JAMES TURNBULL

THE PACKER BOOK

The Packer Book

James Turnbull

April 20, 2018

Version: v1.1.2 (067741e)

Website: The Packer Book

ISBN 9780988820272

Contents

Foreword

Who is this book for?

This book is a hands-on introduction to Packer, the HashiCorp image builder. 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 Packer. If you're a Packer 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.

Finally, Packer is evolving quickly and has only just reached a 1.0.0 release state. That means "here be dragons," and you should take care when using Packer in production.

This book assumes you are using Packer 1.2.2 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.

- Sid Orlando, who not only edited this book but continues to make me laugh at inappropriate moments.
- Gareth Rushgrove and Anna Kennedy for their Packer Serverspec examples.

Packer™ is a registered trademark of HashiCorp, Inc.

Technical reviewers

Thanks to the folks who helped make this book more accurate and useful.

Rickard von Essen

Rickard von Essen works as a Continuous Delivery and Cloud Consultant at Diabol. He helps companies deliver faster, improve continuously, and worry less. In his spare time, he helps maintain Packer and contributes to numerous other FOSS projects. He has been tinkering with Linux and BSD since the late 1990s, and has been hacking since the Amiga era. When not fiddling with computers, he spends his time with his wife and two children in Stockholm, Sweden. Rickard has a Master of Computer Science and Engineering from Linköping University.

Ben Allen

Ben Allen is a SecOps engineer who works with BlueTarp Financial to build secure cloud environments across platforms. Throughout his career he's been an automation evangelist who helps drive organizations to codify processes and build durable, automated assets. Ben's background as a DevOps engineer and a sysadmin has been spent in the healthcare and pharmaceutical industries. He is a certified SaltStack engineer, AWS Certified Solutions Architect, and holds several Microsoft certifications. Ben earned his Bachelor's degree in network engineering

and serves the advisory boards of higher education institutions that help modernize technology curriculums.

Daniel Linder

Daniel Linder is a Systems Architect at West Corporation. He is responsible for the life cycle of all Windows, RedHat, Solaris, and AIX platforms at West. He is also responsible for leading West's transition to private cloud, including their Packer implementation. Daniel is a RedHat Certified System Administrator and has a BS in Computer Science from the University of Nebraska.

Editor

Sid Orlando is a writer and editor (among some other things) who may or may not have recurring dreams about organizing her closet with dreamscape Docker containers. Find her on Twitter: @ohreallysid.

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 Empatico 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/packerbook-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

Figure 1: License

Figure 2: ISBN

Version

This is version v1.1.2 (067741e) of The Packer Book.

Chapter 1

Introduction to Packer

A machine image is a static unit that contains a preconfigured operating system and installed software. You can use images to clone or create new hosts. Images help speed up the process of building and deploying new infrastructure. Images come in many formats, specific to various platforms and deployment environments.

In one of my first jobs I built servers. It wasn't done using an API, the latest cool Cloud service, or even via a configuration management tool. Instead we wheeled a trolley, stacked with four PCs, a CD burner, and an assortment of cables and CDs, around the data center. We plugged directly into freshly racked servers and loaded previously built images, usually called "golden images," onto the new servers manually and slowly.

For many years, this approach remained the state of the art for building servers. The process was slow and painful. It also created hard-to-manage servers. Every image was a snapshot, a point-in-time collection of packages and configuration. Each server was generally fixed at that point in time. This meant your images quickly became out of date and potentially vulnerable to security issues in older packages. Mass updates—for example, package updates—were possible but tricky and, because the "golden image" model predated configuration management tools

like Puppet, mass configuration changes were complex and error prone.

Additionally it was rare that a single image was sufficient for every server. Some servers had shared base configuration but they were rarely entirely fit for purpose. This problem was again because this model predated configuration management. In these ancient times, additional configuration of hosts was rarely more sophisticated than scripts run at startup.

To partially address this issue, images began to multiply. Version 1 of the golden image became Version 2. Version 2 spawned into Version 2 Variant 1, customized for the New Jersey data center, and Version 2 Variant 2, for the data center in Palo Alto. Other variants emerged, perhaps customized for a specific server role. This would be exacerbated by image requirements for different platforms or operating system versions or hardware variants. The result is what can be best described as "image sprawl": your environment ending up containing thousands of images.

Not only would this mean every server was subtly different, but a server's ancestry was often a convoluted mess of inherited images. This combined with the existing entropy and change in an environment resulted in tens, hundreds, or even thousands of servers that were allegedly built the same way but were, in reality, unique.

TIP My friend Luke Kanies, also the founder of Puppet, wrote one of the best pieces on image sprawl entitled Golden Image or Foil Ball?.

This model only started to change as configuration management tools began to evolve. Tools like Puppet now allowed you to build fleets of servers and customize or maintain them via configuration management manifests. Images often remained—to deal with base hardware differences or to provide a starting step or bootstrap for a server—but much of the work they had performed was trimmed away.

The pressure off, image management became a less important concern. The focus on post-install configuration meant that the state of the art for image production and management went largely unchanged for a number of years.

This status quo remained unchanged until new trends in server and compute resource usage started to emerge. This was marked by increased use of virtual machines, both locally in data centers and in cloud services like Amazon Web Services. All of these new technologies use images. Some images underpin base virtual machines like the Open Virtualization Format (OVF); others, like the Amazon Web Services' Amazon Machine Image (AMI) format, underpin Amazon's EC2 compute instances.

In a further evolution, engineer Mitchell Hashimoto released a tool called Vagrant. Vagrant was designed to help manage local development environments and relied on shared images called "boxes" from which virtual machines are cloned and launched. Vagrant also supports provisioners. Provisioners launch post-build activities, such as configuration management tools.

Lastly, container technologies, especially Docker and Kubernetes, have the concept of images as one of their base building blocks. Indeed, container technologies regularly bake applications and application code into images, not just infrastructure.

All of these changes resulted in an increased use of images. They also exposed the fact that many of the same issues—entropy of images, the complexity of image maintenance and management, and keeping images up to date—continued to exist in some form, even when images were used in conjunction with configuration management tools building and customizing on top of base images.

In response to this increased use of images, a number of new image management and pipeline approaches and tools have emerged. One of the first was Veewee. Veewee was written by Patrick Debois and designed to help build Vagrant boxes and virtual machine images. Veewee relies on the concept of building images based on customizable template files. It combines these templates with additional

scripts to build and configure images. For many, building base images and launching them with Vagrant, then using Vagrant's provisioning integration to run a configuration management tool to complete the build process, was an attractive approach.

Building on this concept, Mitchell Hashimoto and the team at Hashicorp released a new tool called Packer.

Introducing Packer

In this book we're going to teach you how to use Packer. Packer is a free and open-source image-building tool, written in Go and licensed under the Mozilla Public License 2.0. It allows you to create identical machine images, potentially for multiple target platforms, from a single configuration source. Packer supports Linux, Microsoft Windows, and Mac OS X, as well as other operating systems, and has support for a wide variety of image formats, deployment targets, and integrations with other tools.

Packer is fast, relatively quick to learn, and easy to automate. When used in combination with configuration management tools, it can create complex and fully functional images with software preinstalled and preconfigured.

Why use Packer?

Building images is tedious. It's also often manual and error prone. Packer can automate the creation of images and integrate well with your existing infrastructure tools. Packer allows you to create pipelines for building and deploying images, which in turn allows you to produce consistent, repeatable images.

Packer is also portable. As it has a central-configuration construct—an image template—it allows you to standardize images across multiple target platforms. You can build images across cloud platforms—like Amazon and Google—that are

consistent with images built on internal platforms like VMware or OpenStack, container environments like Docker and Kubernetes, and even individual development environments located on developer laptops.

You can remove pipelines constructed of multiple tools, some for Linux, some for Windows, some for commercial platforms, and replace them with a single, centrally configured tool. Each image produces identical—or near identical hosts—which reduces the risk of differences in your environment creating regressions or security vulnerabilities, or introducing bugs.

Packer allows you to bake an appropriate and testable portion of your configuration into images without the overhead and complexity of previous image-building solutions. This can significantly reduce the time between launch and operation, as well as minimize the potential for bugs and errors in your builds.

Packer use cases

So what can you use Packer for? A bunch of tasks—but the two biggest wins are for continuous delivery and consistency.

Continuous delivery

Packer integrates well with existing infrastructure tools. It can be slotted into a deployment pipeline and used to ensure the images you build on, test, and deploy are correct and up to date. At Empatico, we use a Continuous Integration/Continuous Deployment (CI/CD) approach to deploy new Amazon EC2 images. Packer builds Amazon Machine Images (AMIs) using Puppet, Terraform uses those AMIs when hosts and services are built, and Puppet runs again (and periodically) to provide final configuration and to keep our hosts correctly configured.

This means that if we need a new host or have to replace a malfunctioning host, the process is fast and consistent. Our infrastructure then becomes disposable, replaceable, and repeatable.

TIP If you're interested in HashiCorp's other useful infrastructure building tool, Terraform, there is also a book available covering it.

Environmental consistency

Do you have a large, complex infrastructure, with numerous environments covering development, testing, staging, and production? Packer is ideal for standardizing images across those environments. As it supports numerous target platforms, you can build standard images for all sorts of fabrics. You can also ensure that consistent configuration for things like patching, time, networking, security, and compliance are maintained across environments. For example, an infrastructure or a security team can use Packer to build images that are then shared with other groups to provide baseline builds that force cross-organizational standards.

Some other image management tools

- Veewee — An image builder for Vagrant; also usable for other image formats.
- Imagefactory - Red Hat imaging tool.
- BoxGrinder - A largely abandoned image builder for a variety of formats and Clouds.

Packer links

- Packer homepage.
- Packer documentation.

- Packer community resources.
- Packer source code.

Chapter 2

Installing Packer

Welcome to the world of image building. In this chapter, we'll take you through the process of installing Packer 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 Packer on:

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

The lessons here for installing Packer can be extended to the other supported platforms.

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

Now, we'll install Packer. In the next chapter, we'll start to learn how to use it to

create, manage, and destroy images. We'll also delve into Packer's configuration language. Finally, we'll build some images using Packer to see it in action.

Installing Packer

Packer is written in Go and shipped as a single binary file. The Packer site contains zip files with the binaries for specific platforms. Currently Packer is supported on:

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

You can also find SHA256 checksums for Packer 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 Packer binary.

Older versions of Packer are available from the Hashicorp releases service.

NOTE At the time of writing, Packer was at version 1.2.2.

Installing Packer on Linux and macOS

To install Packer on a 64-bit Linux or macOS host, you can download the zipped binary file. Use the `wget` or `curl` binaries to get the file from the download site.

Listing 2.1: Downloading the Packer zip file

```
$ cd /tmp
$ wget https://releases.hashicorp.com/packer/1.2.2/Packer_
1.2.2_linux_amd64.zip
```

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

Listing 2.2: Unpacking the Packer binary

```
$ unzip packer_1.2.2_linux_amd64.zip
$ sudo mv packer /usr/local/bin/
$ sudo chown -R root:root /usr/local/bin/packer
```

You can now test if Packer is installed and in your path by checking its version.

Listing 2.3: Checking the Packer version on Linux

```
$ packer version
Packer v1.2.2
```

TIP On some Red Hat and Fedora-based Linux distributions there is another tool named `packer` installed by default. You can check for this using `which -a packer`. If you find this tool already present you should rename the new `packer` binary to an alternate name, such as `packer.io`.

Alternative Mac OS X installation

In addition to being available as a binary for Mac OS X, for which you can use the same installation methodology as for Linux, Packer is also available from Homebrew. If you use Homebrew to provision your Mac OS X hosts then Packer can be installed via the `brew` command.

Listing 2.4: Installing Packer via Homebrew

```
$ brew install packer
```

NOTE Note the Homebrew installation builds from source and does not use the verified Hashicorp binary release.

Homebrew will install the `packer` binary into the `/usr/local/bin` directory. You can test that it's operating via the `packer version` command.

Listing 2.5: Checking the Packer version on Mac OS X

```
$ packer version
Packer v1.2.2
```

Installing Packer on Microsoft Windows

To install Packer on Microsoft Windows, you'll need to download the `packer` executable and put it in a directory. Let's create a directory for the executable using Powershell.

Listing 2.6: Creating a directory on Windows

```
C:\> MKDIR packer
C:\> CD packer
```

Now download the `packer` executable from the download site into the `C:\Packer` directory.

Listing 2.7: Packer Windows download

```
https://releases.hashicorp.com/packer/1.2.2/packer_
1.2.2_windows_amd64.zip
```

Unzip the executable using a tool like 7-Zip into the `C:\packer` directory. Finally, you'll need to add the `C:\packer` directory to the path. This allows Windows to find the executable. To do this you can run this command inside Powershell.

Listing 2.8: Setting the Windows path

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

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

Listing 2.9: Checking the Packer version on Windows

```
C:\> packer version
Packer v1.2.2
```

Alternative Microsoft Windows installation

You can also use a package manager to install Packer on Windows. The Chocolatey package manager has a Packer package available. You can use these instructions to install Chocolatey, then use the `choco` binary to install Packer.

Listing 2.10: Installing Packer via Chocolatey

```
C:\> choco install packer
```

Installing from source

You can also install the latest cutting edition of Packer by installing it from source. You'll need Go installed to do this.

If you don't have Go installed, you'll need to download it and follow the installation instructions.

NOTE Packer requires Go 1.6 or later.

To compile Packer you'll need the `GOPATH` environment variable set and added to your `PATH`, for example:

Listing 2.11: Setting GOPATH

```
$ export GOPATH=$HOME/go
$ export PATH=$PATH:$GOPATH/bin
```

Next, you'll need the Packer source and dependencies. You can get it via the `go get` command.

Listing 2.12: Getting the Packer source

```
$ go get github.com/hashicorp/packer
```

This will place the Packer source code at:

`$GOPATH/src/github.com/hashicorp/packer`

To build a Packer binary you can make use of the `make` command. The source code is also set up if you wish to develop with Packer. To build a binary run:

Listing 2.13: Building the Packer binary

```
$ cd $GOPATH/src/github.com/hashicorp/packer
$ make
go generate .
2017/06/28 16:54:36 Generated command/plugin.go
gofmt -w command/plugin.go
. . .
```

NOTE If you don't have `make` installed you can run `go build -o bin/packer .` to build the binary.

The `packer` binary will be installed into the `bin` directory. You can move it to wherever makes most sense—for example, `/usr/local/bin/`.

Installing via configuration management

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

- A Puppet module for Packer.
- A Chef cookbook for Packer.
- An Ansible role for Packer.
- A Packer Docker container.
- A Packer SaltStack formula.

Summary

In this chapter we've installed Packer and tested that it is operational. In the next chapter we'll see how to create our first image.

Chapter 3

First steps with Packer

Packer's purpose is building images—so we're going to start by building a basic image. Packer calls the process of creating an image a *build*. *Artifacts* are the output from builds. One of the more useful aspects of Packer is that you can run multiple builds to produce multiple artifacts.

A *build* is fed from a *template*. The template is a JSON document that describes the image we want to build—both where we want to build it and what the build needs to contain. Using JSON as a format strikes a good balance between readable and machine-generated.

NOTE New to JSON? Here's a good article to get you started. You can also find an online linter to help validate JSON syntax.

To determine what sort of image to build, Packer uses components called *builders*. A builder produces an image of a specific format—for example, an AMI builder or a Docker image builder. Packer ships with a number of builders, but as we'll discover in Chapter 7, you can also add your own builder in the form of plugins.

We're going to use Packer to build an Amazon Machine Image or AMI. AMIs underpin Amazon Web Services virtual machine instances run from the Elastic Compute Cloud or EC2. You'd generally be building your own AMIs to launch instances customized for your environment or specific purpose. We've chosen to focus on Cloud-based images in this book because they are easy to describe and work with while learning. Packer, though, is also able provision virtual machines.

As we're using AMIs here, before we can build our image, you'll need an Amazon Web Services account. Let's set that up now.

Setting up Amazon Web Services

We're going to use Amazon Web Services to build our initial image. Amazon Web Services have a series of free plans (Amazon calls them tiers) that you can use to test Packer at no charge. We'll use those free 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.

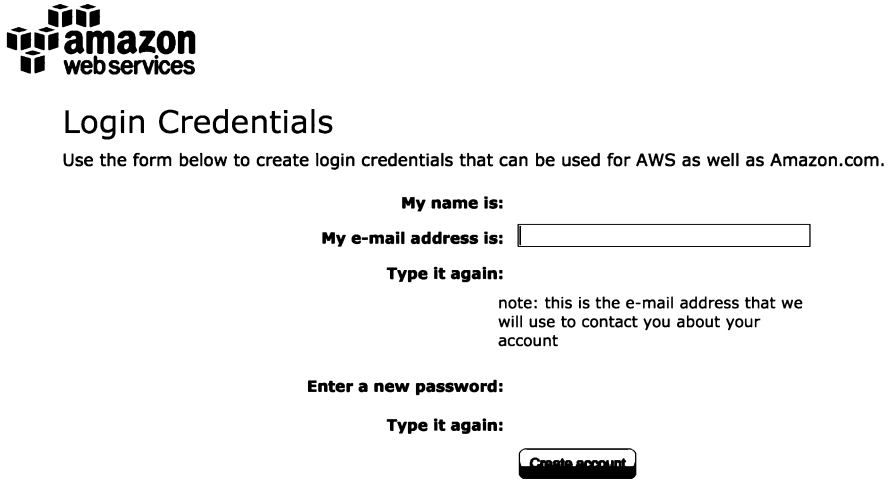

Figure 3.1: Creating an AWS account

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.

We're going to use a `t2.micro` instance to create our first image. If your account is eligible for the free tier (and generally it will be) then this won't cost you anything.

WARNING There is a chance some of the examples in this book might cost you some money. Please be aware of your billing status and the state of your infrastructure.

Running Packer

Packer is contained in a single binary: `packer`. We installed that binary in the last chapter. Let's run it now and see what options are available to us.

Listing 3.1: The packer binary

```
$ packer
Usage: packer [--version] [--help] <command> [<args>]

Available commands are:
    build       build image(s) from template
    fix         fixes templates from old versions of packer
    inspect     see components of a template
    push        push a template and supporting files to a Packer
 build service
    validate    check that a template is valid
    version     Prints the Packer version
```

The `packer` binary builds images using the `build` sub-command and inputting a JSON file called a template.

Let's create a directory hold our Packer templates.

Listing 3.2: Creating a template directory

```
$ mkdir packer_templates
```

Now let's create an initial template to generate our initial AMI.

Creating an initial template

Let's create an empty template file to start.

Listing 3.3: Creating an empty template file

```
$ touch initial_ami.json
```

Now let's open an editor and populate our first template file. The template file defines the details of the image we want to create and some of the how of that creation. Let's see that now.

Listing 3.4: Our initial_ami.json file

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": ""
  },
  "builders": [{
    "type": "amazon-ebs",
    "access_key": "{{user `aws_access_key`}}",
    "secret_key": "{{user `aws_secret_key`}}",
    "region": "us-east-1",
    "source_ami": "ami-a025aeb6",
    "instance_type": "t2.micro",
    "ssh_username": "ubuntu",
    "ami_name": "packer-example {{timestamp | clean_ami_name}}"
  }]
}
```

The template is a nested JSON hash with a series of blocks defined—in this case `variables` and `builders`. Let's take a look at both blocks and their contents.

Variables

The `variables` block contains any user-provided variables that Packer will use as part of the build. User variables are useful in these three ways:

- As shortcuts to values that you wish to use multiple times.
- As variables with a default value that can be overridden at build time.
- For keeping secrets or other values out of your templates.

User variables must be defined in the `variables` block. If you have no variables then you simply do not specify that block. Variables are key/value pairs in string form; more complex data structures are not present. These variables are translated into strings, numbers, booleans, and arrays when parsed into Packer. Packer assumes a list of strings separated by commas should be interpreted as an array.

NOTE Packer also has another type of variable, called template variables, that we'll see later in the book.

User variables can either have a specified value or be defined *null*—for example "". If a variable is null then, for a template to be valid and executed, its value must be provided in some way when Packer runs.

In addition to defined values, variables also support environment variables.

Environment variables

A common configuration approach is to use environment variables to store configuration information. Inside the `variables` block we can retrieve the value of

an environment variable and use it as the value of a Packer variable. We do this using a function called `env`.

Functions allow operations on strings and values inside Packer templates. In this case the `env` function only works in the `variables` block when setting the default value of a variable. We can use it like so:

TIP You can find a full list of the available functions in the Packer engine documentation.

Listing 3.5: Environment variables

```
{
  "variables": {
    "aws_access_key": "{{env `AWS_ACCESS_KEY`}}",
    "aws_secret_key": "{{env `AWS_SECRET_KEY`}}"
  },
```

Note that the function and the environmental variable to be retrieved are enclosed in double braces: `{{ }}`. The specific environment variable to be retrieved is specified inside back ticks.

NOTE You can only use environment variables inside the `variables` block. This is to ensure a clean source of input for a Packer build.

Populating variables

If you're not setting a default value for a variable then variable values must be provided to Packer at runtime. There are several different ways these can be populated. The first is via the command line at build time using the `-var` flag.

Listing 3.6: Specifying a variable on the command line

```
$ packer build \
  -var 'aws_access_key=secret' \
  -var 'aws_secret_key=reallysecret' \
  initial_ami.json
```

You can specify the `-var` flag as many times as needed to specify variable values. If you attempt to define the same variable more than once, the last definition of the variable will stand.

You can also specify variable values in a JSON file. For example, you could create a file called `variables.json` in our `initial_ami` directory and populate it.

Listing 3.7: Specifying variable values in a file

```
{
  "aws_access_key": "secret",
  "aws_secret_key": "reallysecret"
}
```

You can then specify the `variables.json` file on the command line with the `-var -file` flag like so:

Listing 3.8: Specifying the variable file

```
$ packer build \
  -var-file=variables.json \
  initial_ami.json
```

You can define multiple files using multiple instances of the `-var-file` flag. Like variables specified on the command line, if a variable is defined more than once then the last definition of the variable stands.

Builders

The next block is the engine of the template, the `builders` block. The builders turn our template into a machine and then into an image. For our Amazon AMI image we're using the `amazon-ebs` builder. There are a variety of Amazon AMI builders; the `amazon-ebs` builder creates AMI images backed by EBS volumes.

TIP See this storage reference page for more details.

Listing 3.9: The initial_ami builders block

```
"builders": [{
  "type": "amazon-ebs",
  "access_key": "{{user `aws_access_key`}}",
  "secret_key": "{{user `aws_secret_key`}}",
  "region": "us-east-1",
  "source_ami": "ami-a025aeb6",
  "instance_type": "t2.micro",
  "ami_name": "packer-example {{timestamp | clean_ami_name}}"
}]
```

Specify the builder as an element inside a JSON array. Here we've only specified one builder, but you can specify more than one to build a series of images.

TIP We'll see more about multiple builders in Chapter 7.

Specify the builder you want to use using the `type` field, and note that each build in Packer has to have a name. In most cases this defaults to the name of the builder, here specified in the `type` key as `amazon-ebs`. However, if you need to specify multiple builders of the same type—such as if you're building two AMIs—then you need to name your builders using a `name` key.

Listing 3.10: Naming builders blocks

```
"builders": [
  {
    "name": "amazon1",
    "type": "amazon-ebs",
. . .
  },
  {
    "name": "amazon2",
    "type": "amazon-ebs",
  }
]
```

Here we've specified two builders, one named `amazon1`, the other `amazon2`, both with a type of `amazon-ebs`.

NOTE If you specify two builders of the same type, you must name at least one of them. Builder names need to be unique.

To configure the builder, we also specify a number of parameters, called keys, varying according to the needs and configuration of the builder. The first two keys we've specified, `access_key` and `secret_key`, are the credentials to use with AWS to build our image. These keys reference the variables we've just created. We reference variables with the `user` function. Again, our function is wrapped in braces, `{{ }}`, and the input of the variable name is surrounded by back ticks.

Listing 3.11: Referencing variables

```
"access_key": "{{user `aws_access_key`}}",
"secret_key": "{{user `aws_secret_key`}}",
```

This will populate our two keys with the value of the respective variables.

We also specify some other useful keys: the region in which to build the AMI, and the source AMI to use to build the image. This is a base image from which our new AMI will be constructed. In this case we've chosen an Ubuntu 17.04 base AMI located in the `us-east-1` region.

TIP If you're running this build in another region, then you'll need to find an appropriate image.

We also specify the type of instance we're going to use to build our image, in our case `t2.micro` instance type, which should be in the free tier for most accounts.

Lastly, we specify a name for our AMI:

Listing 3.12: Naming our AMI

```
"ami_name": "packer-example {{timestamp | clean_ami_name}}"
```

Our AMI name uses two functions: `timestamp` and `clean_ami_name`. The `timestamp` function returns the current Unix timestamp. We then feed it into the `clean_ami_name` function, which removes any characters that are not supported in an AMI name. This also gives you some idea of how you can call functions and chain functions together to pipe the output from one function as the input of

another.

The resulting output of both functions is then added as a suffix to the name `packer-example`. So the final AMI name would look something like:

```
packer-example 1495043044
```

We do this because AMI images need to be uniquely named.

NOTE There's also a `uuid` function that can produce a UUID if you want a more granular name resolution than time in seconds.

Communicators

Packer builders communicate with the remote hosts they use to build images with a series of connection frameworks called communicators. You can consider communicators as the "transport" layer for Packer. Currently, Packer supports SSH (the default), and WinRM (for Microsoft Windows), as communicators. Communicators are highly customizable and expose most of the configuration options available to both SSH and WinRM. You configure most of these options in the individual builder. We'll see some of this in Chapter 4, and there is also documentation on the Packer site.

Validating a template

Before we build our image, let's ensure our template is correct. Packer comes with a useful validation sub-command to help us with this. It performs syntax checking and validates that the template is complete.

We can run validation with the `packer validate` command.

Listing 3.13: Validating a template

```
$ packer validate initial_ami.json
Failed to parse template: Error parsing JSON: invalid character
'"' after object key:value pair
At line 4, column 6 (offset 49):
    3:     "aws_access_key": ""
    4:     "
           ^
```

Oops. Looks like we have a syntax error in our template. Let's fix that missing comma and try to validate it again.

Listing 3.14: Validating the template again

```
$ packer validate initial_ami.json
Template validated successfully.
```

Now we can see that our `initial_ami.json` template has been parsed and validated. Let's move on to building our image.

TIP You can use the `packer inspect` command to interrogate a template and see what it does.

Building our image

When Packer is run, the `amazon-ebs` builder connects to AWS and creates an instance of the type specified and based on our source AMI. It then creates a new AMI from the instance just created and saves it on Amazon. We execute the build by running the `packer` command with the `build` flag like so:

Listing 3.15: Building our initial AMI image

```
$ packer build initial_ami.json
amazon-ebs output will be in this color.

==> amazon-ebs: Prevalidating AMI Name...
==> amazon-ebs: Error querying AMI: NoCredentialProviders: no
valid providers in chain
==> amazon-ebs: caused by: EnvAccessKeyNotFound: failed to find
credentials in the environment.
==> amazon-ebs: SharedCredsLoad: failed to load profile, .
==> amazon-ebs: EC2RoleRequestError: no EC2 instance role found
==> amazon-ebs: caused by: RequestError: send request failed

. . .

==> Builds finished but no artifacts were created.
```

Ah. Our build has failed as we haven't specified values for our variables. Let's try that again, this time with some variable values defined.

Listing 3.16: Building our initial AMI image again

```
$ packer build \
  -var 'aws_access_key=secret' \
  -var 'aws_secret_key=reallysecret' \
  initial_ami.json
```

TIP The Amazon EC2 builders also support AWS local credential configuration. If we have local credentials specified, we could skip defining the variable values.

Now when we run the build you'll start to see output as Packer creates a new instance and builds an image from it.

Listing 3.17: Building the actual initial_ami image

```
amazon-ebs output will be in this color.

==> amazon-ebs: Prevalidating AMI Name...
    amazon-ebs: Found Image ID: ami-a025aeb6
==> amazon-ebs: Creating temporary keypair: packer_591c9ddd-aff8-
2980-8656-3f4187dc0627
==> amazon-ebs: Creating temporary security group for this
instance...
==> amazon-ebs: Authorizing access to port 22 the temporary
security group...
==> amazon-ebs: Launching a source AWS instance...
    amazon-ebs: Instance ID: i-0da4443cbe9779acc
==> amazon-ebs: Adding tags to source instance
    amazon-ebs: Adding tag: "Name": "Packer Builder"
==> amazon-ebs: Waiting for SSH to become available...
==> amazon-ebs: Connected to SSH!
==> amazon-ebs: Stopping the source instance...
==> amazon-ebs: Waiting for the instance to stop...
==> amazon-ebs: Creating the AMI: initial-ami 1495047645
    amazon-ebs: AMI: ami-6a40397c
==> amazon-ebs: Terminating the source AWS instance...
==> amazon-ebs: Cleaning up any extra volumes...
==> amazon-ebs: No volumes to clean up, skipping
==> amazon-ebs: Deleting temporary security group...
==> amazon-ebs: Deleting temporary keypair...
Build 'amazon-ebs' finished.

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs: AMIs were created:

us-east-1: ami-6a40397c
```

We can see our log output is colored for specific builders. Log lines from a specific builder are prefixed with the name of the builder: `amazon-ebs`.

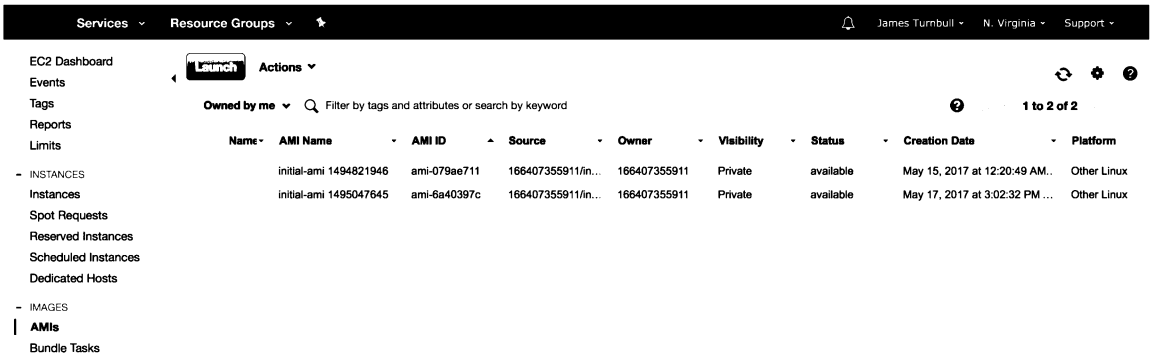

Figure 3.2: Our new AMI

Summary

You've now created your first Packer image. It wasn't very exciting though—just a clone of an existing AMI image. What happens if we want to add something to or change something about our image? In the next chapter we'll learn all about Packer's provisioning capabilities.

Chapter 4

Provisioning with Packer

In the last chapter, we created an Amazon AMI from a stock Ubuntu 17.04 base image. It was a pretty boring image though—essentially a clone of that stock image, renamed. To customize our image a bit we're going to take advantage of a Packer component called a provisioner.

Provisioners execute actions on the image being built. These actions, depending on the provisioner, can run scripts or system commands, and execute third-party tools like configuration management. Provisioners come in a variety of types. You can use one or more types of provisioners during a build—for example, you could use one provisioner to configure and install the requirements for another provisioner.

In this chapter we'll explore several provisioners, starting with the shell provisioner that executes commands and scripts during an image build. We'll also look at two configuration-management–based provisioners: a Puppet server-less approach and a Puppet client-server approach.

Provisioner basics

Provisioners are defined in their own JSON array, `provisioners`, inside your Packer template. Let's create a new template based on our `initial_ami.json` template.

Listing 4.1: Creating a new template

```
$ cp initial_ami.json shell_prov.json
```

Now let's add the `provisioners` block to that new template.

Listing 4.2: Adding the provisioners block

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": ""
  },
  "builders": [{
    "type": "amazon-ebs",
    "access_key": "{{user `aws_access_key`}}",
    "secret_key": "{{user `aws_secret_key`}}",
    "region": "us-east-1",
    "source_ami": "ami-a025aeb6",
    "instance_type": "t2.micro",
    "ssh_username": "ubuntu",
    "ami_name": "shell-prov {{timestamp | clean_ami_name}}"
  }],
  "provisioners": [
  ]
}
```

Each provisioner is defined as an element in the `provisioners` array. Every pro-

visioner has one required key: `type`, the type of provisioner.

As we briefly mentioned in Chapter 3, to allow Packer to communicate with the remote host, Packer uses communicators. You can consider communicators as the "transport" layer for Packer. Currently, Packer supports SSH and WinRM as transports, the default being SSH. As more than just provisioners use communicators, they are configured inside the `builders` block. We could, for example, enable an SSH bastion host.

Listing 4.3: Adding a bastion host

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": ""
  },
  "builders": [{
    "type": "amazon-ebs",
    "access_key": "{{user `aws_access_key`}}",
    "secret_key": "{{user `aws_secret_key`}}",
    "region": "us-east-1",
    "source_ami": "ami-a025aeb6",
    "instance_type": "t2.micro",
    "ssh_username": "ubuntu",
    "ami_name": "shell-prov {{timestamp | clean_ami_name}}",
    "ssh_bastion_host": "bastion.example.com",
    "ssh_bastion_username": "bastion_user",
    "ssh_bastion_password": "bastion_password"
  }],
  "provisioners": [
  ]
}
```

Here we've specified an SSH bastion host with a username and password. When Packer executes a provisioner, it'll now route any SSH connections through this host first.

Let's take a look at some provisioners, starting with the shell provisioner.

Provisioning via the shell provisioner

The shell provisioner executes scripts and commands on the remote image being built. It connects to the remote image via SSH and executes any commands using a shell on that remote host.

The `shell` provisioner can execute a single command, a series of commands, a script, or a series of scripts. Let's start with executing a single command on our remote image.

Listing 4.4: Specifying a single command

```
"provisioners": [{
  "type": "shell",
  "inline": ["sudo apt-get -y install nginx"]
}]
```

The `inline` key can specify an array of one or more commands to be executed when the remote image is created. In this case we're installing the `nginx` package via the `apt-get` command. Let's run our build and see our provisioner in action.

Listing 4.5: Running a single command

```
packer build shell_prov.json
```

We've run the `packer build` command for our new template. Let's see some of the output of that command now.

TIP We skipped specifying the AWS variables because we configured some local AWS credentials.

Listing 4.6: Running our first provisioner

```
. . .

==> amazon-ebs: Waiting for SSH to become available...
==> amazon-ebs: Connected to SSH!
==> amazon-ebs: Provisioning with shell script: /var/folders/tb/
kbv8ry4946j_zplwfb2bbjcc0000gn/T/packer-shell845619060
    amazon-ebs: Reading package lists...
    amazon-ebs: Building dependency tree...

. . .

    amazon-ebs: Setting up nginx-core (1.10.3-1ubuntu3) ...

. . .

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs: AMIs were created:

us-east-1: ami-b3fab6a5
```

TIP There's also two flags, `-debug` flag which provides more information and interaction when building complex templates, and `-on-error`, which tells Packer what to do when something goes wrong. The `-debug` flag also disables parallelization and is more verbose. The Packer documentation has some more general debug tips.

So what's happening here? Our image is created and Packer waits for SSH to be available on the host. When it detects an open SSH service, Packer will connect to that service. Packer will use default credentials to connect to the host; for example, for Amazon, it'll use a temporary key pair and connect to the default operating system user. For Ubuntu, this is `ubuntu`. Packer will have the permissions and capabilities of that user.

Packer will convert our `apt-get` command into a script file, upload it (by default with a name of `script_nnn.sh` and into the `/tmp` directory), and then execute it on the remote host. We can see that the `apt-get install` has been executed, and the `nginx` package has been installed. The script will end; assuming it returns an exit code of `0`, the image will be built and an AMI returned.

NOTE The shell provisioner has a cousin called shell-local that runs commands locally on the host running Packer.

We can also run a series of commands by adding each command to the array.

Listing 4.7: Adding a series of commands

```
"provisioners": [{
  "type": "shell",
  "inline": [
    "sudo apt-get -y install nginx",
    "sudo mkdir -p /var/www/website"
  ]
}]
```

If we execute Packer again we'll see Nginx installed and the `/var/www/website` directory will be created in our image. This assumes both commands exited with a `0` exit code. If either fail then the whole build will fail.

TIP The `inline` commands are executed with a shebang of `/bin/sh -e`. You can adjust this using the inline_shebang key.

Running a script

In addition to a command or series of commands you can run a script using the `script` key. The `script` key is the path to a script to be executed. Let's see it now.

Listing 4.8: Adding the script key

```
"provisioners": [{
    "type": "shell",
    "script": "install.sh"
}]
```

The `script` key points to a script to be executed. The location of the script can be absolute or relative, depending on how it is specified. If it is specified relative, then it is relative to the location of the template file.

In this case our script, `install.sh`, is relatively defined and located with our `shell_prov.json` template. Let's create that script now.

Listing 4.9: The install.sh script

```
#!/bin/sh -x

# Install nginx

sudo apt-get -y install nginx
sudo mkdir -p /var/www/website
sudo sh -c "echo '<html><h1>This is a website</h1></html>' > /
var/www/website/index.html"
```

NOTE The `-x` flag on the shebang is useful as it echo'es the commands executed, allowing you to see what's happening when your script is run.

Our script performs the actions that our previous commands defined in the `inline` key but adds a third step: it creates an `index.html` file with some stock content.

If we execute a Packer build with our updated template we'll see something like the following:

Listing 4.10: Executing a shell script

```
$ packer build shell_prov.json
amazon-ebs output will be in this color.

==> amazon-ebs: Prevalidating AMI Name...
    amazon-ebs: Found Image ID: ami-a025aeb6

. . .

==> amazon-ebs: Waiting for SSH to become available...
==> amazon-ebs: Connected to SSH!
==> amazon-ebs: Provisioning with shell script: install.sh
    amazon-ebs: + sudo apt-get -y install nginx

. . .
```

Packer will upload the `install.sh` script, again to `/tmp`, and execute it. By default, it'll execute the command by `chmod`'ing it into an executable form and running it. This means the script you write needs to have a shebang, then can be executed by being directly called.

TIP You can modify the method by which Packer executes scripts by changing the execute_command key, for example instead of making the script executable you could call it with another binary.

Running more than one script

The last execution method for the `shell` provisioner is to execute a series of scripts, expressed as an array in the `scripts` key. The scripts will be executed in the

sequence in which they are defined.

Again, the scripts are specified absolutely or relatively to the template location. Let's define two scripts: our existing `install.sh` script and a new `post-install.sh` script.

Listing 4.11: Adding a second script

```
"provisioners": [{
  "type": "shell",
  "scripts": [
    "install.sh",
    "post-install.sh"
  ]
}]
```

Now let's create our `post-install.sh` script.

Listing 4.12: The post-install.sh script

```
#!/bin/sh -x

# Run nginx at boot
sudo systemctl enable nginx
```

Now let's run a Packer build and see what happens with two scripts specified.

Listing 4.13: Running two scripts

```
$ packer build shell_prov.json

. . .

==> amazon-ebs: Waiting for SSH to become available...
==> amazon-ebs: Connected to SSH!
==> amazon-ebs: Provisioning with shell script: install.sh
    amazon-ebs: + sudo apt-get -y install nginx

. . .

    amazon-ebs: + sudo mkdir -p /var/www/website
    amazon-ebs: + sudo sh -c echo '<html><h1>This is a website</
h1></html>' > /var/www/website/index.html
==> amazon-ebs: Provisioning with shell script: post-install.sh
    amazon-ebs: + sudo systemctl enable nginx
    amazon-ebs: Synchronizing state of nginx.service with SysV
service script with /lib/systemd/systemd-sysv-install.

. . .
```

We can see each script executed in sequence, and that inside each script, each command is executed in turn. Now when our image is complete we'll have installed Nginx, created a directory and `index.html` file, and enabled Nginx to run when an EC2 instance is launched from our AMI image.

You'll notice one of the things we did in our shell provisioning was create an `index.html` file by echo'ing content into a file—a rather clumsy way to manage this. Packer provides a provisioner that allows us to provision files, like content or configuration files, into our host.

TIP If your provisioning process requires a reboot or restart, you can config-

ure Packer to handle delays, failures, and retries.

File provisioning

The `file` provisioner uploads files, via Packer's communicators, by default SSH from our local host to the remote host. The `file` provisioner is usually used in conjunction with the `shell` provisioner: the `file` provisioner uploads a file then the `shell` provisioner manipulates the uploaded file. (This two-step process primarily caters for file permission issues—Packer typically only has permission to upload files to locations it can write to, such as `/tmp`.) We can then execute the `shell` provisioner with escalated privileges—for example, by prefixing it with the `sudo` command.

Let's update our provisioner example by creating a new template called `file_prov.json`.

Listing 4.14: Creating a new file provisioner template

```
$ cp shell_prov.json file_prov.json
```

Let's remove the creation of our `index.html` file from the `install.sh` script and instead upload a file via the `file` provisioners. Let's add the `file` provisioners to the `file_prov.json` file.

Listing 4.15: Adding the file provisioner

```
"provisioners": [
  {
    "type": "file",
    "source": "index.html",
    "destination": "/tmp/index.html"
  },
  {
    "type": "shell",
    "scripts": [
     "install.sh",
     "post-install.sh"
    ]
  }
]
```

The `file` provisioner specifies a `source` and `destination` for the file. The `source` is defined absolutely or relative to the template. In our case, the `index.html` file is located in the same directory as our template. The `destination` is on the remote host and Packer must be able to write to it. Packer also can't create any parent directories—you'll either need to create those with a `shell` provisioner command or script prior to the upload, or upload to an existing directory. Here we're uploading to `/tmp`, which Packer has write access to, and we know the parent directory, being root, will exist.

We could then update our `install.sh` script to move our `index.html` file into the right place. Or we could even add a second `shell` provisioner to perform the prerequisite actions, like so:

Listing 4.16: Moving index.html

```
"provisioners": [
  {
    "type": "file",
    "source": "index.html",
    "destination": "/tmp/index.html"
  },
  {
    "type": "shell",
    "inline": [
      "sudo mkdir -p /var/www/website",
      "sudo mv /tmp/index.html /var/www/website"
    ]
  },
  {
    "type": "shell",
    "scripts": [
      "install.sh",
      "post-install.sh"
    ]
  }
]
```

Here we've chained together three provisioners: the `file` provisioner to upload our file, several inline shell commands, and then finally executing two shell scripts. You can see how this chaining can allow some complex provisioning operations.

Uploading a directory

In addition to uploading single files, we can also use the `file` provisioner to upload whole directories. Like with single file provisioning, the destination directory must exist. And upload behavior is much like `rsync`: the existence of a trailing slash determines the behavior of the upload. Let's see an example:

Listing 4.17: Uploading a whole directory

```
"provisioners": [
  {
    "type": "file",
    "source": "website",
    "destination": "/tmp"
  },
```

If neither the `source` nor the `destination` have a trailing slash then the local directory will be uploaded into the remote directory—hence the `website` directory would be uploaded to `/tmp/website`.

If the `source` has a trailing slash and the `destination` does not, then the contents of the directory will be uploaded directly into the destination—hence the contents of the `website` directory would be uploaded into the `/tmp` directory.

TIP You can also upload symbolic links with Packer but most provisioners will treat them as regular files.

Provisioning via configuration management

We can see how potentially powerful provisioning with files and shell commands can be. But, like traditional build and configuration processes, this can become unwieldy and complex. Additionally, if you already use a configuration management tool to manage your infrastructure, why replicate that configuration and those capabilities inside Packer? Instead, you could take advantage of the capabilities of a configuration management tool and reuse any existing configuration

you might have available.

We'll demonstrate these capabilities using Puppet. Puppet is an open-source configuration management platform which will help us understand how Packer can apply complex configurations. In addition to Puppet, Packer also supports Ansible, Chef, Salt, and Converge.

Packer can use Puppet in two ways: in a standalone configuration or using Puppet's traditional client-server model. We're not going to show you how to use Puppet or build a Puppet environment—we're going to assume you already have some knowledge and insight there. If you don't, it's still easy to see how to adapt these techniques to other configuration management tools.

Puppet standalone

Let's start by running Puppet in its standalone mode. This is where we provide all the configuration for Puppet locally on our build host and do not connect to a Puppet server. We're going to install Puppet, configure it, and then use it to configure SSH services in our image. We'll use a community SSH module to configure the SSH server and client in our image.

First, let's create a new directory to hold this configuration.

Listing 4.18: Creating Puppet standalone directory

```
$ mkdir puppet_standalone
```

Next we'll create a template for our Packer and Puppet-provisioned image, a shell script to install Puppet, and some directories to hold our Puppet configuration.

Listing 4.19: Creating Puppet standalone template

```
$ cd puppet_standalone
$ touch puppet_standalone.json
$ touch install-puppet.sh
$ mkdir {manifests,hieradata}
```

Let's take a quick look at our template. We'll focus on the `provisioners` block for now, as it's the most relevant section.

Listing 4.20: The Puppet standalone provisioners block

```
"provisioners": [
  {
    "type": "shell",
    "script": "install-puppet.sh"
  },
  {
    "type": "file",
    "source": "hieradata",
    "destination": "/tmp"
  },
  {
    "type": "puppet-masterless",
    "puppet_bin_dir": "/opt/puppetlabs/bin",
    "manifest_file": "manifests/site.pp",
    "hiera_config_path": "hiera.yaml",
    "module_paths": ["modules"]
  }
]
```

Notice that we've chained together three provisioners: the `shell`, `file`, and a new provisioner `puppet-masterless`.

Installing Puppet

Our `shell` provisioner executes a single script called `install-puppet.sh`. Let's look at that script now.

Listing 4.21: The install-puppet.sh shell script

```
#!/bin/sh -x

echo "Adding puppet repo"
sudo curl https://apt.puppetlabs.com/puppetlabs-release-pc1-
xenial.deb -o puppetlabs-release.deb
sudo dpkg -i puppetlabs-release.deb

echo "Updating package info..."
sudo apt-get update -y -qq

echo "Upgrading packages..."
sudo apt-get dist-upgrade -y -qq
sudo apt-get install unzip language-pack-en puppet-agent -y -qq
```

The script adds the Puppet APT repo, then refreshes the package database and installs the `puppet-agent`. The `puppet-masterless` provisioner will later use this agent to execute any Puppet code and configure our host.

Hiera and manifest data

The next provisioner is a `file` provisioner. This loads a Hiera data directory onto the host. Hiera is Puppet's data backend; we're going to use it to tell Puppet what configuration to install.

NOTE We need to upload this directory due to a nuance in the way Puppet is run. Other configuration will remain local.

Hiera configuration is contained in YAML files. We're going to be create two files. The first, `hiera.yaml`, tells Puppet about Hiera's configuration and structure.

Listing 4.22: The hiera.yaml file

```
---
version: 5
defaults:
  datadir: '/tmp/hieradata'
  data_hash: yaml_data
hierarchy:
  - name: "Other YAML hierarchy levels"
    paths:
      - "common.yaml"
```

The `hiera.yaml` file also tells Puppet where to find our second file. We'll create the second file, `common.yaml`, in the `hieradata` directory. This file tells Puppet what configuration to install.

Listing 4.23: The common.yaml file

```
---
classes:
  - ssh
```

This tells Puppet that we want to load the `ssh` module. Let's quickly add a manifest to consume this data. We do that by creating a `site.pp` file in the `manifests` directory.

Listing 4.24: The site.pp file

```
lookup('classes', {merge => unique}).include
```

We've specified one line; your manifest is likely to be more complex. We've used the `lookup` function to query our Hiera data. This will lookup the `classes` array in our `common.yaml` file and load the resulting data into Puppet. Since this array contains one module, `ssh`, this will tell Puppet to load and apply the `ssh` module.

Our `file` provisioner uploads the local `hieradata` directory to the `/tmp` directory on the remote host. Our `puppet-masterless` provisioner will look for it here thanks to our Hiera configuration.

Let's go get the `ssh` module now and install it.

Installing modules

To install modules we're going to use the `librarian-puppet` tool to manage our modules. Let's install Puppet and that tool first.

NOTE To do this, you'll need to have Ruby and Rubygems installed already.

Listing 4.25: Installing librarian-puppet

```
$ sudo gem install puppet librarian-puppet
```

NOTE We've also added a `Gemfile` in the directory so if you have Bundler installed you can just `bundle install`.

Librarian-Puppet uses a `Puppetfile` file to define what modules we want to use. This is much like a Ruby `Gemfile`. Let's create it now.

Listing 4.26: The Puppetfile

```
forge "https://forgeapi.puppetlabs.com"
mod 'saz/ssh'
```

The `Puppetfile` specifies the location of the Puppet Forge from which we'll get the SSH module. It also lists the specific module, `saz/ssh`, that we'll retrieve.

We can then run the `librarian-puppet` command to install the `ssh` module.

Listing 4.27: Installing the SSH module

```
$ librarian-puppet install
```

You'll see that a `Puppetfile.lock` lock file will be created, as will the `modules` directory containing the downloaded module and any supporting modules. We're now ready to configure Puppet.

Configuring Puppet standalone

The next provisioner is the `puppet-masterless` provisioner. It takes the configuration and pieces we've downloaded and constructed earlier and combines them to execute on the remote host. Let's take a look at our provisioner code.

Listing 4.28: The puppet-masterless provisioner

```
{
  "type": "puppet-masterless",
  "puppet_bin_dir": "/opt/puppetlabs/bin",
  "manifest_file": "manifests/site.pp",
  "hiera_config_path": "hiera.yaml",
  "module_paths": ["modules"]
}
```

We've specified the `type` of our provisioner: `puppet-masterless`. We've also told Packer where to find the `puppet` binary we're going to run to provision the host. We do this because the Puppet installer places the binary in the `/opt/puppetlabs/bin` directory which isn't usually in our path.

We then reference the configuration we've done most recently. We tell Packer where to find our `site.pp` manifest file, the path to the `hiera.yaml` configuration file, and the directory containing our modules.

Let's run Packer now and see what happens.

Running Puppet standalone

We've got Packer configured to execute Puppet standalone. If we run Packer now we'll:

1. Install the Puppet agent on the remote host.
2. Upload our Hiera data to the remote host.
3. Run the Puppet agent via Packer to install the `ssh` module.

Let's see that in action.

Listing 4.29: Running Packer and Puppet standalone

```
$ packer build puppet_standalone.json
amazon-ebs output will be in this color.

. . .

==> amazon-ebs: Provisioning with shell script: install-puppet.
sh
    amazon-ebs: Adding puppet repo

. . .

    amazon-ebs: + sudo apt-get install unzip language-pack-en
puppet-agent -y -qq

. . .

==> amazon-ebs: Uploading hieradata => /tmp
==> amazon-ebs: Provisioning with Puppet...
    amazon-ebs: Creating Puppet staging directory...
    amazon-ebs: Uploading hiera configuration...
    amazon-ebs: Uploading local modules from: modules
    amazon-ebs: Uploading manifests...
    amazon-ebs: Uploading manifest file from: manifests/site.pp
    amazon-ebs: Running Puppet: cd /tmp/packer-puppet-masterless
 && FACTER_packer_build_name='amazon-ebs'
FACTER_packer_builder_type='amazon-ebs'  sudo -E /opt/puppetlabs/
bin/puppet apply --verbose --modulepath='/tmp/packer-puppet-
masterless/module-0' --hiera_config='/tmp/packer-puppet-
masterless/hiera.yaml' --detailed-exitcodes /tmp/packer-puppet-
masterless/manifests/site.pp

. . .

    amazon-ebs: Notice: Applied catalog in 0.17 seconds

. . .

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs: AMIs were created:
us-east-1: ami-36105a20
```

 **TIP** There might be some warning messages from Puppet. You can safely ignore these—they are usually deprecations and informational messages.

We've cut out a lot of the log lines and just showed you the various steps executed with some more detail for the Puppet execution. We can see our packages installed, the `hieradata` directory uploaded, and the `puppet-masterless` provisioner executed last. Here our various components are uploaded and placed on the remote host, and the `puppet` binary is executed in `apply` mode to configure the host with the `ssh` module. If we dig into the Puppet execution we'll also see some of the output from the module.

Listing 4.30: Some output from the Puppet module

```
amazon-ebs: Info: Applying configuration version '1496586621'
amazon-ebs: Info: /Stage[main]/Ssh::Client::Config/File[/etc/ssh/
ssh_config]: Filebucketed /etc/ssh/ssh_config to puppet with sum
 d26206bb4490f72d051d2642c3bf03ee
amazon-ebs: Notice: /Stage[main]/Ssh::Client::Config/File[/etc/
ssh/ssh_config]/content: content changed '{md5}
d26206bb4490f72d051d2642c3bf03ee' to '{md5}6
b6e968bce40150262b7ab85822e7c07'
amazon-ebs: Notice: /Stage[main]/Ssh::Server::Config/Concat[/etc/
ssh/sshd_config]/File[/etc/ssh/sshd_config]/mode: mode changed
'0644' to '0600'

. . .

amazon-ebs: Notice: Applied catalog in 0.17 seconds
```

 **TIP** If the Puppet run fails, Packer will fail the build.

You can see that this is a useful way to take advantage of your existing Puppet configuration data, or any of the other configuration management tools that Packer supports. However, we can also make use of an existing Puppet server implementation.

Puppet server

To make use of an existing Puppet server, we use the `puppet-server` provisioner. This provisioner is much simpler than our previous Puppet standalone approach. Here, we just need to point our Packer template at an appropriate Puppet server, configured to provision our remote host building the image. We'll again use our `ssh` module, this time delivered via the Puppet server.

NOTE Need help setting up a Puppet server? Check out the Puppet on AWS documentation for an example.

Let's take a look at a template for that purpose. We'll call it `puppet_server.json` and create it in a new directory called `puppet_server`.

TIP We could also make use of an existing Chef implementation.

Listing 4.31: Creating the puppet_server directory and template

```
$ mkdir puppet_server && cd puppet_server
$ touch puppet_server.json
```

Not let's populate that file with our template. Again we'll build an Amazon AMI and focus on the `provisioners` block inside that template.

Listing 4.32: The puppet_server provisioners blocks

```
"provisioners": [
  {
    "type": "shell",
    "script": "install-puppet.sh"
  },
  {
    "type": "puppet-server",
    "options": "--test",
    "facter": {
      "role": "appserver"
    }
  }
]
```

Again, like with the Puppet standalone approach, we need to install Puppet, as Packer won't do it for us. We'll use the same `shell` provisioner script we used in the previous section to perform that installation.

Next we specify the `puppet-server` provisioner itself. We specify one option, `--test`, to pass to the Puppet server. The `--test` option enables a series of useful options, including log verbosity, that will allow us to see what Puppet is doing.

Our configuration assumes that we've got a configured Puppet master running. By default, Puppet looks for a Puppet server via DNS using `puppet` as the host name. We're also assuming that the Puppet master is configured to appropriately sign agents and deliver our `ssh` module to those agents.

TIP The exact command that Packer executes can be viewed in the provisioner documentation.

We can also control exactly what configuration is delivered by the host via Hiera or customized via Facter facts. You can see that we've specified the `factor` hash, which allows us to set Facter facts during our Puppet run. We've set a new fact called `role` with a value of `appserver`. The `puppet-server` provisioner also creates two facts by default: `packer_build_name`, set to the name of the build, and `packer_builder_type`, set to the type of builder being used.

Both are useful for selectively configuring images based on the type and name of the build.

Now we can run Packer and see the provisioner in action.

Listing 4.33: Executing the Puppet server provisioner

```
$ packer build puppet_server.json

. . .

==> amazon-ebs: Provisioning with shell script: install-puppet.
sh

. . .

==> amazon-ebs: Provisioning with Puppet...
    amazon-ebs: Creating Puppet staging directory...
    amazon-ebs: Running Puppet: FACTER_server_role='webserver'
FACTER_packer_build_name='amazon-ebs' FACTER_packer_builder_type
='amazon-ebs'  sudo -E /opt/puppetlabs/bin/puppet agent --
onetime --no-daemonize --test --detailed-exitcodes

. . .

    amazon-ebs: Notice: Applied catalog in 0.24 seconds

. . .

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs: AMIs were created:

us-east-1: ami-fe9fcee8
```

Again we've cut out a lot of the log, but if you follow along as you run Packer you'll see Puppet installed then initiating a connection to the Puppet server and applying the module. The `--test` flag will run Puppet in a very verbose mode, and you'll be able to see each action taken by the module.

With this, you can ensure every build is built using your existing configuration management resources, and you can consistently and centrally manage the state of your image builds.

 **TIP** If there isn't a provisioner that meets your needs, you can add your own via a custom provisioner plugin. Well look at those in Chapter 7.

Summary

That was a bit more exciting, wasn't it? We've now taken Packer and built some images, customized to suit our environment. To do this we introduced the concept of provisioners that allow you to execute commands, upload files, and integrate with configuration management tools like Puppet and Chef. We started simple with shell commands, then worked all the way up to a Puppet standalone and then a Puppet client-server setup.

But we're not totally done with provisioners. In the next chapter we'll look at building Docker images and combining a Docker builder with a selection of provisioners. Then we'll look at the component in the build chain: post-provisioners that allow you to take action after provisioning.

Chapter 5

Docker and Packer

In the last chapter we saw how to use provisioners to customize our images during the build process. In this chapter, we're going to continue to explore building and provisioning images with Packer. And we're going to look at one of Packer's most interesting and complex use cases: building Docker images.

To build Docker images, Packer uses the Docker daemon to run containers, runs provisioners on those containers, then can commit Docker images locally or push them up to the Docker Hub. But, interestingly, Packer doesn't use `Dockerfiles` to build images. Instead Packer uses the same provisioners we saw in Chapter 3 to build images. This allows us to create a consistent mental model for all our images, across all platforms.

We're going to learn how to build and push Docker images with Packer.

 TIP If you want to learn more about Docker, you can look at the Docker documentation or my book on Docker, originally titled The Docker Book.

Getting started with Docker and Packer

When building Docker images, Packer and the Docker builder need to run on a host that has Docker installed. Installing Docker is relatively simple, and we're not going to show you how to do it in this book. There are, however, a lot of resources available online to help you.

TIP Use a recent version of Docker for this chapter—at least Docker 17.05.0-ce or later.

You can confirm you have Docker available on your host by running the `docker` binary.

Listing 5.1: Running the Docker binary

```
$ docker --version
Docker version 17.06.0-ce-rc1, build 7f8486a
```

A basic Docker build

The Docker builder is just like any other Packer builder: it uses resources, in this case a local Docker daemon, to build an image. Let's create a template for a basic Docker build:

Listing 5.2: Creating a base Docker template

```
$ touch docker_basic.json
```

And now populate that template:

Listing 5.3: The basic Docker template

```
{
  "builders": [{
    "type": "docker",
    "image": "ubuntu",
    "export_path": "docker-basic.tar"
  }]
}
```

The template is much like our template from Chapter 3—simple and not very practical. Currently it just creates an image from the `ubuntu` stock image and exports a tar ball of it. Let's explore each key.

The type of builder we've specified is `docker`. We've specified a base image for the builder to work from; this is much like using the `FROM` instruction in a `Dockerfile`, using the `image` key.

The `type`, as always, and the `image` are required keys for the Docker builder. You must also specify what to do with the container that the Docker builder builds.

The Docker builder has three possible output actions. You must specify one:

- Export - Export an image from the container as a tar ball, as above with the `export_path` key.
- Discard - Throw away the container after the build, using the `discard` key.

- Commit - Commit the container as an image available to the Docker daemon that built it, using the `commit` key.

Let's build our template now and see what happens.

Listing 5.4: Creating a Docker tar ball from Packer

```
$ packer build docker_basic.json
docker output will be in this color.

==> docker: Creating a temporary directory for sharing data...
==> docker: Pulling Docker image: ubuntu
    docker: Using default tag: latest
    docker: latest: Pulling from library/ubuntu
    docker: Digest: sha256:
ea1d854d38be82f54d39efe2c67000bed1b0334...
    docker: Status: Image is up to date for ubuntu:latest
==> docker: Starting docker container...
    docker: Run command: docker run -v /Users/james/.packer.d/
tmp/packer-docker307764002:/packer-files -d -i -t ubuntu /bin/
bash
    docker: Container ID: 6
a872b49ce499f62c37c5a1a1e609c557dc36879...
==> docker: Exporting the container
==> docker: Killing the container: 6
a872b49ce499f62c37c5a1a1e609c557dc36879...
Build 'docker' finished.

==> Builds finished. The artifacts of successful builds are:
--> docker: Exported Docker file: docker_basic.tar
```

We can see that Packer has pulled down a base Docker image, `ubuntu`, that's running a new container, then it's exported the container as `docker_basic.tar`. You could now use the `docker import` command to import that image from the tar ball.

NOTE We'll see other actions we can take with the final image later in this chapter.

Let's do something a bit more complex in our next build.

Provisioning a Docker image

Let's create a new template, `docker_prov.json`, that will combine a Docker build with provisioning of a new image. Rather than export the new image we're going to create, we're going to commit the image to our local Docker daemon. Let's take a look at our template.

Listing 5.5: Creating Docker image with provisioning

```
{
  "builders": [{
    "type": "docker",
    "image": "ubuntu",
    "commit": true
  }],
  "provisioners": [{
    "type": "shell",
    "script": "install.sh"
  }]
}
```

In our new template, we've replaced the `export_path` key with the `commit` key, which is set to `true`. We've also added a `provisioners` block and specified a single script called `install.sh`. Let's look at that script now.

Listing 5.6: The install.sh Docker provisioning script

```
#!/bin/sh -x

# Update apt
apt-get -yqq update

# Install Apache
apt-get -yqq install apache2
```

Our script updates APT and then installs the `apache2` package.

When we run `packer` on this template it'll create a new container from the `ubuntu` image, run the `install.sh` script using the `shell` provisioner, and then commit a new image.

Let's see that now.

Listing 5.7: Running a Docker provisioner

```
$ packer build docker_prov.json

. . .

==> docker: Provisioning with shell script: install.sh
    docker: + apt-get -yqq update
    docker: + apt-get -yqq install apache2

. . .

==> docker: Committing the container
    docker: Image ID: sha256:dd465405e2f4880b50ef6468d9e18a1f...
==> docker: Killing the container: 13786
af85560b9169b71126024aacb6e...
Build 'docker' finished.

==> Builds finished. The artifacts of successful builds are:
--> docker: Imported Docker image: sha256:
dd465405e2f4880b50ef6468d9e18a1f...
```

Here we can see our Docker image has been built from the `ubuntu` image and then provisioned using our `install.sh` and the Apache web server installed. We've then outputted an image, stored in our local Docker daemon.

Instructions and changes

Sometimes a provisioner isn't quite sufficient and you need to take some additional actions to make a container fully functional. The `docker` builder comes with a key called `changes` that allows you to specify some `Dockerfile` instructions.

NOTE The `changes` key behaves in much the same way as the `docker`

`commit --change` command line option.

We can use the `changes` key to supplement our existing template:

Listing 5.8: Using the changes key

```
{
  "type": "docker",
  "image": "ubuntu",
  "commit": true,
  "changes": [
    "USER www-data",
    "WORKDIR /var/www",
    "EXPOSE 80"
  ]
}
```

Here we've added three instructions: `USER`, which sets the default user; `WORKDIR`, which sets the working directory; and `EXPOSE`, which exposes a network port. These instructions will be applied to the image being built and committed to Docker.

You can't change all `Dockerfile` instructions, but you can change the `CMD`, `ENTRYPOINT`, `ENV`, `EXPOSE`, `MAINTAINER`, `USER`, `VOLUME`, and `WORKDIR` instructions.

This is still only a partial life cycle, and we most often want to do something with the artifact generated by our build. This is where post-processors come in.

Post-processing Docker images

Post-processors take actions on the artifacts, usually images, created by Packer. They allow us to store, distribute, or otherwise process those artifacts. The Docker

workflow is ideal for demonstrating their capabilities. We're going to examine two Docker-centric post-processors:

- `docker-tag` - Tags Docker images.
- `docker-push` - Pushes Docker images to an image store, like the Docker Hub.

Post-processors are defined in another template block: `post-processors`. Let's add some post-processing to a new template, `docker_postproc.json`.

Listing 5.9: Docker post-processing

```
{
  "builders": [{
    "type": "docker",
    "image": "ubuntu",
    "commit": true
  }],
  "provisioners": [
    {
      "type": "shell",
      "script": "install.sh"
    }
  ],
  "post-processors": [
     [
      {
        "type": "docker-tag",
        "repository": "jamtur01/docker_postproc",
        "tag": "0.1"
      },
      "docker-push"
    ]
  ]
}
```

Note that we've added a `post-processors` block with an array of post-processors defined. Packer will take the result of any `builder` action and send it through the

post-processors, so if you have one builder the post-processors will be executed once, two builders will result in the post-processors being executed twice, and so on. You can also control which post-processors run for which build—we'll see more of that in Chapter 7.

TIP Also in Chapter 7, we'll see how multiple builders operate and how to control which post-processors execute.

For each post-processor definition, Packer will take the result of each of the defined builders and send it through the post-processors. This means that if you have one post-processor defined and two builders defined in a template, the post-processor will run twice (once for each builder), by default.

There are three ways to define post-processors: simple, detailed, and in sequence. A simple post-processor definition is just the name of a post-processor listed in an array.

Listing 5.10: The simple post-processor definition

```
{
  "post-processors": ["docker-push"]
}
```

A simple definition assumes you don't need to specify any configuration for the post-processor. A more detailed definition is much like a `builder` definition and allows you to configure the post-processor.

Listing 5.11: A detailed post-processor definition

```
{
  "post-processors": [
    {
      "type": "docker-save",
      "path": "container.tar"
    }
  ]
}
```

Like with a builder or provisioner definition, we specify the type of post-processor and then any options. In our case we use the `docker-save` post-processor which saves the Docker image to a file.

The last type of post-processor definition is a sequence. This is the most powerful use of post-processors, chained in sequence to perform multiple actions. It can contain simple and detailed post-processor definitions, listed in the order in which you wish to execute them.

Listing 5.12: A sequential post-processor definition

```
"post-processors": [
  [
    {
      "type": "docker-tag",
      "repository": "jamtur01/docker_postproc",
      "tag": "0.1"
    },
    "docker-push"
  ]
]
```

You can see our post-processors are inside the `post-processors` array and further

nested within an array of their own. This links post-processors together, meaning their actions are chained or executed in sequence. Any artifacts a post-processor generates is fed into the next post-processor in the sequence.

NOTE You can only nest one layer of sequence.

Our first post-processor is `docker-tag`. You can specify a repository and an optional tag for your image. This is the equivalent of running the `docker tag` command.

Listing 5.13: Tagging a built image

```
$ docker tag image_id jamtur01/docker_postproc:0.1
```

This tags our image with a repository name and a tag that makes it possible to use the second post-processor: `docker-push`.

The `docker-push` post-processor pushes Docker images to a Docker registry, like the Docker Hub, a local private registry, or even Amazon ECR. You can provide login credentials for the push, or the post-processor can make use of existing credentials such as your local Docker Hub or AWS credentials.

TIP You can also see a simple definition of a post-processor, in this case the `docker-push` post-processor, in a sequence.

Let's try to post-process our artifact now.

Listing 5.14: Post-processing an image

```
$ packer build docker_postproc.json
docker output will be in this color.

. . .

==> docker: Running post-processor: docker-tag
    docker (docker-tag): Tagging image: sha256:
b5cf683867f9f1d20149bd106db8b423...
    docker (docker-tag): Repository: jamtur01/docker_postproc
:0.1
==> docker: Running post-processor: docker-push
    docker (docker-push): Pushing: jamtur01/docker_postproc:0.1
    docker (docker-push): The push refers to a repository [
docker.io/jamtur01/docker_postproc]

. . .

    docker (docker-push): 1f833f3fe176: Pushed
    docker (docker-push): 0.1: digest: sha256:2
eba43302071a38bf6347f6c06dcc7113d7... size: 1569
Build 'docker' finished.

==> Builds finished. The artifacts of successful builds are:
--> docker: Imported Docker image: sha256:
b5cf683867f9f1d20149bd106db8b...
```

TIP You can tag and send an image to multiple repositories by specifying the `docker-tag` and `docker-push` post-processors multiple times.

We've cut out a lot of log entries, but you can see our Docker image being tagged and then pushed to my Docker Hub account, `jamtur01`. The image has been pushed to the `docker_postproc` repository with a tag of `0.1`. This assumes we've

got local credentials for the Docker Hub. If you need to specify specific credentials you can add them to the template like so:

Listing 5.15: Docker post-processing with credentials

```
. . .
"variables": {
  "hub_username": "",
  "hub_password": ""
},
{
 "post-processors": [
     [
      {
        "type": "docker-tag",
        "repository": "jamtur01/docker_postproc",
        "tag": "0.1"
      },
      {
        "type": "docker-push",
        "login": true,
        "login_username": "{{user `hub_username`}}",
        "login_password": "{{user `hub_password`}}"
      }
    ]
  ]
}
```

Here we've specified some variables to hold our Docker Hub username and password. This is more secure than hard coding it into the template.

TIP We could also use environment variables.

We've used the `user` function to reference them in the post-processor. We've also

specifies the `login` key and set it to `true` to ensure the `docker-push` post-processor logs in prior to pushing the image.

We can then run our template and specify the variables on the command line:

Listing 5.16: Using Docker Hub credentials

```
$ packer build \
  -var 'hub_username=jamtur01' \
  -var 'hub_password=bigsecret' \
  docker_postproc.json
```

 **TIP** We can also do the same with the Amazon ECR container repository.

Summary

In this chapter we've seen how to combine Packer and Docker to build Docker images. We've seen how we can combine multiple stages of the build process:

1. Starting with a Docker image.
2. Adding some `Dockerfile` instructions.
3. Provisioning to build what we need into the image.
4. Post-processing to work with the committed image, potentially uploading it to a container repository like the Docker Hub.

There are also other post-processors that might interest you. You can find a full list in the Packer documentation.

In the next chapter, we're going to see how we can add tests to our Packer build process to ensure that our provisioning and image are correct.

Chapter 6

Testing Packer

In the last few chapters we've seen how to build images with Packer. Part of that process is provisioning images with the configuration you need. When you're running scripts, or even using configuration management tools, there's some risk that the outcomes of your build won't be what you requested. We can ensure that what we get what we've asked for by using software-based tests.

Software tests validate that your software does what it is supposed to do. They are the bread-and-butter validation tools of developers. 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. 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.

We're going to combine Packer with a testing framework called Serverspec. Serverspec is a testing framework and harness that allows you to write RSpec tests for infrastructure. It sits on top of the RSpec testing framework, shares its DSL, and can leverage all of its capabilities and tooling.

Our test scenario

We're going to build an AMI image using Puppet modules and write Serverspec tests to validate that configuration. As a basis for our image we're going to use the Puppet standalone configuration we created in Chapter 4, with some additional modules that we're also going to install and test. We'll create and upload some Serverspec tests onto our remote host as part of that build process and then execute them to validate our configuration.

Let's quickly revisit that configuration and create a directory structure to hold our template and related configuration:

Listing 6.1: Creating a directory

```
$ mkdir -p serverspec/{hieradata,manifests,tests}
```

Here we've created a directory structure for our build, including a `hieradata` directory to hold our Hiera configuration, a `manifests` directory to hold our Puppet code, and a `tests` directory to hold our new Serverspec tests.

Now let's create a template. We'll call it `serverspec.json`.

Listing 6.2: Creating the Serverspec template

```
$ cd serverspec
$ touch serverspec.json
```

Next we'll populate that template. Our template is a copy of the `puppet_standalone.json` template we created in Chapter 4. We've added an additional step in the `provisioners` block to handle our Serverspec installation, so we'll again focus on the `provisioners` block.

Listing 6.3: The Serverspec provisioners block

```
"provisioners": [
  {
    "type": "shell",
    "script": "install-puppet.sh"
  },
  {
    "type": "file",
    "source": "hieradata",
    "destination": "/tmp"
  },
  {
    "type": "puppet-masterless",
    "puppet_bin_dir": "/opt/puppetlabs/bin",
    "manifest_file": "manifests/site.pp",
    "hiera_config_path": "hiera.yaml",
    "module_paths": ["modules"]
  },
  {
    "type": "file",
    "source": "tests",
    "destination": "/tmp"
  },
  {
    "type": "shell",
    "inline": [
      "cd /tmp/tests",
      "sudo /opt/puppetlabs/puppet/bin/gem install serverspec",
      "sudo /opt/puppetlabs/puppet/bin/rake spec"
    ]
  }
]
```

NOTE You can find all the code for these examples on GitHub.

We've chained together three provisioners: the `shell`, `file`, and `puppet-masterless` provisioner.

Our first steps match what we did in Chapter 4: installing Puppet, uploading the Hiera data, and running the Puppet agent to install the modules.

Following that, we've added some final inline shell scripts to install Serverspec and run our tests.

Let's look at each these steps quickly to ensure we understand what's happening.

Installing Puppet

Our `shell` provisioner executes a single script called `install-puppet.sh`. The script adds the Puppet APT repo, then refreshes the package database and installs the `puppet-agent`. The `puppet-masterless` provisioner will later use this agent to execute any Puppet code and configure our host.

Hiera and manifest data

The next provisioner is the `file` provisioner. This loads a Hiera data directory onto the host.

NOTE Remember from Chapter 4 that we need to upload this directory due to a nuance in the way Puppet is run. All of our other configuration will remain local.

Hiera configuration is contained in YAML files. We've created two files. The first, `hiera.yaml`, tells Puppet about Hiera's configuration and structure. It's identical to our configuration in Chapter 4.

Listing 6.4: The hiera.yaml file revisited

```
---
version: 5
defaults:
  datadir: '/tmp/hieradata'
  data_hash: yaml_data
hierarchy:
  - name: "Other YAML hierarchy levels"
    paths:
      - "common.yaml"
```

Remember the `hiera.yaml` file also tells Puppet what configuration to install using the second file: `common.yaml`. We've changed the `common.yaml` file from Chapter 4 to add some new configuration.

Listing 6.5: The updated common.yaml file

```
---
classes:
  - ssh
  - locales
  - motd
  - timezone
  - ntp
timezone:timezone: 'UTC'
ntp:ensure: latest
```

Notice that we've added four new modules: `locales`, `motd`, `timezone`, and `ntp`. These additional modules will be loaded together with our existing `ssh` module. We've also specified some additional configuration to set the time zone of the host and enable NTP to ensure instances launched from our image have the correct time.

We've also maintained the `site.pp` manifest in the `manifests` directory to consume this data.

Listing 6.6: The updated site.pp file

```
lookup('classes', {merge => unique}).include
```

This will look up the `classes` array in our `common.yaml` file and load the resulting modules into Puppet to be processed.

Next, our `file` provisioner uploads the local `hieradata` directory to the `/tmp` directory on the remote host. Our `puppet-masterless` provisioner will look for it here thanks to our Hiera configuration.

Let's go get the new modules and install them.

Installing modules

To install modules we're going to again use the `librarian-puppet` tool to manage our modules. We'll install Puppet and that tool first, if you don't already have them installed.

NOTE We've assumed you have Ruby and Rubygems installed to do this.

Listing 6.7: (Re-)installing librarian-puppet

```
$ sudo gem install puppet librarian-puppet
```

NOTE We've also added a `Gemfile` in the directory so if you have Bundler installed you can just `bundle install`.

Remember that Librarian-Puppet uses a `Puppetfile` file, much like a Ruby `Gemfile`. Let's update ours to add our new modules.

Listing 6.8: The updated Puppetfile

```
forge "https://forgeapi.puppetlabs.com"
mod 'saz/ssh'
mod 'saz/locales'
mod 'saz/motd'
mod 'saz/timezone'
mod 'puppetlabs/ntp'
```

The `Puppetfile` specifies the location of the Puppet Forge from which we'll get the modules. It also lists the specific modules.

We can then run the `librarian-puppet` command to install the modules.

Listing 6.9: Installing the modules

```
$ librarian-puppet install
```

You'll see that a `Puppetfile.lock` lock file will be created, as will the `modules` directory containing the downloaded modules and any supporting modules.

The Puppet standalone provisioner

The next provisioner is the `puppet-masterless` provisioner. It takes the configuration and pieces we downloaded and constructed earlier and combines them to execute on the remote host. Its configuration matches what we saw in Chapter 4.

Uploading our tests

Next we need to upload our tests. We need to do this so the tests can be executed on the image host.

Listing 6.10: Uploading the tests

```
{
  "type": "file",
  "source": "tests",
  "destination": "/tmp"
},
```

This uploads the `tests` directory to `/tmp` on the host. Inside this directory are our tests themselves (which we'll see shortly) as well a `Rakefile` that'll provide an execution wrapper for them, allowing us to run them via a Rake task.

TIP Serverspec also supports running tests remotely. We could make use of the `shell-local` provisioner to run Serverspec in its SSH mode, which connects via SSH and executes the tests. This would save us uploading and installing anything on the image. This blog post discusses running Packer and Serverspec in this mode. Or you can see an example of the configuration in this chapter adapted for SSH in this Gist.

Running Serverspec

Our last provisioning step is to install Serverspec itself, via the `serverspec` gem. We'll make use of the Ruby, Rake, and Rubygem binaries provided for us when we installed Puppet. This ensures we only install what is required for Serverspec, rather than polluting the host with additional packages.

NOTE If we wanted to tidy up after running our tests we could also uninstall the `serverspec` gem.

Listing 6.11: Installing and running Serverspec

```
{
  "type": "shell",
  "inline": [
    "sudo /opt/puppetlabs/puppet/bin/gem install serverspec",
    "cd /tmp/tests && sudo /opt/puppetlabs/puppet/bin/rake spec"
  ]
}
```

The second inline script actually runs the Serverspec tests. We change into the `/tmp/tests` directory we uploaded earlier, then run a Rake task that executes our tests.

Now we've seen the process that will be followed when provisioning our image, let's look at the Serverspec tests we're going to run.

Serverspec tests

Serverspec tests your infrastructure's state by executing commands against that infrastructure. For example, as our Package template builds an image with SSH configured and running, the tests should check SSH is correctly configured and:

1. That an SSH daemon is running.
2. That TCP port 22 is open.

In our case we have installed a series of modules with Puppet. We want to validate that each of those has been successfully applied. Let's see how we might write tests to validate those assertions.

TIP There are alternatives to Serverspec, like InSpec, Goss, or TestInfra that might also meet your testing needs.

Setting up Serverspec

Serverspec uses the same DSL as RSpec. To write tests we define a set of expectations inside a specific context or related collection of tests, usually in an individual file for each item we're testing.

Let's create a `spec` directory inside our `tests` directory to hold the tests themselves.

Listing 6.12: Creating the spec directory

```
$ cd tests
$ mkdir spec
```

RSpec test file names usually end in `_spec`, which tell us they contain the tests relevant for a specific context.

Creating our first Serverspec tests

Let's create a test file to hold our first tests. We'll create the SSH tests first, and call the file `ssh_spec.rb`.

TIP There's also the useful `serverspec-init` command which initializes a set of new tests.

Listing 6.13: Creating the ssh_spec.rb file

```
$ cd spec
$ touch ssh_spec.rb
```

Let's populate that test file now.

Listing 6.14: Our SSH tests

```
require 'spec_helper'

describe 'ssh daemon' do
  it 'has a running service of ssh' do
    expect(service('ssh')).to be_running
  end
  it 'responds on port 22' do
    expect(port 22).to be_listening 'tcp'
  end
end
```

We've now got some tests defined, and we're requiring a `spec_helper`. This helper loads useful configuration for each test and is contained in the `spec` directory in the `spec_helper.rb` file. Let's see it now.

Listing 6.15: The spec_helper.rb file

```
require 'serverspec'

set :backend, :exec
```

Our helper loads the `serverspec` gem and sets the `backend` configuration. Serverspec has two modes of operation—the one we're using now, `exec`, runs all tests locally—and an SSH mode, which, as we mentioned earlier, allows us to run the tests remotely. We're executing locally so we set the `backend` to `exec`.

Back to our SSH tests. We generally want to set a context for our tests; this groups all of the relevant tests together. To do this we use `describe` block. This sets a hierarchy for our tests, here a single-layer hierarchy, that groups all of the SSH test assertions together.

Listing 6.16: A describe block

```
describe 'ssh daemon' do
. . .
end
```

Inside our `describe` block we define a series of individual assertions about our SSH daemon's configuration.

Listing 6.17: A Serverspec test

```
describe 'ssh daemon' do
  it 'has a running service of ssh' do
    expect(service('ssh')).to be_running
  end
  it 'responds on port 22' do
    expect(port 22).to be_listening 'tcp'
  end
end
```

Each assertion articulates a expectation, hence the `expect` inside the assertion. Each assertion is wrapped in an `it ... end` block. Inside that block we use the `expect` syntax to specify the details of our assertion.

Our first assertion says that a service called `ssh` should be running.

```
expect(service('ssh')).to be_running
```

Serverspec supplements the existing RSpec DSL with infrastructure-centric resource types, such as services or ports, which have matchers that support those resources. Here we're using a resource called `service` which allows us to test the state of services on our host. That `service` resource has a matcher called `be_running`. Our assertion asks:

"Serverspec expects that the SSH service will be running"

If this assertion isn't true, the test will fail and Packer will abort.

NOTE Serverspec automatically detects the operating system of the host on which it is being run. This allows it to know what service management tools, package management tools, or the like need to be queried to satisfy a resource. For example, on Ubuntu, Serverspec knows to use APT to query a package's state.

Our second test asks Serverspec to confirm that the network configuration of our SSH daemon is correct.

```
expect(port 22).to be_listening 'tcp'
```

Here we're using a new resource, `port`, to specify the standard SSH port of `22` and a matcher, `be_listening`, from the resource that we've configured to check for TCP ports. This test will validate that TCP port 22 is open on the host; if it is not, the test will fail.

TIP Check out Better Specs for some tips and tricks for writing better RSpec tests.

Let's look at another set of tests, this time for our NTP module.

The Serverspec NTP tests

Let's create a new test file for our NTP tests.

Listing 6.18: Creating the ntp_spec.rb file

```
$ cd spec
$ touch ntp_spec.rb
```

Now let's populate that file with some new tests.

Listing 6.19: The NTP tests

```
require 'spec_helper'

describe 'ntp' do
  it 'ntp package' do
    expect(package('ntp')).to be_installed
  end
  it 'has an enabled service of ntp' do
    expect(service('ntp')).to be_enabled
  end
  it 'has a running service of ntp' do
    expect(service('ntp')).to be_running
  end
end
```

Note that we've again specified the `spec_helper`. We've also created a new `describe` block for our NTP tests; inside it we have three tests. The first uses the `package` resource to test that the `ntp` package is installed using the `be_installed` matcher. The second two tests confirm that the `ntp` service is enabled and running.

TIP You can find the full list of available resources in the Serverspec documentation.

We can then create tests for our time zone, motd, and locales configuration.

NOTE You can find those additional tests in the example code on GitHub. There are also a lot of test examples available in GitHub with a simple search.

The Rakefile

Now that we have our tests, let's create and populate a `Rakefile` so we can run our tests as a Rake task. We'll create it now in the `tests` directory.

Listing 6.20: Creating the Rakefile

```
$ cd tests
$ touch Rakefile
```

Now let's populate our `Rakefile`.

Listing 6.21: The Rakefile

```
require 'rake'
require 'rspec/core/rake_task'

RSpec::Core::RakeTask.new(:spec) do |t|
  t.pattern = 'spec/*_spec.rb'
  t.fail_on_error = true
end
```

Our `Rakefile` requires `rake` and the `rspec` Rake tasks and then creates a new Rake task. That task executes any files ending in `_spec` in the `spec` directory as RSpec

tests. It also ensures that if any of the tests fail that it'll return a non-zero exit code.

Now that our Serverspec configuration and tests are complete, let's run them and see what happens.

Running our tests

Our tests will run as part of our normal Packer build when we execute the `serverspec.json` template.

Listing 6.22: Running the Serverspec tests

```
$ packer build serverspec.json

. . .

==> amazon-ebs: Uploading tests => /tmp
==> amazon-ebs: Provisioning with shell script:

. . .

    amazon-ebs: /opt/puppetlabs/puppet/bin/ruby -I/opt/
puppetlabs/puppet/lib/ruby/gems/2.1.0/gems/rspec-support-3.6.0/
lib:/opt/puppetlabs/puppet/lib/ruby/gems/2.1.0/gems/rspec-core-
3.6.0/lib /opt/puppetlabs/puppet/lib/ruby/gems/2.1.0/gems/rspec-
core-3.6.0/exe/rspec --pattern spec/\*_spec.rb
    amazon-ebs: .......
    amazon-ebs:
    amazon-ebs: Finished in 0.14813 seconds (files took 0.35331
seconds to load)
    amazon-ebs: 7 examples, 0 failures

. . .
```

Here we've run the `serverspec.json` template and grabbed only the test run out-

put. We can see our tests are uploaded, Serverspec is installed, and then our Rake task is executed with `rake spec`. Each dot:

```
.......
```

Indicates a test that has passed. Serverspec has reported at the end that seven examples were run and none failed. Our tests have passed—now our image will be created!

If, alternatively, something wasn't right and a test failed, we'd see that in our output. Let's run this again, this time assuming something has gone wrong with our configuration.

Listing 6.23: A failed test run

```
. . .

    amazon-ebs: ...F...
    amazon-ebs:
    amazon-ebs: Failures:
    amazon-ebs:
    amazon-ebs: 1) ntp has a running service of ntp
    amazon-ebs: Failure/Error: expect(service('ntp')).to
be_running
    amazon-ebs: expected Service "ntp" to be running
    amazon-ebs: /bin/sh -c systemctl\ is-active\ ntp
    amazon-ebs: inactive
    amazon-ebs:
    amazon-ebs: # ./spec/ntp_spec.rb:11:in `block (2 levels) in <
top (required)>'
    amazon-ebs:
    amazon-ebs: Finished in 0.14463 seconds (files took 0.30628
seconds to load)
    amazon-ebs: 7 examples, 1 failure
    amazon-ebs:
    amazon-ebs: Failed examples:
    amazon-ebs:
    amazon-ebs: rspec ./spec/ntp_spec.rb:10 # ntp has a running
service of ntp
    amazon-ebs:

. . .

Build 'amazon-ebs' errored: Script exited with non-zero exit
status: 1

. . .
```

This time it appears something has gone wrong and the `ntp` service is not running. This means our assertion about the `ntp` service has failed—hence the Packer build has failed. We'll need to work out what has gone wrong, fix the issue, and then

run the build again.

TIP When testing like this, it's useful to run Packer with the `-debug` flag enabled, which stops between steps and allows you to debug the server if any issues emerge.

Summary

In this chapter, we've seen how to add tests to a Packer build process. We've learned about Serverspec, an RSpec-based framework, and how to write tests for it.

We reused some of our Puppet standalone configuration from Chapter 4 and added some additional Puppet modules. We've written some basic tests to demonstrate how to validate that that configuration has been successfully applied.

In the next chapter, we'll look at building pipelines for multi-platform images with Packer.

Chapter 7

Pipelines and workflows

In the last few chapters we've seen how to perform a variety of tasks with Packer. We've built images, provisioned them, worked with Docker, seen how post-processing works, and written some tests to validate that our images are correct. We've also targeted a couple of platforms: Amazon EC2 AMI images and Docker container images.

In this chapter we're going to combine many of these concepts to produce a multi-target, consistently provisioned image. Our image will be built both as an Amazon AMI and as a Docker image. We'll also post-process the image selectively to ensure it is deployed to both Amazon and the Docker Hub.

Our build pipeline

We're going to build dual images, a Docker image and an Amazon AWS AMI, using a single Packer template and single provisioning process. We're going to post-process our Docker image and upload it to the Docker Hub, and we're going to rely on the Amazon builder to send the completed AMI to Amazon.

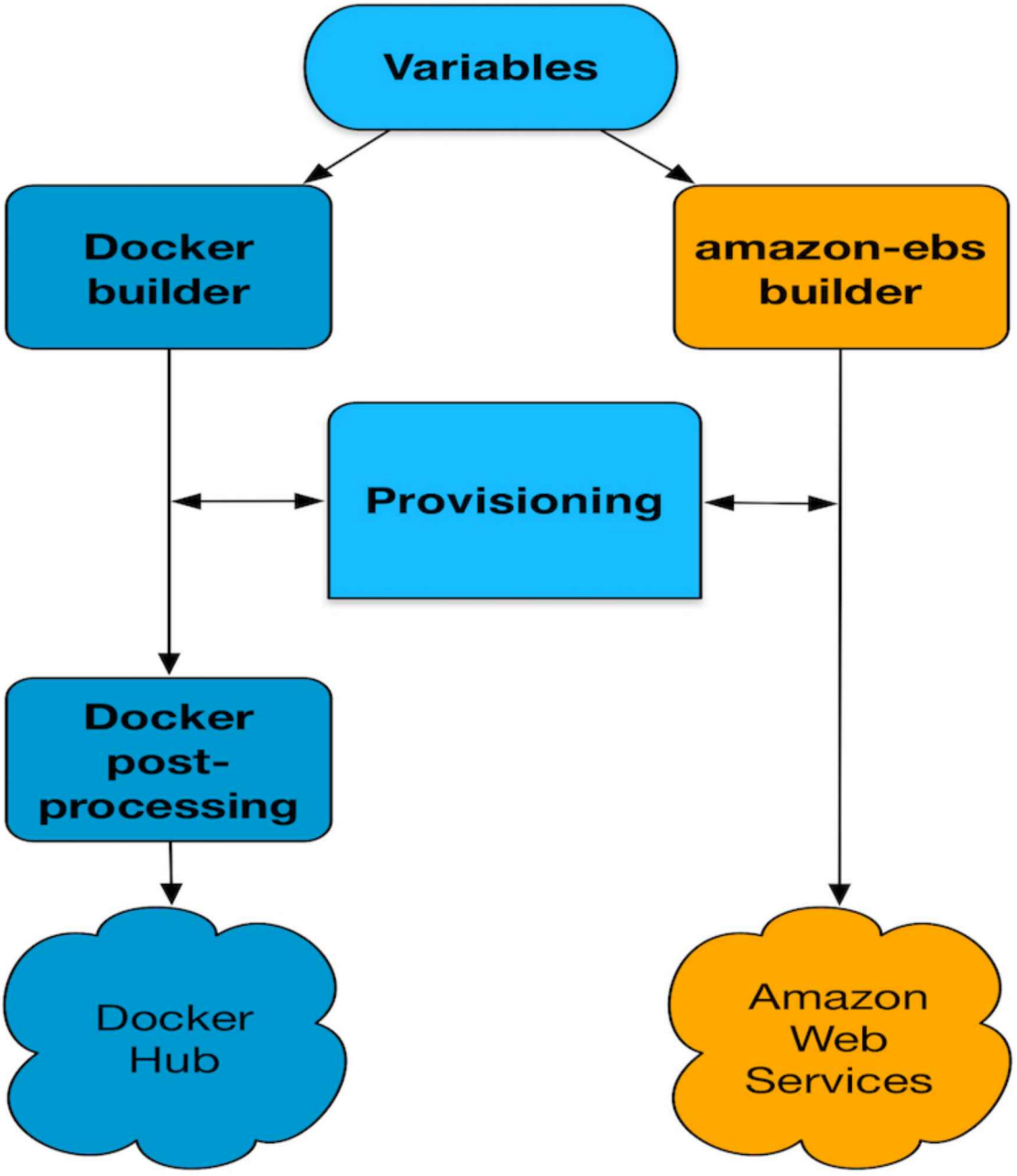

Figure 7.1: Build pipeline

Our pipeline template

Let's take a look at our pipeline template. To examine it, we're going to break it into pieces—it's too big to see here in its entirety. However, you can view the full template in the code for this book on GitHub.

Variables

Let's start with the template's variables. We've specified a few more than we've previously seen.

Listing 7.1: The pipeline template variables

```
"variables": {
  "name": "webimage",
  "namespace": "jamtur01",
  "fqdn": "webapp.example.com",
  "osName": "ubuntu",
  "osVersion": "17.04",
  "ami": "ami-a025aeb6",
  "aws_region": "us-east-1",
  "aws_access_key": "",
  "aws_secret_key": ""
},
```

You can see we've created a series of variables, including our AWS keys. By creating variables we've centralized the configuration for our images and avoided as much repetition as possible. We've created a `name` variable with a value of `webimage` that we'll use to name our images, and several that identify the base operating system we'll use for our image. We'll also use other variables in our build process to decorate our images.

Builders

Let's look at the two builders, one for Docker and one for Amazon, that we're going to use. We'll start with the Docker builder.

Listing 7.2: The Docker builder

```
"builders": [{
  "type": "docker",
  "image": "{{user `osName`}}:{{user `osVersion`}}",
  "commit": true
},
```

Our first builder is the `docker` builder we saw in Chapter 5. In this case, we're using two of our variables to construct the `image` key. Here, the variables will result in the builder using the `ubuntu:17.04` base Docker image. We're committing this image to the Docker daemon with the `commit` key set to `true`.

Next, we have the `amazon-ebs` builder we're going to use for our AMI.

Listing 7.3: The amazon-ebs builder

```
{
  "type": "amazon-ebs",
  "access_key": "{{user `aws_access_key`}}",
  "secret_key": "{{user `aws_secret_key`}}",
  "region": "{{user `aws_region`}}",
  "source_ami": "{{user `ami`}}",
  "instance_type": "t2.micro",
  "ssh_username": "ubuntu",
  "ami_name": "{{user `fqdn`}}-{{timestamp | clean_ami_name}}",
  "ami_description": "Web Application Image",
  "tags": {
    "name": "{{user `name`}}",
    "namespace": "{{user `namespace`}}",
    "fqn": "{{user `fqdn`}}",
    "os": "{{user `osName`}} {{user `osVersion`}}",
    "built": "{{isotime}}"
}
```

This builder is doing a bit more. We've pulled in several variables that we've set, including our AWS keys, the region, and which AMI to use.

TIP Instead of specifying the `source_ami` as a variable we could also use the AMI filtering mechanism to search for a specific AMI.

We're also specifying the name of the AMI to be produced:

Listing 7.4: The AMI name

```
"ami_name": "{{user `fqdn`}}-{{timestamp | clean_ami_name}}",
```

Here we're combining the `fqdn` variable, which contains the fully qualified domain name of our application that we've specified, combined with a timestamp and the `clean_ami_name` function to remove any characters unacceptable in an AMI name. This will result in an AMI named something like:

```
webapp.example.com-1497704817
```

We've also assigned a series of tags to our new image. Tags allow us to characterize images in AWS. Our build will result in images tagged like this:

Key	Value
built	2017-06-17T13:06:57Z
fqn	webapp.example.com
name	webapp
namespace	jamtur01
os	ubuntu 16.04

Figure 7.2: AWS image tags

Each build will execute in, in parallel where possible, when we run Packer. They'll then pass on to the provisioning step.

Provisioners

After our build process, both images will be passed to the provisioning process. Let's look at that part of the template now.

Listing 7.5: The pipeline provisioner

```
"provisioners": [{
  "type": "file",
    "source": "Gemfile",
    "destination": "/tmp/Gemfile"
},
{
  "type": "shell",
  "script": "provision.sh",
  "override": {
    "amazon-ebs": {
      "execute_command": "sudo {{.Path}}"
    }
  }
}]
```

We've got a two-step provisioning process. We first use a `file` provisioner to upload a `Gemfile`; this will populate our application's dependencies. Let's take a quick look at the `Gemfile`.

Listing 7.6: The pipeline Gemfile

```
source 'https://rubygems.org'

gem 'rails', '~> 5.1.0'
```

Our `Gemfile` will install the gems and dependencies required for a Ruby on Rails application.

Our next step runs a shell script called `provision.sh`. This will install some packages and actually install our application's Rubygems. Let's look at this script too.

Listing 7.7: The provision.sh script

```
#!/bin/sh -x

# Install Ruby
apt-get update -yqq
DEBIAN_FRONTEND=noninteractive apt-get install -yqq ruby
rubygems ruby-dev build-essential zlib1g-dev liblzma-dev

# Create app directory
mkdir -p /usr/local/app

# Move Gemfile
mv /tmp/Gemfile /usr/local/app/

# Install gems
cd /usr/local/app
gem install bundler --no-ri --no-rdoc
bundle install
```

NOTE We could easily use a configuration management tool, as we did in Chapter 4, to perform the provisioning.

We first install Ruby and some prerequisite packages. We then create a directory to hold our application at `/usr/local/app`. We move our `Gemfile` into that directory, install the Bundler gem, and then run the `bundle install` command to install the required gems.

This second provisioning script contains an interesting clause we've not seen before: an override.

Listing 7.8: The override clause

```
"override": {
  "amazon-ebs": {
    "execute_command": "sudo {{.Path}}"
  }
}
```

Packer tries to ensure identical images are produced on varying target platforms. But sometimes the the paths to those identical images need to diverge to achieve the required end result. This might be because a specific command or action varies in operation on one of those platforms. Enter overrides.

An override allows you to specify a varying action on a specific build. Each override takes the name of a builder, in our case `amazon-ebs`, and specifies one or more keys that are different for that provisioner when it is executed for that build.

In this case, we override the `execute_command` key. For the `amazon-ebs` build only, we'll run the `provision.sh` script prefixed with the `sudo` command: `sudo {{.Path }}`.

The `{{.Path}}` specified in the command is a variable defined in the `execute_command` key and represents the path to the script to be executed.

NOTE The `execute_command` key also has access to the `Vars` variable, which contains all of the available environmental variables.

Post-processors

Once our two images have been provisioned then the post-processors take over. We've specified two post-processors, both working with our Docker image. Let's look at those now.

Listing 7.9: The pipeline post-processors

```
"post-processors": [
  [
    {
      "type": "docker-tag",
      "repository": "{{user `namespace`}}/{{user `name`}}",
      "tag": "{{user `fqdn`}}",
      "only": ["docker"]
    },
    {
      "type": "docker-push",
      "only": ["docker"]
    }
  ]
]
```

In this section notice we've specified, in sequence, the `docker-tag` and `docker-push` post-processors that we saw in Chapter 5. We've added a twist this time because we're executing two builders. Inside our sequence of post-processors we've used a new key: `only`. The `only` key constrains a post-processor to only run when specific builders are invoked. We can specify an array of builder names—in our case this is the `docker` builder—for which these post-processors will be executed. This prevents the `amazon-web` builder from unnecessarily triggering the post-processors for Docker images.

There's also a second key, `except`, that performs a similar but reversed operation. If you use the `except` key, post-processors will run for all builders except those listed in that key. For example, if we wanted to achieve the same end result as

our `only` key using `except`, we could do the following:

Listing 7.10: The except key

```
"post-processors": [
  [
    {
      "type": "docker-tag",
      "repository": "{{user `namespace`}}/{{user `name`}}",
      "tag": "{{user `fqdn`}}",
      "except": ["amazon-ebs"]
    },
    {
      "type": "docker-push",
      "except": ["amazon-ebs"]
    }
  ]
]
```

This would exclude the builder named `amazon-ebs` from running, but the `docker` builder (or any other builders specified) will execute.

Execution

Now that we've explored our template, let's see it in operation. We're going to run our pipeline build.

Listing 7.11: Running the pipeline build

```
$ packer build pipeline.json
amazon-ebs output will be in this color.
docker output will be in this color.

==> amazon-ebs: Prevalidating AMI Name...
==> docker: Creating a temporary directory for sharing data...
==> docker: Pulling Docker image: ubuntu:17.04
    amazon-ebs: Found Image ID: ami-a025aeb6

. . .

Build 'amazon-ebs' finished.
Build 'docker' finished.

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs: AMIs were created:

us-east-1: ami-0f5b6f19
--> docker: Imported Docker image: sha256:9
be5afdeafb1d0c7f5949f116e6608e1890f1281fb9575e64f7fe4adb80d32ca
```

Here we can see Packer executing both builders. Packer will emit each builder's output in different colors. You can also see the actions from each builder prefixed with the name of the builder:

```
==> amazon-ebs: Prevalidating AMI Name...
```

We've cut out most of the output, but you'll see that each builder individually triggers the provisione, and that each builder gets its own run of the provisioner installing the required directory and RubyGems.

Lastly, the post-processors are run, but only for the `docker` builder, as our image is tagged and pushed to the Docker Hub as `jamtur01/webimage`. If you ever need to only run one builder, there is another command line argument, `-only`, that you can pass to the `packer build` command.

Listing 7.12: Using -only on the command line

```
$ packer build -only=docker pipeline.json
```

Here we're only executing the `docker` builder rather than both builds.

Continuous Integration

It's also easy to extend the pipeline we've built with integration through further tools. This template readily lends itself to use with a Continuous Integration tool like Jenkins CI or commercial CI services like CircleCI or TravisCI. Indeed, there is a Jenkins plugin for Packer available that can help build images during CI runs.

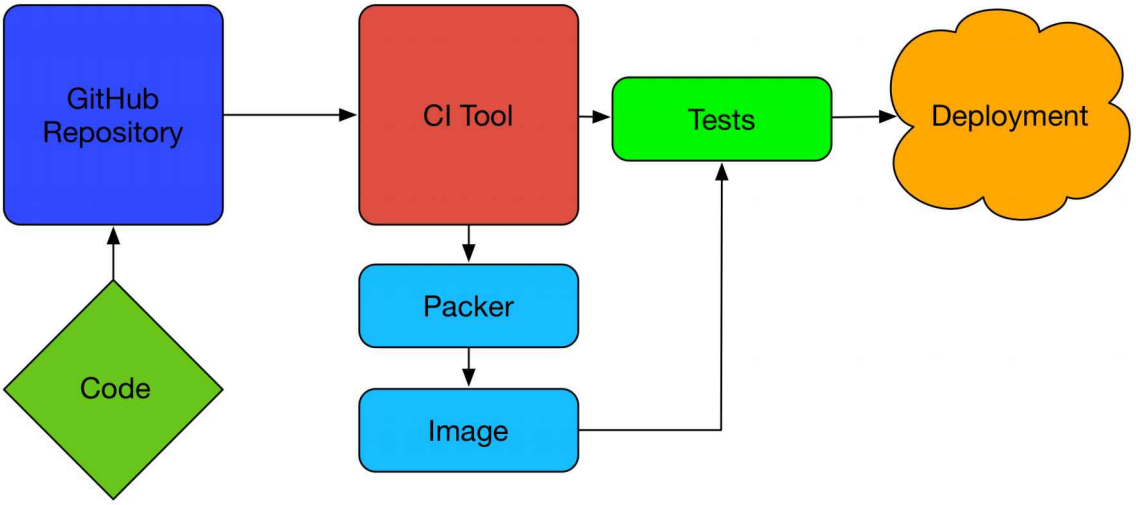

Figure 7.3: CI pipeline

NOTE There's a useful blog post and a tool called Bakery that show some good CI/CD pipeline ideas.

Summary

In this chapter we've seen how to build multi-builder images—in our case an AMI and a Docker image. Both provisioned identically using a shell script. We could extend this to include configuration via a configuration management tool.

We can further see how we can inject Packer into a wider tools landscape as part of a Continuous Integration life cycle.

In the next chapter we'll see how to integrate Packer plugins and write our own extensions to Packer.

Chapter 8

Extending Packer

In this chapter we're going to see the two methods currently available for extending Packer: integrating third-party plugins and writing our own plugins. Packer plugins make it easy to add new functionality without needing to change any core code. Indeed, much of Packer's existing core functionality is provided by plugins that ship with Packer itself. Most of the builders, provisioners, and so on are Packer plugins that are automatically loaded when Packer is run. There is also a small ecosystem of Packer plugins available for you to integrate.

Packer plugins

Packer plugins are standalone applications written in Go that Packer executes and communicates with. They aren't meant to be run manually—the Packer core provides an integration API that has the communication layer between the core and plugins.

Unfortunately there currently isn't a lot of scaffolding around the management of Packer plugins, nor is there a central home for plugins or tooling available to install them automatically or manage them as packages.

As the Packer documentation suggests, currently the best method to find Packer plugins is via a Google search or a search on GitHub.

Installing a plugin

Installing a plugin is currently very manual. Each plugin is a standalone Go binary that needs to be downloaded, named correctly, and placed in an appropriate directory. Let's find a sample Packer plugin, a Packer post-processor plugin for AMI management, to download and install.

We start by creating a directory to hold our plugins. By default, Packer looks for plugins in a configuration directory in the `$HOME` directory: `~/.packer.d`.

Listing 8.1: Creating the Packer plugin directory

```
$ mkdir -p ~/.packer.d/plugins
```

NOTE On Microsoft Windows this directory would be `%APPDATA%/packer.d/plugins`. You can also place plugins into the same directory that contains the `packer` binary or in the current working directory.

We first choose the right binary for our architecture—in our case, a Linux-based AMD64 binary.

Listing 8.2: Getting and unpacking the plugin

```
$ cd ~/.packer.d/plugins
$ wget https://github.com/wata727/packer-post-processor-amazon-
ami-management/releases/download/v0.2.0/packer-post-processor-
amazon-ami-management_linux_amd64.zip
$ unzip packer-post-processor-amazon-ami-management_linux_amd64.
zip
$ ls -la packer-post-processor-amazon-ami-management
-rwxr-xr-x  1 james  staff  18480879 May 27 10:22 packer-post-
processor-amazon-ami-management
```

We can then add some configuration for our new post-processor.

Listing 8.3: The new post-processor plugin

```
{
  "builders": [{
    "type": "amazon-ebs",
    "region": "us-east-1",
    "source_ami": "ami-a025aeb6",
    "instance_type": "t2.micro",
    "ssh_username": "ubuntu",
    "ami_name": "initial-ami {{timestamp | clean_ami_name}}",
    "tags": {
        "Amazon_AMI_Management_Identifier": "packer-plugin-
example"
    }
  }],
  "post-processors": [{
    "type": "amazon-ami-management",
    "region": "us-east-1",
    "identifier": "packer-plugin-example",
    "keep_releases": "3"
  }]
}
```

Here we've added a new post-processor, `amazon-ami-management`, that we acquired as a plugin. When we now execute Packer it will use the installed plugin to manage the life cycle of our AMI images.

Writing our own plugins

Packer's plugins are Go applications. Their architecture is a little unusual. They are loaded at runtime as separate applications, and then IPC and RPC are used to communicate between the core and the various plugins. The core manages starting, stopping, and cleaning up after each plugin execution.

This allows each plugin to be entirely standalone. They're linked via an interface with the core, but have their own dependencies and are isolated from the process space of the Packer core.

Writing plugins does require some Go knowledge but their scaffolding makes for pretty simple structures. Each kind of plugin, builder, provisioner, and post-processor has a corresponding interface that needs to be implemented. The exposure and communication between plugin and core are handled via the Packer plugin package. We'll see how to add this to our plugin shortly.

Getting started with Go

As Packer plugins are written in Go, it's useful to get a good grounding in it. There are some resources available to help with that.

- The Golang tour.
- The Golang learning resources.
- Learn Go in X Minutes.
- Go by example.

The Packer community is also welcoming to newcomers and a good place to find help with your Packer-related development questions.

Building a plugin

We've created our basic plugin on GitHub and called it `packer-slacknotifications`. It's a very simple post-processor that will send notifications to Slack when Packer completes a build. Let's look at what went into making the plugin.

Naming plugins

You need to name your plugins to match their purpose and the type of plugin. To help with this, plugins have a naming convention like so:

```
packer-plugin-name
```

Plugins are prefixed with `packer` and the name of the plugin. So for our post-processor plugin this results in:

```
packer-slacknotifications
```

Plugin basics

Each type of plugin has a defined interface. These take advantage of Go interfaces to define what is required to instantiate each type of plugin. To build a plugin we define the interface required for each type of plugin:

- Builders
- Post-processors
- Provisioners

Each of these interfaces are contained in the `github.com/hashicorp/packer` package, and the plugin server to deliver them is contained in the `github.com/hashicorp/packer/packer/plugin` package. Pretty much every plugin will require both of these packages.

To make a plugin functional, we implement the relevant interface—for example, for our post-processor, we need to implement the `packer.PostProcessor` interface. Let's take a quick look at the current post-processor interface:

Listing 8.4: The PostProcessor interface

```
package packer

type PostProcessor interface {
    Configure(...interface{}) error

    PostProcess(Ui, Artifact) (a Artifact, keep bool, err error)
}
```

Our interface needs to provide both a `Configure` and a `PostProcess` method. Let's crack open our post-processor and see how that works.

Listing 8.5: The post-processor.go file

```
. . .

type PostProcessor struct {
    config Config
}

func (p *PostProcessor) Configure(raws ...interface{}) error {
. . .

}

func (p *PostProcessor) PostProcess(ui packer.Ui, artifact
packer.Artifact) (packer.Artifact, bool, error) {

. . .

}
```

This is from our plugin's source code and shows us defining a `struct` and two methods: `Configure` and `PostProcess`. The first method will take our configuration data from the template that calls the post-processor, and the second will actually execute the post-processing task.

We then register the interface using the plugin server.

Listing 8.6: Registering the interface

```
package main

import (
    "github.com/hashicorp/packer/packer/plugin"
    "github.com/turnbullpress/packer-slacknotifications/
slacknotifications"
)

func main() {
    server, err := plugin.Server()
    if err != nil {
        panic(err)
    }

    server.RegisterPostProcessor(new(slacknotifications.
PostProcessor))
    server.Serve()
}
```

We create a plugin server using the server provided by the Packer plugin package. Then we register a new post-processor:

```
server.RegisterPostProcessor(new(slacknotifications.PostProcessor))
```

Using that server and the server itself, `server.Serve()` will handle all of the communication between the plugin and the core.

TIP The best way to understand how plugins work is to look closely at the existing plugins in the Packer core and the documentation for the specific plugin types.

Plugins are standard Go applications so, like them, they are built using the `go` binary. Binary releases for our sample post-processor are available on GitHub. The sample also comes with a series of scaffolding like a CI service and a `Makefile` that'll give you some hints and tips about how to build and manage Packer plugins.

Further information is available in the Packer documentation.

Summary

In this chapter we've learned how to add existing third-party plugins to Packer's core.

We also saw the basics of how to create our own plugins, including seeing the source code for a full-fledged post-processor of our own.

This wraps up the book, which we hope you've found useful. Thanks for reading!

List of Figures

Listings

Index

Thanks! I hope you enjoyed the book.

ISBN 9780988820272