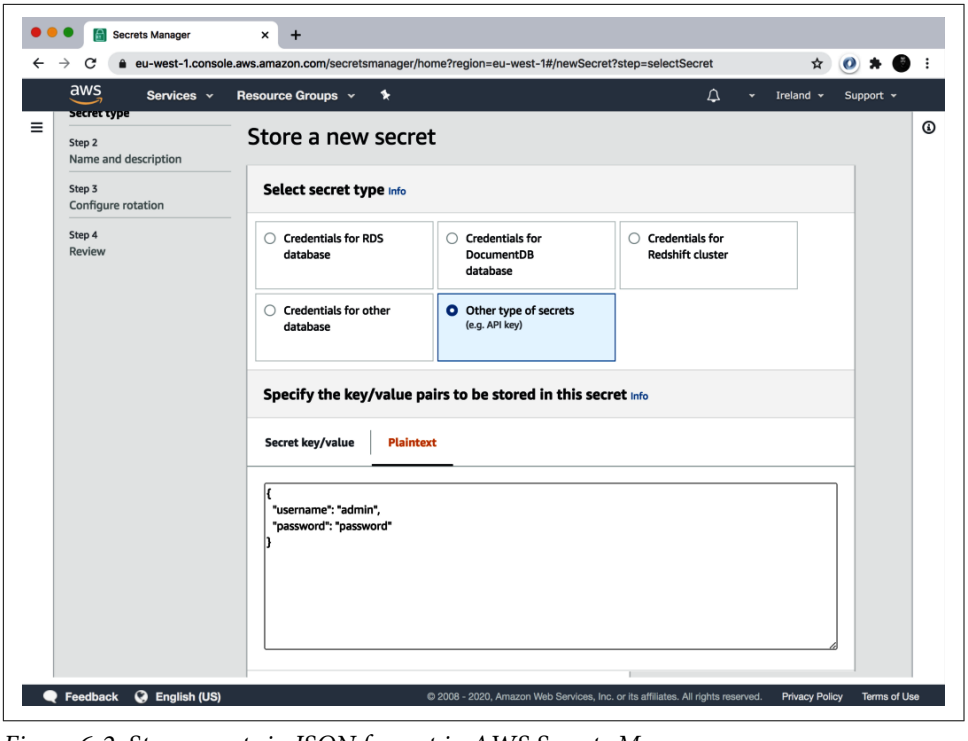


Figure 6-2. Store secrets in JSON format in AWS Secrets Manager.

Note that the secrets in Figure 6-2 are in a JSON format, which is the recommended format for storing data in AWS Secrets Manager.

Go to the next step, and make sure to give the secret a unique name, such as `db-creds`, as shown in Figure 6-3.

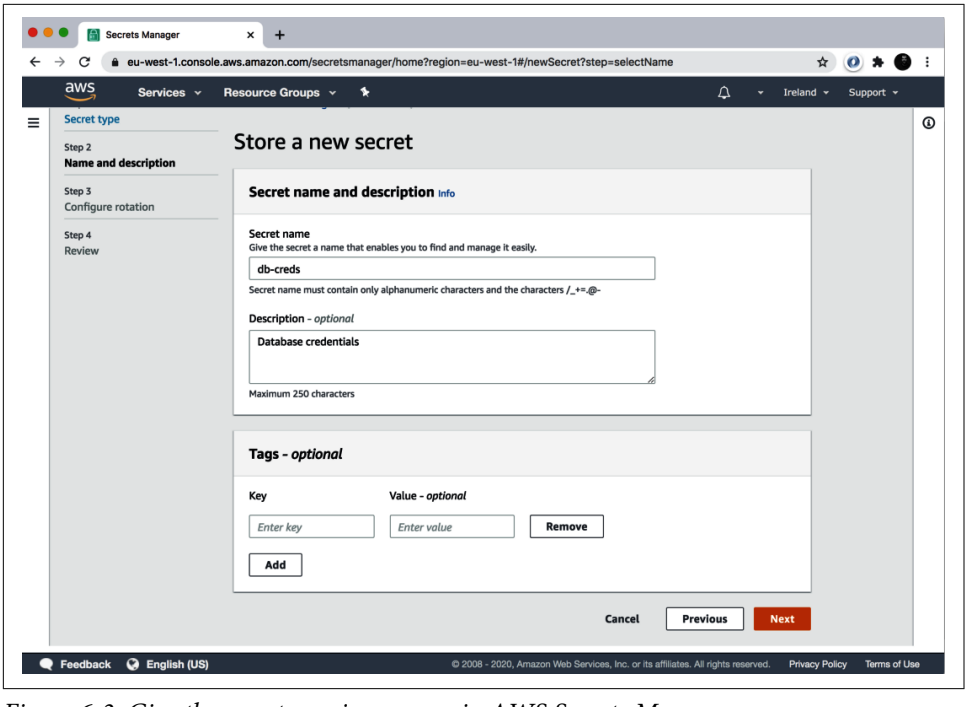


Figure 6-3. Give the secret a unique name in AWS Secrets Manager.

Click Next and Store to save the secret. Now, in your Terraform code, you can use the `aws_secretsmanager_secret_version` data source to read the `db-creds` secret:

```
data "aws_secretsmanager_secret_version" "creds" {
  secret_id = "db-creds"
}
```

Since the secret is stored as JSON, you can use the `jsondecode` function to parse the JSON into the local variable `db_creds`:

```
locals {
  db_creds = jsondecode(
    data.aws_secretsmanager_secret_version.creds.secret_string
  )
}
```

And now you can read the database credentials from `db_creds` and pass them into the `aws_db_instance` resource:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
```

```
  db_name             = var.db_name

  # Pass the secrets to the resource
  username = local.db_creds.username
  password = local.db_creds.password
}
```

Using secret stores has the following advantages:

- Keep plain-text secrets out of your code and version control system.
- Everything is defined in the code itself. There are no extra manual steps or wrapper scripts required.
- Storing secrets is easy, as you typically can use a web UI.
- Secret stores typically support rotating and revoking secrets, which is useful in case a secret gets compromised. You can even enable rotation on a scheduled basis (e.g., every 30 days) as a preventative measure.
- Secret stores typically support detailed audit logs that show you exactly who accessed what data.
- Secret stores make it easier to standardize all your secret practices, as they enforce specific types of encryption, storage, access patterns, etc.

Using secret stores has the following drawbacks:

- Since the secrets are not versioned, packaged, and tested with your code, configuration errors are more likely, such as adding a new secret in one environment (e.g., staging) but forgetting to add it in another environment (e.g., production).
- Most managed secret stores cost money. For example, AWS Secrets Manager charges $0.40 per month for each secret you store, plus $0.05 for every 10,000 API calls you make to store or retrieve data. A typical usage pattern, where you have several dozen secrets stored across several environments and a handful of apps that read those secrets during boot, usually costs around $10–$25/month. With larger deployments, where you have dozens of apps reading hundreds of secrets, the price can go up to hundreds of dollars per month.
- If you're using a self-managed secret store such as HashiCorp Vault, then you're both spending money to run the store (e.g., paying AWS for 3–5 EC2 Instances to run Vault in a highly available mode) and spending time and money to have your developers deploy, configure, manage, update, and monitor the store. Developer time is very expensive, so depending on how much time they have to spend on setting up and managing the secret store, this could cost you thousands of dollars per month.

- Retrieving secrets is harder, especially in automated environments (e.g., an app booting up and trying to read a database password), as you have to solve how to do secure authentication between multiple machines.
- Integrating with automated tests is harder, as much of the code you're testing now depends on a running, external system that either needs to be mocked out or have test data stored in it.

State Files and Plan Files

There are two more places where you'll come across secrets when using Terraform:

- State files
- Plan files

State files

Hopefully, this chapter has convinced you to not store your secrets in plain text and provided you with some better alternatives. However, something that catches many Terraform users off guard is that, no matter which technique you use, *any secrets you pass into your Terraform resources and data sources will end up in plain text in your Terraform state file!*

For example, no matter where you read the database credentials from—environment variables, encrypted files, a centralized secret store—if you pass those credentials to a resource such as `aws_db_instance`:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true
  db_name             = var.db_name

  # Pass the secrets to the resource
  username = local.db_creds.username
  password = local.db_creds.password
}
```

then Terraform will store those credentials in your *terraform.tfstate* file, in plain text. This has been an open issue since 2014, with no clear plans for a first-class solution. There are some workarounds out there that can scrub secrets from your state files, but these are brittle and likely to break with each new Terraform release, so I don't recommend them.

For the time being, no matter which of the techniques discussed you end up using to manage secrets, you must do the following:

Store Terraform state in a backend that supports encryption
: Instead of storing your state in a local *terraform.tfstate* file and checking it into version control, you should use one of the backends Terraform supports that natively supports encryption, such as S3, GCS, and Azure Blob Storage. These backends will encrypt your state files, both in transit (e.g., via TLS) and on disk (e.g., via AES-256).

Strictly control who can access your Terraform backend
: Since Terraform state files may contain secrets, you'll want to control who has access to your backend with *at least* as much care as you control access to the secrets themselves. For example, if you're using S3 as a backend, you'll want to configure an IAM Policy that solely grants access to the S3 bucket for production to a small handful of trusted devs, or perhaps solely just the CI server you use to deploy to prod.

Plan files

You've seen the `terraform plan` command many times. One feature you may not have seen yet is that you can store the output of the plan command (the "diff") in a file:

```
$ terraform plan -out=example.plan
```

The preceding command stores the plan in a file called *example.plan*. You can then run the `apply` command on this saved plan file to ensure that Terraform applies *exactly* the changes you saw originally:

```
$ terraform apply example.plan
```

This is a handy feature of Terraform, but an important caveat applies: just as with Terraform state, *any secrets you pass into your Terraform resources and data sources will end up in plain text in your Terraform plan files!* For example, if you ran `plan` on the `aws_db_instance` code, and saved a plan file, the plan file would contain the database username and password, in plain text.

Therefore, if you're going to use plan files, you must do the following:

Encrypt your Terraform plan files
: If you're going to save your plan files, you'll need to find a way to encrypt those files, both in transit (e.g., via TLS) and on disk (e.g., via AES-256). For example, you could store plan files in an S3 bucket, which supports both types of encryption.

Strictly control who can access your plan files

Since Terraform plan files may contain secrets, you'll want to control who has access to them with *at least* as much care as you control access to the secrets themselves. For example, if you're using S3 to store your plan files, you'll want to configure an IAM Policy that solely grants access to the S3 bucket for production to a small handful of trusted devs, or perhaps solely just the CI server you use to deploy to prod.

Conclusion

Here are your key takeaways from this chapter. First, if you remember nothing else from this chapter, please remember this: you should *not* store secrets in plain text.

Second, to pass secrets to providers, human users can use personal secrets managers and set environment variables, and machine users can use stored credentials, IAM roles, or OIDC. See Table 6-2 for the trade-offs between the machine user options.

Table 6-2. A comparison of methods for machine users (e.g., a CI server) to pass secrets to Terraform providers

	Stored credentials	IAM roles	OIDC
Example	CircleCI	Jenkins on an EC2 Instance	GitHub Actions
Avoid manually managing credentials	x	✓	✓
Avoid using permanent credentials	x	✓	✓
Works inside of cloud provider	x	✓	x
Works outside of cloud provider	✓	x	✓
Widely supported as of 2022	✓	✓	x

Third, to pass secrets to resources and data sources, use environment variables, encrypted files, or centralized secret stores. See Table 6-3 for the trade-offs between these different options.

Table 6-3. A comparison of methods for passing secrets to Terraform resources and data sources

	Environment variables	Encrypted files	Centralized secret stores
Keeps plain-text secrets out of code	✓	✓	✓
All secrets management defined as code	x	✓	✓
Audit log for access to encryption keys	x	✓	✓
Audit log for access to individual secrets	x	x	✓
Rotating or revoking secrets is easy	x	x	✓
Standardizing secrets management is easy	x	x	✓
Secrets are versioned with the code	x	✓	x

	Environment variables	Encrypted files	Centralized secret stores
Storing secrets is easy	✓	x	✓
Retrieving secrets is easy	✓	✓	x
Integrating with automated testing is easy	✓	x	x
Cost	0	$	$$$

And finally, fourth, no matter how you pass secrets to resources and data stores, remember that Terraform will store those secrets in your state files and plan files, in plain text, so make sure to always encrypt those files (in transit and at rest) and to strictly control access to them.

Now that you understand how to manage secrets when working with Terraform, including how to securely pass secrets to Terraform providers, let's move on to Chapter 7, where you'll learn how to use Terraform in cases where you have multiple providers (e.g., multiple regions, multiple accounts, multiple clouds).

CHAPTER 7

Working with Multiple Providers

So far, almost every single example in this book has included just a single `provider` block:

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

This `provider` block configures your code to deploy to a single AWS region in a single AWS account. This raises a few questions:

- What if you need to deploy to multiple AWS regions?
- What if you need to deploy to multiple AWS accounts?
- What if you need to deploy to other clouds, such as Azure or GCP?

To answer these questions, this chapter takes a deeper look at Terraform providers:

- Working with one provider
- Working with multiple copies of the same provider
- Working with multiple different providers

Working with One Provider

So far, you've been using providers somewhat "magically." That works well enough for simple examples with one basic provider, but if you want to work with multiple regions, accounts, clouds, etc., you'll need to go deeper. Let's start by taking a closer look at a single provider to better understand how it works:

- What is a provider?
- How do you install providers?
- How do you use providers?

What Is a Provider?

When I first introduced providers in Chapter 2, I described them as the *platforms* Terraform works with: e.g., AWS, Azure, Google Cloud, DigitalOcean, etc. So how does Terraform interact with these platforms?

Under the hood, Terraform consists of two parts:

Core

This is the `terraform` binary, and it provides all the basic functionality in Terraform that is used by all platforms, such as a command-line interface (i.e., `plan`, `apply`, etc.), a parser and interpreter for Terraform code (HCL), the ability to build a dependency graph from resources and data sources, logic to read and write state files, and so on. Under the hood, the code is written in Go and lives in an open source GitHub repo owned and maintained by HashiCorp.

Providers

Terraform providers are *plugins* for the Terraform core. Each plugin is written in Go to implement a specific interface, and the Terraform core knows how to install and execute the plugin. Each of these plugins is designed to work with some platform in the outside world, such as AWS, Azure, or Google Cloud. The Terraform core communicates with plugins via *remote procedure calls* (RPCs), and those plugins, in turn, communicate with their corresponding platforms via the network (e.g., via HTTP calls), as shown in Figure 7-1.

The code for each plugin typically lives in its own repo. For example, all the AWS functionality you've been using in the book so far comes from a plugin called the Terraform AWS Provider (or just AWS Provider for short) that lives in its own repo. Although HashiCorp created most of the initial providers, and still helps to maintain many of them, these days, much of the work for each provider is done by the company that owns the underlying platform: e.g., AWS employees work on the AWS Provider, Microsoft employees work on the Azure provider, Google employees work on the Google Cloud provider, and so on.

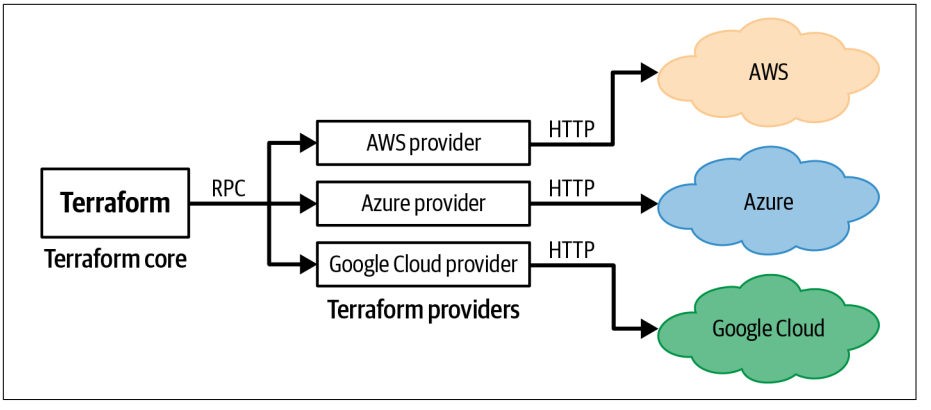


Figure 7-1. The interaction between the Terraform core, providers, and the outside world.

Each provider claims a specific prefix and exposes one or more resources and data sources whose names include that prefix: e.g., all the resources and data sources from the AWS Provider use the `aws_` prefix (e.g., `aws_instance`, `aws_autoscaling_group`, `aws_ami`), all the resources and data sources from the Azure provider use the `azurerm_` prefix (e.g., `azurerm_virtual_machine`, `azurerm_virtual_machine_scale_set`, `azurerm_image`), and so on.

How Do You Install Providers?

For official Terraform providers, such as the ones for AWS, Azure, and Google Cloud, it's enough to just add a `provider` block to your code:[1]

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

As soon as you run `terraform init`, Terraform automatically downloads the code for the provider:

```
$ terraform init

Initializing provider plugins...
- Finding hashicorp/aws versions matching "4.19.0"...
- Installing hashicorp/aws v4.19.0...
- Installed hashicorp/aws v4.19.0 (signed by HashiCorp)
```

1 In fact, you could even skip the `provider` block and just add any resource or data source from an official provider and Terraform will figure out which provider to use based on the prefix: for example, if you add the `aws_instance` resource, Terraform will know to use the AWS Provider based on the `aws_` prefix.

This is a bit magical, isn't it? How does Terraform know what provider you want? Or which version you want? Or where to download it from? Although it's OK to rely on this sort of magic for learning and experimenting, when writing production code, you'll probably want a bit more control over how Terraform installs providers. Do this by adding a `required_providers` block, which has the following syntax:

```
terraform {
  required_providers {
    <LOCAL_NAME> = {
      source  = "<URL>"
      version = "<VERSION>"
    }
  }
}
```

where:

`LOCAL_NAME`
: This is the *local name* to use for the provider in this module. You must give each provider a unique name, and you use that name in the `provider` block configuration. In almost all cases, you'll use the *preferred local name* of that provider: e.g., for the AWS Provider, the preferred local name is `aws`, which is why you write the provider block as `provider "aws" { … }`. However, in rare cases, you may end up with two providers that have the same preferred local name—e.g., two providers that both deal with HTTP requests and have a preferred local name of `http`—so you can use this local name to disambiguate between them.

`URL`
: This is the URL from where Terraform should download the provider, in the format `[<HOSTNAME>/]<NAMESPACE>/<TYPE>`, where `HOSTNAME` is the hostname of a Terraform Registry that distributes the provider, `NAMESPACE` is the organizational namespace (typically, a company name), and `TYPE` is the name of the platform this provider manages (typically, `TYPE` is the preferred local name). For example, the full URL for the AWS Provider, which is hosted in the public Terraform Registry, is `registry.terraform.io/hashicorp/aws`. However, note that `HOSTNAME` is optional, and if you omit it, Terraform will by default download the provider from the public Terraform Registry, so the shorter and more common way to specify the exact same AWS Provider URL is `hashicorp/aws`. You typically only include `HOSTNAME` for custom providers that you're downloading from private Terraform Registries (e.g., a private Registry you're running in Terraform Cloud or Terraform Enterprise).

`VERSION`
: This is a version constraint. For example, you could set it to a specific version, such as `4.19.0`, or to a version range, such as `> 4.0, < 4.3`. You'll learn more about how to handle versioning in Chapter 8.

For example, to install version 4.x of the AWS Provider, you can use the following code:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

So now you can finally understand the magical provider installation behavior you saw earlier. If you add a new `provider` block named `foo` to your code, and you don't specify a `required_providers` block, when you run `terraform init`, Terraform will automatically do the following:

- Try to download provider `foo` with the assumption that the `HOSTNAME` is the public Terraform Registry and that the `NAMESPACE` is `hashicorp`, so the download URL is `registry.terraform.io/hashicorp/foo`.
- If that's a valid URL, install the latest version of the `foo` provider available at that URL.

If you want to install any provider not in the `hashicorp` namespace (e.g., if you want to use providers from Datadog, Cloudflare, or Confluent, or a custom provider you built yourself), or you want to control the version of the provider you use, you will need to include a `required_providers` block.

Always Include required_providers

As you'll learn in Chapter 8, it's important to control the version of the provider you use, so I recommend *always* including a `required_providers` block in your code.

How Do You Use Providers?

With this new knowledge about providers, let's revisit how to use them. The first step is to add a `required_providers` block to your code to specify which provider you want to use:

```
terraform {
  required_providers {
```

```
      aws = {
        source  = "hashicorp/aws"
        version = "~> 4.0"
      }
    }
  }
```

Next, you add a `provider` block to configure that provider:

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

So far, you've only been configuring the `region` to use in the AWS Provider, but there are many other settings you can configure. Always check your provider's documentation for the details: typically, this documentation lives in the same Registry you use to download the provider (the one in the `source` URL). For example, the documentation for the AWS Provider is in the public Terraform Registry. This documentation will typically explain how to configure the provider to work with different users, roles, regions, accounts, and so on.

Once you've configured a provider, all the resources and data sources from that provider (all the ones with the same prefix) that you put into your code will automatically use that configuration. So, for example, when you set the region in the `aws` provider to `us-east-2`, all the `aws_` resources to your code will automatically deploy into `us-east-2`.

But what if you want some of those resources to deploy into `us-east-2` and some into a different region, such as `us-west-1`? Or what if you want to deploy some resources to a completely different AWS account? To do that, you'll have to learn how to configure multiple copies of the same provider, as discussed in the next section.

Working with Multiple Copies of the Same Provider

To understand how to work with multiple copies of the same provider, let's look at a few of the common cases where this comes up:

- Working with multiple AWS regions
- Working with multiple AWS accounts
- Creating modules that can work with multiple providers

Working with Multiple AWS Regions

Most cloud providers allow you to deploy into datacenters ("regions") all over the world, but when you configure a Terraform provider, you typically configure it to

deploy into just one of those regions. For example, so far you've been deploying into just a single AWS region, `us-east-2`:

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

What if you wanted to deploy into multiple regions? For example, how could you deploy some resources into `us-east-2` and other resources into `us-west-1`? You might be tempted to solve this by defining two `provider` configurations, one for each region:

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

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

But now there's a new problem: How do you specify which of these `provider` configurations each of your resources, data sources, and modules should use? Let's look at data sources first. Imagine you had two copies of the `aws_region` data source, which returns the current AWS region:

```
data "aws_region" "region_1" {
}

data "aws_region" "region_2" {
}
```

How do you get the `region_1` data source to use the `us-east-2` provider and the `region_2` data source to use the `us-west-1` provider? The solution is to add an alias to each provider:

```
provider "aws" {
  region = "us-east-2"
  alias  = "region_1"
}

provider "aws" {
  region = "us-west-1"
  alias  = "region_2"
}
```

An *alias* is a custom name for the `provider`, which you can explicitly pass to individual resources, data sources, and modules to get them to use the configuration in that particular provider. To tell those `aws_region` data sources to use a specific provider, you set the `provider` parameter as follows:

```
data "aws_region" "region_1" {
  provider = aws.region_1
}

data "aws_region" "region_2" {
  provider = aws.region_2
}
```

Add some output variables so you can check that this is working:

```
output "region_1" {
  value       = data.aws_region.region_1.name
  description = "The name of the first region"
}

output "region_2" {
  value       = data.aws_region.region_2.name
  description = "The name of the second region"
}
```

And run `apply`:

```
$ terraform apply

(...)

Outputs:

region_1 = "us-east-2"
region_2 = "us-west-1"
```

And there you go: each of the `aws_region` data sources is now using a different provider and, therefore, running against a different AWS region. The same technique of setting the `provider` parameter works with resources too. For example, here's how you can deploy two EC2 Instances in different regions:

```
resource "aws_instance" "region_1" {
  provider = aws.region_1

  # Note different AMI IDs!!
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}

resource "aws_instance" "region_2" {
  provider = aws.region_2

  # Note different AMI IDs!!
  ami           = "ami-01f87c43e618bf8f0"
  instance_type = "t2.micro"
}
```

Notice how each `aws_instance` resource sets the `provider` parameter to ensure it deploys into the proper region. Also, note that the `ami` parameter has to be different on the two `aws_instance` resources: that's because AMI IDs are unique to each AWS region, so the ID for Ubuntu 20.04 in `us-east-2` is different than for Ubuntu 20.04 in `us-west-1`. Having to look up and manage these AMI IDs manually is tedious and error prone. Fortunately, there's a better alternative: use the `aws_ami` data source that, given a set of filters, can find AMI IDs for you automatically. Here's how you can use this data source twice, once in each region, to look up Ubuntu 20.04 AMI IDs:

```
data "aws_ami" "ubuntu_region_1" {
  provider = aws.region_1

  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}

data "aws_ami" "ubuntu_region_2" {
  provider = aws.region_2

  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}
```

Notice how each data source sets the `provider` parameter to ensure it's looking up the AMI ID in the proper region. Go back to the `aws_instance` code and update the `ami` parameter to use the output of these data sources instead of the hardcoded values:

```
resource "aws_instance" "region_1" {
  provider = aws.region_1

  ami           = data.aws_ami.ubuntu_region_1.id
  instance_type = "t2.micro"
}

resource "aws_instance" "region_2" {
  provider = aws.region_2

  ami           = data.aws_ami.ubuntu_region_2.id
  instance_type = "t2.micro"
}
```

Much better. Now, no matter what region you deploy into, you'll automatically get the proper AMI ID for Ubuntu. To check that these EC2 Instances are really deploying into different regions, add output variables that show you which availability zone (each of which is in one region) each instance was actually deployed into:

```
output "instance_region_1_az" {
  value       = aws_instance.region_1.availability_zone
  description = "The AZ where the instance in the first region deployed"
}

output "instance_region_2_az" {
  value       = aws_instance.region_2.availability_zone
  description = "The AZ where the instance in the second region deployed"
}
```

And now run `apply`:

```
$ terraform apply

(...)

Outputs:

instance_region_1_az = "us-east-2a"
instance_region_2_az = "us-west-1b"
```

OK, so now you know how to deploy data sources and resources into different regions. What about modules? For example, in Chapter 3, you used Amazon RDS to deploy a single instance of a MySQL database in the staging environment (*stage/data-stores/mysql*):

```
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"
  skip_final_snapshot = true

  username = var.db_username
  password = var.db_password
}
```

This is fine in staging, but in production, a single database is a single point of failure. Fortunately, Amazon RDS natively supports *replication*, where your data is automatically copied from a primary database to a secondary database—a read-only *replica*—which is useful for scalability and as a standby in case the primary goes down. You can even run the replica in a totally different AWS region, so if one region

goes down (e.g., there's a major outage in `us-east-2`), you can switch to the other region (e.g., `us-west-1`).

Let's turn that MySQL code in the staging environment into a reusable `mysql` module that supports replication. First, copy all the contents of *stage/data-stores/mysql*, which should include *main.tf*, *variables.tf*, and *outputs.tf*, into a new *modules/data-stores/mysql* folder. Next, open *modules/data-stores/mysql/variables.tf* and expose two new variables:

```
variable "backup_retention_period" {
  description = "Days to retain backups. Must be > 0 to enable replication."
  type        = number
  default     = null
}

variable "replicate_source_db" {
  description = "If specified, replicate the RDS database at the given ARN."
  type        = string
  default     = null
}
```

As you'll see shortly, you'll set the `backup_retention_period` variable on the primary database to enable replication, and you'll set the `replicate_source_db` variable on the secondary database to turn it into a replica. Open up *modules/data-stores/mysql/main.tf*, and update the `aws_db_instance` resource as follows:

1. Pass the `backup_retention_period` and `replicate_source_db` variables into parameters of the same name in the `aws_db_instance` resource.
2. If a database instance is a replica, AWS does not allow you to set the `engine`, `db_name`, `username`, or `password` parameters, as those are all inherited from the primary. So you must add some conditional logic to the `aws_db_instance` resource to not set those parameters when the `replicate_source_db` variable is set.

Here's what the resource should look like after the changes:

```
resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  allocated_storage   = 10
  instance_class      = "db.t2.micro"
  skip_final_snapshot = true

  # Enable backups
  backup_retention_period = var.backup_retention_period

  # If specified, this DB will be a replica
  replicate_source_db = var.replicate_source_db
```

```
  # Only set these params if replicate_source_db is not set
  engine   = var.replicate_source_db == null ? "mysql" : null
  db_name  = var.replicate_source_db == null ? var.db_name : null
  username = var.replicate_source_db == null ? var.db_username : null
  password = var.replicate_source_db == null ? var.db_password : null
}
```

Note that for replicas, this implies that the `db_name`, `db_username`, and `db_password` input variables in this module should be optional, so it's a good idea to go back to *modules/data-stores/mysql/variables.tf* and set the `default` for those variables to `null`:

```
variable "db_name" {
  description = "Name for the DB."
  type        = string
  default     = null
}

variable "db_username" {
  description = "Username for the DB."
  type        = string
  sensitive   = true
  default     = null
}

variable "db_password" {
  description = "Password for the DB."
  type        = string
  sensitive   = true
  default     = null
}
```

To use the `replicate_source_db` variable, you'll need set it to the ARN of another RDS database, so you should also update *modules/data-stores/mysql/outputs.tf* to add the database ARN as an output variable:

```
output "arn" {
  value       = aws_db_instance.example.arn
  description = "The ARN of the database"
}
```

One more thing: you should add a `required_providers` block to this module to specify that this module expects to use the AWS Provider, and to specify which version of the provider the module expects.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

You'll see in a moment why this is important when working with multiple regions, too!

OK, you can now use this `mysql` module to deploy a MySQL primary and a MySQL replica in the production environment. First, create *live/prod/data-stores/mysql/variables.tf* to expose input variables for the database username and password (so you can pass these secrets in as environment variables, as discussed in Chapter 6):

```
variable "db_username" {
  description = "The username for the database"
  type        = string
  sensitive   = true
}

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

Next, create *live/prod/data-stores/mysql/main.tf*, and use the `mysql` module to configure the primary as follows:

```
module "mysql_primary" {
  source = "../../../../modules/data-stores/mysql"

  db_name     = "prod_db"
  db_username = var.db_username
  db_password = var.db_password

  # Must be enabled to support replication
  backup_retention_period = 1
}
```

Now, add a second usage of the `mysql` module to create a replica:

```
module "mysql_replica" {
  source = "../../../../modules/data-stores/mysql"

  # Make this a replica of the primary
  replicate_source_db = module.mysql_primary.arn
}
```

Nice and short! All you're doing is passing the ARN of the primary database into the `replicate_source_db` parameter, which should spin up an RDS database as a replica.

There's just one problem: How do you tell the code to deploy the primary and replica into different regions? To do so, create two `provider` blocks, each with its own alias:

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

```
provider "aws" {
  region = "us-west-1"
  alias  = "replica"
}
```

To tell a module which providers to use, you set the `providers` parameter. Here's how you configure the MySQL primary to use the `primary` provider (the one in `us-east-2`):

```
module "mysql_primary" {
  source = "../../../../modules/data-stores/mysql"

  providers = {
    aws = aws.primary
  }

  db_name     = "prod_db"
  db_username = var.db_username
  db_password = var.db_password

  # Must be enabled to support replication
  backup_retention_period = 1
}
```

And here is how you configure the MySQL replica to use the `replica` provider (the one in `us-west-1`):

```
module "mysql_replica" {
  source = "../../../../modules/data-stores/mysql"

  providers = {
    aws = aws.replica
  }

  # Make this a replica of the primary
  replicate_source_db = module.mysql_primary.arn
}
```

Notice that with modules, the `providers` (plural) parameter is a map, whereas with resources and data sources, the `provider` (singular) parameter is a single value. That's because each resource and data source deploys into exactly one provider, but a module may contain multiple data sources and resources and use multiple providers (you'll see an example of multiple providers in a module later). In the `providers` map you pass to a module, the key must match the local name of the provider in the `required_providers` map within the module (in this case, both are set to `aws`). This is yet another reason defining `required_providers` explicitly is a good idea in just about every module.

Alright, the last step is to create *live/prod/data-stores/mysql/outputs.tf* with the following output variables:

```
output "primary_address" {
  value       = module.mysql_primary.address
  description = "Connect to the primary database at this endpoint"
}

output "primary_port" {
  value       = module.mysql_primary.port
  description = "The port the primary database is listening on"
}

output "primary_arn" {
  value       = module.mysql_primary.arn
  description = "The ARN of the primary database"
}

output "replica_address" {
  value       = module.mysql_replica.address
  description = "Connect to the replica database at this endpoint"
}

output "replica_port" {
  value       = module.mysql_replica.port
  description = "The port the replica database is listening on"
}

output "replica_arn" {
  value       = module.mysql_replica.arn
  description = "The ARN of the replica database"
}
```

And now you're finally ready to deploy! Note that running `apply` to spin up a primary and replica can take a long time, some 20–30 minutes, so be patient:

```
$ terraform apply

(...)

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:
primary_address = "terraform-up-and-running.cmyd6qwb.us-east-2.rds.amazonaws.com"
primary_arn     = "arn:aws:rds:us-east-2:111111111111:db:terraform-up-and-running"
primary_port    = 3306
replica_address = "terraform-up-and-running.drctpdoe.us-west-1.rds.amazonaws.com"
replica_arn     = "arn:aws:rds:us-west-1:111111111111:db:terraform-up-and-running"
replica_port    = 3306
```

And there you have it, cross-region replication! You can log into the RDS Console to confirm replication is working. As shown in Figure 7-2, you should see a primary in `us-east-2` and a replica in `us-west-1`.

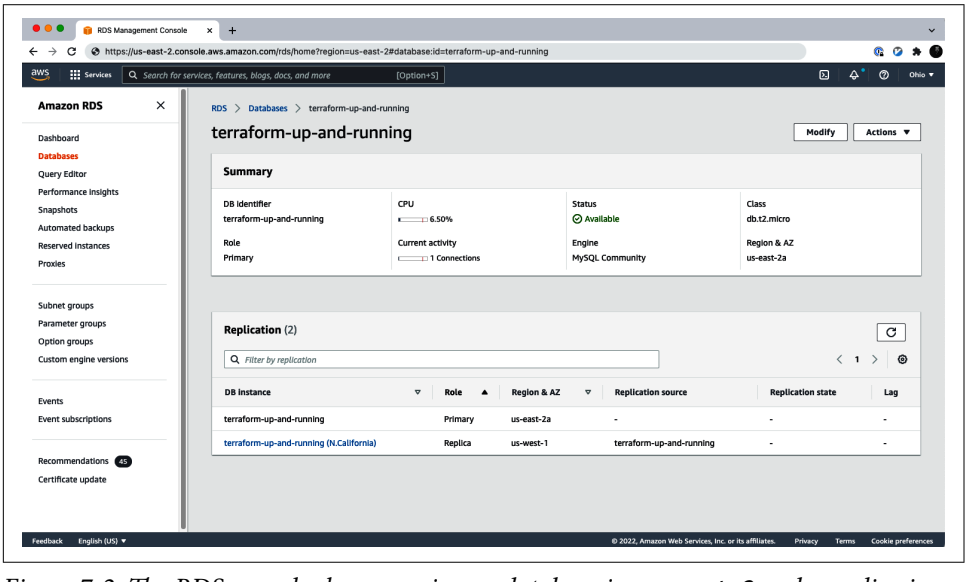


Figure 7-2. The RDS console shows a primary database in `us-east-2` and a replica in `us-west-1`.

As an exercise for the reader, I leave it up to you to update the staging environment (*stage/data-stores/mysql*) to use your `mysql` module (*modules/data-stores/mysql*) as well, but to configure it *without* replication, as you don't usually need that level of availability in pre-production environments.

As you can see in these examples, by using multiple providers with aliases, deploying resources across multiple regions with Terraform is pretty easy. However, I want to give two warnings before moving on:

Warning 1: Multiregion is hard

To run infrastructure in multiple regions around the world, especially in "active-active" mode, where more than one region is actively responding to user requests at the same time (as opposed to one region being a standby), there are many hard problems to solve, such as dealing with latency between regions, deciding between one writer (which means you have lower availability and higher latency) or multiple writers (which means you have either eventual consistency or sharding), figuring out how to generate unique IDs (the standard auto increment ID in most databases no longer suffices), working to meet local data regulations, and so on. These challenges are all beyond the scope of the book, but I figured I'd at least

mention them to make it clear that multiregion deployments in the real world are not just a matter of tossing a few provider aliases into your Terraform code!

Warning 2: Use aliases sparingly

Although it's easy to use aliases with Terraform, I would caution against using them too often, *especially* when setting up multiregion infrastructure. One of the main reasons to set up multiregion infrastructure is so you can be resilient to the outage of one region: e.g., if `us-east-2` goes down, your infrastructure in `us-west-1` can keep running. But if you use a single Terraform module that uses aliases to deploy into both regions, then when one of those regions is down, the module will not be able to connect to that region, and any attempt to run `plan` or `apply` will fail. So right when you need to roll out changes—when there's a major outage—your Terraform code will stop working.

More generally, as discussed in Chapter 3, you should keep environments completely isolated: so instead of managing multiple regions in one module with aliases, you manage each region in separate modules. That way, you minimize the blast radius, both from your own mistakes (e.g., if you accidentally break something in one region, it's less likely to affect the other) and from problems in the world itself (e.g., an outage in one region is less likely to affect the other).

So when does it make sense to use aliases? Typically, aliases are a good fit when the infrastructure you're deploying across several aliased providers is truly coupled and you want to always deploy it together. For example, if you wanted to use Amazon CloudFront as a CDN (Content Distribution Network), and to provision a TLS certificate for it using AWS Certification Manager (ACM), then AWS requires the certificate to be created in the `us-east-1` region, no matter what other regions you happen to be using for CloudFront itself. In that case, your code may have two `provider` blocks, one for the primary region you want to use for CloudFront and one with an `alias` hardcoded specifically to `us-east-1` for configuring the TLS certificate. Another use case for aliases is if you're deploying resources designed for use across many regions: for example, AWS recommends deploying GuardDuty, an automated threat detection service, in every single region you're using in your AWS account. In this case, it may make sense to have a module with a `provider` block and custom `alias` for each AWS region.

Beyond a few corner cases like this, using aliases to handle multiple regions is relatively rare. A more common use case for aliases is when you have multiple providers that need to authenticate in different ways, such as each one authenticating to a different AWS account.

Working with Multiple AWS Accounts

So far, throughout this book, you've likely been using a single AWS account for all of your infrastructure. For production code, it's more common to use multiple AWS accounts: e.g., you put your staging environment in a stage account, your production environment in a prod account, and so on. This concept applies to other clouds too, such as Azure and Google Cloud. Note that I'll be using the term *account* in this book, even though some clouds use slightly different terminology for the same concept (e.g., Google Cloud calls them *projects* instead of accounts).

The main reasons for using multiple accounts are as follows:

Isolation (aka compartmentalization)

You use separate accounts to isolate different environments from each other and to limit the "blast radius" when things go wrong. For example, putting your staging and production environments in separate accounts ensures that if an attacker manages to break into staging, they still have no access whatsoever to production. Likewise, this isolation ensures that a developer making changes in staging is less likely to accidentally break something in production.

Authentication and authorization

If everything is in one account, it's tricky to grant access to some things (e.g., the staging environment) but not accidentally grant access to other things (e.g., the production environment). Using multiple accounts makes it easier to have fine-grained control, as any permissions you grant in one account have no effect on any other account.

The authentication requirements of multiple accounts also help reduce the chance of mistakes. With everything in a single account, it's too easy to make the mistake where you think you're making a change in, say, your staging environment, but you're actually making the change in production (which can be a disaster if the change you're making is, for example, to drop all database tables). With multiple accounts, this is less likely, as authenticating to each account requires a separate set of steps.

Note that having multiple accounts does *not* imply that developers have multiple separate user profiles (e.g., a separate IAM user in each AWS account). In fact, that would be an antipattern, as that would require managing multiple sets of credentials, permissions, etc. Instead, you can configure just about all the major clouds so that each developer has exactly one user profile, which they can use to authenticate to any account they have access to. The cross-account authentication mechanism varies depending on the cloud you're using: e.g., in AWS, you can authenticate across AWS accounts by assuming IAM roles, as you'll see shortly.

Auditing and reporting

A properly configured account structure will allow you to maintain an audit trail of all the changes happening in all your environments, check if you're adhering to compliance requirements, and detect anomalies. Moreover, you'll be able to have consolidated billing, with all the charges for all of your accounts in one place, including cost breakdowns by account, service, tag, etc. This is especially useful in large organizations, as it allows finance to track and budget spending by team simply by looking at which account the charges are coming from.

Let's go through a multi-account example with AWS. First, you'll want to create a new AWS account to use for testing. Since you already have one AWS account, to create new *child accounts*, you can use AWS Organizations, which ensures that the billing from all the child accounts rolls up into the parent account (sometimes called the *root account*) and gives you a dashboard you can use to manage all the child accounts.

Head over to the AWS Organizations Console, and click the "Add an AWS account" button, as shown in Figure 7-3.

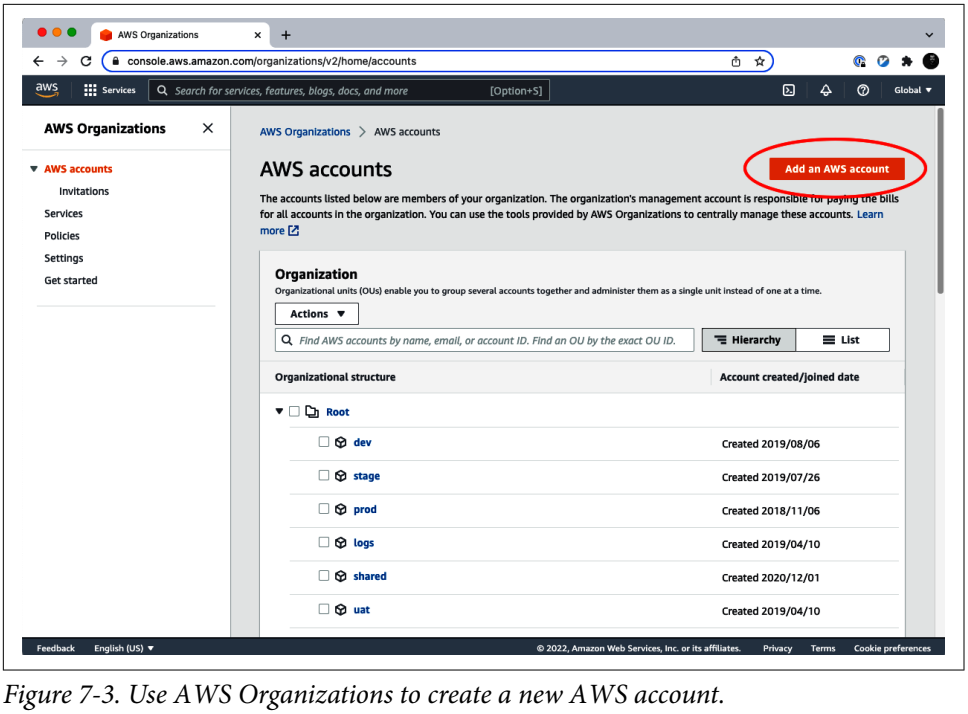


Figure 7-3. Use AWS Organizations to create a new AWS account.

On the next page, fill in the following info, as shown in Figure 7-4:

AWS account name

The name to use for the account. For example, if this account was going to be used for your staging environment, you might name it "staging."

Email address of the account's owner

The email address to use for the root user of the AWS account. Note that every AWS account must use a different email address for the root user, so you can't reuse the email address you used to create your first (root) AWS account (see "How to Get Multiple Aliases from One Email Address" on page 241 for a workaround). So what about the root user's password? By default, AWS does not configure a password for the root user of a new child account (you'll see shortly an alternative way to authenticate to the child account). If you ever do want to log in as this root user, after you create the child account, you'll need to go through the password reset flow with the email address you're specifying here.

IAM role name

When AWS Organizations creates a child AWS account, it automatically creates an IAM role within that child AWS account that has admin permissions and can be assumed from the parent account. This is convenient, as it allows you to authenticate to the child AWS account without having to create any IAM users or IAM roles yourself. I recommend leaving this IAM role name at the default value of `OrganizationAccountAccessRole`.

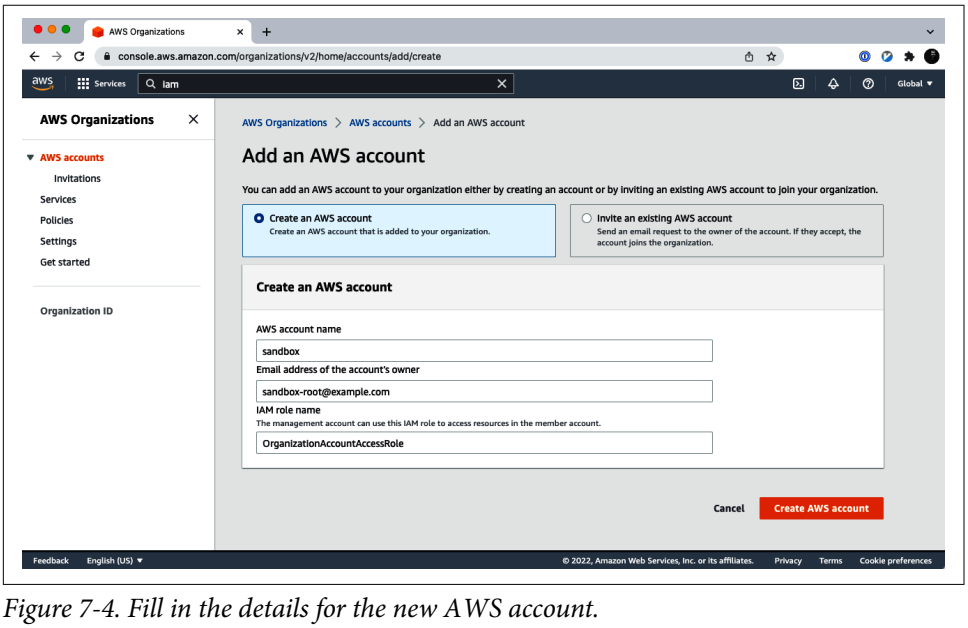


Figure 7-4. Fill in the details for the new AWS account.

How to Get Multiple Aliases from One Email Address

If you use Gmail, you can get multiple email aliases out of a single address by taking advantage of the fact that Gmail ignores everything after a plus sign in an email address. For example, if your Gmail address is *example@gmail.com*, you can send email to *example+foo@gmail.com* and *example+any-text-you-want@gmail.com*, and all of those emails will go to *example@gmail.com*. This also works if your company uses Gmail via Google Workspace, even with a custom domain: e.g., *example+dev@company.com* and *example+stage@company.com* will all go to *example@company.com*.

This is useful if you're creating a dozen child AWS accounts, as instead of having to create a dozen totally separate email addresses, you could use *example+dev@company.com* for your dev account, *example+stage@company.com* for your stage account, and so on; AWS will see each of those email addresses as a different, unique address, but under the hood, all the emails will go to the same account.

Click the Create AWS Account button, wait a few minutes for AWS to create the account, and then jot down the 12-digit ID of the AWS account that gets created. For the rest of this chapter, let's assume the following:

- Parent AWS account ID: `111111111111`
- Child AWS account ID: `222222222222`

You can authenticate to your new child account from the AWS Console by clicking your username and selecting "Switch role", as shown in Figure 7-5.

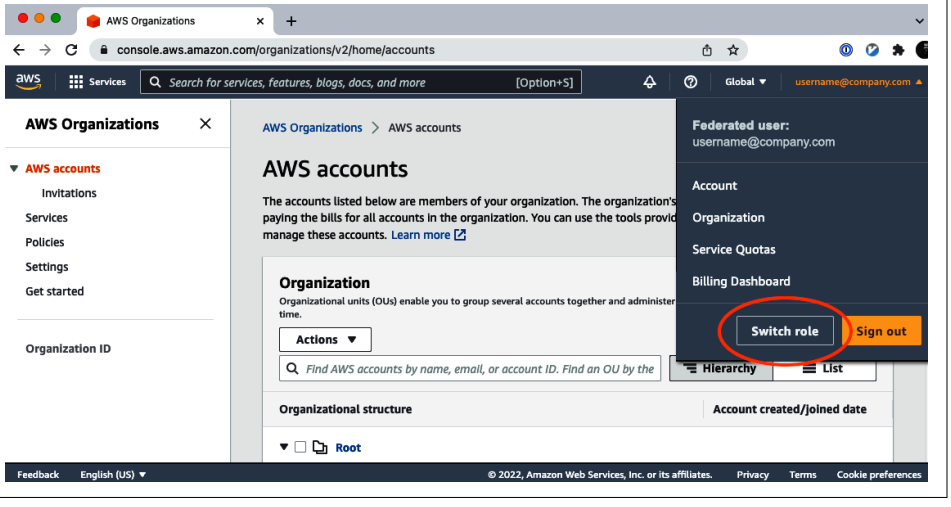


Figure 7-5. Select the "Switch role" button.

Next, enter the details for the IAM role you want to assume, as shown in Figure 7-6:

Account
: The 12-digit ID of the AWS account to switch to. You'll want to enter the ID of your new child account.

Role
: The name of the IAM role to assume in that AWS account. Enter the name you used for the IAM role when creating the new child account, which is `OrganizationAccountAccessRole` by default.

Display name
: AWS will create a shortcut in the nav to allow you to switch to this account in the future with a single click. This is the name to show in this shortcut. It only affects your IAM user in this browser.

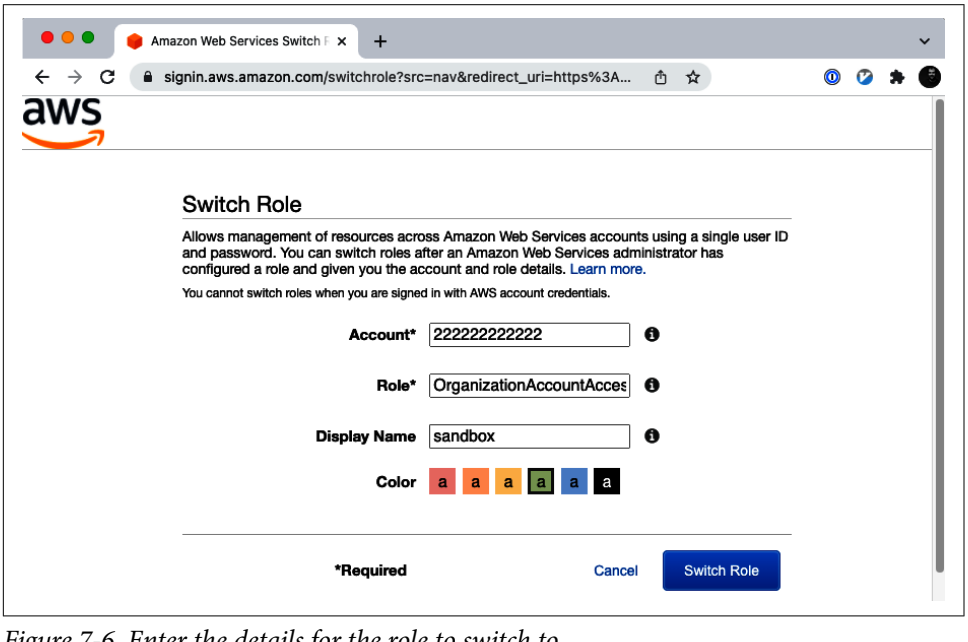


Figure 7-6. Enter the details for the role to switch to.

Click Switch Role and, voilà, AWS should log you into the web console of the new AWS account!

Let's now write an example Terraform module in *examples/multi-account-root* that can authenticate to multiple AWS accounts. Just as with the multiregion AWS example, you will need to add two `provider` blocks in *main.tf*, each with a different alias. First, the `provider` block for the parent AWS account:

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

Next, the `provider` block for the child AWS account:

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

To be able to authenticate to the child AWS account, you'll assume an IAM role. In the web console, you did this by clicking the Switch Role button; in your Terraform code, you do this by adding an `assume_role` block to the child `provider` block:

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

  assume_role {
    role_arn = "arn:aws:iam::<ACCOUNT_ID>:role/<ROLE_NAME>"
  }
}
```

In the `role_arn` parameter, you'll need to replace `ACCOUNT_ID` with your child account ID and `ROLE_NAME` with the name of the IAM role in that account, just as you did when switching roles in the web console. Here's what it looks like with the account ID `222222222222` and role name `OrganizationAccountAccessRole` plugged in:

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

  assume_role {
    role_arn = "arn:aws:iam::222222222222:role/OrganizationAccountAccessRole"
  }
}
```

Now, to check this is actually working, add two `aws_caller_identity` data sources, and configure each one to use a different provider:

```
data "aws_caller_identity" "parent" {
  provider = aws.parent
}

data "aws_caller_identity" "child" {
  provider = aws.child
}
```

Finally, add output variables in *outputs.tf* to print out the account IDs:

```
output "parent_account_id" {
  value       = data.aws_caller_identity.parent.account_id
  description = "The ID of the parent AWS account"
}

output "child_account_id" {
  value       = data.aws_caller_identity.child.account_id
  description = "The ID of the child AWS account"
}
```

Run `apply`, and you should see the different IDs for each account:

```
$ terraform apply

(...)

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

Outputs:

parent_account_id = "111111111111"
child_account_id = "222222222222"
```

And there you have it: by using provider aliases and `assume_role` blocks, you now know how to write Terraform code that can operate across multiple AWS accounts.

As with the multiregion section, a few warnings:

Warning 1: Cross-account IAM roles are double opt-in

In order for an IAM role to allow access from one AWS account to another—e.g., to allow an IAM role in account `222222222222` to be assumed from account `111111111111`—you need to grant permissions in *both* AWS accounts:

- First, in the AWS account where the IAM role lives (e.g., the child account `222222222222`), you must configure its assume role policy to trust the other AWS account (e.g., the parent account `111111111111`). This happened magically for you with the `OrganizationAccountAccessRole` IAM role because AWS Organizations automatically configures the assume role policy of this IAM role to trust the parent account. However, for any custom IAM roles you create, you need to remember to explicitly grant the `sts:AssumeRole` permission yourself.
- Second, in the AWS account from which you assume the role (e.g., the parent account `111111111111`), you must *also* grant your user permissions to assume that IAM role. Again, this happened for you magically because, in Chapter 2, you gave your IAM user `AdministratorAccess`, which gives you permissions to do just about everything in the parent AWS account, including assuming IAM roles. In most real-world use cases, your user won't

be (shouldn't be!) an admin, so you'll need to explicitly grant your user `sts:AssumeRole` permissions on the IAM role(s) you want to be able to assume.

Warning 2: Use aliases sparingly

I said this in the multiregion example, but it bears repeating: although it's easy to use aliases with Terraform, I would caution against using them too often, including with multi-account code. Typically, you use multiple accounts to create separation between them, so if something goes wrong in one account, it doesn't affect the other. Modules that deploy across multiple accounts go against this principle. Only do it when you *intentionally* want to have resources in multiple accounts coupled and deployed together.

Creating Modules That Can Work with Multiple Providers

When working with Terraform modules, you typically work with two types of modules:

Reusable modules

These are low-level modules that are not meant to be deployed directly but instead are to be combined with other modules, resources, and data sources.

Root modules

These are high-level modules that combine multiple reusable modules into a single unit that is meant to be deployed directly by running `apply` (in fact, the definition of a root module is it's the one on which you run `apply`).

The multiprovider examples you've seen so far have put all the `provider` blocks into the root module. What do you do if you want to create a reusable module that works with multiple providers? For example, what if you wanted to turn the multi-account code from the previous section into a reusable module? As a first step, you might put all that code, unchanged, into the *modules/multi-account* folder. Then, you could create a new example to test it with in the *examples/multi-account-module* folder, with a *main.tf* that looks like this:

```
module "multi_account_example" {
  source = "../../modules/multi-account"
}
```

If you run `apply` on this code, it'll work, but there is a problem: all of the `provider` configuration is now hidden in the module itself (in *modules/multi-account*). Defining `provider` blocks within reusable modules is an antipattern for several reasons:

Configuration problems

If you have `provider` blocks defined in your reusable module, then that module controls all the configuration for that `provider`. For example, the IAM role ARN

and regions to use are currently hardcoded in the *modules/multi-account* module. You could, of course, expose input variables to allow users to set the regions and IAM role ARNs, but that's only the tip of the iceberg. If you browse the AWS Provider documentation, you'll find that there are roughly *50* different configuration options you can pass into it! Many of these parameters are going to be important for users of your module, as they control how to authenticate to AWS, what region to use, what account (or IAM role) to use, what endpoints to use when talking to AWS, what tags to apply or ignore, and much more. Having to expose 50 extra variables in a module will make that module very cumbersome to maintain and use.

Duplication problems
: Even if you expose those 50 settings in your module, or whatever subset you believe is important, you're creating code duplication for users of your module. That's because it's common to combine multiple modules together, and if you have to pass in some subset of 50 settings into each of those modules in order to get them to all authenticate correctly, you're going to have to copy and paste a lot of parameters, which is tedious and error prone.

Performance problems
: Every time you include a `provider` block in your code, Terraform spins up a new process to run that provider, and communicates with that process via RPC. If you have a handful of `provider` blocks, this works just fine, but as you scale up, it may cause performance problems. Here's a real-world example: a few years ago, I created reusable modules for CloudTrail, AWS Config, GuardDuty, IAM Access Analyzer, and Macie. Each of these AWS services is supposed to be deployed into every region in your AWS account, and as AWS had ~25 regions, I included 25 `provider` blocks in each of these modules. I then created a single root module to deploy all of these as a "baseline" in my AWS accounts: if you do the math, that's 5 modules with 25 `provider` blocks each, or 125 `provider` blocks total. When I ran `apply`, Terraform would fire up 125 processes, each making hundreds of API and RPC calls. With thousands of concurrent network requests, my CPU would start thrashing, and a single `plan` could take 20 minutes. Worse yet, this would sometimes overload the network stack, leading to intermittent failures in API calls, and `apply` would fail with sporadic errors.

Therefore, as a best practice, you should *not* define any `provider` blocks in your reusable modules and instead allow your users to create the `provider` blocks they need solely in their root modules. But then, how do you build a module that can work with multiple providers? If the module has no `provider` blocks in it, how do you define provider aliases that you can reference in your resources and data sources?

The solution is to use *configuration aliases*. These are very similar to the provider aliases you've seen already, except they aren't defined in a `provider` block. Instead, you define them in a `required_providers` block.

Open up *modules/multi-account/main.tf*, remove the nested `provider` blocks, and replace them with a `required_providers` block with configuration aliases as follows:

```
terraform {
  required_providers {
    aws = {
      source                = "hashicorp/aws"
      version               = "~> 4.0"
      configuration_aliases = [aws.parent, aws.child]
    }
  }
}
```

Just as with normal provider aliases, you can pass configuration aliases into resources and data sources using the `provider` parameter:

```
data "aws_caller_identity" "parent" {
  provider = aws.parent
}

data "aws_caller_identity" "child" {
  provider = aws.child
}
```

The key difference from normal provider aliases is that configuration aliases don't create any providers themselves; instead, they force users of your module to explicitly pass in a provider for each of your configuration aliases using a `providers` map.

Open up *examples/multi-account-module/main.tf*, and define the `provider` blocks as before:

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

provider "aws" {
  region = "us-east-2"
  alias  = "child"

  assume_role {
    role_arn = "arn:aws:iam::222222222222:role/OrganizationAccountAccessRole"
  }
}
```

And now you can pass them into the *modules/multi-account* module as follows:

```
module "multi_account_example" {
  source = "../../modules/multi-account"
```

```
    providers = {
      aws.parent = aws.parent
      aws.child  = aws.child
    }
  }
```

The keys in the `providers` map must match the names of the configuration aliases within the module; if any of the names from configuration aliases are missing in the `providers` map, Terraform will show an error. This way, when you're building a reusable module, you can define what providers that module needs, and Terraform will ensure users pass those providers in; and when you're building a root module, you can define your `provider` blocks just once and pass around references to them to the reusable modules you depend on.

Working with Multiple Different Providers

You've now seen how to work with multiple providers when all of them are the same type of provider: e.g., multiple copies of the `aws` provider. This section talks about how to work with multiple different providers.

Readers of the first two editions of this book often asked for examples of using multiple clouds together (*multicloud*), but I couldn't find much useful to share. In part, this is because using multiple clouds is usually a bad practice,[2] but even if you're forced to do it (most large companies are multicloud, whether they want to be or not), it's rare to manage multiple clouds in a single module for the same reason it's rare to manage multiple regions or accounts in a single module. If you're using multiple clouds, you're far better off managing each one in a separate module.

Moreover, translating every single AWS example in the book into the equivalent solutions for other clouds (Azure and Google Cloud) is impractical: the book would end up way too long, and while you would learn more about each cloud, you wouldn't learn any new Terraform concepts along the way, which is the real goal of the book. If you do want to see examples of what the Terraform code for similar infrastructure looks like across different clouds, have a look at the *examples* folder in the Terratest repo. As you'll see in Chapter 9, Terratest provides a set of tools for writing automated tests for different types of infrastructure code and different types of clouds, so in the *examples* folder you'll find Terraform code for similar infrastructure in AWS, Google Cloud, and Azure, including individual servers, groups of servers, databases, and more. You'll also find automated tests for all those examples in the *test* folder.

In this book, instead of an unrealistic multicloud example, I decided to instead show you how to use multiple providers together in a slightly more realistic scenario (one

2 See "Multi-Cloud is the Worst Practice".

that was also requested by many readers of the first two editions): namely, how to use the AWS Provider with the Kubernetes provider to deploy Dockerized apps. Kubernetes is, in many ways, a cloud of its own—it can run applications, networks, data stores, load balancers, secret stores, and much more—so, in a sense, this is both a multiprovider and multicloud example. And because Kubernetes is a cloud, that means there is a lot to learn, so I'm going to have to build up to it one step at a time, starting with mini crash courses on Docker and Kubernetes, before finally moving on to the full multiprovider example that uses both AWS and Kubernetes:

- A crash course on Docker
- A crash course on Kubernetes
- Deploying Docker containers in AWS using Elastic Kubernetes Service (EKS)

A Crash Course on Docker

As you may remember from Chapter 1, Docker images are like self-contained "snapshots" of the operating system (OS), the software, the files, and all other relevant details. Let's now see Docker in action.

First, if you don't have Docker installed already, follow the instructions on the Docker website to install Docker Desktop for your operating system. Once it's installed, you should have the `docker` command available on your command line. You can use the `docker run` command to run Docker images locally:

```
$ docker run <IMAGE> [COMMAND]
```

where `IMAGE` is the Docker image to run and `COMMAND` is an optional command to execute. For example, here's how you can run a Bash shell in an Ubuntu 20.04 Docker image (note that the following command includes the `-it` flag so you get an interactive shell where you can type):

```
$ docker run -it ubuntu:20.04 bash

Unable to find image 'ubuntu:20.04' locally
20.04: Pulling from library/ubuntu
Digest: sha256:669e010b58baf5beb2836b253c1fd5768333f0d1dbcb834f7c07a4dc93f474be
Status: Downloaded newer image for ubuntu:20.04

root@d96ad3779966:/#
```

And voilà, you're now in Ubuntu! If you've never used Docker before, this can seem fairly magical. Try running some commands. For example, you can look at the contents of */etc/os-release* to verify you really are in Ubuntu:

```
root@d96ad3779966:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.3 LTS (Focal Fossa)"
```

```
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04.3 LTS"
VERSION_ID="20.04"
VERSION_CODENAME=focal
```

How did this happen? Well, first, Docker searches your local filesystem for the `ubuntu:20.04` image. If you don't have that image downloaded already, Docker downloads it automatically from Docker Hub, which is a *Docker Registry* that contains shared Docker images. The `ubuntu:20.04` image happens to be a public Docker image—an official one maintained by the Docker team—so you're able to download it without any authentication. However, it's also possible to create private Docker images that only certain authenticated users can use.

Once the image is downloaded, Docker runs the image, executing the `bash` command, which starts an interactive Bash prompt, where you can type. Try running the `ls` command to see the list of files:

```
root@d96ad3779966:/# ls -al
total 56
drwxr-xr-x   1 root root 4096 Feb 22 14:22 .
drwxr-xr-x   1 root root 4096 Feb 22 14:22 ..
lrwxrwxrwx   1 root root    7 Jan 13 16:59 bin -> usr/bin
drwxr-xr-x   2 root root 4096 Apr 15  2020 boot
drwxr-xr-x   5 root root  360 Feb 22 14:22 dev
drwxr-xr-x   1 root root 4096 Feb 22 14:22 etc
drwxr-xr-x   2 root root 4096 Apr 15  2020 home
lrwxrwxrwx   1 root root    7 Jan 13 16:59 lib -> usr/lib
drwxr-xr-x   2 root root 4096 Jan 13 16:59 media
(...)
```

You might notice that's not your filesystem. That's because Docker images run in containers that are isolated at the userspace level: when you're in a container, you can only see the filesystem, memory, networking, etc., in that container. Any data in other containers, or on the underlying host operating system, is not accessible to you, and any data in your container is not visible to those other containers or the underlying host operating system. This is one of the things that makes Docker useful for running applications: the image format is self-contained, so Docker images run the same way no matter where you run them, and no matter what else is running there.

To see this in action, write some text to a *test.txt* file as follows:

```
root@d96ad3779966:/# echo "Hello, World!" > test.txt
```

Next, exit the container by hitting Ctrl-D on Windows and Linux or Cmd-D on macOS, and you should be back in your original command prompt on your underlying host OS. If you try to look for the *test.txt* file you just wrote, you'll see that it doesn't exist: the container's filesystem is totally isolated from your host OS.

Now, try running the same Docker image again:

```
$ docker run -it ubuntu:20.04 bash
root@3e0081565a5d:/#
```

Notice that this time, since the `ubuntu:20.04` image is already downloaded, the container starts almost instantly. This is another reason Docker is useful for running applications: unlike virtual machines, containers are lightweight, boot up quickly, and incur little CPU or memory overhead.

You may also notice that the second time you fired up the container, the command prompt looked different. That's because you're now in a totally new container; any data you wrote in the previous one is no longer accessible to you. Run `ls -al` and you'll see that the *test.txt* file does not exist. Containers are isolated not only from the host OS but also from each other.

Hit Ctrl-D or Cmd-D again to exit the container, and back on your host OS, run the `docker ps -a` command:

```
$ docker ps -a
CONTAINER ID   IMAGE            COMMAND    CREATED          STATUS
3e0081565a5d   ubuntu:20.04     "bash"     5 min ago    Exited (0) 16 sec ago
d96ad3779966   ubuntu:20.04     "bash"     14 min ago   Exited (0) 5 min ago
```

This will show you all the containers on your system, including the stopped ones (the ones you exited). You can start a stopped container again by using the `docker start <ID>` command, setting `ID` to an ID from the `CONTAINER ID` column of the `docker ps` output. For example, here is how you can start the first container up again (and attach an interactive prompt to it via the `-ia` flags):

```
$ docker start -ia d96ad3779966
root@d96ad3779966:/#
```

You can confirm this is really the first container by outputting the contents of *test.txt*:

```
root@d96ad3779966:/# cat test.txt
Hello, World!
```

Let's now see how a container can be used to run a web app. Hit Ctrl-D or Cmd-D again to exit the container, and back on your host OS, run a new container:

```
$ docker run training/webapp
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

The training/webapp image contains a simple Python "Hello, World" web app for testing. When you run the image, it fires up the web app, listening on port 5000 by

default. However, if you open a new terminal on your host operating system and try to access the web app, it won't work:

```
$ curl localhost:5000
curl: (7) Failed to connect to localhost port 5000: Connection refused
```

What's the problem? Actually, it's not a problem but a feature! Docker containers are isolated from the host operating system and other containers, not only at the filesystem level but also in terms of networking. So while the container really is listening on port 5000, that is only on a port *inside* the container, which isn't accessible on the host OS. If you want to expose a port from the container on the host OS, you have to do it via the `-p` flag.

First, hit Ctrl-C to shut down the `training/webapp` container: note that it's C this time, not D, and it's Ctrl regardless of OS, as you're shutting down a process, rather than exiting an interactive prompt. Now rerun the container but this time with the `-p` flag as follows:

```
$ docker run -p 5000:5000 training/webapp
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Adding `-p 5000:5000` to the command tells Docker to expose port 5000 inside the container on port 5000 of the host OS. In another terminal on your host OS, you should now be able to see the web app working:

```
$ curl localhost:5000
Hello world!
```

Cleaning Up Containers

Every time you run `docker run` and exit, you are leaving behind containers, which take up disk space. You may wish to clean them up with the `docker rm <CONTAINER_ID>` command, where `CONTAINER_ID` is the ID of the container from the `docker ps` output. Alternatively, you could include the `--rm` flag in your `docker run` command to have Docker automatically clean up when you exit the container.

A Crash Course on Kubernetes

Kubernetes is an orchestration tool for Docker, which means it's a platform for running and managing Docker containers on your servers, including scheduling (picking which servers should run a given container workload), auto healing (automatically redeploying containers that failed), auto scaling (scaling the number of containers up and down in response to load), load balancing (distributing traffic across containers), and much more.

Under the hood, Kubernetes consists of two main pieces:

Control plane

The control plane is responsible for managing the Kubernetes cluster. It is the "brains" of the operation, responsible for storing the state of the cluster, monitoring containers, and coordinating actions across the cluster. It also runs the API server, which provides an API you can use from command-line tools (e.g., `kubectl`), web UIs (e.g., the Kubernetes Dashboard), and IaC tools (e.g., Terraform) to control what's happening in the cluster.

Worker nodes

The worker nodes are the servers used to actually run your containers. The worker nodes are entirely managed by the control plane, which tells each worker node what containers it should run.

Kubernetes is open source, and one of its strengths is that you can run it anywhere: in any public cloud (e.g., AWS, Azure, Google Cloud), in your own datacenter, and even on your own developer workstation. A little later in this chapter, I'll show you how you can run Kubernetes in the cloud (in AWS), but for now, let's start small and run it locally. This is easy to do if you installed a relatively recent version of Docker Desktop, as it has the ability to fire up a Kubernetes cluster locally with just a few clicks.

If you open Docker Desktop's preferences on your computer, you should see Kubernetes in the nav, as shown in Figure 7-7.

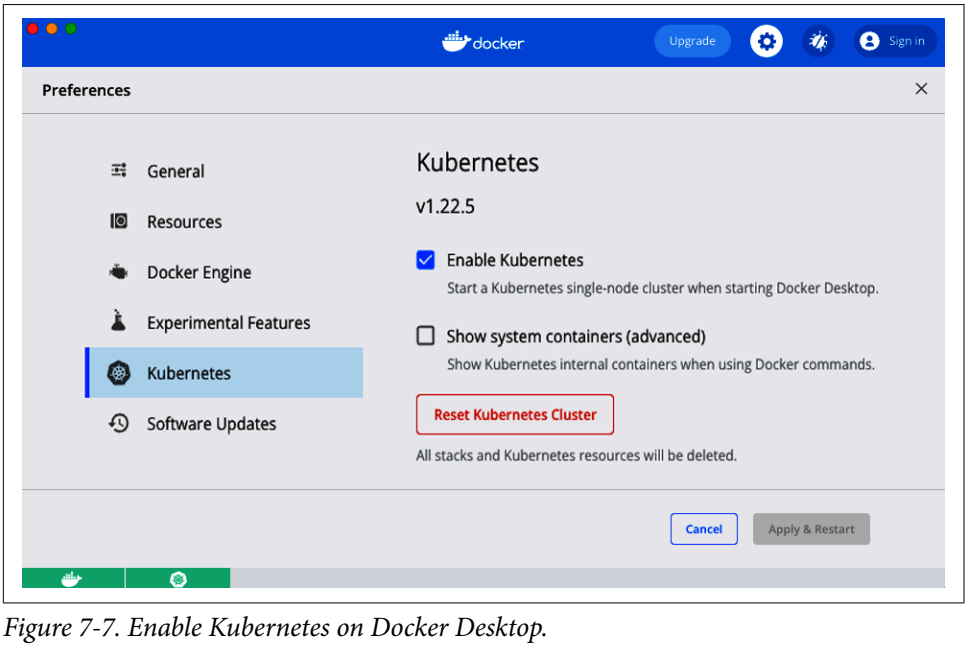


Figure 7-7. Enable Kubernetes on Docker Desktop.

If it's not enabled already, check the Enable Kubernetes checkbox, click Apply & Restart, and wait a few minutes for that to complete. In the meantime, follow the instructions on the Kubernetes website to install `kubectl`, which is the command-line tool for interacting with Kubernetes.

To use `kubectl`, you must first update its configuration file, which lives in *$HOME/.kube/config* (that is, the *.kube* folder of your home directory), to tell it what Kubernetes cluster to connect to. Conveniently, when you enable Kubernetes in Docker Desktop, it updates this config file for you, adding a `docker-desktop` entry to it, so all you need to do is tell `kubectl` to use this configuration as follows:

```
$ kubectl config use-context docker-desktop
Switched to context "docker-desktop".
```

Now you can check if your Kubernetes cluster is working with the `get nodes` command:

```
$ kubectl get nodes
NAME             STATUS   ROLES                  AGE   VERSION
docker-desktop   Ready    control-plane,master   95m   v1.22.5
```

The `get nodes` command shows you information about all the nodes in your cluster. Since you're running Kubernetes locally, your computer is the only node, and it's running both the control plane and acting as a worker node. You're now ready to run some Docker containers!

To deploy something in Kubernetes, you create Kubernetes *objects*, which are persistent entities you write to the Kubernetes cluster (via the API server) that record your intent: e.g., your intent to have specific Docker images running. The cluster runs a *reconciliation loop*, which continuously checks the objects you stored in it and works to make the state of the cluster match your intent.

There are many different types of Kubernetes objects available. For the examples in this book, let's use the following two objects:

Kubernetes Deployment

A *Kubernetes Deployment* is a declarative way to manage an application in Kubernetes. You declare what Docker images to run, how many copies of them to run (called *replicas*), a variety of settings for those images (e.g., CPU, memory, port numbers, environment variables), and the strategy to roll out updates to those images, and the Kubernetes Deployment will then work to ensure that the requirements you declared are always met. For example, if you specified you wanted three replicas, but one of the worker nodes went down so only two replicas are left, the Deployment will automatically spin up a third replica on one of the other worker nodes.

Kubernetes Service

A *Kubernetes Service* is a way to expose a web app running in Kubernetes as a networked service. For example, you can use a Kubernetes Service to configure a load balancer that exposes a public endpoint and distributes traffic from that endpoint across the replicas in a Kubernetes Deployment.

The idiomatic way to interact with Kubernetes is to create YAML files describing what you want—e.g., one YAML file that defines the Kubernetes Deployment and another one that defines the Kubernetes Service—and to use the `kubectl apply` command to submit those objects to the cluster. However, using raw YAML has drawbacks, such as a lack of support for code reuse (e.g., variables, modules), abstraction (e.g., loops, if-statements), clear standards on how to store and manage the YAML files (e.g., to track changes to the cluster over time), and so on. Therefore, many Kubernetes users turn to alternatives, such as Helm or Terraform. Since this is a book on Terraform, I'm going to show you how to create a Terraform module called `k8s-app` (K8S is an acronym for Kubernetes in the same way that I18N is an acronym for internationalization) that deploys an app in Kubernetes using a Kubernetes Deployment and Kubernetes Service.

Create a new module in the *modules/services/k8s-app* folder. Within that folder, create a *variables.tf* file that defines the module's API via the following input variables:

```
variable "name" {
  description = "The name to use for all resources created by this module"
  type        = string
}

variable "image" {
  description = "The Docker image to run"
  type        = string
}

variable "container_port" {
  description = "The port the Docker image listens on"
  type        = number
}

variable "replicas" {
  description = "How many replicas to run"
  type        = number
}

variable "environment_variables" {
  description = "Environment variables to set for the app"
  type        = map(string)
  default     = {}
}
```

This should give you just about all the inputs you need for creating the Kubernetes Deployment and Service. Next, add a *main.tf* file, and at the top, add the `required_providers` block to it with the Kubernetes provider:

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = "~> 2.0"
    }
  }
}
```

Hey, a new provider, neat! OK, let's make use of that provider to create a Kubernetes Deployment by using the `kubernetes_deployment` resource:

```
resource "kubernetes_deployment" "app" {
}
```

There are quite a few settings to configure within the `kubernetes_deployment` resource, so let's go through them one at a time. First, you need to configure the `metadata` block:

```
resource "kubernetes_deployment" "app" {
  metadata {
    name = var.name
  }
}
```

Every Kubernetes object includes metadata that can be used to identify and target that object in API calls. In the preceding code, I'm setting the Deployment name to the `name` input variable.

The rest of the configuration for the `kubernetes_deployment` resource goes into the `spec` block:

```
resource "kubernetes_deployment" "app" {
  metadata {
    name = var.name
  }

  spec {
  }
}
```

The first item to put into the `spec` block is to specify the number of replicas to create:

```
  spec {
    replicas = var.replicas
  }
```

Next, define the `template` block:

```
spec {
  replicas = var.replicas

  template {
  }
}
```

In Kubernetes, instead of deploying one container at a time, you deploy *Pods*, which are groups of containers that are meant to be deployed together. For example, you could have a Pod with one container to run a web app (e.g., the Python app you saw earlier) and another container that gathers metrics on the web app and sends them to a central service (e.g., Datadog). The `template` block is where you define the *Pod Template*, which specifies what container(s) to run, the ports to use, environment variables to set, and so on.

One important ingredient in the Pod Template will be the labels to apply to the Pod. You'll need to reuse these labels in several places—e.g., the Kubernetes Service uses labels to identify the Pods that need to be load balanced—so let's define those labels in a local variable called `pod_labels`:

```
locals {
  pod_labels = {
    app = var.name
  }
}
```

And now use `pod_labels` in the `metadata` block of the Pod Template:

```
spec {
  replicas = var.replicas

  template {
    metadata {
      labels = local.pod_labels
    }
  }
}
```

Next, add a `spec` block inside of `template`:

```
spec {
  replicas = var.replicas

  template {
    metadata {
      labels = local.pod_labels
    }

    spec {
      container {
```

```
            name  = var.name
            image = var.image

            port {
              container_port = var.container_port
            }

            dynamic "env" {
              for_each = var.environment_variables
              content {
                name  = env.key
                value = env.value
              }
            }
          }
        }
      }
    }
```

There's a lot here, so let's go through it one piece at a time:

`container`
: Inside the `spec` block, you can define one or more `container` blocks to specify which Docker containers to run in this Pod. To keep this example simple, there's just one `container` block in the Pod. The rest of these items are all within this `container` block.

`name`
: The name to use for the container. I've set this to the `name` input variable.

`image`
: The Docker image to run in the container. I've set this to the `image` input variable.

`port`
: The ports to expose in the container. To keep the code simple, I'm assuming the container only needs to listen on one port, set to the `container_port` input variable.

`env`
: The environment variables to expose to the container. I'm using a `dynamic` block with `for_each` (two concepts you may remember from Chapter 5) to set this to the variables in the `environment_variables` input variable.

OK, that wraps up the Pod Template. There's just one thing left to add to the `kubernetes_deployment` resource—a `selector` block:

```
    spec {
      replicas = var.replicas
```

```
    template {
      metadata {
        labels = local.pod_labels
      }

      spec {
        container {
          name  = var.name
          image = var.image

          port {
            container_port = var.container_port
          }

          dynamic "env" {
            for_each = var.environment_variables
            content {
              name  = env.key
              value = env.value
            }
          }
        }
      }
    }

    selector {
      match_labels = local.pod_labels
    }
  }
```

The `selector` block tells the Kubernetes Deployment what to target. By setting it to `pod_labels`, you are telling it to manage deployments for the Pod Template you just defined. Why doesn't the Deployment just assume that the Pod Template defined within that Deployment is the one you want to target? Well, Kubernetes tries to be an extremely flexible and decoupled system: e.g., it's possible to define a Deployment for Pods that are defined separately, so you always need to specify a `selector` to tell the Deployment what to target.

That wraps up the `kubernetes_deployment` resource. The next step is to use the `kubernetes_service` resource to create a Kubernetes Service:

```
resource "kubernetes_service" "app" {
  metadata {
    name = var.name
  }

  spec {
    type = "LoadBalancer"
    port {
      port        = 80
```

```
      target_port = var.container_port
      protocol    = "TCP"
    }
    selector = local.pod_labels
  }
}
```

Let's go through these parameters:

`metadata`
: Just as with the Deployment object, the Service object uses metadata to identify and target that object in API calls. In the preceding code, I've set the Service name to the `name` input variable.

`type`
: I've configured this Service as type `LoadBalancer`, which, depending on how your Kubernetes cluster is configured, will deploy a different type of load balancer: e.g., in AWS, with EKS, you might get an Elastic Load Balancer, whereas in Google Cloud, with GKE, you might get a Cloud Load Balancer.

`port`
: I'm configuring the load balancer to route traffic on port 80 (the default port for HTTP) to the port the container is listening on.

`selector`
: Just as with the Deployment object, the Service object uses a selector to specify what that Service should be targeting. By setting the selector to `pod_labels`, the Service and the Deployment will both operate on the same Pods.

The final step is to expose the Service endpoint (the load balancer hostname) as an output variable in *outputs.tf*:

```
locals {
  status = kubernetes_service.app.status
}

output "service_endpoint" {
  value = try(
    "http://${local.status[0]["load_balancer"][0]["ingress"][0]["hostname"]}",
    "(error parsing hostname from status)"
  )
  description = "The K8S Service endpoint"
}
```

This convoluted code needs a bit of explanation. The `kubernetes_service` resource has an output attribute called `status` that returns the latest status of the Service. I've stored this attribute in a local variable called `status`. For a Service of type LoadBalancer, `status` will contain a complicated object that looks something like this:

```
[
  {
    load_balancer = [
      {
        ingress = [
          {
            hostname = "<HOSTNAME>"
          }
        ]
      }
    ]
  }
]
```

Buried within this deeply nested object is the hostname for the load balancer that you want. This is why the `service_endpoint` output variable needs to use a complicated sequence of array lookups (e.g., `[0]`) and map lookups (e.g., `["load_balancer"]`) to extract the hostname. But what happens if the `status` attribute returned by the `kubernetes_service` resource happens to look a little different? In that case, any of those array and map lookups could fail, leading to a confusing error.

To handle this error gracefully, I've wrapped the entire expression in a function called `try`. The `try` function has the following syntax:

```
try(ARG1, ARG2, ..., ARGN)
```

This function evaluates all the arguments you pass to it and returns the first argument that doesn't produce any errors. Therefore, the `service_endpoint` output variable will either end up with a hostname in it (the first argument) or, if reading the hostname caused an error, the variable will instead say "error parsing hostname from status" (the second argument).

OK, that wraps up the `k8s-app` module. To use it, add a new example in *examples/kubernetes-local*, and create a *main.tf* file in it with the following contents:

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000
}
```

This configures the module to deploy the `training/webapp` Docker image you ran earlier, with two replicas listening on port 5000, and to name all the Kubernetes objects (based on their `metadata`) "simple-webapp". To have this module deploy into your local Kubernetes cluster, add the following `provider` block:

```
provider "kubernetes" {
  config_path    = "~/.kube/config"
```

```
  config_context = "docker-desktop"
}
```

This code tells the Kubernetes provider to authenticate to your local Kubernetes cluster by using the `docker-desktop` context from your `kubectl` config. Run `terraform apply` to see how it works:

```
$ terraform apply

(...)

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

service_endpoint = "http://localhost"
```

Give the app a few seconds to boot and then try out that `service_endpoint`:

```
$ curl http://localhost
Hello world!
```

Success!

That said, this looks nearly identical to the output of the `docker run` command, so was all that extra work worth it? Well, let's look under the hood to see what's going on. You can use `kubectl` to explore your cluster. First, run the `get deployments` command:

```
$ kubectl get deployments
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
simple-webapp   2/2     2            2           3m21s
```

You can see your Kubernetes Deployment, named `simple-webapp`, as that was the name in the `metadata` block. This Deployment is reporting that 2/2 Pods (the two replicas) are ready. To see those Pods, run the `get pods` command:

```
$ kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
simple-webapp-d45b496fd-7d447   1/1     Running   0          2m36s
simple-webapp-d45b496fd-vl6j7   1/1     Running   0          2m36s
```

So that's one difference from `docker run` already: there are multiple containers running here, not just one. Moreover, those containers are being actively monitored and managed. For example, if one crashed, a replacement will be deployed automatically. You can see this in action by running the `docker ps` command:

```
$ docker ps
CONTAINER ID   IMAGE             COMMAND           CREATED          STATUS
b60f5147954a   training/webapp   "python app.py"   3 seconds ago    Up 2 seconds
c350ec648185   training/webapp   "python app.py"   12 minutes ago   Up 12 minutes
```

Grab the `CONTAINER ID` of one of those containers, and use the `docker kill` command to shut it down:

```
$ docker kill b60f5147954a
```

If you run `docker ps` again very quickly, you'll see just one container left running:

```
$ docker ps
CONTAINER ID   IMAGE             COMMAND           CREATED          STATUS
c350ec648185   training/webapp   "python app.py"   12 minutes ago   Up 12 minutes
```

But just a few seconds later, the Kubernetes Deployment will have detected that there is only one replica instead of the requested two, and it'll launch a replacement container automatically:

```
$ docker ps
CONTAINER ID   IMAGE             COMMAND           CREATED          STATUS
56a216b8a829   training/webapp   "python app.py"   1 second ago     Up 5 seconds
c350ec648185   training/webapp   "python app.py"   12 minutes ago   Up 12 minutes
```

So Kubernetes is ensuring that you always have the expected number of replicas running. Moreover, it is also running a load balancer to distribute traffic across those replicas, which you can see by running the `kubectl get services` command:

```
$ kubectl get services
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP      10.96.0.1      <none>        443/TCP        4h26m
simple-webapp   LoadBalancer   10.110.25.79   localhost     80:30234/TCP   4m58s
```

The first service in the list is Kubernetes itself, which you can ignore. The second is the Service you created, also with the name `simple-webapp` (based on the `metadata` block). This service runs a load balancer for your app: you can see the IP it's accessible at (`localhost`) and the port it's listening on (80).

Kubernetes Deployments also provide automatic rollout of updates. A fun trick with the `training/webapp` Docker image is that if you set the environment variable `PROVIDER` to some value, it'll use that value instead of the word *world* in the text "Hello, world!" Update *examples/kubernetes-local/main.tf* to set this environment variable as follows:

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Terraform"
  }
}
```

Run `apply` one more time:

```
$ terraform apply

(...)

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

Outputs:

service_endpoint = "http://localhost"
```

After a few seconds, try the endpoint again:

```
$ curl http://localhost
Hello Terraform!
```

And there you go, the Deployment has rolled out your change automatically: under the hood, Deployments do a rolling deployment by default, similar to what you saw with Auto Scaling Groups (note that you can change deployment settings by adding a `strategy` block to the `kubernetes_deployment` resource).

Deploying Docker Containers in AWS Using Elastic Kubernetes Service

Kubernetes has one more trick up its sleeve: it's fairly portable. That is, you can reuse both the Docker images and the Kubernetes configurations in a totally different cluster and get similar results. To see this in action, let's now deploy a Kubernetes cluster in AWS.

Setting up and managing a secure, highly available, scalable Kubernetes cluster in the cloud from scratch is complicated. Fortunately, most cloud providers offer managed Kubernetes services, where they run the control plane and worker nodes for you: e.g., Elastic Kubernetes Service (EKS) in AWS, Azure Kubernetes Service (AKS) in Azure, and Google Kubernetes Engine (GKE) in Google Cloud. I'm going to show you how to deploy a very basic EKS cluster in AWS.

Create a new module in *modules/services/eks-cluster*, and define the API for the module in a *variables.tf* file with the following input variables:

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

variable "min_size" {
  description = "Minimum number of nodes to have in the EKS cluster"
  type        = number
}

variable "max_size" {
  description = "Maximum number of nodes to have in the EKS cluster"
```

```
  type        = number
}

variable "desired_size" {
  description = "Desired number of nodes to have in the EKS cluster"
  type        = number
}

variable "instance_types" {
  description = "The types of EC2 instances to run in the node group"
  type        = list(string)
}
```

This code exposes input variables to set the EKS cluster's name, size, and the types of instances to use for the worker nodes. Next, in *main.tf*, create an IAM role for the control plane:

```
# Create an IAM role for the control plane
resource "aws_iam_role" "cluster" {
  name               = "${var.name}-cluster-role"
  assume_role_policy = data.aws_iam_policy_document.cluster_assume_role.json
}

# Allow EKS to assume the IAM role
data "aws_iam_policy_document" "cluster_assume_role" {
  statement {
    effect  = "Allow"
    actions = ["sts:AssumeRole"]
    principals {
      type        = "Service"
      identifiers = ["eks.amazonaws.com"]
    }
  }
}

# Attach the permissions the IAM role needs
resource "aws_iam_role_policy_attachment" "AmazonEKSClusterPolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
  role       = aws_iam_role.cluster.name
}
```

This IAM role can be assumed by the EKS service, and it has a Managed IAM Policy attached that gives the control plane the permissions it needs. Now, add the `aws_vpc` and `aws_subnets` data sources to fetch information about the Default VPC and its subnets:

```
# Since this code is only for learning, use the Default VPC and subnets.
# For real-world use cases, you should use a custom VPC and private subnets.

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

```
data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}
```

Now you can create the control plane for the EKS cluster by using the `aws_eks_cluster` resource:

```
resource "aws_eks_cluster" "cluster" {
  name     = var.name
  role_arn = aws_iam_role.cluster.arn
  version  = "1.21"

  vpc_config {
    subnet_ids = data.aws_subnets.default.ids
  }

  # Ensure that IAM Role permissions are created before and deleted after
  # the EKS Cluster. Otherwise, EKS will not be able to properly delete
  # EKS managed EC2 infrastructure such as Security Groups.
  depends_on = [
    aws_iam_role_policy_attachment.AmazonEKSClusterPolicy
  ]
}
```

The preceding code configures the control plane to use the IAM role you just created, and to deploy into the Default VPC and subnets.

Next up are the worker nodes. EKS supports several different types of worker nodes: self-managed EC2 Instances (e.g., in an ASG that you create), AWS-managed EC2 Instances (known as a *managed node group*), and Fargate (serverless).[3] The simplest option to use for the examples in this chapter will be the managed node groups.

To deploy a managed node group, you first need to create another IAM role:

```
# Create an IAM role for the node group
resource "aws_iam_role" "node_group" {
  name               = "${var.name}-node-group"
  assume_role_policy = data.aws_iam_policy_document.node_assume_role.json
}

# Allow EC2 instances to assume the IAM role
data "aws_iam_policy_document" "node_assume_role" {
  statement {
    effect  = "Allow"
    actions = ["sts:AssumeRole"]
```

3 For a comparison of the different types of EKS worker nodes, see the Gruntwork blog.

```
    principals {
      type        = "Service"
      identifiers = ["ec2.amazonaws.com"]
    }
  }
}

# Attach the permissions the node group needs
resource "aws_iam_role_policy_attachment" "AmazonEKSWorkerNodePolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"
  role       = aws_iam_role.node_group.name
}

resource "aws_iam_role_policy_attachment" "AmazonEC2ContainerRegistryReadOnly" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly"
  role       = aws_iam_role.node_group.name
}

resource "aws_iam_role_policy_attachment" "AmazonEKS_CNI_Policy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy"
  role       = aws_iam_role.node_group.name
}
```

This IAM role can be assumed by the EC2 service (which makes sense, as managed node groups use EC2 Instances under the hood), and it has several Managed IAM Policies attached that give the managed node group the permissions it needs. Now you can use the `aws_eks_node_group` resource to create the managed node group itself:

```
resource "aws_eks_node_group" "nodes" {
  cluster_name    = aws_eks_cluster.cluster.name
  node_group_name = var.name
  node_role_arn   = aws_iam_role.node_group.arn
  subnet_ids      = data.aws_subnets.default.ids
  instance_types  = var.instance_types

  scaling_config {
    min_size     = var.min_size
    max_size     = var.max_size
    desired_size = var.desired_size
  }

  # Ensure that IAM Role permissions are created before and deleted after
  # the EKS Node Group. Otherwise, EKS will not be able to properly
  # delete EC2 Instances and Elastic Network Interfaces.
  depends_on = [
    aws_iam_role_policy_attachment.AmazonEKSWorkerNodePolicy,
    aws_iam_role_policy_attachment.AmazonEC2ContainerRegistryReadOnly,
    aws_iam_role_policy_attachment.AmazonEKS_CNI_Policy,
  ]
}
```

This code configures the managed node group to use the control plane and IAM role you just created, to deploy into the Default VPC, and to use the name, size, and instance type parameters passed in as input variables.

In *outputs.tf*, add the following output variables:

```
output "cluster_name" {
  value       = aws_eks_cluster.cluster.name
  description = "Name of the EKS cluster"
}

output "cluster_arn" {
  value       = aws_eks_cluster.cluster.arn
  description = "ARN of the EKS cluster"
}

output "cluster_endpoint" {
  value       = aws_eks_cluster.cluster.endpoint
  description = "Endpoint of the EKS cluster"
}

output "cluster_certificate_authority" {
  value       = aws_eks_cluster.cluster.certificate_authority
  description = "Certificate authority of the EKS cluster"
}
```

OK, the `eks-cluster` module is now ready to roll. Let's use it and the `k8s-app` module from earlier to deploy an EKS cluster and to deploy the `training/webapp` Docker image into that cluster. Create *examples/kubernetes-eks/main.tf*, and configure the `eks-cluster` module as follows:

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

module "eks_cluster" {
  source = "../../modules/services/eks-cluster"

  name         = "example-eks-cluster"
  min_size     = 1
  max_size     = 2
  desired_size = 1

  # Due to the way EKS works with ENIs, t3.small is the smallest
  # instance type that can be used for worker nodes. If you try
  # something smaller like t2.micro, which only has 4 ENIs,
  # they'll all be used up by system services (e.g., kube-proxy)
  # and you won't be able to deploy your own Pods.
  instance_types = ["t3.small"]
}
```

Next, configure the `k8s-app` module as follows:

```
provider "kubernetes" {
  host = module.eks_cluster.cluster_endpoint
  cluster_ca_certificate = base64decode(
    module.eks_cluster.cluster_certificate_authority[0].data
  )
  token = data.aws_eks_cluster_auth.cluster.token
}

data "aws_eks_cluster_auth" "cluster" {
  name = module.eks_cluster.cluster_name
}

module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Terraform"
  }

  # Only deploy the app after the cluster has been deployed
  depends_on = [module.eks_cluster]
}
```

The preceding code configures the Kubernetes provider to authenticate to the EKS cluster, rather than your local Kubernetes cluster (from Docker Desktop). It then uses the `k8s-app` module to deploy the `training/webapp` Docker image exactly the same way as you did when deploying it to Docker Desktop; the only difference is the addition of the `depends_on` parameter to ensure that Terraform only tries to deploy the Docker image after the EKS cluster has been deployed.

Next, pass through the service endpoint as an output variable:

```
output "service_endpoint" {
  value       = module.simple_webapp.service_endpoint
  description = "The K8S Service endpoint"
}
```

OK, now you're ready to deploy! Run `terraform apply` as usual (note that EKS clusters can take 10–20 minutes to deploy, so be patient):

```
$ terraform apply

(...)

Apply complete! Resources: 10 added, 0 changed, 0 destroyed.

Outputs:
```

```
service_endpoint = "http://774696355.us-east-2.elb.amazonaws.com"
```

Wait a little while for the web app to spin up and pass health checks, and then test out the `service_endpoint`:

```
$ curl http://774696355.us-east-2.elb.amazonaws.com
Hello Terraform!
```

And there you have it! The same Docker image and Kubernetes code is now running in an EKS cluster in AWS, just the way it ran on your local computer. All the same features work here too. For example, try updating `environment_variables` to a different `PROVIDER` value, such as "Readers":

```
module "simple_webapp" {
  source = "../../modules/services/k8s-app"

  name           = "simple-webapp"
  image          = "training/webapp"
  replicas       = 2
  container_port = 5000

  environment_variables = {
    PROVIDER = "Readers"
  }

  # Only deploy the app after the cluster has been deployed
  depends_on = [module.eks_cluster]
}
```

Rerun `apply`, and just a few seconds later, the Kubernetes Deployment will have deployed the changes:

```
$ curl http://774696355.us-east-2.elb.amazonaws.com
Hello Readers!
```

This is one of the advantages of using Docker: changes can be deployed very quickly.

You can use `kubectl` again to see what's happening in your cluster. To authenticate `kubectl` to the EKS cluster, you can use the `aws eks update-kubeconfig` command to automatically update your *$HOME/.kube/config* file:

```
$ aws eks update-kubeconfig --region <REGION> --name <EKS_CLUSTER_NAME>
```

where `REGION` is the AWS region and `EKS_CLUSTER_NAME` is the name of your EKS cluster. In the Terraform module, you deployed to the `us-east-2` region and named the cluster `kubernetes-example`, so the command will look like this:

```
$ aws eks update-kubeconfig --region us-east-2 --name kubernetes-example
```

Now, just as before, you can use the `get nodes` command to inspect the worker nodes in your cluster, but this time, add the `-o wide` flag to get a bit more info:

```
$ kubectl get nodes
NAME                              STATUS   AGE   EXTERNAL-IP    OS-IMAGE
xxx.us-east-2.compute.internal    Ready    22m   3.134.78.187   Amazon Linux 2
```

The preceding snippet is highly truncated to fit into the book, but in the real output, you should be able to see the one worker node, its internal and external IP, version information, OS information, and much more.

You can use the `get deployments` command to inspect your Deployments:

```
$ kubectl get deployments
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
simple-webapp   2/2     2            2           19m
```

Next, run `get pods` to see the Pods:

```
$ kubectl get pods
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
simple-webapp   2/2     2            2           19m
```

And finally, run `get services` to see the Services:

```
$ kubectl get services
NAME            TYPE           EXTERNAL-IP                                PORT(S)
kubernetes      ClusterIP      <none>                                     443/TCP
simple-webapp   LoadBalancer   774696355.us-east-2.elb.amazonaws.com      80/TCP
```

You should be able to see your load balancer and the URL you used to test it.

So there you have it: two different providers, both working in the same cloud, helping you to deploy containerized workloads.

That said, just as in previous sections, I want to leave you with a few warnings:

Warning 1: These Kubernetes examples are very simplified!

Kubernetes is complicated, and it's rapidly evolving and changing; trying to explain all the details can easily fill a book all by itself. Since this is a book about Terraform, and not Kubernetes, my goal with the Kubernetes examples in this chapter was to keep them as simple and minimal as possible. Therefore, while I hope the code examples you've seen have been useful from a learning and experimentation perspective, if you are going to use Kubernetes for real-world, production use cases, you'll need to change many aspects of this code, such as configuring a number of additional services and settings in the `eks-cluster` module (e.g., ingress controllers, secret envelope encryption, security groups, OIDC authentication, Role-Based Access Control (RBAC) mapping, VPC CNI, kube-proxy, CoreDNS), exposing many other settings in the `k8s-app` module (e.g., secrets management, volumes, liveness probes, readiness probes, labels, annotations, multiple ports, multiple containers), and using a custom VPC with

private subnets for your EKS cluster instead of the Default VPC and public subnets.[4]

Warning 2: Use multiple providers sparingly

Although you certainly can use multiple providers in a single module, I don't recommend doing it too often, for similar reasons to why I don't recommend using provider aliases too often: in most cases, you want each provider to be isolated in its own module so that you can manage it separately and limit the blast radius from mistakes or attackers.

Moreover, Terraform doesn't have great support for dependency ordering between providers. For example, in the Kubernetes example, you had a single module that deployed both the EKS cluster, using the AWS Provider, and a Kubernetes app into that cluster, using the Kubernetes provider. As it turns out, the Kubernetes provider documentation explicitly recommends *against* this pattern:

> When using interpolation to pass credentials to the Kubernetes provider from other resources, these resources SHOULD NOT be created in the same Terraform module where Kubernetes provider resources are also used. This will lead to intermittent and unpredictable errors which are hard to debug and diagnose. The root issue lies with the order in which Terraform itself evaluates the provider blocks vs. actual resources.

The example code in this book is able to work around these issues by depending on the `aws_eks_cluster_auth` data source, but that's a bit of a hack. Therefore, in production code, I always recommend deploying the EKS cluster in one module and deploying Kubernetes apps in separate modules, after the cluster has been deployed.

Conclusion

At this point, you hopefully understand how to work with multiple providers in Terraform code, and you can answer the three questions from the beginning of this chapter:

What if you need to deploy to multiple AWS regions?

Use multiple `provider` blocks, each configured with a different `region` and `alias` parameter.

4 Alternatively, you can use off-the-shelf production-grade Kubernetes modules, such as the ones in the Gruntwork Infrastructure as Code Library.

What if you need to deploy to multiple AWS accounts?
: Use multiple `provider` blocks, each configured with a different `assume_role` block and an `alias` parameter.

What if you need to deploy to other clouds, such as Azure or GCP or Kubernetes?
: Use multiple `provider` blocks, each configured for its respective cloud.

However, you've also seen that using multiple providers in one module is typically an antipattern. So the real answer to these questions, especially in real-world, production use cases, is to use each provider in a separate module to keep different regions, accounts, and clouds isolated from one another, and to limit your blast radius.

Let's now move on to Chapter 8, where I'll go over several other patterns for how to build Terraform modules for real-world, production use cases—the kind of modules you could bet your company on.

CHAPTER 8
Production-Grade Terraform Code

Building production-grade infrastructure is difficult. And stressful. And time consuming. By *production-grade infrastructure*, I mean the kind of infrastructure you'd bet your company on. You're 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 might go out of business. That's what's 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 one to two 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 two to four 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 two to four 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 to 36 months of work, with

small companies typically being closer to six months and larger companies typically taking several years.

Table 8-1 shows a summary of this data.

Table 8-1. 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 in an ASG	2–4 weeks
Self-managed distributed system (stateful)	Elasticsearch cluster	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?" or "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 overconfident 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, it might 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:

- Why it takes so long to build production-grade infrastructure
- The production-grade infrastructure checklist
- Production-grade infrastructure modules
  - Small modules
  - Composable modules
  - Testable modules
  - Versioned modules
  - Beyond Terraform modules

Example Code

As a reminder, you can find all of the code examples in the book on GitHub.

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 that you thought would take five minutes takes up the entire day; the minor new feature that you estimated at a day of work takes two weeks; the app that 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.

I think there are three major reasons for this. The first reason is that DevOps, as an industry, is still in the Stone Age. 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-2000s, and tools like Terraform, Docker, Packer, and Kubernetes were all initially released in the mid- to late-2010s. All of these tools and techniques are relatively new, and all of them are changing rapidly. This means that 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…"

1 Douglas R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*, 20th anniversary ed. (New York: Basic Books, 1999).

2 Seth Godin, "Don't Shave That Yak!" Seth's Blog, March 5, 2005, *https://bit.ly/2OK45uL*.

> "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.

Yak shaving consists of all the tiny, seemingly unrelated tasks you must 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 uncover a bug in your app configuration. You try to deploy a fix for the app configuration, but that's blocked by a TLS certificate issue. After spending hours on Stack Overflow, you try to roll out a fix for the TLS issue, but that fails due to a problem 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-shaving 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 one wire from a box of tangled wires—it just tends to 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 why DevOps work takes so long. The first two reasons—DevOps is in the Stone Age and yak shaving—can be classified as accidental complexity. *Accidental complexity* refers to the problems imposed by the particular tools and processes you've chosen, as opposed to *essential complexity*, which refers to the problems inherent in whatever it is that you're working on.[3] For example, if you're using C++ to write stock-trading algorithms, 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. On the other hand, 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.

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

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; and 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 can 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 8-2. This list covers most of the key items that you need to consider to deploy infrastructure to production.

Table 8-2. The Production-Grade Infrastructure Checklist

Task	Description	Example tools
Install	Install the software binaries and all dependencies.	Bash, Ansible, Docker, Packer
Configure	Configure the software at runtime. Includes port settings, TLS certs, service discovery, leaders, followers, replication, etc.	Chef, Ansible, Kubernetes
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.	ASG, Kubernetes, ECS
High availability	Withstand outages of individual processes, servers, services, datacenters, and regions.	Multi-datacenter, multi-region
Scalability	Scale up and down in response to load. Scale horizontally (more servers) and/or vertically (bigger servers).	Auto scaling, replication
Performance	Optimize CPU, memory, disk, network, and GPU usage. Includes query tuning, benchmarking, load testing, and profiling.	Dynatrace, Valgrind, VisualVM
Networking	Configure static and dynamic IPs, ports, service discovery, firewalls, DNS, SSH access, and VPN access.	VPCs, firewalls, Route 53

Task	Description	Example tools
Security	Encryption in transit (TLS) and on disk, authentication, authorization, secrets management, server hardening.	ACM, Let's Encrypt, KMS, Vault
Metrics	Availability metrics, business metrics, app metrics, server metrics, events, observability, tracing, and alerting.	CloudWatch, Datadog
Logs	Rotate logs on disk. Aggregate log data to a central location.	Elastic Stack, Sumo Logic
Data backup	Make backups of DBs, caches, and other data on a scheduled basis. Replicate to separate region/account.	AWS Backup, RDS snapshots
Cost optimization	Pick proper Instance types, use spot and reserved Instances, use auto scaling, and clean up unused resources.	Auto scaling, Infracost
Documentation	Document your code, architecture, and practices. Create playbooks to respond to incidents.	READMEs, wikis, Slack, IaC
Tests	Write automated tests for your infrastructure code. Run tests after every commit and nightly.	Terratest, tflint, OPA, InSpec

Most developers are aware of the first few tasks: install, configure, provision, and deploy. It's all the ones that come after them 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 datacenter 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 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 that you need 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 I'll cover:

- Small modules
- Composable modules
- Testable modules
- Versioned modules
- Beyond Terraform modules

Small Modules

Developers who are new to Terraform, and IaC 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 "State File Isolation" on page 93, 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 of your infrastructure is defined in one Terraform module, running any command will take a long time. I've seen modules grow so large that `terraform plan` takes 20 minutes to run!

Large modules are insecure
: If all your infrastructure is managed in a single large module, to change anything, you need permissions to access everything. This means that almost every user must be an admin, which goes against the *principle of least privilege*.

Large modules are risky
: If all your eggs are in one basket, 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 database.

Large modules are difficult to understand
: The more code you have in one place, the more difficult 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 difficult to review
: Reviewing a module that consists of several dozen lines of code 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 difficult to test
: Testing infrastructure code is hard; testing a large amount of infrastructure code is nearly impossible. I'll come back to this point in Chapter 9.

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.

Imagine you were using a general-purpose programming language such as Java or Python or Ruby, and you came across a single function that was *20,000 lines* long—you would immediately know this is a code smell. The better approach is to refactor this code into a number of small, standalone functions that each do one thing. You should use the same strategy with Terraform.

Imagine that you came across the architecture shown in Figure 8-1.

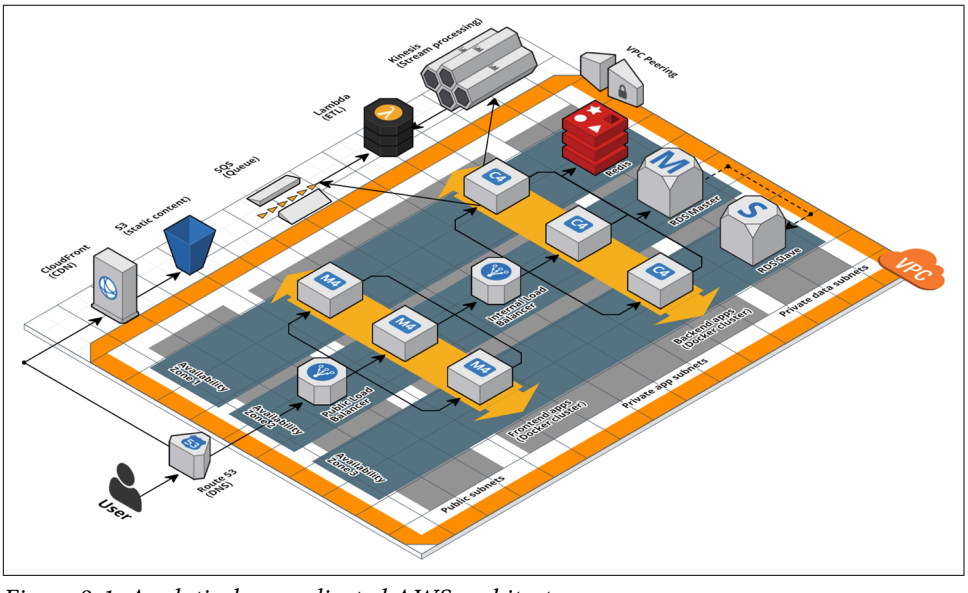
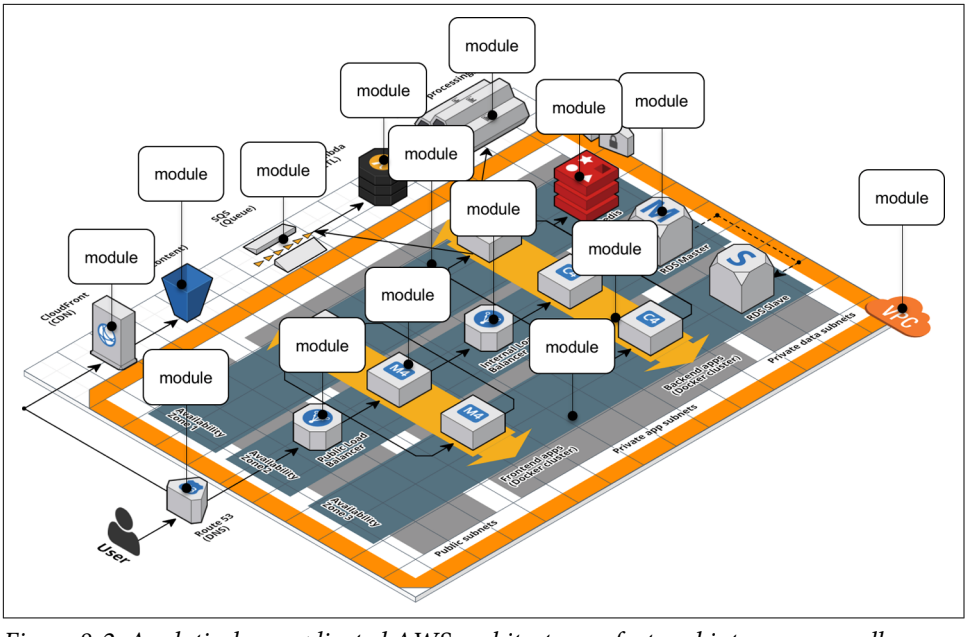


Figure 8-1. A relatively complicated AWS architecture.

If this architecture was defined in a single Terraform module that was 20,000 lines long, you should immediately think of it as a code smell. The better approach is to refactor this module into a number of small, standalone modules that each do one thing, as shown in Figure 8-2.

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


module
module
module

`modules/services/hello-world-app`
: A module specifically for deploying the "Hello, World" app, which uses the `asg-rolling-deploy` and `alb` modules under the hood.

Before getting started, make sure to run `terraform destroy` on any `webserver-cluster` deployments you have from previous chapters. After you do that, you can start putting together the `asg-rolling-deploy` and `alb` modules. Create a new folder at *modules/cluster/asg-rolling-deploy*, and move 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` (just the one rule for the Instances but not those for the ALB)
- `aws_cloudwatch_metric_alarm` (both of them)

Next, move the following variables from *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 the one 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_subnets.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 GitHub.

Composable Modules

You now have two small modules—`asg-rolling-deploy` and `alb`—that each 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 is 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.

One way to do this is through *function composition*, in which 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
```

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

```
# 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, since 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, return 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 ELB 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     = null
}
```

The first variable, `subnet_ids`, tells the `asg-rolling-deploy` module what subnets to deploy into. Whereas the `webserver-cluster` module was hardcoded 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, and so on.

Take these three new input variables and pass them through to the `aws_autoscaling_group` resource in *modules/cluster/asg-rolling-deploy/main.tf*, replacing the previously hardcoded settings that were referencing resources (e.g., the ALB) and data sources (e.g., `aws_subnets`) that we didn't copy into the `asg-rolling-deploy` module:

```
resource "aws_autoscaling_group" "example" {
  name                 = var.cluster_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

  # (...)
}
```

The fourth variable, `user_data`, is for passing in a User Data script. Whereas the `webserver-cluster` module had a hardcoded User Data script that could only be used to deploy a "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:

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

You'll also want to add a couple of 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, since 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 to use these 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 have only the following resources and data sources left in *module/services/hello-world-app/main.tf*:

- `aws_lb_target_group`
- `aws_lb_listener_rule`
- `terraform_remote_state` (for the DB)
- `aws_vpc`
- `aws_subnets`

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 the `asg-rolling-deploy` module that 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
  instance_type = var.instance_type

  user_data     = templatefile("${path.module}/user-data.sh", {
    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
  })

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

  subnet_ids        = data.aws_subnets.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, also that 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_subnets.default.ids
}
```

Note the use of 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 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
  }
}
```

Now, 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 {
    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).

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           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  min_size           = 1
  max_size           = 1
  enable_autoscaling = false

  subnet_ids        = data.aws_subnets.default.ids
}

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

```
data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}

data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}
```

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 the following:

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.

An automated test harness
: As you will see in Chapter 9, this example code is also how you create automated tests for your modules. I typically recommend that tests go into the *test* folder.

Executable documentation
: If you commit this example (including *README.md*) into version control, 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 might want to add other examples for the `asg-rolling-deploy` module that show how to use it with auto scaling policies, how to hook up load balancers to it, 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 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 begin with the implementation, it's too easy to become lost 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 difficult to use. On the other hand, if you begin 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 backward to the implementation. Because the example code is the primary way of testing modules anyway, this is a form of *Test-Driven Development* (TDD); I'll dive more into this topic in Chapter 9, which is entirely dedicated to testing.

In this section, I'll focus on creating *self-validating modules*: that is, modules that can check their own behavior to prevent certain types of bugs. Terraform has two ways of doing this built in:

- Validations
- Preconditions and postconditions

Validations

As of Terraform 0.13, you can add *validation blocks* to any input variable to perform checks that go beyond basic type constraints. For example, you can add a `validation` block to the `instance_type` variable to ensure not only that the value the user passes in is a string (which is enforced by the `type` constraint) but that the string has one of two allowed values from the AWS Free Tier:

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

  validation {
    condition     = contains(["t2.micro", "t3.micro"], var.instance_type)
    error_message = "Only free tier is allowed: t2.micro | t3.micro."
  }
}
```

The way a `validation` block works is that the `condition` parameter should evaluate to `true` if the value is valid and `false` otherwise. The `error_message` parameter allows you to specify the message to show the user if they pass in an invalid value. For example, here's what happens if you try to set `instance_type` to `m4.large`, which is not in the AWS Free Tier:

```
$ terraform apply -var instance_type="m4.large"
| Error: Invalid value for variable
|
|   on main.tf line 17:
|    1: variable "instance_type" {
|     ├────────────────
|     | var.instance_type is "m4.large"
|
| Only free tier is allowed: t2.micro | t3.micro.
|
| This was checked by the validation rule at main.tf:21,3-13.
```

You can have multiple `validation` blocks in each variable to check multiple conditions:

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

  validation {
    condition     = var.min_size > 0
    error_message = "ASGs can't be empty or we'll have an outage!"
  }

  validation {
    condition     = var.min_size <= 10
    error_message = "ASGs must have 10 or fewer instances to keep costs down."
```

```
  }
}
```

Note that `validation` blocks have a major limitation: the `condition` in a `validation` block can *only* reference the surrounding input variable. If you try to reference any other input variables, local variables, resources, or data sources, you will get an error. So while `validation` blocks are useful for basic input sanitization, they can't be used for anything more complicated: for example, you can't use them to do checks across multiple variables (such as "exactly one of these two input variables must be set") or any kind of dynamic checks (such as checking that the AMI the user requested uses the x86_64 architecture). To do these sorts of more dynamic checks, you'll need to use `precondition` and `postcondition` blocks, as described next.

Preconditions and postconditions

As of Terraform 1.2, you can add `precondition` and `postcondition` blocks to resources, data sources, and output variables to perform more dynamic checks. The `precondition` blocks are for catching errors before you run `apply`. For example, you could use a `precondition` block to do a more robust check that the `instance_type` the user passes in is in the AWS Free Tier. In the previous section, you did this check using a `validation` block and a hardcoded list of instance types, but these sorts of lists quickly go out of date. You can instead use the `instance_type_data` data source to always get up-to-date information from AWS:

```
data "aws_ec2_instance_type" "instance" {
  instance_type = var.instance_type
}
```

And then you can add a `precondition` block to the `aws_launch_configuration` resource to check that this instance type is eligible for the AWS Free Tier:

```
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.
  lifecycle {
    create_before_destroy = true
    precondition {
      condition     = data.aws_ec2_instance_type.instance.free_tier_eligible
      error_message = "${var.instance_type} is not part of the AWS Free Tier!"
    }
  }
}
```

Just like `validation` blocks, `precondition` blocks (and `postcondition` blocks, as you'll see shortly) include a `condition` that must evaluate to `true` or `false` and an `error_message` to show the user if the `condition` evaluates to `false`. If you now try to run `apply` with an instance type not in the AWS Free Tier, you'll see your error message:

```
$ terraform apply -var instance_type="m4.large"
| Error: Resource precondition failed
|
|   on main.tf line 25, in resource "aws_launch_configuration" "example":
|   18:    condition = data.aws_ec2_instance_type.instance.free_tier_eligible
|     ├────────────────
|     | data.aws_ec2_instance_type.instance.free_tier_eligible is false
|
| m4.large is not part of the AWS Free Tier!
```

The `postcondition` blocks are for catching errors after you run `apply`. For example, you can add a `postcondition` block to the `aws_autoscaling_group` resource to check that the ASG was deployed across more than one Availability Zone (AZ), thereby ensuring you can tolerate the failure of at least one AZ:

```
resource "aws_autoscaling_group" "example" {
  name                 = var.cluster_name
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = var.subnet_ids

  lifecycle {
    postcondition {
      condition     = length(self.availability_zones) > 1
      error_message = "You must use more than one AZ for high availability!"
    }
  }

  # (...)
}
```

Note the use of the `self` keyword in the `condition` parameter. *Self expressions* use the following syntax:

```
self.<ATTRIBUTE>
```

You can use this special syntax solely in `postcondition`, `connection`, and `provisioner` blocks (you'll see examples of the latter two later in this chapter) to refer to an output `ATTRIBUTE` of the surrounding resource. If you tried to use the standard `aws_autoscaling_group.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 this sort of use case.

If you run `apply` on this module, Terraform will deploy the module, but after, if it turns out that the subnets the user passed in via the `subnet_ids` input variable were

all in the same AZ, the `postcondition` block will show an error. This way, you'll always be warned if your ASG isn't configured for high availability.

When to use validations, preconditions, and postconditions

As you can see, `validation`, `precondition`, and `postcondition` blocks are all similar, so when should you use each one?

Use `validation` *blocks for basic input sanitization*
: Use `validation` blocks in all of your production-grade modules to prevent users from passing invalid variables into your modules. The goal is to catch basic input errors *before* any changes have been deployed. Although `precondition` blocks are more powerful, you should still use `validation` blocks for checking variables whenever possible, as `validation` blocks are defined with the variables they validate, which leads to a more readable and maintainable API.

Use `precondition` *blocks for checking basic assumptions*
: Use `precondition` blocks in all of your production-grade modules to check assumptions that must be true *before* any changes have been deployed. This includes any checks on variables you can't do with `validation` blocks (such as checks that reference multiple variables or data sources) as well as checks on resources and data sources. The goal is to catch as many errors as early as you can, before those errors can do any damage.

Use `postcondition` *blocks for enforcing basic guarantees*
: Use `postcondition` blocks in all of your production-grade modules to check guarantees about how your module behaves *after* changes have been deployed. The goal is to give users of your module confidence that your module will either do what it says when they run `apply` or exit with an error. It also gives maintainers of that module a clearer signal of what behaviors you want this module to enforce, so those aren't accidentally lost during a refactor.

Use automated testing tools for enforcing more advanced assumptions and guarantees
: `validation`, `precondition`, and `postcondition` blocks are all useful tools, but they can only do basic checks. This is because you can only use data sources, resources, and language constructs built into Terraform to do these checks, and those are often not enough for more advanced behavior. For example, if you built a module to deploy a web service, you might want to add a check after deployment that the web service is able to respond to HTTP requests. You could try to do this in a `postcondition` block by making HTTP requests to the service using Terraform's http provider, but most deployments happen asynchronously, so you may need to retry the HTTP request multiple times, and there is no retry mechanism built into that provider. Moreover, if you deployed an internal web service, it might not be accessible over the public internet, so you'd need

to connect to some internal network or VPN first, which is also tricky to do in pure Terraform code. Therefore, to do more robust checks, you'll want to use automated testing tools such as OPA and Terratest, both of which you'll see in Chapter 9.

Versioned Modules

There are two types of versioning you'll want to think through with modules:

- Versioning of the module's dependencies
- Versioning of the module itself

Let's start with versioning of the module's dependencies. Your Terraform code has three types of dependencies:

Terraform core
: The version of the `terraform` binary you depend on

Providers
: The version of each provider your code depends on, such as the `aws` provider

Modules
: The version of each module you depend on that are pulled in via `module` blocks

As a general rule, you'll want to practice *versioning pinning* with all of your dependencies. That means that you pin each of these three types of dependencies to a specific, fixed, known version. Deployments should be predictable and repeatable: if the code didn't change, then running `apply` should always produce the same result, whether you run it today or three months from now or three years from now. To make that happen, you need to avoid pulling in new versions of dependencies accidentally. Instead, version upgrades should always be an explicit, deliberate action that is visible in the code you check into version control.

Let's go through how to do version pinning for the three types of Terraform dependencies.

To pin the version of the first type of dependency, your Terraform core version, you can use the `required_version` argument in your code. At a bare minimum, you should require a specific major version of Terraform:

```
terraform {
  # Require any 1.x version of Terraform
  required_version = ">= 1.0.0, < 2.0.0"
}
```

This is critical, because each major release of Terraform is backward incompatible: e.g., the upgrade from 1.0.0 to 2.0.0 will likely include breaking changes, so you don't

want to do it by accident. The preceding code will allow you to use only 1.x.x versions of Terraform with that module, so 1.0.0 and 1.2.3 will work, but if you try to use, perhaps accidentally, 0.14.3 or 2.0.0, and run `terraform apply`, you immediately get an error:

```
$ terraform apply

Error: Unsupported Terraform Core version

This configuration does not support Terraform version 0.14.3. 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, you may want to pin not only the major version but the minor and patch version too:

```
terraform {
  # Require Terraform at exactly version 1.2.3
  required_version = "1.2.3"
}
```

In the past, before the Terraform 1.0.0 release, this was absolutely required, as every release of Terraform potentially included backward-incompatible changes, including to the state file format: e.g., a state file written by Terraform version 0.12.1 could not be read by Terraform version 0.12.0. Fortunately, after the 1.0.0 release, this is no longer the case: as per the officially published Terraform v1.0 Compatibility Promises, upgrades between v1.x releases should require no changes to your code or workflows.

That said, you might still not want to upgrade to a new version of Terraform *accidentally*. New versions introduce new features, and if some of your computers (developer workstations and CI servers) start using those features but others are still on the old versions, you'll run into issues. Moreover, new versions of Terraform may have bugs, and you'll want to test that out in pre-production environments before trying it in production. Therefore, while pinning the major version is the bare minimum, I also recommend pinning the minor and patch version and applying Terraform upgrades intentionally, carefully, and consistently throughout each environment.

Note that, occasionally, you may have to use different versions of Terraform within a single codebase. For example, perhaps you are testing out Terraform 1.2.3 in the stage environment, while the prod environment is still on Terraform 1.0.0. Having to deal with multiple Terraform versions, whether on your own computer or on your CI servers, can be tricky. Fortunately, the open source tool tfenv, the Terraform version manager, makes this much easier.

At its most basic level, you can use `tfenv` to install and switch between multiple versions of Terraform. For example, you can use the `tfenv install` command to install a specific version of Terraform:

```
$ tfenv install 1.2.3
Installing Terraform v1.2.3
Downloading release tarball from
https://releases.hashicorp.com/terraform/1.2.3/terraform_1.2.3_darwin_amd64.zip
Archive:  tfenv_download.ZUS3Qn/terraform_1.2.3_darwin_amd64.zip
  inflating: /opt/homebrew/Cellar/tfenv/2.2.2/versions/1.2.3/terraform
Installation of terraform v1.2.3 successful.
```

tfenv on Apple Silicon (M1, M2)

As of June 2022, `tfenv` did not install the proper version of Terraform on Apple Silicon, such as Macs running M1 or M2 processors (see this open issue for details). The workaround is to set the `TFENV_ARCH` environment variable to `arm64`:

```
$ export TFENV_ARCH=arm64
$ tfenv install 1.2.3
```

You can list the versions you have installed using the `list` command:

```
$ tfenv list
  1.2.3
  1.1.4
  1.1.0
* 1.0.0 (set by /opt/homebrew/Cellar/tfenv/2.2.2/version)
```

And you can select the version of Terraform to use from that list using the `use` command:

```
$ tfenv use 1.2.3
Switching default version to v1.2.3
Switching completed
```

These commands are all handy for working with multiple versions of Terraform, but the real power of `tfenv` is its support for *.terraform-version* files. `tfenv` will automatically look for a *.terraform-version* file in the current folder, as well as all the parent folders, all the way up to the project root—that is, the version control root (e.g., the folder with a *.git* folder in it)—and if it finds that file, any `terraform` command you run will automatically use the version defined in that file.

For example, if you wanted to try out Terraform 1.2.3 in the stage environment, while sticking with Terraform 1.0.0 in the prod environment, you could use the following folder structure:

```
live
 └ stage
   └ vpc
   └ mysql
   └ frontend-app
   └ .terraform-version
 └ prod
```

```
└ vpc
└ mysql
└ frontend-app
└ .terraform-version
```

Inside of *live/stage/.terraform-version*, you would have the following:

```
1.2.3
```

And inside of *live/prod/.terraform-version*, you would have the following:

```
1.0.0
```

Now, any `terraform` command you run in *stage* or any subfolder will automatically use Terraform 1.2.3. You can check this by running the `terraform version` command:

```
$ cd stage/vpc
$ terraform version
Terraform v1.2.3
```

And similarly, any `terraform` command you run in *prod* will automatically use Terraform 1.0.0:

```
$ cd prod/vpc
$ terraform version
Terraform v1.0.0
```

This works automatically on any developer workstation and in your CI server so long as everyone has `tfenv` installed. If you're a Terragrunt user, tgswitch offers similar functionality to automatically pick the Terragrunt version based on a *.terragrunt-version* file.

Let's now turn our attention to the second type of dependency in your Terraform code: providers. As you saw in Chapter 7, to pin provider versions, you can use the `required_providers` block :

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

This code pins the AWS Provider code to any 4.x version (the `~> 4.0` syntax is equivalent to `>= 4.0, < 5.0`). Again, the bare minimum is to pin to a specific major version number to avoid accidentally pulling in backward-incompatible changes. With Terraform 0.14.0 and above, you don't need to pin minor or patch versions

for providers, as this happens automatically due to the *lock file*. The first time you run `terraform init`, Terraform creates a *.terraform.lock.hcl* file, which records the following information:

The exact version of each provider you used
: If you check the *.terraform.lock.hcl* file into version control (which you should!), then in the future, if you run `terraform init` again, on this computer or any other, Terraform will download the *exact* same version of each provider. That's why you don't need to pin the minor and patch version number in the `required_providers` block, as that's the default behavior anyway. If you want to explicitly upgrade a provider version, you can update the version constraint in the `required_providers` block and run `terraform init -upgrade`. Terraform will download new providers that match your version constraints and update the *.terraform.lock.hcl* file; you should review those updates and commit them to version control.

The checksums for each provider
: Terraform records the checksum of each provider it downloads, and on subsequent runs of `terraform init`, it will show an error if the checksum changed. This is a security measure to ensure someone can't swap out provider code with malicious code in the future. If the provider is cryptographically signed (most official HashiCorp providers are), Terraform will also validate the signature as an additional check that the code can be trusted.

Lock Files with Multiple Operating Systems

By default, Terraform only records checksums for the platform you ran `init` on: for example, if you ran `init` on Linux, then Terraform will only record the checksums for Linux provider binaries in *.terraform.lock.hcl*. If you check that file in and, later on, you run `init` on that code on a Mac, you'll get an error, as the Mac checksums won't be in the *.terraform.lock.hcl* file. If your team works across multiple operating systems, you'll need to run the `terraform providers lock` command to record the checksums for every platform you use:

```
terraform providers lock \
  -platform=windows_amd64 \ # 64-bit Windows
  -platform=darwin_amd64 \  # 64-bit macOS
  -platform=darwin_arm64 \  # 64-bit macOS (ARM)
  -platform=linux_amd64     # 64-bit Linux
```

Finally, let's now look at the third type of dependencies: modules. As discussed in "Module Versioning" on page 133, I strongly recommend pinning module versions

by using `source` URLs (rather than local file paths) with the `ref` parameter set to a Git tag:

```
source = "git@github.com:foo/modules.git//services/hello-world-app?ref=v0.0.5"
```

If you use these sorts of URLs, Terraform will always download the exact same code for the module every time you run `terraform init`.

Now that you've seen how to version your code's dependencies, let's talk about how to version the code itself. As you saw in "Module Versioning" on page 133, you can version your code by using Git tags with semantic versioning:

```
$ 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"
}

module "hello_world_app" {
  # TODO: replace this with your own module URL and version!!
  source = "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
  ami                = data.aws_ami.ubuntu.id
}

data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}
```

Next, 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"
}
```

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 encounter 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 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:[6]

- The module must live in a public GitHub repo.
- 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., `rds`).
- 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 filenames.
- 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 in to the Terraform Registry with your GitHub account and using the web UI to publish the module. Once your modules are in the Registry, your team can use a web UI to discover modules and learn how to use them.

6 You can find the full details on publishing modules on the Terraform website.

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 an open source RDS module from the Terraform Registry:

```
module "rds" {
  source  = "terraform-aws-modules/rds/aws"
  version = "4.4.0"

  # (...)
}
```

If you are a customer of HashiCorp's Terraform Cloud or 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 reside in the *modules* folder directly alongside your Terraform code: e.g., you might have a *modules/packer* folder that contains a Packer template and some Bash scripts you use to configure an AMI right next to the *modules/asg-rolling-deploy* Terraform module you use to deploy that AMI.

However, occasionally, you 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:

- Provisioners
- Provisioners with `null_resource`
- 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), and `file` (copy files to a remote resource).[7]

You can add provisioners 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           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

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

When you run `terraform apply` on this code, it prints "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.
```

7 You can find the full list of provisioners on the Terraform website.

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 do the following:

Communicate with the EC2 Instance over the network
: You already know how to allow this with a security group.

Authenticate to the EC2 Instance
: The `remote-exec` provisioner supports SSH and WinRM connections.

Since the examples in this book have you launch Linux (Ubuntu) EC2 Instances, you'll want to use SSH authentication. And that means you'll need to configure SSH keys. Let's begin 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 out this code, 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 is 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 begin writing the code for the EC2 Instance:

```
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}

resource "aws_instance" "example" {
  ami                    = data.aws_ami.ubuntu.id
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]
  key_name               = aws_key_pair.generated_key.key_name
}
```

Just about all of this code should be familiar to you: it's using the `aws_ami` data source to find Ubuntu AMI and using the `aws_instance` resource to deploy that AMI on a `t2.micro` instance, associating that instance with the security group you created earlier. The only new item is the use of the `key_name` attribute in the `aws_instance` resource to instruct 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, add the `remote-exec` provisioner to the `aws_instance` resource:

```
resource "aws_instance" "example" {
  ami                    = data.aws_ami.ubuntu.id
  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. You do this by using a `connection` block:

```
resource "aws_instance" "example" {
  ami                    = data.aws_ami.ubuntu.id
  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 autogenerated private key. 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 five minutes, but you can configure it). 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 that 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 after you run `terraform destroy`, 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 instruct 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:

- 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 earlier 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 when using Terraform to deploy to AWS).
- 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 take effect only while Terraform is running and don't work with ASGs at all.
- 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 real advantage to using a provisioner to execute code on an EC2 Instance is that User Data scripts are limited to a length of 16 KB, whereas provisioner scripts can be arbitrarily long.

Provisioners with null_resource

Provisioners can be defined only 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 that 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 re-created, therefore forcing any provisioners within it to be reexecuted. For example, if you want to execute a provisioner within a `null_resource` every single time you run `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 correct 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 echoes back to stdout any data it receives on stdin. Therefore, any data you 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," since 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 of 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 8-2, and explicitly identify the items you'll be implementing and the items you'll be skipping. Use the results of this checklist, plus Table 8-1, 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 the versions for all your dependencies, including Terraform core, your Terraform providers, and Terraform modules you depend on.
4. Create a *test* folder, and write automated tests for each example.

That last bullet point—writing automated tests for your infrastructure code—is what we'll focus on next, as we move on to Chapter 9.

CHAPTER 9

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 need 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, 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 will 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:

- 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, you can find all of the code examples in the book on GitHub.

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'
    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
```

This code will send a 200 response with the body "Hello, World" for the `/` URL, a 201 response with a JSON body for the `/api` 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
```

You can test this server using a web browser or `curl`:

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

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, you can't use localhost. This applies to most IaC tools, 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 why it's essential to have easy-to-deploy examples in the *examples* folder for each module, as described in Chapter 8. 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"
}

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

  alb_name   = "terraform-up-and-running"
  subnet_ids = data.aws_subnets.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 deployment 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, because the infrastructure you're testing includes a load balancer that responds to HTTP requests. For other types of infrastructure, you'll need 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 might need to SSH to the server and execute some commands locally to test it; and so on. So although you can use the same basic test structure described in this chapter with any type of infrastructure, the validation steps will change depending on what you're testing.

In short, 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*, in which 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 can 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.

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 root account, as you saw in Chapter 7.

At a minimum, you should create a culture in which developers clean up whatever they deploy when they are done testing by running `terraform destroy`. Depending on your deployment environment, you might also be able to find tools that you can run on a regular schedule (e.g., a cron job) to automatically clean up unused or old resources, such as cloud-nuke and aws-nuke.

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 48 hours old, based on the assumption that any infrastructure a developer fired up for manual testing is no longer necessary after a couple of days:

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

Warning: Lots of Coding Ahead

Writing automated tests for infrastructure code is not for the faint 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, 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.

Automated Tests

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 10, you can set up a CI server to run these tests after every single commit and then 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:

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—for example, databases, web services, even the filesystem—are replaced with *test doubles* or *mocks* that allow you to finely control the behavior of those dependencies (e.g., by returning a hardcoded response from a database mock) to test that your code handles a variety of scenarios.

Integration tests

Integration tests verify that multiple units work together correctly. In a general-purpose programming language, an integration test consists of code that validates that 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 communicates with the database, you might want to test it with a real database, but mock out other dependencies, such as the app's authentication system.

End-to-end tests

End-to-end tests involve running your entire architecture—for example, your apps, your data stores, your load balancers—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 that 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 ensure the basic building blocks fit together correctly. And just because different parts of your system work correctly doesn't mean they will 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:

```
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 that calls the `do_GET` method directly is tricky, as you'd have to either instantiate real `WebServer`, `request`, and `response` objects, or create test doubles of them, both of which require a fair bit of work. When you find it difficult to write unit tests, that's often a code smell and indicates that 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:

```
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:

Simple values as inputs
: The `Handlers` class does not depend on `HTTPServer`, `HTTPRequest`, or `HTTPResponse`. Instead, all of its inputs are simple values, such as the `path` of the URL, which is a string.

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:

```
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 are three unit tests that check each endpoint in the `Handlers` class:

```
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

  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
end
```

And here's how you run the tests:

```
$ 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 whether 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.

Unit testing Terraform code

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 reusable module such as the `alb` module you created in Chapter 8. How would you test this module?

With Ruby, to write unit tests, you needed 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, and so on—you'll realize that 99% of what this code is doing is communicating with 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 reusable module) to get feedback as quickly as possible.

This means that the basic strategy for writing unit tests for Terraform is as follows:

1. Create a small, 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: for example, for an ALB, you'd validate

2 In limited cases, it is possible to override the endpoints Terraform uses to communicate with providers, such as overriding the endpoints Terraform uses to talk to AWS to instead talk to a mocking tool called LocalStack. This works 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 ASGs and ALBs, your `terraform apply` might succeed, but does that tell you anything useful about whether your code would have actually deployed a working app on top of that infrastructure?

it by sending an HTTP request and checking that you receive 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, all of 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) across a wide variety of environments (e.g., AWS, Google Cloud, Kubernetes). 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 just described, where you `terraform apply` some code, validate that it works, and then run `terraform destroy` at the end to clean up.

To use Terratest, you need to do the following:

1. Install Go (minimum version 1.13).
2. Create a folder for your test code: e.g., a folder named *test*.
3. Run `go mod init <NAME>` in the folder you just created, where `NAME` is the name to use for this test suite, typically in the format `github.com/<ORG_NAME>/<PROJECT_NAME>` (e.g., `go mod init github.com/brikis98/terraform-up-and-running`). This should create a *go.mod* file, which is used to track the dependencies of your Go code.

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:

```
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:

```
go test -v
```

The `-v` flag means verbose, which ensures that the test always shows all log output. You should see output that looks something like this:

```
=== RUN   TestGoIsWorking

If you see this text, it's working!

--- PASS: TestGoIsWorking (0.00s)
PASS
ok      github.com/brikis98/terraform-up-and-running-code       0.192s
```

If that's working, feel free to delete *go_sanity_test.go*, and 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:

```
package test

import (
        "testing"
)

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

The first step is to direct Terratest to where your Terraform code resides by using the `terraform.Options` type:

```
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 (you should update the relative path in `TerraformDir` to point to the folder where you created that example).

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 `InitAndApply` 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 preceding code is already a fairly useful unit test, since 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 output 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 returns the output of the given name, or it fails the test if that output doesn't exist or is empty. The preceding code 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 using the `http_helper` package (make sure to add `github.com/gruntwork-io/terratest/modules/http-helper` as an import):

```
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,
                nil,
                expectedStatus,
                expectedBody,
                maxRetries,
                timeBetweenRetries,
        )
}
```

The `http_helper.HttpGetWithRetry` method will make an HTTP GET request to the URL you pass in and check that the response has the expected status code and body. If it doesn't, the method will retry up to the specified maximum number of retries, with the specified amount of time between retries. If it eventually achieves the expected response, the test will pass; if the maximum number of retries is reached without the expected response, the test will fail. This sort of retry logic is very common in infrastructure testing, as there is usually a period of time between when `terraform apply` finishes and when the deployed infrastructure is completely ready (i.e., it takes time for health checks to pass, DNS updates to propagate, and so on),

and as you don't know exactly how long that'll take, the best option is to retry until it works or you hit a timeout.

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, if any of the code before that causes a test failure (e.g., `HttpGetWithRetry` fails because the ALB is misconfigured), 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 by 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,
                nil,
                expectedStatus,
                expectedBody,
                maxRetries,
                timeBetweenRetries,
        )
}
```

Note that the `defer` is added early in the code, even before the call to `terraform.InitAndApply`, to ensure that 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!

Terratest Version

The test code in this book was written with Terratest v0.39.0. Terratest is still a pre-1.0.0 tool, so newer releases may contain backward-incompatible changes. To ensure the test examples in this book work as written, I recommend installing Terratest specifically at version v0.39.0, and not the latest version. To do that, go into *go.mod* and add the following to the end of the file:

```
require github.com/gruntwork-io/terratest v0.39.0
```

Since this is a brand-new Go project, as a one-time action, you need to tell Go to download dependencies (including Terratest). The easiest way to do that at this stage is to run the following:

```
go mod tidy
```

This will download all your dependencies and create a *go.sum* file to lock the exact versions you used.

Next, since this test deploys infrastructure to AWS, before running the test, you need to authenticate to your AWS account as usual (see "Other AWS Authentication Options" on page 44). 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 might 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.

After you've authenticated to an AWS account that 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]

(...)

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, not only causing the tests to fail but also preventing the cleanup code (i.e., `terraform destroy`) from running. This ALB test should take closer to five minutes, but whenever running a Go test that deploys real infrastructure, it's safer to set an extra-long timeout to avoid the test being killed partway 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 of the key stages 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 five minutes, you can now automatically find out whether 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.

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 making an HTTP GET to `example.org` and proxying the response. When you `curl` this endpoint, you get the following:

```
$ 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, your unit tests would be subject to the behavior of an external dependency (in this case, `example.org`). This has a number of downsides:

- If that dependency has an outage, your tests will fail, even though there's nothing wrong with your code.
- 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 need to constantly keep updating the test code, even though there's nothing wrong with the implementation.
- If that dependency were slow, your tests would be slow, which negates one of the main benefits of unit tests, the fast feedback loop.
- 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.

Although working with real dependencies might 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*, in which you make it possible to pass in (or "inject") external dependencies from outside your code, rather than hardcoding them within your code.

For example, the `Handlers` class shouldn't need to deal with all of 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)
    @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`:

```
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:

```
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:

```
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
```

Rerun 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 three 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"
}

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
  ami                = data.aws_ami.ubuntu.id
}

data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
}
```

The dependency problem becomes apparent when you spot the parameters `db_remote_state_bucket` and `db_remote_state_key`: the `hello-world-app` module assumes that 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 these two parameters. The goal here is to create a unit test for the `hello-world-app` module, and although 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 might 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_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
```

```
  }
}
```

This convention makes it easier for users of your code to know, 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 that 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 `null`.

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, as soon as 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
  ami                = data.aws_ami.ubuntu.id
}

module "mysql" {
  source = "../../../modules/data-stores/mysql"

  db_name     = var.db_name
  db_username = var.db_username
  db_password = var.db_password
}
```

Because 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 also type-safe function composition.

But before that can work, you'll need to finish refactoring the code. Because the MySQL configuration can be passed in as an input, this means that 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`:

```
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_subnets" "default" {
  count = var.subnet_ids == null ? 1 : 0
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}
```

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_subnets.default[0].ids
      : var.subnet_ids
  )
}
```

Note that because the data sources use the `count` parameters, they are now arrays, so any time you reference them, you need 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 one of the local variables you just added instead. Start by updating the `aws_subnets` data source to use `local.vpc_id`:

```
data "aws_subnets" "default" {
  count = var.subnet_ids == null ? 1 : 0
  filter {
    name   = "vpc-id"
    values = [local.vpc_id]
  }
}
```

Then, set the `subnet_ids` parameter of the `alb` module to `local.subnet_ids`:

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

  alb_name   = "hello-world-${var.environment}"
  subnet_ids = local.subnet_ids
}
```

In the `asg` module, make the following updates: set the `subnet_ids` parameter to `local.subnet_ids`, and in the `user_data` variables, update `db_address` and `db_port` to read data from `local.mysql_config`.

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

  cluster_name  = "hello-world-${var.environment}"
  ami           = var.ami
  instance_type = var.instance_type

  user_data = templatefile("${path.module}/user-data.sh", {
    server_port = var.server_port
    db_address  = local.mysql_config.address
    db_port     = local.mysql_config.port
    server_text = var.server_text
  })

  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
}
```

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 parameters because 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
  }
}
```

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
  ami                = data.aws_ami.ubuntu.id
}
```

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)

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

        http_helper.HttpGetWithRetryWithCustomValidation(
                t,
                url,
                nil,
                maxRetries,
                timeBetweenRetries,
                func(status int, body string) bool {
                        return status == 200 &&
                                strings.Contains(body, "Hello, World")
                },
        )
}
```

This code is nearly identical to the unit test for the `alb` example, with only two differences:

- The `TerraformDir` setting is pointing to the `hello-world-app` example instead of the `alb` example (be sure to update the path as necessary for your filesystem).
- Instead of using `http_helper.HttpGetWithRetry` to check for a 404 response, the test is using the `http_helper.HttpGetWithRetryWithCustomValidation` method to check for a 200 response and a body that contains the text "Hello, World." That's because the User Data script of the `hello-world-app` module

returns a 200 OK response that includes not only the server text but also other text, including HTML.

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, as soon as it gets back the expected response, 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. Although four to five 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 want to run as many tests in parallel as you can.

To instruct 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—for example, the ASG, security group, and ALB—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 that 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"
}
```

Next, 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_subnets.default.ids
}
```

Now, set this variable to a unique value in *test/alb_example_test.go*:

```
package test

import (
        "fmt"
        "github.com/stretchr/testify/require"

        "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()),
                },
        }

        // (...)
}
```

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, six-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 by 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
  ami                = data.aws_ami.ubuntu.id
}
```

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.

Note that, by default, the number of tests Go will run in parallel is equal to how many CPUs you have on your computer. So if you only have one CPU, then by default, the tests will still run serially, rather than in parallel. You can override this setting by setting the `GOMAXPROCS` environment variable or by passing the `-parallel` argument to the `go test` command. For example, to force Go to run up to two tests in parallel, you would run the following:

```
$ go test -v -timeout 30m -parallel 2
```

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 because they all try to run `terraform init` and end up overwriting one another'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 that 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. 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 do the following:

1. Run the web server on localhost so that 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 down the web server. 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. Let's run all of 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 slowdown of roughly 387 times. 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. I'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, 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 hardcoded 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"
}
```

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_username": "admin",
                        "db_password": "password",
                    },
            }
    }
```

Not much new so far: the code points `terraform.Options` at the passed-in directory and sets the `db_name`, `db_username`, 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 hardcoded values:

```
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 hardcoded values are a big problem for testing, because 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" on page 94), but that would still require access to the 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" on page 91. 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:

```
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_REGION_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_username": "admin",
                        "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 *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 values 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
  ami                = data.aws_ami.ubuntu.id
}
```

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 to which the `mysql` module just wrote. 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,
                nil,
                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 whether it worked:

```
$ go test -v -timeout 30m -run "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 it takes more than twice as long (10–15 minutes rather than 4–5 minutes). In general, there's not much that 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 might 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 of 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 making changes only to the `hello-world-app` module, rerunning 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 to 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. Rerun `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 quickly do that inner loop in step 3 is the key to fast, iterative development with Terraform. To support this, you need to break your test code into *stages*, in which you can choose the stages to execute and those that you can skip.

Terratest supports this natively with the `test_structure` package. The idea is that you wrap each stage of your test in a function with a name, and you can then direct 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 then filling in the underlying methods later:

```
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:

`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()`.

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.

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 `dbOpts` 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 can run each test stage as part of a 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 might want to run `deployDb` but skip `teardownDb`, and then 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. 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 need to do a little work to save and load data to and from disk. These simple changes unlock an important ability: you can instruct 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. Because 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 direct 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 rerun the tests, but this time, direct them to skip not only the teardown stages but also the `mysql` module deploy stage (because `mysql` is still running) so that the only things that execute are `deploy app` 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 to 15 minutes after every change, you can try out new changes in 10 to 60 seconds (depending on the change). Given that you're likely to rerun 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 that only the teardown stages are 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

After 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`:

```
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 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 rerun:

```
$ 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 that you might 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 and 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 run `terraform destroy`—this is rarely done in practice. The reason for this has to do with the *test pyramid*, which you can see in Figure 9-1.

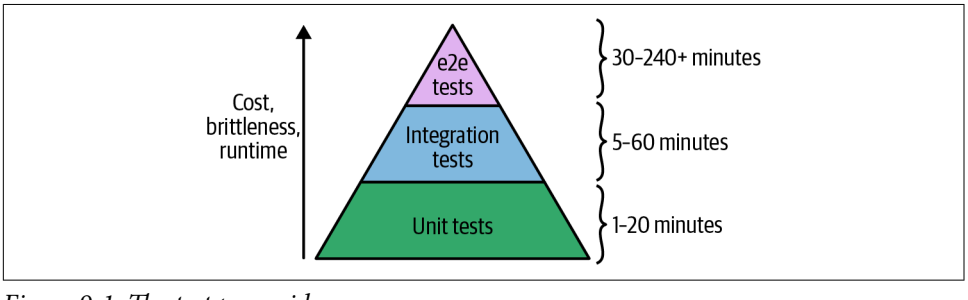


Figure 9-1. The test pyramid.

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 runtime 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 required a fair amount of work with name-spacing, dependency injection, retries, error handling, and test stages to test even a relatively simple `hello-world-app` module. With larger and more complicated infrastructure, this only becomes more difficult. Therefore, you want to do as much of your testing as low in the pyramid as you can because 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:

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 because the feedback loop is simply too slow. You'd probably run such a test suite only 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 whether it worked. That limits you to roughly one bug fix attempt per day. In these sorts of situations, what actually happens is developers begin blaming others for test failures, convince management to deploy despite the test failures, and eventually ignore the test failures entirely.

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, suppose that a single resource (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). This 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? 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% × 99.9% = 99.8%. With three resources, the odds are 99.9% × 99.9% × 99.9% = 99.7%. With N resources, the formula is $99.9\%^N$.

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%. This 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, suppose that you had an integration test of 3 modules that deployed 60 resources. Now the odds of success are $99.9\%^{60}$ = 94.1%. Again, with enough retry logic, you can typically make these tests stable enough to be useful. So what happens if you want 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%. This 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 because end-to-end tests take so long, you get only one attempt, maybe two, 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 code, the end-to-end test does the following:
   a. Applies the infrastructure change to the test environment.
   b. Runs validations against the test environment (e.g., uses 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 applying only incremental changes, you're reducing the number of resources that are being deployed at test time from several hundred to just a handful so that 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 also that the *deployment process* for that infrastructure works correctly, too.

Other Testing Approaches

Most of this chapter has focused on testing your Terraform code by doing a full `apply` and `destroy` cycle. This is the gold standard of testing, but there are three other types of automated tests you can use:

- Static analysis
- Plan testing
- Server testing

Just as unit, integration, and end-to-end tests each catch different types of bugs, each of the testing approaches just mentioned will catch different types of bugs as well, so you'll most likely want to use several of these techniques together to get the best results. Let's go through these new categories one at a time.

Static analysis

Static analysis is the most basic way to test your Terraform code: you parse the code and analyze it without actually executing it in any way. Table 9-1 shows some of the tools in this space that work with Terraform and how they compare in terms of popularity and maturity, based on stats I gathered from GitHub in February 2022.

Table 9-1. A comparison of popular static analysis tools for Terraform

	`terraform validate`	`tfsec`	`tflint`	Terrascan
Brief description	Built-in Terraform command	Spot potential security issues	Pluggable Terraform linter	Detect compliance and security violations
License	(same as Terraform)	MIT	MPL 2.0	Apache 2.0
Backing company	(same as Terraform)	Aqua Security	(none)	Accurics
Stars	(same as Terraform)	3,874	2,853	2,768
Contributors	(same as Terraform)	96	77	63
First release	(same as Terraform)	2019	2016	2017
Latest release	(same as Terraform)	v1.1.2	v0.34.1	v1.13.0
Built-in checks	Syntax checks only	AWS, Azure, GCP, Kubernetes, DigitalOcean, etc.	AWS, Azure, and GCP	AWS, Azure, GCP, Kubernetes, etc.
Custom checks	Not supported	Defined in YAML or JSON	Defined in a Go plugin	Defined in Rego

The simplest of these tools is `terraform validate`, which is built into Terraform itself, which can catch syntax issues. For example, if you forgot to set the `alb_name` parameter in *examples/alb*, and you ran `validate`, you would get output similar to the following:

```
$ terraform validate

| Error: Missing required argument
|
|   on main.tf line 20, in module "alb":
|   20: module "alb" {
|
| The argument "alb_name" is required, but no definition was found.
```

Note that `validate` is limited solely to syntactic checks, whereas the other tools allow you to enforce other types of policies. For example, you can use tools such as `tfsec` and `tflint` to enforce policies, such as:

- Security groups cannot be too open: e.g., block inbound rules that allow access from all IPs (CIDR block `0.0.0.0/0`).
- All EC2 Instances must follow a specific tagging convention.

The idea here is to *define your policies as code*, so you can enforce your security, compliance, and reliability requirements as code. In the next few sections, you'll see several other policy as code tools.

Strengths of static analysis tools

- They run fast.
- They are easy to use.
- They are stable (no flaky tests).
- You don't need to authenticate to a real provider (e.g., to a real AWS account).
- You don't have to deploy/undeploy real resources.

Weaknesses of static analysis tools

- They are very limited in the types of errors they can catch. Namely, they can only catch errors that can be determined from statically reading the code, without executing it: e.g., syntax errors, type errors, and a small subset of business logic errors. For example, you can detect a policy violation for static values, such as a security group hardcoded to allow inbound access from CIDR block `0.0.0.0/0`, but you can't detect policy violations from dynamic values, such as the same security group but with the inbound CIDR block being read in from a variable or file.
- These tests aren't checking functionality, so it's possible for all the checks to pass and the infrastructure still doesn't work!

Plan testing

Another way to test your code is to run `terraform plan` and to analyze the plan output. Since you're executing the code, this is more than static analysis, but it's less than a unit or integration test, as you're not executing the code fully: in particular, `plan` executes the read steps (e.g., fetching state, executing data sources) but not the write steps (e.g., creating or modifying resources). Table 9-2 shows some of the tools that do `plan` testing and how they compare in terms of popularity and maturity, based on stats I gathered from GitHub in February 2022.

Table 9-2. A comparison of popular plan testing tools for Terraform

	Terratest	Open Policy Agent (OPA)	HashiCorp Sentinel	Checkov	terraform-compliance
Brief description	Go library for IaC testing	General-purpose policy engine	Policy-as-code for HashiCorp enterprise products	Policy-as-code for everyone	BDD test framework for Terraform
License	Apache 2.0	Apache 2.0	Commercial / proprietary license	Apache 2.0	MIT
Backing company	Gruntwork	Styra	HashiCorp	Bridgecrew	(none)
Stars	5,888	6,207	(not open source)	3,758	1,104
Contributors	157	237	(not open source)	199	36
First release	2016	2016	2017	2019	2018

	Terratest	Open Policy Agent (OPA)	HashiCorp Sentinel	Checkov	terraform-compliance
Latest release	v0.40.0	v0.37.1	v0.18.5	2.0.810	1.3.31
Built-in checks	None	None	None	AWS, Azure, GCP, Kubernetes, etc.	None
Custom checks	Defined in Go	Defined in Rego	Defined in Sentinel	Defined in Python or YAML	Defined in BDD

Since you're already familiar with Terratest, let's take a quick look at how you can use it to do `plan` testing on the code in *examples/alb*. If you ran `terraform plan` manually, here's a snippet of the output you'd get:

```
Terraform will perform the following actions:

  # module.alb.aws_lb.example will be created
  + resource "aws_lb" "example" {
      + arn                        = (known after apply)
      + load_balancer_type         = "application"
      + name                       = "test-4Ti6CP"
      (...)
    }

  (...)

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

How can you test this output programmatically? Here's the basic structure of a test that uses Terratest's `InitAndPlan` helper to run `init` and `plan` automatically:

```
func TestAlbExamplePlan(t *testing.T) {
        t.Parallel()

        albName := fmt.Sprintf("test-%s", random.UniqueId())

        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": albName,
                },
        }

        planString := terraform.InitAndPlan(t, opts)
}
```

Even this minimal test offers some value, in that it validates that your code can successfully run `plan`, which checks that the syntax is valid and that all the read API calls work. But you can go even further. One small improvement is to check that you

get the expected counts at the end of the plan: "5 to add, 0 to change, 0 to destroy." You can do this using the `GetResourceCount` helper

```
// An example of how to check the plan output's add/change/destroy counts
resourceCounts := terraform.GetResourceCount(t, planString)
require.Equal(t, 5, resourceCounts.Add)
require.Equal(t, 0, resourceCounts.Change)
require.Equal(t, 0, resourceCounts.Destroy)
```

You can do an even more thorough check by using the `InitAndPlanAndShowWithStructNoLogTempPlanFile` helper to parse the `plan` output into a `struct`, which gives you programmatic access to all the values and changes in that `plan` output. For example, you could check that the `plan` output includes the `aws_lb` resource at address `module.alb.aws_lb.example` and that the `name` attribute of this resource is set to the expected value, as follows:

```
// An example of how to check specific values in the plan output
planStruct :=
        terraform.InitAndPlanAndShowWithStructNoLogTempPlanFile(t, opts)

alb, exists :=
        planStruct.ResourcePlannedValuesMap["module.alb.aws_lb.example"]
require.True(t, exists, "aws_lb resource must exist")

name, exists := alb.AttributeValues["name"]
require.True(t, exists, "missing name parameter")
require.Equal(t, albName, name)
```

The strength of Terratest's approach to plan testing is that it's extremely flexible, as you can write arbitrary Go code to check whatever you want. But this very same factor is also, in some ways, a weakness, as it makes it harder to get started.

Some teams prefer a more declarative language for defining their policies as code. In the last few years, Open Policy Agent (OPA) has become a popular *policy-as-code* tool, as it allows your to capture you company's policies as code in a declarative language called Rego.

For example, many companies have tagging policies they want to enforce. A common one with Terraform code is to ensure that every resource that is managed by Terraform has a `ManagedBy = terraform` tag. Here is a simple policy called *enforce_tagging.rego* you could use to check for this tag:

```
package terraform

allow {
   resource_change := input.resource_changes[_]
   resource_change.change.after.tags["ManagedBy"]
}
```

This policy will look through the changes in a `terraform plan` output, extract the tag `ManagedBy`, and set an OPA variable called `allow` to `true` if that tag is set or `undefined` otherwise.

Now, consider the following Terraform module:

```
resource "aws_instance" "example" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}
```

This module is not setting the required `ManagedBy` tag. How can we catch that with OPA?

The first step is to run `terraform plan` and to save the output to a plan file:

```
$ terraform plan -out tfplan.binary
```

OPA only operates on JSON, so the next step is to convert the plan file to JSON using the `terraform show` command:

```
$ terraform show -json tfplan.binary > tfplan.json
```

Finally, you can run the `opa eval` command to check this plan file against the *enforce_tagging.rego* policy:

```
$ opa eval \
  --data enforce_tagging.rego \
  --input tfplan.json \
  --format pretty \
  data.terraform.allow

undefined
```

Since the `ManagedBy` tag was not set, the output from OPA is `undefined`. Now, try setting the `ManagedBy` tag:

```
resource "aws_instance" "example" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  tags = {
    ManagedBy = "terraform"
  }
}
```

Rerun `terraform plan`, `terraform show`, and `opa eval`:

```
$ terraform plan -out tfplan.binary

$ terraform show -json tfplan.binary > tfplan.json

$ opa eval \
  --data enforce_tagging.rego \
```

```
  --input tfplan.json \
  --format pretty \
  data.terraform.allow

true
```

This time, the output is `true`, which means the policy has passed.

Using tools like OPA, you can enforce your company's requirements by creating a library of such policies and setting up a CI/CD pipeline that runs these policies against your Terraform modules after every commit.

Strengths of plan testing tools

- They run fast—not quite as fast as pure static analysis but much faster than unit or integration tests.
- They are somewhat easy to use—not quite as easy as pure static analysis but much easier than unit or integration tests.
- They are stable (few flaky tests)—not quite as stable as pure static analysis but much more stable than unit or integration tests.
- You don't have to deploy/undeploy real resources.

Weaknesses of plan testing tools

- They are limited in the types of errors they can catch. They can catch more than pure static analysis but nowhere near as many errors as unit and integration testing.
- You have to authenticate to a real provider (e.g., to a real AWS account). This is required for `plan` to work.
- These tests aren't checking functionality, so it's possible for all the checks to pass and the infrastructure still doesn't work!

Server testing

There are a set of testing tools that are focused on testing that your servers (including virtual servers) have been properly configured. I'm not aware of any common name for these sorts of tools, so I'll call it *server testing*. These are not general-purpose tools for testing all aspects of your Terraform code. In fact, most of these tools were originally built to be used with configuration management tools, such as Chef and Puppet, which were entirely focused on launching servers. However, as Terraform has grown in popularity, it's now very common to use it to launch servers, and these tools can be helpful for validating that the servers you launched are working. Table 9-3 shows some of the tools that do server testing and how they compare in terms of popularity and maturity, based on stats I gathered from GitHub in February 2022.

Table 9-3. A comparison of popular server testing tools

	InSpec	Serverspec	Goss
Brief description	Auditing and testing framework	RSpec tests for your servers	Quick and easy server testing/validation
License	Apache 2.0	MIT	Apache 2.0
Backing company	Chef	(none)	(none)
Stars	2,472	2,426	4,607
Contributors	279	128	89
First release	2016	2013	2015
Latest release	v4.52.9	v2.42.0	v0.3.16
Built-in checks	None	None	None
Custom checks	Defined in a Ruby-based DSL	Defined in a Ruby-based DSL	Defined in YAML

Most of these tools provide a simple *domain-specific language* (DSL) for checking that the servers you've deployed conform 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
```

Strengths of server testing tools

- They make it easy to validate specific properties of servers. The DSLs these tools offer are much easier to use for common checks than doing it all from scratch.
- You can build up a library of policy checks. Because each individual check is quick to write, per the previous bullet point, these tools tend to be a good way to validate a checklist of requirements, especially around compliance (e.g., PCI compliance, HIPAA compliance, etc.).
- They can catch many types of errors. Since you actually have to run `apply` and you validate a real, running server, these types of tests catch far more types of errors than pure static analysis or plan testing.

Weaknesses of server testing tools

- They are not as fast. These tests only work on servers that are deployed, so you have to run the full `apply` (and perhaps `destroy`) cycle, which can take a long time.
- They are not as stable (some flaky tests). Since you have to run `apply` and wait for real servers to deploy, you will hit various intermittent issues and occasionally have flaky tests.
- You have to authenticate to a real provider (e.g., to a real AWS account). This is required for the `apply` to work to deploy the servers, plus, these server testing tools all require additional authentication methods—e.g., SSH—to connect to the servers you're testing.
- You have to deploy/undeploy real resources. This takes time and costs money.
- They only thoroughly check that servers work and not other types of infrastructure.
- These tests aren't checking functionality, so it's possible for all the checks to pass and the infrastructure still doesn't work!

Conclusion

Everything in the infrastructure world is continuously changing: Terraform, Packer, Docker, Kubernetes, AWS, Google Cloud, Azure, and so on are all moving targets. This 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've put into keeping the code clean, testing it manually, and doing code reviews, as soon as I've taken the time to write automated tests, I've found numerous nontrivial bugs. Something magical happens when you take the time to automate the testing process and, almost without exception, it flushes out problems that 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 caught bugs not only in my own code but also in the tools I was using, including nontrivial 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 it takes 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 3 a.m. dealing with an outage.

This Book Has Tests, Too!

All of the code examples in this book have tests, too. You can find all of the code examples, and all of their corresponding tests, at GitHub.

Throughout this chapter, you saw the basic process of testing Terraform code, including the following key takeaways:

When testing Terraform code, you can't use localhost
: Therefore, you need to do all of your manual testing by deploying real resources into one or more isolated sandbox environments.

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.

Regularly clean up your sandbox environments
: Otherwise, the environments will become unmanageable, and costs will spiral out of control.

You must namespace all of your resources
: This ensures that multiple tests running in parallel do not conflict with one another.

Smaller modules are easier and faster to test
: This was one of the key takeaways in Chapter 8, and it's worth repeating in this chapter, too: smaller modules are easier to create, maintain, use, and test.

You also saw a number of different testing approaches throughout this chapter: unit testing, integration testing, end-to-end testing, static analysis, and so on. Table 9-4 shows the trade-offs between these different types of tests.

Table 9-4. A comparison of testing approaches (more black squares is better)

	Static analysis	Plan testing	Server testing	Unit tests	Integration tests	End-to-end tests
Fast to run	■■■■■	■■■■□	■■■□□	■■□□□	■□□□□	□□□□□
Cheap to run	■■■■■	■■■■□	■■■□□	■■□□□	■□□□□	□□□□□
Stable and reliable	■■■■■	■■■■□	■■■□□	■■□□□	■□□□□	□□□□□
Easy to use	■■■■■	■■■■□	■■■□□	■■□□□	■□□□□	□□□□□
Check syntax	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■	■■■■■
Check policies	■■□□□	■■■■□	■■■■□	■■■■■	■■■■■	■■■■■
Check servers work	□□□□□	□□□□□	■■■■■	■■■■■	■■■■■	■■■■■
Check other infrastructure works	□□□□□	□□□□□	■■□□□	■■■■□	■■■■■	■■■■■
Check all the infrastructure works together	□□□□□	□□□□□	□□□□□	■□□□□	■■■□□	■■■■■

So which testing approach should you use? The answer is: a mix of all of them! Each type of test has strengths and weaknesses, so you have to combine multiple types of tests to be confident your code works as expected. That doesn't mean that you use all the different types of tests in equal proportion: recall the test pyramid and how, in general, you'll typically want lots of unit tests, fewer integration tests, and only a small number of high-value end-to-end tests. Moreover, you don't have to add all the different types of tests at once. Instead, pick the ones that give you the best bang for your buck and add those first. Almost any testing is better than none, so if all you can add for now is static analysis, then use that as a starting point, and build on top of it incrementally.

Let's now move on to Chapter 10, where 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.

CHAPTER 10

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 a 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, you can find all of the code examples in the book on GitHub.

Adopting IaC in Your Team

If your team is used to managing all of your infrastructure by hand, switching to infrastructure as code 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. Because 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:

- Convince your boss
- Work incrementally
- Give your team the time to learn

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 although this developer sees all the benefits of adopting an IaC tool such as Terraform, they aren't seeing all the costs. Here are just a few of the costs of adopting IaC:

Skills gap
: The move to IaC means that your Ops team will need to spend most of its time writing large amounts of code: Terraform modules, Go tests, Chef recipes, and so on. Whereas 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 might 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 may 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, they are used to making all of their changes *directly*: for example, 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; 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 and resources in one project, you are implicitly choosing not to invest that time and resources in other projects. What projects will have to be put on hold so that 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 need 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 Payment Card Industry (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 it has value but that it will 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: for example, Terraform is declarative, it's popular, it's open source. This is one of many areas where developers would do well to learn from salespeople. Most salespeople 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 infrastructure will be far easier to maintain. Instead of talking about the fact that Terraform is popular, talk about how you'll be able to leverage lots of existing modules and plugins to get things done faster. 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 salespeople know an even more effective strategy: focus on the problems. If you watch a great salesperson talking to a customer, you'll notice that it's actually the customer that does most of the talking. The salesperson 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 salespeople 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 the most important problems they are working on that quarter or that year. You might find that those problems would not be solved by IaC. And that's OK! It might 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 although it will pay off in the long term for some scenarios, it won't for others; for example, if you're at a tiny startup with just one Ops person, or you're working on a prototype that might be thrown away in a few months, or you're just working on a side project for fun, 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 might 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 uptime. 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 rollout 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 by half." I guarantee this will get your boss's attention. Use this opportunity to paint a picture for your boss of a world in which 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 double deployments. And then mention that, based on your research, you believe you can deliver this future world using Terraform.

There's no guarantee that 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 3 out of 4 small IT projects (less than $1 million) are completed successfully, only 1 out of 10 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 datacenters must be shut down, and that everyone will "do DevOps" (whatever that means), all within six 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, two to three years later, every one of these companies is still working on the migration, the old datacenter is still running, and no one can tell whether they are really doing DevOps.

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 splitting up the work in such a way that 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] Suppose that 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, because it

1 The Standish Group, "CHAOS Manifesto 2013: Think Big, Act Small," 2013, *https://oreil.ly/ydaWQ*.

2 Dan Milstein, "How to Survive a Ground-Up Rewrite Without Losing Your Sanity," OnStartups.com, April 8, 2013, *https://oreil.ly/nOGrU*.

relies on a new backend. Then, you rewrite the backend, but you don't launch that either, because 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 begin realizing 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 is canceled or put on hold or significantly changed partway 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 three months to clean up tech debt, but after 12 months, it's time to begin shipping new products), and the project ends up getting canceled before completion. With false incrementalism, this gives you the worst possible outcome: you've paid a huge cost and received 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 right away, and making one team successful, you'll begin to build momentum. That team can become your cheerleader and help convince other teams to migrate, too. Fixing the specific deployment issue can 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, 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 happen only 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 that it needs, then your IaC migration is unlikely to be successful. No matter how nice your code is, if your entire team isn't on board with it, here's how it will 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. Then, the inevitable happens: an outage. One of your team members needs to deal with it, and they have two options: either (A) fix the outage the way they've always done it, by making changes manually, which takes a few minutes, or (B) 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.
4. Now, as a result of the manual change, the Terraform code no longer matches what's actually deployed. Therefore, next time someone on your team tries to use Terraform, there's a chance that 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 in which team members make more and more manual changes.
5. 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.

This scenario isn't hypothetical but something I've seen happen at many different companies. They have large, expensive codebases full of beautiful Terraform code that are just gathering dust. To avoid this scenario, you need to not only convince your boss that you should use Terraform but also give everyone on the team the time they need to learn the tool and internalize how to use it so that 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'll 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 will 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 more than 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 ActiveRecord), 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 check out the code repo for this book:

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

By default, this checks out the `main` 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 by 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 9, which you can run as follows:

```
$ cd code/ruby/10-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
```

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 your 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 begin making changes. This is an iterative process in which you make a change, rerun your manual or automated tests to see whether the change worked, make another change, rerun 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
```

You might also update and rerun the automated tests. The idea in this part of the workflow is to optimize the feedback loop so that the time between making a change and seeing whether 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 with a separate code review tool (e.g., Phabricator or Review Board) 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 then click Create. Your team members will now be able to review the changes, as shown in Figure 10-1.

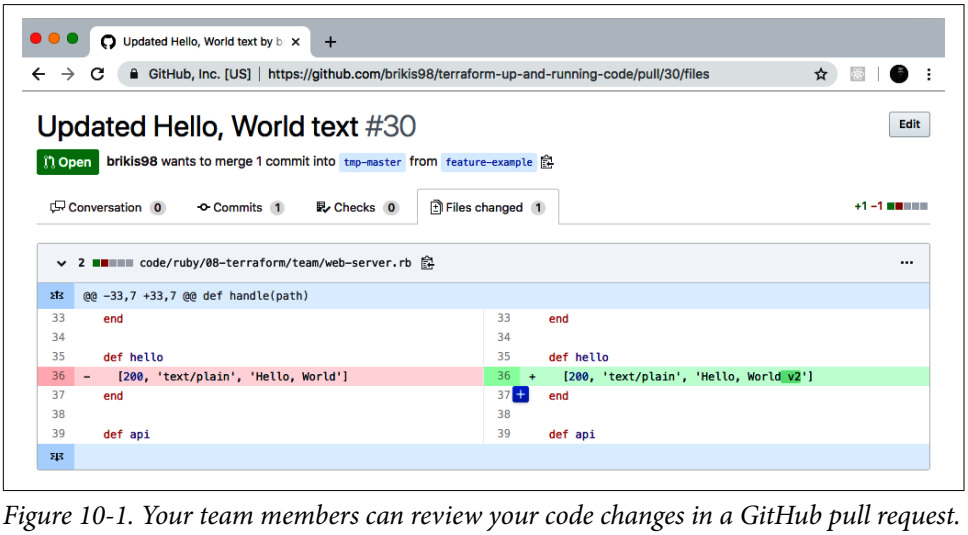


Figure 10-1. Your team members can review your code changes in a GitHub pull request.

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* (CI) server, such as Jenkins, CircleCI, or GitHub Actions. 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 10-2.

You can see in Figure 10-2 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.

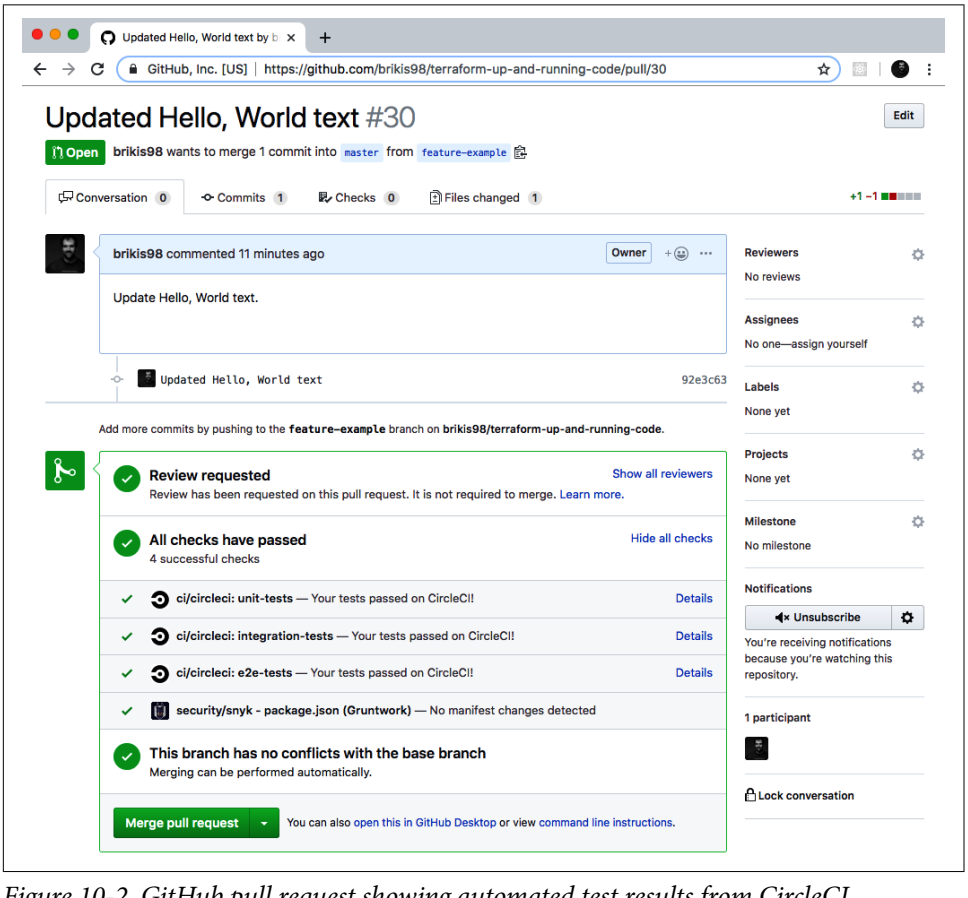


Figure 10-2. GitHub pull request showing automated test results from CircleCI.

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 that the existing tests passed, and ensuring that you've added tests for any new behavior. If everything looks good, your code can be merged into the `main` branch.

The next step is to release the code. If you're using immutable infrastructure practices (as discussed in "Server Templating Tools" on page 7), releasing application code means packaging that code into a new, immutable, versioned artifact. Depending on how you want 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 of 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 preceding code 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 .
```

Thus, when you're 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 want to run it, your architecture, what tools you're using, and so on. Here are a few of the key considerations:

- Deployment tooling
- Deployment strategies
- Deployment server
- Promotion across environments

Deployment tooling

There are many different tools that you can use to deploy your application, depending on how you package it and how you want to run it. Here are 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 ASG. 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`.

Orchestration tools
: There are a number of orchestration tools designed to deploy and manage applications, such as Kubernetes (arguably the most popular Docker orchestration tool), Amazon ECS, HashiCorp Nomad, and Apache Mesos. In Chapter 7, you saw an example of how to use Kubernetes to deploy Docker containers.

Scripts
: Terraform and most orchestration tools support only a limited set of deployment strategies (discussed in the next section). If you have more complicated requirements, you may have to write custom scripts to implement these requirements.

Deployment strategies

There are a number of different strategies that you can use for application deployment, depending on your requirements. Suppose that you have five 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 one 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 of the old copies have been replaced. Rolling deployment with replacement ensures that you never have more than five 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 (you can replace more than one copy of the app at a time if 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 one 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 five copies of it running at the same time. The advantage is that you never have less than five copies of the app running, so you're not running at a reduced capacity during deployment. Note that this deployment strategy can also work with larger batch sizes (if you have the capacity for it, you can deploy five new copies all at once) 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 five 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 at the same time, 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 five 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 five copies of the app running, so you're not running at a reduced capacity during deployment.

Canary deployment

Deploy one 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 becomes worse.

The name comes from the "canary in a coal mine" concept, where miners would take canary birds with them down into the tunnels, and if the tunnels filled with dangerous gases (e.g., carbon monoxide), those gases would affect the canary before the miners, thus providing an early warning to the miners that something was wrong and that they needed to exit immediately, before more damage was 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 affected only 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*, in which 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. Because 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, later on you can enable the feature toggle for a portion of your users via an internal web interface. For example, you might initially enable the new feature only 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.

Consistent environment
: If developers run deployments from their own computers, you'll run into bugs due to differences in how their computer is configured: for example, different operating systems, different dependency versions (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 of these issues by deploying everything from the same CI server.

Better permissions management
: Instead of giving every developer permissions to deploy, you can give solely 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 do for numerous developers with production access.

Promotion across environments

If you're using immutable infrastructure practices, 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 and 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 and 2 again to promote `v0.0.4` to prod.

Because you're running the exact same artifact everywhere, there's a good chance that if it works in one environment, it will work in another. And if you do hit any issues, you can roll back anytime 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 mean code written with any IaC tool (including, of course, Terraform) that you can use to deploy arbitrary infrastructure changes beyond a

single application: for example, 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

On 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. This means that you'll use `git clone` to check out your code, just as before. However, version control for infrastructure code has a few extra requirements:

- *Live* repo and *modules* repo
- Golden Rule of Terraform
- 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 8; that is, 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 it must deploy and manage the infrastructure for every team. Instead, the Ops team can spend most of its time writing infrastructure code, and all of the other teams will be able to work independently, using these modules to get themselves up and running. And because 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, because it means that your infrastructure code matches what's actually deployed. If the output sometimes shows a small diff, and you hear the occasional excuse from your team members ("Oh, right, I tweaked that one thing by hand and forgot to update the code"), your code doesn't match reality, and you might soon be in trouble. If `terraform plan` fails completely with weird errors, or every `plan` shows a gigantic diff, 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 Golden Rule of Terraform*:

> The main 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:

3 See the Gruntwork Infrastructure as Code Library.

"…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*. After you begin 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 IaC 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-of-band changes so that 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 have only 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" on page 94, 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 copying and pasting.

"The main branch…"

You should have to look at only a single branch to understand what's actually deployed in production. Typically, that branch will be `main`. This means that all changes that affect the production environment should go directly into `main` (you can create a separate branch but only to create a pull request with the intention of merging that branch into `main`), and you should run `terraform apply` only for the production environment against the `main` 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. Even though 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, suppose that 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           = data.aws_ami.ubuntu.id
  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           = data.aws_ami.ubuntu.id
  instance_type = "t2.medium"
}
```

Here's what Anna sees when she runs `terraform plan`:

```
$ terraform plan

(...)

Terraform will perform the following actions:

  # aws_instance.foo will be updated in-place
  ~ resource "aws_instance" "foo" {
        ami                          = "ami-0fb653ca2d3203ac1"
        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           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  tags = {
    Name = "foo"
  }
}
```

Note that Anna's changes are already deployed in staging, but because 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 following log output 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-0fb653ca2d3203ac1"
        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 might 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. Because 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"; for example, you can't deploy an AWS ASG onto your own laptop. As discussed in "Manual Testing Basics" on page 317, 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 by 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 begin to make changes, just as with application code. Every time you make a change, you can rerun `terraform apply` to deploy those changes and rerun `curl` to see whether those changes worked:

```
$ curl hello-world-stage-477699288.us-east-2.elb.amazonaws.com
Hello, World v2
```

Or you can rerun `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 that you can get feedback on your changes as quickly as possible. In "Test

stages" on page 354, you saw that you can use these test stages to rerun only specific stages of a test suite, dramatically shortening the feedback loop.

As you make changes, be sure to regularly commit your work:

```
$ git commit -m "Updated Hello, World text"
```

Submit Changes for Review

After 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. Although well-written code can tell you *what* it does, no programming language that 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 that 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, because 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 might 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 begins 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 8, 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 and 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 9 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. Because the file layout for Terraform also determines the way Terraform state is stored, you should be especially mindful of how file layout affects your ability to provide isolation guarantees, such as ensuring that changes in a staging environment cannot accidentally cause problems in production. In a code review, you might want to enforce the file layout described in "Isolation via File Layout" on page

4 Writing the README first is called Readme-Driven Development.

100, 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 more important thing is that you are consistent throughout your codebase.

Terraform has a built-in `fmt` command that can reformat code to a consistent style automatically:

```
$ terraform fmt
```

I recommend running this command as part of a commit hook to ensure that all code committed to version control uses 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 9. 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 10-3.

Figure 10-3. Atlantis can automatically add the output of the `terraform plan` command as a comment on your pull requests.

Terraform Cloud and Terraform Enterprise, HashiCorp's paid tools, both support running `plan` automatically on pull requests as well.

Merge and Release

After 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 `main` 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 VM 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:

- Deployment tooling
- Deployment strategies
- Deployment server
- 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 might 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. Although 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 Cloud and Terraform Enterprise
: HashiCorp's paid products provide a web UI that 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 copying and pasting.

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

For most types of infrastructure changes, Terraform doesn't offer any built-in deployment strategies: for example, there's no way to do a blue-green deployment for a VPC change, and there's no way to feature toggle a database change. You're essentially limited to `terraform apply`, which either works or it doesn't. A small subset of changes do support deployment strategies, such as the zero-downtime rolling deployment in the `asg-rolling-deploy` module you built in previous chapters, but these are the exceptions and not the norm.

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; for example, 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, that's because there is no reasonable way to roll back many types of infrastructure changes: for example, 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!

Therefore, you should expect errors to happen and ensure you have a first-class way to deal with them:

Retries

Certain types of Terraform errors are transient and go away if you rerun `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 partway 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 Amazon 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 cleaning up the workspace after a build)! If you can still access this file after a failed deployment, as soon as 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 showing the ID of the lock. If you're absolutely sure this is an accidentally leftover 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 of 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, GitHub Actions, Terraform Cloud, 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; for example, to deploy to an ASG, the CI server typically needs only a few specific `ec2` and `autoscaling` permissions. However, to be able to deploy arbitrary infrastructure code changes (e.g., your Terraform code might try to deploy a database or a VPC or an entirely new AWS account), the CI server needs arbitrary permissions—that is, admin permissions. And that's a problem.

The reason it's a problem is that CI servers are (a) notoriously hard to secure,[5] (b) accessible to all the developers at your company, and (c) used to execute arbitrary code. Adding permanent admin permissions to this mix is just asking for trouble! You'd effectively be giving every single person on your team admin permissions and turning your CI server into a very high-value target for attackers.

There are a few things you can do to minimize this risk:

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.).

5 See 10 real-world stories of how we've compromised CI/CD pipelines for some eye-opening examples.

Don't expose your CI server on the public internet

That is, run the CI server in private subnets, without any public IP, so that it's accessible only over a VPN connection. That way, only users with valid network access (e.g., via a VPN certificate) can access your CI server at all. Note that this does have a drawback: webhooks from external systems won't work. For example, GitHub won't automatically be able to trigger builds in your CI server; instead, you'll need to configure your CI server to poll your version control system for updates. This is a small price to pay for a significantly more secure CI server.

Enforce an approval workflow

Configure your CI/CD pipeline to require that every deployment be approved by at least one person (other than the person who requested the deployment in the first place). During this approval step, the reviewer should be able to see both the code changes and the `plan` output, as one final check that things look OK before `apply` runs. This ensures that every deployment, code change, and `plan` output has had at least two sets of eyes on it.

Don't give the CI server permanent credentials

As you saw in Chapter 6, instead of manually managed, permanent credentials (e.g., AWS access keys copy/pasted into your CI server), you should prefer to use authentication mechanisms that use temporary credentials, such as IAM roles and OIDC.

Don't give the CI server admin credentials

Instead, isolate the admin credentials to a totally separate, isolated *worker*: e.g., a separate server, a separate container, etc. That worker should be extremely locked down, so no developers have access to it at all, and the only thing it allows is for the CI server to trigger that worker via an extremely limited remote API. For example, that worker's API may only allow you to run specific commands (e.g., `terraform plan` and `terraform apply`), in specific repos (e.g., your live repo), in specific branches (e.g., the `main` branch), and so on. This way, even if an attacker gets access to your CI server, they still won't have access to the admin credentials, and all they can do is request a deployment on some code that's already in your version control system, which isn't nearly as much of a catastrophe as leaking the admin credentials fully.[6]

6 Check out Gruntwork Pipelines for a real-world example of this worker pattern.

Promote artifacts across environments

Just as with application artifacts, you'll want to promote your immutable, versioned infrastructure artifacts from environment to environment: for example, promote `v0.0.6` from dev to stage to prod.[7] The rule here is also simple:

> Always test Terraform changes in pre-prod before prod.

Because 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 must 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 approval step, as mentioned in the previous section, where you run `terraform plan` and have someone manually review the output and approve the deployment. 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, for instance, `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; for example, 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.

7 Credit for how to promote Terraform code across environments goes to Kief Morris: Using Pipelines to Manage Environments with Infrastructure as Code.

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 10-4.

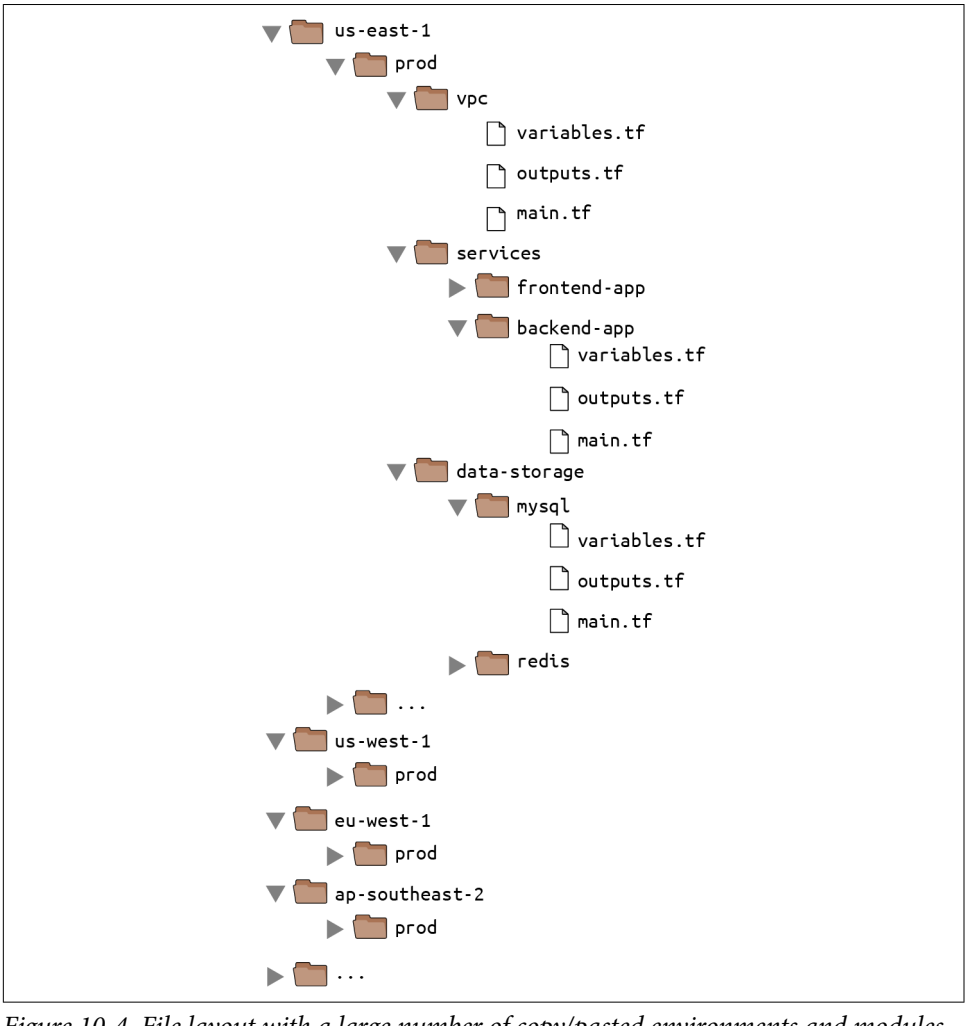


Figure 10-4. File layout with a large number of copy/pasted environments and modules within each environment.

This *live* repo has a large number of regions, and within each region, a large number of modules, most of which are copied and pasted. Sure, each module has a *main.tf* that references a module in your *modules* repo, so it's not as much copying and pasting 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:

- The `provider` configuration
- The `backend` configuration
- The input variables to pass to the module
- The output variables to proxy from the module

This can add up to dozens or hundreds of lines of mostly identical code in each module, copied and 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 you run all of the 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. In particular, Terragrunt allows you to define all of your Terraform code exactly once in the *modules* repo, whereas 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 10-5.

To get started, install Terragrunt by following the instructions on the Terragrunt website. Next, 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"
}
```

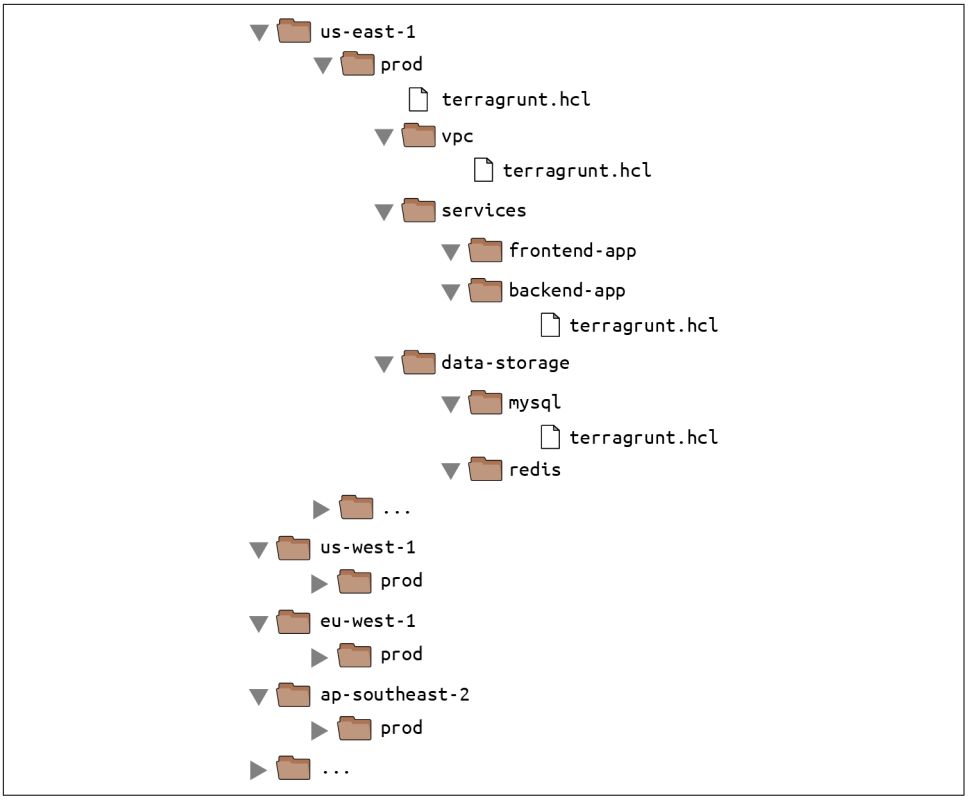


Figure 10-5. Use Terragrunt in your live repos to reduce the amount of code duplication.

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 copied and 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"

  # Set the username using the TF_VAR_db_username environment variable
```

```
  # Set the password using the TF_VAR_db_password environment variable
}
```

As you can see, *terragrunt.hcl* files use the same HashiCorp Configuration Language (HCL) syntax as Terraform itself. When you run `terragrunt apply` and it finds the `source` parameter in a *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 (with a `ref` parameter, as in the preceding example), 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 contains only 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`, and so on 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"

  generate = {
    path      = "backend.tf"
    if_exists = "overwrite"
  }

  config = {
    bucket         = "<YOUR BUCKET>"
    key            = "${path_relative_to_include()}/terraform.tfstate"
    region         = "us-east-2"
    encrypt        = true
    dynamodb_table = "<YOUR_TABLE>"
  }
}
```

From this one `remote_state` block, Terragrunt can generate the `backend` configuration dynamically for each of your modules, writing the configuration in `config` to the file specified via the `generate` param. Note that the `key` value in `config` uses a Terragrunt built-in function called `path_relative_to_include()`, which 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*, 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"

  # Set the username using the TF_VAR_db_username environment variable
  # 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-log-level debug
DEBU[0001] Reading Terragrunt config file at terragrunt.hcl
DEBU[0001] Included config live/stage/terragrunt.hcl
DEBU[0001] Downloading Terraform configurations into .terragrunt-cache
DEBU[0001] Generated file backend.tf
DEBU[0013] Running command: terraform init

(...)

Initializing the backend...

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

(...)

DEBU[0024] Running command: terraform apply

(...)

Terraform will perform the following actions:

(...)
```

```
Plan: 5 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: yes

(...)

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

Normally, Terragrunt only shows the log output from Terraform itself, but as I included `--terragrunt-log-level debug`, the preceding output shows what Terragrunt does under the hood:

1. Read the *terragrunt.hcl* file in the *mysql* folder where you ran `apply`.
2. Pull in all the settings from the included root *terragrunt.hcl* file.
3. Download the Terraform code specified in the `source` URL into the *.terragrunt-cache* scratch folder.
4. Generate a *backend.tf* file with your `backend` configuration.
5. Detect that `init` has not been run and run it automatically (Terragrunt will even create your S3 bucket and DynamoDB table automatically if they don't already exist).
6. Run `apply` to deploy changes.

Not bad for a couple of tiny *terragrunt.hcl* files!

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()
}

dependency "mysql" {
  config_path = "../../data-stores/mysql"
}

inputs = {
  environment = "stage"
  ami         = "ami-0fb653ca2d3203ac1"
```

```
  min_size = 2
  max_size = 2

  enable_autoscaling = false

  mysql_config = dependency.mysql.outputs
}
```

This *terragrunt.hcl* file uses the `source` URL and `inputs` just as you saw before and 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. The one new thing in this *terragrunt.hcl* file is the `dependency` block:

```
dependency "mysql" {
  config_path = "../../data-stores/mysql"
}
```

This is a Terragrunt feature that can be used to automatically read the output variables of another Terragrunt module, so you can pass them as input variables to the current module, as follows:

```
  mysql_config = dependency.mysql.outputs
```

In other words, `dependency` blocks are an alternative to using `terraform_remote_state` data sources to pass data between modules. While `terraform_remote_state` data sources have the advantage of being native to Terraform, the drawback is that they make your modules more tightly coupled together, as each module needs to know how other modules store state. Using Terragrunt `dependency` blocks allows your modules to expose generic inputs like `mysql_config` and `vpc_id`, instead of using data sources, which makes the modules less tightly coupled and easier to test and reuse.

Once you've got `hello-world-app` working 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 10-1 shows an overview of the two workflows side by side.

Table 10-1. 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
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 • Once a pull request is merged, deploy automatically	• Deploy with Terraform, Atlantis, Terraform Cloud, Terraform Enterprise, Terragrunt, scripts • Limited deployment strategies (make sure to handle errors: retries, `errored.tfstate`!) • Run deployment on a CI server • Give CI server temporary credentials solely to invoke a separate, locked-down worker that has admin permissions • Promote immutable, versioned artifacts across environments • Once a pull request is merged, go through an approval workflow where someone checks the `plan` output one last time, and then deploy automatically

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 10-6.

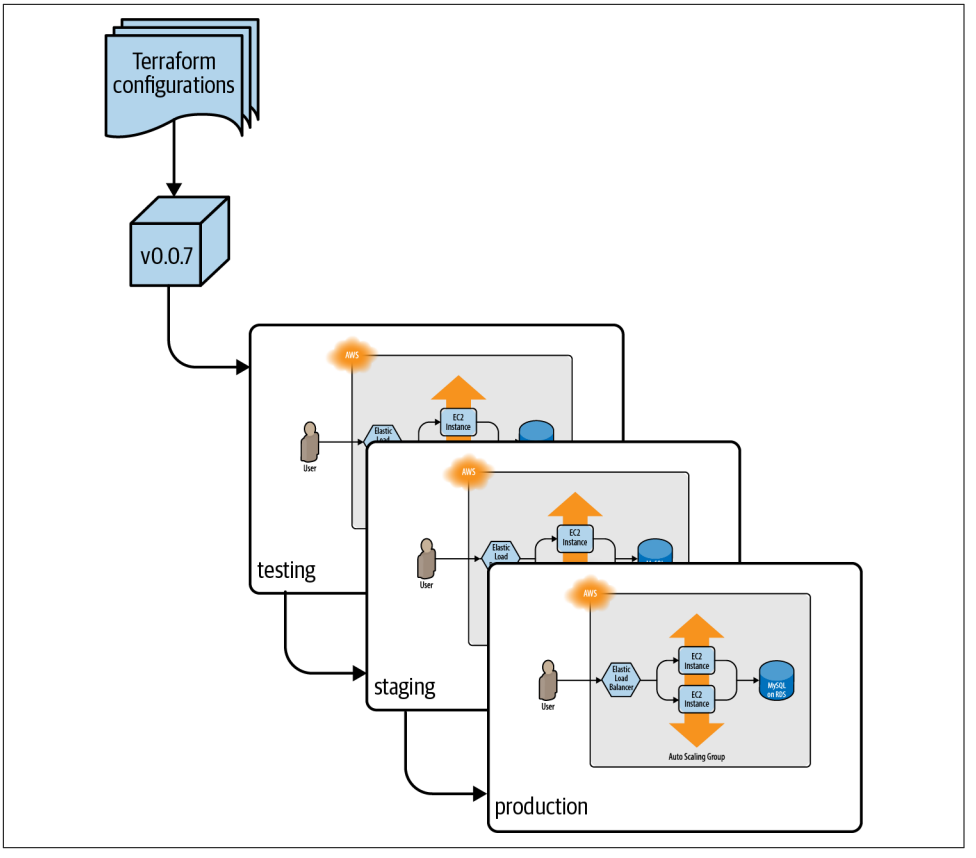


Figure 10-6. Promoting an immutable, versioned artifact of Terraform code from environment to environment.

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 manage secrets; how to work with multiple regions, accounts, and clouds; 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, scheduled actions, CloudWatch alarms, IAM users, reusable modules, zero-downtime deployment, AWS Secrets Manager, Kubernetes clusters, 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, IaC, 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, IaC, and DevOps, head over to Appendix 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!

APPENDIX

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: Dynamic Systems for the Cloud Age* 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, and 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 Kevin Behr, Gene Kim, and George Spafford (Information Technology Process Institute)
- *Effective DevOps* by Jennifer Davis and Ryn 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
- AWS News 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
- "Automated Testing for Terraform, Docker, Packer, Kubernetes, and More" 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
- "Real Software Engineering" by Glenn Vanderburg

Newsletters

- DevOps Weekly
- Gruntwork Newsletter
- Terraform: Up & Running Newsletter
- Terraform Weekly Newsletter

Online Forums

- Terraform subforum of HashiCorp Discuss
- Terraform subreddit
- DevOps subreddit

Index

A

Access Key IDs (in AWS), 42, 44, 196-199, 200-201
accessing secrets, interfaces for, 195
    (see also secrets management)
accidental complexity, 278
accounts (in AWS)
    configuring multiple, 238-245
    setup, 40-42
ad hoc scripts, 4-5
agent versus agentless software, 28-29
ALB (Application Load Balancer), 71-78, 283-285
aliases, 227
    configuration aliases, 247-248
    for Gmail, 241
    KMS CMK aliases, 210
    when to use, 236-237, 245
Amazon S3 (Simple Storage Service), as remote backend, 85-91
Amazon Web Services (see AWS)
AMI (Amazon Machine Image)
    ami parameter, 46
    Packer template, 10-11
    zero-downtime deployment, 169-179
AMI IDs, managing, 229-230
ami parameter, 14, 46
Ansible, comparison with other IaC tools, 20-36
application code workflow
    automated tests, 385
    changing code, 383-384
    comparison with infrastructure code workflow, 412
    deployment, 387-390
    locally run code, 383
    merge and release, 386
    reviewing code changes, 384
    steps in, 382
    version control, 382
Application Load Balancer (ALB), 71-78, 283-285
arguments for resources, 45
arrays
    lookup syntax, 144
    of resources, 145
ASGs (Auto Scaling Groups), 66-70
    instance refresh, 181-184
    scheduled actions, 127-129
    zero-downtime deployment, 171-179
assume role policies, 201
Atlantis, 399-401
auditing multiple AWS accounts, 239
authentication (see secrets management)
auto healing, 12
auto scaling, 12, 127-129, 160-162
automated tests, 398
    in application code workflow, 385
    end-to-end tests, 362-364
    in infrastructure code workflow, 399-400
    integration tests
        with multiple modules, 349-354
        retries in, 360-361
        Ruby comparison, 348-349
        stages of, 354-360
    plan testing, 366-370
    purpose of, 320
    server testing, 370-372

static analysis, 364-366
types of, 320-321
unit tests
dependency injection, 332-343
Ruby comparison, 321-324
running in parallel, 343-347
with Terratest, 324-332
when to use, 297
Availability Zones (AZs), 45, 179-180
AWS (Amazon Web Services)
accounts
configuring multiple, 238-245
setup, 40-42
AZs (Availability Zones), 45, 179-180
benefits of, 39
configuring as provider, 45
regions, 45, 226-237
aws eks update-kubeconfig command, 270
AWS Provider, xx, 222
AZs (Availability Zones), 45, 179-180

B

backends
configuring with Terragrunt, 409-411
local backends, 84
remote backends
limitations of, 91-93
shared storage with, 84-91
Bash scripts, 305
as ad hoc scripts, 4-5
externalizing, 111-113
best practices for testing, 373
blocking public access to S3 buckets, 87
blue-green deployment, 388
branches, reasons to avoid, 393-395
bucket parameter, 86, 88
built-in functions, explained, 111-113
bus factor, 16

C

canary deployment, 389
centralized secret stores, 194
changing code
in application code workflow, 383-384
in infrastructure code workflow, 396
checklist for production-grade infrastructure, 279-280
Chef, comparison with other IaC tools, 20-36
child accounts, creating, 239-241
CI servers
automated tests, 385
as deployment servers, 389, 403-404
machine user authentication, 199-206, 219
CI/CD workflow
for application code
automated tests, 385
changing code, 383-384
comparison with infrastructure code workflow, 412
deployment, 387-390
locally run code, 383
merge and release, 386
reviewing code changes, 384
steps in, 382
version control, 382
CircleCI, 199-201
EC2 Instances with IAM roles, 201-204
GitHub Actions, 204-206
for infrastructure code
automated tests, 399-400
changing code, 396
comparison with application code workflow, 412
deployment, 401-412
locally run code, 395
merge and release, 400-401
reviewing code changes, 397-399
steps in, 390
version control, 391-395
CIDR blocks, 54, 366
CircleCI, 199-201, 385
CircleCI Context, 200
CLB (Classic Load Balancer), 71
cleaning up manual tests, 319-320
CLI (command-line interface), 195
cloud providers (see providers)
CloudFormation, comparison with other IaC tools, 20-36
cluster of web servers
deploying, 66-70
refactoring, 283-285
clusters (Kubernetes), 12
deploying Docker containers, 264-272
inspecting, 262-264
CMK (Customer Managed Key), 209
code examples
fair use permissions, xx-xxi
location of, xix-xx

versions used for, xx
coding guidelines, 397-399
collaboration using Git repositories, 51
combining IaC tools, 34-36
command history, avoiding writing to, 197
command-line interface (CLI), 195
comments, as documentation, 398
community size in IaC tool comparison, 31-33
comparison of IaC tools, 20-36
agent versus agentless, 28-29
combining tools, 34-36
configuration management versus provisioning, 21
general-purpose versus domain-specific language, 25-26
large versus small community, 31-33
master versus masterless, 26-27
mature versus cutting edge, 33
mutable versus immutable infrastructure, 21-22
paid versus free, 30-31
procedural versus declarative language, 22-25
composable modules for production-grade infrastructure, 285-291
concat built-in function, 165
conditionals
count parameter
if-else-statements with, 163-165
if-statements with, 160-162
for_each and for expressions, 165-167
if string directive, 167-168
ternary syntax, 99, 161, 165
types and purpose of, 160
configurable web servers, deploying, 60-66
configuration aliases, 247-248
configuration drift, 2
configuration files
explained, 18
naming conventions, 102-103
configuration management tools, 5-7
combining with provisioning, 34
provisioning tools versus, 21
container engines, 9
containers, 9
deploying with EKS (Elastic Kubernetes Service), 264-272
explained, 250-252
Pods, 13, 257-258
control plane (Kubernetes), 253, 266
cost in IaC tool comparison, 30-31
count parameter
conditionals with
if-else-statements, 163-165
if-statements, 160-162
limitations, 147-149, 179-180
loops with, 142-149
create_before_destroy lifecycle setting, 68, 141, 171, 181, 188
creation-time provisioners, 309
credentials (see secrets)
Customer Managed Key (CMK), 209
customer secrets, 193

D

data sources
external, 311-312
replicating in multiple AWS regions, 230-236
secrets management, 206
with encrypted files, 209-213
with environment variables, 207-209
with secret stores, 213-217
for subnets, 69-70
terraform_remote_state, 105-113
declarative language
procedural versus, 22-25
Terraform as, 141
declaring variables, 60-66
default parameter, 60
Default VPC (in AWS), 42, 59, 69
default_tags block, 155
defining listeners, 72
dependencies
implicit, 55
in Terragrunt, 412
types of, 298
versioning pinning, 298-303
dependency injection, 332-343
depends_on parameter, 65
deploying
application code, 387-390
Docker with EKS (Elastic Kubernetes Service), 264-272
infrastructure code, 401-412
load balancers, 70-78
modules in multiple AWS regions, 230-236
resources in multiple AWS regions, 228-230

servers
cluster of web servers, 66-70
configurable web servers, 60-66
single servers, 44-52
web servers, 52-59
with zero-downtime deployment, 169-179
deployment servers, 389, 403-404
deployment strategies, 387-389, 402-403
deployment tools, 387, 401
Deployments (Kubernetes), 13, 254-259
description parameter, 60, 64
destroy-time provisioners, 309
DevOps, 1-3
project time estimates, 277-279
resources for information, 417-419
Docker
containers (see containers)
deploying with EKS (Elastic Kubernetes Service), 264-272
explained, 249-252
installing, 249
purpose of, 11
Docker Hub, 250
docker kill command, 263
docker ps command, 262
Docker Registry, 250
docker rm command, 252
docker run command, 249
docker start command, 251
documentation
IaC as, 16
for Terraform, xix, 46
types of, 397-398
DRY (Don't Repeat Yourself) principle, 60, 93, 407
DSL (domain-specific language) versus GPL (general-purpose programming language), 25-26
DynamoDB tables
limitations of remote backends, 91-93
locking with, 87
dynamodb_table parameter, 89

E

EC2 Instances, 45, 49
(see also servers)
deploying in multiple AWS regions, 228-230
IAM roles for, 201-204
EKS (Elastic Kubernetes Service), 264-272
ELB (Elastic Load Balancer) service, 70
enabling
server-side encryption, 87
versioning in Amazon S3, 86
encrypt parameter, 89
encryption
secrets management with, 209-213
server-side, enabling, 87
end-to-end tests, 321, 362-364
enforcing tagging standards, 155
environment variables, secrets management with, 107, 197-199, 207-209
environments
folders for, 101-102
promotion across, 390, 405-412
sandbox, 319-320, 330, 396
errors in deployment, 402-403
essential complexity, 278
example code, as documentation, 398
examples
fair use permissions, xx-xxi
location of, xix-xx
versions used for, xx
executing scripts
for external data source, 311-312
with provisioners, 306-310
expressions, 55
external data source, 311-312
externalizing Bash scripts, 111-113

F

fair use permissions for code examples, xx-xxi
false incrementalism, 379
feature toggles, 389
file layout
conventions for, 398
isolating state files, 100-105
file paths
local, 138
path references for, 129-130
file-based secret stores, 194
folders
for environments, 101-102
running commands across multiple, 104
for expressions
conditionals with, 165-167
loops with, 156-158
for string directive, loops with, 158-159
format built-in function, 111

for_each expressions
conditionals with, 165-167
limitations, 179-180
loops with, 149-154
function composition, 285-291

G

get deployments command, 262, 271
get nodes command, 254, 270
get pods command, 262, 271
get services command, 263, 271
git commit command, 52
git push command, 52
Git repositories
collaborating with teammates, 51
SSH authentication for, 137
GitHub Actions, 204-206
GitHub, pull requests, 384
.gitignore file, 51
Gmail aliases, 241
Go, installing, 325
Golden Rule of Terraform, 392-393
GPL (general-purpose programming language) versus DSL (domain-specific language), 25-26

H

HCL (HashiCorp Configuration Language), 44
health checks, 74
heredoc syntax, 54, 167
history files, avoiding writing to, 197
Hofstadter's Law, 277
human users, secrets management and, 197-199
hypervisors, 8

I

IaC (infrastructure as code), 275
(see also production-grade infrastructure)
ad hoc scripts, 4-5
benefits of, 16-17
configuration management tools, 5-7
importing into state files, 185
orchestration tools, 12-14
out of band resources, 185
provisioning tools, 14
purpose of, 3
refactoring, 186-189
resources for information, 417-419
server templating tools, 7-11
team adoption of
incrementalism in, 379-380
learning curve, 380-381
return on investment, 376-379
tool comparison, 20-36
agent versus agentless, 28-29
combining tools, 34-36
configuration management versus provisioning, 21
general-purpose versus domain-specific language, 25-26
large versus small community, 31-33
master versus masterless, 26-27
mature versus cutting edge, 33
mutable versus immutable infrastructure, 21-22
paid versus free, 30-31
procedural versus declarative language, 22-25
validating, 319
IAM (Identity and Access Management) service, 40
IAM OIDC identity providers, 204
IAM Policies, 41
IAM roles
child account authentication, 243
cross-account authentication, 244
for EC2 Instances, 201-204
IAM users
changing policies, 163-165
creating, 40, 142-152
errors, 184-186
idempotence, 6, 47, 89
if string directive, conditionals with, 167-168
if-else-statements with count parameter, 163-165
if-statements with count parameter, 160-162
images (of servers), 7-11
immutable infrastructure
explained, 11
mutable versus, 21-22
promotion across environments, 390, 405
releasing application code, 386
workflows, 412-413
implicit dependencies, 55
importing infrastructure, 185
incrementalism in IaC (infrastructure as code) adoption, 379-380

infrastructure as code (see IaC)
infrastructure code workflow
automated tests, 399-400
changing code, 396
comparison with application code workflow, 412
deployment, 401-412
locally run code, 395
merge and release, 400-401
reviewing code changes, 397-399
steps in, 390
version control, 391-395
infrastructure secrets, 194
inline blocks
creating multiple with for_each expressions, 152-154
separate resources versus, 130-132
input variables, 60-64
in composable modules, 286-291
in modules, 121-124
inspecting Kubernetes clusters, 262-264
installing
Docker, 249
Go, 325
kubectl, 254
providers, 223-225
Terraform, 43-44, 300
Terragrunt, 407
instance metadata endpoints, 203
instance profiles, 202
instance refresh, 181-184
instance_type parameter, 46
integration tests
purpose of, 321
retries in, 360-361
Ruby comparison, 348-349
stages of, 354-360
with multiple modules, 349-354
interfaces, accessing secrets, 195
interpolation, 64
isolating
multiple AWS accounts, 238
at network level, 133
state files, 83
purpose of, 93-94
via file layout, 100-105
via workspaces, 94-100

K

kernel, 9
kernel space, 9
key parameter, 88
key policies, 209
KMS (key management system), 194
kubectl, 254, 262-263
Kubernetes
clusters, 12
deploying Docker containers, 264-272
inspecting, 262-264
control plane, 253, 266
EKS (Elastic Kubernetes Service), 264-272
explained, 252-264
kubectl, 254, 262-263
objects in, 254
Pods, 13, 257-258
running, 253
worker nodes, 253, 266
Kubernetes Deployment, 13, 254-259
Kubernetes Service, 255, 259-261

L

large modules, limitations of, 281
launch configurations, 67
launch templates, 67
learning curve in IaC (infrastructure as code)
adoption, 380-381
length built-in function, 144
lifecycle settings, 68, 86, 171
list comprehensions (Python), 156
listener rules, creating, 75
listeners, defining, 72
literals, 55
live repository, 391-393, 406-412
load balancing, 12, 70-78
local backends, 84
local file paths, 138
local names, 224
local references, 126
local values in modules, 125-127
locally run code
in application code workflow, 383
in infrastructure code workflow, 395
lock files, 302
locking
with DynamoDB tables, 87
with remote backends, 84
state files, 83

version control disadvantages of, 84
lookup syntax for arrays, 144
loops
count parameter, 142-149
for expressions, 156-158
for string directive, 158-159
for_each expressions, 149-154
types and purpose of, 142

M

machine users, secrets management and, 199-206, 219
managed node groups, 266-268
Managed Policies (in AWS), 41
manual tests
cleanup, 319-320
explained, 317-319
Ruby comparison, 316-317
maps of resources, 150
master versus masterless servers, 26-27
maturity in IaC tool comparison, 33
merging code
in application code workflow, 386
in infrastructure code workflow, 400-401
mocks, 320
modules
authenticating multiple AWS accounts, 242-244
configuring for Kubernetes, 261-262
count parameter, 146-147
creating for Kubernetes, 255-261
creating reusable, 118-120
deploying in multiple AWS regions, 230-236
for_each expressions, 152
inline blocks versus separate resources, 130-132
input variables, 121-124
integration tests, 349-354
local values, 125-127
for multiple providers, 245-248
output variables, 127-129
path references, 129-130
for production-grade infrastructure
composable modules, 285-291
non-Terraform code in, 305-312
small modules, 281-285
testable modules, 291-298
versioned modules, 298-305
publishing, 304-305
purpose of, 92, 117
repository for, 391-392
self-validating, 293
precondition/postcondition blocks in, 295-297
validation blocks in, 294-295
tagging standards, 155
types of, 118, 245
unit tests for, 324-332
versioning, 133-138
versioning pinning, 302-303
multiple AWS accounts, configuring, 238-245
multiple AWS regions, configuring, 226-237
multiple folders, running commands across, 104
multiple inline blocks, creating with for_each expressions, 152-154
multiple modules, integration tests with, 349-354
multiple providers
Docker, explained, 249-252
EKS (Elastic Kubernetes Service), 264-272
Kubernetes, explained, 252-264
multiple AWS accounts, 238-245
multiple AWS regions, 226-237
multiple copies of different providers, 248-272
multiple copies of same provider, 226-248
reusable modules for, 245-248
mutable versus immutable infrastructure, 21-22

N

naming
configuration files, 102-103
resources, 49
network isolation, 133
network security
CIDR blocks, 54, 366
data sources for subnets, 69
Default VPC (in AWS), 42
network isolation, 133
port numbers and, 53
security groups, creating, 54, 73
VPC subnets, 59
NLB (Network Load Balancer), 71
non-Terraform code, running in modules, 305-312
null_resource, 310-311

O

objects (Kubernetes), 254
OIDC (Open ID Connect), 204-206
one built-in function, 165
OPA (Open Policy Agent), 368-370
OpenStack Heat, comparison with other IaC tools, 20-36
orchestration tools, 12-14
  combining with provisioning and server templating, 35-36
  for deployment, 387
  purpose of, 252
out of band resources, 185
output variables, 64-66
  in composable modules, 286-291
  in modules, 127-129

P

Packer, 7-11
parallel unit tests, running, 343-347
partial configurations, 92, 351-352
passing secrets via environment variables, 107, 197-199
path references in modules, 129-130
permissions for code examples, xx-xxi
personal secrets, 193
plain text, avoiding secrets in, 192-193
plan files, secrets management, 218-219
plan testing, 366-370, 399
plugins, providers as, 222-223
Pod Template, 257-258
Pods, 13, 257-258
policy-as-code tools, 368-370
port numbers, 53
postcondition blocks, 295-297
precondition blocks, 295-297
preferred local names, 224
prevent_destroy parameter, 86
private API, state files as, 83
private Git repositories, 137
private keys, 194
private subnets, 60
Private Terraform Registry, 305
procedural versus declarative language, 22-25
production-grade infrastructure
  checklist for, 279-280
  definition of, 275
  modules for
    composable modules, 285-291
    non-Terraform code in, 305-312
    small modules, 281-285
    testable modules, 291-298
    versioned modules, 298-305
  time estimates for, 275-279
promotion across environments, 390, 405-412
providers, 249
  (see also multiple providers)
  configuring, 45, 225-226
    multiple AWS accounts, 238-245
    multiple AWS regions, 226-237
    in reusable modules, 245-248
  installing, 223-225
  as plugins, 222-223
  secrets management, 196-197
    human users and, 197-199
    machine users and, 199-206, 219
  transparent portability, 19
  versioning pinning, 301-302
provisioners
  executing scripts, 306-310
  with null_resource, 310-311
  User Data scripts versus, 310
provisioning tools, 14
  combining with configuration management, 34
  combining with orchestration and server templating, 35-36
  combining with server templating, 35-35
  configuration management tools versus, 21
public access to S3 buckets, blocking, 87
public keys, 194
public subnets, 59
Public Terraform Registry, 304-305
publishing modules, 304-305
pull requests, 384
Pulumi, comparison with other IaC tools, 20-36
Puppet, comparison with other IaC tools, 20-36

R

random integers, generating, 180
RDS (Relational Database Service), 105
README files, 398
reconciliation loops, 254
refactoring
  Terraform code, 186-189
  web server clusters, 283-285
references, 55
region parameter, 89

regions (in AWS), 45, 226-237
Rego, 368-370
Relational Database Service (RDS), 105
relative filepaths, 129
releasing code
    in application code workflow, 386
    in infrastructure code workflow, 400-401
remote backends
    limitations of, 91-93
    shared storage with, 84-91
remote procedure calls (RPCs), 222
removing resources, 79-80
replicas, 13
replicating data sources in multiple AWS regions, 230-236
repositories
    sharing, 51
    SSH authentication for, 137
    for version control, 391-392
required_providers block, 224-225
required_version parameter, 298
resource attribute references, 55
resource dependencies, 105
resources
    arrays of, 145
    creating, 45
    creating out of band, 185
    deploying in multiple AWS regions, 228-230
    lifecycle settings, 68
    maps of, 150
    naming, 49
    null_resource, 310-311
    removing, 79-80
    secrets management, 206
        with encrypted files, 209-213
        with environment variables, 207-209
        with secret stores, 213-217
    separate resources versus inline blocks, 130-132
    tagging standards, 155
resources for information, 417-419
retries
    in integration tests, 360-361
    with known errors, 402
return on investment for IaC (infrastructure as code), 376-379
reusability of code, 17
reusable modules, 245
    creating, 118-120
    for multiple providers, 245-248
reviewing code changes
    in application code workflow, 384
    for infrastructure code, 397-399
rolling deployments, 7, 388-388
root accounts, 239
root modules, 118, 245
root users, 40
    port numbers and, 53
RPCs (remote procedure calls), 222
running
    Kubernetes, 253
    parallel unit tests, 343-347

S

S3 buckets
    creating, 85-86
    limitations of remote backends, 91-93
    public access, blocking, 87
    server-side encryption, enabling, 87
    versioning, enabling, 86
sandbox environments, 319-320, 330, 396
scheduled actions, 127, 160
scripts
    for deployment, 387, 401
    executing
        for external data source, 311-312
        with provisioners, 306-310
Secret Access Keys (in AWS), 42, 44
secret stores, secrets management with, 213-217
secrets
    access interface for, 195
    avoiding in plain text, 192-193
    passing via environment variables, 107
    with remote backends, 85
    storage methods, 194-195
    types of, 193-194
    version control disadvantages, 84
secrets management
    CI/CD workflow
        CircleCI, 199-201
        EC2 Instances with IAM roles, 201-204
        GitHub Actions, 204-206
    multiple AWS accounts, 238, 242-244
    for plan files, 218-219
    for providers, 196-197
        human users and, 197-199
        machine users and, 199-206, 219

for resources and data sources, 206
with encrypted files, 209-213
with environment variables, 207-209
with secret stores, 213-217
rules of, 192-193
for state files, 217-218
Terraform authentication options, 44
tool comparison, 195-196
security (see network security; secrets management)
security groups, creating, 54, 73
selector block, 259
self-validating modules, 293
precondition/postcondition blocks in, 295-297
validation blocks in, 294-295
semantic versioning, 136
sensitive parameter, 61, 65
separate resources versus inline blocks, 130-132
server templating tools, 7-11
combining with orchestration and provisioning, 35-36
combining with provisioning, 35
server testing, 370-372
server-side encryption, enabling, 87
servers, deploying
cluster of web servers, 66-70
configurable web servers, 60-66
single servers, 44-52
web servers, 52-59
service discovery, 12
shared storage for state files, 83-91
with remote backends, 84-91
version control disadvantages, 83-84
sharing Git repositories with teammates, 51
small modules for production-grade infrastructure, 281-285
snowflake servers, 2
software delivery, 1
sops, 212
splat expression, 145
Spolsky, Joel, 382
SSH authentication for Git repositories, 137
stages of integration tests, 354-360
state files
explained, 82-83
importing infrastructure, 185
isolating, 83
purpose of, 93-94
via file layout, 100-105
via workspaces, 94-100
locking, 83
as private API, 83
secrets management, 217-218
shared storage for, 83-91
with remote backends, 84-91
version control disadvantages, 83-84
terraform_remote_state data source, 105-113
static analysis, 364-366
storing secrets, 194-195
string directives, 158
strip markers, 168
structural types, 62
style guidelines, 399
subnets
data sources for, 69-70
of VPCs, 59
subnets parameter, 72
subnet_ids parameter, 69

T

tagging standards, enforcing, 155
target groups, creating, 74
TDD (Test-Driven Development), 293
team adoption of IaC
incrementalism in, 379-380
learning curve, 380-381
return on investment, 376-379
templatefile built-in function, 111-113, 129
ternary syntax, 99, 161, 165
Terraform
authentication options, 44
changes, 2017-2019, xvi-xviii
comparison with other IaC tools, 20-36
agent versus agentless, 28-29
combining tools, 34-36
configuration management versus provisioning, 21
general-purpose versus domain-specific language, 25-26
large versus small community, 31-33
master versus masterless, 26-27
mature versus cutting edge, 33
mutable versus immutable infrastructure, 21-22
paid versus free, 30-31

procedural versus declarative language, 22-25
core, 222, 298-301
as declarative language, 141
as deployment tool, 387
documentation, xix, 46
Golden Rule of Terraform, 392-393
installing, 43-44, 300
new features, xiv-xvi
operational overview, 17-19
purpose of, ix
refactoring, 186-189
transparent portability, 19
version used, xx
terraform apply command, 48
Terraform Cloud, 401
Terraform configurations, 18
terraform console command, 111
terraform destroy command, 79
Terraform Enterprise, 401
terraform fmt command, 399
terraform graph command, 56
terraform import command, 185
terraform init command, 47, 89
terraform output command, 65
terraform plan command, 47, 184-186, 366-370, 399
Terraform state files (see state files)
terraform validate command, 365
terraform version command, 301
terraform workspace list command, 97
terraform workspace new command, 96
terraform workspace select command, 97
terraform workspace show command, 95
terraform_remote_state data source, 105-113
Terragrunt, 93, 401
installing, 407
promotion across environments, 407-412
tgswitch, 301
Terratest, 248
integration tests with, 355-360
plan tests with, 367-368
retries in integration tests, 360-361
unit tests with, 324-332
versions of, 330
test doubles, 320
test pyramid, 362
Test-Driven Development (TDD), 293
testable modules for production-grade infrastructure, 291-298
testing
automated tests, 398
in application code workflow, 385
end-to-end tests, 362-364
in infrastructure code workflow, 399-400
integration tests, 348-361
plan testing, 366-370
purpose of, 320
server testing, 370-372
static analysis, 364-366
types of, 320-321
unit tests, 321-347
when to use, 297
best practices, 373
comparison of approaches, 373
manual tests
cleanup, 319-320
explained, 317-319
Ruby comparison, 316-317
tfenv, 299-301
tgswitch, 301
time estimates for production-grade infrastructure, 275-279
transparent portability, 19
type constraints, 61
type parameter, 61

U

Ubuntu, 249
UI (user interface), 195
unit tests
dependency injection, 332-343
purpose of, 320
Ruby comparison, 321-324
running in parallel, 343-347
with Terratest, 324-332
units, defined, 320
User Data scripts versus provisioners, 310
user interface (UI), 195
user space, 9
user_data parameter, 14, 53
user_data_replace_on_change parameter, 54

V

Vagrant, purpose of, 11
validating
code, 17

infrastructure, 319
validation blocks, 294-295
validation parameter, 61
values built-in function, 151
variable references, 64
variables, 121
(see also input variables; output variables)
declaring, 60-66
version control, 51
in application code workflow, 382
disadvantages of, 83-84
importance of, 17
in infrastructure code workflow, 391-395
secrets in, 192
versioning
in Amazon S3, 85, 86
modules, 133-138
for production-grade infrastructure, 298-305
versioning pinning, 298-303
VM images, 8
VMs (virtual machines), 8
VPCs (virtual private clouds)
Default VPC (in AWS), 42
network isolation, 133
subnets, 59

W

web servers
deploying, 52-59
cluster of web servers, 66-70
configurable web servers, 60-66
refactoring, 283-285
worker nodes (Kubernetes), 253, 266
workflows (see CI/CD workflow)
workspaces
isolating state files, 94-100
limitations of, 99

Y

yak shaving, 277-278

Z

zero-downtime deployment, 169-179
limitations, 181-184

About the Author

Yevgeniy (Jim) Brikman loves programming, writing, speaking, traveling, and lifting heavy things. He is the cofounder of Gruntwork, a company that provides DevOps as a Service. He's also the author of another book published by O'Reilly Media called *Hello, Startup: A Programmer's Guide to Building Products, Technologies, and Teams*. Previously, he worked as a software engineer at LinkedIn, TripAdvisor, Cisco Systems, and Thomson Financial and got his BS and master's degrees at Cornell University. For more info, check out *ybrikman.com*.

Colophon

The animal on the cover of *Terraform: Up and Running* is a flying dragon lizard (*Draco volans*), a small reptile so named for its ability to glide using winglike flaps of skin known as *patagia*. The patagia are brightly colored and allow the animal to glide for up to eight meters. The flying dragon lizard is commonly found in many Southeast Asian countries, including Indonesia, Vietnam, Thailand, the Philippines, and Singapore.

Flying dragon lizards feed on insects and can grow to more than 20 centimeters in length. They live primarily in forested regions, gliding from tree to tree to find prey and avoid predators. Females descend from the trees only to lay their eggs in hidden holes in the ground. The males are highly territorial and will chase rivals from tree to tree.

Although once thought to be poisonous, flying dragon lizards pose no threat to humans and are sometimes kept as pets. They are not currently threatened or endangered. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from *Johnson's Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY®

Learn from experts.
Become one yourself.

Books | Live online courses
Instant Answers | Virtual events
Videos | Interactive learning

Get started at oreilly.com.

©2022 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. | 175