



This book is provided in digital form with the permission of the rightsholder as part of a Google project to make the world's books discoverable online.

The rightsholder has graciously given you the freedom to download all pages of this book. No additional commercial or other uses have been granted.

Please note that all copyrights remain reserved.

About Google Books

Google's mission is to organize the world's information and to make it universally accessible and useful. Google Books helps readers discover the world's books while helping authors and publishers reach new audiences. You can search through the full text of this book on the web at <http://books.google.com/>

JAMES TURNBULL

THE
TERRAFORM
BOOK

The Terraform Book

James Turnbull

September 6, 2018

Version: v1.4.7 (ad2b141)

Website: [The Terraform Book](#)



Some rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical or photocopying, recording, or otherwise, for commercial purposes without the prior permission of the publisher.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit [here](#).

© Copyright 2016 - James Turnbull <james@lovedthanlost.net>

ISBN 978-0-9888202-5-8 US\$14.99

A standard linear barcode representing the ISBN 978-0-9888202-5-8. Below the barcode, the numbers 9 780988 820258 are printed vertically.

9 780988 820258

Contents

	Page
Foreword	1
Who is this book for?	1
Credits and Acknowledgments	2
Technical Reviewers	2
Jennifer Davis	2
Thom May	2
Peter Miron	3
James Nugent	3
Paul Stack	3
Editor	4
Author	4
Conventions in the book	4
Code and Examples	5
Colophon	5
Errata	5
Disclaimer	5
Copyright	6
Version	7
Chapter 1 Introduction	8
Infrastructure as Code	9
Wait ... what's infrastructure?	10

Introducing Terraform	11
So why not a configuration management tool?	12
So why not CloudFormation et al?	13
So what can you use Terraform for?	14
Multi-tier applications	14
Self-service infrastructure	14
Production, development, and testing environments	15
Continuous delivery	15
Managing your management tools	15
What's in the book?	16
Links and information	17
Chapter 2 Installing and introducing Terraform	18
Installing Terraform	19
Installing Terraform on Linux	20
Installing Terraform on Microsoft Windows	21
Alternative Microsoft Windows installation	22
Alternative Mac OS X installation	22
Getting started with Terraform	24
Adding some configuration	25
Terraform configuration files	28
Setting up Amazon Web Services	30
Our first configuration file	32
Building our first resource	37
Terraform state	41
Showing our resource	44
Adding a second resource	45
Outputting plans	49
Targeting resources	53
Failed plans	54
Showing all the resources	54

Viewing the Terraform graph	56
Destroying infrastructure	60
Summary	62
Chapter 3 Building an application stack with Terraform	63
Our application stack	63
Parameterizing our configuration	65
Variables	66
Maps	69
Lists	72
Variable defaults	74
Populating variables	75
Starting our stack	80
Using AWS shared credentials	82
First resources	85
Modules	88
Defining a module	89
Module structure	93
Using our module	103
Getting our module	106
Moving our module to a repository	107
Counts and counting	111
Sets of counted resources using splat	112
Setting values with count indexes	113
Wrapping counts with the element function	118
Conditionals	119
Locals	121
Provisioning our stack	123
Finishing up our stack	125
Committing our configuration	126
Validating and formatting	127

Initializing Terraform	127
Planning our stack	128
Applying our stack	130
Graphing our stack	133
Seeing the results	134
Destroying the web stack resources	136
Summary	136
Chapter 4 Provisioning with Terraform	138
What does a provisioner do?	139
Provisioning an instance	140
Connection block	142
The file provisioner	143
Remote execution provisioning	152
Destroying the old web stack resources	158
Provisioning our web stack	159
Failed execution of a provisioner	161
Summary	162
Chapter 5 Collaborating with Terraform	164
Terraform state	165
Remote state	166
No state protection	166
Not all backends have locking	167
Creating an S3 remote state backend	168
Creating a module for an S3 remote state	168
Planning the remote state module	173
Applying our remote state module	174
Uploading the remote state module	176
Configuring Terraform to use remote state	176
Setting the remote state	181

Disabling remote state	186
Using and sharing remote state	188
Sharing state between users	189
Externally loading remote state backends	191
Using remote state data	193
Moving our base state remote	194
Adding the data source to the base configuration	197
State and service discovery	204
Creating a Consul cluster	205
Using Consul and remote state	213
Other tools for managing Terraform state	221
Summary	221
Chapter 6 Building a multi-environment architecture	223
Creating a data center	223
Creating a directory structure	225
Workflow	227
Develop	227
Plan	228
Apply in development	229
Automation and testing	229
Deploy to production	230
The development environment	230
Getting our VPC module	236
Adding some outputs for the development environment	236
Planning the development environment	237
Applying the development environment	238
Configuring remote state	240
Adding the web service	240
The web module	243
Using a data source in the web module	246

Web instances	250
A Cloudflare record	253
Committing our module	254
Getting our web module	254
Planning our web service	254
Applying the web service	255
Testing our web service	257
Removing our web service	259
Adding the API service	260
The API module	261
API instances	261
The API service outputs	263
Getting our module	263
Planning the API service	264
Applying the API service	264
Testing our API service	265
Adding a production environment	267
Adding services to production	268
State environments	268
Other resources for Terraform environments	269
Summary	270
Chapter 7 Infrastructure testing	271
Test Kitchen	272
InSpec	272
How Test Kitchen works	273
Prerequisites	273
Creating a test configuration	278
Creating our first control	284
Decorating controls with metadata	286
Adding another test to our control	287

Setting up a Test Kitchen instance	288
Running the controls	292
Adding a new control	294
Building custom InSpec resources	298
Alternative infrastructure testing frameworks	302
Summary	302
List of Figures	304
List of Listings	315
Index	316

Foreword

Who is this book for?

This book is a hands-on introduction to Terraform. The book is for engineers, developers, sysadmins, operations staff, and those with an interest in infrastructure, configuration management, or DevOps. It assumes no prior knowledge of Terraform. If you're a Terraform guru, you might find some of this book too introductory.

There is an expectation that the reader has basic Unix/Linux skills and is familiar with Git version control, the command line, editing files, installing packages, managing services, and basic networking.

An Amazon Web Services account and a basic understanding of how Amazon Web Services works is also useful. It would be relatively easy, however, to recreate the book's examples to use platforms like Google Cloud Platform or Azure.

Some of the examples in this book that use Amazon Web Services will cost you money—not a lot of money, but some. Y'all have been warned.

Finally, Terraform is evolving quickly and is currently in a pre-1.0.0 release state. That means “Here Be Dragons,” and you should take care when using Terraform in production.

The book assumes you are using Terraform 0.11.8 or later. Earlier versions may not work as described.

Credits and Acknowledgments

- Ruth Brown, who continues to humor these books and my constant tap-tap-tap of keys late into the night.
- Stephanie Coleman, for her friendship and good humor.
- Paul Stack, who answered many dumb questions.
- John Vincent and Dean Wilson, whose experiments with Terraform infrastructure testing were very useful.

Terraform™ is a registered trademark of HashiCorp, Inc.

Technical Reviewers

Jennifer Davis

Jennifer is the co-author of [Effective DevOps](#). She is a core organizer for devopsdays, co-organizer of [devopsdays Silicon Valley](#), and the founder of [Coffeeops](#). She blogs occasionally on [jendavis.org](#) and contributes content to SysAdvent and AWS Advent. She has spoken at a number of industry conferences about DevOps, tech culture, monitoring, and automation. In her role as an engineer at Chef, Jennifer develops Chef cookbooks to simplify building and managing infrastructure. When she's not working, she enjoys hiking Bay Area trails, learning to make things, and quality time with Brian and George. Find her on Twitter: [@sigje](#).

Thom May

Thom is an operations engineer with a keen interest in getting code to our customers as safely as possible. When he's not doing open source for Chef, he's generally skiing, running, or eating. Thom (re)tweets at [@thommay](#).

Peter Miron

Pete Miron scales engineering teams and software systems. He's currently building software to make it easier for engineering teams to scale software systems at Apcera. Find him on Twitter: [@petemiron](#).

James Nugent

James is an engineer at HashiCorp. He has been a long standing contributor to open source, including a contributor to both Packer and Terraform. He is now focusing on Terraform full time and ensuring the project continues to grow with its community.

Prior to HashiCorp, James was an architect at Boundary building a SaaS around network profiling and monitoring, where huge amounts of data had to be collected, aggregated and analyzed in real time. In his free time, he is a contributor to [EventStore](#), working on a database to do Complex Event Processing.

When James isn't working, he is often traveling for both pleasure and open source evangelism. He is a vintage guitar, recording equipment collector, and a coffee connoisseur.

Paul Stack

Paul Stack is an infrastructure coder who is passionate about continuous integration, continuous delivery, good operational procedures, and how they should be part of what developers and system administrators do on a day-to-day basis. He believes that reliably delivering software is as important as its development.

Editor

Sid Orlando is a writer and editor (among some other things), currently word-nerding out as Managing Editor at Kickstarter. She may or may not be experiencing recurring dreams about organizing her closet with dreamscape Docker containers. Find her on Twitter: [@ohrealsysid](#).

Author

James is an author and engineer. His most recent books are [The Packer Book](#); [The Terraform Book](#); [The Art of Monitoring](#); [The Docker Book](#), about the open-source container virtualization technology; and [The Logstash Book](#), about the popular open-source logging tool. James also authored two books about Puppet, [Pro Puppet](#) and [Pulling Strings with Puppet](#). He is the author of three other books: [Pro Linux System Administration](#), [Pro Nagios 2.0](#), and [Hardening Linux](#).

He is currently CTO at Emaptico and was formerly CTO at Kickstarter, VP of Services and Support at Docker, VP of Engineering at Venmo, and VP of Technical Operations at Puppet. He likes food, wine, books, photography, and cats. He is not overly keen on long walks on the beach or holding hands.

Conventions in the book

This is an [inline code statement](#).

This is a code block:

Listing 1: Sample code block

```
This is a code block
```

Long code strings are broken. If you see  in a code block it indicates that the output has been shortened for brevity's sake.

Code and Examples

The code and example configurations contained in the book are available on [GitHub](#) at:

<https://github.com/turnbullpress/tfb-code>

Colophon

This book was written in Markdown with a large dollop of LaTeX. It was then converted to PDF and other formats using PanDoc (with some help from scripts written by the excellent folks who wrote [Backbone.js on Rails](#)).

Errata

Please email any errata you find to james+errata@lovedthanlost.net.

Disclaimer

This book is presented solely for educational purposes. The author is not offering it as legal, accounting, or other professional services advice. While best efforts

have been used in preparing this book, the author makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The author shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. Every company is different and the advice and strategies contained herein may not be suitable for your situation. You should seek the services of a competent professional before beginning any infrastructure project.

Copyright

Some rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical or photocopying, recording, or otherwise, for commercial purposes without the prior permission of the publisher.



Figure 1: License

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit [here](#).

© Copyright 2016 - James Turnbull & Turnbull Press



Figure 2: ISBN

Version

This is version v1.4.7 (ad2b141) of The Terraform Book.

Chapter 1

Introduction

Besides black art, there is only automation and mechanization.

Federico Garcia Lorca

Building infrastructure is (pretty) easy. Building it in an automated, repeatable, and reusable way is much, much harder.

Perhaps you are a systems administrator, a developer, or you're doing DevOps work, and you've discovered that you need to manage your infrastructure components more efficiently and effectively. Maybe you've inherited a bespoke Cloud, Docker, or virtualized environment, or maybe you need to migrate to the Cloud or a virtualization environment—but you're finding the tools available to you aren't sufficient.

In some cases you might have a mix of manually constructed infrastructure, hand-edited configuration files, and 10,000-line shell scripts copied from one host to another, with undocumented fixes accumulating each time. If you're lucky, you might have a configuration management tool like Puppet or Chef to deploy your applications and services, or one or more bespoke deployment tools written to

deploy infrastructure and applications. Solutions are often undocumented, not scalable, and boutique to your existing infrastructure and architecture.

In addition to a lack of tooling, you have colleagues and customers who want their environments built more quickly than ever before. They want changes to compute and storage and networking to happen promptly—all error-free and done as automatically as possible. The combination of these challenges means you need to find a better way of doing things.

Thankfully in the last few years a lot of tools have been developed to help you out. These include tools that allow you to build applications and infrastructure like you write code. These tools enable a new concept:

Infrastructure as Code

Infrastructure as Code

“Infrastructure as Code” (IaC) enables us to describe our infrastructure and applications in source code. This enables us to treat our infrastructure like we treat our applications and take advantage of development practices and tools.

Infrastructure represented, and programmatically manipulatable, in code can be versioned, shared, re-used, debugged, and potentially reverted when something goes wrong. We can apply the tenets of our application development life cycle to our infrastructure including code reviews, linting, and automated testing. Instead of poorly and infrequently maintained documentation or state stored in the minds of your teammates, we can create managed environments that we can easily introspect, monitor, and debug.

Once our infrastructure is articulated in code it becomes equivalent to our application code. Instead of roadblocks or detours to handle infrastructure differently from applications we can instead:

- Build and provision automated development, testing, and production environments.
- Provision our applications and services on top of these environments.
- Automate the deployment workflow between them, beginning to head towards the Holy Grail of continuous deployment and delivery.

Wait ... what's infrastructure?

The first generation of these IaC tools, like Puppet and Chef, focused heavily on configuring operating systems and applications. They did branch into building infrastructure but generally in a limited form. You could often build the operating system's settings and application configuration for complex applications but not the hosts, services, and networking that they were built on.

Infrastructure is the building blocks upon which you construct your applications. The dividing line between what's infrastructure and what isn't is blurry in many cases. Generally the data center services (or Cloud services in other environments), hosts, virtual machines or Docker containers, networking (including routing, switching, and firewalls), and storage were all traditionally considered infrastructure. In many modern environments, infrastructure also includes more complex services or Software-as-a-Service (SaaS) products delivered by third parties such as DNS, Content Delivery Networks (CDN), databases, job scheduling, queues, and monitoring.

To fully realize the promise of IaC, we're starting to see tools that cater for these building blocks and integrate with some of the existing tools to bridge into managing operating systems and applications. In this book we're going to see one of those tools: Terraform.

Introducing Terraform

Terraform is an “Infrastructure as Code” tool for building and managing infrastructure efficiently and elegantly. Terraform provides declarative execution plans for building and running applications and infrastructure. What does declarative mean? Declarative plans are ones in which you specify outcomes. These outcomes are high-level states, problems, or definitions of the infrastructure you want to build.

In a traditional procedural build process you specify the steps (and sub-steps) to achieve these outcomes:

- Create host using VM management tool.
 - Launch VM management tool.
 - Click on “New VM”.
 - ...
- Install operating system using PXE boot.
- Configure networking using `ifconfig`.

In a declarative tool you declare the required end state:

- A virtual machine with two CPUs and 1GB of RAM.
- Ubuntu 16.04 installation.
- Two network interfaces with IP addresses `10.0.0.1/24` and `10.0.1.1/24`.

You let the tool take care of the steps required and focus on the outcome. This means you no longer need to worry about what specific commands to run, buttons to push, or settings to tweak. The tool underneath takes care of those details.

This is how Terraform operates: you rely on it to take care of the details while you articulate the end state you want. You define the list of resources and configuration required for your infrastructure. Terraform examines each resource and uses a [graph-based approach](#) to model and apply the desired state. Each resource is placed inside the graph, its relationships with other resources are calculated, and then each resource is automatically built in the correct order to produce your infrastructure. This means you never have to think about modeling these complex relationships. Terraform does it for you.

You can also import existing resources to get a jump start on your configuration. As your infrastructure grows and changes you can perform incremental updates and changes.

Terraform also integrates with configuration management and provisioning tools to allow you to control application and service level components. You can easily leverage and reuse existing configuration management content you may already have that defines your applications and infrastructure.

Right now, Terraform is primarily used to manage Cloud-based and Software-as-a-Service (SaaS) infrastructure but also supports various on-premise resources such as Docker and VMWare vSphere. Terraform can manage hosts, compute resources, data and storage, networking, as well as SaaS services like CDNs, monitoring tools, and DNS.

Terraform is written by the team at [HashiCorp](#). It is open source and licensed under the [Mozilla Public License, version 2.0](#). HashiCorp also sells an [enterprise version of Terraform](#).

So why not a configuration management tool?

Using Terraform versus configuration management is not an either/or. It's more “Why not just a configuration management tool?” As we discussed above, configuration management tools—like Puppet, Chef, and Ansible—are excellent at

managing applications and services, but they're often focused on software configuration rather than building the infrastructure components that underpin our applications. Terraform is instead focused on building and deploying infrastructure.

This makes Terraform a very complementary tool in most organizations when building and deploying applications and infrastructure. We recommend using Terraform to build your infrastructure and then using your existing configuration management tools, with their pre-existing configuration stores, to configure your applications.

So why not CloudFormation et al?

Many Cloud tools and SaaS services come with their own bootstrapping tools. An example is [AWS CloudFormation](#), which performs a similar function to Terraform, exclusively for AWS. Why not use one of these tools instead of Terraform? There are two big downsides to these tools.

First, they (generally) only work for their own resources. This can lead to a vendor lock-in, where you are so tightly tied to the vendor's tool that you can't use or migrate to other vendors' products. Vendor lock-in isn't always terrible. But it should be a choice that you consciously make, not something accidentally get locked into because you chose a specific vendor's tool.

Second, also because they just provision their own infrastructure, these tools alone don't necessarily make your environment fully functional. Some of these tools are best-of-breed for managing their own resources and don't need to extend beyond that. In some cases this means you may have a substantial coverage of your infrastructure. In others, you have to glue multiple tools together or write your own tool to manage disparate resources.

In comparison, Terraform provides a holistic solution that allows you to skip building that tool and limits the amount of infrastructure glue you need to create to

manage resources from different services.

So what can you use Terraform for?

There are a wide variety of potential use cases for Terraform.

Multi-tier applications

Terraform is focused on infrastructure deployment and supports a diverse collection of components including compute, storage, and networking assets. This makes it ideal to build N-tier applications. It's ideal for classic web applications with database back ends right up to multi-tier applications with web, cache, application, middleware, and database tiers. Each tier can be configured in Terraform, independent and isolated, and because it's all articulated in code, they can be easily scaled and managed by changing that code.

Self-service infrastructure

Often operations teams aspire to being infrastructure coordinators and controllers rather than bottlenecks in the infrastructure provisioning and management process. In this world, operations provides scalable and cost-effective self-service infrastructure to their customers.

The operations team can then focus on providing quality and value to consumers while managing risk through standardizing resources. Terraform enables operations folks to configure those standardized resources, creating standard stacks and packages that can then be provided to consumers as self-service infrastructure.

Production, development, and testing environments

A common problem in complex environments is that development and testing environments do not always reflect the state of the production environments. With Terraform, as your infrastructure is now codified, it's easy to ensure that your production configuration can be shared with your development and testing environments to ensure consistency and compatibility.

We can also start to move away from the idea of fixed environments, where production is always the same environment. Instead we can see infrastructure and environments like we see application code: as versioned and iterative. Instead of being locked to a single conception of “production,” we can deploy a new environment that can evolve into or become a new “production.”

Continuous delivery

If you're operating in an environment where applications are packaged in artifacts and your application configuration is done with configuration management tools, then adding Terraform allows you to also codify your infrastructure. This means all of the pieces of your environment are now automatically deployable, scalable, and manageable. You can chain together your infrastructure, your configuration management, and your build artifacts to produce continuous delivery pipelines.

Managing your management tools

In addition to managing Cloud-based resources, Terraform also configures many of the tools and infrastructure components you use to manage your infrastructure. Terraform can configure components like GitHub organizations and repositories, PagerDuty alerts, Grafana monitoring consoles, and DataDog metrics. This means you can not only use Terraform to build your infrastructure, but you can also start to codify the configuration of your management tools. Want to spin up a

new environment? With Terraform you not only get the compute, network, and storage components, but also the management framework around it.

What's in the book?

This is a hands-on introduction to Terraform. We'll use Terraform to build actual infrastructure and application stacks. The book will take you through Terraform basics into more advanced features and examples. It's an introductory book designed for folks who have never used Terraform, so if you're already a Terraform guru then some of the material may not be of interest.

Let's look at what's in each chapter.

- Chapter 1: This introduction.
- Chapter 2: Installation & Basics. Shows you how to install Terraform and introduces you to using it.
- Chapter 3: Building an infrastructure stack with Terraform. Takes you through building an initial infrastructure stack with Terraform.
- Chapter 4: Provisioning with Terraform. Shows you how you can provision applications on resources deployed with Terraform, including integrating with existing configuration management tools.
- Chapter 5: Collaborating with Terraform. Introduces you to collaborating and sharing Terraform configuration.
- Chapter 6: Building a multi-environment architecture. Demonstrates an architecture and workflow that allows you to develop and manage infrastructure and safely make change to that infrastructure.
- Chapter 7: Testing infrastructure changes. Takes your architecture and shows you how to write tests to validate it and how to use those tests and workflow to safely manage change in your infrastructure.

Links and information

- The [Terraform site](#).
- You can find the Terraform documentation on the HashiCorp site.
- You can get help from [the Terraform community](#).
- You can find the Terraform source code on GitHub.

Chapter 2

Installing and introducing Terraform

Welcome to the world of infrastructure as code. In this chapter, we'll take you through the process of installing Terraform on a variety of platforms. This isn't the full list of supported platforms but a representative sampling to get you started. We'll look at installing Terraform on:

- Linux.
- Microsoft Windows.
- Mac OS X.

You'll discover that the lessons here from installing Terraform can be extended to the other supported platforms.



NOTE We've written the examples in this book assuming Terraform is running on a Linux distribution. The examples should also work for Mac OS X but might need tweaking for Microsoft Windows.

After we've installed Terraform, we'll start to learn about how to use it to create and manage infrastructure. We'll learn about Terraform's workflow for creating, managing, and destroying infrastructure. We'll also start to delve into Terraform's configuration language. Finally, we'll build some simple infrastructure using Terraform to see it in action.

Installing Terraform

Terraform is shipped as a single binary file. The [Terraform site](#) contains zip files containing the binaries for specific platforms. Currently Terraform is supported on:

- Linux: 32-bit, 64-bit, and ARM.
- Max OS X: 32-bit and 64-bit.
- FreeBSD: 32-bit, 64-bit, and ARM.
- OpenBSD: 32-bit and 64-bit
- Illumos Distributions: 64-bit
- Microsoft Windows: 32-bit and 64-bit

You can also find [SHA256 checksums](#) for Terraform releases, and you can verify the checksums signature file has been signed using HashiCorp's GPG key. This allows you to validate the integrity of the Terraform binary.

Older versions of Terraform are available from the Hashicorp [releases service](#).



NOTE At the time of writing Terraform was at version 0.11.8.

Installing Terraform on Linux

To install Terraform on a 64-bit Linux host we can download the zipped binary file. We can use `wget` or `curl` to get the file from the download site.

Listing 2.1: Download the Terraform zip file

```
$ cd /tmp  
$ wget https://releases.hashicorp.com/terraform/0.11.8/terraform_0.11.8_linux_amd64.zip
```

Now let's unpack the `terraform` binary from the zip file, move it somewhere useful, and change its ownership to the `root` user.

Listing 2.2: Unpack the Terraform binary

```
$ unzip terraform_0.11.8_linux_amd64.zip  
$ sudo mv terraform /usr/local/bin/  
$ sudo chown -R root:root /usr/local/bin/terraform
```

We can now test if Terraform is installed and in our path by checking its version.

Listing 2.3: Checking the Terraform version on Linux

```
$ terraform version  
Terraform v0.11.8
```

Installing Terraform on Microsoft Windows

To install Terraform on Microsoft Windows we need to download the `terraform` executable and put it in a directory. Let's create a directory for the executable using Powershell.

Listing 2.4: Creating a directory on Windows

```
C:\> MKDIR terraform  
C:\> CD terraform
```

Now download the `terraform` executable from the download site into the `C:\terraform` directory:

Listing 2.5: Terraform Windows download

```
https://releases.hashicorp.com/terraform/0.11.8/terraform\_0.11.8\_windows\_amd64.zip
```

Unzip the executable using a tool like [7-Zip](#) into the `C:\terraform` directory. Finally, add the `C:\terraform` directory to the path. This will allow Windows to find the executable. To do this run this command inside Powershell.

Listing 2.6: Setting the Windows path

```
$env:Path += ";C:\terraform"
```

You should now be able to run the `terraform` executable.

Listing 2.7: Checking the Terraform version on Windows

```
C:\> terraform version  
Terraform v0.11.8
```

Alternative Microsoft Windows installation

You can also use a package manager to install Terraform on Windows. The [Chocolatey](#) package manager has a Terraform package available. You can use [these instructions to install Chocolatey](#) and then use the [choco](#) binary to install Terraform.

Listing 2.8: Installing Terraform via Chocolatey

```
C:\> choco install terraform
```

Alternative Mac OS X installation

In addition to being available as a binary for Mac OS X, Terraform is also available from [Homebrew](#). If you use Homebrew to provision your Mac OS X hosts then you can install Terraform via the [brew](#) command.

Listing 2.9: Installing Terraform via Homebrew

```
$ brew install terraform
```

 **NOTE** Please note that the brew installation builds from source and does

not use the verified Hashicorp binary release.

Homebrew will install the `terraform` binary into the `/usr/local/bin` directory. We can test it is operating via the `terraform version` command.

Listing 2.10: Checking the Terraform version on Mac OS X

```
$ terraform version  
Terraform v0.11.8
```

 **TIP** Another approach to OS X or Linux installation is [tfenv](#), which is much like Ruby's [rbenv](#) tool. It allows you to run and manage multiple Terraform versions at the same time.

Installing via configuration management

There are also configuration management resources available for installing Terraform. You can find:

- A [Puppet module](#) for Terraform.
- A [Chef cookbook](#) for Terraform.
- An [Ansible role](#) for Terraform.
- A Terraform [Docker container](#).

Getting started with Terraform

Now that we've got Terraform installed, let's see what it can do. The core of Terraform's functionality is provided by the `terraform` binary. Terraform behaves like most command line tools: we specify commands with arguments to execute specific functions. We've already returned Terraform's version with the `version` command. To return Terraform's help text we run the `terraform` binary with the `help` command.

Listing 2.11: Seeing the Terraform command options

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

Common commands:
  apply           Builds or changes infrastructure
  destroy         Destroy Terraform-managed infrastructure
  fmt             Rewrites config files to canonical format
  get             Download and install modules...
.

All other commands:
  state          Advanced state management
```

We can see the list of commands available to us (we've abbreviated that list in this output). In this chapter we're going to focus on three of those commands:

- `plan`: Shows us what changes Terraform will make to our infrastructure.
- `apply`: Applies changes to our infrastructure.
- `destroy`: Destroys infrastructure built with Terraform.

 **TIP** To get detailed help on a specific command run it with the `--help` flag—for example, `terraform plan --help`.

Let's start by creating some configuration to work with.

Adding some configuration

Let's define some infrastructure to be built by Terraform. We'll begin by creating a directory under our home directory called `terraform` to hold this Terraform configuration.

Listing 2.12: Create a home for Terraform

```
$ mkdir ~/terraform  
$ cd terraform
```

Let's now make a directory to hold our first example of Terraform configuration inside this directory. We'll call it `base`.

Listing 2.13: Create our first Terraform configuration directory

```
$ mkdir base  
$ cd base
```

Now we'll initialize the `base` directory as a Git version control repository. You'll want to always store your Terraform configuration in a version control system. We're going to use Git in the book to track our Terraform code. A version control system allows us to:

- Manage and version our Terraform code, just like we should manage and version our application code.
- Share our configuration across a team or organization.
- Ensure our infrastructure is tracked like our code.
- Allow us to apply development principles and life cycle to our infrastructure. In Chapters 5, 6, and 7, we'll see more details on sharing, collaboration, and development workflow for our infrastructure.

We like Git because it's easy, distributed, and integrates with GitHub, which is useful for hosting our source code.

 **NOTE** If you're interested in learning more about using Git [you can find some excellent resources on GitHub](#).

Listing 2.14: Creating a Git repository

```
$ git init
Initialized empty Git repository in /home/james/terraform/base/ .
git/
```

Let's also be good citizens and add a [README.md](#) file to tell folks what code is in this directory.

Listing 2.15: The base README.md file

```
# Our first AWS configuration for Terraform

An initial configuration for Terraform.

## Usage

```
$ terraform plan
$ terraform apply
```

## License

MIT
```

We can then add our `README.md` to Git and commit it.

Listing 2.16: Committing the README.md file

```
$ git add README.md
$ git commit -m "Adding our initial README file"
```

Now the next person who comes along will at least know something about this configuration.

 **TIP** Seeing a pattern here? When building infrastructure as code we try to use good software development practice: version control, documentation, comments, etc. Your colleagues will thank you for it!

Terraform configuration files

When Terraform runs inside a directory it will load any Terraform configuration files. Any non-configuration files are ignored and Terraform will not recurse into any sub-directories. Each file is loaded in alphabetical order, and the contents of each configuration file are appended into one configuration.

 **TIP** Terraform also has an “override” file construct. [Override files are merged rather than appended.](#)

Terraform then constructs a DAG, or Directed Acyclic Graph, of that configuration. The vertices of that graph—its nodes—are resources—for example, a host, subnet, or unit of storage; the edges are the relationships, the order, between resources. For example, a network may be required before a host can be created in order to assign it an IP address. The graph determines this relationship and then ensures Terraform builds your configuration in the right order to satisfy this.

 **TIP** This is a [similar technique to what Puppet uses](#) to construct its configuration catalog.

The configuration loading model allows us to treat individual directories, like `base`, as standalone configurations or environments. A common Terraform directory structure might look like:

Listing 2.17: Common directory structure

```
terraform/  
  base/  
  development/  
  production/  
  staging/
```

Each directory could represent an environment, stack, or application in our organization.

 **TIP** We'll learn more about the best approaches to structuring and managing our Terraform configurations in Chapter 6.

Let's create an initial configuration file inside the `~/terraform/base` directory. We'll call it `base.tf`.

Listing 2.18: Creating the base.tf file

```
$ pwd  
terraform/base  
$ touch base.tf
```

Terraform configuration files are normal text files. They are suffixed with either `.tf` or `.tf.json`. Files suffixed with `.tf` are in Terraform's native file format, and `.tf.json` files are **JSON**-formatted.

The two configuration file formats are for two different types of audiences:

- Humans.

- Machines.

Humans

The `.tf` format, also called the HashiCorp Configuration Language or HCL, is broadly human-readable, allows inline comments, and is generally recommended if humans are crafting your configuration.

Machines

The `.tf.json` format is pure JSON. The `.tf.json` format is meant for machine interactions, where a machine is building your configuration files.

You can use JSON if you'd prefer, but the HCL file format is definitely easier to consume and we recommend using it primarily.



TIP You can specify a mix of the Terraform file formats in a directory.

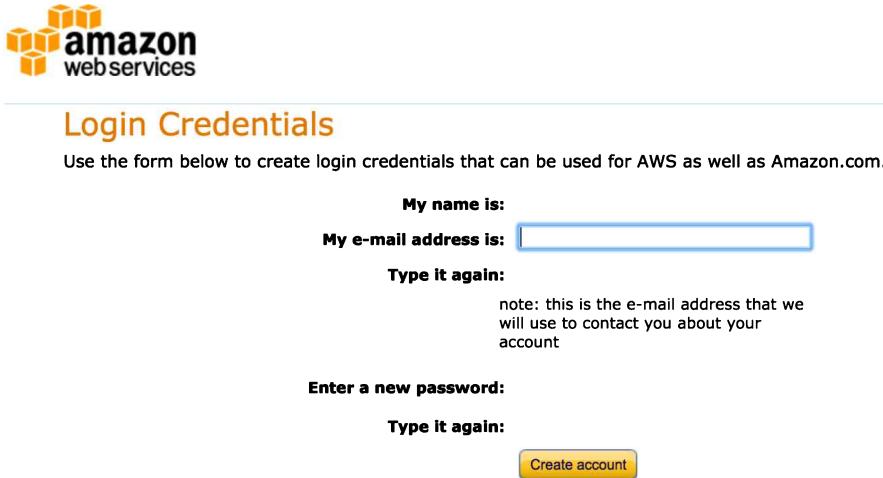
Setting up Amazon Web Services

We're going to use [Amazon Web Services](#) to build our initial infrastructure. Amazon Web Services have a series of [free plans](#) (they call them tiers) that we can use to test Terraform at no charge. We'll use those tiers in this chapter.

If you haven't already got a free AWS account, you can create one at:

<https://aws.amazon.com/>

Then follow [the Getting Started process](#).



The screenshot shows the 'Login Credentials' page from the AWS website. At the top left is the Amazon Web Services logo. Below it, the heading 'Login Credentials' is displayed in orange. A sub-instruction 'Use the form below to create login credentials that can be used for AWS as well as Amazon.com.' follows. The form contains several input fields and labels:

- 'My name is:' followed by a text input field.
- 'My e-mail address is:' followed by a text input field.
- 'Type it again:' followed by a text input field.
- A note: 'note: this is the e-mail address that we will use to contact you about your account'
- 'Enter a new password:' followed by a text input field.
- 'Type it again:' followed by a text input field.

At the bottom right of the form is a yellow 'Create account' button.

Figure 2.1: Creating an AWS account

As part of the Getting Started process you'll receive an access key ID and a secret access key. If you have an Amazon Web Services (AWS) account you should already have a pair of these. Get them ready. You'll use them shortly.

Alternatively, you should look at IAM or AWS Identity and Access Management. IAM allows multi-user role-based access control to AWS. It allows you to create access credentials per user and per AWS service.

Configuring it is outside the scope of this book, but here are some good places to learn more:

- IAM [getting started guide](#).
- [Root versus IAM credentials](#).
- [Best practices for managing access keys](#).

💡 TIP You should also [create and keep a key pair handy](#). We'll use this to allow us to SSH into any instances we're going to create.

Our first configuration file

Let's now populate the `base.tf` file with some initial Terraform configuration.

Listing 2.19: Initial base.tf configuration

```
/* This is a multi-line comment. This is a multi-line comment.  
This is a multi-line comment. This is a multi-line comment. This  
is a multi-line comment. This is a multi-line comment. */  
provider "aws" {  
    access_key = "abc123"  
    secret_key = "abc123"  
    region     = "us-east-1"  
}  
  
# This is a single-line comment.  
resource "aws_instance" "base" {  
    ami          = "ami-0d729a60"  
    instance_type = "t2.micro"  
}
```

 **TIP** You can add single-line comments to configuration by starting the line with `#`. Multi-line comments can be wrapped in `/* multi-line comment */`. White-space is not meaningful in Terraform configurations.

You can see we've specified two configuration objects in the file: a `provider` and a `resource`.

Providers

Providers connect Terraform to the infrastructure you want to manage—for example, AWS, Microsoft Azure, or a variety of other Cloud, network, storage, and SaaS services. They provide configuration like connection details and authentication credentials. You can think about them as a wrapper around the services whose infrastructure we wish to manage.

Providers are not shipped with Terraform since Terraform 0.10. In order to download the providers you’re using in your environment you need to run the `terraform init` command to install any required providers. Let’s do that now to get the `aws` provider.

Listing 2.20: Adding the aws provider

```
$ cd base  
$ terraform init
```

This will initialize Terraform’s setup and download the `aws` provider and make it available for our Terraform configuration. In this case, we’re using the `aws` provider. The `aws` provider lets us create and manage Amazon Web Services infrastructure. We’ll need to specify some connections details—like to which AWS region we’d like to connect. We’ll also need to specify some authentication credentials—the access and secret key we discussed above—that will allow us to use our AWS account from Terraform.

You can find a full list of the available providers and what they can manage in the [Terraform provider documentation](#).

 **TIP** You can specify multiple providers in a Terraform configuration to manage resources from multiple services or from multiple regions or parts of a service.

We can see our `provider` specifies a name, here `aws`, and a block of configuration inside braces: `{ }`. Our configuration is a series of `key = value` pairs. In this case we're specifying three pairs:

- `access_key`: Our AWS access key ID.
- `secret_key`: Our AWS secret access key.
- `region`: The AWS region in which to manage infrastructure.

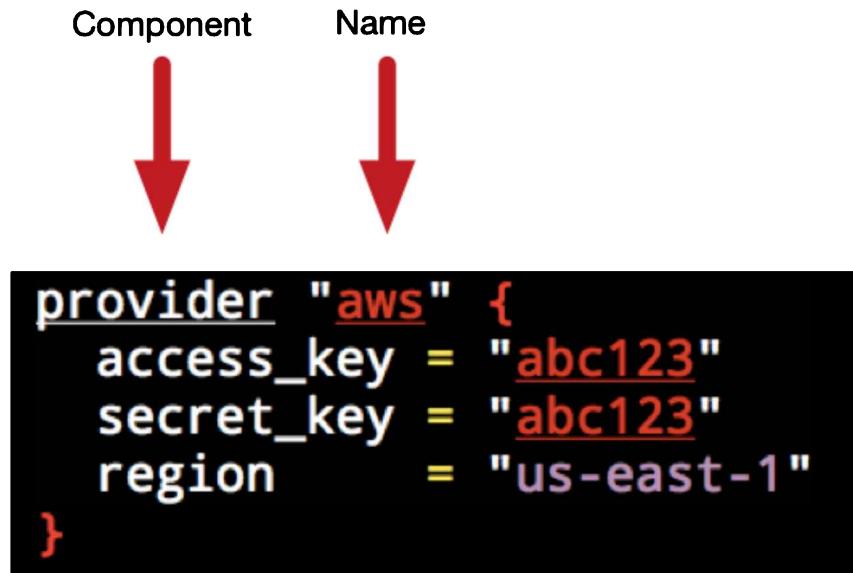


Figure 2.2: Provider definition

With this information Terraform can connect to AWS and perform any operations in the `us-east-1` region.

TIP Instead of hard-coding the AWS credentials you can also use [environment](#)

variables or AWS shared credentials.

Resources

Our second object is a `resource`. Resources are the bread and butter of Terraform. They represent the infrastructure components you want to manage: hosts, networks, firewalls, DNS entries, etc. The `resource` object is constructed of a type, name, and a block containing the configuration of the resource.

The type of resource here is `aws_instance`, which manages AWS EC2 host instances. Each type of the resource is linked to a provider; you can tell which by the leftmost value in the type, here `aws`. This indicates that this type of resource is provided by the `aws` provider.

The name of the resource is specified next. This name is defined by you—here we've named this resource `base`. The name of the resource should generally describe what the resource is or does.

The combination of type and name must be unique in your configuration.

Hence there can be only one `aws_instance` named `base` in your configuration. If you specify more than one resource type with the same name you'll see an error like so:

```
* aws_instance.base: resource repeated multiple times
```

 **NOTE** Your configuration is defined as the scope of what configuration Terraform loads when it runs. You can have a resource with a duplicate name in another configuration—for example, another directory of Terraform files.

This restriction is what allows Terraform's graph to be constructed correctly. Each

resource is a vertex or node in that graph. Each node is uniquely identified, its edges examined to determine its relationship with other nodes in the graph, and an order determined for the creation of those resources.

Inside the block we specify the configuration of the resource. Each resource type has a different set of attributes you can configure. In our `aws_instance.base` resource we're defining two configuration attributes:

- `ami`
- `instance_type`

The `ami` specifies which [Amazon Machine Image](#) we want to launch, and the `instance_type` is the class of instance we want to launch. In this case we're launching one of Amazon's free tier AMIs of the `t2.micro` class.

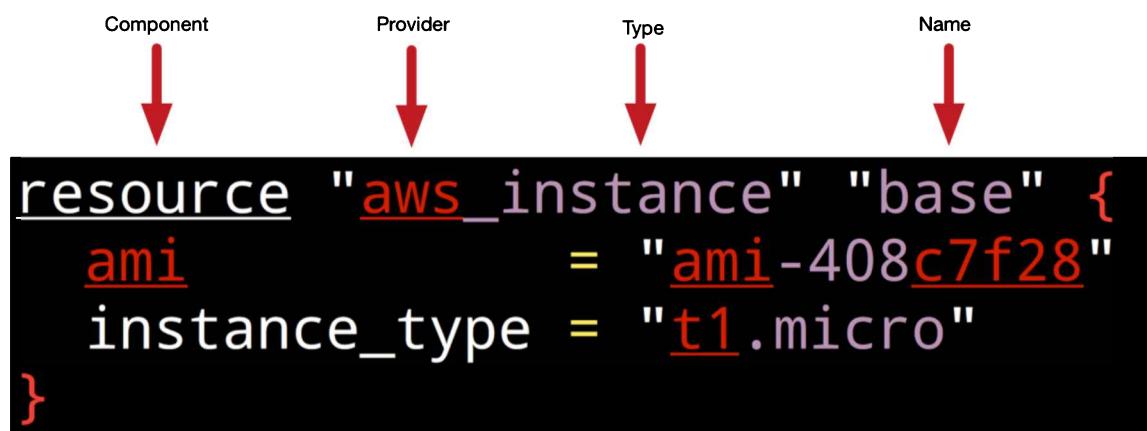


Figure 2.3: Resource definition

NOTE This assumes you're using the `us-east-1` region and that your Amazon account has a default VPC and is not in EC2 Classic. If you are in EC2 Classic you can replace these values with an AMI of `ami-408c7f28` and a type of `t1.micro`. If you're not in `us-east-1` then replace the AMIs with appropriate stock Ubuntu

16.04 AMIs from your region.

Building our first resource

So we've got a connection to AWS defined and a single resource, an AWS instance, defined. Now how do we get Terraform to create that resource? Terraform calls the process of creating and managing infrastructure an execution plan. You will likely run execution plans in one of two modes:

- Plan - Display the proposed changes to make.
- Apply - Apply the changes.

Let's start with showing our execution plan by running the `terraform` binary with the `plan` command inside the `~/terraform/base` directory. The `plan` command lets us see what Terraform is proposing to do, a check that we're going to make the right changes. This is useful for ensuring you're doing the right thing and not about to break your infrastructure. We recommend always running a `plan` before you apply any changes.

Listing 2.21: Displaying the Terraform plan

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...

.

+ aws_instance.base
  ami:          "ami-0d729a60"
  availability_zone:  "<computed>"
  instance_state:   "<computed>"
  instance_type:    "t2.micro"
  key_name:        "<computed>"
  network_interface_id:  "<computed>"
  placement_group:  "<computed>"
  private_dns:      "<computed>"
  private_ip:       "<computed>"
  public_dns:       "<computed>"
  public_ip:        "<computed>"

.

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

 **NOTE** Remember Terraform commands load all the Terraform configuration in the current directory. They do not recurse into sub-directories. So here Terraform will only load our `base.tf` file.

We can see our AWS instance is listed and that Terraform has created an identifier for the resource by merging the type and name:

`aws_instance.base`

You can see the resource identifier is prefixed with a `+`. This indicates that this

resource is a planned change that has not yet been made. Think about the `+` like you would in a version control system. It indicates a new, pending change. There are a series of similar indicators:

- `+`: A resource that will be added.
- `-`: A resource that will be destroyed.
- `-/+`: A resource that will be destroyed and then added again.
- `:`: A resource will be changed.

We then see the configuration of the resource. Some of the attributes have values, like the `ami` and `instance_type` attributes we configured earlier, but others say `<computed>`. A `<computed>` value is one that Terraform does not know the value of yet. The value of the configuration item will only be known when the resource is actually created—that is to say Terraform does not yet know what IP address AWS will assign to our instance.

After reviewing our `plan` output, everything looks good. The next stage of the process is to actually create our resource. This is done using the `apply` command. Let's build our instance now.

 **TIP** Since Terraform 0.11 the `terraform apply` command now prompts interactively like the `terraform plan` command. You can override this behavior with the `-auto-approve` flag.

Listing 2.22: Applying the base resources

```
$ terraform apply
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

+ aws_instance.base
  id:                      <computed>
  ami:                     "ami-0d729a60"
  associate_public_ip_address: <computed>
  availability_zone:       <computed>

  ...

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

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

Enter a value:
```

You'll first see a breakdown of Terraform's proposed actions and an interactive prompt asking whether you want to continue. You'll need to answer **yes** to continue or anything else (or **Ctrl-C**) to kill the command.

If you enter **yes** then Terraform will start creating our resources.

Listing 2.23: Creating the base resources

```
aws_instance.base: Creating...
  ami:          "" => "ami-0d729a60"
  availability_zone: "" => "<computed>"

  ...
  tenancy:          "" => "<computed>"
  vpc_security_group_ids.#: "" => "<computed>"

aws_instance.base: Still creating... (10s elapsed)
aws_instance.base: Still creating... (20s elapsed)
aws_instance.base: Still creating... (30s elapsed)
aws_instance.base: Creation complete

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

We can see that Terraform has created our resource. You'll see that the `+` is now missing from the resource identifier, meaning that Terraform is creating our resource in this execution and not just planning it.

So now that it's created our resource, how does Terraform keep track of it and its state?

Terraform state

After creating our resource, Terraform has saved the current state of our infrastructure into a file called `terraform.tfstate` in our `base` directory. This is called a state file. The state file contains a map of resources and their data to resource IDs.

The state is the canonical record of what Terraform is managing for you. This file is important because it is canonical. If you delete the file Terraform will not know what resources you are managing, and it will attempt to apply all configuration

from scratch. This is bad. You should ensure you preserve this file.

NOTE Terraform also creates a backup file of our state from the most recent previous execution in a file called `terraform.tfstate.backup`.

Some Terraform documentation recommends putting this file into version control. We do not. The state file contains everything in your configuration, including any secrets you might have defined in them. We recommend instead adding this file to [your .gitignore configuration](#).

Listing 2.24: Adding the state file and backup to .gitignore

```
$ pwd  
~/terraform/base  
$ echo "terraform.tfstate*" >> .gitignore  
$ git add .gitignore  
$ git commit -m "Adding .gitignore file"
```

NOTE We'll look at managing and sharing this state with other folks in Chapter 5.

Let's take a peek into the `terraform.tfstate` file.

Listing 2.25: The Terraform state file

```
{  
  "version": 3,  
  "terraform_version": "0.11.8",  
  "serial": 14,  
  "lineage": "019fad87-3f05-425d-9dec-3ab78db61db8",  
  "modules": [  
    {  
      "path": [  
        "root"  
      ],  
      "outputs": {},  
      "resources": {  
        "aws_instance.base": {  
          "type": "aws_instance",  
          "depends_on": [],  
          "primary": {  
            "id": "i-1c763b0a",  
            "attributes": {  
              "ami": "ami-0d729a60",  
              "availability_zone": "us-east-1e",  
            }  
          }  
        }  
      }  
    }  
  ]  
}
```

We can see that our `terraform.tfstate` file is a JSON representation of all the resources you're managing. The JSON is topped with some metadata, including the version of Terraform that created the state file.

As this file is the source of truth for the infrastructure being managed, it's critical to only use Terraform to manage that infrastructure. If you make a change to your infrastructure manually, or if you use another tool, it can be easy for this state to get out of sync with reality. You can then lose track of the state of your infrastructure and its configuration, or have Terraform reset your infrastructure back to a potentially non-functioning configuration when it runs.

We strongly recommend that if you want to manage specific infrastructure with

Terraform that it becomes the sole way of managing that infrastructure.

Showing our resource

Now that we've created the `aws_instance.base` resource, let's take a closer look at it. To do this we can use the `terraform show` command.

Listing 2.26: Displaying the base resource

```
$ terraform show
aws_instance.base:
  id = i-1c763b0a
  ami = ami-0d729a60
  availability_zone = us-east-1d
  instance_state = running
  instance_type = t2.micro
  private_dns = ip-172-31-22-157.ec2.internal
  private_ip = 172.31.22.157
  public_dns = ec2-52-87-195-107.compute-1.amazonaws.com
  public_ip = 52.87.195.107

  ...
  subnet_id = subnet-8dc734d6
  vpc_security_group_ids.717946557 = sg-4f713c35
```

We can see the full list of the resource's attributes and their values. This includes the values of the attributes we configured earlier and some new values, computed when we created the resource. This means all the `<computed>` attributes have now been replaced with actual values from Amazon. For example, now that we've created our `aws_instance.base` resource, we know its IP address, DNS, ID, and other useful information.

If we go to the Amazon console we'll see our EC2 instance is running.

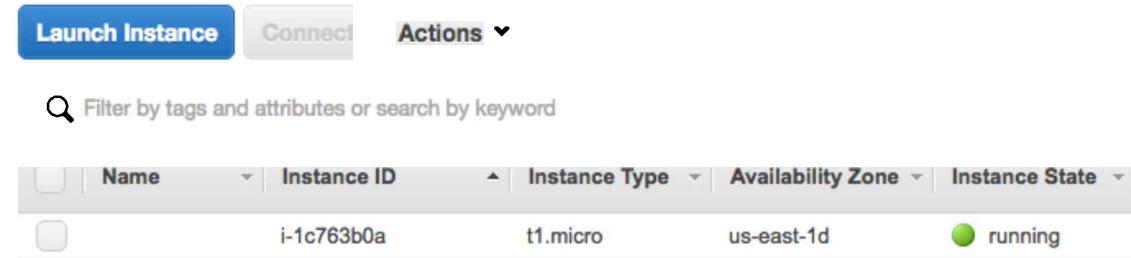


Figure 2.4: AWS EC2 instance

TIP Have existing infrastructure? Terraform can import it. You can read about how in the [Terraform import documentation](#). In some cases, it's also possible to recreate an accidentally deleted state file by importing resources.

Adding a second resource

Managing one resource isn't very exciting. Let's add another resource to our `base.tf` configuration file.

Listing 2.27: Adding a second resource

```
resource "aws_instance" "base" {
    ami           = "ami-0d729a60"
    instance_type = "t2.micro"
}

resource "aws_eip" "base" {
    instance = "${aws_instance.base.id}"
}
```

After our `aws_instance.base` resource, we've specified a new resource called

`aws_eip.base`. The `aws_eip` type manages [Amazon Elastic IP Addresses](#). We've named the resource `base`. The configuration of our `aws_eip.base` resource contains one attribute: `instance`. The value of our `instance` attribute is especially interesting:

```
${aws_instance.base.id}
```

Here we've referenced an attribute, `id`, from our `aws_instance.base` resource. Attribute references are variables and are very useful. They allow us to use values from one resource in another resource.

Variables are wrapped in Terraform's [interpolation syntax](#): `${ }` . When Terraform runs, this attribute would be interpolated from:

```
${aws_instance.base.id}
```

to the actual value:

```
i-1c763b0a
```

 **TIP** We'll learn more about variables and interpolation in Chapter 3.

Terraform uses this interpolated value to connect our Elastic IP address to the EC2 instance we've just created. You can consider this as creating an implicit dependency in the graph Terraform creates when it runs between our Elastic IP address and our EC2 instance.

Validating and formatting

Two useful commands we might run before planning our configuration are `terraform validate` and `terraform fmt`. The `validate` command checks the syntax and validates your Terraform configuration files and returns any errors. The `fmt` command neatly formats your configuration files.

You could even specify both as a [pre-commit hook](#) in your Git repository. There's an example of a hook like this in [this gist](#).

Planning our second resource

Now, let's see what happens when we run `terraform plan`.

Listing 2.28: Planning a second resource

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but
will not be persisted to local or remote state storage.

aws_instance.base: Refreshing state... (ID: i-1c763b0a)

. . .

+ aws_eip.base
  allocation_id:      "<computed>"
  association_id:    "<computed>"
  domain:            "<computed>"
  instance:          "i-1c763b0a"
  network_interface: "<computed>"
  private_ip:         "<computed>"
  public_ip:          "<computed>"
  vpc:                "<computed>"

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

We can see Terraform is refreshing our state to determine the current state of our infrastructure. To do this it connects to AWS and checks that the configuration of our `aws_instance.base` resource matches the definition.

Every Terraform plan or apply follows the same process:

1. We query the state of the resources, if they exist now.
2. We compare that state against any proposed changes to be made, building the graph of resources and their relationships. As a result of the graph, Terraform will only propose the set of required changes.
3. If they are not the same, either show the proposed change, if in the plan phase, or make the change, if in the apply phase.

Our plan process also lets us know that this is just a plan and running this command won't change the current state.

We can then see our new resource, `aws_eip.base`, prefixed with a `+` to indicate it is a new, pending change. Again, most of the attributes are `<computed>`, but we can see that our `instance` attribute has been interpolated to the ID of our `aws_instance.base` resource. When we apply the configuration this will connect our Elastic IP to our EC2 instance.

If any other changes had been made to our infrastructure, outside of Terraform, then Terraform would also show us what would be needed to bring the infrastructure back in line with our Terraform configuration.

Applying our second resource

Let's apply our configuration now.

 **NOTE** In this command and future `terraform apply` commands we're going to skip the interactive prompt and assume you've typed `yes` to save some space in the output.

Listing 2.29: Applying our second resource

```
$ terraform apply
aws_instance.base: Refreshing state... (ID: i-1c763b0a)
aws_eip.base: Creating...
  allocation_id:      "" => "<computed>"
  association_id:    "" => "<computed>"
  domain:            "" => "<computed>"
  instance:          "" => "i-1c763b0a"
  network_interface: "" => "<computed>"
  private_ip:        "" => "<computed>"
  public_ip:         "" => "<computed>"
  vpc:               "" => "<computed>"
aws_eip.base: Creation complete

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

We can see our new Elastic IP address has been created and assigned to our `aws_instance.base` instance. You'll note that Terraform did not recreate our whole configuration; rather it examined the current state, compared the differences, and only applied the relevant changes.

Where possible, Terraform will aim to perform the smallest incremental change rather than rebuilding every resource. In some cases, however, changing a resource requires recreating it. Since this is a destructive action, you should always carefully read the proposed actions in a `terraform apply` before saying `yes` or run `terraform plan` first to understand the impact of executing the change. The last thing you want to do is inadvertently destroy a running application.

Outputting plans

Terraform has an approach for trying to limit the risk of large-scale destructive changes to our environment and allowing us to make increment changes. To do this Terraform captures the proposed changes by outputting the plan it intends to

run to a file.

Terraform calls this a plan output. We capture the plan by specifying the `-out` flag on a `terraform plan` command. This will capture the proposed changes in a file we specify. The plan output means we can make small, incremental changes to our infrastructure.

Let's assume we haven't applied our configuration with our second resource: `aws_eip.base`. Let's run `terraform plan` and output the contents of the plan.

Listing 2.30: Creating a plan output

```
$ terraform plan -out base-`date +'%s'`.plan
...
aws_instance.base: Refreshing state... (ID: i-e729627e)
...
Your plan was also saved to the path below. Call the "apply" subcommand
with this plan file and Terraform will exactly execute this execution
plan.

Path: base-1477640557.plan

+ aws_eip.base
  allocation_id:      "<computed>"
  association_id:    "<computed>"
  domain:            "<computed>"
  instance:          "i-e729627e"
  network_interface: "<computed>"
  private_ip:         "<computed>"
  public_ip:          "<computed>"
  vpc:                "<computed>"

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

We can see we've run `terraform plan` with a new flag, `-out`. As the value of the `-out` flag, we've specified the location of a plan output. We've also appended the current Unix epoch date to the file name to ensure it is unique using the `date +'%s'` command. You can structure your plan output file names any way that suits.

We will now see a new file, `base-1477640557.plan`—our plan output has been

created in the `~/terraform/base` directory. This captures the changes proposed in our planning run: the addition of the `aws_eip.base` resource.

We can use this plan when applying our configuration.

Listing 2.31: Applying an output plan

```
$ terraform apply base-1477640557.plan
aws_eip.base: Creating...
  allocation_id:      "" => "<computed>"
  association_id:    "" => "<computed>"
  domain:            "" => "<computed>"
  instance:          "" => "i-e729627e"
  network_interface: "" => "<computed>"
  private_ip:         "" => "<computed>"
  public_ip:          "" => "<computed>"
  vpc:               "" => "<computed>"
aws_eip.base: Creation complete

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

  . . .
```

Here we've run the `terraform apply` command, appending the plan file name to command. Terraform will then apply **only** the proposed changes in the plan output we specified. You can see it has also skipped the interactive prompt asking you to okay the action!

 **TIP** You can also convert your plan file into JSON using [the useful tfjson tool](#).

So this looks like we made the same change as if we'd run `terraform apply` alone. Why is this useful? First, we don't have to apply this change immediately. The

plan output can be kept and stored as a potential incremental change. Let's say we go on to make a series of additional changes to our Terraform configuration after saving our plan output, adding more resources or changing other configuration variables. We can continue to make changes, run `terraform plan`, and save each additional change as a plan output. These plan outputs then become small steps we could apply to our infrastructure, incrementally and carefully.

We can then change as much of the infrastructure as we want in the Terraform configuration and apply each specific plan output as we wish in a systematic and careful way.

This is also the way we'd typically run automated Terraform actions, for example in a script or continuous integration tool. This avoids `terraform apply`'s interactive mode because obviously in most scripts you can't answer `yes`. If you don't want to use plans (and this is fairly risky!) you can run `terraform apply -auto-approve` to also skip the interactive prompt.



WARNING The generated plan file will contain all your variable values, potentially including any credentials or secrets. It is not encrypted or otherwise protected. Handle this file with appropriate caution!

Targeting resources

To help with the systematic and incremental rollout of resources, Terraform has another useful flag: `-target`. You can use the `-target` flag on both the `terraform plan` and `terraform apply` commands. It allows you to target a resource—or more if you specify multiple `-target` flags—to be managed in an execution plan.

Listing 2.32: Targeting a resource

```
$ terraform plan -target aws_eip.base
```

Here we've told Terraform to plan only for the `aws_eip.base` resource. The `target` flag will produce a reduced version of our graph that only includes the specified resources and any resources linked or dependent on it. In the case of our `aws_eip.base` resource, this will be the sole resource. If we were to target the `aws_instance.base` resource, Terraform would plan for it and the `aws_eip.base` resource, as they are linked.

Failed plans

If our execution plan had failed, then Terraform would not roll back the resources. It'll instead mark the failed resource as `tainted`. The tainted state is Terraform's way of saying, "This resource may not be right." Why tainting instead of rolling back? Terraform always holds to the execution plan: it was asked to create a resource, not delete one. If you run the execution plan again, Terraform will attempt to destroy and recreate any tainted resources.

Showing all the resources

Let's use `terraform show` to see both of our resources.

Listing 2.33: Showing the second resource

```
$ terraform show
aws_eip.base:
  id = eipalloc-5ed0a061
  association_id = eipassoc-a4e85398
  domain = vpc
  instance = i-1c763b0a
  network_interface = eni-8754ff81
  private_ip = 172.31.2.44
  public_ip = 52.0.192.210
  vpc = true
aws_instance.base:
  id = i-1c763b0a

  . . .
```

We can see our new `aws_eip.base` resource in the output. You'll note that it appears first in our configuration output, but we specified it last in our `base.tf` file. What does this mean for the ordering of resources in our configuration files? Terraform configurations do not depend on the order in which they are defined.

Terraform is a [declarative system](#); you specify the proposed state of your resources rather than the steps needed to create those resources. When you specify resources, Terraform builds a [dependency graph](#) of your configuration. The dependency graph represents the relationships between the resources in your configuration. When you plan or apply that configuration, Terraform walks that graph, works out which resources are related, and hence knows the order in which to apply them.

 **TIP** As a result of the dependency graph, Terraform tries to perform as many operations in parallel as it can. It can do this because it knows what it has to sequence and what it can create stand-alone.

For example, we've specified the `aws` provider and two AWS resources. We've told Terraform we'd like the EC2 resource to use the Elastic IP address resource. This is an implicit dependency. When we plan or apply the configuration, Terraform builds the dependency graph, walks the graph, and knows to create the EC2 instance prior to the Elastic IP address because of that dependency.

 **NOTE** Generally you always want the graph to dictate resource ordering. But sometimes we do need to force order in our resources. If we need to do this we can use a special attribute called `depends_on`.

Viewing the Terraform graph

One of the nice features of Terraform is that we can actually view this graph. To do that we run the `graph` command.

Listing 2.34: Creating the dependency graph

```
$ terraform graph
digraph {
    compound = "true"
    newrank = "true"
    subgraph "root" {
        "[root] aws_eip.base" [label = "aws_eip.base", shape = "box"]
        "[root] aws_instance.base" [label = "aws_instance.base", shape =
= "box"]
        "[root] provider.aws" [label = "provider.aws", shape =
diamond]
        "[root] aws_eip.base" -> "[root] aws_instance.base"
        "[root] aws_instance.base" -> "[root] provider.aws"
    }
}
```

The `graph` command outputs our dependency graph in the DOT graph format. That output can be piped to a file so we can visualize the graph.

Listing 2.35: Piping the graph command

```
$ terraform graph > base.dot
```

We can then view this graph in an application like [Graphviz](#).

Installing Graphviz

If you need to install Graphviz, you can do it on OS X via `brew`.

Listing 2.36: Install graphviz via brew

```
$ brew install graphviz
```

Or on Ubuntu via [apt](#).

Listing 2.37: Installing graphviz on Ubuntu

```
$ sudo apt install graphviz
```

Or on Microsoft Windows you can download a binary MSI installer from [the Graphviz site](#).

Or if you used [Chocolatey](#) to install Terraform earlier, you can also install Graphviz via the [choco](#) binary.

Listing 2.38: Installing graphviz with Chocolatey

```
C:\> choco install graphviz
```

If you don't want to install Graphviz then you can use the online [WebGraphviz](#) tool.

Creating an image

We can then use the Graphviz tools to create a viewable image from our [base.dot](#) file.

Listing 2.39: Creating a viewable graph

```
$ dot base.dot -Tsvg -o base.svg
```

This will create a `base.svg` image file we should be able to view.

Viewing our graph

If we now load the `base.svg` file in an image viewer or browser we can see:

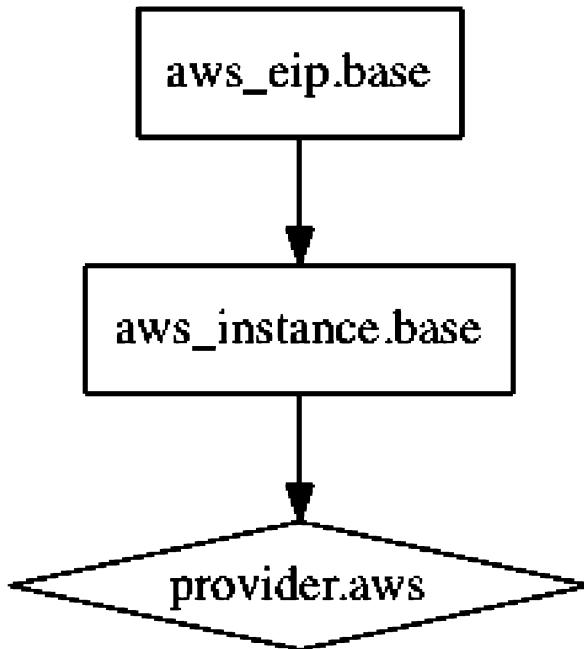


Figure 2.5: The `base.tf` graph

We can see that both resources and the provider that instantiated them are defined and connected: the provider at the base, the instance above it, and the Elastic IP address above the instance. This represents the dependency relationship—and the order of creation—of these resources.

Destroying infrastructure

In addition to planning and applying configuration with Terraform we can also destroy it. Let's destroy our EC2 instance and associated Elastic IP address using the `destroy` command.

Listing 2.40: Destroying our resources

```
$ 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: yes

aws_instance.base: Refreshing state... (ID: i-1c763b0a)
aws_eip.base: Refreshing state... (ID: eipalloc-5ed0a061)
aws_eip.base: Destroying...
aws_eip.base: Destruction complete
aws_instance.base: Destroying...
aws_instance.base: Still destroying... (10s elapsed)
aws_instance.base: Still destroying... (20s elapsed)
aws_instance.base: Destruction complete

Destroy complete! Resources: 2 destroyed.
```

You can see we're first prompted to enter `yes` to confirm we really want to destroy these resources. The state is then queried for each resource and it is destroyed. Finally, Terraform reports the success or failure of the destruction.

The `terraform destroy` command, without any options, destroys everything!

⚠️ WARNING If the above paragraph isn't already sending warning signals ... be very, very careful with `terraform destroy`. You can easily destroy your

entire infrastructure.

If you only want to destroy a specific resource then you can use the `-target` flag. For example, to only destroy the `aws_eip.base` resource we'd use:

Listing 2.41: Destroying a single resource

```
$ terraform destroy -target=aws_eip.base
Do you really want to destroy?
  Terraform will delete the following infrastructure:
    aws_eip.base
  There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes

aws_eip.base: Refreshing state... (ID: eipalloc-ca1466f5)
aws_eip.base: Destroying...
aws_eip.base: Destruction complete

Destroy complete! Resources: 1 destroyed.
```

You can see only our `aws_eip.base` resource is destroyed. The `-target` flag will also destroy any dependencies of the resource specified. So using the `-target` flag to destroy the `aws_instance.base` resource would also destroy the `aws_eip.base` resource.

You can also plan the `terraform destroy` process by passing the `-destroy` flag to the `terraform plan` command and saving a plan file.

Listing 2.42: Planning a Terraform destroy

```
$ terraform plan -destroy -out base-destroy-'date +'%s''.plan
```

This command will save a plan that will destroy all resources as `base-destroy-epochtime.plan`.

The destruction of resources will update and remove them from the state file.

 **TIP** Terraform also has the concept of `tainting` and `untainting` resources. Tainting resources marks a single resource to be destroyed and recreated on the next `apply`. It doesn't change the resource but rather the current state of the resource. Untainting reverses the marking.

Summary

In this chapter we've learned how to install Terraform on a variety of platforms. We then were introduced to Terraform's configuration language and files. We covered the basics of working with the `terraform` binary by building some basic infrastructure.

We learned about how to define basic resources and how to plan their execution. We then created those resources, were introduced to the basics of Terraform state, and learned how to graph the state. Finally, we destroyed our resources.

In the next chapter, we're going to build a much more complex infrastructure. We'll learn more about Terraform's configuration language and how to manage interactions between resources.

Chapter 3

Building an application stack with Terraform

In the last chapter we installed Terraform and got a crash course in the basics of creating, managing, and destroying infrastructure. We learned about Terraform configuration files and the basics of Terraform syntax.

In this chapter we're going to build a more complex infrastructure: a multi-tier web application. We're going to use this exercise to learn more about Terraform configuration syntax and structure.

Our application stack

We're going to build a two-tier web application stack. We're going to build this stack in Amazon Web Services (AWS) in an Amazon VPC environment. We'll create that VPC and the supporting infrastructure as well as the stack itself. The stack will be made up of two components:

- An Amazon Elastic Load Balancer (ELB).
- Two EC2 instances.

The ELB will be load balancing between our two EC2 instances.

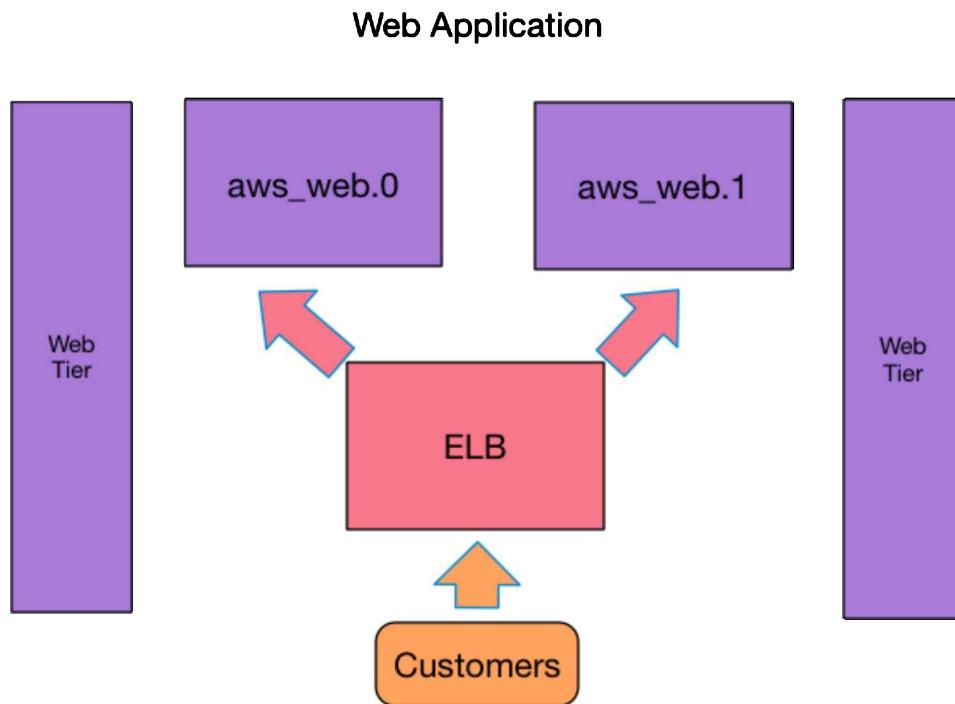


Figure 3.1: Our web application stack

Before we build the stack, we're going to learn about a new concept: parameterizing your configuration.

⚠️ WARNING If you're following along with this and subsequent chapters, note that we will be creating infrastructure in AWS that will cost small amounts of money to run. We recommend monitoring the infrastructure you're launching in your AWS console and destroying or terminating infrastructure when it is no longer needed.

Parameterizing our configuration

In the previous chapter we created some configuration in our `~/terraform/base/base.tf` configuration file.

Listing 3.1: Our original configuration

```
provider "aws" {  
    access_key = "abc123"  
    secret_key = "abc123"  
    region     = "us-east-1"  
}  
  
resource "aws_instance" "base" {  
    ami          = "ami-0d729a60"  
    instance_type = "t2.micro"  
}  
  
resource "aws_eip" "base" {  
    instance = "${aws_instance.base.id}"  
    vpc      = true  
}
```

You can see we've hard-coded several attributes in this configuration: the AWS credentials, the AMI, the instance type, and whether our Elastic IP is in a VPC. If we were to expand upon this configuration, we'd end up repeating a number of these values. This is not very **DRY**. DRY is an abbreviation for “Don’t Repeat Yourself,” a software principle that recommends reducing the repetition of information. It’s also not very practical or efficient if we have to change these values in multiple places.

 **TIP** A little later, in the **Using AWS shared credentials** section, we'll talk more about AWS credentials and how to better protect them.

Variables

In order to address this we're going to parameterize our configuration using [variables](#). Variables allow us to centralize and manage values in our configuration. Let's use the configuration from Chapter 2 to learn more about variables.

We start by creating a file, called `variables.tf`, to hold our variables. We create the file in the `~/terraform/base` directory.

Listing 3.2: Creating the `variables.tf` file

```
$ cd ~/terraform/base  
$ touch variables.tf
```

 **TIP** The file can be called anything. We've just named it `variables.tf` for convenience and identification. Remember all files that end in `.tf` will be loaded by Terraform.

Let's create a few variables in this file now. Variables can come in a number of types:

- Strings — String syntax. Can also be Boolean's: `true` or `false`.
- Maps — An associative array or hash-style syntax.
- Lists — An array syntax.

Let's take a look at some string variables first.

Listing 3.3: Our first variables

```
variable "access_key" {
  description = "The AWS access key."
}

variable "secret_key" {
  description = "The AWS secret key."
}

variable "region" {
  description = "The AWS region."
  default     = "us-east-1"
}
```

Terraform variables are created with a `variable` block. They have a name and an optional type, default, and description.

Our first two variables are the `access_key` and `secret_key` we need for our `aws` provider. We've only set a `description` for these variables. The value of these variables is currently undefined. The third variable is the `region`. This variable has a default value set with the `default` attribute.

You can also specify a variable type, either `string`, `map`, or `list`.

Listing 3.4: Variable type specified

```
variable "region" {
  type = "string"
  description = "The AWS region."
  default = "us-east-1"
}
```

If you omit the `type` attribute then Terraform assumes your variable is a string, unless the `default` is in the format of another variable type. Here Terraform

would assume the first variable is a string but that the second is a list.

Listing 3.5: Variable type specified

```
variable "region" {
  description = "The AWS region."
  default = "us-east-1"
}

variable "region_list" {
  description = "AWS availability zones."
  default = ["us-east-1a", "us-east-1b"]
}
```

You can supply an optional description of the variable using the `description` attribute.

Listing 3.6: Variable descriptions

```
variable "region" {
  description = "The AWS region."
  default = "us-east-1"
}
```

 **TIP** We recommend you always add variable descriptions. You never know who'll be using your code, and it'll make their (and your) life a lot easier if every variable has a clear description. Comments are fun too.

Let's update our `provider` with the new variables we've just created.

Listing 3.7: Adding our new variables

```
provider "aws" {  
    access_key = "${var.access_key}"  
    secret_key = "${var.secret_key}"  
    region     = "${var.region}"  
}
```

We've specified Terraform's interpolation syntax: `${ }` . Inside that syntax we've specified our three variables. Each variable is identified as a variable by the `var.` prefix. Currently only one of these variables has a value, the `default` of `us-east-1` we've set for the `var.region` variable. Soon we'll see how to populate values for the other variables.

 **TIP** Since Terraform 0.8 there is a command called `terraform console`. The console is a Terraform REPL that allows you to work with interpolations and other logic. It's a good way to explore working with Terraform syntax. You can read about it in the [console command documentation](#).

Maps

Most of our variable examples have, thus far, been strings. We can also specify two other types of variables: maps and lists. Let's look at maps first.

Maps are associative arrays and ideal for situations where you want to use one value to look up another value. For example, one of our potential configuration attributes is the EC2 instance's AMI. AMIs are region specific, so if we change region we will need to look up a new AMI. Terraform's maps are ideal for this

task.

Let's define a map in our `variables.tf` file.

Listing 3.8: A map variable

```
variable "ami" {
  type = "map"
  default = {
    us-east-1 = "ami-0d729a60"
    us-west-1 = "ami-7c4b331c"
  }
  description = "The AMIs to use."
}
```

We can see our new variable is called `ami`. We've specified a `type` of `map` and default values for two keys: the `us-east-1` and `us-west-1` regions.

 **NOTE** We don't need to specify the variable `type` if the variable's `default` is in the form of a map. In that case Terraform will automatically assume you've defined a map.

So how do we use this map variable? Let's update `base.tf` with the `ami` variable.

Listing 3.9: Using map variables in base.tf

```
provider "aws" {  
    access_key = "${var.access_key}"  
    secret_key = "${var.secret_key}"  
    region     = "${var.region}"  
}  
  
resource "aws_instance" "base" {  
    ami          = "${lookup(var.ami, var.region)}"  
    instance_type = "t2.micro"  
}  
  
resource "aws_eip" "base" {  
    instance = "${aws_instance.base.id}"  
    vpc      = true  
}
```

You can see we've specified an interpolated value for the `ami` attribute. Inside that interpolation we've added something new: a **function**. Terraform has a set of built-in functions to make it easier to work with variables and values.

 **NOTE** You can find a full list of functions in [the Terraform documentation](#).

In this case we're using a function called `lookup`, which performs a key lookup on a specific map like so:

`${lookup(map, key)}`

In our earlier example, we're using another variable we defined, `var.region`, to perform a lookup of the `var.ami` variable. So if our `var.region` variable was set to `us-west-1`, then our `ami` attribute would receive a value of `ami-7c4b331c`.

You can also look up maps explicitly—for example, `var.ami["us-west-1"]` will get the value of the `us-west-1` key from the `ami` map variable. Or you can even nest interpolations to look up a variable, like so:

```
 ${var.ami[var.region]}
```

You'll see that the second variable is not wrapped in `${ }` but specified bare. The interpolation syntax recognizes it is a variable and interpolates it too.

Lists

The last type of variable available in Terraform is the list. Let's assume we have a list of security groups we'd like to add to our instances. Our list would be constructed like so:

Listing 3.10: Constructing a list

```
variable "security_group_ids" {
  type    = "list"
  description = "List of security group IDs."
  default = ["sg-4f713c35", "sg-4f713c35", "sg-4f713c35"]
}
```

A list is wrapped in `[]` when it is defined and when it is used in your configuration. We can drop the `type` if the `default` is a list. We can specify a list directly as the value of a variety of attributes, for example:

Listing 3.11: Using a list

```
resource "aws_instance" "base" {  
    . . .  
    vpc_security_group_ids = ["${var.security_group_ids}"]  
}
```

 **NOTE** You'll need to create some security groups if you want to test this and use the resulting IDs in your list.

Lists are [zero-indexed](#). We can retrieve a single element of a list using the syntax:

`${var.variable[element]}`

Like so:

Listing 3.12: Retrieving a list element

```
resource "aws_instance" "base" {  
    . . .  
    vpc_security_group_ids = "${var.security_group_ids[1]}"  
}
```

 **NOTE** You can also use the `element` function to retrieve a value from a list.

This will populate the `vpc_security_group_ids` attribute with the second element in our `var.security_group_ids` variable.

Variable defaults

Variables with and without defaults behave differently. A defined, but empty, variable is a required value for an execution plan.

Listing 3.13: An empty variable

```
variable "access_key" {
  description = "The AWS access key."
}
```

If you run a Terraform execution plan then it will prompt you for the value of `access_key` (and any other empty variables).

Let's try that now.

Listing 3.14: Empty and default variables

```
$ terraform plan
var.access_key
Enter a value: abc123

var.secret_key
Enter a value: abc123

...
```

We can see that Terraform has prompted us to provide values for two variables: `var.access_key` and `var.secret_key`. Again, the `var` prefix indicates this is a variable, and the suffix is the variable name. Setting the variables for the plan

will not persist them. If you re-run `terraform plan`, you'll again be prompted to set values for these variables.

So how does Terraform populate and persist variables?

Populating variables

Of course, inputting the variable values every time you plan or apply Terraform configuration is not practical. To address this, Terraform has a variety of methods by which you can populate variables. Those ways, in order of descending resolution, are:

1. Loading variables from command line flags.
2. Loading variables from a file.
3. Loading variables from environment variables.
4. Variable defaults.

Loading variables from command line flags

The first method allows you to pass in variable values when you run `terraform` commands.

Listing 3.15: Command line variables

```
$ terraform plan -var 'access_key=abc123' -var 'secret_key=abc123'
```

We can also populate maps via the `-var` command line flag:

Listing 3.16: Setting a map with var

```
$ terraform plan -var 'ami={ us-east-1 = "ami-0d729a60", us-west-1 = "ami-7c4b331c" }'
```

And lists via the command line:

Listing 3.17: Populating a list via command line flag

```
$ terraform plan -var 'security_group_ids=[ "sg-4f713c35", "sg-4f713c35", "sg-4f713c35"]'
```

You can pass these variables on both the `plan` and `apply` commands. Obviously, like the input prompt, this does not persist the values of variables. Next time you run Terraform, you'll again need to specify these variables values.

Loading variables from a file

Our next method, populating variable values via files, does allow persistence. When Terraform runs it will look for a file called `terraform.tfvars`. We can populate this file with variable values that will be loaded when Terraform runs.

Let's create that file now.

Listing 3.18: Creating a variable assignment file

```
$ touch terraform.tfvars
```

We can then populate this file with variables—here a string, map, and list respectively.

Listing 3.19: Adding variable assignments

```
access_key = "abc123"
secret_key = "abc123"
ami = {
    us-east-1 = "ami-0d729a60"
    us-west-1 = "ami-7c4b331c"
}
security_group_ids = [
    "sg-4f713c35",
    "sg-4f713c35",
    "sg-4f713c35"
]
```

When Terraform runs it will automatically load the `terraform.tfvars` file and assign any variable values in it. The file can contain Terraform configuration syntax or JSON, just like normal Terraform configuration files.

Any variable for which you define a value needs to exist. In our case, the variables `access_key`, `secret_key`, and `security_group_ids` need to be defined with `variable` blocks in our `variables.tf` file. If they do not exist you'll get an error like so:

Listing 3.20: Variable doesn't exist error

```
module root: 1 error(s) occurred:

* provider config 'aws': unknown variable referenced: 'access_key'. define it with 'variable' blocks
```

You can also name the `terraform.tfvars` file something else—for example, we could have a variable file named `base.tfvars`. If you do specify a new file name, you will need to tell Terraform where the file is with the `-var-file` command line flag.

Listing 3.21: Running Terraform with a custom variable file

```
$ terraform plan -var-file base.tfvars
```

 **TIP** You can use more than one `-var-file` flag to specify more than one file. If you specify more than one file, the files are evaluated from first to last, in the order specified on the command line. If a variable value is specified multiple times, the last value defined is used.

Loading variables from environment variables

Terraform will also parse any environment variables that are prefixed with `TF_VAR`. For example, if Terraform finds an environment variable named:

`TF_VAR_access_code=abc123`

it will use the value of the environment variable as the string value of the `access_code` variable.

We can populate a map via an environment variable:

`TF_VAR_ami='{us-east-1 = "ami-0d729a60", us-west-1 = "ami-7c4b331c"}'`

and a list.

`TF_VAR_roles='["sg-4f713c35", "sg-4f713c35", "sg-4f713c35"]'`

 **TIP** Variable files and environment variables are a good way of protecting passwords and secrets. This avoids storing them in our configuration files, where they might end up in version control. A better way is obviously some sort of

secrets store. Since Terraform 0.8 there is now support for [integration with Vault for secrets management](#).

Variable defaults

Lastly, you can specify variable defaults for your variables.

Listing 3.22: Variable defaults

```
variable "region" {
  description = "The AWS region."
  default = "us-east-1"
}
```

Variable defaults are specified with the `default` attribute. If nothing in the above list of variable population methods resolves the variable then Terraform will use the default.

 **TIP** Terraform also has an “override” file construct. When Terraform loads configuration files it appends them. With [an override the files are instead merged](#). This allows you to override resources and variables.

Our new variables are useful syntax. Let’s start our build using some of this new syntax.

Starting our stack

Now that we've learned how to parameterize our configuration, let's get started with building a new application stack. Inside our `~/terraform` directory let's create a new directory called `web` to hold our stack configuration, and let's initialize it as a Git repository.

Listing 3.23: Creating the web directory

```
$ cd ~/terraform  
$ mkdir web  
$ cd web  
$ git init
```

Let's add our `.gitignore` file too. We'll exclude any state files to ensure we don't commit any potentially sensitive variable values.

Listing 3.24: Adding the state file and backup to `.gitignore`

```
$ echo "terraform.tfstate*" >> .gitignore  
$ git add .gitignore  
$ git commit -m "Adding .gitignore file"
```

It's important to note that this is a new configuration. Terraform configurations in individual directories are isolated. Our new configuration in the `web` directory will, by default, not be able to refer to, or indeed know about, any of the configuration in the `base` directory. We'll see how to deal with this in Chapter 5, when we talk more about state.



NOTE You can find the code for this chapter [on GitHub](#).

Let's create a new file to hold our stack configuration, a file to define our variables, and a file to populate our variables.

Listing 3.25: Creating the stack files

```
$ touch web.tf variables.tf terraform.tfvars
```

Let's begin by populating our `variables.tf` file.

Listing 3.26: Our `variables.tf` file

```
variable "access_key" {
  description = "The AWS access key."
}
variable "secret_key" {
  description = "The AWS secret key."
}
variable "region" {
  description = "The AWS region."
}
variable "key_name" {
  description = "The AWS key pair to use for resources."
}
variable "ami" {
  type    = "map"
  description = "A map of AMIs."
  default = {}
}
variable "instance_type" {
  description = "The instance type."
  default = "t2.micro"
}
```

Note that we've used variables similar to our example in Chapter 2. We've also

added a few new variables, including the name of a key pair we're going to use for our instances, and a map that will specify the AMI we wish to use.



TIP You should have [created a key pair](#) when you set up AWS.

Let's populate some of these variables by adding definitions to our `terraform.tfvars` file.

Listing 3.27: The web `terraform.tfvars` file

```
access_key = "abc123"
secret_key = "abc123"
region = "us-east-1"
ami = {
    us-east-1 = "ami-f652979b"
    us-west-1 = "ami-7c4b331c"
}
```

You can see we've provided values for our AWS credentials, the region, and a map of AMIs for the `us-east-1` and `us-west-1` regions.

Using AWS shared credentials

We mentioned earlier that we don't have to specify our credentials in the `terraform.tfvars` file. Indeed, it's often a very poor security model to specify these credentials in a file that could easily be accidentally distributed or added to version control. Instead of specifying the credentials in your configuration, you should configure the AWS client tools. These provide a shared credential configuration that Terraform can consume, removing the need to specify credentials.

To install the AWS client tools on Linux, we'd use Python `pip`:

Listing 3.28: Installing AWS CLI on Linux

```
$ sudo pip install awscli
```

On OS X we can use `pip` or `brew` to install the AWS CLI:

Listing 3.29: Installing AWS CLI on OSX

```
$ brew install awscli
```

On Windows, we'd use the [MSI installer from AWS](#), or if you've used the Chocolatey package manager, we'd install via the `choco` binary:

Listing 3.30: Installing awscli via choco

```
C:\> choco install awscli
```

We then run the `aws` binary with the `configure` option.

Listing 3.31: Running aws configure

```
$ aws configure
AWS Access Key ID [None]: abc123
AWS Secret Access Key [None]: abc123
Default region name [None]: us-east-1
Default output format [None]:
```

You would replace each `abc123` with your AWS credentials and specify your pre-

ferred default region. This will create a file in `~/.aws/credentials` with your credentials that will look like:

Listing 3.32: The aws/credentials file

```
[default]
aws_access_key_id = abc123
aws_secret_access_key = abc123
```

And a file called `~/.aws/config`, with our default region:

Listing 3.33: The aws/config file

```
[default]
region = us-east-1
```

 **TIP** Due to a bug with Terraform, you will still need to specify `region = us-east-1` (or your region) in your Terraform configurations. This is because Terraform does not seem to read the `config` file in some circumstances.

Now we can remove the `var.access_key` and `var.secret_key` variables from our `variables.tf` and `terraform.tfvars` files if we wish.

For the rest of the book we'll assume you have configured shared credentials, and we'll remove references to the access and secret keys!

For the other variables in our `variables.tf` file, we're going to rely on their defaults.

 **TIP** We could also use the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables to specify our credentials. Terraform also automatically consumes these variables. If you're on OS X, you should also look at `envchain`, which uses the OS X Keychain to help manage environment variables.

First resources

Now that we've got the inputs for our stack defined, let's start to create our resources and their configuration in the `web.tf` file.

Listing 3.34: Our web.tf file

```
provider "aws" {
    region = "${var.region}"
}

module "vpc" {
    source      = "./vpc"
    name        = "web"
    cidr        = "10.0.0.0/16"
    public_subnet = "10.0.1.0/24"
}

resource "aws_instance" "web" {
    ami           = "${lookup(var.ami, var.region)}"
    instance_type = "${var.instance_type}"
    key_name      = "${var.key_name}"
    subnet_id     = "${module.vpc.public_subnet_id}"
    associate_public_ip_address = true
    user_data     = "${file("files/web_bootstrap.sh")}"
}

vpc_security_group_ids = [
    "${aws_security_group.web_host_sg.id}",
]

count = 2
}

resource "aws_elb" "web" {
    name          = "web-elb"
    subnets       = ["${module.vpc.public_subnet_id}"]
    security_groups = ["${aws_security_group.web_inbound_sg.id}"]
    listener {
        instance_port = 80
        .
        .
    }
    instances     = ["${aws_instance.web.*.id}"]
}

resource "aws_security_group" "web_inbound_sg" {
```



TIP You'll find code you can download and use for this example on [GitHub](#).

You can see we've first added the `aws provider` to allow us to provision our resources from AWS. We've omitted the access and secret access keys from our provider because we've assumed we're using our AWS shared configuration to provide them. The only option we have specified for the provider is the `region`.

As we discussed in Chapter 2, you can define multiple providers, both for different services and for different configurations of service. A common Terraform pattern is to define multiple providers aliased for specific attributes—for example, being able to create resources in different AWS regions. Here's an example:

Listing 3.35: Multiple providers

```
provider "aws" {
    region      = "${var.region}"
}

provider "aws" {
    alias      = "west"
    region    = "us-west-2"
}

resource "aws_instance" "web" {
    provider = "aws.west"
}
.
```

We've defined an `aws provider` that uses our `var.region` variable to define the AWS region to which we'll connect. We've then defined a second `aws provider` with an `alias` attribute of `west` and the `region` hard-coded to `us-west-2`.

We can now refer to this specific provider by using `the provider attribute`. The

`provider` is a special type of attribute called a meta-parameter. Meta-parameters are attributes you can add to any resources in Terraform. Terraform has [a number of meta-parameters](#) available, and we'll see others later in the book.

 **TIP** The `depends_on` attribute we mentioned in the last chapter is also a meta-parameter.

We're going to stick with our single `aws` provider for now and use a single AWS region.

You can also see that we've added some new configuration syntax and structures to our `web.tf` file. Let's look at each of these now, starting with the `module` structure.

Modules

[Modules](#) are defined with the `module` block. Modules are a way of constructing reusable bundles of resources. They allow you to organize collections of Terraform code that you can share across configurations.

Often you have a configuration construct such as infrastructure like an [AWS VPC](#), an application stack, or other collection of resources that you need to repeat multiple times in your configurations. Rather than cutting and pasting and repeating all the resources required to configure that infrastructure, you can bundle them into a module. You can then reuse that module in your configurations.

You can configure inputs and outputs for modules: an API interface to your modules. This allows you to customize them for specific requirements, while your code remains as DRY and reusable as possible.

 **TIP** Hashicorp makes available a collection of verified and community modules in the [Terraform Module Registry](#). These include modules for a large number of purposes and are a good point to start if you need a module. You can learn more about the Terraform Module Registry in [the documentation](#).

Defining a module

To Terraform, every directory containing configuration is automatically a module. Using modules just means referencing that configuration explicitly. References to modules are created with the `module` block.

Listing 3.36: The `vpc` module

```
module "vpc" {  
    source = "./vpc"  
    name   = "web"  
    cidr   = "10.0.0.0/16"  
    public_subnet = "10.0.1.0/24"  
}
```

As you can see, modules look just like resources only without a type. Each module requires a name. The module name must be unique in the configuration.

Modules only have one required attribute: the module's `source`. The `source` tells Terraform where to find the module's source code. You can [store modules](#) locally in your filesystem or remotely in repositories such as GitHub. In our case the `vpc` module is located in a directory called `vpc` inside our `~/terraform/web` directory.

You can specify a module multiple times in a configuration by giving it a new name but specifying the same source. For example:

Listing 3.37: Multiple vpc modules

```
module "vPCA" {  
    source = "./vpc"  
    . . .  
}  
  
module "vPCB" {  
    source = "./vpc"  
    . . .  
}
```

Here Terraform would create two VPCs, one from `vPCA` and the other from `vPCB`. We would configure each differently.

Let's create the `vpc` directory first and initialize it as a Git repository, because ultimately we want to store our module on GitHub.

Listing 3.38: Creating the vpc module directory

```
$ pwd  
~/terraform/web  
$ mkdir vpc  
$ cd vpc  
$ git init
```

Inside our `source` attribute we specify the `vpc` directory relative to the `~/terraform/web` directory. Remember Terraform uses the current directory it's in when executed as its root directory. To ensure Terraform finds our module we need to specify the `vpc` directory relative to the current directory.

 **TIP** This path manipulation in Terraform is often tricky. To help with this,

Terraform provides a built-in variable called `path`. You can read about how to use the `path` variable in [the interpolation path variable documentation](#).

Instead of storing them locally, you can also specify remote locations for your modules. For example:

Listing 3.39: The vpc module with a remote source

```
module "vpc" {
  source = "github.com/turnbullpress/tf_vpc"
  ...
}
```

This will load our module from a GitHub repository:

https://github.com/turnbullpress/tf_vpc

This allows us to reference module configurations without needing to store them in directories underneath or adjacent to our configuration.

This also allows us to create versioning for modules. Terraform can refer to a specific repository branch or tag as the source of a module. For example,

Listing 3.40: Referencing a module version

```
module "vpc" {
  source = "git::https://github.com/turnbullpress/tf_vpc.git?ref=production"
}
```

The `git::` prefix tells Terraform that the `source` is a Git repository. The `ref=` suffix can be a branch name, a tag, or a commit. Here we're downloading the `production` branch of the module in the `tf_vpc` repository.

Or if you want to get a module specifically from the [Terraform Registry](#) then you can use syntax like so:

Listing 3.41: Referencing a registry module

```
module "vpc" {  
    source = "terraform-aws-modules/vpc/aws"  
}
```

The source path format for Terraform Registry modules looks like this:

`namespace/name/provider`

The `namespace` is like an organization or source of the module. The `name` is the module's name and the `provider` is the specific provider it uses. The [module's homepage](#) will contain full documentation on how to use it, including any required inputs and any outputs.

 **NOTE** Modules with a blue tick on the Terraform Registry are [verified](#) and from a Hashicorp partner. These modules should be more resilient and tested than others. You can also [publish](#) your own modules on the Registry.

Terraform Registry modules can also be versioned and you can use a specific version of a module like so:

Listing 3.42: Referencing a registry module's version

```
module "vpc" {  
    source = "terraform-aws-modules/vpc/aws"  
    version = "1.3.0"  
}
```

 **TIP**

You can find a full list of the potential sources and how to configure them in the [module source documentation](#).

Module structure

Inside our `vpc` directory our module is identical to any other Terraform configuration. It will have variables, variable definitions, and resources.

Variables

Let's start with creating a file to hold the module's variables. We'll use a file called `interface.tf`.

 **TIP** The explicit file name makes it clear that this is the module's API, the interface to the module.

Listing 3.43: Creating the vpc module variables

```
$ cd vpc  
$ touch interface.tf
```

We populate this file with the variables we'll use to configure the VPC that the module is going to build.

Listing 3.44: The vpc module's variables

```
variable "name" {  
    description = "The name of the VPC."  
}  
variable "cidr" {  
    description = "The CIDR of the VPC."  
}  
variable "public_subnet" {  
    description = "The public subnet to create."  
}  
variable "enable_dns_hostnames" {  
    description = "Should be true if you want to use private DNS  
within the VPC"  
    default      = true  
}  
variable "enable_dns_support" {  
    description = "Should be true if you want to use private DNS  
within the VPC"  
    default      = true  
}
```

You can see that we've defined a number of variables. Some of the variables will be required: `name`, `cidr`, and `public_subnet`. These variables currently have no defaults, so we must specify a value for each of them. We've specified the values in the `module` block in our `web.tf` file. This represents the incoming API for the

vpc module.

Listing 3.45: The vpc module's default variables

```
module "vpc" {
  source = "./vpc"
  name   = "web"
  cidr   = "10.0.0.0/16"
  public_subnet = "10.0.1.0/24"
}
```

If we do not specify a required variable, then Terraform will fail with an error:

```
Error loading Terraform: module root: module vpc: required variable
cidr not set
```

 **NOTE** So what's `module root`? Well, remember that modules are just folders containing files. Terraform considers every folder of configuration files a module. Terraform has created an implicit module, called the `root` module, from the stack configuration contained in the `/terraform/web` directory.

We also have several variables with defaults that we can override when configuring our module.

Listing 3.46: Overriding vpc module's default variables

```
module "vpc" {
  source = "./vpc"
  .
  .
  enable_dns_hostnames = false
}
```

This will override the default value of the `enable_dns_hostnames` variable and set it to `false`.

Module resources

Now let's add the resources to configure our VPC. We'll create a configuration file called `main.tf` to hold the resources and then populate it.

Listing 3.47: The vpc module resources

```
resource "aws_vpc" "tfb" {
  cidr_block          = "${var.cidr}"
  enable_dns_hostnames = "${var.enable_dns_hostnames}"
  enable_dns_support    = "${var.enable_dns_support}"
  tags {
    Name = "${var.name}"
  }
}

resource "aws_internet_gateway" "tfb" {
  vpc_id = "${aws_vpc.tfb.id}"
  tags {
    Name = "${var.name}-igw"
  }
}

resource "aws_route" "internet_access" {
  route_table_id      = "${aws_vpc.tfb.main_route_table_id}"
  destination_cidr_block = "0.0.0.0/0"
  gateway_id          = "${aws_internet_gateway.tfb.id}"
}

resource "aws_subnet" "public" {
  vpc_id           = "${aws_vpc.tfb.id}"
  cidr_block       = "${var.public_subnet}"
  tags {
    Name = "${var.name}-public"
  }
}
```

 **TIP** Our VPC module is very simple. It does not expose anywhere near the complexity of a complete VPC configuration. For a more fully featured module take a look at the [Terraform Community VPC module](#).

You can see we've added a number of new resources but what you can't see is an `aws` provider definition. This is because we don't need one. The module will, by default, inherit the provider configuration from the `web.tf` file and use that to connect to AWS.

Module provider inheritance

However if you have multiple providers specified in the `web.tf` file, as we saw earlier using the `alias` attribute, then you must explicitly tell the module which provider to use!

 **TIP** This change occurred in Terraform 0.11 and later.

So if our `web.tf` file has defined two providers:

Listing 3.48: Multiple aliased providers

```
provider "aws" {  
    alias  = "use1"  
    region = "us-east-1"  
}  
  
provider "aws" {  
    alias  = "usw2"  
    region = "us-west-2"  
}
```

One aliased `use1` and one aliased `usw2` then you must explicitly tell the module which provider to use. We do this using the `providers` meta-parameter.

Listing 3.49: The vpc module's default variables

```
module "vpc" {
  source = "./vpc"
  name   = "web"
  cidr   = "10.0.0.0/16"
  public_subnet = "10.0.1.0/24"
  providers = {
    "aws" = "aws.use1"
  }
}
```

Here we've specified a new attribute for our module: `providers`. The `providers` attribute contains a list of each of the providers our module uses and the alias name of the specific provider to use. So in this case for our `aws` provider the `vpc` module will use the `aws.use1` definition of the provider.

 **TIP** You can read more about module provider inheritance in the [modules documentation](#).

Our module resources

So back to our module's resources. You can see that we've used a series of new AWS resource types: the VPC itself, gateways, routes, and subnets. Let's look at the `aws_vpc` resource in more detail.

Listing 3.50: The aws_vpc resource

```
resource "aws_vpc" "tfb" {
  cidr_block      = "${var.cidr}"
  enable_dns_hostnames = "${var.enable_dns_hostnames}"
  enable_dns_support    = "${var.enable_dns_support}"
  tags {
    Name = "${var.name}"
  }
}
```

We've called our VPC resource `aws_vpc.tfb`.¹ Inside the resource, we've passed some of our variables in—for example, the `var.cidr` variable—to be interpolated and to configure the resource. We need to ensure each of these variables is defined and that they're either populated in the `module` block or that a default exists in the module for them.

In the `main.tf` file we also configure a series of other resources using a mix of variables and resource references as the values of our attributes—for example, in the `aws_subnet` resource:

Listing 3.51: The aws_subnet resource

```
resource "aws_subnet" "public" {
  vpc_id          = "${aws_vpc.tfb.id}"
  cidr_block      = "${var.public_subnet}"
  tags {
    Name = "${var.name}-public"
  }
}
```

Our resource is named `aws_subnet.public` and references attributes from earlier

¹tfb for The Terraform Book

configured resources. For example, the `vpc_id` attribute is populated from the ID of the `aws_vpc.tfb` resource we created earlier in the module.

```
 ${aws_vpc.tfb.id}
```

Another interesting attribute is the `Name` tag we've created. Here we've used the interpolated `var.name` variable inside a string.

```
 ${var.name}-public
```

This will create a value that combines the value of the `var.name` variable with the string `-public`.

The combination of these resources will create an Amazon VPC with access to the Internet, internal routing, and a single public subnet, specified in [CIDR notation](#).

Outputs

Lastly, we need to specify outputs from our module. This is essentially the API response from using the module. They can contain useful data like the IDs of resources created or other configuration that we might want to use outside of the module to configure other resources. To add these outputs we use a new construct called an `output`.

The `output` construct can be used in any Terraform configuration, not just in modules. It is a way to highlight specific information from the attributes of resources we're creating. This allows us to selectively return critical information to the user or to another application rather than returning all the possible attributes of all resources and having to filter the information down.

Let's add some outputs to the end of our `interface.tf` file.

Listing 3.52: The vpc module outputs

```
output "public_subnet_id" {
  value = "${aws_subnet.public.id}"
}

output "vpc_id" {
  value = "${aws_vpc.tfb.id}"
}

output "cidr" {
  value = "${aws_vpc.tfb.cidr_block}"
}
```

Here one of our outputs is the VPC ID. We've called the output `vpc_id`. The output will return the `aws_vpc.tfb.id` attribute value from the `aws_vpc.tfb` resource we created inside the module.

You can see that, like a variable, an `output` is configured as a block with a name. Each `output` has a value, usually an interpolated attribute from a resource being configured.

 **TIP** Since Terraform 0.8, you can also add a `description` attribute to your outputs, much like you can for your variables.

Outputs can also be marked as containing sensitive material by setting the `sensitive` attribute.

Listing 3.53: The vpc module outputs

```
output "public_subnet_id" {
    value      = "${aws_subnet.public.id}"
    sensitive = true
}
```

When outputs are displayed—for instance, at the end of the application of a plan—sensitive outputs are redacted, with `<sensitive>` displayed instead of their value.

 **NOTE** This is purely a visual change. The outputs are not encrypted or protected.

We'll see how to use these outputs inside our stack configuration shortly.

 **NOTE** We recommend using a naming convention for Terraform files inside modules. This isn't required but it makes code organization and comprehension easier. We use `interface.tf` for variables and outputs and `main.tf` for resources.

With that our module is complete. Now let's see it at work.

Using our module

Back in our `web.tf` configuration file we've already defined our `module` block.

Listing 3.54: The vpc module block revisited

```
module "vpc" {
  source = "./vpc"
  name   = "web"
  cidr   = "10.0.0.0/16"
  public_subnet = "10.0.1.0/24"
}
```

Inside the `module` block we've passed in all of our required variables and when Terraform runs it will:

1. Load the module code.
2. Pass in the variables.
3. Create all the resources in the module.
4. Return the outputs.

Let's see how we can use those outputs in our stack's configuration.

We have the `aws_elb.web` or AWS Elastic Load Balancer resource. To configure it we need to provide at least one piece of information from our `vpc` module: the subnet ID of the subnet to which our load balancer is connected.

Listing 3.55: Our web aws_elb resource

```
resource "aws_elb" "web" {
  name          = "web-elb"
  subnets       = ["${module.vpc.public_subnet_id}"]
  security_groups = ["${aws_security_group.web_inbound_sg.id}"]
  listener {
    instance_port      = 80
    instance_protocol  = "http"
    lb_port            = 80
    lb_protocol        = "http"
  }
  instances       = ["${aws_instance.web.*.id}"]
}
```

We've specified a module output for the value of the `subnets` attribute:

```
["${module.vpc.public_subnet_id}"]
```

This variable is an output from our `vpc` module. Module outputs are prefixed with `module`, the module name—here `vpc`—and then the name of the output. You can access any output you've defined in the module.

It's important to remember that a module's resources are isolated. You only see the data you define. You must specify outputs for any attribute values you want to expose from them.

Like resources, modules automatically create dependencies and relationships. For example, by using the `module.vpc.public_subnet_id` output from the `vpc` module we've created a dependency relationship between the `aws_elb.web` resource and the `vpc` module.

 **TIP** Since Terraform 0.8, you can also specify the `depends_on` meta-parameter to explicitly create a dependency on a module. You can reference a module via

name, for example `module.vpc`.

We can use this combination of variables and outputs as a simple API for our modules. It allows us to define standard configuration in the form of modules and then use the outputs of those modules to ensure standardization of our resources.

 **TIP** The fine folks at Segment.io have released an excellent tool called `terraform-docs`. The `terraform-docs` tool reads modules and produces Markdown or JSON documentation for the module based on its variables and outputs.

Getting our module

Before you can use a module in your configuration, you need to load it or `get` it. You do that from the `~/terraform/web` directory, via the `terraform get` command.

Listing 3.56: The Terraform `get` command

```
$ pwd  
~/terraform/web  
$ terraform get  
Get: file:///Users/james/terraform/web/vpc
```

This gets the module code and stores it in the `.terraform/modules` directory inside the `~/terraform/web` directory.

If you change your module, or the module you're using has been updated, you'll need to run the `get` command again, with the `-update` flag set.

Listing 3.57: Updating a module

```
$ terraform get -update
```

If you run the `terraform get` command without the `-update` flag, Terraform will not update the module.

Moving our module to a repository

Currently our `vpc` module is located in our local filesystem. That's cumbersome if we want to reuse it. Let's instead move it to a GitHub repository.

You'll need a GitHub account to do this. You can [join GitHub on their site](#). There's also some [useful sign-up documentation](#) available.

After we've created our GitHub account, we can [create a new GitHub repository](#). We're calling ours `turnbullpress/tf_vpc`.



NOTE You'd use your own GitHub username and repository name.

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner / 

Great repository names are short and memorable. Need inspiration? How about [silver-potato](#).

Description (optional)
A VPC module for The Terraform book

 **Public**
Anyone can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾ Add a license: **None** ▾

Create repository

Figure 3.2: Creating a GitHub repository

Let's add a `README.md` file to our `~/terraform/web/vpc` directory to tell folks how to use our module.

Listing 3.58: The README.md file

```
# AWS VPC module for Terraform

A lightweight VPC module for Terraform.

## Usage

module "vpc" {
  source = "github.com/turnbullpress/tf_vpc"
  name   = "vpc_name"
  cidr   = "10.0.0.0/16"
  public_subnet = "10.0.1.0/24"
}

See `interface.tf` for additional configurable variables.

## License

MIT
```

Let's create a `.gitignore` file to ensure we don't accidentally commit any state or variables values we don't want in our module repository.

Listing 3.59: Creating a .gitignore file

```
$ echo ".terraform/" >> .gitignore
$ echo "terraform.tfvars" >> .gitignore
$ git add .gitignore
```

We can then commit and push our `vpc` module.

Listing 3.60: Committing and pushing our vpc module

```
$ pwd  
~/terraform/web/vpc  
$ git add .  
$ git commit -m "First commit of VPC module"  
$ git tag -a "v0.0.1" -m "First release of vpc module"  
$ git remote add origin git@github.com:turnbullpress/tf_vpc.git  
$ git push -u origin master --tags
```

Here we've added all the `vpc` module files and committed them. We've also tagged that commit as `v0.0.1`. We add the newly created remote repository and push up our code and tag.

Now we can update our `module` configuration in `web.tf` to reflect the new location of the `vpc` module.

Listing 3.61: Updating our vpc module configuration

```
module "vpc" {  
  source = "github.com/turnbullpress/tf_vpc.git?ref=v0.0.1"  
  name   = "web"  
  cidr   = "10.0.0.0/16"  
  public_subnet = "10.0.1.0/24"  
}
```

We'll need to get our module again since we've changed its source.

Listing 3.62: Getting the new vpc module

```
$ terraform get  
Get: git::https://github.com/turnbullpress/tf_vpc.git?ref=v0.0.1
```

Any time we want to use the `vpc` module, we can now just reference [the module on GitHub](#). This also means we can manage multiple versions of the module—for example, we could create `v0.0.2` of the module, and then use the `ref` parameter to refer to that.

```
git::https://github.com/turnbullpress/tf_vpc.git?ref=v0.0.2
```

This allows us to test a new version of a module without changing the old one.

Counts and counting

Let's go back to our `web.tf` file and look at our remaining resources. We know we want to create two EC2 instances in our stack. We know we can only specify a resource named `aws_instances.web` once. It doesn't make sense to duplicate the resource with a new name, especially if its configuration is otherwise identical.

In a traditional programming language this is when you'd break out a `for` loop. Terraform has a solution for this: counts. A count is another meta-parameter and can be added to any resource.

 **TIP** Terraform has [a number of meta-parameters available](#).

You add a `count` to a resource to have Terraform iterate and create the number of resources equal to the value of the `count`. Let's look at how `count` works with our `aws_instances.web` resource.

Listing 3.63: The aws_instances count

```
resource "aws_instance" "web" {
    ami           = "${lookup(var.ami, var.region)}"
    instance_type = "${var.instance_type}"
    key_name      = "${var.key_name}"
    subnet_id     = "${module.vpc.public_subnet_id}"
    associate_public_ip_address = true
    user_data     = "${file("files/web_bootstrap.sh")}"
    vpc_security_group_ids = [
        "${aws_security_group.web_host_sg.id}",
    ]
    count         = 2
}
```

We've added the `count` meta-parameter and specified a value of `2`. When Terraform creates the `aws_instances.web` resource it will iterate and create two of these resources. It'll create each resource with the index of the count suffixed to the resource name, like so:

- `aws_instance.web.0`
- `aws_instance.web.1`

We can now refer to these resources and their attributes using these names. For example, to access the `id` of one of these instances we'd use:

`aws_instance.web.0.id`

Sets of counted resources using `splat`

Sometimes we want to refer to the set of resources created via a `count`. To do this Terraform has a `splat` syntax: `*`. This allows us to refer to all of these resources in a variable. Let's see how that works in the `aws_elb.web` resource.

Listing 3.64: The aws_elb resource

```
resource "aws_elb" "web" {
    name          = "web-elb"
    subnets       = ["${module.vpc.public_subnet_id}"]
    security_groups = ["${aws_security_group.web_inbound_sg.id}"]

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

    instances = ["${aws_instance.web.*.id}"]
}
```

The `instances` attribute in our `aws_elb.web` resource needs to contain a list of the IDs of all the EC2 instances that are connected to our load balancer. To provide this we make use of the splat syntax like so:

```
["${aws_instance.web.*.id}"]
```

The value assigned to the attribute is a list interpolated from the IDs of all our EC2 instances.

Setting values with count indexes

We can also use the `count` to allow us to specify different values for an attribute for each iteration of a resource. We do this by referring to the index of a count in a variable.

Let's take a look at how this works. We start by declaring a list variable with a value for each iteration.

Listing 3.65: Using count indexes in variables.tf

```
variable "instance_ips" {  
    description = "The IPs to use for our instances"  
    default = ["10.0.1.20", "10.0.1.21"]  
}
```

We've defined a new list variable called `instance_ips` that contains two IP addresses in our VPC subnet. We're going to match the index of the count with the relevant element of the list.

Listing 3.66: Looking up the count index

```
resource "aws_instance" "web" {  
    . . .  
    private_ip      = "${var.instance_ips[count.index]}"  
    . . .  
    count = "${length(var.instance_ips)}"  
}
```

You can see we've updated our `aws_instance.web` resource to add the `private_ip` attribute. The value of the attribute uses the list element lookup we saw earlier in this chapter.

For the element lookup we specify the name of the variable we just defined: `var.instance_ips` and the index of the count using `count.index`. The `count.index` is a special function on the `count` meta-parameter to return the index.

When each `aws_instance.web` resource is created, the matching list element will be retrieved with the index from the `count.index`. The `var.instance_ips` will

return each value, and the instance will get the correct IP address, hence:

- `aws_instance.web.0` will get the IP address `10.0.1.20`.
- `aws_instance.web.1` will get the IP address `10.0.1.21`.

This makes it easier to customize individual resources in a collection.

You'll notice we also changed the value of the `count` meta-parameter. Instead of hard-coding a number, we used the length of the `var.instance_ips` list as the value of the `count`. We know the `var.instance_ips` list needs to have an IP address for each instance otherwise instance creation will fail. So we know that we can only have as many instances as the number of elements in this list. The `length function` allows us to count the element in this list and return an integer, in our case `2`. We can use this to populate the `count` attribute. This means we can increment the number of instances created by just adding new private IP addresses, rather than having to change and track the instance count in two places.

Listing 3.67: Using the length function

```
resource "aws_instance" "web" {  
    . . .  
    count = "${length(var.instance_ips)}"  
}
```

We can also use the `count.index` in other places. For example, to add a unique name to our EC2 instances we could do the following:

Listing 3.68: Naming using the count.index

```
resource "aws_instance" "web" {  
    . . .  
  
    tags {  
        Name = "web-${format("%03d", count.index)}"  
    }  
    count = "${length(var.instance_ips)}"  
}
```

This will populate the `Name` tag of each instance with a name based on the `count.index`. We've also used a new function called `format`. The `format` function formats strings according to a specified format. Here we're turning the `count.index` of `0` or `1` into a three-digit number.

The `format` function is essentially a `sprintf` and is a wrapper around Go's `fmt` library syntax. So `%03d` is constructed from `0`, indicating that you want to pad the number to the specified width with leading zeros. Then `3` indicates the width that you want, and `d` specifies a base 10 integer. The flags together will pad single digits with a leading `0` but ignore numbers larger than three digits in length.

This will produce `Name` tags `web-000` and `web-001` respectively. Having a `web-000` is a bit odd though. It comes from `count`'s zero index. Alternately we can use some math in our interpolated string like so:

Listing 3.69: Interpolated math

```
tags {  
    Name = "web-${format("%03d", count.index + 1)}"  
}
```

This would add one to every `count.index` value producing the tags `web-001` and

`web-002` respectively. We can do other math: subtract, multiple, divide, etc., on any integer or float variables.

We can also iterate through list elements with `count.index`. Let's create a list variable with some tags we'd like to add to our instances.

Listing 3.70: AWS owner tags in variables.tf

```
variable "owner_tag" {
    default = ["team1", "team2"]
}
```

We'd like to distribute our instances between these two tag values in `web.tf`.

Listing 3.71: Splitting up the count instances

```
resource "aws_instance" "web" {
    ...
    tags {
        Owner = "${var.owner_tag[count.index]}"
    }
    count   = "${length(var.instance_ips)}"
}
```

This returns the element matching the `count.index` from the specified list variable. When we create the resources, one instance will be tagged `team1` and the second `team2`.

If we specify more instances than the number of elements in our list, then Terraform will fail with an error like:

Listing 3.72: Count exhausted failure

```
* index 2 out of range for list var.owner_tag (max 2) in:  
${var.owner_tag[count.index]}
```

Wrapping counts with the element function

We can, however, cause Terraform to wrap the list using [the element function](#). The `element` function pulls an element from a list using the given index and wraps when it reaches the end of the list.

Let's update our code to do that.

Listing 3.73: Wrapping the count instances list

```
resource "aws_instance" "web" {  
    . . .  
  
    tags {  
        Owner = "${element(var.owner_tag, count.index)}"  
    }  
    count    = "${length(var.instance_ips)}"  
}
```

Now, if our `var.instance_ips` variable had 12 elements, then our `count` will create 12 instances. Terraform would select each element then wrap to the start of the list and select again. This way we'd end up with six instances tagged with `team1` and six instances tagged with `team2`.

Conditionals

The `count` meta-parameter also allows us to explore Terraform's [conditional logic](#). Terraform has a [ternary operation](#) conditional form.

 **NOTE** Conditional logic was introduced in Terraform 0.8. It will not work in earlier releases.

A ternary operation looks like this:

Listing 3.74: A ternary operation

```
condition ? true : false
```

We specify a condition, followed by a `?`, and then the result to return if the condition is true or false, separated by `:`.

Let's see how we might use a conditional to set the `count` meta-parameter as an alternative to the methods we've seen thus far.

Listing 3.75: Using ternary with count

```
variable "environment" {
  default = "development"
}

resource "aws_instance" "web" {
  ami           = "${lookup(var.ami, var.region)}"

  ...
  count         = "${var.environment == "production" ? 4 : 2}"
}
```

Here we've set a variable called `environment` with a default of `development`. In our resource we've configured our `count` attribute with a conditional. If the `var.environment` variable equals `production` then launch 4 instances, if it is the default of `development`, or any other value, then only launch 2 instances.

The condition can be any interpolation: a variable, a function, or even chaining another conditional. The true or false values can also return any interpolation or valid value. The true and false values must return the same type though.

The condition supports a bunch of operators. We've already seen equality, `==`, and Terraform supports the opposite operator `!=` for inequality. It also supports numeric comparisons like greater or less, `>` and `<`, and the related `>=` and `<=`. It also supports Boolean logic like: `&&`, `||` and unary `.`.

We don't have to use conditionals with just `count` though. They work on any resource or module attribute, for example:

Listing 3.76: A conditional attribute

```
module "vpc" {  
    . . .  
  
    cidr = "${var.region} != "us-east-1" ? "172.16.0.0/12" :  
          "172.18.0.0/12"  
  
    . . .  
}
```

Here we're setting the value of the `cidr` attribute using a ternary conditional. If the `var.region` variable is not equal to `us-east-1` then use the CIDR of `172.16.0.0/12`. If it is equal then use `172.18.0.0/12`.

 **TIP** You can read more about conditionals in [their documentation](#).

Locals

Terraform also has the concept of local value configuration. Local values assign a name to an expression, essentially allowing you to create repeatable function-like values.

 **NOTE** Local values have been available since Terraform version 0.10.3.

We define local values in `locals` blocks.

Listing 3.77: A local definition

```
locals {  
    instance_ip_count = ${length(var.instance_ips)}  
}
```

Here we've created a local value from the application of the `length` function to our `var.instance_ips` variable. This assigns to the resulting count of IPs in that variable to a local value of `instance_ip_count`. We can then use this local value in our resources without needing to repeat the function, for example:

Listing 3.78: Using a local in a resource

```
resource "aws_instance" "web" {  
    ...  
  
    tags {  
        Owner = "${element(var.owner_tag, count.index)}"  
    }  
    count    = instance_ip_count  
}
```

Local expressions can refer to or use previously defined locals too but can't be self-referential. For example, you can't use a local within the expression that defines that local.

 **TIP** A local is only available in the context of the module it is defined in. It will not work cross-module.

You can specify one or many `locals` blocks in a module. We'd recommend grouping them together for maintainability. If you use more than one `locals` block in a module then the names of the locals defined must be unique across the module.

Now let's look at provisioning some application configuration on our EC2 instances.

Provisioning our stack

Provisioning is the process of adding configuration, packages, applications, and services to the infrastructure we're creating. It usually involves making more granular changes to our infrastructure than we do with Terraform—for example, installing Apache on an EC2 instance. For complex provisioning we're likely to hand off the task to a dedicated tool like Puppet, Chef, or Ansible. For our stack, however, we're going to do some simple provisioning using [EC2 user data](#). With user data, you can specify some commands or actions that should be run when the EC2 instance is launched.

 **TIP** We'll learn more about provisioning and integration with configuration management tools in Chapter 4.

To make use of user data in Terraform we add the `user_data` attribute to our `aws_instance.web` resources in the `web.tf` file.

Listing 3.79: Adding user data to our instances

```
resource "aws_instance" "web" {  
    . . .  
    user_data = "${file("files/web_bootstrap.sh")}"  
    . . .  
    count      = "${length(var.instance_ips)}"  
}
```

We can see that the value of our `user_data` attribute is:

```
${file("files/web_bootstrap.sh")}
```

This uses a new function, `file`, to load the contents of a file as the value of an attribute. In this case we're loading a shell script called `web_bootstrap.sh` from a directory called `files`. The location of the `files` directory is relative to the current directory.

Let's create that directory and file now.

Listing 3.80: Creating the files directory

```
$ pwd  
~/terraform/web  
$ mkdir files  
$ cd files  
$ touch web_bootstrap.sh
```

Let's add some commands to the `web_bootstrap.sh` script.

Listing 3.81: The `web_bootstrap.sh` script

```
#!/bin/bash
sudo apt-get update
sudo apt-get install -y nginx
sudo service nginx start
```

Now when our instances launch Nginx will automatically be installed and started. We'll see the results of this when we apply our configuration.

 **TIP** It might take some time after the instance is launched to complete the installation process. Be patient! You can SSH into the instances to check the progress if required.

We're going to focus on more complex provisioning in Chapter 4. For now let's finish building our stack.

Finishing up our stack

At the bottom of our configuration file there's some security group configuration, providing security groups for some of the resources in our `web.tf` file. We're not going to show you this here because security group configuration is long and complex, but you can see it in [the book's source code](#).

Finally, let's add some outputs to our stack. We'll create a new file in the `~/terraform/web` directory called `outputs.tf` and populate it.

Listing 3.82: The web outputs.tf file

```
output "elb_address" {
    value = "${aws_elb.web.dns_name}"
}

output "addresses" {
    value = "${aws_instance.web.*.public_ip}"
}

output "public_subnet_id" {
    value = "${module.vpc.public_subnet_id}"
}
```

We've specified three outputs. These outputs will be displayed at the end of our `terraform apply` run. We've specified the DNS name of our Elastic Load Balancer resource and a list of the public IP addresses of our EC2 instances. We've used the splat syntax of `*` to return the `public_ip` values of all of the EC2 instances we're going to create.

We've also specified an output that returns one of the outputs of the `vpc` module: `public_subnet_id`. Outputs allow us to bubble up attributes from all of our configurations, including modules.

Now let's tidy up a few loose ends by better managing our configuration.

Committing our configuration

Now is a good time to commit our configuration to Git. This will allow us to go back to a known good state if we need to, and to potentially share our configuration with others.

 **NOTE** We're going to assume you know the basics of how Git works, and that you'll be regularly committing. This is just a reminder that it's a good idea to store your configuration in version control.

Listing 3.83: Committing our configuration

```
$ pwd  
terraform/web  
$ git add .  
$ git commit -a "First draft of our web stack"
```

This will commit our current Terraform configuration to our Git repository. We could then push our configuration upstream to a shared repository for others to use.

Validating and formatting

Don't forget the `terraform validate` and `terraform fmt` commands we introduced in Chapter 2. The `validate` command checks the syntax, validates your Terraform configuration files, and returns any errors. The `fmt` command neatly formats your configuration files. These are both very useful, especially as your configurations get more complex.

Now let's see what happens when we plan the stack.

Initializing Terraform

Before we go any further we need to initialize this Terraform configuration and download our provider. We do this using the `terraform init` command.

Listing 3.84: Initialiazing the web configuration

```
$ terraform init
```

This will get our `aws` provider and update our local configuration.

Planning our stack

Now that our stack's configuration and initialization is complete we can build it. But before we do, it's always a good idea to run `terraform plan` to ensure the configuration is going to do what we expect.

Listing 3.85: Planning our web configuration

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but
will not be persisted to local or remote state storage.

.

+ aws_elb.web
  availability_zones.#:                      "<computed>"
  connection_draining:                         "false"
  connection_draining_timeout:                 "300"
  cross_zone_load_balancing:                  "true"
  dns_name:                                    "<computed>"
  health_check.#:                            "<computed>"

.

+ module.vpc.aws_internet_gateway.tfb
  tags.%:          "1"
  tags.Name:       "web-igw"
  vpc_id:         "vpc-3c35d65a"

.

Plan: 9 to add, 0 to change, 0 to destroy.
```

We've included a sample of the `terraform plan` output. It shows us each resource that will be created, the values of the attributes that we know about now, and which of those will be computed when we apply the configuration. We can see that nine total resources will be created.

Now we're comfortable Terraform is going to do the right thing!

Applying our stack

Let's apply our execution plan and build our stack. To do this we run the `terraform apply` command.



NOTE In this command and future `terraform apply` commands we're going to skip the interactive prompt and assume you've typed `yes` to save some space in the output.

Listing 3.86: Applying our web stack

```
$ terraform apply
module.vpc.aws_vpc.tfb: Creating...
  cidr_block:          "" => "10.0.0.0/16"
  default_network_acl_id:    "" => "<computed>"
  default_route_table_id:    "" => "<computed>"

  ...
  subnets.#:           "" => "1"
  subnets.248256935:      "" => "subnet-8f0afcb3"
  "
  zone_id:            "" => "<computed>"

aws_elb.web: Creation complete

Apply complete! Resources: 9 added, 0 changed, 0 destroyed.

  ...
State path: terraform.tfstate

Outputs:

addresses = [
  54.167.183.26,
  54.167.186.170
]
elb_address = web-elb-1083111107.us-east-1.elb.amazonaws.com
public_subnet_id = subnet-ae6bacf5
```

It might take a couple minutes to create all of our configuration and finish. We should see that nine resources have been created. We can also see our outputs are the last items returned. We see the IP addresses of both our EC2 instances and the DNS name of our Elastic Load Balancer. We also see the ID of the `public` subnet we created with the `vpc` module.

If we want to see these outputs again, rather than applying the configuration again,

we can run the `terraform output` command.

Listing 3.87: Showing the outputs only

```
$ terraform output
addresses = [
    54.167.183.26,
    54.167.186.170
]
elb_address = web-elb-108311107.us-east-1.elb.amazonaws.com
public_subnet_id = subnet-ae6bacf5
```

 **TIP** Remember if you want to see the full list of all our resources and their attributes you can run the `terraform show` command.

We can also make use of this data in other tools by outputting it in a machine-readable JSON format. To do this we can use the `terraform output` command with the `-json` flag.

Listing 3.88: Outputs as JSON

```
$ terraform output -json
{
    "addresses": {
        "sensitive": false,
        "type": "list",
        "value": [
            "54.167.183.26",
            "54.167.186.170"
        ]
    },
    "elb_address": {
        "sensitive": false,
        "type": "string",
        "value": "web-elb-1083111107.us-east-1.elb.amazonaws.com"
    }
    "public_subnet_id": {
        "sensitive": false,
        "type": "string",
        "value": "subnet-ae6bacf5"
    }
}
```

We can consume this data in another service. For example, we could pass it to a provisioning tool such as Chef, Puppet, or Ansible.

Graphing our stack

Lastly, let's look at our stack's graph to see how the resources are interrelated. To output the graph we use the `terraform graph` command, pipe the result to a `.dot` file, and then convert it to an SVG file.

Listing 3.89: Graphing the web stack

```
$ terraform graph > web.dot
$ dot web.dot -Tsvg -o web.svg
```

We can then display the `web.svg` file.

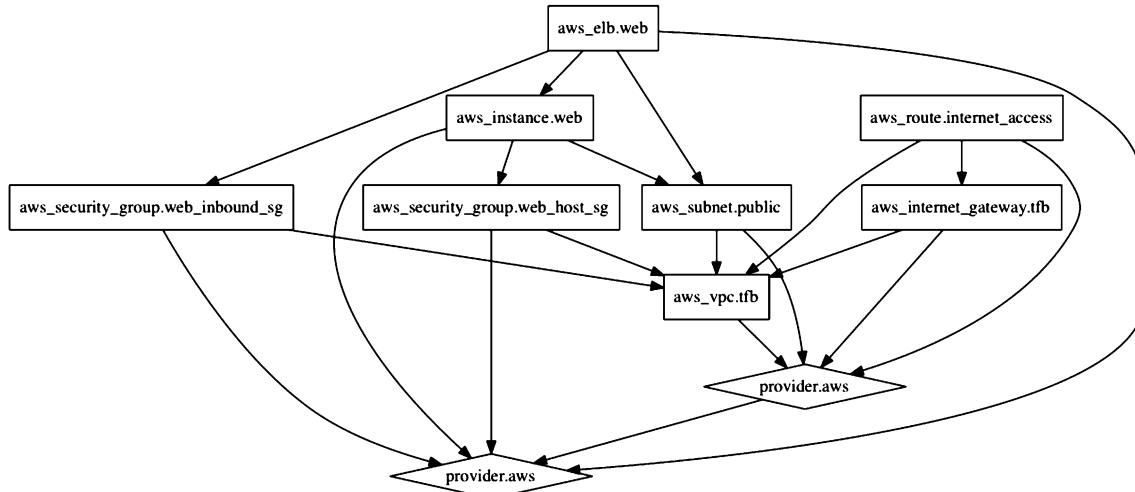


Figure 3.3: The graph of our web application stack

We can see two instances of the `aws` provider, one for our root configuration and the other for the `vpc` module. You can also see the relationships between the various resources we've just created.

Seeing the results

Finally, we can actually see the results of our Terraform plan being executed by viewing the URL of the Elastic Load Balancer we just created. We can take the DNS name of the `aws_elb.web` resource from the outputs, in our case:

`web-elb-1083111107.us-east-1.elb.amazonaws.com`

We can browse to that URL and, if everything works, see the default Nginx index page.

💡 TIP Remember, it might take a few minutes to complete the post-launch installation using our `user_data` script.



Figure 3.4: Our stack in action

Voilà—we've created a simple, easily repeatable infrastructure stack!

💡 TIP In addition to building a stack from your configuration, you can do the reverse and import existing infrastructure. You can read more about the [import](#) process in the Terraform documentation.

Destroying the web stack resources

If you're done with your web stack you can then destroy it (and stop spending any money on AWS resources) with the `terraform destroy` command.

Listing 3.90: Destroy our web stack

```
$ 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: yes

module.vpc.aws_vpc.tfb: Refreshing state... (ID: vpc-a22c10c5)

.

module.vpc.aws_vpc.tfb: Destruction complete

Destroy complete! Resources: 9 destroyed.
```

Now our web stack has been destroyed.

Summary

In this chapter we've put our burgeoning Terraform knowledge into action to build a simple web application with a load balancer. We've been introduced to the concept of parameterizing our configuration, allowing us to be more flexible in how we build our configuration. We've explored the types of variables available to us and how to use them.

We were introduced to modules, Terraform's approach to reusable infrastructure. We've learned how to build and use modules in our configuration. We've also

learned about some of Terraform's meta-parameters. Finally, we learned about outputs and how to make use of them.

In the next chapter we'll learn more about how to provision software with Terraform, including how to connect it to existing provisioners like Puppet and Chef.

Chapter 4

Provisioning with Terraform

In Chapter 3 we saw some simple provisioning, installing Nginx on our web stack. This made use of [AWS EC2 user data](#). This “run a shell script” when the compute resource launches is configuration management and provisioning at its most basic. It’s not very flexible or repeatable.

Indeed, like many organizations, you likely have a configuration management tool like Puppet, Chef, or Ansible installed. This means you have existing configuration management modules, roles, or cookbooks that already know how to build your applications and services.

Terraform isn’t designed to replace your configuration management tool—rather, it’s made to integrate with it. Terraform’s sweet spot is the management of your infrastructure components rather than the nitty-gritty of your Apache virtual hosts or firewall configuration. In this chapter, we’re going to learn how to use Terraform’s [provisioners](#). We can use provisioners to do simple bootstrapping, installation, and configuration tasks. They are especially ideal to install and manage a configuration management tool that can further configure your hosts and applications.

 **TIP** Terraform is not a configuration management tool. Really, it isn't. Use it to set up your configuration management tool; do not use it to build your hosts.

What does a provisioner do?

When you create and delete a resource, you can run [provisioners](#) in those resources to configure them. This can be used to bootstrap the resource—for example, to add it to a cluster or to update a network device. Provisioners can also trigger other tools, like configuration management tools or patch and update systems.

 **TIP** In releases prior to Terraform 0.9 you could only provision when creating resources.

The most important piece of information here is that provisioners only run when a resource is *created* or *destroyed*. They do not run when a resource is changed or updated. This means that if you want to trigger provisioning again, you will need to destroy and recreate the resource, which is often not convenient or practical. Again, Terraform's provisioning is not a replacement for configuration management.

Let's look at an example of a provisioner at work by installing a Puppet agent.

Provisioning an instance

Let's take our web stack configuration from Chapter 3 and add provisioning to it. Provisioning can be added to any resource that supports making an SSH or Windows Remote Management (WinRM) connection—generally these are going to be compute resources of some kind.

 **NOTE** To provision you'll need to be able to make an appropriate WinRM or SSH connection from the machine running Terraform to the resource being provisioned. In AWS, we'd need to have SSH access allowed in our security groups.

We're going to do three things to our new instances:

1. We're going to install the configuration management tool Puppet.
2. We're going to provision an Nginx server and configuration with Puppet.
3. We're going to install a custom `index.html` for each instance. This will mimic the installation of a web application or website.

 **TIP** Another approach is to bake your provisioning or configuration management tool into your compute images—for example, by using a tool like [Packer](#) to create virtual machine images or AMIs. This reduces the requirement to do provisioning with Terraform.

We'll add our provisioning to the `aws_instance.web` instances we created in Chapter 3. Let's look at that resource now.

Listing 4.1: Adding a provisioner to an AWS instance

```
resource "aws_instance" "web" {
    ami          = "${lookup(var.ami, var.region)}"
    ...
    count        = "${length(var.instance_ips)}"
    connection {
        user      = "ubuntu"
        private_key = "${file(var.key_path)}"
    }
    provisioner "file" {
        content = "${element(data.template_file.index.*.rendered,
count.index)}"
        destination = "/tmp/index.html"
    }
    provisioner "remote-exec" {
        script = "files/bootstrap_puppet.sh"
    }
    provisioner "remote-exec" {
        inline = [
            "sudo mv /tmp/index.html /var/www/html/index.html"
        ]
    }
}
```

In our configuration we've removed the `user_data` attribute and replaced it with some new blocks of configuration.

You can see we've added new blocks inside our resource: a `connection` block and several `provisioner` blocks.

Connection block

The `connection` block sets various configuration options for the SSH or WinRM connections that our provisioners use. The provisioners provide some defaults, but in some cases we want to override those defaults.

A good example of this are the Amazon Web Services Ubuntu AMIs. Provisioners using an SSH connection default to a connecting user of `root`, but Ubuntu AMIs disable login via this user name. Instead they use the `ubuntu` user. Using the `connection` block we've overridden the `user` attribute to ensure all connections to these instances connect with the `ubuntu` user.

Listing 4.2: Connection block attributes

```
connection {  
    user = "ubuntu"  
    private_key = "${file(var.key_path)}"  
}
```

We've also specified the `private_key` attribute which tells Terraform where to find the AWS key we've specified for our instances. Here it uses the `file` function to reference a location specified in a variable: `var.key_path`.

Listing 4.3: The `key_path` variable

```
variable "key_path" {  
    default = "/Users/james/.ssh/james_aws"  
}
```

This would tell Terraform to load the private key file located at `/Users/james/.ssh/james_aws` and to use that for the SSH connections to our instances when provisioning.

 **TIP** You could also add your AWS key to `ssh-agent` so that Terraform can find it.

You can find a full list of `connection` options in the [connection block documentation](#).

The file provisioner

Now we come to the first of our three `provisioner` blocks. Provisioners are the exception to the rule for Terraform resources: they are executed in the sequence they are specified. This means you can daisy chain provisioning actions and trust them to work in the order specified.

Our first `provisioner` block uses the `file` provisioner. The `file` provisioner allows you to upload content, templates, files, or directories to the remote host. Let's see it in action now.

Listing 4.4: Adding the file provisioner to an AWS instance

```
resource "aws_instance" "web" {
    ...
    provisioner "file" {
        content = "${element(data.template_file.index.*.rendered,
        count.index)}"
        destination = "/tmp/index.html"
    }
    ...
}
```

Our `file` provisioner is being used here to upload a template file, a new `index.html` that our web stack is going to serve. We're going to specify a unique `index.html` for each instance created. To create this template we've specified the `content`

attribute. We've set the value of the `content` attribute to a new type of variable: [a data source](#).

Data sources provide read-only data that can be used in your configuration. Data sources are linked to providers. Not every provider has data sources—generally they exist if there are sources of information that are useful in the configuration managed by the provider. For example, the `aws` provider has a data source called `aws_ami` that allows you to search for and return the IDs of specific AMIs.



TIP We'll see more [data sources](#) later in the book.

We define data sources like we define resources. Let's define one for our template using the `template` provider. The `template provider` isn't connected to a specific infrastructure service, rather it allows us to work with template files.

Let's see it now.

Listing 4.5: A template provider data source

```
data "template_file" "index" {
  count    = "${length(var.instance_ips)}"
  template = "${file("files/index.html.tpl")}"
  vars {
    hostname = "web-${format("%03d", count.index + 1)}"
  }
}
```

Our data source is defined with a `data` block that looks like our regular resources. It has a resource type: `template_file`. The `template` prefix indicates that it's a data source from the `template` provider. The data source also has a name: `index`. Like resources, this combination of type and name needs to be unique in our

configuration.

 **TIP** The `template` provider only has two resources: `template_file`, which creates template files, and `template_cloudinit_config`, which allows you to template `cloud-init` configurations.

The `template_file.index` data source contains three attributes. The first is the `count` attribute. We're going to use the length of the `var.instance_ips` variable as its value. As with a resource, the `count` attribute allows us to iterate through the data source. We can then generate a unique template for each instance using the `count.index`.

The second attribute is `template`, which specifies the template to render. We've used the `file` built-in function to specify the location of our template. In our case the `index.html.tpl` template file is in our `files` directory.

 **NOTE** The `.tpl` extension is not required but useful to identify that this is a template file.

The third attribute, `vars`, lists any variables we want to pass to our template. Terraform templates are very simple. They can only contain string primitives and don't support lists or maps. They do not support control flow logic like optional content, `if/else` clauses, or loops.

 **TIP** This is another reason you should use a configuration management tool. Almost all of them have fully featured template generation and syntax.

We've created one variable, called `hostname`. The `hostname` variable is constructed using the same function we used to set the `Name` tag of our instance:

Listing 4.6: The `hostname` variable

```
"web-${format("%03d", count.index + 1)}"
```

This will create a `hostname` value based on `web` and a formatted `count.index` for each iteration of the template. In our case, with `count` of `2`, we'd get two iterations: `web-001` and `web-002`.

When our configuration is applied this variable will be populated and passed into the template.

Let's look at the `index.html.tpl` template file in the `files` directory.

Listing 4.7: The `files/index.html.tpl` template

```
<html>
  <head>
    <title>Web service - ${hostname}</title>
  </head>
  <body>
    <h1>The Terraform Book web service running on ${hostname}</h1>
  </body>
</html>
```

We can see we've used our variable, `${hostname}`, in two places in the template. As each instance is created and each template iterated, that variable will be interpolated and a new file will be created. For our first instance the `index.html` will look like:

Listing 4.8: The populated index.html

```
<html>
  <head>
    <title>Web service - web-001</title>
  </head>
  <body>
    <h1>The Terraform Book web service running on web-001</h1>
  </body>
</html>
```

Now lets actually use our template in the `file` provisioner.

Listing 4.9: Using the template in a provisioner

```
provisioner "file" {
  content = "${element(data.template_file.index.*.rendered,
count.index)}"
  destination = "/tmp/index.html"
}
```

The first attribute, `content`, is the actual file content we want to provision. We've specified [an element function](#), which we introduced in Chapter 3, as the value of the `content` attribute. The `element` function pulls an element from a list using a given index. We've specified our data source as the list and the `count.index` as the element index. This will create an individual template for each element of the list.

We specify a data source much like we specify a resource. We start with `data` as the prefix, then the type of data source—here, `template_file`. We then specify the name of the resource, `index`. Remember how we specified a `count` in the `data.template_file.index` data source? This means we can use the splat syntax, `*`, to create a list of all the possible templates. The last part of the data source

is `rendered`. The `rendered` method is a special call on the `template_file` data source. It returns the rendered template.

Lastly, we're then using our `count.index` as the index to return a specific iteration of the template.

You can see we're not actually passing around a variable between the instance resource and the template resource. Rather we're relying on the value of the `count.index` to ensure our template is uniquely provisioned. This is because of the sequence in which Terraform creates the dependency graph. The template is created when the resource is created, so we cannot use attribute values—for example, the instance's IP address or DNS name—in the template because they will not yet exist when the template is created.

The next attribute, `destination`, tells the `file` provisioner where to upload the file content. We're uploading to `/tmp/index.html`.

 **TIP** The `file` provisioner uses the permissions of the user we connect to the instance with. For us, this is the `ubuntu` user we specified in the `connection` block. This user must have permission to write to the chosen destination. If it does not have permission, the provisioner will fail.

So what happens when we create our instances? Let's walk through the process.

1. The `aws_instance.web` resource is processed and Terraform sees the value of the count from the length of the `var.instance_ips` variable—in our case `2`.
2. The first instance, `aws_instance.web.0`, gets created.
3. The provisioner is started as the resource is created, and Terraform grabs an iteration of the template, `data.template_file.index.0`. The `0` matches the

`count.index` of the current instance.

4. The `data.template_file.index.0` template is created. The `hostname` variable is populated as `web-001` and passed into the template file.
5. The `data.template_file.index.0` is rendered, returned to the `file` provisioner, and uploaded to `/tmp/index.html`.
6. The next instance is created, likely in parallel, etc.

A single template

If we didn't need a unique template for each instance, we could just specify `data.template_file.index.rendered` as the value of the `content` attribute, without the splat syntax or the `count` configuration.

Listing 4.10: A single template

```
data "template_file" "index" {  
    template = "${file("files/index.html.tpl")}"  
}  
  
resource "aws_instance" "web" {  
    . . .  
    connection {  
        user = "ubuntu"  
        private_key = "${file(var.key_path)}"  
    }  
    provisioner "file" {  
        content = "${data.template_file.index.rendered}"  
        destination = "/tmp/index.html"  
    }  
    . . .
```

You can see we've stripped out the `element` function, the counts, and the `hostname` variable. This would create a single template for each iteration of the instance,

rather than a unique one.

We'd then update the template file to remove the `hostname` variable.

Listing 4.11: The populated single index.html

```
<html>
  <head>
    <title>Web service</title>
  </head>
  <body>
    <h1>The Terraform Book web service</h1>
  </body>
</html>
```

 **TIP** Another useful place for the `template_file` data source is as the value of the `user_data` attribute. We can render a template file that will be executed to provision our hosts, along with any variables we might find useful.

Provisioning content

The `file` provisioner can also upload content or strings to a file on a remote resource without using a template at all—for example:

Listing 4.12: Uploading file content

```
provisioner "file" {
  content = "Instance ID: ${self.id}"
  destination = "/etc/instance_id"
}
```

Here we're populating a file, `/etc/instance_id`, with the contents of a `self.id` variable. What's `self`? It refers to the value of one of the resource's own attributes. An attribute name prefixed with `self` refers to the attribute in that resource. For example, `self.id` would refer to `id` attribute of an `\text{aws_instance}` resource. You can only use `self` variables in provisioners. They do not work anywhere else.

Provisioning files or directories

Finally, the `file` provisioner can also upload files or whole directories of files. To upload a file, we replace the `content` attribute with a `source` attribute.

Listing 4.13: Uploading a file

```
resource "aws_instance" "web" {  
    . . .  
    provisioner "file" {  
        source = "files/nginx.conf"  
        destination = "/tmp/nginx.conf"  
    }  
    . . .
```

Here we've specified a file, `nginx.conf`, located in the `files` directory, and will upload it to `/tmp/nginx.conf`.

If instead we want to upload a whole directory, we can specify the directory as the `source`:

Listing 4.14: Uploading a directory

```
resource "aws_instance" "web" {  
    . . .  
    provisioner "file" {  
        source = "files/"  
        destination = "/root"  
    }  
    . . .
```

This would upload the contents of the `files` directory locally to the `/root` directory on the remote instance. The destination path on the remote host must be specified as an absolute path so that Terraform can find it.

The `file` provisioner loosely follows the rules of the `rsync` tool and uses the presence or absence of a trailing `/` to determine its upload behavior.

If the `source` directory has a trailing `/`, the contents of the directory will be uploaded into the `destination` directory. So a `source` of `files/` will upload the contents of the `files` directory to the destination.

If the `source` directory doesn't have a trailing `/`, a new directory will be created inside the `destination` directory. So a `source` of `files` will upload the `files` directory to the `destination`, creating, for example, `/root/files`.

Remote execution provisioning

Our next `provisioner` block is a `remote-exec` provisioner.

Listing 4.15: The second provisioner

```
resource "aws_instance" "web" {
    ami          = "${lookup(var.ami, var.region)}"

    ...
    provisioner "remote-exec" {
        script = "files/bootstrap_puppet.sh"
    }
    ...
}
```

The `remote-exec` provisioner runs scripts or commands on a remote instance. It can run in three modes:

- Run a single script.
- Run a list of scripts in the order specified.
- Run a list of commands in the order specified.

Single-script mode

This `remote-exec` provisioner runs in single-script mode. It runs one script, located at `files/bootstrap_puppet.sh`. When it connects to the instance, the `remote-exec` provisioner uploads the script and then runs it on the remote host. Let's look at the `bootstrap_puppet.sh` script.

Listing 4.16: The bootstrap_puppet script

```
#!/bin/bash
curl https://apt.puppetlabs.com/puppet-release-xenial.deb -o
puppet-release-xenial.deb
sleep 15
sudo dpkg -i puppet-release-xenial.deb
sudo apt-get -qq update
sudo apt-get install -yq puppet-agent
sudo /opt/puppetlabs/bin/puppet module install puppet-nginx
sudo /opt/puppetlabs/bin/puppet module install puppetlabs-apt
cat >/tmp/nginx.pp << "EOF"
class{'nginx': }
EOF
sudo /opt/puppetlabs/bin/puppet apply /tmp/nginx.pp
```

Our script downloads the `puppetlabs-release` DEB package (which adds Puppet's package repositories to Apt) and installs it. It then refreshes our repository caches and installs the Puppet agent.

 **TIP** We could skip a lot of these steps by building an instance with an AMI that has Puppet pre-installed. You can easily build an AMI with a tool like [Packer](#).

It also replicates the functionality of our script in Chapter 3 and installs Nginx and a dummy virtual host, this time using a Puppet module and the `puppet apply` command. You could instead connect to a Puppet master and do the same thing.

 **TIP** The `remote-exec` provisioner has a counterpart: `local-exec`. The `local-exec` provisioner runs commands locally on the host running Terraform.

You can read about the `local-exec` provisioner [in its provisioner documentation](#).

Multiple scripts

If instead we had multiple scripts we wanted to run, we could update our provisioner to execute a list of scripts.

Listing 4.17: The second provisioner

```
resource "aws_instance" "web" {
    ami          = "${lookup(var.ami, var.region)}"

    ...
    provisioner "remote-exec" {
        scripts = [
            "files/bootstrap_puppet.sh",
            "files/another_script.sh",
            "files/a_third_script.sh"
        ]
    }
    ...
}
```

You can see we've replaced the `script` attribute with a `scripts` attribute. The `scripts` attribute takes a list of scripts to be uploaded and then executed on the resource. Here we'd run, in sequence:

1. `files/bootstrap_puppet.sh`
2. `files/another_script.sh`

3. files/a_third_script.sh

There is one shortcoming with both the single- and multiple-script execution modes: you can't pass any arguments to the scripts being run.

Remote script execution with arguments

If you do want to run a script with an argument, there's a workaround available. We use the `file` provisioner to first upload the script and then run it with `remote-exec` in its final mode, running inline commands.

Listing 4.18: Adding a remote execution script with arguments

```
resource "aws_instance" "web" {
  ...
  connection {
    user = "ubuntu"
    private_key = "${file(var.key_path)}"
  }
  provisioner "file" {
    source = "files/bootstrap_puppet_args.sh"
    destination = "/tmp/bootstrap_puppet_args.sh"
  }
  provisioner "remote-exec" {
    inline = [
      "chmod +x /tmp/bootstrap_puppet_args.sh",
      "/tmp/bootstrap_puppet_args.sh server=puppet.example.com"
    ]
  }
}
```

We've specified a `file` provisioner with a `source`: the location of the file we'd like to upload. In our case this is our `files/bootstrap_puppet_args.sh` script. It also has a `destination`, to which it'll upload the file.

We've also modified the `remote-exec` provisioner to use an `inline` block, which runs a list of specified commands in the order they are written. If one of those commands is a script, it does not get automatically uploaded. It assumes the script is already present on the resource, hence using the `file` provisioner to upload the script.

Here we're making our `bootstrap_puppet_args.sh` script executable, and then we're running it, with an argument of `server=puppet.example.com`. This gets around the inability of the `remote-exec` provisioner to accept scripts with arguments.

Our final provisioner

The last provisioner block also uses the `inline` script mode of the `remote-exec` provisioner.

Listing 4.19: The final provisioner block

```
resource "aws_instance" "web" {
    ami          = "${lookup(var.ami, var.region)}"

    ...
    provisioner "remote-exec" {
        inline = [
            "sudo mv /tmp/index.html /usr/share/nginx/html/index.html"
        ]
    }
}
```

Our final provisioner moves the template `index.html` file into our virtual host's root directory: `/usr/share/nginx/html/`.

Why do we need to do this? Remember that provisioners are executed with the privileges of the user with which we connect to the resource—in our case, `ubuntu`.

The `ubuntu` user has limited privileges and can't write to the `/usr/share/nginx/html/` directory. Instead we upload the template file to `/tmp`. Once the file is there we can use an inline command, prefixed with the `sudo` command, to step up our privileges and allow us to write the `index.html` file to the `/usr/share/nginx/html/` directory.

Now that we have a complete picture of our provisioning process, let's see it in action.

Destroying the old web stack resources

If we've been creating and destroying our web stack during this book, we might want to destroy it with the `terraform destroy` command before we continue. Remember provisioning only occurs when a resource is newly created. Starting from a clean slate is a good idea.

Listing 4.20: Destroy our web stack

```
$ 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: yes

module.vpc.aws_vpc.tfb: Refreshing state... (ID: vpc-a22c10c5)

...
module.vpc.aws_vpc.tfb: Destruction complete

Destroy complete! Resources: 9 destroyed.
```

Provisioning our web stack

Now let's recreate our stack with our provisioner enabled. Remember, you will need to be able to connect to your infrastructure via SSH. You'll also need the key pair—we're using the `james` key pair—with which you created the instances. You could also add that key pair to your SSH agent to ensure Terraform can find it and use it.

Listing 4.21: Running our first provisioning

```
$ terraform apply
module.vpc.aws_vpc.tfb: Creating...

.

aws_instance.web.0: Still creating... (20s elapsed)
aws_instance.web.0: Provisioning with 'remote-exec'...
aws_instance.web.0 (remote-exec): Connecting to remote host via
SSH...
aws_instance.web.0 (remote-exec):   Host: 54.152.161.48
aws_instance.web.0 (remote-exec):   User: ubuntu

.

aws_instance.web.0 (remote-exec): Connected!

.

aws_instance.web.0: Creation complete
aws_instance.web.1: Creation complete

.

Apply complete! Resources: 9 added, 0 changed, 0 destroyed.
```

Here you can see snippets from our `terraform apply` execution plan output. The execution will show you the provisioning steps, including any output from the

resources, as they are provisioned. This is useful to see if everything is working, and if anything breaks, it's a source of debugging information.

You'll see in the plan output that as soon as the `aws_instance.web.*` instances are created, our provisioner will try to connect to them. It'll continue to try to connect until it makes an SSH connection. It'll then run our three provisioners, one after another:

1. The `file` provisioner will render our template. The `.rendered` method contains the rendered template file. The `content` attribute will be populated with this template file and uploaded to `/tmp/index.html`.
2. Our `remote-exec` provisioner will run the `bootstrap_puppet.sh` script, install Puppet, and run `puppet apply` to install Nginx.
3. The last `remote-exec` provisioner will move the template `index.html` file from `/tmp` to `/usr/share/nginx/html/`. We do this because we need to use `sudo` to write the `index.html` into the new directory, as the `ubuntu` user doesn't have the right permissions.

The resulting `index.html` file for our first instance will look like:

Listing 4.22: The final populated index.html

```
<html>
  <head>
    <title>Web service - web-001</title>
  </head>
  <body>
    <h1>The Terraform Book web service running on web-001</h1>
  </body>
</html>
```

When our execution plan is completed this will result in two instances with Puppet installed and configured. Our Nginx server will be installed by Puppet, and a unique `index.html` file will be populated.

If we were to browse to our load balancer we'd see this:

C ⓘ web-elb-1441275151.us-east-1.elb.amazonaws.com

The Terraform Book Web service running on web-001

Figure 4.1: Our web stack custom index.html

NOTE It might take a few minutes to finish the provisioning and for AWS to register the load balancer and instance DNS. Don't panic if you can't connect to the load balancer immediately.

If we refresh the page, we'd see the output change to reflect each instance's custom `index.html` page.

TIP There are several other useful provisioners including a [Chef provisioner](#) and an [Ansible provisioner](#).

Failed execution of a provisioner

If the provisioning process fails then our execution plan will fail. A failure will look a bit like this:

Listing 4.23: A failed provisioning run

Error applying plan:

2 error(s) occurred:

```
* Script exited with non-zero exit status: 100  
* Script exited with non-zero exit status: 100
```

Terraform does not automatically rollback `in` the face of errors. Instead, your Terraform state file has been partially updated with any resources that successfully completed. Please address the error above and apply again to incrementally change your infrastructure.

The failed resources will be marked as tainted. Remember that Terraform always holds to the execution plan and refuses to roll back and delete resources that have failed. You can fix the issue and then rerun `terraform apply`, and it'll destroy and recreate the tainted resources and rerun provisioning.

 **TIP** There is a workaround for decoupling the destroy/recreate life cycle for provisioning. It involves using the `null_resource` provisioner, which allows you to [create centralized provisioning configuration](#) tied to triggers.

Summary

In this chapter we've learned how to use a variety of Terraform provisioners. These allow us to perform actions on our resources to provision them or allow

us to install and execute configuration management tools such as Puppet.

We've also been introduced to a new construct: data sources. These allow us to use data from providers to configure our resources. We'll see a few more data sources in subsequent chapters.

In the next chapter we'll learn about how to collaborate with others on your configuration. We'll also look at how to manage your state when multiple people are working on your configuration.

Chapter 5

Collaborating with Terraform

In the previous chapters we've been running Terraform locally and with the assumption that you're an individual developer. This makes sense when you're getting started with Terraform. But in the real world, it's likely you'll be working with others on your team on your Terraform configurations. This means maintaining the local state on your host is no longer feasible. Your colleagues need to know and be able to query the state of your infrastructure. We like to think about managing state like this as service discovery for infrastructure.

In this chapter we're going to look at solutions for sharing Terraform state and working with others to collaboratively build infrastructure. Later in the chapter we'll also extend our state management directly into the service discovery world by using [Consul](#) to store our remote state. Consul is a highly available key-value store, also written by the folks at HashiCorp, that we're going to use to store configuration values and state from Terraform. Other folks can then query that data and state from Consul to use in their own configurations.

Terraform state

In Chapter 2 we introduced the Terraform state. The state is populated when we apply our configuration and ends up locally in the `terraform.tfstate` file. This is a JSON representation of all the resources in our configuration. If we delete this state file, Terraform loses track of the state of our infrastructure.



NOTE When the state is changed you'll see that a backup file is created called `terraform.tfstate.backup`.

Also in Chapter 2, we recommended not adding your Terraform state to version control. This omission is acceptable for a single user. It's relatively easy to protect your state from accidental deletion and provides very simple management of your state. You could even potentially share the state between multiple users via an out-of-band mechanism. But, if someone else uses your configuration and adjusts it, they will end up with a new state file of their own.

If you're sharing the state file, you'll then need to merge others' changes, and you'll need to incorporate them into your own state. The frequency of changes to infrastructure, the potential for conflicts, and the complexity of resolving those conflicts makes this approach problematic.

To help address this issue Terraform offers two options. The first is making use of Hashicorp's [Terraform Enterprise product](#). This option requires spending some money and the book won't cover it directly. The second option is to make use of Terraform's built-in remote state storage capability. We're going to focus on the latter and learn how to manage our remote state.

Remote state

Terraform has the ability to store [its state remotely](#), in a variety of backends. This includes tools like Artifactory, etcd, and Consul, as well as going directly to storage such as with Amazon's S3 file system.

You can store some or all of your Terraform state in a backend. Why would you only store some state? This allows delegation to different groups of different parts of your infrastructure. Let's think about the VPC we configured in Chapter 3. We might have a central group that manages and controls VPC configuration and provides this infrastructure to other groups. By sharing that state with other teams, they don't need to replicate it and can focus on their own infrastructure.

You can start using Terraform with remote state or, as we'll see shortly, you can migrate your existing state into remote storage. You can also go back to local state storage from remote if required.

Sadly, this remote state management is not quite perfect and some common problems emerge with it.

No state protection

The specific remote state for a configuration is identified via a path or key. For example, you might have a state file for every environment: development, testing, production... In your remote backend these environments may be identified something like this:

Listing 5.1: Example of remote state environments

```
development/terraform.tfstate  
testing/terraform.tfstate  
production/terraform.tfstate
```

If you misconfigure your remote state configuration—for example, if you run it in the wrong environment—it's easy to overwrite the wrong state on your remote backend. This is again why many folks wrap Terraform in a script.

 **TIP** Terraform has another command that can be useful here: `refresh`. The `refresh` command reconciles the state file with real state of your infrastructure. It modifies your state file but does not change any infrastructure. Any proposed change will take place during the next `plan` or `apply`.

Not all backends have locking

First, some remote states storage are missing a key component: locking. Currently, state is managed when you apply configuration. During the execution process Terraform will download the current state and apply your infrastructure and any changes based on it. At the end of execution it'll push up the new state to the remote backend. For some of the remote backends, there is no way to lock access to this state while the plan is being executed. This means if someone else applies configuration at the same time as you, your configuration could get out of sync.

 **TIP** Terraform does come with a command line tool for editing the state. It's called `terraform state`. You can list the contents of the current state using the `terraform state list` command. You can use the `terraform state show` command to show the state of a specific resource. You can move items in the state or to another state. You can also remove items from the state.

We'll look at ways to address some of these concerns in this chapter. We'll also

look at a few ideas for managing some of the risks of Terraform’s remote state. Let’s start looking at remote state by setting up a backend.

 **TIP** Remote state backends significantly changed in Terraform 0.9. If you’re coming from an earlier version of Terraform you should read [this guide to upgrading from legacy backends](#).

Creating an S3 remote state backend

We’re going to be very meta in this chapter and use Terraform to configure our remote state backends. We’ll start with our initial remote state backend: an S3 bucket. The S3 bucket is a good choice for a backend as it’s highly available and yet still a simple file system.

Configuring our S3 backend is a straightforward task—all we need to do is create the bucket we want to store our configuration in and then configure it as Terraform’s remote state backend. We’ll create this configuration as a module to allow us to reuse it elsewhere.

Creating a module for an S3 remote state

Let’s start by creating our remote state module to make our S3 bucket. First we create a new directory under our `~/terraform` directory to hold our remote backend configuration, create a `README.md` file, and initialize our directory as a Git repository.

Listing 5.2: Creating an S3 remote backend directory

```
$ cd ~/terraform  
$ mkdir remote_state  
$ cd remote_state  
$ touch README.md  
$ git init
```

Let's create a `.gitignore` file to ensure we don't accidentally commit any state or variables values we don't want in our repository.

Listing 5.3: Creating a `.gitignore` file

```
$ echo ".terraform/" >> .gitignore  
$ echo "terraform.tfvars" >> .gitignore  
$ git add .gitignore
```

Next, let's add an `interface.tf` file to hold our Terraform variables.

Listing 5.4: The `remote_state` interface.`tf`

```
variable "region" {
  default    = "us-east-1"
  description = "The AWS region."
}

variable "prefix" {
  default    = "examplecom"
  description = "The name of our org, i.e. examplecom."
}

variable "environment" {
  default    = "development"
  description = "The name of our environment, i.e. development."
}
```

We've defined a simple set of variables, the region we want to launch the bucket in, a configurable prefix we'll apply to our bucket name, and the name of an environment or stack we'll use to identify our bucket. We've assumed you're using shared credentials for your secret and access keys rather than specifying them locally.

Lastly, let's define a `main.tf` to create our actual S3 bucket.

Listing 5.5: The `remote_state` `main.tf` file

```
resource "aws_s3_bucket" "remote_state" {
  bucket = "${var.prefix}-remote-state-${var.environment}"
  acl    = "authenticated-read"

  versioning {
    enabled = true
  }

  tags {
    Name = "${var.prefix}-remote-state-${var.environment}"
    Environment = "${var.environment}"
  }
}
```

We've defined a single resource, [an `s3_bucket` resource](#), to hold our remote state.

The `s3_bucket.remote_state` resource starts with a `bucket` name. Since AWS bucket names are globally unique, we've started with a configurable prefix contained in the `var.prefix` variable. You would add an appropriate prefix for your team and organization. We have then appended the value of our `var.environment` variable to the bucket name.

We've also specified a default Access Control List (ACL) to protect our configuration file. The `authenticated-read` ACL is one of AWS's [canned policies](#). This ACL allows the owner to read and write and authenticated users to read. This ACL mimics the delegation behavior we discussed earlier in this chapter: the owner can write to the state but other teams, if authenticated, can only read.

 **NOTE** This gives **any** authenticated AWS user access to your state. You should select a more restrictive ACL that works for you or develop your own.

We've also enabled S3 bucket versioning, which keeps a version history of any changes to the bucket. This helps protect us from accidental deletion of state data and allows us to potentially revert to a previous state.

We could have also used a new meta-parameter called `lifecycle`. The `lifecycle` meta-parameter provides the ability to control the life cycle of a resource. It has several options you can configure:

- `create_before_destroy` — If a resource is going to be recreated, then the new resource is created before the old resource is deleted. This is useful for creating resources that replace others, such as creating a new DNS record before you delete an old one.
- `ignore_changes` — Allows you to specify a list of attributes that will be ignored by Terraform.
- `prevent_destroy` — Does not delete the resource, even if a plan requires it. Any plan or execution that proposes destroying this resource will exit with an error.

If we want to protect the resource from deletion, we could enable this last attribute, `prevent_destroy`, as an extra safeguard against someone accidentally deleting the bucket. Once enabled, any execution plan that tries to delete the bucket will fail.

We've also added some tags to name and identify our bucket further.

Finally, let's add an output for our `remote_state` module in our `interface.tf` file.

Listing 5.6: The `remote_state` outputs

```
output "s3_bucket_id" {
  value = "${aws_s3_bucket.remote_state.id}"
}
```

This will return the ID of the bucket we're creating.

Planning the remote state module

Let's test that our module does what we expect. To do that we need to initialize it and then plan it. Let's initialize it first.

Listing 5.7: Initializing the remote state module

```
$ terraform init  
.  
.
```

Now we can run the plan to see.

Listing 5.8: The remote_state plan

```
$ terraform plan

. . .

+ aws_s3_bucket.remote_state
  acceleration_status:          "<computed>"
  acl:                          "authenticated-read"
  arn:                          "<computed>"
  bucket:                       "examplecom-remote-state-
development"
  force_destroy:                 "false"
  hosted_zone_id:               "<computed>"
  region:                       "<computed>"
  request_payer:                "<computed>"
  tags.%:                        "2"
  tags.Environment:             "development"
  tags.Name:                     "examplecom-remote-state-
development"
  versioning.#:                  "1"
  versioning.69840937.enabled:   "true"
  website_domain:               "<computed>"
  website_endpoint:              "<computed>"

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

Okay! Looks good. We can see our new S3 bucket, `aws_s3_bucket.remote_state`, is going to be created using our default `var.environment` of `development`.

Applying our remote state module

Let's now apply our configuration to absolutely confirm it all works.

Listing 5.9: Applying our `remote_state` configuration

```
$ terraform apply

aws_s3_bucket.remote_state: Creating...
  acceleration_status:      "" => "<computed>"
  acl:                      "" => "authenticated-read"
  arn:                      "" => "<computed>"
  bucket:                   "" => "examplecom-remote-state-
development"
  force_destroy:             "" => "false"
  hosted_zone_id:           "" => "<computed>"
  region:                   "" => "<computed>"
  request_payer:             "" => "<computed>"
  tags.%:                   "" => "2"
  tags.Environment:          "" => "development"
  tags.Name:                 "" => "examplecom-remote-state-
development"
  versioning.#:              "" => "1"
  versioning.69840937.enabled: "" => "true"
  website_domain:            "" => "<computed>"
  website_endpoint:          "" => "<computed>"

aws_s3_bucket.remote_state: Creation complete

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

. . .

State path: terraform.tfstate

Outputs:

s3_bucket_id = examplecom-remote-state-development
```

Neat! We can also see that our new S3 bucket has been created and that we're ready to use this module to configure Terraform for our remote state.

Uploading the remote state module

Let's now commit our code. We'll then create a repository on GitHub and upload our new module.

Listing 5.10: Committing our new `remote_state` module

```
$ git add .
$ git commit -a -m "Initial remote_state module"
```

You can see our final module, [tf_remote_state](#), on GitHub.

Configuring Terraform to use remote state

Let's configure the web stack configuration we've used in the last couple of chapters to use a remote state. Let's change into that configuration's home directory to get started.

Listing 5.11: Changing into the web configuration

```
$ cd ~/terraform/web
```

Let's add the `var.prefix` and `var.environment` variables we're going to use to name our bucket to the `variables.tf` file.

Listing 5.12: Adding the var.prefix and var.environment variables

```
....  
variable "prefix" {  
    default      = "examplecom"  
    description = "The name of our org, i.e. examplecom."  
}  
  
variable "environment" {  
    default = "web"  
    description = "The name of the environment."  
}  
....
```

We've specified defaults of `examplecom` and `web`, the name of our current stack.

We then add our `remote_state` module to our web stack configuration inside the `web.tf` file.

Listing 5.13: Adding the remote state module

```
provider "aws" {  
    region      = "${var.region}"  
}  
  
module "remote_state" {  
    source      = "github.com/turnbullpress/tf_remote_state.git"  
    prefix      = "${var.prefix}"  
    environment = "${var.environment}"  
}  
  
module "vpc" {  
....
```

Now let's get our new module.

Listing 5.14: Getting the remote_state module

```
$ terraform get -update
Get: git:::https://github.com/turnbullpress/tf_remote_state.git (
  update)
Get: git:::https://github.com/turnbullpress/tf_vpc.git?ref=v0.0.1
  (update)
```

We've fetched our new module and can now plan and apply our configuration to confirm our new bucket is created and ready.

Listing 5.15: Planning our web remote_state bucket

```
$ terraform plan

. . .

+ module.remote_state.aws_s3_bucket.remote_state
  acceleration_status:          "<computed>"
  acl:                          "authenticated-read"
  arn:                          "<computed>"
  bucket:                       "examplecom-remote-state-web"
  force_destroy:                 "false"
  hosted_zone_id:               "<computed>"
  region:                       "<computed>"
  request_payer:                "<computed>"
  tags.%:                        "2"
  tags.Environment:              "web"
  tags.Name:                     "examplecom-remote-state-web"
  versioning.#:                  "1"
  versioning.69840937.enabled:   "true"
  website_domain:                "<computed>"
  website_endpoint:              "<computed>"
```

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

We can see our new proposed bucket, `aws_bucket.remote_state`, with a name of `examplecom-remote-state-web`. Let's create it now.

Listing 5.16: Creating the web remote_state bucket

```
$ terraform apply

. . .

module.remote_state.aws_s3_bucket.remote_state: Creating...
  acceleration_status:      "" => "<computed>"
  acl:                      "" => "authenticated-read"
  arn:                      "" => "<computed>"
  bucket:                   "" => "examplecom-remote-state-
web"
  force_destroy:             "" => "false"
  hosted_zone_id:           "" => "<computed>"
  region:                   "" => "<computed>"
  request_payer:             "" => "<computed>"
  tags.%:                   "" => "2"
  tags.Environment:          "" => "web"
  tags.Name:                 "" => "examplecom-remote-state-
web"
  versioning.#:              "" => "1"
  versioning.69840937.enabled: "" => "true"
  website_domain:            "" => "<computed>"
  website_endpoint:          "" => "<computed>"

module.remote_state.aws_s3_bucket.remote_state: Creation
complete

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

. . .
```

Our new bucket, `examplecom-remote-state-web`, is now associated with the `web` configuration.

Setting the remote state

Configuring where Terraform stores state is done using standard Terraform code. Let's create a configuration now. in our `web` module.

Listing 5.17: Adding the remote state module

```
provider "aws" {
    region      = "${var.region}"
}

terraform {
    backend "s3" {
        bucket = "examplecom-remote-state-web"
        key    = "terraform.tfstate"
        region = "us-east-1"
    }
}

...
```

To configure the remote state backend we specify a new configuration block: `terraform`. The `terraform` block is designed to hold Terraform specific configuration.

Listing 5.18: The terraform remote state block

```
terraform {
    backend "s3" {
        region = "us-east-1"
        bucket = "examplecom-remote-state-web"
        key    = "terraform.tfstate"
    }
}
```

 **TIP** Your `backend` configuration cannot contain interpolated variables. This is because this configuration is initialized prior to Terraform parsing these variables.

We then specify a `backend` block inside our `terraform` block. Like a Terraform resource, each `backend` has a `type`, in our case `s3`. Other types include Consul, etcd and an ironically named local remote state backend.

Each `backend type` has a set of available configuration options. For the `S3` backend we need to specify the region in which the bucket is located. In our case this is the `us-east-1` region. We also need to specify the name of the `bucket`, in our case `examplecom-remote-state-web`. You'll need to update the `examplecom` to reflect the value of the `var.prefix` variable you've set. Last, we need to specify the `key`. The `key` is the location of the state file. In an S3 bucket this is the path to the file.

In our case, we've chosen to place the state file in a dedicated bucket named after our prefix variable, `examplecom`, and our environment variable, `web`. Remember each directory containing Terraform configuration generates its own state file. This means we can maintain individual states for each stack or application configured in Terraform.

Instead of bucket per environment or configuration, we could specify a single bucket and use the `key` configuration setting to specify a state file per directory. For example:

```
key = web/terraform.tfstate
```

We prefer the stand-alone bucket approach to protect our state as much as possible from mishaps.

After configuring our backend we need to initialize it. We do this using the `terraform init` command. You won't be able to run Terraform until you initialize your backend. Indeed, let's try and run `terraform plan` now and see what

happens.

Listing 5.19: Running terraform plan pre-initialization

```
$ terraform plan
Backend reinitialization required. Please run "terraform init".
Reason: Initial configuration of the requested backend "s3"

...
Failed to load backend: Initialization required. Please see the
error message above.
```

We've cut a bit of text here but Terraform is telling us we can't proceed without the `terraform init` command.

Let's run the `terraform init` command now and see if we can resolve this.

Listing 5.20: Running the `terraform init` command

```
$ terraform init
Downloading modules (if any)...
Get: git::https://github.com/turnbullpress/tf_remote_state.git
Get: git::https://github.com/turnbullpress/tf_vpc.git?ref=v0.0.1
Initializing the backend...

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

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform
plan" to see
any changes that are required for your infrastructure. All
Terraform commands
should now work.

If you ever set or change modules or backend configuration for
Terraform,
rerun this command to reinitialize your environment. If you
forget, other
commands will detect it and remind you to do so if necessary.
```

First, Terraform downloads our modules and then configures our backend. If we were to run `terraform plan` again now we'd be able to proceed.

 **TIP** If you're upgrading your configuration from a legacy backend in Terraform 0.8 and earlier you'll be automatically prompted to upgrade when you run a newer Terraform release.

So what just happened? There are two major changes. The first is that the `terraform.tfstate` file has disappeared from our `web` directory. This is because our state has been pushed to our S3 bucket.

 **NOTE** If a `terraform.tfstate.backup` file has been created it'll still be in your `web` directory.

The second change is the presence of a new `terraform.tfstate` file in the `.terraform` directory.

 **NOTE** Remember Terraform stores modules in the `.terraform` directory. Your local cache of the remote state file is also stored there. As we've discussed elsewhere, you'll want to add this directory and the `terraform.tfstate` file to your `.gitignore` file to ensure neither are committed to version control.

Let's look at this file and see how it's different from the state we saw in Chapter 2.

Listing 5.21: The new `terraform.tfstate` file

```
{  
  "version": 3,  
  "serial": 0,  
  "lineage": "6d79daf5-6cc0-415f-b45d-447bcd723653",  
  "backend": {  
    "type": "s3",  
    "config": {  
      "bucket": "examplecom-remote-state-web",  
      "key": "terraform.tfstate",  
      "region": "us-east-1"  
    }  
  }  
}
```

We can see our state file contains a new section: `backend`. This tells us that the original state for this configuration is stored remotely in our S3 bucket.

The S3 backend also provides a [locking capability](#). When you run Terraform it will use a DynamoDB database under the covers to provide a lock. This means if anyone else tries to run a Terraform apply using this configuration they'll receive an error message indicating you already have a Terraform operation in progress.

 **NOTE** Your AWS user needs to have access to DynamoDB for you to take advantage of the locking capabilities.

Disabling remote state

If we want to disable remote state we remove the `backend` block from our Terraform code. This will require also re-running the `terraform init` command to return to our previous, locally-stored, state. Terraform will warn us of this before

letting us plan or apply any new configuration.

Listing 5.22: Warning on remote state removal

```
$ terraform plan
Backend reinitialization required. Please run "terraform init".
Reason: Unsetting the previously set backend "s3"
. . .
```

So let's see what happens when we re-run the `terraform init` command to disable our remote state storage.

Listing 5.23: Disabling remote state storage

```
$ terraform init
Downloading modules (if any)...
Get: git::https://github.com/turnbullpress/tf_remote_state.git
Get: git::https://github.com/turnbullpress/tf_vpc.git?ref=v0.0.1
Do you want to copy the state from "s3"?
  Terraform has detected you're unconfiguring your previously
  set backend.
  Would you like to copy the state from "s3" to local state?
Please answer
  "yes" or "no". If you answer "no", you will start with a blank
  local state.

  Enter a value: yes
. . .
```

We can see that Terraform has again downloaded our modules but it has also asked us if we'd like to copy our existing state back to the local host. If we answer `yes`, this will remove the state from the backend and rewrite it into a local `terraform.tfstate` state file in our `~/terraform/web/.terraform` directory. If you answer `no` you'll get a blank local state.

In addition to our state being synchronized, when we configure the remote backend, it is also updated whenever we run Terraform operations like `plan` or `apply`. It is refreshed when commands are run and updated if any resources are changed.

You can also manually sync the state via the `terraform state pull` and `terraform state push` commands. Let's see what happens when we pull our state.

Listing 5.24: Pulling the remote state

```
$ terraform state pull
{
  "version": 3,
  "terraform_version": "0.9.0",
  "serial": 0,
  "lineage": "bf841743-b81f-41ce-8100-61e63761bd0e",
  "modules": [
    {
      "path": [
        "root"
      ],
      ...
    }
  ]
}
```

We can see that our current state being outputted in its JSON form. The `terraform state push` will try to push a local state configuration to the remote backend. It's generally done automatically but can be used manually if you have failed to connect to the backend for some reason.

Using and sharing remote state

Now that we've configured our Terraform remote state, how do we use it? There are two principal ways:

- Sharing state between users.

- Using data from the remote state.

Sharing state between users

Our first way is to ensure that when more than one person edits and manages our configuration, we don't duplicate configuration or make unnecessary changes. Let's look at a simple example. Say we've committed our `web` stack code to a Git repository and shared it with our team. A teammate then wants to make changes to our configuration. She clones that repository, `terraform get`'s our modules, and she runs the `terraform plan` command to check the current state of our infrastructure.

Listing 5.25: First run of terraform plan

```
$ terraform plan
Backend reinitialization required. Please run "terraform init".
Reason: Initial configuration of the requested backend "s3"

...
Failed to load backend: Initialization required. Please see the
error message above.
```

Oops. We need to initialize our backend state configuration. Let's follow the instructions and run `terraform init`.

Listing 5.26: Running terraform init for a new stack

```
$ terraform init
Downloading modules (if any)...
Get: git::https://github.com/turnbullpress/tf_remote_state.git
Get: git::https://github.com/turnbullpress/tf_vpc.git?ref=v0.0.1
Initializing the backend...

Terraform has been successfully initialized!
```

...

Woot! Now if we run `terraform plan` we should get our existing state.

Listing 5.27: Running terraform plan with existing state

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but
will not be
persisted to local or remote state storage.

data.template_file.master: Refreshing state...
aws_vpc.tfb: Refreshing state... (ID: vpc-53366435)
aws_s3_bucket.remote_state: Refreshing state... (ID: exampleco...
e-consul)
aws_internet_gateway.tfb: Refreshing state... (ID: igw-2e120549)
aws_subnet.public: Refreshing state... (ID: subnet-39117c05)
...
No changes. Infrastructure is up-to-date.
```

This means that Terraform did not detect any differences between
your
configuration and real physical resources that exist. As a
result, Terraform
doesn't need to `do` anything.

Excellent! We can see that Terraform has refreshed our remote state and validated it against the running resources. Each resource has been checked and Terraform has identified that no changes are currently planned.

To use Terraform in a shared environment you'll need to develop a workflow and process for collaborating.

 **TIP** We'll talk more about workflow in Chapter 6.

Externally loading remote state backends

One thing you remember we mentioned is that the `backend` block doesn't support interpolated variables. This means hard-coding variables, especially variables like the AWS region, that are likely to change regularly in dynamic environment. There is a work around for this issue though that can you can use. The `terraform init command` has a flag that allows you to pass a remote state backend configuration to it.

You'll still need to configure the skeleton of a remote state in your Terraform code, for example for an S3 backend:

Listing 5.28: A skeleton remote state backend

```
terraform { backend ""s3 {} }
```

We then create a file to hold the rest of our backend configuration. Let's call ours `s3_backend`.

Listing 5.29: Creating the s3_backend file

```
$ touch s3_backend
```

The `s3_backend` file contains key/value pairs containing any required configuration for our backend. Let's populate that file now:

Listing 5.30: The s3_backend file

```
region = "us-east-1"
bucket = "examplecom-remote-state-web"
key    = "terraform.tfstate"
```

We can then run the `terraform init` command with the `-backend-config` flag.

Listing 5.31: The backend-config flag

```
$ terraform init -backend-config=s3_backend
```

Terraform will complete our S3 remote state backend configuration with the values from the `s3_backend` file. This allows you to do a few different things:

- Stops you needing to store authentication keys or passwords in your Terraform code.
- Allows you to programmatically construct a backend configuration, suitable for scripting or in more dynamic environments.

You can now easily write a wrapper script that uses the `terraform init` command and populates your backend configuration appropriately, depending on the required context.

 **TIP** You can also specify individual key/value pairs to the `-backend-config` configuration flag like so: `-backend-config=key=value`.

Using remote state data

The second use of our state data is to provide data to help us build other infrastructure. Let's say we've built our VPC environment using the `vpc` module we created in Chapter 3. We can make use of the data that module outputs to configure other resources and environments.

To make use of remote state data we can take advantage of a [data source](#) called `terraform_remote_state` enabled by [the terraform provider](#). Remember that data sources provide read-only data that can be used in your configuration. The `terraform_remote_state` data source allows us to query remote state. Let's see it in action.

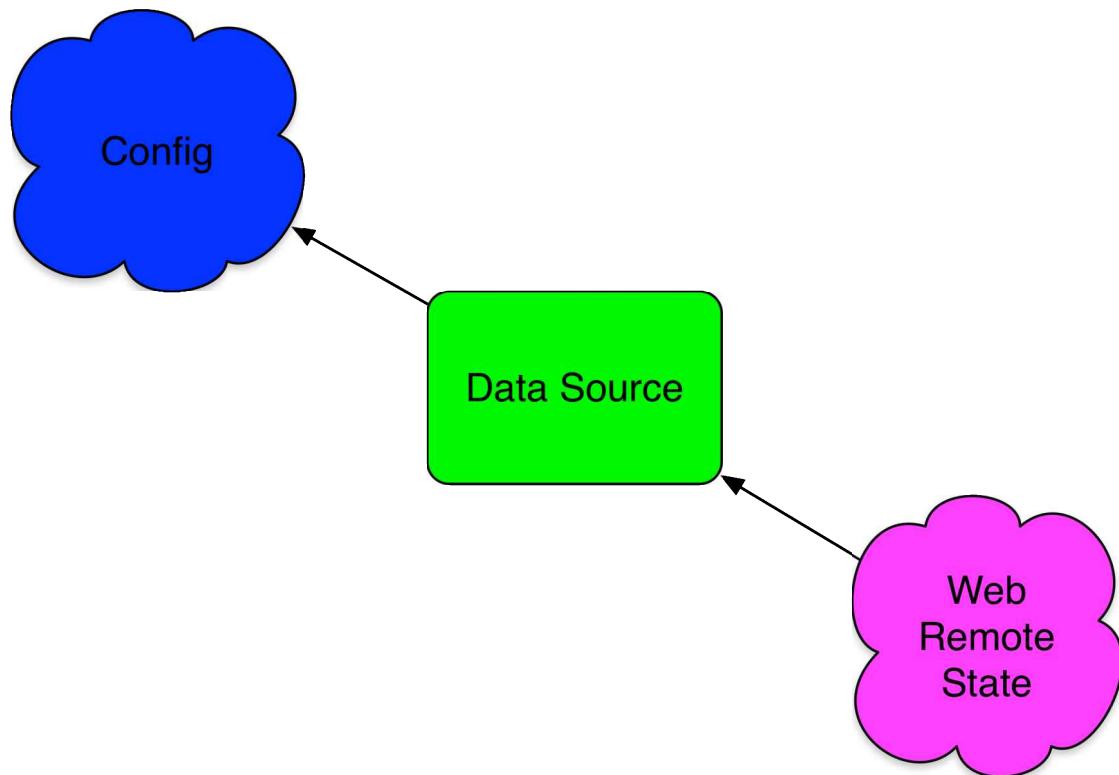


Figure 5.1: Querying remote state

To do this we're going to use some of the state of our `web` stack configuration in the `base` configuration we created in Chapter 2. Normally you would not be able to refer to values in another configuration; we cannot use attribute references between two configurations because Terraform doesn't know the other configuration exists. Using the `terraform_remote_state` data source makes this possible.

Moving our `base` state remote

Let's start by moving our `base` configuration's state to a remote state using an S3 bucket for consistency. We'll also add the `var.environment` variable to the `variables.tf` file.

Listing 5.32: Adding the environment variable to the base variables.tf

```
....  
variable "environment" {  
  default    = "base"  
  description = "The environment name."  
}  
....
```

And we'll add our `remote_state` module to the `base.tf` configuration file.

Listing 5.33: Adding the remote_state module to base.tf

```
provider "aws" {  
  region = "${var.region}"  
}  
  
module "remote_state" {  
  source = "github.com/turnbullpress/tf_remote_state.git"  
  prefix    = "${var.prefix}"  
  environment = "${var.environment}"  
}  
  
resource "aws_instance" "base" {  
  ....
```

Then we'll get the module.

Listing 5.34: Getting the remote_state module for base

```
$ terraform get  
Get: git::https://github.com/turnbullpress/tf_remote_state.git
```

And then we'll apply it.

Listing 5.35: Applying the remote_state module for base

```
$ terraform apply
...
module.remote_state.aws_s3_bucket.remote_state: Creating...
  acceleration_status:      "" => "<computed>"
  acl:                      "" => "authenticated-read"
  arn:                      "" => "<computed>"
  bucket:                   "" => "examplecom-remote-state-
base"
  force_destroy:             "" => "false"
  hosted_zone_id:           "" => "<computed>"
  region:                   "" => "<computed>"
  request_payer:             "" => "<computed>"
  tags.%:                   "" => "2"
  tags.Environment:          "" => "base"
  tags.Name:                 "" => "examplecom-remote-state-
base"
  versioning.#:              "" => "1"
  versioning.69840937.enabled: "" => "true"
  website_domain:            "" => "<computed>"
  website_endpoint:          "" => "<computed>"
module.remote_state.aws_s3_bucket.remote_state: Creation
complete
...
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

We can see our new bucket, `examplecom-remote-state-base`, has been created. We can now configure the `base` configuration to move our state to that bucket.

Listing 5.36: Moving the base state remote

```
provider "aws" {
    region = "${var.region}"
}

terraform {
    backend "s3" {
        region = "us-east-1"
        bucket = "examplecom-remote-state-base"
        key    = "terraform.tfstate"
    }
}

module "remote_state" {
    source = "github.com/turnbullpress/tf_remote_state.git"
    ...
}
```

We then run `terraform init` to initialize the remote state and now our `base` configuration state is uploaded to our `examplecom-remote-state-base` bucket. We can also check in the `.terraform` directory for our local cached state file.

Adding the data source to the base configuration

Let's add the `terraform_remote_state` data source to the `base` configuration so we can read the web stack's remote state. We'll add the data source to the top of our `base.tf` file after the `aws` provider.

Listing 5.37: The `terraform_remote_state` data source

```
...
data "terraform_remote_state" "web" {
  backend = "s3"
  config {
    region = "${var.region}"
    bucket = "examplecom-remote-state-web"
    key    = "terraform.tfstate"
  }
}
...
```

Our data source is specified within the `data` block. The type of the data source is `terraform_remote_state` and the name is `web`.

Inside our `terraform_remote_state` data source we configure the remote state we wish to query. We first define a `backend` to query. Ours is `s3`, the backend we configured earlier in this chapter. We then specify the connection details of our data source inside a map called `config`. These are the same options we specify using the `backend` block in the `terraform` configuration block. We set the `region` we're querying. We also set the `bucket` name—we're using our `web` stack's state and the specific `key` we'd like to query.

 **TIP** We can query more than one remote state by specifying the `terraform_remote_state` data source multiple times. Remember that each data source, like our resources, needs to be named uniquely.

Let's see what this offers us. We can update the Terraform configuration by using

the `terraform apply` command.

We need to use the `terraform apply` command here, rather than the `terraform plan` command, because the data source is a resource. If we only plan then this resource is not added to the `base` configuration.

Listing 5.38: Using terraform data source

```
$ terraform apply
data.terraform_remote_state.web: Refreshing state...
aws_instance.base: Refreshing state... (ID: i-eaf428fc)
aws_eip.base: Refreshing state... (ID: eipalloc-b9286486)

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

We can see that Terraform has connected to our `data.terraform_remote_state.web` data source and refreshed its state, together with our other resources.

 **NOTE** For a data source to be read, the remote state needs to exist. If the configuration you're fetching doesn't exist—for example, if it's been destroyed or not yet applied—then your remote state will be returned empty. Any variables you are populating will return an error.

Now let's look at the data that Terraform has gotten from our remote state. We can do this with the `terraform show` command.

Listing 5.39: Showing our remote state data source

```
$ terraform show
aws_eip.base:
  id = eipalloc-b9286486
  association_id = eipassoc-a01e699c
  domain = vpc
  instance = i-eaf428fc

  . . .

data.terraform_remote_state.web:
  id = 2016-10-08 02:25:31.134813974 +0000 UTC
  addresses.# = 2
  addresses.0 = 52.87.170.139
  addresses.1 = 54.83.165.201
  backend = s3
  config.% = 3
  config.bucket = examplecom-remote-state-web
  config.key = terraform.tfstate
  config.region = us-east-1
  elb_address = web-elb-67413180.us-east-1.elb.amazonaws.com
  public_subnet_id = subnet-ae6bacf5
```

All of our resources, including our new `data.terraform_remote_state.web` data source, are present in the output. We can also see a list of attributes available from our data source.

 **NOTE** Only the root-level outputs from the remote state are available. Outputs from modules within the state cannot be accessed. If you want a module output to be accessible via a remote state, you must expose the output in the top-level configuration.

These attributes are now available for us to use in our `base` configuration. Let's use one of them now. We're going to use the VPC public subnet ID created in our web stack in our `base` configuration. We'll get this value by querying the web stack's remote state using the `terraform_remote_state` data source.

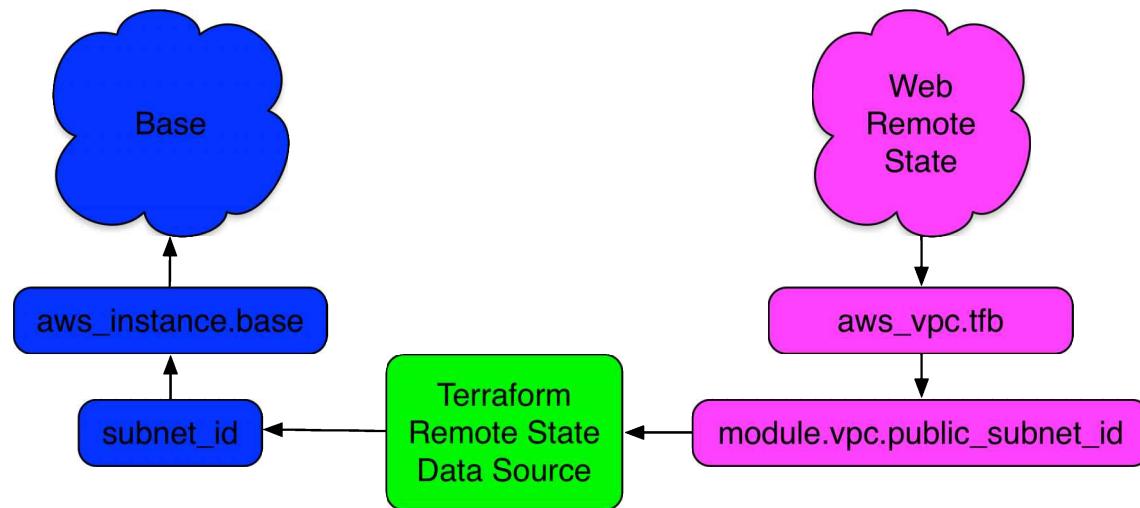


Figure 5.2: Querying the web remote state

Let's start by adding a value from the web stack remote state via a data source variable.

Listing 5.40: Using our remote state

```
resource "aws_instance" "base" {
    ami          = "${lookup(var.ami, var.region)}"
    instance_type = "t2.micro"
    subnet_id    = "${data.terraform_remote_state.web.
public_subnet_id}"
}

resource "aws_eip" "base" {
    instance = "${aws_instance.base.id}"
    vpc      = true
}
```

You can see we've added the attribute:

`data.terraform_remote_state.web.public_subnet_id`

As the value of the `subnet_id` of our AWS instance. Note the variable is constructed with a prefix of `data`, the type of data source, the name of the data source, and finally the specific output we're using. If we run `terraform plan` again, we'll see the proposed change.

Listing 5.41: Planning using our data source

```
$ terraform plan

. . .

~ aws_eip.base
  instance: "i-eaf428fc" => "${aws_instance.base.id}"

-/+ aws_instance.base
  ami:                      "ami-4cc3b85b" => "ami-4cc3b85b"
  availability_zone:        "us-east-1a" => "<computed>"
  ebs_block_device.#:       "0" => "<computed>"
  ephemeral_block_device.#: "0" => "<computed>"
  instance_state:           "running" => "<computed>"
  instance_type:             "t1.micro" => "t1.micro"
  key_name:                  "" => "<computed>"
  network_interface_id:     "eni-d36f5bd5" => "<computed>"
  placement_group:          "" => "<computed>"
  private_dns:               "ip-172-31-11-212.ec2.internal" =>
    "<computed>"
  private_ip:                 "172.31.11.212" => "<computed>"
  public_dns:                "ec2-52-5-148-165.compute-1.
    amazonaws.com" => "<computed>"
  public_ip:                  "52.5.148.165" => "<computed>"
  root_block_device.#:      "1" => "<computed>"
  security_groups.#:        "0" => "<computed>"
  source_dest_check:         "true" => "true"
  subnet_id:                  "subnet-a48f23ed" => "subnet-
    ae6bacf5" (forces new resource)
  tenancy:                   "default" => "<computed>"
  vpc_security_group_ids.#: "1" => "<computed>"

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

We can see our proposed change will cause Terraform to recreate the resource (because the subnet it belongs to can only be changed at launch) as indicated by the `(forces new resource)` in the output. If we were to run `terraform apply`

now, our resources would be recreated or changed. Terraform would update the subnet our `aws_instance.base` is in to be the same subnet as our web stack resources and would change its Elastic IP address. This is a simple and powerful way to ensure standardization of your resources and environments.

The classic use case for this is shared infrastructure, such as a VPC, load balancer, or storage. You can have one group responsible for building the infrastructure and outputting the relevant information required by other teams to use it: addresses, versions, ID, etc. Teams can then consume this in their Terraform configurations without risking others changing the base configuration.

State and service discovery

We've seen how we can use our remote state in another configuration. Let's extend the principle to enable service discovery. Service discovery allows the automatic detection of applications and services in your environment. Clients or other applications query a service discovery tool, and it returns the IP address or port numbers or other relevant information that allows them to find the service they want.

So when should you use remote state versus service discovery? Remote state best lends itself to provisioning use. There are variables or data you want to make use of when you build your stack. They're often only used once or twice during that process. They don't require you to regularly query that data source while your application or service is being run. Service discovery tends to be used at runtime and exercised regularly when applications and services run—for example, by querying the address of a required service. It generally requires a more resilient and faster service with a higher availability than our remote state.

We're going to enable service discovery using another HashiCorp tool: [Consul](#). Consul includes a highly available key-value store that we're going to use to store configuration values from Terraform. Other folks can then query that data and

state from Consul for use in their own configurations.

We'll create a Consul cluster using Terraform, show you how other Terraform configurations can be made aware of Consul, and then we'll look at how to store their service information in Consul.

Creating a Consul cluster

In order to use Consul, we first need to create a Consul server cluster. We've created a new module to do this, [which you can find on GitHub](#). Our module, about which we're not going to go into deep detail, creates a three-node Consul cluster in a VPC in a public subnet. The module does not produce an overly secure Consul cluster. It will allow external access to the Consul nodes and is only lightly secured, with ACL enabled and set to default `allow` and encryption between nodes.

You should review the module or look at [alternative modules](#). The module will output the public DNS address of the first Consul node and the private IP addresses of the cluster nodes. We'll store this data in remote state and make use of it to connect to our Consul server later in this chapter.

Let's create a new configuration to use this module and create our cluster.

Listing 5.42: Making a Consul configuration

```
$ cd ~/terraform
$ mkdir consul
```

We'll then start with a `variable.tf` file to hold our variables.

Listing 5.43: The consul module's variables.tf file

```
variable "region" {
  default      = "us-east-1"
  description = "The AWS region."
}
variable "prefix" {
  description = "The name of our org, i.e. examplecom."
  default      = "examplecom"
}
variable "environment" {
  description = "The name of our environment, i.e. development."
}
variable "private_key_path" {
  default      = "~/.ssh/james.key"
  description = "The path to the AWS key pair to use for
resources."
}
variable "key_name" {
  default      = "james"
  description = "The AWS key pair to use for resources."
}
variable "vpc_cidr" {
  description = "The CIDR of the VPC."
}
variable "public_subnet" {
  description = "The public subnet to populate."
}
variable "token" {
  description = "Consul server token"
}
variable "encryption_key" {
  description = "Consul encryption key"
}
```

We've seen a lot of these variables before, but we've added a few new ones: `token` and `encryption_key`. These will configure access control lists, ACLs, and cluster encryption to provide some basic security for Consul.

We'll also create values for our `token`, `encryption_key`, and other variables in a `terraform.tfvars` file.

Listing 5.44: The new Consul variable values in `terraform.tfvars`

```
token = "yourtoken"
encryption_key = "yourkey"
environment = "consul"
vpc_cidr = "10.0.0.0/16"
public_subnet = "10.0.5.0/24"
```

Your `encryption_key` needs to be a 16-byte Base64-encoded string. There are several ways you can generate one. The first is to [download Consul](#) and use [the consul keygen command](#). Or you can generate one on the command line using something like this:

Listing 5.45: Generating an encryption key via openssl

```
$ openssl rand -base64 16
```

This assumes you have the `openssl` binary available, and it uses the `rand` command to return a Base64-encoded key. If you don't have the `openssl` binary you can do something like:

Listing 5.46: Generating an encryption key via SHA

```
$ date +%s | shasum -a 256 | base64 | head -c 16 ; echo
```

This method uses the current date, runs it through a SHA256, Base64 encodes it, and then returns the first 16 bytes.

For the `token` we tend to use a UUID. You can generate one on most hosts like so:

Listing 5.47: Creating a UUID token

```
$ uuid  
3a21ec16-aa46-11e6-84a5-9b7a59f45343
```

Now let's create some outputs for our configuration in an `outputs.tf` file.

Listing 5.48: The new Consul outputs

```
output "consul_server_address" {  
    value = ["${module.consul.consul_dns_address}"]  
}  
output "consul_host_addresses" {  
    value = ["${module.consul.consul_host_addresses}"]  
}
```

We're outputting both of the Consul module's output variables here.

Last, we want to add a `consul.tf` file to instantiate our cluster.

Listing 5.49: The `consul.tf` file

```
provider "aws" {
    region      = "${var.region}"
}

module "remote_state" {
    source      = "github.com/turnbullpress/tf_remote_state.git"
    prefix      = "${var.prefix}"
    environment = "${var.environment}"
}

module "vpc" {
    source      = "github.com/turnbullpress/tf_vpc.git?ref=v0
.0.1"
    name        = "consul"
    cidr        = "${var.vpc_cidr}"
    public_subnet = "${var.public_subnet}"
}

module "consul" {
    source      = "github.com/turnbullpress/tf_consul.git"
    environment = "${var.environment}"
    token       = "${var.token}"
    encryption_key = "${var.encryption_key}"
    vpc_id      = "${module.vpc.vpc_id}"
    public_subnet_id = "${module.vpc.public_subnet_id}"
    region      = "${var.region}"
    key_name    = "${var.key_name}"
    private_key_path = "${var.private_key_path}"
}
```

Our `consul.tf` uses three modules: the remote state module that will create an S3 bucket to put our remote state in, the VPC module that will create a new VPC to hold our Consul cluster, and the Consul module to create the cluster itself.

Let's initialize the configuration and grab those modules now.

Listing 5.50: Getting the Consul configuration's modules

```
$ pwd  
~/terraform/consul  
$ terraform init  
  
...  
  
$ terraform get -update  
Get: git::https://github.com/turnbullpress/tf_remote_state.git (update)  
Get: git::https://github.com/turnbullpress/tf_vpc.git?ref=v0.0.1 (update)  
Get: git::https://github.com/turnbullpress/tf_consul.git (update)
```

We can then `terraform plan` and `terraform apply` our Consul configuration. Let's assume we've run `terraform plan` and jump straight to an `apply`.

Listing 5.51: Applying our Consul configuration

```
$ terraform apply  
.  
.  
Apply complete! Resources: 8 added, 0 changed, 0 destroyed.  
  
Outputs:  
  
consul_host_addresses = [  
    10.0.5.57,  
    10.0.5.171,  
    10.0.5.192  
]  
consul_server_address = [  
    ec2-54-221-122-241.compute-1.amazonaws.com  
]
```

We can see our Consul cluster has been created and our outputs emitted.

We can confirm our Consul cluster is running by browsing to the web UI by adding port **8500** to the DNS address in the **consul_server_address** output.



Figure 5.3: The Consul Web UI

We can move the state we've just created into the S3 bucket. We create our **backend** block inside our **consul.tf** file.

Listing 5.52: Moving the Consul state to S3

```
...
terraform {
  backend "s3" {
    region = "us-east-1"
    bucket = "examplecom-remote-state-consul"
    key    = "terraform.tfstate"
  }
}
```

This, combined with the `terraform init` command, will put our state into the `examplecom-remote-state-consul` S3 bucket we just created.

Alternatively, we could use [Consul itself to store the remote state](#). We can store our remote state as a key inside Consul's key/value store.

Listing 5.53: Storing remote state in Consul

```
...
terraform {
  backend "consul" {
    path      = "state/consul"
    access_token = "yourtoken"
    address     = "ec2-54-221-122-241.compute-1.amazonaws.com
:8500"
    datacenter   = "consul"
  }
}
...
```

This will upload our state as a key `state/consul` to our Consul server. We've

specified the address of our Consul server, suffixed with the `8500` port. We've also specified the `datacenter` to use, which we defined with the `var.environment` variable, here with a value of `consul`, in our Consul module.

TIP Rather than specify the `access_token` you could use the `CONSUL_HTTP_TOKEN` environment variable or specify it via the `-backend-config` command line flag. This keeps your token out of your local state.

You should now be able to see our remote state in the web UI.



Figure 5.4: The state/consul key

Let's make use of that state.

Using Consul and remote state

At this point we should revisit our [web](#) configuration. Remember our configuration creates some AWS instances and an Elastic Load Balancer. Let's add some Consul integration to that configuration. We're going to:

- Allow the `web` configuration to query the Consul remote state for the Consul server's DNS address.
- Store some configuration from the `web` environment in our new Consul cluster.
- Query that configuration data.

The first step is to add our Consul configuration's remote state to our `web` environment, as we saw earlier in the chapter. Let's assume we're still storing it in S3 rather than Consul. Let's open up the `~/terraform/web/web.tf` file and add the `terraform_remote_state` data source.

Listing 5.54: Adding the Consul remote state

```
...
.
.
.
data "terraform_remote_state" "consul" {
  backend = "s3"
  config {
    region = "${var.region}"
    bucket = "examplecom-remote-state-consul"
    key    = "terraform.tfstate"
  }
}
.
.
```

Here we've used the `terraform_remote_state` data source to query the remote state of the Consul cluster we've just created. It'll look inside the S3 bucket's remote state and make the outputs from that configuration available in our `web` configuration. Let's run `terraform apply` now to update the configuration.

Listing 5.55: Updating the web configuration

```
$ terraform apply  
data.terraform_remote_state.consul: Refreshing state...  
.  
. . .
```

Now the `web` configuration knows about the Consul state. Let's explore what it knows in a bit more depth.

Listing 5.56: Showing the Consul state

```
$ terraform show  
. . .  
  
data.terraform_remote_state.consul:  
  id = 2016-11-10 12:48:44.453771993 +0000 UTC  
  backend = s3  
  config.% = 3  
  config.bucket = examplecom-remote-state-consul  
  config.key = terraform.tfstate  
  config.region = us-east-1  
  consul_host_addresses.# = 3  
  consul_host_addresses.0 = 10.0.5.12  
  consul_host_addresses.1 = 10.0.5.11  
  consul_host_addresses.2 = 10.0.5.163  
  consul_server_address.# = 1  
  consul_server_address.0 = ec2-54-221-122-241.compute-1.  
amazonaws.com  
. . .
```

We can see that our configuration knows where to find the Consul server via the `consul_server_address.0` output in the remote state. Let's take advantage of that

now.

We're going to add a new provider to the `web` configuration: `consul`. The `consul` provider allows you to connect to a Consul server and interact with it. Let's add it to the `web.tf` file.

Listing 5.57: Adding the `consul` provider

```
    . . .

provider "consul" {
    address      = "${data.terraform_remote_state.consul.
    consul_server_address.0}:8500"
    datacenter   = "consul"
}

    . . .
```

Here we've added a new `provider` block. The `consul` provider has two required attributes: an `address` and a `datacenter`. We're extracting the `address` from the Consul configuration's remote state with our data source, querying:

`data.terraform_remote_state.consul.consul_server_address.0`

We've added the port to which Consul is bound on that address, `8500`, to the variable. In [our Consul configuration and module](#) we've used the value of the `environment` variable as the value of the Consul datacenter.

Lastly, to connect to our Consul server and interact with it, we need to add a single variable: `token`. Remember we added some basic security to our Consul cluster, including specifying a token used for updating data in the cluster. Let's add that variable to `variables.tf` now.

Listing 5.58: Adding the token variable

```
...  
  
variable "token" {  
  description = "The Consul server token"  
}
```

We then want to add a value to our `terraform.tfvars` file for this token.

Listing 5.59: Adding the Consul token value

```
...  
  
token = "yourtoken"
```

Now let's add a resource that makes use of this new provider and our `token` variable. We're going to use the `consul_key_prefix` resource to add some key/values to Consul. The `consul_key_prefix` resource configures a namespace containing a series of keys. We'll populate those keys with attributes from our `web` configuration.

Listing 5.60: The `consul_key_prefix` resource

```
 . . .

resource "consul_key_prefix" "web" {
    token = "${var.token}"

    path_prefix = "web/config/"
    subkeys = {
        "public_dns"      = "${aws_elb.web.dns_name}"
    }
}

. . .
```

Our new resource consumes our `var.token` variable. We've specified a `path_prefix`, here `web/config/`, that will prefix any key/value pairs we add to Consul. This allows us to group our keys under a namespace. We've also added a single key to that namespace. The key is called `public_dns`—or with our prefix, `web/config/public_dns`—and its value is the DNS name of the Elastic Load Balancer we're creating in the `web` configuration: `aws_elb.web.dns_name`.

Let's instantiate this key by applying our configuration.

Listing 5.61: Applying our new web configuration

```
$ terraform apply
data.terraform_remote_state.consul: Refreshing state...
.

consul_key_prefix.web: Creating...
  datacenter:      "" => "<computed>"
  path_prefix:    "" => "web/config/"
  subkeys.%:      "" => "1"
  subkeys.public_dns: "" => "web-elb-1887901826.us-east-1.elb.
amazonaws.com"
  token:          "" => "yourtoken"
consul_key_prefix.web: Creation complete

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

.
```

We can see our new key has been created and populated with the public DNS name of the Elastic Load Balancer. You'll also note that the `token` is in the state and the output.

We can now query the Consul cluster to use this key. We could use the key in another Terraform configuration using the `consul_keys` data source. Or we could use a Consul client or HTTP client to fetch the ELB address from the key. Let's see it from the command line using `curl`.

Listing 5.62: Curling our Consul key

```
$ curl -s 'ec2-54-221-122-241.compute-1.amazonaws.com:8500/v1/kv/web/config/public_dns?token=yourtoken' | jq
[
  {
    "LockIndex": 0,
    "Key": "web/config/public_dns",
    "Flags": 0,
    "Value": "d2ViLWVsYi0x0Dg30TAx0DI2LnVzLWWhc3QtMS5lbGIuYW1hem9uYXdzLmNvbQ==",
    "CreateIndex": 1402,
    "ModifyIndex": 1402
  }
]
```

Here we've curled to the Consul key/value HTTP API and pulled the `web/config/public_dns` key. This returns a JSON hash, which we've piped through `jq` to prettify it. Our ELB address is in the `Value` key (don't panic—it's not gibberish, just Base64-encoded to allow you to use non-UTF8 characters). Let's decode it.

Listing 5.63: Decoding the value

```
$ echo
d2ViLWVsYi0x0Dg30TAx0DI2LnVzLWWhc3QtMS5lbGIuYW1hem9uYXdzLmNvbQ==
| base64 --decode
web-elb-1887901826.us-east-1.elb.amazonaws.com
```

Or you can fancily do the `curl` and decode all in one step.

Listing 5.64: Decoding the value in one step

```
$ curl -q ec2-54-221-122-241.compute-1.amazonaws.com:8500/v1/kv/web/config/public_dns?token=yourtoken | jq '.[] | .Value' | tr -d '"' | base64 --decode | awk '{ print $1 }'
```

Or you could do this via a programmatic process in your application or client code. We can now use that value in a service or application. We get provisioning, updates to our service discovery tool, and service discovery data available to be consumed by applications, services, or monitoring.

Other tools for managing Terraform state

There are some other tools available to help manage Terraform's state:

- [Terrahelp](#): Go utility that provides Vault-based encryption and decryption of state files.
- [Terragrunt](#): Go tool for managing locking and state that can be used as glue in a multi-environment setup.
- [Terraform_exec](#): Go wrapper that allows Terraform projects to have multiple environments synced to S3.

Summary

In this chapter we've learned more about managing Terraform's state and sharing that state with others. We've learned how to configure remote state, setup an S3 bucket backend, and manage our state with others.

We can also see that remote state management isn't quite perfect with Terraform yet. It requires some cooperation and collaboration between folks working on shared configuration.

We've also seen how to use our remote state data in our configurations and potentially how to integrate remote state with service discovery.

In the next chapter we're going to build on this to introduce a much more fully featured and complex state management architecture across multiple environments.

Chapter 6

Building a multi-environment architecture

In Chapter 5 learned how to manage, share, and use Terraform's remote state. We also saw some of the challenges involved in using that state. We're going to combine that state knowledge with our previous learning about Terraform itself to create a multi-environment and multi-service architecture. Importantly, we're also going to describe a workflow for developing and deploying Terraform infrastructure and changes to that infrastructure.

Terraform users will tell you that working out how to organize and lay out your code is crucial to having a usable Terraform environment. Inside our new architecture, we're going to create new Terraform configuration in the form of an example data center. We'll include a file and directory layout, state management, multiple providers, and data sources.

Creating a data center

We're going to create a multi-environment data center to demonstrate what we've learned over the last few chapters. It's going to be hosted in AWS and will look

like this:

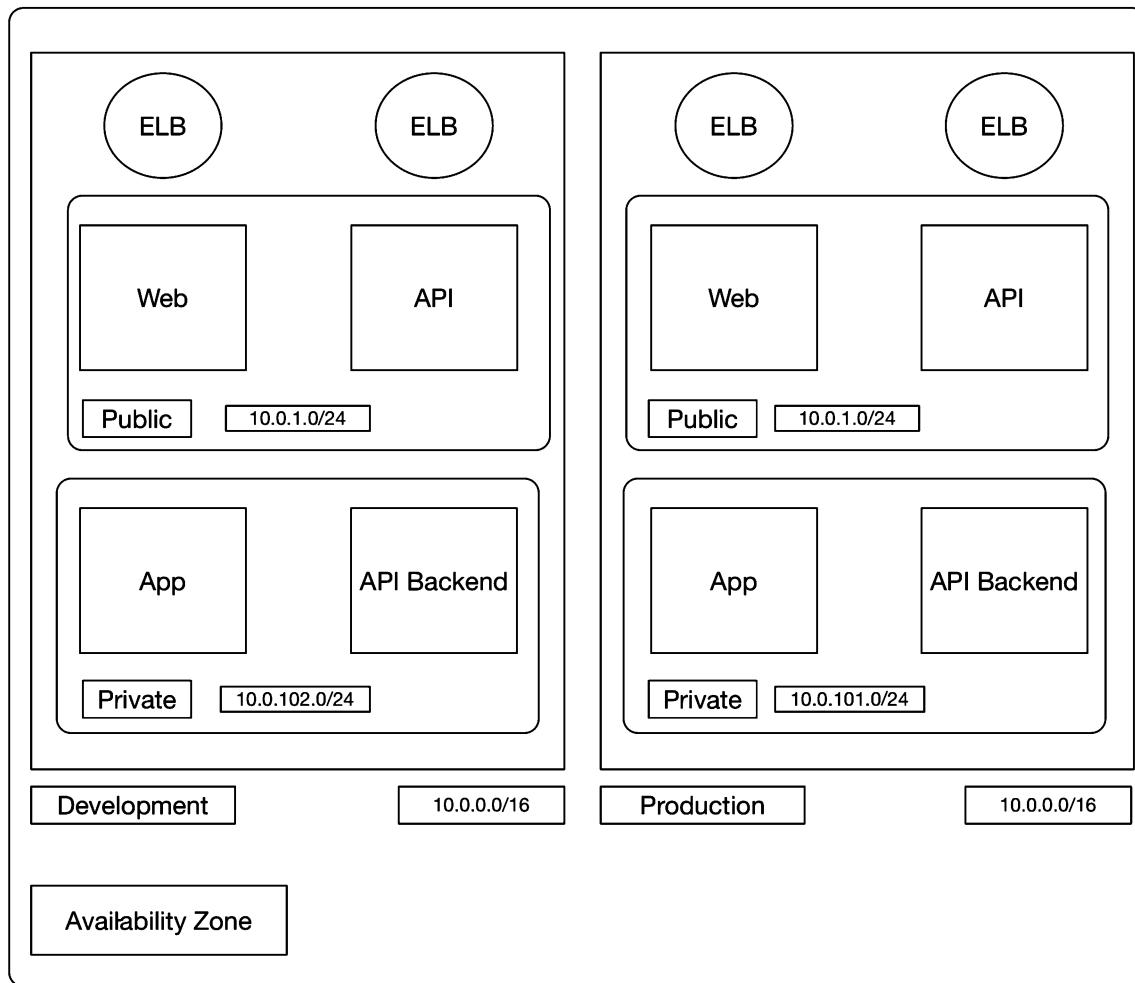


Figure 6.1: Our data center

We have a **development** environment and a **production** environment, both built in AWS. We have two services, a web service and a backend API service, in each environment.

- Our web service stack will consist of a [Cloudflare](#) website service, a load balancer, two EC2 instances running a web server, and two EC2 instances as backend application servers for persistent data.

- Our API stack will consist of a load balancer with five EC2 instances.

We're going to create a module for each service that we're going to build. This will allow us to reuse our Terraform configuration across environments and better manage versions of infrastructure.

Creating a directory structure

Let's start by creating a directory and file structure to hold our Terraform configuration. We'll create it under our home directory.

Listing 6.1: Creating the base directory

```
$ cd ~  
$ mkdir -p dc/{development,production,modules/{web,api,vpc,  
remote_state}}  
$ cd dc  
$ git init
```

We've created a base directory called `dc` and a directory for each environment we want to create—`development`, `production`—as well as a `modules` directory. The `modules` directory will contain each of the modules we'll use in our environment. These directories will contain the module source code. We'll upload the modules to GitHub and use that as their source so we can more easily reuse them.

Our new directory structure looks like this:

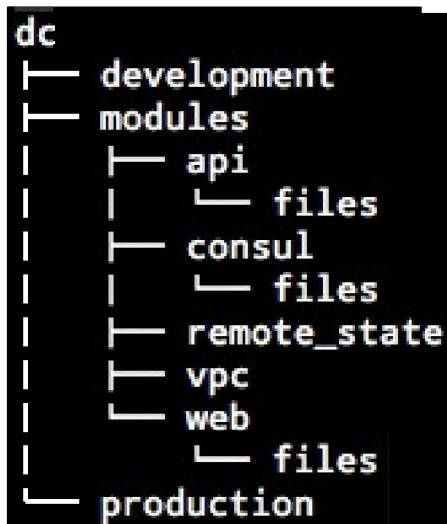


Figure 6.2: Directory structure

Each environment will contain environment configuration, and then each service’s module will be declared and configured. We’ll add variables and outputs for the environment, as well as for each service.

Let’s add a `.gitignore` file too. We’ll exclude any state files to ensure we don’t commit any potentially sensitive variable values.

Listing 6.2: Adding the state file and backup to `.gitignore`

```
$ echo "terraform.tfstate*" >> .gitignore
$ echo ".terraform/" >> .gitignore
$ git add .gitignore
$ git commit -m "Adding .gitignore file"
```

We’ve also created a Git repository for our data center environment. You can find it [on GitHub here](#).

Workflow

One of the key aspects of managing infrastructure is the workflow you establish to do so. Our multiple environments provide a framework for testing changes prior to production. The `terraform plan` command provides a mechanism for conducting that testing.

We've run the `terraform plan` command before pretty much every change we've made in the book. This is intentional: the output of planning is critical to ensure we deploy the right changes to our environment. When we're working with a multi-environment architecture we can make use of our framework and the planning process.

We recommend a simple workflow.

Develop

We develop our infrastructure in the `development` environment. We'll see some patterns in this chapter for how to build each environment and how to package services using modules to ensure appropriate isolation for each environment and each service we wish to deploy.

We should ensure our code is clean, simple, and well documented. There are some good guidelines to work within to help with this:

- All code should be in version control.
- Always comment code when it requires explanation.
- Add `description` fields to all variables.
- Include `README` files or documentation for your module and their interfaces.
- Running `terraform fmt` and `terraform validate` prior to committing or as a commit hook is strongly recommended to ensure your code is well formatted and valid.

Plan

When you're ready to test your infrastructure, always run `terraform plan` first. Use this plan to ensure the `development` version of your infrastructure is accurate and that any changes you're proposing are viable. Between validation and the planning process you are likely to catch the vast majority of potential errors.

Remember from Chapter 2 that the `terraform plan` also allows you to save the generated plan from each run. To do this we run the `terraform plan` command with the `-out` flag.

Listing 6.3: Running the terraform plan command with -out

```
$ terraform plan -out development-`date +'%s'`.plan
```

This saves the generated plan, including any differences, in a file—here, `development-epochtime.plan`.

We can now use this file as a blueprint for our change. This ensures that even if the configuration or state has drifted or changed since we ran the `plan` command, the changes applied will be the changes we expect.

Listing 6.4: Applying the generated plan

```
$ terraform apply development-epochtime.plan
```

⚠️ WARNING The generated plan file will contain all your variable values—potentially including any credentials or secrets. It is not encrypted or otherwise protected. Handle this file with appropriate caution!

Apply in development

Even with a `plan` things can still go wrong. The real world is always ready to disappoint us. Before you apply your configuration in your `production` environment, apply it in your `development` environment. Apply it using plan output files in iterative pieces to confirm it is working correctly.

You can use scaled-down versions of infrastructure—for example, in AWS we often use smaller instance types to build infrastructure in `development`, in order to save money and capacity. Actually deploying your configuration further validates it and likely reduces the risk of potential errors.

Automation and testing

If things have worked in your `development` environment, then it's time to consider promoting it to your `production` environment. This doesn't mean, however, that you shouldn't make your changes jump through a few more hoops.

You should have your code in version control so it becomes easy to pass changes through an appropriate workflow. Many people use the [GitHub Flow](#) to review and promote code. Broadly, this entails:

1. Creating a branch.
2. Developing your changes.
3. Creating a pull request with your changes.
4. Reviewing your code changes.
5. Merging your changes to `master` and deploying them.

As part of Step 4 above, you can also run tests, often automatically, using continuous integration tools like [Jenkins](#) or third-party services like [TravisCI](#) or [CircleCI](#).

 **NOTE** We'll look at creating tests for our infrastructure in Chapter 7.

Deploy to production

Once we're satisfied that our changes are okay, any tests have passed, and our colleagues have reviewed and approved our code, then it's time to merge and deploy. This is another opportunity to again run `terraform plan`. You can never be too sure that what you're deploying is correct.

You might even automate this process upon the merge of your code into `master`—for example, by using one of the CI tools we've mentioned or a Git hook of some kind. This is an excellent way to combine plan output files. Each commit generates a plan output file that is applied iteratively to your environments.

If this planning reveals that everything is still okay, then it's time to run the `terraform apply` command and push your changes live!

The development environment

Now that we've got an idea of how a workflow might operate, let's dive into the `development` directory and start to define our `development` environment. We're going to start with a `variables.tf` file for the environment.

Listing 6.5: The development variables.tf

```
variable "region" {
    description = "The AWS region."
}
variable "prefix" {
    description = "The name of our org, e.g., examplecom."
}
variable "environment" {
    description = "The name of our environment, e.g., development"
}
variable "key_name" {
    description = "The AWS key pair to use for resources."
}
variable "vpc_cidr" {
    description = "The CIDR of the VPC."
}
variable "public_subnets" {
    default     = []
    description = "The list of public subnets to populate."
}
variable "private_subnets" {
    default     = []
    description = "The list of private subnets to populate."
}
```

We've defined the basic variables required for our environment:

- The region and an AWS key name for any instances created in the environment.
- The name of the environment.
- The network configuration of the VPC containing the environment, including the [CIDR block](#) and any public and private subnets.

Let's populate a [terraform.tfvars](#) file to hold our initial variable values.

Listing 6.6: The development terraform.tfvars file

```
region          = "us-east-1"
prefix         = "examplecom"
environment    = "development"
key_name       = "james"
vpc_cidr       = "10.0.0.0/16"
public_subnets = ["10.0.1.0/24", "10.0.2.0/24"]
private_subnets = ["10.0.101.0/24", "10.0.102.0/24"]
```

We've added default values for all of our `development` environment variables, including the name of the environment itself. This includes the base CIDR block of our VPC, and two lists, each containing public and private subnets.

Now we're going to define a base configuration for our environment. We'll put it in a file called `main.tf`.

Listing 6.7: The development environment main.tf file

```
provider "aws" {
    region      = "${var.region}"
}

module "remote_state" {
    source      = "github.com/turnbullpress/tf_remote_state.git"
    prefix      = "${var.prefix}"
    environment = "${var.environment}"
}

module "vpc" {
    source          = "github.com/turnbullpress/tf_vpc.git?ref=v0
.0.2"
    environment    = "${var.environment}"
    region         = "${var.region}"
    key_name       = "${var.key_name}"
    vpc_cidr       = "${var.vpc_cidr}"
    public_subnets = ["${var.public_subnets}"]
    private_subnets = ["${var.private_subnets}"]
}
```

We've specified the `aws` provider to provide our AWS resources. We're configuring it with variables we've specified, and we've assumed you're using either [AWS environment variables or shared credentials](#). We could also enable [profiles in our shared credentials](#): allowing different users for different environments or configurations.

We've next specified our `remote_state` module to create an S3 bucket for our `development` environments state.

Then we add the `vpc` module. We've specified its `source` as the `v0.0.2` tag of the `tf_vpc` repository on GitHub:

```
github.com/turnbullpress/tf_vpc.git?ref=v0.0.2
```

This is the second generation of the module we created in Chapter 3. It creates a

VPC, public and private subnets, NAT gateways, and all the required routing and security rules for them. The VPC build is modeled on [this AWS scenario](#) which results in a configuration much like:

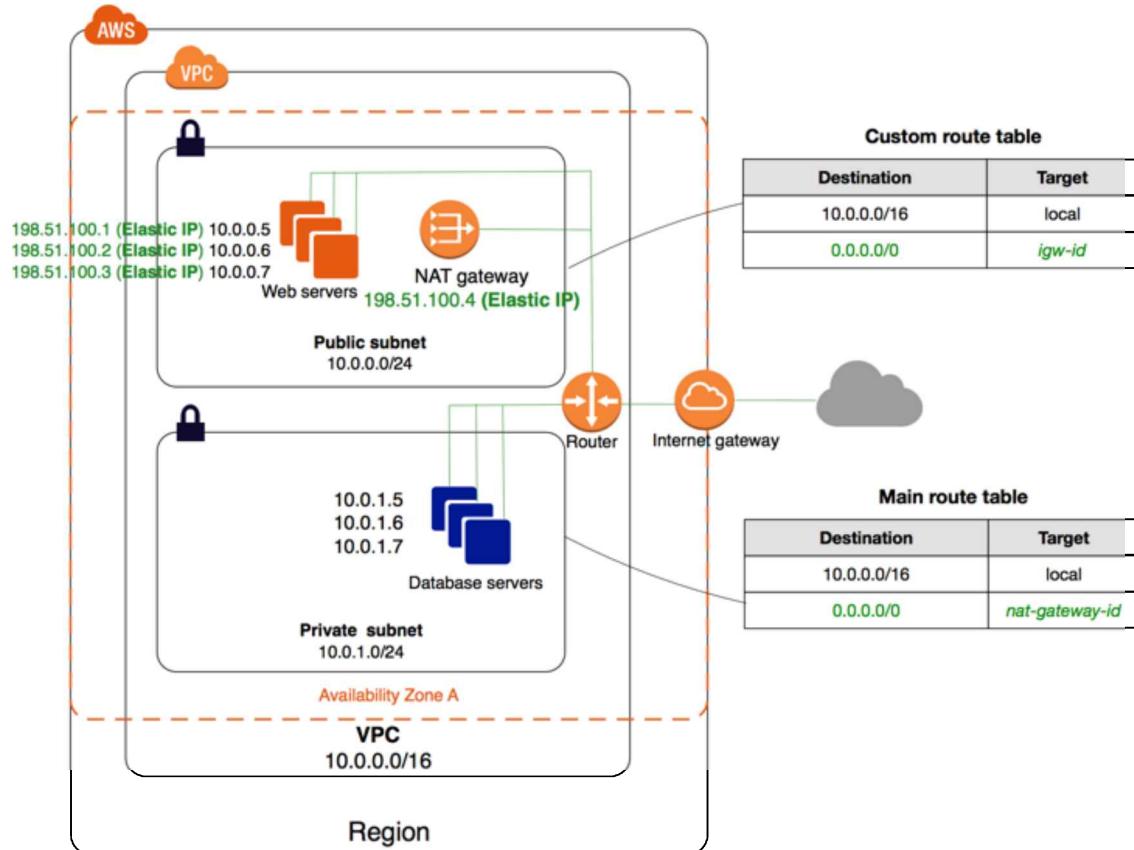


Figure 6.3: AWS VPC Configuration

You can find the updated module in the [tf_vpc repository on GitHub](#). We're not going to step through it in detail, as most of the code is self-explanatory and builds on the work we saw in Chapter 3.

We will, however, quickly look at the outputs of the new version of the module as we're going to reuse several of these shortly.

Listing 6.8: The vpc module outputs

```
output "vpc_id" {
    value = "${aws_vpc.environment.id}"
}
output "vpc_cidr" {
    value = "${aws_vpc.environment.cidr_block}"
}
output "bastion_host_dns" {
    value = "${aws_instance.bastion.public_dns}"
}
output "bastion_host_ip" {
    value = "${aws_instance.bastion.public_ip}"
}
output "public_subnet_ids" {
    value = "${aws_subnet.public.*.id}"
}
output "private_subnet_ids" {
    value = "${aws_subnet.private.*.id}"
}
output "public_route_table_id" {
    value = "${aws_route_table.public.id}"
}
output "private_route_table_id" {
    value = "${aws_route_table.private.*.id}"
}
output "default_security_group_id" {
    value = "${aws_vpc.environment.default_security_group_id}"
}
```

We can see our `vpc` module outputs the key configuration details of our VPC: ID, CIDR block, subnet and route IDs, as well as the DNS and public IP of a bastion host we can use to connect to hosts in our private subnets or without public IP addresses.

Getting our VPC module

Before we use it we'll need to initialize our environment and get our `remote_state` and `vpc` modules from GitHub. Inside the `~/dc/development` directory we need to run the `terraform init` command and then the `terraform get` command to download our modules.

Listing 6.9: Getting the development `remote_state` and `vpc` modules

```
$ cd ~/dc/development
$ terraform init

...
$ terraform get
Get: git::https://github.com/turnbullpress/tf_remote_state.git
Get: git::https://github.com/turnbullpress/tf_vpc.git?ref=v0.0.2
```

This has downloaded the `remote_state` module and `v0.0.2` tag of our `vpc` module and installed it in the `.terraform/modules` directory.

Adding some outputs for the development environment

Let's add some outputs, all based on outputs of the `vpc` module, to the end of the `main.tf` file in our `development` environment. This will expose them for us for later use.

Listing 6.10: The development outputs in main.tf

```
 . . .

output "public_subnet_ids" {
    value = ["${module.vpc.public_subnet_ids}"]
}
output "private_subnet_ids" {
    value = ["${module.vpc.private_subnet_ids}"]
}
output "bastion_host_dns" {
    value = "${module.vpc.bastion_host_dns}"
}
output "bastion_host_ip" {
    value = "${module.vpc.bastion_host_ip}"
}
```

We've added four outputs from the `vpc` module. For example, we've added an output called `bastion_host_dns` with a value of `module.vpc.bastion_host_dns` to expose the bastion host's fully qualified domain name from the `vpc` module.

Planning the development environment

We've passed all of our current variables into the module. If we now run `plan`, we should see our proposed VPC and environment. Let's try that now.

Listing 6.11: Planning our development environment

```
$ terraform plan
. . .

+ module.vpc.aws_eip.environment.0
  allocation_id:      "<computed>"
  association_id:    "<computed>"
  domain:            "<computed>"
  instance:          "<computed>"
  network_interface: "<computed>"
  private_ip:        "<computed>"
  public_ip:         "<computed>"
  vpc:               "true"

. . .

Plan: 23 to add, 0 to change, 0 to destroy.
```

We can see that we'll create 23 resources.

Applying the development environment

Now let's apply our resources.

Listing 6.12: Applying our development VPC

```
$ terraform apply
module.vpc.aws_vpc.environment: Creating...
  cidr_block:          "" => "10.0.0.0/16"
  default_network_acl_id:  "" => "<computed>"

...
Apply complete! Resources: 23 added, 0 changed, 0 destroyed.

...
Outputs:

bastion_host_dns = ec2-52-90-119-131.compute-1.amazonaws.com
bastion_host_ip = 52.90.119.131
private_subnet_ids = [
  subnet-5056af6c,
  subnet-f6bb74ad
]
public_subnet_ids = [
  subnet-f5bb74ae,
  subnet-f4bb74af
]
```

We can see that our 23 resources have been added and that Terraform has outputted some useful data about our new VPC, including details for the bastion host that will allow us to connect to any instances that we launch in the VPC.

It will also have created a `examplecom-remote-state-development` S3 bucket in which we can store our remote state. Let's enable that now by adding the `backend` block to our configuration.

Configuring remote state

Let's configure remote state. We enable the remote state like so:

Listing 6.13: Configuring development environment remote state

```
terraform {
  backend "s3" {
    region = "us-east-1"
    bucket = "examplecom-remote-state-development"
    key    = "terraform.tfstate"
  }
}
```

We've specified our remote state in an S3 bucket and can now run the `terraform init` command to initialize our remote state backend.

Adding the web service

Now let's add our first service to the `development` environment. We're going to configure the web service first. We're going to add all of the variables, the web service module, and the outputs in a single file. Let's call it `web.tf`, add it to our `~/dc/development` directory, and populate it. Note this is the first time we're going to see a second provider added to our environment.

Listing 6.14: The development web.tf file

```
variable "cloudflare_email" {}
variable "cloudflare_token" {}
variable "domain" {
    default = "turnbullpress.com"
}
variable "web_instance_count" {
    default = 2
}
variable "app_instance_count" {
    default = 2
}

provider "cloudflare" {
    email = "${var.cloudflare_email}"
    token = "${var.cloudflare_token}"
}

module "web" {
    source          = "github.com/turnbullpress/tf_web"
    environment     = "${var.environment}"
    vpc_id          = "${module.vpc.vpc_id}"
    public_subnet_ids = "${module.vpc.public_subnet_ids}"
    private_subnet_ids = "${module.vpc.private_subnet_ids}"
    web_instance_count = "${var.web_instance_count}"
    app_instance_count = "${var.app_instance_count}"
    domain          = "${var.domain}"
    region          = "${var.region}"
    key_name         = "${var.key_name}"
}

output "web_elb_address" {
    value = "${module.web.web_elb_address}"
}
output "web_host_addresses" {
    value = ["${module.web.web_host_addresses}"]
}
output "app_host_addresses" {
    value = ["${module.web.app_host_addresses}"]
}
```

We've started by adding some new variables. These variables are specifically for our `web` service, so we're adding them in here instead of the `variables.tf` file, which handles the base variables for our environment. We've added two variables to configure Cloudflare (more on this in a moment), and a third variable to hold the domain name of our web service. We also have two variables for the instance count for our web and app servers.

We've also added a second provider—because it's specific to the web service, we're adding it here and not in the environment's base configuration in `main.cf`. Our second provider, `cloudflare`, manages Cloudflare records and websites. We've used the two variables we just created to specify the email address and API token of our Cloudflare account. We'll need to specify some values for these variables. We're going to put ours in the `terraform.tfvars` file with our other credentials.

Listing 6.15: The Cloudflare variable in `terraform.tfvars`

```
...
environment = "development"
cloudflare_email = "james@example.com"
cloudflare_token = "abc123"
key_name = "james"
...
```

 **TIP** If you'd like to follow along at home you can create a [free Cloudflare account](#) and use their free tier of service to manage a domain name record of your own.

We've next defined a new module, called `web`, for our web service. The `source` of the module is a GitHub repository:

github.com/turnbullpress/tf_web

We're retrieving the `HEAD` of the Git repository rather than a specific tag or commit.

We've also passed in several variables. Some of them are defined in our `variables.tf` file, like the name of our environment and the region in which to launch the service. The key variables, though, are extracted from our `vpc` module.

```
vpc_id = "${module.vpc.vpc_id}"
```

The `module.vpc.vpc_id` references the `vpc_id` output from our VPC module:

Listing 6.16: The vpc_id output

```
output "vpc_id" {
  value = "${aws_vpc.environment.id}"
}
```

This allows us to daisychain modules. The values created by one module can be used in another. We get another benefit from this variable reference: resource ordering. By specifying the `module.vpc.vpc_id` variable, we place the `vpc` module before the `web` module in our dependency graph. This ensures our modules are created in the right sequence but still sufficiently isolated.

Finally, we specify a few useful outputs that will be outputted when we apply our configuration.

Let's take a glance inside our `web` module and see what it looks like.

The web module

We're going to build the `web` module inside our `~/dc/modules` directory.

Listing 6.17: Building the web module

```
$ cd ~/dc/modules
$ mkdir -p web
$ cd web
$ touch {interface,main}.tf
$ mkdir -p files
$ git init
```

We've created a directory, `web`, and a sub-directory, `files`, in our `modules` directory. We've also `touch`'ed the base files—`interface.tf` and `main.tf`—for our module, and initialized the directory as a Git repository.

Let's start with our module's variables in `interface.tf`. We know we need to define a variable for each incoming variable used in our `module` block.

Listing 6.18: The web module's variables

```
variable "region" {}
variable "ami" {
  default = {
    "us-east-1" = "ami-f652979b"
    "us-west-1" = "ami-7c4b331c"
  }
}
variable "instance_type" {
  default = "t2.micro"
}
variable "key_name" {
  default = "james"
}
variable "environment" {}
variable "vpc_id" {}
variable "public_subnet_ids" {
  type = "list"
}
variable "private_subnet_ids" {
  type = "list"
}
variable "domain" {}
variable "web_instance_count" {}
variable "app_instance_count" {}
```

You can see we've defined the inputs from our `module` block plus some additional variables for the Ubuntu 16.04 AMIs to use to create any EC2 instances and the instance type.

We also need to define some outputs for our module to match the outputs we specified in `web.tf`. We'll add these to end of the `interface.tf` file.

Listing 6.19: The web module outputs

```
output "web_elb_address" {
    value = "${aws_elb.web.dns_name}"
}
output "web_host_addresses" {
    value = "${aws_instance.web.*.private_ip}"
}
output "app_host_addresses" {
    value = "${aws_instance.app.*.private_ip}"
}
```

Finally, we've gotten to the heart of our module: the resources we want to create. These are contained in the `main.tf` file. It's too big to show here in its entirety, but you can find it in [the book's source code](#). For now, let's look at a couple specific resources.

Using a data source in the web module

The first resource we're going to look at will show us a useful pattern for using data sources in modules.

Listing 6.20: The web module resources

```
data "aws_vpc" "environment" {
  id = "${var.vpcar.vpc_id}"
}

. . .

resource "aws_security_group" "web_host_sg" {
  name      = "${var.environment}-web_host"
  description = "Allow SSH and HTTP to web hosts"
  vpc_id      = "${data.aws_vpc.environment.id}"

  ingress {
    from_port  = 22
    to_port    = 22
    protocol   = "tcp"
    cidr_blocks = ["${data.aws_vpc.environment.cidr_block}"]
  }

  ingress {
    from_port  = 80
    to_port    = 80
    protocol   = "tcp"
    cidr_blocks = ["${data.aws_vpc.environment.cidr_block}"]
  }

. . .

}
```

You'll first note we haven't specified an AWS provider or credentials. This is because the module will inherit the existing `provider` configuration provided by our `development` environment.

We instead have the `aws_vpc` data source defined as our first resource. The `aws_vpc` data source returns data from a specific VPC. In our `web` module we've used the VPC ID we received from the `vpc` module and passed into the `web`

module.

Data source filters

There's another approach we could take to querying the data source, where we can search for the VPC using a filter. Filtering allows us to search for a specific resource within a data source.

Listing 6.21: A filtered data source

```
data "aws_vpc" "environment" {
  filter {
    name  = "tag:Name"
    values = ["${var.environment}"]
  }
}
```

You can see we've specified a `filter` attribute with two sub-attributes: `name` and `values`.

The `name` attribute should be the name of a specific field to query. In the case of the `aws_vpc` data source, the field to query is derived from [the AWS DescribeVPC API](#), a `tag` called `Name`.

 **TIP** The data source documentation will direct you to the list of potential field names.

If you look at the `vpc` module you'll discover that it creates the VPC and [applies a Name tag using the name of the environment](#):

Listing 6.22: The VPC Name tag

```
    . . .
    tags {
      Name = "${var.environment}"
    }
    . . .
```

The `values` attributes contains a list of potential values for the tag specified. The data source will find any VPCs that match this list. You can only return one VPC though, so the combination of your `name` and `values` attributes must be sufficiently constrained to return the correct resource.

Using the VPC data source data

The `aws_vpc` data source returns a variety of information. Here's an example of what the data source returns:

Listing 6.23: The data.aws_vpc.environment data source

```
module.web.data.aws_vpc.environment:
  id = vpc-c96f7eae
  cidr_block = 10.0.0.0/16
  default = false
  dhcp_options_id = dopt-1ef3017b
  instance_tenancy = default
  state = available
  tags.% = 1
  tags.Name = development
  . . .
```

There's a lot of interesting information here, but most importantly for us, it returns the CIDR block of the VPC. We've updated several security groups in our `main.tf`

file to use that CIDR block in the form of a data source variable as the value of the `cidr_block` attribute:

Listing 6.24: The CIDR block data source variable

```
cidr_block = ["${data.aws_vpc.environment.cidr_block}"]
```

The variable is prefixed with `data`, the type of our data source `aws_vpc`, and the name we've assigned it: `environment`. The suffix is the name of the value returned from the data source query: `cidr_block`.

You can see we've used `data.aws_vpc.environment.id` as a variable as well. This is the VPC ID returned by our data source. We've specified this for consistency to ensure all our data is coming from the same source and specified in the same form, but we could instead use our `var.vpc_id` value.

 **TIP** We could also pull this information from the Terraform remote state via the `terraform_remote_state` provider. We saw this approach in Chapter 5.

Web instances

Our next resources are the AWS instances that will run our web servers.

Listing 6.25: The web instance

```
resource "aws_instance" "web" {
    ami           = "${lookup(var.ami, var.region)}"
    instance_type = "${var.instance_type}"
    key_name      = "${var.key_name}"
    subnet_id     = "${var.public_subnet_ids[0]}"
    user_data     = "${file("${path.module}/files/web_bootstrap.sh"
  )}"
    vpc_security_group_ids = [
        "${aws_security_group.web_host_sg.id}"
    ]
    tags {
        Name = "${var.environment}-web-${count.index}"
    }
    count = "${var.web_instance_count}"
}
```

This is very similar to the EC2 resources we created earlier in the book. There are a couple of interesting features we should note.

The first feature is the selection of the subnet in which to run the instance. We're taking advantage of a sorting quirk of Terraform to select a specific subnet for our instances. We're passing in the list of public and private subnet IDs into the `web` module. These are generated in the `vpc` module and emitted as an output of that module. We've assigned them in the `web` module to a variable called `var.public_subnet_ids`.

The original subnets were created using values from our `development` environment's `terraform.tfvars` file:

Listing 6.26: The public and private subnets

```
public_subnets = [ "10.0.1.0/24", "10.0.2.0/24" ]
private_subnets = [ "10.0.101.0/24", "10.0.102.0/24" ]
```

Each subnet is created in the order specified—here `10.0.1.0/24` and then `10.0.2.0/24`. The resulting subnet IDs are also outputted in the order they are created from the `vpc` module.

Listing 6.27: The public_subnet_ids output

```
public_subnet_ids = [
    subnet-f5bb74ae,
    subnet-f4bb74af
]
```

We select the first element in the `var.public_subnet_ids` and know we're getting the `10.0.1.0/24` subnet and so on.

The second interesting feature is contained in the `user_data` attribute for our instance:

```
user_data = "${file("${path.module}/files/web_bootstrap.sh")}"
```

The `user_data` uses the `file` function to point to a file inside our module. To make sure we find the right path, we've used the `path` variable to help us locate this file. We briefly saw the `path` variable in Chapter 3.

The `path` variable can be suffixed with a variety of methods to select specific paths. For example:

- `path.cwd` for current working directory.
- `path.root` for the root directory of the root module.

- `path.module` for the root directory of the current module.

Here we're using `path.module`, which is the root directory of the `web` module. Hence our `web_bootstrap.sh` script is located in the `files` directory inside the root of the `web` module.

 **NOTE** The Terraform interpolation documentation [explains the path variable](#) in more detail.

A Cloudflare record

We've also specified a resource to manage our Cloudflare record. Let's take a look at that now.

Listing 6.28: The Cloudflare record resource

```
resource "cloudflare_record" "web" {  
    domain = "${var.domain}"  
    name = "${var.environment}.${var.domain}"  
    value = "${aws_elb.web.dns_name}"  
    type = "CNAME"  
    ttl = 3600  
}
```

The `cloudflare_record` resource creates a DNS record—in our case a `CNAME` record. We've created our `CNAME` record by joining our environment name, `development`, with the domain name, `turnbullpress.com`, we specified for the `var.domain` variable in our `~/dc/development/web.tf` file. Our `CNAME` record points to the DNS name of our AWS load balancer provided by the variable

`aws_elb.web.dns_name`. Specifying this variable guarantees our Cloudflare record will be created after our load balancer in the dependency graph.

Committing our module

Finally, we commit and add our module to GitHub.

Listing 6.29: Committing the web module

```
$ git add .
$ git commit -a -m "Adding initial module"
```

We would then [create a GitHub repository](#) to store our module and push it.

Getting our web module

Now that we've added our module to the `development` environment, we need to get it.

Listing 6.30: Getting the web module

```
$ cd ~/dc/development
$ terraform get
Get: git::https://github.com/turnbullpress/tf_vpc.git?ref=v0.0.2
Get: git::https://github.com/turnbullpress/tf_web.git
```

We've now downloaded the `web` module to join our `vpc` module.

Planning our web service

If we again run `terraform plan`, we'll see some new resources to be added.

Listing 6.31: Planning for our web module

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but
will not be persisted to local or remote state storage.

module.vpc.aws_vpc.environment: Refreshing state... (ID: vpc-782
c281f)
module.vpc.aws_eip.environment.0: Refreshing state... (ID:
eipalloc-52336f6d)
module.vpc.aws_eip.environment.1: Refreshing state... (ID:
eipalloc-bf366a80)

...
Plan: 9 to add, 0 to change, 0 to destroy.
```

Terraform has refreshed our state to check what's new and has identified nine resources, the contents of our `web` module, that will be added to the `development` environment.

Applying the web service

When we run `terraform apply` we'll get our first service installed in the development environment.

Listing 6.32: Applying the web module

```
$ terraform apply
module.vpc.aws_vpc.environment: Refreshing state... (ID: vpc-782
c281f)

...
Apply complete!! Resources: 9 added, 0 changed, 0 destroyed.

...
Outputs:

app_host_addresses = [
  10.0.101.238,
  10.0.101.174
]
bastion_host_dns = ec2-52-90-119-131.compute-1.amazonaws.com
bastion_host_ip = 52.90.119.131
private_subnet_ids = [
  subnet-5056af6c,
  subnet-f6bb74ad
]
public_subnet_ids = [
  subnet-f5bb74ae,
  subnet-f4bb74af
]
web_elb_address = development-web-elb-1020554483.us-east-1.elb.
amazonaws.com
web_host_addresses = [
  10.0.1.56,
  10.0.1.233
]
```

Terraform has created our new resources and outputted their information, in addition to the previous `vpc` outputs. We can also see our new Cloudflare domain record has been created.

Listing 6.33: The Cloudflare record

```
module.web.cloudflare_record.web: Creating...
  domain:    ""  => "turnbullpress.com"
  hostname:  ""  => "<computed>"
  name:      ""  => "development.turnbullpress.com"
  proxied:   ""  => "false"
  ttl:       ""  => "3600"
  type:     "CNAME"
  value:    "development-web-elb-1020554483.us-east-1.elb.
amazonaws.com"
  zone_id:   ""  => "<computed>"
module.web.cloudflare_record.web: Creation complete
```

This shows us that our second provider is working too.

Testing our web service

Let's try to use a couple of those resources now. We'll browse to our load balancer first.

→ C ① development-web-elb-1020554483.us-east-1.elb.amazonaws.com

The Terraform Book Web service

Figure 6.4: Our web service

Let's also sign in to our bastion host, 52.90.119.131, and then bounce into one of our web server hosts.



NOTE This assumes we have a copy of the `james` key pair that we used to

create our instances locally on our Terraform host.

Listing 6.34: SSH'ing into the bastion host

```
$ ssh -A ubuntu@52.90.119.131
The authenticity of host '52.90.119.131 (52.90.119.131)' can't
be established.
ECDSA key fingerprint is SHA256:g/Jfap5CgjZVEQoxDsAVDMILToEHfY/
mQ13mzLjXJe8.
Are you sure you want to continue connecting (yes/no)? yes
.
.
.
ubuntu@ip-10-0-1-224:~$
```

We've connected to the bastion host, specifying the `-A` flag on the SSH command to enable agent forwarding so we can use the `james` key pair on subsequent hosts.

From the bastion host we should be able to `ping` one of our web hosts and SSH into it by using its private IP address in the `10.0.1.0/24` subnet.

Listing 6.35: Ping a web host

```
ubuntu@ip-10-0-1-224:~$ ping 10.0.1.56
PING 10.0.1.56 (10.0.1.56) 56(84) bytes of data.
64 bytes from 10.0.1.56: icmp_seq=1 ttl=64 time=1.17 ms
64 bytes from 10.0.1.56: icmp_seq=2 ttl=64 time=0.613 ms
64 bytes from 10.0.1.56: icmp_seq=3 ttl=64 time=0.598 ms
^C
ubuntu@ip-10-0-1-224:~$ ssh ubuntu@10.0.1.56
.
.
.
ubuntu@ip-10-0-1-56:~$
```

Excellent! It all works and we're connected. We now have a running service inside a custom built VPC that's secured and can be managed via a bastion host. What's even more awesome is that this is totally repeatable in other environments.

Removing our web service

If we want to remove the web service, all we need to do is remove the `web.tf` file from the `~/dc/development` directory and run `terraform apply` to remove the web service resources. If we can remove the file and run `terraform plan` we can see the proposed deletions.

Listing 6.36: Removing the web service

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but
will not be persisted to local or remote state storage.

. . .

- module.web.aws_elb.web
- module.web.aws_instance.app.0
- module.web.aws_instance.app.1
- module.web.aws_instance.web.0
- module.web.aws_instance.web.1
- module.web.aws_security_group.app_host_sg
- module.web.aws_security_group.web_host_sg
- module.web.aws_security_group.web_inbound_sg

Plan: 0 to add, 0 to change, 8 to destroy.
```

But instead of removing these resources, let's finish our `development` environment by adding our second service, an API service.

Adding the API service

Like our web service, the API service lives in a file in our `~/dc/development` directory. We've called this file `api.tf`.

Listing 6.37: The API service in the api.tf file

```
variable "api_instance_count" {
    default = 5
}

module "api" {
    source          = "github.com/turnbullpress/tf_api"
    environment     = "${var.environment}"
    vpc_id          = "${module.vpc.vpc_id}"
    public_subnet_ids = "${module.vpc.public_subnet_ids}"
    private_subnet_ids = "${module.vpc.private_subnet_ids}"
    region          = "${var.region}"
    key_name         = "${var.key_name}"
    api_instance_count = "${var.api_instance_count}"
}

output "api_elb_address" {
    value = "${module.api.api_elb_address}"
}

output "api_host_addresses" {
    value = ["${module.api.api_host_addresses}"]
}
```

In our `api.tf` file we've specified a new variable for this module: the count of API servers we'd like to create. We've set a default of `5`. We've also specified the module that will provision our API service. Our module is sourced from the `github.com/turnbullpress/tf_api` repository on GitHub. We've passed in an identical set of variables as we did to our `web` module, plus our new API server count.

The API module

Our `api` module will create five AWS instances, a load balancer in front of our instances, and appropriate security groups to allow connectivity. We're not going to create each individual file; rather we're going to check out the existing module from GitHub.

Listing 6.38: Checking out the API module

```
$ cd ~/dc/modules  
$ git clone https://github.com/turnbullpress/tf_api.git api
```

The resulting file tree will look like:

```
api  
├── LICENSE  
├── README.md  
├── files  
│   └── api_bootstrap.sh  
└── interface.tf  
    └── main.tf
```

Figure 6.5: Our API service directory tree

API instances

Let's take a peak at the `aws_instance` resource in our `api` module's `main.tf` file to see how we've used our new count variable.

Listing 6.39: The API aws_instance resource

```
resource "aws_instance" "api" {
    ami           = "${lookup(var.ami, var.region)}"
    instance_type = "${var.instance_type}"
    key_name      = "${var.key_name}"
    subnet_id     = "${var.public_subnet_ids[1]}"
    user_data     = "${file("${path.module}/files/api_bootstrap.sh"
  )}"

    vpc_security_group_ids = [
        "${aws_security_group.api_host_sg.id}",
    ]

    tags gs {
        Name = "${var.environment}-api-${count.index}"
    }

    count = "${var.api_instance_count}"
}
```

Our `aws_instance.api` resource looks very much like our earlier instance resources from the `web` module. We've specified a different subnet, the second public subnet in our VPC: `10.0.2.0/24`.

We've also used the `var.api_instance_count` variable as the value of the `count` meta-parameter. This allows us to avoid hard-coding a number of API servers in our configuration. Instead we can set the value of this variable, either in an `terraform.tfvars` file, or via one of the variable population methods we explored in Chapter 3. This means we can raise and lower the number of instances in our API cluster simply and quickly.

 **NOTE** You can find the full module source code in the [tf_api](#) repository on GitHub.

The API service outputs

Back in our `~/dc/development/api.tf` file we've also specified two outputs:

Listing 6.40: The API module outputs

```
output "api_elb_address" {
  value = "${module.api.api_elb_address}"
}
output "api_host_addresses" {
  value = ["${module.api.api_host_addresses}"]
```

One will return the DNS address of the API service load balancer, and one will return the host addresses for the API servers.

Getting our module

In order to use our API module, we need to get it.

Listing 6.41: Getting the API module

```
$ cd ~/dc/development
$ terraform get -update
Get: git::https://github.com/turnbullpress/tf_api.git (update)
Get: git::https://github.com/turnbullpress/tf_vpc.git?ref=v0.0.2
    (update)
Get: git::https://github.com/turnbullpress/tf_web.git (update)
```

Like our `web` module, we're pulling the `HEAD` of the Git repository, rather than a tag or commit.

Planning the API service

With the `api` module in place we can then plan the service.

Listing 6.42: The API service plan

```
$ terraform plan

.

.

+ module.web.aws_elb.web
  availability_zones.#:                      "<computed>"
  connection_draining:                         "false"
  connection_draining_timeout:                 "300"
  cross_zone_load_balancing:                  "true"

.

.

Plan: 8 to add, 0 to change, 0 to destroy.
```

You can see we'll add eight resources to our existing infrastructure.

Applying the API service

Let's do that now by applying our configuration.

Listing 6.43: The API service

```
$ terraform apply
module.vpc.aws_vpc.environment: Refreshing state... (ID: vpc-782
c281f)
module.vpc.aws_eip.environment.1: Refreshing state... (ID:
eipalloc-bf366a80)

...
Apply complete! Resources: 8 added, 0 changed, 0 destroyed.

...
Outputs:

api_elb_address = development-api-elb-1764248438.us-east-1.elb.
amazonaws.com
api_host_addresses = [
    10.0.2.120,
    10.0.2.172,
    10.0.2.64,
    10.0.2.72,
    10.0.2.239
]
...
```

Now we have two services running in our `development` environment. We can see that our API service has created a new load balancer and five API hosts located behind it in our `10.0.2.0/24` public subnet.

Testing our API service

Let's test our API is working by `curl`'ing an API endpoint on the load balancer.

Listing 6.44: Testing the API service

```
$ curl development-api-elb-1764248438.us-east-1.elb.amazonaws.com/api/users/1
{
  "id": 1,
  "name": "user1"
}
```

Our API is very simple and only has one endpoint: `/api/users`. With `/api/users/1` we can return a single user entry, or with `/api/users` we can return all of the users in our API's database.

We can also test the API is working by visiting the load balancer via browser.

← C ⓘ development-api-elb-1764248438.us-east-1.elb.amazonaws.com

The Terraform Book API service

Figure 6.6: Our API service

Finally, we can easily remove our API service by removing the `api.tf` file and running `terraform apply` to adjust our configuration. We can also adjust the API instance count to grow or shrink our pool of API instances.

⚠️ WARNING Remember that our environment has a dependency graph. If we remove the `main.tf` file containing our VPC configuration, the VPC and everything in it will be destroyed!

Adding a production environment

Now that we've got a functioning `development` environment, we can extend this architecture to add new environments. Let's populate a `production` environment now. Our architecture makes this very easy: we can just duplicate our existing environment and update its variables.

Listing 6.45: Adding a production environment

```
$ cd ~/dc
$ mkdir -p production
$ cp -R development/{main.tf,variables.tf,terraform.tfvars}
production/
```

We've copied the `main.tf`, `variables.tf`, and `terraform.tfvars` into the `production` directory. We haven't included any of the services yet. Let's update our configuration files, starting with `terraform.tfvars`, for our new environment.

Listing 6.46: The production terraform.tfvars

```
region = "us-east-1"
environment = "production"
key_name = "james"
vpc_cidr = "10.0.0.0/16"
public_subnets = [ "10.0.1.0/24", "10.0.2.0/24" ]
private_subnets = [ "10.0.101.0/24", "10.0.102.0/24" ]
```

You can see we've updated our `environment` variable for the `production` environment. We could also adjust other settings, like the `region` or the public and private subnets, to suit our new environment.

Adding services to production

We can now add services, such as our web service, to the environment by duplicating—and potentially adjusting—the `web.tf` file from our `development` environment. With the core of our modules driven by variables, we’re likely to only need to adjust the scale and size of the various services rather than their core configuration.

Listing 6.47: Adding the `web.tf` file

```
$ cd ~/dc
$ cp development/web.tf production/
$ vi production/web.tf
. . .
```

We would wrap this whole process in the workflow we introduced at the beginning of the chapter to ensure our changes are properly managed and tested.

 **NOTE** Remember, we’ve also created a Git repository for our data center environment. You can find it [on GitHub here](#).

State environments

Another useful feature when thinking about workflow are [Terraform state environments](#). State environments were introduced in Terraform 0.9. You can think about a state environment as branching version control for your Terraform resources.

 **NOTE** Currently, state environments are only supported by the S3 and Consul backends.

A state environment is a namespace, much like a version control branch. They allow a single folder of Terraform configuration to manage multiple states of resources. They are useful for isolating a set of resources to test changes during development. Unlike version control though, state environments do not allow merging, any changes you make in a state environment need to be re-applied to any other environments.

Right now you are operating in a state environment, a special, always present, environment called `default`. If you're running a Consul backend, you can see that by running the `terraform workspace` command with the `list` flag.

Listing 6.48: Running the `terraform workspace list` command

```
$ terraform workspace list  
default
```

As they are pretty limited right now we're not going to cover them in more details but you can learn more about [state environments in the documentation](#).

Other resources for Terraform environments

There are also some tools, blog posts, and resources designed to help run Terraform in a multi-environment setup.

- [Terrenv](#) — RubyGem for setting up Terraform environments.
- [Terragrunt](#) — Go tool for managing locking and state that can be used as glue in a multi-environment setup.

- [Terraform, VPC, and why you want a Tfstate file per env](#) — Great post by Charity Majors on running Terraform in multiple environments.
- [How I structure Terraform configurations](#) — Useful post from Anton Babenko on structuring Terraform code and environments.
- [Terraform at Scale](#) — Video from Calvin French-Owen (Co-founder and CTO at Segment) talking about running Terraform at scale.

Summary

In this chapter we've seen how to build a multi-environment architecture with Terraform. We've proposed a workflow to ensure our infrastructure is managed and changed as carefully as possible. We've built a framework of environments to support that workflow. We've also seen how to build isolated and portable services that we can deploy safely and simply using modules and configuration.

In the next chapter we'll look at how to expand upon one section of our workflow: automated testing. We'll learn how to write tests to help validate changes to our infrastructure.

Chapter 7

Infrastructure testing

In the last chapter we built a multi-environment architecture and workflow for Terraform. This helps us manage change in our infrastructure. One of the steps in our workflow was testing. If you're an application developer or software engineer, you are probably familiar with tests and testing.

Software tests validate that your software does what it is supposed to do. Loosely, they're a combination of quality measures and correctness measures. We're going to apply some of the principles of software testing to our infrastructure. Wait, what? Yep. A software unit test, at its heart, confirms that an isolated unit of code performs as required. Inputs to the unit of code are applied, the code is run, and the outputs are confirmed as valid. A Terraform resource is a unit of isolated code about which we can reason and write tests to ensure the combination of the inputs and execution result in the correct outputs. With Terraform this is made even easier by the declarative nature of resources.

We can even apply more advanced software testing approaches like [Test-driven development](#) (TDD) to infrastructure testing. In TDD the requirements of your software are turned into a series of unit tests. These unit tests are written before the code itself. This means all of our unit tests fail initially, but slowly begin to pass as we code to satisfy our requirements. TDD ensures a developer focuses on

the requirements before writing the code.

In this chapter we're going to look at a testing framework on top of our multi-environment architecture. We're going to build on top of the code we developed in Chapter 6, write tests for some of that code, and demonstrate how we use those tests.

Sadly, testing on Terraform is still in the early stages and has limitations. At the moment there are a limited set of testing frameworks and harnesses that support Terraform. We're going to see what we can achieve now by looking at a tool called Test Kitchen. This chapter will evolve as tools get better and easier to use.

Test Kitchen

[Test Kitchen](#) is a test harness to execute infrastructure and configuration management code on isolated platforms. It builds your infrastructure, configuration, or environment, and then validates it against a series of tests. It relies on a series of drivers that target various providers and platforms: Amazon, Digital Ocean, Vagrant, Docker, etc. One of the plugin drivers available is [Terraform](#).

As Test Kitchen is a test harness, a wide variety of testing frameworks are supported including [InSpec](#), [Serverspec](#), and [RSpec](#).

InSpec

We're going to use [InSpec](#) with Test Kitchen to test our Terraform-built infrastructure. InSpec is an infrastructure-testing framework built around the concept of compliance controls. You write a series of “controls”—compliance statements backed with individual tests, for example:

- Control — SSH should be enabled and securely configured.
 - Tests

- * Is the `sshd` daemon running?
- * Is `Protocol 2` set?
- * Is the `sshd` daemon running on port 2222?
- * Etc.

How Test Kitchen works

Test Kitchen works by creating the infrastructure we want to test, connecting to it, and running a series of tests to validate the right infrastructure has been built.

The integration with Terraform comes via an add-on called `kitchen-terraform`. The `kitchen-terraform` add-on is installed via a gem. It is made up of a series of plugins:

- Driver — This is a wrapper around the `terraform` binary and allows Test Kitchen to execute Terraform commands.
- Provisioner — The bridge between Terraform and Test Kitchen. It manages the Terraform configuration and works with the driver to process Terraform state during test runs.
- Transport — The networking code that allows Test Kitchen to connect to your Terraform-built hosts.
- Verifier — The verifier is a wrapper around `InSpec`. The verifier runs our actual tests and returns the results.

We're going to need to install some prerequisites to get Test Kitchen up and running.

Prerequisites

The biggest prerequisite is that Test Kitchen requires SSH access to any hosts upon which you want to run tests. It currently doesn't support using a bastion or bounce

host to get this connectivity. This means that unless you have SSH access to your hosts, Test Kitchen will not be able to run tests on hosts in private subnets or without public IP addresses.

 **NOTE** This is a [big](#) limitation right now. There's [an issue open to address using a bastion host to run tests on GitHub](#) but nothing's available yet.

Test Kitchen is written in Ruby and requires Ruby 2.3.1 or later installed. If your platform doesn't have native Ruby 2.3.1 installed, you can manage Ruby with tools like [rvm](#) and [rbenv](#).

Listing 7.1: Testing the Ruby version

```
$ ruby -v  
ruby 2.2.2p95 (2015-04-13 revision 50295) [x86_64-darwin15]
```

rbenv

Assuming we don't have Ruby 2.3.1 available, let's quickly install [rbenv](#) and add the correct Ruby version.

Listing 7.2: Installing rbenv

```
$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
```

 **TIP** On OS X you can also do [brew install rbenv](#) to install rbenv.

Add the `rbenv` binary to your path. On Bash we'd run:

Listing 7.3: Adding rbenv to the path

```
$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bash_profile
```

We then enable auto-activation of `rbenv`. To do this follow the instructions emitted from:

Listing 7.4: Auto-activating rbenv

```
$ ~/.rbenv/bin/rbenv init
# Load rbenv automatically by appending
# the following to ~/.bash_profile:

eval "$(rbenv init -)"
```

So, if we were running a Bash shell we'd add the `eval`.

Listing 7.5: Adding the rbenv activating

```
$ echo 'eval "$(rbenv init -)"' >> ~/.bash_profile
```

Restart your terminal to update the path and activate `rbenv`. You can then confirm it is working using:

Listing 7.6: Confirming rbenv worked

```
$ type rbenv  
rbenv is a function
```

We can now install the required Ruby version with the `rbenv` binary.

Listing 7.7: Installing Ruby with rbenv

```
$ rbenv install 2.3.1  
Downloading ruby-2.3.1.tar.bz2...  
-> https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.1.tar.bz2  
...
```

 **NOTE** On OS X El Capitan you might strike [this issue](#).

We can then confirm that Ruby 2.3.1 is installed like so:

Listing 7.8: Checking Ruby 2.3.1 is installed

```
$ rbenv versions  
* system (set by /Users/james/.rbenv/version)  
2.3.1
```

Our Ruby install should also provide RubyGems and the `gem` binary.

Gems and Bundler

Next we need to install the Bundler gem and bundle some gems. The gem installation is specific to each Test Kitchen installation so we need to do this inside our `~/dc` directory.

If we've installed `rbenv`, we need to ensure our local Terraform directory is using it.

Listing 7.9: Enabling Ruby 2.3.1

```
$ cd ~/dc  
$ rbenv local 2.3.1
```

This will set the local Ruby version to 2.3.1 and create a file called `.ruby-version` with our Ruby version in it. You can add this file to your repository to set the version for other users if they clone your repository.

Listing 7.10: Adding the `.ruby-version` file

```
$ git add .ruby-version  
$ git commit -a -m "Added Ruby version file"
```

We can now install Bundler.

Listing 7.11: Install Bundler

```
$ gem install bundler
```

Next, we create a `Gemfile` in `~/dc` to hold the list of gems to install. We populate it like so:

Listing 7.12: Our Test Kitchen Gemfile

```
source 'https://rubygems.org/'  
ruby '2.3.1'  
  
gem 'test-kitchen'  
gem 'kitchen-terraform'
```

Now we run Bundler to install the gems.

Listing 7.13: Running Bundler for Test Kitchen

```
$ bundle install  
Installing artifactory 2.5.0  
Installing builder 3.2.2  
  
...  
  
Installing kitchen-inspec 0.15.2  
Installing kitchen-terraform 0.3.0  
Bundle complete! 2 Gemfile dependencies, 48 gems now installed.  
Use `bundle show [gemname]` to see where a bundled gem is  
installed.
```

Creating a test configuration

Now that we have everything we need for Test Kitchen installed, we can create a test configuration. We're going to create a configuration in each environment, currently `development` and `production`. Let's start with `development`.

Test Kitchen stores all of its information about state in a special directory called `.kitchen` at the root of our environment. Test Kitchen also uses a special YAML configuration file, `.kitchen.yml`, that tells Test Kitchen how and what to test.

Let's create a `.kitchen.yml` at the root of the `development` environment, `~/dc/development`.

Listing 7.14: The `development/.kitchen.yml` file

```
---
driver:
  name: terraform

provisioner:
  name: terraform
  variable_files:
    - terraform.tfvars

platforms:
  - name: ubuntu

transport:
  name: ssh
  ssh_key: ~/.ssh/james_aws

verifier:
  name: terraform
  format: doc
  groups:
    - name: bastion
      hostnames: bastion_host_dns
      username: ubuntu

suites:
  - name: default
```

Driver

Our first piece of configuration, the `driver`, specifies which driver our tests will use. Ours is the `terraform` driver and it doesn't have, nor require, any configura-

tion options.

Provisioner

Next we specify the `provisioner`, again `terraform`. The provisioner needs to know about our Terraform configuration and where to find it. By default it looks in the current directory for any Terraform configuration files—in our case that will be the files in the `~/dc/development` directory. When the `terraform` driver runs Terraform `plan` and `apply` commands, it'll expect to find everything it needs in this directory.

We also need to tell the provisioner about any Terraform variable definition files, our `terraform.tfvars` file, and any other variable definition files we've created. In our case we've used the `variable_files` option to tell Test Kitchen about our `terraform.tfvars` file. This will load all the variable values prior to any infrastructure being built or tests being run.

Platforms

The `platforms` block specifies the target operating systems for our tests. Here we only have one: `ubuntu`. All of our AMIs are Ubuntu 16.04 hosts. This option allows you to customize tests for specific operating systems.

Transport

The `transport` block configures our SSH connections to the hosts upon which we wish to run tests. We specify that Test Kitchen should use the `ssh` transport and provide the location of an SSH key, using the `ssh_key` option, that can be used to log on to the hosts. In our case this is the AWS key pair we've used when creating our EC2 instances.

 **NOTE** You'll need to ensure the host running Test Kitchen can connect via SSH to the hosts upon which you wish to run tests.

Verifier

The next block is the `verifier`. The `verifier` block defines our test configuration and performs the actual test verification. Our verifier is named `terraform`, and we've configured a few different attributes for it.

The first attribute is the format of our test output. We've selected `doc` which is neat and structured documentation-style output. We could also output in `json` or other formats. If you're reviewing your test output on screen, then `doc` is probably going to work best. The `doc` output looks something like:

Listing 7.15: The doc test format

```
Service sshd
  should be enabled
  should be running

Finished in 0.30054 seconds (files took 2.53 seconds to load)
2 examples, 0 failures
```

This should be fairly familiar if you've used Ruby-based testing frameworks like RSpec before. The tests that are executed are shown, and a summary is presented with the number of tests that ran as well as the number of failures.

In our `verifier` block we also specify `groups`. Each group is a collection of tests and infrastructure we'd like to test against. We've only specified one group: `bastion`.

Listing 7.16: The verifier block

```
verifier:  
  name: terraform  
  format: doc  
  groups:  
    - name: bastion  
      hostnames: bastion_host_dns  
      username: ubuntu
```

There are a number of attributes you can configure for each group. The `hostnames` attribute contains the list of hostnames of the hosts that belong to this group. Test Kitchen needs the hostnames so that it can connect to them using SSH.

Test Kitchen expects to see the value of the `hostnames` attribute as an output from your current configuration. In our case we've specified the `bastion_host_dns` output.

Listing 7.17: The bastion_dns_host

```
output "bastion_host_dns" {  
  value = "${module.vpc.bastion_host_dns}"  
}
```

We also specify the `username` Test Kitchen should use to sign in to the target hosts—in our case `ubuntu` to match the default Ubuntu user.

Suites

Finally, we specify test suites. Test suites are collections of tests. We've specified one suite, called `default`.

Directory structure

The `suites` block also dictates the Test Kitchen directory structure. Test Kitchen expects to find a directory called `default` under a `test/integration` directory in our `development` environment. Let's create the structure now.

Listing 7.18: Creating the Test Kitchen directory structure

```
$ mkdir -p ~/dc/development/test/integration/default/controls
```

The `default` directory is the root of our test suite, and the `controls` directory is going to hold our controls and their associated tests.

We then need to create an `inspec.yml` inside the `default` directory to let Test Kitchen know this is the right place.

Listing 7.19: Creating the inspec.yml file

```
$ touch ~/dc/development/test/integration/default/inspec.yml
```

Populating the `inspec.yml` file:

Listing 7.20: Creating the inspec.yml file

```
---
name: default
title: 'Default suite of tests'
summary: 'A collection of controls to test baseline host
configuration'
maintainer: 'James Turnbull'
version: 1.0.0
```

You can populate the `inspec.yml` file with a variety of metadata to identify the suite of tests. The `name` setting is the only required setting, but other settings help to describe the purpose of your suite.

We can test whether the resulting suite of tests is valid is using the `inspec` binary. Let's do that now.

Listing 7.21: Testing the suite configuration

```
$ cd ~/dc/development
$ bundle exec inspec check test/integration/
Location: test/integration/
Profile: default
Controls: 0
Timestamp: 2016-11-05T13:55:43-04:00
Valid: true

! test/integration/default/inspec.yml:0:0: Missing profile
copyright in inspec.yml
! No controls or tests were defined.

Summary: 0 errors, 2 warnings
```

We've used the `bundle exec` command to run the `inspec` binary. We specify the `check` command and the location of our test suites, `test/integration`.

Creating our first control

We specified a single test suite in our `.kitchen.yml` configuration: `default`. Test Kitchen will run all of the controls it finds in the `test/integration/default/controls` directory.

We're going to start by creating a control called `operating_system`. Our `operating_system` control will test that the right AMI is being built for our bastion host instance with a series of tests. Let's create our base control and an

initial test now.

For a control named `operating_system`, Test Kitchen expects to find a file called `operating_system_spec.rb` in the `test/integration/default/controls` directory. We put our `operating_system` controls inside that file.

Listing 7.22: The operating_system control

```
control 'operating_system' do
  describe command('lsb_release -a') do
    its('stdout') { should match (/Ubuntu 16.04/) }
  end
end
```

Test Kitchen's InSpec controls are expressed in a Ruby DSL (Domain Specific Language) that will be familiar to anyone who has used RSpec, as it's built on top of it. In our example, we can see two of the most common DSL blocks: `control` and `describe`.

The `control` block wraps a collection of controls. The `describe` block wraps individual controls. The `describe` block must contain at least one control. A `control` block must contain at least one `describe` block, but may contain as many as needed.

The `describe` block is constructed like so:

Listing 7.23: The operating_system control

```
describe command('lsb_release -a') do
  its('stdout') { should match (/Ubuntu 16.04/) }
end
```

Each control is made up of resources and matchers that are combined into tests. Resources are components that execute checks of some kind for a test: run a

command, check a configuration setting, check the state of a service, and so on. InSpec has a long list of [built-in resources](#) and has the ability for you to [write your own custom resources](#). Matchers are [a series of methods](#) that check, by various logic, if output from a resource matches the output you expect. So a matcher might test equality, presence, or a regular expression.

Inside our `describe` block, we use a resource to perform an action on the host. For example, this block uses the `command` resource to run a command on the host. The `command` resource's output is captured in `stdout`, and then we specify a block for the test itself. So our test unfolds like so:

1. The `lsb_release` binary is run via the `command` resource and captures the output.
2. The `stdout` of the `command` is then checked for a regular expression `match` of `Ubuntu 16.04`.
3. If the match is made, the test passes. Otherwise, it fails.

Decorating controls with metadata

You can also decorate `control` blocks with metadata. For example:

Listing 7.24: The operating_system control

```
control 'operating_system' do
  title 'Operating system controls'
  desc "
    Checks that the host's operating system is correctly
    configured.
  "
  tag 'operating_system', 'ubuntu'

  .
  .

end
```

This adds metadata to the test to help folks understand what the test does and, importantly, why the test failing matters. In this example we've decorated our control with a title and a description—a plain-English explanation of what it is and how it works. We've also added a couple of tags to the control.

 **TIP** This is a small selection of the metadata available. You can find more in the [InSpec DSL documentation](#).

Adding another test to our control

Let's add another test to our control. We're going to confirm that several services are enabled and running. We're also going to see how we can intermingle Ruby with our InSpec controls.

Listing 7.25: Adding a second test

```
control 'operating_system' do
  ...
  services = [ 'cron', 'rsyslog' ]
  services.each do |service|
    describe service(service) do
      it { should be_enabled }
      it { should be_running }
    end
  end
end
```

We've added an array of service names: `cron` and `rsyslog`. We've then iterated through that array and passed each element into a `describe` block. Inside the `describe` block we've now got two tests. These tests use the `service` resource, which helps you test the state of services from a variety of service managers. For each service passed to the block, we test if they are enabled and if they are running. If one, or both, fail, we'll see corresponding output.

Now let's finish setting up our Test Kitchen environment, then run our new controls and see what happens!

Setting up a Test Kitchen instance

Test Kitchen has a simple workflow.



Figure 7.1: The Test Kitchen workflow

Test Kitchen runs tests, in our case our InSpec controls, inside what it calls an “instance.” You can think about the instance as the test environment state. An instance usually exists for the period of a test run: it is created, converges our infrastructure, verifies that the tests pass, and then it’s destroyed.

The name of the instance is the combination of a test suite and a platform. In our case we have one platform defined, `ubuntu`, and one test suite defined, `default`—so our instance is called `default-ubuntu`.

Let’s create that instance now.

Listing 7.26: Creating the Test Kitchen instance

```

$ cd ~/dc/development
$ bundle exec kitchen create
-----> Starting Kitchen (v1.13.2)
      Terraform 0.11.8

-----> Creating <default-ubuntu>...
      Finished creating <default-ubuntu> (0m0.00s).
-----> Kitchen is finished. (0m0.94s)
  
```

We can then list the created instance using the `kitchen list` command.

Listing 7.27: Listing the Test Kitchen instance

```
$ bundle exec kitchen list
Terraform v0.11.8
Instance      Driver  Provisioner Verifier Transport Last
Action
default-ubuntu Terraform Terraform  Terraform Ssh      Created
```

The listing shows our instance, `default-ubuntu`, from the concatenation of the platform and test suite, its driver, provisioner, verifier, and transport. It also shows the last action taken on the instance. Here our instance is in the `Created` state because we've just run `kitchen create`.

Our next step is to converge our infrastructure. This ensures our infrastructure is running and up-to-date. Let's do that now.

Listing 7.28: Converging Test Kitchen

```
$ bundle exec kitchen converge
-----> Starting Kitchen (v1.13.2)
      Terraform v0.11.8

-----> Converging <default-ubuntu>...
      Get: git::https://github.com/turnbullpress/tf_api.git (
update)
      Get: git::https://github.com/turnbullpress/
tf_remote_state (update)
      Get: git::https://github.com/turnbullpress/tf_vpc.git?ref
=v0.0.2
      (update)
      Get: git::https://github.com/turnbullpress/tf_web.git (
update)
      Refreshing Terraform state in-memory prior to plan...
      The refreshed state will be used to calculate this plan,
but
      will not be persisted to local or remote state storage.

      module.vpc.aws_vpc.environment: Refreshing state... (ID:
vpc-9dcbd0fa)
      . . .
```

We've used the `bundle exec` command to run the `kitchen converge` command. This checks the current state of our infrastructure against the Terraform state. If it varies, it'll apply the plan until the infrastructure is up-to-date.

Let's look at our instance now.

Listing 7.29: Reviewing the Test Kitchen instance again

```
$ bundle exec kitchen list
Terraform v0.11.8

Instance      Driver  Provisioner Verifier Transport Last
Action
default-ubuntu Terraform Terraform  Terraform Ssh
Converged
```

Note the last action is now **Converged**. From here we can actually run our tests.



NOTE We're going to skip the login step. We don't need it in this case.

Running the controls

Now that our instance is ready and our infrastructure converged, we can run the controls using the `kitchen verify` command.

Listing 7.30: Running the controls

```
$ bundle exec kitchen verify
----> Starting Kitchen (v1.13.2)
      Terraform v0.11.8
----> Setting up <default-ubuntu>...
      Finished setting up <default-ubuntu> (0m0.00s).
----> Verifying <default-ubuntu>...
      Terraform v0.11.8

.
.
.

      Verifying host 'ec2-107-23-238-219.compute-1.amazonaws.com' of group 'bastion'

Command lsb_release -a
  stdout
    should match /Ubuntu 16.04/

Service cron
  should be enabled
  should be running

Service rsyslog
  should be enabled
  should be running

Finished in 0.65346 seconds (files took 4.72 seconds to load)
5 examples, 0 failures
```

The `kitchen verify` command executes our controls. We can see it ensuring our instance, `default-ubuntu`, is set up. It then uses the `bastion` group to determine the hosts on which to run the tests. It takes the list of hosts from the output we created, in our case our one bastion host, and runs each control and their tests.

We can see the output from each control and each test. We now know:

- That our host has the right operating system and version installed.

- That the `cron` and `rsyslog` services are enabled and running.

Finally, we see a summary of the verify run that shows that all five examples have passed.

Adding a new control

Let's now add another control to our `bastion` group. We're going to test the settings of our SSH daemon. We'll add our control to the `default` test suite (as we have no others). We add a file called `sshd_spec.rb` in the `test/integration/default/controls` directory. Let's populate that file now.

Listing 7.31: The `sshd_spec.rb` control file

```
control 'sshd' do
  title 'SSHd controls'
  desc "
    Checks that the host's SSH daemon is correctly configured.
"
  tag 'sshd', 'ubuntu'

  describe service('sshd') do
    it { should_be_enabled }
    it { should_be_running }
  end

  describe sshd_config do
    its('Protocol') { should eq '2' }
    its('Port') { should eq('2222') }
  end
end
```

We've added a new control called `sshd` with some useful metadata. Inside our control are two `describe` blocks. The first ensures our `sshd` daemon is enabled and running. The second block uses a new resource: `sshd_config`. The

`sshd_config` resources allows you to test the SSH daemon's configuration is correct. We're testing two matchers: the `Protocol` and `Port` settings inside the `/etc/ssh/sshd_config` configuration file. We want to ensure that `Protocol` is set to `2` and the SSH daemon is listening on port `2222`.

Let's run our new controls. We can use a new Test Kitchen command, `kitchen test`, to do this. The `kitchen test` command runs all the steps in our workflow: `create`, `converge`, and `verify`. We're also going to pass in a command line flag: `--destroy passing`. The `--destroy` flag potentially destroys the instance after our tests run. The `passing` option constrains it to only destroy the infrastructure if all the tests pass. An alternative is the `--destroy always` flag, which always destroys the instance.

Listing 7.32: Running kitchen test

```
$ bundle exec kitchen test --destroy passing
.
.
.
Service sshd
  should be enabled
  should be running

SSH Configuration
  Protocol
    should eq "2"
  Port
    should eq "2222" (FAILED - 1)

Failures:

1) SSH Configuration Port should eq "2222"
   Failure/Error: DEFAULT_FAILURE_NOTIFIER = lambda { |failure,
   _opts| raise failure }

     expected: "2222"
     got: "22"

     (compared using ==)
# ./test/integration/default/controls/sshd_spec.rb:9:in `block (3 levels) in load_with_context'

Finished in 0.84653 seconds (files took 18.67 seconds to load)
9 examples, 1 failure
```

Damn! Our test failed. The SSH daemon is set to port 22 still. Now we can go and fix that issue. As we specified the `--destroy passing` command flag, our instance still exists. Let's jump onto the host and fix the issue, then run our controls again.

Listing 7.33: Running the controls again

```
$ bundle exec kitchen test --destroy passing
. .
.
Service sshd
  should be enabled
  should be running

SSH Configuration
  Protocol
    should eq "2"
  Port
    should eq "2222"

Finished in 1.05 seconds (files took 20.03 seconds to load)
9 examples, 0 failures

      Finished verifying <default-ubuntu> (0m3.84s).
-----> Destroying <default-ubuntu>...
      Finished destroying <default-ubuntu> (0m0.00s).
      Finished testing <default-ubuntu> (0m20.78s).
-----> Kitchen is finished. (0m21.76s)
```

Excellent. All tests are passing and the instance is destroyed as a result. This doesn't mean our infrastructure is destroyed though. If you want to remove the infrastructure, you'll need to run `terraform destroy`.

Listing 7.34: Destroying the development infrastructure

```
$ terraform destroy
. . .
```

We could now extend this to cover testing beyond these examples. Other areas we could test are [network configuration](#) and [host status and connectivity](#), [files](#), [JSON](#)

or YAML configuration files, configuration, or other operating system properties. Or you can use custom resources you create yourself.

Building custom InSpec resources

InSpec also offers you the capability to create your own custom resources. Let's look at a simple example that tests the contents of the /etc/resolv.conf DNS resolver configuration file. Test Kitchen expects to find custom resources in the libraries directory under the test/integration/default directory (or the name of our suite of controls in which we use the resource). Let's create that directory now.

Listing 7.35: Making the libraries directory

```
$ cd ~/dc/development  
$ mkdir -p test/integration/default/libraries
```

With this directory created, let's now create a custom resource. Like Test Kitchen, custom resources are written in Ruby. Let's create and populate a file called resolv_conf.rb to hold our resolv.conf custom resource.

Listing 7.36: The resolv_conf custom resource

```

class ResolvConf < Inspec.resource(1)
  name 'resolv_conf'

  desc '
    Checks resolv.conf configuration.

  example "
    describe resolv_conf do
      its('nameserver') { should eq('10.0.0.2') }
    end
  "

  def initialize
    @path = "/etc/resolv.conf"
    @file = inspec.file(@path)

    begin
      @params = Hash[*@file.content.split("\n")
                    .reject{ |l| l =~ /^#/ }
                    .collect { |v| [ v.chomp.split ] } ]
      .flatten]
    rescue StandardError
      return skip_resource "#{@file}: #{$!}"
    end
  end

  def exists?
    @file.file?
  end

  def method_missing(name)
    @params[name.to_s]
  end
end

```

This code parses the `resolv.conf` file, ignores comments, and parses each line

assuming it is a rough key/value pair. For example:

Listing 7.37: The resolv.conf

```
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by
# resolvconf(8)
# DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE
OVERWRITTEN
nameserver 8.8.8.8
nameserver 8.8.4.4
```

We can then create a control that uses this custom resource. Let's create a file called `resolv_conf_spec.rb` in the `test/integration/default/controls` directory.

Listing 7.38: The resolv_conf_spec.rb file

```
control 'resolv.conf' do
  title 'Resolv.conf tests'
  desc '
    Tests the contents of the resolv.conf file.
  '
  tag 'resolv.conf', 'ubuntu'

  describe resolv_conf do
    its('nameserver') { should eq('10.0.0.2') }
  end
end
```

Here we're testing that the `resolv.conf` file has a name server of `10.0.0.2` configured. We can now run our new control.

Listing 7.39: Running the resolv.conf control

```
$ bundle exec kitchen test --destroy passing
-----> Starting Kitchen (v1.13.2)
      Terraform v0.11.8

.
.

resolv_conf
  nameserver
    should eq "10.0.0.2"

.

Finished in 0.91426 seconds (files took 19.6 seconds to load)
10 examples, 0 failures
```

We can see our new control has been executed and passed.

Again, this process only destroys the Test Kitchen instance. It does not destroy our infrastructure. If you want to destroy the stack we've used for the tests, we can use the `terraform destroy` command.

We've also created a Git repository for our data center environment and tests which you can find [on GitHub here](#).

 **TIP** There's a useful, [detailed Test Kitchen example](#) including fixtures and more complex configuration in the `kitchen-terraform` GitHub repository.

Alternative infrastructure testing frameworks

As it's the early days for Terraform, there aren't a lot of alternatives for testing. The current Test Kitchen solution requires direct SSH access to connect, which is an unfortunate limitation.

There is, however, an alternative using another framework called [Serverspec](#). [John Vincent](#) has provided a [Gist showing how you might integrate Serverspec with Terraform](#). This chapter will be updated as alternatives and new solutions evolve.

Another approach, specifically for AWS and using [awspec](#), has been documented by [Dean Wilson](#).

Summary

In this chapter we've learned how we can write tests for our infrastructure and its configuration. We saw how to install and integrate a testing harness, Test Kitchen, into our multi-environment architecture.

We also learned about the InSpec testing framework and how to write tests to determine if our infrastructure is correct. We saw some of the built-in resources in InSpec available to test infrastructure settings. We also saw how to create your own custom resources.

From here you should be able to build infrastructure with Terraform, manage the life cycle of that infrastructure, and write appropriate tests to validate the correctness of your infrastructure.

List of Figures

1 License	6
2 ISBN	6
2.1 Creating an AWS account	31
2.2 Provider definition	34
2.3 Resource definition	36
2.4 AWS EC2 instance	45
2.5 The base.tf graph	59
3.1 Our web application stack	64
3.2 Creating a GitHub repository	108
3.3 The graph of our web application stack	134
3.4 Our stack in action	135
4.1 Our web stack custom index.html	161
5.1 Querying remote state	194
5.2 Querying the web remote state	201
5.3 The Consul Web UI	211
5.4 The state/consul key	213
6.1 Our data center	224
6.2 Directory structure	226
6.3 AWS VPC Configuration	234
6.4 Our web service	257
6.5 Our API service directory tree	261

List of Figures

6.6 Our API service	266
7.1 The Test Kitchen workflow	289

Listings

1 Sample code block	5
2.1 Download the Terraform zip file	20
2.2 Unpack the Terraform binary	20
2.3 Checking the Terraform version on Linux	20
2.4 Creating a directory on Windows	21
2.5 Terraform Windows download	21
2.6 Setting the Windows path	21
2.7 Checking the Terraform version on Windows	22
2.8 Installing Terraform via Chocolatey	22
2.9 Installing Terraform via Homebrew	22
2.10 Checking the Terraform version on Mac OS X	23
2.11 Seeing the Terraform command options	24
2.12 Create a home for Terraform	25
2.13 Create our first Terraform configuration directory	25
2.14 Creating a Git repository	26
2.15 The base README.md file	27
2.16 Committing the README.md file	27
2.17 Common directory structure	29
2.18 Creating the base.tf file	29
2.19 Initial base.tf configuration	32
2.20 Adding the aws provider	33
2.21 Displaying the Terraform plan	38

2.22 Applying the base resources	40
2.23 Creating the base resources	41
2.24 Adding the state file and backup to .gitignore	42
2.25 The Terraform state file	43
2.26 Displaying the base resource	44
2.27 Adding a second resource	45
2.28 Planning a second resource	47
2.29 Applying our second resource	49
2.30 Creating a plan output	51
2.31 Applying an output plan	52
2.32 Targeting a resource	54
2.33 Showing the second resource	55
2.34 Creating the dependency graph	57
2.35 Piping the graph command	57
2.36 Install graphviz via brew	58
2.37 Installing graphviz on Ubuntu	58
2.38 Installing graphviz with Chocolatey	58
2.39 Creating a viewable graph	59
2.40 Destroying our resources	60
2.41 Destroying a single resource	61
2.42 Planning a Terraform destroy	61
3.1 Our original configuration	65
3.2 Creating the variables.tf file	66
3.3 Our first variables	67
3.4 Variable type specified	67
3.5 Variable type specified	68
3.6 Variable descriptions	68
3.7 Adding our new variables	69
3.8 A map variable	70
3.9 Using map variables in base.tf	71
3.10 Constructing a list	72

3.11 Using a list	73
3.12 Retrieving a list element	73
3.13 An empty variable	74
3.14 Empty and default variables	74
3.15 Command line variables	75
3.16 Setting a map with var	76
3.17 Populating a list via command line flag	76
3.18 Creating a variable assignment file	76
3.19 Adding variable assignments	77
3.20 Variable doesn't exist error	77
3.21 Running Terraform with a custom variable file	78
3.22 Variable defaults	79
3.23 Creating the web directory	80
3.24 Adding the state file and backup to .gitignore	80
3.25 Creating the stack files	81
3.26 Our variables.tf file	81
3.27 The web terraform.tfvars file	82
3.28 Installing AWS CLI on Linux	83
3.29 Installing AWS CLI on OSX	83
3.30 Installing awscli via choco	83
3.31 Running aws configure	83
3.32 The aws/credentials file	84
3.33 The aws/config file	84
3.34 Our web.tf file	86
3.35 Multiple providers	87
3.36 The vpc module	89
3.37 Multiple vpc modules	90
3.38 Creating the vpc module directory	90
3.39 The vpc module with a remote source	91
3.40 Referencing a module version	91
3.41 Referencing a registry module	92

3.42 Referencing a registry module's version	93
3.43 Creating the vpc module variables	94
3.44 The vpc module's variables	94
3.45 The vpc module's default variables	95
3.46 Overriding vpc module's default variables	96
3.47 The vpc module resources	97
3.48 Multiple aliased providers	98
3.49 The vpc module's default variables	99
3.50 The aws_vpc resource	100
3.51 The aws_subnet resource	100
3.52 The vpc module outputs	102
3.53 The vpc module outputs	103
3.54 The vpc module block revisited	104
3.55 Our web aws_elb resource	105
3.56 The Terraform get command	106
3.57 Updating a module	107
3.58 The README.md file	109
3.59 Creating a .gitignore file	109
3.60 Committing and pushing our vpc module	110
3.61 Updating our vpc module configuration	110
3.62 Getting the new vpc module	110
3.63 The aws_instances count	112
3.64 The aws_elb resource	113
3.65 Using count indexes in variables.tf	114
3.66 Looking up the count index	114
3.67 Using the length function	115
3.68 Naming using the count.index	116
3.69 Interpolated math	116
3.70 AWS owner tags in variables.tf	117
3.71 Splitting up the count instances	117
3.72 Count exhausted failure	118

3.73 Wrapping the count instances list	118
3.74 A ternary operation	119
3.75 Using ternary with count	120
3.76 A conditional attribute	121
3.77 A local definition	122
3.78 Using a local in a resource	122
3.79 Adding user data to our instances	124
3.80 Creating the files directory	124
3.81 The web_bootstrap.sh script	125
3.82 The web outputs.tf file	126
3.83 Committing our configuration	127
3.84 Initialiazing the web configuration	128
3.85 Planning our web configuration	129
3.86 Applying our web stack	131
3.87 Showing the outputs only	132
3.88 Outputs as JSON	133
3.89 Graphing the web stack	134
3.90 Destroy our web stack	136
4.1 Adding a provisioner to an AWS instance	141
4.2 Connection block attributes	142
4.3 The key_path variable	142
4.4 Adding the file provisioner to an AWS instance	143
4.5 A template provider data source	144
4.6 The hostname variable	146
4.7 The files/index.html.tpl template	146
4.8 The populated index.html	147
4.9 Using the template in a provisioner	147
4.10 A single template	149
4.11 The populated single index.html	150
4.12 Uploading file content	150
4.13 Uploading a file	151

4.14 Uploading a directory	152
4.15 The second provisioner	153
4.16 The bootstrap_puppet script	154
4.17 The second provisioner	155
4.18 Adding a remote execution script with arguments	156
4.19 The final provisioner block	157
4.20 Destroy our web stack	158
4.21 Running our first provisioning	159
4.22 The final populated index.html	160
4.23 A failed provisioning run	162
5.1 Example of remote state environments	166
5.2 Creating an S3 remote backend directory	169
5.3 Creating a .gitignore file	169
5.4 The remote_state interface.tf	170
5.5 The remote_state main.tf file	171
5.6 The remote_state outputs	172
5.7 Initializing the remote state module	173
5.8 The remote_state plan	174
5.9 Applying our remote_state configuration	175
5.10 Committing our new remote_state module	176
5.11 Changing into the web configuration	176
5.12 Adding the var.prefix and var.environment variables	177
5.13 Adding the remote state module	177
5.14 Getting the remote_state module	178
5.15 Planning our web remote_state bucket	179
5.16 Creating the web remote_state bucket	180
5.17 Adding the remote state module	181
5.18 The terraform remote state block	181
5.19 Running terraform plan pre-initialization	183
5.20 Running the terraform init command	184
5.21 The new terraform.tfstate file	186

5.22 Warning on remote state removal	187
5.23 Disabling remote state storage	187
5.24 Pulling the remote state	188
5.25 First run of terraform plan	189
5.26 Running terraform init for a new stack	190
5.27 Running terraform plan with existing state	190
5.28 A skeleton remote state backend	191
5.29 Creating the s3_backend file	192
5.30 The s3_backend file	192
5.31 The backend-config flag	192
5.32 Adding the environment variable to the base variables.tf	195
5.33 Adding the remote_state module to base.tf	195
5.34 Getting the remote_state module for base	195
5.35 Applying the remote_state module for base	196
5.36 Moving the base state remote	197
5.37 The terraform_remote_state data source	198
5.38 Using terraform data source	199
5.39 Showing our remote state data source	200
5.40 Using our remote state	202
5.41 Planning using our data source	203
5.42 Making a Consul configuration	205
5.43 The consul module's variables.tf file	206
5.44 The new Consul variable values in terraform.tfvars	207
5.45 Generating an encryption key via openssl	207
5.46 Generating an encryption key via SHA	207
5.47 Creating a UUID token	208
5.48 The new Consul outputs	208
5.49 The consul.tf file	209
5.50 Getting the Consul configuration's modules	210
5.51 Applying our Consul configuration	211
5.52 Moving the Consul state to S3	212

5.53 Storing remote state in Consul	212
5.54 Adding the Consul remote state	214
5.55 Updating the web configuration	215
5.56 Showing the Consul state	215
5.57 Adding the consul provider	216
5.58 Adding the token variable	217
5.59 Adding the Consul token value	217
5.60 The <code>consul_key_prefix</code> resource	218
5.61 Applying our new web configuration	219
5.62 Curling our Consul key	220
5.63 Decoding the value	220
5.64 Decoding the value in one step	221
6.1 Creating the base directory	225
6.2 Adding the state file and backup to <code>.gitignore</code>	226
6.3 Running the <code>terraform plan</code> command with <code>-out</code>	228
6.4 Applying the generated plan	228
6.5 The development variables.tf	231
6.6 The development <code>terraform.tfvars</code> file	232
6.7 The development environment <code>main.tf</code> file	233
6.8 The <code>vpc</code> module outputs	235
6.9 Getting the development <code>remote_state</code> and <code>vpc</code> modules	236
6.10 The development outputs in <code>main.tf</code>	237
6.11 Planning our development environment	238
6.12 Applying our development VPC	239
6.13 Configuring development environment <code>remote state</code>	240
6.14 The development <code>web.tf</code> file	241
6.15 The <code>Cloudflare</code> variable in <code>terraform.tfvars</code>	242
6.16 The <code>vpc_id</code> output	243
6.17 Building the <code>web</code> module	244
6.18 The <code>web</code> module's variables	245
6.19 The <code>web</code> module outputs	246

6.20 The web module resources	247
6.21 A filtered data source	248
6.22 The VPC Name tag	249
6.23 The data.aws_vpc.environment data source	249
6.24 The CIDR block data source variable	250
6.25 The web instance	251
6.26 The public and private subnets	252
6.27 The public_subnet_ids output	252
6.28 The Cloudflare record resource	253
6.29 Committing the web module	254
6.30 Getting the web module	254
6.31 Planning for our web module	255
6.32 Applying the web module	256
6.33 The Cloudflare record	257
6.34 SSH'ing into the bastion host	258
6.35 Ping a web host	258
6.36 Removing the web service	259
6.37 The API service in the api.tf file	260
6.38 Checking out the API module	261
6.39 The API aws_instance resource	262
6.40 The API module outputs	263
6.41 Getting the API module	263
6.42 The API service plan	264
6.43 The API service	265
6.44 Testing the API service	266
6.45 Adding a production environment	267
6.46 The production terraform.tfvars	267
6.47 Adding the web.tf file	268
6.48 Running the terraform workspace list command	269
7.1 Testing the Ruby version	274
7.2 Installing rbenv	274

7.3 Adding rbenv to the path	275
7.4 Auto-activating rbenv	275
7.5 Adding the rbenv activating	275
7.6 Confirming rbenv worked	276
7.7 Installing Ruby with rbenv	276
7.8 Checking Ruby 2.3.1 is installed	276
7.9 Enabling Ruby 2.3.1	277
7.10 Adding the .ruby-version file	277
7.11 Install Bundler	277
7.12 Our Test Kitchen Gemfile	278
7.13 Running Bundler for Test Kitchen	278
7.14 The development/.kitchen.yml file	279
7.15 The doc test format	281
7.16 The verifier block	282
7.17 The bastion_dns_host	282
7.18 Creating the Test Kitchen directory structure	283
7.19 Creating the inspec.yml file	283
7.20 Creating the inspec.yml file	283
7.21 Testing the suite configuration	284
7.22 The operating_system control	285
7.23 The operating_system control	285
7.24 The operating_system control	287
7.25 Adding a second test	288
7.26 Creating the Test Kitchen instance	289
7.27 Listing the Test Kitchen instance	290
7.28 Converging Test Kitchen	291
7.29 Reviewing the Test Kitchen instance again	292
7.30 Running the controls	293
7.31 The sshd_spec.rb control file	294
7.32 Running kitchen test	296
7.33 Running the controls again	297

7.34 Destroying the development infrastructure	297
7.35 Making the libraries directory	298
7.36 The resolv_conf custom resource	299
7.37 The resolv.conf	300
7.38 The resolv_conf_spec.rb file	300
7.39 Running the resolv.conf control	301

Index

- .gitignore, 42, 80, 109, 185, 226
- .kitchen.yml, 279
- .terraform directory, 106, 185, 236
- .tf, 29
- .tf.json, 29
- AMI, 36
- Ansible, 13, 23, 123, 133, 138
- AWS, 30, 31, 35, 67, 233
 - Access Key ID, 31
 - Profiles, 233
 - Secret Access Key, 31
 - Shared credentials, 35, 233
- aws
 - configure, 83
- AWS IAM, *see* Identity and Access Management
- awspec, 302
- Chef, 13, 23, 123, 133, 138
- Chocolatey, 22
- CircleCI, 229
- cloud-init, 145
- Compliance, 271
- Conditionals, 119
- Configuration Management, 138
- Consul, 164, 204
 - encryption key, 207
 - installing, 205
 - remote state, 212
 - token, 207
- consul
 - keygen, 207
- DAG, 12, 28
- Puppet, 28
- Data sources, 144, 248
 - aws_ami, 144
 - filtering, 248
 - terraform_remote_state, 197, 214
- Declarative, 56
- depends_on, 56, 106
- Docker, 23
- File formats
 - .tf, 29
 - .tf.json, 29
- Function, 71
 - element, 73, 118, 147
 - file, 124, 252

format, 116
lookup, 71
Functions
length, 115
Git, 25, 80, 127, 185
GitHub, 107
Graph, 12, 28
Graphviz, 57
HashiCorp, 12
HCL, 30
Homebrew, 22
IAM, *see* Identity and Access Management
Identity and Access Management, 31
Import infrastructure, 135
InSpec, 272, 273, 285
 control, 285
 Custom resources, 298
 custom resources, 286
 inspec.yml, 283
 matchers, 286
 metadata, 287
 resources, 286
 Ruby, 288
 service, 288
Inspec
 control, 285
 describe, 285
inspec
 check, 284
 inspec.yml, 283
Installation, 19
 Linux, 20
 Mac OS X, 22
 Microsoft Windows, 21, 22
 Windows, 21
Interpolation, 46
Jenkins, 229
JSON, 29, 30
kitchen
 converge, 289
 create, 289
 list, 289
 test, 296
 verify, 292
kitchen-terraform, 273
 driver, 273
 provisioner, 273
 transport, 273
 verifier, 273
Local values, 121
Locals, 121
Meta-parameter
 count, 111, 262
 depends_on, 56, 88, 106
 lifecycle, 172
module
 providers, 98

source
 registry modules, 92
 versioned Terraform Registry
 modules, 92

Modules, 88
 documentation, 106
 getting, 106
 Git, 91
 outputs, 101
 registry, 92
 resources, 96
 source, 89
 sources, 91, 111
 structure, 93
 updates, 106
 usage, 104
 versions, 111

multiple providers, 88

Overrides, 28, 79

Packer, 140, 154

Path, 252

Provider, 33, 134
 AWS, 67, 134
 aws, 33, 87
 cloudflare, 242
 consul, 216

providers
 alias, 88

Provisioners
 connection, 141

file, 143, 150
local-exec, 155
null_resource, 162
provisioner, 141
remote-exec, 153

Puppet, 13, 23, 123, 133, 138, 163
 DAG, 28

rbenv, 274

Remote state, 166

REPL, 69

resource providers, 88

RSpec, 272, 285

RVM, 274

Serverspec, 272, 302

State, 41, 165, 185

State environments, 268

State file, 42

Supported platforms, 19

TDD, 271

Templates, 144

Ternary operation, 119

Terraform
 +, 39
 -, 39
 -/, 39
 backup state, 165
 count, 111, 262
 data source, 144, 248
 Declarative, 56

dependencies, 46, 56, 105
environment variables, 78
failed execution, 54
functions, 71
graph, 12, 28
import, 45
Import infrastructure, 135
installation
 Linux, 20
 OSX, 22
 Windows, 21
Interpolation, 46
isolation, 80, 197
lifecycle, 172
lists, 72
locking, 186
maps, 70
module, 88, 91
multiple providers, 242
null_resource, 162
outputs, 101, 125
outputting plans, 50
override, 28, 79
parallelization, 55
path, 91, 252
plan output, 53, 229, 230
provider
 template, 144
Provisioning, 138
Remote state, 165, 166, 250
remote state
 initialization, 182
 legacy, 168
 locking, 186
 REPL, 69
 self variables, 151
 State, 165
 state, 41, 42, 165, 185
 list, 167
 show, 167
 state environments, 268
 taint, 54, 62, 162
 testing, 271
 untaint, 54, 62
 variable
 math, 116
 variable defaults, 74
 variables, 66, 67
terraform
 -help, 25
 -var-file, 77
 apply, 39, 130, 238
 -auto-approve, 39, 53
 command help, 25
 console, 69
 destroy, 60, 136, 297, 301
 -target, 61
 fmt, 47
 get, 106
 -update, 106
 graph, 56
 init, 33, 127, 182, 192

-backend-config, 192
output, 132
-json, 132
plan, 37, 237
-out, 51, 228
refresh, 167
remote
 config, 181, 185
 pull, 188
 push, 188
show, 44, 54, 132, 199
validate, 47
var, 75
version, 20
workspace, 269
 list, 269
terraform apply interactivity, 52, 53
Terraform Enterprise, 165
Terraform file formats, 29
Terraform Module Registry, 89
Terraform Registry, 92
terraform-docs, 106
terraform.tfstate, 41, 185
terraform.tfvars, 76
Test Kitchen, 272
 .kitchen, 279
 .kitchen.yml, 279
 converge, 291
 instance, 290
 verify, 292
Test-Driven Development, 271

Thanks! I hope you enjoyed the book.

© Copyright 2016 - James Turnbull <james@lovedthanlost.net>

