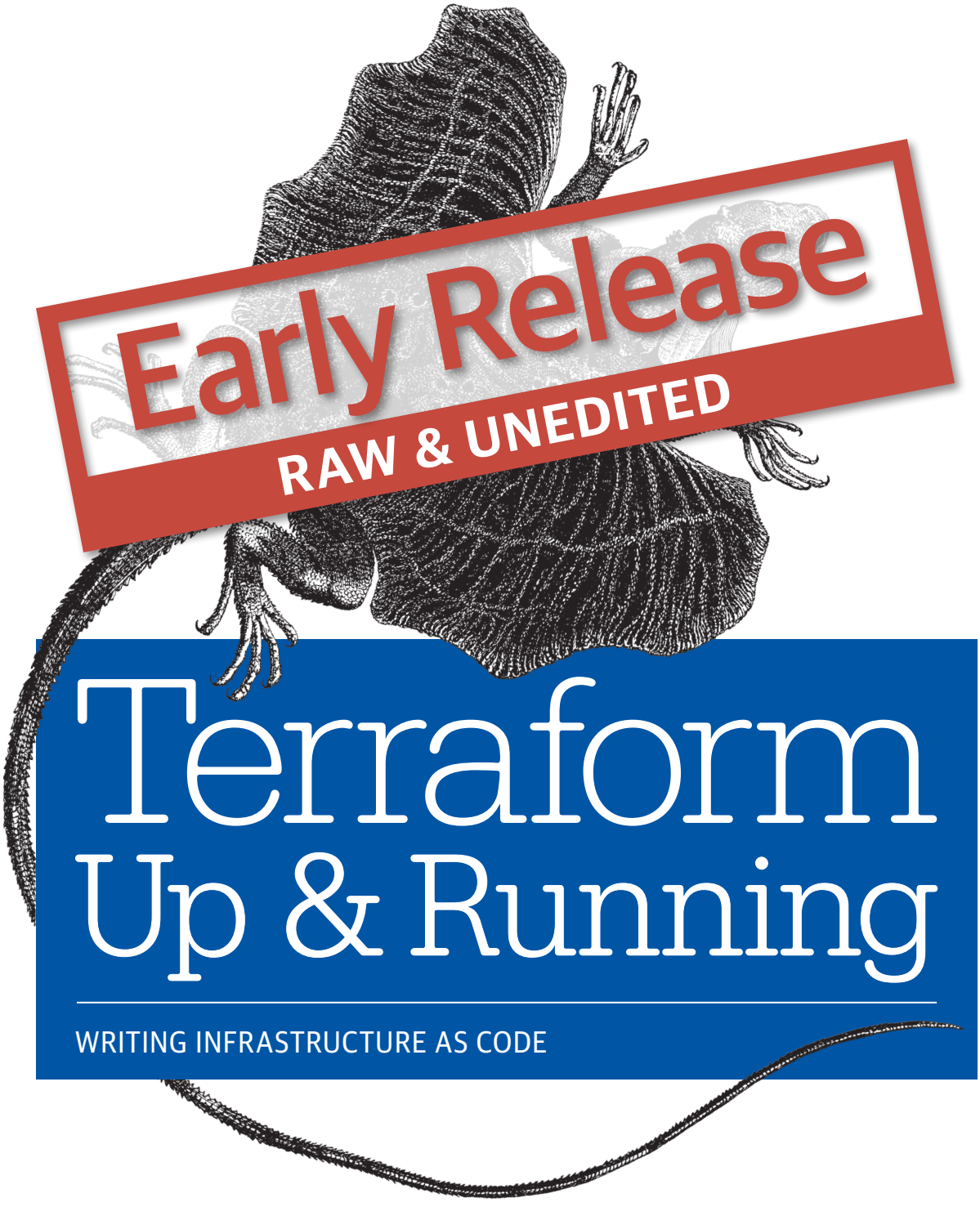
O'REILLY®



Early Release

RAW & UNEDITED

# Terraform Up & Running

WRITING INFRASTRUCTURE AS CODE

Yevgeniy Brikman

1ST EDITION

# Terraform

## *Up and Running*

*Yevgeniy Brikman*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

**Terraform: Up and Running**
by Yevgeniy Brikman



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://safaribooksonline.com* ). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

**Editors:** Brian Anderson and Virginia Wilson
**Production Editor:** FILL IN PRODUCTION EDITOR
**Copyeditor:** FILL IN COPYEDITOR
**Proofreader:** FILL IN PROOFREADER
**Indexer:** FILL IN INDEXER
**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery
**Illustrator:** Rebecca Demarest

January -4712: First Edition

**Revision History for the First Edition**
2016-11-08: First Early Release
2016-12-01: Second Early Release
2016-12-08: Third Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491977033* for release details.



978-1-491-97703-3

[FILL IN]

*To Mom, Dad, Lyalya, and Molly*

# Table of Contents

# Preface

A long time ago, in a data center far, far away, an ancient group of powerful beings known as sysadmins used to deploy infrastructure manually. Every server, every route table entry, every database configuration, and every load balancer 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 Rebel Alliance, there is now a better way to do things: Terraform.

Terraform is an open source tool that allows you to define your infrastructure as code using a simple, declarative programming language, and to deploy and manage that infrastructure across a variety of cloud providers (including Amazon Web Services, Azure, Google Cloud, DigitalOcean, and many others) using a few commands. For example, instead of manually clicking around a webpage or running dozens of commands, here is all the code it takes to configure a server:

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

And to deploy it, you just run one command:

```
> terraform apply
```

Thanks to its simplicity and power, Terraform is rapidly emerging as a dominant player in the DevOps world. It's replacing not only manual sysadmin work, but also many older infrastructure as code tools such as Chef, Puppet, Ansible, SaltStack, and CloudFormation.

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 above!) all the way up to running a full tech stack (server cluster, load

balancer, database) capable of supporting a large amount of traffic and a large team of developers—all in the span of just a few chapters. This is a hands-on-tutorial that not only teaches you DevOps and infrastructure as code principles, but also walks you through dozens of code examples that you can try at home, so make sure you have your computer handy.

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

## Who should read this book

This book is for Sysadmins, Operations Engineers, Release Engineers, Site Reliability Engineers, DevOps Engineers, Infrastructure Developers, Full Stack Developers, Engineering Managers, CTOs, and anyone else responsible for the code *after* it has been written. 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 3AM, then this book is for you.

Collectively, all of these tasks are usually referred to as "operations." While there are many developers who know how to write code, far fewer understand operations. You could get away with that in the past, but in the modern world, as cloud computing and the DevOps movement become ubiquitous, just about every developer will need to add operational skills to their toolbelt.

You don't need to already be an operations expert to read this book—a basic familiarity with programming, the command line, and server-based software (e.g. websites) should suffice—but you should read this book if you want to *become* an expert in one of the most critical aspects of modern operations: managing infrastructure as code. In fact, you'll learn not only how to manage infrastructure as code using Terraform, but also how this fits into the overall DevOps world. Here are some of the questions you'll be able to answer by the end of the book:

- Why use infrastructure as code at all?
- When should you use configuration management tools like Chef, Ansible, and Puppet?
- When should you use orchestration tools like Terraform, CloudFormation, and OpenStack Heat?
- When should you use server templating tools such as Docker and Packer?
- How does Terraform work and how do I use it to manage my infrastructure?
- How do you make Terraform a part of your automated deployment process?
- How do you make Terraform a part of your automated testing 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 with 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: age.

At the time of writing, Terraform is barely 2 years old. As a result, it's hard to find books, blog posts, or experts to help you master the tool. If you try to learn Terraform from the official documentation, you'll find that it does a good job of introducing the basic syntax and features, but it includes almost no information on idiomatic patterns, best-practices, testing, reusability, or team workflows. It's like trying to become fluent in French by studying only the vocabulary and not any of the grammar or idioms.

The reason I wrote this book is to help developers become fluent in Terraform. I've been using Terraform for more than half of its life, much of it in a professional context at my company Gruntwork, and I've spent many of those months figuring out what works and what doesn't primarily through trial-and-error. My goal is to share what I've learned so you can avoid that lengthy process and become fluent in a matter of hours.

Of course, you can't become fluent just by reading. To become fluent in French, you'll have to spend time talking with native French speakers, watching French TV shows, and listening to French music. To become fluent in Terraform, you'll have to write real Terraform code, use it to manage real software, and deploy that software on real servers (don't worry, all the examples use Amazon Web Service's free tier, so it shouldn't cost you anything). 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, orchestration, and server templating; the benefits of infrastructure as code; a comparison of Terraform, Chef, Puppet, Ansible, Salt Stack, and CloudFormation.

*Chapter 2, An Introduction to Terraform Syntax*
: Installing Terraform; an overview of Terraform syntax; an overview of the Terraform CLI tool; how to deploy a single server; how to deploy a web server; how to deploy a cluster of web servers; how to deploy a load balancer; how to clean up resources you've created.

*Chapter 3, How to manage Terraform state*
: What is Terraform state; how to store state so multiple team members can access it; how to lock state files to prevent concurrency issues; how to isolate state files to limit the damage from errors; a best-practices file and folder layout for Terraform projects; how to use read-only state.

*Chapter 4, How to create reusable infrastructure with Terraform modules*
: What are modules; how to create a basic module; how to make a module configurable; versioned modules; module tips and tricks; using modules to define reusable, configurable pieces of infrastructure.

*Chapter 5, Terraform tips & tricks: loops, if-statements, deployment, and gotchas*
: Advanced Terraform syntax; loops; if-statements; if-else statements; interpolation functions; zero-downtime deployment; common Terraform gotchas and pitfalls.

*???, How to use Terraform as a team*
: Version control; the golden rule of Terraform; coding guidelines; Terraform style; automated testing for Terraform; documentation; a workflow for teams; automation with Terraform.

Feel free to read the book from start to finish or jump around to the chapters that interest you the most. At the end of the book, in ???, you'll find a list of recommended reading where you can learn more about Terraform, operations, infrastructure as code, and DevOps.

## What you won't find in this book

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

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

To demonstrate these patterns, I've included a number of code examples. Just about all of them are built on top of Amazon Web Services (AWS), although the basic patterns should be largely the same no matter what cloud provider you're using. My goal was to make it as easy as possible for you to try the examples at home. That's why the code examples all focus on one cloud provider (you only have to register with one service, AWS, which offers a free tier for the first year). That's why I omitted just about all other third party dependencies in the book (including HashiCorp's paid service, Atlas). And that's why I've released all the code examples as open source.

## Open source code examples

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

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

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

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

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

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

**A note about versions**

All of the examples in this book were tested against Terraform 0.7.x, which was the most recent major release at the time of writing. Since Terraform is a relatively new tool and has not hit version 1.0.0 yet, it is possible that future releases will contain backwards incompatible changes, and it is likely that some of the best practices will change and evolve over time.

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

## Using the code examples

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

Attribution is appreciated, but not required. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Terraform: Up & Running* by Yevgeniy Brikman (O'Reilly). Copyright 2016 Yevgeniy Brikman, 9781491977088."

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

## Conventions Used in This Book

The following typographical conventions are used in this book:

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

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

**`Constant width bold`**
: Shows commands or other text that should be typed literally by the user.

*`Constant width italic`*
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

## Safari® Books Online

*Safari Books Online* is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

## How to Contact O'Reilly Media

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

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

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

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

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

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

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

# Acknowledgments

*Josh Padnick*

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

*O'Reilly Media*

Thank you for publishing another one of my books. Reading and writing have profoundly transformed my life and I'm proud to have your help in sharing some of my writing with others. A special thanks to Brian Anderson for helping me get this book out in record time.

*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 an order of magnitude 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.

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

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

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

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

## The rise of DevOps

In the not-so-distant past, if you wanted to build a software company, you also had to manage a lot of hardware. You would set up cabinets and racks, load them up with servers, hook up wiring, install cooling, build redundant power systems, and so on. It made sense to have one team, typically called Operations ("Ops"), dedicated to managing this hardware, and a separate team, typically called Developers ("Devs"), dedicated to writing the software.

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

This works well for a while, but as the company grows, you eventually run into problems. It typically plays out like this: since releases are done manually, as the number of servers increases, releases become slow, painful, and unpredictable. The Ops team occasionally makes mistakes, so you end up with *snowflake servers*, where each one has a subtly different configuration from all the others (an 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 3AM after every release, reduce the release cadence to once per week. Then to once per month. Then once every six months. Weeks before the biannual release, teams start trying to merge all their projects together, leading to a huge mess of merge conflicts. No one can stabilize the release branch. Teams start blaming each other. Silos form. The company grinds to a halt.

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

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

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

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

Instead of multi-day 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 their organization, they were able to double the number of features they delivered per month, reduce defects by 50%, reduce *lead times* (the time from coming up with an idea to running code in production) by 60%, and reduce the number of production incidents by 60% to 90%. After HP's LaserJet Firmware division began using DevOps practices, the amount of time their 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 25-50 times per day.[1]

There are four core values in the DevOps movement: Culture, Automation, Measurement, and Sharing (sometimes abbreviated as the acronym CAMS).[2] This book is not meant as a comprehensive overview of DevOps (check out ??? 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 webpage or manually executing shell commands, but through code. This is a concept that is typically called infrastructure as code.

## What is infrastructure as code

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

There are four broad categories of IAC tools:

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

1 *http://itrevolution.com/devops-handbook*

2 *http://devopsdictionary.com/wiki/CAMS*

- Orchestration tools

Let's look at these one at a time.

## Ad hoc scripts

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

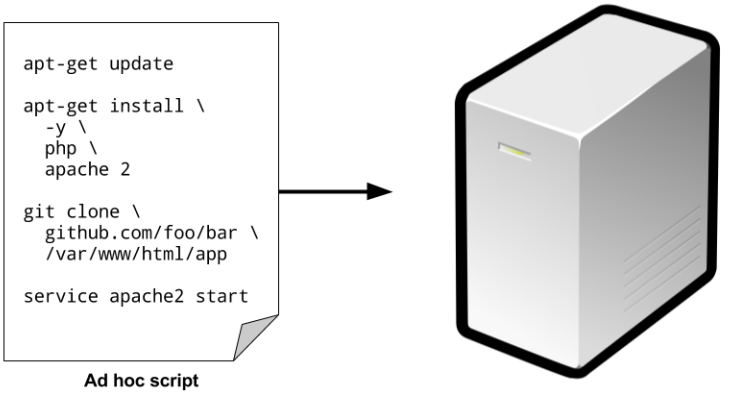



*Figure 1-1. Running an ad hoc script on your server*

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

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

# Install PHP
sudo apt-get install -y php

# Install Apache
```

```
sudo apt-get install -y apache2

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

# Start Apache
sudo service apache2 start
```

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

Whereas tools that are purpose-built for IAC provide concise APIs for accomplishing complicated tasks, if you're using a general-purpose programming language, you have to write completely custom code for every task. Moreover, tools designed for IAC usually enforce a particular structure for your code, whereas with a general-purpose programming language, each developer will use 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 hundreds of servers, databases, load balancers, network configurations, and so on.

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

## Configuration management tools

Chef, Puppet, Ansible, and SaltStack are all *configuration management tools*, which means they are designed to install and manage software on existing servers. For example, here is an *Ansible Role* that configures the same Apache web server as in the previous section:

```
- 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 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, except that Ansible enforces a consistent, predictable structure, including documentation and clearly named parameters. Note how Ansible has support built-in for common tasks, such as installing packages using `apt` and checking out code using `git`. While this simple Ansible Role is of comparable size to the Bash script, once you start using Ansible's higher level commands, the difference becomes more pronounced.

Moreover, unlike ad hoc scripts, which are designed for running on the local machine, Ansible and other configuration management tools are designed specifically for managing large numbers of remote servers, as shown in Figure 1-2.

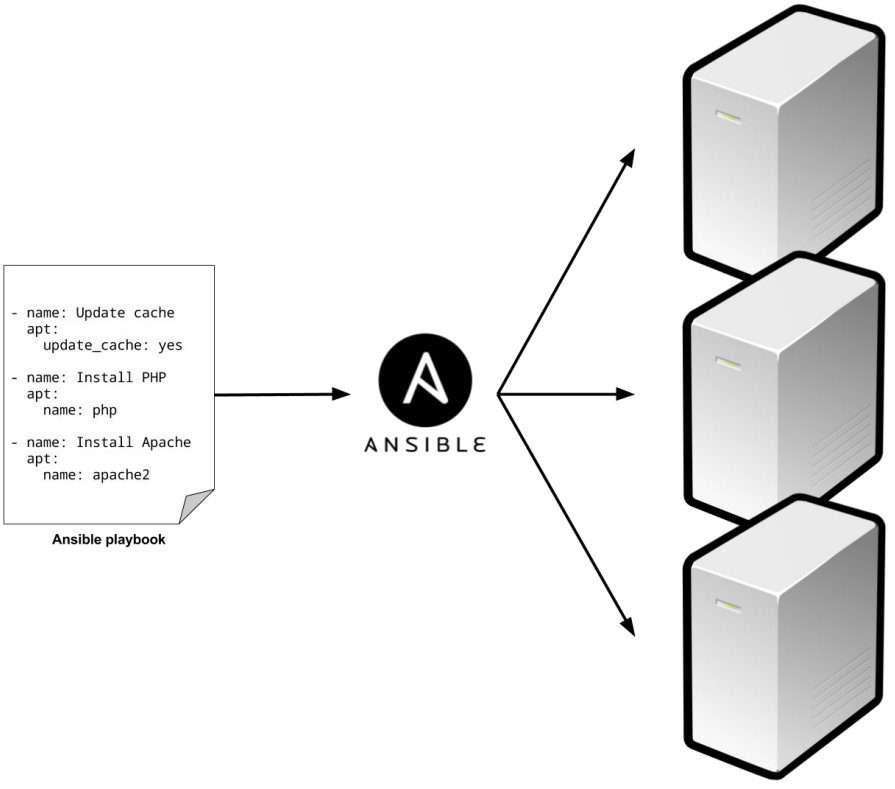



*Figure 1-2. A configuration management tool like Ansible can execute your code across a large number of servers*

For example, to apply the webserver Role above to 5 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
```

## Server templating tools

An alternative to configuration management that has been growing in popularity recently are *server templating tools* such as Docker, Packer, and Vagrant. Instead of launching a bunch of servers and configuring them by running the same code on each one, the idea behind server templating tools is to create an *image* of a server that captures a fully self-contained "snapshot" of the operating system, the software, the files, and all other relevant details. You can give each image a unique version number, deploy the image to any environment, and roll back to previous versions if anything goes wrong. Of course, to deploy images across all your servers, you still need some other IAC tool. For example, you could use Terraform, as you'll see in the next section, or you could use Ansible, as shown in Figure 1-3.

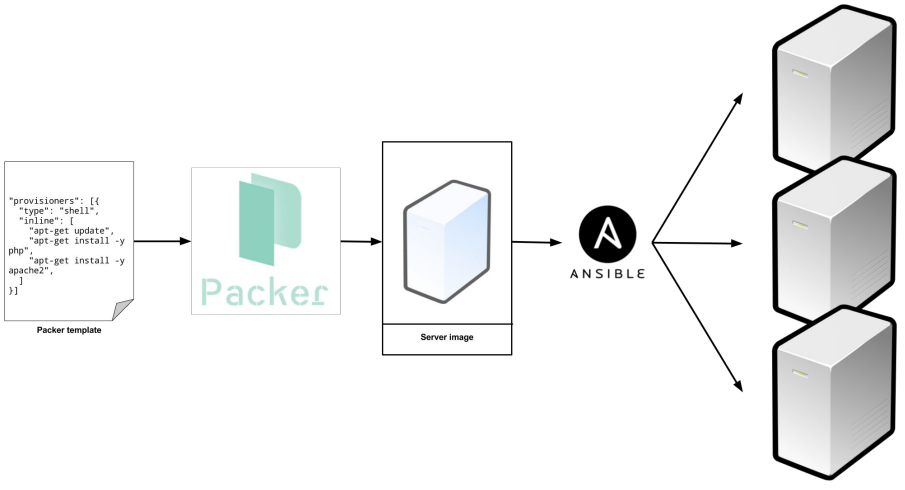



*Figure 1-3. A server templating tool like Packer can be used to create a self-contained image of a server. You can then use other tools, such as Ansible, to deploy that image across all of your servers.*

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

*Virtual Machines*

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

*Containers*

A *container* emulates the user space of an operating system. You run a *container engine*, such as Docker or CoreOS rkt, to create isolated processes, memory, mount points, and networking. Any container (e.g. a Docker Container) you run on top of the container engine can only see its own isolated user space, so it cannot see the host machine or other containers, and will run exactly the same way in all environments (e.g. your computer, a QA server, a production server, etc). Since containers run directly on top of the host machine, the isolation is not as secure as with VMs, but you have virtually no CPU or memory overhead, and

containers can boot up in milliseconds. You can define Container Images as code using tools such as Docker and CoreOs rkt.

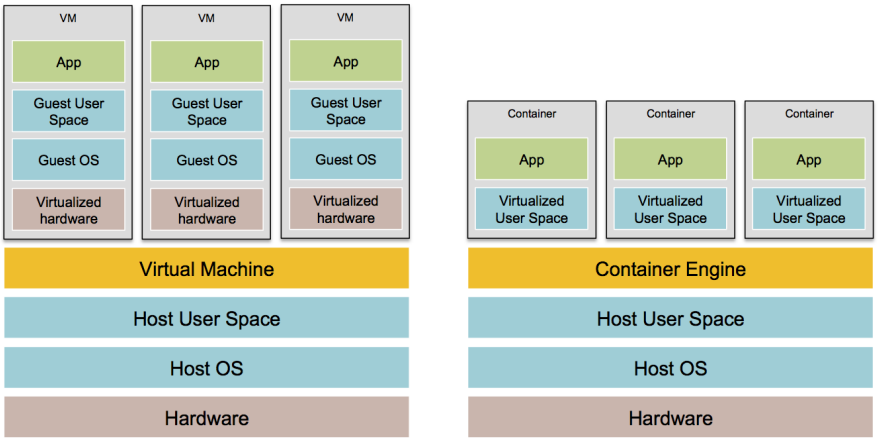



*Figure 1-4. The two main types of images: VMs, on the left, and containers, on the right. VMs virtualize the hardware, whereas containers only virtualize user space.*

For example, here is a Packer template that creates an Amazon Machine Image (AMI), which is a VM Image you can run on Amazon Web Services (AWS):

```
{
  "builders": [{
    "ami_name": "packer-example",
    "instance_type": "t2.micro",
    "region": "us-east-1",
    "type": "amazon-ebs",
    "source_ami": "ami-40d28157",
    "ssh_username": "ubuntu"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sudo apt-get update",
      "sudo apt-get install -y php",
      "sudo apt-get install -y apache2",
      "sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app"
    ]
  }]
}
```

This Packer template configures the same Apache web server you saw in the earlier sections, except for one difference: unlike the previous examples, this Packer template does not start the Apache webserver (e.g. by calling `sudo service apache2 start`).

That's because server templates are typically used to install software in images, but it's only when you run the image (e.g. by deploying it on a server) that you should actually run that software.

You can build an AMI from this template by running `packer build web server.json`, and once the build completes, you can deploy that AMI across all of your AWS servers, tell each one to fire up Apache on boot (you'll see an example of this in the next section), and they will all run exactly the same way.

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

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

## Orchestration tools

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

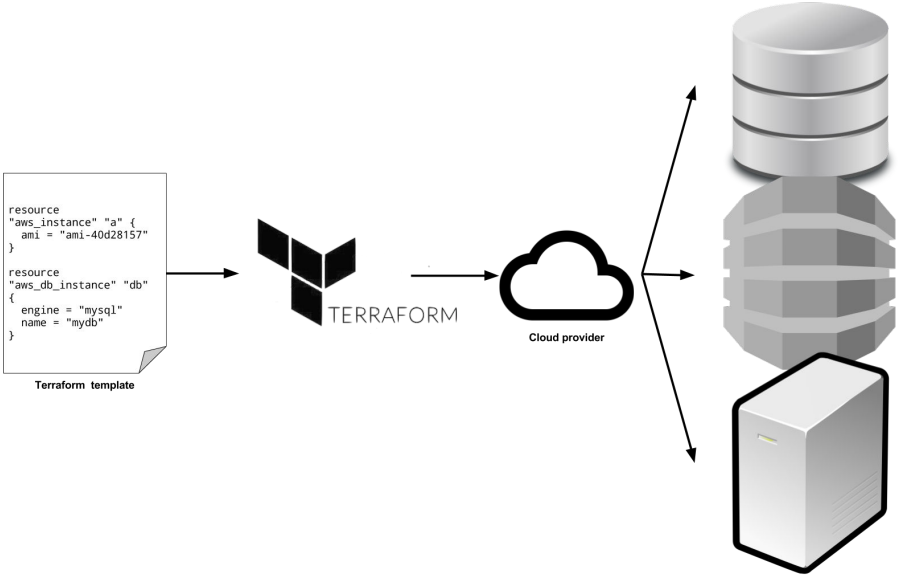



*Figure 1-5. Orchestration tools can be used with your cloud provider to provision servers, databases, load balancers and all other parts of your infrastructure.*

For example, below is Terraform code that deploys a web server, database, and load balancer:

```
resource "aws_instance" "app" {
  instance_type = "t2.micro"
  availability_zone = "us-east-1a"
  ami = "ami-40d28157"

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

resource "aws_db_instance" "db" {
  allocated_storage = 10
  engine = "mysql"
  instance_class = "db.t2.micro"
  name = "mydb"
  username = "admin"
  password = "password"
}

resource "aws_elb" "load_balancer" {
```

```
    name = "frontend-load-balancer"
    instances = ["${aws_instance.app.id}"]
    availability_zones = ["us-east-1a"]

    listener {
      instance_port = 8000
      instance_protocol = "http"
      lb_port = 80
      lb_protocol = "http"
    }
  }
```

Don't worry if some of the syntax isn't familiar to you yet. For now, just focus on the `resource "aws_instance"` part, which is the web server, and note two parameters:

*ami*
: This parameter specifies the ID of an AMI to deploy on the server. You could set it to the ID of an AMI built from the 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 example above uses this script to boot up Apache.

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

## Benefits of infrastructure as code

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

The answer is that code is powerful. In exchange for the up-front investment of converting your manual practices to code, you get dramatic improvements in your ability to deliver software. According to the 2016 State of DevOps Report, organizations that use DevOps practices, such as IAC, deploy 200 times more frequently, recover from failures 24 times faster, and have lead times that are 2,555 times lower.[3]

When you 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:

3 *https://puppet.com/resources/white-paper/2016-state-of-devops-report*

*Self-service*

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

*Speed and safety*

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

*Documentation*

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

*Version control*

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

*Validation*

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

*Reuse*

You can package your infrastructure into reusable modules, so that instead of doing every deployment for every product in every environment from scratch, you can build on top of known, documented, battle-tested pieces.[4]

4 Check out the Gruntwork Infrastructure Packages for an example: *https://medium.com/@gruntwork/gruntwork-infrastructure-packages-7434dc77d0b1*

*Happiness*

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

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

## How Terraform works

Terraform is an open source tool written in the Go programming language. The Go code compiles down into a single binary (or rather, one binary for each of the supported operating systems) called, not surprisingly, `terraform`.

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

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

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

resource "dnsimple_record" "example" {
  domain = "example.com"
  name = "test"
  value = "${aws_instance.example.public_ip}"
  type = "A"
}
```

Even if you've never seen Terraform code before, you shouldn't have too much trouble reading it. The snippet above tells Terraform to make API calls to AWS to deploy a server and then make API calls to DNSSimple to create a DNS entry pointing to that 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 templates and commit those templates 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.

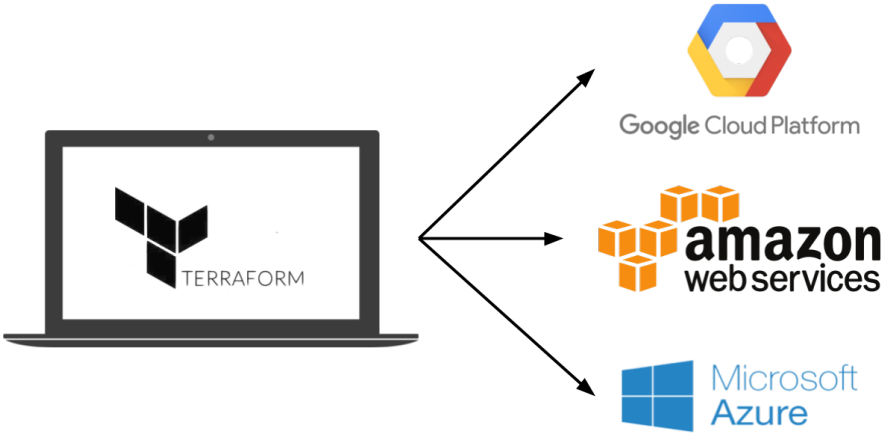



*Figure 1-6. Terraform is a binary that translates the contents of your templates into API calls to cloud providers*

When someone on your team needs to make changes to the infrastructure, instead of updating the infrastructure manually, they make their changes in the Terraform templates, 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.

# How Terraform compares to other infrastructure as code tools

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

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

In the following sections, I'm going to do a detailed comparison between the most popular configuration management and orchestration tools: Terraform, Chef, Puppet, Ansible, SaltStack, CloudFormation, and OpenStack Heat. My goal is to help you decide if Terraform is a good choice by explaining why my company, Gruntwork, picked Terraform as our IAC tool of choice at and, in some sense, why I wrote this book.[5] As with all technology decisions, it's a question of trade-offs and priorities, and while your particular priorities may be different than mine, I hope that sharing this thought process will help you make your own decision.

Here are the main trade-offs to consider:

- Configuration management vs orchestration
- Mutable infrastructure vs immutable infrastructure
- Procedural language vs declarative language
- Client/server architecture vs client-only architecture
- Large community vs small community
- Mature vs cutting-edge

## Configuration management vs orchestration

As you saw earlier, Chef, Puppet, Ansible, and SaltStack are all configuration management tools, whereas CloudFormation, Terraform, and OpenStack Heat are all orches-

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

tration tools. Although the distinction is not entirely clear cut, as configuration management tools can typically do some degree of orchestration (e.g. you can deploy a server with Ansible) and orchestration 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 such as Docker or Packer, the vast majority of your configuration management needs are already taken care of. Once you have an image created from a Dockerfile or Packer template, all that's left to do is provision the infrastructure for running those images. And when it comes to provisioning, an orchestration tool is going to be your best choice.

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

## Mutable infrastructure vs immutable infrastructure

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

If you're using an orchestration tool such as Terraform to deploy machine images created by Docker or Packer, then most "changes" are actually deployments of a new server. For example, to deploy a new version of OpenSSL, you would create a new image using Packer with the new version of OpenSSL already installed, deploy that image across a set of new servers, and then undeploy the old 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.

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 orchestration 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 redpeloying all your servers for a trivial change can take a long time. Moreover, immutability only lasts until you actually run the image. Once a server is up and running, it'll start making changes on the hard drive and experiencing some degree of configuration drift (although this is mitigated if you deploy frequently).

## Procedural language vs declarative language

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

To demonstrate the difference, let's go through an example. Imagine you wanted to deploy 10 servers ("EC2 Instances" in AWS lingo) to run an AMI with ID `ami-40d28157` (Ubuntu 16.04). Here is a simplified example of an Ansible template that does this using a procedural approach:

```
- ec2:
    count: 10
    image: ami-40d28157
    instance_type: t2.micro
```

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

```
resource "aws_instance" "example" {
  count = 10
  ami = "ami-40d28157"
  instance_type = "t2.micro"
}
```

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

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

```
- ec2:
    count: 5
    image: ami-40d28157
    instance_type: t2.micro
```

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

```
resource "aws_instance" "example" {
  count = 15
  ami = "ami-40d28157"
```

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

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

```
> terraform plan

+ aws_instance.example.11
    ami:                      "ami-40d28157"
    instance_type:            "t2.micro"
+ aws_instance.example.12
    ami:                      "ami-40d28157"
    instance_type:            "ami-40d28157"
+ aws_instance.example.13
    ami:                      "ami-40d28157"
    instance_type:            "t2.micro"
+ aws_instance.example.14
    ami:                      "ami-40d28157"
    instance_type:            "t2.micro"
+ aws_instance.example.15
    ami:                      "ami-40d28157"
    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-408c7f28`? With the procedural approach, both of your previous Ansible templates are again not useful, so you have to write yet another template to track down the 10 servers you deployed previous (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 template once again and simply change the `ami` parameter to `ami-408c7f28`:

```
resource "aws_instance" "example" {
  count = 15
  ami = "ami-408c7f28"
  instance_type = "t2.micro"
}
```

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

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

With Terraform's declarative approach, the code always represents the latest state of your infrastructure. At a glance, you can tell what's currently deployed and how it's configured, without having to worry about history or timing. This also makes it easy to create reusable code, as you don't have to manually account for the current state of the world. Instead, you just focus on describing your desired state, and Terraform figures out how to get from one state to the other automatically. As a result, Terraform codebases tend to stay small and easy to understand.

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

## Client/server architecture vs client-only architecture

Chef, Puppet, SaltStack, CloudFormation, and Heat all use a client/server architecture by default, as shown in Figure 1-7. The client, which could be a web UI or a CLI tool, is what you use to issue commands (e.g "deploy X"). Those commands go to a server, which is responsible for executing your commands and storing the state of the system. To execute those commands, the server talks to agents, which must be running on every sever you want to configure.

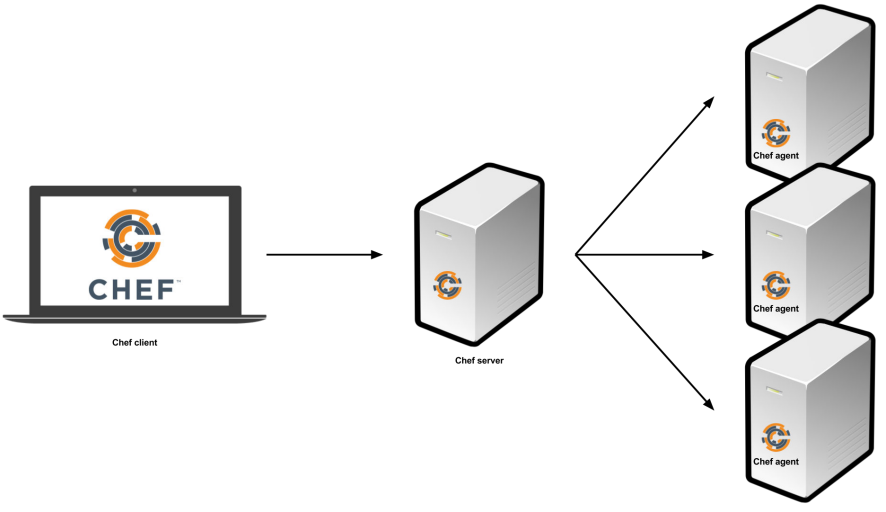



*Figure 1-7. Chef, Puppet, SaltStack, CloudFormation, and Heat all use a client/server architecture. The image above shows a Chef client running on your computer, which talks to a Chef server, which deploys changes by talking to Chef agents running on all your other servers.*

The client-server architecture has some drawbacks:

- You have to install and run extra software on every one of your servers.
- You have to deploy an extra server (or even a cluster of servers for high availability) just for configuration management.
- You not only have to install this extra software and hardware, but you also have to maintain it, upgrade it, make backups of it, monitor it, and restore it in case of outages.
- Since the client, server, and agents all need to communicate over the network, you have to open extra ports for them, and configure ways for them to authenticate to each other, all of which increases your surface area to attackers.
- All of these extra moving parts introduce a large number of new failure modes into your infrastructure. When you get a bug report at 3AM, you'll have to figure out if it's a bug in your application code, or your IAC code, or the configuration management client software, or the configuration management agent software, or the configuration management server software, or the ports all those configuration management pieces use to communicate, or the way they authenticate to each other, or…

Ansible, and Terraform, use a client-only architecture, as shown in Figure 1-8. The Ansible client works by connecting directly to your servers over SSH. Terraform uses cloud provider APIs to provision infrastructure, so there are no new authentication mechanisms beyond what you're using with the cloud provider already, and there is no need for direct access to your servers.

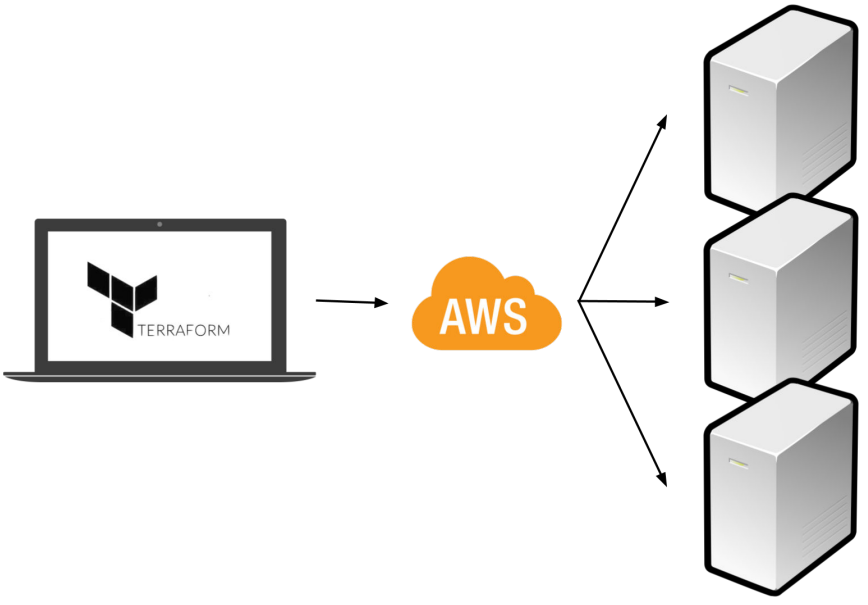



*Figure 1-8. Terraform uses a client-only architecture. All you need to run is the Terraform client and it takes care of the rest by using the APIs of cloud providers, such as AWS.*

It's worth mentioning that while CloudFormation uses a client/server architecture, AWS handles all the server and agent details (including installing them, deploying them, and updating them), so as an end user, you have an experience that feels like client-only.

## Large community vs 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 StackOverflow), and how easy it is to hire someone to help you (e.g. an employee, consultant, or support company).

It's hard to do an accurate comparison between communities, but you can spot some trends by searching online. Table 1-1 shows a comparison of popular IAC tools from September, 2016, including whether they are open source or closed source, what cloud providers they support, the total number of contributors and stars on GitHub, how many active changes and issues there were in the month of September, how many open source libraries are available for the tool, the number of questions listed for that tool on StackOverflow, and the number of jobs that mention the tool on Indeed.com.[6]

*Table 1-1. A comparison of IAC communities*

| | Source | Cloud | Contributors | Stars | Commits in Sept | Bugs in Sept | Libraries | StackOverflow | Jobs |
|---|---|---|---|---|---|---|---|---|---|
| Chef | Open | All | 477 | 4,439 | 182 | 58 | 3,052[a] | 4,187 | 5,631[b] |
| Puppet | Open | All | 432 | 4,158 | 79 | 130[c] | 4,435[d] | 2,639 | 5,213[e] |
| Ansible | Open | All | 1,488 | 18,895 | 340 | 315 | 8,044[f] | 3,633 | 3,901 |
| SaltStack | Open | All | 1,596 | 6,897 | 689 | 347 | 240[g] | 614 | 454 |
| CloudFormation | Closed | AWS | ? | ? | ? | ? | 240[h] | 613 | 665 |
| Heat | Open | All | 283 | 283 | 83 | 36[i] | 0[j] | 52 | 72[k] |
| Terraform | Open | All | 653 | 5,732 | 440 | 480 | 40[l] | 131 | 392 |

[a] This is the number of cookbooks in the Chef Supermarket: *https://supermarket.chef.io/cookbooks*
[b] To avoid false positives for the term "chef", I searched for "chef engineer"
[c] Based on the Puppet Labs JIRA account: *https://tickets.puppetlabs.com/secure/Dashboard.jspa*
[d] This is the number of modules in the Puppet Forge: *https://forge.puppet.com/*
[e] To avoid false positives for the term "puppet", I searched for "puppet engineer"
[f] This is the number of reusable roles in Ansible Galaxy: *https://galaxy.ansible.com/*
[g] This is the number of formulas in the Salt Stack Formulas GitHub account: *https://github.com/saltstack-formulas*
[h] This is the number of templates in the awslabs GitHub account: *https://github.com/awslabs*
[i] Based on the OpenStack bug tracker: *https://bugs.launchpad.net/openstack/*
[j] I could not find any collections of community Heat templates
[k] To avoid false positives for the term "heat", I searched for "openstack"
[l] This is the number of modules in the terraform-community-modules repo: *https://github.com/terraform-community-modules*

Obviously, this is not a perfect apples-to-apples comparison. For example, some of the tools have more than one repository, some use other methods for bug tracking and questions, searching for jobs with common words like "chef" or "puppet" is tricky, 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

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

is closed source, and only works with AWS. Second, Chef, Puppet, and Ansible seem to be the most popular tools, although SaltStack and Terraform aren't too far behind.

### Mature vs cutting edge

Another key factor to consider when picking any technology is maturity. Table 1-2 shows a comparison of the initial release dates and current version number (as of October, 2016), for of each of the IAC tools:

*Table 1-2. A comparison of IAC maturity*

| | Initial release | Current version |
|---|---|---|
| Chef | 2009 | 12.16.39 |
| Puppet | 2005 | 4.9.0 |
| Ansible | 2012 | 2.2.0 |
| SaltStack | 2011 | 2016.3.3 |
| CloudFormation | 2011 | ? |
| Heat | 2012 | 7.0.0 |
| Terraform | 2014 | 0.7.10 |

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

## Conclusion

Putting it all together, below is a table that shows how the most popular IAC tools stack up:

*Table 1-3. A comparison of IAC tools*

| | Source | Cloud | Type | Infrastructure | Language | Architecture | Community | Maturity |
|---|---|---|---|---|---|---|---|---|
| Chef | Open | All | Config Mgmt | Mutable | Procedural | Client/Server | Large | High |
| Puppet | Open | All | Config Mgmt | Mutable | Declarative | Client/Server | Large | High |
| Ansible | Open | All | Config Mgmt | Mutable | Procedural | Client-only | Large | Medium |
| SaltStack | Open | All | Config Mgmt | Mutable | Declarative | Client/Server | Medium | Medium |
| CloudFormation | Closed | AWS | Orchestration | Immutable | Declarative | Client/Server | Small | Medium |
| Heat | Open | All | Orchestration | Immutable | Declarative | Client/Server | Small | Low |
| Terraform | Open | All | Orchestration | Immutable | Declarative | Client-only | Medium | Low |

At Gruntwork, what we wanted was an open source, cloud-agnostic orchestration tool that supported immutable infrastructure, a declarative language, a client-only architecture, had a large community, and was mature. From the table above, Terraform, while not perfect, comes the closest to meeting all of our criteria.

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

CHAPTER 2

# An Introduction to Terraform Syntax

In this chapter, you're going to learn the basics of Terraform syntax. It's an easy language to learn, so in the span of about 25 pages, you'll go from running your first Terraform commands all the way up to using Terraform to deploy a cluster of servers with a load balancer that distributes traffic across them. This infrastructure is a good starting point for running scalable, highly-available web services and microservices. In subsequent chapters, you'll evolve this example even further.

Terraform can provision infrastructure across many different types of cloud providers, including Amazon Web Services (AWS), Azure, Google Cloud, DigitalOcean, and many others. For just about all of the code examples in this chapter and the rest of the book, you are going to use AWS. AWS is a good choice for learning Terraform because:

- AWS is the most popular cloud infrastructure provider, by far. It has a 45% share in the cloud infrastructure market, which is more than the next three biggest competitors (Microsoft, Google, and IBM) combined.[1]
- AWS provides a huge range of reliable and scalable cloud hosting services, including Elastic Compute Cloud (EC2), AKA virtual servers, Auto Scaling Groups (ASGs), AKA a cluster of servers,, and Elastic Load Balancing (ELB) AKA a load balancer.[2].

1 *http://www.geekwire.com/2016/study-aws-45-share-public-cloud-infrastructure-market-microsoft-google-ibm-combined/*

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

- AWS offers a generous Free Tier which should allow you to run all of these examples for free. If you already used up your free tier credits, the examples in this book should still cost you no more than a few dollars.

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

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

**Example code**

As a reminder, all of the code examples in the book can be found at the following URL:

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

## Set up your AWS account

If you don't already have an AWS account, head over to *https://aws.amazon.com/* and sign up. When you first register for AWS, you initially sign in as *root user*. This user account has access permissions to do absolutely anything in the account, so from a security perspective, it's not a good idea to use the root user on a day-to-day basis. In fact, the *only* thing you should use the root user for is to create other user accounts with more limited permissions, and switch to one of those accounts immediately.[3]

To create a more limited user account, you will need to use the *Identity and Access Management (IAM)* service. IAM is where you manage users accounts as well as the permissions for each user. To create a new *IAM user*, head over to the IAM Console, click "Users", and click the blue "Create New Users" button. Enter a name for the user and make sure "Generate an access key for each user" is checked, as shown in Figure 2-1.

3 For more details on AWS user management best practices, see *https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html*

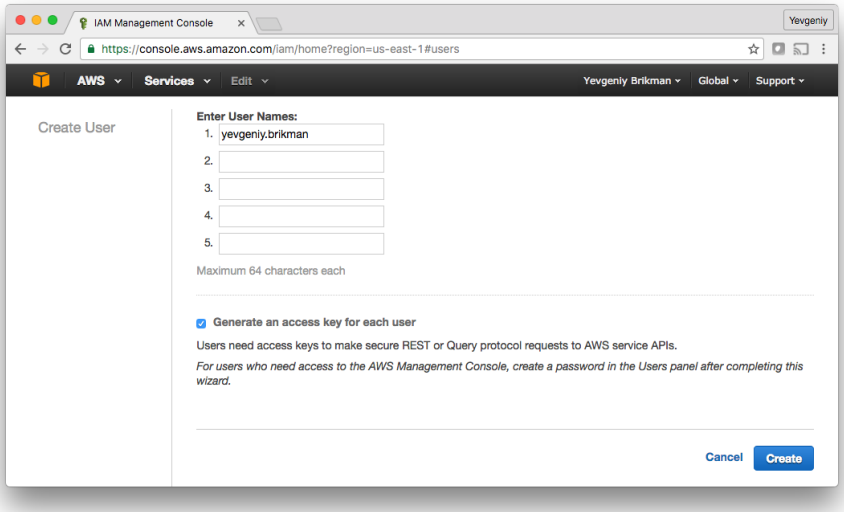



*Figure 2-1. Create a new IAM user*

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

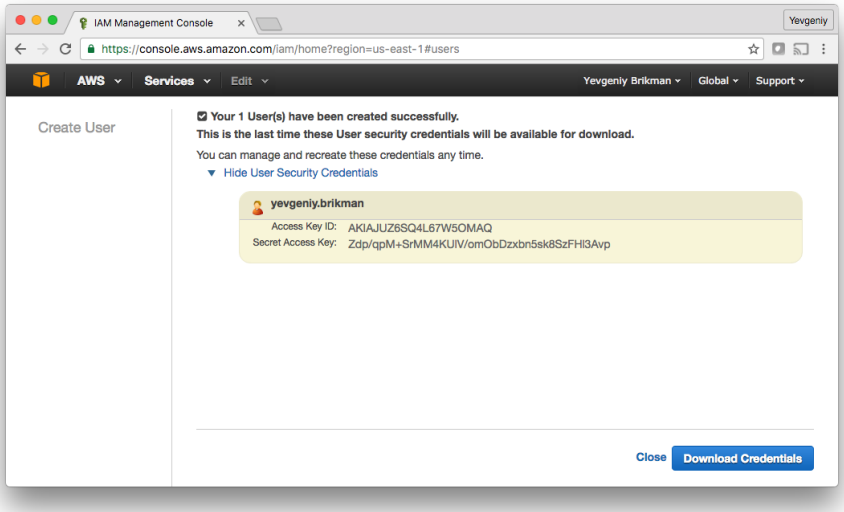



*Figure 2-2. Store your AWS credentials somewhere secure. Never share them with anyone. Don't worry, the ones in the screenshot are fake.*

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

To give an IAM user permissions to do something, you need to associate one or more IAM Policies with that user's account. An *IAM Policy* is a JSON document that defines what a user is or isn't allowed to do. You can create your own IAM Policies or use some of the pre-defined IAM Policies, which are known as *Managed Policies*.[4]

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

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

4 You can learn more about IAM Policies here: *https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_managed-vs-inline.html*

5. `CloudWatchFullAccess`: required for Chapter 5.
6. `IAMFullAccess`: required for Chapter 5.

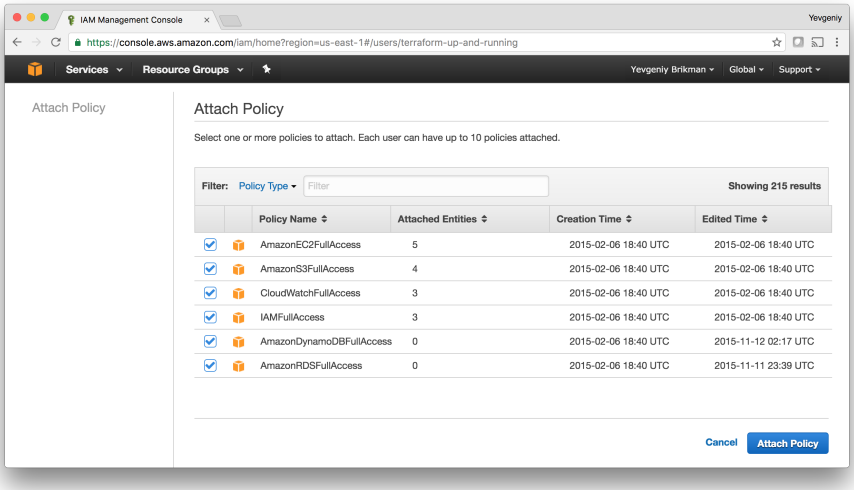



*Figure 2-3. Add several Managed IAM Policies to your new IAM user*

**A note on Default VPCs**

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

# Install Terraform

You can download Terraform from the Terraform homepage. Click the download link, select the appropriate package for your operating system, download the zip archive, and unzip it into the directory where you want Terraform to be installed. The

archive will extract a single binary called `terraform`, which you'll want to add to your `PATH` environment variable.

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

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

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

In order for Terraform to be able to make changes in your AWS account, you will need to set the AWS credentials for the IAM user you created earlier as the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`:

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

**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 `~/.aws/credentials`, which are automatically generated if you are run the `configure` command on the AWS CLI, or IAM Roles, which you can add to almost any resource in AWS. For more info, see Configuring the AWS Command Line Interface.

# Deploy a single server

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

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

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

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

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

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

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

The general syntax for a Terraform resource is:

```
resource "PROVIDER_TYPE" "NAME" {
  [CONFIG ...]
}
```

Where `PROVIDER` is the name of a provider (e.g. `aws`), `NAME` is an identifier you can use throughout the Terraform code to refer to this resource (e.g. `example`), and `CONFIG` consists of one or more configuration parameters that are specific to that resource (e.g. `ami = "ami-40d28157"`). For the `aws_instance` resource, there are many different configuration parameters, but for now, you only need to set the following ones:[6]

*ami*

The Amazon Machine Image (AMI) to run on the EC2 Instance. You can find free and paid AMIs in the AWS Marketplace or create your own using tools such as Packer (see "Server templating tools" on page 23 for a discussion of machine

5 You can learn more about AWS regions and availability zones here: *http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html*

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

images and server templating). The example above sets the `ami` parameter to the ID of an Ubuntu 16.04 AMI in `us-east-1`.

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

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

```
> terraform plan

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

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

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

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

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

```
> terraform apply

aws_instance.example: Creating...
  ami:                      "" => "ami-40d28157"
  availability_zone:        "" => "<computed>"
```

```
  instance_state:           "" => "<computed>"
  instance_type:            "" => "t2.micro"
  key_name:                 "" => "<computed>"
  private_dns:              "" => "<computed>"
  private_ip:               "" => "<computed>"
  public_dns:               "" => "<computed>"
  public_ip:                "" => "<computed>"
  security_groups.#:        "" => "<computed>"
  subnet_id:                "" => "<computed>"
  vpc_security_group_ids.#: "" => "<computed>"
  (...)

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

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

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

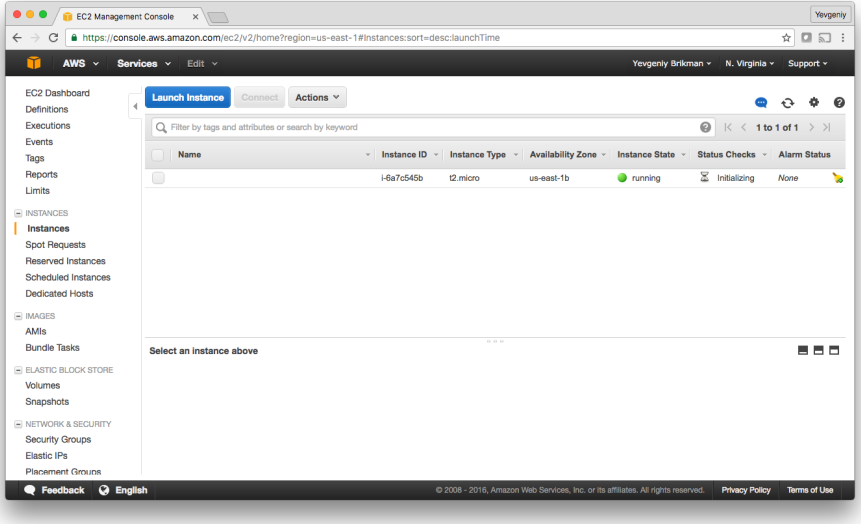



*Figure 2-4. A single EC2 Instance*

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

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

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

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

```
> terraform plan

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

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

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

Terraform keeps track of all the resources it already created for this set of templates, so it knows your EC2 Instance already exists (notice Terraform says "Refreshing state…" when you run the `plan` command), and it can show you a diff between what's currently deployed and what's in your Terraform code (this is one of the advantages of using a declarative language over a procedural one, as discussed in "How Terraform compares to other infrastructure as code tools" on page 32). The diff above shows that Terraform wants to create a single tag called "Name", which is exactly what you need, so run the `apply` command again.

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

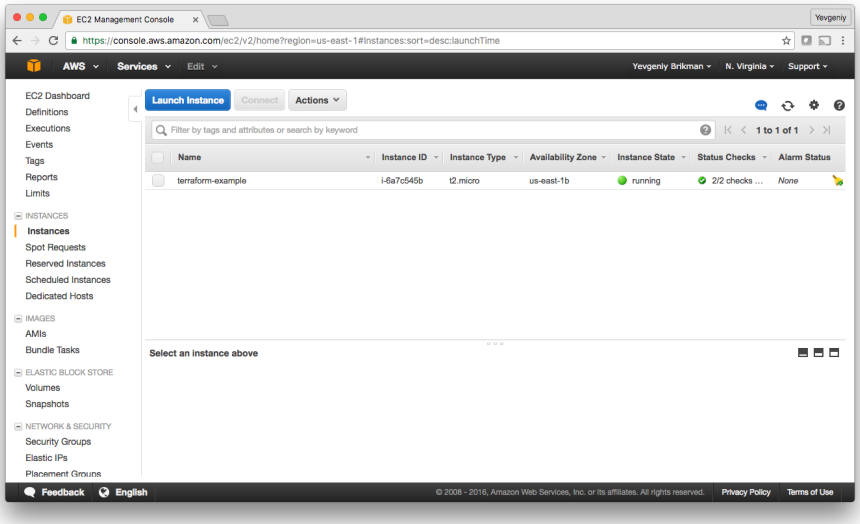



*Figure 2-5. The EC2 Instance now has a name tag*

# Deploy a single web server

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

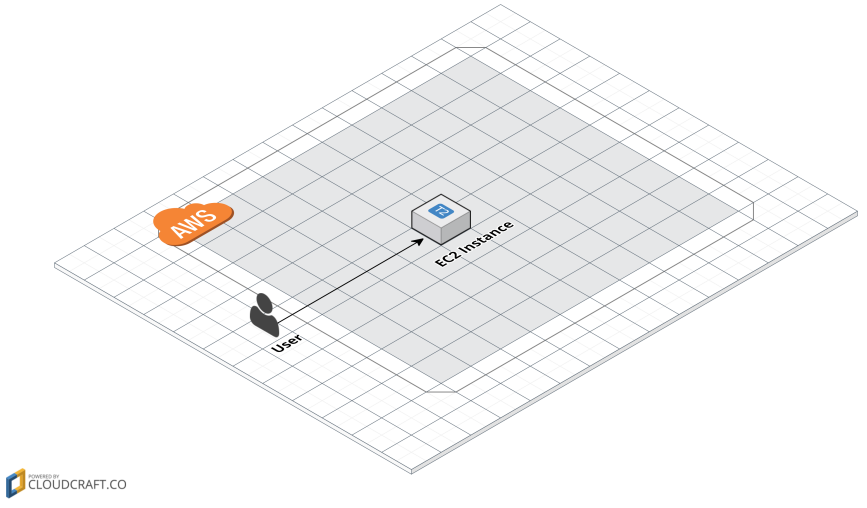



*Figure 2-6. Start with a simple architecture: a single web server running in AWS that responds to HTTP requests*

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

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

This is a bash script that writes the text "Hello, World" into `index.html` and runs a tool called busybox (which is installed by default on Ubuntu) to fire up a web server on port 8080 to serve that file. I wrapped the `busybox` command with `nohup` and `&` so that the web server runs permanently in the background, while the bash script itself can exit.

How do you get the EC2 Instance to run this script? Normally, as discussed in "Server templating tools" on page 23, 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, there is nothing to install. Therefore, in such a simple case, you can just run the script above as part of the EC2 Instance's *User Data* configuration, which AWS will execute when the Instance is booting:

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

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

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

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

The `<<-EOF` and `EOF` are Terraform's *heredoc* syntax, which allows you to create multi-line strings without having to insert new-line characters all over the place.

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

```
resource "aws_security_group" "instance" {
  name = "terraform-example-instance"
  ingress {
    from_port = 8080
    to_port = 8080
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

The code above creates a new resource called `aws_security_group` (notice how all resources for the AWS provider start with `aws_`) and specifies that this group allows incoming TCP requests on port 8080 from the CIDR block 0.0.0.0/0. *CIDR blocks* are a concise way to specify IP address ranges. For example, a CIDR block of 10.0.0.0/24 represents all IP addresses between 10.0.0.0 and 10.0.0.255. The CIDR block 0.0.0.0/0 is an IP address range that includes all possible IP addresses, so the security group above allows incoming requests on port 8080 from any IP.[8]

Note that port 8080 is now duplicated in both the security group and the User Data configuration, so it would be easy to update it in one place but forget to make the same change in the other place. To reduce duplication and to make your code more configurable, Terraform allows you to define *input variables*.

8 To learn more about how CIDR works, see *https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing*. For a handy calculator that converts between IP address ranges and CIDR notation, see *http://www.ipaddressguide.com/cidr*.

The declaration for an input variable consists of the keyword `variable`, followed by a name for the variable, and then the body. The body can contain three parameters, all of them optional:

*description*
: It's always a good idea to use this parameter to document how a variable is used. Your teammates will not only be able to see this description while reading the code, but also when running the `plan` or `apply` commands (you'll see an example of this below).

*default*
: There are a number of ways to provide a value for the variable, including passing it in at the command line (using the `-var` option), via a file (using the `-var-file` option), or via an environment variable (Terraform looks for environment variables of the name `TF_VAR_<variable_name>`). If no value is passed in, the variable will fall back to this default value. If there is no default value, Terraform will interactively prompt the user for one.

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

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

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

And here's a map:

```
variable "map_example" {
  description = "An example of a map in Terraform"
  type = "map"
  default = {
    key1 = "value1"
    key2 = "value2"
    key3 = "value3"
  }
}
```

For the web server example, all you need is a number, which in Terraform, are automatically coerced to strings, so you can omit the type:[9]

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

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

```
> terraform plan

var.server_port
  The port the server will use for HTTP requests

Enter a value:
```

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

```
> terraform plan -var server_port="8080"
```

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

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

To use an input variable in your resources, you need to use Terraform's *interpolation syntax*:

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

Whenever you see a dollar sign and curly braces inside of double quotes, that means Terraform is going to process the contents in a special way. You'll see many different uses for this syntax throughout the book. The first and simplest version is to use interpolation syntax to set a parameter to the value of a variable, such as setting 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 {
```

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

```
    from_port = "${var.server_port}"
    to_port = "${var.server_port}"
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

The syntax for looking up a variable is `"${var.VARIABLE_NAME}"`. You can use the same syntax to set the port number used by `busybox` in the User Data of the EC2 Instance:

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

One last thing to do: you need to tell the EC2 Instance to actually use the new security group. To do that, you need to pass the ID of the security group into the `vpc_security_group_ids` parameter of the `aws_instance` resource. How do you get this ID?

In Terraform, every resource has attributes that you can reference using interpolation with the syntax `${TYPE.NAME.ATTRIBUTE}` (you can find the list of available attributes in the documentation for each resource). For example, here is how you set the `vpc_security_group_ids` parameter of the `aws_security_group` resource to the ID of the security group:

```
resource "aws_instance" "example" {
  ami = "ami-40d28157"
  instance_type = "t2.micro"
  vpc_security_group_ids = ["${aws_security_group.instance.id}"]

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

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

When one resource references another resource using interpolation syntax, you create an *implicit dependency*. Terraform parses these dependencies, builds a dependency graph from them, and uses that to automatically figure out in what order it should create resources. For example, Terraform knows it needs to create the security group before the EC2 Instance, since the EC2 Instance references the ID of the security

group. You can even get Terraform to show you the dependency graph by running the `graph` command:

```
> terraform graph

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

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

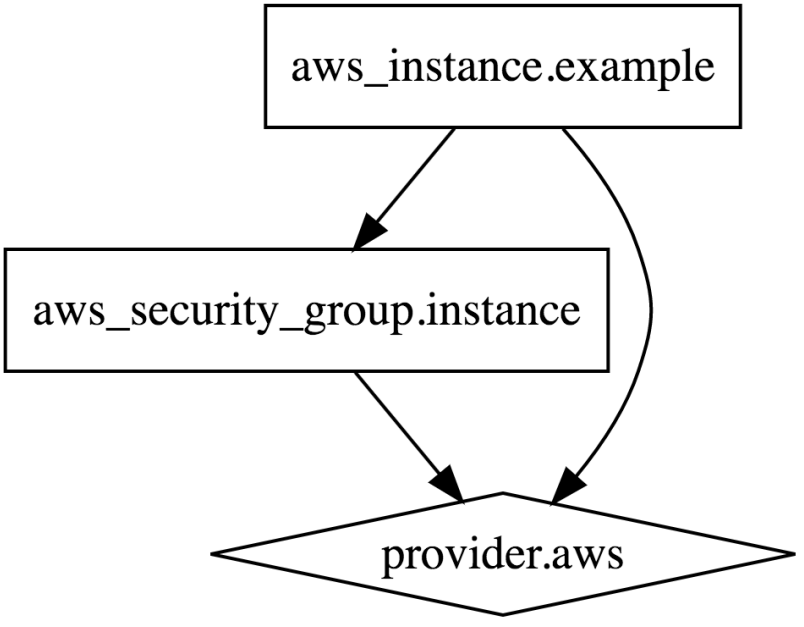



*Figure 2-7. The dependency graph for the EC2 Instance and its security group*

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

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

```
> terraform plan

(...)


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

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

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

In Terraform, most changes to an EC2 Instance, other than metadata such as tags, actually create a completely new Instance. This is an example of the immutable infrastructure paradigm discussed in "Server templating tools" on page 23. It's worth mentioning that while the web server is being replaced, your users would experience downtime; you'll see how to do a zero-downtime deployment with Terraform in Chapter 5.

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

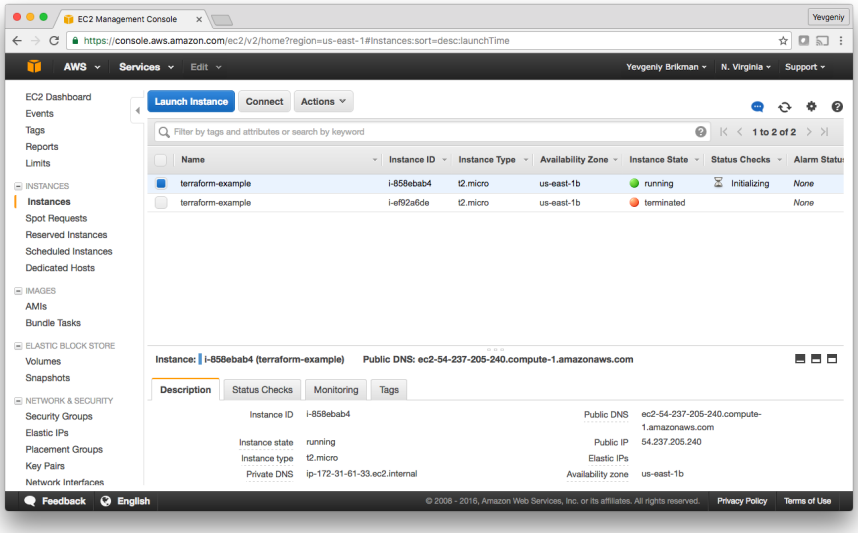



*Figure 2-8. The new EC2 Instance with the web server code replaces the old Instance*

In the description panel at the bottom of the screen, you'll also see the public IP address of this EC2 Instance. Give it a minute or two to boot up and then try to curl this IP address at port 8080:

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

Yay, a working web server running in AWS! However, having to manually poke around the EC2 console to find this IP address is no fun. Fortunately, you can do better by specifying an *output variable* in your Terraform code:

```
output "public_ip" {
  value = "${aws_instance.example.public_ip}"
}
```

The code above uses interpolation syntax again, this time to reference the `public_ip` attribute of the `aws_instance` resource. If you run the `apply` command again, Terraform will not apply any changes (since you haven't changed any resources), but it will show you the new output:

```
> terraform apply

aws_security_group.instance: Refreshing state... (ID: sg-db91dba1)
aws_instance.example: Refreshing state... (ID: i-61744350)

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

```
Outputs:

public_ip = 54.174.13.5
```

As you can see, output variables show up in the console after you run `terraform apply`. You can also use the `terraform output` command to list outputs without applying any changes and `terraform output OUTPUT_NAME` to see the value of a specific output. Input and output variables are essential ingredients in creating reusable infrastructure code, a topic you'll see more of in Chapter 4.

**Network security**

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

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

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

## Deploy a 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.[10]

10 For a deeper look at how to build highly-available and scalable systems on AWS, see: *https://www.airpair.com/aws/posts/building-a-scalable-web-app-on-amazon-web-services-p1*

Managing such a cluster manually is a lot of work. Fortunately, you can let AWS take care of it for by you using an *Auto Scaling Group (ASG)*, as shown in Figure 2-9. An ASG takes care of a lot of tasks for you completely automatically, including launching a cluster of EC2 Instances, monitoring the health of each Instance, replacing failed Instances, and adjusting the size of the cluster in response to load.

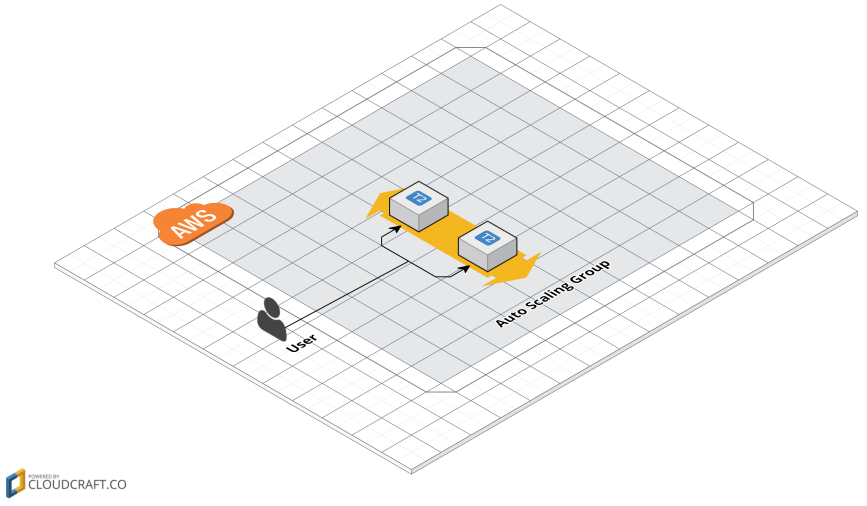



*Figure 2-9. Instead of a single web server, run a cluster of web servers using an Auto Scaling Group*

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

```
resource "aws_launch_configuration" "example" {
  image_id = "ami-40d28157"
  instance_type = "t2.micro"
  security_groups = ["${aws_security_group.instance.id}"]

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

  lifecycle {
    create_before_destroy = true
  }
}
```

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

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

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

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

  ingress {
    from_port = "${var.server_port}"
    to_port = "${var.server_port}"
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  lifecycle {
    create_before_destroy = true
  }
}
```

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

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

  min_size = 2
  max_size = 10

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

This ASG will run between 2 and 10 EC2 Instances (defaulting to 2 for the initial launch), each tagged with the name "terraform-example." The ASG references the launch configuration you created earlier using Terraform's interpolation syntax.

To make this ASG work, you need to specify one more parameter: `availability_zones`. This parameter specifies into which availability zones (AZs) the EC2 Instances should be deployed. Each AZ represents an isolated AWS data center, so by deploying your Instances across multiple AZs, you ensure that your service can keep running even if some of the AZs have an outage. You could hard-code the list of AZs (e.g. set it to `["us-east-1a", "us-east-1b"]`), but each AWS account has access to a slightly different set of AZs, so a better option is to use the `aws_availability_zones` data source to fetch the AZs specific to your AWS account:

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

A *data source* represents a piece of read-only information that is fetched from the provider (in this case, AWS) every time you run Terraform. Adding a data source to your Terraform templates does not create anything new; it's just a way to query the provider's APIs for data. There are data sources to not only get the list of availability zones, but also AMI IDs, IP address ranges, and the current user's identity.

To use the data source, you reference it using the interpolation syntax of the format `${data.TYPE.NAME.ATTRIBUTE}`. For example, here is how you pass the names of the AZs from the `aws_availability_zones` data source into the `availability_zones` parameter of the ASG:

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

  min_size = 2
  max_size = 10

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

# Deploy a load balancer

At this point, you can deploy your ASG, but you'll have a small problem: now you have several different servers, each with their own IP addresses, but you typically want to give your end users only a single IP to hit. One way to solve this problem is to deploy a *load balancer* to distribute traffic across your servers and to give all your users the IP (actually, the DNS name) of the load balancer. Creating a load balancer

that is highly available and scalable is a lot of work. Once again, you can let AWS take care of it for you, this time by using Amazon's *Elastic Load Balancer (ELB)* service, as shown in Figure 2-10.

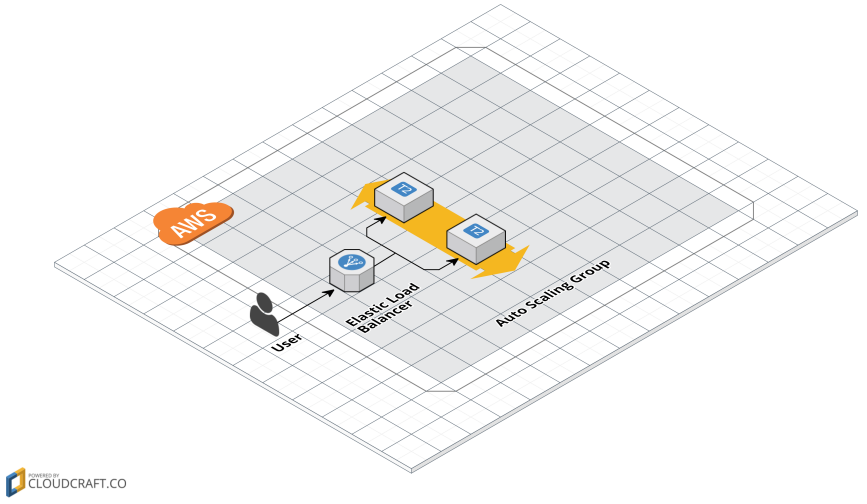



*Figure 2-10. Use an Elastic Load Balancer to distribute traffic across the Auto Scaling Group*

To create an ELB with Terraform, you use the `aws_elb` resource:

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

This creates an ELB that will work across all of the AZs in your account. Of course, the definition above doesn't do much until you tell the ELB how to route requests. To do that, you add one or more *listeners* which specify what port the ELB should listen on and what port it should route the request to:

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

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

The code above tells the ELB to receive HTTP requests on port 80 (the default port for HTTP) and to route them to the port used by the Instances in the ASG. Note that, by default, ELBs don't allow any incoming or outgoing traffic (just like EC2 Instances), so you need to create a new security group that explicitly allows incoming requests on port 80:

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

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

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

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

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

The ELB has one other trick up its sleeve: it can periodically check the health of your EC2 Instances and, if an Instance is unhealthy, it will automatically stop routing traffic to it. To configure a health check for the ELB, you add a `health_check` block. For examle, here is a `health_check` block that sends an HTTP request every 30 seconds to the "/" URL of each of the EC2 Instances in the ASG and only considers an Instance healthy if it responds with a 200 OK:

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

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

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

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

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

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

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

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

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

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

  min_size = 2
  max_size = 10

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

```
  }
}
```

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

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

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

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

```
Outputs:
elb_dns_name = terraform-asg-example-123.us-east-1.elb.amazonaws.com
```

Copy this URL down. It'll take a couple minutes for the Instances to boot and show up as healthy in the ELB. In the meantime, you can inspect what you've deployed. Open up the ASG section of the EC2 console, and you should see that the ASG has been created, as shown in Figure 2-11.

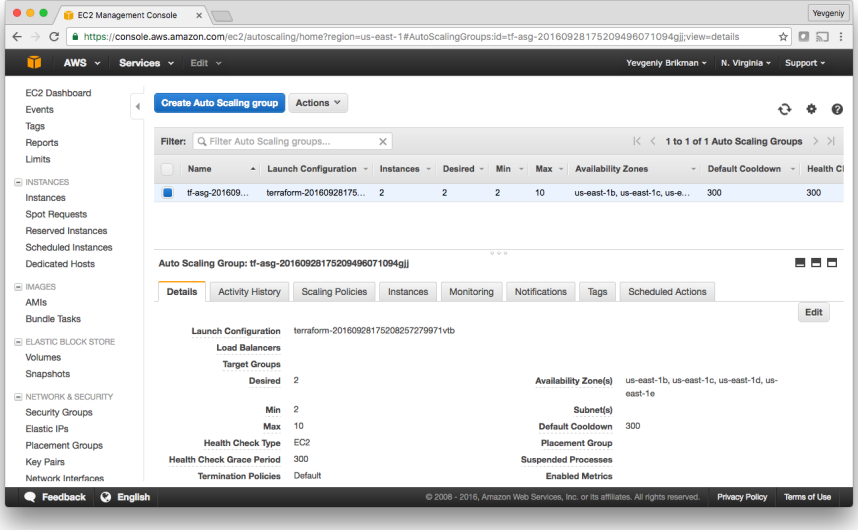



*Figure 2-11. The Auto Scaling Group*

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

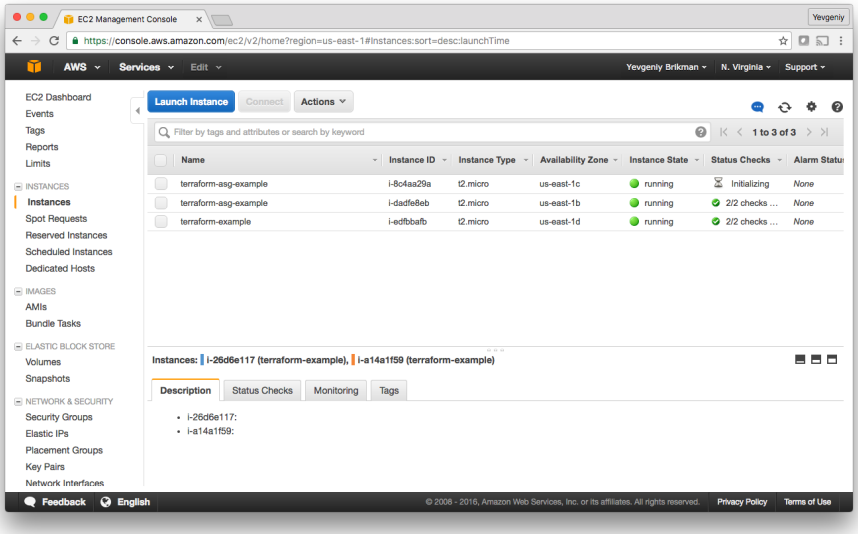



*Figure 2-12. The EC2 Instances in the ASG are launching*

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

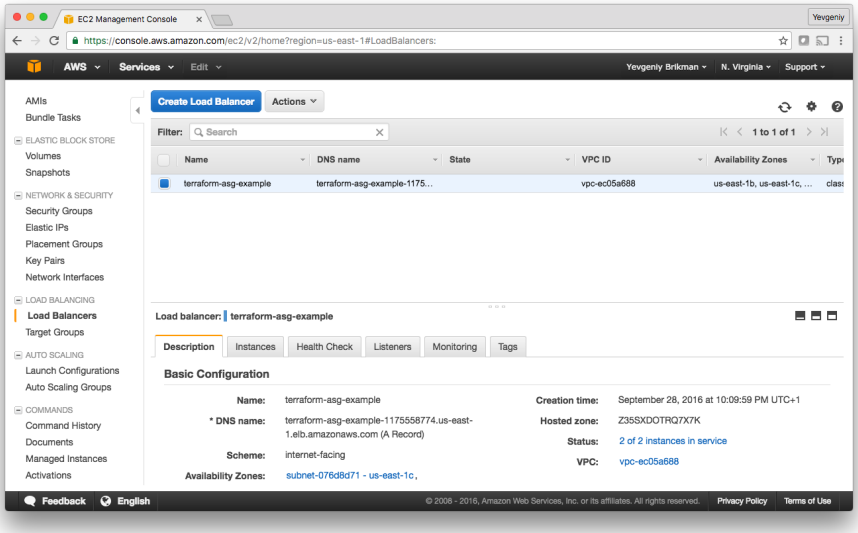



*Figure 2-13. The Elastic Load Balancer*

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

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

Success! The ELB is routing traffic to your EC2 Instances. Each time you hit the URL, it'll pick a different Instance to handle the request. You now have a fully working cluster of web servers!

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

## Clean up

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

```
> terraform destroy

Do you really want to destroy?
  Terraform will delete all your managed infrastructure.
  There is no undo. Only 'yes' will be accepted to confirm.

Enter a value:
```

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

Note that later in the book, you will continue to evolve this example, so don't delete the Terraform code! However, feel free to run `destroy` on the actual deployed resources. After all, the beauty of infrastructure as code is that all of the information about those resources is captured in code, so you can recreate all of them at any time with a single command: `terraform apply`.

## Conclusion

You now have a basic grasp of how to use Terraform. The declarative language makes it easy to describe exactly the infrastructure you want to create. The `plan` command allows you to verify your changes and catch bugs before deploying them. Variables, interpolation, and dependencies allow you to remove duplication from your code and make it highly configurable.

However, you've only scratched the surface. In Chapter 3, you'll learn how Terraform keeps track of what infrastructure it has already created, and the profound impact that has on how you should structure your Terraform code. In Chapter 4, you'll see how to create reusable infrastructure with Terraform modules.

CHAPTER 3

# How to manage Terraform state

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

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

1. What is Terraform state
2. Shared storage for state files
3. Locking state files
4. Isolating state files
5. File layout
6. Read-only state

**Example code**

As a reminder, all of the code examples in the book can be found at the following URL:

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

## What is Terraform state

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

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

After running `terraform apply`, here is a small snippet of the contents of the `terraform.tfstate` file:

```
{
  "aws_instance.example": {
    "type": "aws_instance",
    "primary": {
      "id": "i-66ba8957",
      "attributes": {
        "ami": "ami-40d28157",
        "availability_zone": "us-east-1d",
        "id": "i-66ba8957",
        "instance_state": "running",
        "instance_type": "t2.micro",
        "network_interface_id": "eni-7c4fcf6e",
        "private_dns": "ip-172-31-53-99.ec2.internal",
        "private_ip": "172.31.53.99",
        "public_dns": "ec2-54-159-88-79.compute-1.amazonaws.com",
        "public_ip": "54.159.88.79",
        "subnet_id": "subnet-3b29db10"
      }
    }
  }
}
```

Using this simple JSON format, Terraform knows that `aws_instance.example` corresponds to an EC2 Instance in your AWS account with ID `i-66ba8957`. 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 templates to determine what changes need to be applied.

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

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

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

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

In the following sections, I'll dive into each of these problems and show you how to solve them.

## Shared storage for state files

The most common technique for allowing multiple team members to access a common set of files is to put them in version control (e.g. Git). With Terraform state, this is a **bad idea** for two reasons:

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

*Secrets*
: All data in Terraform state files is stored in plaintext. 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 with no encryption whatsoever. Storing plaintext secrets *anywhere* is a bad idea, including version control. As of November, 2016, this is an open issue in the Terraform community, and there are only partial solutions available at this point, as I will discuss shortly.

Instead of using version control, the best way to manage shared storage for state files is to use Terraform's built-in support for *Remote State Storage*. Using the `terraform`

`remote config` command, you can configure Terraform to fetch and store state data from a remote store every time it runs. Several remote stores are supported, such as Amazon S3, Azure Storage, HashiCorp Consul, and HashiCorp Atlas.

I typically recommend Amazon S3 (Simple Storage Service), which is Amazon's managed file store, for the following reasons:

- It's a managed service, so you don't have to deploy and manage extra infrastructure to use it.
- It's designed for 99.999999999% durability and 99.99% availability, which effectively means it'll never lose your data or go down.[1]
- It supports encryption, which reduces worries about storing sensitive data in state files. Anyone on your team who has access to that S3 bucket will be able to see the state files in an unencrypted form, so this is still a partial solution, but at least the data will be encrypted at rest (S3 supports server-side encryption using AES-256) and in transit (Terraform uses SSL to read and write data in S3).
- It supports *versioning*, so every revision of your state file is stored, and you can always roll back to an older version if something goes wrong.
- It's inexpensive, with most Terraform usage easily fitting into the free tier.[2]

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

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

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

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

  versioning {
    enabled = true
  }

  lifecycle {
    prevent_destroy = true
  }
}
```

The code above sets three parameters:

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

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

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

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

*prevent_destroy*
: `prevent_destroy` is the second `lifecycle` setting you've seen (`create_before_destroy` was the first). 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 users from accidentally deleting an important resource, such as this S3 bucket, which will store all of your Terraform state. Of course, if you really meant to delete it, you can just comment that setting out.

Run `terraform plan`, and if everything looks OK, create the bucket by running `terraform apply`. After this completes, you will have an S3 bucket, but your Terraform state is still stored locally. To configure Terraform to store the state in your S3 bucket (with encryption), run the following command, filling in your own values where specified:

```
> terraform remote config \
    -backend=s3 \
    -backend-config="bucket=(YOUR_BUCKET_NAME)" \
    -backend-config="key=global/s3/terraform.tfstate" \
    -backend-config="region=us-east-1" \
    -backend-config="encrypt=true"

Remote configuration updated
Remote state configured and pulled.
```

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

3 See here for more information on S3 bucket names: *http://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html#bucketnamingrules*

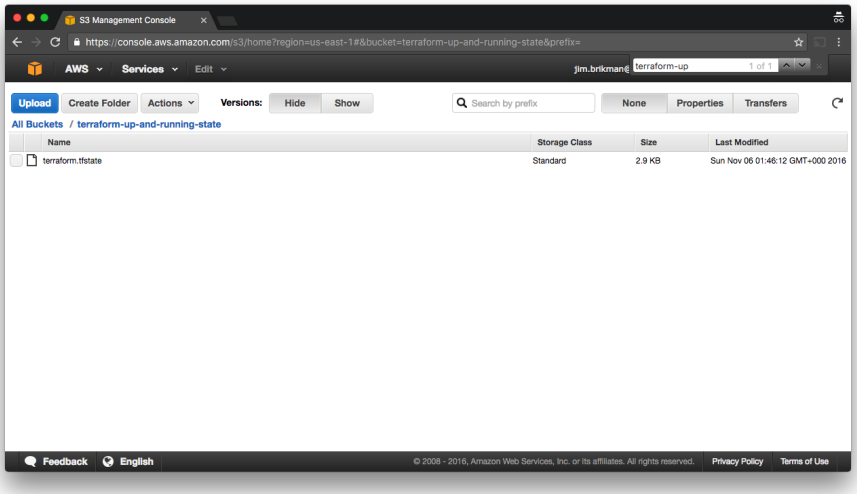



*Figure 3-1. Terraform state file stored in S3*

With remote state 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 variable:

```
output "s3_bucket_arn" {
  value = "${aws_s3_bucket.terraform_state.arn}"
}
```

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

```
> terraform apply

aws_s3_bucket.terraform_state: Refreshing state... (ID: terraform-up-and-running-state)

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

Outputs:

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

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

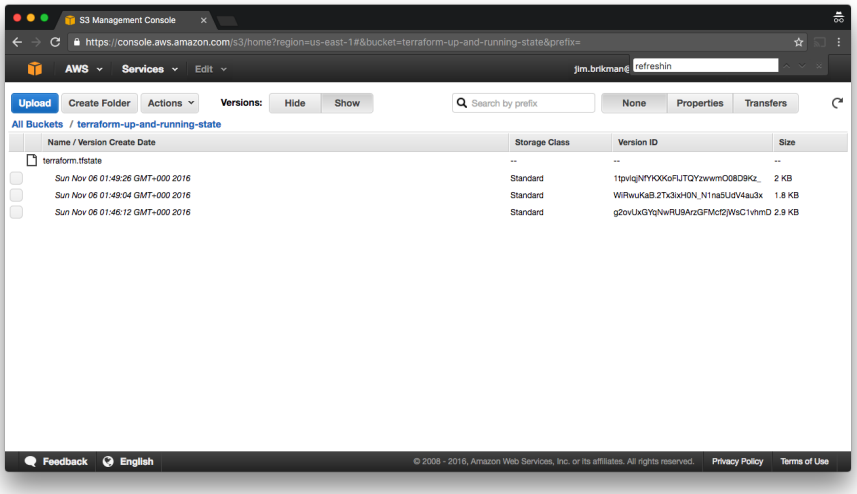



*Figure 3-2. Multiple versions of the Terraform state file in S3*

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

## Locking state files

Enabling remote state solves the problem of how you share state files with your teammates, but it creates two new problems:

1. Each developer on your team needs to remember to run the `terraform remote config` command for every Terraform project. It's easy to mess up or forget to run this long command.
2. While Terraform remote state storage ensures your state is stored in a shared location, it does **not** provide locking for that shared location (unless you are using HashiCorp's paid product Atlas for remote state storage). Therefore, race conditions are still possible if two developers are using Terraform at the same time on the same state files.

One way to solve both of these problems is to use an open source tool called Terragrunt. Terragrunt is a thin wrapper for Terraform that manages remote state for you automatically and provides locking by using Amazon DynamoDB. DynamoDB is also part of the AWS free tier, so using it for locking should be free for most teams.

Head over to the Terragrunt GitHub page and follow the instructions in the Readme to install the appropriate Terragrunt binary for your operating system. Next, create a file called `.terragrunt` in the same folder as the Terraform templates for your S3 bucket, and put the following code in it, filling in your own values where specified:

```
# Configure Terragrunt to use DynamoDB for locking
lock = {
  backend = "dynamodb"
  config {
    state_file_id = "global/s3"
  }
}

# Configure Terragrunt to automatically store tfstate files in S3
remote_state = {
  backend = "s3"
  config {
    encrypt = "true"
    bucket = "(YOUR_BUCKET_NAME)"
    key = "global/s3/terraform.tfstate"
    region = "us-east-1"
  }
}
```

The `.terragrunt` file uses the same language as Terraform, HCL. The first part of the configuration tells Terragrunt to use DynamoDB for locking. The `state_file_id` should be unique for each set of Terraform templates, so they each have their own lock. The second part of the configuration tells Terragrunt to use an S3 bucket for remote state storage using the exact same settings as the `terraform remote config` command you ran earlier.

Once you check this `.terragrunt` file into source control, everyone on your team can use Terragrunt to run all the standard Terraform commands:

```
> terragrunt plan
> terragrunt apply
> terragrunt output
> terragrunt destroy
```

Terragrunt forwards almost all commands, arguments, and options directly to Terraform, using whatever version of Terraform you already have installed. However, before running Terraform, Terragrunt will ensure your remote state is configured according to the settings in the `.terragrunt` file. Moreover, for any commands that could change your Terraform state (e.g. `apply` and `destroy`), Terragrunt will acquire and release a lock using DynamoDB.

Here's what it looks like in action:

```
> terragrunt apply
```

```
[terragrunt] Configuring remote state for the s3 backend
[terragrunt] Running command: terraform remote config
[terragrunt] Attempting to acquire lock in DynamoDB
[terragrunt] Attempting to create lock item table terragrunt_locks
[terragrunt] Lock acquired!
[terragrunt] Running command: terraform apply

terraform apply

aws_instance.example: Creating…
 ami: "" => "ami-0d729a60"
 instance_type: "" => "t2.micro"

(...)

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

[terragrunt] Attempting to release lock
[terragrunt] Lock released!
```

In the output above, you can see that Terragrunt automatically configured remote state as declared in the `.terragrunt` file, acquired a lock from DynamoDB, ran `terraform apply`, and then released the lock. If anyone else already had the lock, Terragrunt would have waited until the lock was released to prevent race conditions. Future developers need only checkout the repository containing this folder and run `terragrunt apply` to achieve an identical result!

## Isolating state files

With remote state storage and locking, collaboration is no longer a problem. However, there is still one more problem remaining: isolation. When you first start using Terraform, you may be tempted to define all of your infrastructure in a single Terraform file or a 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 each other, so if you are managing all the environments from a single set of Terraform templates, you are breaking that isolation. Just as a ship has bulkheads that act as bar-

4 For a colorful example of what happens when you don't isolate Terraform state, see: *https://charity.wtf/2016/03/30/terraform-vpc-and-why-you-want-a-tfstate-file-per-env/*

riers to prevent a leak in one part of the ship from immediately flooding all the others, you should have "bulkheads" built into your Terraform design, as shown in Figure 3-3.

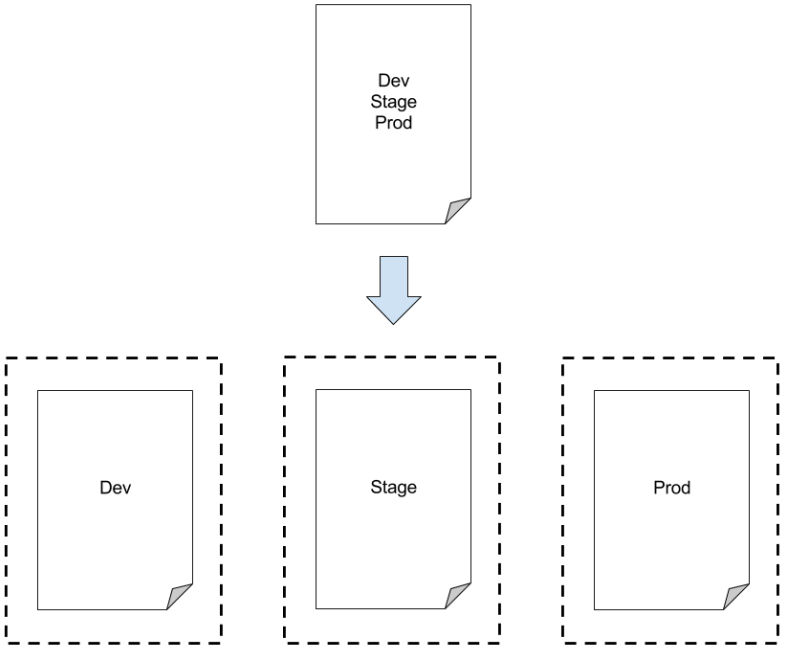



*Figure 3-3. Instead of defining all your environments in a single set of Terraform templates (top), you want to define each environment in a separate set of templates (bottom), so a problem in one environment is completely isolated from the others.*

The way to do that is to put the Terraform templates for each environment into a separate folder. For example, all the templates for the staging environment can be in a folder called `stage` and all the templates for the production environment can be in a folder called `prod`. That way, Terraform will use a separate state file for each environment, which makes it significantly less likely that a screw up in one environment can have any impact on another.

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

of a web server multiple times per day. If you manage the infrastructure for both the VPC component and the web server component in the same set of Terraform templates, you are unnecessarily putting your entire network topology at risk of breakage multiple times per day.

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

## File layout

Here is the file layout for my typical Terraform project:

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

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

- **stage**: An environment for non-production workloads (i.e. testing).
- **prod**: An environment for production workloads (i.e. user-facing apps).
- **mgmt**: An environment for DevOps tooling (e.g. bastion host, Jenkins).

- **global**: A place to put resources that are used across all environments, such as user management (e.g. S3, IAM).

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

- **vpc**: The network topology for this environment.
- **services**: The apps or microservices to run in this environment, such as a Ruby on Rails frontend or a Scala backend. Each app could even live in its own folder to isolate it from all the other apps.
- **data-storage**: The data stores to run in this environment, such as MySQL or Redis. Each data store could even live in its own folder to isolate it from all other data stores.

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

- **vars.tf**: Input variables.
- **outputs.tf**: Output variables.
- **main.tf**: The actual resources.
- **.terragrunt**: Locking and remote state configuration for Terragrunt.

When you run Terraform, it simply looks for files in the current directory with the `.tf` extension, so you can use whatever file names you want. Using a consistent convention like the one above makes your code easier to browse, since you always know where to look to find a variable, an output, or a resource. If your templates are becoming massive, it's OK to break out certain functionality into separate files (e.g. `iam.tf`, `s3.tf`, `database.tf`), but that may also be a sign that you should break your code up into smaller modules instead, a topic I'll dive into in Chapter 4.

Let's take the web server cluster code you wrote in Chapter 2, plus the S3 bucket code you wrote in this chapter, and rearrange it using the following folder structure:

```
stage
  └ services
      └ webserver-cluster
          └ vars.tf
          └ outputs.tf
          └ main.tf
          └ .terragrunt
global
  └ s3
    └ outputs.tf
    └ main.tf
    └ .terragrunt
```

The s3 bucket you created in this chapter should be moved into the `global/s3` folder. Note that you'll need to move the `s3_bucket_arn` output variable to `outputs.tf`. Make sure to copy the `.terragrunt` file and the `.terraform` folder (both of which are "hidden") to the new location too.

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 `vars.tf`, and output variables into `outputs.tf`. You should also give the web server cluster templates their own `.terragrunt` file in the `stage/services/webserver-cluster` folder:

```
# Configure Terragrunt to use DynamoDB for locking
lock = {
  backend = "dynamodb"
  config {
    state_file_id = "stage/services/webserver-cluster"
  }
}

# Configure Terragrunt to automatically store tfstate files in S3
remote_state = {
  backend = "s3"
  config {
    encrypt = "true"
    bucket = "(YOUR_BUCKET_NAME)"
    key = "stage/services/webserver-cluster/terraform.tfstate"
    region = "us-east-1"
  }
}
```

Notice that the `state_file_id` and `key` match the file path of the `.terragrunt` file (`state/services/webserver-cluster`) so that if you're browsing S3 or DynamoDB, there is a 1:1 mapping between the layout of your Terraform code, your state files, and your locks, which makes it easy to connect all the pieces together.

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

Of course, this very same property is, in some ways, a drawback too: splitting components into separate folders prevents you from breaking multiple components in one command, but it also prevents you from creating all the components in one command. If all of the components for a single environment were defined in a single Terraform template, you could spin up an entire environment with a single call to

`terraform apply`. But if all the components are in separate folders, then you need to run `terraform apply` separately in each one. If you create and tear down environments often, you'll want to create a script to make all these calls to `terraform apply` for you.

There is another problem with this file layout: it makes it harder to use resource dependencies. If your app code was defined in the same Terraform template as the database code, then that app could directly access attributes of the database (e.g. the database address and port) using Terraform's interpolation syntax (e.g. `${aws_db_instance.foo.address}`). But if if the app code and database code live in different folders, as recommended above, you can no longer do that. Fortunately, Terraform offers a solution: read-only state.

## Read-only state

In Chapter 2, you used data sources to fetch read-only information from AWS, such as the `aws_availability_zones` data source, which returns a list of availability zones in the current region. 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 templates in a completely read-only manner.

Let's go through an example. Imagine that your web server cluster needs to talk to a MySQL database. Running a database that is scalable, secure, durable, and highly available is a lot of work. Once again, you can let AWS take care of it for you, this time by using the *Relational Database Service (RDS)*, as shown in Figure 3-4. RDS supports a variety of databases, including MySQL, PostgreSQL, SQL Server, and Oracle.

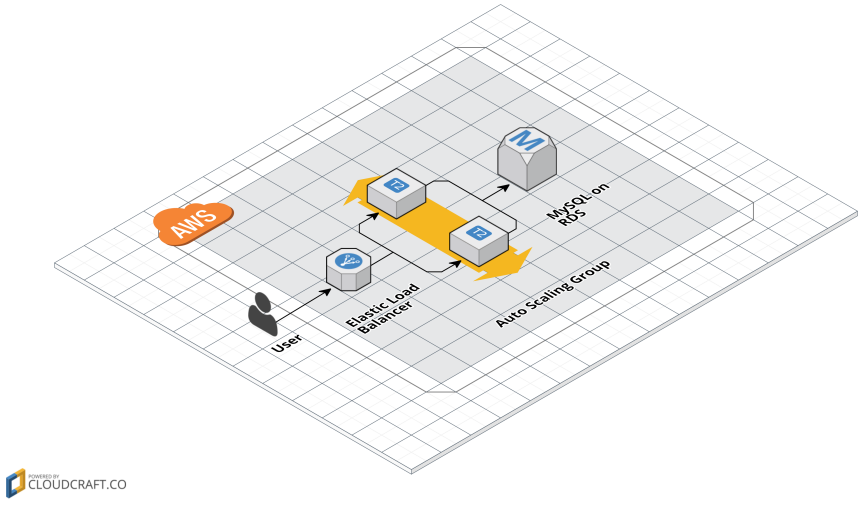



*Figure 3-4. The web server cluster talks to MySQL, which is deployed on top of Amazon's Relational Database Service*

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

```
stage
  └ services
      └ webserver-cluster
          └ vars.tf
          └ outputs.tf
          └ main.tf
          └ .terragrunt
  └ data-stores
      └ mysql
          └ vars.tf
          └ outputs.tf
          └ main.tf
          └ .terragrunt
global
  └ s3
    └ outputs.tf
    └ main.tf
    └ .terragrunt
```

Fill in the `.terragrunt` file in the `mysql` folder to configure locking and remote state storage for the database templates, matching the `state_file_id` and `key` parameters to the `stage/data-stores/mysql` folder path:

```
# Configure Terragrunt to use DynamoDB for locking
lock = {
  backend = "dynamodb"
  config {
    state_file_id = "stage/data-stores/mysql"
  }
}

# Configure Terragrunt to automatically store tfstate files in S3
remote_state = {
  backend = "s3"
  config {
    encrypt = "true"
    bucket = "(YOUR_BUCKET_NAME)"
    key = "stage/data-stores/mysql/terraform.tfstate"
    region = "us-east-1"
  }
}
```

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

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

resource "aws_db_instance" "example" {
  engine = "mysql"
  allocated_storage = 10
  instance_class = "db.t2.micro"
  name = "example_database"
  username = "admin"
  password = "${var.db_password}"
}
```

At the top of the file, you see the typical `provider` resource, but just below that is a new resource: `aws_db_instance`. This resource creates a database in RDS. The settings in the code above configure RDS to run MySQL with 10GB of storage on a `db.t2.micro` instance, which has 1 virtual CPU, 1GB of memory, and is part of the AWS free tier. Notice that in the code above, the `password` parameter is set to the `var.db_password` input variable, which you should declare in `stage/data-stores/mysql/vars.tf`:

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

Note that this variable does not have a `default`. This is intentional. You should not store your database password or any sensitive information in plaintext. Instead, you should store all secrets using a password manager that will encrypt your sensitive data (e.g. 1Password, LastPass, OS X Keychain) and expose those secrets to Terraform via environment variables. For each input variable `foo` defined in your Terraform templates, you can provide Terraform the value of this variable using the environment variable `TF_VAR_foo`. For the `var.db_password` input variable above, you can use the `TF_VAR_db_password` environment variable:

```
> export TF_VAR_db_password="(YOUR_DB_PASSWORD)"
> terragrunt plan
```

As a reminder, Terraform stores all variables in its state file in plain text, including this database password, which is why your `.terragrunt` file should always enable encryption for remote state storage in S3.

If the plan looks good, run `terragrunt apply` to create the database. Note that RDS can take as long as 10 minutes to provision even a small database, so be patient!

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

```
output "address" {
  value = "${aws_db_instance.example.address}"
}

output "port" {
  value = "${aws_db_instance.example.port}"
}
```

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

```
> terragrunt apply

(...)

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

Outputs:

address = tf-2016111123.cowu6mts6srx.us-east-1.rds.amazonaws.com
port = 3306
```

These outputs are now also stored in the remote state for the database, which, as configured by the `.terragrunt` file, is in your S3 bucket at the path `stage/data-stores/mysql/terraform.tfstate`. You can get the web server cluster code to read the data

from this state file by adding the `terraform_remote_state` data source in `stage/services/webserver-cluster/main.tf`:

```
data "terraform_remote_state" "db" {
  backend = "s3"
  config {
    bucket = "(YOUR_BUCKET_NAME)"
    key = "stage/data-stores/mysql/terraform.tfstate"
    region = "us-east-1"
  }
}
```

Whereas the database's `.terragrunt` file configured it to write its state to the `stage/data-stores/mysql/.terragrunt` folder in your S3 bucket, the `terraform_remote_state` data source above configures the web server cluster code to read that same state file from the same folder in your S3 bucket, as shown in Figure 3-5. 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.

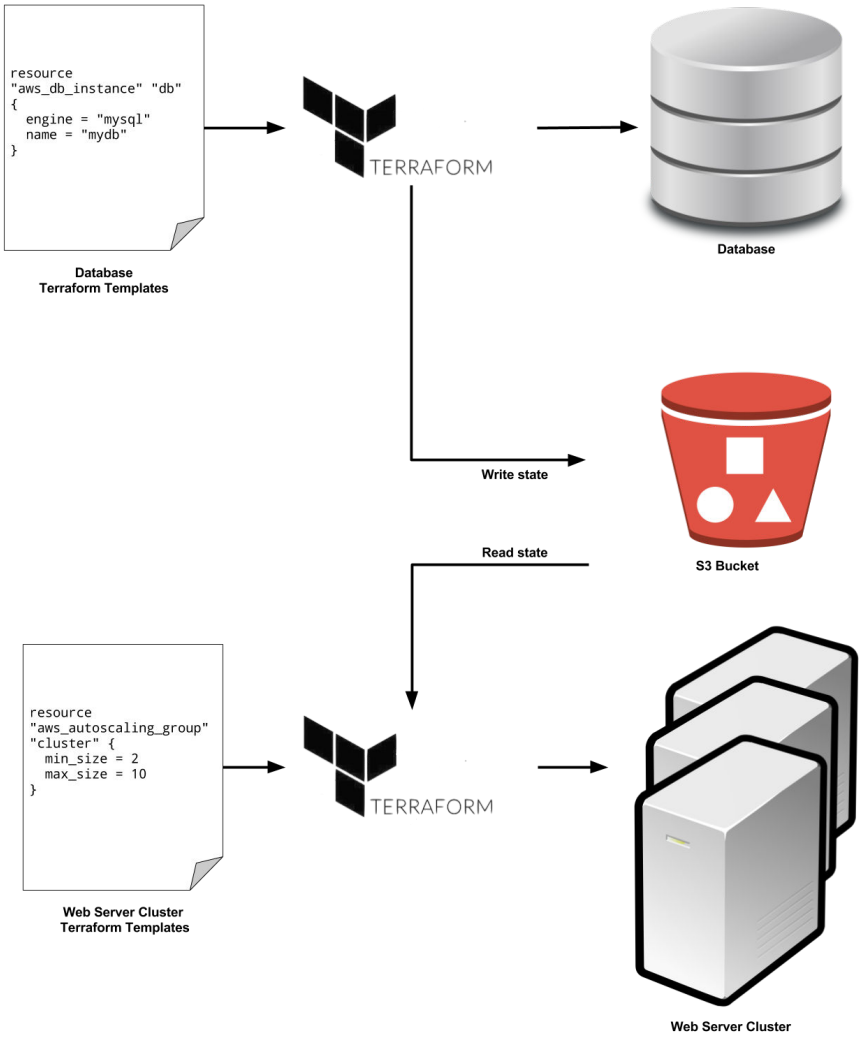



*Figure 3-5. The database writes its state to an S3 bucket (top) and the web server cluster reads that state from the same bucket (bottom)*

All the database's output variables are stored in the state file and you can access them on the `terraform_remote_state` data source using interpolation syntax, just as you would with any output 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 "${data.terraform_remote_state.db.address}" >> index.html
echo "${data.terraform_remote_state.db.port}" >> index.html
nohup busybox httpd -f -p "${var.server_port}" &
EOF
```

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

An *interpolation function* is a function you can use within Terraform's interpolation syntax. There are a number of built-in functions that can be used to manipulate strings, numbers, lists, and maps. One of them is the file interpolation function, which can be used to read a file from disk and return its contents as a string. For example, you could put your User Data script into stage/services/webserver-cluster/user-data.sh and load its contents into the user_data parameter of the aws_launch_configuration resource as follows:

```
user_data = "${file("user-data.sh")}"
```

The catch is that the User Data script for the web server cluster needs some dynamic data from Terraform, including the server port, database address, and database port. When the User Data script was embedded in the Terraform code, you used interpolation syntax to fill in these values. This does not work with the file interpolation function. However, it does work if you use a template_file data source.

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

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

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

```
  }
}
```

You can see that the code above sets the `template` parameter to the contents of the `user-data.sh` script and the `vars` parameter to the three variables the User Data script needs: the server port, database address, and database port. To use these variables, here's what the `stage/services/webserver-cluster/user-data.sh` script should look like:

```
#!/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 the Bash script above has a few changes from the original:

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

**A note on externalized files**

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

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

./user-data.sh

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

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

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

```
resource "aws_launch_configuration" "example" {
  image_id = "ami-40d28157"
  instance_type = "t2.micro"
  security_groups = ["${aws_security_group.instance.id}"]
  user_data = "${data.template_file.user_data.rendered}"

  lifecycle {
    create_before_destroy = true
  }
}
```

Ah, that's much cleaner than writing bash scripts in-line!

If you deploy this cluster using `terragrunt apply`, wait for the Instances to register in the ELB, and open the ELB URL in a web browser, you'll see something similar to Figure 3-6.

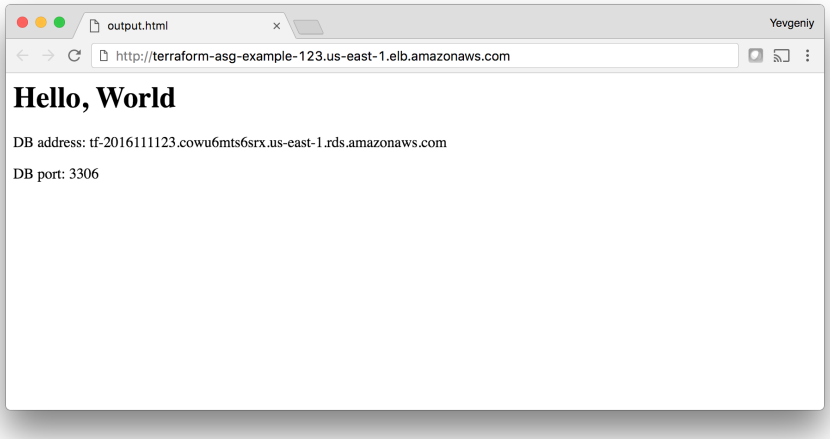



*Figure 3-6. The web server cluster can programmatically access the database address and port*

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

# Conclusion

The reason you need to put so much thought into isolation, locking, and state is that infrastructure as code (IAC) has different trade-offs than normal coding. When you're writing code for a typical app, most bugs are relatively minor and only break a small part of a single app. When you're writing code that controls your infrastructure, bugs tend to be more severe, as they can break all of your apps—and all of your data stores and your entire network topology and just about everything else. Therefore, I recommend including more "safety mechanisms" when working on IAC than with typical code.[5]

A common concern of using the recommended file layout is that it leads code duplication. If you want to run the web server cluster in both staging and production, how do you avoid having to copy & paste a lot of code between `stage/services/`

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

`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.

CHAPTER 4

# How to create reusable infrastructure with Terraform modules

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

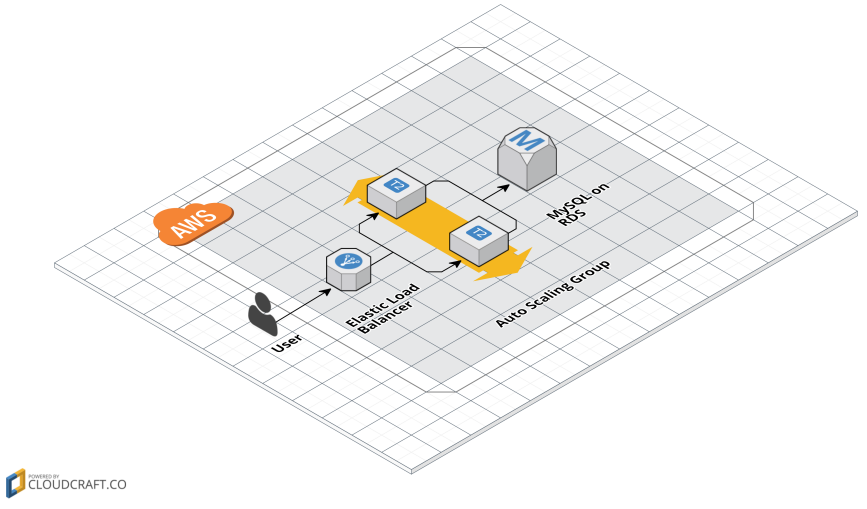



*Figure 4-1. A load balancer, web server cluster, and database*

This works great as a staging environment, but what about the production environment? You don't want to your users accessing the same environment your employees use for testing, and it's too risky testing in production, so you typically need two environments, staging and production, as shown in Figure 4-2. Ideally, the two environ-

ments are nearly identical, though you may run slightly fewer/smaller servers in staging to save money.

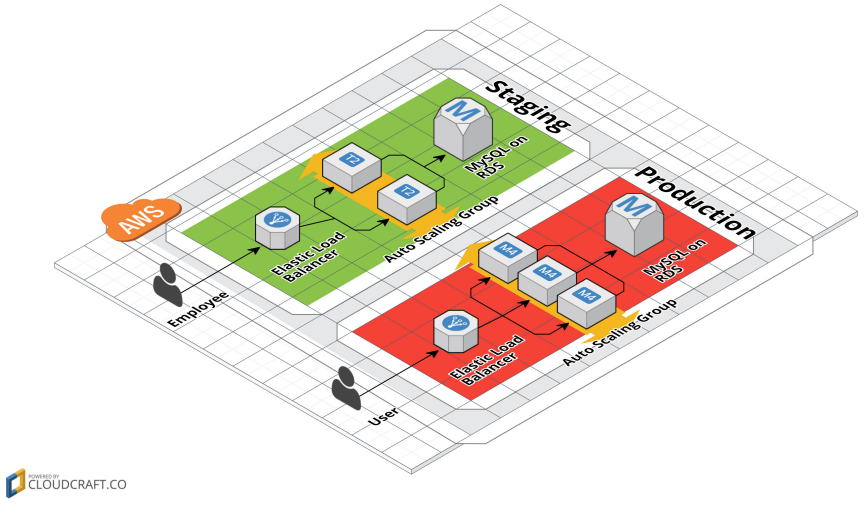



*Figure 4-2. Two environments, each with its own load balancer, web server cluster, and database*

With just a staging environment, the file layout for your Terraform code looked something like this:

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

If you were to add a production environment, you'd end up with the following file layout:

```
stage
  └ services
      └ webserver-cluster
          └ main.tf
          └ (etc)
```

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

How do you avoid duplication between the staging and production environments? How do you avoid having to copy and paste all the code in `stage/services/webserver-cluster` into `prod/services/webserver-cluster` and all the code in `stage/data-stores/mysql` into `prod/data-stores/mysql`?

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

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

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

With Terraform, you can put your code inside of a *Terraform module* and reuse that module in multiple places throughout your code. The `stage/services/webserver-cluster` into `prod/services/webserver-cluster` templates can both reuse code from the same module without the need to copy and paste:

```
stage                               modules
  └ services                          └ services
      └ webserver-cluster                 └ webserver-cluster
          └ main.tf ------------------> └ main.tf
          └ (etc)                     |   └ (etc)
  └ data-stores                       |
      └ mysql                         |
          └ main.tf                   |
          └ (etc)                     |
prod                                  |
  └ services                          |
      └ webserver-cluster             |
          └ main.tf ---------------/
```

```
            └ (etc)
    └ data-stores
        └ mysql
            └ main.tf
            └ (etc)
global
    └ s3
        └ main.tf
        └ (etc)
```

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

- Module basics
- Module inputs
- Module outputs
- Module gotchas
- Versioned modules

**Example code**

As a reminder, all of the code examples in the book can be found at the following URL:

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

# Module basics

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

As an example, let's turn the code in `stage/services/webserver-cluster`, which includes an Auto Scaling Group (ASG), Elastic Load Balancer (ELB), security groups, and many other resources, into a reusable module. As a first step, run `terraform destroy` in the `stage/services/webserver-cluster` to clean up any resources you created earlier. Next, create a new top-level folder called `modules` and move all the files from `stage/services/webserver-cluster` to `modules/services/webserver-cluster` *except* for the `.terragrunt` file, as that's going to be different for every environment. You should end up with a folder structure that looks something like this:

```
modules
  └ services
```

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

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

You can now make use of this module in the stage environment by creating a new `stage/services/webserver-cluster/main.tf` file with the following contents:

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

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

As usual, the code includes a `provider` block at the top, but there is also something new below that: the `module` block. The syntax for using a module is the keyword `module` followed by a unique identifier for the module that you can use throughout the Terraform code. Within the module definition, you must define the `source` parameter to specify the folder where the module's code can be found.

You can re-use the 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-1"
}

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

And there you have it: code re-use in multiple environments without any copy/paste! Note that whenever you add a module to your Terraform templates or modify its `source` parameter, you need to run the `get` command before you run `plan` or `apply`:

```
> terragrunt get
Get: /modules/frontend-app

> terragrunt plan

(...)
```

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

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

## Module inputs

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

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

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

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

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

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

variable "db_remote_state_key" {
```

```
  description = "The path for the database's remote state in S3"
}
```

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

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

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

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

Notice how the `name` parameter is set to `"${var.cluster_name}"`. You'll need to make a similar change to the other `aws_security_group` resource (e.g. give it the name `"${var.cluster_name}-instance"`), the `aws_elb` 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 data from the right database:

```
data "terraform_remote_state" "db" {
  backend = "s3"
  config {
    bucket = "${var.db_remote_state_bucket}"
    key = "${var.db_remote_state_key}"
    region = "us-east-1"
  }
}
```

Now, in the staging environment, you can set these new input variables accordingly:

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

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

You should do the same in the production environment:

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

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

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

As you can see, you set input variables for a module using the same syntax as setting input parameters for a resource. The input variables are the API of the module, controlling how it will behave in different environments. The example above uses different names in different environments, but you may want to make other parameters configurable too. For example, in staging, you 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/vars.tf`:

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

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

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

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-40d28157"
  instance_type = "${var.instance_type}"
  security_groups = ["${aws_security_group.instance.id}"]
  user_data = "${data.template_file.user_data.rendered}"

  lifecycle {
    create_before_destroy = true
  }
}
```

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

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

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

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

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

On the other hand, in the production environment, you can use a larger `instance_type` with more CPU and memory, such as m4.large (note: this instance type is NOT part of the AWS free tier, so if you're just using this for learning and don't want to be charged, use "t2.micro" for the `instance_type`), and you can set `max_size` to ten 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
}
```

How do you make the cluster shrink or grow in response to load? One option is to use an *auto scaling schedule*, which can change the size of the cluster at a scheduled time during the day. For example, if traffic to your cluster is much higher during normal business hours, you can use an auto scaling schedule to increase the number of servers at 9AM and decrease it at 5PM.

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

## Module outputs

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

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

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

The code above uses one `aws_autoscaling_schedule` resource to increase the number of servers to ten in the morning (the `recurrence` parameter uses cron syntax, so `"0 9 * * *"` means "9AM every day") and a second `aws_autoscaling_schedule` resource to decrease the number of servers at night (`"0 17 * * *"` means "5PM every day"). However, both usages of `aws_autoscaling_schedule` above are missing a required parameter, `autoscaling_group_name`, which specifies the name of the ASG. The ASG itself is defined within the `webserver-cluster` module, so how do

you access its name? In a general-purpose programming language, functions can return values:

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

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

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

```
output "asg_name" {
  value = "${aws_autoscaling_group.example.name}"
}
```

Now, in `prod/services/webserver-cluster/main.tf`, you can access that output using interpolation syntax and use it 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}"
}
```

Just as you can set module input variables the same way as resource parameters, you can also access module output variables the same way as resource output attributes. The syntax is `${module.MODULE_NAME.OUTPUT_NAME}` (e.g. `${module.frontend.asg_name}`).

You may want to expose one other output in the `webserver-cluster` module: the DNS name of the ELB, 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 "elb_dns_name" {
  value = "${aws_elb.example.dns_name}"
}
```

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

```
output "elb_dns_name" {
  value = "${module.webserver_cluster.elb_dns_name}"
}
```

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, there are a few gotchas to watch out for:

- File paths
- Inline blocks

## File paths

In Chapter 3, you moved the User Data script for the web server cluster into an external file, `user-data.sh`, and used the `file` interpolation function to read this file from disk. The catch with the `file` function is that the file path you use has to be relative (since you could run Terraform on many different computers)—but what is it relative to?

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

To solve this issue, you can use `path.module` to convert to a path that is relative to the module folder. Here is how the `template_file` data source should look in `modules/services/webserver-cluster/main.tf`:

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

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

```
  }
}
```

## Inline blocks

The configuration for some Terraform resources can be defined either as inline blocks or as separate resources. When creating a module, you should always prefer using a separate resource. For example, the `aws_security_group` resource allows you to define ingress and egress rules via inline blocks, as you saw in the `webserver-cluster` module (`modules/services/webserver-cluster/main.tf`):

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

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

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

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

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

resource "aws_security_group_rule" "allow_http_inbound" {
  type = "ingress"
  security_group_id = "${aws_security_group.elb.id}"

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

resource "aws_security_group_rule" "allow_all_outbound" {
  type = "egress"
  security_group_id = "${aws_security_group.elb.id}"
```

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

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

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

```
output "elb_security_group_id" {
  value = "${aws_security_group.elb.id}"
}
```

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

```
resource "aws_security_group_rule" "allow_testing_inbound" {
  type = "ingress"
  security_group_id = "${module.webserver_cluster.elb_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, the above code would not work. Note that this same type of problem affects a number of Terraform resources, such as:

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

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

**Network isolation**

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

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

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

# Versioned modules

If both your staging and production environment are pointing to the same module folder, then as soon as you make a change in that folder, it will affect both environments on the very next deployment. This sort of coupling makes it harder to test out a change in staging without any chance of affecting prod. 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 prod (e.g. v0.0.1):

```
stage                                modules
  └ services                           └ services
      └ webserver-cluster                  └ webserver-cluster
          └ main.tf ---------------\       └ main.tf
          └ (etc)                  |       └ (etc)
  └ data-stores                    |          |
      └ mysql                      |          |
          └ main.tf                \----->   \-- v0.0.2
          └ (etc)                            |
prod                                         |
  └ services                       /----->   \-- v0.0.1
      └ webserver-cluster          |
          └ main.tf ---------------/
          └ (etc)
  └ data-stores
      └ mysql
```

```
            └ main.tf
            └ (etc)
    global
      └ s3
        └ main.tf
        └ (etc)
```

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

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

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

The updated folder structure for your Terraform code will now look something like this:

```
    modules
      └ services
          └ webserver-cluster
    live
      └ stage
          └ services
              └ webserver-cluster
          └ data-stores
              └ mysql
      └ prod
          └ services
              └ webserver-cluster
          └ data-stores
              └ mysql
        global
          └ s3
```

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

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

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

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

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

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

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

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

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

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

```
source =
  "git@github.com:brikis98/terraform-up-and-running-code.git//
  code/terraform/04-terraform-module/module-example/modules/
  services/webserver-cluster?ref=v0.0.1"
```

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

A particularly useful naming scheme for tags is *semantic versioning*.[2] This is a versioning scheme of the format `MAJOR.MINOR.PATCH` (e.g. `1.0.4`) with specific rules on when you should increment each part of the version number. In particular, you should increment the…

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

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

Since you've updated your Terraform code to use a versioned module URL, you need to run `terraform get -update`:

```
> terragrunt get -update
Get: git::ssh://git@github.com/foo/modules.git?ref=v0.0.1

> terragrunt plan
(...)
```

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

**Private Git repos**

If your Terraform module is in a private Git repository, you will need to ensure the computer you're using has SSH keys configured correctly that allow Terraform to access that repository. In other words, before using the URL `ssh://git@github.com/foo/modules.git` in the `source` parameter of your module, make sure you can `git clone` that URL in your terminal:

```
> git clone ssh://git@github.com/foo/modules.git
```

If that command fails, you need to set up your SSH keys first. GitHub has excellent documentation on how to do that: *https://help.github.com/articles/generating-an-ssh-key/*

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

2 *http://semver.org/*

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

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

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

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

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

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

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

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

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

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

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

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

**Developing modules**

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

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

# Conclusion

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

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

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

CHAPTER 5

# Terraform tips & tricks: loops, if-statements, deployment, and gotchas

Terraform is a declarative language. As discussed in Chapter 1, infrastructure as code in a declarative language tends to provides 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 code base small. However, without access to a full programming language, certain types of tasks become more difficult in a declarative language.

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

Fortunately, Terraform provides a few primitives—namely, a meta-parameter called `count`, a lifecycle block called `create_before_destroy`, plus a large number of interpolation functions—that allow you to do certain types of loops, if-statements, and zero-downtime deployments. You probably won't need to use these too often, but when you do, it's good to be aware of what's possible and what the gotchas are. Here are the topics I'll cover in this chapter:

- Loops
- If-statements
- If-else-statements
- Zero-downtime deployment

- Terraform Gotchas

**Example code**

As a reminder, all of the code examples in the book can be found at the following URL:

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

# Loops

In Chapter 2, you created an IAM user by clicking around the AWS console. Now that you have this user, you can create and manage all future IAM users with Terraform. Consider the following Terraform code, which should live in `live/global/iam/main.tf`:

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

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

This code uses the `aws_iam_user` resource to create a single new IAM user. What if you wanted to create three IAM users? In a normal 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 the syntax above will not work. However, almost every Terraform resource has a meta-parameter you can use called `count`. This parameter defines how many copies of the resource to create. Therefore, you can create three IAM users as follows:

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

One problem with the code above is that all three IAM users would have the same name, which would cause an error, since user names 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 code above, you will see that Terraform wants to create three IAM users, each one with a different name ("neo.0", "neo.1", "neo.2"):

```
+ aws_iam_user.example.0
    arn:           "<computed>"
    force_destroy: "false"
    name:          "neo.0"
    path:          "/"
    unique_id:     "<computed>"

+ aws_iam_user.example.1
    arn:           "<computed>"
    force_destroy: "false"
    name:          "neo.1"
    path:          "/"
    unique_id:     "<computed>"

+ aws_iam_user.example.2
    arn:           "<computed>"
    force_destroy: "false"
    name:          "neo.2"
    path:          "/"
    unique_id:     "<computed>"


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

Of course, a user name like "neo.0" isn't particularly usable. If you combine `count.index` with some interpolation functions built into Terraform, you can customize each "iteration" of the "loop" even more. You saw the `file` interpolation func-

tion in Chapter 3. Terraform has many other built-in interpolation functions that can be used to manipulate strings, numbers, lists, and maps.[1]

For example, you could define all of the IAM user names you want in an input variable in `live/global/iam/vars.tf`:

```
variable "user_names" {
  description = "Create IAM users with these names"
  type = "list"
  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.index` and two interpolation functions: `element` and `length`. The `element(list, index)` function returns the item located at `index` in the given `list`.[2] The `length(list)` function returns the number of items in `list`. Putting these together, you get:

```
resource "aws_iam_user" "example" {
  count = "${length(var.user_names)}"
  name = "${element(var.user_names, count.index)}"
}
```

Now when you run the `plan` command, you'll see that Terraform wants to create three IAM users, each with a unique name:

```
+ aws_iam_user.example.0
    arn:           "<computed>"
    force_destroy: "false"
    name:          "neo"
    path:          "/"
    unique_id:     "<computed>"

+ aws_iam_user.example.1
    arn:           "<computed>"
```

1 You can find the full list of interpolation functions here: *https://www.terraform.io/docs/configuration/interpolation.html*

2 If the `index` is greater than the number of elements in `list`, the `element` function will automatically "wrap" around using a standard mod function.

```
    force_destroy: "false"
    name:          "trinity"
    path:          "/"
    unique_id:     "<computed>"

+ aws_iam_user.example.2
    arn:           "<computed>"
    force_destroy: "false"
    name:          "morpheus"
    path:          "/"
    unique_id:     "<computed>"


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

Note that once you've used `count` on a resource, it becomes a list of resources, rather than just one resource. For example, if you wanted to provide the Amazon Resource Name (ARN) of one of the IAM users as an output variable, you would need to do the following:

```
output "neo_arn" {
  value = "${aws_iam_user.example.0.arn}"
}
```

Since `aws_iam_user.example` is now a list of IAM users, instead of using the standard syntax to read an attribute from that resource (`TYPE.NAME.ATTRIBUTE`), you have to specify which IAM user you're interested in by specifying its index in the list (`TYPE.NAME.INDEX.ATTRIBUTE`). If you want the ARNs of *all* the IAM users, you need to use the *splat* syntax (`TYPE.NAME.*.ATTRIBUTE`) and wrap your output variable in brackets to indicate it's a list:

```
output "all_arns" {
  value = ["${aws_iam_user.example.*.arn}"]
}
```

When you run the `apply` command, the `all_arns` output will contain the list of ARNs:

```
> terragrunt 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
]
```

Note that since the splat syntax returns a list, you can combine it with other interpolation functions, such as `element`. For example, let's say you wanted to give each of these IAM users read-only access to EC2. You may remember from Chapter 2 that by default, new IAM users have no permissions whatsoever, and that to grant permissions, you can attach IAM policies to those IAM users. An IAM policy is a JSON document:

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["ec2:Describe*"],
      "Resource": ["*"]
    }
  ]
}
```

An IAM policy consists of one or more *statements*, each of which specifies an *effect* (either "Allow" or "Deny"), on one or more *actions* (e.g. "ec2:Describe*" allows all API calls to EC2 that start with the name "Describe"), on one or more *resources* (e.g. "*" means "all resources"). Although you can define IAM policies in JSON, Terraform also provides a handy data source called the `aws_iam_policy_document` which gives you a more concise way to define the same IAM policy:

```
data "aws_iam_policy_document" "ec2_read_only" {
  statement {
    effect = "Allow"
    actions = ["ec2:Describe*"]
    resources = ["*"]
  }
}
```

To create a new managed IAM policy from this document, you need to use the `aws_iam_policy` resource and set its `policy` parameter to the JSON output of the `aws_iam_policy_document` you created above:

```
resource "aws_iam_policy" "ec2_read_only" {
  name = "ec2-read-only"
  policy = "${data.aws_iam_policy_document.ec2_read_only.json}"
}
```

Finally, to attach the IAM policy to your new IAM users, you use the `aws_iam_user_policy_attachment` resource:

```
resource "aws_iam_user_policy_attachment" "ec2_access" {
  count = "${length(var.user_names)}"
  user = "${element(aws_iam_user.example.*.name, count.index)}"
  policy_arn = "${aws_iam_policy.ec2_read_only.arn}"
}
```

The code above uses the `count` parameter to "loop" over each of your IAM users and the `element` interpolation function to select each user's ARN from the list returned by `aws_iam_user.example.*.arn`.

# If-statements

Using `count` lets you do a basic loop. If you're clever, you can use the same mechanism to do a basic if-statement as well. Let's start by looking at simple if-statements in the next section and then move on to more complicated ones in the section after that.

## Simple if-statements

In Chapter 4, you created a Terraform module that could be used as "blueprint" for deploying web server clusters. The module created an Auto Scaling Group (ASG), Elastic Load Balancer (ELB), security groups, and a number of other resources. One thing the module did *not* create was the auto scaling schedule. Since you only want to scale the cluster out in production, you defined the `aws_autoscaling_schedule` resources directly in the production templates 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/vars.tf` that can be used to specify whether the module should enable auto scaling:

```
variable "enable_autoscaling" {
  description = "If set to true, enable auto scaling"
}
```

Now, if you had a general-purpose programming language, you could use this input variable in an if-statement:

```
# This is just pseudo code. It won't actually work in Terraform.
if ${var.enable_autoscaling} {
  resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
    scheduled_action_name = "scale-out-during-business-hours"
    min_size = 2
    max_size = 10
    desired_capacity = 10
    recurrence = "0 9 * * *"
    autoscaling_group_name = "${aws_autoscaling_group.example.name}"
  }

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

```
    desired_capacity = 2
    recurrence = "0 17 * * *"
    autoscaling_group_name = "${aws_autoscaling_group.example.name}"
  }
}
```

Terraform doesn't support if-statements, so the code above won't work. However, you can accomplish the same thing by using the `count` parameter and taking advantage of two properties:

1. In Terraform, if you set a variable to a boolean `true` (that is, the word `true` without any quotes around it), it will be converted to a 1 and if you set it to a boolean `false`, it will be converted to a 0.
2. If you set `count` to 1 on a resource, you get one copy of that resource and if you set `count` to 0, that resource is not created at all.

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}"

  scheduled_action_name = "scale-out-during-business-hours"
  min_size = 2
  max_size = 10
  desired_capacity = 10
  recurrence = "0 9 * * *"
  autoscaling_group_name = "${aws_autoscaling_group.example.name}"
}

resource "aws_autoscaling_schedule" "scale_in_at_night" {
  count = "${var.enable_autoscaling}"

  scheduled_action_name = "scale-in-at-night"
  min_size = 2
  max_size = 10
  desired_capacity = 2
  recurrence = "0 17 * * *"
  autoscaling_group_name = "${aws_autoscaling_group.example.name}"
}
```

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 the `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 the `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
}
```

## More complicated if-statements

The approach above works well if the user passes an explicit boolean value to your module, but what do you do if the boolean is the result of a more complicated conditional, such as string equality? Unfortunately, there is no general purpose answer to this, as Terraform doesn't have any official ways to do boolean comparisons. However, in certain cases, if you get creative with some of the built-in interpolation functions, you can make things work.

Let's go through an example. Imagine that as part of the `webserver-cluster` module, you wanted to create a set of CloudWatch alarms. A *CloudWatch alarm* can be configured to notify you via a variety of mechanisms (e.g. email, text message) if a specific

metric exceeds a pre-defined threshold. For example, here is how you could use the `aws_cloudwatch_metric_alarm` resource in `modules/services/webserver-cluster/main.tf` to create an alarm that goes off if the average CPU utilization in the cluster is over 90% during a 5 minute period:

```
resource "aws_cloudwatch_metric_alarm" "high_cpu_utilization" {
  alarm_name = "${var.cluster_name}-high-cpu-utilization"
  namespace = "AWS/EC2"
  metric_name = "CPUUtilization"
  dimensions = {
    AutoScalingGroupName = "${aws_autoscaling_group.example.name}"
  }
  comparison_operator = "GreaterThanThreshold"
  evaluation_periods = 1
  period = 300
  statistic = "Average"
  threshold = 90
  unit = "Percent"
}
```

This works fine for a CPU Utilization alarm, but what if you wanted to add another alarm that goes off when CPU Credits are low? Certain EC2 Instances are configured with a baseline level of CPU performance, plus the ability to burst above the baseline level. The amount of time that you can burst is governed by *CPU credits*. A single CPU credit provides the performance of a full CPU core for one minute. Each EC2 Instance starts with a certain number of CPU credits, accumulates more credits when it's idle, and uses up credits when it's doing work. If the EC2 Instance runs out of CPU credits, it will only be able to use the baseline CPU level, which can be relatively low. Therefore, you may want to add a CloudWatch alarm that goes off if your web server cluster is almost out of CPU credits:

```
resource "aws_cloudwatch_metric_alarm" "low_cpu_credit_balance" {
  alarm_name = "${var.cluster_name}-low-cpu-credit-balance"
  namespace = "AWS/EC2"
  metric_name = "CPUCreditBalance"
  dimensions = {
    AutoScalingGroupName = "${aws_autoscaling_group.example.name}"
  }
  comparison_operator = "LessThanThreshold"
  evaluation_periods = 1
  period = 300
  statistic = "Minimum"
  threshold = 10
  unit = "Count"
}
```

The catch is that CPU Credits only apply to tXXX Instances (e.g. t2.micro, t2.medium, etc). Larger instance types (e.g. m4.large) have access to full CPU cores and don't have to worry about baseline vs. burst performance. Is there a way to create the alarm above only if `var.instance_type` starts with the letter "t"?

You could add a new boolean input variable called `var.is_t2_instance`, but that would be redundant with `var.instance_type`, and you'd most likely forget to update one when updating the other. An alternative is to do a string comparison by using the `replace` interpolation function.

`replace(string, search, replacement)` replaces all instances of `search` in `string` with `replacement`. For example, `replace("hello world", "l", "")` would return "heo word". Note that if `search` is wrapped in forward slashes, it is treated as a regular expression, so `replace("hello world", "/[a-h]/", "")` would return "llo worl".

You can use (or rather, abuse) the `replace` function to do a string comparison as follows:

```
resource "aws_cloudwatch_metric_alarm" "low_cpu_credit_balance" {
  count = "${replace(replace(var.instance_type, "/^[^t].*/", "0"),
             "/^t.*$/", "1")}"

  alarm_name = "${var.cluster_name}-low-cpu-credit-balance"
  namespace = "AWS/EC2"
  metric_name = "CPUCreditBalance"
  dimensions = {
    AutoScalingGroupName = "${aws_autoscaling_group.example.name}"
  }
  comparison_operator = "LessThanThreshold"
  evaluation_periods = 1
  period = 300
  statistic = "Minimum"
  threshold = 10
  unit = "Count"
}
```

The alarm code is the same as before, except for the relatively complicated `count` parameter that makes nested calls to the `replace` function. To understand how it works, start with the inner call to `replace`:

```
replace(var.instance_type, "/^[^t].*/", "0")
```

This function will replace `var.instance_type` with 0 if `var.instance_type` does not start with the letter "t". Now look at the outer call to `replace`:

```
replace(<INNER>, "/^t.*$/", "1")
```

This statement takes whatever string the inner statement returned, which will be either a 0 or something that starts with "t", and if that string starts with "t", it replaces it with a 1. In other words, these two replace statements produce a 1 for instance types that start with "t" and a 0 otherwise, ensuring the alarm is only created for instance types that actually have a CPUCreditBalance metric.

# If-else-statements

Now that you know how to do an if-statement, what about an if-else-statement? Let's again start by looking at simple if-else-statements in the next section and move on to more complicated ones in the section after that.

## Simple if-else-statements

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. However, whether neo got only read or both read and write access would be determined by the person applying the Terraform templates. This is a slightly contrived example, but it makes it easy to demonstrate a simple type of if-else-statement, where all that matters is that one of the if or else branches gets executed, and the rest of the Terraform code doesn't need to know which one.

Here is an IAM policy that allows read-only access to CloudWatch:

```
resource "aws_iam_policy" "cloudwatch_read_only" {
  name = "cloudwatch-read-only"
  policy = "${data.aws_iam_policy_document.cloudwatch_read_only.json}"
}

data "aws_iam_policy_document" "cloudwatch_read_only" {
  statement {
    effect = "Allow"
    actions = ["cloudwatch:Describe*", "cloudwatch:Get*", "cloudwatch:List*"]
    resources = ["*"]
  }
}
```

And here is an IAM policy that allows full (read and write) access to CloudWatch:

```
resource "aws_iam_policy" "cloudwatch_full_access" {
  name = "cloudwatch-full-access"
  policy = "${data.aws_iam_policy_document.cloudwatch_full_access.json}"
}

data "aws_iam_policy_document" "cloudwatch_full_access" {
  statement {
    effect = "Allow"
    actions = ["cloudwatch:*"]
    resources = ["*"]
  }
}
```

The goal is to attach one of these IAM policies to neo, based on the value of a new input variable called `give_neo_cloudwatch_full_access`:

```
variable "give_neo_cloudwatch_full_access" {
  description = "If true, neo gets full access to CloudWatch"
}
```

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 again use the `count` parameter and a boolean, but this time, you also need to take advantage of the fact that Terraform allows simple math in interpolations:

```
resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full_access" {
  count = "${var.give_neo_cloudwatch_full_access}"

  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 = "${1 - var.give_neo_cloudwatch_full_access}"

  user = "${aws_iam_user.example.0.name}"
  policy_arn = "${aws_iam_policy.cloudwatch_read_only.arn}"
}
```

The code above contains two `aws_iam_user_policy_attachment` resources. The first one, which attaches the CloudWatch full access permissions, sets its `count` parameter to `var.give_neo_cloudwatch_full_access`, so this resource only gets created if `var.give_neo_cloudwatch_full_access` is `true` (this is the if-clause). The second one, which attaches the CloudWatch read-only permissions, sets its `count` parameter to `1 - var.give_neo_cloudwatch_full_access`, so it will have the inverse behavior, and only be created if `var.give_neo_cloudwatch_full_access` is `false` (this is the else-clause).

## More complicated if-else-statements

The approach above works well if your Terraform code doesn't need to know which of the if or else clauses actually got executed. But what if you need to access some output attribute on the resource that comes out of the if or else clause? For example, what if you wanted to offer two different User Data scripts in the `webserver-cluster` module and allow users to pick which one gets executed? Currently, the `webserver-cluster` module pulls in the `user-data.sh` script via a `template_file` data source:

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

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

The current `user-data.sh` script looks like this:

```
#!/bin/bash

cat > index.html <<EOF
<h1>Hello, World</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

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

Now, imagine that you wanted to allow some of your web server clusters to use this alternative, shorter script, called `user-data-new.sh`:

```
#!/bin/bash

echo "Hello, World, v2" > index.html
nohup busybox httpd -f -p "${server_port}" &
```

To use this script, you need a new `template_file` data source:

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

  vars {
    server_port = "${var.server_port}"
  }
}
```

The question is, how can you allow the user of the `webserver-cluster` module to pick from one of these User Data scripts? As a first step, you could add a new boolean input variable in `modules/services/webserver-cluster/vars.tf`:

```
variable "enable_new_user_data" {
  description = "If set to true, use the new User Data script"
}
```

If you were using a general-purpose programming language, you could add an if-else-statement to the launch configuration to pick between the two User Data `template_file` options as follows:

```
# This is just pseudo code. It won't actually work in Terraform.
resource "aws_launch_configuration" "example" {
  image_id = "ami-40d28157"
  instance_type = "${var.instance_type}"
  security_groups = ["${aws_security_group.instance.id}"]

  if ${var.enable_new_user_data} {
    user_data = "${data.template_file.user_data_new.rendered}"
  } else {
    user_data = "${data.template_file.user_data.rendered}"
  }

  lifecycle {
    create_before_destroy = true
  }
}
```

To make this work with real Terraform code, you first need to use the if-else-statement trick from before to ensure that only one of the `template_file` data sources is actually created:

```
data "template_file" "user_data" {
  count = "${1 - var.enable_new_user_data}"

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

  vars {
    server_port = "${var.server_port}"
    db_address = "${data.terraform_remote_state.db.address}"
    db_port = "${data.terraform_remote_state.db.port}"
  }
}

data "template_file" "user_data_new" {
  count = "${var.enable_new_user_data}"

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

  vars {
    server_port = "${var.server_port}"
  }
}
```

If `var.enable_new_user_data` is `true`, then `data.template_file.user_data_new` will be created and `data.template_file.user_data` will not; if it's `false`, it'll be the other way around. All you have to do now is to set the `user_data` parameter of the `aws_launch_configuration` resource to the `template_file` that actually exists. To do this, you can take advantage of the `element` interpolation function, which you already know, plus the `concat(list1, list2, ...)` interpolation function, which combines two or more lists into a single list:

```
resource "aws_launch_configuration" "example" {
  image_id = "ami-40d28157"
  instance_type = "${var.instance_type}"
  security_groups = ["${aws_security_group.instance.id}"]

  user_data = "${element(
    concat(data.template_file.user_data.*.rendered,
           data.template_file.user_data_new.*.rendered),
    0)}"

  lifecycle {
    create_before_destroy = true
  }
}
```

Let's break the large value for the `user_data` parameter down. First, take a look at the inner part:

```
concat(data.template_file.user_data.*.rendered,
       data.template_file.user_data_new.*.rendered)
```

Note that the two `template_file` resources are both lists, as they both use the `count` parameter. One of these lists will be of length 1 and the other of length 0, depending on the value of `var.enable_new_user_data`. The code above uses the `concat` function to combine these two lists into a single list, which will be of length 1. Now consider the outer part:

```
user_data = "${element(<INNER>, 0)}"
```

This code simply takes the list returned by the inner part, which will be of length 1, and uses the `element` function to extract that one value.

You can now try out the new User Data script in the staging environment by setting the `enable_new_user_data` parameter to true in `live/stage/services/webserver-cluster/main.tf`:

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

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

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

  enable_autoscaling = false
  enable_new_user_data = true
}
```

On the other hand, in the production environment, you can stick with the old version of the script by setting `enable_new_user_data` to false in `live/prod/services/webserver-cluster/main.tf`:

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

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

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

  enable_autoscaling = true
  enable_new_user_data = false
}
```

Using `count` and interpolation 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 above, it allows you to conceal lots of complexity from your users so that they get to work with a clean and simple API.

## Zero-downtime deployment

Now that your module has a clean and simple API for deploying a web server cluster, an important question to ask is, how do you update that cluster? That is, when you have made changes to your code, how do you deploy a new AMI across the cluster? And how do you do it without causing downtime for your users?

The first step is to expose the AMI as an input variable in `modules/services/webserver-cluster/vars.tf`. In real-world examples, this is all you would need, as the actual web server code would be defined in the AMI. However, in the simplified examples in this book, all of the web server code is actually in the User Data script, and the AMI is just a vanilla Ubuntu image. Switching to a different version of Ubuntu won't make for much of a demonstration, so in addition to the new AMI input variable, you can also add an input variable to control the text the User Data script returns from its one-liner HTTP server:

```
variable "ami" {
  description = "The AMI to run in the cluster"
  default = "ami-40d28157"
}

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

Earlier in the chapter, to practice with if-else statements, you created two User Data scripts. Let's consolidate that back down to one to keep things simple. First, in `modules/services/webserver-cluster/vars.tf`, remove the `enable_new_user_data` input variable. Second, in `modules/services/webserver-cluster/main.tf`, remove the `template_file` resource called `user_data_new`. Third, in the same file, update the other `template_file` resource, called `user_data`, to no longer use the `enable_new_user_data` input variable, and to add the new `server_text` input variable to its `vars` block:

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

  vars {
    server_port = "${var.server_port}"
    db_address = "${data.terraform_remote_state.db.address}"
    db_port = "${data.terraform_remote_state.db.port}"
    server_text = "${var.server_text}"
  }
}
```

Now you need to update the `modules/services/webserver-cluster/user-data.sh` bash script to use this `server_text` variable in the `<h1>` tag it returns:

```
#!/bin/bash

cat > index.html <<EOF
<h1>${server_text}</h1>
<p>DB address: ${db_address}</p>
<p>DB port: ${db_port}</p>
EOF

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

Finally, find the launch configuration in `modules/services/webserver-cluster/main.tf`, set its `user_data` parameter to the remaining `template_file` (the one called `user_data`), and set its `ami` parameter to the new `ami` input variable:

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

```
  user_data = "${data.template_file.user_data.rendered}"

  lifecycle {
    create_before_destroy = true
  }
}
```

Now, in the staging environment, in `live/stage/services/webserver-cluster/main.tf`, you can set the new `ami` and `server_text` parameters and remove the `enable_new_user_data` parameter:

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

  ami = "ami-40d28157"
  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"

  cluster_name = "webservers-stage"
  db_remote_state_bucket = "${var.db_remote_state_bucket}"
  db_remote_state_key = "${var.db_remote_state_key}"

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

  enable_autoscaling = false
}
```

The code above uses the same Ubuntu AMI, but changes the `server_text` to a new value. If you run the `plan` command, you should see something like the following (I've omitted some of the output for clarity):

```
~ module.webserver_cluster.aws_autoscaling_group.example
    launch_configuration:
      "terraform-2016182624wu" => "${aws_launch_configuration.example.id}"

-/+ module.webserver_cluster.aws_launch_configuration.example
    ebs_block_device.#:  "0" => "<computed>"
    ebs_optimized:       "false" => "<computed>"
    enable_monitoring:   "true" => "true"
    image_id:            "ami-40d28157" => "ami-40d28157"
    instance_type:       "t2.micro" => "t2.micro"
    key_name:            "" => "<computed>"
    name:                "terraform-2016wu" => "<computed>"
    root_block_device.#: "0" => "<computed>"
    security_groups.#:   "1" => "1"
    user_data:           "416115339b" => "3bab6ede8dc" (forces new resource)
```

```
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 to reference the new launch configuration. The problem is that merely referencing the new launch configuration will have no effect until the Auto Scaling Group launches new EC2 Instances. So how do you tell the Auto Scaling Group to deploy new Instances?

One option is to destroy the ASG (e.g. by running `terraform destroy`) and then recreate 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, this is exactly what the `create_before_destroy` lifecycle setting does!

Here's how you can take advantage of this lifecycle setting to get a zero-downtime deployment:[3]

1. Configure the `name` parameter of the ASG to depend directly on the name of the launch configuration. That way, each time the launch configuration changes (which it will when you update the AMI or User Data), Terraform will try to replace the ASG.
2. Set the `create_before_destroy` parameter of the ASG to `true`, so each time Terraform tries to replace it, it will create the replacement 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 register in the ELB before it'll start destroying the original ASG.

Here is what the updated `aws_autoscaling_group` resource should look like in `modules/services/webserver-cluster/main.tf`:

```
resource "aws_autoscaling_group" "example" {
  name = "${var.cluster_name}-${aws_launch_configuration.example.name}"
  launch_configuration = "${aws_launch_configuration.example.id}"
  availability_zones = ["${data.aws_availability_zones.all.names}"]
  load_balancers = ["${aws_elb.example.name}"]
  health_check_type = "ELB"
```

3 Credit for this technique goes to Paul Hinze: *https://groups.google.com/d/msg/terraform-tool/7Gdhv1OAc80/iNQ93riiLwAJ*

```
  min_size = "${var.min_size}"
  max_size = "${var.max_size}"
  min_elb_capacity = "${var.min_size}"

  lifecycle {
    create_before_destroy = true
  }

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

As you may remember, a gotcha with the `create_before_destroy` parameter is that if you set it to true on a resource R, you also have to set it to true on every resource that R depends on. In the web server cluster module, the `aws_autoscaling_group` resource depends on one other resource, the `aws_elb`. The `aws_elb`, in turn, depends on one other resource, an `aws_security_group`. Set `create_before_destroy` to `true` on both of those resources.

If you re-run the `plan` command, you'll now see something that looks like this (I've omitted some of the output for clarity):

```
-/+ module.webserver_cluster.aws_autoscaling_group.example
    availability_zones.#:      "4" => "4"
    default_cooldown:          "300" => "<computed>"
    desired_capacity:          "2" => "<computed>"
    force_delete:              "false" => "false"
    health_check_type:         "ELB" => "ELB"
    launch_configuration:      "terraform-20161wu" =>
"${aws_launch_configuration.example.id}"
    max_size:                  "2" => "2"
    min_elb_capacity:          "" => "2"
    min_size:                  "2" => "2"
    name:                      "tf-asg-200170wox" => "${var.cluster_name}
-${aws_launch_configuration.example.name}" (forces new resource)
    protect_from_scale_in:     "false" => "false"
    tag.#:                     "1" => "1"
    tag.2305202985.key:        "Name" => "Name"
    tag.2305202985.value:      "webservers-stage" => "webservers-stage"
    vpc_zone_identifier.#:     "1" => "<computed>"
    wait_for_capacity_timeout: "10m" => "10m"

-/+ module.webserver_cluster.aws_launch_configuration.example
    ebs_block_device.#:  "0" => "<computed>"
    ebs_optimized:       "false" => "<computed>"
    enable_monitoring:   "true" => "true"
    image_id:            "ami-40d28157" => "ami-40d28157"
    instance_type:       "t2.micro" => "t2.micro"
```

```
    key_name:              "" => "<computed>"
    name:                  "terraform-20161118182404wu" => "<computed>"
    root_block_device.#: "0" => "<computed>"
    security_groups.#:     "1" => "1"
    user_data:             "416115339b" => "3bab6edc" (forces new resource)

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 new resource" next to its name parameter, which means Terraform will replace it with a new Auto Scaling Group running the new version of your code (or new version of your 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:

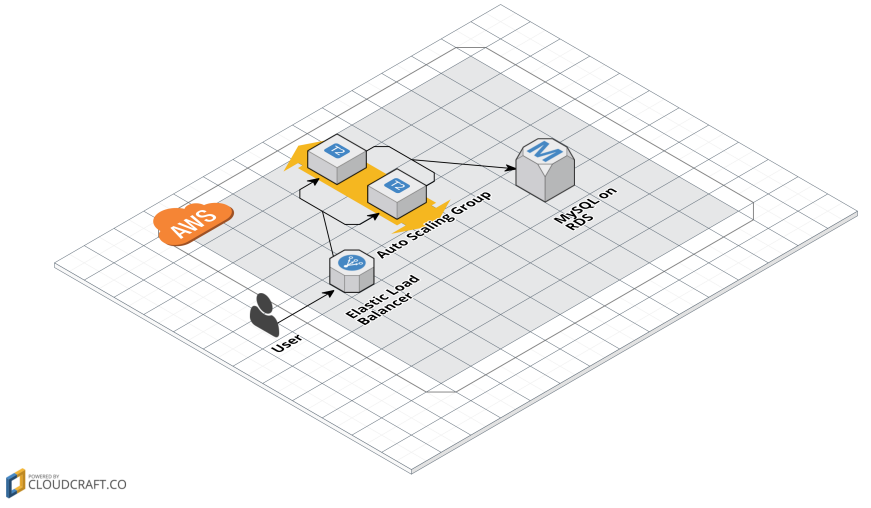



*Figure 5-1. Initially, you have the original ASG running v1 of your code*

You make an update to some aspect of the launch configuration, such as switching to an AMI that contains v2 of your code, and run the `apply` command. This forces Terraform to start deploying a new ASG with v2 of your code:

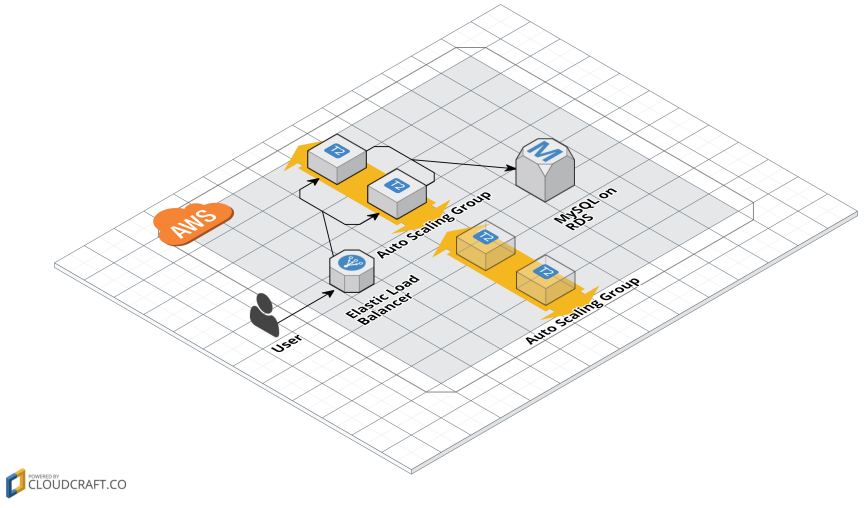



*Figure 5-2. Terraform begins deploying the new ASG with v2 of your code*

After a minute or two, the servers in the new ASG have booted, connected to the database, and registered in the ELB. 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 ELB happens to route them to:

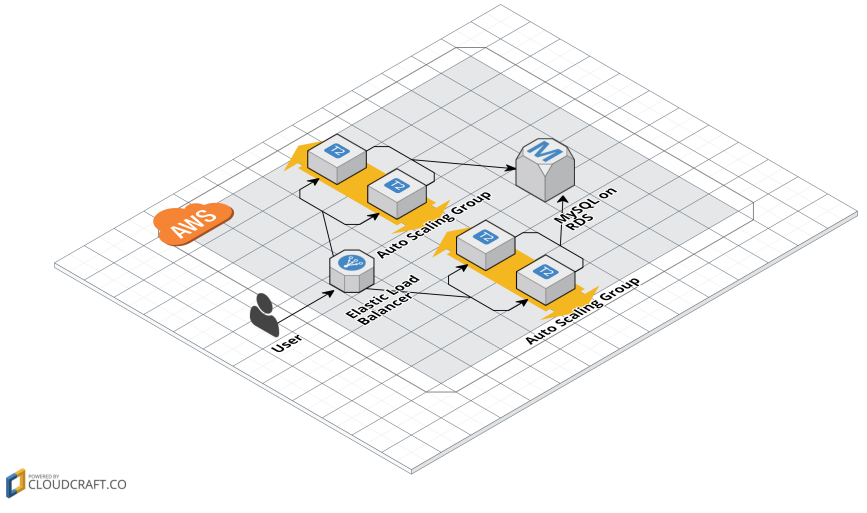



*Figure 5-3. The servers in the new ASG boot up, connect to the DB, register in the ELB, and start serving traffic*

Once `min_elb_capacity` servers from the v2 ASG cluster have registered in the ELB, Terraform will begin to undeploy the old ASG, first by deregistering the servers in that ASG from the ELB, and then by shutting them down:

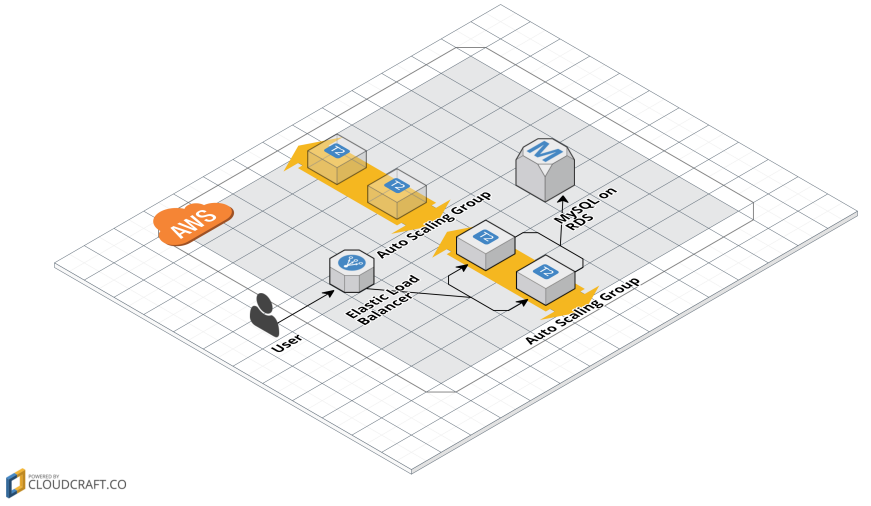



*Figure 5-4. The servers in the old ASG begin to shut down*

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:

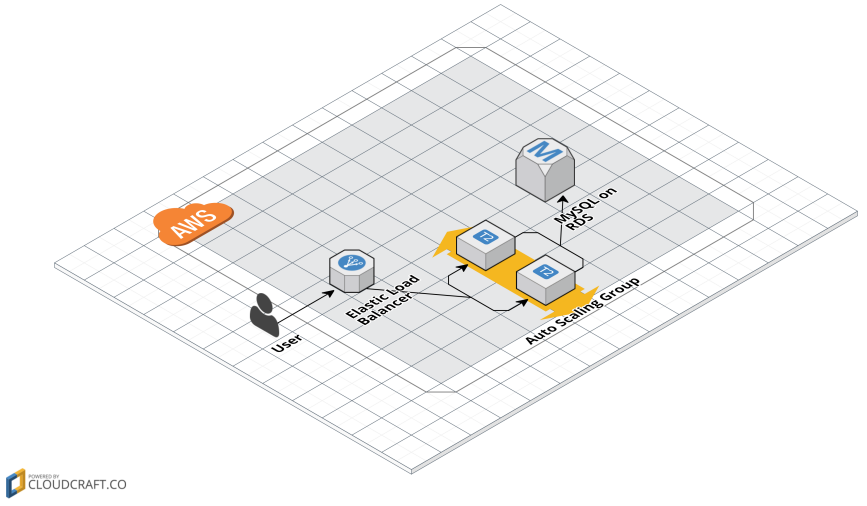



*Figure 5-5. Now, only the new ASG remains, which is running v2 of your code*

During this entire process, there are always servers running and handling requests from the ELB, so there is no downtime. Open the ELB URL in your browser and you should see something like Figure 5-6.

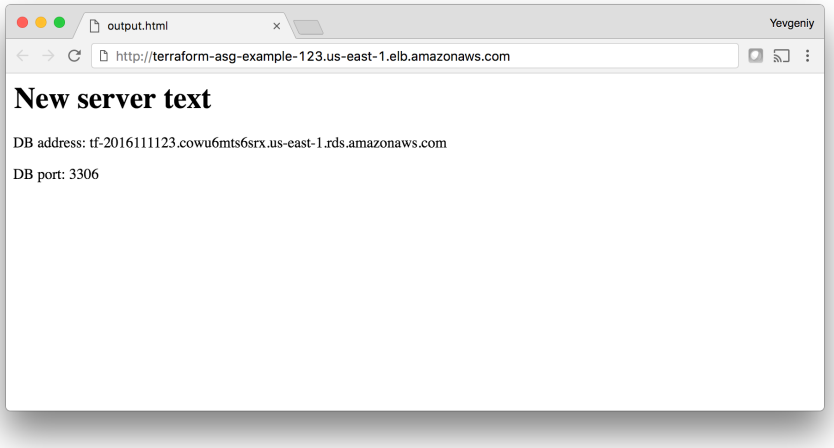



*Figure 5-6. The new code is now deployed*

Success! The new server text has deployed. As a fun experiment, make another change to the `server_text` parameter (e.g. update it to say "foo bar"), and run the

`apply` command. In a separate terminal tab, run `curl` in a loop, hitting your ELB once per second, and you'll be able to see the zero-downtime deployment in action:

```
> while true; do curl http://<load_balancer_url>; sleep 1; done
```

For the first minute or so, you should see the same response that says "New server text". Then, you'll start seeing it alternate between the "New server text" and "foo bar". This means the new Instances have registered in in the ELB. After another minute, the "New server text" will disappear, and you'll only see "foo bar", which means the old ASG has been shut down. The output will look something like this (for clarity, I'm listing only the contents of the `<h1>` tags):

```
New server text
New server text
New server text
New server text
New server text
New server text
foo bar
New server text
foo bar
New server text
foo bar
New server text
foo bar
New server text
foo bar
New server text
foo bar
foo bar
foo bar
foo bar
foo bar
foo bar
```

As an added bonus, if something went wrong during the deployment, Terraform will automatically roll back! For example, if there was a bug in v2 of your app and it failed to boot, then the Instances in the new ASG will not register with the ELB. 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 ELB, after which it will consider the deployment a failure, delete the v2 ASG, and exit with an error (meanwhile, v1 of your app continues to run just fine in the original ASG).

# Terraform Gotchas

After going through all these tips and tricks, it's worth taking a step back and pointing out a few gotchas, including those related to the loop, if-statement, and deployment techniques above, as well as those related to more general problems that affect Terraform as a whole:

- Count has limitations
- Zero-downtime deployment has limitations
- Valid plans can fail
- Refactoring can be tricky
- Eventual consistency is consistent… eventually

## Count has limitations

In the examples above, you made extensive use of the `count` parameter in loops and if-statements. This works well, but there is a significant limitation: you cannot use dynamic data in the count parameter. By "dynamic data," I mean any data that is fetched from a provider (e.g. from a data source) or is only available after a resource has been created (e.g. an output attribute of a resource).

For example, imagine you wanted to deploy multiple EC2 Instances, and for some reason did not want to use an Auto Scaling Group to do it. The code might look something like this:

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

resource "aws_instance" "example" {
  count = 3
  ami = "ami-40d28157"
  instance_type = "t2.micro"
}
```

What if you wanted to deploy one EC2 Instance per availability zone (AZ) in the current AWS region? You might be tempted to use the `aws_availability_zones` data source to retrieve the list of AZs and update the code as follows:

```
resource "aws_instance" "example" {
  count = "${length(data.aws_availability_zones.all.names)}"
  availability_zone =
    "${element(data.aws_availability_zones.all.names, count.index)}"
  ami = "ami-40d28157"
  instance_type = "t2.micro"
}
```

The code above uses the `length` interpolation function to set the `count` parameter to the number of available AZs, and the `element` interpolation function with `count.index` to set the `availability_zone` parameter to a different AZ for each EC2 Instance. This is a perfectly reasonable approach, but unfortunately, if you run this code, you'll get an error that looks like this:

```
aws_instance.example:resource count can't reference resource variable:
data.aws_availability_zones.all.names
```

The cause is that Terraform tries to resolve all the `count` parameters *before* fetching any dynamic data. Therefore, it's trying to parse `${length(data.aws_availability_zones.all.names)}` as a number *before* it has fetched the list of AZs and computed their length. This is an inherent limitation in Terraform's design and, as of November, 2016, it's an open issue in the Terraform community, with no clear indication of when it will be fixed.

For now, your only option is to manually look up how many AZs you have in your AWS region (every AWS account has access to different AZs, so check your EC2 console) and to set the `count` parameter to that hard-coded value:

```
resource "aws_instance" "example" {
  count = 3
  availability_zone =
    "${element(data.aws_availability_zones.all.names, count.index)}"
  ami = "ami-40d28157"
  instance_type = "t2.micro"
}
```

Alternatively, you can set the `count` parameter to a variable:

```
resource "aws_instance" "example" {
  count = "${var.num_availability_zones}"
  availability_zone =
    "${element(data.aws_availability_zones.all.names, count.index)}"
  ami = "ami-40d28157"
  instance_type = "t2.micro"
}
```

However, the value for that variable must also be hard-coded somewhere along the line (e.g. via a `default` defined with the variable or a value passed in via the command-line `-var` option) and not depend on any dynamic data:

```
variable "num_availability_zones" {
  description = "The number of Availability Zones in the AWS region"
  default = 3
}
```

## Zero-downtime deployment has limitations

Using `create_before_destroy` with an ASG is a great technique for getting zero-downtime deployment, but there is one limitation: it doesn't work with auto scaling policies. Or, to be more accurate, it resets your ASG size back to its `min_size`, which can be a problem if you had used auto scaling policies to increase the number of running servers.

For example, the web server cluster module includes a couple `aws_autoscaling_schedule` resources that increase the number of servers in the cluster from two to ten at 9 AM. If you ran a deployment at, say, 11 AM, the replacement ASG would

boot up with only two servers, rather than ten, and would stay that way until 9AM the next day.

There are a number of possible workarounds, such as:

- Change the `recurrence` parameter on the `aws_autoscaling_schedule` from "0 9 * * *", which means "run at 9 AM", to something like "0-59 9-17 * * *", which means "run every minute from 9 AM to 5 PM." If the ASG already has ten servers, re-running this auto scaling policy will have no effect, which is just fine; and if the ASG was just deployed, then running this policy ensures that the ASG won't be around for more than a minute before the number of Instances is increased to ten. This approach is a bit of a hack, and while it may work for scheduled auto scaling actions, it does not work for auto scaling policies triggered by load (e.g. "add two servers if CPU utilization is over 95%").
- Create a custom script that uses the AWS API to figure out how many servers are running in the ASG before deployment, use that value as the `desired_size` parameter of the ASG in the Terraform templates, and then kick off the deployment. After the new ASG has booted, the script should remove the `desired_size` parameter so that the auto scaling policies can control the size of the ASG. On the plus side, the replacement ASG will boot up with the same number of servers as the original, and this approach works with all types of auto scaling policies. The downside is that it requires a custom and somewhat complicated deployment script rather than pure Terraform code.

Ideally, Terraform would have first-class support for zero-downtime deployment, but as of November, 2016, this is an open issue in the Terraform community.

## Valid plans can fail

Sometimes, you run the `plan` command and it shows you a perfectly valid-looking plan, but when you run `apply`, you'll get an error. For example, try to add an `aws_iam_user` resource with the exact same name you used for the IAM user you created in Chapter 2:

```
resource "aws_iam_user" "existing_user" {
  # You should change this to the username of an IAM user that already
  # exists so you can practice using the terraform import command
  name = "yevgeniy.brikman"
}
```

If you now run the `plan` command, Terraform will show you a plan that looks reasonable:

```
+ aws_iam_user.existing_user
    arn:           "<computed>"
    force_destroy: "false"
```

```
    name:          "yevgeniy.brikman"
    path:          "/"
    unique_id:     "<computed>"

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

If you run the `apply` command, you'll get the following error

```
Error applying plan:

* aws_iam_user.existing_user: Error creating IAM User yevgeniy.brikman:
EntityAlreadyExists: User with name yevgeniy.brikman already exists.
```

The problem, of course, is that an IAM user with that name already exists. This can happen not only with IAM users, but almost any resource. Perhaps someone created it manually or with a different set of Terraform templates, but either way, some identifier is the same, and that leads to a conflict. There are many variations on this error, and Terraform newbies are often caught off-guard by them.

The key realization is that `terraform plan` only looks at resources in its Terraform state file. If you create resources *out-of-band*—such as by manually clicking around the AWS console—they will not be in Terraform's state file, and therefore, Terraform will not take them into account when you run the `plan` command. As a result, a valid-looking plan may still fail.

There are two main lessons to take away from this:

*Once you start using Terraform, you should only use Terraform*
: Once a part of your infrastructure is managed by Terraform, you should never make changes manually to it. Otherwise, you not only set yourself up for weird Terraform errors, but you also void many of the benefits of using infrastructure as code in the first place, as that code will no longer be an accurate representation of your infrastructure.

*If you have existing infrastructure, use the import command*
: If you created infrastructure before you started using Terraform, you can use the `terraform import` command to add that infrastructure to Terraform's state file, so Terraform is aware of and can manage that infrastructure. The `import` command takes two arguments. The first argument is the "address" of the resource in your Terraform templates. This makes use of the same syntax as interpolations, such as `TYPE.NAME` (e.g. `aws_iam_user.existing_user`). The second argument is a resource-specific ID that identifies the resource to import. For example, the ID for an `aws_iam_user` resource is the name of the user (e.g. yevgeniy.brikman) and the ID for an `aws_instance` is the EC2 Instance ID (e.g. i-190e22e5). The documentation for each resource typically specifies how to import it at the bottom of the page.

For example, here is the `import` command you can use to sync the `aws_iam_user` you just added in your Terraform templates with the IAM user you created back in Chapter 2 (obviously, you should replace "yevgeniy.brikman" with your own user name in the command below):

```
> 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 templates. From then on, when you run the `plan` command, Terraform will know that IAM user already exists and not try to create it again.

Note that if you have a lot of existing resources that you want to import into Terraform, writing the Terraform code for them from scratch and importing them one at a time can be painful, so you may want to look into a tool such as Terraforming, which can import both code and state from an AWS account automatically.

## Refactoring can be tricky

A common programming practice is *refactoring*, where you restructure the internal details of an existing piece of code without changing its external behavior. The goal is to improve the readability, maintainability, and general hygiene of the code. Refactoring is an essential coding practice that you should do regularly. However, when it comes to Terraform, or any infrastructure as code tool, you have to be careful about what defines the "external behavior" of a piece of code, or you will run into unexpected problems.

For example, a common refactoring practice is to rename a variable or a function to give it a clearer name. Many IDEs even have built-in support for refactoring and can rename the variable or function for you, automatically, across the entire codebase. While such a renaming is something you might do without thinking twice in a general purpose programming language, you have to be very careful in how you do it in Terraform, or it could lead to an outage.

For example, the `webserver-cluster` module has an input variable named `cluster_name`:

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

Perhaps you start using this module for deploying microservices, and you are tempted to rename the variable to `service_name`. It seems like a trivial change and an IDE like IntelliJ can apply it for you automatically. However, if you do that, you will cause an outage for all services that use this module on their next deployment.

That's because the `cluster_name` variable is used in a number of parameters, including the `name` parameters of the ELB and two security groups. If you change the `name` parameter of certain resources, Terraform will delete the old versions and create new ones to replace them. If the resource you are deleting happens to be an ELB, there will be nothing to route traffic to your web server cluster until the new ELB boots up. Similarly, if the resource you are deleting happens to be a security group, your servers will reject all network traffic until the new security group is created.

Another refactor you may be tempted to do is to change a Terraform identifier. For example, consider the `aws_security_group` resource in the `webserver-cluster` module:

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

  lifecycle {
    create_before_destroy = true
  }
}
```

The identifier for this resource is called `instance`. Perhaps you were doing a refactor and you thought it would be clearer to change this name to `cluster_instance`. What's the result? Yup, you guessed it: downtime.

As you saw in the previous section, Terraform associates each resource identifier with a 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. Well, if you change the resource identifier, such as changing the `aws_security_group` identifier from `instance` to `cluster_instance`, then as far as Terraform knows, you deleted the old resource and have added a completely new one. As a result, if you `apply` these changes, Terraform will delete the old security group and create a new one, and in the time period in between, your servers will reject all network traffic.

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

*Always use the plan command*
: All the gotchas above are easily caught by running the `plan` command, carefully scanning the output, and noticing that Terraform plans to delete a resource that you probably don't want deleted.

*Create before destroy*
: If you do want to replace a resource, then think carefully about whether its replacement should be created before you delete the original. If so, then you may be able to use `create_before_destroy` to make that happen. Alternatively, you can also accomplish the same effect manually through two steps: first, add the

new resource to your templates and run the `apply` command; second, remove the old resource from your templates and run the `apply` command again.

*All identifiers are immutable*
: The identifier you associate with each resource or module is immutable. If you change it, Terraform will delete the old resource or module and create a new one to replace it. Therefore, don't rename identifiers unless absolutely necessary, and even then, use the `plan` command, and consider whether you should use a create before destroy strategy.

*Some parameters are immutable*
: The parameters of many resources are immutable, so if you change them, Terraform will delete the old resource and create a new one to replace it. The documentation for each resource often specifies what happens if you change a parameter, so RTFM. And, once again, make sure to always use the `plan` command, and consider whether you should use a create before destroy strategy.

## Eventual consistency is consistent… eventually

The APIs for some cloud providers, such AWS, are asynchronous and eventually consistent. *Asynchronous* means the API may send a response immediately, without waiting for the requested action to complete. *Eventually consistent* means it takes time for a change to propagate throughout the entire system, so for some period of time, you may get inconsistent responses depending on which data store replica happens to respond to your API calls.

For example, let's say you make an API call to AWS asking it to create an EC2 Instance. The API will return a "success" (i.e. 201 Created) response more or less instantly, without waiting for the EC2 Instance creation to complete. If you tried to connect to that EC2 Instance immediately, you'd most likely fail because AWS is still provisioning it or the Instance hasn't booted yet. Moreover, if you made another API call to fetch information about that EC2 Instance, you may get an error in return (i.e. 404 Not Found). That's because the information about that EC2 Instance may still be propagating throughout AWS, and it'll take a few seconds before it's available everywhere.

In short, whenever you use an asynchronous and eventually consistent API, you are supposed to wait and retry for a while until that action has completed and propagated. Unfortunately, Terraform does not do a great job of this. As of version 0.7.x, Terraform still has a number of eventual consistency bugs that you will hit from time-to-time after running `terraform apply`.

For example, there is #5335:

```
> terraform apply
aws_route.internet-gateway:
```

```
error finding matching route for Route table (rtb-5ca64f3b)
and destination CIDR block (0.0.0.0/0)
```

And #5185:

```
> terraform apply
Resource 'aws_eip.nat' does not have attribute 'id' for variable 'aws_eip.nat.id'
```

And #6813:

```
> terraform apply
aws_subnet.private-persistence.2: InvalidSubnetID.NotFound:
The subnet ID 'subnet-xxxxxxx' does not exist
```

These bugs are annoying, but fortunately, most of them are harmless. If you just re-run `terraform apply`, everything will work fine, since by the time you re-run it, the information has propagated throughout the system.

It's also worth noting that eventual consistency bugs may be more likely if the place from where you're running Terraform is geographically far away from the provider you're using. For example, if you're running Terraform on your laptop in California and you're deploying code to the AWS region `eu-west-1`, which is thousands of miles away in Ireland, you are more likely to see eventual consistency bugs. I'm guessing this is because the API calls from Terraform get routed to a local AWS data center (e.g. `us-west-1`, which is in California), and the replicas in that data center take a longer time to update if the actual changes are happening in a different data center.

# 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`, `create_before_destroy`, and interpolation 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 interpolation documentation and let your inner hacker go wild. OK, maybe not too wild, as someone still has to maintain your code, but just wild enough that you can create clean, beautiful APIs for your users.

These users will be the focus of the next chapter, which describes how to use Terraform as a team. This includes a discussion of what workflows you can use, how to manage environments, how to test your Terraform templates, and more.